Odysseus

Thomas Vogelgesang

thomas.vogelgesang@informatik.uni-oldenburg.de

Abstract: Odysseus stellt ein generisches Framework zur Entwicklung von Datenstrommanagementsystemen dar und ist unter anderem wegen der Verwendung der OSGi Service Platform leicht an verschiedene Szenarien anpassbar. Im Rahmen der Projektgruppe StreamCars soll es beispielsweise zur Fusion datenstrombasierter Sensordaten für Fahrassistenzsysteme eingesetzt werden. In dieser Arbeit wir daher der Aufbau des Frameworks und die Funktionalität seiner Komponenten erläutert. Darüber hinaus werden die Grundlagen der OSGi Service Platform erläutert und die Erweiterbarkeit von Odysseus anhand eines Tutroials gezeigt, in dem ein neuer Datenstromoperator beispielhaft implementiert und integriert wird.

1 Einleitung

Ziel der Projektgruppe *StreamCars* ist es, die für intelligente Fahrassistenzsysteme erforderliche Sensordatenfusion durch ein allgemeines Datenstrommanagementsystem (DSMS) zu realisieren und die Funktionsfähigkeit des Konzepts durch eine prototypische Implementierung zu zeigen. Dabei soll für die Datenverarbeitung *Odysseus*, ein erweiterbares Framework zur Entwicklung von DSMS, zum Einsatz gebracht werden.

In der vorliegenden Arbeit wird daher zunächst auf den generellen Aufbau des Frameworks und die Funktionalität der einzelnen Komponenten sowie deren Zusammenhang eingegangen. Anschließend werden einige Grundlagen der OSGi Service Platform erläutert, welche in Odysseus zur Modularisierung verwendet wird. Danach folgt ein Tutorial, das die Erweiterung des Frameworks anhand der Implementierung eines Split-Operators beispielhaft darstellt. Den Abschluss dieser Arbeit bildet eine kurze Zusammenfassung.

2 Architektur von Odysseus

Bei Odysseus handelt es sich um ein Framework zur Entwicklung von Datenstrommanagementsystemen (DSMS), das das erforderliche Grundgerüst zur Verarbeitung von Datenströmen bereitstellt. Durch die komponentenbasierte Architektur ist es leicht an verschiedene Anwendungen anpassbar. Innerhalb der Komponenten werden *Fixpunkte* und *Variationspunkte* unterschieden. Fixpunkte sind interne Verwaltungsstrukturen, die die Ausführung einzelner Verarbeitungsschritte übernehmen und fest ins Framework integriert sind. Variationspunkte sind einzelne, veränderbare Verarbeitungsschritte [BGJ+09].

Abbildung 1 zeigt die Architektur von Odysseus, dessen Komponenten im Folgenden erläutert werden.

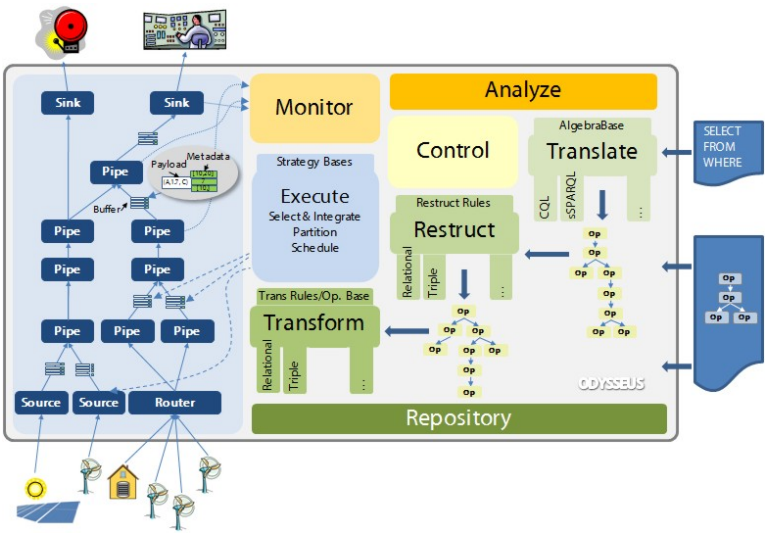

Abbildung 1: Architekturübersicht von Odysseus [BGJ+09]

