Institut für Informatik der Freien Universität Berlin
Arbeitsgruppe Datenbanken und Informationssysteme

Ein generisches ereignisgesteuertes System für komplexe Ereignisverarbeitung

Masterarbeit

13. Mai 2014

Autor: Ralf Öchsner

Gutachter: Prof. Dr. Agnès Voisard
Prof. Dr. Adrian Paschke

Betreuer: Daniel Kressner

Zusammenfassung

In vielen Bereichen der Informatik werden heutzutage ereignisgesteuerte Softwaresysteme eingesetzt. Bisher wurden diese Systeme, trotz vieler Gemeinsamkeiten, fast immer für jede Anwendungsdomäne neu entwickelt.

Im Rahmen dieser Arbeit wurde ein generisches ereignisgesteuertes System namens *Crepe* entwickelt, das das Ziel verfolgt, für möglichst viele Anwendungsdomänen verwendbar zu sein. Hierfür wurde eine allgemeingültige Ereignisspezifikation und eine Ereignisalgebra herausgearbeitet, die von der Software für die Verarbeitung der Ereignisse verwendet werden. Weiterhin ist die Software modular aufgebaut, so dass sie leicht erweiterbar ist und so noch besser an die Anforderungen von einzelnen Domänen angepasst werden kann. Durch Schnittstellen für das Importieren und Exportieren von Ereignissen wird eine Anbindung an externe Systeme ermöglicht.

Abstract

Nowadays, event-based software systems are used in many areas of computer science. Despite these systems having a lot in common, a new system has so far been developed for every application domain.

Throughout the course of this thesis, an event-based system has been developed, that pursues the goal of being useable for as many application domains as possible. Therefore, a universal event specification and algebra is developed that is used for the processing of the events. Furthermore, the software is built with a modular design so that it can easily be extended and therefore be adjusted to the requirements of particular domains. Through import and export interfaces, an integration into external systems is enabled.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation und Themengebiet

Durch die schnelle Entwicklung von Software- und Hardwaresystemen in verschiedenen Bereichen der Informatik sind viele neue dynamische und komplexe Systeme entstanden. Häufige Anforderungen an solche Systeme sind Verteiltheit, Nebenläufigkeit und das asynchrone Auftreten diskreter Ereignisse. Diese Systeme lassen sich unter dem Begriff ereignisgesteuerte Systeme (Event Based Systems) zusammenfassen, da Ereignisse (Events) im Mittelpunkt stehen und dort als Modellierungsgrundlage dienen [3].
Ereignisgesteuerte Systeme werden beispielsweise benutzt um Geschäftsprozesse zu modellieren, Straßenverkehr zu regulieren oder Lieferkettenmanagement (Supply-Chain-Management) zu betreiben [22]. Hierfür können Daten aus unterschiedlichen Quellen, wie zum Beispiel Sensornetzwerken dienen. Das Verarbeiten von Ereignissen aus verschiedenen Quellen wird häufig als *komplexe Ereignisverarbeitung* (engl. *Complex Event Processing*) bezeichnet [26].
In ereignisgesteuerten Systemen werden anhand von beobachteten Ereignissen bestimmte Aktionen ausgelöst. Ein einfaches Beispiel für eine solche Aktion wäre das Senden einer Benachrichtigung. Geschieht ein bestimmtes Ereignis so, wird eine Nachricht an einen Interessenten verschickt. Beispiele für Ereignis wären etwa, das Überschreiten einer bestimmten Temperatur, die von einem Sensor ausgelesen wird, eine Wertänderung eines Börsenkurses oder das entstehen eines Verkehrsstaus.
Ein Ereignis ist im Allgemeinen das Auftreten eines beobachteten Geschehens [26] bzw. eine Zustandsänderung in einem System[6]. Da Ereignisse immer einen Zeitpunkt haben zu dem sie auftreten, kann eine Abfolge von Ereignissen als Datenstrom betrachtet werden. Aktionen werden innerhalb des ereignisgesteuerten Systems anhand von Regeln ausgelöst, die auf die Ereignisse aus dem Datenstrom reagieren.
Hierzu überprüfen Regeln die eintreffenden Ereignisse auf Bedingungen. Die von den Regeln ausgelösten Aktionen können dann wiederum als Ereignisse betrachtet und somit erneut von Regeln ausgewertet werden. Um Regeln zu definieren werden Operatoren verwendet. Durch Kombination verschiedener Operatoren lassen sich komplexe Regeln erstellen. Ein ereignisgesteuertes System enthält daher, allgemein betrachtet, eine Algebra, die für die Ereignisverarbeitung zuständig ist, wobei die Auswertung von Operatoren als Rechnen mit Ereignissen betrachtet werden kann.
Ein Beispiel für das oben genannte Rechnen mit Ereignissen, wäre etwa ein Sensorsystem in einem Kühlhaus. Mehrere Sensoren im Kühlhaus messen die Temperatur und senden

diese in regelmäßigen Abständen an ein Softwaresystem. Dort sind Regeln hinterlegt, die die Temperaturen auswerten. Wenn beispielsweise alle Sensoren die gleiche niedrige Temperatur melden, außer der Sensor, der in der nähe eines Fensters angebracht ist, eine höhere Temperatur gemessen hat, so kann das System etwa mit einem Alarm reagieren, der anzeigt, dass ein Fenster offen steht. Die Rechenoperationen der Regel bestünden in diesem Fall aus logischen Und-Verknüpfungen und Größer-, bzw. Gleichheitsoperationen. Das aus der Regel resultierende Ereignis wäre, dass ein Alarm ausgelöst wurde.

Vorwiegend wurden Ereignisse bisher je nach Anwendungszweck unterschiedlich modelliert [19, 18, 4, 41, 39, 7], obwohl die Strukturen der Ereignisse starke Gemeinsamkeiten aufweisen. Ziel soll es daher sein, ein einheitliches Datenmodell zu erschaffen, das möglichst alle Arten von Ereignissen berücksichtigt.

Wie oben bereits beschrieben, ist ein Ereignis im Allgemeinen eine Zustandsänderung in einem System[6] und hat einen Zeitpunkt zu dem es auftritt. Weiterhin hat jedes Ereignis mindestens ein Attribut, das das beobachtete Geschehen repräsentiert. Beispielsweise kann in einem Sensorsystem das Überschreiten einer bestimmen Temperatur ein Ereignis auslösen. Das Attribut wäre dann die Temperatur und der Attributwert beispielsweise 30 Grad Celsius.

Je nach Ereignisart können auch mehrere, verschiedene Attribute vorkommen. Beim vorherigen Beispiel etwa das Überschreiten der Temperatur kombiniert mit einer bestimmten Luftfeuchtigkeit. Jeder Wert eines Attributs hat dabei einen Wertebereich[37], der sich aus dem Datentyp des Attributs ergibt. Es liegt daher nahe, Ereignisse als ein dreier Tupel aus einer eindeutigen ID, dem Zeitpunkt und einer *Collection* aus Attributen-Werte-Paaren, wie bei Voisard und Ziekow[38] beschrieben, zu modellieren. Die ID soll dabei der besseren Referenzierbarkeit dienen. Bei einem objektorientierten Modell kann hierfür die Objekt-ID verwendet werden. Die Attribut-Werte-Paare müssen dabei nicht in einer bestimmten Reihenfolge gespeichert werden, der Attributname sollte jedoch, zwecks Referenzierbarkeit, eindeutig sein. In Java bietet sich etwa eine *Map* als abstrakter Datentyp an, bei der der Attributname ein Schlüssel vom Typ String ist.

Abbildung 1.1 zeigt die Ereignisspezifikation noch einmal in Form eines UML-Klassendiagramms mit Beispielen für Attributtypen. Das Attribut *Temperature* ist eine Gleitkommazahl (`Double`). Die *StationsID* ist eine `Integer`-Klasse und *Location* vom Typ `GeoData`. Alle Typen sind von `Object` abgeleitet und können daher als Wert in der *Map* eingefügt werden.

1.2. Ziele dieser Arbeit

Ziel der Arbeit soll es sein, ein Softwaresystem zu erstellen, das die Gemeinsamkeiten von Ereignissen unterschiedlicher Domänen ausnutzt. Hierfür soll die oben beschriebene allgemeingültige Spezifikation von Ereignissen als interne Datenstruktur dienen. Zur

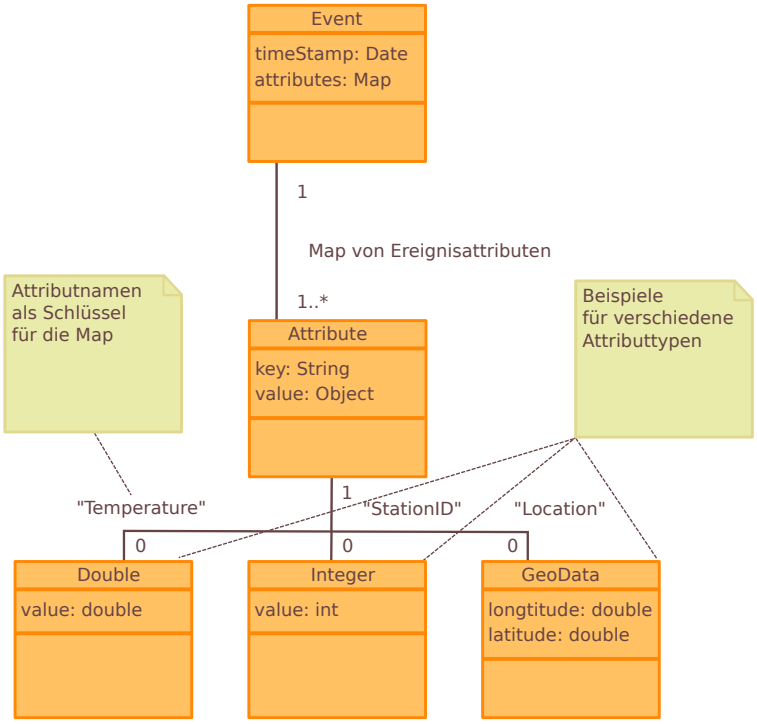

Abbildung 1.1.: Ereignisspezifikation als Klassendiagramm

Anbindung an externe Systeme sollen Komponenten für den Import und Export von Ereignissen implementiert werden.

Für den Import werden Ereignisse von einem Anwendungsspezifischen Format in die oben beschriebene, intern verwendete Datenstruktur (siehe Abbildung 1.1) umgewandelt. Das Laden, sowie die Umwandlung von Ereignissen soll mittels Adaptern stattfinden, von denen sich beliebig viele für verschiedene Anwendungszwecke in das System integrieren lassen sollen.

Liegen die Ereignisse im internen Format vor, können sie mit Hilfe einer geeigneten Algebra verarbeitet werden. Übliche Operatoren für eine solche Algebra sind etwa Disjunktionen, Konjunktionen, Negierungen oder zeitliche Bedingungen wie, dass ein Ereignis vor oder nach einem anderen Ereignis oder innerhalb eines bestimmten Zeitintervalls auftritt [38].

Um die Regeln der Algebra festzuhalten, soll geprüft werden, ob sich eine domänenspezifische Sprache, inklusive entsprechender Sprachwerkzeuge, bei der Implementierung der Software, dafür eignet. Hierbei soll stets eine leichte Erweiterbarkeit des Systems im

Vordergrund stehen. Neue Operatoren müssen unkompliziert integriert werden können, damit das System Akzeptanz in möglichst vielen Domänen finden kann.
Nach der Verarbeitung durch das Algebrasystem sollen die Ereignisse zurück in das ursprüngliche Ereignisformat exportiert werden können. Auch hierfür kommen wieder Adapter zum Einsatz, von denen sich erneut beliebig viele in das System integrieren lassen sollen. Sowohl für die Einlese- und Ausgabekomponenten sollen Schnittstellen geschaffen werden, so dass Erweiterungen leicht dem System hinzugefügt werden und untereinander ausgetauscht werden können. Durch die erweiterbaren Import- und Exportmöglichkeiten soll das System universell eingesetzt und in bestehende Softwareumgebungen integriert werden können.
Die Architektur der Software soll so gestaltet sein, dass es eine Aufteilung in ein Backend und ein Frontend gibt. Das Backend ist für die gesamte Ereignisverarbeitung inklusive Einlesen der Ereignisse, Auswertung der Regeln und die Ausgabe an externe Systeme zuständig. Im Frontend sollen die Ereignisse visualisiert werden und die Regeln der Algebra mit Hilfe einer GUI erstellt werden können. Im Rahmen dieser Arbeit soll nur das Backend des Softwaresystems implementiert werden, jedoch ohne konkrete Schnittstellen zu einem Frontend. Die Anbindung an ein Frontend soll hier nur angedacht werden, um eine Grundlage für eine zukünftige Weiterentwicklung zu einem vollständigen System zu schaffen. Das vorgestellte System stellt somit nur die verarbeitende Schicht des Backends dar und trägt den Namen *Crepe* (*Complex rule-based event processing engine*).
Abbildung 1.2 verdeutlicht den generellen Ablauf des beschriebenen Systems noch einmal grafisch. Zuerst werden Daten aus einem externen System geladen. Zum Beispiel Daten von einem RSS-Feed oder von Sensoren. Die Daten werden anschließend in das interne Format umgewandelt und an die Algebra weitergegeben. Weiterhin kann hier eine Schnittstelle zum Frontend integriert werden, welches dann die Visualisierung der Daten und eine Interaktion mit dem Algebrasystem übernimmt. Auf die umgewandelten Daten werden dann die Regeln der Algebra angewendet und gegebenenfalls an ein externes System exportiert. Analog zum Import werden die Daten beim Export zunächst in das jeweilige externe Datenformat umgewandelt und danach in das externe System geladen.
Beim gesamten Softwareentwurf soll die Universalität des Systems im Vordergrund stehen. Diese soll durch eine leichte Erweiterbarkeit im Bezug auf Import-, Export- und Operatorkomponenten gewährleistet werden. Je leichter es ist, das System in eine vorhandene Softwareumgebung zu integrieren, desto mehr steigt die Bereitschaft der Anwender, dieses zu verwenden.
Für das Backend ist es daher nötig, die Benutzbarkeit für Softwareentwickler möglichst einfach zu gestalten, ohne dabei zu viel Einschränkungen zu machen. Dennoch soll langfristig das Gesamtsystem Anwender und nicht Softwareentwickler als Hauptzielgruppe haben. Dies soll bei der Konzeption berücksichtigt werden. Hierbei sind als Anwender Domänenexperten zu verstehen, die die Software dauerhaft einsetzen wollen. Eine gewisse Einarbeitungszeit kann daher in Kauf genommen werden.

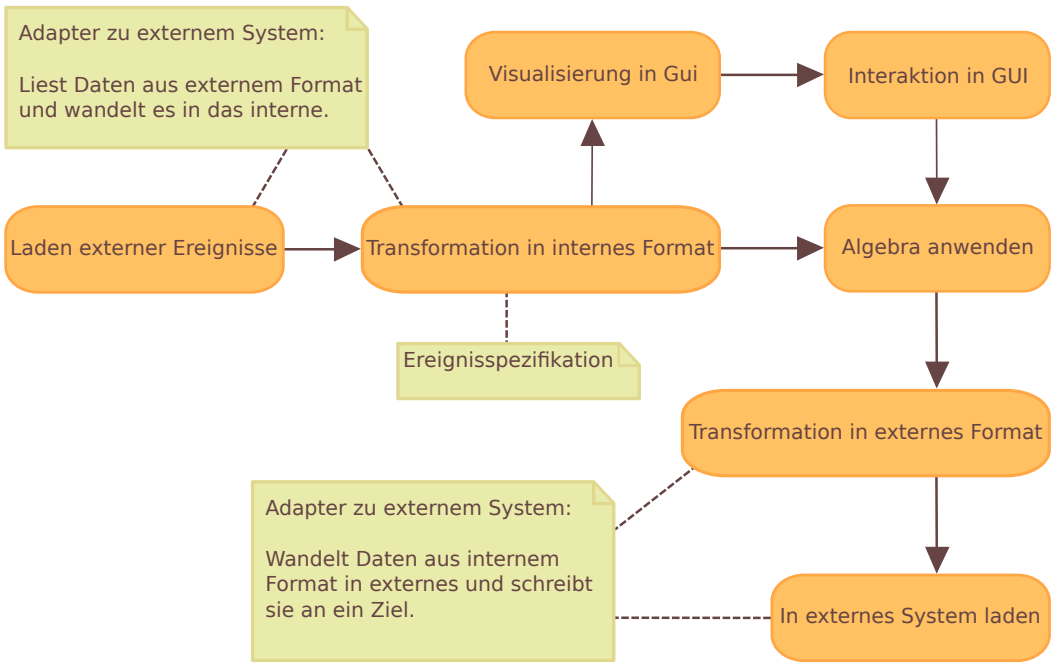

Abbildung 1.2.: Ablauf der Ereignisverarbeitung vom Import bis zum Export

1.3. Aufbau dieser Arbeit

Kapitel 2 geht auf verwandte Arbeiten ein und gibt einen Überblick, welche Ansätze für komplexe ereignisgesteuerte System bereits existieren und erläutert kurz, worin sie sich zu dem hier vorgestellten unterscheiden.
Kapitel 3 soll auf die Grundlagen ereignisgesteuerter Systeme eingehen und später verwendete Begriffe definieren. Weiterhin wird der allgemeine Aufbau von ereignisgesteuerten Systemen analysiert und die Anforderungen an ein universelles System, wie das hier vorgestellte *Crepe*, skizziert.
Kapitel 4 beschreibt die Algebra von *Crepe* und definiert ihre Operatoren und Fensterfunktionen.
Kapitel 5 erklärt den technischen Aufbau von *Crepe* und gibt Details zur Implementierung. Es werden dabei Entwurfsentscheidungen begründet und ein Überblick über die Schnittstellen des Systems gegeben.
Kapitel 6 demonstriert die Funktionalität der Implementierung anhand von Fallstudien aus zwei unterschiedlichen Anwendungsbereichen. Es wird auf die Beschaffenheiten der Daten eingegangen und geeignete Auswertungen herausgearbeitet, die mithilfe von *Crepe* durchgeführt und abschließend als Ergebnis bewertet werden. Weiterhin wird die

Durchführung eines kurzen Lasttests beschrieben, um zu zeigen, dass das System für praktische Aufgaben schnell genug arbeitet.
Kapitel 7 diskutiert die in dem Projekt gewonnenen Erfahrungen und gibt einen Ausblick auf zukünftige Erweiterungen von *Crepe*.
In der gesamten Arbeit sind wichtige Begriffe mit *kursiver Schrift* gekennzeichnet. Programmcode und Bestandteile der domänenspezifischen Sprache von *Crepe* sind in `nicht proportionaler Schrift` angegeben.

2. Verwandte Arbeiten

Die Spezifikation von Ereignissen und Regeln und deren Semantik wurden in der Forschung in vielen Bereichen untersucht. Im ersten Abschnitt soll ein kurzer Überblick über Arbeiten über ereignisgesteuerte Systeme gegeben und die dort vorgestellten Ansätze kurz umrissen werden.
Da im Mittelpunkt dieser Arbeit vor allem die praktische Umsetzung eines generischen ereignisgesteuerten Systems steht, sollen auch bereits existierende Softwaresysteme kurz betrachtet werden.

