
TUHH

Technische Universität Hamburg-Harburg

**Diplomarbeit**

# Ein inhaltsbasiertes Publish/Subscribe Architekturkonzept zur individuellen Sensordatenversorgung

erstellt von
**Sylvia Melzer**

*Erstgutachter:*
**Prof. Dr. Joachim W. Schmidt, TUHH**

*Zweitgutachter:*
**Prof. Dr. Helmut Weberpals, TUHH**

*Externe Betreuung:*
**Dr. Vera Kamp, Plath GmbH**

Technische Universität Hamburg–Harburg
21073 Hamburg
GERMANY

*erstellt*
***Januar* 2006**

## Danksagung

An dieser Stelle möchte ich allen danken, die zum Gelingen dieser Arbeit beigetragen haben:

Herrn Prof. Dr. Schmidt danke ich dafür, dass er die Erstgutachtung meiner Diplomarbeit übernommen hat und dass ich die Diplomarbeit in der Firma Plath GmbH absolvieren durfte.

Dr. Vera Kamp danke ich für eine sehr interessante, praxisbezogene und anspruchsvolle Aufgabenstellung, für Ihre Unterstützung und Tipps sowie für Ihre engagierte und lehrreiche Betreuung und die Bereitstellung der vielen Literatur.

Dipl.-Inform. Patrick Hupe danke ich ebenfalls für die engagierte Betreuung meiner Diplomarbeit und für die viele in langen Diskussionen gewonnenen Anregungen und für die Literatur, die er mir ausgeliehen hat.

Prof. Dr. Weberpals danke ich die Zweitgutachtung meiner Diplomarbeit, für die vielen Tipps bzgl. des Layouts und für die Zwischenkorrekturen meiner Arbeit.

Allen Mitarbeitern der Plath GmbH danke ich für die praxisbezogenen Beispiele, die ich für meine Arbeit benötigte sowie für Ihre Unterstützung und Tipps.

Allen Professoren und Mitarbeitern aus dem Fachbereich STS danke ich dafür, dass sie mir stets bei Fragen bzgl. Ontologien und Geosprachen mit Rat und Tat zur Seite standen.

Ein großer Dank geht auch an die Mitarbeiter aus dem Fachbereich Wasserbau und an die Mitarbeiter von deegree, die mir in Bereich der geographischen Sprachen wertvolle Informationen lieferten.

Dipl.-Inform. Thomas Dammann danke ich auch ganz herzlich für seine Unterstützung und die vielen Tipps.

Meinem Ehemann Martin und meinen Eltern danke ich vom ganzen Herzen für die liebevolle Unterstützung und Motivierung während meiner gesamten Studienzeit.

## Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, den 16. Januar 2005

Sylvia Melzer

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

Im Kontext strategischer und taktischer Überwachungssysteme spielen sensorbasierte Systeme eine zentrale Rolle. Sensoren sind Bauteile, die durch die Messung chemischer oder physikalischer Größen qualitativ und quantitativ chemische oder physikalische Eigenschaften erfassen können. Sie ermöglichen Systemen die Wahrnehmung der Umwelt und erlauben über die Verarbeitung dieser Informationen erst die intelligente Reaktion auf die wahrgenommene Umwelt. Sensorbasierte Systeme sind z.B. Frühwarnsysteme oder Funküberwachungssysteme, bei denen einerseits Messdaten langfristig gesammelt, gespeichert und überwacht werden, um diese für Vorhersagen auszuwerten. Andererseits werden die Messdaten so überwacht, dass bei einem verdächtigen Messwert eine sofortige Reaktion ausgelöst wird. In Abbildung 1.1 wird ein solches sensorbasiertes System skizziert.

Im Erfassungssystem befinden sich erfasste Sensordaten, die an interessierte Auswertekomponenten ausgespielt werden. Diese können dabei automatisch (Push) oder erst bei einer direkten Anfrage zu einem bestimmten Zeitpunkt weitergegeben werden (Pull/ Query on demand) [10]. Die Auswertung der Sensordaten erfolgt in der jeweiligen Auswertekomponente.
Die Daten (Sensordaten) werden im Erfassungssystem werden in kontinuierlichen Strömen erfasst. Neben der kontinuierlichen Verwaltung und Verarbeitung großer Datenmengen bildet die Datenweitergabe an nachfolgende Auswertesysteme einen wichtigen Aspekt in dem Bereich des so genannten „sensor based computing“. Die Datenweitergabe bzw. der Zugriff auf die Sensordaten stellt einen zentralen Aspekt dar und hat großen Einfluss auf die Architektur eines sensorbasierten Systems zur Informationsversorgung.