Die Translate-Komponente übersetzt Anfragen in logische Anfragepläne. Die Anfragen können in unterschiedlichen Anfragesprachen formuliert sein und auf verschiedenen Datenmodellen basieren. Eine abstrakte Parser-Schnittstelle ermöglicht es eigene Parser anzubinden. Die erzeugten logischen Anfragepläne stellen die Metadaten einer Anfrage als Baumstruktur bereit. Die dazu erforderlichen Verwaltungsstrukturen werden von einer logischen Algebra-Basis [KS09] in Form abstrakter logischer Algebra-Operatoren als Fixpunkt bereitgestellt. Die logische Algebra-Basis ist erweiterbar. Durch die Entwicklung datenmodellabhängiger Operatoren können verschiedene Datenmodelle unterstützt werden. Mittels Vererbung können die speziellen Operatoren auf die Verwaltungsstrukturen der logischen Algebra zugreifen. Dazu definiert Odysseus eine Operatoren-Hierarchie. Die oberste Ebene bilden die abstrakten Operatoren, die die notwendigen Verwaltungsstrukturen für den Aufbau von Anfragepläne bereitstellen. Darunter befinden sich die konkreten allgemeinen Operatoren, die auf verschiedenen Datenmodellen anwendbar sind. Die unterste Ebene bilden die datenmodellspezifischen Operatoren, die sowohl abstrakte als auch allgemeine Operatoren erweitern können. Zur Zeit bietet Odysseus eine relationale Datenstromalgebra sowie eine Datenstromalgebra für RDF-Daten an [BGJ+09].

Da logische Anfragepläne stets auf gleiche Art und Weise aus einem datenmodellabhängigen, abstrakten Syntaxbaum generiert werden, müssen sie oft zur Optimierung der Verarbeitungseffizienz restrukturiert werden. Dies übernimmt in Odysseus die Restrukturierungskomponente. Sie arbeitet regelbasiert und ist flexibel gehalten, um neue Forschungsergebnisse integrieren zu können. Sie besteht aus einer einheitlichen Schnittstelle, über die eine Regelengine als Eingabe einen logischen Anfrageplan und eine Menge von Restrukturierungsregeln erhält und den restruktierten Anfrageplan zurückgibt. Zur Unterstützung

verschiedener Datenmodelle erlaubt Odysseus die Definition und Verarbeitung beliebiger Regeln. Als Regelengine kommt Drools[1] zum Einsatz [BGJ+09].

Nach der Restrukturierung muss jeder logische Anfrageplan in einen äquivalenten physischen Anfrageplan transformiert werden, da logische Anfragepläne nur Metadaten über die Anfrage enthalten und zu einem logischen Operator mehrere Implementierungen existieren können [KS09]. Dies wird in Odysseus durch die Transformationskomponente realisiert, die ebenfalls aus einer einheitlichen Schnittstelle zu einer Regelengine besteht. Dabei werden im Anfrageplan logische Operatoren durch physische ersetzt. Da es mehrere Implementierungen eines Operators geben kann, können mehrere physische Anfragepläne aus der Transformation resultieren, von denen einer anhand eines Kostenmodells ausgewählt wird. Die Transformationskomponente erlaubt es, beliebige Transformationsregeln zu definieren. Außerdem existiert eine einheitliche Schnittstelle für die Auswertung von Kostenmodellen. Somit lassen sich durch das Implementieren dieser Schnittstelle beliebige Kostenmodelle definieren [BGJ+09].

In Odysseus werden die Verwaltungsstrukturen für physische Anfragepläne als Fixpunkt von der physischen Algebra-Basis [KS09] bereitgestellt. Auch in der physischen Algebra sind abstrakte Operatoren enthalten, die verschiedene Schnittstellen anbieten. Die Ausführung der physischen Anfragepläne ist einheitlich geregelt. Dennoch können Anfragepläne über verschiedene Datenmodelle ausgeführt werden. Erweiterungen können sich durch die einheitliche Schnittstelle auf die eigentliche Datenverarbeitung beschränken. Dazu implementiert die Algebra-Basis eine push-basierte Variante des Open-Next-Close-Protokolls [Gra93]. Die Verknüpfung physischer Operatoren durch einen Publish-Subscribe-Mechanismus ist in der Algebra-Basis gekapselt. Durch Java-Generics und das Strategy-Pattern [GHJV08] wird die Unabhängigkeit der physischen Algebra-Basis vom Datenmodell realisiert. Die abstrakten Operatoren der physischen Algebra-Basis beschreiben lediglich Verwaltungsstrukturen und nicht wie die Daten verarbeitet werden. In einigen Operatoren existieren abstrakte Algorithmen, in denen datenmodellunabhängige Schritte durch in Strategien gekapselte datenmodellabhängige Schritte ergänzt werden. Durch die Implementierung geeigneter Strategien können zusätzliche Datenmodelle integriert und Algorithmen wiederverwendet werden [BGJ+09]. Ein Beispiel für einen solchen Operator stellt der Ripple-Join [HH99] dar.

