Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Pascal Jäger

Umsetzung der Kommunikationsmodellierungssprache ModoCom mit Esper

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Pascal Jäger

Umsetzung der Kommunikationsmodellierungssprache ModoCom mit Esper

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Olaf Zukunft
Zweitgutachter : Prof. Dr. rer. nat. Bettina Buth

Abgegeben am 5. Juli 2010

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einführung

1.1 Motivation

Im heutigen Informationszeitalter ist die Verarbeitung und Gewinnung neuer Informationen wichtiger denn je. Dabei ist es auf Grund der Masse an Daten schwierig, relevante Daten von nicht relevanten Daten zu unterscheiden. Erschwerend kommt hinzu, dass teilweise das Verarbeiten der Daten auf Grund ihrer Verschlüsselung sehr aufwändig ist und im Vorfeld entschieden werden muss, welche Daten einer weiteren Verarbeitung zugeführt werden sollen. Das heißt, es müssen Daten, unabhängig von ihrem Inhalt, als Ganzes betrachtet und ausgewertet werden.

Dieses Problem besteht beispielsweise im Bereich der Kommunikation via Funk und Kommunikation im Internet. Für die Kommunikation via Funk kann verallgemeinernd angenommen werden, dass Informationen verschlüsselt und “ziellos” als Broadcast in die Umgebung abgesetzt werden. Da Anfrage und Antwort zunächst in keinem offensichtlichen Zusammenhang stehen, ist es hier besonders schwer, eine Kommunikation zwischen zwei oder mehr Personen zu identifizieren. Im Idealfall ist nur der Sender, aber nicht der oder die angedachten Empfänger bekannt.

Betrachtet man Funkemissionen als Ganzes, lassen sich unter anderen folgende Gemeinsamkeiten feststellen, aus denen Rückschlüsse auf eine stattfindende Kommunikation gezogen werden können:

- Lokalität des Senders
- Frequenz der Emission
- Startzeit der Emission
- Dauer der Emission

Kommunikation im Internet, basierend auf IP-Paketen, hat den Vorteil, dass im IP-Header sowohl Quell- als auch Zieladresse mitgeführt werden, sich hier also Emitter und Empfänger einer Emission feststellen lassen. Die in der Transportschicht verwendeten Protokolle (TCP und UDP) nutzen Ports, um den Empfänger einer Emission an einem Zielhost eindeutig zu bestimmen. Dennoch ist ein Zielhost dadurch nicht eindeutig beschrieben, da der eigentliche Empfänger meistens hinter einer so genannten NAT-Box sitzt, welche beispielsweise die Adressen eines Firmennetzwerks von Adressen des Internets entkoppelt. Daher sind in der Regel die Quell- und Ziel-IP eines IP-Paketes Adressen von NAT-Boxen, welche jeweils das Sender-Netz und das Empfänger-Netz vom Internet entkoppeln. Die

eigentlichen Kommunikationsteilnehmer in den dahinter liegenden Netzen lassen sich nicht mehr bestimmen.

Basierend auf dem Verständnis, dass Kommunikation ein Informationsaustausch ist, also die Kombination von zwei oder mehr Emissionen zwischen zwei oder mehr Emittern, lassen sich allgemeine Modelle erstellen, mit denen sich bestimmte Kommunikationsformen innerhalb eines bestimmten Bereiches (Funk, Internet, Handy) beschreiben lassen. Als Beispiele seien hier aus dem Funk-Bereich die so genannte *Simplex Kommunikation* erwähnt, bei der zwei Sender sequentiell Emissionen erzeugen und die so genannte *Duplex Kommunikation*, bei der zwei Sender eine andauernde Verbindung auf unterschiedlichen Frequenzen nutzen. Die Begriffe *Simplex* und *Duplex* aus dem Funk-Bereich lassen sich jedoch nicht ohne weiteres auf Kommunikation im Internet übertragen, da die Internetkommunikation nicht notwendigerweise nur über ein Medium abläuft. Mögliche Medien sind Kupfer- oder Glasfaserkabel sowie WLAN- und Satelliten-Verbindungen, die für eine Kommunikation alle gleichzeitig genutzt werden können, abhängig von der Anbindung der Kommunikationspartner und den getroffenen Routing-Entscheidungen. Betrachtet man Internetkommunikation als Ganzes, lassen sich auch hier gemeinsame Eigenschaften für Emissionen feststellen:

- Ziel-IP-Adresse
- Quell-IP-Adresse
- Ziel-Port
- Quell-Port (abhängig vom verwendeten Protokoll)
- Startzeit der Emission
- Anzahl übertragener Bytes

Verknüpft man nun die Eigenschaften von Emissionen mit den Einschränkungen, die sich aus einer Kommunikationsform ergeben, so ist es möglich, einen großen Anteil aller empfangenen Emissionen als uninteressant auszuschließen, so dass nur noch ein kleiner Teil der Daten einer weiteren Verarbeitung zugeführt werden muss.

Um die Kommunikationsformen unabhängig von einer technischen Umsetzung[1]

[1] Die technische Umsetzung bezieht sich hier auf das System, das die ModoCom Programme verarbeitet und nicht auf die technische Umsetzung einer Kommunikation. Dieses Wissen ist zwangsweise notwendig für die Beschreibung einer Kommunikation.

beschreiben zu können, wurde bei der Fa. Plath GmbH in Hamburg die Sprache ModoCom (Modelling of Communications) entwickelt [7] [8], welche es Anwendern in diesem Bereich ermöglicht, eine Kommunikation zu beschreiben ohne dabei Wissen über das verarbeitende System zu besitzen.

Generell beruht ModoCom auf einem Ereignis-Modell (*event-model*). Dabei können die Ereignisse (*events*), die einem solchen Modell zu Grunde liegen, in folgende Kategorien eingeteilt werden, wie sie in [2] beschrieben sind:

transient event Die Sensorik stellt in kleinen zyklischen Abständen (Abtasten) das Eintreffen einer Emission fest.

silent events Die Sensorik stellt beim Abtasten das Ausbleiben eines Emissionsereignisses fest. Also die Abwesenheit eines *transient events*.

atomic events Die Sensorik erkennt den Zusammenhang der *transient*- und *silent events* und verkettet diese zu einem so genannten atomaren Event. Ein atomarer Event bildet die Grundlage für ModoCom, stellt also eine Funkemission (ein Funkspruch) oder eine “IP-Emission” (ein IP-Paket) dar, die mittels ModoCom zu einer Kommunikation verknüpft werden können.

compound events Atomare Events können zu *compound events* zusammengesetzt werden. Innerhalb eines *compound events* haben atomare events Relationen die zeitlicher, kausaler und lokaler Ausprägung sind. Das heißt, zwei Events können gleichzeitig (*fully parallel*), teilweise gleichzeitig (*partial parallel*) oder nacheinander (*sequential*) stattfinden. Sie können kausal abhängig sein oder nicht und sie können am selben Ort stattfinden oder nicht.

Bei der Umsetzung von ModoCom konnten, zusammen mit der Fa. Plath, zwei grundlegende Möglichkeiten identifiziert werden:

- Online-Analyse
- Offline-Analyse

Die Begriffe *Offline* und *Online* bringen zum Ausdruck, dass eingehende Ereignisse entweder direkt bei ihrem Eintreffen in der Anwendung verarbeitet werden (*Online*) oder Ereignisse gesammelt und zwischengespeichert werden, um zu einem späteren Zeitpunkt ausgewertet zu werden (*Offline*).

Bei einer ersten Umsetzung von ModoCom wurde der Offline-Ansatz gewählt. Die von der Sensorik generierten Daten werden in eine Datenbank geschrieben, in

der sie mittels Clustering auf so genannte EventGroups heruntergebrochen werden. Eine EventGroup ist eine Menge von Events mit gleichen Eigenschaften. Im Falle von Clustering als Vorverarbeitungsschritt entspricht eine EventGroup also genau einem Cluster.
Ein Ziel, neben der Datenreduktion, ist dabei die Gewinnung statistischer Informationen. So lässt sich beispielsweise bei Funkemissionen nach dem Ursprungsort der Emissionen clustern, wodurch Rückschlüsse auf erhöhte Aktivität in einem Gebiet möglich sind. Bei Internet-Emissionen wären hier beispielsweise Subnetze ein mögliches Ziel für Clustering. Diese EventGroups bilden die Grundlage für eine Auswertung durch ModoCom. Das ModoCom Programm eines Anwenders wird in ein LISP-Programm übersetzt und mit den Daten aus der Datenbank evaluiert.

Das Problem des Offline-Ansatzes ist die zeitliche Differenz zwischen dem Auftreten eines oder mehrerer Events (also dem Stattfinden einer Kommunikation) und dem Erkennen dieser Kommunikation. Das zeitnahe Erkennen kann im Bereich der Fernmeldeaufklärung, aber auch bei der so genannten Intrusion Detection in Netzwerken von Interesse sein.

Das Paradigma Complex Event Processing (CEP - siehe Abs. 2.2) bietet die Möglichkeit einer solchen Online-Analyse von Events. Das Thema CEP hat im Laufe der letzten Jahre den Sprung aus dem universitären Umfeld in die Industrie geschafft. Mittlerweile gibt es diverse Tools und Engines, auf Basis derer sich CEP-Anwendungen entwickeln lassen. Als wichtigstes OpenSource Tool ist hier die Event Processing Engine Esper (Abs. 2.3) zu nennen, welches von der Firma EsperTech entwickelt wird. Die Esper Engine bietet die Möglichkeit Events und Event Streams in nahezu Echtzeit zu verarbeiten und auf sie zu reagieren. Daher untersucht diese Arbeit, wie am Beispiel von Esper, ModoCom-Programme in einem Online-Ansatz umgesetzt werden können.

1.2 Ziel der Arbeit

Die Arbeit untersucht, inwieweit die Sprache ModoCom der Fa. Plath GmbH auf eine Event Processing Language (EPL), hier am Beispiel von Esper, abgebildet werden kann. Dabei sollen nicht nur die Möglichkeiten für die Abbildung nach Esper aufgezeigt, sondern auch eventuell auftretende Probleme identifiziert werden. Zusätzlich soll in dieser Arbeit ein Framework entworfen werden, mit dem es möglich ist, ModoCom-Programme mit in einer CEP-Engine auszuführen. Teile des Framework sollen in Form eines Prototyps realisiert werden.

1.3 Aufbau der Arbeit

Zunächst werden in dieser Arbeit die Grundlagen (Kapitel 2) für die Problemstellung erarbeitet. Dabei wird auf die Sprache ModoCom, der Fa. Plath GmbH, eingegangen, sowie auf ihre Umsetzung als Offline-Ansatz. Des Weiteren werden Complex Event Processing im Allgemeinen und Esper im Speziellen beschrieben. In Kapitel 3 wird untersucht, welche Konzepte in ModoCom existieren und diskutiert, wie diese Konzepte für eine Realisierung mit einer CEP-Engine umgesetzt werden können. Anschließend wird in Kapitel 4 auf Basis der besprochenen Konzepte eine Framework konzipiert, mit dem der Anwender später ModoCom-Programme ausführen können soll. Zusätzlich wird ein Prototyp eingeführt, dessen Implementierung in Kapitel 5 besprochen werden soll. In Kapitel 6 werden alle Ergebnisse der Arbeit zusammengefasst, bewertet und ein Ausblick auf zukünftige Aufgaben und zu lösende Probleme gegeben.

2 Grundlagen

Im ersten Teil dieses Kapitels werden der Aufbau und die Bestandteile der Sprache ModoCom beschrieben. Dabei wird das, von der Fa. Plath GmbH entwickelte ModoCom zuerst unabhängig von seiner Implementierung beleuchtet, um anschließend auf die Aspekte der Offline-Ansatzes einzugehen. Im zweiten Teil wird ein Einblick in Complex Event Processing (CEP) gegeben, um anschließend die Event Processing Engine Esper und ihre Event Processing Language (EPL) [9] vorzustellen.

2.1 ModoCom

Die Sprache ModoCom (Language for Modelling of Communications) wurde von der Fa. Plath GmbH entwickelt, um Kommunikationsmodelle in Funk-Emissionsdaten beschreiben zu können, mit denen dann Funkemissionen gefunden werden können, die eine mögliche Lösung des Modells darstellen. Dabei ist ModoCom nicht auf die Verwendung von Funkemissionen limitiert, sondern kann mit Emissionen im Allgemeinen umgehen. Der Begriff Emission wird in dieser Arbeit als die Abgabe beziehungsweise Ausstrahlung von Informationen an die Umwelt verstanden (vgl. engl: emission) und entsprechend verwendet.

In [11] werden Events ganz allgemein als Ereignisse von Interesse beschrieben. Das Ereignis von Interesse ist in diesem Fall das Eintreffen bzw. Auftreten einer Emission, weshalb in dieser Arbeit Emission und Event synonym verwendet werden. ModoCom selbst definiert einen Event als ein beobachtetes, primitives Objekt, das durch die mit einem Sensor empfangenen Daten wie beispielsweise Ortung und Dauer beschrieben ist [8].

2.1.1 Sprachelemente

In diesem Abschnitt werden die umzusetzenden Sprachelemente untersucht. Zunächst soll ein Überblick über den Zusammenhang der einzelnen Elemente gegeben werden, an dem sich der Leser orientieren kann (Abbildung 1).

ModoCom gruppiert Events in so genannten *EventGroups*. Alle Events innerhalb dieser EventGroup sind bezüglich einer Eigenschaft, beispielsweise der Modulation, gleich. Diese EventGroups werden an so genannten *CommunicationOperators* übergeben, welche zusammen mit *Predicates* (Einschränkungen für Events) diese Untersuchen und in *Communications* einteilen. Der verwendete *CommunicationOperator* und die Predicates werden im *CommunicationModel* vom Anwender

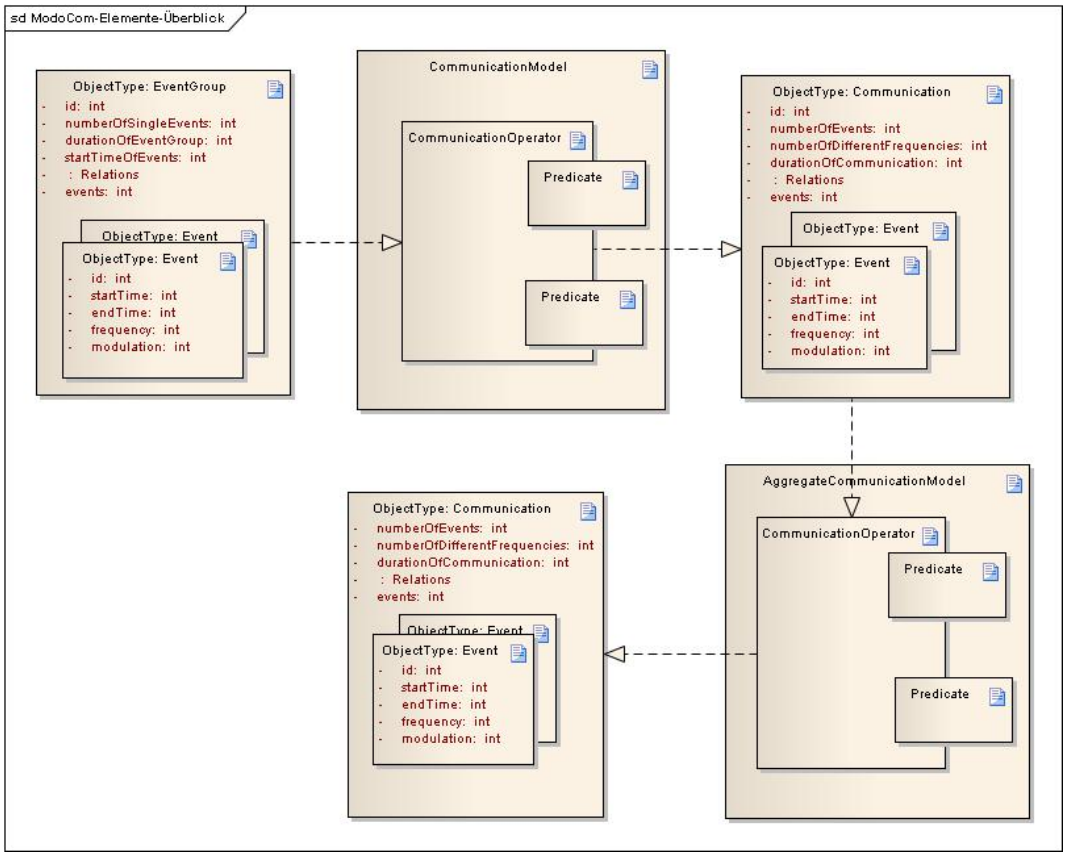

Abbildung 1: ModoCom-Sprachelemente im Überblick

definiert. Der Anwender hat anschließend die Möglichkeit mit Hilfe von *AggregateCommunicationModels* die gefundenen Communications weiter zu untersuchen. Dabei können diese aggregierten Modelle nur auf Communications, nicht aber auf EventGroups arbeiten.

Die nachfolgenden Abschnitte gehen nun näher auf diese Sprachelemente ein und zeigen, wie diese in ModoCom definiert werden.

ObjectTypes ObjectTypes definieren die in ModoCom verwendeten Datentypen. Das können Events, EventGroups oder Communications sein, auf die später im Kapitel näher eingegangen wird.
ObjectTypes werden mit dem Keyword *ObjectType* und einem Namen definiert. Abhängig von der jeweiligen Kommunikation beziehungsweise den Sensoren, welche Emissionen einer Kommunikation auffangen, haben die ObjectTypes unter-

schiedliche Attribute (vgl. Abschnitt 1.1). Listing 1 zeigt einen ObjectType, der als vordefinierter Typ durch die Fa. Plath GmbH spezifiziert wurde.

Listing 1: Definition eines EventGroup-ObjectTypes für Funk-Emissionen [8]

```
ObjectType EventGroup [
    About: "Definition of an event group";
    Parameter:
        (name: id, default: nil);
        (name: numberOfSingleEvents, default: nil);
        (name: durationOfEventGroup, default: nil);
        (name: startTimeOfEvents, default: nil);
        (name: centerLongitude, default: nil);
        (name: centerLatitude, default: nil);
    Relations:
        (name: events, objecttype: SingleEvent);
    Sorting:
        (property: events, function: <,
         key: SingleEvent-startTime);
]
```

- **About** Mit *About:* kann eine Beschreibung für den ObjectType angegeben werden.
- **Parameter** Mit *Parameter:* werden die Attribute des ObjectTypes definiert. Dabei besteht jeder Parameter aus einem Namen und einem Default-Wert.
- **Inherit** ModoCom ist in der Lage, Attribute von einem vorher definierten ObjectType zu erben. Diese Vererbung wird im Feld *Inherit:* angegeben (in Listing 1 nicht enthalten).
- **Relations** *Relations* bieten über eine Listenstruktur Zugriff auf Elemente innerhalb des ObjectTypes. Dies können beispielsweise die Events in einer EventGroup sein.
- **Sorting** *Sorting* erlaubt das Sortieren der Elemente in einer *Relations*-Liste. Darüber lässt sich steuern, in welcher Reihenfolge die Elemente der Relation evaluiert werden.

Restrictions Eine Kommunikation unterliegt gewissen Eigenschaften. Ziel ist es also, mit ModoCom Eigenschaften zu beschreiben, anhand derer Events ausgewählt werden können, die zur Kommunikation gehören. Dies geschieht in ModoCom mit so genannten Restrictions, die in drei Arten unterteilt werden können:

a) Einfache Restrictions Einfache Restrictions sind logische und arithmetische Ausdrücke. Sie sind Teil eines Predicates und müssen nicht explizit definiert werden.

b) Predicates Ein Predicate wird mit dem Keyword *Predicate* und einem Namen definiert und bekommt als Parameter eine Anzahl an Events übergeben. Sie dienen zur Steuerung der CommunicationOperators. Im Rumpf der Predicate-Definition (*PredicateRestrictions*) werden einfache Restrictions aufgelistet. Diese Restrictions sind implizit UND-verknüpft.

Listing 2: Definition eines Predicates[8]

```
1  Predicate EquivalencePredicate (SingleEvent e1, SingleEvent e1){
2      About: "Defines when two events are equal, i.e.
3              the frequencies are equal and the
4              modulationtypes are equal or one is unkown";
5
6      PredicateRestrictions:
7              e1.frequency = e2.frequency &&
8              (e1.modulationtype = e2.modulationtype ||
9               e1.modulationtype = "UNK" ||
10              e2.modulationtype = "UNK");
11 }
```

c) CommunicationOperator Ein CommunicationOperator wird nicht in ModoCom definiert, sondern ist Teil des Systems auf dem ModoCom aufsetzt. Ein CommunicationOperator bekommt als Parameter EventGroups übergeben auf denen er operiert. Als zusätzliche Parameter bekommt er Predicates, anhand derer der Anwender den Operator steuern kann. CommunicationOperators können nicht direkt in ModoCom modelliert beziehungsweise definiert werden, sondern sind Teil der ModoCom-Implementierung. Ein beispielhafter Aufruf des CommunicationOperators *FindAlternatingCommunicationsInTwoGroups*, wie er im Offline-Ansatz implementiert ist, wird in Listing 3 gezeigt.

CommunicationModel Ein CommunicationModel beschreibt die Eigenschaften einer Kommunikationsform. Das Modell fasst also alle notwendigen Einschränkungen (CommunicationOperators und Predicates) zusammen. Zusätzlich werden in einem CommunicationModel die verwendeten EventGroups definiert.

ModoCom unterscheidet zwei Typen von CommunicationModels:

a) CommunicationModel Ein einfaches Kommunikationsmodell wird mit dem Keyword (*CommunicationModel*) und einem Namen definiert und modelliert eine Kommunikation auf Basis von EventGroups. Es liefert als Ergebnis eine Liste von Lösungen, so genannte *Communications* . Eine Lösung ist eine mögliche Menge von Events, die die Eigenschaften des CommunicationModels erfüllen. Listing 3 zeigt die Definition eines Modells der Fa. Plath GmbH.

Listing 3: Definition eines einfachen CommunicationModel [8]

```
CommunicationModel TwoPartnerSimplex (delay: 12){
    About: "The communication can have an arbitrary number
            of events, but those should have same frequency
            and modulation type, should belong to distinct
            event groups and shoud succeed each other with
            a default maximum delay of 12 seconds.";
    EventGroups:
            (name: m, objecttype Main);
            (name: s, objecttype Sub);
    Restrictions:
        FindAlternatingCommunicationsInTwoGroups(m, s,
            EquivalencePredicate,
            BetweenGroupRestrictions(delay),
            InGroupRestrictions(delay),
            RepresentativePredicate);
}
```

b) AggregateCommunicationModel Aggregierte CommunicationModels werden mit dem Keyword *AggregateCommunicationModel* und einem Namen definiert. Im Unterschied zu einfachen Modellen, modellieren aggregierte Kommunikationsmodelle Kommunikation auf Basis von Lösungen, die von einfachen oder anderen zusammengesetzten Modellen geliefert werden. Sie lassen sich beliebig schachteln, wodurch eine beliebig komplexe Kommunikation modelliert werden kann. Aggregierte CommunicationModels, respektive die in einem Modell verwendeten Predicates und CommunicationOperators, arbeiten nur auf den Attributen der Communications, nicht aber auf den Attributen der Events, die sich in der Communication befinden. Listing 4 zeigt die Definition eines aggregierten Modells der Fa. Plath GmbH.

