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 kurz 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), welches das erforderliche Grundgerüst zur Verarbeitung von Datenströmen bereitstellt. Durch seine komponentenbasierte Architektur ist es leicht an verschiedene Anwendungen unterschiedlicher Domänen anpassbar. Dabei stellen die Komponenten die notwendige Infrastruktur zur Verfügung, um den gesamten Workflow eines DSMS abzudecken. Innerhalb dieser Komponenten wird zwischen *Fixpunkten* und

Variationspunkten unterschieden. Fixpunkte sind interne Verwaltungsstrukturen, die die Ausführung einzelner Verarbeitungsschritte übernehmen und unabänderlich in das Framework integriert sind. Variationspunkte sind hingegen einzelne, veränderbare Verarbeitungsschritte [BGJ+09]. Abbildung 1 zeigt eine Übersicht der Architektur von Odysseus und seiner Komponenten. Im Folgenden werden die einzelnen Komponenten mit ihren Aufgaben erläutert.

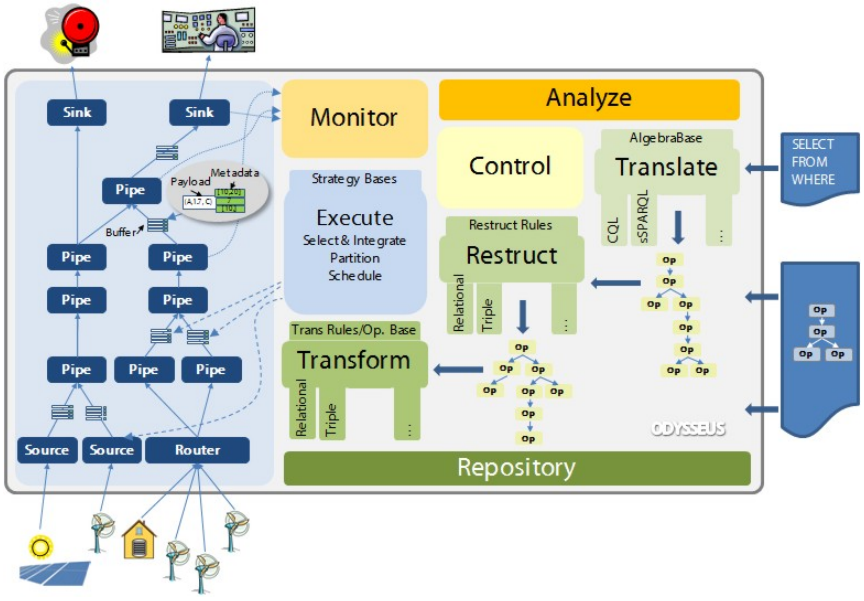

Abbildung 1: Architekturübersicht von Odysseus [BGJ+09]

Die Aufgabe der Translate-Komponente besteht in der Übersetzung von Anfragen in logische Anfragepläne. Dabei können die Anfragen in unterschiedlichen Anfragesprachen formuliert sein und auf verschiedenen Datenmodellen basieren. Eine abstrakte Parser-Schnittstelle ermöglicht es eigene Parser anzubinden. Die bei der Übersetzung erzeugten logischen Anfragepläne stellen die Metadaten einer Anfrage in einer Baumstruktur bereit. Die dazu erforderlichen Verwaltungsstrukturen werden von einer logischen Algebra-Basis [KS09] in Form abstrakter logischer Algebra-Operatoren als Fixpunkt bereitgestellt. Für den Aufbau logischer Anfragepläne bieten die abstrakten logischen Operatoren Mechanismen zur Verknüpfung von Operatoren sowie zum Bereitstellen von Ein- und Ausgabeschemata an. Dabei ist die logische Algebra-Basis erweiterbar gehalten. So können von Odysseus durch die Entwicklung datenmodellabhängiger Operatoren verschiedene Datenmodelle unterstützt werden. Mittels Vererbung können diese speziellen Operatoren dann auf die Verwaltungsstrukturen der logischen Algebra zugreifen. Um dieses zu ermöglichen definiert Odysseus eine Hierarchie für die logischen Operatoren. Die oberste Ebene bilden die bereits erwähnten abstrakten Operatoren, die die notwendigen Verwaltungsstrukturen für den Aufbau von Anfragepläne bereitstellen. Darunter befinden sich die allgemeinen Operatoren. Dies sind konkrete Operatoren, die auf verschiedenen Datenmodellen

anwendbar sind. Die unterste Ebene bilden die datenmodellspezifischen Operatoren. Sie können sowohl als Erweiterung abstrakter Operatoren, als auch als Erweiterung allgemeiner Operatoren implementiert werden. Zur Zeit bietet das Framework eine relationale Datenstromalgebra sowie eine Datenstromalgebra für RDF-Daten an [BGJ+09].