In Odysseus lassen sich die Datenstromelemente mit Metadaten, z.B. Gültigkeitsintervallen, versehen. Diese werden grundsätzlich jedem Element im Datenstrom angehängt. Dazu wird eine parametrisierte Schnittstelle für Datenstromelemente bereitgestellt, in denen die Parameter die Metadaten repräsentieren. Die Metadaten müssen zudem eine weitere Schnittstelle implementieren. Die Verarbeitung der Metadaten ist in Strategien gekapselt, sodass die physische Algebra-Basis leicht erweiterbar ist. Es können aber auch spezielle Operatoren für den Umgang mit Metadaten entwickelt werden. Die Kapselung der Metadaten ermöglicht eine leichte Erweiterung um neue Verarbeitungsmodi [BGJ+09].

Odysseus verarbeitet Anfragen auf Datenströmen grundsätzlich push-basiert, d.h. dass jedes Element sofort nach dem Eintreffen verarbeitet wird. Dafür wird eine Ausführungsumgebung bereitgestellt, die als Fixpunkt einen Scheduler bietet. Dieser ist durch Puf-

[1] `http://www.jboss.org/drools/` (letzter Aufruf: 27.4.2010)

ferplatzierungsstrategien (PPS) und Schedulingstrategien (ScS) steuerbar [BGJ+09]. Dabei bestimmt die PPS wo Steuerungspunkte (Puffer) im Anfrageplan integriert werden [CHK+07]. Wie Quelloperatoren werden auch Puffer vom Scheduler gesteuert. Die übrigen Operatoren werden durch Pufferausgaben angestoßen, sodass alle direkt verbundenen Operatoren in einem einzigen Schedulingschritt ausgeführt werden. So wird das Scheduling durch die PPS beeinflusst und parametrisiert. In einem Anfrageplan sind in der Regel mehrere Puffer bzw. Quelloperatoren enthalten, deren Ausführungsreihenfolge unterschiedlich gewählt werden kann. Deshalb lässt sich der Scheduler durch verschiedene ScS parametrisieren. Zur Zeit bietet Odysseus mehrere verschiedene PPS und ScS [BGJ+09].

Über die Monitoringkomponente lassen sich zur Laufzeit physische Anfragepläne überwachen. Dazu kann sie sich über einen Publish-Subscribe-Mechanismus an beliebiger Stelle im Anfrageplan registrieren und die von den Operatoren gefeuerten Events über eine abstrakte EventListener-Schnittstelle entgegennehmen. Zur Verarbeitung der Events müssen spezialisierte EventListener implementiert werden. Eine Erweiterung der Event-Bibliothek um eigene Events ist möglich. Die Verarbeitungsreihenfolge der Events wird durch einen über ScS parametrisierbaren Monitoring-Scheduler gesteuert.

3 OSGi Service Platform

Die *OSGi Service Platform* ist eine Softwareumgebung, die ein dynamisches Modulsystem in Java zur Verfügung stellt. Sie wird von der OSGi Alliance, einem Zusammenschluss mehrerer Unternehmen, spezifiziert und erlaubt die dynamische Integration sowie das Fernmanagement von Softwarekomponenten und Diensten. Dazu stellt die Service Platform als Basiskomponente das sogenannte *OSGi-Framework* zur Verfügung, welches einen Container für die Module und Dienste, sowie deren Verwaltung, implementiert [WHKL08].

Die Module (*Bundles*) stellen fachlich oder technisch zusammenhängende Einheiten von Klassen und Ressourcen dar, die eigenständig im Framework installiert und deinstalliert werden können. Dabei sind grundsätzlich alle in einem Bundle enthaltenen Klassen und Ressourcen für andere Bundles unsichtbar. Allerdings lassen sich die Java-Packages eines Bundles explizit exportieren, wodurch darin enthaltene Klassen und Ressourcen auch für andere Bundles nutzbar werden. Um auf exportierte Packages anderer Bundles zugreifen zu können, müssen diese vom verwendenden Bundle explizit importiert werden. Der Import und Export der Packages erfolgt dabei deklarativ über eine im Bundle gespeicherte *Manifest-Datei*. Das OSGi-Framework sorgt dann zur Laufzeit dafür, dass die deklarierten Abhängigkeiten automatisch aufgelöst und alle importierten Packages bereitgestellt werden. Sollte jedoch ein benötigtes Package fehlen, so kann das gesamte Bundle nicht ausgeführt werden. Die übrigen Bundles sind davon aber erstmal nicht betroffen [WHKL08].

Ein Bundle besteht im Allgemeinen aus einem einfachen Java-Archiv (Jar-Datei), das alle Klassen und Ressourcen zusammenfasst. Zusätzlich können weitere Bibliotheken in Form von Jar-Dateien enthalten sein. Das ebenfalls auf oberster Ebene enthaltene Manifest stellt neben den Import- und Export-Abhängigkeiten weitere Metainformationen zur Verfügung, die vom Framework ausgewertet und verwendet werden. Jedes Bundle wird über einen

symbolischen Namen und eine Version eindeutig identifiziert [WHKL08].