2.1. Forschungsarbeiten

Cugola und Margara [9] geben eine Übersicht über *komplexe Ereignisverarbeitung* und zeigen Gemeinsamkeiten zu anderen Themenfeldern, wie beispielsweise *Database Management Systems* (*DBMS*) auf. Weiterhin verallgemeinern sie Konzepte aus ereignisgesteuerten Systemen und geben einen Überblick über existierende Systeme.
In der Vergangenheit wurden Datenströme oft mit Hilfe von Datenbanksystemen verarbeitet [9], doch war deren Verarbeitungsgeschwindigkeit oft nicht ausreichend [35]. Dieser Umstand hat die Entstehung von *Data Stream Management Systems (DSMSs)* hervorgerufen [5, 2]. Während *Database Management Systems (DBMS)* Anfragen mit Hilfe einer relationalen Abfragesprache, wie z.B. SQL, einmalig ausführen, werten *DSMSs* diese kontinuierlich auf dem Datenstrom aus [27].
Active Database Systems erweitern *DBMS* um eine *ereignisgesteuerte Architektur* und können dadurch anhand von *Event-Condition-Actions (ECA)* auf Zustandsänderungen der Datenbanken reagieren [32, 22]. Im Bereich von *Active Databases* wurde die Erstellung von Regeln anhand von Ereignissen, auch im Zusammenhang mit *Composite Events* (vgl. Kapitel 3.2), ausgiebig erforscht [18, 17].
Viele der *DSMSs*-Systeme basieren auf Datenbanksystemen und verwenden deren Abfragesprachen [35, 38]. Die meisten der Systeme sind so entworfen, dass sie die Ausdruckskraft ihrer Sprachen zugunsten der Verarbeitungsgeschwindigkeit einschränken [35]. Weiterhin fehlt ihnen in der Regel die Möglichkeit, Auswertungen auf Ereignissen aus unterschiedlichen Quellen auszuführen [9].
Systeme für *komplexe Ereignisverarbeitung (Complex Event Processing)* [26] erweitern diese Systeme, so dass verschiedene Datenquellen berücksichtigt [12] und umfangreichere Auswertungen durchgeführt werden können [10]. Beispiele für solche *CEP*-Systeme sind

SASE [33], Cayuga [36] oder die unten vorgestellten Esper [11] (Abschnitt 2.2.2) und *StreamInsight* [8] (Abschnitt 2.2.1).
Voisard und Ziekow [38] beschreiben einen ähnlichen Ansatz, wie den in dieser Arbeit erläuterten. Ausgehend von einer allgemeinen Ereignisdefinition wird ein Framework namens *ARCHITECT* entwickelt, das sauber in einzelne Schichten aufgetrennt ist, so dass es sich leicht erweitern und pflegen lässt. Die dort als *Capturing Layer* bezeichnete Schicht ist für das Sammeln von Ereignissen aus verschiedenen Quellen zuständig, was oben als Import bezeichnet wurde. Die Schicht *Communication* verteilt Ereignisse innerhalb des Systems weiter. *Execution Layer* ist für die Auswertung der Ereignisalgebra verantwortlich und *Language Layer* beinhaltet eine Beschreibungssprache, mit der die Regeln der Ereignisalgebra festgehalten werden können.
Wu und Dube [39] beschreiben ein ereignisgesteuertes Framework namens *PLAN*, welches zur Modellierung und Spezifizierung von medizinischen Leitlinien dienen soll. Zwar reden die Autoren von einem generischen Framework, jedoch haben sie immer die Anwendungsdomäne der medizinischen Leitlinien im Fokus. Auch wenn einige der dort vorgestellten Konzepte sich verallgemeinern lassen ist es anders, als in dieser Arbeit, nicht die Absicht der Autoren ein universales ereignisgesteuertes System zu entwickeln.
Jung und Jones [23] stellen eine grafische Sprache zur Definition von Profilen (Regeln) vor. Sie behandeln dabei die Operatoren *Sequence*, *Composite conjunction*, *Composite disjunction*, *Selection* (vgl. Kapitel 4.2) und führen dabei für alle Operatoren eine Dauer ein, in der sie zugetroffen haben müssen. Das Ziel der grafischen Sprache zur Profilerstellung ist, es dem Anwender es einfacher zu machen Regeln zu erstellen. *Crepe* implementiert die Operatoren in textueller Form und setzt die Dauer der Operatoren in Form einer Fensterfunktion um – die grafische Sprache könnte als leicht in ein später implementiertes Frontend integriert werden.
Voisard und Hinze [22] stellen eine parametrisierte Ereignisalgebra namens *EVA* (*EVent Algebra*) vor und definieren die Semantik komplexer Ereignisse so, dass sie leicht für verschiedene Anwendungsfälle angepasst werden können. Um die Algebra an externe Systeme anzubinden, führen die Autorinnen einen *Meta Service* ein. Dieser stellt eine Zwischenschicht, zwischen Algebra und externen Systemen dar. Der *Mediator Server* kümmert sich um Transformation von Profilen und Ergebnissen aus den Auswertungen. So können vorhandene ereignisgesteuerte Systeme ihre Profile in ihrem eigenen Format an den *Meta Service* senden und bekommen die Ergebnisse der Auswertung erneut im eigenen Format zurück.

2.2. Softwaresysteme

Hierfür wird exemplarisch auf die Systeme *Esper* und *StreamInsight* eingegangen, da sie einen recht ähnlichen Ansatz verfolgen. Weiterhin wird kurz diskutiert, worin sie sich zu dem hier vorgestellten *Crepe* unterscheiden.

2.2.1. Microsoft StreamInsight

Die Firma Microsoft bietet unter dem Namen *StreamInsight* eine Softwarelösung für komplexe Ereignisverarbeitung an [8]. *StreamInsight* verwendet Microsofts *.NET-Framework* und richtet sich an Softwareentwickler. Die Software ist auf einen hohen Datendurchsatz und eine geringe Latenz optimiert und eignet sich daher besonders für die Entwicklung von Systemen, bei denen Geschwindigkeit eine große Rolle spielt.
Die *StreamInsight* zugrunde liegende Architektur ähnelt sehr stark der von *Crepe* (vergleiche Kapitel 5.1). StreamInsight ist als Server bzw. Backend konzipiert und besteht aus ähnlichen Komponenten:
Ereignisquellen werden mit so genannten *Input-Adaptern* in ein internes Ereignisformat umgewandelt, danach von der *StreamInsight Engine* verarbeitet und bei einer Ausgabe mittels *Output-Adaptern* in ein externes Format umgewandelt und anschließend an ein Ziel geschrieben. Die *StreamInsight Engine* entspricht dem Algebrateil eines ereignisgesteurten Systems und verwendet für die Definition der Regeln eine domänenspezifische Sprache namens *LINQ*, die recht ähnlich zu SQL ist.
Microsofts Ereignisspezifikation ist sehr allgemein gehalten: Ereignisse bestehen aus einem Zeitpunkt oder wahlweise einem Zeitinterval und enthalten eine beliebige *.NET*-Datenstruktur, die den Inhalt eines Ereignisses beschreibt.
Der Hauptunterschied zum Konzept von *Crepe* liegt darin, dass *StreamInsight* primär ein Framework für Softwareentwickler ist, während *Crepe* zumindest langfristig als Anwendersoftware konzipiert ist. Weiterhin wurde *Crepe* in Java entwickelt und läuft somit auf vielen Plattformen, während *StreamInsight* nur unter Windows läuft.

2.2.2. Esper

Die Firma EsperTech bietet mit der Software *Esper* [11] eine sehr umfassende Softwarekomponente für komplexe Ereignisverarbeitung an. Esper ist eine Java-Software und existiert zusätzlich noch in einer zweiten Variante namens *NEsper* für *.NET*.
Ähnlich wie *StreamInsight* verwendet *Esper* eine SQL-ähnliche eigene domänenspezifische Sprache namens *EPL* (*Event Processing Language*)). Das Ereignismodell entspricht in etwa dem in dieser Arbeit vorgestellten Modell, mit dem Unterschied, dass bei Attributen noch unterschieden wird, ob innerhalb eines Attributtyps auf weitere Unterelemente zugegriffen werden kann. Beispielsweise unterscheidet Esper, ob ein Attribut ein einfaches Objekt, eine Liste oder eine *Map* ist, in der man durch Schlüssel auf Unterelement zugreifen kann.
Esper ist im Gegensatz zu *Crepe* kein System, das sich um alle Aufgaben eines ereignisgesteuerten Systems kümmert. So gibt es in *Esper* keine Einlese- oder Ausgabekomponenten, da es sich lediglich als Komponente für die Ereignisverarbeitung, also den Algebrateil, versteht.

3. Grundlagen ereignisgesteuerter Systeme

Ereignisgesteuerte Systeme werden zwar zur Lösung verschiedenster Problemstellungen eingesetzt, dennoch weisen sie viele Gemeinsamkeiten auf. In diesem Kapitel sollen daher die grundlegenden Elemente von ereignisgesteuerten Systemen beschrieben und definiert werden. Gewisse Problemstellungen treten in allen Systemen gleichermaßen auf, weshalb diese Gemeinsamkeiten im Folgenden herausgearbeitet und näher erläutert werden. Weiterhin werden unterschiedliche Lösungsansätze für die Aufgaben von ereignisgesteuerten Systemen erörtert.

3.1. Ereignis

Ereignisse sind das zentrale Element von ereignisgesteuerten Systemen. Wie bereits erwähnt, ist ein Ereignis im Allgemeinen das Auftreten eines beobachteten Geschehens[26] bzw. eine Zustandsänderung eines Objektes in einem (Computer-)System[6, 28]. Eine Zustandsänderung geschieht zu einem bestimmten Zeitpunkt [30] und wird durch eine Veränderung von Eigenschaften (*Attributen*) des beobachteten Objektes verursacht. Ein Ereignis lässt sich daher wie folgt definieren (vgl. [22]):

Definition 1. *Ein Ereignis ist eine Zustandsänderung eines Objektes von Interesse zu einem bestimmten Zeitpunkt, d.h. die Veränderung von mindestens einem Attributwert.*

Beispielsweise bestünde ein Ereignis bei einem Temperatursensor aus einem Zeitpunkt (etwa in Millisekunden) wann eine Temperatur gemessen wurde und aus dem Attribut *Temperatur*. Attribute können unterschiedlichen Typs sein und dementsprechend einen anderen Wertebereich haben. Beim Temperatursensor etwa der Temperaturbereich, den er messen kann. Überdies kann ein Ereignis auch mehrere Attribute enthalten, die das Geschehen beschreiben. Beim bisherigen Beispiel wäre etwa denkbar, dass der Sensor neben der Temperatur noch die Luftfeuchtigkeit misst.
Es ist möglich, dass Ereignisse mit gleichen Attributen und gleichen Attributwerten zum selben Zeitpunkt auftreten [41]. Gerade bei geringer zeitlicher Auflösung ist dies leicht möglich. Das ereignisgesteuerte System muss jedoch, bei manchen Anwendungsbereichen, in der Lage sein, zwischen den Ereignissen zu unterscheiden. Hierfür wird eine eindeutige ID für jedes Ereignis benötigt [38]. Beim vorigen Temperatursensorbeispiel wäre es z.B. möglich, dass mehrere Sensoren gleichzeitig ihre Temperaturmessungen melden. Bei hoher zeitlicher Auflösung wäre es relativ unwahrscheinlich, dass der Zeitpunkt

sich nicht unterscheidet, übermitteln sie jedoch nur einmal pro Minute oder noch seltener ihre gemessene Temperatur kann eine Gleichzeitigkeit von Ereignissen leicht vorkommen. In manchen Anwendungsdomänen kann es notwendig sein, dass Ereignisse auftreten, obwohl sich kein Attributwert eines Objektes verändert hat, wie z.B. beim periodischen Auslesen eines Sensors. Hat sich der Sensorwert seit dem letzten Auslesen nicht verändert, hat sich dennoch der Zeitpunkt des Auslesens geändert und wird daher dem ereignisgesteurten System als neues Ereignis mit verändertem Zeitpunkt mitgeteilt [21].

3.2. Basisereignisse und Composite Events

Ereignisse, die in ein ereignisgesteuertes System von Außen eingespeist werden, wie z.B. Sensordaten, werden im Folgenden als *Basisereignisse* bezeichnet.
Mit Hilfe einer Ereignisalgebra kann man Basisereignisse kombinieren und daraus neue Ereignisse erzeugen [22, 34]. Solche zusammengesetzten Ereignisse (*Composite Events*) lassen sich wiederum erneut mit Basisereignissen oder anderen zusammengesetzten Ereignissen kombinieren.

3.3. Ereignisraum

Ereignisgesteuerte Systeme beobachten Ereignisse, um diese dann zu verarbeiten. Durch eine Ereignisalgebra können Composite Events gebildet werden, die dann ebenfalls vom System beobachtet werden. Die Menge aller Ereignisse, die von einem System beobachtet werden können, nennt sich Ereignisraum (vgl. [22]):

Definition 2. *Der Ereignisraum ist die Menge aller von einem System beobachtbaren Ereignisse. Der Ereignisraum ist die Vereinigung der Menge aller Basisereignissen mit der Menge aller Composite Events.*

3.4. Ereignisstrom

Ereignisgesteuerte Systeme beobachten Ereignisse und werten diese nacheinander aus, wodurch ein *Ereignisstrom* entsteht [25]. Die zeitliche Ordnung der Auswertung muss dabei nicht zwingenderweise mit der Ordnung der Zeitpunkte der Ereignisse übereinstimmen. So können Sensoren Werte mit einem zugehörigen Zeitstempel ausgeben und ein System beispielsweise zwei Sensoren *s1* und *s2* überwachen, wobei die Sensorwerte von *s1* zeitlich vor denen von *s2* ausgewertet wurden. Im ereignisgesteuerten System kommen jedoch die Werte von *s2* vor denen von *s1* an. Dieser Umstand muss bei der Erstellung von Profilen berücksichtigt werden, da es sonst zu unerwartetem Verhalten kommen kann.

3.5. Profile und Benachrichtigungen

Ein Profil (engl. *profile*) ist definiert als Bedingung (bzw. Regeln), die kontinuierlich auf die im Ereignisstrom eintreffenden Ereignisse geprüft wird [22, 20]. Trifft die Regel zu, so wird eine *Benachrichtigung* (engl. *notification*) ausgelöst. Eine Benachrichtigung kann an ein externes System gehen oder auch als neues Ereignis betrachtet werden, welches wiederum in den Ereignisstrom gesendet wird und somit ebenfalls ausgewertet werden kann.

Definition 3. *Ein Profil ist eine Bedingung (bzw. Regel), die kontinuierlich auf im Ereignisstrom eintreffende Ereignisse geprüft wird. Beim Erfüllen der Bedingung werden zugehörige* Benachrichtigungen *ausgelöst.*

3.6. Ereignisselektion

Ereignisse treten zeitlich gesehen hintereinander oder gleichzeitig auf [41]. Ein ereignisgesteuertes System, arbeitet auf einem Ereignisstrom und reagiert auf Ereignisse, indem es die vorhandenen Regeln auf Treffer überprüft. Ein neu eingetroffenes Ereignis kann also bewirken, dass eine zuvor noch nicht zutreffende Regel nun einen Treffer ergibt.
Eine Regel besteht aus Operatoren die jeweils einen Zustand haben, der angibt, ob sie zutreffen oder ob sie nicht zutreffen. Da die Zustände durch eintreffende Ereignisse verändert werden, besteht eine Zuordnung zwischen dem Ereignis und dem Operator. Prüft beispielsweise ein Operator `attribute("Temperatur")` auf das Vorhandensein eines Attributs mit dem Namen “Temperatur” (siehe 4.2.3) und es trifft im Ereignisstrom ein Ereignis, mit dem Attribut “Temperatur” und dem Wert 25 ein, so lässt sich dieses Ereignis zum `attribute`-Operator als Treffererereignis zuordnen.
Operatoren, die direkt auf Ereignissen arbeiten, wie etwa der genannte `attribute`-Operator, behalten für gewöhnlich ihren Zustand bei. Beim Prüfen auf das Vorhandensein eines Attributs heißt das also, dass der `attribute`-Operator im Trefferzustand bleibt, sobald das erste passende Ereignis eintrifft. Erst wenn eine komplette Regel zutrifft werden die Zustände zurückgesetzt, so dass die Auswertung der Operatoren von vorne beginnt.
Um dies noch mal an einem Beispiel zu verdeutlichen: Nimmt man den `attribute`-Operator und erstellt eine Regel `and(attribute("Temperatur"), attribute("Luftfeuchtigkeit"))`, wobei `and` ein logisches Und darstellt (siehe Kapitel 4.2.4). Die Regel trifft also zu, sobald das Attribut “Temperatur” und das Attribut “Luftfeuchtigkeit” im Ereignisstrom auftreten. Dies kann sowohl innerhalb eines einzigen Ereignisses sein, als auch in zwei verschiedenen Ereignissen.
Betrachtet man nun den Fall, dass die Attribute in zwei verschiedenen Ereignissen vorkommen und das Ereignis *e1*, das das Attribut “Temperatur” enthält, vor dem Ereignis

e2 auftritt, dass das Attribut "Luftfeuchtigkeit" enthält. Beim Auftreten von *e1* geht der `attribute("Temperatur")`-Operator in einen zutreffenden Zustand über.
Treten nun Ereignisse auf, bei denen das Attribute *Temperatur* nicht vorkommt, so trifft der Operator zwar auf diese Ereignisse nicht zu, behält aber seinen Trefferzustand bei. Erst wenn *e2* auftritt, geht auch der `attribute("Luftfeuchtigkeit")`-Operator in den Trefferzustand über und somit ist die gesamte Regel erfüllt. Um zu vermeiden, dass die Regeln von diesem Zeitpunkt an für immer erfüllt ist und somit keine weiteren Ereignisse mehr überprüft werden, wird die Regel wieder in ihren Grundzustand zurück versetzt.
Für den Fall, dass zwischen *e1* und *e2* nur Ereignisse auftreten, für die der Operator `attribute("Temperatur")` nicht zutrifft, ist die Zuordnung des Operators zum Treffereignis *e1* eindeutig. Anders sieht es jedoch aus, sobald zwischen *e1* und *e2* Ereignisse auftreten, die ebenfalls für `attribute("Temperatur")` zutreffen.
Dann stellt sich die Frage, welches Ereignis dem Operator zugeordnet wird und wie die Regel darauf reagiert. Generell gibt es hierfür drei verschiedene Ansätze, die je nach Anwendungsfall ihre Vor- und Nachteile haben [22]. Diese sollen im Folgenden näher besprochen werden.

3.6.1. Erster Treffer

Ein Ansatz ist es, nur das erste Ereignis, welches einen Trefferzustand auslöst, zu nehmen. Alle weiteren zutreffenden Ereignisse werden ignoriert. Dieser Ansatz eignet sich besonders, bei Problemen, bei denen ein neues Ereignis keine neue Information liefert z.B. wenn beim Auslesen eines Sensors nur auf das Überschreiten eines Grenzwertes geprüft wird.

3.6.2. Letzter Treffer

Eine weitere Möglichkeit ist es, immer das letzte Ereignis, welches einen Trefferzustand auslöst, zu nehmen. Das Treffereignis wird also immer mit dem Ereignis überschrieben, welches zuletzt auf den Operator zugetroffen hat. Dieses Vorgehen eignet sich beispielsweise für Systeme, die immer die aktuellsten Informationen auswerten. So werten z.B. Temperaturregelungssysteme in der Regel nur den letzten und somit aktuellsten wert eines Sensors aus.

3.6.3. Jeder Treffer

Weiterhin kann auch jedes Ereignis, welches einen Trefferzustand auslöst, betrachtet werden. Dies stellt eine sehr aufwändige Methode dar, da für jede Trefferkombination eine Auswertung erfolgen muss. Sie eignet sich daher nur für Systeme, wie z.B. Sicherheitssysteme, bei denen jeder Treffer explizit berücksichtigt werden muss.