Da logische Anfragepläne stets auf die gleiche Art und Weise aus einem vom Datenmodell abhängigen, abstrakten Syntaxbaum generiert werden, müssen sie in der Regel zur Optimierung der Verarbeitungseffizienz restrukturiert werden. Diese Aufgabe wird in Odysseus von der Restrukturierungskomponente übernommen. Die Komponente arbeitet regelbasiert und ist flexibel gehalten, um neue Forschungsergebnisse integrieren zu können. Sie besteht aus einer für alle Datenmodelle einheitlichen Schnittstelle zu einer Regelengine. Darüber erhält die Regelengine als Eingabe einen logischen Anfrageplan und eine Menge von Restrukturierungsregeln. Auch der restruktierte Anfrageplan wird über die Schnittstelle an die Regelausführungsumgebung zurück gegeben. Da verschiedene Datenmodelle unterschiedliche Regelmengen erfordern, erlaubt Odysseus an dieser Stelle die Definition und Verarbeitung beliebiger Regeln. Als Regelengine kommt hier Drools[1] zum Einsatz [BGJ+09].

Nach der Restrukturierung muss jeder logische Anfrageplan in einen äquivalenten physischen Anfrageplan transformiert werden, da logische Anfrageläne nur Metadaten über die Anfrage enthalten und zu einem logischen Operator mehrere Implementierungen existieren können [KS09]. Diese Transformationen werden in Odysseus durch die Transformationskomponente realisiert, die wie die Restrukturierungskomponente aus einer für alle Datenmodelle einheitlichen Schnittstelle zu einer Regelengine besteht. Dabei werden die logischen Operatoren im Anfrageplan durch physische Operatoren ersetzt. Da es wie bereits erwähnt mehrere Implementierungen eines Operators geben kann, können mehrere physische Anfragepläne aus der Transformation resultieren. Deshalb wird aus der Menge von möglichen physischen Anfrageplänen anhand eines Kostenmodells ein geeigneter Plan ermittelt und ausgewählt. Auch die Transformationskomponente bietet einige Variationspunkte. So lassen sich beispielsweise beliebige Regeln für die Transformation 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, insbesondere die, die für deren Aufbau notwendig sind, 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. Um dieses zu gewährleisten implementiert die Algebra-Basis eine push-basierte Variante des Open-Next-Close-Protokolls [Gra93], wodurch eine Kapselung der *Open*-, *Process_Next*- und *Close*-Methode für datenmodellspezifische Algorithmen erreicht wird. Die Verknüpfung physischer Operatoren durch einen Publish-Subscribe-Mechanismus ist zudem in der Algebra-Basis gekapselt. Durch die Verwendung von Java-Generics und des Strategy-Patterns [GHJV08] wird die Unabhängigkeit

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

der physischen Algebra-Basis vom Datenmodell realisiert. Die in der physischen Algebra-Basis enthaltenen abstrakten Operatoren beschreiben lediglich Verwaltungsstrukturen und nicht wie die Daten verarbeitet werden. In einigen Operatoren, die die Kapselung von datenmodellspezifischen Eigenschaften erlauben, existieren sogar abstrakte Algorithmen. In einem solchen Algorithmus werden die datenmodellunabhängigen Schritte durch datenmodellabhängige Schritte, die durch das Strategy-Pattern gekapselt sind, ergänzt. Somit ist es möglich, die Erweiterung um ein zusätzliches Datenmodell durch die Implementierung einer geeigneten Strategie zu integrieren und gleichzeitig den Algorithmus wiederzuverwenden [BGJ+09]. Ein Beispiel für einen solchen Operator stellt der Ripple-Join [HH99] dar.