Als weiteres Mittel zur Entkopplung stehen innerhalb des OSGi-Frameworks Dienste (*Services*) zur Verfügung. Dabei handelt es sich um ein Java-Objekt, das unter einem Interface-Namen innerhalb des Frameworks bekannt gemacht wird. Zu diesem Zweck wird eine *Service Registry* bereitgestellt, die eine bundleübergreifende Registrierung der Services zentral ermöglicht. So kann die Service Registry von beliebigen Bundles nach angemeldeten Diensten abgefragt werden. Das abfragende Bundle braucht dabei nicht einmal zu wissen, welches Bundle diesen Dienst bereitstellt und wie dieser implementiert ist. Services können sich dynamisch bei der Service Registry an- und abmelden, ohne dabei Rücksicht auf die Bundles nehmen zu müssen, von denen sie verwendet werden. So muss das Bundle immer selbst sicherstellen, dass die verwendeten Services auch tatsächlich zur Verfügung stehen. Das Framework bietet mehrere Konzepte, mit denen auf die dynamische Registrierung und Abmeldung von Services reagiert werden kann [WHKL08].

Von der Implementierung des Frameworks wird stets ein sogenannter *Management Agent* bereitgestellt. Dieser ermöglicht die dynamische Verwaltung der Bundles zur Laufzeit. So lassen sich Bundles installieren, deinstallieren, starten, stoppen oder aktualisieren, ohne das gesamte Softwaresystem anhalten oder neustarten zu müssen. Im einfachsten Fall handelt es sich bei dem Management Agent um eine einfache textbasierte Konsole, die mittels Kommandos eine Verwaltung der Bundles erlaubt [WHKL08].

Außerdem werden vom Framework eine Reihe von *Standard Services* implementiert, welche verschiedene grundlegende und häufig benötigte Dienste als einheitliche und standardisierte Infrastruktur zur Verfügung stellen. So ermöglicht beispielsweise der *Log-Service* das Schreiben von Fehlermeldungen und Warnungen in Protokolldateien [WHKL08].

Das Odysseus-Framework ist auf Basis der OSGi Service Platform implementiert, wobei seine Teilkomponenten in zahlreichen Bundles gekapselt sind. Die Implementierungsstrategie für Odysseus sieht vor, dass Abhängigkeiten zwischen den Bundles grundsätzlich über Services realisiert werden. So können zur Laufzeit z.B. mehrere verschiedene Implementierungen des Parsers, der Transformation und der Restrukturierung im System vorgehalten werden. Der Import von Java-Packages, welche von anderen Bundles zur Verfügung gestellt werden, wird in Odysseus nur zur Auslagerung von Funktionalität genutzt. Der ebenfalls in OSGi mögliche explizite Import ganzer Bundles sollte hingegen unter allen Umständen vermieden werden, da das betreffende Bundle nicht durch ein anderes Bundle ersetzt werden kann, das dieselben Packages exportiert. Bei der Erweiterung bestehender Komponenten (z.B. der Transformationsregeln) werden insbesondere *Fragment Bundles* verwendet. Dies sind spezielle Bundles, die den Classpath eines anderen Bundles (*Host Bundle*) erweitern. So lassen sich dem Host Bundle weitere Klassen und Ressourcen hinzufügen, ohne dieses ändern zu müssen [WHKL08].

4 Tutorial

Für den Erfolg der Projektgruppe ist es unerlässlich, das Odysseus-Framework um weitere Komponenten zu erweitern. Daher wird hier ein Tutorial zur Erweiterung von Odysseus

gegeben, das auf der Präsentation [Gra] basiert und entsprechend erweitert wurde. Im Tutorial wird beispielhaft ein neuer Datenstromoperator implementiert und ins Framework integriert, wobei eine vertikale Anpassung des gesamten Frameworks gezeigt wird. Dafür wird eine aktuelle Eclipse IDE (*Eclipse for RCP/Plug-in Developers*[2]) mit einer installierten OSGi-Implementierung (*Equinox*[3]) vorausgesetzt. Es wird angenommen, dass der Eclipse Workspace alle Bundles von Odysseus in Form eigenständiger Projekte enthält.

Als Beispiel wird ein *Split-Operator* implementiert und in das Odysseus-Framework integriert. Dieser Operator dient der Verteilung eingehender Datenstromelemente auf verschiedene ausgehende Datenströme anhand von benutzerdefinierten Prädikaten [GAW+08]. Formal werden für den Split-Operator n Prädikate und $n + 1$ ausgehende Datenströme definiert. Für jedes eingehende Datum werden die Prädikate der Reihe nach überprüft. Wird ein Prädikat P_x mit $x \leq n$ erfüllt, so wird das Datum an den Ausgabestrom x weitergeleitet und die Verarbeitung mit dem nächsten Element fortgesetzt. Kann ein Datum jedoch keines der Prädikate erfüllen, so wird es an den Ausgabestrom $n+1$ weitergeleitet.