3.7. Aufbau ereignisgesteuerter Systeme

Zwar können ereignisgesteuerte System im Detail sehr verschieden sein, dennoch ist deren prinzipieller Aufbau immer gleich[9, 38]: Ereignisse werden aus einer Quelle ausgelesen, anschließend ausgewertet und wenn definierte Bedingungen erfüllt sind, eine Aktion ausgeführt. Verwendet man die oben eingeführten Definitionen, so heißt das, dass durch das Einlesen von Quellen ein Ereignisstrom erzeugt wird, der auf das Zutreffen von Profilen geprüft wird und bei einem Treffer die im Profil hinterlegten Benachrichtigungen ausgelöst werden. Eine Benachrichtigung kann entweder ein neues Ereignis im Ereignisstrom erzeugen oder aber aus dem System herausgehen, wie beispielsweise eine Datei schreiben oder eine Aktion in einem Regelungssystem hervorrufen [38]. Als Gegenstück zur Quelle soll ein externes Ausgabeziel einer Benachrichtigung im Folgenden als Senke bezeichnet werden (vgl. Abbildung 1.2).
Betrachtet man diesen allgemeinen Aufbau, der allen ereignisgesteuerten Systemen gemein ist, so stellt man leicht fest, dass die Unterschiede der einzelnen Systeme vor allem im Ereignisformat, im Auslesen aus Quellen und in der Ausgabe an Senken besteht [38]. Will man ein universal einsetzbares System haben, so sind daher zwei Dinge notwendig: Ein einheitliches Ereignisformat, welches möglichst alle Anwendungsdomänen berücksichtigen kann [41], sowie ein Trennen der Ereignisalgebra vom Format der Quellen und Senken [9].
Die Gemeinsamkeiten der Systeme bestehen vor allem in den Auswertungen: Zwar gibt es domänenspezifische Auswertungen, dennoch kommen gewisse Operatoren in vielen Systemen immer wieder vor. Beispielsweise werden logische Verknüpfungen wie Konjunktionen, Disjunktionen oder Negationen in vielen Bedingungen verwendet [21, 22]. Hat man ein einheitliches Ereignisformat, kann man Operatoren definieren, die allgemein mit Ereignissen umgehen können, ohne dabei auf domänenspezifische Eigenschaften angewiesen zu sein. Diese Operatoren können dann für die Definition von Bedingungen in Profilen in allen Domänen benutzt werden. So lässt sich auf Dauer ein nicht unerheblicher Implementierungsaufwand einsparen.
Dieser Vorteil lässt sich auch auf domänenspezifische Operatoren ausweiten: Selbst innerhalb einer Anwendungsdomäne gibt es oft viele verschiedene Systeme mit proprietären Ereignisformaten. Dennoch werden die gleichen Arten von Auswertungen vorgenommen, was bedeutet, dass die einzelnen Operatoren jeweils neu für das verwendete Ereignisformat implementiert werden müssen. Verwendet man ein einheitliches Ereignisformat, so lassen sich zumindest innerhalb einer Domäne die Operatoren wiederverwenden.
Analysiert man das Einlesen von Quellen genauer, so kann man zwei getrennte Schritte erkennen [35]. Zunächst werden Daten im Urprungsformat der Quelle ausgelesen und danach daraus Ereignisse erzeugt, so dass sich ein Ereignisstrom ergibt. Das Format der Daten aus der Quelle muss zwar zur Weiterverarbeitung nicht zwingenderweise verändert werden, dennoch ist dies zwecks besserer Handhabbarkeit oft der Fall. Liest man beispielsweise Daten aus einem Sensor aus, so ist deren Format oft abhängig von Hard-

warebegebenheiten und wird in der Regel erst in eine Datenstruktur überführt, mit der dann die eigentliche Verarbeitung stattfindet. Will man nun ein einheitliches Ereignisformat erreichen, so ist dieser Schritt sogar unumgänglich.
Im Allgemeinen besteht das Einlesen also aus zwei Teilen: dem reinen Laden von Daten und das Umwandeln vom Ursprungsformat in das intern verwendete Ereignisformat [9]. Hat man zwischen dem Laden und dem Umwandeln eine einheitliche Schnittstellen, so kann weiterer Implementierungsaufwand eingespart werden, denn die Ladekomponenten und Umwandlungskomponenten lassen sich beliebig kombinieren. Durch eine Parametrisierung der Komponenten lässt sich die Wiederverwendbarkeit weiter steigern.
Dies lässt sich an einem Beispiel leicht verdeutlichen: Hat man etwa zwei Ladekomponenten, eine die lokale Dateien und eine die Dateien von HTTP-Servern lädt und eine Umwandlungskomponente, die CSV-Dateien umwandelt. So kann man CSV-Dateien lokal oder per HTTP laden, muss die Umwandlungskomponente aber nicht zweimal implementieren. Eine Parametrisierung wäre bei der Ladekomponente etwa, dass man bei einer lokalen Datei deren Pfad oder bei der HTTP-Datei die URL und eventuell andere Optionen wie Timeout oder Benutzername und Passwort angeben kann. Bei der CSV-Umwandlung wären als Parameter etwa Trennungszeichen und Zeichensatz denkbar.
Genauso wie beim Einlesen lässt sich dieses Prinzip auch bei der Ausgabe umsetzen. Wird durch das Zutreffen eines Profils eine Ausgabe aus einem System verursacht, so wird zunächst die Treffererereignisse in das gewünschte Ausgabeformat gebracht und anschließend an ein Ziel geschrieben. Beispielsweise könnte ein XML-Dokument, das aus Daten der Treffererereignisse besteht, auf einen FTP-Server geladen werden. Hat man Zwischen Umwandlungs- und Schreibkomponente eine einheitliche Schnittstelle, so sind auch hier beliebige Kombinationen der Komponenten möglich.
Das in dieser Arbeit vorgestellte komplexe ereignisgesteuerte System *Crepe* setzt die genannten Ideen konsequent um. Im nächsten Kapitel soll das Algebrasystem von *Crepe* im Detail näher beschrieben werden, während in Kapitel 5 auf die technischen Einzelheiten der Implementierung näher eingegangen wird.

4. Algebra von *Crepe*

Für die Verarbeitung von Ereignissen ist es wichtig, genau festzulegen, wie sich ein Algebrasystem verhält. Es ist wichtig, dass die Bedeutung der verwendeten Operatoren, sowie die äußeren Begebenheiten, wie etwa das Auswählen von Ereignissen (Ereignisselektion) oder eventuell vorhandene Fensterfunktionen genau definiert sind. Es soll daher erläutert werden, welche Ansätze für das im Rahmen dieser Arbeit implementierte System *Crepe*, verwendet wurden. In diesem Kapitel sollen zunächst die Grundlagen der Algebra theoretisch erörtert werden, während im anschließenden Kapitel 5 auf die Implementierungsdetails eingegangen werden soll. Da die hier definierte Algebra jedoch eng mit der Implementierung zusammenhängt, ist es an manchen Stellen nicht vermeidbar, auf technische Notwendigkeiten einzugehen.

4.1. Allgemeines

In *Crepe* werden Profile in Form einer Regel und mindestens einer Benachrichtigung definiert. Trifft eine Regel zu, so werden alle Benachrichtigungen ausgelöst und der Ereignisstrom weiter auf das Zutreffen des Profils geprüft. Der genaue Aufbau von Regeln wird in Kapitel 4.2 genauer beschrieben.
Crepe arbeitet verwendet bei der Ereignisselektion immer den *letzten Treffer* (vgl. Kapitel 3.6.2), d.h. Treffereignisse werden grundsätzlich überschrieben. Dieser Ansatz wurde gewählt, da er für viele Anwendungsszenarien geeignet ist und die Umsetzung von mehreren Selektionsvarianten den Rahmen dieser Arbeit gesprengt hätte. Dennoch ist es möglich diese in Zukunft noch in das System zu integrieren.

4.2. Operatoren

Zentraler Bestandteil eines Algebrasystems sind dessen Operatoren. Sie dienen zur Definition von Regeln und bilden somit die Grundlage von Profilen. Von den Operatoren hängt ab, welche Art von Auswertungen mit einem System gemacht werden können.
Da in dieser Arbeit das Ziel verfolgt wurde, ein System zu erschaffen, dass möglichst jede Art von Auswertung durchführen kann, wurde es so gestaltet, dass beliebige Operatoren hinzugefügt werden können, um den Funktionsumfang des Systems zu erweitern. Es gibt jedoch Operatoren, die bei den meisten Auswertungen vorkommen und nicht Domänenspezifisch sind. Diese Grundoperatoren wurden in dem im Rahmen dieser Arbeit

erstellten Softwaresystem implementiert und sollen daher im Folgenden näher besprochen werden.

4.2.1. Grundlegendes

Prinzipiell können Operatoren auf Ereignissen oder auf anderen Operatoren arbeiten. Arbeiten Operatoren auf Ereignissen so werten sie entweder deren Attribute oder deren Zeitstempel aus.
Wenn sie auf anderen Operatoren arbeiten, dann arbeiten sie auf dem Ergebnis von deren Auswertung. Regeln sind immer boolsch, d.h. sie treffen entweder zu oder nicht zu. Betrachtet man eine Regel als Operatorbaum, bei dem der äußerste Operator an der Wurzel steht, so muss dieser ebenfalls boolsch sein. Unterhalb im Baum ist es möglich, dass Operatoren auch andere Rückgabewerte haben, um beispielsweise numerische Auswertungen durchzuführen. Die hier vorgestellten Operatoren sind alle so gestaltet, dass sie auch als Wurzel im Operatorbaum und somit als Regel dienen können.
Arbeiten Operatoren auf Ereignissen, dann stellen sie im Operatorbaum Blätter dar. Für die Auswertung von Attributen ist immer die Angabe eines Attributnamens nötig, um das auszuwertende Attribut des Ereignisses zu finden. Da viele Auswertungen von Attributen ebenfalls boolsch sind, entsteht ein neues Problem: Wenn ein Ereignis einen Attributwert auswertet, das Ereignis jedoch dieses Attribut nicht besitzt, so ist das Ergebnis zunächst unklar.
Prüft ein Operator beispielsweise, ob ein Attribut namens *Temperatur* einen Wert kleiner 25 hat und das gerade eintreffende Ereignis hat kein solches Attribut, sondern beispielsweise nur das Attribut *Luftfeuchtigkeit*, so gäbe es bei einer rein boolschen Auswertung für den Rückgabewert des Operators nur die Möglichkeiten wahr oder falsch. Eine mögliche Interpretation wäre anzunehmen, dass das Attribut dann nicht kleiner 25 ist und der Operator somit falsch zurück gibt. In den meisten Fällen ist dies jedoch nicht das gewünschte Verhalten. Vielmehr möchte man, dass der Operator das Ereignis einfach ignoriert und nur Ereignisse auswertet, die auch das auszuwertende Attribut besitzen. Daher gibt es für Operatoren, die auf Attributen arbeiten noch die Option Ereignisse zu ignorieren und somit auf diesen keinen boolschen Rückgabewert zu haben. Dieses Verhalten ist vor allem deswegen sinnvoll, weil Ereignisströme oft gemischte Ereignistypen enthalten, jedes Profil und somit all dessen Operatoren, aber auf jedes Ereignis angewendet werden.

4.2.2. Referenzierung

In *Crepe* ist es möglich, Operatoren zu referenzieren. Dadurch ist es möglich, auf das Treffereignis des jeweiligen Operators zuzugreifen und Auswertungen mit diesem Ereignis durchzuführen. Viele der unten beschriebenen Operatoren bieten daher die Möglichkeit

mit Operatorreferenzen als Operanden zu arbeiten. Sie lesen dann zunächst das letzte Treffereignis aus und benutzen dann dies als eigentlichen Operanden für die Operation. Der Umweg zum Treffereignis über die Referenz auf den Operator ist aus technischer Sicht aus zwei Gründen sinnvoll: Zum einen wurde *Crepe* in Java programmiert und die Operatorargumente würden bereits bei der Profildefinition ausgewertet werden. Möchte man also auf Treffereignissen arbeiten, so sind diese offensichtlich bei der Profildefinition noch nicht vorhanden. Die Operatorfunktion würde aber mit der Referenz arbeiten, die bei Profildefinition übergeben wurde und daher dauerhaft mit einer Nullreferenz arbeiten. Operatoren sind jedoch bereits bei der Profildefinition vorhanden und haben Zugriff auf ihre Treffereignisse. So kann ein Operator das benötigte Treffereignis bei Bedarf vom als Parameter übergebenen Operator auslesen.
Zum anderen erleichtert die Referenzierung auf den Operator die Syntax in der internen Java-DSL, die für Profildefinitionen bereitgestellt wurde. Per Inline-Definition lässt sich der der Operator angeben, dessen Treffereignis referenziert werden soll. Dies vermeidet es, dass weitere Funktionen zur Referenzierung in einer Regeldefinition angegeben werden müssen. Um die Syntax weiter besser leserlich zu machen wurde eine abstrakte Operatorklasse namens `Match` eingeführt, die zur Referenzierung verwendet werden soll. Somit sieht dann eine Referenzierung beispielsweise wie folgt aus:

```
Match m1;
and(m1 = attribute("Temperatur"), equals("Temperatur", m1, 12));
```

Die Bedeutung der im Beispiel verwendeten Operatoren wird weiter unten genauer beschrieben. Die angegebene Regel soll hier nur der Verdeutlichung der Referenzierung dienen. Sie besagt, dass überprüft werden soll, ob ein Attribut namens *Temperatur* existiert (`attribute`) und (`and`) gleichzeitig, das Ereignis, für das dies zutrifft (`m1`) der Wert von *Temperatur* gleich (`equals`) zwölf ist.

4.2.3. Basisoperatoren

Die folgenden Operatoren dienen in *Crepe* zur grundlegenden Erstellung von Regeln und passen in keine der anderen Kategorien.

Attributexistenz

Der Attribut-Operator (`attribute`) hat einen Attributnamen als Argument und trifft zu, sobald ein eintreffendes Ereignis ein Attribut mit diesem Namen hat.

Gleichheit

Der Gleichheitsoperator (`equal`) kann entweder zwei oder drei Parameter haben. Das erste Argument ist immer der Name des Attributs, auf das der Operator angewandt

werden soll. Im Fall von zwei Parametern trifft der Operator zu, sobald ein eintreffendes Ereignis ein Attribut mit dem angegebenem Namen hat und dessen Wert gleich dem zweiten Parameter ist. In der zweiten Variante sind die beiden zusätzlichen Parameter Operatoren. In diesem Fall trifft der Operator zu, sobald die Treffererereignisse der Operatoren gleich sind.

Zähloperator

Der Zähloperator (`count`) kann zum Zählen von Ereignissen verwendet werden. Er hat einen Operator und eine Zahl n als Argument und erhöht einen Zähler jedes mal um eins, wenn der Operator zutrifft. Erreicht der Zähler den Wert des zweiten Arguments, so trifft der `count`-Operator zu und enthält als Treffererereignis das Ereignis durch das der Zähler gleich n wurde.

forEvent-Operator

Der `forEvent`-Operator unterscheidet sich grundlegend von den anderen Operatoren. Er führt keine Auswertung auf einem Ereignis durch, sondern überschreibt lediglich das zu verarbeitende Ereignis mit dem Treffererereignis eines Operators. Bei Operatoren, die direkt auf Ereignissen arbeiten, werden normalerweise die Auswertungen auf dem gerade eintreffenden Ereignis durchgeführt. Möchte man jedoch das Ereignis, das für einen anderen Operator zugetroffen hat, auf weitere Bedingungen prüfen, so kann man den `forEvent`-Operator verwenden, der eine Referenz auf einen Operator als Argument hat. Dies erspart bei Operatoren die auf Ereignissen arbeiten, dass man diesen Sonderfall für jeden Operator erneut implementieren muss. Er stellt also eine Möglichkeit der Referenzierung auf Treffer dar.

match-Operator

Mit dem `match`-Operator hat ein Ereignis als Parameter und erzeugt ein Operatorobjekt, welches das übergebene Ereignis als Treffererereignis hat. Der Operator selbst führt keinerlei Auswertung durch sondern repräsentiert lediglich ein Ereignis in Form eines Operators. Dies hat den Zweck, dass das Ereignis an Operatoren übergeben werden kann, die einen Operator als Parameter erwarten. Vor allem durch Verwendung von Operatorreferenzierung und den im folgenden Abschnitt erläuterten `setMatchTo`-Operator macht dies weitere Auswertungen möglich, was im Folgenden auch anhand eines Beispiels deutlich wird.

setMatchTo-Operator

Ähnlich wie der `forEvent`-Operator ist auch der `setMatchTo`-Operator ein Hilfsoperator, der selbst keine Auswertung durchführt, sondern ein Operator, der die Referenzierungs-

möglichkeiten erweitert. Der `setMatchTo`-Operator hat zwei Operatoren als Parameter. Der erste Operator wird ausgeführt und falls er einen Treffer hat, wird dessen Treffererereignis als Treffererereignis des zweiten Operators gesetzt. Dadurch lassen sich die Treffererereignisse von referenzierten Operatoren überschreiben und die Referenz kann gleichzeitig als Parameter in dem Operator verwendet werden, der das Treffererereignis überschreibt. Um das an einem einfach Beispiel zu verdeutlichen:

```
Match m1 = match(event("Temperatur", 100.0));
profile(setMatchTo(less("Temperatur", m1), m1),
        compositeEventNotification("Treffer", "Neues Temperaturtief!"));
```

Zunächst wird hier der `match`-Operator verwendet, um einen Operator mit einem Treffererereignis zu erzeugen, das ein Attribut namens Temperatur und den Wert 100.0 hat. Dieser Operator wird außerhalb des Profils definiert und wird somit nur beim Laden der Profildatei ausgeführt. Er dient lediglich dem Zweck, dass beim ersten Aufruf von `less("Temperatur", m1)` die Referenz `m1` keine Nullreferenz ist und ein Operator mit einem Treffererereignis existiert.
Der `setMatchTo`-Operator wertet nun bei jedem eintreffenden Ereignis den `less`-Operator aus und falls dieser einen Treffer hat, überschreibt er das Treffererereignis von `m1` mit diesem Ereignis. Da der `less`-Operator immer genau dann zutrifft, wenn ein Ereignis aus dem Ereignisstrom im angegebenen Attribut (hier *Temperatur*) kleiner ist als der Wert des ersten Parameters, löst das obige Beispielprofil immer eine Benachrichtigung aus, wenn ein neues Temperaturminimum erreicht wird.

4.2.4. Logische Operatoren

Mit logischen Operatoren andere Operatoren so verknüpft, dass sich daraus komplexere Regeln ergeben. Sie arbeiten also nicht direkt auf Ereignissen sondern auf anderen Operatoren.

Und

Der Und-Operator (`and`) hat zwei Operatoren a und b als Argumente und trifft zu, wenn sowohl a, als auch b zutrifft. Die zeitliche Reihenfolge spielt dabei keine Rolle, d.h. der Operator trifft zu, sowohl wenn a vor b als auch wenn b vor a auftritt (vgl. [22]).

Oder

Der Oder-Operator (`or`) hat zwei Operatoren a und b als Argumente und trifft zu, wenn a oder b oder a und b zutreffen. Auch hier ist die zeitliche Reihenfolge, ob a oder b zuerst zutrifft, egal (vgl. [22]).

Exklusives Oder

Der exklusive Oder-Operator (`xor`) hat ebenfalls zwei Operatoren a und b als Argumente und trifft zu, wenn entweder a oder b, aber nicht a und b zusammen zutreffen. Trifft also a oder b auf, so trifft der Operator solange zu, bis das jeweils noch nicht zutreffende Argument zutrifft.

Negation

Der Negationsoperator (`not`) hat einen Operator a als Argument und trifft zu, solange a nicht zutrifft (vgl. [22]).

4.2.5. Zeitliche Operatoren

Zeitliche Operatoren arbeiten auf den Zeitstempeln der Ereignisse. Es ist ein Ereignis also *vor* einem anderen Ereignis, wenn dessen Zeitstempel weiter in der Vergangenheit liegt, als der des anderen. Die Reihenfolge im Ereignisstrom spielt also für das Zutreffen eines zeitlichen Operators keine Rolle (vgl. [34]). Zeitliche Operatoren können entweder das aktuelle Ereignis mit einem absoluten Zeitpunkt vergleichen oder die Treffererereignisse zweier Operatoren miteinander vergleichen, um beispielsweise herauszufinden, ob ein Treffer vor einem anderen Treffer war.

Vorher

Der Vorher-Operator (`before`) hat entweder einen absoluten Zeitpunkt als Argument und trifft dann zu, wenn ein Ereignis eintrifft, das vor diesem Zeitpunkt lag oder es hat zwei Operatoren als Argument. In diesem trifft er dann zu, wenn der Zeitpunkt des Treffererereignisses des ersten Operators vor dem Zeitpunkt des Treffererereignisses des zweiten Operators lag (vgl. [41]).

Nachher

Der Nachher-Operator (`after`) verhält sich wie der Vorher-Operator, nur dass er zutrifft, wenn der Zeitpunkt eines eintreffendes Ereignis nach dem angegebenen absoluten Zeitpunkt ist. Für den Fall, dass zwei Operatoren als Argumente dienen, trifft er zu, wenn der Zeitpunkt des Treffererereignisses des ersten Operators nach dem Zeitpunkt des Treffererereignisses des zweiten Operators ist (vgl. [41]).

Gleichzeitig

Der Gleichzeitig-Operator (`concurrent`) kann ebenfalls mit einem absoluten Zeitpunkt oder mit den Zeitpunkten der Treffererereignisse von zwei Operatoren arbeiten. Er trifft zu, wenn die Zeitpunkte exakt gleich sind (vgl. [34, 41]).