In Odysseus lassen sich die Datenstromelemente mit Metadaten, wie beispielsweise Gültigkeitsintervallen, versehen. Solche Metadaten 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. Darüber hinaus können auch spezielle Operatoren für den Umgang mit Metadaten entwickelt werden. Die Kapselung der Metadaten ermöglicht es, Odysseus leicht um neue Verarbeitungsmodi zu erweitern [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. Der Scheduler ist durch Pufferplatzierungsstrategien (PPS) und Schedulingstrategien (ScS) steuerbar [BGJ+09]. Dabei bestimmt die PPS wo Steuerungspunkte (Puffer) im Anfrageplan integriert werden [CHK+07]. Puffer sind Elementarspeicher, die im physischen Anfrageplan zwischen physischen Operatoren platziert und, ebenso wie Quellzugriffsoperatoren, vom Scheduler gesteuert werden. Die übrigen Operatoren werden durch Pufferausgaben angestoßen, sodass alle physischen Operatoren, die direkt (ohne Puffer) miteinander verbunden sind, in einem einzigen Schedulingschritt ausgeführt werden. Dadurch wird das Scheduling durch die PPS maßgeblich beeinflusst und somit parametrisierbar. In einem Anfrageplan sind in der Regel mehrere Puffer bzw. Quelloperatoren enthalten. Die Ausführungsreihenfolge dieser Operatoren kann unterschiedlich gewählt werden, weshalb sich der Scheduler durch verschiedene ScS parametrisieren lässt. Zur Zeit sind mehrere verschiedene PPS und ScS in Odysseus implementiert [BGJ+09].

Zur Überwachung von Anfrageplänen stellt die Monitoringkomponente eine einheitliche Schnittstelle bereit, die das Auslesen und Auswerten von Informationen über physische Anfragepläne zur Laufzeit ermöglicht. Dies wird über einen Publish-Subscribe-Mechanismus realisiert, der die Verarbeitung von Events, die von einem der physischen Operatoren gefeuert werden, erlaubt. Abstrakte Events dienen hierbei als Fixpunkt und beinhalten Informationen über Art und Ursprung des Events. Über eine abstrakte EventListener-Schnittstelle werden die Events entgegen genommen, jedoch nicht weiter verarbeitet. Für die Verarbeitung der Events müssen spezialisierte EventListener implementiert werden. Die Registrierung von EventListenern an Operatoren ist dank des Publish-Subscribe-Mechanismus auch zur Laufzeit und an beliebigen Stellen im Anfrageplan möglich. Darüber

hinaus lässt sich die Event-Bibliothek um eigene Events erweitern. Das Feuern von spezialisierten Events während der Verarbeitung durch einen Operator erfordert jedoch eine Anpassung des Operators, was in der Regel am einfachsten durch einen spezialisierten Operator mit angepasster Datenstromverarbeitung umgesetzt werden kann. Ein durch ScS parametrisierbarer Monitoring-Scheduler steuert die verschiedenen im Anfrageplan registrierten EventListener und somit die Verarbeitung der Events [BGJ+09].

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, im OSGi-Kontext *Bundle* genannt, 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 die darin enthaltenen Klassen und Ressourcen auch für andere Bundles nutzbar werden. Um auf die exportierten Packages von anderen 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), dass 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 innerhalb des Frameworks über einen symbolischen Namen und eine Version eindeutig identifiziert [WHKL08].

Als ein weiteres Mittel zur Entkopplung stehen innerhalb des OSGi-Frameworks Dienste, sogenannte *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 imple-

mentiert ist. Das Besondere an den Services ist, dass sie sich dynamisch bei der Service Registry an- und abmelden können, 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. Allerdings stehen innerhalb des Frameworks mehrere Konzepte zur Verfügung, 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 anzuhalten 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 dabei vor, dass Abhängigkeiten zwischen den Bundles grundsätzlich über Services realisiert werden. So können zur Laufzeit beispielsweise 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 sogenannte *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, insbesondere um neue Datenstromoperatoren, zu erweitern. Daher wird in diesem Abschnitt ein Tutorial zur Erweiterung von Odysseus gegeben, das auf der Präsentation [Gra] basiert und an den entsprechenden Stellen erweitert wurde. In diesem Tutorial wird beispielhaft ein neuer Datenstromoperator implementiert und ins Framework integriert, wobei eine vertikale Anpassung des gesamten Frameworks gezeigt wird.

Für dieses Tutorial wird eine aktuelle Eclipse IDE (*Eclipse for RCP/Plug-in Developers*[2])

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

mit einer installierten OSGi-Implementierung (*Equinox*[3]) vorausgesetzt. Des weiteren wird angenommen, dass der Eclipse Workspace alle Bundles von Odysseus in Form eigenständiger Eclipse-Projekte enthält.

Als Beispiel wird in diesem Tutorial ein sogenannter *Split-Operator* implementiert und in das Odysseus-Framework integriert. Dieser Operator dient der Verteilung der eingehenden Datenstromelemente auf verschiedene ausgehende Datenströme, die anhand von benutzerdefinierten Prädikaten erfolgt [GAW+08]. Formal werden für einen 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 in der Verarbeitung befindliche 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 als Zielplattform (*Target Platform*) das OSGi-Framework Equinox ausgewählt (siehe Abbildung 2).