4.1 Definition eines neuen Algebra-Operators

Die Implementierung und Integration des Split-Operators beginnt mit dem Erstellen des logischen Operators. Zunächst wird dazu in Eclipse ein neues OSGi-Plugin (Projekttyp: *Plug-in Project*) mit dem Namen `de.uniol.inf.is.odysseus.logicaloperator.split` angelegt und unter *Target Platform* Equinox ausgewählt.

Im Ordner `src` wird eine neue Java-Klasse angelegt und in Abhängigkeit von der Zahl der Eingangsdatenströme von der entsprechenden Subklasse von `AbstractLogicalOp` abgeleitet. Da der Split-Operator nur einen eingehenden Datenstrom hat, wird hier von `UnaryLogicalOp` abgeleitet. Die Klasse erhält den Namen `SplitAO`, da nach der Namenskonvention von Odysseus alle logischen Operatoren mit `AO` enden müssen.

Damit von der Oberklasse geerbt werden kann, muss das Package `de.uniol.inf.is.odysseus.logicaloperator.base` importiert werden. Dazu wird die Manifest-Datei im Verzeichnis `META-INF` geöffnet und das Package unter *Dependencies* im Bereich *Imported Packages* importiert. Nun erhält die Klasse eine Liste als Attribut, in der später die Prädikate vom Typ `IPredicate` gespeichert werden sollen. Im Manifest muss unter *Dependencies* im Bereich *Required Plugins* das Plugin `de.uniol.inf.is.odysseus.base` hinzugefügt werden, da dort das Interface `IPredicate` liegt.

Anschließend müssen die Methoden `clone()`, `hashCode()` und `equals()` überschrieben, ein Standard-Konstruktor und ein Kopier-Konstruktor erstellt werden. In Odysseus wird die `clone()`-Methode typischerweise durch einen Aufruf des Kopier-Konstruktors implementiert. Um den Zugriff auf Prädikate zu kapseln, sind noch eine Setter- und eine Getter-Methode für die Prädikatliste und eine Methode zum Hinzufügen von Prädikaten erforderlich. Zudem muss noch die abstrakte Methode `getOutputSchema()`

[2] `http://www.eclipse.org/downloads/packages/eclipse-rcpplug-developers/galileosr2` (letzter Aufruf: 23.4.2010)

[3] `http://www.eclipse.org/equinox/` (letzter Aufruf: 23.4.2010)

implementiert werden, die das Ausgabeschema des Operators liefert. In Zukunft kann der Methode auch eine Portnummer als Parameter übergeben werden. Da die Datenelemente im Falle des Split-Operators unverändert bleiben, kann das Eingabeschema zurückgegeben werden. Listing 1 zeigt das zu implementierende Interface der Klasse `SplitAO`.

```
1  public class SplitAO extends UnaryLogicalOp {
2    List<IPredicate<?>> predicates;
3    public SplitAO() {}
4    public SplitAO(SplitAO splitAO) {}
5    public int hashCode() {}
6    public boolean equals(Object obj) {}
7    public AbstractLogicalOperator clone() {}
8    public List<IPredicate<?>> getPredicates() {}
9    public void setPredicates(List<IPredicate<?>> predicates) {}
10   public void addPredicate(IPredicate<?> predicate) {}
11   public SDFAttributeList getOutputSchema() {}
12 }
```

Listing 1: Interface der Klasse `SplitAO`

4.2 Erstellen des physischen Operators

Nun muss ein passender physischer Operator erstellt werden, der die tatsächliche Verarbeitung der Datenstromelemente realisiert. Dazu wird ein neues OSGi-Bundle mit dem Namen `de.uniol.inf.is.odysseus.physicaloperator.split` erstellt. Darin wird eine neue Klasse `SplitPO` angelegt und von `AbstractPipe<R,W>` abgeleitet. Die Namenskonvention schreibt die Endung `PO` für physische Operatoren vor. Damit abgeleitet werden kann, muss das Package `de.uniol.inf.is.odysseus.physical operator.base` unter *Dependencies* in der Manifest-Datei importiert werden.

Da der Split-Operator die Datenstromelemente nicht verändert, sind im Beispiel die eingehenden und die ausgehenden Datenobjekte vom selben Typ. Folglich wird die Klasse `SplitPO` nur durch den generischen Datentyp `T` parametrisiert. Listing 2 zeigt das zu implementierende Interface der Klasse `SplitPO`.

