Fachhochschule
Hannover

University of Applied Sciences and Arts

Fakultät IV – Wirtschaft und Informatik

Evaluierung der CEP Engine Drools anhand Esper und einer Fallstudie

Bachelorarbeit im Studiengang Angewandte Informatik der Fachhochschule Hannover

Jan Naumann

August 2010

Autor	
Name:	Jan Naumann
Adresse:	Schmiedeweg 3 32699 Extertal
Email:	jan-naumann@gmx.de

Prüfer	
Name:	Ralf Bruns
Adresse:	FH Hannover Fakultät IV Ricklinger Stadtweg 120 30459 Hannover
Email:	ralf.bruns@fh-hannover.de

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die eingereichte Bachelorarbeit selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum Unterschrift

Inhaltsverzeichnis

1. Einführung in diese Bachelorarbeit....4
1.1 Ziele und Inhalt der Arbeit....4
1.2 Abgrenzungen dieser Arbeit....4
1.3 Motivation....4
2. Einführung in CEP und Ereignisbasierte Architekturen....6
2.1 Ereigniserläuterung....6
2.2 Was Komplexe Ereignisse sind....7
2.3 Ereignishierachie....7
2.4 Warum Ereignisbasierte Systeme bauen?....8
3. Anwendungsszenario....9
3.1 Das Modell....9
4. Open-Source CEP-Engines Drools und Esper....13
4.1 Drools....13
4.2 Anwendung von Drools auf Anwendungsszenario....16
4.3 Esper....26
4.4 Anwendung von Esper auf Anwendungsszenario....26
5. Vergleich der Ereignisverarbeitenden Sprachen und ihren Engines....31
5.1 Sprachparadigma....31
5.2 Ereignisdarstellung....33
5.2.1 Definition und Relation von Ereignissen....33
5.2.2 Ereignisherkunft....35
5.2.3 Zeitkonzept....37
5.3 Sprachumfang....40
5.3.1 Verknüpfungsoperatoren....40
5.3.2 Temporale Operatoren....41
5.3.3 Sequenzoperatoren....43
5.3.4 Kausale Operatoren....44
5.3.5 Fenster....45
5.3.6 Ereignisaggregation....49
5.4 Erweiterter Sprachumfang und Engine Eigenschaften....50
5.4.1 Absenz von Ereignissen....50
5.4.2 Konsumierung von Ereignissen und Garbage Collecting....52
5.4.3 Content-Enrichment....54
5.4.4 In-Line Datenmodifikation....56
5.4.5 Performance....56
5.5 Kriterien zur Arbeit mit den Plattformen....58
5.5.1 Installation und Integration in Entwicklungsumgebungen....58
5.5.2 Support und Dokumentation der Plattform....59
5.6 Zusammenfassende Gegenüberstellung....61
5.7 Nicht betrachtete Kriterien....63
6. Fazit und Schlussbemerkungen....64

1. Einführung in diese Bachelorarbeit

1.1 Ziele und Inhalt der Arbeit

Das Ziel dieser Bachelorarbeit ist eine Evaluation der Open-Source ereignisverarbeitenden Sprache Drools. Hierbei soll Wert gelegt werden sowohl auf die Mächtigkeit der Sprache als auch die Fähigkeiten und Funktionen der Engine, sowie zuguterletzt die Nutzbarkeit der angebotenen Funktionen zur Anwendungsimplementierung. Zur Evaluation der Engine soll dabei ein möglichst direkter Vergleich mit der ereignisverarbeitenden Sprache/Engine Esper durchgeführt werden. Dazu wurde eine Implementation von einem vorgegebenen Anwendungsfall durchgeführt. Die Vergleichskriterien die zur Evaluation genutzt werden wurden dabei anhand von Anforderungen aus der Industrie sowie wissenschaftlichen Methoden erarbeitet und sollen im Zuge dieser Studienarbeit vorgestellt werden. Die Fähigkeiten der Sprachen/Engines sollen auf diese dann folgend abgeglichen werden. Ziel der Arbeit ist eine Empfehlung für oder gegen die Verwendung von Drools im Bereich der Ereignisverarbeitenden Systeme auszusprechen, sowie einen Einblick in die Verwendung beider Sprachen, anhand des Anwendungsbeispiels, zu geben.

Die Arbeit soll dabei zuerst einen kurzen Einblick auf die Grundlagen im Bereich Complex Event Processing oder kurz CEP gewähren und eine Übersicht über den derzeitigen wissenschaftlichen Stand in jenem Bereich bieten. Danach soll eine kurze Übersicht über die beiden Sprachen Esper und Drools sowie ihre Hintergründe gegeben werden. Zuguterletzt soll der konkrete Vergleich erfolgen und eine Wertung stattfinden.

Diese Arbeit richtet sich dabei an eine Personengruppe, die bereits grundlegende Kenntnisse mit dem objektorientierten Programmierparadigma besitzen. Beide betrachteten Plattformen verwenden als Basisplattform Java, weswegen zumindest Grundlagen der Sprache beherrscht werden sollten.

Für diese Arbeit wurde die Drools Version 5.0 sowie Esper Version 3.4 benutzt.

1.2 Abgrenzungen dieser Arbeit

Der in dieser Arbeit referenzierte Anwendungsfall soll lediglich dem Zwecke dienlich sein die Evaluation der Sprachen/Engines zu ermöglichen. Der Anwendungsfall wird zwar in Folge dieser Arbeit vorgestellt, der Autor nimmt sich aber die Freiheit im Zuge des Vergleichs von dem implementierten Beispiel abzuweichen, oder weitere Beispiele einzuführen sollte dies im Sinne der Evaluation nützlich sein, bzw. Falls der Anwendungsfall nicht ausreichend abdeckend ist um eine korrekte Evaluation zu ermöglichen. Die Implementation der Esper Beispiele wurde bereits in [1] durchgeführt und der Autor wird wann immer möglich auf diese Implementierung referenzieren. Schwerpunkt dieser Arbeit ist dabei eindeutig die Engine Drools, wobei Esper als Vergleichender Aspekt gezeigt werden soll. Im Zuge dieser Bachelorarbeit können nicht alle Aspekte der beiden Sprachen abgedeckt werden und die Vorstellung von allen Funktionen würde den Rahmen dieser Arbeit weit sprengen.

1.3 Motivation

Die Fakultät IV der FH-Hannover führt zurzeit eine Projektgruppe mit dem Thema Event-Driven-Architectures.(http://eda.inform.fh-hannover.de/) Ein Projekt dieser Projektgruppe ist eDraft welches in den Jahren 2008 bis 2010 stattfand, welches mit Hilfe von Ereignisbasierten Systemen ein Verkehrssteuerungssystem realisierte. Ein solches System muss dabei viele kleine Vorkommnisse im Straßenverkehr mit Hilfe von Sensoren wahrnehmen und diese dann korrekt verarbeiten. Dazu müssen kleine Ereignisse, wie die Geschwindigkeit eines Fahrzeugs welches einen Sensor passiert, die in sich selbst genommen unwichtig sind, betrachtet und aus diesen Mengen von Ereignissen Schlüsse gezogen werden. Zur Verarbeitung dieser Ereignisse wurde die Open-Source Event-Engine

"Esper" benutzt. Die Kunst ist dabei, den Anwendungsentwickler mit Hilfe von Sprachkonstrukten und einer asynchronen Verarbeitung von vielen Ereignissen zu unterstützen Muster in Ereignissen zu finden, diese korrekt zu interpretieren und gegebenenfalls auf diese Ereignisse zu reagieren.

Zurzeit gibt es auf dem Markt der Ereignisverarbeitenden Systeme eine Vielzahl an kommerziellen und zum Teil aufgrund dessen auch stark spezialisierten Lösungen für solch einen Anwendungsfall, im Open-Source Bereich haben sich bisher aber nur zwei ernsthafte Bewerber hervorgetan. Zum einen das von EsperTech oben bereits erwähnte Esper, zum anderen das von JBoss angebotene Drools.

Die Fakultät IV der FH-Hannover hat durch eDraft und weitere Projekte große Erfahrung im Umgang mit Esper gesammelt, wohingegen bis jetzt eine tiefer gehende Nutzung von Drools zur Anwendungsentwicklung noch nicht stattfand. Die Idee zu dieser Bachelorarbeit entstand aus diesen Vorraussetzungen, herausgehend aus der Frage, wie reif das noch recht junge Modul "Drools Fusion" zur Ereignisverarbeitung ist, besonders mit Hinblick auf den "Open-Source-Platzhirsch" Esper.

2. Einführung in CEP und Ereignisbasierte Architekturen

Dieser Bereich soll dazu dienen eine Einführung in ereignisbasierte- Systeme sowie Programmierparadigmen zu zeigen. Dieser Bereich dient auch dazu die Bedeutung von später eingeführten Evaluationskriterien zu rechtfertigen.

2.1 Ereigniserläuterung

Ein Ereignis wird in der Informationstechnologie gemeinhin als: "*a significant change in state*"[2] bezeichnet. Zu Deutsch der Wechsel oder zumindest die Aufzeichnung desselben in einem System(-konstrukt). Ereignisse bestehen laut Luckham aus [3] :

- Form eines Ereignisses: Die spezifische Ausprägung eines Ereignisses, beispielsweise als Java Objekt welches Daten wie zum Beispiel Zeichenketten oder Gleitkommazahlen beinhaltet. Die Form referenziert dabei also auf so etwas wie die "physikalische" Existenz eines Ereignisses.
- Signifikanz eines Ereignisses: Die fachliche Beschreibung was ein Ereignis eigentlich ausmacht, bzw. welche Aktivität es darstellt. Die Signifikanz eines Ereignisses muss dabei aus Daten seiner Form beschrieben werden.
- Relativität eines Ereignisses: Ereignisse besitzen immer relative Bezüge zueinander, so kann ein Ereignis vor oder nach einem anderen stattfinden, ein Ereignis kann hierbei von einem anderen Ereignis verursachend abhängig sein und ein Ereignis kann aus einer Menge von beinhaltenden Ereignissen bestehen. Luckham nennt für diese Zusammenhänge die Begriffe: Time,Causality und Aggregation.Die Relativität beschreibt diese Zusammenhänge.

Wir leiten also aus obigen Erkenntnissen nun zwei signifikante Aussagen her:

- Die Form eines Ereignisses muss fähig sein die Signifikanz, sprich die Aktivität eines Ereignisses korrekt abzubilden.(Eine korrekte Analogie zwischen Form und Signifikanz ist die eines Objekts und einer Klasse.) Eine Wirtssprache wie Java oder C++ übernimmt diesen Arbeitsschritt dabei schon weitestgehend.
- Die Beobachtung der Relativität von Ereignissen erlaubt uns eine klarere Kategorisierung von selbigen: Ein Ereignis kann nun aus mehreren Ereignissen bestehen, weswegen wir hier eine Unterscheidung zwischen zwei Typen vornehmen können, einmal ein einfaches Ereignis welches lediglich stattfand und ein aggregiertes Ereignis welches mehrere Ereignisse zusammenfasst.

Ein aggregiertes Ereignis ist dabei bestehend aus mehreren einfachen Ereignissen und hat auch eine Dauer, wobei diese durch das erste und letzte aggregierte Ereignis festgelegt werden kann. (Da ein Ereignis die Beschreibung einer Aktivität ist und solch eine Aktivität auch andauern kann ist es nicht falsch davon auszugehen, dass auch einfache Ereignisse eine Dauer haben, allerdings ist eine Dauer keine Notwendigkeit eines einfachen Ereignisses.) Die Kausalität ist bei aggregierten Ereignissen bestehend aus einfachen natürlich auch zwingend sicher, sowie die Zeit: Einfache Ereignisse geschehen vor Aggregierten.

Ein aggregiertes Ereignis muss dabei nicht nur aus Einzel-Ereignissen bestehen, es kann auch während es erzeugt wird durch umgebendes Wissen aus dem enthaltenen Modell "angereichert" werden. Man spricht hierbei vom "Content-Enrichment".

Einfache Ereignisse entstehen dabei immer aus Beobachtungen, zur Erzeugung eines einfachen Ereignisses muss also im Falle einer Verkehrssteuerung ein Sensor oder im Falle von einer Börsenanwendung eine Kursüberwachungsanwendung Ereignisse aus den Veränderungen des Beobachteten Systems heraus erzeugen.

2.2 Was Komplexe Ereignisse sind

Bis jetzt wurden zwei Typen von Ereignissen vorgestellt, einfache Ereignisse und die aus ihnen aggregierten Ereignisse. Ist gilt nun den dritten Typ von Ereignissen und fachlich auch komplexesten vorzustellen, das so genannte Komplexe Ereignis.

Während aggregierte Ereignisse lediglich eine recht aussagelose Summe von Einzelereignissen sind, die unter Umständen mit Fakten aus dem Modell angereichert wurden, so ist ein komplexes Ereignis ein aus Ereignismustern entstandenes Ereignis, welches die Aktivitäten einer Menge von Ereignissen zusammenfasst. "*Event pattern rules are used in CEP to create complex events signifying the activities of sets of events*."[3]

Die Unterscheidung zwischen einfachen aggregierten Ereignissen und komplexen Ereignissen scheint dabei fließend, generell gilt allerdings, dass komplexe Ereignisse aus zum Teil komplizierten Schlussfolgerungen entstehen, komplexe Ereignisse sind aufgrund dessen auch der schwierigste zu entwerfende Part der sogenannten Ereigniswolke.

Die Ereigniswolke ist dabei die Summe der Ereignisse die in einem definierten System stattfinden.

Ein Beispiel für ein komplexes Ereignis wäre unter anderem die Erkennung eines Unfalls auf einer Verkehrsstrecke. Hierbei muss aus einer Summe von Einzelereignissen, nämlich Passagen von Fahrzeugen über im Boden eingelassene Induktionsspulen, eine logische Schlussfolgerung stattfinden, wie der derzeitige Verkehrsstand aussieht. Ein über die Summe der Einzelereignisse räsonierendes System ist gezwungen über eine grosse Anzahl an Verkehrspassagen die relevanten Daten zu filtern und diese dann in eine Beurteilung der derzeitigen Verkehrssituation umzuwandeln. Die Beurteilung dieser Situation ist dann im Kontext als komplexes Ereignis zu verstehen.

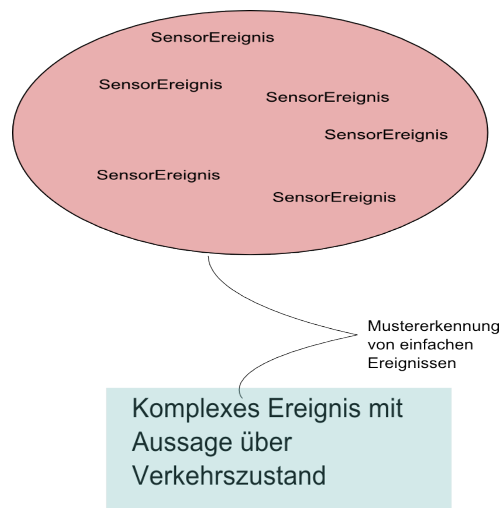

Abbildung 1: Räsonierung über komplexes Ereignis

2.3 Ereignishierachie

Mit den in den vorherigen Abschnitten gewonnenen Erkenntnissen kann eine Hierarchie von Ereignissen gefolgert werden. Die Hierarchie besteht dementsprechend aus drei Stufen:

- **Einfache Ereignisse**, die von Ereignisgeneratoren kreiert werden und eine simple Tätigkeit, Tatsache oder Veränderung in einem System darstellen.
- **Aggregierte Ereignisse**, die mehrere einfache Ereignisse zusammenfassen, im Allgemeinen aber keine tiefere Bedeutung haben aber unter Umständen Informationen beinhalten, die nur aus einer Menge von einfachen Ereignissen entstanden sind, wie beispielsweise

Summenwerte von einzelnen Ereignissen.

- **Komplexe Ereignisse**, die eine komplexe Tatsache oder Erkenntnis darstellen, die aus einzelnen und aggregierten Ereignissen geschlussfolgert wurde. Dazu dürfen nicht nur Ereignisse zusammengefasst werden, sondern Muster in Strömen erkannt werden.

Insgesamt wird die gesamte eben beschriebene Disziplin als "Complex Event Processing" bezeichnet, namentlich, die Erkennung, Erzeugung und Verarbeitung von komplexen Ereignissen aus einem beliebigen Informationsverarbeitenden System heraus.

Die folgende Anwendung soll versuchen, diesen Sachverhalt und ihre Hierarchie abzubilden.

2.4 Warum Ereignisbasierte Systeme bauen?

Fast alle Vorgänge in unserem täglichen Leben basieren auf dem Geschehen von Ereignissen, so ist der Fall eines Aktienkurses ein Ereignis wie auch die Bestellung eines Buches oder der Vorgang das Besteller selbigen Buches eine Mahnung erhält, weil er genanntes Buch schon seit einem Monat nicht bezahlt hat.

Ein Architekturmodell für solche Systemvorgänge ist eine ereignisbasierte Architektur.[4]

In einer Ereignisbasierten Architektur wird auf Ereignisse asynchron reagiert. Die Komponenten dieser Architektur sind dabei sehr lose gekoppelt, eine untergeordnete Komponente weiss dabei weder etwas von übergeordneten Komponenten, noch ist ihm bekannt was die von ihm erzeugten Nachrichten/Ereignisse in höheren Komponenten an Diensten auslösen.

Grob kann solch eine Architektur in drei Komponenten aufgeteilt werden.

- **Ereignisgeneratoren**, die die Rohdaten erzeugen, welche später in der Anwendung betrachtet werden. Beispiele wären der Fall eines Aktienkurses, das Senden eines Netzwerkpakets oder die Änderung des Warenkorbs in einem Online-Shop. Jeder Ereignisgenerator muss dabei Adapter unterstützen, die sich um Weiterleitung von Ereignissen kümmern.
- **Ereignisverarbeitung**, welches die zentrale Komponente einer CEP-Anwendung ist. Diese kümmert sich um das Erkennen von Mustern und komplexen Ereignissen im System und muss mit den Daten der Ereignisgeneratoren versorgt werden. Dabei ist die Ereignisverarbeitung in der Regel selber ein Ereignisgenerator, welcher zirkulär Ereignisse an seinen Ereigniseingang leitet.
- **Ereignisbehandlung,** ist die höchste Komponente in einer ereignisbasierten Architektur. Diese stößt Dienste an, welche bei Erhalten von Ereignissen ausgelöst werden sollen. Beispielhaft die Aktualisierung einer Datenbank, Das Versenden von Nachrichten an nebenliegende Systeme oder der Beginn eines Prozesses.

3. Anwendungsszenario

Hier folgt eine Beschreibung des betrachteten Anwendungsszenarios.

Das betrachtete Anwendungsszenario ist dabei im Modell aus dem Buch "Event Driven Architectures" entnommen und wurde nur falls dies nötig war angepasst. Die Ereignismodellierung, -Generierung und -Verarbeitung wurde im Zug dieser Studienarbeit umgesetzt.[1]

Das Anwendungsszenario ist dabei thematisch einer Verkehrssteuerung entnommen, wobei hier das Schema von einfachen bis komplexen Ereignissen abgedeckt wird. Einfache Ereignisse werden während des Strassenverkehrs erzeugt, worauf im Folgenden diese aggregiert sowie komplexe Ereignisse erzeugt werden.

3.1 Das Modell

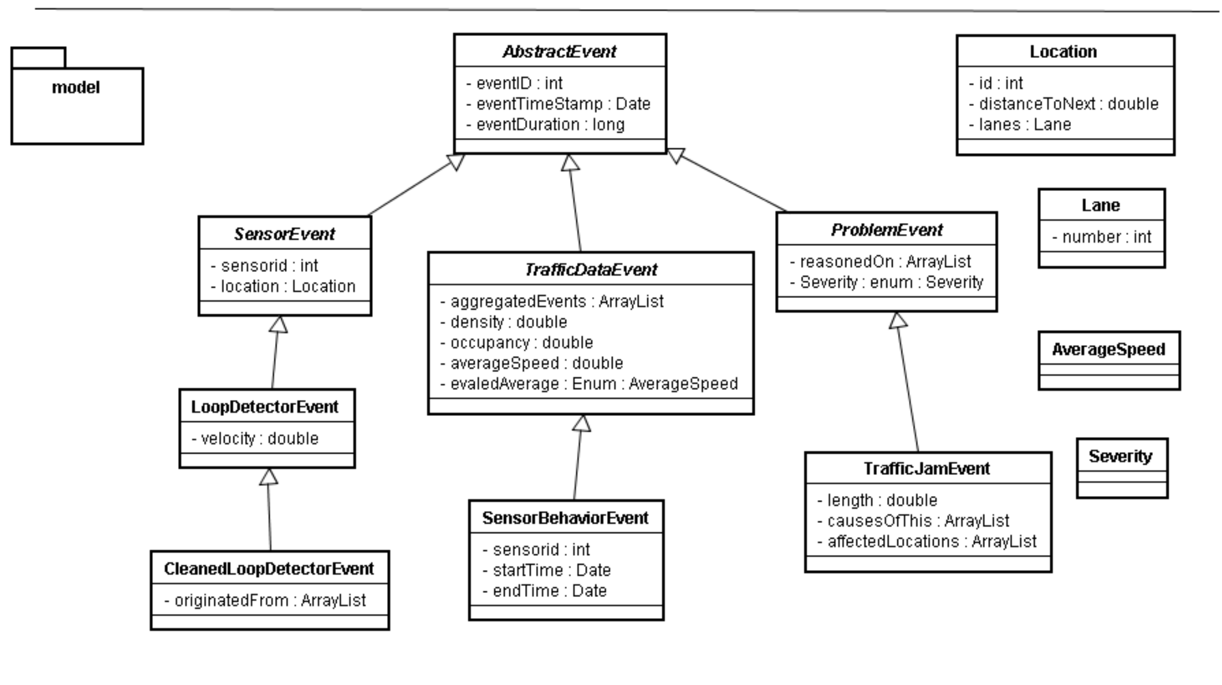

Abbildung 2: Modell

Im Package model befindet sich das Modell, welches sowohl Java-Klassen als Objekte enthält, sowie die Ereignisklassen. Weitere Klassen stellen Enums dar, die zur späteren Ereignisbewertung benutzt werden.

AbstractEvent:

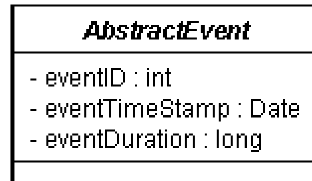

Abbildung 3: AbstractEvent

AbstractEvent ist eine abstrakte Klasse wovon jede weitere Ereignisklasse erbt. Jedes Ereignis besitzt dabei eine ihm einzigartige EventID. Die EventID wird dabei aus der utils Klasse IDGetter bei Erzeugung eines neuen Ereignisses zugewiesen und ist vom Typ Integer. Jedes Ereignis hat weiterhin gemein das es einen Ereignis-Zeitpunkt vom Typ Date hat, an dem das Ereignis geschehen ist, wobei jegliche Räsonierungen in der Event-Engine über diesen Zeitstempel stattfinden soll. Der Zeitstempel wird bei Erzeugung des Ereignisses in Java festgelegt. Ausserdem besitzt jedes Ereignis eine Dauer vom Typ long,welche anzeigt wie lange ein Ereignis als stattfindend deklariert werden soll. Einige Ereignisse sind hierbei lediglich für einen festgelegten Zeitpunkt gültig und haben die Dauer 0, aggregierte oder komplexe Ereignisse haben jedoch eine Zeitdauer die hinweist wie lange die Start- und Endzeitstempel der beinhaltenen Ereignisse, bzw. Im Falle von komplexen Ereignissen wie lange die temporalen Operatoren(Kapitel 5.3.4) der Event-Engines über die Erzeugung eines komplexen Ereignisses sinniert haben.