Anschließend wird im Ordner `src` eine neue Java-Klasse angelegt und in Abhängigkeit von der Zahl der eingehenden Datenströme von der entsprechenden Unterklasse der abstrakten Klasse `AbstractLogicalOp` abgeleitet. Operatoren mit einem eingehenden Datenstrom erben von der Klasse `UnaryLogicalOp`, Operatoren mit zwei eingehenden Datenströmen von `BinaryLogicalOp`. Da der Split-Operator nur einen eingehenden Datenstrom hat, wird in diesem Fall von `UnaryLogicalOp` abgeleitet. Die neue Klasse erhält den Namen `SplitAO`, da nach der Namenskonvention von Odysseus alle logischen Operatoren mit `AO` enden müssen.

Damit von der Superklasse geerbt werden kann, muss das Odysseus-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 wird der Klasse eine Liste als Attribut hinzugefügt, in der später die Prädikate vom Typ `IPredicate` gespeichert werden sollen. In der Manifest-Datei mus unter *Dependencies* im Bereich *Required Plugins* das Plugin `de.uniol.inf.is.odysseus.base` hinzugefügt werden, da dort das Interface `IPredicate` liegt (siehe Abbildung 3). Zudem lässt man sich von Eclipse das Attribut *serialversionUID* automatisch generieren.

Anschließend müssen die Methoden `clone()`, `hashCode()` und `equals()` überschrieben und ein Standard-Konstruktor sowie ein Kopier-Konstruktor erstellt werden. In Odysseus wird dabei typischerweise die `clone()`-Methode durch einen Aufruf des Kopier-Konstruktors implementiert. Um den Zugriff auf Prädikate zu kapseln, sind noch

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

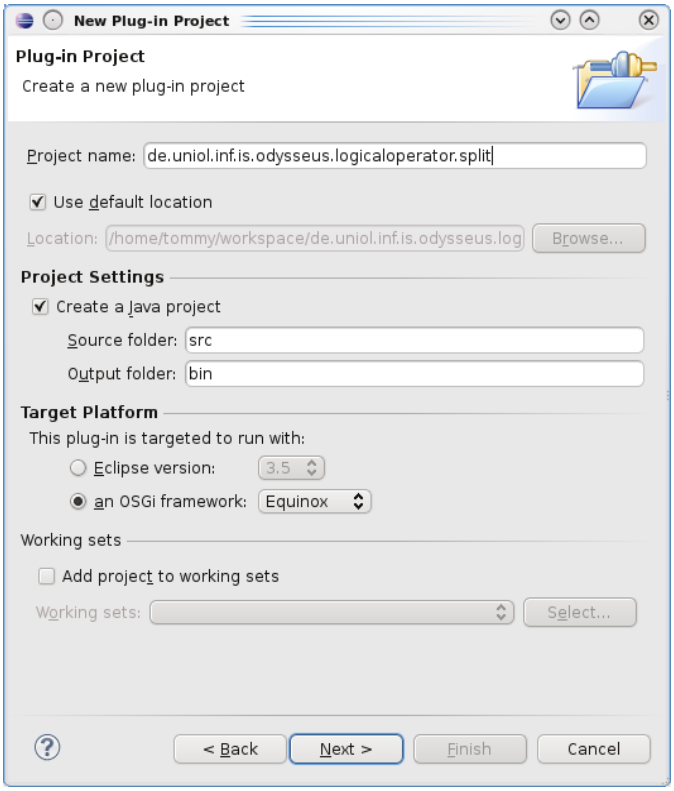

Abbildung 2: Erstellen eines neuen OSGi-Bundles für den logischen Operator

eine Setter- und eine Getter-Methode für die Prädikatliste und eine Methode zum einfachen Hinzufügen von Prädikaten erforderlich. Zudem muss noch die abstrakte Methode `getOutputSchema()` implementiert werden, die das Ausgabeschema des Operators liefert. In Zukunft kann der Methode auch eine Portnumemr als Parameter übergeben werden. Da die Datenelemente im Falle des Split-Operators unverändert bleiben, kann in diesem Fall einfach über die geerbte Methode `getInputSchema()` das Eingabeschema zurückgegeben werden. Code-Listing 1 gibt einen Überblick über das zu implementierende Interface der Klasse `SplitAO`.