Auch bei physischen Operatoren müssen die geerbten Methoden `clone()`, `equals()` und `hashCode()` überschrieben und ein Kopier-Konstruktor erstellt werden. Auch eine Liste für die Prädikate wird als Attribut benötigt. Ebenso muss das Plugin `de.uniol .inf.is.odysseus.base` wieder importiert werden. Dem Konstruktor der Klasse `SplitPO` muss eine Liste von Prädikaten als Parameter übergeben werden, damit der Algebra-Operator seine Prädikate an den physischen Operator weiterreichen kann. Der Algebra-Operator darf jedoch nicht direkt an den Konstruktor übergeben werden, da sonst eine Abhängigkeit zwischen physischem und logischem Operator geschaffen würde.

Des weiteren muss die Methode `getOutputMode()` überschrieben werden, um eine konfliktäre Manipulation der Daten zu verhindern und die Zahl der Objektkopien zu minimieren. Die Methode muss `INPUT` zurückgegeben, falls die Daten unverändert bleiben.

```
1  public class SplitPO<T> extends AbstractPipe<T,T> {
2    List<IPredicate<? super T>> predicates;
3    public SplitPO(List<IPredicate<? super T>> predicates) {}
4    public SplitPO(SplitPO<T> splitpo) {}
5    public int hashCode() {}
6    public boolean equals(Object obj) {}
7    public SplitPO<T> clone() {}
8    public OutputMode getOutputMode() {}
9    protected void process_next(T object, int port) {}
10   protected void process_open() throws OpenFailedException {}
11   protected void process_done() {}
12 }
```

Listing 2: Interface der Klasse `SplitPO`

Werden die Daten manipuliert, muss `MODIFIED_INPUT` zurückgegeben werden. Werden neue Elemente erzeugt, muss `NEW_ELEMENT` geliefert werden. Im Beispiel bleiben die eingehenden Daten unverändert, sodass `INPUT` zurückgegeben wird.

Da die Verarbeitung der Datenstromelemente nach dem Open-Next-Close-Protokoll erfolgt, muss zur Implementierung der Funktionalität nur die Methode `process_next()` implementiert werden. Darin werden sämtliche Verarbeitungsschritte umgesetzt. Die Methoden `process_open()` bzw. `process_done()` müssen hingegen nur implementiert, wenn der Operator einmalig bestimmte Vor- bzw. Nachbereitungen für die Verarbeitung des eingehenden Datenstroms erfordert.

Bevor der Split-Operator die Prädikate auswerten kann, müssen diese initialisiert werden. Dazu wird in `process_open()` über die Prädikatenliste iteriert und für jedes darin enthaltene Prädikat die `init()`-Methode aufgerufen (siehe Listing 3). Dabei muss `process_open()` zuerst für die Oberklasse aufgerufen werden. Eine Implementierung der `process_done()`-Methode ist nur dann erforderlich, wenn z.B. nach der Verarbeitung eine Verbindung zu externen Programmen oder Datenquellen geschlossen werden muss.

Die Implementierung der Funktionalität ist für den Split-Operator ebenfalls sehr einfach (siehe Listing 3). So wird lediglich mit Hilfe einer Zählervariablen `i` über die Prädikatenliste iteriert und jedes Prädikat für das aktuelle Datum evaluiert. Erfüllt das Datum das Prädikat, so wird es über die Methode `transfer(object, i)` an den i-ten Ausgangsdatenstrom weitergeleitet und die Verarbeitung anschließend beendet. Kann das Datum keines der Prädikate erfüllen, so wird es an den letzten Ausgabestrom übergeben.

4.3 Erstellen der Tranformations- und Restrukturierungsregeln

Nun werden die Transformationsregeln erstellt, anhand derer die Transformationskomponente einen logischen in einen physischen Operator übersetzen kann. Auch die Restrukturierung des logischen Anfrageplans benötigt spezielle Restrukturierungsregeln. Da Drools in Odysseus als Regelengine eingesetzt wird, müssen die Regeln in der *Drools Rule Language* geschrieben werden.

```
protected void process_open() throws OpenFailedException {
  super.process_open();
  for (IPredicate<? super T> p: predicates) {
    p.init();
  }
}
protected void process_next(T object, int port) {
  for (int i = 0; i < predicates.size(); i++) {
    if (predicates.get(i).evaluate(object)) {
      transfer(object, i);
      return;
    }
  }
  transfer(object, predicates.size());
}
```