Listing 4: Definition eines aggregierte CommunicationModel [8]

```
AggregateCommunicationModel TwoPartnersSimplexInSameEventGroups (
        delay: 120000,
        minNumberOfEventGroups: 2)
[
    About: "Combine simplex communications";
    EventGroups:
        (name: m, objecttype: Main);
        (name: s, objecttype: Sub);
    InvolvedModels:
        (name: tp, modeltype: TwoPartnerSimplex,
        solutions: tpe);
    Restrictions:
        EvalModel(tp, m, s, delay);
        AggregateCommunicationsOfOneSubmodel(tpe,
            EquivalencePredicate,
            CommunicationRestrictions(
                    minNumberOfEventGroups),
            FilterPredicate);
]
```

- **EventGroups** *EventGroups* definieren die zu verwendenden Eingabe-ObjectTypes. In den Listings 3 und 4 werden zwei EventGroups (Main und Sub) verwendet, die mittels *Inherit* erzeugt wurden.

Listing 5: Erben von einem ObjectType [8]

```
ObjectType Main [
    About: "Definition of a main communication station";
    Inherit: EventGroup
]
```

Dabei entspricht der unter *Inherit* aufgeführte ObjectType genau dem aus Listing 1. Das Definieren von Sub erfolgt analog zu Listing 5.

- **InvolvedModels** Dieses Keyword steht nur in AggregateCommunicationModels zur Verfügung und definiert das dem aggregierten Modell zu Grunde liegende CommunicationModel. Dabei kann das zu Grunde liegende Modell ein einfaches oder aggregiertes Modell sein. Wie in Listing 4 zu sehen ist, wird der Name des *involved models* (tp) dazu verwendet, um das Modell mittels dem Operator *EvalModel* aufzurufen. Die Solutions (tpe) werden dem CommunicationOperator des aggregierten Modells übergeben.

- **Restrictions** Hier wird der im CommunicationModel verwendete CommunicationOperator aufgerufen. Dabei ist es im aggregierten Modell wichtig, vor-

her das zu Grunde liegende Modell auszuwerten. Dies geschieht mit dem Operator *EvalModel*.
In Listing 4 lässt sich gut erkennen, dass das aggregierte Modell alle notwendigen Parameter, auch die für das *involved model* (hier: delay) mit einem Wert belegen muss, welche mittels *EvalModel* weitergereicht wird. Ebenso muss das aggregierte Model, die im *involved model* benötigten EventGroups definieren.

2.1.2 Offline-Ansatz

Die Plath GmbH hat zusammen mit dem HITEC e.V.[2] ModoCom als Constraint-Systems in LISP realisiert. Die Umsetzung entspricht dem Offline-Ansatz, wie er in Abschnitt 1.1 beschrieben ist. Die Einteilung in EventGroups erfolgt in einem Vorverarbeitungsschritt mittels Clustering , wobei aktuell nur so genannte *spatial cluster* erzeugt werden, da die LISP-Implementierung interne Optimierungen für diesen Fall vornimmt. Frequenz-Cluster (oder Event-Gruppen mit anderen Merkmalen) werden aktuell nicht unterstützt.

CommunicationOperators CommunicationOperators werden nicht, wie Predicates, direkt im ModoCom-Programm definiert, sondern sind Teil des Systems in dem ModoCom realisiert wird. Das hat zur Folge, dass der Anwender ohne Wissen über die zu Grunde liegende Implementierung nur die in der Implementierung vorgegebenen CommunicationOperators verwenden kann. Die ModoCom-Implementierung in LISP kennt vier CommunicationOperator:

- FindAlternatingCommunicationsInTwoGroups
- FindCommunicationsInOneGroup
- AggregateCommunicationsOfOneSubmodel
- EvalModel

Die ersten beiden CommunicationOperator erhalten EventGroups als Eingabe und erzeugen einfache CommunicationModels. Der dritte Operator arbeitet mit den Ergebnissen der ersten beiden Operatoren und erzeugt zusammengesetzte CommunicationModels.

Der vierte Operator ermöglicht es einem zusammengesetzten CommunicationModel ein einfaches CommunicationModel zu evaluieren und dessen Ergebnisse zu

[2] Hamburger Informatik Technologie-Center e.V.

nutzen (siehe Listing 4).

Am Namen des dritten Operators lässt sich erkennen, dass die LISP-Umsetzung von ModoCom nur die Verwendung eines Sub-Modells zum Erstellen komplexerer CommunicationModels erlaubt. Dies soll jedoch keine Einschränkung darstellen, da sich mit diesem Operator zusammengesetzte CommunicationModels schachteln lassen können [8].

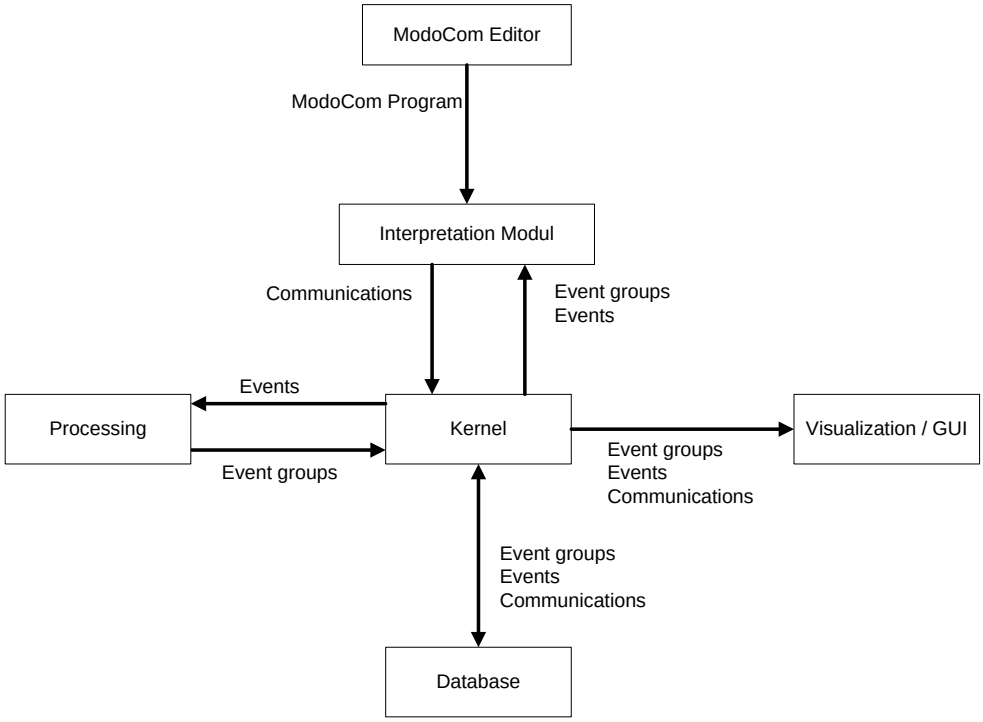

Abbildung 2: System Overview aus [8]

Allgemeiner Ablauf Der Anwender kann, in einer für ihn leicht verständlichen Sprache (ModoCom), ein Modell für eine Kommunikation beschreiben, ein so genanntes ModoCom-Programm. Dabei kann er auch einzelne (einfache) Modelle zu komplexeren Modellen verknüpfen. Bevor Modelle mit ModoCom interpretiert beziehungsweise evaluiert werden können, müssen die Events in einem Vorverarbeitungsschritt in EventGroups eingeteilt werden. Diese EventGroups bilden die Grundlage für eine Evaluation mit ModoCom und sind in [8] definiert als

> „. . . eine Menge von Events, die durch einen Vorverarbeitungsschritt (beispielsweise Clustering) entstanden sind. Abhängig von diesem

> Vorverarbeitungsschritt haben die Events einer solchen Menge Gemeinsamkeiten bezüglich ihrer Ortung, Frequenz oder anderen Merkmalen.“

Die Einteilung der Events in EventGroups hat folgenden Hintergrund. Zum einen wird die Komplexität reduziert, da nicht jeder Event mit allen anderen verglichen wird, sondern man nun Events innerhalb einer EventGroup oder gezielt die Events zweier EventGroups vergleichen kann. Zum anderen liefert der Vorverarbeitungsschritt zusätzliche Informationen. So kann man beispielsweise durch ein Clustering, bezogen auf die Ortung der Emissionen, Gebiete finden, in denen erhöhtes Funkaufkommen ist und gezielt diese Gebiete untersuchen.

Für den Offline-Ansatz bietet sich, wie bereits oben erwähnt, als Vorverarbeitungsschritt ein Clustering an. Dafür werden die Events in einer Datenbank gespeichert. Das Ergebnis des Clusterings kann ebenfalls in der Datenbank gespeichertwerden, so dass das Interpretationsmodul bei der Evaluierung dort die notwendigen Daten abrufen kann.

Die Evaluierung eines ModoCom-Programms erfolgt im Interpretationsmodul (vgl. Abbildung 2). Ein ModoCom-Programm beschreibt ein CommunicationModel, welches in [8] definiert wird als „. . . Restriktionen, die für eine bestimmte Art von Kommunikation erfüllt sein müssen“.

Im Interpretationsmodul wird das Modell evaluiert, das heißt es werden Events in den EventGroups gesucht, welche die im Modell beschriebenen Restriktionen erfüllen.

Ein beispielhafter Systemaufbau, mit einer Datenbank als Grundlage für die Datenhaltung, ist in Abbildung 2 zu sehen.

2.1.3 Bewertung und Zusammenfassung

Die ursprüngliche Intention für die Entwicklung von ModoCom war die Modellierung und Erkennung von Kommunikation in Funkemissionsdaten [7]. ModoCom ist aber nicht auf diese Domäne beschränkt, sondern kann Kommunikation allgemein beschreiben und erkennen.

Problematisch ist die unklare Trennung zwischen der Sprache ModoCom und dem realisierenden System. Diese wird vor allem bei den Strategy-Objects ([8]) deutlich, welche hier nicht besprochen wurden.

Hinzu kommen Relikte früherer ModoCom-Versionen, die leider noch immer in der

aktuellen Dokumentation ([8]) enthalten, aber nicht mehr Teil der aktuellen Sprache sind. Dazu gehören:

- **primaryKeys** Diese können zur Definition von Attributen in ObjectTypes verwendet werden.
- **InvolvedModels** Die Variablen first und next, die in [8] in Figure 10 und 12 auftauchen. Sie sollen dem CommunicationOperator Zugriff auf die in den Communications enthaltenen Lösungen geben. Die entsprechende Zeile aus Figure 10 vereinfacht sich also zu

```
InvolvedModels:
(name: tp,modeltype: TwoPartnerSimplex, solutions: tpe);
```

2.2 Complex Event Processing (CEP)

Wie bereits in Abschnitt 1.1 beschrieben, liefert CEP die Grundlage für den in dieser Arbeit zu untersuchenden Online-Ansatz. Eckert und Bry beschreiben in ihrem Artikel für die GI[3] CEP wie folgt:

> „. . . CEP ist ein Sammelbegriff für Methoden, Techniken und Werkzeuge, um Ereignisse zu verarbeiten während sie passieren, also kontinuierlich und zeitnah. CEP leitet aus Ereignissen höheres, wertvolleres Wissen in Form von so genannten komplexen Ereignissen, das heißt Situationen die sich nur als Kombination mehrerer Ereignisse erkennen lassen, ab.“[4]

Dabei lässt sich CEP in zwei Kategorien unterteilen[4]:

- Bisher unbekannte Muster in Ereignisströmen als komplexe Ereignisse erkennen mittels maschinellem Lernen und Datamining (Pattern Discovery).
- Feststellen komplexer Ereignisse als a-priori spezifizierte Muster mittels Ereignisanfragesprachen (Pattern Matching).

Die zweite Kategorie entspricht genau dem Ziel, das mit ModoCom verfolgt wird. Eigenschaften für Events werden mit einem ModoCom-Programm beschrieben, welches dann in eine Anfrage der Ereignisanfragesprache überführt wird. Für diese Arbeit wurde die Ereignisanfragesprache (*Event Processing Language, EPL*) von Esper ausgewählt. Der Abschnitt 2.3 beschreibt Esper und seine EPL im Einzelnen.

[3] Gesellschaft für Informatik e.V.

2.2.1 Ereignisanfragen

Das Problem beim Online-Ansatz ist, dass man bei CEP konzeptionell auf einem Strom unendlich vieler Ereignisse arbeitet, wohingegen Abfragen auf Datenbanken immer gegen eine endliche Menge gestellt werden. Da nicht alle Anfragen gegen eine unendliche Menge möglich sind[4], ergeben sich nach [4] folgende Anforderungen an eine Ereignisanfragesprache.

Extraktion von Daten Ereignisse transportieren Daten, die Grundlage für Entscheidungen sind oder zu neuen Ereignisse aggregiert werden. Diese Daten müssen für die Anfragesprache verfügbar sein.

Komposition Mehrere Ereignisse müssen kombiniert werden können, so dass ihr zeitlich verteiltes Auftreten zu einem neuen Ereignis zusammengefasst werden kann.

Zeitliche Zusammenhänge Die zeitliche Reihenfolge von Ereignissen und vor allem auch das Auftreten eines Ereignisses in einem bestimmen Zeitfenster, müssen mit der Ereignisanfragesprache darstellbar sein. Neben der zeitlichen Anordnung ist unter anderem auch eine kausale Verknüpfung wichtig.

Akkumulation Das Ausbleiben eines Ereignisses auf einem unendlichen Strom kann nicht modelliert werden. Daher muss eine Ereignisanfragesprache einen endlichen Ausschnitt des Stromes (Fenster,*view*) bereitstellen, auf dem Operationen wie die Aggregation von Ereignisdaten wohldefiniert sind.

2.2.2 Regeltypen und Anfragesprache

Es können zwei Regeltypen unterschieden werden: deduktive und reaktive Regeln. Deduktive Regeln definieren neue (komplexe) Ereignisse auf Basis von Ereignisanfragen und haben keine Seiteneffekte. Reaktive Regeln definieren, ob und wie auf (komplexe) Ereignisse reagiert werden kann und soll. Darüber hinaus kann man drei Arten von Anfragesprachen unterscheiden[4]:

Kompositionsoperatoren Sie entstammen den aktiven Datenbanken und verwenden Operatoren wie Konjunktion, Sequenz, Negation in der Sequenz, welche sich zu komplexeren Anfragen schachteln lassen. Mit Selektion lassen sich nur

[4] So ist beispielsweise die Durchschnittsbildung nur auf einer endlichen Menge möglich.

bestimmte Ereignisse (beispielsweise das erste) zur Aggregation eines komplexen Ereignisses auswählen. Um die Wiederverwendung einfacher Ereignisse in komplexen Ereignissen zu verhindern, können sie konsumiert werden.

Datenstrom-Anfragesprachen Zu diesen gehört auch die Anfragesprache von Esper. Sie basieren auf SQL und haben folgendes Grundprinzip: Datenströme, die Ereignisse als Tupel enthalten, werden in Relationen umgewandelt, auf die dann reguläre SQL-Abfragen angewendet werden. Dies geschieht konzeptionell zu jedem Zeitpunkt einer diskreten Zeitachse. Dabei gibt es verschiedene Fensteroperationen (siehe Abs. 2.3.3).

Produktionsregeln Produktionsregeln arbeiten eng mit der Wirtssprache zusammen und bieten im Gegensatz zu den anderen beiden Sprachen kein so hohes Abstraktionsniveau, da sie zustands- und nicht ereignisorientiert sind.

2.2.3 Pattern in CEP

Design Patterns (Entwurfsmuster) stellen bewährte und wohl dokumentierte Lösungen für wiederkehrende Entwurfsprobleme dar. Sie sind als eine Art *Best Practice* zu verstehen und sollen dem Entwickler helfen, das Rad nicht neu erfinden zu müssen.
Ein guter Überblick über die Möglichkeiten von Complex Event Processing ist in [3] zu finden. Dort werden zehn Design Pattern beschrieben. Diese können kombiniert werden, um eine bestimmte Aufgabe zu erfüllen. Durch eine Kombination der Pattern lassen sich unter anderen die CommunicationOperators realisieren.
Nachfolgend werden die zehn Pattern kurz beschrieben, damit der Leser einen Eindruck über die Möglichkeiten von CEP erhält. Sie dienen später als Anhaltspunkt für die Konzeption einer Umsetzung von ModoCom mit dem CEP-System Esper.

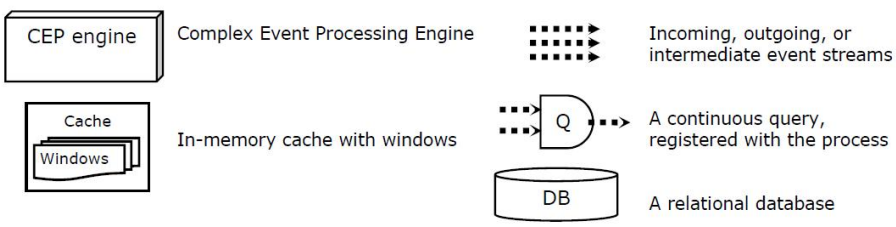

Abbildung 3: Legende zur Darstellung der Pattern.[3]

Filtern Die Anfrage wird auf einem Stream platziert und evaluiert zu wahr oder falsch bzgl. der Event-Attribute. Evaluiert das Query zu wahr, werden die Events in einen vorher definierten Strom weitergeleitet. (siehe Abbildung 4)

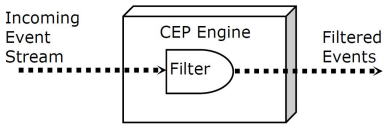

Abbildung 4: Filtern [3]

In Memory Caching Events werden in so genannten Windows festgehalten. Aber auch Ergebnisse von Datenbankanfragen können im Cache zwischengespeichert werden. Hinzu kommt, dass hier zwar von *in memory* gesprochen wird, es aber oft und vor allem bei langlebigen Windows (Stunden oder Tage) wichtig ist, die dort gehaltenen Daten zu persistieren um auch gegenüber einem Systemabsturz robust zu sein. Caching/Windows werden für Joins und Aggregations gebraucht (siehe Abbildung 5).

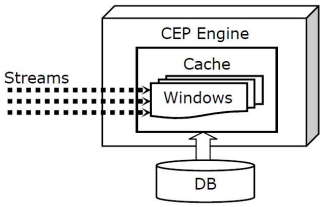

Abbildung 5: In Memory Caching [3]

Aggregation over Windows Die Daten werden zwischengespeichert um darauf Berechnungen wie beispielsweise Min, Max oder Average auszuführen (siehe Abbildung 6).

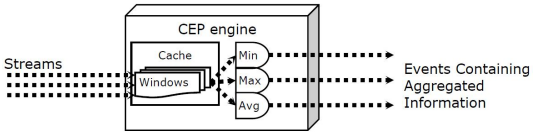

Abbildung 6: Aggregation over Windows [3]

Database Lookups

Oft ist es notwendig, Events um so genannten historische Daten zu erweitern. Dafür sind Datenbankabfragen notwendig, welche beispielsweise einen Key aus einem eintreffenden Event erhalten, mit diesem eine Anfrage an die Datenbank stellen und das Ergebnis der Anfrage dem Event hinzufügen (siehe Abbildung 7).

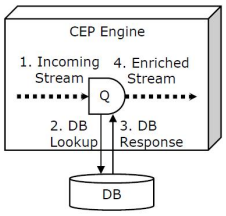

Abbildung 7: Database Lookups [3]

Database Writes

Neben dem Abfragen historischer Daten können diese natürlich auch mit Datenbank-Writes angelegt werden. Dies kann nach einer Verarbeitung in der Engine passieren oder die Daten können direkt vom Stream in die Datenbank persistiert werden (siehe Abbildung 8). Dabei kann nur der Inhalt eines Streams in die Datenbank geschrieben werden.

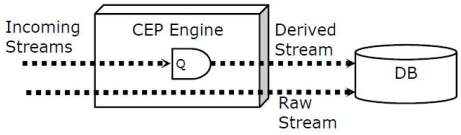

Abbildung 8: Database Writes [3]

Correlation (Join)

Um Informationen aus mehreren Streams zu bekommen, müssen diese betrachtet und zusammengeführt werden. Dafür muss auf mindestens einem Stream ein Window platziert sein, dass Events festhält und diese dann mit eintreffenden Events des anderen Streams vergleicht (siehe Abbildung 9).

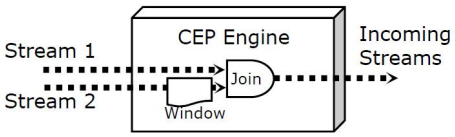

Abbildung 9: Correlation(Join) [3]

Event Pattern Matching Event Pattern Matching ist eine mächtige und einfache Möglichkeit um zeitliche Zusammenhänge zwischen Events mittels einem *follow by*-Operator (siehe Abbildung 10) zu beschreiben (vgl. Abs. 2.3.3).

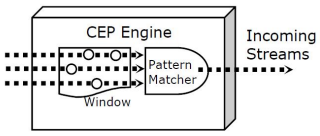

Abbildung 10: Event Pattern Matching [3]

State Machines Es ist möglich, Zustandsautomaten zu beschreiben, deren Transitionen durch Events getriggert werden. Die Definition des Automaten erfolgt allerdings nicht mit einer EPL, sondern getrennt in der Engine oder in einer Datenbank (siehe Abbildung 11).

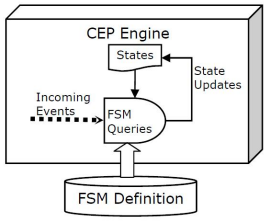

Abbildung 11: State Machines [3]

Hierachical Events Events können aus mehreren Einzelteilen bestehen und formen zusammen einen komplexen (hierarchischen) Event (siehe Abbildung 12). Dabei wird zwischen *structured* und *semi-structured* Events unterschieden [11]. Die Events werden als Records modelliert, die aus Key-Value-Paaren bestehen. Bei den structured Events ist ein Attributname (ein Key) eindeutig (unique). Bei den semi-structured Events kann ein Attributname mehrfach auftauchen.

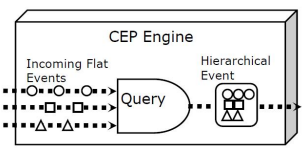

Abbildung 12: Hierachical Events [3]

Dynamic Queries Oft benötigt die Anwendung das Registrieren von Abfragen, das Starten oder Stoppen solcher Abfragen und ähnliches, ohne dabei die Engine neu starten zu müssen. Diese Aufgaben können entweder programmatisch oder vom Anwender gesteuert werden (siehe Abbildung 13).