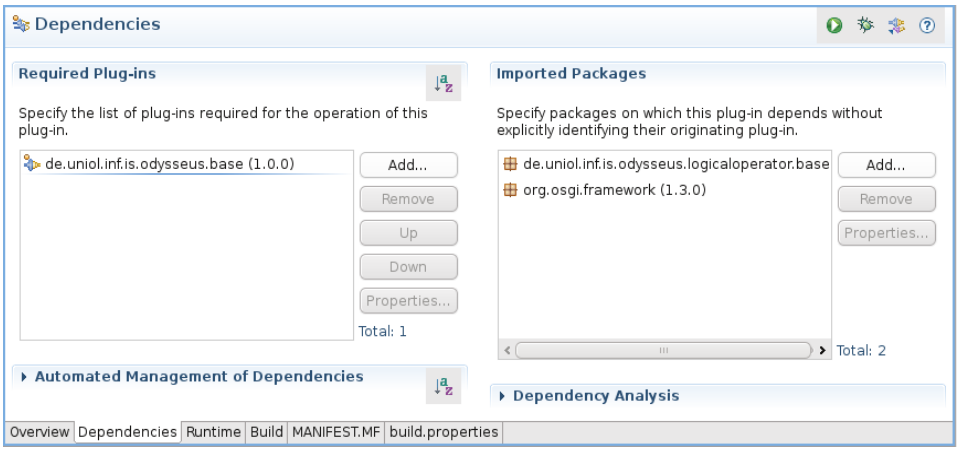

Abbildung 3: Bundle-Abhängigkeiten für den logischen Operator

```
1  public class SplitAO extends UnaryLogicalOp {
2
3    private static final long serialVersionUID;
4    List<IPredicate<?>> predicates;
5
6    public SplitAO() {}
7    public SplitAO(SplitAO splitAO) {}
8    public int hashCode() {}
9    public boolean equals(Object obj) {}
10   public AbstractLogicalOperator clone() {}
11   public List<IPredicate<?>> getPredicates() {}
12   public void setPredicates(List<IPredicate<?>> predicates) {}
13   public void addPredicate(IPredicate<?> predicate) {}
14   public SDFAttributeList getOutputSchema() {}
15 }
```

Listing 1: Interface der Klasse `SplitAO`

4.2 Erstellen des physischen Operators

Nachdem im vorangegangenen Abschnitt ein logischer Algebra-Operator implementiert wurde, muss nun ein passender physischer Operator erstellt werden, der die tatsächliche Verarbeitung der Datenstromelemente realisiert. Dazu wird wieder ein neues OSGi-Bundle mit dem Namen `de.uniol.inf.is.odysseus.physicaloperator.split` erstellt. Darin wird eine neue Klasse `SplitPO` angelegt, wobei die Namenskonvention die Endung `PO` für physische Operatoren vorschreibt. Die neu erstellte Klasse wird von der abstrakten Klasse `AbstractPipe<R,W>` abgeleitet. Der generische Typ `R` beschreibt den Objekttyp, der von dem physischen Operator gelesen werden soll, wohingegen `W` den Objekttyp der zu schreibenden Datenelemente angibt. Damit von der Klasse geerbt werden kann, muss das Package `de.uniol.inf.is.odysseus.physicaloperator.base` unter *Dependencies* in der Manifest-Datei importiert werden (siehe Abbildung 4).

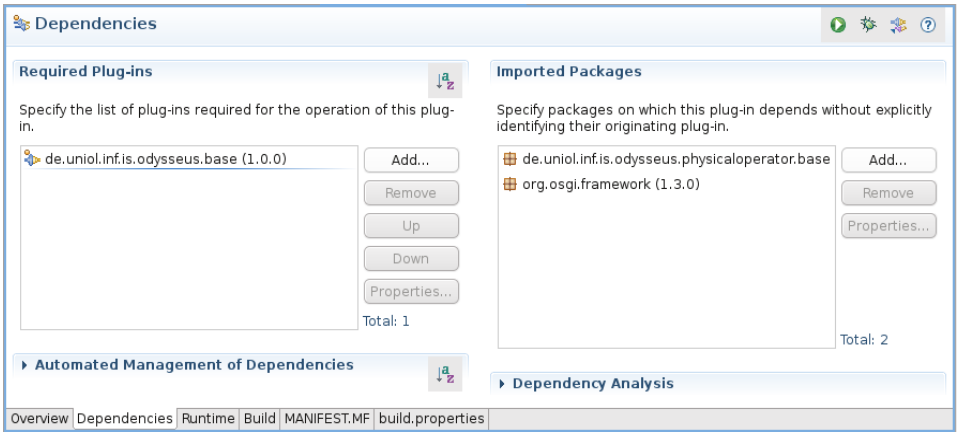

Abbildung 4: Bundle-Abhängigkeiten für den physischen Operator

Da der Split-Operator die Datenstromelemente nicht verändert, sondern lediglich auf verschiedene Datenströme verteilt, sind in diesem Beispiel die eingehenden und die ausgehenden Datenobjekte vom selben Typ. Folglich wird die Klasse `SplitPO` nur durch den generischen Datentyp `T` parametrisiert, welcher bei der Erweiterung von `AbstractPipe<R,W>` sowohl für `R` als auch für `W` gesetzt wird. Dies ist in der ersten Zeile von Listing 2 zu sehen, welches das Interface der Klasse `SplitPO` beschreibt.