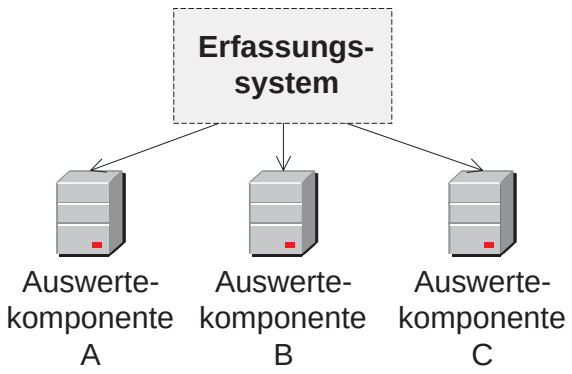


Abbildung 1.1: Sensorbasiertes System

Für die dynamische Interaktion der Systeme bieten sich die drei bekanntesten asynchronen Kommunikationsmodelle Message Passing, Publish/Subscribe und Broadcast an [48]. In diesem Kontext basieren neue Ansätze auf dem Publish/Subscribe Paradigma [59, 25], das einen robusten und flexiblen Mechanismus für eine omnipräsente und lose gekoppelte Informationsversorgung bietet. Dabei können sich die Auswertesysteme für die Sensordaten registrieren (subscribe) und die Sensordaten an die Auswertesysteme ausgespielt werden (publish).

## 1.1 Motivation

In neuartigen Anwendungen aus den Bereichen E-Business, E-Science, Katastrophenmanagement, Medizin und Militär entstehen Kooperationsstrukturen in hochgradig dynamischer Weise und der übliche Weg einer Interaktion von eng gekoppelten Systemen auf Basis des Request-Response-Mechanismus ist nicht mehr adäquat. Es steigt die Nachfrage nach einer Service-orientierten Architektur (SOA), die den Auswertekomponenten die Nutzung der Dienste eines Erfassungssystems erlaubt und je nach Aufgabenstellung, die sich jederzeit ändern kann, die relevanten Daten mit Hilfe eines inhaltsbasierten Mechanismus' liefert. Dies hat nicht nur Auswirkungen auf der Ebene des Kommunikationsmodells, sondern auch auf der Ebene der Datenrepräsentation.

Das Publish/Subscribe Paradigma eignet sich für entkoppelte Systeme und für eine individuelle Sensordatenversorgung. Die ersten Publish/Subscribe Systeme waren topic-based [59]. In diesen Systemen registrierten sich die Subscriber für spezielle Themen. Alle Informationen zu diesen Themen wurden an die Subscriber weitergeleitet. Neuere Publish/Subscribe Systeme unterstützen das inhaltsbasierte Paradigma [59], bei dem sich die Subscriber mit einer Regel, die zur Filterung von Sensordaten dient, registrieren kann. Die Sensordaten im System, die von Publishern veröffentlicht werden, werden mit den registrierten Regeln abgeglichen. Wenn die Sensordaten zu den Regeln passen, werden die Sensordaten zu den korrespondierenden Subscribern weitergeleitet, andernfalls werden sie verworfen. Ein inhaltsbasiertes Publish/Subscribe System ist also flexibler und effizienter als ein topic-based Publish/Subscribe System [59].

Zum erfolgreichen Austausch von Informationen in solch lose gekoppelten Systemen ist es notwendig, dass das Erfassungssystem und die Auswertekomponenten mit dem selben Vokabular und der selben Interpretation der Informationen (Semantik) arbeiten. Alle bisher verfügbaren Spezifikationen definieren hauptsächlich einen syntaktischen Rahmen und die Semantik des Services, die von den Diensten angeboten werden, ist nicht maschinenauswertbar. Diese fehlende Beschreibung der Semantik ist ein Hindernis, um eine individuelle Informationsversorgung und vollständige Automatisierung des Datenaustauschs zu gewährleisten [15]. Durch einen zu entwickelnden und zu beschreibenden Dienst kann der Aufwand zur Interpretation der erhaltenen Informationen minimiert oder die Informationen können sogar vollautomatisch verarbeiten werden. Die Probleme der Datenrepräsentation lassen sich auf verschiedenen Ebenen lösen - von der technischen (XML, XML Schema, etc.) bis zur ontologischen (RDF, RDF-S, OWL, etc.) Ebene. Je nach Aufgabenstellung muss abgeschätzt werden, wie komplex die Sprache des Erfassungssystems und der Auswertekomponenten sein sollen oder können, um allen Anforderungen zu genügen. Für die Mitteilung der individuellen Interessen der Auswertekomponenten werden Interessensprofile eingesetzt.

Besondere Herausforderungen entstehen bei der Echtzeitverarbeitung volatiler Datenströme

[40, 31, 9]. Dabei stellen neben den großen Datenmengen auch die Qualität der Sensordaten (Rohdaten, verdichtete Daten) sowie die Komplexität der Interessensprofile wichtige Einflussfaktoren dar.