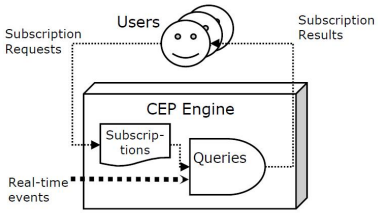

Abbildung 13: Dynamic Queries [3]

2.3 Esper

Esper ist ein OpenSource Projekt der Firma EsperTech und hat eine eigene Event Processing Language (EPL), die an SQL angelehnt ist. Die EPL ist eine Datenstrom-Anfragesprache, wie sie im vorherigen Abschnitt erwähnt wurde. Das Herz von Esper ist die Esper-Engine, welche man sich wie eine umgedrehte Datenbank vorstellen kann. In einer Datenbank werden Daten gespeichert, um mit Anfragen darauf zu operieren. Bei Esper werden die Anfragen in der Engine gespeichert und die Daten zur Auswertung durch die Engine geschoben [9].

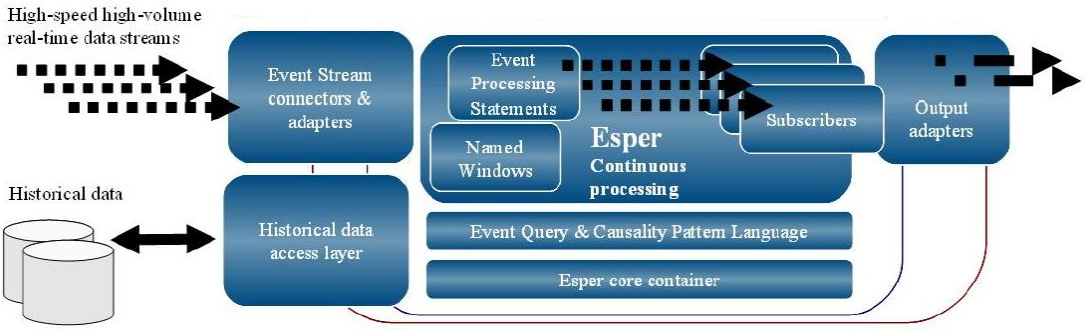

Abbildung 14: Esper Architektur [6]

2.3.1 Event Processing Language

Wie bereits in Abschnitt 2.2.2 erwähnt, ist Espers Event Processing Language eine Datenstrom-Anfragesprache und basiert auf SQL. Elemente wie *select*, *from*, *where* oder *group by* existieren auch in Espers EPL und können dort ähnlich oder genau wie in SQL verwendet werden.
In den nachfolgenden Abschnitten sollen wichtige Unterschiede und neue Konzepte der EPL besprochen werden. Das sind unter anderen

- Views
- Insert- und RemoveStreams
- Filter und Where-Klauseln
- Match-Recognize und
- Pattern Matching

Für ausführliche Informationen über die EPL von Esper sei auf [9] verwiesen.

2.3.2 Events

Esper bietet eine Reihe von Adaptern [10] (siehe Abbildung 14), die es erlauben, Events mittels verschiedener Protokolle zu empfangen. Darunter auch CSV und JMS.
Intern bietet Esper die Möglichkeit die Events als POJO[5], JavaBean, Map oder XML-Dokument zu repräsentieren [9]. Mittels diverser Output-Adapter [10] verlassen die Events die Engine.
Dadurch lässt sich Esper leicht auch in bereits bestehende Strukturen integrieren.

2.3.3 Konzepte in Esper

EPL und UpdateListener/Subscriber Mit der EPL werden Anfragen an den Datenstrom gestellt, Ereignisse selektiert, zu neuen Ereignissen aggregiert und ggf. in einen neuen Strom gesendet. Jede dieser Anfragen wird in der Esper-Engine registriert. Zusätzlich kann auf jeder Anfrage maximal ein *Subscriber* und beliebig viele *UpdateListener* registriert werden. Diese werden dann, entsprechend der Abfrage, über die Ergebnisse der Abfrage informiert. Die Unterschiede sowie Vor- und Nachteile zwischen UpdateListener und Subscribern sind in [9] im Detail ausgeführt.

[5] Plain Old Java Object

View Nicht alle Anfragen sind für eine konzeptionell unendliche Menge von Daten, als die Streams anzusehen sind, definiert. Esper bietet daher das Konzept der Views an, welches bereits in Abs. 2.2.3 (In Memory Caching) vorgestellt wurde, mit dem Events nach bestimmten Kriterien festgehalten werden können, damit Anfragen gegen die Menge von Events in der View gestellt werden können.

Die in Esper eingebauten Views sind in vier *namespaces* organisiert.

- **win** bietet Fenster an, die entweder zeitlich oder durch eine Anzahl enthaltener Events beschränkt werden können. Dabei kann ein solches Fenster kontinuierlich (also jede Sekunde oder immer wenn ein neuer Event hereinkommt), oder aber stapelweise (*batched*) verschoben werden. Dabei können Zeit und Länge auch kombiniert werden. Außerdem gibt es die Möglichkeit ein Zeitfenster nicht über die interne Uhr der Engine, sondern über externe Zeitstempel aufzubauen. Dabei wird der Zeitstempel jedes Events ausgewertet.
- **ext** Views aus diesem Namensraum sortieren Events nach bestimmten Kriterien (*timestamp* oder andere Event-Properties).
- **stat** Diese Views bieten statistische Informationen über die Daten wie beispielsweise Regression, Korrelation oder gewichteter Durchschnitt an.
- **std** Bietet “Standard”-Sichten wie *groupby, size* und *firstevent* oder *lastevent* an.

Insert und Remove Stream Jedes Mal, wenn die Esper-Engine ein Ereignis verarbeitet, werden alle Update Listener, die beispielsweise an der Abfrage

*select * from Withdrawal*

hängen, benachrichtigt und das Objekt (Withdrawal-Event) an diese weitergegeben. Der Begriff *Insert Stream* spielt darauf an, dass die Ereignisse dem Listener als eingehende Events überreicht werden. Wird mit Views gearbeitet, beschreibt der *Insert Stream* die Events, die in die View hineinkommen. Gefilterte Events werden nicht in den *Insert Stream* übernommen. Ein *Remove Stream* existiert nur bei der Verwendung von Views und beinhaltet die Events, die eine View verlassen. Die folgende Anfrage definiert beispielsweise eine Fenster der Länge 5.

*select * from Withdrawal.win:length[5]*

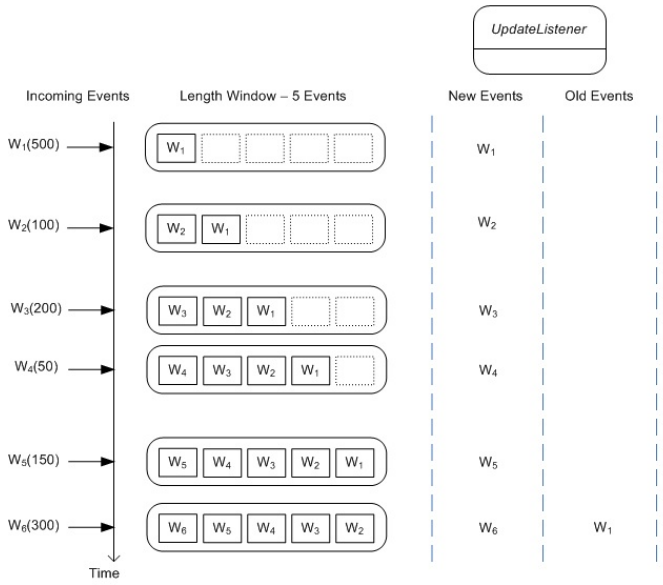

Abbildung 15: Beispielausgabe für ein Längen-Fenster.[8]

Der Inhalt des *Insert Stream* und *Remove Stream* sind in Abbildung 15 verdeutlicht.

Wird in der select-Klausel kein Stream angegeben, wird standardmäßig der Insert Stream (*istream*) verwendet, mit der Konsequenz, dass den Update-Listenern nur die Events des Insert Streams mitgeteilt werden. Mit den Schlüsselwörtern *rstream* und *irstream* lassen sich entweder nur der Remove Stream oder beide Streams auswählen.

Filter und Where-Klauseln Mit Hilfe von Filtern können Ereignisse herausgefiltert werden, so dass sie nicht in die View (hier ein Längen-Fenster) übernommen werden. Eine Anfrage mit einem Filter sieht wie folgt aus:

*select * from Withdrawal(amount >= 200).win:length(5)*

Die entsprechende Ausgabe ist in Abbildung 16 zu finden. Wird eine entsprechende Einschränkung in der Where-Klausel formuliert,

*select * from Withdrawal.win:length(5) where amount >= 200*

werden alle eingehenden Events in das Fenster übernommen, aber nur die Events, welche die Einschränkung erfüllen, werden in den Insert-Stream übergeben. Welche Events in die View beziehungsweise den Insert-Stream übergeben werden, ist in Abbildung 17 auf Seite 32 dargestellt.

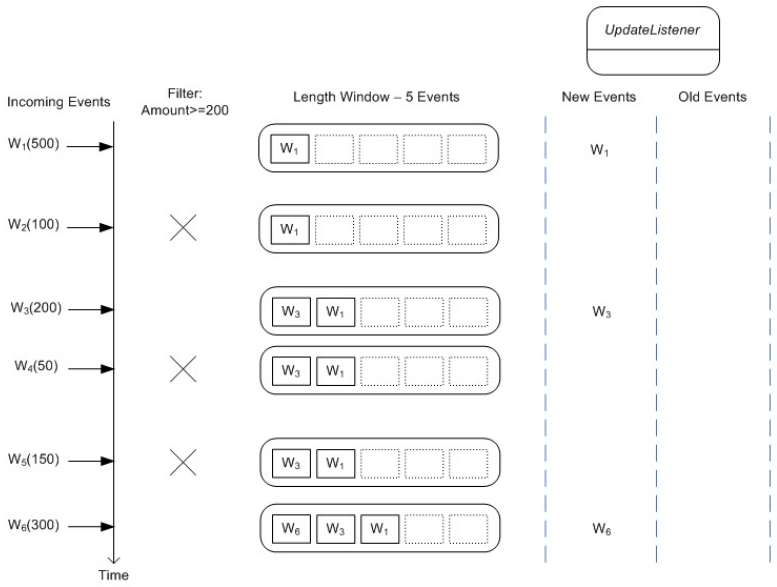

Abbildung 16: Beispielausgabe für ein Längen-Fenster mit Event Stream Filter.[8]

Pattern Matching Pattern Matching erlaubt das Beschreiben von Beziehungen zwischen Events zeitlich geordneten Events. Das folgende Beispiel berechnet den Gesamtpreis, wenn innerhalb einer Minute eine ServiceOrder von einer ProductOrder gefolgt wird in einem Gesamtzeitraum von 2 Stunden.

Listing 6: Beispiel eines Pattern Matchings [9]

```
1 select a.custId, sum(a.price + b.price)
2 from pattern [every a=ServiceOrder ->
3                 b=ProductOrder(custId = a.custId)
4                 where timer:within(1 min)
5               ].win:time(2 hour)
6 where a.name in ('Repair', b.name)
7 group by a.custId
8 having sum(a.price + b.price) > 100
```

Die meisten der Pattern Matching Ausdrücke können auch mit SQL ausgedrückt werden, diese sind aber meist umständlich und komplex. Dabei ist Pattern Matching nicht vergleichbar mit Pattern Discovery aus dem Data-Mining Bereich. Pattern Discovery, also das Finden eines Patterns in Daten ist ein schwieriges Problem und wird in der Regel auf Offline-Daten gelöst. Pattern Matching hat die Aufgabe das Auftreten eines bereits erkannten Patterns festzustellen. Als CEP-Engine unterstützt Esper natürlich auch Pattern Matching. Allerdings unterstütz ModoCom nicht das Beschreiben von Patterns.

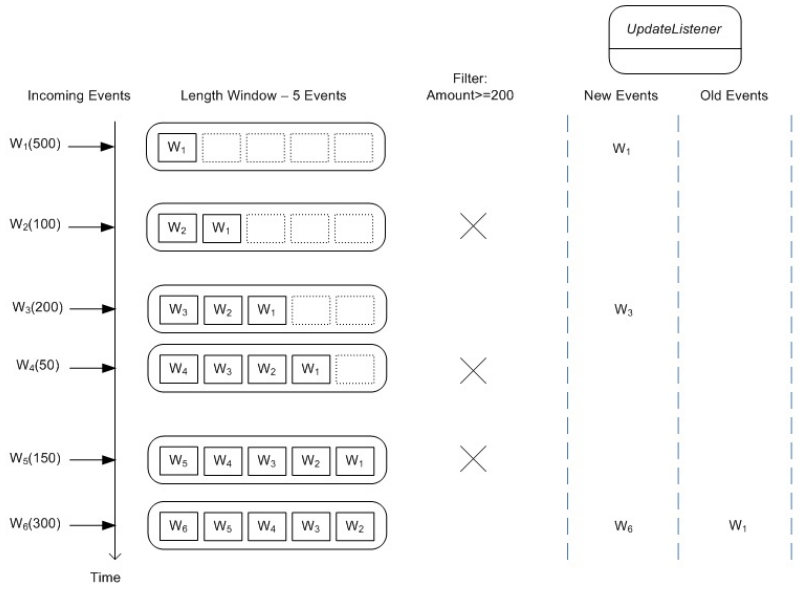

Abbildung 17: Beispielausgabe für ein Längen-Fenster mit Where-Klausel.[8]

Match Recognize Mit Match Recognize bietet Esper eine alternative Möglichkeit um Patterns mit Hilfe regulärer Ausdrücke zu beschreiben. Match-Recognize wurde als Erweiterung des SQL-Standards vorgeschlagen [6][1]. Im Gegensatz zu einer Pattern-Definition kann ein *match recognize* nur direkt nach *from* stehen.

2.3.4 Zusammenfassung

CEP unterstützt mit seinen Streams und Views sehr gut die angestrebte Online-Analyse und bietet mit Pattern Matching sogar Konzepte, die über die aktuellen Möglichkeiten von ModoCom hinausgehen. Durch die SQL-ähnliche EPL und die Verwendung von Java innerhalb des Esper-Engine ist der Zugang zu Esper relativ einfach.

Neben den hier beschriebenen Möglichkeiten bietet Esper noch weitere Features wie beispielsweise Annotations, die für diese Arbeit aber nicht relevant sind.

2.4 Fazit

Das von der Fa. Plath GmbH ModoCom zerlegt die Beschreibung einer Kommunikation in verschiedene Bestandteile, deren Eigenschaften der Anwender mit Prädikaten(Predicates) beschreiben kann. Dabei werden die Elemente einer Menge

kombiniert und für jede Kombination wird überprüft, ob sie gewisse Eigenschaften besitzt. Ist dies der Fall, werden die Elemente in ein andere (Unter-)Menge verschoben, wo sie erneut kombiniert und ausgewertet werden können.
Dadurch, dass der Anwender die Möglichkeit hat, die Ergebnisse eines Modells durch weitere Prädikate und Operatoren weiter einzuschränken, lassen sich komplexe Kommunikationsmodelle beschreiben.

Mit CEP kann der Anwender, ähnlich wie auch in Datenbanken, Daten aus einer Menge selektieren, indem er mit Queries/Statements die Eigenschaften der Daten beschreibt, die er haben möchte.

Der Unterschied zwischen CEP und einer Datenbank ist, dass mit CEP eine Menge oft nicht vollständig mit einem Query bearbeitet werden kann, da Elemente der Menge entweder noch nicht in der CEP-Engine eingetroffen sind oder sie bereits wieder verlassen haben, beispielsweise weil der Hauptspeicher, in dem die Elemente vorgehalten werden, nicht mehr ausreicht.
Die Frage ist daher, ob es trotz dieser Einschränkung möglich ist, die Möglichkeiten von ModoCom vollständig mit CEP zu realisieren oder ob einige Konzepte von ModoCom nicht abgebildet werden können.
Damit überhaupt klar ist, welche Konzepte in ModoCom und CEP existieren, werden diese im nächsten Kapitel untersucht.

3 Konzeption einer Abbildung

In diesem Kapitel werden Konzepte und Sprachteile in der von der Fa. Plath GmbH entwickelten Kommunikationsmodellierungsprache ModoCom identifiziert, die in den Kontext einer CEP-Applikation abgebildet werden müssen. Ziel dieses Kapitels ist es, eine grobe Struktur für ein CEP-Framework zur Umsetzung von ModoCom zu beschreiben.

3.1 Online- vs. Offline Analyse

In diesem Abschnitt werden die Unterschiede der beiden Analyse-Verfahren untersucht und es wird auf Einzelheiten und Probleme der Online-Analyse eingegangen.

3.1.1 Charakteristik der Offline-Analyse

Die bei der Fa. Plath GmbH entwickelte Offline-Analyse zeichnet sich dadurch aus, dass die von der Sensorik bereitgestellten Events “aufgezeichnet”, also für eine bestimmte, endliche Zeit in eine Datenbank geschrieben werden (Abbildung 18). Erst wenn das Schreiben in die Datenbank abgeschlossen ist, werden die Events

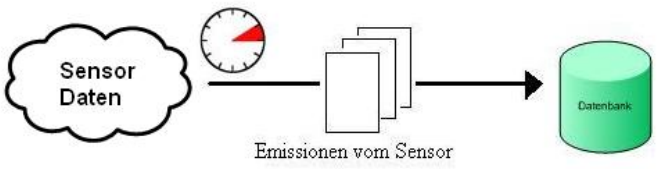

Abbildung 18: Aufzeichnen von Emissionen in eine Datenbank.

wieder ausgelesen und die ModoCom-Modelle evaluiert (Abbildung 19). Durch die

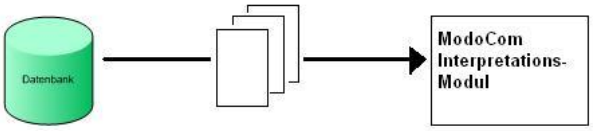

Abbildung 19: Auslesen der Emissionen aus der Datenbank für die Evaluation.

Aufzeichnung sind Anfang und Ende festgelegt und folglich sind EventGroups und Communications als endliche Mengen zu betrachten.

3.1.2 Charakteristik der Online-Analyse

Bei der Online-Analyse werden im Gegensatz zu Offline-Analyse keine Daten für eine Evaluation gespeichert. Die Events werden direkt vom Sensor in die CEP-Engine weitergeleitet und ausgewertet. Für die Analyse ist kein festes Ende definiert, was zur Folge hat, dass EventGroups und Communications als unendliche Mengen von Events betrachtet werden müssen. Das Problem unendlicher Men-

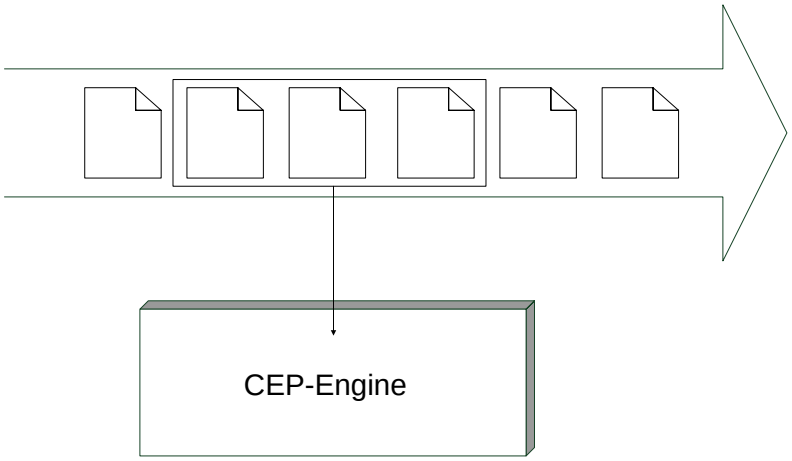

Abbildung 20: Auswerten eines Streams.

gen ist, dass man sie nicht sortieren kann. Durch das Eintreffen der Events in der Reihenfolge ihres Auftretens unterliegen die Events implizit einer zeitlichen Sortierung. Das Sortieren nach anderen Attributen als der Zeit ist aber nur möglich, wenn der Stream von eintreffenden Events zeitlich oder mengenmäßig begrenzt wird. Für dieses Problem bietet CEP die Möglichkeit von Fenstern (vgl. Abschnitt 2.2.3 und Abschnitt 2.3.3), die sowohl zeitlich als auch über die Anzahl der im Fenster befindlichen Events definiert werden können. Die Fenster können zusätzlich entweder gleitend (sliding) oder fallend (tumbling)[8] sein. Ein gleitendes Fenster wird kontinuierlich in der Zeit verschoben, wohingegen ein fallendes Fenster immer nur dann verschoben wird, wenn das Zeitfenster abgelaufen ist oder eine bestimmte Anzahl Events sich im Fenster befinden.
Abbildung 20 zeigt den Zugriff auf einen Stream mittels eines Fenster mit einer Größe von 3 Events. Die Definition des Fensters ist also entscheidend für die Auswertung der Events, da ein zu kleines oder zu kurzes Fenster möglicherweise dazu führt, dass einige Events nicht mit anderen (entscheidenden) Events verglichen

werden können.

Da bei der Online-Analyse keine Zwischenspeicherung der Eingabedaten erfolgt, muss die Online-Analyse ohne die, durch die Zwischenspeicherung gewonnenen Metadaten auskommen. Das heißt, dass zu Beginn der Analyse nicht klar ist, wie viele EventGroups auszuwerten sind. Bei der Offline-Analyse ist die Anzahl der EventGroups durch das im Vorfeld stattfindende Clustering bekannt (siehe Abbildung 21) und das Interpretationsmodul lässt sich vor der Evaluation entsprechend einstellen. Bei der Online-Analyse hingegen, sind die Daten im Vorfeld unbekannt

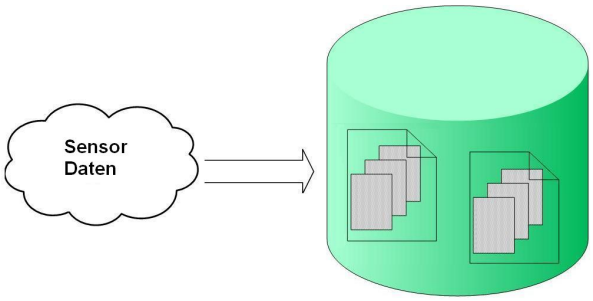

Abbildung 21: Offline-Analyse, Gewinnung von Meta-Daten.

und das CEP-Framework muss dynamisch auf die Daten reagieren können (Abbildung 22). Das heißt, es müssen während der Analyse neue Abfragen in der

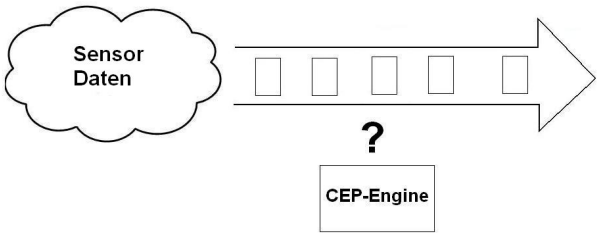

Abbildung 22: Online-Analyse, Arbeiten ohne Meta-Daten.