Wie bei Algebra-Operatoren müssen auch bei physischen Operator die geerbten Methoden `clone()`, `equals()` und `hashCode()` überschrieben und ein Kopier-Konstruktor erstellt werden. Auch eine Liste für die Prädikate wird wieder als Attribut benötigt. Damit die Liste mit dem Java-Interface `IPredicate` typisiert werden kann, muss zusätzlich das Plugin `de.uniol.inf.is.odysseus.base` importiert werden. Dem Konstruktor der Klasse `SplitPO` muss eine Liste gleichen Typs als Parameter übergeben werden, damit später die Prädikatenliste vom Algebra-Operator an den physischen Operator weitergereicht werden kann. Der Algebra-Operator darf jedoch nicht direkt an den Konstruktor übergeben werden, da sonst eine Abhängigkeit des physischen Operators vom logischen

```
1  public class SplitPO<T> extends AbstractPipe<T,T> {
2    List<IPredicate<? super T>> predicates;
3
4    public SplitPO(List<IPredicate<? super T>> predicates) {}
5    public SplitPO(SplitPO<T> splitpo) {}
6    public int hashCode() {}
7    public boolean equals(Object obj) {}
8    public SplitPO<T> clone() {}
9    public OutputMode getOutputMode() {}
10   protected void process_next(T object, int port) {}
11   protected void process_open() throws OpenFailedException {}
12   protected void process_done() {}
13 }
```

Listing 2: Interface der Klasse `SplitPO`

Operator geschaffen würde.

Des weiteren muss die Methode `getOutputMode()` überschrieben werden. Sie wird benötigt, um eine konfliktäre Manipulation der Datenstromelemente zu verhindern und gleichzeitig die Zahl der Objektkopien so gering wie möglich zu halten. Der Rückgabewert dieser Methode gibt in Form eines Enum-Wertes an, wie der Operator mit den eingehenden Datenelementen umgeht. Der Wert `INPUT` gibt dabei an, dass die eingehenden Datenelemente vom Operator ausschließlich gelesen werden, wie es z.B. bei der Selektion der Fall ist. Verändert der Operator hingegen wie bei der Projektion die gelesenen Elemente, so muss `MODIFIED_INPUT` zurückgegeben werden. Der Wert `NEW_ELEMENT` gibt hingegen an, dass ein Operator neue Elemente erzeugt und in den ausgehenden Datenstrom einfügt, wie es z.B. der Join-Operator tut. Im Fall des Split-Operators bleiben die eingehenden Daten unverändert, sodass der Enum-Wert `INPUT` zurückgegeben werden muss.

Da die Verarbeitung der Datenstromelemente, wie in Abschnitt 2 bereits erläutert, nach dem Open-Next-Close-Protokoll erfolgt, muss zur Implementierung der eigentlichen Funktionalität lediglich 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 zunächst einmal initialisiert werden. Dazu wird in der Methode `process_open()` über die Prädikatenliste iteriert und für jedes darin enthaltene Prädikat die `init()`-Methode aufgerufen (siehe Listing 3). In der Implementierung ist ebenfalls zu sehen, dass `process_open()` zuallererst für die Oberklasse aufgerufen werden muss. Eine Implementierung der `process_done()`-Methode ist erforderlich, wenn z.B. ein Prädikat oder eine Verbindung zu externen Programmen oder Datenquellen nutzt, die nach dem Ende der Verarbeitung 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 über die gleiche Methode an den letzten Ausgabestrom übergeben.

```
1  protected void process_open() throws OpenFailedException {
2    super.process_open();
3    for (IPredicate<? super T> p: predicates) {
4      p.init();
5    }
6  }
7
8  protected void process_next(T object, int port) {
9    for (int i = 0; i < predicates.size(); i++) {
10     if (predicates.get(i).evaluate(object)) {
11       transfer(object, i);
12       return;
13     }
14   }
15   transfer(object, predicates.size());
16 }
```

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

4.3 Erstellen der Tranformations- und Restrukturierungsregeln

Nachdem sowohl der logische als auch der physische Operator fertiggestellt wurden, muss nun der Zusammenhang zwischen diesen hergestellt werden. In Odysseus geschieht dies durch das Erstellen einer oder mehrerer Transformationsregeln, 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.