Listing 3: Implementierung der Methoden `process_next()` und `process_open()`

Zunächst werden zwei neue Fragment Bundles (Projekttyp: *Fragment Project*) erstellt. Das Bundle für die Restrukturierungsregeln benötigt dabei das Paket `de.uniol.inf.is.odysseus.rewrite.drools` als Host Bundle und erhält den Namen `de.uniol.inf.is.odysseus.rewrite.split`. Das Transformations-Bundle wird `de.uniol.inf.is.odysseus.transformation.split` genannt und `de.uniol.inf.is.odysseus.transformation.drools` als Host Bundle ausgewählt. Anschließend werden die Regel-Dateien (*Rule Resource*) angelegt. Die Dateinamen müssen mit `T` (Transformationsregeln) bzw. `R` (Restrukturierungsregeln) beginnen, werden also `TSplitAO.drl` bzw. `RSplitAO.drl` genannt. Sie werden unter `resources/transformation/rules` bzw. `resources/rewrite/rules` gespeichert. Obwohl das Beispiel keine Restrukturierungsregeln erfordert, müssen beide Regel-Dateien erstellt werden.

```
rule "SplitAO -> SplitPO"
  salience 0
  ruleflow-group "transformation"
  no-loop true
  when
    $SplitAO : SplitAO( allPhysicalInputSet == true )
    $trafo : TransformationConfiguration(dataType == "relational")
  then
    SplitPO splitpo = new SplitPO($splitAO.getPredicates());
    splitPO.setOutputSchema($splitAO.getOutputSchema());
    Collection<ILogicalOperator> toUpdate = TransformationHelper.
        replace($splitAO, splitpo);
    for (ILogicalOperator o: toUpdate) {
      uodate(o);
    }
    retract($splitAO)
end
```

Listing 4: Tranformationsregel für den Split-Operator

Die Übersetzungsregel ist in Listing 4 zu sehen. Die Transformation erfolgt von der Quelle zur Senke, sodass die Regel nur auf Objekte vom Typ `SplitAO` mit bereits vollständig übersetzten Eingangsoperatoren angewendet wird. Über die Abfrage der `TransformationConfiguration` wird sichergestellt, dass die Regel nur ausgeführt wird, wenn die eingehenden Datenelemente relationale Tupel sind. Folglich kann der Split-Operator nur relationale Tupel verarbeiten. Auf gleiche Weise kann die Verwendung bestimmter Metadatentypen sichergestellt werden. Werden beide Bedingung erfüllt, so wird ein neuer physischer Split-Operator erzeugt und dessen Ausgabeschema gesetzt. Anschließend wird der logische Operator mittels der Hilfsklasse `TransformationHelper` durch den neuen physischen Operator ersetzt. Danach müssen über die Methode `update(o)` die internen Speicherstrukturen von Drools aktualisiert werden. Zum Schluss wird mit `retract($splitAO)` der logische Operator aus dem Drools-Objektspeicher entfernt.

Über die Drools-Variable `ruleflow-group` wird die Regel dem Regelpaket `transformation` zugewiesen werden. Konflikte mit bereits existierenden Regeln müssen vermieden werden. Da die Regelbasis auch spezielle Regeln enthalten kann, wird über die Variable `salience` die Priorität der Regel angegeben. In Odysseus gilt die Konvention, dass für jeden von einem Operator unterstützten Metadatentyp (außer `ITimeInterval` und `IPN`) die Priorität, von 0 beginnend, um 5 Punkte erhöht wird.

4.4 Anpassen des PQL-Parsers

Um den neuen Operator auch in prozeduralen Anfragen verwenden zu können, muss die Grammatik des Parsers in der Datei `ProceduralExpressionParser.jjt` im Package `de.uniol.inf.is.odysseus.pqlhack` angepasst werden. Zunächst wird ein neues Token für den Split-Operator definiert, wozu die Produktionsregel für die Schlüsselwort-Token erweitert wird (siehe Listing 5, Zeile 3). Die Produktionsregel für die Algebra-Operatoren wird um den Split-Operator erweitert (Listing 5, Zeile 6) und neue Produktionsregel eingeführt, die die Syntax des Split-Operators beschreibt (Listing 5, Zeile 9-11).

```
1  TOKEN : /* Keywords */ {
2          ...
3  |       <K_SPLIT: "SPLIT">
4  }
5  void AlgebraOp():{}{
6          SplitOp() |
7          ...
8  }
9  void SplitOp(): {}{
10         <K_SPLIT>"(" AlgebraOp() "," (Predicate())*")"
11 }
```