Sequenz

Der Sequenz-Operator (`sequence`) hat beliebig lange Liste an Operatoren als Parameter. Er trifft zu, wenn alle Operatoren auf hintereinander eintreffenden Ereignissen zutreffen, ohne dass ein nicht zutreffendes Ereignis dazwischen auftritt. Formal: Sei n die Anzahl an Operatoren und $e_1, ..., e_n$ im Ereignisstrom hintereinander eintreffende Ereignisse, dann trifft der Operator zu, wenn für die als Parameter angegebenen Operatoren $op_1, ..., op_n$ gilt, dass alle e_i mit $i \in \{1, ..., n\}$ Treffererereignisse von op_i mit $i \in \{1, ..., n\}$ sind (vgl. [41]).

4.2.6. Numerische Operatoren

Numerische Operatoren sind Operatoren, die mit numerischen Werten arbeiten, d.h. sie benutzen für ihre Auswertungen Attributwerte. Da der Attributtyp numerisch sein muss, hängen die Operatoren vom Ereignistyp ab. Sie sind also anders als die vorherigen Operatoren nicht allgemein verwendbar. Da numerische Auswertungen jedoch sehr häufig vorkommen, wurden sie in das Grundsystem integriert. Für alle numerischen Operatoren gilt, dass für den Fall, dass ein zu überprüfendes Attribut nicht existiert oder dessen Typ nicht numerisch ist, das Ereignis ignoriert wird. Dies setzt den in 4.2.1 beschriebenen Gedanken um, dass der Rückgabewert nicht wahr oder falsch ist, wenn der Operator auf einen Ereignistyp nicht anwendbar ist.

Größer und größer gleich

Der Größer-Operator (`greater`) hat zwei Argumente – einen Attributnamen und einen Vergleichswert. Er trifft zu, sobald ein Ereignis eintrifft, bei dem das angegebene Attribut existiert und dessen Wert größer ist als der Vergleichswert. Der Vergleichswert kann auch ein Operator sein, dann wird vom letzten Treffererereignis des Operators der Wert des angegeben Attributs für den Vergleich hergenommen.
Analog dazu existiert ein Größer-Gleich-Operator (`greaterEqual`), der wie der Größer-Operator funktioniert, jedoch beim Vergleich zutrifft, wenn der zweite Parameter größer oder gleich dem zu prüfenden Wert ist.

Kleiner und kleiner gleich

Als Gegenstück zu den Größer- und Größer-Gleich-Operatoren existieren die Kleiner- (`less`) und Kleiner-Gleich-Operatoren (`lessEqual`). Sie treffen zu, sobald ein als erster Parameter angegebenes Attribut existiert und dessen Wert kleiner bzw. kleiner oder gleich dem Vergleichswert ist. Auch hier kann der Vergleichswert wieder ein Operator sein, von dem dann das letzte Treffererereignis verwendet wird.

Plus, Minus, Mal, Geteilt

Die Plus-, Minus-, Mal- und Geteilt-Operatoren (`plus`, `minus`, `times`, `divide`) nehmen eine besondere Rolle unter den numerischen Operatoren ein. Anders als die restlichen Operatoren, haben sie keine Bedingungen, die erfüllt werden müssen, damit sie zutreffen. Sie dienen vielmehr zum Rechnen innerhalb von Regeln und geben immer `true` zurück, sobald eine Rechnung durchgeführt werden konnte. Sie implementieren die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division und sind auf ein Attribut anwendbar, dass als erster Parameter angeben wird.

Die beiden weiteren Parameter a und b dienen als Operanden und es wird die jeweilige Rechenoperation ausgeführt, solange diese die Objekttypen von a und b unterstützt. Dies ist etwa für alle primitiven der Programmiersprache Java und deren zugehörigen Wrapper-Klassen, wie beispielsweise `int` oder `double` bzw. `Integer` oder `Double`, der Fall. Eine beliebige Erweiterung um andere Objekttypen wäre möglich. Weiterhin ist es möglich, statt Zahlen auch Operatoren als Parameter zu verwenden. In diesem Fall wird das entsprechende Attribut aus dem Treffereignis des übergebenen Operators als Operand verwendet.

Konnte die Rechenoperation durchgeführt werden, so wird das Ergebnis als Treffereignis für den Operator gesetzt. Dadurch kann man das Ergebnis der Rechnung als Parameter für andere Operatoren verwenden, die das Treffereignis eines Operators verwenden.

4.2.7. Ausdruckskraft

Es stellt sich die grundlegende Frage, was mit der Algebra von *Crepe* alles ausgedrückt werden kann. *Crepe* wurde so konzipiert, dass beliebig viele Operatoren hinzugefügt werden können. Weiterhin wurde bei der Entwicklung der Schnittstelle für Operatoren darauf geachtet, dass die Auswertungsfunktionen der Operatoren auf alle Sprachmittel von Java Zugriff haben. Somit ist es möglich mit Operatoren all das zu tun, was auch mit Java möglich ist. Um eine Aussage über die Ausdruckskraft der Algebra von *Crepe* zu treffen, ist es daher nötig, die Operatoren zu betrachten.

Die durch die Operatoren erzeugte Sprache kann Aussagen über die Existenz von (zusammengesetzten) Ereignissen im Ereignisraum treffen. Es ist möglich logische Aussagen über Attributwerte oder Zeitpunkte zu zutreffen. Die Operatoren Konjunktionen (`and`), Disjunktionen (`or`) und Negationen (`not`) lassen sich für jeden Operator der Algebra verwenden. Die Operatoren machen dabei nicht zwingend auf ein und demselben Ereignis eine Aussage über ihre Operanden. Dies liegt daran, dass bei der Ereignisselektion jeweils der letzte Treffer genutzt wird. Es kann jedoch eine Aussage über ein einzelnes Ereignis durch die Verwendung von Operatorreferenzierung in Verbindung mit dem `forEvent`-Operator getroffen werden.

Dadurch, dass alle Operatoren von *Crepe* als Operatorobjekt in einem Operatorbaum angesehen werden, ergibt sich die Möglichkeit einer Klammerung. Operatoren lassen sich verschachteln und werden entsprechend einer Klammerung ausgeführt. Überdies ist die Trefferreihenfolge der Operatoren für eine Regelauswertung nicht relevant, wodurch die Verknüpfungen sowohl *assoziativ* als auch *kommutativ* sind.

4.3. Fensterfunktionen

Der Ereignisstrom auf den die Profile angewendet werden, ist zunächst einmal nicht begrenzt. Es können beliebig viele Ereignisse eintreffen die alle ausgewertet werden müssen. Wie bereits erwähnt, haben viele Operatoren Zustände, die sie ab einem Treffer solange beibehalten, bis die gesamte Regel zutrifft. Dieser Umstand kann manchmal zu einem unerwünschten Verhalten führen. Möchte man beispielsweise bei einem Temperaturregelungssystem wissen, ob von Sensoren ausgewertete Temperaturen und Luftfeuchtigkeiten einen gewissen Grenzwert überschreiten, so möchte man dies in der Regel innerhalb von gewissen Zeiträumen tun. Standardmäßig verhalten sich die Operatoren jedoch nicht so, sondern wenn beispielsweise der Operator, der die Überschreitung des Temperaturgrenzwertes abfrägt, durch ein entsprechendes Ereignis in den Trefferzustand, übergeht, so könnten auch Jahre vergehen, bis ein weiteres Ereignis eintrifft, dass den Grenzwert für die Luftfeuchtigkeit überschreiten lässt – das Profil, würde trotzdem zutreffen und eine Benachrichtigung würde ausgelöst werden.
Es ist also nötig, eine Möglichkeit zu haben, das Verhalten von Operatoren so zu verändern, dass sich deren Auswertung auf bestimmte Ereignisse begrenzt. Hierfür werden üblicherweise Fensterfunktionen verwendet. Bei ereignisgesteuerten Systemen bieten sich vor allem Schiebefensterfunktionen (engl. *sliding windows*) an. Betrachtet man den Ereignisstrom als Zeitstrahl, so begrenzen Schiebefenster den Gültigkeitsbereich der Operatoren immer ausgehend vom aktuellen Ereignis in Richtung Vergangenheit. Das Fenster verschiebt sich also mit jedem neu eintreffenden Ereignis um eins nach vorne. Zwei solcher Schiebefensterfunktionen wurden auch in das im Rahmen dieser Arbeit implementierte System integriert und sollen daher im Folgenden kurz besprochen werden.

4.3.1. Letzten n Ereignisse

Eine Variante, die Gültigkeit von Operatoren einzuschränken ist es, nur die letzten n Ereignisse für die Auswertung her zu nehmen. Trifft ein Operator nicht innerhalb der letzten n-Ereignisse zu, so wird er in seinen Ausgangszustand versetzt. Der Vorteil ist, dass man im Bezug auf Systemressourcen immer genau weiß, wie viele Ereignisse maximal vorgehalten werden müssen.
Bei manchen Ereignisquellen hat man jedoch keine genaue Kontrolle darüber wie viele Ereignisse innerhalb einer gewissen Zeit eintreffen, d.h. dass Operatoren zeitlich gesehen

eine sehr lange Gültigkeit haben können, wenn nur wenige Ereignisse eintreffen oder aber eine sehr kurze Gültigkeit, wenn etwa viele Ereignisse schnell hintereinander eintreffen.

4.3.2. Innerhalb eines Zeitraums

Um eine zeitliche Kontrolle der Gültigkeit der Operatoren zu ermöglichen, gibt es noch Zeitfenster, die festlegen, innerhalb welcher Zeit, alle Treffererereignisse aufgetreten sein müssen (vgl. [23]). Hierbei sind keine absoluten Zeitpunkte gemeint, da hierfür die zeitlichen Operatoren, wie *Vorher* oder *Nachher* dienen, sondern relative Zeiträume. Der Zeitraum gilt immer vom Zeitstempel des zuletzt eingetroffenen Ereignisses in die Vergangenheit. Beispielsweise könnte man festlegen, dass alle Treffererereignisse innerhalb einer Stunde vor dem zuletzt eingetroffenen Ereignis stattgefunden haben müssen. Nachteil dieser Methode ist, dass man keinerlei Kontrolle über die Anzahl an Ereignissen hat, die innerhalb dieser Zeit stattgefunden haben.

5. Implementierung von Crepe

Im Rahmen dieser Arbeit wurde ein ereignisgesteuertes System namens *Crepe* (*Complex rule-based event processing engine*) entwickelt, das viele der zuvor genannten Prinzipien umsetzt. Im Mittelpunkt des Entwurfs stand dabei, das System möglichst so zu gestalten, dass es leicht erweiterbar ist. Dadurch soll gewährleistet sein, dass für zukünftige Anwendungsfälle noch eventuell fehlende Funktionalität leicht hinzugefügt werden kann. Hierfür wurden Schnittstellen an den entsprechenden Stellen ausgearbeitet.
Im Folgenden soll zunächst die grundlegende Architektur der Software beschrieben und die verwendeten Technologien erwähnt werden. Anschließend wird auf die Funktionalität der einzelnen Bestandteile eingegangen und diese näher erläutert. Abschließend werden die jeweiligen Schnittstellen zum Hinzufügen von Erweiterungen, wie etwa Operatoren, Fensterfunktionen oder auch Schnittstellen zu externen Systemen, besprochen.

5.1. Architektur

Das ereignisgesteuerte System wurde so konzipiert, dass das System in ein Frontend und ein Backend aufgeteilt ist. Das Backend übernimmt dabei die komplette Ereignisverarbeitung, d.h. das Laden und Umwandeln von Ereignis, sowie die Regelverarbeitung und das anschließende, erneute Umwandeln in ein Ausgabeformat inklusive herausladen an eine externe Schnittstelle.
Das Frontend dient als Benutzeroberfläche und hilft dem Anwender bei der Erstellung von Regeln. Da eine saubere Implementierung von Backend und Frontend den Rahmen dieser Arbeit bei weitem gesprengt hätte, ist hier zunächst nur das Backend implementiert worden. Um dennoch eine erste Benutzung des Systems, vor allem zu Testzwecken, zu ermöglichen, wurde eine dateibasierte Benutzerschnittstelle integriert. Die einzelnen Bestandteile des Backends werden im Folgenden genauer beschrieben und die zugrundeliegenden Entwurfsentscheidungen erläutert.

5.1.1. Verwendete Technologien

Das ereignisgesteuerte System wurde komplett mit der Programmiersprache *Java* [31] entwickelt, wobei eine vollständige Kompatibilität zur Version 6 eingehalten wurde. Als Build-Management-Tool wurde *Maven* [15] von der *Apache Software Foundation* gewählt. Maven erstellt Java-Programme nach einem standardisierten Verfahren und sorgt

unter anderem automatisch dafür, dass alle zum kompilieren benötigten Softwarebibliotheken gegebenenfalls aus dem Internet nachgeladen werden.
Zur einfacheren Handhabung der Ereignisströme wurde die *Apache-Camel*-Bibliothek benutzt [13]. Apache Camel ist nachrichtenorientierte Middleware, die es erlaubt, Routen mit Hilfe einer auf Java basierenden Domain Specific Language (Java-DSL) zu definieren. So lassen sich recht einfach Ereignisse innerhalb des Systems in Form von Apache-Camel-Nachrichten versenden. Beispielsweise werden so Ereignisse von Quellen zur Algebra gesendet, die nach einer Auswertung entweder *Composite Events* zurück in die Algebra sendet oder *Benachrichtigungen* auslöst, die Ereignisse zu den Senken weiterleiten (siehe 5.1.2).
Weiterhin wurde die *JUnit*-Bibliothek [24] verwendet, um automatisierte Tests in den Entwicklungsprozess zu integrieren. Dadurch soll eine bessere Code-Qualität erreicht und Regressionen vermieden werden. Als Logger kam Log4j [14] – ebenfalls von der Apache Software Foundation – zum Einsatz.

5.1.2. Allgemeiner Aufbau

Das Backend ist ein Konsolenprogramm, das nach dem Starten zunächst eine Konfigurationsdatei einliest und eventuell per Kommandozeile übergebene Parameter auswertet. Anschließend werden aus einem Profilordner alle Profildateien (siehe 5.2.3) ausgelesen und die darin spezifizierten Profile zum Algebrasystem hinzugefügt.
Danach läd das Backend alle Source-Dateien (vergleiche 5.2.1) aus dem Source-Ordner. Jede Source-Datei repräsentiert eine Quelle, aus der Ereignisse geladen werden sollen. Dabei sind in jeder Datei ein *Loader* und ein *InputAdapter* und zusätzlich deren Parameter angegeben. Die Source-Dateien werden also ausgewertet und danach die Ereignisse mit Hilfe des entsprechenden Loader-Plugins geladen, durch den InputAdapter in das interne Ereignisformat umgewandelt und anschließend an das Algebrasystem weitergeleitet. Sowohl der Profilordner, als auch der Source-Ordner werden dabei solange auf Veränderungen beobachtet, wie das Programm läuft, das heißt, dass neue Profile oder Source-Dateien im laufenden Betrieb nachgeladen werden können.
Im Algebrasystem wird dann jedes Profil auf jedes Ereignis angewendet. Ergibt die Regel eines Profils einen Treffer, der durch das zuletzt verarbeitete Ereignis ausgelöst wurde, so werden alle Benachrichtigungen des Profils ausgelöst. Beinhaltet eine Benachrichtigung ein *Composite Event*, so wird dieses zurück in das Algebrasystem gesendet.
Das Laden von Profildateien, Source-Dateien und die Ereignisverarbeitung durch das Algebrasystem stellen jeweils eine Route in der Apache-Camel-Middleware dar. Die Grafik 5.1 verdeutlicht diesen Aufbau nochmals. Die Knoten in der Grafik repräsentieren dabei *Components* in Camel und die Kanten werden durch Nachrichten in Camel repräsentiert. Durch die Java-DSL von Camel ist es sehr leicht, die Knoten mit neuen Punkten zu verbinden. So könnte etwa für eine spätere Anbindung an ein Frontend sehr leicht eine weitere Komponente hinzugefügt werden.

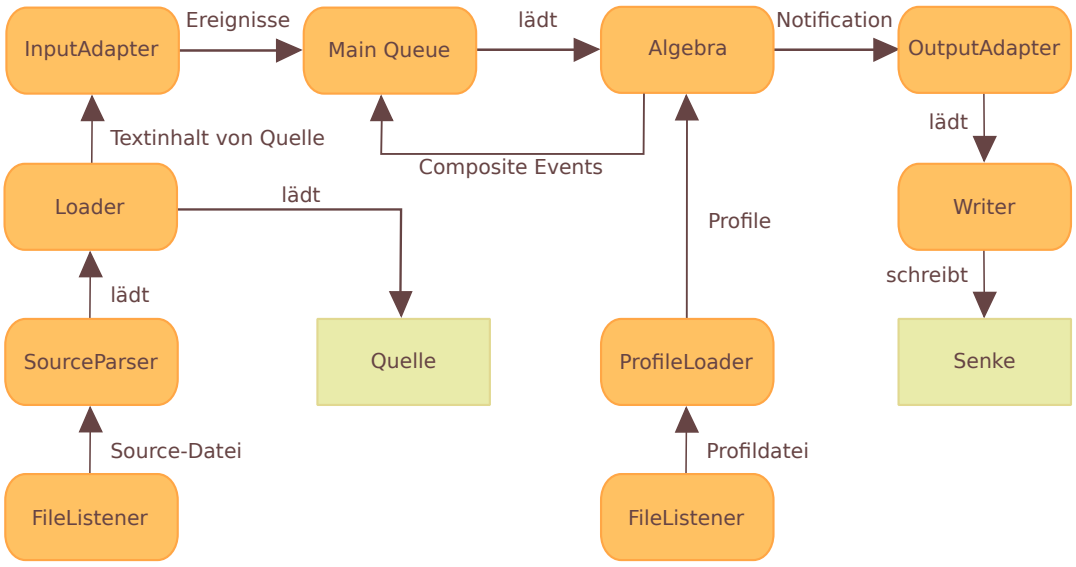

Abbildung 5.1.: Architektur von *Crepe*

Will man Beispielsweise über eine GUI, ein Profil laden, so wäre es einfach möglich eine entsprechende Camel-Nachricht an den `ProfileLoader` zu senden. Genauso einfach wäre es für eine GUI Quellen zu laden: Eine einfache Nachricht mit dem Inhalt einer *Source*-Datei an den `SourceLoader` senden und schon wird die Quelle geladen und die Auswertung der Ereignisse gestartet.

5.1.3. Ereignismodell

Um Ereignisse und Attribute innerhalb des Systems abzubilden, wurden die zwei Schnittstellen *IEvent* und *IAttribute* geschaffen. Weiterhin gibt es eine Klasse *Event* und eine Klasse *Attribute*, die die jeweiligen Schnittstellen implementieren.

Ereignisse wurden dabei so umgesetzt, dass sie aus einem Zeitstempel und einer *Map* von Attributen bestehen (vgl. Abbildung 1.1) und für diese entsprechende Getter und Setter enthalten. In der Map sind die Schlüssel vom Typ *String* und die Werte vom Typ *IAttribute*. Dies repräsentiert den Ansatz, beliebige Attribute anhand eines Namens aufrufen zu können. Für den Zeitstempel wurde die *Date*-Klasse aus der Java-API gewählt, welche eine Genauigkeit von Millisekunden aufweist.

Die Klasse Attribute hält lediglich den Attributwert intern als Datentyp *Object* vor und bietet Getter und Setter dafür. Es ist ein reines Wrapper-Objekt und dient lediglich der Datenkapselung.

5.2. Komponenten

An dieser Stelle soll auf die wichtigsten Komponenten von *Crepe* im Detail eingegangen werden. Die Komponenten sind, wie zuvor bei Kapitel 5.1.2 beschrieben, durch die *Apache-Camel*-Middleware miteinander verbunden.

5.2.1. Loader und Input-Adapter

Das Hineinladen von Ereignissen in das Algebrasystem wurde in zwei Schritte unterteilt. Als erstes werden Ereignisse von einer Quelle geladen und intern als String gespeichert. Dieser String wird dann an einen so genannten *Input Adapter* weitergeleitet, der aus dem String Ereignisse extrahiert und in das intern verwendete Ereignisformat umwandelt.

Ein *Loader* ist also nur für das reine Laden von Ereignissen zuständig, während ein *Input-Adapter* ausschließlich für die Datentransformation verantwortlich ist. Durch die Trennung in zwei Komponenten, können zahlreiche Kombinationen von Ereignisquellen und Formaten im System verwendet werden. Beispielsweise kann ein Loader der Dateien per HTTP lädt, eine CSV-Datei laden. Der zuständige CSV-Inputadapter wandelt dann diese Datei ins interne Ereignisformat und sendet die einzelnen Ereignisse nacheinander in das Algebrasystem.