SensorEvent:

Abbildung 4: SensorEvent

SensorEvent ist eine abstrakte Klasse welches als Grundlage für alle ihm folgenden einfachen Ereignisklassen dienen soll. Die Daten des SensorEvents werden hierbei im Falle dieser Anwendung aus speziell formatierten Textdateien gelesen. In der echten Welt würden diese Ereignisse dabei natürlich aus einem Ereignisgenerator stammen, welcher Nachrichten über von Sensoren an der Fahrbahn erhält. Die sensorid vom Typ Integer soll dem Sensorevent erlauben es eindeutig einem Ursprungssensor zuzuordnen. Die location ist dabei eine Faktenklasse, die spezfische Daten über den Ort des Sensors enthält.

LoopDetectorEvent:

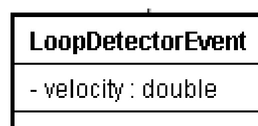

Abbildung 5: LoopDetectorEvent

LoopDetectorEvent ist eine konkrete Klasse welche ein einfaches Ereignis darstellt und von

SensorEvent erbt. Dieses Event soll in der Ereignishierarchie die erste konkrete Ausprägung eines einfachen Ereignisevents sein.Dieses Event beschreibt die Passage eines Fahrzeugs über eine in der Straße eingelassene Induktionsspule, wobei die Induktionsspule die Geschwindigkeit des passierenden Fahrzeugs aufnimmt und in das Attribut velocity vom Typ double einfügt. Die LoopDetectorEvents werden in der Beispielanwendung vom EventReader ausgelesen und dann durch die EventFeederKlasse in den Speicher Ereignisverarbeitenden Engine eingefügt, wo dann über die Ereignisse räsoniert wird.

CleanedLoopDetectorEvent.

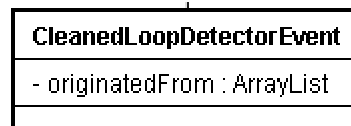

Abbildung 6: CleanedLoopDetectorEvent

Dieses Ereignis ist eine weitere Spezialisierung von LoopDetectorEvent. Wird ein LoopDetectorEvent in den Ereignisstrom hineingeführt, so behandeln die definierten Regeln dieses Ereignis als "nicht sauber", es wird nicht in weitere Ereignisakkumulierungen einbezogen. Dies bedeutet im weiteren Sinne der Anwendung, dass das Ereignis nicht den Anforderungen für Gültigkeit entpricht. Der in Event-Driven Architectures definierte Anwendungsfall verlangt von Ereignissen die eine Übertretung einer Induktionsspule beschreiben, dass sie nicht mehrfach vorkommen können(Es ist sicherlich vorstellbar, dass ein Sensor versehentlich die gleiche Fahrzeugpassage mehrfach feststellt.), sowie dass sie keine fehlerhaften Geschwindigkeitswerte wie eine Geschwindigkeit weniger als 0 oder eine zu hohe Geschwindigkeit haben. Die Logik dieser Prüfung soll natürlich in den Regeln und Statements abgebildet werden.

TrafficDataEvent:

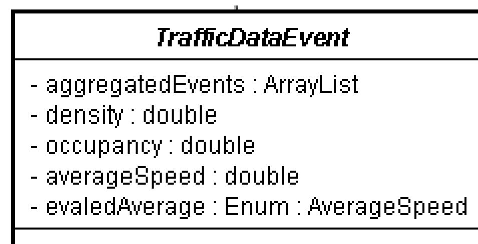

Abbildung 7: TrafficDataEvent

Stellt ein abstraktes Ereignis welches eine Menge von CleanedLoopDetectorEvents aggregiert. Ist damit das oberste Aggregations-Ereignis in der Ereignishierarchie. Es beinhaltet die Ereignisse in einer ArrayList, aus dem das TrafficData Event zusammengefasst wurde. Die beiden double-Werte density und occupancy sollen die Verkehrsstärke sowie Verkehrsdichte angeben, diese Werte können dabei durch Content-Enrichment aus dem Modell stammen. Der double-Wert averageSpeed wird über ein Akkumulation von mehreren CleanedLoopDetectorEvents gesetzt und die ermittelten Daten werden dann an den EventCreator bzw. Dem Listener gegeben, der die TrafficDataEvents schließlich erzeugt. Der Enum evaledAverage soll dabei eine qualitative Bewertung des averageSpeed sein.

SensorBehaviorEvent:

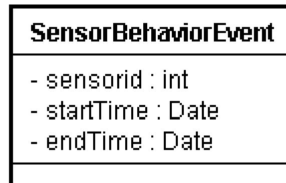

Abbildung 8: SensorBehaviorEvent

Ist eine Spezialisierung des TrafficDataEvents. Das Ereignis stellt dabei eine Beobachtung eines spezifischen Sensors, welcher festgelegt ist über seine SensorID, über einen spezifischen Zeitraum dar. Die Start und Endzeit wird festgelegt über die Zeitstempel der aggregatedEvents und soll dabei zur späteren Erkennung, über welchen Zeitraum die Daten gesammelt wurden dienen. Die sensorID wird zur Erkennung welcher Sensor benutzt wurde verwendet.

ProblemEvent:

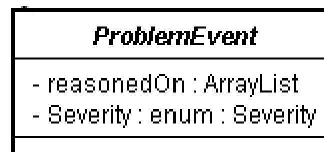

Abbildung 9: ProblemEvent

Ist ein abstraktes Ereignis und in der Ereignishierarchie das erste komplexe Ereignis. Die Erzeugung dieses Ereignisses wird dabei über eine Analyse der Muster von SensorBehaviorEvents in der Drools-Engine angestoßen. Das ProblemEvent ist dabei als komplexes Ereignis einzuordnen, da seine Erzeugung keine einfache Aggregierung von Daten darstellt, sondern eine komplexe Bewertung von TrafficDataEvents. Es enthält die SensorBehaviorEvents über die räsoniert wurde und eine Bewertung über die Brisanz des festgestellten Problems.

TrafficJamEvent:

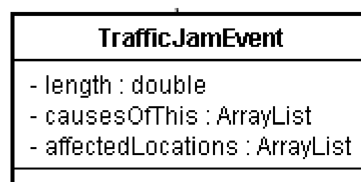

Abbildung 10: TrafficJamEvent

Ist eine Spezialisierung eines ProblemEvents, welches die Beobachtung eines Staus darstellt. Hierbei wird die Länge des Staus aus den beobachteten Sensoren, die in den aggregierten TrafficDataEvents enthalten sind sowie ihren Locations berechnet. Das TrafficJamEvent hat weiterhin eine Liste von Gründen aus denen der Stau entstanden ist sowie eine Liste von Locations die durch den Stau betroffen sind.

4. Open-Source CEP-Engines Drools und Esper

Der folgende Abschnitt soll dazu dienen die beiden zu vergleichenden Engines dieser Evaluation vorzustellen. Dazu soll ein Überblick über die Entwicklungsgeschichte der beiden Engines gegeben werden sowie ein grober Überblick über die Funktionsweise der Engines sowie ihre generellen Sprach- und Laufzeiteigenschaften. Genauere Nutzungsdetails und Anwendungen auf das Fallbeispiel werden dann in den folgenden Kapiteln durchgeführt.

4.1 Drools

Drools ist eine frei verfügbares Framework für ein regelbasiertes System mit beinhaltendem Regelinterpreter welche im Jahre 2001 von Bob McWhirter ersonnen wurde. Zwischenzeitig unter dem Namen Jboss Rules bekannt, hat sich das Projekt mittlerweile zu einer führenden Open-Source Regel-Sprache/Engine entwickelt. Drools selber wurde in Java geschrieben und kann als Wirtssprachen sowohl Java wie auch .NET nutzen. Zurzeit kann Drools in der Version 5.0 von der Seite http://jboss.org/drools/downloads.html frei heruntergeladen werden, die CEP-Komponente Fusion ist dabei bereits enthalten.

Regelbasierte Systeme bestehen aus drei Komponenten:

Eine **Regelbasis** welche die zu verwendenden Regeln besitzt. Eine Regel folgt grundlegend immer aus zwei Bestandteilen. Einen Wenn-Teil, der die Voraussetzung für die Regel beschreibt, sowie einem Dann-Teil, der beschreibt was die Wirkung einer Regel ist. Die Regelbasis kann dabei durchaus aus einer gewaltigen Menge an Regeln bestehen, was es unter Umständen recht schwierig macht solch große Regelbasen vernünftig zu pflegen. Ist die Voraussetzung, also der Wenn-Teil einer Regel wahr, so wird die im Dann-Teil festgelegte Anweisung durchgeführt. Drools nutzt nativ den mvel Logikdialekt, beherrscht aber auch sprachliche Konstrukte aus Java.[5]

Ein kurzes Beispiel einer Regel soll verdeutlichen wie Drools funktioniert:

```
rule "Monkey is hungry"
	when
		m : Monkey( a : alive == true)
	then
		modify(m){
			setHunger(m.getHunger + 1)
		}
	System.out.println("Monkey wants Banana ");
end
```

Der Regelname lautet `Monkey is hungry`, die Regel wird gefeuert wenn ein Objekt von der Klasse `Monkey` existiert, dessen Attribut `alive` wahr ist. Dem gewählten Monkey wird dabei die lokale Variable `m` zugewiesen. Im Dann-Teil wird eine Modifikation von m vorgenommen. Dabei wird die Methode `setHunger()` aufgerufen, mit dem Hunger des Affen, der um eins erhöht wird. Da keine weiteren Regelattribute eingestellt worden sind (`no-loop` vor dem **`when`** würde beispielsweise eine erneute Feuerung einer Regel verhindern), wird die Regel wiederholt feuern, zumindest wie noch ein Affe in der Laufzeitumgebung vorhanden ist. Das oben gezeigte Beispiel zeigt bereits wie viele logische Vorgänge innerhalb von nur wenig Code gesteckt werden kann, was sicherlich zur Charmanz von Regelsprachen beiträgt.

Jegliche Fakten die ausgewertet werden sollen müssen in das **Working-Memory** eingefügt werden. Dieses repräsentiert alles Wissen, was der Sprache bekannt ist und muss vom Benutzer vor Nutzung der Regeln eingefügt werden.

Der letzte Teil ist der **Regelinterpreter**, welcher beide Komponenten miteinander verbindet und auswählt, welche Konditionen nun wahr sind und die festgelegten Konsequenzen setzt. Im Gegensatz

zu einem üblich stattfindenden prozeduralen Ablauf in einer Programmiersprache muss der Regelinterpreter auswählen, welche Regel zuerst ausgewertet wird, weil ein regelbasiertes System nur einen festgelegten Zeitpunkt kennt, an dem alle Regeln ausgewertet werden. (Im Falle von Drools ist dies die Java-Methode `fireAllRules()`.) Dem geneigten Leser fällt sicherlich auf, dass dies nicht unbedingt vereinbar ist mit einem endlos andauernden Ereignisstrom eines Ereignisverarbeitenden Systems, womit klar ist, dass ein regelbasiertes System extra angepasst werden muss. Es ist weiterhin durchaus möglich, dass mehrere Regeln zum selben Zeitpunkt wahr sind, was die Auswahl welche Regel nun zuerst ausgeführt werden soll zu keiner trivialen Angelegenheit macht, insbesondere nicht, wenn vom Dann-Teil durchgeführte Modifikationen die Faktenbasis noch einmal verändern, was sich natürlich auf andere Bedingungen auswirken könnte.

Ein kurzer Exkurs über den Rete-Algorithmus:

Drools verwendet den Rete-Algorithmus welcher gut geeignet ist Regeln schnell abzuarbeiten, auch unter Änderung der Basisfakten.

Dabei werden zu überprüfende Regelprämissen in ihre atomaren Einzelteile zerlegt und in Knoten gespeichert. Komplexe Bedingungen werden dabei analog zu einem Baum verknüpft.. Diese atomaren Knoten werden dann über Kanten verknüpft, wobei die Kanten Verknüpfungen von Teilregeln darstellen. Das ist soweit nicht neu. Die erste Besonderheit ist jedoch, dass gleiche atomare Regelteile auch als gleiche Knoten dargestellt werden. Dies hält das Netzwerk möglichst klein.

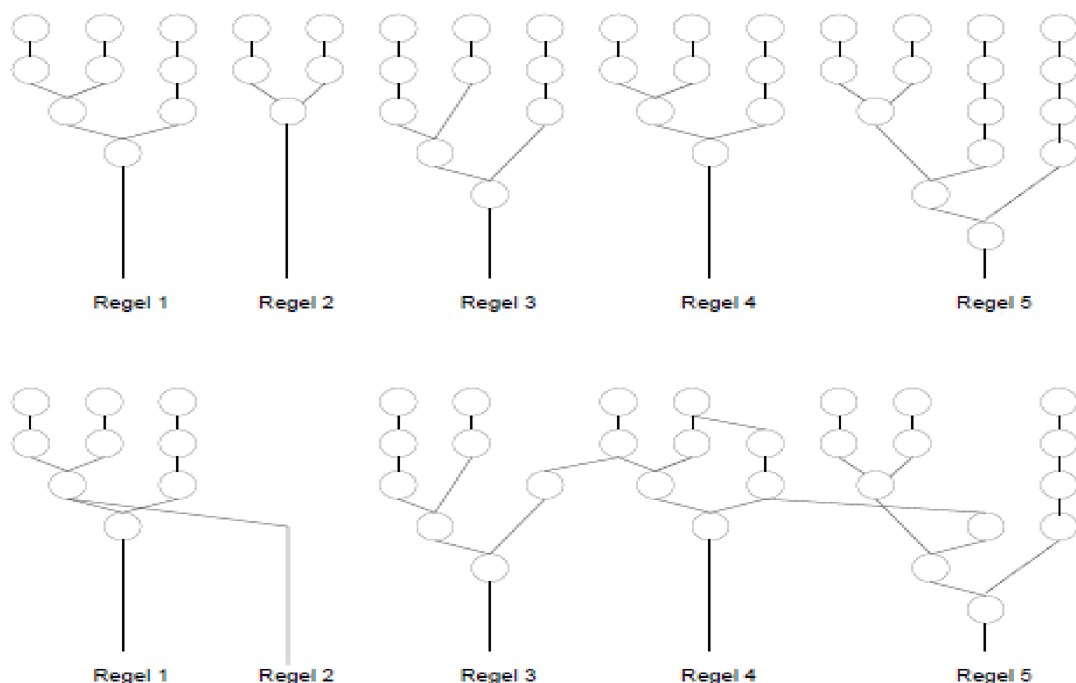

Abbildung 11: Regelzusammenführung in Rete [6]

Die oben beschriebenen atomaren Datenknoten werden auch als Alpha-Knoten bezeichnet. In den Knoten selber befinden sich die jeweils passenden Fakten, die aus dem Working-Memory stammen. Und zwar in Reihenfolge von großen Faktenmengen, zu immer kleiner werdenden. Nach den Alpha-Knoten folgen die Beta-Knoten, in denen Joins der vorher reduzierten Faktenmengen stattfinden. Dies erlaubt es Joins relativ günstig auszuführen, da die Faktenmenge möglichst klein ist.

Bei Einfügen von Daten in das Working-Memory werden die Ergebnisse aus den bereits

stattgefundenen Joins lediglich wieder verwendet, nur veränderte Daten werden durch das Netzwerk neu propagiert.[6]

Abbildung 12: Darstellung von Faktenverarbeitung in Alpha und Beta Knoten[6]

Drools-Fusion:

Es ist ja mit Sicherheit bereits aufgefallen, dass der einfache Sprachumfang einer Regelsprache nicht genügt um Ereignisverarbeitung zu realisieren. Es fehlen zeitliche oder sequentielle Konstrukte um Ordnungen innerhalb der Datenströme zu erkennen, als auch die Möglichkeit fortlaufend auf ankommende Ereignisse zu reagieren. Da diese Arbeit sehr schnell ihren Sinn ohne Ereignisverarbeitung verlieren würde, haben die Entwickler von Drools ein Paket geschnürt, welches sich Drools Fusion nennt und alle für Complex Event Processing benötigten Werkzeuge beinhalten soll. Die folgende Anwendung und Kriterien sollen spezifisch den Sprachumfang von Drools Fusion beleuchten, als auch einige Sprachkonstrukte aus dem normalen Drools-Paket vorstellen. Das Drools Fusion Paket ist eng mit dem Mutterpaket Drools verbunden, der Benutzer merkt es im Zweifelsfall gar nicht wenn er Sprachumfang aus Fusion verwendet, da kaum umfangreiche Konfigurationen vorgenommen werden müssen. Wird in dieser Arbeit nur von Drools gesprochen, so ist das Fusion Paket hier bereits implizit enthalten.

4.2 Anwendung von Drools auf Anwendungsszenario

Zuerst soll eine Beschreibung des Klassenmodells folgen, darauf die Umsetzung welche Klasse welche Verantwortlichkeiten übernimmt, zuguterletzt wird ein Blick auf die Drools Regeln geworfen, die zur Ereigniserzeugung genutzt werden.

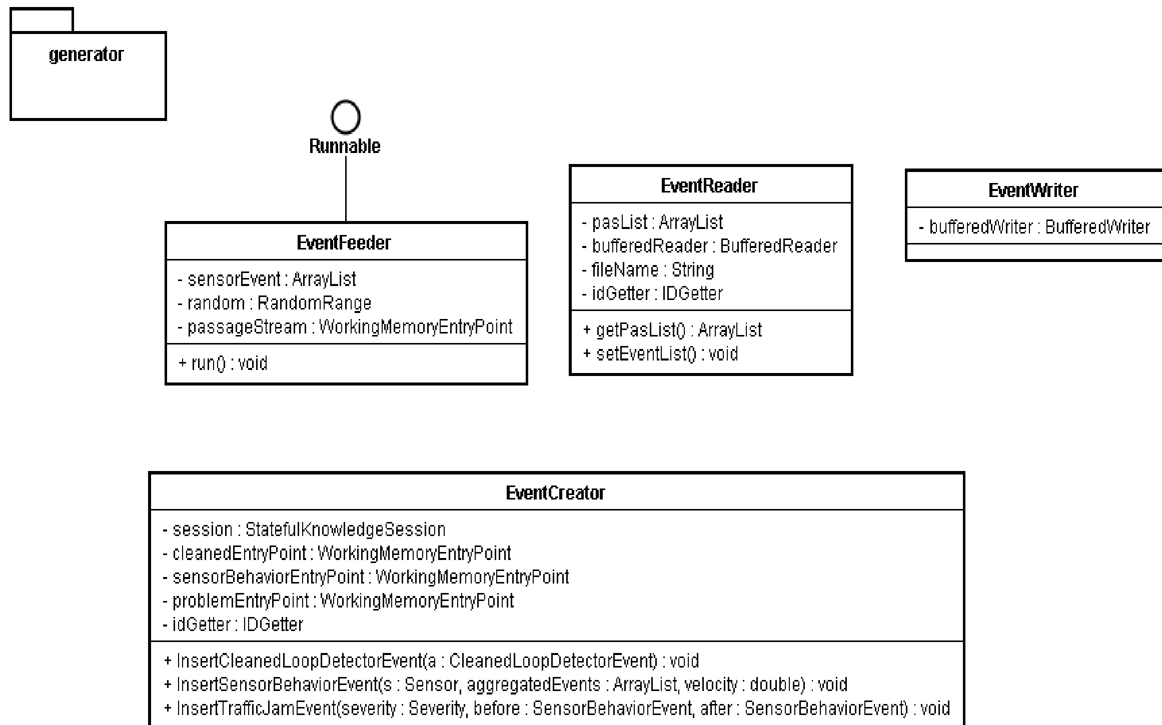

Abbildung 13: Ereignisgenerator

Das Package generator:

Das Package generator beinhaltet einmal Klassen zur Erzeugung von Ereignissen, welche aus den Drools-Regeln aufgerufen werden, sowie Klassen zum zufälligen Erzeugen von Rohdaten für Ereignisse, welche später ausgelesen werden können und als Ereignisse erzeugt werden können.

EventWriter:

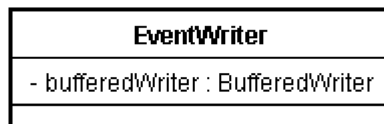

Abbildung 14: EventWriter

Der EventWriter erzeugt eine Liste von Rohdaten, die für LoopDetectorEvents genutzt werden können. Diese Rohdaten werden per Zufallsprinzip erstellt und über einen BufferedWriter in eine Text-Datei geschrieben, von der sie mit der Klasse EventReader ausgelesen werden können. Ursprünglich war geplant, dass der EventWriter mehrere Modi unterstützt die unterschiedliche Daten in Textform erzeugt, zurzeit unterstützt der EventWriter allerdings nur eine Random Erzeugung von Daten, wobei der Typ der Erzeugung dabei dem Konstruktor übergeben werden soll. Der Konstruktor der Klasse sieht dabei so aus:

```
public EventWriter(String type, int countOfPassages,int numberOfSensors, int
numberOfLanes, String filename)
```

Als erster Parameter wird ein Typ welches Ereignis geschrieben werden soll als String übermittelt. Der EventReader liest hierbei "s" als ein LoopDetectorEvent. Der zweite Parameter soll angeben wieviele Passagen, sprich wieviele einfache Events insgesamt erzeugt werden sollen, der dritte Parameter gibt an wieviele unterschiedliche Sensoren es geben soll und der vierte wieviele Lanes. (Der Einfachheit halber wird angenommen, alle Locations hätten dieselbe Anzahl an Lanes.) Die Anzahl der Locations wird in den Ereignissen dabei der Anzahl der Sensoren gleichgesetzt. Zuguterletzt schreibt der EventWriter noch die Geschwindigkeit der Passage als einen double-Wert zwischen 80 und 200. Das fertige Textdokument für die Ereignisrohdaten sieht dann wie folgt aus:

EventTyp	Ereignisnummer	SensorID	LocationID	Distanz zur nächsten Location	Fahrbahn	Geschwindigkeit
s	1	7	7	1	2	158
s	2	4	4	1	4	177
s	3	6	6	1	4	128
s	4	3	3	1	3	127
s	5	8	8	1	3	188
s	6	2	2	1	1	169
s	7	3	3	1	2	139
s	8	8	8	1	1	134
s	9	2	2	1	3	161

Grundsätzlich sind weitere Implementierungen und Definitionen von Ereignisrohdaten denkbar, zur Evaluation von den Engines soll solch eine Implementierung aber erstmal genügen.

EventReader:

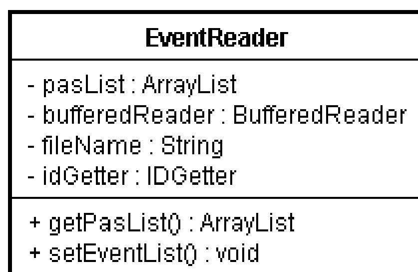

Abbildung 15: EventReader

Diese Klasse ist das Gegenstück zur Writer Klasse. Ziel dieser Klasse ist es aus den im Writer erzeugten Text-Rohdaten eine Liste von Objekten zu serialisieren, welche dann zur Weiterverarbeitung genutzt werden können.
Der Konstruktor des EventReader sieht wie folgt aus:

```
public EventReader(String file,IDGetter idget)
```

Es wird hierbei der Name der Datei, aus der die Ereignisse ausgelesen werden festgelegt, sowie ein IDGetter-Objekt übergeben, welches den zu erzeugenden Ereignissen jeweils eine einzigartige ID zuweist. Der Reader hält sich eine ArrayListe von Ereignissen, die beim Aufruf der Methode setEventList() mit einem BufferedReader aus der per Dateiname annotierten Datei ausgelesen werden. Es gilt zu beachten, dass die Datei den im Abschnitt EventWriter definierten Aufbau exakt entsprechen muss, um eine korrekte Erzeugung einer Liste von Ereignissen zu garantieren. Mit der getter-Methode getPasList() kann nach dem Setzen der Liste schließlich selbige aus dem Reader-Objekt geholt werden, nachdem dieser seinen Dienst getan hat.