CEP-Engine platziert werden, um die neue EventGroups auswerten zu können, während die bereits existierenden EventGroups weiterhin evaluiert werden müssen.

3.1.3 Evaluation von CommunicationModels

Bei der Offline-Analyse, bei der die Daten in einer Datenbank gespeichert sind, wird ein Modell vollständig evaluiert und die Ergebnisse werden wieder in die Da-

tenbank geschrieben, bevor die Ergebnisse aus der Evaluation des Modells in weiteren Modellen analysiert werden.
Die Online-Analyse hat das Ziel, die zeitliche Differenz zwischen dem Auftreten der Events und dem Erkennen einer Kommunikation möglichst klein zu halten.
Für die Online-Analyse ergeben sich zwei Vorgehensweisen, die unterschiedliche Auswertegeschwindigkeiten mit sich bringen, aber auch andere Anforderungen bezüglich Speicher und Komplexität haben.

Finished Communications Die Auswertung innerhalb der Offline-Analyse erfolgt schrittweise, wobei jeder Schritt vollständig abgeschlossen ist, bevor ein neuer Schritt beginnt. Konkret heißt das, dass zuerst alle Daten aufgezeichnet werden (vgl. Abbildung 18 S. 34) und anschließend EventGroups mittels einem Clustering-Verfahren ermittelt werden. Anschließend steht die Anzahl gefundener EventGroups fest, wie sie in Abbildung 21 in der Datenbank angedeutet sind. Im nächsten Schritt werden alle diese EventGroups ausgewertet und, entsprechend dem CommunicationModel, nach möglichen Kommunikationen gesucht. Am Ende dieses Schritts steht eine feste Anzahl von Communications zur weiteren Verarbeitung zur Verfügung.

Durch den zeitlichen Versatz, welcher durch das Aufzeichnen entsteht und bedingt durch die Endlichkeit der Aufzeichnung, ist jede gefundene Communication zum Zeitpunkt des Erkennens bereits beendet. In jedem Fall ist eine Kommunikation entweder durch das Modell selbst beendet worden, da das Modell der Kommunikation keine neuen Events zugeordnet hat oder es gibt keine neuen Events, da die Aufzeichnung beendet wurde.
Die Arbeit führt für diese Kategorie von Communications den Begriff *finished Communications* ein.
In der Offline-Analyse ist ein Modell also vollständig evaluiert, wenn alle Events evaluiert wurden.

In der Online-Analyse kann nicht festgestellt werden, ob ein Modell vollständig evaluiert wurde, da immer neue Events im Modell eintreffen können. Jedoch kann für eine Communication festgestellt werden, dass sie abgeschlossen (finished) ist, wenn auf Grund der Predicates kein neuer Event mehr der Communication hinzugefügt werden kann.

Finished Communications im Kontext der Online-Analyse bedeutet, dass ein CommunicationModel eine Communication erkennt und alle zugehörigen Events und Meta-Informationen zwischenspeichert, bis diese Communication abgeschlossen ist, das heißt keine neuen Events mehr für diese Communication gefunden wer-

den können. Erst dann wird diese Communication dem Framework zur weiteren Verarbeitung übergeben.

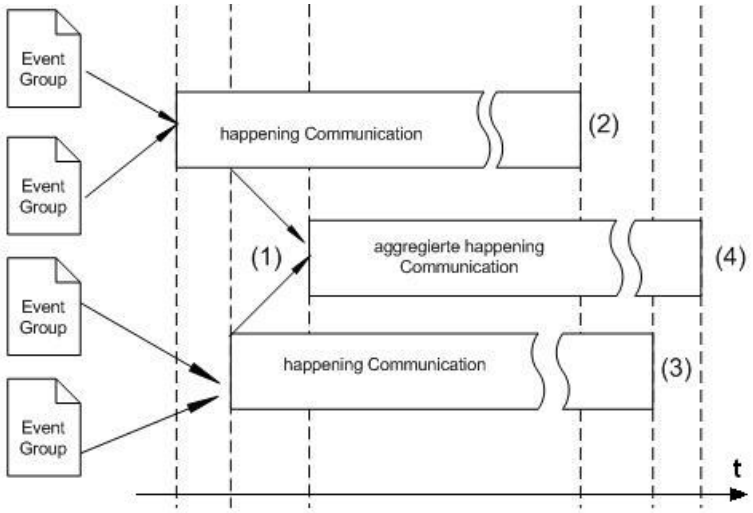

Abbildung 23: Ablauf bei einer Realisierung mit happening Communications.

Happening Communications Bei der für die Fa. Plath GmbH zu entwickelnden Online-Analyse ist es von Interesse, möglichst wenig zeitlichen Versatz zwischen Stattfinden und Erkennen einer Kommunikation zu haben. Um diesen Versatz möglichst klein zu halten, ist es naheliegend, gefundene Communications dem Framework direkt zur weiteren Verarbeitung weiter zu reichen, auch wenn die gefundene Kommunikation noch in vollem Gange ist. Zum Zeitpunkt, da eine Communication weiteren Modellen zur Verfügung gestellt wird, ist also nicht klar ob, wann und wie viele Events noch für diese Communication gefunden werden. Für diese Kategorie Communications wird in der Arbeit der Begriff *happening Communications* verwendet. Werden dann zwei (oder mehr) *happening Communications* kombiniert und ausgewertet und entsteht daraus wieder eine *happening Communication*, kann diese ebenfalls direkt nach dem Erkennen dem Framework zur Verfügung gestellt werden. Daraus kann sich eine beliebig lange Kette von CommunicationModels entwickeln, die Auswertungen auf Grundlage einer noch nicht abgeschlossenen Kommunikation betreiben.

Nun kann es passieren, dass zwei happening Communications ausgewertet werden, um festzustellen, ob sich diese beiden Communications zu einer komplexeren Communication aggregieren lassen (siehe Abbildung 23 (1)). Während dieser Evaluation kann es passieren, dass eine Communication abgeschlossen wird (Abbildung 23 (2)). Implementiert das Framework *happening Communications*, muss

die Auswertung entsprechend darauf reagieren, da nur noch eine Communication neue Events generiert.
Das Framework muss auch reagieren, wenn eine der Kommunikationen abbricht, beziehungsweise wenn es feststellt, dass die komplexe Communication nicht mehr erfüllt werden kann und die Auswertung abgebrochen werden muss. Das ist genau dann der Fall, wenn eine Communication zunächst die vorgegebenen Predicates erfüllt, im Laufe der Auswertung aber die Predicates irgendwann nicht mehr erfüllt sind. Ein Beispiel für ein solches Predicate könnte sein

```
... wherenumberOfEvents < 15
```

Bei diesem Beispiel werden zunächst weniger als 15 Events zur Communication hinzugefügt. Im Laufe der Auswertung aber wird diese Grenze überschritten.

Das heißt, eine Communication kann sich nicht nur im Zustand *happening* oder *finished* befinden, sondern auch in den Zuständen *fulfilled* oder *canceled*. Eine Communication ist dann *canceled*, wenn sie zuvor *fulfilled* war, das Predicate aber nun nicht länger erfüllt werden kann. Da, gemäß dem CommunicationModel eine Kommunikation nur dann

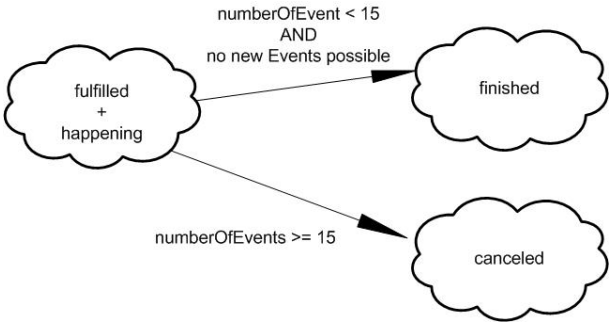

Abbildung 24: Mögliche Zustände einer Communication in der Online-Analyse.

stattfindet, wenn diese das Predicate erfüllt, fallen die Zustände *happening* und *fulfilled* zu einem gemeinsamen Zustand zusammen (Abbildung 24). Entweder erfüllt sie weiterhin das Predicate und wird abgeschlossen (*finished*) oder das Prediacte kann nicht mehr erfüllt werden, und die Communication muss abgebrochen werden (*canceled*).

3.1.4 Zusammenfassung

Der Hauptunterschied zwischen der Offline- und der Online-Analyse ist die Fenstergröße. Der Zeitraum, bei dem in der Offline-Analyse aufgezeichnet wird, ist prinzipiell der gleiche, wie das Fenster auf einem Eventstream. Der Unterschied ist, dass der Offline-Zeitraum um ein Vielfaches größer ist als das Fenster der Online-Analyse, da das Fenster der Online-Analyse im Arbeitsspeicher und nicht in einer Datenbank gehalten wird. Im Gegenzug wird das Fenster in der Online-Analyse über die Zeit verschoben, wobei das Fenster der Offline-Analyse fest ist.

Wählt man bei der Online-Analyse eine ungünstige Fenstergröße, läuft man Gefahr, vorhandene Kommunikationen zu zerstückeln, mit der Folge, dass sie nicht mehr erkannt werden können.

Allerdings ist auch die Offline-Analyse kein Garant dafür, alle Kommunikationen erkennen zu können, da auch hier das Aufnahmefenster ungünstig platziert sein kann, so dass Teile einer Kommunikation abgeschnitten und somit nicht mehr erkennbar sind.

Hinsichtlich der Vollständigkeit gibt es keinen Unterschied zwischen den happening Communications und den finished Communications. Diese wirken sich nur auf die Auswertegeschwindigkeit und auf die Komplexität hinsichtlich der Implementierung aus. Allerdings erlaubt die Online-Analyse dem Anwender zu entscheiden, die Analyse zu stoppen oder sie noch weiter laufen zu lassen, wenn beispielsweise eine interessante Kommunikation stattfindet. Bei der Offline-Analyse hat der Anwender keinen Einblick in die Daten und kann daher nicht entscheiden, ob es ggf. besser sein könnte, die Aufnahme noch einige Minuten weiterlaufen zu lassen.

3.2 Das CEP-Framework

ModoCom gibt dem Anwender die Möglichkeit Kommunikationsmodelle zu beschreiben. Diese Modelle benötigen einen Kontext, in dem sie zur Evaluation ausgeführt werden können. Dieser Kontext ist das CEP-Framework. Abbildung 25 zeigt die Bestandteile dieses Frameworks und den Zusammenhang zwischen den Eingabedaten des Sensors, dem ModoCom-Programm und dem CEP-Framework.
Ein durchgängiger Rahmen gibt an, an welcher Stelle etwas definiert wird. Ein durchgehender Pfeil gibt die Richtung an, in die die Definitionen abgebildet werden. Ein gestrichelter Pfeil zeigt zum einen den Weg der Events durch das Framework und zum anderen den Zugriffe auf eine Datenbank.

Zunächst lässt sich erkennen, dass der Sensor die Eingabedaten bestimmt (siehe 3.2.1). Die Sensordaten liegen in der Regel im XML-Format vor und werden innerhalb des Frameworks verwendet (b). Neben diesen Sensordaten (Emissionen die durch den ObjectType repräsentiert werden), gibt es noch weitere Datentypen im CEP-Framework. Diese werden innerhalb des Frameworks definiert (b), da ModoCom zur Beschreibung nicht ausreicht. Zu diesen Datentypen gehören EventGroups und Communications. Alle Datentypen, die in einem ModoCom-Programm verwendet werden sollen, müssen in diesem deklariert werden (c). Neben den Datentypen sind auch CommunicationOperators im CEP-Framework definiert und werden im ModoCom-Programm nur verwendet (d).
Der Anwender beschreibt in einem ModoCom-Programm ein Kommunikationsmodell, in dem er Einschränkungen (Predicates) definiert, mit denen die Operatoren gesteuert werden können. Er definiert die Anordnung der Operatoren und wählt die zu evaluierenden EventGroups. Die im ModoCom-Programm beschriebene Struktur wird dann in das CEP-Framework abgebildet (e).

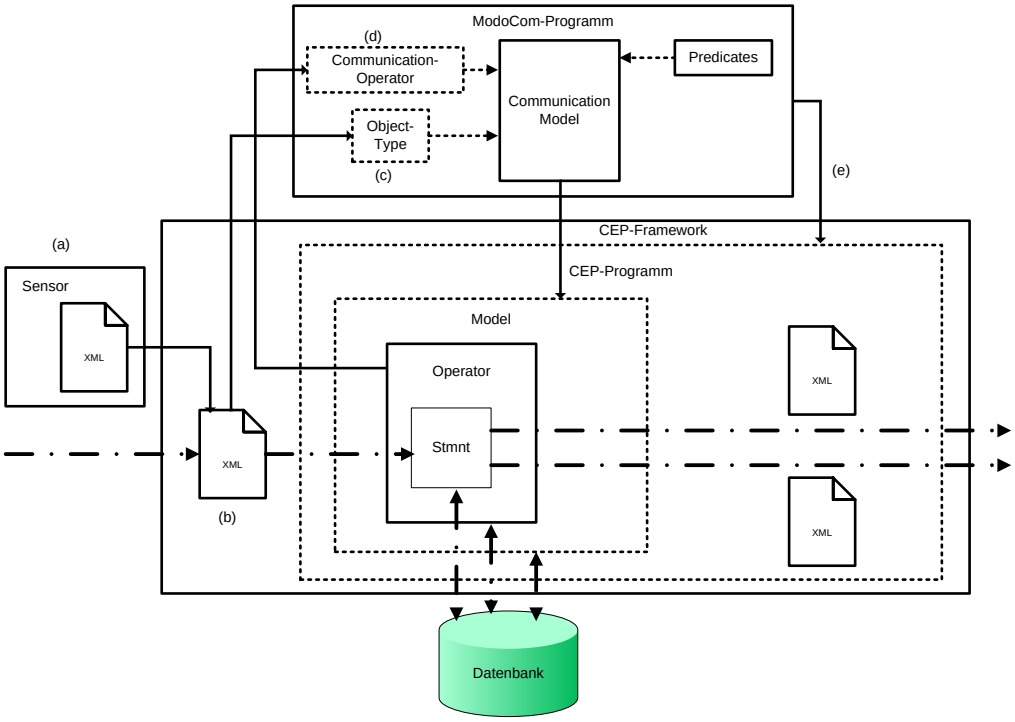

Abbildung 25: CEP-Framework im Kontext.

Während der Online-Analyse gelangen die Sensordaten in das CEP-Framework, welches durch das ModoCom-Programm initialisiert wurde. Dort werden sie ggf. in ein für die CEP-Engine passendes Format umgewandelt und in EventGroups eingeteilt. Die Einteilung kann innerhalb der CEP-Engine geschehen oder in einem anderen Teil des CEP-Frameworks. Die in EventGroups eingeteilten Events erreichen als Stream das erste CommunicationModel, wo sie durch den CommunicationOperator verarbeitet werden. Sowohl die in der Engine platzierten Statements, als auch der CommunicationOperator und das CommunicationModel haben die Möglichkeit Daten kurz- oder langfristig zwischen zu speichern. Diese Tatsache ist in Abbildung 25 vereinfachend durch Datenbank-Zugriffe dargestellt. Es ist aber auch möglich Daten innerhalb des Frameworks selbst vorzuhalten.
Nachdem die Events alle CommunicationModels durchlaufen haben, werden die Ergebnisse dem Anwender übergeben.

Abbildung 26 verdeutlicht noch einmal, welche Komponenten und Konzepte in welche Richtung abzubilden sind. Die drei letzen Pfeile machen deutlich, das auf der Seite von ModoCom einige Erweiterungen notwendig sind, um das CEP-Framework vollständig aus ModoCom heraus steuern zu können.

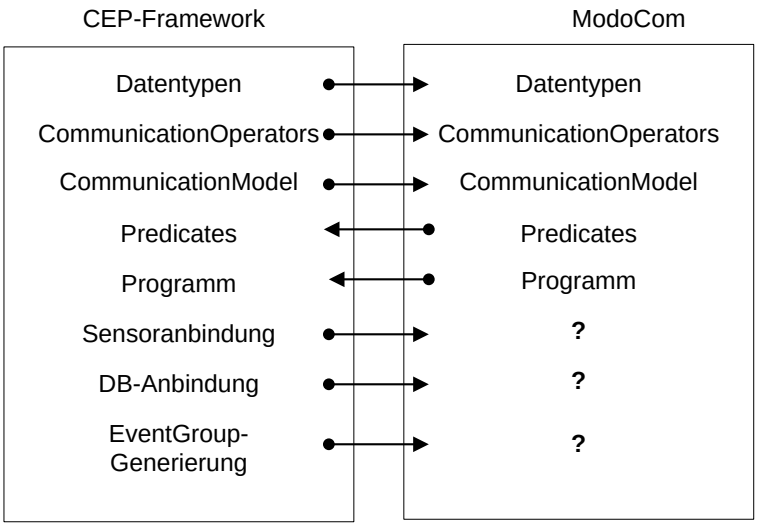

Abbildung 26: Abbildung der Konzepte und Komponenten im Überblick.

3.2.1 Abbildung: CEP-Framework nach ModoCom

Abbildung 26 zeigt, welche Konzepte und Bestandteile woher entstammen und wohin sie für ein funktionierendes Framework abgebildet werden müssen. In diesem Abschnitt werden die Teile beschrieben, die durch das CEP-Framework vorgegeben sind und in das ModoCom-Programm integriert werden müssen, damit das ModoCom-Programm mit den Daten in Framework arbeiten kann. Abschnitt 3.2.2 widmet sich dann den Konzepten, die in ModoCom definiert werden und in das CEP-Framework hinein abgebildet werden müssen.

Datentypen Grundlage einer Evaluation sind die Daten. Diese werden von einem Sensor in das Framework eingespeist. Daher sind die Eigenschaften einer Emission durch den jeweils einspeisenden Sensor vorgegeben.
Typischerweise sind die Daten eines Sensors in XML spezifiziert. Können also, falls notwendig, in ein anderes Format überführt werden, bevor sie in die CEP-Engine gesendet werden.

ModoCom unterscheidet drei Kategorien von ObjectTypes:

- Events
- EventGroups
- Communications

Die Daten der Sensoren werden auf ObjectTypes der Kategorie Events abgebildet. Auf diese Kategorie von ObjectTypes wird nur lesend zugegriffen. Daher ist es wichtig, dass

die Engine auf alle Attribute des Events, welche in ModoCom beschrieben wurden, auch zugreifen kann. Dabei kann der Anwender weniger Attribute in ModoCom beschreiben, als der Event tatsächlich besitzt. Attribute zu beschreiben, die der Sensor-Event aber nicht besitzt, ist nicht möglich.

Auf ObjectTypes der beiden anderen Kategorien wird auch schreibend zugegriffen, da in diesen ObjectTypes die Ergebnisse der Analysen beziehungsweise der EventGroup-Generierung gespeichert werden.
Tendenziell wäre es für diese beiden Kategorien möglich, dass diese, anders als die Events, von ModoCom ins Framework abgebildet werden, doch fehlt es ModoCom an sprachlichen Mitteln, damit eine Abbildung in diese Richtung realisiert werden könnte.
Denn neben dem Beschreiben, welche Attribute der ObjectType besitzt, muss auch beschrieben werden, wie Werte für die einzelnen Attribute erzeugt werden sollen. Findet ein CommunicationOperator Events, die zu einer bestimmten Kommunikation gehören, dann müssen die Attribute der Kommunikation aktualisiert werden.
Der CommunicationOperator soll unabhängig der Daten beschrieben werden, mit denen er arbeitet und daher kein Wissen darüber besitzen, wie die einzelnen Attribute der Events in eine Kommunikation einzupflegen sind. In ModoCom werden nur die Attribute beschrieben, die eine Kommunikation besitzt, damit die Predicates, und damit die Operatoren auf die Attribute zugreifen können. Doch auch in ModoCom kann nicht beschrieben werden, wie die Werte für die Attribute zu bilden sind. Daher bleibt im Moment nur die Möglichkeit übrig, die ObjectTypes der Kategorien EventGroup und Communication auch im Framework, in Form eines abstrakten Datentyps, zu beschreiben. Diesem kann der Operator die gefundenen Events übergeben und der Datentyp weiß dann, wie er an die für ihn wichtigen Informationen kommt.

CommunicationOperators Die CommunicationOperators bilden den Kern der Evaluation. Sie beschreiben, in welcher Reihenfolge die Events mit den definierten Einschränkungen ausgewertet und wie sie weitergereicht werden.
Da sich ModoCom noch nicht im produktiven Einsatz befindet, wurden bei der Fa. Plath GmbH bisher erst drei CommunicationOperators für die Offline-Analyse implementiert. Diese sind:

- FindAlternatingCommunicationsInTwoGroups
- FindCommunicationsInOneGroup
- AggregateCommunicationsOfOneSubmodel

Nachfolgend soll mit Hilfe bekannter CEP-Pattern (vgl. 2.2.3) die Funktion der drei bekannten Operatoren allgemein für eine CEP-Engine umgesetzt werden.

a) FindAlternatingCommunicationsInTwoGroups Dieser erste Operator arbeitet auf EventGroups und ist somit für den Einsatz in einem einfachen CommunicationModel bestimmt. Mit Hilfe von vier Predicates lässt sich das Verhalten dieses Operators steuern. Die Predicates haben dabei folgende Aufgaben:

EquivalencePredicate Das EquivalencePredicate unterteilt eine EventGroup in Äquivalenzklassen. Dabei hat der Anwender sehr große Freiheit bei der Definition von Äquivalenzklassen. Eine mögliche Äquivalenzklasse kann beispielsweise Events mit gleicher Frequenz und gleicher Modulation enthalten. Dabei kann der Anwender definieren, was im Falle der Modulation als gleich anzusehen ist. So kann der Anwender in jeder Klasse auch die Events mit einbeziehen, deren Modulation unbekannt ist.
Es ist aber auch möglich, Äquivalenzklassen zu definieren, deren Äquivalenz sich über das Verhältnis der Werte eines Attributes zueinander bestimmt. So kann beispielsweise die Differenz von Endzeit zu Startzeit von zwei Events als Kriterium definiert werden.

RepresentativePredicate Das *RepresentativePredicate* wählt einen Event aus der Äquivalenzklasse aus, der diese Klasse vertreten kann. Es kann auch leer gelassen werden, so dass jeder Event als Repräsentant fungieren kann. Konkret wird immer der erste auftretende Event genommen.