Um noch mehr Flexibilität zu ermöglichen, können die Loader und Input Adapter parametrisiert werden. Beispielsweise beim Laden einer Datei per HTTP etwa die URL als Parameter für den Loader und beim Umwandeln einer CSV-Datei das Trennzeichen, wie etwa ein Komma- oder Semikolonzeichen. Da langfristig die Benutzer des Systems selbst Loader und Input Adapter hinzufügen können sollen, wurde eine recht einfache Schnittstelle entworfen, um diese zu Schreiben und mit Parametern zu versehen.

Sowohl die Loader als auch die Input Adapter werden beim Start des Programms dynamisch aus Pluginordnern geladen (je ein Ordner für Loader und ein Ordner für Input Adapter). Diese Ordner lassen sich optional in der Konfigurationsdatei festlegen. Konkret wurde das dynamische Laden der Klassen in Java per *Reflection* umgesetzt. Der sogenannte *SourceLoader* lädt alle *.class*-Dateien aus den Ordner und überprüft, ob sie entsprechende Interfaces, die sie als Loader oder Input-Adapter ausweisen, implementieren. In den Konstruktoren der Klassen können den Parametern *Tags* gegeben werden, die dem Parameter einen Namen in den Source-Dateien geben.

Die Source-Dateien selbst sind *Java-Properties*-Dateien (*.properties*). Properties-Dateien bestehen aus Schlüssel- und Wertepaaren und lassen sich innerhalb von Java sehr leicht verarbeiten. Von sich aus sind Properties-Dateien nicht hierarchisch, weswegen künstlich eine kleine Hierarchie eingeführt wurde, um die Einstellungen der Loader und der Input-Adapter voneinander zu trennen. Die Trennung erfolgt dabei durch einen Punkt, wobei die Parameter der Loader mit *loader* beginnen und die der Input Adapter mit *inputAdapter*. Für den *URLLoader* (der Dateien von HTTP, FTP oder der Festplatte

lädt) kombiniert mit dem *CSVInputAdapter* sieht eine *Source*-Datei beispielsweise wie folgt aus:

```
loader = URLLoader
loader.url = http://www.datenquelle.de/wetterdaten.csv
inputAdapter = CSVInputAdapter
inputAdapter.timeStampCaption = Datum
inputAdapter.timeStampFormat = yyyy-MM-dd
inputAdapter.delimiter = ;
inputAdapter.columnTypes = java.lang.Integer, java.lang.String, \
                           java.lang.Integer, java.lang.Double, \
                           java.lang.Double, java.lang.Double, \
                           java.lang.Double, java.lang.Double
```

Es müssen in einer Source-Datei immer Loader und Input Adapter zusammen angegeben werden, so dass dem System eine eindeutige Zuordnung zwischen der Quelle und der Ereignisformattransformation möglich ist. Die Zeile `loader = URLLoader` spezifiziert den Loader, der verwendet werden soll und muss deshlab genauso heißen, wie der Klassenname des Loaders (hier `URLLoader`). Alle weiteren Parameter des URLLoaders (hier nur ein einziger) werden dann mit einem Punkt abgetrennt. Analog dazu verhält es sich mit dem InputAdapter.
Prinzipiell ist es möglich, sowohl für den Loader, als auch für den Input Adapter, die Anzahl und Art der Parameter variieren zu lassen. Wenn etwa beim Weglassen von bestimmten Parametern an deren Stelle Standardwerte verwendet werden oder gar sich der Loader oder InputAdapter durch andere Parameter unterschiedlich Verhalten sollen. Ein Beispiel wäre hier beim CSVInputAdapter, dass man den Parameter `columnTypes` weglassen könnte, um etwa den Input Adapter selbst den Datentyp anhand einer Heuristik bestimmen zu lassen.
Die Plugin-API für Loader und Input-Adapter gestaltet sich wie folgt: Die Parameter der Loader und Input-Adapter werden in den Konstruktoren der Klassen festgelegt und mittels der *Annotation* `Tag` der Name für den Schlüssel in der *Properties*-Datei, ohne die Präfixe `loader` oder `inputAdapter` spezifiziert. Für den oben genannten CSVInputAdapter sähe die Deklaration des Konstruktors beispielsweise wie folgt aus:

```
public CSVInputAdapter(@Tag("timeStampCaption") String caption,
                       @Tag("timeStampFormat") String format,
                       @Tag("delimiter") String delimiter,
                       @Tag("columnTypes") String columnTypes)
```

Loader müssen die Schnittstelle `ILoader` und Input-Adapter die Schnittstelle `IInputAdapter` implementieren. `ILoader` enthält eine Methode namens `load()`, die das Laden der Quelle durchführt und eine Methode namens `getText()`, die den Inhalt der Quelle als `String` zurück gibt.

Die Schnittstelle `IInputAdapter` enthält die Methoden `load(String text)` und `getEvents()`. Erstere bekommt als Parameter den vom Loader erzeugten `String` und sorgt für die Transformation des Textes in eine Liste von Ereignissen. Die Liste wird dann von `getEvents()` zurückgegeben und automatisch vom Algebrasystem ausgelesen.
Möchte man verschiedenen Konfigurationsmöglichkeiten für ein und denselben Loader oder Input-Adapter haben, so gestaltet sich dies dank Verwendung von *Annotations* recht einfach: Die Konstruktoren der Loader oder Input Adapter werden einfach überladen und deren Parameter entweder mit unterschiedlichen Tags versehen oder aber die Anzahl der Tags variiert. Der `SourceLoader` sucht dann automatisch den Konstruktor heraus, der zu den angegebenen Parametern passt. Die Reihenfolge der Parameter wird jedoch, anders als in Java selbst, nicht berücksichtigt.

5.2.2. Operatoren und Regeln

Den Kern des Algebrasystems bilden die Operatoren. Da das gesamte System den Anspruch hat, universal einsetzbar zu sein, war es ebenfalls notwendig, es zu ermöglichen Operatoren zur Algebra hinzuzufügen. Dabei war es Ziel, dies so komfortabel wie möglich zu gestalten, ohne jedoch den Programmierer, der eine Erweiterung implementiert, zu sehr einzuschränken. Dies wurde erreicht, indem für viele Funktionen, die ein Operator haben muss, bereits Standardimplementierungen integriert wurden. Will man einen neuen Operator hinzufügen, muss man ein Klasse von der abstrakten *Operator*-Klasse ableiten und nur die Teile überschreiben, die für die Funktionalität des neuen Operators relevant sind.
Jeder Operator besteht im wesentlichen aus zwei Teilen: Zum einen aus der Operatorklasse selbst, welche die eigentliche Implementierung des Operators darstellt und zum anderen aus einer statischen Wrapper-Methode, welche den Operator in die DSL integriert (vgl. 5.2.4).
Die Operatorklasse selbst fungiert immer als Knoten in einem Operatorbaum, der eine Wurzel besitzt und bei Profilen auch *Regel* genannt wird. Ein Operator kann also Unteroperatoren haben. Ist dies nicht der Fall, dann ist der Operator ein Blatt im Operatorbaum. Wie bereits erwähnt haben Profile immer genau eine Regel und mindestens eine Benachrichtigung. Beim Auswerten der Profile wird also das gerade sich im Ereignisstrom befindliche Ereignis der Regel des Profils übergeben und der Operatorbaum von der Wurzel hin zu den Blättern abgearbeitet (*Top-Down*-Verfahren). Die Blätter stellen dabei die Terminalsymbole des Syntaxbaumes dar und sind üblicherweise die einzigen Operatoren, die direkt auf den Ereignissen arbeiten, während die übergeordneten Operatoren typischerweise ihre untergeordneten Knoten nur Auswerten und das gerade zu verarbeitende Ereignis an diese weiterreichen.
Das ganze lässt sich an einem einfachen Beispiel verfolgen: Betrachte man den Operator `attribute`, welcher einen Treffer ergibt, sobald ein angegebenes Attribut existiert und weiterhin den logischen Und-Operator (`and`). Verknüpft man nun zwei `attribute`-

Operatoren, einer der auf das Attribut mit dem Schlüssel *Temperatur* und einer der auf das Attribut mit dem Schlüssel *Luftfeuchtigkeit* überprüft, mit einem logischen Und. In der DSL sähe das dann wie folgt aus:

```
and(attribute("Temperatur"), attribute("Luftfeuchtigkeit"))
```

Der `and`-Operator würde zuerst aufgerufen werden. Dieser würde wiederum die beiden `attribute`-Operatoren aufrufen und ihm das aktuelle Ereignis übergeben. Die beiden `attribute`-Operatoren überprüfen, ob der jeweilige Schlüssel existiert und geben entsprechend `true` oder `false` zurück. Der `and`-Operator überprüft die Rückgabewerte und gibt das Ergebnis der logischen Und-Verknüpfung zurück.
Die konkrete Implementierung eines neuen Operators in *Crepe* gestaltet sich dann wie folgt: Die Klasse des neuen Operators erbt von der abstrakten Klasse `Operator` und implementiert die Methode `apply(IEvent event)`. In dieser Methode findet die Auswertung statt und über den Parameter `event` hat man Zugriff auf das gerade im Ereignisstrom eintreffende Ereignis. Die Methode gibt `true` zurück, wenn der Operator zutrifft, `false` wenn er nicht zutrifft und kann eine `OperatorNotFoundException` werfen, wenn das Ereignis ignoriert werden soll (siehe 4.2.1). Über die Getter und Setter `getMatchingEvent` und `setMatchingEvent` lässt sich auf das Treffererereignis zugreifen. Um beispielsweise den `attribute`-Operator zu implementieren wäre folgender Code ausreichend:

```
public boolean apply(IEvent event)
                        throws OperatorNotSupportedException {

        if (event.getAttributes().containsKey(this.attribute)){
                this.setMatchingEvent(event);
        }
        else {
                throw new OperatorNotSupportedException();
        }

        return this.getMatchingEvent() != null;
}
```

Das Setzen der Parameter eines Operators, wie im obigen Beispiel der Attributname `attribute`, geschieht über den Konstruktor. Dieser wird dann in der DSL über eine statische Wrapper-Funktion aufgerufen (siehe 5.2.4).
Viele der Operatoren von *Crepe*, die direkt auf Attributen arbeiten, behalten ihren Zustand, ab dem ersten Treffererereignis bei. Sie werden erst wieder in den Grundzustand versetzt, wenn das gesamte Profil zugetroffen hat. Um dies zu berücksichtigen existiert in der `Operator`-Klasse eine Methode namens `reset()`. Sie wird automatisch rekursiv

für den gesamten Operatorbaum aufgerufen, sobald das Profil des Operators zutrifft. Für das obige Beispiel sähe ein Implementierung etwa wie folgt aus:

```
@Override
public void reset() {

        this.setMatchingEvent(null);
}
```

Stellt ein Operator kein Blatt im Operatorbaum dar, sondern hat selbst Unteroperatoren, so werden einfach die `apply`-Methoden der Unteroperatoren aufgerufen. Zusätzlich müssen einmalig mit dem Setter `setChildren(Operator ... children)` die Unterknoten des Operatorbaums gesetzt werden, damit dieser für die `reset`- und Fensterfunktionen traversiert werden kann. Ein einfaches Beispiel für die Implementierung eines Operators mit Unteroperatoren ist der `and`-Operator, dessen `apply`-Methode wie folgt implementieren lässt:

```
public boolean apply(IEvent event)
                throws OperatorNotSupportedException {

        return getChildren()[0].apply(event)
                & getChildren()[1].apply(event);
}
```

5.2.3. Profile und Benachrichtigungen

Profile bestehen aus einer Regel und einer oder mehreren Benachrichtigungen. Wie bereits erwähnt besteht eine Regel aus einem Operatorbaum, d.h. konkret, dass ein Profil eine Referenz auf die Wurzel eines Operatorbaumes enthält. Das Algebrasystem wertet Profile in der Reihenfolge aus, in der sie dem System hinzugefügt wurden. Die Auswertung besteht dann naheliegenderweise aus dem Aufruf des Wurzeloperators der Regel, wobei diesem jeweils das aktuell zu verarbeitende Ereignis übergeben wird.
Gibt die `apply`-Methode des Wurzeloperators `true` zurück, so trifft die gesamte Regel zu und alle Benachrichtigungen des Profils werden ausgelöst. Nach dem die Benachrichtigungen abgearbeitet wurden, wird die Regel in den Anfangszustand zurückgesetzt, d.h. dass alle Operatoren ihre Treffer auf vergangene Ereignisse wieder verwerfen. Würde dies nicht geschehen, so würde die Regel ihren Zustand für immer beibehalten und unaufhörlich zutreffen.
Profile werden vom Nutzer über eine domänenspezifische Sprache definiert, die im nächsten Abschnitt beschrieben wird. Dort wird auch ein Beispiel für die Definition eines Profils gegeben.

5.2.4. Domänenspezifische Sprache

Ein wichtiger beim Punkt beim Entwurf der Software war es, dem Benutzer eine einfache Möglichkeit zu geben, Profile zu definieren. Zielgruppe sollten dabei nicht nur technisch versierte Benutzer, wie etwa Programmierer sein. Aus diesem Grund sollte der intern verwendete Java-Code bei der Definition von Regeln weitestgehend verborgen bleiben. Dennoch sollte das System aufgrund seines Anspruchs erweiterbar und universal zu sein, flexibel genug sein, um komplexere Sachverhalte zu bewältigen.
Ein verbreiteter Ansatz, der auch hier verwendet wurde, ist es, eine domänenspezifische Sprache (engl. *domain specific language*, kurz *DSL*) zu verwenden. DSLs sind formale Sprachen, die speziell für eine bestimmte Anwendungsdomäne entworfen und implementiert werden [16]. Die Sprache wird so entworfen, dass sie zwar das ganze Anwendungsfeld abdeckt, aber möglichst nichts was außerhalb der Domäne liegt. Durch die Beschränkung der Sprache auf das Anwendungsfeld ist es einem Domänenspezialisten ohne Zusatzwissen möglich, die DSL zu verwenden.
Man unterscheidet im Allgemeinen zwischen externen und internen (auch eingebetteten) DSLs [16]. Externe DSLs sind von Grund auf neu entwickelte Sprachen, während interne DSLs eine Untermenge einer anderen, meist komplexeren, Sprache darstellen.
Externe DSLs haben den Vorteil, das man sowohl deren Syntax, als auch deren Semantik frei definieren kann. Man hat die volle Kontrolle über die Sprache, muss sich aber im Gegenzug auch um alle Bestandteile, wie etwa Parser, Interpreter oder Compiler, selbst kümmern. Dies stellt einen nicht unerheblich Aufwand und ein großes Potential für Fehlerquellen dar. Durch die Verwendung von Werkzeugen zur Erstellung von externen DSLs, wie beispielsweise Xtext [40], lassen sich diese jedoch erheblich reduzieren.
Interne DSLs verwenden zur Sprachimplementierung Komponenten ihrer Wirtssprache und sind daher immer eine Untermenge der Wirtssprache. Aufgrund dessen ist der Implementierungsaufwand üblicherweise erheblich kleiner als bei einer externen DSL. Ein Nachteil ist, dass man auf Sprachkonstrukte, die auch in der Wirtssprache nicht möglich sind, in der DSL verzichten muss. Beispielsweise können in der Programmiersprache Java keine Operatoren überladen werden, weshalb es auch nicht möglich ist, dies in einer DSL, die in Java eingebettet ist, zu verwenden.
Für das vorliegende System wurde eine interne Java-DSL zur Definition von Profilen und deren Regeln gewählt, da bei diesem Ansatz ein gutes Gleichgewicht zwischen dem Verbergen von unnötigem Java-Code und der leichten Erweiterbarkeit des Systems herrscht. Eine interne DSL ermöglicht es sehr leicht, neue Operatoren oder andere Sprachelemente hinzuzufügen. Für *Crepe* wurde die DSL durch Verschachtelung statischer Funktionen realisiert (siehe [16]).
Um beispielsweise einen neuen Operator der DSL hinzuzufügen, ist es lediglich nötig eine Wrapper-Funktion mit einer einzigen `return`-Anweisung, die das Operator-Objekt zurück gibt, zu erstellen. Für das Beispiel des `attribute`-Operators (siehe 5.2.2) sähe die Wrapper-Funktion wie folgt aus:

```
public static Operator attribute(String attribute) {

        return new Attribute(attribute);
}
```

Wegen der Einbettung in Java wird für die Kompilierung der DSL der Java-Compiler verwendet. Der Benutzer erstellt also Profildateien, die den DSL-Code enthalten, welche dann aus dem Profilordner per Apache-Camel geladen und an den `ProfileLoader` übergeben werden. Dieser fügt für die Kompilierung benötigten Text, wie z.B. Klassendeklaration und Import-Deklarationen, oberhalb und unterhalb zum Inhalt der Profildatei hinzu. Die Profildefinitionen werden dabei in eine statische Funktion gepackt, die eine Liste mit Profilen zurückgibt.
Anschließend kompiliert der `ProfileLoader` den so erzeugten Java-Code und ruft die vorher erzeugte statische Funktion auf. Die zurückgegebene Liste mit Profilen wird dem Algebrasystem hinzugefügt und von da an auf den Ereignisstrom angewendet. Die Profildefinition sieht aus Sicht des Anwenders dann beispielsweise so aus:

```
profile(and(attribute("Temperatur"), attribute("Luftfeuchtigkeit")),
compositeEventNotification(event("Beschreibung", "Beides gefunden!")));
```

Der erste Parameter von `profile` ist der Wurzeloperator der Regel (hier `and`) und der zweite Parameter eine variable Argumentliste zum übergeben von Benachrichtigungen (hier `compositeEventNotification`). Es können also beliebig viele Benachrichtigungen pro Profil ausgelöst werden.
Es sei angemerkt, dass es sich bei dem obigen Beispiel um den vollständigen Inhalt einer Profildatei handelt. Der weitere zur Kompilierung notwendige Java-Code wird vollständig automatisch hinzugefügt. Dennoch ist es sowohl möglich mehrere Profile pro Profildatei zu definieren, als auch Java-Code außerhalb einer Profildefinition anzugeben. Dieser Code wird beim Laden eines Profils mit ausgeführt, was vor allem für die Referenzierung von Operatoren verwendet wird (siehe Kapitel 4.2.2).

5.2.5. Output-Adapter und Writer

Um Daten aus dem System heraus exportieren zu können, gibt es *Output-Adapter* und *Writer*. Ähnlich zu den Input-Adaptern und Loadern übernimmt hier der Output-Adapter die Umwandlung der Ereignisse in das Ausgabeformat, während der Writer dafür zuständig ist den umgewandelten Text an ein Ausgabeziel zu schreiben.
Beim Laden von Ereignissen wurden die Quellen in Form von Java-Properties-Dateien realisiert. Dadurch ist es möglich, die Quellen von den Profilen zu trennen. Man kann also Dateien aus beliebigen Quellen laden und die daraus erzeugten Ereignisse werden von jedem Profil ausgewertet. So ist es auch möglich im laufenden Betrieb der Software noch neue Dateien nachzuladen.