EventCreator:

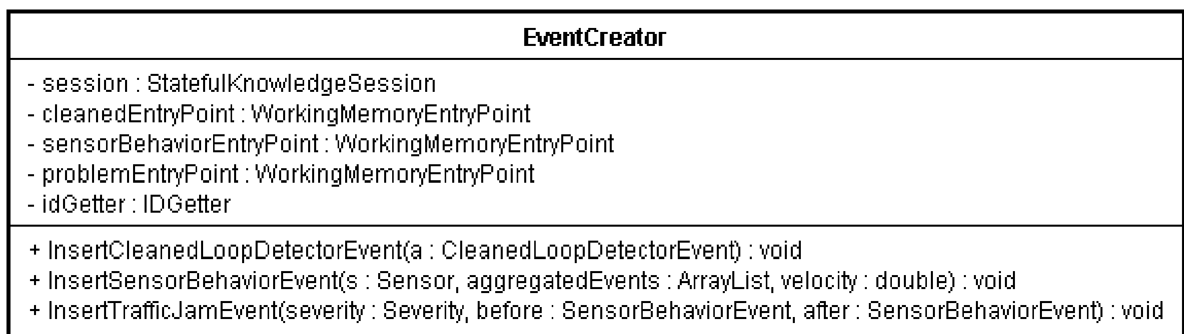

Abbildung 16: EventCreator

Diese Klasse dient dazu höherwertige Ereignisse in das Drools-Working-Memory zur Laufzeit einzufügen. Die Methoden des Inserten werden dabei statisch der Drools-Regelbasis zur Verfügung gestellt, sodass diese nur aus den Drools-Regeln aufgerufen werden müssen. Der Konstruktor des EventCreator sieht dabei wie folgt aus.

```
public EventCreator(WorkingMemoryEntryPoint clean,WorkingMemoryEntryPoint
sb,WorkingMemoryEntryPoint pr,IDGetter idget)
```

Dem EventCreator wird zum Einfügen der Ereignisse die erzeugt werden sollen mehrere WorkingMemoryEntryPoints zur Verfügung gestellt. Ein WorkingMemoryEntryPoint dient dabei als Eingang zu einem bestimmten Ereignisstrom. Drools unterscheidet dabei die WorkingMemoryEntryPoints über ihren Namen, der als String übergeben wird. Der Name als String kann also in den Drools-Regeln genutzt werden um zu erkennen über welchen Eingangsstrom ein Ereignis die Inferenzmaschine betreten hat. Dem EventCreator gilt es schließlich noch dasselbe IDGetter-Objekt wie dem EventReader zu übermitteln, was wiederum sicherstellt das jedes Ereignis eine einzigartige ID hat, indem es von anderen Ereignissen unterschieden werden kann. Die Methoden nutzen jeweils einen ihnen zugeordneten WorkingMemoryEntryPoint, wobei die Parameter direkt aus den Drools Regeln stammen. Betrachten wir solch eine Methode nun einmal beispielhaft:

```
public static void InsertSensorBehaviourEvent(Sensor s,
ArrayList<SensorEvent> aggregated,double velocity){

      if(aggregated.size() > 0){
            Date startStamp = aggregated.get(0).getEventtimestamp();
            Date endStamp = new Date();
            if(aggregated.size() == 1)
            endStamp = aggregated.get(0).getEventtimestamp();
            if(aggregated.size() > 1)
            endStamp  = aggregated.get(aggregated.size()-
            1).getEventtimestamp();


            SensorBehaviorEvent sb = new
            SensorBehaviorEvent(idgetter.getnextID(),aggregated,
            0.0,0.0,velocity,s.getSensorid(),startStamp,endStamp,
            new Date(),120, utils.StaticMethods.evalAverage(velocity));
            sensorBehaviorEntryPoint.insert(sb);
      }
}
```

Hier wird ein SensorBehaviourEvent erstellt und anschließend in das Drools Working-Memory eingefügt. Das Wissen welches die Methode benötigt besteht aus dem betroffenen Sensor an dem die Daten gesammelt wurden, sowie die Liste an Ereignissen die gesammelt wurden und die ausgerechnete Durchschnittsgeschwindigkeit(velocity). Die von dem aggregierten Ereignis benötigten Start und Endzeitstempel werden dabei aus der Liste von Ereignissen ausgelesen, ist dies geschehen wird dem Ereignis neben den übermittelten Daten noch eine einzigartige ID zugewiesen, ein Zeitpunkt für den Moment an dem das Ereignis in das WorkingMemory eingefügt wurde, bzw. An dem das Ereignis stattgefunden hat, eine Dauer, für die das Ereignis als gültig gilt(In diesem Beispiel 120 Sekunden). Sowie einen mit der Durchschnittsgeschwindigkeit evaluierten enum-Wert, der eine qualitative Bewertung der Durchschnittsgeschwindigkeit geben soll, der bei Mustererkennung zur Erzeugung von komplexen Ereignissen genutzt wird.

EventFeeder:

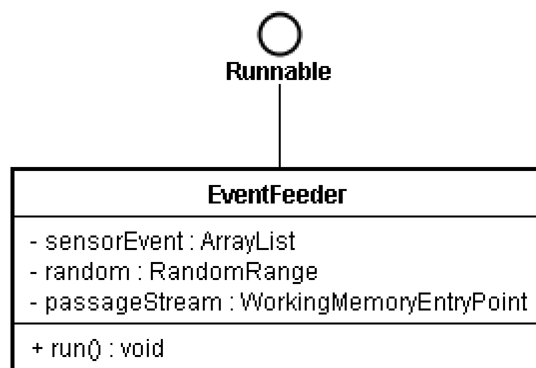

Abbildung 17: EventFeeder implementiert Runnable

Das Ziel des EventFeeders ist es, dass Drools Working-Memory mit SensorEvents zu versorgen. Dazu läuft diese Klasse in einem eigenen Thread

```
public EventFeeder(ArrayList<SensorEvent> events, WorkingMemoryEntryPoint
passageStream){
          eventList = events;
          rd = new RandomRange();
          this.passageStream = passageStream;
      }
```

Der Konstruktor dieser Klasse wird von der Main-Klasse aufgerufen, wobei diese Klasse dabei mit der von dem EventReader aufbereiteten Liste von sensorEvents versorgt wird. Um einen Eintrittspunkt für diese "Low-Level Events" in das WorkingMemory zu haben wird dieser ebenfalls noch vom Konstruktor mitgeliefert. Der Konstruktor erstellt sich noch ein Exemplar von RandomRange, welches dazu genutzt wird zu zufälligen Zeitpunkten SensorEvents einzufügen.

```
      public void run() {

            Random rand = new Random();
            Iterator it = eventList.iterator();
            while(it.hasNext()){
                  passageStream.insert( it.next() );
                  try {
                        Thread.sleep(rd.showRandomInteger(1000, 2000, rand));
                  } catch (InterruptedException e) {

                        e.printStackTrace();
                  }
            }
      }
```

Innerhalb der run-Methode wird nun über die Liste an Ereignissen iteriert und diese mit Zeitverzögerung zwischen einer und zwei Sekunden in den passageStream(der Strom für einfache SensorEvents) eingefügt, so soll ein kontinuierlicher Strom an Ereignissen ähnlich einem echten Verkehrsablauf simuliert werden.

Das Package execution:

Main:
Die Main-Klasse wird erwartungsgemäß natürlich zuerst ausgeführt, ihr gebührt die ehrenvolle Aufgabe alle "Stricke" zusammen zu führen.

```
IDGetter idget = new IDGetter();
EventReader r = new EventReader("events.txt",idget);
```

Zuerst wird die EventReader-Klasse erzeugt, welche die einfachen Sensorereignisse aus der "events.txt" auslesen soll und ihnen eine ID verpassen soll

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add(ResourceFactory.newClassPathResource("TrafficRules.drl"),
ResourceType.DRL);
KnowledgeBuilderErrors errors = kbuilder.getErrors();
if (errors.size() > 0) {
                  for (KnowledgeBuilderError error: errors) {
                        System.err.println(error);
                  }
                  throw new IllegalArgumentException("Could not parse
knowledge.");
            }
```

Im nächsten Schritt wird der Knowledgebuilder kreiert, welcher die "TrafficRules.drl" kompiliert und in sogenannte Knowledgepackages umwandelt, welche wiederum in eine KnowledgeBase eingefügt werden müssen. Es ist vorstellbar, dass der KnowledgeBuilder noch weitere Regelbeinhaltende Dateien kompilieren kann, für dieses Beispiel wurde aber nur ein eine einzige Regeldatei genutzt.

```
KnowledgeBaseConfiguration kbaseconf =
KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
kbaseconf.setOption( EventProcessingOption.STREAM );
```

Hier wird eine KnowledgeBaseConfiguration angelegt. Es ist wichtig der Engine mitzuteilen, dass von nun an im sogenannten Stream-Modus gearbeitet werden soll. Dieser soll der Engine ermöglichen Zugriff auf die Systemzeit zu haben und so Zeitstempel von Ereignissen auf den jetzigen Zeitpunkt im System zu prüfen. Dies ist zum Beipiel notwendig zum Nutzen von Fenstern oder zum Warten auf Eintreffen eines Ereignisses. Das Gegenteil wäre der Cloud-Modus, welcher wie eine normale Inferenzmaschine arbeitet und kein Wissen über die Zeit hat..

```
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase(kbaseconf);
kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
```

Die vom KnowledgeBuilder erzeugten KnowledgePackages werden in die KnowledgeBase eingefügt,die KnowledgeBase arbeitet dabei als Container für die KnowledgePackages und bewahrt diese auf. Die oben erstellte Konfiguration wird ebenfalls übermittelt.

```
KnowledgeSessionConfiguration conf =
KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
conf.setOption( ClockTypeOption.get( "realtime" ) );
```

Es kann noch eine Konfiguration für die Session an sich erstellt werden. Der Drools-Engine wird mit dem Stichwort "realtime" mitgeteilt, dass sie die lokale Systemzeit heranziehen soll. Drools erlaubt auch eine Pseudo-Uhr, die dem Anwender explizit erlaubt die Zeit voranschreiten zu lassen mit:

```
SessionPseudoClock clock = session.getSessionClock();
clock.advanceTime( 360, TimeUnit.SECONDS );
```

Dies gibt dem Benutzer die Möglichkeit den Zeitverlauf selber durchzuführen und ist damit ein gutes Werkzeug zum Debuggen der geschriebenen Regeln.

```
StatefulKnowledgeSession session = kbase.newStatefulKnowledgeSession( conf,
null );
            for(int i = 1; i<=8; i++){
                  session.insert(new Sensor(i));
            }
```

Erzeugen einer neuen Stateful-Session mit der oben erstellten Konfiguration. Hier werden zum Start Sensoren als Fakten aus dem erstellten Modell eingefügt, auf die Ereignissdatenstöme dann in den Regeln verknüpft(gejoint) werden.

```
WorkingMemoryEntryPoint passageStream = session.getWorkingMemoryEntryPoint(
"LoopDetectorEventStream" );
WorkingMemoryEntryPoint cleanStream =
session.getWorkingMemoryEntryPoint("CleanedLoopDetectorEventStream");
WorkingMemoryEntryPoint sensorBehaviorStream =
session.getWorkingMemoryEntryPoint("SensorBehaviorEventStream");
WorkingMemoryEntryPoint problemStream =
session.getWorkingMemoryEntryPoint("ProblemEventStream");
```

Die Erstellung von Eingangspunkten in die Engine. Mehrere Eingangsströme können dabei

nebenläufig in das Working-Memory eingefügt werden. Somit kann eine verteilte Architektur erstellt werden, aus der Ereignisse von unterschiedlichen Generatoren und Ereignisverarbeitungsstationen stammen.

```
EventCreator c = new
EventCreator(cleanStream,sensorBehaviorStream,problemStream,idget);
EventFeeder f = new EventFeeder(r.getPaslist(),passageStream);
Thread EventThread = new Thread(f);
EventThread.start();
```

Dies sind die letzten Bereitstellungsarbeiten. Der EventCreator wird später aus den Drools Regeln aufgerufen um aggregierte sowie komplexe Ereignisse zu erstellen und in die Engine einzuführen. Weiterhin wird ein Thread erstellt um einfache Ereignisse einzufügen.

Vorstellung der verwendeten Regeln:

Dieser Abschnitt beschreibt die TrafficRules.drl, die zur Mustererkennung und Ereigniserzeugung genutzt wird, diese Datei ist direkt in der Drools-Logiksprache geschrieben.

```
package Execution
import model.LoopDetectorEvent
import model.CleanedLoopDetectorEvent
import model.SensorBehaviorEvent
import generator.EventCreator
import model.Sensor
import java.util.ArrayList
import model.AverageSpeed
import model.Severity
import model.ProblemEvent

dialect "mvel"
```

Nach der package-Deklarierung und Importen von Klassen aus dem Modell wird festgelegt, dass der mvel-Dialekt verwendet werden soll. Dieser ist eine Sprache welches es erlaubt logische Ausdrücke zu formulieren und der übliche Standarddialekt bei Verwendung von Drools.[5]

```
declare LoopDetectorEvent
      @role( event )
      @expires( 3s )
      @timestamp ( eventtimestamp )
      @duration ( eventduration )
end

declare CleanedLoopDetectorEvent
      @role( event )
      @expires( 20m )
      @timestamp ( eventtimestamp )
      @duration ( eventduration )
end

declare SensorBehaviorEvent
      @role ( event )
      @expires ( 20m )
      @timestamp ( eventtimestamp )
      @duration ( eventduration )
end
```

```
declare ProblemEvent
      @role ( event )
      @expires ( 1h )
      @timestamp ( eventtimestamp )
      @duration ( eventduration )
end
```

Hier werden die verwendeten Ereignisse für die Laufzeitumgebung definiert. Dabei muss Ereignissen diese Rolle explizit zugewiesen werden, wenn diese mit den Funktionen der Drools-Fusion Umgebung verwendet werden sollen. Die Ereignisse erhalten noch eine Verweildauer welche angibt wie lange sie im Speicher erhalten bleiben sollen, bis diese wieder entfernt werden. Aus dem Speicher entfernte Ereignisse werden vom Garbage-Collector abgeholt und können in Zukunft nicht mehr für Mustererkennung genutzt werden. Die Parameter `@timestamp` und `@duration` zeigen der Drools-Engine die Namen der Java Objektattribute auf, welche für diese Rollen gedacht sind. Bis auf die Rolle als Event sind diese Angaben lediglich optional, die Session weist den Ereignissen im Zweifelsfall beim Einfügen selber eigene Attribute zu, die diese Aufgabe übernehmen sollen und führt auch automatisches Garbage-Collecting durch. In diesem Fall werden aber Attribute aus dem Modell genommen, damit diese später in der Wirtssprache Java verwendet werden können.

Regel 1:

```
rule "putCleanEventsForward"
      when $p : LoopDetectorEvent (velocity > 0 && < 280, $i : sensorid) from
            entry-point "LoopDetectorEventStream"
            not (  LoopDetectorEvent (sensorid == $i, this after [0,1s] $p) from
            entry-point "LoopDetectorEventStream")
      then
            EventCreator.InsertCleanedLoopDetectorEvent($p)
      end
```

Diese Regel erzeugt aus "rohen" `LoopDetectorEvents` saubere `CleanedLoopDetectorEvent`, die auf Fehler in den erzeugenden Sensoren geprüft wurden. Dies soll geschehen wenn im WorkingMemory ein LoopDetectorEvent aus dem Eingangsstrom `LoopDetectorEventStream`, (Wird über den Namen als String referenziert, welcher in dem WorkingMemoryEntryPoint in der Main-Klasse definiert ist) welches im korrekten Geschwindigkeitsbereich von 0 bis 280 Stundenkilometern ist, kommt. Dabei darf gleichzeitig kein falsches Duplikat mit der selben `sensorid` im Zeitraum einer Sekunde nach dem ersten erzeugt werden. Diese Regel feuert im Falle von Duplikaten nur einmal, für das zweite ankommende Ereignis, für das Duplikat greift dann die Nicht-Bedingung nicht mehr, sodass keine Feuerung stattfindet. Feuert die Regel wird vom `EventCreator` ein neues `CleanedLoopDetectorEvent` eingereicht. Jeglicher Zugriff auf Attribute und Überprüfungen von Attributen werden in den Klammern nach der Faktennamensdefinition eingefügt. Hier ist auch die polnische Notation sowie eine ODER-Verknüpfung zu sehen. Mit dem für Drools Fusion neu hinzugekommenen temporalen-Operator:

```
this after [0,1s] $p
```

kann ein temporaler Vergleich von zwei als Ereignisse definierte Fakten durchgeführt werden.

Regel 2:

```
rule "accumulate average Speed on certain sensor and create Event for Change
      Notification"
      when
            $s :  Sensor ($i : sensorid)
            $p : Number($average : doubleValue) from
            accumulate(CleanedLoopDetectorEvent(sensorid == $i, $velocity :
            velocity) over window:time(5m)
            from entry-point "CleanedLoopDetectorEventStream",
            average($velocity))
            $ev : ArrayList() from collect(CleanedLoopDetectorEvent(sensorid ==
            $i) over window:time(5m) from entry-point
            "CleanedLoopDetectorEventStream")
      then
            EventCreator.InsertSensorBehaviourEvent($s,$ev,$average)
end
```

Diese Regel führt eine Aggregierung einer Liste von `CleanedLoopDetectorEvents` in ein `SensorBehaviourEvent` durch welches dem Typus der zweiten Stufe der Ereignishierarchie entspricht. Hierbei werden mit

```
accumulate (CleanedLoopDetectorEvent(sensorid == $i, $velocity :
      velocity) over window:time(50s)
```

über ein zeitbasiertes Schiebefenster der Größe der letzten fünf Minuten `CleanedLoopDetectorEvents` gesammelt. Die Verknüpfungsbedingung für diese Sammlung ist dabei alle Ereignisse, die die gleiche `sensorid` teilen. Mit der `average($velocity)`-Funktion wird für diese Sammlung ein Durchschnittswert für alle Passagen errechnet. Um dem aggregierten Ereignis noch die Liste der für seine Erzeugung verantwortlichen simplen Ereignisse zu geben wird noch eine ArrayListe mit dem selben Verfahren erzeugt mit dem Befehl

```
from collect
```

die dann dem EventCreator gegeben wird. Das verschiebende Fenster ist ein aus Fusion stammendes neu hinzugekommenes Sprachkonstrukt, welches es erlaubt Sammlungen über einen bestimmten Zeitraum umzusetzen.

Regel 3:

```
rule "check for rapid changes"
      when
            $se1 : SensorBehaviorEvent($sid : sensorid)
            from entry-point "SensorBehaviorEventStream"
            $se2 : SensorBehaviorEvent(sensorid == $sid,(evaledAverage ==
            AverageSpeed.HIGH && $se1.evaledAverage == AverageSpeed.LOW),  this
            after[0,2m] $se1  ) from entry-point "SensorBehaviorEventStream"
      then
            EventCreator.InsertTrafficJamEvent(Severity.HIGH,$se1,$se2)
end
```

Mit dieser Regel soll eine Mustererkennung für zwei innerhalb einer Minute aufeinander folgende `SensorBehaviorEvents` welche die selbe `sensorid` haben durchgeführt werden, ob sich die evaluierte Durchschnittsgeschwindigkeit an diesem Sensor plötzlich von hoch auf langsam verändert hat. Ist dies der Fall soll ein komplexes `TrafficJamEvent` aus der Spitze der Ereignishierarchie eingefügt werden, welches auf einen Verkehrsstau hinweist. Hier ist diesmal eine UND-Verknüpfung in der Bedingung zu sehen, sowie erneut ein temporaler Operator.

Regel 4:

```
rule "check for continous changing pattern of sinking speed"
      when
            $se1 : SensorBehaviorEvent($sid : sensorid, $vel1 : averageSpeed)
            from entry-point "SensorBehaviorEventStream"
            $se2 : SensorBehaviorEvent(sensorid == $sid,$vel2 :
            averageSpeed,averageSpeed < $vel1,  this after[1m,2m] $se1  ) from
            entry-point "SensorBehaviorEventStream"
            $se3 : SensorBehaviorEvent(sensorid == $sid,$vel3 :
            averageSpeed,averageSpeed < $vel2,  this after[1m,2m] $se2  ) from
            entry-point "SensorBehaviorEventStream"
            $se4 : SensorBehaviorEvent(sensorid == $sid,$vel4 :
            averageSpeed,averageSpeed < $vel3,  this after[1m,2m] $se3  ) from
            entry-point "SensorBehaviorEventStream"
            $se5 : SensorBehaviorEvent(sensorid == $sid,averageSpeed < $vel4,
            this after[1m,2m] $se4  ) from entry-point
            "SensorBehaviorEventStream"
      then
            System.out.println("Ein kontinuierlicher Abfall der
            Geschwindigkeiten");
      end
```

Diese Regel soll auf einen kontinuierlichen Abfall der gemessenen Geschwindigkeiten prüfen. Dabei werden mehrere innerhalb einer Minute jeweils aufeinanderfolgende Ereignisse geprüft, ob sich ein Muster von sinkenden Geschwindigkeiten entwickelt. Mit dieser Regel wird quasi eine Form von Sequenzmustererkennung realisiert, weil die Regel durch diesen Aufbau genau spezifiziert, wie Ereignisse aufeinander zu folgen haben. Unschönerweise kennt Drools Fusion kein sprachliches Konstrukt, welches es erlaubt genau **das** nächste Ereignis nach einem bestimmten stattgefundenen aus dem Ereignisstrom zu betrachten. Dies zwingt den Entwickler dazu genau zu wissen in welchem zeitlichen Abstand die Ereignisse eintreffen. Würde ein `SensorBehaviorEvent` die Grenze von einer Minute überschreiten und erst dann eintreffen, so wäre die Kette durchbrochen und die Mustererkennung würde nicht mehr funktionieren.

Regel 5:

```
rule "there is a problem"
      when
            $p1 : ProblemEvent()  from entry-point "ProblemEventStream"
      then
            System.out.println("Ein Ereignis der Klasse ProblemEventStream");
end
```

Hier folgt eine kurze Ausgabe wenn ein ProblemEvent vom Eingangspunkt ProblemEventStream durchkommt.

Regel 6:

```
rule "waitingforProblem"
when $a : model.LoopDetectorEvent () from entry-point "LoopDetectorEventStream"
model.ProblemEvent (this after  $a) from entry-point "ProblemEventStream"
then
System.out.print(ProblemEvent gefunden );
end
```

Diese Regel wird im Performance-Test im Evaluationskapitel verwendet.

4.3 Esper

Der Softwarearchitekt Thomas Bernhardt führte 2004 eine Evaluierung für ein großes Finanzhaus von unterschiedlichen Regelbasierten Engines für die Live-Überwachung von Geschäften durch. Im Zuge dieser Evaluierung kam er zu dem Entschluß, dass es das Beste wäre, wenn er selber ein eigenes System für solch eine Aufgabe schreibt. Herausgekommen ist dabei schlußendlich das in Java geschriebene Esper-Projekt. Grundlegend ist Esper stark verschieden von dem auf einen regelbasierten System aufbauenden Drools. Esper verwendet eine SQL-ähnliche Sprache die Continous Query Language genannt wird. Eingehende Ereignisse, die auf POJO's basieren werden in ein Fenster eingefügt und dort gesammelt und bei Aufruf eines CQL-Statements an einem registrierten Listener weitergereicht, wo die Ereignisse dann ausgelesen oder im Zuge der Statements erzeugte Daten weiterverarbeitet werden. Somit kann Esper im weitesten Sinne als ein System, welches mit ähnlichen Funktionalitäten wie ein aktives Datenbanksystem ausgestattet ist, bezeichnet werden, welches keine Anfragen an eine statische Datenbank durchführt sondern Anfragen an einen kontinuierlichen Datenstrom. Esper arbeitet dabei eng mit der Wirtssprache Java zusammen, besitzt weiterhin jedoch eine Version welche für .NET verfügbar ist. Die getestete Version von Esper in dieser Evaluation ist 3.4, während diese Arbeit geschrieben wurde, wurde von den Machern bereits eine weitere stabile Version 3.5 herausgegeben.

Ein kurzes Beispiel aus dem Sprachumfang von Esper:

```
select  *
from StockTick(symbol='AAPL').win:length(2)
having avg(price) > 6.0
```

Wie zu sehen ist, ist der Aufbau der Sprache an der einer SQL Datenbankabfrage-Sprache angelehnt. Der obige Ausdruck soll alle StockTick Ereignisse mit dem Symbol AAPL auslesen, wenn zwei hintereinander folgende Stockticks im Mittel einen höheren Preis als 6 haben. Wie ersichtlich kann hier mit wenigen Befehlen bereits sehr spezifisch in einen Datenstrom eingegriffen werden und Daten aus einem bestimmten Muster herausgeholt werden.

4.4 Anwendung von Esper auf Anwendungsszenario

Hier folgt die Implementierung des Anwendungsszenarios in Esper. Die CQL-Statements von Esper wurden dabei zu einem guten Teil aus Event-Driven Architecture[1] übernommen. Zur Veranschaulichung der Sprache wurden dabei einige Anpassungen an den Regeln durchgeführt.

EventFeeder:

Der Ablauf wie LoopDetectorEvents in die Laufzeitumgebung von Esper eingeführt werden ist hier gleich wie in der Drools-Implementierung. Der Befehl zum Einfügen eines Ereignisses:

```
cepRT.sendEvent( it.next() );
```

ist dabei analog zum Befehl:

```
passageStream.insert( it.next() );
```

innerhalb der Drools-Implementierung. Beide Befehle fügen ein Ereignis manuell in den Datenstrom ein. CepRT stellt dabei die erstellte Esper Laufzeitumgebung dar.

Main:

```
Configuration cepConfig = new Configuration();
cepConfig.addEventType("LoopDetectorEvent",LoopDetectorEvent.class.getName());
cepConfig.addEventType("SensorBehaviorEvent",
SensorBehaviorEvent.class.getName());
cepConfig.addEventType("ProblemEvent",ProblemEvent.class.getName());
cepConfig.addEventType("CleanedLoopDetectorEvent",
CleanedLoopDetectorEvent.class.getName());
cepConfig.addImport("utils.*");
```

Um Ereignisse an Esper bekannt zu geben müssen diese in einem Konfigurationsobjekt eingefügt werden. Das obige Beispiel gibt einen Teil der im Modell definierten Ereignisse bekannt, dabei wird bei Bekanntgabe dem Objekt einen qualifizierenden Namen, welcher in den folgenden Statements benutzt wird, als String sowie der komplette Name der Klasse gegeben. In der Beispielimplementierung sind diese Ereignisse als POJO's implementiert, Esper erlaubt jedoch auch weitere mögliche Ereignisdefinitionsmodi. Neben Ereignisdefinitionen können in der Konfiguration auch noch Importe von verwendeten Bibliotheken bekanntgegeben werden. Dort können auch noch weitere Anpassungen an das spätere Verhalten der Laufzeitumgebung durchgeführt werden, auf die hier jedoch nicht näher eingegangen werden soll.