Listing 5: Erweiterung der Grammatik

Nun muss mit *JJTree* aus der Grammatik die Datei `ProceduralExpressionParser.jj` erzeugt werden. Mittels *JavaCC* lässt sich daraus dann der Parser generieren. Da-

bei dürfen bereits existierende Dateien nicht überschrieben werden. Nach der Erzeugung müssen die Visitor-Klassen des Parsers händisch angepasst werden. Für die Klasse `DefaultVisitor` ist es ausreichend, die fehlenden Methoden automatisch von Eclipse erzeugen zu lassen. In der Klasse `CreateLogicalPlanVisitor` muss innerhalb der zu implementierenden visit-Methode aus dem Split-Operator-Knoten des Syntaxbaums ein logischer Split-Operator erzeugt werden. Dazu wird ein Objekt der Klasse `SplitAO` erzeugt. Der erste Kindknoten des Syntaxbaums enthält den eingehenden Operator, der ausgewertet und vom Split-Operator subskribiert werden muss. Alle weiteren Kindknoten stellen die Prädikate dar, die ebenfalls ausgewertet, initialisiert und an den Split-Operator übergeben werden müssen. Abbildung 2 zeigt den Aufbau des Syntaxbaums für den Split-Operator und Listing 6 eine beispielhafte Implementierung der visit-Methode. Damit ist der Split-Operator fertiggestellt und kann nun in PQL-Anfragen genutzt werden.

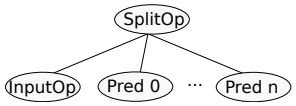

Abbildung 2: Aufbau des Syntaxbaums für den Split-Operator

```
1  public Object visit(ASTSplitOp node, Object data) {
2    SplitAO split = new SplitAO();
3    ArrayList newData = new ArrayList();
4    newData.add(((ArrayList)data).get(0));
5    AbstractLogicalOperator inputForSplit = (AbstractLogicalOperator
         )((ArrayList)node.jjtGetChild(0).jjtAccept(this, newData)).
         get(1);
6    split.subscribeTo(inputForSplit, inputForSplit.getOutputSchema()
         );
7    newData = new ArrayList();
8    newData.add(((ArrayList)data).get(0));
9    for(int i = 1; i<node.jjtGetNumChildren(); i++){
10     IPredicate predicate = (IPredicate)((ArrayList)node.
           jjtGetChild(i).jjtAccept(this, newData)).get(1);
11     initPredicate(predicate, split.getInputSchema(), null);
12     split.addPredicate(predicate);
13   }
14   ((ArrayList)data).add(split);
15   return data;
16 }
```

Listing 6: Implementierung der visit-Methode

5 Zusammenfassung

Odysseus ist ein komponentenbasiertes Framework zur Entwicklung von DSMS, dessen Fixpunkte alle grundlegenden Verwaltungsstrukturen zur Verarbeitung von Datenströmen

bereitstellen. Aufgrund von zahlreichen Variationspunkten ist es darüber hinaus leicht zu erweitern und somit an verschiedene Anwendungskontexte anpassbar. Dies wird durch den Einsatz der OSGi Service Platform unterstützt, welche ein flexibles Modularisierungssystem zur Verfügung stellt und den einfachen Austausch von Softwarekomponenten sogar zur Laufzeit ermöglicht.

Die einfache Erweiterbarkeit von Odysseus wurde in einem Tutorial gezeigt, in der beispielhaft ein neuer Datenstromoperator implementiert und ins Framework integriert wurde. Dabei wurden zunächst der logische Split-Operator sowie der physische Split-Operator erstellt. Anschließend wurde die Regelbasis um eine Übersetzungsregel zur Abbildung des logischen auf den physischen Operator erweitert. Zu guter Letzt wurde die Anpassung der Parser-Grammatik gezeigt, sodass sich der Operator auch in einer PQL-Anfrage verwenden lässt.

Literatur

[BGJ+09] André Bolles, Marco Grawunder, Jonas Jacobi, Daniela Nicklas und Hans-Jürgen Appelrath. Odysseus: Ein Framework für maßgeschneiderte Datenstrommanagementsysteme. *Informatik 2009 - Im Fokus das Leben*, 2009.

[CHK+07] Michael Cammert, Christoph Heinz, Jürgen Krämer, Bernhard Seeger, Sonny Vaupel und Udo Wolske. Flexible Multi-Threaded Scheduling for Continuous Queries over Data Streams. *Data Engineering Workshops, 22nd International Conference on*, 0, 2007.

[GAW+08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu und Myungcheol Doo. SPADE: the system s declarative stream processing engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, Seiten 1123–1134, New York, NY, USA, 2008. ACM.

[GHJV08] Erich Gamma, Richard Helm, Ralph E. Johnson und John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 2008.

[Gra] Marco Grawunder. How to create new query operators in odysseus.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), 6 1993.

[HH99] Peter J. Haas und Joseph M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, Seiten 287–298, New York, NY, USA, 1999. ACM.

[KS09] Jürgen Krämer und Bernhard Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.*, 34(1):1–49, 2009.

[WHKL08] Gerd Wütherich, Nils Hartmann, Bernd Kolb und Matthias Lübken. *Die OSGi Service Platform*. dpunkt.verlag, 2008.