InGroupRestrictionPredicate Mit den hier definierten Einschränkungen werden Events aus einer Äquivalenzklasse einer EventGroup getestet. Ein solches Predicate könnte, nachdem die Events in Äquivalenzklassen mit gleicher Frequenz und Modulation eingeteilt wurden die Events herausfiltern, die zeitlich einen zu großen Abstand zu einander haben.

BetweenGroupRestrictionPredicate Mit diesem Predicate werden Einschränkungen für Events definiert, die in der gleichen Äquivalenzklasse aber in verschiedenen EventGroups sind. Hier werden die Events der beiden Klassen zusammengebracht und und mit dem *BetweenGroupRestrictionPredicate* weiter aussortiert. Die Ergebnisse aus diesem Predicate münden in einer Kommunikation.

Der Ablauf im Operator stellt sich wie folgt dar: Zunächst werden die EventGroups jeweils für sich mit Hilfe des *EquivalencePredicate* in Äquivalenzklassen eingeteilt. Diese Äquivalenzklassen werden dann jeweils mit dem *InGroupRestrictionPredicate* weiter verfeinert und anschließend wird, mit dem *RepresentativePredicate*, ein Repräsentant für die Äquivalenzklasse bestimmt. Nun werden gleiche Äquivalenzklassen der beiden EventGroups zusammengebracht, um mit dem *BetweenGroupRestrictionPredicate* ausgewertet zu werden. Die Events, die jetzt noch übrig geblieben sind, bilden eine Kommunikation.

Abbildung 27 zeigt, wie dieser Ablauf mit CEP und den in Abschnitt 2.2.3 besprochenen Pattern realisiert werden kann. Da die Daten und auch das *EquivalencePredicate* vorher unbekannt sind, weiß der Operator nicht, in wie viele Äquivalenzklassen eine EventGroup

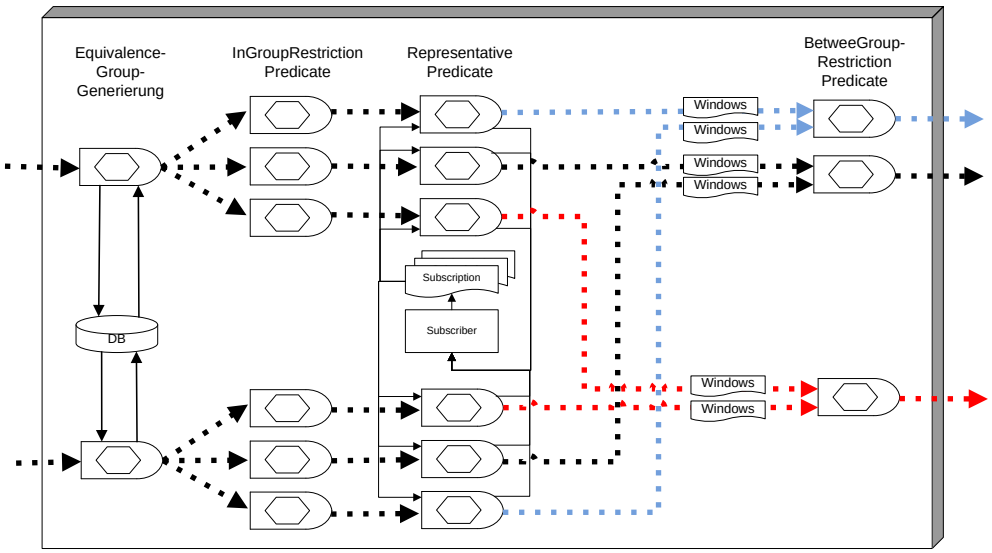

Abbildung 27: Umsetzung des FindAlternatingCommunicationsInTwoGroups-Operator. Dies Symbole sind aus [3] entnommen. Das Sechseck in den Queries sind Predicates, wie sie in Abs. 3.2.2 definiert werden.

zerfallen kann. Für jede Äquivalenzklasse wird innerhalb des Operators ein neuer Stream erzeugt. Damit der Operator für jede Äquivalenzklasse den Stream wiederfindet und nicht einen neuen Stream generiert, muss er die bereits definierten Stream-Äquivalenzklassen-Zuordnung mitspeichern. Dies ist in Abbildung 27 durch die Datenbank dargestellt. Anschließend wird auf jedem Äquivalenzklassen-Stream eine Abfrage platziert, die mit dem *InGroupRestrictionPredicate* entsprechende Events aussortiert, um dann mit der nächsten Abfrage einen Repräsentanten zu bestimmen.
Nun muss der CommunicationOperator für jede Äquivalenzklasse überprüfen, ob es die gleiche Äquivalenzklasse auch in der anderen EventGroup gibt, und diese dann in einem Stream zusammenführen. Auf diesem Stream kann anschließend die Abfrage mit dem *BetweenGroupRestrictionPredicate* platziert werden. Dies ist durch das *Dynamic Query Pattern* mit seinen Subscriptions und Subscribern dargestellt. Der Subscriber ist in diesem Fall der CommunicationOperator selbst.

b) FindCommunicationsInOneGroup Dieser Operator funktioniert ähnlich wie der *FindAlternatingCommunicationsInTwoGroups*-Operator mit dem Unterschied, dass er nur auf einer EventGroup arbeitet, daher kein *BetweenGroupRestrictionPredicate* benötigt und auch keine *RepresentativePredicate* besitzt. Der Anwender kann ihn also mit einem *Equi-*

valencePredicate und einem *InGroupRestrictionPredicate* steuern.
Der Ablauf ist ähnlich dem des *FindAlternatingCommunicationsInTwoGroups-Operators*. Zuerst werden die Events der EventGroup in Äquivalenzklassen eingeteilt und diese anschließend mit dem *InGroupRestrictionPredicate* gefiltert. Die übriggebliebenen Events bilden eine Kommunikation.
Abbildung 28 zeigt den Ablauf des Operators, wie er in einer CEP-Engine ablaufen könnte.

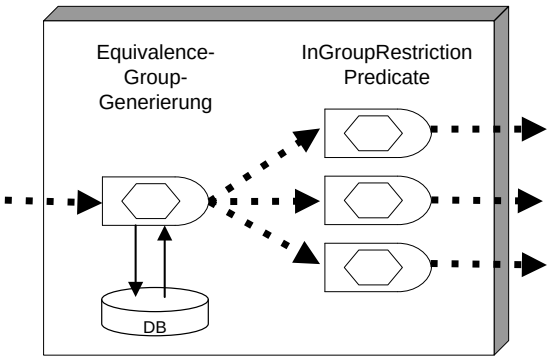

Abbildung 28: Umsetzung des FindCommunicationsInOneGroup-Operator.

c) AggregateCommunicationsOfOneSubmodel Dieser Operator arbeitet im Unterschied zu den ersten beiden Operatoren nicht auf EventGroups, sondern auf Communications und ist derzeit der einzige Operator, der in der Offline-Analyse in einem AggregateCommunicationModel verwendet werden kann.
Der Anwender kann den Operator mit folgenden Predicates steuern:

FilterCommunicationsPredicate Mit diesem Predicate kann der Anwender Communications aussortieren, die nicht den Mindestanforderungen entsprechen, um überhaupt weiter verarbeitet zu werden.

EquivalencePredicate Diese Predicate hat die gleiche Aufgabe, wie auch schon bei den Operatoren zuvor. Nur werden jetzt ganze Communications in Äquivalenzklassen eingeteilt und nicht einzelne Events.

CommunicationRestrictionPredicate Das *CommunicationrestrictionPredicate* funktioniert wie ein *InGroupRestrictionPredicate*, denn es filtert den Inhalt einer Äquivalenzklasse. In diesem Fall also Communications.

Der Ablauf des Operators stellt sich wie folgt dar: Zuerst werden die Communications mit dem *FilterCommunicationsPredicate* auf ihre Tauglichkeit für eine weitere Evaluation

überprüft. Die Communications, die dann noch im Stream sind, werden mit dem *EquivalencePredicate* in Äquivalenzklassen eingeteilt. Diese werden anschließend mit dem *CommunicationRestrictionPredicate* weiter ausgedünnt. Die Events, die jetzt noch im Stream sind, bilden eine komplexe Kommunikation.

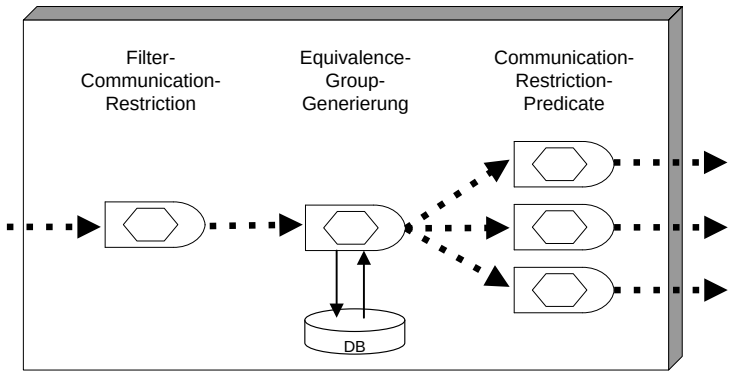

Abbildung 29: AggregateCommunicationOfOneSubmodel-Operator.

EventGroup-Generierung ModoCom arbeitet auf EventGroups. Die Sensoren liefern aber nur einzelne Events. Die Events müssen also in EventGroups eingeteilt werden, bevor sie mit dem ModoCom-Programm des Anwenders in der CEP-Engine untersucht werden können.
Für den Offline-Ansatz wird ein Clustering-Verfahren eingesetzt, um für die Analyse im Interpretationsmodul EventGroups zu finden. Die Arbeit von Jens Ellenberg [5] hat sich mit der Umsetzung von Datamining-Algorithmen mit Hilfe der Esper-Engine beschäftigt, deren Aussagen durchaus eine Allgemeingültigkeit für alle CEP-Engines haben. Ellenberg gibt an, dass lediglich Classification als möglicher Datamining-Algorithmus vollständig innerhalb der Esper-Engine realisiert werden kann. Für alle anderen Verfahren (auch beim Clustering) werden externe Komponenten benötigt, für die sich eine CEP-Engine nur zur Vor- und Nachbearbeitung der Daten eignet[5].

Mit dieser Aussage werden nun zwei Operatoren entworfen, mit Hilfe derer der Anwender im CEP-Framework EventGroups generieren kann. Dabei wird einmal ein Vorgehen beschrieben, dass eine externe Datamining-Komponente einbindet, wie es Ellenberg in [5] beschrieben hat. Der zweite Operator beschreibt ein ganz einfaches Vorgehen, um so etwas wie *spatial clusters* zu erstellen, wie sie derzeit im Offline-Ansatz verwendet werden. Dabei platziert der Operator beliebig viele Anfragen auf dem eintreffenden EventStream, die jeweils ein Rechteck über einer bestimmten Geo-Region aufspannen und nur Events durchlassen, die in diese Region fallen.

a) EventGroup-Generierung mittels externer Datamining-Komponente Zunächst werden eventuell enthaltene Null-Werte in den Event-Parametern durch Default-Werte ersetzt, damit die Events nicht für die Analyse verloren sind. Anschließend werden Events aussortiert, die nicht einer gewissen Mindestanforderung entsprechen. Nun können die Events in ein anderes Format transformiert werden, falls die externe Mining-Komponente ein anderes Format benötigt, und werden dann an die externe Komponente weitergeleitet. Dort werden die Events geclustert und wieder zurück in die Engine geschickt, wo sie nun von einem CommunicationModel ausgewertet werden können.

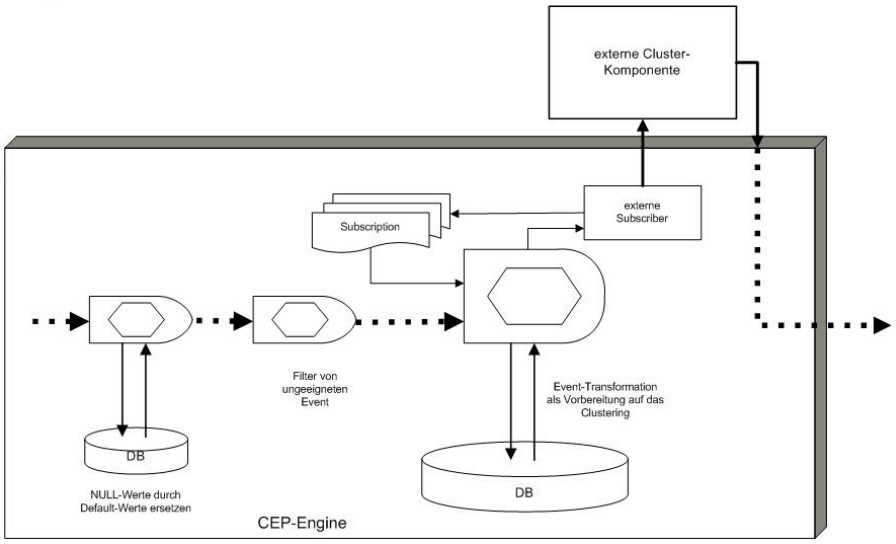

Abbildung 30: EventGroup-Generierung mittels Datamining.

b) EventGroup-Generierung mittels einfachen spatial clusters Das Vorgehen ist ähnlich dem ersten Operator. Die Events können in diversen Schritten für die anschließende Auswertung vorbereitet werden. Als letzter Schritt im Operator werden mehrere Statements, abhängig von der Anzahl der gewünschten EventGroups, auf dem Stream platziert. Jedes Statement filtert Events heraus, die in eine bestimmte Geo-Region fallen und versendet sie in einem eigenen Stream.

Sensor-Anbindung Durch die fehlenden Möglichkeiten in ModoCom Datenquellen zu spezifizieren, müssen die Sensoren programmatisch oder mit Konfigurationsdateien an das CEP-Framework angeschlossen werden.

Die Daten werden vom Sensor in das Framework gesendet. Dabei stellt sich die Frage, wie mit der Situation umgegangen werden muss, wenn der Sensor mehr Events in das Framework sendet, als das Framework verarbeiten kann. Entweder werden die Events in

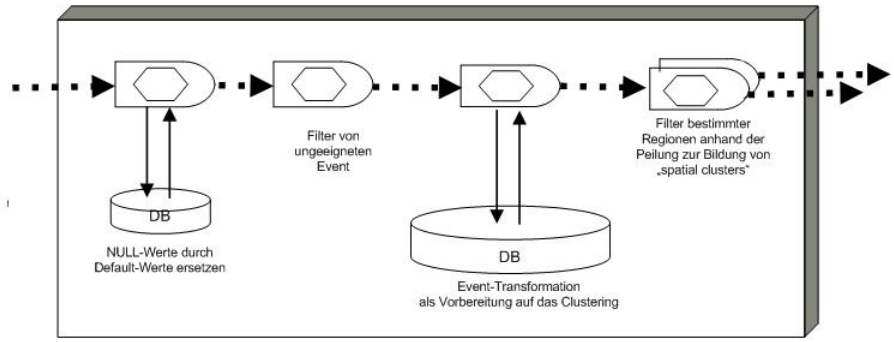

Abbildung 31: einfache EventGroup-Generierung ohne Clustering-

einer Queue gepuffert und dann von dort abgearbeitet, so dass kein Event für die Analyse verloren geht, oder diejenigen Events, welche zu schnell in das Framework gesendet werden, werden verworfen und stehen für die Analyse nicht zur Verfügung. Letzteres Vorgehen bringt das Problem mit sich, dass beim Verlust von Events, sich die Engine auf die neuen Events synchronisieren muss. Bei Realisierung mit Puffer würde der zeitliche Abstand zwischen Eintreffen der Events (am Sensor) und Erkennen einer Kommunikation größer werden.

Allerdings können CEP-Engines eine sehr große Menge von Events pro Minute verarbeiten, so dass ein Überschwemmen des Frameworks durch den Sensor relativ unwahrscheinlich ist.

Datenbank-Anbindung Auch für die Anbindung von Datenbanken gilt, dass diese nicht mittels ModoCom definiert werden kann. Der Fokus von ModoCom liegt natürlich auf dem Modellieren von Kommunikation. Dafür spielt eine Datenbankanbindung eine sehr untergeordnete Rolle. ModoCom verwendet CommunicationOperators , die außerhalb von ModoCom beschrieben sind und eigentlich nur diese Operatoren Zugriffe auf Datenbanken ausführen. Das Framework kann eine entsprechende Schnittstelle anbieten, über die die Operatoren ihre Zugriffe abwickeln können. Denkbar wäre aber auch ein Konfigurationsabschnitt innerhalb des ModoCom-Programms, mit dem sich das Framework vor dem Ausführen des Programms initialisieren lässt.

3.2.2 Abbildung: ModoCom nach CEP-Framework

Dieser Abschnitt behandelt diejenigen Teile, die in ModoCom definiert werden und daher in das CEP-Framework hinein abgebildet werden müssen. Diese Punkte sind: das ModoCom-Programm, das Konzept der CommunicationModels sowie die Predicates, welche in ModoCom definiert werden und die außerhalb von ModoCom beschriebenen CommunicationOperators steuern.

ModoCom-Programm Ein ModoCom-Programm bildet den alles umfassenden Rahmen. Das Programm bestimmt die zu verwendenden Datentypen und enthält die vom Anwender spezifizierten Predicates. Außerdem werden hier die CommunicationModels definiert. Die im Programm aufgebaute Struktur wird dann in ein CEP-Programm abgebildet. Das CEP-Programm hat Zugriff auf das CEP-Framework und auf die CEP-Engine, um diese konfigurieren zu können.
Um das Framework vollständig initialisieren zu können, müsste das ModoCom-Programm allerdings auch beschreiben, wie die Sensoren an das Framework angebunden werden, wo und wie das Framework auf eine Datenbank zugreifen kann und vorallem wie die Events in EventGroups einzuteilen sind.

CommunicationModels Im CommunicationModel verknüpft der Anwender die zuvor im Programm definierten Datentypen mit CommunicationOperators aus dem Framework und steuert diese mit den Predicates. Zudem wird im Modell die Auswertereihenfolge der einzelnen Modelle festgelegt, indem jedes Modell das Modell bestimmt, welches jeweils vor ihm auszuwerten ist.
Ein CommunicationModel verwendet jeweils einen, der in Abschnitt 3.2.1 beschriebe-

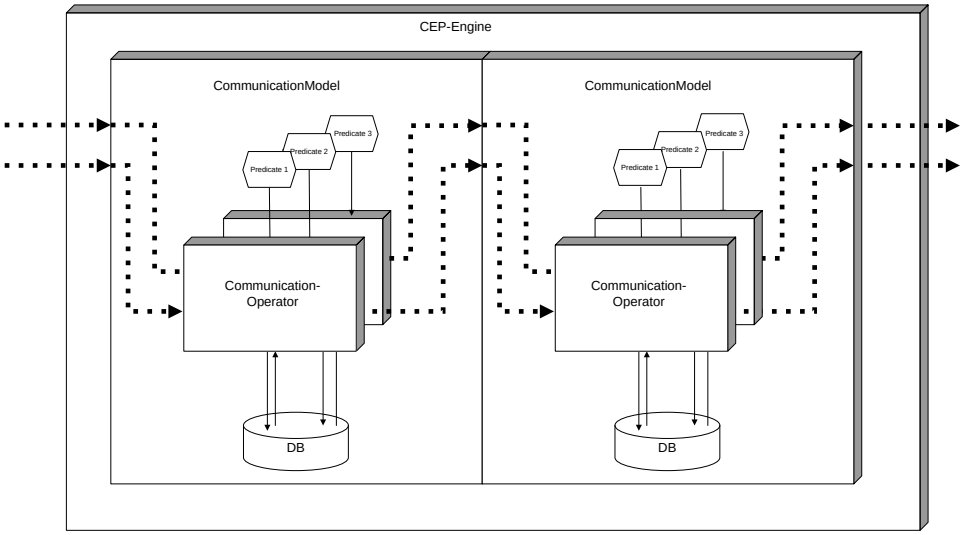

Abbildung 32: Einbindung der CommunicationModels mit ihren Communication-Operators in die CEP-Engine.

nen CommunicationOperators und verankert diese in der CEP-Engine. Zu diesem Zweck

muss das Modell wissen, welche Streams in der Engine existieren, um darauf den Operator zu platzieren. Damit das Modell einem nachfolgenden aggregierten Modell ebenfalls die entsprechenden Informationen zur Verfügung stellen kann, speichert es die Meta-Informationen der vom eigenen Operator neu generierten Streams.
Das heißt, das Modul, welches die EventGroups generiert, teilt dem einfachen CommunicationModel mit, welche Streams in der Engine erzeugt wurden und auf welchen sich das Model registrieren soll.
Das einfache CommunicationModel platziert daraufhin seinen CommunicationOperator. Der Operator teilt dem CommunicationModel mit, welche Communications gefunden wurden und in welchen Streams deren Informationen zu finden sind. Diese Information reicht das CommunicationModel an das nach ihm folgende aggregierte CommunicationModel weiter, welches dann auf diesen Streams seinen Operator platzieren kann.

Die ModoCom-Beschreibung eines CommunicationModels gibt eine bestimmte Anzahl EventGroups oder Communications vor, die benötigt werden, um die im Modell beschriebene Kommunikation zu erkennen. Im Idealfall liefert der Sensor weitaus mehr Events beziehungsweise EventGroups als die in der Modellbeschreibung angegebene Anzahl. Das heißt, dass jede weitere EventGroup oder Communication eine neue Möglichkeit liefert die EventGroups zu kombinieren um sie dann auszuwerten. Doch auch bereits bei der im Modell geforderten Mindestanzahl an EventGroups oder Communications kann es, abhängig vom Operator sein, dass gegenebenfalls. eine Permutation der existierenden EventGroups zu neuen Ergebnissen führen kann. Bei der späteren Realisierung muss entschieden werden, an welcher Stelle diese Kombinationen und Permutationen einer Kombination erzeugt werden und wie ein CommunicationOperator damit umzugehen hat. Da dem CommunicationModel eine verwaltende Rolle zugeteilt wird, wird es sicherlich sinnvoll sein, das CommunicationModel als einzelne Instanz zu realisieren und für jede neue Kombination an EventGroups oder Communications eine neue Instanz des CommunicationOperators zu erzeugen, welche dann ihre Ergebnisse an die zentrale Instanz des CommunicationModels zurückliefern kann. Abbildung 32 zeigt, wie eine solche Struktur aussehen kann. Dort sind pro Modell jeweils zwei Operator-Instanzen dargestellt.

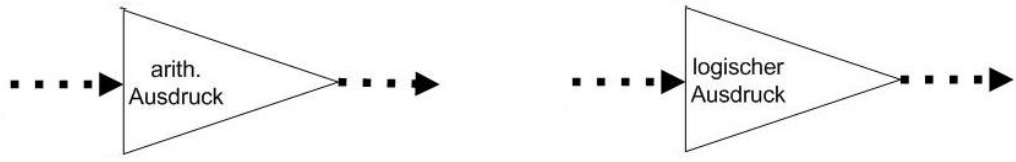

Abbildung 33: Restrictions: arithmetischer und logischer Ausdruck für ein Predicate.