```
EPServiceProvider cep = EPServiceProviderManager.getProvider("Feeder",
cepConfig);
EPRuntime cepRT = cep.getEPRuntime();
```

Hier wird eine Instanz einer Laufzeitumgebung mit Namen "Feeder" von Esper erzeugt. Die oben erstellte Konfiguration wird der Laufzeitumgebung übergeben und ein sogenannter Service-Provider erzeugt der über einen String-Namen qualifiziert wird. Wird getProvider() dabei erneut mit dem gleichen String-Qualifikator aufgerufen, so erhält man Zugriff auf die gleiche zuvor in der Esper-Engine erstellte Laufzeitumgebung. Natürlich können so mehrere Laufzeitumgebungen zur selben Zeit erstellt werden. Jegliche Daten können nachfolgend über das EPRuntime-Objekt eingefügt werden, wie oben in der Beschreibung des EventFeeders gezeigt.

```
EPAdministrator cepAdm = cep.getEPAdministrator();
```

Mit diesem Befehl wird eine Schnittstelle zum Einfügen von CQL-Statements bereitgestellt.

Statement 1:

```
EPStatement LDEStatement = cepAdm.createEPL("
select a
from pattern [ every timer:interval(1 sec) -> (a = LoopDetectorEvent(velocity >
0)  ->  not b = LoopDetectorEvent(sensorid=a.sensorid))
where  timer : within(1 seconds )]  "
);
```

Das erste Statement ist analog zur Regel 1 aus der Drools-Implementierung , wobei mit createEPL ein Statement erzeugt wird. Dieses Statement soll noch einmal komplett wie im Quellcode vorhanden gezeigt werden, weitere werden ohne anhängende Befehle dargestellt. Hier soll ein LoopDetectorEvent mit select a, wobei dies der im Pattern benutzte Variablenname ist, aus dem Strom an Daten genommen werden. Dieses Statement zeigt bereits die Benutzung eines Patterns. Als Grundregel gilt, dass Pattern grundsätzlich von links nach rechts zu lesen sind. Das Pattern wird wahr, wenn eine Sekunde vergeht und danach ein LoopDetectorEvent mit einer Geschwindigkeit

höher als 0 gefolgt von keinem weiteren LoopDetector Event mit der selben SensorID in der gleichen Sekunde erscheint. Der Ausdruck

```
where timer : within(1 seconds)
```

ist dabei ein sogenannter Post-Guard, der sich auf den vor ihm stehenden Ausdruck bezieht. Man beachte dabei die Klammerung, die somit den Ausdruck

```
a = LoopDetectorEvent(velocity > 0)  ->  not b =
LoopDetectorEvent(sensorid=a.sensorid)
```

zu einem kompletten eingebetteten Statement macht. Der Grundablauf in der Laufzeitumgebung ist dabei, dass zuerst geprüft wird ob es wahr ist, dass eine Sekunde vergangen ist, gefolgt von der in den Klammern angegebenen Musterprüfung. Wird diese dann als wahr angenommen, wird der Post-Guard betrachtet und prüft ob sich der geklammerte Ausdruck als innerhalb einer Sekunde bewahrheitet hat. Ist dies der Fall so wird das gesamte Pattern wahr und das Statement feuert, wobei die selektierten Daten an möglicherweise registrierte Listener gereicht werden. Die immer wiederkehrende Prüfung ob eine Sekunde vergangen ist zu Anfang des Patterns ist dabei ein Kniff um das Pattern zur korrekten Ausführung zu bewegen. Wenn dieser Teil fehlen würde, wäre das Pattern nur für eine einzige Instanz eines ankommenden LoopDetectorEvents gültig und dies führt dazu das die Laufzeitumgebung das Statement verwirft sobald es einmal wahr geworden ist (Einmal wahr gewordene Statements ohne Wiederholungsoperator wie every werden nicht noch einmal von der Laufzeitumgebung betrachtet.). Wäre der every Operator wiederum in das geklammerte Pattern selber eingefügt,

```
every a = LoopDetectorEvent(velocity > 0)  ->  not every b =
LoopDetectorEvent(sensorid=a.sensorid)
```

so würde das Pattern immer erneut für das erste eingefügte Ereignis feuern. Zwar gibt es unterschiedliche Wege eine korrekte Realisierung zu erzwingen (u.a. mit dem every-distinct Operator), allerdings ist dieser Weg auch ein Gängiger.

```
LDEStatement.addListener(new LoopDetectorEventListener());
```

Hiermit wird noch schnell ein Listener für das oben erstellte Statement registriert. Eine beispielhafte Implementierung eines Listeners folgt weiter unten.

Statement 2:

```
select sensorid, avg(velocity) ,StaticMethods.evalAverage(avg(velocity))
from CleanedLoopDetectorEvent.std:groupby(sensorid).win:time_batch(5 min)
group by sensorid
```

Das zweite Statement führt eine Aggregation von CleanedLoopDetectorEvents durch und ist so bis auf weiteres aus Event-Driven Architectures übernommen.[1] Im Gegensatz zum ersten Statement wurde hier keine Mustererkennung durchgeführt. Der Feuermoment des Statements berechnet sich einzig und allein aus dem Batch-Fenster, und zwar genau Fünf Minuten nach Eintreffen des ersten Ereignisses. Der dritte Parameter innerhalb des Select-Teils zeigt einen Aufruf einer Methode aus dem Modell zur Bewertung der Durchschnittsgeschwindigkeit. Im Select-Teil wird die Aggregationsmethode avg zur Berechnung der Durchschnittsgeschwindigkeit benutzt wobei diese dabei mit

```
group by sensorid
```

für jede mögliche sensorID berechnet wird. Ein letztes ganz besonders intressantes Konstrukt ist dieser Befehl:

```
CleanedLoopDetectorEvent.std:groupby(sensorid).win:time_batch(5 min)
```

Hier wird ein als View bekanntes Konstrukt, also ein spezifisches Fenster auf einen Datenbestand betrachtet. Es würde uns nichts daran hindern, den Befehl auch nur als solchen zu schreiben:

```
CleanedLoopDetectorEvent.win:time_batch(20 sec)
```

Wie ersichtlich fehlt hier der View `.std:groupby(sensorid)`. Dieser erlaubt es für jede sensorID ein eigenes Batch-Fenster zu erstellen. Ist der View nicht vorhanden, so würde es nur ein einziges Batch Fenster für alle SensorID's geben. Zwar beinhaltet die Ausgabe dann immer noch die Ergebnisse für jede SensorID, allerdings nur alle fünf Minuten für ein einziges Fenster auf alle SensorID's. So hat jede SensorID sein eigenes Fenster, welches auch nur dann startet, sobald ein passendes Ereignis eintrifft.

Statement 3:

```
select s2.sensorid,s2.evaledAverage
from pattern [ every(s1=SensorBehaviorEvent -> s2=SensorBehaviorEvent(sensorid =
s1.sensorid)) where timer:within(1 min)]
where s1.evaledAverage = model.AverageSpeed.HIGH and s1.evaledAverage =
model.AverageSpeed.LOW"
```

Das letzte Statement stammt ebenfalls aus [1], und ist analog zur Drools Regel 3. Das Pattern prüft ob zwei SensorBehaviorEvents nacheinander innerhalb einer Minute existieren, wobei die ausgewertete mittlere Geschwindigkeit von hoch auf niedrig umspringt. Ist dies der Fall springt der registrierte Listener an und behandelt das entdeckte Pattern.

Statement 4:

```
Select avg(cl.velocity)
from pattern [ every cl = CleanedLoopDetectorEvent and not
CleanedLoopDetectorEvent(velocity > 200)].win:time_batch(20 sec)
```

In diesem Muster werden alle CleanedLoopDetectorEvents über 20 Sekunden gesammelt und das Muster aktiviert sich, solange Absenz von einem Ereignis vorhanden ist, dessen Geschwindigkeit über 200 km/h schnell ist. Ist das Muster aktiviert, wird der Mittelwert aller sich im Fenster befindenden Ereignisse weitergeleitet.

Statement 5:

```
select *
from pattern [ a = LoopDetectorEvent -> ProblemEvent ]
```

Dieses Statement wird im Performance-Test verwendet.

Statement 6:

```
select *
from pattern [every a = LoopDetectorEvent -> b = ProblemEvent ]
```

Letztes Statement wird im Abschnitt "Absenz von Ereignissen" verwendet.

LoopDetectorEventListener:

Hier ist beispielhaft ein LoopDetectorEventListener aufgeführt.:

```
public class LoopDetectorEventListener implements UpdateListener {

public void update(EventBean[] newData, EventBean[] oldData) {
      EPServiceProvider taa = EPServiceProviderManager.getProvider("Feeder");
      EPRuntime runtime = taa.getEPRuntime();
      LoopDetectorEvent l = (LoopDetectorEvent) newData[0].get("a");
      CleanedLoopDetectorEvent clean = new
      CleanedLoopDetectorEvent(idgetter.getnextID(),l.getSensorid(),
      l.getLocation(),orglist,l.getVelocity(),new Date(),120);
      runtime.sendEvent(clean);
      }
}
```

Der Code soll hier kurz aufzeigen wie es möglich ist, Daten aus den durch das Select-Statement erzeugte EventBean auszulesen. Es fällt aus, dass die vom UpdateListener zu implementierende Methode update() in ihrer Signatur zwei Parameter EventBeans beinhaltet. Der erste newData betitelte Parameter ist hierbei der sogenannte Insert-Stream der alle Ereignisse die sich zur Zeit in einem von der Select-Klausel definierten Fenster befinden beinhaltet(Bei Feuerung des Listeners). Der zweite Parameter oldData ist der Remove-Stream eines Fensters und enthält alle Ereignisse, die das Fenster bereits wieder verlassen haben, aber noch nicht an Listener gepostet wurden(Beispielsweise wenn ein Fenster die Größe 5 hat, so werden die bereits verdrängten Ereignisse hier landen, bei Erkennung eines Muster). Mit get() ist es möglich mit dem innerhalb der Select-Klausel festgelegten Name an die beinhalteten Daten zu gelangen. Zuguterletzt sendet der Listener das neu erzeugte Ereignis wieder an die Laufzeitumgebung.

5. Vergleich der Ereignisverarbeitenden Sprachen und ihren Engines

Dieser Abschnitt dient zur Vorstellung der erarbeiteten Vergleichskriterien aus welchen die Evaluation der beiden Engines erfolgen soll. Die Anforderungen wurden dabei mit den Anforderungen der Industrie an Ereignisverarbeitenden Engines sowie mit den gewonnenen Erkenntnissen im Zuge der Implementierung des Anwendungsfalles erstellt. Die Quellen die für die Evaluierung herangezogen wurden sind:

- [7] welches eine vom kommerziellen Lösungsanbieter Coral erarbeitete, selbstbetitelt unabhängige Richtlinie zur Evaluierung von ereignisverarbeitenden Laufzeitumgebungen ist.
- [8] Ein von Tibco in ihrem Blog gepostete Checkliste zur Evaluierung von CEP-Engines
- [9] Ein von den Drools Entwicklern geschriebener Liste von Anforderungen an eine CEP-Engine

5.1 Sprachparadigma

In der Welt der Ereignisverarbeitenden Regelsprachen haben sich drei sprachlichen[10] Herangehensweisen mittlerweile etabliert, wobei diese Evaluation die beiden geläufigsten Systeme unter die Lupe nimmt.

Einerseits existieren die sogenannten Continous Query Languages, welches Sprachen sind, die an SQL angelehnt worden sind. Im Gegensatz zu SQL wird dabei allerdings nicht einmalig eine Anfrage auf eine Datenmenge durchgeführt sondern es wird durchgängig auf einem ständig wechselnden Datenstrom geprüft ob die in CQL definierte Anfrage auf die derzeit bekannte Datenmenge greift. Tritt dies ein, so wird die Anfrage durchgeführt und die betroffenen Tupel aus dem Datenstrom genommen und zu einer Relation, welches wiederum ein aggregiertes oder komplexes Ereignis darstellt, verknüpft. Weiterhin wird ein Listening-Paradigma genutzt, wo die ausgewählten Ereignisse nun weiterverarbeitet werden bzw. benutzerdefinierte Funktionen oder Methoden aufgerufen werden. [11] Der Vorteil dieser Sprachen liegt in der weiten Verbreitung von SQL als Grundkenntnis der meisten Anwendungsentwickler, was es in vielen Fällen erleichtert Einstieg in diesen Sprachstil zu finden. Auch ist das Schema Datenströme zuerst in verknüpfte Relationen umzuwandeln bestens geeignet den Prozess der Aggregierung von Ereignissen darzustellen. Die Nähe zu Datenbankabfragesprachen ist verständlicherweise auch gut geeignet zur Datenbankintegration. Der Nachteil ist, dass die Erweiterung der Sprachen um die in vorherigen Kapiteln benannten Ereigniseigenschaften zum Teil zu komplizierten Ausdrücken führen kann, was den Entwickler letztenendes dazu zwingt seinen Code sehr strukturiert und sauber zu programmieren, diese Eigenschaften verhindern dabei zum Teil auch, dass sich Industriestandards etablieren können. Auch ist es eher unnatürlich, dass kontinuierliche Datenströme zu statischen Relationen zusammengefasst werden um interpretiert werden zu können und Bedingungen und Aktionen klar voneinander getrennt sind.

Der zweite Sprachtyp ist eine in regelbasierten Systemen verwendete Regelsprache. Regelsprachen sind dabei eigentlich ausgelegt für Fakten in einem sogenannten "Working-Memory" mit Hilfe von Bedingungen bestimmte Aktionen auszulösen. Dazu wird ein Wenn-Dann Konstrukt benutzt, welches bei Korrektheit einer definierten Bedingung von Fakten die in der Regel festgelegte Aktion ausführt. Um die Brücke zu ereignisbasierter Verarbeitung zu schlagen müssen die Ereignisse bei diesem Sprachtyp dabei als Fakten definiert werden und als solche in das Working-Memory eingefügt werden, über welche schließlich fortschreitend entschieden muss ob diese einem bestimmten Regelmuster entsprechen. Vorteile dieses Sprachtyps sind eine direkte Verknüpfung von Bedingung und Reaktion sowie eine deklarative und damit unter Umständen übersichtlichere Sprache. Der

Nachteil ist, dass für Ereignisverarbeitung notwendige Sprachkonstrukte wie Akkumulationen oder temporale Konstrukte über Fenster, zumindest wenn die Sprache urpsrünglich nicht für Ereignisverarbeitung gedacht ist, nicht nativ in der Sprache enthalten sind und erst in den Sprachumfang eingewoben werden müssen um die volle Funktionalität zu gewährleisten. Dies führt ebenfalls zu einer recht komplexen und unter Umständen schwer verständlichen Syntax und hoher Einarbeitungszeit.

Der letzte Typus ist der Kompositionsoperator basierend auf aktiven Datenbanksystemen. Hierbei werden aus einer Reihe von Ereignissen über zum Teil geschachtelte Sprachkonstrukte Kompositionen von selbigen Ereignissen erstellt. Da dies bereits dem Vorgang der Aggregation von Ereignissen entspricht ist dies ein guter Ansatz für Complex Event Processing. Ereignisse die aus der Ereignismenge genommen wurden werden dabei als "konsumiert" bezeichnet. Durch die Verknüpfung von mehreren Operatoren kann diese Variante allerdings zu einer gewissen Unübersichtlichkeit in der Definition der Kompositionen führen.[12]

Insgesamt lässt sich sagen, dass allen Typen ein inhärenter Hang zur Sprachkomplexität innewohnt, was im Hinblick auf die Schwierigkeit des eigentlichen Themenfelds,nämlich der Erkennung von komplexen Mustern eigentlich zu erwarten ist. Hilfreich wäre hier jedoch eine Standardisierung der Sprache, etwas bei dem sich zumindest im Zuge von Continous Query Languages zumindest allmählich etwas tut[13]. Letztendlich bleibt zu erwähnen das es bei der Evaluierung von Sprachtypen davon abhängt, wie die Zielgruppe die eine Sprache verwenden soll aussieht. Dabei sind Geschmäcker letztenendes doch verschieden.

Sprachparadigma in Drools:

Drools nutzt wie bereits gesagt ein regelbasiertes System, wobei die Sprache nicht ursprünglich für CEP vorgesehen wurde, sondern erst um CEP-Funktionalitäten erweitert wurde. Die Sprache selbst ist dabei recht leicht zu nutzen, solange keine zu komplizierten Sprachkonstrukte erstellt werden. Das Problem ist, dass alle Regeln grundlegend von der Engine gleich behandelt werden und es so schwer fällt einen übersichtlichen Ablauf über eine mehrfach gestaffelte Ereignishierarchie zu verwirklichen, zumindest wenn diese schlecht entworfen wurde.

Sprachparadigma in Esper:

Esper wurde direkt für die Entwicklung von CEP Anwendungen entworfen, der Sprachumfang ist ebenfalls ausserordentlich umfangreich. Zwar fällt der Einstieg dank der SQL ähnlichen Syntax leicht, ein genaues Studium des Referenzhandbuchs ist zur vollen Kenntnis der Sprache allerdings dringend notwendig. Ist Erfahrung mit Esper jedoch vorhanden, lässt die Sprache kaum etwas zu wünschen übrig. Statements können mit Listenern verknüpft werden, die asynchron auf Ereignisse reagieren. Statements und Listener können dabei lose gekoppelt werden, was eine gute Skalierung einer gesamten Softwarearchitektur erlaubt.

5.2 Ereignisdarstellung

5.2.1 Definition und Relation von Ereignissen

Eine Sprache sollte um die Relevanz von Ereignissen zu kennen möglichst wissen, welche Objekte im Modell genau nun Ereignisse sind und welche nicht. Zur einfacheren Benutzung von komplizierteren sprachlichen Konstrukten sollte die Sprache wissen welche Ereignisse aus einer Summe von einfachen Ereignissen aggregiert wurden oder welche Ereignisse dem Typ eines komplexen Ereignisses entspricht.

- Die schlichte **Definition** welche Objekte nun überhaupt **einfache Ereignisse** sind , um sie im Speicher von einfachen Objekten oder Fakten unterscheiden zu können. Auch ist notwendig zu wissen, welchen Typs sie sind.
- Die **Definition** welche Ereignisse **aggregierter oder komplexer** Natur sind.
- Die **Festlegung der Relationen** zwischen Ereignissen. Denkt man dies weiter, könnte es eine Sprache erlauben mit Hilfe einer einfachen Funktion Ereignisse automatisch zu aggregieren anstatt zuerst den Entwickler dazu zu nötigen umfangreiche sprachliche Konstrukte zu schreiben. Vorstellbar wäre zum Beispiel:

  ```
  declare SensorBehaviourEvent
  Role (aggregated from SensorEvent)
  aggregate on (sensorid)
  interval (1m)
  entry-point ("SensorBehaviorEventStream")
  ```

  Dies wäre eine Definiton eines aggregierten Ereignisses, wobei die Engine auch weiss, dass es eines ist. Sie könnte so automatisch auf einem bestimmten Zeitpunkt Ereignisse zusammenfassen und in die Laufzeitumgebung einfließen lassen, ohne dass der Entwickler dabei die eigentliche Sprache bemühen müsste. Dies würde gewaltig der Automatisierung dienen, und der Entwickler könnte sich dabei der Hauptaufgabe, welche die Mustererkennung von komplexen Ereignissen ist, widmen. Es ist dabei allerdings zu beachten, inwieweit das sprachliche Paradigma überhaupt solche Definitionen zulässt bzw. Wie weit die Engine solche umfassenden Definitionen überhaupt stützen kann.

Definition und Rollen von Ereignissen in Drools:

Ereignisse müssen innerhalb der .drl-Regeldatei für Drools mit

```
declare LoopDetectorEvent
    @role( event )
```

explizit bekannt gegeben werden, der Compiler kann keine Ereignisse On-the-Fly per Reflection zuordnen ohne diese vorher in der .drl-Datei definiert zu haben.