Zunächst einmal müssen zwei neue Fragment Bundles (Projekttyp: *Fragment Project*) erstellt werden. 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 Bundle für die Transformation wird `de.uniol.inf.is.odysseus.transformation.split` genannt und `de.uniol.inf.is.odysseus.transformation.drools` als Host Bundle ausgewählt (siehe Abbildungen 5 und 6). Anschließend werden die Dateien für die Regeln (*Rule Resource*) angelegt. Zu beachten ist, dass die Dateien mit `T` (Transformationsregeln) bzw. `R` (Restrukturierungsregeln) beginnen müssen und im Verzeichnis `resour`

ces/transformation/rules bzw. resources/rewrite/rules gespeichert werden. So erhalten die Regeln die Dateinamen TSplitAO.drl bzw. RSplitAO.drl. Wichtig ist, dass beide Regel-Dateien erstellt werden, auch wenn eine Restrukturierungsregel für dieses Beispiel nicht erforderlich ist.

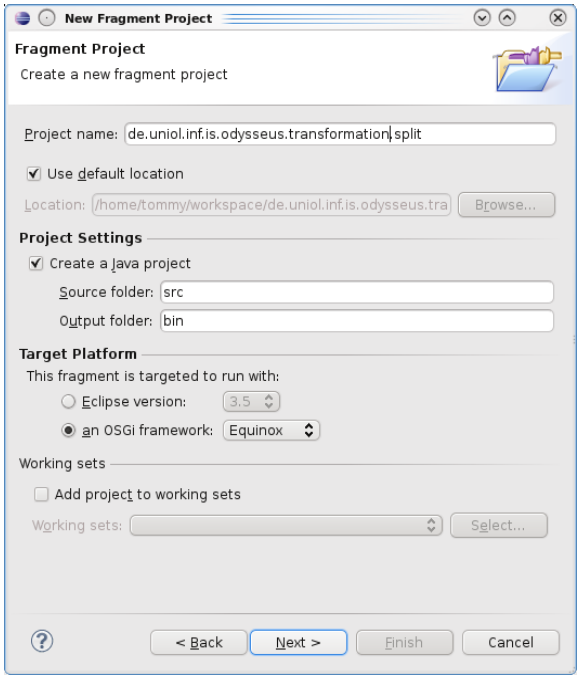

Abbildung 5: Erstellen des Fragment Bundles für die Transformationsregeln

Die Regel zur Übersetzung des logischen Split-Operators in den entsprechenden physischen Operator ist in Listing 4 zu sehen. Eine Erläuterung der einzelnen Sprachelemente kann an dieser Stelle nicht gegeben werden, da das den Umfang dieser Arbeit bei weitem übersteigen würde. Daher sei auf die Onlinedokumentation[4] von Drools verwiesen. Stattdessen wird im Folgenden die Logik der Transformationsregel anhand des Beispiels näher betrachtet und auf einige odysseusspezifische Aspekte eingegangen.

Die Transformation erfolgt bottom-up, also von der Quelle zur Senke, sodass die Regel nur auf Objekte vom Typ SplitAO angewendet wird, deren Eingangsoperatoren bereits vollständig in physische Operatoren übersetzt wurden. Über die Abfrage der TransformationConfiguration wird zudem 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 auch die Verwendung bestimmter Metadatentypen sichergestellt werden. Werden beide Bedingung erfüllt, so wird zunächst ein neuer physischer Split-Operator erzeugt und dessen Ausgabeschema

[4] http://downloads.jboss.com/drools/docs/5.0.1.26597.FINAL/drools-expert/html/index.html (letzter Aufruf: 24.4.2010)