Bei der Verwendung der Output-Adapter und Writer wurde dies anders gelöst. Je nachdem welches Profil zutrifft, ist eine andere Ausgabe notwendig. Daher wurde die Ausgabe in Form einer Benachrichtigung implementiert, welche einen Output-Adapter und einen Writer als Argumente erwartet. So entsteht eine feste Zuordnung von Ausgaben und Profilen und es ist problemlos möglich, beliebig viele Ausgaben pro Profil durchführen zu lassen.
Durch den gewählten Ansatz ergibt sich auch die API für das Hinzufügen neuer Output-Adapter und Writer: Die Benachrichtigung für die Ausgabe wird mittels der statischen Funktion`writerNotification` in der DSL aufgerufen. Diese hat zwei Parameter, eine Klasse, die die Schnittstelle `IOutputAdapter` und eine, die `IWriter` implementiert.
`IOutputAdapter` und `IWriter` sind umgekehrt zu den Schnittstellen für das Einlesen aufgebaut (vgl. 5.2.1): Erstere hat eine Methode `load(IEvent[] events)`, der eine Liste mit Treffererereignissen aus referenzierten Operatoren übergeben wird. Diese Liste wird vom Benutzer als Parameter in der `writerNotification` spezifiziert und enthält jene Operatoren, deren Ereignisse exportiert werden sollen.
Wird die `writerNotification` ausgelöst, so wird die `load`-Methode aufgerufen und durch diese eine Transformation in das gewünschte Ausgabeformat vorgenommen. Nach der Transformation wird der erzeugte Text durch `getOutput()` ausgelesen und dem `Writer` durch `setText(String text)` als `String` übergeben. Abschließend wird dessen `write`-Methode aufgerufen und der Text so an ein Ziel geschrieben.
Für den Entwickler von Writer- und Output-Adapter-Plugins ist es also nötig die beiden oben genannten Schnittstellen zu implementieren. Zwecks Konsistenz sollten für jedes hinzugefügte Plugin noch eine Wrapper-Funktionen zur DSL hinzugefügt werden. Nötig ist dies jedoch nicht, da auch innerhalb eines Profils Java-Klassen instanziiert werden können. Ein Aufruf einer `writerNotification` sieht dann beispielsweise wie folgt aus:

```
writerNotification(fileWriter("/pfad/zu/datei"),
                   stringOutputAdapter("Ausgabe: %f", "Temperatur"), m)
```

Diese Benachrichtigung würde eine Textdatei an den angegebenen Pfad schreiben. Der `stringOutputAdapter` enthält einen Java-Format-String und eine Liste mit Attributnamen als Parameter. Er erstellt aus dem Format-String und den Werten der Attribute, deren Namen als Parameter übergeben wurden, einen Text. Die Attributwerte werden hierbei aus den Treffererereignissen, der als variable Argumentliste (hier nur `m`) übergebenen Operatoren, bezogen. Ausführlichere Beispiele finden sich in Kapitel 6.

5.2.6. Fensterfunktionen

In Kapitel 4.3 wurden Fensterfunktionen besprochen und beschrieben welche Rolle sie im Algebrasystem spielen. Fensterfunktionen wurden in *Crepe* so integriert, dass man

sehr leicht neue Fensterfunktionen hinzufügen kann. Die beiden in 4.3.1 und 4.3.2 besprochenen Fensterfunktionen wurden implementiert und können über die DSL für die Profile verwendet werden.
Fensterfunktionen werden an Operatoren angehängt und im Operatorbaum integriert. Jedes Blatt im Knoten hat ein eigenes Fensterobjekt, welches die Schnittstelle `IWindow` implementiert. Um das Erzeugen neuer Fensterobjekte zu automatisieren hat die Schnittstelle eine `newInstance`-Methode, die ein neues Fensterobjekt im Grundzustand, erzeugt. Diese Methode wird dann rekursiv für den Operatorbaum aufgerufen, so dass allen Operatoren ein Fensterobjekt zugewiesen wird. Ausgehend vom Grundzustand, wird für jedes hereinkommende Ereignis dessen `apply`-Methode aufgerufen, welche für eine Zustandsänderung sorgen kann und `false` zurück gibt, sobald der Operator zurückgesetzt werden soll. Gibt die Methode `true` zurück, so befinden sich alle Ereignisse noch innerhalb des Fensters und es ist kein Zurücksetzen des Operators nötig. Um beispielsweise die letzten n Ereignisse zu prüfen, zählt diese Methode alle Ereignisse, die vom Operator ausgewertet werden und gibt `false` zurück, sobald die Anzahl größer als n ist.
Da *Crepe*, wie in Kapitel 3.6.2 beschrieben, Treffereignisse immer mit dem zuletzt eingetroffenen, passenden Ereignis überschreibt, verändern sich die Fensterzustände mit jedem Treffer. Am Beispiel des Letzten-n-Ereignisse-Fensters etwa, dass der Zähler, der die ausgewerteten Ereignisse zählt, zurückgesetzt wird, sobald ein Treffer auftritt. Um auf Treffer zu reagieren, gibt es daher die Methode `onMatch`, welche jedes Mal aufgerufen wird, wenn ein Operator zutrifft.
Als Schnittstelle für den Benutzer wurden die Fensterfunktionen in die DSL integriert. Mit der statischen Funktion `setWindow` kann man eine Fensterfunktion setzen, die dann solange für alle anschließend definierten Profile gilt, bis erneut eine `setWindow`-Funktion aufgerufen wird. So kann für jedes Profil eine andere Fensterfunktion verwendet werden. Setzt man keine Fensterfunktion, so wird standardmäßig die Funktion `EndlessWindow` verwendet, was bedeutet, dass das Fenster unendlich groß ist, bzw. keine Fensterfunktion angewendet wird.

6. Evaluation

Es soll nun gezeigt werden, dass das implementierte System praxistauglich ist. Eines der Hauptziele bei der Entwicklung war, das System so zu gestalten, dass es universell einsetzbar ist. Durch Verwendung passender *Loader*, *Input-Adapter*, *Output-Adapter* und *Writer* kann die Anbindung an unterschiedliche externe Systeme hergestellt werden. Fehlen passende Adapter, Loader oder Writer, so können diese recht einfach durch die bereitgestellte API neu hinzugefügt werden. Ebenso verhält es sich mit den Operatoren. Da es unmöglich ist, erschöpfend zu zeigen, welche Anwendungsfälle mit dem System abgedeckt werden können, sollen hier statt dessen die Universalität anhand zweier Fallstudien belegt werden. Weiterhin ist es nötig, dass das System unter praktischen Bedingungen ausreichend schnell arbeitet. Dabei soll an dieser Stelle kein umfassendes Benchmarking durchgeführt, sondern lediglich der Nachweis einer annehmbaren Geschwindigkeit erbracht werden.

6.1. Fallstudie Wetteranalyse

Die erste Fallstudie soll anhand von Wetterdaten durchgeführt werden. Hierbei soll zunächst auf die Motivation eingegangen werden und danach erläutert werden, welche Vorbereitungen getroffen wurden. Anschließend wird besprochen, wie die Fallstudie mit *Crepe* durchgeführt wurde und zu welchem Ergebnis es dabei kam.

6.1.1. Motivation

Wetterstationen sammeln täglich eine ganze Reihe an Daten, die von Meteorologen ausgewertet werden. Die Daten kommen in der Regel von Sensoren, die periodisch ausgewertet werden. Das Auslesen der Daten als Ereignis zu betrachten und Auswertungen deshalb mit einem ereignisgesteuerten System zu machen.
Häufig sind die Daten, die von Wetterstationen übermittelt werden bereits aggregiert und werden nur in vergleichsweise großen zeitlichen Abständen, wie etwa einmal pro Tag, ausgewertet. Beispielsweise kann eine Station eine mittlere Windstärke und einen mittleren Luftdruck einmal täglich ausgeben. Da die Daten einen starken zeitlichen Bezug zueinander haben, liegt es also nahe, eine Datenübermittlung als ein einziges Ereignis mit mehreren Attributen zu betrachten. Im vorigen Beispiel würde also ein Ereignis pro Tag erzeugt werden und dieses hätte die zwei Attribute *mittlere Windstärke* und *mittlerer Luftdruck*.

Die Wetterdaten stellen also eine gute Möglichkeit dar, zu demonstrieren, wie durch *Crepe* Ereignisse mit vielen Attributen verarbeitet werden. Es werden also mehrere Operatoren auf unterschiedliche Attribute von ein und demselben Ereignissen angewendet, um ein zusammengesetztes Ereignis zu erzeugen.

6.1.2. Vorbereitung

Für die Wetterdatenfallstudie wurden Daten des *Deutschen Wetterdiensts* (*DWD*) verwendet. Der hier verwendete Datensatz wurde als CSV-Datei vom Lehrstuhl für Didaktik der Informatik an der Universität Bayreuth im Rahmen des *DBup2date!*-Projekts bereit gestellt [1].
Der Datensatz enthält Daten aus 80 aktiven Wettermessstationen, die vom DWD veröffentlicht werden. Die Messungen reichen ca. ein Jahr zurück und werden täglich aktualisiert. Jede Wetterstation ist in der Datei mit einer ID vermerkt und enthält pro Tag zwölf Messwerte. Daraus ergeben sich zum Zeitpunkt der Erstellung dieser Fallstudie genau 27753 Messungen mit je 12 Messwerten. Die Datei enthält pro Stations-ID und Tag genau eine Zeile mit den Messwerten in den Spalten 3 – 14.
Die Daten in der CSV-Datei bestehen aus den folgenden Spalten (hier Attributen):
Datum, *Qualitaet*, *Min_5cm*, *Min_2m*, *Mittel_2m*, *Max_2m*, *Relative_Feuchte*, *Mittel_Windstaerke*, *Max_Windgeschwindigkeit*, *Sonnenscheindauer*, *Mittel_Bedeckungsgrad*, *Niederschlagshoehe*, *Mittel_Luftdruck*.

6.1.3. Durchführung

Zunächst mussten die Daten in *Crepe* geladen und ein Ereignisstrom aus den Zeilen der CSV-Datei erzeugt werden. Die Daten lagen lokal vor und wurden daher mit dem `FileLoader` geladen. Als *Input-Adapter* wurde der `CSVInputAdapter` verwendet. Die*Source*-Datei für *Crepe* sah dabei wie folgt aus:

```
loader = FileLoader
loader.path = use_case_weather/data/wetterdaten_Wettermessung_short.csv
inputAdapter = CSVInputAdapter
inputAdapter.timeStampCaption = Datum
inputAdapter.timeStampFormat = yyyy-MM-dd
inputAdapter.delimiter = ;
inputAdapter.columnTypes = java.lang.Integer, java.lang.String, \
                           java.lang.Integer, java.lang.Double, \
                           java.lang.Double, java.lang.Double, \
                           java.lang.Double, java.lang.Double, \
                           java.lang.Double, java.lang.Double, \
                           java.lang.Double, java.lang.Double, \
                           java.lang.Double, java.lang.Double
```

Die erste Zeile gibt an, dass der `FileLoader` zum Laden verwendet werden soll. Dieser lädt lokale Dateien und benötigt einen Dateipfad als Parameter, der in der zweiten Zeile als relativer Pfad angegeben ist. In der dritten Zeile wird angegeben, dass der `CSVInputAdapter` zum Umwandeln der Daten verwendet werden soll.
Mit der Option `timeStampCaption` wird festgelegt, welche Spalte die Daten für die Zeitstempel der Ereignisse enthält. Um die Zeitstempel richtig parsen zu können, muss noch das richtige Datumsformat mit der Option `timeStampFormat` angegeben werden. In der Datumsspalte der CSV-Datei sind die Tage der Wetterdaten als vierstellige Jahreszahl gefolgt von einem zweistelligen Monat und einem zweistelligen Tag angegeben. Die Datumsbestandteile sind mit einem Bindestrich voneinander getrennt. Beim `CSVInputAdapter` wird dies durch die Zeichenkette `yyyy-MM-dd` repräsentiert.
Trennzeichen (`delimiter`) für die Spalten ist das Semikolon. Die Objekttypen der Werte lassen sich mit `columnTypes` in der Reihenfolge der Spalten angeben, wobei der Typ für die bei `timeStampCaption` angegebene Spalte beliebig ist. Diese Spalte ist für den Zeitstempel reserviert und wird ohnehin getrennt behandelt. Dass trotzdem ein Typ angegeben werden muss, dient lediglich der Einhaltung der Reihenfolge der Spalten. Es wurde deshalb für die Spalte *Datum* die Klasse `java.lang.String` als Platzhalter verwendet. Die Attribute selbst sind Ganz- oder Gleitkommazahlen und sollen daher `java.lang.Integer` bzw. `java.lang.Double` als Attributtyp haben. Für die Namen der Attribute wählt der `CSVInputAdapter` die Spaltenüberschriften.

Sonnentage im Jahr

Die erste Auswertung die durchgeführt werden soll, ist herauszufinden, wie viel Sonnentage es an der Messstation der FU Berlin hatte (Stations_ID 10381). Das Attribut *Sonnenscheindauer* enthält die Summe der Sonnenstunden pro Messtag [1]. Es gilt also die Summe aller Sonnenstunden über 365 Tage zu finden und durch 24h zu teilen. Hierfür wurde folgendes Profil erstellt:

```
Match stationsDaten, sonnenTag;
Match sonnenTage = match(event("Sonnenscheindauer", 0.0));

profile(
count(
and(and(and(stationsDaten = equal("Stations_ID", 10381),
            not(attribute("Treffer"))),
        sonnenTag = divide("Sonnenscheindauer", stationsDaten, 24.0)),
   forEvent(sonnenTag,
            setMatchTo(plus("Sonnenscheindauer", sonnenTage, sonnenTag),
                       sonnenTage))
), 365),
```

```
writerNotification(consoleWriter(),
stringOutputAdapter("Sonnentage: %f", "Sonnenscheindauer"), sonnenTage),
compositeEventNotification("Treffer", "gefunden"));
```

Zunächst werden drei Referenzen (`Match`) auf Operatoren erstellt, um auf die Ergebnisse der referenzierten Operatoren zurückzugreifen. Die Regel des Profils besteht an äußerster Stelle aus einem `count`-Operator, der einen `and`-Operator und die Zahl *365* als Parameter enthält. Der Operator trifft genau dann zu, wenn der `and`-Operator 365 mal zugetroffen hat. Innerhalb des logischen Und-Operators sollen also die Tage aufsummiert werden, so dass beim Zutreffen der Regel insgesamt 365 mal aufaddiert wurde.
Das Aufaddieren geschieht folgendermaßen: Zuerst wird die Wetterstation mittels `equal`-Operator herausgefiltert und mittels `divide`-Operator ein Vierundzwanzigstel der Sonnenstunden eines Messtages errechnet. Auf diesen Wert kann dann per `sonnenTag`-Referenz zugegriffen werden. Anschließend wird mittels `forEvent`-Operator auf dem Treffereignis von `sonnenTag` der `setMatchTo`-Operator angewendet. Dieser setzt das Treffereignis von `plus` als Treffereignis von `sonnenTage`, dessen Treffereignis anfangs mittels `match` auf ein statisches Ereignis gesetzt wurde. Die eigentliche Addition von `sonnenTage` und `sonnenTag` findet durch den `plus`-Operator statt.
Beim Treffer wird als Benachrichtigung eine `writerNotification` ausgelöst, die als *Output-Adapter* den `stringOutputAdapter` verwendet und dessen Inhalt per `consoleWriter` auf die Konsole ausgibt. Der `stringOutputAdapter` enthält als Parameter einen Format-String und eine Liste an Attributnamen, deren Werte über den Format-String ausgegeben werden sollen.
Eine Besonderheit des Profils ist, dass es eine zweite Benachrichtigung auslöst. Neben der Ausgabe des gesuchten Wertes wird noch ein neues Ereignis in den Ereignisstrom gesendet (`compositeEventNotification`). Dieses wird im Ereignisstrom unmittelbar dort eingefügt, wo das Profil zugetroffen hat, d.h. es wird von *Crepe* verarbeitet, bevor das nächste Ereignis der Wetterdaten eintrifft.
Das Senden des neuen Ereignisses hat den Sinn, für das Profil zu markieren, dass bereits ein komplettes Jahr gefunden wurde. Zwar werden durch den `equals`-Operator die nicht gewollten Station herausgefiltert, doch trifft der Operator auch zu, wenn ein Ereignis mit der *Stations_ID* 10381 nach den 365 Tagen eintrifft. Der *equals*-Operator ist so definiert, dass er seinen Trefferzustand beibehält, bis das gesamte Profil zutrifft (siehe `equals`). Da `divide` und `plus` immer zutreffen, würden auch die umschließenden Operatoren `forEvent`, `setMatchTo` und `and` zutreffen. Dadurch würde `count` erneut 365 mal die Addition erlauben.
Um dies zu vermeiden wurde das `equals` durch ein logisches Und mit einem verneinten (`not`) `attribute` verknüpft. Dadurch wird zusätzlich überprüft, ob noch kein Ereignis mit einem Attribut namens *Treffer* im Ereignisstrom vorgekommen ist. Kam ein solches Attribut vor, so kann die gesamte Regel nie mehr zutreffen und es kommt zu keiner weiteren Addition. Dies ist das hier gewünschte Verhalten.

Auf den verwendeten Wetterdaten gibt die Benachrichtigung den Text „Sonnentage: 75.679167“ aus. Was genau den Sonnentagen vom 07.05.2013 (erster Tag in den Daten) bis zum 06.05.2014 an der Messtation der Freien Universität Berlin entspricht. Das berechnete Ergebnis ist also korrekt.

Tagesfolge ohne Regen

Wetterdaten können beispielsweise auch von Regelungssystemen, wie etwa automatischen Bewässerungssystemen, ausgewertet werden. So könnte z.B. von einem solchen System eine Bewässerung eingeleitet werden, wenn es eine bestimmte Zeit lang nicht geregnet hat. Exemplarisch wurde hierfür ein Profil erstellt, was bei drei Tagen in Folge ohne Regen, eine Benachrichtigung ausgibt. Als Messstation wurde wieder die der Freien Universität Berlin gewählt:

```
Match m1, m2, m3;

profile(
    sequence(
        and(m1 = equal("Stations_ID", 10381),
        forEvent(m1, equal("Niederschlagshoehe", 0.0))),
        and(m2 = equal("Stations_ID", 10381),
        forEvent(m2, equal("Niederschlagshoehe", 0.0))),
        and(m3 = equal("Stations_ID", 10381),
        forEvent(m3, equal("Niederschlagshoehe", 0.0)))
    ),
    writerNotification(fileWriter("wetter_TageOhneRegen.csv"),
        CSVOutputAdapter("Datum", "yyyy-MM-dd", ";"), m1, m2, m3)
);
```

Die Regel des Profils filtert wieder, wie im vorherigen Beispiel, die Messtation anhand der *Stations_ID* mit dem `equal`-Operator heraus. Durch den `forEvent`-Operator wird dann auf dessen Treffererereignis ein weiterer `equal`-Operator ausgeführt. Dieser überprüft das Attribut *Niederschlagshoehe* auf den Wert *0.0*, da dies bedeutet, dass es nicht geregnet hat. Die Operatoren werden durch ein `and` verknüpft und drei Mal als Parameter für `sequence` verwendet.
Der `sequence`-Operator überprüft, ob alle seine Parameteroperatoren hintereinander zu treffen, ohne dass ein nicht zutreffendes Ereignis dazwischen liegt (vgl. Kapitel 4.2.5). Das Profil trifft deswegen zu, weil die Daten innerhalb der CSV-Datei nach *Stations_ID* und innerhalb eines *Stations_ID*-Bereiches nach Datum sortiert sind. Treffen drei Ereignisse mit derselben *Stations_ID* hintereinander ein, so repräsentieren die Ereignisse drei Messtage der Station. Haben diese Ereignisse eine *Niederschlagshoehe* von *0.0*, so hat es drei Tage lang nicht geregnet und das Profil löst seine Benachrichtigung aus.

Hier wurde für die Benachrichtigung eine Ausgabe in eine CSV-Datei gewählt. Der `fileWriter` schreibt den von `CSVOutputAdapter` ausgegebenen Inhalt in eine lokale Datei. Die Daten in der Ausgabedatei wurden manuell mit den Daten der Quelldatei verglichen und für richtig befunden.

6.1.4. Ergebnis

Es hat sich gezeigt, dass sich mit Hilfe von *Crepe* sinnvolle Auswertungen auf Wetterdaten durchführen lassen. Alle von *Crepe* erkannten zusammengesetzten Ereignisse wurden manuell überprüft und stellten sich als korrekt heraus. Weiterhin wurde demonstriert, wie man durch Senden einer `compositeEventNotification` markieren kann, dass der Ereignisraum einen bestimmten Zustand erreicht hat.
Im Übrigen wurde demonstriert, wie Quellen in *Crepe* geladen und durch Benachrichtigungen Daten an Senken geschrieben werden können. Es wurden zwei verschiedene *Output-Adapter* für die Profile verwendet und so gezeigt, wie verschieden Ausgabeformate erzeugt werden können.

6.2. Fallstudie Finanzanalyse

Zur Demonstration der Universalität von *Crepe* wurde eine zweite Fallstudie mit Daten und Auswertungen einer anderen Domäne durchgeführt. Hierfür wurden Finanzdaten analysiert, was im Folgenden näher beschrieben werden soll.