```
/*
declare CleanedLoopDetectorEvent
    @role( event )
    @expires( 5h )
    @timestamp ( eventtimestamp )
    @duration ( eventduration )
end
*/
```

führt zu:

```
Exception caught while executing action:
org.drools.reteoo.PropagationQueuingNode$PropagateAction@1d3cdaa
java.lang.ClassCastException: org.drools.common.DefaultFactHandle cannot be cast
to org.drools.common.EventFactHandle
```

Allerdings ist dies möglich für einfache Fakten, die keine Ereignisse sind, was darauf hindeutet das diese Funktionalität schlichtweg noch nicht implementiert worden ist.

Im Anwendungsfall wurden die Ereignisse als Java-Objekte vordefiniert und dann nur noch in die Regeldatei importiert. Die Drools-Regelsprache erlaubt es allerdings auch Ereignisse direkt inline in der Regeldatei zusammen mit ihren Attributen zu deklarieren:

```
declare LoopDetectorEvent
    @role( event )
    eventID : int
    sensorID : int
    velocity : double
    timestamp : java.util.Date
```

Abgesehen von der Rolle als Ereignisse führt Drools nicht weiter Buch über weitere Ereignistyp-Metadaten, zwischen einfachen Ereignissen und höheren Ereignissen in der Hierarchie wird also nicht unterschieden, was den Entwickler dazu zwingt Ereignisaggregationen von Hand zu vollziehen.

Die Unterscheidung von einfachen Fakten und Ereignissen führt Drools mit der

```
@role( event )    oder        @role( fact )
```

Annotation.

Definition und Rollen von Ereignissen in Esper:

Im Anwendungsfall wurden Ereignisse als Java Klassen definiert und mit

```
cepConfig.addEventType("CleanedLoopDetectorEvent",CleanedLoopDetectorEvent.class.
getName());
```

der Esper Laufzeitumgebung bekannt gegeben.

Esper erlaubt weiterhin Ereignisse In-Code zur Laufzeit zu definieren durch Anlegen einer Map:

```
Map LDEvent = new HashMap();
event.put("sensorid", sensorid);
event.put("velocity", velocity);
cepRT.sendEvent(event, "LoopDetectorEvent");
```

Für das neue Ereignis wird automatisch ein neuer Datenstrom unter dem Namen

"LoopDetectorEvent" erstellt, der durch CQL-Statements identifiziert und ausgelesen werden kann. Esper unterstützt noch weitere Ereignisdefinitionen unter anderem per XML und führt eine Ereignisdefinitions API, auf die im Zuge dieser Arbeit nicht näher eingegangen werden soll. Weitere Informationen sind enthalten in der Esper Dokumentation.

Esper kann zwar beliebig Ereignisse ineinander schachteln und so beinhaltende Untereeignisse abbilden, besitzt jedoch keine tieferen Metadaten zu Ereignisdefinition. Hier ist Esper auf Funktionen der Wirtssprache Java angewiesen.

5.2.2 Ereignisherkunft

Die Ereignisherkunft sollte von der Engine bestimmbar und unterscheidbar sein.Vorzugsweise sollte die Plattform fähig sein mehrere Ereignisse des gleichen Typs aus unterschiedlichen Quellen unterscheiden zu können. Dies würde sowohl den Speicherverbrauch für nicht mehr benötigte Ereignisse senken, wobei nur Ereignisse aus bestimmter Quelle für eine Bedingung in Betracht kommen könnten, so dass nicht mehr benötigte Ereignisse desselben Typs aber aus unterschiedlicher Quelle aus dem Speicher genommen werden können, als auch unterstützend für eine lose Kopplung der unterschiedlichen Ereignisgeneratoren wirken. Dies könnte eine bessere Spezifizierung des Architekturentwurfes erlauben.

Ereignisherkunft in Drools:

Um erkennen zu können aus welchem Ereignisstrom nun welche Ereignisse stammen nutzt Drools ein **Entry-Point** lautendes Konzept.

```
when
  $p1 : ProblemEvent()  from entry-point "ProblemEventStream"
then
```

Jedes Ereignis behält wenn es über einen Entry-Point in das Working-Memory geleitet wird die Referenz von welchem es ursprünglich einmal stammte. Dies erlaubt es Ereignisse vom gleichen Typ aus mehreren unterschiedlichen Eingangspunkten stammend voneinander in den Regeln zu trennen. Dies ist natürlich ein Vorteil sowohl zur Übersicht, als auch für die Engine um entscheiden zu können welche Ereignisse noch im Speicher gebraucht werden und welche nicht.

```
when
  $p1 : ProblemEvent()  from entry-point "ProblemEventStream"
then
.
.
.
when
  $p1 : ProblemEvent()  from entry-point "TheOtherProblemEventStream"
then
```

Ereignisse die durch unterschiedliche Entry-Points in das Working-Memory gelangen, können natürlich trotzdem ohne Probleme über ein enthaltenes Attribut verknüpft werden. Siehe hierzu Regel 3 und 4 für eine Verknüpfung über das Attribut sensorID.

Eine Schwäche ist, dass Drools zwar erkennt, welche Ereignisse über welchen Entry-Point hereingekommen sind, nicht aber aus der Engine selber heraus erlaubt Ereignisse in das Working-Memory einzufügen. Zum Einfügen von Ereignissen muss immer eine externe Java-Methode aus der Regelbasis heraus aufgerufen werden, wie auch in der Beispielanwendung gezeigt.

In TrafficRules.drl:

```
EventCreator.InsertCleanedLoopDetectorEvent($p);
```

In EventCreator:

```
cleanedEntryPoint.insert(clean);
```

Die Insert-Methode des Drools Working-Memory kann zwar auch über die Regeldatei aufgerufen werden, es gibt allerdings keine Möglichkeit hier eine Referenz auf einen Entry-Point (wie den `cleanedEntryPoint` oben.) zu bekommen. Da der Entry-Point erst zur Laufzeit der Anwendung selbst erstellt wird müsste die Engine bei Lesen der Knowledge-Base hier ein referenzierbares Objekt von sich aus erstellen, wenn solch eine Funktionalität in der Zukunft eingefügt werden soll. Zurzeit ist dies jedoch nicht möglich, was den Benutzer zwingt ein Objekt bereit zu stellen, welches eine statische Referenz auf einen definierten Entry-Point hält, sowie eine statische Methode zum Zugriff. (In diesem Fall der `EventCreator`.)

Ereignisherkunft in Esper:

Sobald die Definition eines Ereignisses an Esper mit

```
cepConfig.addEventType("SensorBehaviorEvent",SensorBehaviorEvent.class.getName())
;
```

bekannt gemacht wurde, wird der Datenstrom unter dem selben Namen mit Hilfe der from-Klausel des CQL-Statements wieder ausgelesen.

```
from CleanedLoopDetectorEvent.std:groupby(sensorid).win:time_batch(5 min)
```

obiges Beispiel zeigt den relevanten Codeausschnitt aus Statement 2. Die from Klausel kann nicht nur verwendet werden um einfach aus einem oder mehreren bennanten Datenströmen zu lesen, sondern es gibt auch den sehr mächtigen `from pattern [...]` Ausdruck, der es ermöglicht Mustererkennung über mehrere Datenströme hinweg zu führen. Auf Möglichkeiten dieses Ausdrucks wird noch in kommenden Kapiteln eingegangen.

Von der Java-Wirtssprache aus wird in ein Ereignis mit dem Befehl:

```
cepRT.sendEvent(LDEvent);
```

an die Esper-Runtime versendet. Esper kann des weiteren Ereignisse direkt über Statements versenden, hier ein etwas verändertes Statement 2.

```
insert into SensorBehaviorEvent(sid,averagevelocity,Averageevaled)
select sensorid, avg(velocity) ,utils.StaticMethods.evalAverage(avg(velocity))
from CleanedLoopDetectorEvent.std:groupby(sensorid).win:time_batch(5 sec)
group by sensorid
```

die insert into Klausel nimmt sich hierbei die gezogenen Werte der select Klausel, übernimmt dabei sogar bereits automatisch ihren Datentyp und schert sich auch nicht um typographische Feinheiten, was alles in allem recht komfortabel zu bedienen ist.

5.2.3 Zeitkonzept

Da der Ereignisstrom ein unendlicher und auch andauernder Strom an Daten ist, ist es notwendig sowohl für Vergleiche wie auch für Aggregationen von Ereignissen zu wissen, wann eigentlich ein Ereignis geschehen ist. Eine ereignisverarbeitende Laufzeitumgebung benötigt deswegen:

- **Ein Zeitkonzept** wie Ereignisse eigentlich zeitlich sortiert werden sollen. Die Engine soll deswegen die Verarbeitung von Zeitstempeln inhärent unterstützen. Ereignisse müssen Zeitstempel haben und die Engine muss erkennen können wofür diese dienen, sodass Vergleiche auf diese Zeitstempel unterstützt werden. Dabei soll es möglich sein Ereignissen feste Zeitstempel benutzerseits zuzuweisen wie auch das Ereignisse bei "Bekanntgabe" in einen Ereignisstrom von der Engine selbst einen Zeitstempel zugewiesen wird. Aggregierte und komplexe Ereignisse sollen dabei nicht nur einen Zeitstempel haben, sondern auch einen Zeitintervall, also einen Startzeitpunkt und einen Endzeitpunkt für den dieses Ereignis "aktiv" war.
- Die Engine soll dem Benutzer erlauben **Kontrolle über die verwendete Systemzeit** auszuüben, so ist es denkbar die Uhrzeit aus einem externen System zu nutzen, anstatt die Systemzeit des lokalen Systems auf dem die Engine läuft(beispielsweise um Daten aus einer anderen Zeitzone korrekt zu verarbeiten). Schön wäre es weiterhin dem Benutzer Kontrolle über den Fluss der Zeit zu Testzwecken zu ermöglichen.

Zeitkonzept in Drools:

Zeitkonzept :

Ereignisse werden, wenn sie in das Drools Working-Memory eingefügt werden implizit mit einem Zeitstempel versehen, womit sie im Datenstrom geordnet werden. Diese Zeitstempel entsprechen dabei dem Typ java.util.Date. Will der Entwickler auf diese Zeitstempel auch außerhalb der Regelsprache zugreifen muss er dies der Engine mitteilen, welches Ereignisattribut aus der Java-Klasse den Zeitstempel repräsentieren soll:

TrafficRules.drl:

```
declare LoopDetectorEvent
    @timestamp ( eventtimestamp )
end
```

EventCreator:

```
Date startStamp = aggregated.get(0).getEventtimestamp();
System.out.println(startStamp);
```

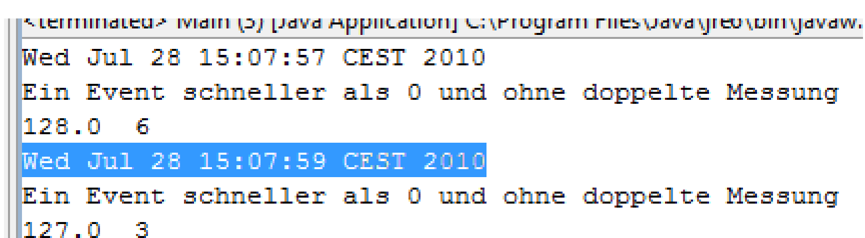

Abbildung 18: Ein von Drools automatisch zugewiesener Zeitstempel

Ereignisse in Drools müssen nicht unbedingt nur einen Zeitstempel haben sondern können auch eine

Dauer zugewiesen bekommen, aus der sich dann der Endzeitstempel, also der Zeitstempel der aussagt wie lange die Tätigkeit, welche vom Ereignis dargestellt wurde, andauerte, berechnet.

```
@duration ( eventduration )
```

Die duration entspricht dem Typ long und im Zuge der Nutzung von temporalen Operatoren wird diese auf den timestamp aufgerechnet um festzustellen wie lange ein Ereignis andauerte.

Kontrolle über die verwendete Systemzeit:

Die Uhr welche Drools verwendet wird per Standardeinstellung der Systemzeit entnommen, auf dem die Anwendung läuft. Drools nennt diese Konfigurationseinstellung `"realtime"`. Angegeben werden kann dies mit:

```
KnowledgeSessionConfiguration conf =
KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
conf.setOption( ClockTypeOption.get( "realtime" ) );
```

Drools erlaubt ausserdem die Nutzung einer Pseudo-Uhr zu Testzwecken. Der Anwender kann Ereignisse in das Working-Memory einfügen und dann gezielt die Uhr voranschreiten lassen und weitere Ereignisse einfügen. So hat man die Kontrolle über den gesamten Zeitfluss, was sicherlich ein probates Mittel ist zu Testzwecken und Debugging von Regeln.

```
KnowledgeSessionConfiguration config =
KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
config.setOption( ClockTypeOption.get("pseudo") );


clock.advanceTime(4, TimeUnit.HOURS );
```

Zeitkonzept in Esper:

Zeitkonzept :

Esper besitzt eine komplette Implementierung für zeitliche Abhängigkeiten, jegliche Ereignisse, die die Laufzeitumgebung betreten werden intern mit einem Zeitstempel versehen und geordnet in der Engine abgelegt. Standardmäßig verwendet Esper dabei die auf der ausführenden Maschine laufende Systemzeit.

Der Zeitstempel wird von Esper explizit selbst kontrolliert und es ist im Gegensatz zu Drools nicht möglich, Ereignisse mit einem externen Zeitstempel via Attribut zu versehen. Ein einziger Eingriff in die zeitliche Ereignisreihenfolge ist mit dem `ext:time_order` View erlaubt:

```
insert into DelayedDetectorEvents
select rstream * from LoopDetectorEvent.ext:time_order(eventtimestamp, 20 sec)
```

Das Attribut `eventtimestamp` ist im LoopDetectorEvent enthalten und gibt den Zeitstempel des Ereignisses an. Die 20 Sekunden ist der Zeitpunkt, die das Ereignis von der Engine zurückgehalten werden sollen. Differiert die derzeitige Engine-Zeit 20 Sekunden von dem angegebenen Zeitstempel, sprich sind 20 Sekunden in der Engine-Zeit nach dem Attribut `eventtimestamp` vergangen, wird das Ereignis in den DelayedDetectorEvents Strom eingefügt. So kann manuell eine Ordnung in dem Ereignisstrom eingefügt werden sowie auf möglicherweise verhinderte Ereignisse von Sensoren gewartet werden, sodass Ereignisse in korrekter Reihenfolge in die Engine gehen.

Kontrolle über die verwendete Systemzeit:

Genau wie Drools kann in Esper die interne Uhr direkt vom Anwender kontrolliert werden. Dazu muss in der Konfiguration die interne Uhrzeit ausgeschaltet werden.

```
Configuration conf = new Configuration();
conf.getEngineDefaults().getThreading().setInternalTimerEnabled(false);
```

Gefolgt von einem CurrentTimeEvent, welches die zu benutzende Zeit in Millisekunden angibt.

```
long timeInMillis = System.currentTimeMillis();
CurrentTimeEvent timeEvent = new CurrentTimeEvent(timeInMillis);
epService.getEPRuntime().sendEvent(timeEvent);
```

Der Fluss der Zeit kann nun explizit gesteuert werden und Ereignisse kontrolliert eingefügt werden. Es ist natürlich auch vorstellbar, dass mit dieser Methode externe Uhren genutzt werden können, wenn auch etwas kompliziert.

```
cepRT.sendEvent(it.next());
cepRT.sendEvent(new CurrentTimeEvent(1000));
cepRT.sendEvent(it.next());
```

5.3 Sprachumfang

5.3.1 Verknüpfungsoperatoren

Die Unterstützung von Verknüpfungsoperatoren in der Sprache ist eine Grundvorraussetzung zur korrekten Ereignisverarbeitung und sollte dementsprechend wirklich in jeder ereignisverarbeitenden Sprache, die sich so nennt, vorhanden sein.

- Vorhandensein des **ODER-Operators**, der es erlaubt zu prüfen ob zumindest ein Zustand einer bestimmten Menge von Zuständen eingetreten ist.
- Vorhandensein des **UND-Operators**, der es erlaubt zu prüfen ob mehrere Zustände zur selben Zeit eingetreten sind.
- **Weitere Operatoren** sind etwas syntaktischer Zucker und können mit den obigen drei Operatoren auch kreiert werden, dennoch ist es nett wenn eine Sprache noch NAND, NOR und XOR direkt unterstützt.

Verknüpungsoperatoren in Drools:

Aufgrund seiner Identifikation als regelbasierte Sprache besitzt Drools einen ganzen Schwall an zum Teil sogar exotischen Bedingungsoperatoren für verschiedenste Verwendungszwecke.

Schauen wir noch einmal auf Regel 1 aus dem Anwendungsfall:

```
when
$p : model.LoopDetectorEvent (velocity > 0 && < 280, $i : sensorid) from
entry-point "LoopDetectorEventStream"
not (  model.LoopDetectorEvent (sensorid == $i , this after [0,500ms] $p) from
entry-point "LoopDetectorEventStream")
```

Zu sehen ist bereits die Benutzung des Verknüpfungsoperators **&&** (UND) welcher die Attributbedingung der Geschwindigkeit zwischen 0 und 280 liegen muss zusammenfasst. Der Verknüpfungsoperator || (ODER) wird ebenfalls von Drools gestützt. Die beiden jeweils atomaren Bedingungen startend aus Zeile 2 und 4 sind bereits implizit mit einer UND-Bedingung verknüpft, beide atomaren Bedingungen müssen wahr sein, damit die Regel feuert.

Der **,** Operator zwischen

`sensorid == $i` und `this after [0,500ms] $p`

wird von der Sprache ebenfalls wie ein UND behandelt, besitzt jedoch eine niedrigere Priorität und damit niedrigere Bindung als der der konventionelle **&&** Operator.

Der Ausdruck:

```
velocity > 0 && < 280, $i : sensorid
```

verknüpft zuerst die `&&` und sobald dieser wahr ist, den `,` Operator mit dem vorherigen Ausdruck.

Drools erlaubt allerdings auch noch explizite Verknüpfungen von Bedingungen:

```
when
$p : model.LoopDetectorEvent (velocity > 0 && < 280, $i : sensorid) from
entry-point "LoopDetectorEventStream"
and
not (  model.LoopDetectorEvent (sensorid == $i,this after [0,500ms] $p) from
entry-point "LoopDetectorEventStream")
.
.
.

when
$p : model.LoopDetectorEvent (velocity > 0 && < 280, $i : sensorid) from
entry-point "LoopDetectorEventStream"
or
not (  model.LoopDetectorEvent (sensorid == $i,this after [0,500ms] $p) from
entry-point "LoopDetectorEventStream")
```

Somit lassen sich in Drools komplexe Bedingungen sehr einfach zusammenfassen, ohne dass der Überblick schnell verloren geht.

Verknüpfungsoperatoren in Esper:

Operatoren zur Verknüpfung von komplexen Mustern sind in Esper ebenfalls wie in Drools vorhanden. Esper führt hier noch einige weitere Operatoren ein, die im Abschnitt 5.4.1 und 5.4.2 vorgestellt werden. Die Operatoren werden im `pattern[...]`-Ausdruck verwendet.Die Keywords für die Operatoren sind: `or` , `not` und `and` Hier soll noch einmal kurz ein Beispiel einer UND-Verknüpfung gezeigt werden, stammend aus Statement 3:

```
where s1.evaledAverage = model.AverageSpeed.HIGH and s1.evaledAverage =
model.AverageSpeed.LOW"
```

5.3.2 Temporale Operatoren

Sobald eine Ereignisverarbeitende Sprache ein Zeitkonzept hat muss sie natürlich auch Operatoren zur Prüfung von Zeitstempeln bieten.

- Die Unterstützung von auch komplexeren **temporalen Operatoren** zum Vergleich von Zeitstempeln muss her. Allen definiert dabei die folgenden temporalen Operatoren für zeitliche Vergleiche [14] :

Relation	Symbol	Symbol for Inverse	Pictoral Example
X *before* Y	<	>	XXX YYY
X *equal* Y	=	=	XXX YYY
X *meets* Y	m	mi	XXXYYY
X *overlaps* Y	o	oi	XXX YYY
X *during* Y	d	di	XXX YYYYYY

Abbildung 19: Von Allen definierte temporale Relationen sowie ihre Operatoren[14]

Dabei gilt, das eine Ereignisverarbeitende Engine sowohl die oben dargestellte zeitliche Relation bewerten können muss, als auch die Absenz von einer zeitlichen Relation.(Kapitel 5.4.1) Der Sprachumfang von Sequenz-Operatoren ist dabei bereits in temporalen Operatoren enthalten, Sequenzoperatoren sind schlichtweg temporale Operatoren, die keine nähere zeitliche Definitionen beinhalten, wann genau eine Bedingung greifen soll. Ein komplexerer temporaler Operator könnte prüfen, ob ein spezifisches Ereignis seinen Zeitstempel genau zwischen 3 bis 4 Sekunden nach einem anderen Ereignis hat. Ein Sequenzoperator prüft nur, ob ein Ereignis nun danach existiert oder nicht.

Temporale Operatoren in Drools:

Drools unterstützt, wie in ihrer Dokumentation selbst behauptet, alle von Allen definierten Operatoren. Innerhalb der Drools Regel 3 ist die Benutzung eines solchen Operators zu sehen:

```
$se2 : SensorBehaviorEvent(sensorid == $sid,(evaledAverage == AverageSpeed.HIGH
&& $se1.evaledAverage == AverageSpeed.LOW),  this after[0,1m] $se1  ) from entry-
point "SensorBehaviorEventStream"
```

Der Operator `this after` wird hier benutzt zu prüfen, ob ein SensorBehaviorEvent im Zeitraum bis zu einer Minute nach einem $se1 betitelten Ereignis eintrifft. Der Operator könnte auch ohne die geklammerten zeitlichen Beschränkungen verwendet werden, dann wird lediglich geprüft ob das Ereignis danach stattgefunden hat.

Folgend sind die von Drools Fusion verwendeten temporalen Operatoren für Ereignisse:

- `Event1 this after[...] Event2`

  Prüft ob Ereignis 1 nach Ereignis 2 stattgefunden hat.

- `Event1 this before[...] Event2`

  Prüft ob Ereignis 1 vor Ereignis 2 stattgefunden hat.

- `Event1 this during[...] Event2`

  Prüft ob Ereigniss 1 begonnen und geendet hat während Ereignis 2 stattfand.

- `Event1 this includes[...] Event2`

  Prüft ob Ereigniss 2 begonnen und geendet hat während Ereignis 1 stattfand.

- `Event1 this finishes[...] Event2`

  Prüft ob Ereignis 1 nach Ereignis 2 begonnen hat, sie beide aber zur selben Zeit geendet haben

- `Event1 this finishedby[...] Event2`

  Prüft ob Ereignis 2 nach Ereignis 1 begonnen hat, sie beide aber zur selben Zeit geendet haben

- `Event1 this meets[...] Event2`

  Prüft ob Ereignis 1 zum selben Zeitpunkt endet, wie Ereignis 2 beginnt.

- `Event1 this metby[...] Event2`

  Prüft ob Ereignis 2 zum selben Zeitpunkt endet, wie Ereignis 1 beginnt.

- `Event1 this overlaps[...] Event2`

  Prüft ob Ereignis 1 begonnen hat bevor dies Ereignis 2 getan hat und auch seinen Endzeitpunkt hat bevor Ereignis 2 geendet hat

- `Event1 this overlappedby[...] Event2`

  Prüft ob Ereignis 2 begonnen hat bevor dies Ereignis 1 getan hat und auch seinen Endzeitpunkt hat bevor Ereignis 1 geendet hat

- `Event1 this starts[...] Event2`

  Prüft ob Ereignis 1 zur selben Zeit begonnen hat als Ereignis 2, allerdings vor Ereignis 2 geendet hat

- `Event1 this startedby[...] Event2`

  Prüft ob Ereignis 2 zur selben Zeit begonnen hat als Ereignis 1, allerdings vor Ereignis 1 geendet hat

- `Event1 this coincides[...] Event2`

  Prüft ob Ereignis 1 zur selben Zeit geschehen ist, als Ereignis 2 (sowohl Zeitpunkt, als auch wenn angegeben zusätzlich Dauer)

Die Liste der temporalen Operatoren die Drools kennt ist also geradezu gewaltig und es fällt teilweise schwer, sich vorzustellen wozu diese Operatoren alle genutzt werden könnten. Da in der Drools Fusion Dokumentation allerdings ein signifikanter Aussschnitt eben diesen Operatoren gewidmet ist, soll dies hier dementsprechend gewürdigt werden.

Temporale Operatoren in Esper:

Esper kennt keine temporalen Operatoren wie Drools, besitzt jedoch ein genauso mächtiges Konzept mit dem Sequenzoperator `->` und dem where timer:within() Operator, vorgestellt im nächsten Kapitel.

5.3.3 Sequenzoperatoren

Die Nutzung von temporalen Operatoren ist eine Möglichkeit zur Prüfung von temporalen Beziehungen zwischen Ereignissen. Besitzt eine Laufzeitumgebung jedoch Wissen über den Zeitpunkt, wann Ereignisse sie erreicht haben, so ist es ihr auch möglich diese in eine sequentielle Reihenfolge zu bringen. Eine Sequenzierung könnte sogar zur einfacheren Abarbeitung von Ereignissen dienlich sein. Mit Hilfe eines Sequenzoperators wie beispielsweise **->** könnten Bedingungen erzeugt werden, die nicht die zeitliche Komponente eines Ereignisses berücksichtigt, sondern lediglich den Punkt, an dem Ereignisse in den Datenstrom eingeflossen sind, was es erlaubt eine Trennung von Zeitstempel des Ereignisses sowie Zeitpunkt des Fließens in die Laufzeitumgebung zu führen. Merken wir uns: Ein Ereignis stellt wohlmöglich die Abbildung einer Tätigkeit aus der realen Welt dar. Diese Tätigkeit hat einen Zeitpunkt der eben nicht notwendigerweise (höchstunwahrscheinlich sogar) den selben Zeitpunkt darstellt, wie das Ereignis in die Laufzeitumgebung gekommen ist. Ist ein Ereignis erst an einem Sensor erstellt worden, so wird sehr wahrscheinlich Zeit vergangen sein bis dieses Ereignis seinen Weg zur Ereignisverabeitenden Station gefunden hat. Aus diesem Grund ist es durchaus vertretbar eine Trennung zwischen temporalen Operatoren sowie sequentiellen Operatoren, die eine Reihenfolge im Speicher betrachten durchzuführen.

Nicht zuletzt ist ein sequentieller Operator ein sehr mächtiges Konstrukt, der es einfach macht Ereignisse in Reihenfolgen zu betrachten, ohne sich dabei um zeitliche Bedingungen Gedanken machen zu müssen.

Sequenzoperatoren in Drools: Drools führt keine Sequenzoperatoren, jegliche Überprüfungen auf Ordnungen der Ereignisse müssen in der Drools Regelsprache mit dem allerdings großen Umfang der temporalen Operatoren getätigt werden.

Sequenzoperatoren in Esper: Die Sequenzoperatoren in Esper können wohl als eines der mächtigsten Werkzeuge zur Ereigniserkennung des Sprachumfangs bezeichnet werden. Der Sequenzoperator

```
->
```

wird innerhalb des Pattern-Operator verwendet und legt fest, nach welchem zeitlichen Muster Ereignisse ankommen müssen. Zur Beispielverwendung einen Ausschnitt aus Statement 3.