Predicates Predicates steuern die CommunicationOperators. Sie werden den Operatoren übergeben, welche sie dann entsprechend der verwendeten Engine (siehe Abs. 2.2.2 -

Anfragesprachen) in einem Query platzieren. Zu überlegen ist sicherlich, in welcher Form die Predicates gespeichert oder weitergereicht werden können. Das hängt von der Implementierung und Generizität der CommunicationOperators ab. Ob Operatoren für jede Engine Predicates in Form eines Strings einbetten können, oder ob Predicates in einem Predicate-Datentyp gehalten werden müssen, von wo aus sie in die passende Form für einen Operator transformiert werden können, bleibt an dieser Stelle offen.
Zunächst bestehen Predicate aus logischen und arithmetischen Ausdrücken (den Restrictions), die direkt in ein EPL-Query übernommen werden können und in der weiteren Untersuchung durch die in Abbildung 33 abgebildeten Symbole dargestellt werden. Der An-

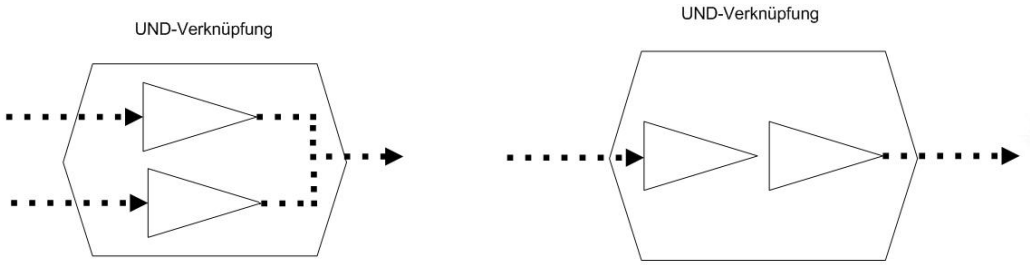

Abbildung 34: (a) UND-Verknüpfung von zwei Streams, (b) UND-Verknüpfung auf einem Stream.

wender hat im Predicate die Möglichkeit diese Ausdrücke UND und ODER zu verknüpfen. Ein Predicate kann für ein oder mehrere EventGroups oder Communications definiert werden. So verknüpft bspw. Abbildung 34 (a) zwei Streams miteinander in einem Predicate, während in 34 (b) zwei Ausdrücke auf einem Stream platziert werden.

In Abbildung 35 werden zwei Ausdrücke ODER-verknüpft.

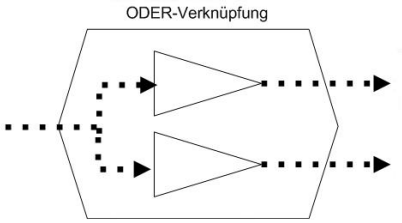

Abbildung 35: ODER-Verknüpfung

Außerdem erlaubt ModoCom dem Anwender die Restrictions in einem IF THEN ELSE Statement zu platzieren. Dieses kann wie in Abbildung 36 in CEP umgesetzt werden. IF THEN ELSE Statements müssen für eine CEP-Engine in logische Statements transformiert werden. Dies ist aber einfach, wie folgendes Beispiel darstellt:

$$\text{if a then b else c} \Rightarrow (a \wedge b) \vee (\neg a \wedge c)$$

Ersetzt man nun b durch “if x then y else z” ergibt sich daraus

$$(a \wedge ((x \wedge y) \vee (\neg x \wedge y))) \bigvee (\neg a \wedge c)$$

Die ELSE-Restriction ist also die negierte IF-Restriction. Diese Restrictions und Predicates können dann beliebig tief UND und ODER verknüpft werden.

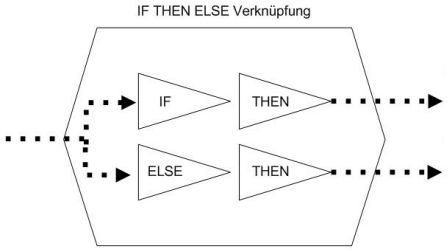

Abbildung 36: IF THEN ELSE Verknüpfungen von Restrictions.

Sorting ModoCom erlaubt dem Anwender, bei der Definition eines ObjectTypes, eine Sortierfunktion zu definieren, mit der der Anwender Einfluss auf die Auswertungsreihenfolge der einzelnen ObjectType-Instanzen nehmen kann.
Da die Daten vorgegeben sind, kann diese Eigenschaft nicht in die Daten hinein abgebildet werden. Das Sortieren passiert zudem ja nicht in den Daten, sondern auf den Daten, wird als vom Framework beziehungsweise der CEP-Engine vorgenommen.

Mit Hilfe eines Fensters und darauf operierenden Funktionen aus der CEP-Engine, können ObjectTypes entsprechend einer vom Anwender angegeben Sortierfunktion neu angeordnet werden.

3.3 Fazit

Zusammenfassend lässt sich feststellen, dass der Fokus der, von der Fa. Plath GmbH entwickelten Sprache ModoCom ganz klar auf dem Beschreiben von Kommunikationsmodellen liegt. Massiven Einfluss auf ein solches Modell hat aber auch die Generierung der EventGroups, für die ModoCom aktuell noch keine Möglichkeit zur Beschreibung bereithält.
Die Anbindung der Sensoren und der Datenbank kann auch als Teil des Frameworks betrachtet und muss nicht notwendigerweise in ModoCom umgesetzt werden.

Festzustellen ist auch, dass ModoCom bezüglich der verwendbaren Datentypen nicht so generisch ist, wie Kapitel 2 zunächst vermuten lässt. In ModoCom können nur Datentypen beschrieben werden, die auch im Framework zur Verfügung stehen.

Ein weiteres Problem ist, dass mit ModoCom nur die Felder eines Datentyps in ModoCom wiedergegeben werden können, der Anwender aber nicht beschreiben kann, wie die Werte der Felder gebildet werden sollen.

Zusammenfassend sollte ModoCom um folgende Fähigkeiten erweitert werden, um entsprechend generisch auf dem Unterbau aufsetzen zu können

- ModoCom muss beschreiben können, wie Attribute eines ObjectTypes gebildet werden.
- ModoCom muss beschreiben können, wie und nach welchen Kriterien oder mit welchen Verfahren EventGroups erzeugt werden sollen. Dies ist unabhängig davon, ob die Einteilung mit einer externen Mining-Komponente oder mit einem einfacheren Verfahren direkt in der CEP-Engine realisiert werden soll.
- Wird ModoCom um eine Möglichkeit zu Bildung von EventGroups erweitert, sollte auch eine Möglichkeit bereitgestellt werden, mit der die Sortierung und das dafür notwendige Fenster definiert werden können.

Darüber hinaus bleibt zu überlegen, wo die Anbindung der Sensoren und der Datenbank beschrieben werden können.

4 Konzeption einer Architektur

In diesem Kapitel wird eine Architektur entworfen, welche die in Kapitel 3 beschriebenen Konzepte in einem Framework umsetzt, das auf einer CEP-Engine aufsetzt. Prinzipiell spiegeln die einzelnen Komponenten die im vorherigen Kapitel besprochenen Konzepte wider. Ausgehend von einer allgemeinen Architektur, in der alle Komponenten enthalten sind, wird dann ein Prototyp-Framework abgeleitet, welches einige der Konzepte beispielhaft umsetzt.

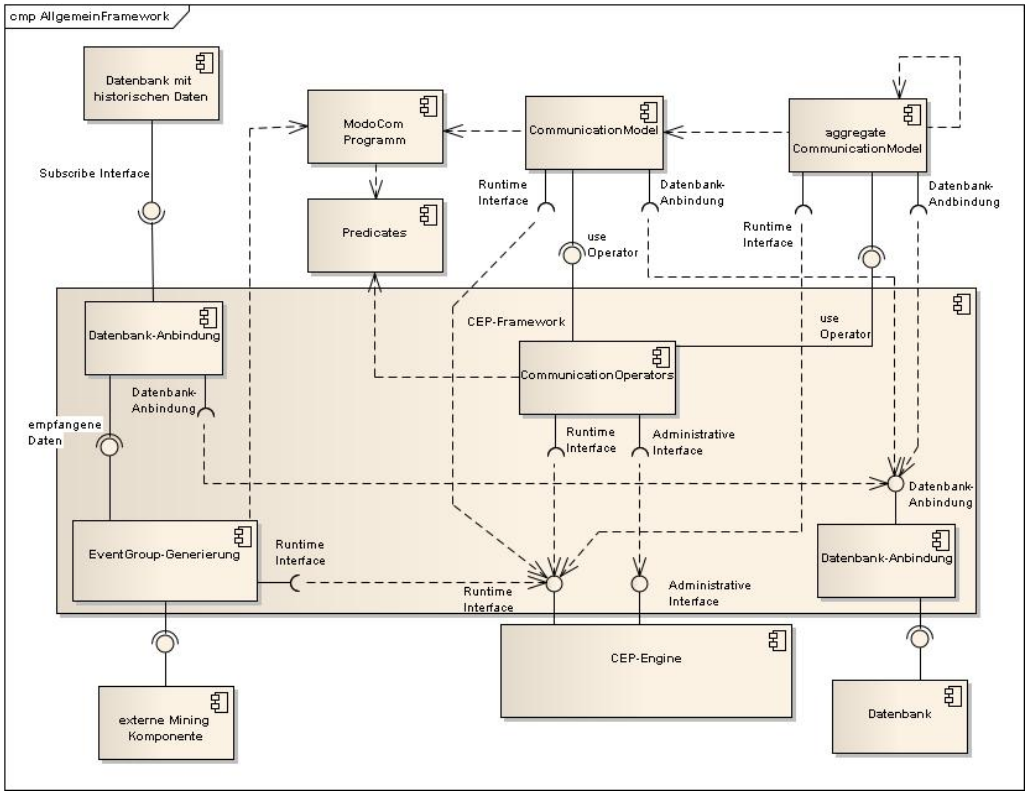

Abbildung 37: Komponenten eines möglichen CEP-Frameworks für ModoCom.

4.1 Sensor und Sensor-Daten

Ein Sensor ist ein Bauteil, das Eigenschaften beziehungsweise Änderungen der Eigenschaften seiner Umgebung aufnimmt und diese qualitativ oder als quantitative Messgröße weitergibt. Das kann ein Funkpeiler oder aber auch ein Knoten in einem Computer -Netzwerk sein, die jeweils versendete Daten empfangen und entsprechend aufbereiten, damit sie von anderen Systemen verarbeitet werden können.
Dabei bestimmt der Sensor, wie die Daten aussehen, die er bereit stellt. Die Daten sind typischer Weise im XML-Format beschrieben, so dass sie sehr leicht an andere Systeme

weitergegeben werden können.

Der Sensor ist eine externe Komponente und nicht Teil des Frameworks.

4.2 Sensor-Anbindung

Diese Komponente ist das Bindeglied zwischen den Sensoren und dem Framework. Zum Framework hin realisiert es die Schnittstelle, um Daten der Komponente für die EventGroup-Generierung zuzuführen. Zur Sensor-Seite hin muss es die Schnittstelle des jeweils anzuschließenden Sensors implementieren.

Diese Komponente muss auch ein eventuelles Überfluten des Frameworks durch den Sensor behandeln. In ihr enthalten sein, kann also eine Queue, welche die Events zwischenspeichert, bis sie vom Framework verarbeitet werden können. Oder aber die zu viel gesendeten Events werden von dieser Komponente verschluckt und nicht an das System weitergeleitet. Das Framework sollte entsprechend über beide der Situationen informiert werden.

4.3 EventGroup Generierung

Bevor die Events mit ModoCom untersucht werden können, müssen sie in EventGroups eingeteilt werden. Wie in Abschnitt 3.2.1 bereits angesprochen, ist für die Generierung sinnvoller und aussagekräftiger EventGroups eine externe Mining-Komponente sehr wahrscheinlich unumgänglich. In dieser Komponente werden die Daten für ein Datamining-Verfahren vorbereitet, um sie dann entweder in der CEP-Engine oder einer externen Mining-Komponente in EventGroups einzuteilen.
Dabei wird für jede EventGroup ein eigener Stream erzeugt, in den die Daten dann gesendet werden können. Gleichzeitig wird dem ersten CommunicationModel mitgeteilt, auf welchen Streams es seine Operatoren platzieren muss.

4.4 Datenbank und Datenbank-Anbindung

Die Anbindung einer Datenbank ist nicht in allen Fällen notwendig, doch gibt es einige Aufgaben, die mit einer Datenbank gelöst werden können. So können beispielsweise historische Daten aus einer Datenbank gelesen werden, mit Hilfe derer Entscheidungen bei der Auswertung einzelner Events getroffen werden. Aber auch Erkenntnisse die während der Auswertung gewonnen werden und zur Steuerung der Operatoren wichtig sind (beispielsweise welche Äquivalenzklassen bereits gefunden wurden und in welchem Stream deren Events liegen), können in einer Datenbank gespeichert und wieder ausgelesen werden.
Nicht zuletzt die Zwischen- und Endergebnisse der CommunicationModels können in der Datenbank gespeichert werden.
Das Framework bietet für alle Komponenten eine zentrale Schnittstelle an, so dass die

dahinter angebundene Datenbank ausgetauscht werden kann, ohne die einzelnen Komponenten anpassen zu müssen.

4.5 CommunciationOperators

Diese Komponenten werden vom Anwender selbst entwickelt und müssen gegen das Interface des CommunicationModels implementiert werden, welches die Operatoren verwendet. Da die Operatoren Statements in einer CEP-Engine platzieren, müssen sie auch gegen das Interface der entsprechenden Engine implementiert werden. Ob es möglich ist, eine Abstraktionsschicht für CEP-Engines des gleiches Anfragetyps zu entwickeln, so dass die Engine ausgetauscht werden muss, ohne die CommunicationOperators neu schreiben zu müssen, muss untersucht werden.

Die Umsetzung der CEP-Pattern aus 2.2.3 hängt von der jeweils verwendeten Engine ab.

4.6 Predicates

Predicates dienen zum Steuern der CommunicationOperators. Müssen also von diesen verwendet werden können. In welcher Form die aus dem ModoCom-Programm gewonnen Informationen eines Predicates für die Verwendung im Framework zwischengespeichert werden, hängt also von den CommunicationOperators ab.

4.7 CommunicationModels

Ein CommunicationModel verbindet alle in einem ModoCom-Programm definierten Elemente miteinander. Es teilt dem CommunicationOperator mit, auf welchen Streams EventGroups und Communications sind und steuert den Operator mit den, vom Anwender definierten Predicates. Zusätzlich teilt das CommunicationModel die Ergebnisse seines CommunicationOperators nachfolgenden aggregierten CommunicationModels mit.

EventGroups und Communications existieren in einer CEP-Engine in Form von Streams die, ähnlich wie Tabellennamen, durch Stream-Namen repräsentiert werden. Das einfache CommunicationModel erhält von der EventGroup-Generierungskomponente die Namen der Streams, in denen sich EventGroups befinden. Diese kombiniert das Modell dann entsprechend mit Instanzen des CommunicationOperators. Sobald der Operator neue Streams mit Ergebnissen erzeugt, teilt er diese dem CommunicationModel mit, damit dieses nachfolgende Modelle informieren kann. Da aggregierte CommunicationModels für die Auswertung der Communications nicht mehr an den einzelnen Events interessiert sind, sondern nur an den Meta-Informationen einer Communication, wird bei der Realisierung der Communications wie folgt vorgegangen:
Für jeden Ergebnis-Stream verwaltet das CommunicationModel ein zusätzliches Objekt, das *CommunicationObject*, in dem die Meta-Informationen gespeichert werden.

- Bei der Realisierung der *happening Communication* wird, sobald ein neuer Event einer Communication zugeordnet wurde, vom *CommunicationObject* ein *Communication-Event* generiert, der den aktuellen Zustand der Communication widerspiegelt. Diese Events werden dann in den Operatoren aggregierter Modelle permanent ausgewertet. Dies kann dann zu einer neuen *happening Communication* führen, die abgebrochen oder beendet werden kann.

- Wird die *finished Communication* realisiert, erfährt ein nachfolgendes CommunicationModel erst von der Existenz einer Communication, wenn diese beendet ist. Bei diesem Vorgehen wird, wie auch bei der *happening Communication*, ein *CommunicationObject* erzeugt, welches die Events einer Kommunikation sammelt. Es werden aber nicht permanent neue *CommunicationEvents* generiert. Erst, wenn die Kommunikation abgeschlossen ist, teilt das CommunicationModel den nachfolgenden Modellen mit, dass es eine Communication gefunden hat. Die nachfolgenden Modelle können nun auf das CommunicationObject zugreifen und sich von ihm einen *CommunicationEvent* generieren lassen, den sie dann auswerten können. Im Falle der *finished Communications* wird also pro Auswertung einer Kommunikation im Operator nur einmal genau ein Event benötigt.

5 Prototypische Implementierung

Auf den folgenden Seiten soll ein Prototyp entworfen werden, mit dem es möglich ist, Kommunikationsmodelle in einer CEP-Engine zu evaluieren. Zunächst wird eine Architektur für den Prototyp vorgestellt. Anschließend werden die einzelnen Komponenten besprochen. Im letzten Teil dieses Kapitels wird der Prototyp getestet.

5.1 Die Architektur des Prototypen

In diesem Abschnitt wird das in Kapitel 4 beschriebene Framework für eine Realisierung als Prototyp angepasst. Dafür werden einige Komponenten weggelassen oder durch weniger komplexe Komponenten ersetzt. Abbildung 38 zeigt die Komponenten des Prototypen. Der Prototyp wird wegen der geringeren Komplexität die *finished Communication* realisie-

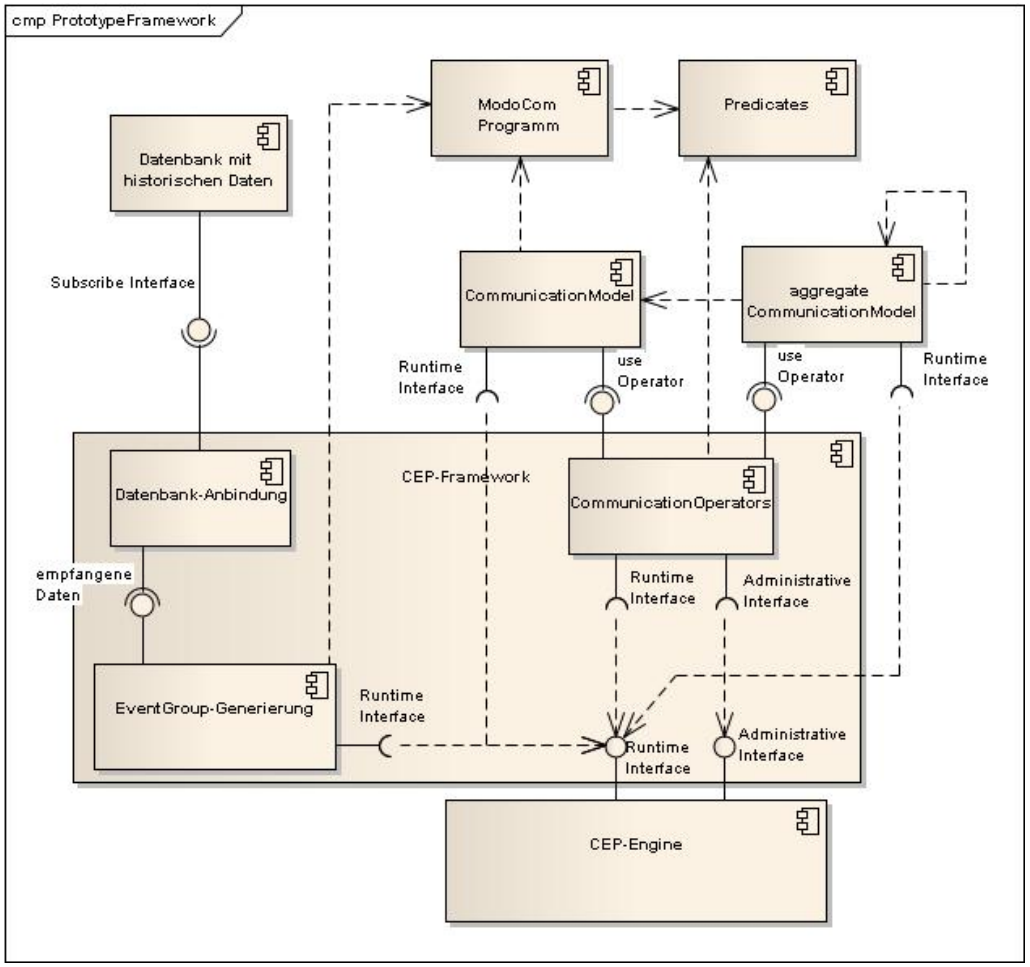

Abbildung 38: Komponenten des CEP-Framework-Prototyps für ModoCom.

ren, da hierfür nicht überwacht werden muss, ob eine Kommunikation abgebrochen werden muss. Dadurch wird das Verwalten der CEP-Engine einfacher, da bei einem Abbruch die jeweiligen Statements wieder aus der Engine hätten entfernt werden müssen. Ebenso wird auf die Realisierung einer zentralen Datenbankanbindung verzichtet.

5.1.1 Allgemeine Konzepte

Zunächst sollen einige Konzepte erwähnt werden, die auf alle, oder zumindest auf viele Komponenten zutreffen.

Informationsfluss Es wird unterschieden, zwischen dem Informationsfluss und dem Datenfluss. Der Informationsfluss beinhaltet alle zur Steuerung notwendigen Daten. Dazu gehören Informationen darüber, welche Event-Menge auf welchem Stream gefunden werden kann oder ob eine Kommunikation *finished* ist.

Diese Informationen werden fast ausschließlich mittels des Observer-Patterns zwischen den einzelnen Komponenten ausgetauscht.

Datenfluss Der Datenfluss meint den Weg der Events. Diese werden von der Sensor-Anbindungskomponente beziehungsweise der Test-Datenbank-Anbindung in die Esper-Engine gesendet und wandern dann mittels Streams durch die einzelnen Auswertungskomponenten.

Eventrepräsentation Der Prototyp arbeitet mit Events, die durch *java.util.Map*-Objekte realisiert sind. Die Entscheidung wurde willkürlich getroffen und bietet im ersten Moment keine Vor- oder Nachteile gegenüber den anderen Datentypen.