6.2.1. Motivation

In der Anwendungsdomäne der Finanzanalyse werden häufig aus Kurs- und Umsatzhistorien Schlüsse gezogen, um Vorhersagen für Kursentwicklungen zu treffen. Dies geschieht oft in Form einer so genannten *technischen Analyse* oder auch *Chartanalyse*. Eine Kursänderung kann als Ereignis betrachtet werden und legt somit nahe, für die Vorhersagen ein ereignisgesteuertes System zu verwenden. Kursdaten werden von Banken und Börsen veröffentlicht und sind daher leicht verfügbar. Dies macht sie besonders geeignet für ein Fallbeispiel.
Da Kursdaten in ihrer Natur numerisch sind und sich die Vorhersagen in der Regel am zeitlichen Kontext des Kursverlaufes orientieren, eignen sich Daten besonders, um numerische Operatoren zu evaluieren. Es kann an Beispielen demonstriert werden, wie sie sich für Auswertungen einsetzen lassen und leicht überprüft werden, ob die durch das *Crepe*-System automatisch generierten Vorhersagen mit einer manuellen Auswertung übereinstimmen.

6.2.2. Vorbereitung

Um die Fallstudie durchzuführen, gilt es zunächst geeignete Daten auszuwählen und festzulegen, welche Auswertungen damit getätigt werden sollen. Als Datensatz wurde hierbei der Euro-US-Dollar-Kurs gewählt. Dessen eigentliche Daten werden von der Europäischen Zentralbank täglich herausgegeben.

Der Datensatz wird von der Deutschen Bundesbank als Zeitreihe mit der Kennung *BBEX3.D.USD.EUR.BB.AC.000* in Form einer CSV- oder XML-Datei veröffentlicht. Er enthält ein Kursdatum pro Tag seit der Einführung des Euros als Buchgeld, d.h. seit dem 01.01.1999. Konkret besteht die CSV-Datei also aus einer Spalte für den Tag des Kursdatums und einer Spalte mit dem jeweiligen Wert.

Für Wochenenden und Bankfeiertage sind keine Kurswerte vorhanden, was sich in der CSV-Datei durch einen jeweils leeren Kurswert widerspiegelt. Es muss also aus jeder Zeile der CSV-Datei ein Ereignis erzeugt werden. Jedes der Ereignisse hat den obligatorischen Zeitstempel und genau ein Attribut, nämlich den Kurswert. Zum Zeitpunkt der Erstellung dieser Fallstudie befanden sich insgesamt 3921 Kursdaten im Datensatz.

6.2.3. Durchführung

Um die gewünschten Auswertungen durchzuführen, mussten die Daten zunächst in *Crepe* geladen werden und ein Ereignisstrom aus den Daten erzeugt werden. Da die Daten als CSV-Daten vorlagen, wurde als *Loader* der `FileLoader` und als *Input-Adapter* der `CSVInputAdapter` verwendet. Der Inhalt der *Source*-Datei zum Laden der Daten in *Crepe* sieht wie folgt aus:

```
loader = FileLoader
loader.path = testData/use_case_finance/data/euro_dollar.csv
inputAdapter = CSVInputAdapter
inputAdapter.timeStampCaption = Datum
inputAdapter.timeStampFormat = yyyy-MM-dd
inputAdapter.delimiter = ;
inputAdapter.columnTypes = java.lang.String, java.lang.Double
```

Das Einlesen gestaltet sich wie bei der Fallstudie mit den Wetterdaten (Kapitel 6.1.3). Der einzige Unterschied ist hier, dass es nur zwei Spalten gibt. Eine mit dem Datum und eine mit dem Attribut *Kurs*.

Nun, da die Daten geladen werden können, werden Profile für die Auswertungen erstellt. Profile bestehen aus einer Regel und mindestens einer Benachrichtigung. Als Benachrichtigung wurde hier wieder die Ausgabe in eine CSV-Datei gewählt.

Kursmaxima und -minima

Die erste Auswertung die vorgenommen werden soll, ist es herauszufinden, wann jeweils der Eurokurs ein neues Hoch oder Tief hat. Das Profil zur Erkennung der Kursmaxima soll also jedes mal zutreffen, wenn der zuletzt maximale Kurswert überschritten wird. Analog dazu erfolgt die Erkennung der Kursminima über einen Vergleich mit dem letzten Minimum des Kurses.
Hierzu wurden folgende *Crepe*-Profile erstellt:

```
// Maxima
Match m1 = match(event("Kurs", 0.0));

profile(
    setMatchTo(greater("Kurs", m1), m1),
    writerNotification(fileWriter("finance_maxima.csv"),
                       CSVOutputAdapter("Datum", "yyyy-MM-dd", ";"), m1)
);

// Minima
Match m2 = match(event("Kurs", 100.0));

profile(
    setMatchTo(less("Kurs", m2), m2),
    writerNotification(fileWriter("finance_minima.csv"),
                       CSVOutputAdapter("Datum", "yyyy-MM-dd", ";"), m2)
);
```

Die Profile benutzen die Größer- bzw. Kleiner-Operatoren `greater` und `less`, denen jeweils ein referenzierter statischer Operator (`m1` und `m2`) übergeben wird. Sie vergleichen also jeweils den Wert des Attributs *Kurs* von `m1` bzw. `m2` mit dem *Kurs*-Attributwert des eintreffenden Ereignisses aus dem Ereignisstrom. Wenn der Wert größer bzw. kleiner ist, dann werden die Werte von `m1` bzw. `m2` mit dem letzten Treffer überschrieben. So werden die Maxima und Minima gespeichert. Die Initialwerte wurden auf 0.0 und 100.0 mittels `match`-Operator gesetzt, da sie damit im Bezug auf den Eurokurs ausreichend klein bzw. groß sind.
Man beachte, dass bei jedem neu gefundenen Höchst- oder Tiefstwert die Regel des Profils erfüllt wird und die Benachrichtigung ausgelöst wird. Der `FileWriter` hängt dabei jeweils eine Zeile an die vorhandene Datei an. Die Dateien `finance_maxima.csv` und `finance_minima.csv` enthalten also eine Liste Höchst- und Tiefstwerten und den zugehörigen Tagen.

Gleitende Durchschnitte

Für Chartanalysen werden häufig *gleitende Durchschnitte* (engl. *moving averages*) als Trendindikatoren verwendet [29]. Der gleitende Durchschnitt ist eine Funktion mit einem Parameter n, die das arithmetische Mittel aus dem aktuellen und den n - *1* vorhergehenden Werten bildet. Bei dem hier betrachteten Eurokurs also das Mittel aus den Kurswerten des jeweils aktuellen Tags und der n - *1* vorhergehenden Tagen.
Betrachtet man den Kurswert und einen gleitenden Durchschnitt grafisch, so stellt man fest, dass der gleitende Durchschnitt eine Glättung bewirkt, die umso stärker wird, je größer n ist. Chartanalysten verwenden diese Glättung, um Kursschwankungen zu ignorieren und so Trends leichter zu erkennen. Einzelne Kauf- oder Verkaufentscheidungen werden dann beispielsweise anhand von Schnittpunkten unterschiedlicher gleitender Durchschnitte getroffen. Abbildung 6.1 zeigt einen kleinen Ausschnitt von den Eurokursdaten zwischen dem 02.01.2004 und dem 27.02.2004 zusammen mit den gleitenden Durchschnitten aus fünf und zwölf Tagen.

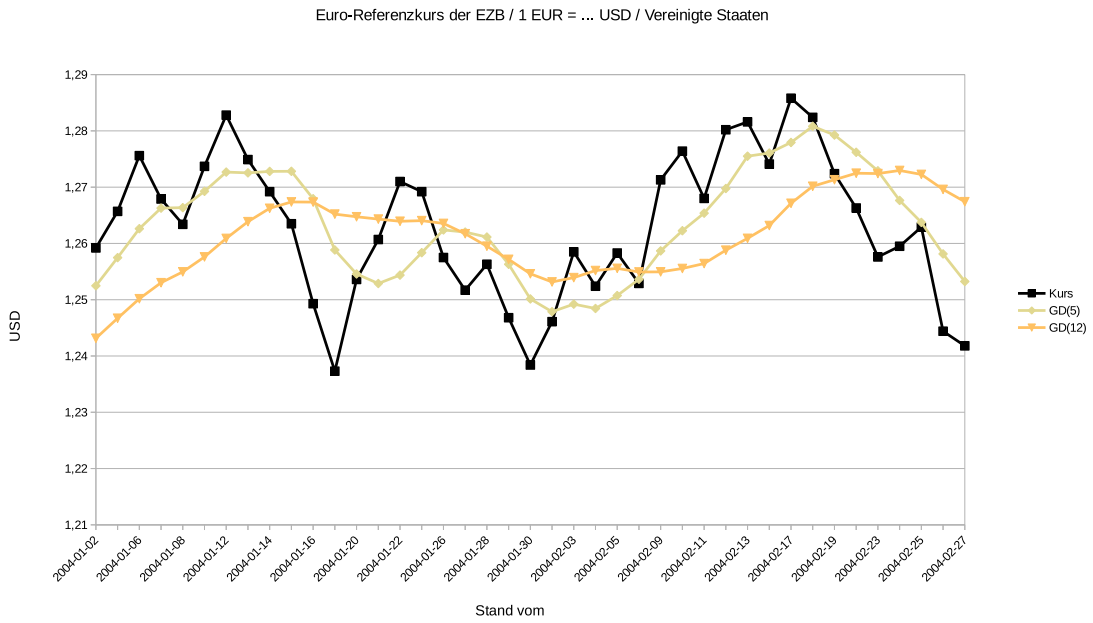

Abbildung 6.1.: Kurs mit gleitenden Durchschnitten über fünf und zwölf Tagen

Crepe hatte zunächst keinen Operator, um einen gleitenden Durchschnitt zu bilden, weswegen ein solcher für die Fallstudie implementiert wurde. Durch das Implementieren zeigte sich, dass es nur mit wenig Aufwand verbunden ist, einen neuen Operator zu *Crepe* hinzuzufügen und das Entwurfsziel der Erweiterbarkeit somit gut umgesetzt wurde.

Der `movingAverage`-Operator hat drei Parameter, einen Namen des Attributs, von dem der Durchschnitt berechnet werden soll, sowie einen Attributnamen unter dem der Durchschnitt gespeichert werden soll und die Anzahl der Werte, die für die Berechnung mit einbezogen werden sollen. Der Operator trifft immer zu und enthält als Treffererereignis ein Ereignis mit einem Attribut, dass den gleitenden Durchschnittswert enthält.
Somit können durch folgendes Profil die gleitenden Durchschnitte über fünf und zwölf Tage berechnet werden:

```
Match gd5, gd12;
profile(
    and(
        gd5 = movingAverage("Kurs", "GD5", 5),
        gd12 = movingAverage("Kurs", "GD12", 12)
    )
    ,writerNotification(fileWriter("finance_gd5.csv"),
        CSVOutputAdapter("Datum", "yyyy-MM-dd", ";"), gd5),
     writerNotification(fileWriter("finance_gd12.csv"),
        CSVOutputAdapter("Datum", "yyyy-MM-dd", ";"), gd12)
);
```

Das Profil verwendet zwei Benachrichtigungen in zwei CSV-Dateien `finance_gd5.csv` und `finance_gd12.csv`. Es wäre auch ohne Weiteres möglich, beide Werte in ein und dieselbe Datei zu schreiben. Die zwei Dateien sollen lediglich demonstrieren, wie mit einem Profil mehrere Benachrichtigungen gleichzeitig ausgelöst werden können.
Das logische Und (`and`) dient hier nur dem Zweck, dass zwei gleitende Durchschnitte von einem Profil berechnet werden, da ein Profil immer nur einen Operator als Regel haben kann. Da der `movingAverage`-Operator immer zutrifft, trifft das Profil für jedes Ereignis aus dem Ereignisstrom genau einmal zu. Die Treffererereignisse haben jeweils ein Attribut mit dem Namen *GD5* bzw. *GD12*.
Das Profil soll nun so umgeschrieben werden, dass Schnittpunkte von *GD5* und *GD12* erkannt werden.

```
Match gd5, gd12;
profile(
    and(
        and(
            gd5 = movingAverage("Kurs", "GD", 5),
            gd12 = movingAverage("Kurs", "GD", 12)
        ),
        and(
            forEvent(gd5, greater("GD", gd12)),
```

```
            forEvent(gd5, less("GD", gd12))
        )
    ),
    writerNotification(fileWriter("finance_gd_schnittpunkte.csv"),
        CSVOutputAdapter("Date", "yyyy-MM-dd", ";"), gd5, gd12)
);
```

Der Regelteil aus dem vorherigen Profil wurde in einem weiteren `and`-Operator als erster Parameter verschachtelt. Der zweite `and`-Parameter vergleicht die berechneten Durchschnittswerte miteinander. Hierfür wird der `forEvent`-Operator verwendet (vgl. 4.2.3): Er wendet den `greater`- bzw. `less`-Operator auf dem Treffererereignis von `gd5` an. Die Größer- und Kleiner-Vergleiche von `gd12` werden also nicht mit den Ereignissen aus dem Ereignisstrom getätigt, sondern mit den Treffererereignissen von `gd5`.
Die Bedingung, dass das *GD* Attribut von `gd12` sowohl größer als auch kleiner als das von `gd5` sein soll, mag zunächst unsinnig erscheinen. Man beachte jedoch, dass die `greater`- und `less`-Operatoren ihren Trefferzustand beibehalten, sobald sie einmal zugetroffen haben. Die beiden Operatoren innerhalb eines logischen Unds müssen also nicht zwingenderweise auf demselben Ereignis zutreffen und die Reihenfolge, ob der erste oder zweite Parameter zuerst in den Trefferzustand übergeht ist ebenfalls egal. Dieser Umstand wird hier ausgenutzt um Schnittpunkte zu erkennen: Ist *GD12* erst größer als *GD5* und dann kleiner als *GD5*, so haben sich deren beiden Graphen geschnitten. Genauso im umgekehrten Fall, dass *GD12* erst kleiner und danach größer als *GD5* war.

6.2.4. Ergebnis

Die von *Crepe* erzeugten CSV-Dateien wurden zur Kontrolle der Auswertungen manuell mit der Originaldatei der deutschen Bundesbank verglichen. Zur einfacheren Datensichtung wurde eine Tabellenkalkulationssoftware eingesetzt, was allerdings keinen Einfluss auf die Korrektheit der Ergebnisse hat. Es ist damit jedoch einfacher die Spalten der CSV-Dateien nebeneinander zu betrachten, um sie so zu vergleichen.
Crepe hat alle Maxima und Minima der Kurswerte korrekt erkannt. Die von den Profilen erzeugten CSV-Dateien hatten aufsteigende bzw. absteigende Wertelisten. Weiterhin wurden keine Werte, die korrekte Hoch- oder Tiefpunkte darstellen, ausgelassen. Auch sind keine nicht korrekten Werte in die Liste aufgenommen worden.
Bei der Erstellung der gleitenden Durchschnitte wurden ebenfalls per Tabellenkalkulationssoftware zunächst die berechneten Durchschnittswerte überprüft. Anschließend wurden die Werte von *GD5* und *GD12* nebeneinander aufgetragen und die von *Crepe* errechneten Schnittpunkte überprüft. Auch hier wurden wieder alle Auswertungen von *Crepe* korrekt vorgenommen.
Es hat sich insgesamt gezeigt, dass *Crepe* für Chartanalysen geeignet ist. Durch dessen numerische Operatoren lassen sich Zahlen aus der Finanzdomäne mit Hilfe von Regeln

auswerten. Weiterhin wurde demonstriert, wie man sich die Erweiterbarkeit von *Crepe* zu nutze machen kann. Hierfür wurde ein neuer Operator zur Berechnung von gleitenden Durchschnitten implementiert, um ihn dann mit bereits in *Crepe* vorhandenen Operatoren zu kombinieren.

6.3. Lasttest

Neben den beiden Fallstudien wurde noch ein kurzer Leistungstest durchgeführt. Auch hier soll wieder kurz auf die Motivation und Vorbereitung eingegangen werden und anschließend besprochen werden, wie der Test durchgeführt wurde und zu welchem Ergebnis er kam.

6.3.1. Motivation

Auch wenn beim Entwurf des Systems die Geschwindigkeit nicht im Vordergrund stand, ist es wichtig zu zeigen, dass das System für praktische Anwendungen schnell genug ist. Das gesamte System befindet sich am Anfang seiner Entwicklung und soll in Zukunft ausgebaut und optimiert werden. Daher wäre es unrealistisch, anzunehmen, dass das System in Sachen Geschwindigkeit bereits konkurrenzfähig zu jahrelang optimierten Systemen wie etwa Microsofts *StreamInsight* [8] wäre.

Dennoch soll es möglich sein, bereits jetzt mit größeren Ereignisströmen arbeiten zu können. Heutige Computer sind sehr leistungsfähig und können in der Regel problemlos mit größeren Datenströmen umgehen. Schwierigkeiten wären nur zu erwarten, wenn etwa einzelne Algorithmen – also hier Operatoren – große Laufzeiten haben. Da jedoch keiner der hier vorgestellten Operatoren eine große Laufzeit, sondern alle eine konstante Laufzeit haben, sollte die Ausführung der einzelnen Operatoren hier keine große Auswirkungen haben.

Für eine genaue, theoretische Laufzeitanalyse des gesamten Systems ist das System zu komplex und aufgrund der ausgiebig verwendeten Apache-Camel-Bibliothek zu undurchsichtig. Vielmehr lässt sich, ohne den Rahmen dieser Arbeit zu sprengen, also nur das Gesamtsystem bezüglich seiner Leistung testen. Da der hier durchgeführte Leistungstest nur einen Proof-of-Concept darstellen soll, wird die Leistung exemplarisch anhand einiger größeren Ereignisquellen durchgeführt. Nach einem erfolgreichen Testdurchlauf wird die Testdatenmenge jeweils verdreifacht, um zu überprüfen, wie *Crepe* skaliert. Durch den Aufbau von *Crepe* wäre eine lineare Skalierung zu erwarten und auch gewünscht. Die dreifache Datenmenge sollte also in etwa die dreifache Zeit bei der Auswertung benötigen.

6.3.2. Vorbereitung

Der Lasttest wurde auf einem PC mit einer Intel Core i7-2700K Quad-Core (3,5 GHz) CPU und 8 GB DDR3 Speicher durchgeführt. Alle Tests wurden auf einem Linux-System (Ubuntu 14.04) mit Kernel 3.12.0 64-Bit durchgeführt. Als Java VM kam *OpenJDK* in der Version 1.7.0_55 in der 64-Bit-Variante zum Einsatz.
Die geladenen Daten passen alle problemlos in den Arbeitsspeicher, wobei noch genügend freie Speicherreserven vorhanden sind, um dadurch jegliches Swapping zu vermeiden. Weiterhin wurden bei den Tests nur Benachrichtigungen für die Profile ausgewählt, die keine Festplattenzugriffe tätigen. Zusätzlich wurde jegliches Logging abgeschalten, um ein Verlangsamen durch die Konsolenausgabe zu vermeiden.
Dadurch ist gewährleistet, dass nur die Verarbeitungsgeschwindigkeit der Operatoren und des Grundsystems getestet wird, nicht aber eventuelle Festplattenzugriffe einen Flaschenhals bilden. Um Systemschwankungen auszugleichen wurden die Tests jeweils mehrfach durchgeführt und die Werte anschließend gemittelt. Es stellte sich heraus, dass der jeweils erste Durchlauf deutlich länger als die restlichen benötigte. Dies liegt daran, dass zu diesem Zeitpunkt noch nicht alle Objekte geladen sind. Der erste Aufruf wurde daher als statistischer Ausreißer bewertet und nicht mit in den Durchschnittswert einbezogen.

6.3.3. Durchführung

Als Daten wurden die Datensätze aus den beiden zuvor besprochenen Fallstudien verwendet. Da die Datensätze einzeln zu klein sind, um eine größere Systemlast zu erzeugen, wurden sie mehrfach eingelesen. Alle Messungen wurden mit Millisekunden-Genauigkeit durchgeführt.
Da das Augenmerk des Lasttests auf der Verarbeitungsgeschwindigkeit der Algebra lag, wurden nur die Zeitpunkte zwischen dem ersten in der Algebra eintreffenden Ereignis und dem Auslösen der letzten Benachrichtigung gemessen. So ist wiederum sichergestellt, dass alle Daten sich bereits im Arbeitsspeicher befinden und nicht Festplattenzugriffe einen weiteren Flaschenhals bilden können. Um auch bei der Benachrichtigung keine Ausgabe zu haben wurde kurzer Hand eine Benachrichtigung namens `BlankNotification` zu *Crepe* hinzugefügt, die keinerlei Funktion hat, sondern lediglich als Platzhalter in der DSL dient.
Da die Daten der beiden Fallstudien unterschiedliche Attribute haben, kommt es bei der Mischung der Daten zu einem einzigen Ereignisstrom dazu, dass Regeln, die nicht für die jeweiligen Ereignistypen geschrieben wurden, nicht alle Operatoren ausführen. Dies ist dem Umstand geschuldet, dass die Operatoren Ereignisse ignorieren, falls das vom Operator geprüfte Attribut nicht vorhanden ist. In *Crepe* ist das durch das Werfen einer `OperatorNotFoundException` gelöst. Da das die Auswertungszeit einer Regel aber erheblich verkürzt, wurden die Daten getrennt mit jeweils passenden Profilen ausgewertet.