```
from pattern [ every(s1=SensorBehaviorEvent -> s2=SensorBehaviorEvent(sensorid =
s1.sensorid)) where timer:within(1 min)]
```

Der Ablauf des des obigen Statements wurde ja bereits hinreichend geklärt. Schlussendlich bleibt nur zu sagen, das der Sequenzoperator in Verbindung mit dem `timer:within(...)` Operator es ermöglicht sehr einfach komplizierte Muster zusammenzufassen und dies nach einer simplen und benutzernahen Logik.

5.3.4 Kausale Operatoren

Luckham schreibt in [3] das neben der zeitlichen Beziehung von Ereignissen die kausale Beziehung ebenfalls zu den wichtigen Beziehungstypen zwischen Ereignissen gehört. Beide Beziehungen sind dabei ineinander übergreifend. Luckham beschreibt dabei das Ursache-Zeit Axiom, nämlich, dass jedes Ereignis, welches ein anderes erzeugt hat auch zeitlich vor dem erzeugten Ereignis stattgefunden haben muss. Somit ist es logisch das kausale Operatoren bereits im Funktionsumfang von temporalen Operatoren enthalten sind. Die Menge "ist Grund von" ist mathematisch eine Untermenge von "ist geschehen vor" und somit ein stärker selektierender Vergleichsoperator. Somit macht es durchaus Sinn:

- Die Deklaration eines **Kausalitäts-Vektors**, also die Menge von Ereignissen die zur Entstehung von einem spezifischen Ereignis geführt hat, zu erlauben. Die Engine sollte dabei wissen, welcher Datentyp auf diesen Beziehungstyp hinweist und diesen bei Erzeugung eines aggregierten Ereignisses auch einfügen. Dabei sollte sie fähig sein transitive Abhängigkeiten zwischen Ereignissen zu erkennen.
- **Kausale Operatoren** zum vergleichen dieser Relation. Dabei soll sowohl "X ist Grund von Y" als auch "X ist nicht Grund von Y" unterstützt werden.

Kausalität und kausale Operatoren in Drools:

Ein eigentlicher Kausalitäts-Vektor wird in Drools nicht unterstützt und kann auch nicht per Annotation deklariert werden, will der Entwickler einen solchen erzeugen ist er auf die Funktionen der Java-Wirtssprache angewiesen. Ein Beispiel wie so etwas gemacht wird, kann in Regel 2 gesehen werden.

```
$ev : ArrayList() from collect(CleanedLoopDetectorEvent(sensorid == $i) over
window:time(50s) from entry-point "CleanedLoopDetectorEventStream")
```

Die `collect` Funktion des normalen Drools Expert Sprachumfangs kann zum Glück die Erstellung einer Liste durchführen, was es zumindest leicht macht in Regeln die Sammlung von auslösenden

Ereignissen zu führen.

Der Operator `memberOf` kann prüfen, ob ein Ereignis Teil einer erstellten Liste ist, was es in Verbindung mit obigem Code ermöglicht, zumindest um die Ecke Listen von verursachenden Ereignissen zu führen.

```
$SBEvent( $p memberOf $ev )
```

Kausalität und kausale Operatoren in Esper:

Wie bereits in Kapitel 5.2.1 angemerkt besitzt Esper keine umfangreichen Methoden zur Ereignisdefinition, was jegliche Kausalität einschließt. Soll so etwas verwirklicht werden, muss man wie auch ähnlich in Drools verursachende Ereignisse aus dem Strom auslesen und selbst per Hand Buch darüber führen, welche Ereignisse nun aus welchen entstanden sind.

5.3.5 Fenster

Bei einem grossen Strom an Ereignissen ist es oftmals notwendig die Menge von Ereignissen die betrachtet werden sollen von vornherein zu verringern. Einerseits um die Menge von Ereignissen über die die Laufzeitumgebung räsonieren muss zu senken(und damit auch den Aufwand), andererseits weil von Benutzerseite oft nur eine Menge von Ereignissen über einen bestimmten Zeitraum oder nur die letzten paar stattgefundenen Ereignisse von Interesse zur Betrachtung sind. Die Engine muss deswegen ein sogenanntes "Fenster" erlauben. Fenster legen dabei fest welche Ereignisse für die Bedingung von Ereignisregeln relevant sind und schränken so alle Ereignisse auf die Menge von Interessanten ein. Dies stellt ein Grundkonzept von Ereignisverarbeitung dar. Die Kriterien für Fenster sind deswegen:

- **Zeitbasierte Fenster**, die es erlauben über einen bestimmten Zeitraum Ereignisse zu sammeln. Dabei sowohl vor "jetzt" als auch nach oder vor Geschehen eines bestimmten Ereignisses.
- **Längenbasierte Fenster**, die es erlauben eine bestimmte Menge von Ereignissen zu sammeln. Dabei ist es wünschenswert sowohl die letzten Ereignisse vor "jetzt" zu sammeln, als auch Ereignisse nach oder vor Geschehen eines bestimmten Ereignisses.
- Untersützung von "**Sliding Windows**".(Nicht zu verwechseln mit der Definition von Sliding-Windows aus dem Netzwerkprotokollbereich.) Dies sind Fenster, welche kontinuierlich bei Veränderung ihres Betrachtungskriteriums vorangeschoben werden. Das Betrachtungskriterium kann dabei die Zeit sein, als auch die letze Menge an stattgefundenen Ereignissen. Betrachtet man die Menge an stattgefundenen Ereignissen und legt dabei eine Fenstergrösse von 10 Ereignissen fest, so wird das Fenster vorangeschoben wenn ein elftes Ereignis in das Fenster eintritt. Das erste Ereignis wird dann aus der Betrachtung herausgenommen, das Fenster verschiebt sich also. Selbiges kann auch über einen Zeitraum geschehen, das Sliding Window verschiebt sich also immer dann, wenn die Systemzeit voranschreitet, Ereignisse die älter als das die definierte Fensterzeit sind verschwinden also aus der Betrachtung.

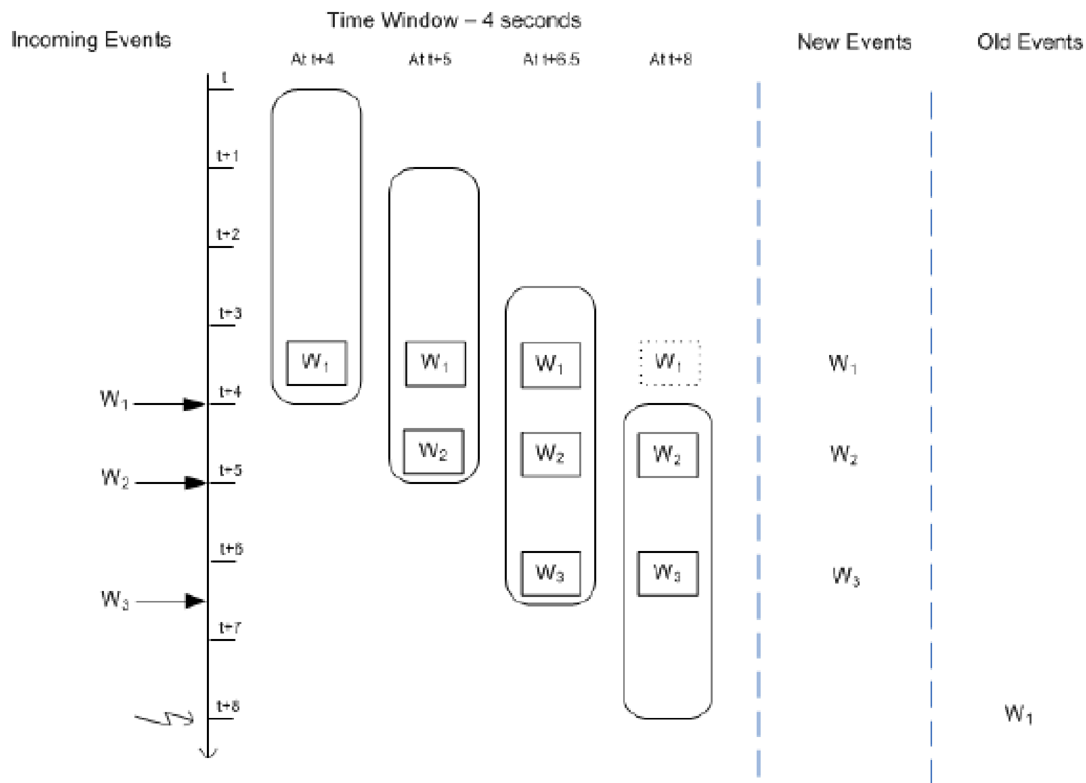

Abbildung 20: Beispiel eines zeitbasierten Sliding Windows, übernommen aus der Esper Dokumentation

- Unterstützung von "**Tumbling Windows**". Das sind Fenster die einen bestimmten Startpunkt haben und Ereignisse von dort an sammeln. Das Sammeln von Ereignissen findet dabei solange statt bis ein Sättigungskriterium erfüllt worden ist. Sobald dies stattgefunden hat wird der Inhalt des Fensters geleert und das Sammeln findest erneut von vorne statt. Das Sättigungskriterium kann dabei wieder das Vergehen einer Zeitdauer oder Überschreiten einer bestimmten Menge an Ereignissen sein. Vorstellbar wäre zum Beispiel das Sammeln von Ereignissen über exakt einen Monat und nach Monatsüberschreitung wird die Sammelmenge im Fenster geleert und das Sammeln beginnt erneut von vorne.

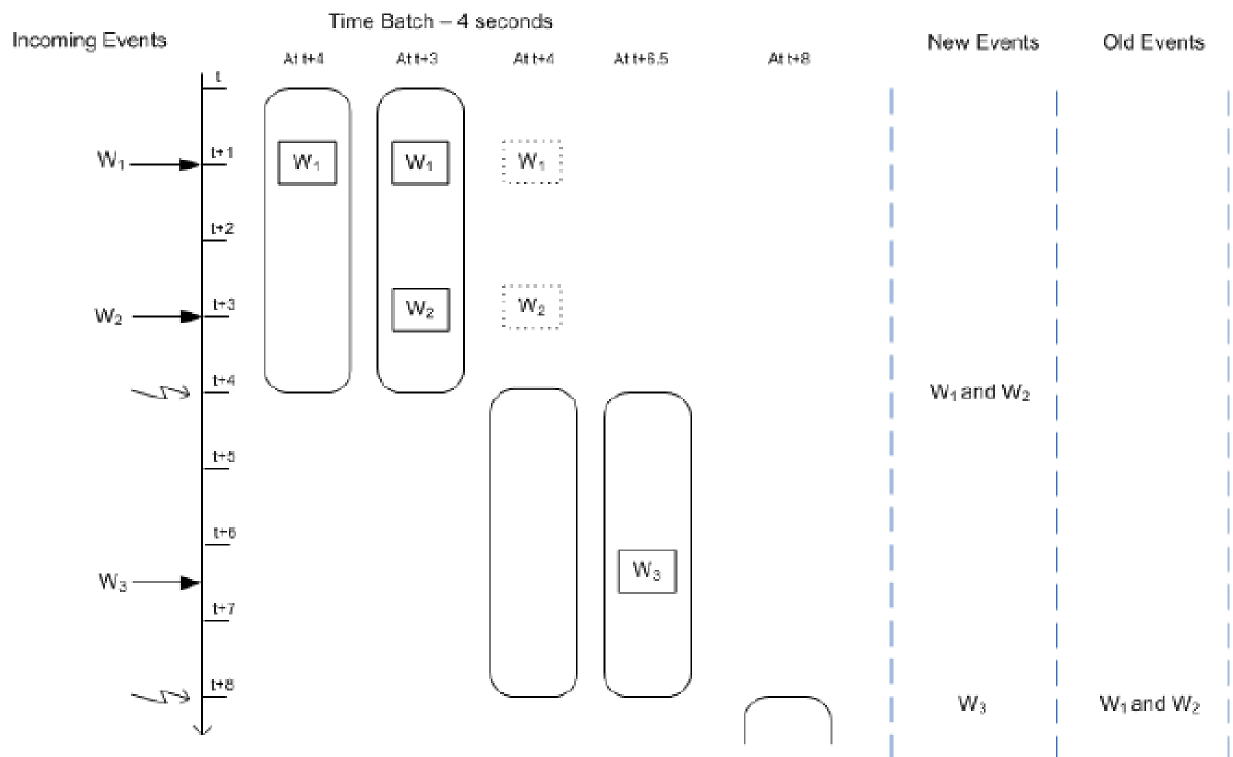

Abbildung 21: Beispiel eines zeitbasierten Tumbling Windows, übernommen aus der Esper Dokumentation

- Unterstützung von **komplexeren Fenstern**, Beispielsweise das Sammeln von Ereignissen mit einem bestimmten Datenwert kleiner als ein Kriterium oder sammeln von Ereignissen mit den höchsten oder niedrigsten Datenwerten.

Fenster in Drools:

Drools besitzt eine Implementation von Sliding Windows für die Betrachtung von sowohl chronologischen Fenstern als auch mengenbasierten Fenstern. Das Sliding Window muss dabei in Kombination mit einer Ereignisakkumulierung verwendet werden, da Drools im Gegensatz zu einer CQL-Sprache mit Listenern natürlich keinen Ausgabestrom kennt, sondern nur Entscheidungen über Ereignisse im Working-Memory. Eine Verwendung eines solchen Sliding Windows wurde in Regel 2 durchgeführt.

```
when
      from entry-point "CleanedLoopDetectorEventStream", average($velocity))
      $ev : ArrayList() from collect(CleanedLoopDetectorEvent(sensorid == $i)
      over window:time(5m) from entry-point "CleanedLoopDetectorEventStream"

then
      EventCreator.InsertSensorBehaviourEvent($s,$ev,$average);
end
```

Der Regelinterpreter schaut sich bei Benutzung des Fensters alle CleanedLoopDetectorEvents seit den letzten zwei Minuten an. Sind keine weiteren Bedingungen im Dann-Konstrukt festgelegt worden feuert diese Regel kontinuierlich im Stream-Modus bei Eintreffen eines neuen Ereignisses und erzeugt ein akkumulierendes Ereignis. Neben der Verwendung von zeitlichen Fenstern kennt

Drools Fusion auch längenbasierte Fenster mit

```
over window:length(20)
```

Weitere Fensterimplementierungen ausser Sliding Windows sind in Drools Fusion nicht vorhanden, man kann sich aber durchaus vorstellen, dass diese noch in zukünftigen Releases eingefügt werden, da logisch nichts gegen die Einführung weiterer Fenster-Funktionalität sprechen würde.

Eine weitere Kleinigkeit, die aufgefallen ist, ist dass nicht gewartet wird bis das Fenster zumindest einmal hinreichend gefüllt ist, die Drools-Regel wird hingegen sofort gefeuert. Anstatt dass bei einem Längenfenster von 20 auch gewartet wird, bis 20 Ereignisse vorhanden sind, hindert eine Fensterakkumulation den Regelinterpreter anscheinend nicht daran, die Wenn-Bedingung als wahr auszugeben, sobald alle anderen Bedingungen stimmen, auch wenn noch nicht genügend Ereignisse vorhanden sind. Im Zweifelsfall muss der Entwickler also immer noch zeitliche Abläufe selbst koordinieren und die Inferenzmaschine daran hindern frühzeitig zu feuern.

Fenster in Esper:

Esper bietet sowohl Unterstützung von Sliding Windows, als auch von Tumbling Windows, im Esper Fachjargon auch "Batch Windows" genannt, sowohl für Ereignismengen, als auch für zeitliche Dauer. In Statement 2 wird solch ein zeitliches Batch-Window verwendet:

```
select sensorid, avg(velocity) ,StaticMethods.evalAverage(avg(velocity))
from CleanedLoopDetectorEvent.std:groupby(sensorid).win:time_batch(5 min)
group by sensorid
```

Ein Sliding Window auf Länge wird mit dem Befehl:

```
select sensorid, avg(velocity) ,StaticMethods.evalAverage(avg(velocity))
from CleanedLoopDetectorEvent.std:groupby(sensorid).win:length(4)
group by sensorid
```

genutzt. In diesem Fall werden immer wieder bei jedem Auslösen des Listeners, was wiederum bei jedem Eintreffen eines CleanedLoopDetectorEvents geschieht, da keine weiteren Bedingungen definiert worden sind, die Geschwindigkeit der letzten 4 CleanedLoopDetectorEvents gemittelt und weitergereicht.

Esper besitzt ebenfalls noch eine Vielzahl an komplexeren Fenstern, von denen nur einige hier genannt werden sollen, für den Rest soll noch hier noch einmal auf die exzellente Dokumentation verwiesen werden. Dort wird noch einmal semantisch zwischen "windows" und "views" unterschieden, die Funktionsweise ist allerdings die Gleiche, beide können dabei noch ineinander verschachtelt werden.

```
CleanedLoopDetectorEvent.std:groupby(sensorid).win:time_length_batch(10 sec, 5 )
```

Hier wird eine Kombination von Zeit und Längenbasiertem Fenster verwendet. Das Statement löst dabei aus, wenn entweder 5 Ereignisse ankommen oder 10 Sekunden vergangen sind.

```
CleanedLoopDetectorEvent.win:time(20 sec).ext:sort(1, velocity desc)
```

Obiges Beispiel führt ein Sliding Window auf die letzten 20 Sekunden, wobei das CleanedLoopDetectorEvent mit der höchsten Geschwindigkeit aus betrachteten 20 Sekunden weitergereicht wird.

Alles in allem kann man behaupten, dass die Möglichkeiten zur Fensternutzung in Esper wenig zu wünschen übrig lassen, hier ist Esper auch erstaunlich simpel zu nutzen, was es hier zum klaren

Sieger macht.

5.3.6 Ereignisaggregation

Das Sammeln von Ereignissen über Fenster ist ein starker Sprachumfang einer ereignisverarbeitenden Sprache. Die einfache Aggregation von Ereignissen ist allerdings wie man mittlerweile aus der Ereignshierachie und -wolke gesehen hat nur der erste Schritt. Von Interesse innerhalb von zu erkennenden Ereignismustern sind normalerweise nicht die eigentlichen Ereignisse, sondern die beinhaltenden Daten wie beispielsweise ein Durchschnittswert eines Integer-Wertes über alle beinhaltenden Ereignisse einer Aggregation, oder der Maximalwert aus einer Menge von Ereignissen. Zwar sind solche Berechnungen auch über zu schreibende Methoden einer Wirtssprache realisierbar, es ist jedoch sowohl angenehmer, als auch dem eigentlichen Vorgang der Ereignisverarbeitung näher, wenn diese Funktionen bereits im Sprachumfang der Ereignissprache vorhanden sind. Die Sprache sollte deswegen:

- Die **klassischen Aggregatsfunktionen** zur Berechnung von Maximum, Minimum, Durchschnittswert, Summe und Anzahl unterstützen.[15]
- Erlauben das Ergebnis einer Aggregatsfunktion in einer **Variable** zu speichern, das auch On-The-Fly, soll heißen die Engine soll auch wissen, welcher Datentyp die Aggregation hat.
- Unter Umständen noch **mehr Funktionen** aus der Statistik unterstützen, wie den Median, Standardabweichung oder Top-N Agggregationen.

Ereignisaggregation in Drools:

In der Sprachmenge von Drools sind die fünf klassischen Aggregatsfunktionen Summe, Anzahl, arithmetisches Mittel, Maximum und Minimum für Mengen von Ereignissen standardmäßig enthalten und können sowohl über einfache Mengen durchgeführt werden, als auch über Sliding Windows.

```
$p : Number($average : doubleValue) from
accumulate(CleanedLoopDetectorEvent(sensorid == $i, $velocity : velocity) over
window:time(2m)
from entry-point "CleanedLoopDetectorEventStream", average($velocity))
```

Weitere Funktionen sind nicht enthalten, können vom Benutzer aber durch Implementierung des Interfaces

```
org.drools.base.acumulators.AccumulateFunction
```

sowie dem Anpassen der Konfigurationsdatei eingefügt werden.

Ereignisaggregation in Esper:

Esper kann sowohl die klassischen Aggregatsfunktionen verwenden, als auch noch weitere wie Funktionen zur Berechnung von Standardabweichung, Median und mehr. Die Verwendung der Aggregatsfunktionen ist dabei ähnlich wie in SQL, hier Statement 2.

```
select sensorid, avg(velocity) ,StaticMethods.evalAverage(avg(velocity))
from CleanedLoopDetectorEvent.std:groupby(sensorid).win:time_batch(5 min)
group by sensorid
```

Benutzerdefinierte Funktionen zur Ereignisaggregierung können hier ebenfalls einfach und ohne Registration aus Statements aufgerufen werden.

5.4 Erweiterter Sprachumfang und Engine Eigenschaften

5.4.1 Absenz von Ereignissen

Neben dem Sammeln von existierenden Ereignissen muss eine Ereignisverarbeitende Sprache auch Möglichkeiten unterstützen zu prüfen ob keine Ereignisse eines ausdrücklichen Typus vorhanden sind. Dies entspricht in der grundlegenden Ausführung dabei dem booleschen Operator NOT. Dies geht allerdings im Normalfall nicht weit genug, die Engine muss dabei auch während Datenaggregationen über Fenstern auf Nichtvorhandensein von Daten prüfen, was sie wiederum zwingt auf das Nichteintreffen eines Ereignisses zu warten, weil in Strömen natürlich Daten jederzeit fortdauernd eintreffen könnten und unter Umständen eine gerade stattfindene Aggregierung abgebrochen werden muss bei Unwahr werdendem NOT-Muster. Der Härtetest ist hierbei noch, ob die Engine in Bedingungen referenzierte nicht stattgefundene Ereignisse, zumindest wenn es logisch ist in Regelkonstrukten verwenden kann und unter Umständen sogar noch weiter verarbeiten kann. Vorstellbar wäre dabei eine Prüfung auf ein Ereignis, welches seinen Zeitstempel in einem bestimmten Zeitraum nach Nichtvorhandensein eines anderen Ereignis besitzt. Dies ist ein guter Test um zu prüfen ob es auch möglich ist in die Vergangenheit zu schauen.

- Unterstützung des **NOT-Operators.**
- Bei **Fortlaufenden Aggregierungen** soll die **Umgebung abbrechen**, wenn sich der Operator NOT bewahrheitet.Das Kriterium liegt hierbei darauf, dass die Prüfung dabei kontiniuierlich stattfinden soll und nicht nur einmalig bei Aggregationsbeginn.

Absenz von Ereignissen in Drools:

Drools stützt, wie bereits zu erwarten war den **NOT-Operator** zu Prüfung auf Nichtvorhandensein eines Faktums.