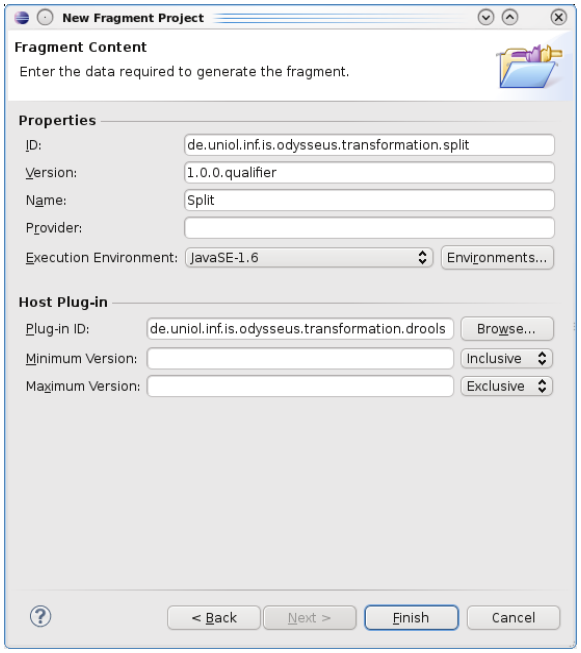

Abbildung 6: Auswahl des Fragment Hosts

gesetzt. Anschließend wird der logische Operator mittels der Hilfsklasse `TransformationHelper` durch den neu erzeugten physischen Operator ersetzt. Danach müssen über die Methode `update(o)` die internen Speicherstrukturen von Drools aktualisiert werden. Damit ist die Transformation abgeschlossen, sodass zu guter Letzt mit `retract($splitAO)` der logische Operator aus dem Drools-Objektspeicher entfernt wird.

Beim Erstellen der Regeln ist zu beachten, dass die Regel dem richtigen Regelpaket zugewiesen wird. Dies geschieht über das Setzen der Drools-Variablen `ruleflow-group`. So müssen Transformationsregeln beispielsweise dem Paket `transformation` zugewiesen werden. Außerdem muss beim Erstellen von Regeln stets auf Konflikte mit bereits existierenden Regeln geachtet werden. Da die Regelbasis neben allgemeinen auch speziellere Regeln enthalten kann, wird über die Variable `salience` die Priorität der Regel angegeben. In Odysseus gilt die Konvention, dass für jeden von einem Operator unterstützen Metadatentyp (mit Ausnahme von `ITimeInterval` und `IPN`) die Priorität, von 0 beginnend, um 5 Punkte erhöht wird.

```
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

4.4 Anpassen des PQL-Parsers

Nachdem auch die Transformationsregel erstellt wurde, ist der Split-Operator in Odysseus benutzbar. Allerdings kann er nur innerhalb eines logischen Algebraplans an Odysseus übergeben werden. Die Möglichkeit, ihn beispielsweise in einer Anfragesprache zu verwenden, besteht allerdings noch nicht. Daher wird in diesem Abschnitt der Parser der prozeduralen Anfragesprache PQL erweitert, mit der logische Anfragepläne direkt definiert werden können.

Zur Integration des neuen Operators muss die Grammatik des Parsers angepasst werden, welche sich in der Datei `ProceduralExpressionParser.jjt` im Package `de.uniol.inf.is.odysseus.pqlhack` befindet. Zunächst muss ein neues Token für den Split-Operator definiert werden, wozu die Produktionsregel für die Schlüsselwort-Token erweitert wird (siehe Listing 5, Zeile 4). Anschließend muss die Produktionsregel für die Algebra-Operatoren um den Split-Operator erweitert werden (Listing 5, Zeile 9). Zu guter Letzt wird noch eine neue Produktionsregel eingeführt, die die Syntax des Split-Operators beschreibt (Listing 5, Zeile 13-15).

```
1  TOKEN : /* Keywords */ {
2          <K_ACCESS: "ACCESS">
3          ...
4  |       <K_SPLIT: "SPLIT">
5  }
6
7  void AlgebraOp():{}{
8          ...
9          SplitOp() |
10         PredictionOp()
11 }
12
13 void SplitOp(): {}{
14         <K_SPLIT>"(" AlgebraOp() "," (Predicate())*")"
15 }
```

Listing 5: Erweiterung der Grammatik

Nachdem die Grammatik erweitert wurde, muss daraus über das Werkzeug *JJTree* zunächst die Datei `ProceduralExpressionParser.jj` erzeugt werden. Mittels *JavaCC* lässt sich daraus im Anschluss der Parser automatisch generieren. Jedoch ist dabei zu beachten, dass bereits existierende Dateien nicht überschrieben werden. Nach der Erzeugung müssen die Visitor-Klassen des Parsers noch händisch angepasst werden. Für die Klasse `DefaultVisitor` ist es ausreichend, die noch fehlende Methode 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 zunächst ein Objekt der Klasse `SplitAO` erzeugt. Der erste Kindknoten des Syntaxbaums enthält den eingehenden Operator, der zunächst ausgewertet und anschließend 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 7 zeigt den Aufbau des Syntaxbaums für einen Split-Operator. Eine beispielhafte Implementierung der visit-Methode ist in Listing 6 zu sehen.

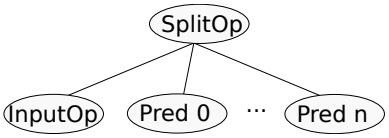

Abbildung 7: Aufbau des Syntaxbaums für den Split-Operator

```
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

Damit ist die Implementierung des Split-Operators abgeschlossen. Dieser lässt sich nun in Anfragen der PQL verwenden.

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.