Benchmark mit Wetterdaten

Für den Lasttest geht es nicht um sinnvolle Auswertungen. Es wurde daher ein Profil willkürlich erstellt, das allerdings einige logische Operatoren enthalten soll, die ineinander verschachtelt sind, um einen nicht flachen Operatorbaum in der Regel zu haben. Weiterhin soll das Profil aus einigen numerischen Operatoren bestehen. Das für den Benchmark verwendete Profil wurde daher wie folgt gewählt:

```
profile(or(and(equal("Qualitaet", 3),
               not(greaterEqual("Relative_Feuchte", 92.5))),
           xor(times("Mittel_Bedeckungsgrad", 20, 3.0),
               divide("Sonnenscheindauer", 30, 3.4))),
        blankNotification());
```

Dieses Profil wurde 100 mal kopiert und in eine einzelne Profildatei gespeichert. Um auf einer größeren Datenmenge zu testen, wurde der Inhalt der CSV-Datei durch Kopieren verdoppelt. Somit wurden insgesamt 55506 Ereignisse erzeugt, wobei jedes einzelne von 100 Profilen überprüft wurde. Das Einlesen der CSV-Datei geschah, wie in Kapitel 6.1 beschrieben, mit einer entsprechenden *Source*-Datei. Danach wurden fünf Messungen durchgeführt und aus den Werten das arithmetische Mittel gebildet:

Messungen mit 55506 Daten	
Durchlauf	Dauer [ms]
1	3640
2	3809
3	3687
4	3771
5	3740
Durchschnitt:	3729,4

Es lässt sich gut erkennen, dass es zwischen den Testdurchläufen keine nennenswerten Schwankungen gab. Im Schnitt wurden 3,7 Sekunden für die Verarbeitung der 55506 Ereignisse benötigt. Anschließend sollte die Skalierbarkeit im Bezug auf die Ereignismenge überprüft werden. Hierfür wurde die Datenmenge auf insgesamt 166518 Ereignisse verdreifacht und danach erneut gemessen:

Messungen mit 166518 Daten	
Durchlauf	Dauer [ms]
1	11630
2	11550
3	11444
4	11556
5	11617
Durchschnitt:	11559,4

Auch hier traten wieder kaum Schwankungen auf und es wurde ziemlich genau die dreifache Zeit wie im vorigen Testfall benötigt.

Benchmark mit Eurokursdaten

Auch für die Eurokursdaten wurde wieder ein Profil willkürlich erstellt. Um wie in der Fallstudie, wieder bei numerischen Auswertungen zu bleiben, wurden nur logische und numerische Operatoren verwendet. Konkret wurde folgendes Profil für den Test verwendet:

```
profile(and(or(not(less("Kurs", 3.0)), greater("Kurs", 1.1)),
            xor(plus("Kurs", 20, 3.0), minus("Kurs", 30, 3.4))),
        blankNotification());
```

Das Profil wurde, wie beim Benchmark mit den Wetterdaten, 100 mal kopiert und in eine Profildatei geschrieben. Anschließend wurde wieder, wie in Kapitel 6.2 beschrieben, die CSV-Datei mit den Kursdaten in *Crepe* geladen. Wie beim Wetterdaten-Benchmark wurde der Inhalt der CSV-Datei künstlich vergrößert. Hierfür wurden alle Daten zehn mal kopiert, um aus den ursprünglich 3921 Kursdaten insgesamt 39210 Daten zu machen. Insgesamt wurde also die Auswertung von 100 Regeln mit je 39210 Daten gemessen:

Messungen mit 39210 Daten	
Durchlauf	Dauer [ms]
1	2211
2	2196
3	2221
4	2159
5	2158
Durchschnitt:	2189

Im Schnitt wurden also ca. 2,2 Sekunden für die Auswertung der 100 Profile und den 39210 Ereignissen benötigt. Man erkennt leicht, dass die Schwankungen zwischen den Messungen nur sehr gering sind. Um wieder die Skalierbarkeit zu testen, wurden anschließend die Daten verdreifacht, so dass insgesamt 117630 Daten mit den 100 Profilen ausgewertet wurden:

Messungen mit 117630 Daten	
Durchlauf	Dauer [ms]
1	6827
2	6900
3	6992
4	6931
5	6560
Durchschnitt:	6842

Durchschnittlich wurden also ca. 6,8 Sekunden für die Auswertung von 117630 Ereignissen und je 100 Profilen benötigt. Wie erwartet, entspricht die Zeit für die Auswertung der dreifachen Datenmenge auch ziemlich genau der dreifachen Zeit einer einfachen Datenmenge.

6.3.4. Ergebnis

Es konnte gezeigt werden, dass *Crepe* problemlos mit größeren Datenmengen ohne Leistungseinbrüche umgehen kann. Auch wenn Vergleichswerte zu anderen Systemen fehlen und deswegen eine Einordnung der Leistung schwer fällt, kann man mit Sicherheit sagen, dass *Crepe* für sehr viele Anwendungsfälle ausreichend schnell ist. Wie erwartet kam es bei den Messungen zu keinen größeren Leistungsschwankungen.

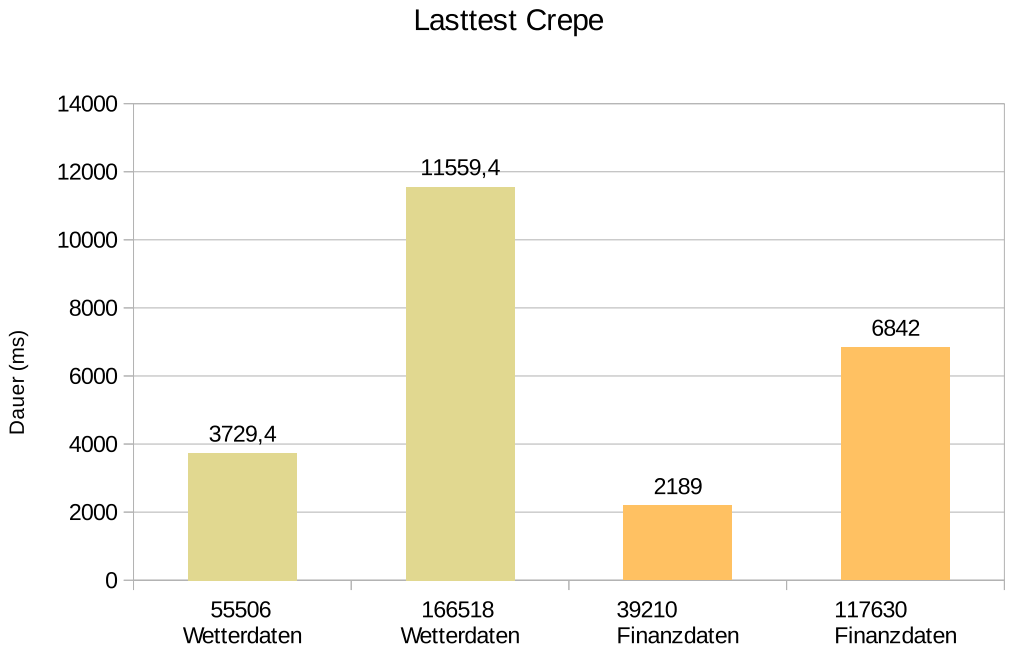

Abbildung 6.2.: Ergebnisse des Benchmarks als Säulendiagramm

Durch die Tests konnte bestätigt werden, dass das System im Bezug auf die Ereignismenge linear skaliert und das Grundsystem keinen nennenswerten Overhead hinzufügt. An Abbildung 6.2 wird dies noch mal deutlich sichtbar. Alles in allem verliefen die Tests wie erwartet und bestätigen die eingangs erwähnten theoretischen Überlegungen. Das Ziel des kurzen Lasttests wurde also erreicht.

7. Diskussion und Ausblick

In diesem Kapitel wird zusammengefasst, welche gesetzten Ziele erreicht und welche Erfahrungen dabei gewonnen wurden. Weiterhin werden Anregungen für eine langfristige Entwicklungsrichtung von *Crepe* gegeben und Erweiterungsmöglichkeiten skizziert.

7.1. Zusammenfassung und Erfahrungen

Mit dem im Rahmen dieser Arbeit entwickelten *Crepe*-System wurde der Grundstein für ein umfassendes generisches ereignisgesteuertes System geschaffen. Es wurde eine universelle Ereignisspezifikation geschaffen und als Datentyp in der Software integriert. Weiterhin wurden Schnittstellen für *Loader*, *Input-Adapter*, *Operatoren*, *Fensterfunktionen*, *Output-Adapter* und *Writer* geschaffen (siehe Kapitel 5).
Das System enthält bereits häufig benötigte Operatoren und lässt sich schon für praktische Aufgaben verwenden. Dies wurde mit Hilfe von zwei Fallbeispielen demonstriert (siehe Kapitel 6). Ebenso sind Implementierungen von *Loadern*, *Input-Adaptern*, *Output-Adaptern* und *Writern* integriert, die bereits für einzelne Anwendungsfälle genutzt werden können.
Obwohl das Thema Leistung nur angeschnitten wurde, konnte gezeigt werden, dass das System zumindest für viele Anwendungszwecke schnell genug arbeitet (Kapitel 6.3). Verarbeitungsgeschwindigkeit und Speicherverbrauch waren zwar beim Entwurf von *Crepe* kein Kriterium mit hoher Priorität, stellen aber durch die Art und Weise, wie das System die Ereignisauswertungen durchführt, kein Problem dar. Für jedes eintreffende Ereignis werden die in den Profilen definierten Operatoren ausgeführt und das Ereignis wieder aus dem Speicher entfernt, es sei denn ein Operator speichert es, um später darauf zugreifen zu können. Bei den in Kapitel 4.2 vorgestellten Operatoren werden bei den Operatoren, die auf Ereignissen arbeiten, jeweils nur die Trefferereignisse gespeichert. Operatoren, die nur auf anderen Operatoren arbeiten, wie beispielsweise logische Operatoren, speichern selbst gar keine Ereignisse.
Weiterhin haben alle in dieser Arbeit beschriebenen Operatoren eine konstante Laufzeit. Zwar ist es denkbar, auch komplexere Operatoren mit größeren Laufzeiten und höherem Speicherverbrauch zu integrieren, jedoch entsteht nicht bereits durch das darunterliegende Grundsystem ein großer Overhead.
Die Entscheidung *Apache-Camel* als Middleware zu verwenden erwies sich als gute Entscheidung. Durch die Middleware bleiben technische Aspekte im Bezug auf die Handha-

bung der Ereignisströme weitestgehend im Quelltext verborgen. Dies macht den Quelltext nicht nur besser wartbar, sondern vermeidet Fehler bei der Implementierung der Softwareteile, die durch die Middleware bereit gestellt werden. *Apache Camel* wird seit Jahren von einer Open-Source-Entwicklergemeinde aktiv gepflegt und verbessert und bietet damit einen stabilen Unterbau für *Crepe*.

7.2. Ausblick

Die Entwicklung von *Crepe* steht noch ganz am Anfang und soll in Zukunft weiter fortgeführt werden. Um eine bessere Anbindung an externe Systeme zu ermöglichen, ist es nötig, dem System weitere *Loader*, *Input-Adapter*, *Output-Adapter* und *Writer* hinzuzufügen. Auch wenn für manche Anwendungsfelder sicher Speziallösungen nötig sein werden, so ist es doch sinnvoll, dem Grundsystem weitere Komponenten für häufig auftretende Fälle hinzuzufügen. Dies erspart Arbeit bei den Anwendern und sorgt für eine höhere Akzeptanz. Je weniger Arbeit ein Anwender in die Verwendung eines Systems hineinstecken muss, desto eher ist er bereit es zu verwenden. Weiterhin unterstreicht dies den Ansatz eines universellen Systems.
Genauso verhält es sich mit den Operatoren: Je größer die Menge an verwendbaren Operatoren ist, desto mehr Anwendungsfelder können mit *Crepe* erschlossen werden, ohne dass zusätzlicher Implementierungsaufwand entsteht. Sinnvoll wäre es beispielsweise in naher Zukunft weitere numerische Operatoren zu integrieren, da diese in vielen Domänen benötigt werden.
Wie bereits erwähnt, wurde *Crepe* so konzipiert, dass es als Backend dient. Die zugrunde liegende Idee ist, dass in Zukunft ein Frontend in Form einer grafischen Benutzeroberfläche entwickelt wird, der es Benutzern ermöglicht, auf einfache Weise Quellen zu Laden und Profile zu erstellen. Idealerweise wählt der Benutzer dann nur noch im Frontend einzelne Datenquellen aus und legt das Format fest, indem er die *Input-Adapter* grafisch konfiguriert.
Das Erstellen von Regeln könnte sogar teilweise grafisch geschehen, wie bei Jung und Jones beschrieben [23]. Doch auch bei der textuellen Erstellung von Regeln könnten Anwender weiter unterstützt werden: Bisher werden Profile von *Crepe* nicht auf ihre Syntax geprüft. Es wird lediglich versucht das Profil mithilfe des Java-Compilers zu kompilieren und falls dies fehlschlägt wird der Profilimport abgebrochen. Zwar läuft das Backend im Fehlerfall einwandfrei weiter und kann erneut mit Profilen beladen werden, jedoch wäre vorherige Abfrage auf Korrektheit eines Profils sinnvoll. Dies könnte beispielsweise im Rahmen eines Profileditors umgesetzt werden.

Literaturverzeichnis

[1] Universität Bayreuth. Datenbank „Wetter in Deutschland". `http://dbup2date.uni-bayreuth.de/wetterdaten.html`, 2014. aufgerufen am 02.03.2014.

[2] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 215–226. VLDB Endowment, 2002.

[3] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

[4] Shubha Chakravarty and Yuval Shahar. A Constraint-Based Specification of Periodic Patterns in Time-Oriented Data. In *Proceedings of the Sixth International Workshop on Temporal Representation and Reasoning*, TIME '99, pages 29–, Washington, DC, USA, 1999. IEEE Computer Society.

[5] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[6] Mani K. Chandy. Event-driven applications: Costs, benefits and design approaches, 2006.

[7] Tsz S. Cheng and Shashi K. Gadia. The Event Matching Language for Querying Temporal Data. *IEEE Trans. on Knowl. and Data Eng.*, 14(5):1119–1125, September 2002.

[8] Microsoft Corporation. Microsoft StreamInsight. `http://technet.microsoft.com/en-us/library/ee362541%28v=sql.111%29.aspx`, 2014. aufgerufen am 02.01.2014.

[9] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[10] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422. www.cidrdb.org, 2007.

[11] EsperTech. Esper. `http://esper.codehaus.org/`, 2014. aufgerufen am 02.01.2014.

[12] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[13] Apache Software Foundation. Apache Camel. `http://camel.apache.org/`, 2013. aufgerufen am 01.11.2013.

[14] Apache Software Foundation. Apache Log4j 2. `http://logging.apache.org/log4j/2.x/`, 2013. aufgerufen am 12.11.2013.

[15] Apache Software Foundation. Apache Maven. `http://maven.apache.org/`, 2013. aufgerufen am 01.11.2013.

[16] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[17] Stella Gatziu and Klaus R. Dittrich. Events in an active object-oriented database system. Technical report, 1993.

[18] Narain H. Gehani, Hosagrahar V. Jagadish, and Oded Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 327–338, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[19] Narain H. Gehani, Hosagrahar V. Jagadish, and Oded Shmueli. Event specification in an active object-oriented database. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, pages 81–90, New York, NY, USA, 1992. ACM.

[20] Annika Hinze. Efficient Filtering of Composite Events. In *Proceedings of the 20th British National Conference on Databases*, BNCOD'03, pages 207–225, Berlin, Heidelberg, 2003. Springer-Verlag.

[21] Annika Hinze, Kai Sachs, and Alejandro Buchmann. Event-based Applications and Enabling Technologies. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 1:1–1:15, New York, NY, USA, 2009. ACM.

[22] Annika Hinze and Agnès Voisard. Eva: An event algebra supporting adaptivity and collaboration in event-based systems. Technical Report TR-09-006, ICSI, 2009.

[23] Doris Jung and Steve Jones. Interactive Definition of Single-user Profiles for Alerting Systems. In *Proceedings of the 22Nd Conference of the Computer-Human Interaction Special Interest Group of Australia on Computer-Human Interaction*, OZCHI '10, pages 33–40, New York, NY, USA, 2010. ACM.

[24] JUnit. A programmer-oriented testing framework for Java. `http://www.junit.org/`, 2013. aufgerufen am 01.11.2013.

[25] Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner, and Murali Mani. Event stream processing with out-of-order data arrival. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, ICDCSW '07, pages 67–, Washington, DC, USA, 2007. IEEE Computer Society.

[26] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[27] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation ina data stream management system. In *CIDR 2003*. Stanford InfoLab, 2002.

[28] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed event-based systems*, volume 1. Springer, 2006.

[29] J.J. Murphy. *Technische Analyse der Finanzmärkte: Grundlagen, Methoden, Strategien, Anwendungen ; incl. „Workbook Technische Analyse"*. Trading Seminar Reihe. FinanzBuch-Verlag, 2011.

[30] Christopher Mutschler and Michael Philippsen. Reliable Speculative Processing of Out-of-order Event Streams in Generic Publish/Subscribe Middlewares. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 147–158, New York, NY, USA, 2013. ACM.

[31] Oracle. Java Programming Language. `http://www.java.com`, 2014. aufgerufen am 01.03.2014.

[32] Adrian Paschke. Eca-ruleml: An approach combining eca rules with temporal interval-based kr event/action logics and transactional update logics. *CoRR*, abs/cs/0610167, 2006.

[33] SASE. SASE. `http://sase.cs.umass.edu/`, 2013. aufgerufen am 04.12.2013.

[34] Scarlet Schwiderski-Grosche and Ken Moody. The SpaTeC Composite Event Language for Spatio-temporal Reasoning in Mobile Systems. In *Proceedings of the*

Third ACM International Conference on Distributed Event-Based Systems, DEBS '09, pages 11:1–11:12, New York, NY, USA, 2009. ACM.

[35] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pages 43–50, New York, NY, USA, 2011. ACM.

[36] Cornell University Database Systems. Cayuga. `http://www.cs.cornell.edu/bigreddata/cayuga/`, 2014. aufgerufen am 03.04.2014.

[37] Christian Tominski. *Event-Based Visualization for User-Centered Visual Analysis*. PhD thesis, University of Rostock, Germany, University of Rostock, November 2006.

[38] Agnès Voisard and Holger Ziekow. ARCHITECT: A layered framework for classifying technologies of event-based systems . *Information Systems*, 36(6):937 – 957, 2011.

[39] Bing Wu and Kudakwashe Dube. PLAN: A Framework and Specification Language with an Event-Condition-Action (ECA) Mechanism for Clinical Test Request Protocols. In *In Proceedings of 34th Annual Hawaii International Conference on System Sciences (HICSS-34*. IEEE Press, 2001.

[40] XTEXT. Framework for development of programming languages and domain specific languages. `http://www.eclipse.org/Xtext/`, 2013. aufgerufen am 01.09.2013.

[41] Robert Zhang and Elizabeth A. Unger. Event Specification and Detection. Technical report, 1996.

Abbildungsverzeichnis

A. CD-ROM-Inhalt

Dieser Arbeit liegt eine CD-ROM bei, die diesen Text als PDF-Datei enthält. Weiterhin befinden sich der gesamte Quelltext von *Crepe* mit dem Stand des Abgabedatums dieser Arbeit, sowie die für die Evaluation verwendeten Testdaten auf der CD. Der jeweils aktuelle Stand des Quelltextes von *Crepe* ist auch als Projekt auf Github unter der Adresse `https://github.com/kressnerd/ems` verfügbar. *Crepe* steht unter der Apache Lizenz Version 2.0 (`http://www.apache.org/licenses/`).

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, 12. Mai 2014 Ralf Öchsner