5.1.2 Einspeisen der Sensor-Daten

Für eine beispielhafte Implementierung ist der Anschluss an einen realen Sensor zu aufwendig, zumal das Mithören und Auswerten von Kommunikation ohne entsprechende Genehmigung verboten ist. Daher arbeitet der Prototyp mit Test-Daten, welche die Firma Plath für Testzwecke in einer Datenbank gespeichert hat. Der Vorteil dieser gespeicherten Daten ist, dass sich damit Tests auf dem Prototypen reproduzieren lassen, was mit Live-Daten nicht möglich ist.
Die Komponente für die Sensor-Anbindung wird im Falle des Prototyps zur Anbindung für die Datenbank mit den Testdaten. Dort werden die Daten in *java.util.Map* Instanzen verpackt und in die Esper-Engine versendet.

Die in der Datenbank gespeicherten Emissionen werden nach End-Zeitpunkt sortiert und können somit in der Reihenfolge in die Esper-Engine gesendet werden, wie sie auch ein Sensor liefern würde.

Da die Evaluation der Daten auf den Start- und Endzeiten der Emissionen beruht, ist es möglich, die Testdaten zeitlich sehr komprimiert in die Esper-Engine zu senden. Lediglich die Zeitfenster auf einem Stream müssen dieser Komprimierung angepasst werden.
Der Vorteil dieser Komprimierung ist, dass auch größere Testdatensätze in relativ kurzer Zeit ausgewertet werden können. Es ist natürlich ebenso möglich, die Differenz zwischen den chronologisch sortieren Emissionen zu berechnen, so dass die Emissionen in "Echtzeit" in die Esper-Engine gesendet werden können.

5.1.3 Generieren der EventGroups

Für den Prototyp wird die Generierung der EventGroups relativ einfach gehalten. EventGroups werden, wie in Abschnitt 3.2.1 und Abbildung 31 dargestellt, mit einfachen Rechtecken über der Peilung generiert. Dies hat den Vorteil, dass hier der Aufwand eine Datamining-Komponente zu integrieren entfällt.

5.1.4 CommunicationObjects

Da mit ModoCom nicht beschrieben werden kann, wie die Werte für die Attribute gebildet werden können, wird dieses Wissen in sog. *CommunicationObjects* gespeichert.
Zum Instanzieren eines *CommunicationObjects* wird der Name des Streams benötigt, auf dem die Events der Kommunikation gesendet werden, sowie das Modell, zu dem die Kommunikation gehört und eine CommunicationID, welche die Kommunikation eindeutig identifiziert.
Mit der Methode *addEvent()* kann der CommunicationOperator neue Events zur Kommunikation hinzufügen. Die Kommunikation kümmert sich dann selbst darum, wie die Informationen der ihr übergebenen Events ausgewertet werden müssen.
Mit *getCommunicationEvent()* kann vom *CommunicationObject* ein Event angefordert werden, der den Zustand der Kommunikation wiedergibt. Dieser wird auf dem im Konstruktor angegebenen Stream versendet.

Für den Prototyp wurden zwei CommunicationObjects implementiert:

- SimpleCommunication und
- AggModelCommunication

Ersteres verwertet mit seiner *addEvent()*-Methode einzelne Events. Letzteres erwartet Events vom Typ *CommunicationEvent* um seinen Status zu aktualisieren.

Mit dem Erzeugen eines CommunicationObjects, wird ein Timer gestartet. Ist der Timer abgelaufen, bedeutet dies, dass für diese Kommunikation keine neuen Events mehr gefunden werden können und die Kommunikation in den Zustand *finished* übergeht. Dieser Zustandswechsel, wird ebenfalls mittels des Observer-Patterns an das CommunicationModel der Kommunikation gemeldet. Durch das Hinzufügen eines neuen Events zur Kommunikation wird der Timer neu gestartet.

5.1.5 CommunicationModels

Der Informationsfluss der CommunicationModels wird über das Observer-Pattern realisiert. Dabei entscheidet das ModoCom-Programm, welche Modelle wo als Observer registriert werden.

Ein CommunicationModel instanziert den durch das Modell vorgegebenen Operator und verwaltet dessen Ergebnisse in einer Liste. Beide Aufgaben konnten in einen abstrakten Supertyp ausgelagert werden, so dass sich die Implementierung eines konkreten Modells auf die Predicates und das Implementieren der update()-Methode beschränkt, durch die das CommunicationModel über neue Ergebnisse informiert wird und den Communication-Operator startet.

5.1.6 CommunicationOperators

In Anlehnung an die von der Fa. Plath GmbH entwickelten Beispiele aus [8], wurden für den Prototyp zwei CommunicationOperator implementiert. Je einen ür ein einfaches CommunicationModel und einen für ein aggregiertes CommunicationModel.
Für das einfache Modell wurde der *FindAlternatingCommunicationsInTwoGroups* Operator implementiert, für die aggregierten Modelle der *AggregateCommunicationOfOneSubmodel* Operator.

FindAlternatingCommunicationsInTwoGroups Der Operator bekommt vom CommunicationModel die Namen zweier Streams übergeben, welche jeweils eine EventGroup repräsentieren. Der Operator beginnt zunächst jeden Stream getrennt zu verarbeiten. Abbildung 39 verdeutlicht den Ablauf.

Als erstes wird jeder Stream in Äquivalenzklassen eingeteilt. Ein Statement liest alle Events vom Stream ein und reicht sie an einen UpdateListener weiter. Dieser prüft, ob der Event in eine bereits existierende Äquivalenzklasse passt. Hierfür wird das EquivalencePredicate in eine Java-Methode überführt, welche true oder false als Return-Wert hat. Die Methode vergleicht den neuen Event mit dem Repräsentant der Äquivalenzklasse repräsentiert. Erfüllen beide das EquivalencePredicate, wird der Event in die Äquivalenzklasse einsortiert, in dem er auf den Stream weitergeleitet wird, welcher die Klasse in der Esper-Engine repräsentiert.
Existiert noch keine Äquivalenzklasse oder wurde keine passende Klasse gefunden, wird mit Hilfe des RepresentativePredicates festgestellt, ob der Event als Repräsentant einer neuen Äquivalenzklasse fungieren kann. Hierfür wird das RepresentativePredicate ebenfalls in eine Java-Methode überführt.
Erfüllt der Event das Predicate, wird ein neuer Stream erzeugt. Der UpdateListener erzeugt ein Äquivalenzklassen-Objekt, in welchem er den Repräsentanten zusammen mit dem Stream-Namen speichert. Für jeden Stream verwaltet der UpdateListener eine Liste an Äquivalenzklassen-Objekten. Für jeden neuen Event wird nun über die Liste iteriert und jeder Repräsentant mit dem neuen Event mit dem EquivalencePredicate untersucht.

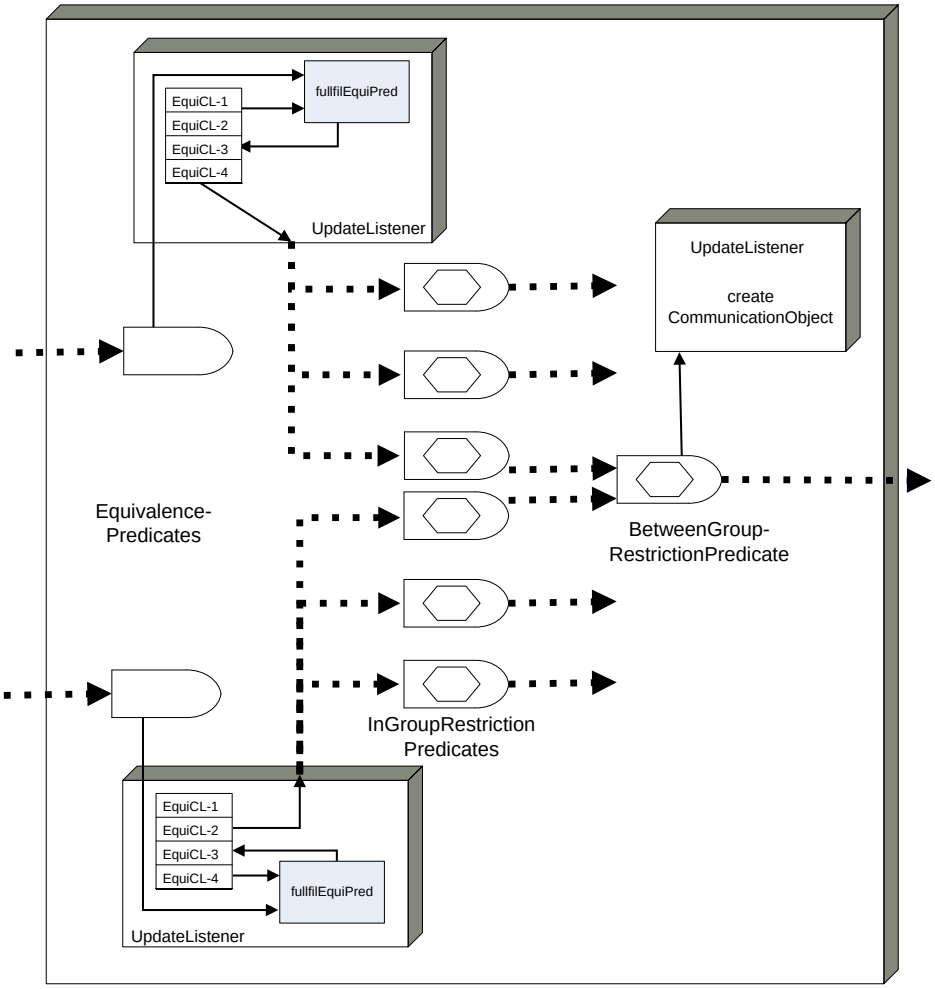

Abbildung 39: FindAlternatingCommunicationsInTwoGroups-Operator für Esper.

Für jede Äquivalenzklasse platziert der Operator ein Statement auf dem entsprechenden Stream, welches das InGroupRestrictionPredicate auswertet. Hierfür wurde das Predicate in einen String umgewandelt, welcher bequem in der Where-Klausel des Statements platziert werden kann. Gleichzeitig mit dem Platzieren des InGroupRestrictionPredicates wird auch das BetweenGroupRestrictionPredicate platziert. Hierfür wird der Repräsentant der neuen Äquivalenzklasse mit den existierenden Äquivalenzklassen aus dem anderen Stream verglichen. Wie beim Finden einer existierenden Äquivalenzklasse innerhalb desselben Streams, wird auch hier die Methode mit dem EquivalencePredicate benutzt, um den Repräsentanten des einen Streams mit allen Äquivalenzklassen des anderen Streams zu vergleichen. Wird eine passende Klasse gefunden, wird für die Streams beider Klassen ein Join-Statement in der Engine platziert, welches das BetweenGroup-

RestrictionPredicate ebenfalls in Form eines Strings in der Where-Klausel platziert und auswerten lässt. Die Ergebnisse dieses Statements sind Teil einer Kommunikation und werden einem UpdateListener weitergereicht. Dort wird überprüft, ob bereits ein CommunicationObject erzeugt wurde und die Events diesem übergeben werden können. Existiert noch kein *CommunicationObject*, wird dieses erzeugt.

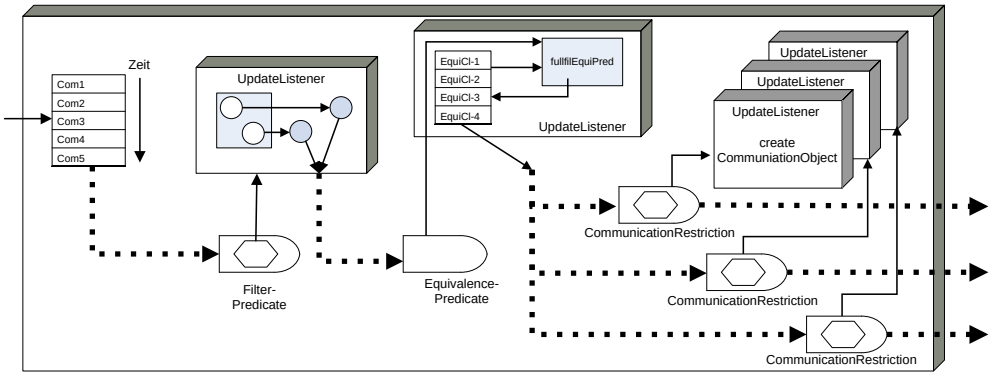

Abbildung 40: AggregateCommunicationOfOneSubmodel-Operator für Esper.

AggregateCommunicationOfOneSubmodel Der Operator bekommt ein CommunicationObject übergeben, von welchem er sich einen CommunicationEvent generieren lässt. Dieser wird wie folgt ausgewertet: (vgl. Abbildung 40)

Zunächst wird der CommunicationEvent chronologisch nach Startzeit in eine Liste bereits eingetroffener CommunicationEvents einsortiert. Der Operator des Offline-Ansatzes iteriert nun über die vollständige Liste und vergleicht jede Kommunikation mit ihrer Vorgänger- und Nachfolger-Kommunikation.
Für die Online-Analyse kann nicht gewartet werden, bis alle Kommunikationen des Modells eingetroffen sind, da nicht festgestellt werden kann, wann dieser Zeitpunkt erreicht ist. Daher wird jede eintreffende Kommunikation direkt ausgewertet. Dadurch, dass die Kommunikationen nicht chronologisch nach Startzeit sortiert eintreffen, aber nach der chronologischen Sortierung jeweils der Vorgänger und Nachfolger zum jeweiligen Zeitpunkt bestimmt werden, entstehen in diesem Operator mehr Kombinationen von Kommunikationen, die ausgewertet werden müssen. Dabei sortiert das FilterPredicate ungültige Kombinationen aus.
Für diesen Zweck wird das FilterPredicate in einen String umgewandelt, der in der Where-Klausel eines Statements platziert wird. Das Ergebnis dieses Statements ist ein komplexer Event, welcher die beiden ausgewerteten Events beinhaltet. In einem an das Statement angeschlossenen UpdateListener wird dieser Event wieder in zwei einzelne Events zerlegt, welche wieder in die Esper-Engine gesendet werden.
Dort werden sie wie auch bereits beim *FindAlternatingCommunicationsInTwoGroups-*

Operator in Äquivalenzklassen eingeteilt. Der Ablauf ist der gleiche, mit dem einzigen Unterschied, dass dieser Operator kein RepresentativePredicate verwendet. Das bedeutet, dass immer der erste gefundene Event zum Vergleichen der Äquivalenzklassen verwendet wird.
Auf dem Stream jeder gefundenen Äquivalenzklasse wird ein weiteres Statement platziert, welches in der Where-Klauel das *CommunicationRestrictionPredicate* enthält.
Auf diesem Statement ist wie auch im *FindAlternatingCommunicationsInTwoGroups*-Operator ein UpdateListener platziert, der sich um die Instanzierung und Befüllung eines CommunicationObjects kümmert.

5.1.7 Predicates

Wie bereits im Abschnitt 5.1.6 zu erkennen ist, werden für den Prototyp die Predicates eines CommunicationModels an verschiedenen Stellen in verschiedenen Formaten benötigt.
Wichtig für eine spätere Übersetzung aus einem ModoCom-Programm heraus ist, dass die im ModoCom-Programm verwendeten Variablennamen für die Events auch in den Statements verwendet werden, damit die Engine darauf zugreifen kann.

5.2 Kommunikationsmodelle des Prototypen

Für den Prototyp werden drei Kommunikationsmodelle realisiert. Alle drei Modelle wurden von der Fa. Plath GmbH enwickelt und sind aus [8] entnommen. Sie können auf Anfrage über die Fa. Plath GmbH bezogen werden.

5.2.1 TwoPartnerSimplex

Bei *TwoPartnerSimplex* handelt es sich um ein einfaches Kommunikationsmodell. Es verwendet den *FindAlternatingCommunicationsInTwoGroups*-Operator, den es mit entsprechenden Predicates initialisiert.

Das Modell bildet Äquivalenzklassen mit Emissionen gleicher Frequenz und gleicher Modulation. Emissionen deren Modulation unbekannt ist werden auf alle Äquivalenzklassen verteilt, welche die gleiche Frequenz haben.
Die Äquivalenzklassen werden dann auf Events untersucht, zwischen denen der zeitliche Abstand nicht größer als der eingestellt *delay* ist. Die Äquivalenzklassen werden mit passenden Äquivalenzklassen aus der zweiten EventGroup kombiniert und wieder werden Events gesucht, deren zeitlicher Abstand den *delay* nicht überschreitet. Diese Events bilden dann eine Kommunikation.

5.2.2 MultiplexPairsWithTimeOffset

Hier handelt es sich ebenfalls um ein einfaches Kommunikationsmodell. Wie auch *TwoPartnerSimplex* verwendet es den *FindAlternatingCommunicationsInTwoGroups*-

Operator, jedoch mit andere Predicates.

In diesem Modell werden die Äquivalenzklassen wie folgt gebildet: Der erste Event dient als Repräsentant. Alle nachfolgenden Events werden solange dieser Klasse zugeteilt, bis ihr zeitlicher Abstand zum Repräsentant zu groß ist. Der erste nicht mehr einsortierte Event wird Repräsentant der nächsten Äquivalenzklasse. Die Events der Äquivalenzklassen werden nicht weiter aussortiert, sondern werden gleich mit Klassen aus der zweiten EventGroup verglichen. Hier müssen die Events eine Mindestlänge besitzen, dürfen aber eine maximale Dauer nicht überschreiten. Sind diese beiden Kriterien erfüllt, wird geprüft, wie weit sich die beiden Emissionen überlappen. Wird der Grenzwert nicht überschritten, gehören die Events zur Kommunikation.

5.2.3 MultiplePartnersMultiplexWithTimeClassesOffset

Bei diesem Modell handelt es sich um ein aggregiertes Kommunikationsmodel. Es basiert auf dem *MultiplexPairsWithTimeOffset*. Die Ergebnisse dieses einfachen Modells werden mit dem *AggregateCommunicationOfOneSubmodel*-Operator ausgewertet.
Dabei hält das Modell neben den Predicates für den *AggregateCommunicationOfOneSubmodel*-Operator auch Parameter zur Steuerung des Submodells *MultiplexPairsWithTimeOffset* bereit.

Das komplexe Modell nutzt den *AggregateCommunicationOfOneSubmodel*-Operator nur gering. Es werden weder Kommunikationen im Vorfeld aussortiert noch werden Äquivalenzklassen gebildet. Alle Kommunikationen fallen in die gleiche Äquivalenzklasse und werden dort auf folgende Punkte überprüft:

- zunächst wird überprüft, dass die beiden Kommunikationen nicht gleich sind
- dann wird überprüft, ob die erste Kommunikation eine bestimmte Dauer nicht überschreitet.
- im dritten Schritt wird getestet, ob entweder der letzte Event der ersten Kommunikation auch der erste Event der zweiten Kommunikation ist oder ob die beiden Startzeiten der Kommunikationen keinen zu großen Abstand haben und ob eine maximale Frequenz- und EventGroup-Anzahl zusammen nicht überschritten wird.

Erfüllen die Kommunikationen alle Anforderungen sind sie Teil der Lösung dieses komplexen Modells.

5.3 Testen des Prototyps

Wie jedes Stück Software, muss auch dieser Prototyp getestet werden. Doch zuerst müssen die Fragen geklärt werden, was getestet werden soll und welche Aussagen aus den Testergebnissen gewonnen werden können.

Was soll getestet werden? Ziel des Systemtests ist es, Fehler im Prototyp zu finden und zu beseitigen. Dafür müssen die einzelnen Komponenten des Prototypen im Einzelnen als auch ihr Zusammenspiel im Ganzen funktionieren.
Das Problem beim Testen der CommunicationOperators ist, dass diese ohne eingebundene Predicates nicht funktionsfähig sind. Daher kann die Funktionsweise der Operatoren nur im Zusammenhang eines Kommunikationsmodells getestet werden. Daher wird getestet, ob der Prototyp ausgewählte Modelle korrekt erkennt. Diese sind:

- TwoPartnerSimplex
- MultiplePartnersMultiplexWithTimeClassesOffset

Hierfür wird in zwei Durchgängen getestet.
Zunächst werden die Modelle beziehungsweise das korrekte Erkennen der Daten, die diesem Modell entsprechen an Hand eigener Testfälle überprüft. Im zweiten Schritt sollen dann die Modelle des Prototypen mit Ergebnissen der Offline-Analyse verglichen werden. Für diesen Test, werden die Daten aus einer Datenbank mit Testdaten entnommen.

Aussagekraft der Testfälle Mit den selbst erstellten Testfällen kann die korrekte Funktionsweise der Operatoren und des Prototyps nachgewiesen werden, da für diese Fälle bekannt ist, welche Events und Kommunikationen gefunden werden müssen, die dem Modell entsprechen.

Für den Vergleichstest mit den Ergebnissen der Offline-Analyse verhält sich die Interpretation der Ergebnisse wesentlich schwieriger. Die zu Grunde liegenden Testdaten sind zu groß, als dass sie von Hand evaluiert werden könnten, um die Ergebnisse des Prototypen zu bestätigen. Beide Analyse-Verfahren wurden getestet, bevor sie auf den Daten der Datenbank gearbeitet haben. Es kann aber keine hundert prozentige Aussage darüber getroffen werden, ob eines der beiden Analyse-Verfahren korrekt arbeitet. Sollten Unterschiede oder Abweichungen in den Auswertungen der Daten durch die verschiedenen Verfahren auftreten, müssen diese einzeln Überprüft werden um überhaupt eine Aussage treffen zu können.

5.3.1 Test mit eigenen Testfällen

TwoPartnerSimplex Um dieses Modell zu testen werden von einem Thread Events generiert, wie sie in Abbildung 41 zu sehen sind. Die Parameter für dieses Modell sind:

- delay = 2000

Für die Events bedeutet das, dass der zeitliche Abstand zwischen zwei Events nicht größer als 2 Sekunden sein darf, damit sie in der Äquivalenzklasse bleiben können (InGroupRestrictionPredicate). Der Abstand zwischen zwei Events aus den beiden EventGroups darf ebenfalls nicht größer als 2 Sekunden sein, damit die Events zur Kommunikation gehören (BetweenGroupRestrictionPredicate).