```
not (  model.LoopDetectorEvent (sensorid == $i,this after [0,500ms] $p) from
entry-point "LoopDetectorEventStream")
```

In Kombination mit einem temporalen Operator lassen sich so Beziehungen von absenten Ereignissen mit anderen Ereignissen beschreiben. Zu bemerken ist, dass die Engine natürlich gezwungen ist auf das Nichteintreffen eines Ereignisses zu warten. Das widerspricht natürlich dem üblichen Ablauf eines regelbasierten Systems welches keine Verzögerung bei Teilwahrheit eines Wenn-Ausdrucks kennt, sondern lediglich die Status Wahr oder Unwahr für eine Bedingung. Somit ist mit Drools Fusion ein wichtiger Punkt, namentlich die zeitlich verzögerte Feuerung von Regeln erfolgreich implementiert worden. Zum Beweis kurz ein Test:

Die Regel 1 aus dem Anwendungsbeispiel wird so verändert, dass sie nicht nur eine halbe Sekunde auf Nichteintreffen warten soll, sondern 5 Sekunden.

```
when $p : model.LoopDetectorEvent (velocity > 0 && < 280, $i : sensorid) from
entry-point "LoopDetectorEventStream"
not (  model.LoopDetectorEvent (sensorid == $i,this after [0,5s] $p) from entry-
point "LoopDetectorEventStream")
```

```
<terminated> Main (3) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (29.07.2010 18:35:46)
LoopDetectorEvent ist angekommen Die Uhrzeit ist: Thu Jul 29 18:35:48 CEST 2010
LoopDetectorEvent ist angekommen Die Uhrzeit ist: Thu Jul 29 18:35:49 CEST 2010
LoopDetectorEvent ist angekommen Die Uhrzeit ist: Thu Jul 29 18:35:50 CEST 2010
LoopDetectorEvent ist angekommen Die Uhrzeit ist: Thu Jul 29 18:35:51 CEST 2010
LoopDetectorEvent ist angekommen Die Uhrzeit ist: Thu Jul 29 18:35:52 CEST 2010
LoopDetectorEvent ist angekommen Die Uhrzeit ist: Thu Jul 29 18:35:52 CEST 2010
Ein CleanedLoopDetectorEvent schneller als 0 und ohne doppelte Messung
LoopDetectorEvent ist angekommen Die Uhrzeit ist: Thu Jul 29 18:35:53 CEST 2010
```

Abbildung 22: Das Warten auf das Nichteintreffen eines Ereignisses wird korrekt durchgeführt

Tatsächlich wartet die Inferenzmaschine 5 Sekunden auf das Nichteintreffen eines LoopDetectorEvents, bis die Regel gefeuert wird, wie im Ausschnitt bewiesen.

Wird der NOT-Operator in Wenn-Bedingungen mit dem impliziten UND zusammen mit einer Ereignisakkumulation verbunden lassen sich sich so **Abbruchbedingungen für Akkumulationen** fest legen, womit der Entwickler Optionen zur Steuerung von Akkumulationen erhält.

Was ausdrücklich nicht möglich ist, ist eine Referenz auf ein Nicht-Vorhandenes Ereignis zu haben. Der Drools Parser konnte folgendes Satzkonstrukt nicht verwenden:

```
not($t : SensorBehaviorEvent() from entry-point "SensorBehaviorEventStream")
not (SensorBehaviorEvent (this after [0,500ms] $t) from entry-point
"SensorBehaviorEventStream")
```

```
//Absenz von Ereignissen:
BuildError: Unable to create restriction '[VariableRestriction: after[0,500ms] $t ]' for field 't
not (SensorBehaviorEvent (this after [0,500ms] $t)
```

Abbildung 23: Fehler bei Referenzierung von Nicht-Vorhandenen Ereignissen

Da die Drools Engine mit dem NOT-Operator lediglich eine Evaluation durchführt, die entweder true oder false zurückgibt, scheint die Engine keine Referenz auf $t zu erstellen. Die Tatsache, dass das SensorBehaviorEvent $t und damit eine Referenz ja eigentlich garnicht existiert bei Wahrheit des Ausdrucks sei hier einmal ausgeklammert, wäre aber natürlich ein weiteres Hindernis.

Absenz von Ereignissen in Esper:

Die Kontrolle ob ein Ereignis nicht vorhanden ist wird innerhalb des `pattern` Operators mit dem `not` Operator geführt.

Statement 1 zeigt die Verwendung des **NOT-Operators**.

```
from pattern [ every timer:interval(1 sec) -> (a = LoopDetectorEvent(velocity >
0)  ->  not b = LoopDetectorEvent(sensorid=a.sensorid))
where  timer : within(1 seconds )]
```

Das Kriterium der **Abbruchbedingungen für Akkumulationen** lässt sich ebenfalls mit der Esper Mustererkennung lösen, im folgenden Statement 4 kann noch einmal ein Rundumschlag von Allem betrachtet werden, was die Esper Sprache hergibt:

```
Select avg(cl.velocity)
from pattern [ every cl = CleanedLoopDetectorEvent and not
CleanedLoopDetectorEvent(velocity > 200)].win:time_batch(20 sec)
```

5.4.2 Konsumierung von Ereignissen und Garbage Collecting

Die Anzahl an Ereignissen im Speicher wächst bei fortlaufendem Programmablauf unaufhörlich. Es muss also eine Idee her, wie mit dem Ausschuss an Ereignissen umgegangen werden soll, der garnicht mehr gewollt ist oder zu alt ist. In diesem Abschnitt soll darauf eingegangen werden, wie die Engine das Loswerden von ungewollten oder nicht mehr benötigten Ereignisse handhabt. Einerseits sollen sich Ereignisse automatisch, wenn sie nicht mehr von Interesse sind, aus dem Speicher verabschieden, andererseits sollen Sprachkonstrukte her, die angeben welche Ereignisse nicht konsumiert werden sollen.

- Ereignisse sollen **automatisch "konsumiert"** werden, sobald diese nicht mehr in Regeln/Statements gebraucht werden.
- Der Benutzer soll eine Kontrolle darüber haben, welche Ereignisse ausdrücklich **noch nicht konsumiert** werden sollen und weiterhin im Speicher verweilen und auch noch für zukünftige Mustererkennungen benutzbar sein sollen. Dies soll sowohl endlos geschehen können, als auch für ein einzustellende Anzahl an Wiederholungen von Mustern.

Konsumierung von Ereignissen und Garbage Collecting in Drools:

Drools kann Ereignissen mit der

```
@expires( 5s )
```

Meta-Information innerhalb der Ereignisdeklarierung eine definitiv feste Zeit zuweisen wie lange diese Ereignisse im Working-Memory verbleiben sollen.

Im folgenden soll kurz der Beweis erbracht werden, dass Drools Ereignisse die nicht mehr benötigt werden auch tatsächlich aus dem Speicher entfernt. Der Ablauf der Testumgebung bleibt derselbe, lediglich Regel 1 wird verwendet.

Die `@expires( ... )` Information des LoopDetectorEvents wird auf 200 Sekunden gesetzt:

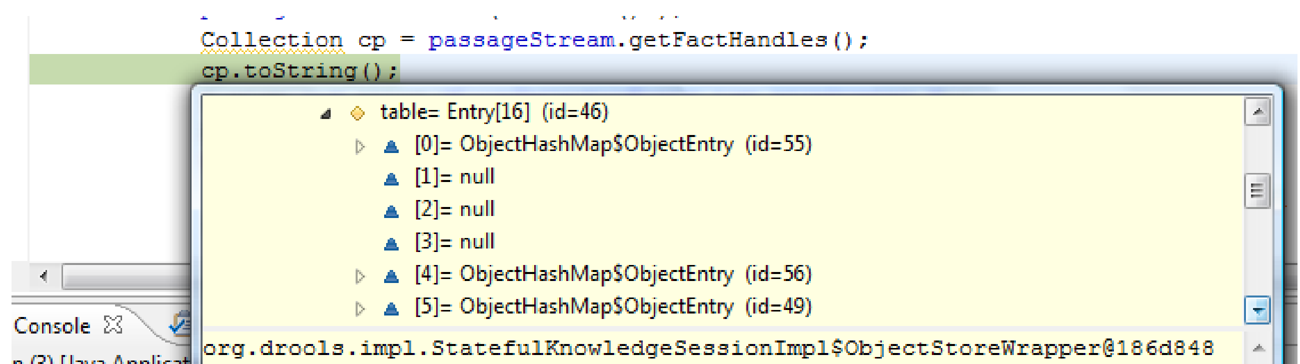

Abbildung 24: Die Hashmap ist durch den expires-Parameter weiterhin gefüllt

Der passageStream bezeichnete Entry-Point behält seine Ereignisse weiterhin im Speicher, obwohl diese bereits von Regel 1 verarbeitet wurden.

Nun soll die `@expires( ... )` Information entfernt werden, womit der Drools Engine die Arbeit

überlassen ist, unnötige Ereignisse zu entfernen:

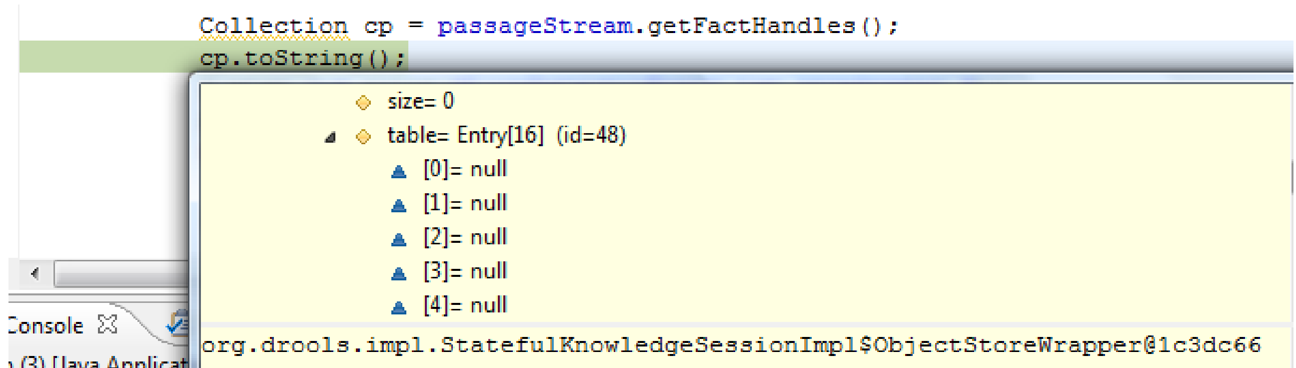

Abbildung 25: Der Beweis, das automatisches Garbage-Collecting durchgeführt wird

Tatsächlich ist die HashMap leer, da die Regel direkt nach Einfügen des LoopDetectorEvents ausgeführt wurde, womit kurz bewiesen wäre, dass Drools Fusion automatische Speicherverwaltung durchführt, wenn keine Regel mehr auf ein bereits verarbeitetes Ereignis hinweist.

Die `@expires( ... )` Information führt übrigens nicht, wie man villeicht nach obigem Beispiel annehmen könnte dazu, dass Muster für das selbe Ereignisse erneut feuern, sondern lediglich dazu, dass das Ereignis im Speicher verbleibt. Es dient also lediglich für den Fall, dass niemals ein Muster eintritt in welchem das Ereignis konsumiert wird und es somit für immer im Speicher ist, weil die Engine annimmt, dass es noch von Interesse ist. Jedes Ereignis wird wenn es für alle referenzierenden Muster genutzt wurde als verbraucht betrachtet und nicht mehr benutzt.

Es gibt für Drools Fusion leider keine äquivalentes Konstrukt zum `every`-Operator der ein Muster immer wieder erneut für ein Ereignis feuern lässt. Die Nichtexistenz dieses Operators hat sich in der Evaluation als eklatantes Hindernis erwiesen, was Drools hier vergleichsweise schlecht aussehen lässt. Ein `repeat`-Operator wird ebenfalls schmerzlich vermisst, auch wenn ein Sprachkonstrukt wie in Regel 4 verwendet die selbe Funktion umsetzt, wenn auch komplizierter.

Will man ein Ereignis einfach manuell entfernen, so kann die Methode:

```
retract($p);
```

aus den Regeln direkt heraus benutzt werden.

Konsumierung von Ereignissen und Garbage Collecting in Esper:

Esper berechnet automatisch anhand der in die Engine gelegten Statements, wie lange Ereignisse im Speicher verbleiben sollen. Dabei schaut Esper auf die benutzten Fenster und verwirft Ereignisse, sobald die definierten Fenstergrößen für die Mustererkennung überschritten worden sind.

```
select avg(cl.velocity)
from pattern [ every cl = CleanedLoopDetectorEvent and not
CleanedLoopDetectorEvent(velocity > 200)].win:time_batch(20 sec)
```

Die zeitliche Fenstergröße für das Statement 4 beträgt 20 Sekunden. Jegliche CleanedLoopDetectorEvents die zwar das Pattern betreten, es jedoch nicht auslösen, weil die zweite Bedingung sich nicht bewahrheitet, werden nicht an den implementierten Listener durchgereicht, sondern verworfen. Das geschieht zumindest immer dann,sobald diese auch in keinem weiteren Eingangsstrom eines anderen Statements mehr in Betracht zur Weiterreichung an Listener gezogen werden können.

Esper scheint keine "harten" Konfigurationsmöglichkeiten zu besitzen, wie lange Ereignisse im Speicher behalten werden sollen, im FAQ auf der Esper Seite lässt sich lediglich folgende Aussage finden:

"The policy for holding events in memory depends on your statement, especially the data window used (sliding, tumbling and time or length etc), and whether it uses patterns, output rate limiting or is a join.

For example, if your statement employs no data window, Esper keeps no events in memory.

(...)

For views that derive values from an event stream, no events are kept in memory. For aggregations, only the aggregation values are kept in memory. For patterns, the events that participate in the pattern are kept in memory only if tagged"

Zum Testen ob dies auch der Wahrheit entspricht, wird das definierte Statement 6 verwendet:

```
select *
from pattern [every a = LoopDetectorEvent -> b = ProblemEvent ]
```

Das Statement behält alle LoopDetectorEvents, bis ein einziges ProblemEvent darauf folgt. Dummerweise ist die Anwendung in diesem Fall so geschrieben, dass nie ein ProblemEvent kommen wird, was die Esper Laufzeitumgebung dazu zwingen wird, alle LoopDetectorEvents im Speicher zu behalten, da auch keine Fenster definiert sind, die dem Treiben ein Ende setzen können, zumindest so die Theorie nach dem was oben steht.

Der EventFeeder ist so geschrieben, dass eine Million LoopDetectorEvents eingeführt werden.

Nach etwa 540000 Ereignissen geht die Anwendung merklich in die Knie

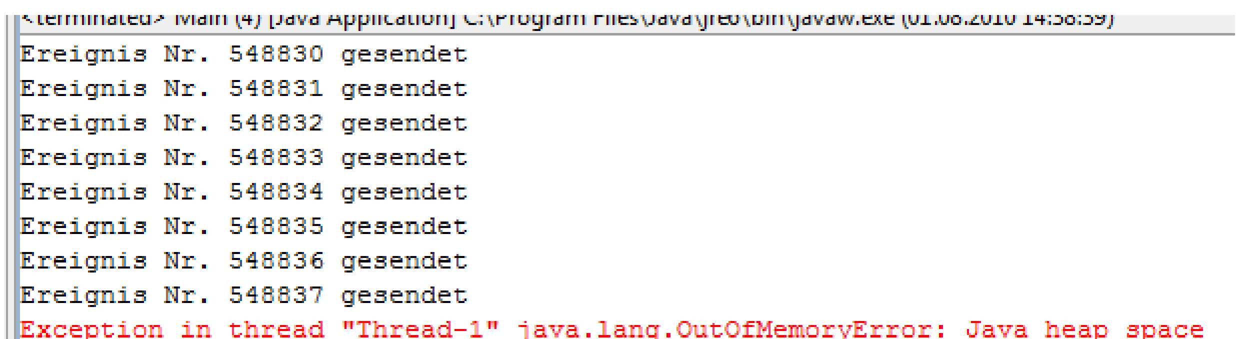

Abbildung 26: Exception bei Überlast unter Benutzung des every-Operators

Nach noch etwas mehr Ereignissen terminiert sich die Anwendung schließlich per Exception. Somit ist die Aussage bewiesen, dass Ereignisse solange im Speicher bleiben, wie Esper sie in den Statements betrachten muss, alle verarbeiteten Ereignisse werden entfernt. Zugegebermaßen ist ein solches Statement nicht sehr clever, dennoch zeigt dies, dass es villeicht von Vorteil ist, konfigurieren zu können, dass alte Ereignisse verworfen werden, selbst wenn sie noch von Interesse sind, sobald Speichermangel vorherrscht, eine komplette Terminierung der Anwendung ist wohl schlimmer als einige Ereignisse zu verlieren.

5.4.3 Content-Enrichment

Die oben genannte Ereignisdefinition beschreibt ein Ereignis als die Beschreibung einer Zustandsveränderung in einem System. Das impliziert im weitesten Sinne, dass Ereignisse, wenn sie

denn geschehen sind eigentlich nicht geändert werden sollten, da jede mögliche Veränderung eines Ereignisses die Abbildung der Ereignisse auf den "eigentlichen" Zustand des Systems inkonsistent werden lässt. Natürlich lässt sich der Entwickler von solch strikten Definitionen nicht klein kriegen und in der Praxis ist es unter Umständen notwendig Daten auch nach Eintreffen eines Ereignisses im System noch zu verändern, beispielsweise um erst später bekannte Fehler zu korrigieren. Es ist auch schlicht vorstellbar, dass bestimmte Daten erst nach Ereignisauftritt berechnet werden können oder das für weiterführende Mustererkennung Daten die sich nicht im Verarbeitungsspeicher der Laufzeitumgebung befinden noch mit Daten aus anderen Systemen "gefüttert" werden. Die "Content-Enrichment" genannte Funktion kann dabei zwar auch von der Wirtssprache übernommen werden, Funktionen zur Werteveränderung in der Ereignissprache sind allerdings ebenfalls eine nette Sache, wenn denn unterstützt.([1] Seite 120-121)

- **Modifizieren und Füllen von Daten** eines Ereignisses in der Ereignissprache sollte unterstützt werden.

Content-Enrichment in Drools

Bisher wurde weitestgehend die "linke" Seite der Drools Regeln, also die Wenn-Bedingung betrachtet. Nun soll kurz die andere Seite, die Dann-Aktion angeschaut werden. Diese eignet sich nämlich ideal dafür, Modifikationen an Ereignissen durch zu führen. Drools unterstützt dies mit dem modify Befehl:

```
then
modify( $p ){
        setVelocity( $p.getVelocity + 15 )
}
end
```

Die Sinnigkeit die Geschwindigkeit eines Ereignisses einfach um 15 zu erhöhen sei kurz beiseite geschoben, wichtig ist jedoch, dass Methoden sowohl von Ereignissen als auch von Fakten im Working-Memory und statische Methoden aus importierten Klassen aufgerufen werden können, womit Änderungen an Ereignissen vorgenommen werden können.

Die Engine muss jedoch noch mit

```
update($p);
```

aufgefordert werden, noch in dieser Inferenzrunde die Daten neu zu bewerten, sonst werden Änderungen erst bei der nächsten Inferenz bedacht.

Content-Enrichment in Esper

Traditionell wird Content-Enrichment in Esper in den an den Statements registrierten Listenern durchgeführt. Der Ablauf ist somit weitestgehend äquivalent zum Dann-Teil der Drools Fusion Regeln:

```
LoopDetectorEvent l = (LoopDetectorEvent) newData[0].get("a");
ArrayList<LoopDetectorEvent> orglist = new ArrayList<LoopDetectorEvent>();
orglist.add(l);
CleanedLoopDetectorEvent clean = new
CleanedLoopDetectorEvent(idgetter.getnextID(),l.getSensorid(),l.getLocation(),org
list,l.getVelocity(),new Date(),120);
runtime.sendEvent(clean);
```

Somit wird dieser Part von der Wirtssprache Java in dem registrierten Listenern übernommen.

5.4.4 In-Line Datenmodifikation

Ein weiterer Punkt um einen Parser zu testen ist die Frage, ob er es schafft in der Regelsprache geschriebene Datenwertverschiebungen direkt verarbeiten zu können. Dies kann sowohl direkt in Datenwertzuweisungen geschehen, als auch in Vergleichen.

- **Arithmetische Datenwertverschiebungen** sollten auch vom Compiler unterstützt werden. Man stelle sich vor, der Nutzer möchte Integer-Daten von zwei Ereignissen vergleichen:

  Event1.countedAverage > Event2.countedaverage + 15

  (Man achte auf die + 15.) Unterstützt der Compiler auch solche sprachlichen Konstrukte, oder ist man gezwungen auf Methoden der Wirtssprache zurück zu greifen, ist dabei die Frage.

In-Line Datenmodifikation in Drools:

Der Drools-Parser besitzt kurioserweise nur die Möglichkeit arithmetische Wertverschiebungen in Bedingungen zu nutzen, wenn der eval-Operator verwendet wird. Dieser wertet einen Ausdruck aus kehrt mit true zurück bei Wahrheit. Der Ausdruck:

```
$p : model.LoopDetectorEvent (eval(velocity > 0)  && eval( velocity < 280 + 25),
$i : sensorid)
```

ist kompilierbar und zumindest nominal äquivalent zu:

```
$p : model.LoopDetectorEvent (velocity > 0  && velocity < (280 + 25), $i :
sensorid)
```

welcher wiederum nicht kompilierbar ist, trotz der Klammerung.

In-Line Datenmodifikation in Esper:

Im Gegensatz zu Drools ist der Parser von Esper etwas standfester was Inline Datenwertverschiebungen betrifft.

Der Befehl:

```
a = LoopDetectorEvent(velocity >( 0 + 40))
```

konnte ohne Probleme vom Parser verarbeitet werden, womit Esper hier mehr Reife beweist.

5.4.5 Performance

In diesem Unterkapitel wird ein kurzer Überblick über die Performance der Laufzeitumgebungen gegeben. Dabei soll eine große Menge an Ereignissen in sehr kurzer Zeit in den Speicher eingefügt werden. Dabei soll die Abarbeitungsgeschwindigkeit betrachtet werden. Absichtlich wird alles simpel gehalten, um jegliches Drumherum auszuschalten, sodass nur die Engine selbst Arbeit verrichtet und sprachliche Unterschiede minimal sind. Die Testumgebung ist bei beiden die Gleiche und auch keine Prozesse werden zwischenzeitig gestartet. Dieser Abschnitt soll keineswegs einen wirklich

umfangreichen Performance-Test ersetzen, sondern nur eine Idee geben, welche Engine schneller ist. Natürlich ist es möglich das sich bei anderen Daten und Konfigurationen andere Ergebnisse herausstellen

Die Testanwendung ist wie folgt aufgebaut:

- Lediglich zwei Ereignisse werden in der Sprache deklariert, LoopDetectorEvent und ProblemEvent.
- Nur ein einziges Statement bzw. eine einzige Regel wird geschrieben. Für Drools ist dies:

```
rule "waitingforProblem"
when $a : model.LoopDetectorEvent () from entry-point "LoopDetectorEventStream"
model.ProblemEvent (this after  $a) from entry-point "ProblemEventStream"
then
System.out.print(ProblemEvent gefunden );
end
```

Für Esper lautet das Statement:

```
select *
from pattern [ a = LoopDetectorEvent -> ProblemEvent ]
```

Beide Pattern zwingen die Engine auf ein ProblemEvent zu warten.Nach Einfügen von einer Anzahl von LoopDetectorEvents wird die Engine erlöst und eine Ausgabe über die Zeit, wie lange die Verarbeitung gedauert hat wird angezeigt. Das Modell und auch der Code ist für beide Seiten so identisch wie möglich gehalten. Die Funktion der Regel/des Statements ist, wie oben ersichtlich ebenfalls gleich. Das Ziel des Tests ist es, zu erkennen wie stark die Last beim Einfügen von vielen Ereignissen zur selben Zeit ist.

Performance in Drools:

Folgende Ergebnisse haben sich herausgestellt:

Anzahl an LoopDetectorEvents	Vergangene Zeit in Millisekunden bis zur Terminierung
1000	123
10000	522
100000	3875
1000000	35871

Performance in Esper:

Anzahl an LoopDetectorEvents	Vergangene Zeit in Millisekunden bis zur Terminierung
1000	81
10000	215
100000	1607
1000000	14683

5.5 Kriterien zur Arbeit mit den Plattformen

5.5.1 Installation und Integration in Entwicklungsumgebungen

Dieser Abschnitt soll dazu dienen einen Überblick über die Installation und Bereitstellung der beiden betrachteten Engines zu geben. Die Entwicklungsumgebung beider Engines war während Erstellung dieser Arbeit dabei die Eclipse IDE und die Wirtssprache Java. Hier soll kurz dargestellt werden wie schnell und subjektiv gut die Laufzeitumgebung in der Entwicklungsumgebung Eclipse integriert worden ist und wie einfach die Benutzung der bereitgestellten Bibliotheken ist. Dabei soll auch kurz ein Überblick auf den Umfang der zu verwendenden Bibliotheken und damit das Gewicht der Plattformen geworfen werden.

Installation und Integration für Drools:

Drools besitzt eine eigene Eclipse Perspektive, was ein angenehmer Vorteil zur Arbeit mit dieser Plattform ist.

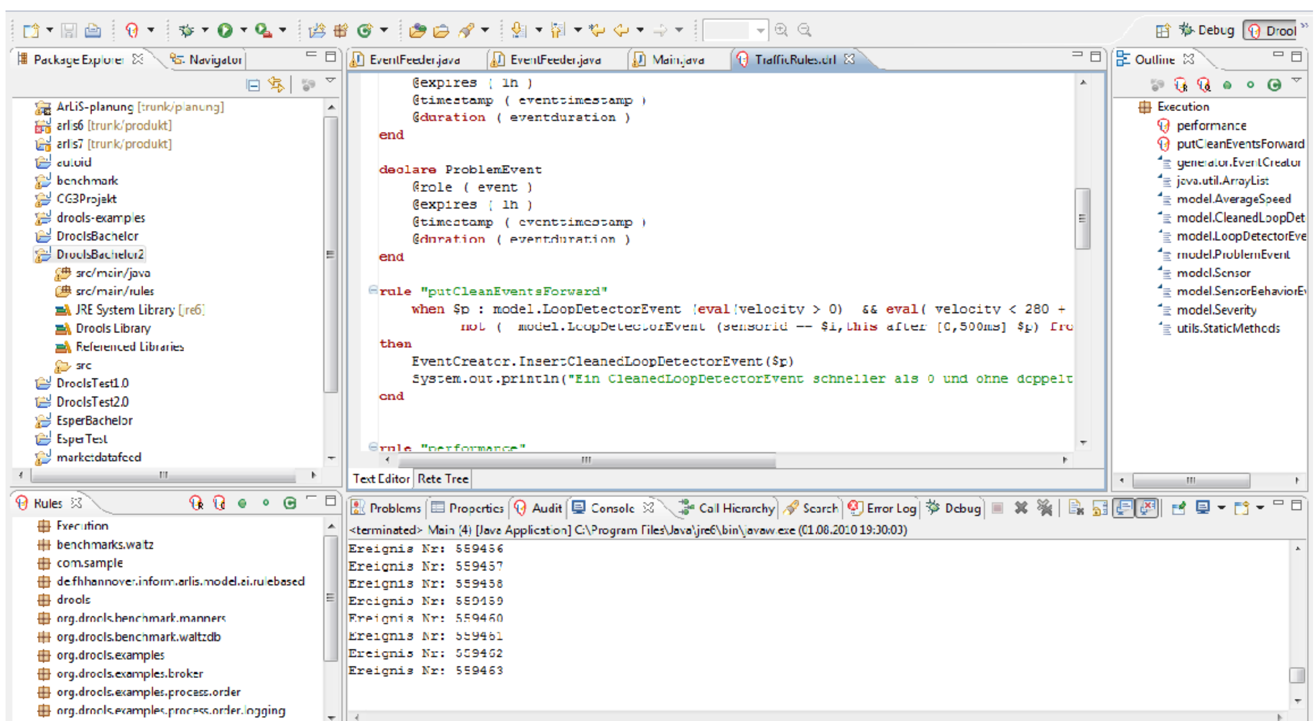

Abbildung 27: Die Drools Perspektive

Zur Installation der Perspektive muss lediglich das Eclipse Plugin von der Drools Webseite runtergeladen werden, oder der Update Mechanismus von Eclipse verwendet werden.

Die Drools Perspektive erlaubt eine Ansicht auf die definierten Regeln und ihre Packages und besitzt Dialoge zu Erstellung von neuen Drools Projekten, eine eigene Syntaxkorrektur für Regeldateien sowie einige weitere Komfortfunktionen.

Danach kann in den Eclipse Preferences eine neue Drools Runtime erstellt werden. Die Runtime wird in Zukunft dafür sorgen, dass die zur Ausführung benötigten Bibliotheken in ein neues Projekt automatisch eingebunden werden.

Eine genaue und einfache Beschreibung der Installation ist auf der Drools Webseite zu finden.

Die für Drools benötigten Bibliotheken:

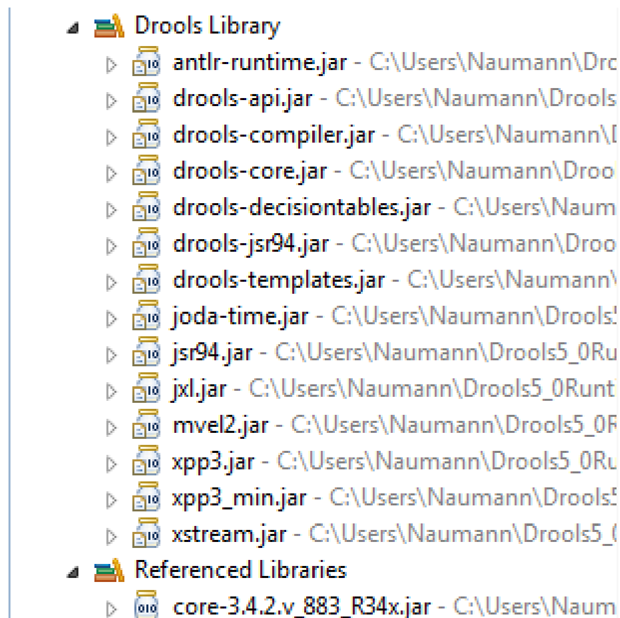

Abbildung 28: Benötigte Libraries für Drools

Die core-3.4.2.v_883_R34x.jar ist die einzige Bibliothek, die nicht in der Drools Runtime vorhanden ist und noch zur Ausführung referenziert werden muss. Insgesamt wiegen alle benötigten Bibliotheken 9,45 MB. Der Sprachinhalt von Fusion ist enthalten.

Installation und Integration für Esper:

Im Gegensatz zu Drools besitzt Esper keine eigene Eclipse-Perspektive, ist jedoch dafür sehr viel leichtgewichtiger.

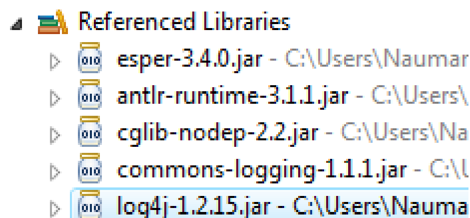

Abbildung 29: Benötigte Libraries für Esper

In diese referenzierten Bibliotheken befindet sich alles, was in dieser Evaluation benutzt und benötigt wurde. Insgesamt wiegen alle Bibliotheken 3,6 MB. Esper liefert im 3.4 Paket noch weitere .jars mit, zur Integration in Frameworks sowie weiteres.

5.5.2 Support und Dokumentation der Plattform

Um mit irgendeiner Sprache arbeiten zu können, ist es natürlich zuerst notwendig zu verstehen wie eine Sprache überhaupt funktioniert. Dafür ist eine gute Dokumentation und Beispiele, die zeigen wie Sprachkonstrukte funktionieren, notwendig. Dieser kurze Abschnitt soll eine subjektive Meinung über dieses Thema darlegen.

Support und Dokumentation für Drools:

Die Drools 5.0 Expert Dokumentation ist ziemlich umfangreich, dabei dennoch übersichtlich und zeigt gut, wie Funktionen aus dem normalen Sprachumfang von Drools verwendet werden. Das Problem ist jedoch, das die eigentliche Drools Fusion Dokumentation kurz gehalten ist und nur dessen Funktionen zeigt, ohne dabei näher auf Implementierungsdetails einzugehen. Im Zuge dieser Arbeit war deswegen eine Menge an Ausprobieren notwendig, um zum gewünschten Ergebnis zu gelangen. Für den normalen Expert Sprachgebrauch sind eine Plethora an Beispielen sowohl direkt vom Entwickler, als auch im Netz zu finden, für Fusion sind diese rarer und der Entwickler sorgt nur für ein einziges, wenn auch umfangreiches, Beispiel in Form einer Börsenanwendung.

Dokumentation und Beispiele sind zu finden unter:

http://www.jboss.org/drools/downloads.html

Support und Dokumentation für Esper:

Die Esper Dokumentation ist umfangreich und lässt wenig zu wünschen übrig, auch wenn sie an einigen Stellen unübersichtlich wirkt. Genaues Studium der Dokumentation ist aufgrund des hohen und zum Teil abstrakten Sprachumfangs dringend notwendig. Esper liefert direkt zusammen mit der Anwendung selbstlaufende Beispiele, die viele Aspekte der Sprache abdecken.

Dokumentation ist zu finden unter:

http://esper.codehaus.org/esper/documentation/documentation.html

Beispiele sind im normalen Esper Paket enthalten unter

http://esper.codehaus.org/esper/download/download.html

5.6 Zusammenfassende Gegenüberstellung

Dieses Kapitel soll einen Überblick über alle ausgearbeiteten Kriterien geben und wie sich die beiden Engines in den Kriterien geschlagen haben. Dazu soll eine gegenüberstellende Tabelle vorgestellt werden und Entscheidungen begründet werden.

Kriterium	**Benotung von Drools**	**Benotung von Esper**	**Grund für Entscheidung**
Sprachparadigma	O	O	Die Entscheidung für ein Sprachparadigma obliegt dem spezifischen Nutzer, jeder selbst muss entscheiden, ob CQL-Syntax oder deklarative Regeln vorgezogen werden.
Definition und Relation von Ereignissen	O(+)	O(-)	Beide Sprachen besitzen keine nach dem Kriterium festgelegten tieferen Definitionsmöglichkeiten für Ereignisse. Beiden Sprachen liegt zuallererst die Verarbeitung von Ereignissen am Herz, nicht die Definition. Aufgrund der externen Definition von Ereignissen in Drools bleibt für die Zukunft zu hoffen, dass hier bald mehr möglich ist.
Ereignisherkunft	-	+	Die Möglichkeit zur Festlegung von Entry-Points für Ereignisse in Drools ist ein guter Schritt für die komplexe Ereignisverarbeitung, zur Zeit ist es jedoch nicht möglich, Ereignisse in Entry-Points aus den Regeln selbst einzufügen, weswegen dieses Kriterium, wenn auch knapp, an Esper geht.
Zeitkonzept und Kontrolle über Zeit	O(+)	O(-)	In Drools besitzt der Nutzer etwas mehr Kontrolle über das Zeitkonzept als in Esper, aufgrund der Möglichkeit Zeitstempel direkt zu deklarieren, was jedoch sowohl Vor- als auch Nachteil sein kann. Esper gibt kümmert sich dagegen explizit selbst um Reihenfolgen im Speicher. Beide Sprachen erlauben die direkte Kontrolle über den Fluß der Zeit. Beide Sprachen sind in diesem Bereich ausgefeilt genug für CEP-Anwendungen.
Verknüpfungsoperatoren	+	-	Verknüpfungen von Strömen lassen sich in Drools dank implizitem UND sowie dem allgemeinen Sprachparadigma etwas simpler und übersichtlicher gestalten, alles in allem nehmen sich aber beide Sprachen, erwartungsgemäß nicht viel in diesem Kriterium.
Temporale Operatoren/Sequentielle Operatoren	O(-)	O(+)	Temporale Operatoren für Drools, sowie sequentielle für Esper sind im Grunde zwei Unterschiedliche Herangehensweisen an die gleiche Funktion. Mit beiden lassen sich die selben Problemstellungen lösen, wobei Esper jedoch für komplexere Aufgaben die Unterstützung von Fenstern und Post-Guard

			Operatoren inhärent benötigt. Dafür ist die Benutzung von Sequenzoperatoren in Esper für einige andere Problemstellungen etwas simpler. Insgesamt ein recht subjektives Kriterium, wobei die Benutzung von Esper gefühlt etwas einfacher war.
Kausalität und kausale Operatoren	-	-	Beide Sprachen besitzen kein Wissen über Kausalität von Ereignissen, in beiden muss dieses Kriterium deswegen negativ ausfallen. Aufgrund der näher zusammen liegenden Definition von Regelkonstrukten und Ereignissen in der selben Regeldatei, bleibt zu hoffen, dass sich für Drools hier in der Zukunft mehr tut.
Fenster	--	++	Die Implementation von Fenstern sowohl in Funktionsumfang als auch im Handling ist für Drools im Vergleich zu Esper mehr als dürftig. Esper kennt nicht nur viel mehr Fenster, sondern die Verwendung ist auch simpler und geht leichter von der Hand.
Absenz von Ereignissen	-	+	Drools und Esper besitzen sowohl die Möglichkeit auf absente Ereignisse zu warten, als auch Akkumulationen zur Laufzeit abbzubrechen, sobald Absenz durchbrochen ist. Aufgrund der in der Evaluation festgestellten Unmöglichkeit Nichtvorhandene Ereignisse zu referenzieren scheint Drools hier allerdings unausgewogener und verliert deswegen.
Ereignisaggregation	-	+	Esper kann hier mehr und es fällt auch etwas leichter eigene Aggregationen in den Sprachumfang einzufügen, Grundlegend Wichtiges beherrschen jedoch beide Sprachen.
Konsumierung von Ereignissen und Garbage Collecting	-	+	Beide Engines besitzen automatisches Garbage-Collecting für unwichtig gewordene Ereignisse, Drools besitzt jedoch aufgrund des fehlenden every- und repeat-Operators schlechtere Kontrolle zur Mustererkennung für nicht zu konsumierende Ereignisse, was Kontrolle vom Anwender wegnimmt. Drools kann eine definitiv feste Zeit zum Garbage-Collecting einstellen,auch wenn das Ereignis noch von Interesse ist, was ein kleiner Vorteil ist.
Content-Enrichment	O	O	Beide Sprachen besitzen Möglichkeiten zum Content-Enrichment direkt aus der Sprache heraus.
In-Line Datenmodifikation	-	+	Hier beweist Esper das es definitiv standfester ist.
Performance	O(-)	O(+)	Siehe Tabelle, wobei ein tieferer Performance-Test als der durchgeführte jedoch notwendig ist

			um ein endgültiges Urteil zu fällen.
Installation und Integration in Entwicklungsumgebunge n	+	-	Die Eclipse Perspektive ist gut integriert, dafür ist Drools im Gesamtpaket schwergewichtiger.
Support und Dokumentation der Plattform	-	+	Die Fusion Doku von Drools ist eher kurz und die Entwickler liefern nur ein Beispiel.

5.7 Nicht betrachtete Kriterien

Für eine endgültige Evaluation sind noch weitere Kriterien denkbar, die allerdings im Zuge dieser Bachelorarbeit nicht betrachtet wurden. Die Integration von einer Plattform mit Frameworks wie Spring, das Verhalten bei Multithreading, die Möglichkeit der automatischen Persistierung im Falle eines Systemabsturzes, sowie ein umfangreicher Performance-Test und weitere sind sicherlich noch denkbar. Da diese Arbeit auch das Ziel hat einen Evaluierungleitfaden darzustellen, ist es nur fair diese weiteren Kriterien, auch wenn sie nicht betrachtet wurden, noch zum Abschluss zu erwähnen.

6. Fazit und Schlussbemerkungen

Drools Fusion hat in dieser kurzen Evaluation bereits gezeigt, dass es grundsätzlich alle von einer Ereignisverarbeitende Sprache und Engine benötigten Anforderungen erfüllt. Die Verarbeitung von unterschiedlichen Strömen ist in den Rete-Algorithmus erfolgreich implementiert worden. Die automatische Ereignisverarbeitung und Konsumierung ist ebenfalls funktionabel. Im Vergleich zu Esper ist der Sprachumfang weitaus geringer und auch die Funktionen sind weniger ausgereift. Die Dokumentation ist noch nicht so tiefgreifend und Beispiele speziell für Fusion sind auch noch nicht so viele vorhanden wie für Esper. Drools hat dennoch bewiesen, dass Ereignisverarbeitung auf Grundlage eines Regelbasierten Systems generell eine gute Idee ist.

Für die Zukunft ist zu hoffen, dass Drools insbesondere seine zumindest potentielle Stärke im Bereich der Ereignisdefinition ausbauen kann, da Esper hier vom Sprachparadigma stärker verhindert ist, sodass Ereignisverarbeitung mit einem an der Realität nah angelehnten Modell und logischer, deklarativer Sprache möglich ist. Funktionen wie der every-Operator oder sequentielle Operatoren sind dabei ein Muss für die Drools Plattform der Zukunft. So wie es aussieht scheint die nächste Drools Version bereits in den Startlöchern und ist in der nicht supporteten Version 5.1 RC1 bereits ausprobierbar.

Für eine professionelle Anwendung sollte aber zumindest im Augenblick Esper der Vorzug gegeben werden, es hat sich Alles in Allem als die bessere Plattform in den meisten Kriterien dieser Evaluation erwiesen.

Abbildungsverzeichnis

Abbildung 1: Räsonierung über komplexes Ereignis......7
Abbildung 2: Modell......9
Abbildung 3: AbstractEvent......10
Abbildung 4: SensorEvent......10
Abbildung 5: LoopDetectorEvent......10
Abbildung 6: CleanedLoopDetectorEvent......11
Abbildung 7: TrafficDataEvent......11
Abbildung 8: SensorBehaviorEvent......12
Abbildung 9: ProblemEvent......12
Abbildung 10: TrafficJamEvent......12
Abbildung 11: Regelzusammenführung in Rete [6]......14
Abbildung 12: Darstellung von Faktenverarbeitung in Alpha und Beta Knoten[6]......15
Abbildung 13: Ereignisgenerator......16
Abbildung 14: EventWriter......16
Abbildung 15: EventReader......17
Abbildung 16: EventCreator......18
Abbildung 17: EventFeeder implementiert Runnable......19
Abbildung 18: Ein von Drools automatisch zugewiesener Zeitstempel......37
Abbildung 19: Von Allen definierte temporale Relationen sowie ihre Operatoren[14]......41
Abbildung 20: Beispiel eines zeitbasierten Sliding Windows, übernommen aus der Esper Dokumentation......45
Abbildung 21: Beispiel eines zeitbasierten Tumbling Windows, übernommen aus der Esper Dokumentation......46
Abbildung 22: Das Warten auf das Nichteintreffen eines Ereignisses wird korrekt durchgeführt......49
Abbildung 23: Fehler bei Referenzierung von Nicht-Vorhandenen Ereignissen......50
Abbildung 24: Die Hashmap ist durch den expires-Parameter weiter gefüllt......51
Abbildung 25: Der Beweis, das automatisches Garbage-Collecting durchgeführt wird......52
Abbildung 26: Exception bei Überlast unter Benutzung des every-Operators......53
Abbildung 27: Die Drools Perspektive......58
Abbildung 28: Benötigte Libraries für Drools......58
Abbildung 29: Benötigte Libraries für Esper......59

Literaturverzeichnis

1: Event-Driven Architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse; Ralf Bruns , Jürgen Dunkel; 2010; Springer Verlag

2: Event-driven architecture; Wikipedia; http://en.wikipedia.org/wiki/Event-driven_architecture; abgerufen 30.07.2010

3: Event-Driven Applications: Costs, Benefits and Design Approaches; Mani Chandy; California Institute of Technology; http://www.infospheres.caltech.edu/node/38; abgerufen 30.07.2010

4: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems; David Luckham; 2002; Addison-Wesley Pearson Education

5: MVEL; Wikipedia; http://en.wikipedia.org/wiki/MVEL; abgerufen 30.07.2010

6: Der Rete-Algorithmus; FH Bonn-Rhein-Sieg; http://www2.inf.fh-rhein-sieg.de/~pbecke2m/xps/regel2.pdf; abgerufen 30.07.2010

7: Comprehensive Guide to Evaluating Event Stream Processing Engines; Coral Inc.; http://complexevents.com/wp-content/uploads/2008/11/coral8guidetoevaluatingespengines.pdf; abgerufen 30.07.2010

8: Complex Event Processing: a technology evaluation check-list; Paul Vincent; TIBCO Software Inc; http://tibcoblogs.com/cep/2010/03/04/complex-event-processing-a-technology-evaluation-check-list/; abgerufen 30.07.2010

9: Event Stream Processing; Complex Event Processing and Rules Engines; Edson Tirelli; JBoss Rules; http://blog.athico.com/2007/05/event-stream-processing-complex-event.html; abgerufen 30.07.2010

10: Complex Event Processing; Michael Eckert , Francois Bry, Gesellschaft für Informatik; http://www.gi-ev.de/service/informatiklexikon/informatiklexikon-detailansicht/meldung/complex-event-processing-cep-221.html; abgerufen 30.07.2010

11: Technology Overview; Espertech; http://esper.codehaus.org/esper-3.5.0/doc/reference/en/html/technology_overview.html#technology_overview_intro_cep; abgerufen 30.07.2010

12: The Situation Manager Component of Amit - Active Middleware Technology; Asaf Adi , David Botzer , Opher Etzion; IBM Research, Haifa; http://www.research.ibm.com/haifa/projects/software/amit/papers/AMIT_Situation_Manager-NGITS.pdf; abgerufen 30.07.2010

13: Towards a streaming SQL Standard; Namit Jain , Shailendra Mishra , Anand Srinavasan , Johannes Gehrke , Uğur Çetintemel , Mitch Cherniack , Richard Tibbets , Stan Zdonik , Jennifer Widom , Hari Balakrishnan; http://www.cs.brown.edu/~ugur/streamsql.pdf; Oracle Corp. , Stanford University , Cornell University , StreamBase Inc.; abgerufen 30.07.2010

14: An Interval-Based Representation of Temporal Knowledge; James Allen; Departement of Computer Science , University of Rochester; http://dli.iiit.ac.in/ijcai/IJCAI-81-VOL%201/PDF/045.pdf; abgerufen 30.07.2010

15: Aggregation(OLAP); Wikipedia; http://de.wikipedia.org/wiki/Aggregation_(OLAP); abgerufen 30.07.2010