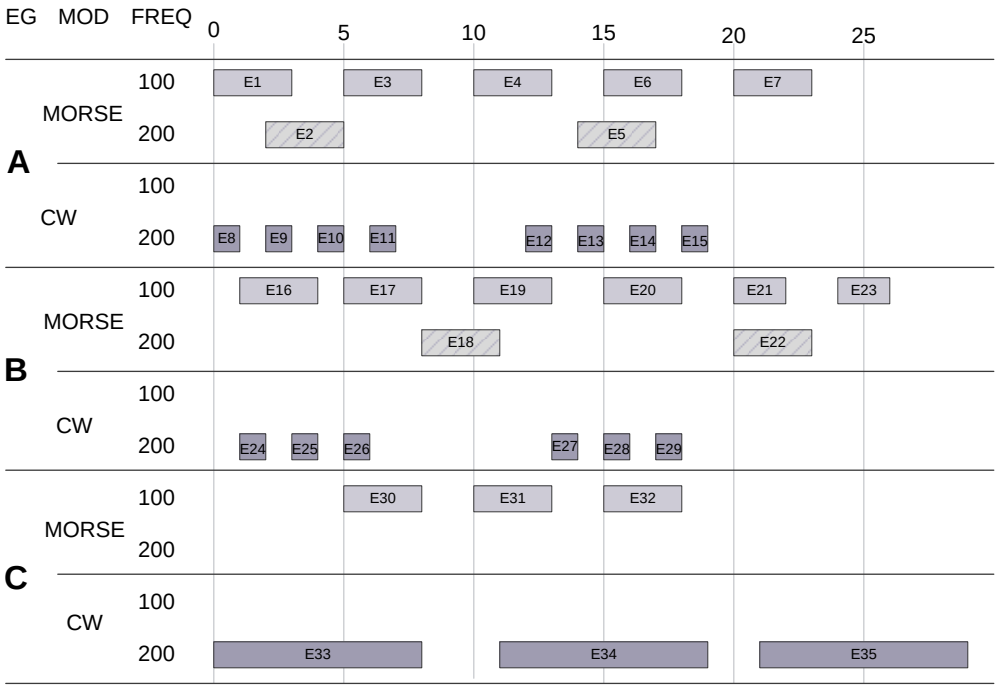

Abbildung 41: Testdaten für das TwoPartnerSimplex-Modell.

Äquivalenzklassen entstehen pro EventGroup (A,B, C) und dort pro Modulation und Frequent. Jeder Zeile ist also eine EventGroup.
Da der Delay-Wert für beide Predicates gleich ist, überlagern sich in der Kommunikation die Events beider EventGroups. Folgende Kommunikationen werden gefunden:

- E1, E15, E16, E3, E4, E19, E7, E21, E23
- E16, E17, E30, E31, E19, E20, E32, E21
- E1, E3, E30, E31, E4, E6, E32, E7
- E12, E27, E13, E28, E14, E26, E15
- E8, E24, E9, E25, E10, E26, E11

Für die Testdurchläufe mit diesen Events wurden die Events beim Senden nicht komprimiert. Das heißt, der in Abbildung 41 dargestellte Abstand zwischen den Events wurde nicht verkürzt.

a) Verpassen des ersten Events einer Kommunikation Bei den ersten Durchläufen wurde festgestellt, dass immer der erste Event einer Kommunikation verpasst wurde, wenn dieser auch der erste Event einer Äquivalenzklasse ist. Die Ursache des

Fehlers wurde beim Kombinieren der Äquivalenzklassen lokalisiert: Wird eine neue Äquivalenzklasse erzeugt, so wird versucht, eine passende Klasse in der zweiten EventGroup zu finden. Existiert noch keine passende Klasse, wird der Event auf den Stream seiner Äquivalenzklasse gesendet, ohne dass eine Statement später diesen Event noch einmal aufgreifen kann. Wird nun im zweiten Stream die passende Äquivalenzklasse erzeugt, so wird die Klasse aus dem ersten Stream gefunden und ein Join-Statement wird platziert. Erst jetzt wird der Event aus der Äquivalenzklasse gesendet und daher auch von dem Join-Statement aufgegriffen.
Als Lösung wird nun zu jeder Äquivalenzklasse die letzten 5 Events mitgespeichert. Werden zwei Äquivalenzklassen kombiniert, können diese letzten 5 Event erneut versendet werden und sind somit für das Join-Statement nicht verloren.
Dies ist natürlich keine optimale Lösung, denn je nach Modell muss diese Zahl angepasst werden. Doch für den Prototyp ist die Lösung ausreichend.

b) Finished-Timer-Problem bei TwoPartnerSimplex Werden die Events nach Endzeit sortiert abgesendet und der letzte Event für eine Communication sehr lange ist, kann es passieren, dass die Zeit, die eine Communication wartet bis sie in den Zustand “finished” wechselt, zu klein ist und dadurch zusammenhängende Kommunikationen zerreißt. Andererseits kann es passieren, dass die Zeit zu groß eingestellt wird und zwei getrennte Kommunikationen auf den gleichen Äquivalenzklassen zusammenfallen.
Das Problem hängt damit zusammen, dass die einzelnen Events nicht in der Länge beschränkt sind und daher keine Maximalzeit berechnet werden kann.

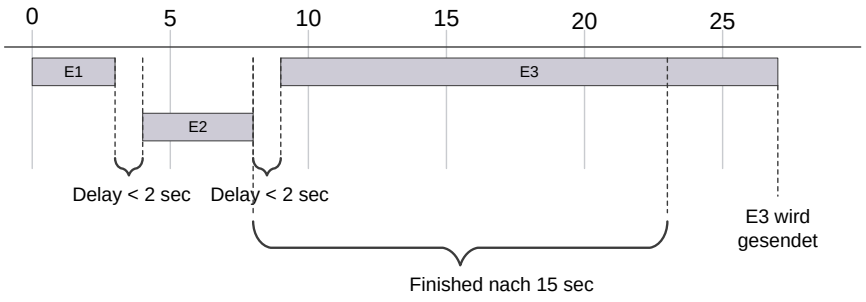

Abbildung 42: Finished-Timer-Problem: Wann kann eine Kommunikation beendet werden.

c) Verpassen einzelner Events Nach dem ein Puffer bei der Kombination von Äquivalenzklassen eingebaut wurde, wurden die Tests erneut durchgeführt. Nun wurden fast alle Events erkannt und den Kommunikationen zugeordnet. Jedoch wurde bei fast jedem Durchlauf ein oder zwei Events verpasst. Zwar sind es immer die gleichen Events, die nicht erkannt werden, aber mal fehlen zwei Events und mal nur einer oder kein Event.

Trotz längerer Untersuchung konnte die Ursache hierfür nicht festgestellt werden. Es wird vermutet, dass das Threading beim Versenden der Events eventuell daran Schuld sein könnte. Da der Thread zwischen dem Versenden zweier Events pausiert wird. Die Dauer der Pause ist jedoch nur als Mindestwert zu verstehen, so dass eine Pause auch länger dauern kann und somit das Timing des Events zu spät ist und er verpasst wird.

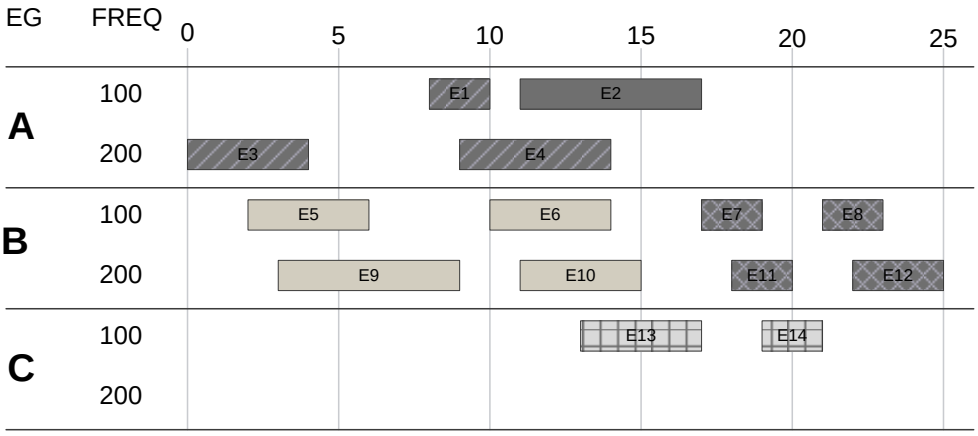

Abbildung 43: Testdaten für das MultiplePartnersMultiplexWithTimeClassesOffset-Modell.

MultiplePartnersMultiplexWithTimeClassesOffset Für dieses Modell wird zunächst das MultiplexPairsWithTimeOffset-Modell ausgewertet. Hierfür müssen einige Parameter vom aggregierten Modell an das einfache Modell durchgereicht werden. Die Parameter des aggregierten Modells sind wie folgt:

- multiDay = 20000
- delay = 2000
- maxDuration = 20000
- minimumEventLength = 1000
- maxNumberOfPartners = 2
- maxNumberOfFrequencies = 2

Das MultiplexPairsWithTimeOffset-Modell wird instanziert mit

- delay = delay
- minEventLength = minimumEventLength

- maxEventLength = maxDuration

Die Events werden wie in Abbildung 43 dargestellt in Äquivalenzklassen eingeteilt. Dabei dienen E3, E2, E5, E13 und E7 als Repräsentant. Da es kein InGroupRestrictionPredicate gibt werden die Events nur mit dem BetweenGroupRestrictionPredicate gefiltert, so dass folgende Kommunikationen gebildet werden:

- E13, E7, E11, E14, E8 (Kommunikation 1)
- E2, E7 (Kommunikation 2)
- E3, E5, E9, E4, E6 (Kommunikation 3)
- E2, E13 (Kommunikation 4)
- E6, E2 (Kommunikation 5)

Diese werden als Ergebnis des MultiplexPairsWithTimeOffset-Modell an das MultiplePartnersMultiplexWithTimeClassesOffset-Modell weitergereicht, wo sie weiter untersucht werden.
Das Modell besteht nur aus einem CommunicationRestrictionPredicate und findet zwei komplexe Kommunikationen in den Ergebnissen, bestehend aus den Events:

- E6, E2, E7 (Kommunikation 2 und Kommunikation 5)
- E6, E2, E13 (Kommunikation 4 und Kommunikation 5)

5.3.2 Vergleichstest mit Testfällen der Offline-Analyse

In diesem Test-Schritt werden existierende Analyse-Ergebnisse aus der Offline-Analyse der Fa. Plath GmbH mit den Ergebnissen des Prototyps verglichen.

TwoPartnerSimplex Um überhaupt Testdaten miteinander vergleichen zu können, mussten entsprechende Kommunikationen in den Ergebnissen der Offline-Analyse gefunden werden. Da der Prototyp sehr einfach gehalten ist kann er nur Event-Attribute als Gleich erkennen, wenn diese auch exakt gleich sind. Bei Funkemissionen kann es aber immer wieder zu Ungenauigkeiten bezüglich der Peilung und der erkannten Frequenz kommen. Da das TwoPartnerSimplex-Modell Äquivalenzklassen auf gleichen Frequenzen aufbaut, mussten Testdaten gefunden werden, in denen die gefundenen Events die gleiche Frequenz haben.
Um einen möglichst aussagekräftigen Test zu erstellen, wurde auf die Generierung eigener EventGroups verzichtet und aus der Datenbank direkt die Cluster der Offline-Analyse übernommen.

Fünf, mit der Offline-Analyse gefundene Kommunikationen entsprechen den eben genannten Anforderungen und wurden ebenfalls mit dem Prototypen evaluiert. Dabei wurden in

einem Fall die gleichen Events gefunden. In einem zweiten Fall wurden neben den drei Events, welche auch im Ergebnis der Offline-Analyse auftauchen, ein weiterer Event gefunden und in den drei anderen Fällen, konnte leider nichts gefunden werden.

MultiplePartnersMultiplexWithTimeClassesOffset Für dieses Modell wurden aus der Testdatenbank alle Ergebnisse selektiert, die dem MultiplePartnersMultiplexWithTimeClassesOffset zugeordnet sind. Für jede dort aufgelistete Kommunikation wurden jeweils zwei Events als Teil des Ergebnisses aufgelistet.
Das sind allerdings zu wenig Events, um das Modell zu erfüllen. Denn jede einfache Kommunikation aus dem MultiplexPairsWithTimeOffset-Modell besteht aus mindestens zwei Events. Diese Kommunikationen werden im aggregierten Modell zusammengeführt und dürfen sich maximal im ersten beziehungsweise dem letzten Event einer Kommunikation überlappen, so dass eine Lösung des MultiplePartnersMultiplexWithTimeClassesOffset-Modells mindestens drei Events beinhalten muss.

Dennoch wurden willkürlich 3 Kommunikationen ausgewählt und ihre Ergebnisse mit dem Prototypen verglichen. Der Prototyp konnte in keinem einzigen Fall eine einfache und dem entsprechend auch keine aggregierte Kommunikation finden.

5.3.3 Auswertung der Testergebnisse

Die Tests mit den eigenen TestDaten haben gezeigt, dass der Prototyp prinzipiell in der Lage ist, die beiden umgesetzten Modelle auszuwerten. Der Vergleich mit der Offline-Analyse zeigt, dass es sehr schwierig ist, die vollständige Auswertung eines Modells bei großer Datenmenge nachzuweisen. Wie auch beim Testen der Software gilt hier, dass das Nicht-Finden einer Kommunikation nicht bedeutet, dass sie nicht in den Daten enthalten ist. Dies zeigt vor allem auch der Vergleichsfall, bei dem ein Event mehr, als in der Offline-Analyse gefunden wurde.

6 Fazit und Ausblick

Diese Kapitel fasst die Arbeit rückblickend zusammen und hebt wichtige Punkte noch einmal hervor. Anschließend wird ein Ausblick gegeben, wie mit den Erkenntnissen dieser Arbeit weiter umgegangen werden kann und welche Möglichkeiten sich für die Zukunft ergeben.

6.1 Fazit

In dieser Arbeit wurde untersucht, ob es möglich ist die Kommunikationsmodellierungssprache ModoCom der Fa. Plath GmbH mit Mitteln des Complex Event Stream Processing umzusetzen. Dafür wurde untersucht, welche Sprachmittel in ModoCom existieren und welche Konzepte damit realisiert werden. Diese Konzepte wurden in den CEP-Bereich übertragen und Teile davon in einem Prototyp, basierend auf ESPER, realisiert. Für diese ModoCom-Umsetzung wurde der Begriff *Online-Analyse* geprägt. Für eine bereits existierende ModoCom-Umsetzung mit LISP wurde der Begriff *Offline-Analyse* verwendet.

Der Begriff *Online-Analyse* soll ausdrücken, dass bei dem, in dieser Arbeit gewählte Ansatz, die Analyse der Daten direkt während ihrem Auftreten stattfinden soll. Also noch während die gesuchte Kommunikation stattfindet.
Basierend auf diesem Verständnis wurde geprüft, in wie weit diese Vorstellung umgesetzt werden kann. Es wurden zwei mögliche Formen der Realisierung beschrieben:

happening Communication Bei dieser Form werden die eintreffenden Daten untersucht und gefundene Ergebnisse direkt an interessierte Komponenten weitergeleitet. Diese Komponenten führen ihrerseits sofort weitere Untersuchungen auf den Daten durch. Dabei wird in Kauf genommen, dass begonnene Auswertungen gegebenenfalls wieder abgebrochen werden müssen, sollte sich im Laufe der vorgelagerten Auswertung herausstellen, dass die Daten nicht mehr valide sind.
Der Vorteil dieser Form ist, dass zeitnah Ergebnisse präsentiert werden können. Die Nachteile hierbei sind, dass Ergebnisse eventuell revidiert werden müssen und gleichzeitig ein höherer Verwaltungsaufwand für die Auswertungen betrieben werden muss.

finished Communication Hier werden Ergebnisse erst an andere Komponenten weitergeleitet, wenn die Kommunikation als abgeschlossen gilt und damit valide ist. Dies hat den Vorteil, dass der Verwaltungsaufwand für die einzelnen Analyse-Schritte sinkt. Der Nachteil ist die verzögerte Weiterleitung der Ergebnisse.

Bei der Untersuchung der Sprachmöglichkeiten von ModoCom musste festgestellt werden, dass ModoCom noch an einigen Stellen erweitert werden muss, damit es generisch mit Events und Kommunikationsformen umgehen kann. So ist es beispielsweise mit ModoCom aktuell nicht möglich zu Beschreiben, wie sich die Werte der Kommunikations-Attribute zusammensetzen. Ebenso ist die Generierung von EventGroups derzeit noch nicht Teil der

Sprache.
Die Tatsache, dass aktuell nur ein Submodell zu einem komplexeren Modell aggregiert werden kann liegt an dem Operator und ist nicht durch ModoCom selbst limitiert. Ein Anwendungsfall für die Zukunft wird sein, zwei verschiedene Modelle in einem komplexen Modell zusammenzuführen, um so Kommunikation über verschiedene Medien finden zu können.
Darüber hinaus konnten vorhandene Konzepte wie Event, EventGroup, CommunicationOperator und CommunicationModel in den CEP-Bereich abgebildet werden. Die Abbildung von Events ist trivial. Mengen von Events (EventGroups und Communications) werden als Streams realisiert. Dabei werden für einzelnen Streams, Objekte mit Meta-Informationen vorgehalten (bspw. Communications) um daraus Events für den jeweiligen Stream zu generieren.
Nicht abgebildet wurden *StrategyObjects*[8], da diese aus der Offline-Analyse mit LISP heraus motiviert sind und für eine Realisierung im CEP-Bereich keine Bedeutung haben.

Für den Prototyp wurden zwei CommunicationOperator, wie sie für die Offline-Analyse existieren, nachgebaut, um darauf aufbauend zwei einfache und aggregiertes Kommunikationsmodell zu realisieren.
Abschließend wurde der entwickelte Prototyp getestet und mit der Offline-Analyse verglichen.

Mit der Fertigstellung des Prototyps wurden alle in Kapitel 1.2 formulierten Ziele erreicht. Tabelle 1 auf Seite 76 gibt einen zusammenfassenden Überblick über die einzelnen Komponenten und ihre Realisierung.

6.2 Ausblick

Die Arbeit hat gezeigt, dass es möglich ist, ModoCom mit Complex Event Stream Processing zu realisieren. ModoCom noch nicht ausgereift und muss an einigen Stellen speziell an die Bedürfnisse für eine Realisierung mit CEP-Mitteln angepasst werden (vgl. StrategyObjects der Offline-Analyse).

Es bleibt zu klären, ob die algorithmische Umsetzung eines Operators der Offline-Analyse auch geeignet für den Einsatz in CEP ist, oder ob es bessere und einfachere Abläufe in CEP gibt. In einem ersten Versuch konnten die Modelle teilweise innerhalb eines einzelnen EPL-Statements spezifiziert werden. Diese Statements sind dann komplexer, als die bisherigen Predicates, haben allerdings den Vorteil, dass die CEP-Engine die Abfragen optimieren kann. Die Operatoren der Offline-Analyse hingegen haben bereits eine Abstraktion erfahren, so dass sie allgemeiner verwendet werden können.

Auch der Vergleich des Prototypen mit der Offline-Analyse hat gezeigt, dass auch hier noch einige Punkte zu klären sind und einer genaueren Untersuchung bedürfen.

Ein weiteres noch offenes Problem, ist das Problem der Messungenauigkeit. ModoCom

arbeitet mit exakten Werten, die von den Sensoren oft nicht geliefert werden können. Für die Offline-Analyse wurde bereits eine Lösung implementiert, die damit umgehen kann. Für die Online-Analyse bleibt zu klären wie eine solche Lösung aussehen kann. Gegebenenfalls ist auch eine Lösung in ModoCom selbst realisierbar. Auch dies muss untersucht werden.
Die vielen Erweiterungen und Anpassungen für eine Realisierung mit CEP legen eine Um- beziehungsweise Restrukturierung von ModoCom nahe, bei der ein ModoCom Kern herausgearbeitet wird und abhängig vom realisierenden System, zusätzliche Sprachmittel realisiert werden.

Eine weitere Funktion von ModoCom könnte sein, dass statt nur zu Erkennen, ob eine Menge von Events ein Modell erfüllen oder nicht, dem Anwender auch mitzuteilen, welche Events fehlen, damit ein Modell erfüllt wäre und diese Füll-Events zu generieren. Dadurch könnten auch Kommunikationen erkannt werden, die beispielsweise auf Grund von Übertragungsstörungen nicht hätten erkannt werden könne.

Analyseverfahren	ModoCom	LISP	CEP	Prototyp	Bemerkungen
Offline		ja	ja	ja	
Online		nein	ja		
- Happening Communication		nein	ja	nein	
- Finished Communication		nein	ja	ja	
ObjectTypes	ja	ja	ja	ja	nicht von ModoCom definiert, nur dort verwendet
- Event	ja	ja	ja	ja	
- EventGroup	ja	ja	ja	ja	als Stream
- Communication	ja	ja	ja	ja	als Stream und abstrakter Datentyp mit Meta-Informationen
- Sorting	ja	ja	ja	nein	aber nur eine bestimmte Menge (Windwo)
- Inherit	ja	ja	ja	nein	
- Setzen der Attribute	-		-		Datentyp weiß, wie Attribute gebildet werden
Predicate	ja	ja	ja	ja	In der Where-Klausel eines EPL-Statments oder als Java-Methode
CommunicationOperator	nein	ja	ja	ja	werden im System definiert
- FindAltComInTwoGroups	nein	ja	ja	ja	
- AggComOfOneSubModel	nein	ja	ja	ja	
- FindComInOneGroup	nein	ja	nein	nein	
StrategyObjects	ja	ja	nein	nein	sind zur Optimierung für Constraint Systeme
CommunicationModel	ja	ja	ja	ja	
- TwoPartnerSimplex		ja	ja	ja	
- MultiplePairsTimeOffset		ja	ja	ja	
- MultPartMultWithTimeOff		ja	ja	ja	
ModoCom-Programm	ja	ja	ja	ja	
Aggregation von Com-Models	ja	ja	ja	ja	kann durch das Hinzufügen weiterer Operatoren auch auf mehrere Submodelle erweitert werden
EventGroup-Generierung	nein		ja	ja	aber nur sehr einfach im Prototyp
Sensor-Anbindung	nein	nein	ja	ja	im Prototyp auch nur ein DB angeschlossen

Tabelle 1: Überblick

Literatur

[1] Anonymus. Pattern matching in sequences of rows (11). http://dist.codehaus.org/esper//row-pattern-recogniton-11-public.pdf, März 2007.

[2] Pradeep K. Atrey. A hierarchical model for representation of events in multimedia observation systems. In *EiMM '09: Proceedings of the 1st ACM international workshop on Events in multimedia*, pages 57–64, New York, NY, USA, 2009. ACM.

[3] Inc Coral8. Complex Event Processing: Ten Design Patterns, 2006.

[4] Michael Eckert and Francois Bry. Complex Event Processing. *Informatik-Spektrum*, 32(2):1633–167, März 2009.

[5] Jens Ellenberg. Event Stream Processing mit ESPER unter Einsatz von Datamining Verfahren. Bachelorarbeit, Hochschule für Angewandte Wissenschaften Hamburg, 2009.

[6] Esper Inc. Esper Architektur. http://esper.codehaus.org.

[7] Matthias Haringer, Lothar Hotz, and Vera Kamp. Two stage knowledge discovery for spatio-temporal radio-emission data. In *ECAI*, pages 673–677, 2008.

[8] Lothar Hotz and Stephanie Knab. ModoCom - Language Modeling of Communications, Dezember 2009.

[9] EsperTech Inc. Esper Reference Documentation V-3.4.0, März 2010.

[10] EsperTech Inc. EsperIO Reference Documentation V-3.4.0, März 2010.

[11] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

Index

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 5. Juli 2010	
Ort, Datum	Unterschrift