## 1.2 Ziel der Arbeit

Die Arbeit hat zum Ziel für das sensorbasierte Informationssystem zur Signalüberwachung von der Firma Plath GmbH einen adäquaten Architekturvorschlag zu machen. Die zentralen Aspekte, die durch ein solchen Architekturvorschlag adressiert werden, sind die Erfassung von Sensordaten in kontinuierlichen Strömen, die kontinuierliche Verwaltung und Verarbeitung von großen Datenmengen nahe der Echtzeit und die Datenausspielung an nachfolgende Auswertekomponenten nahe der Echtzeit. Die Datenausspielung bzw. der Zugriff auf die Daten stellt einen zentralen Aspekt dar und hat großen Einfluss auf die Architektur eines sensorbasierten Systems zur individuellen Informationsversorgung. Damit die Auswertekomponenten Daten zu ihren individuellen Interessen erhalten, soll ein entsprechender Dienst konzeptionell entwickelt werden, über den die Daten an die Auswertekomponenten nahe der Echtzeit ausgespielt werden. Dazu eignet sich besonders das Publish/Subscribe Paradigma, weil die Auswertekomponenten ihr Interesse mit ihren Interessensprofilen registrieren können, die mit den Daten aus dem Erfassungssystem abgeglichen und an sie ausgespielt werden. Für einige Anfragen, die sich auf Rohdaten beziehen, wie beispielsweise *„Liefere mir alle Emissionen“* oder *„Liefere mir alle Frequenzen von 8 bis 10 Uhr“* soll ein geeignetes Datenmodell festgelegt werden. Da ein einfaches topic-based Publish/Subscribe Sytem nicht immer ausreichend ist geeignete Antworten auf komplexeren Anfragen zu liefern, müssen die relevanten Daten mit Hilfe eines inhaltsbasierten Publish/Subscribe-Sytem geliefert werden. Aus dem konkret entwickelten Datenmodell, soll ein allgemeines Datenmodell abstrahiert werden, sodass eine einfache Erweiterbarkeit von Anfragen nach verdichteten Daten garantiert wird. Für den Entwurf ist zu überlegen, welche Standards es für den Datenaustausch und die Datenbeschreibung gibt und welche geeigneten Frameworks man zur Dienstrealisierung einsetzen kann. Diese Arbeit soll sich mit einer prototypischen Realisierung eines Testsystems und einer Evaluation abrunden.

## 1.3 Gliederung der Arbeit

Im folgenden Kapitel werden die Grundlagen strategischer und taktischer Überwachungssysteme erläutert. Damit wird ein Einblick in die generelle Funktionsweise des Sensorsystems der Plath GmbH und die damit verbundenen Anforderungen gegeben. Kapitel 3 gibt einen Überblick über die konzeptionellen und technologischen Grundlagen. Es werden grundlegende Begriffe definiert und erläutert, verschiedene Technologien und Sprachen vorgestellt. Auf dieser Grundlage soll eine konzeptionelle Architektur entworfen werden. Bevor es zu dem Entwurf kommt, wird erst eine Analyse der bestehenden Technologien und Sprachen bzgl. der Anforderungen in Kapitel 4 vorgenommen. In Kapitel 5 wird der Entwurf der zu erstellenden Architektur präsentiert. In Kapitel 6 wird die prototypische Implementierung des Architekturkonzepts beschrieben und geprüft. Abschließend gibt es in Kapitel 7 eine Zusammenfassung und einen Ausblick.

# Kapitel 2

# Grundlagen strategischer und taktischer Überwachungssysteme

Der Themenbereich dieser Arbeit umfasst ein sehr komplexes Aufgabengebiet. Dieses Kapitel bildet die Grundlage für das technische Verständnis in Kapitel 3. Es wird darauf eingegangen wie ein Sensorsystem der Plath GmbH aufgebaut ist. Dazu wird der Prozess der Sensordatenerfassung und -ausspielung im Detail beschrieben. In diesem Zusammenhang wird das eingeschränkte technische Konzept zur Anfragebeantwortung der Plath GmbH erläutert. Folglich wird darauf eingegangen wie die Wissensbasis aufgebaut ist, die für die individuelle Anfrageauswertung benötigt wird und welche Kommunikationsmodelle sich für die Interaktion von verteilten Systemen eignen.

## 2.1 Einleitung

Ein strategisches und taktisches Überwachungsystem überwacht ein bestimmtes Gebiet über einen langen oder mittelfristigen Zeitraum. Für solche Systeme stellt die individuelle Informationsversorgung eine wichtige Anforderung dar. Je nach aktueller Situation und Fragestellung ist es ad hoc erforderlich, Informationsproduzenten zu finden, die schnell bedarfsgerechte (inhaltsorientierte) und relevante (Sensor-)Daten an den Informationskonsumenten liefern. In neuartigen Anwendungsbereichen erfordert der dynamische situationsbezogene Informationsbedarf die effektive Nutzung von einer Vielzahl von Metadaten wie zum Bespiel Interessensprofile [29]. Diese dienen zum einen der Informationsauffindung (dynamische Kooperation) und zum anderen dem inhaltsbasierten Informationszugriff (dynamische Zuordnung von Sensordaten zu den korrespondierenden Konsumenten, Verknüpfen von Datenbeschreibungen und Interessensprofilen bzw. Anfragefiltern). Neben den Datenmengen stellen auch die Qualität der Daten (Rohdaten, verdichtete Daten) sowie die Komplexität der Interessensprofile wichtige Einflussfaktoren dar. Die in diesem Zusammenhang benötigten Kooperationsstrukturen sind hochgradig dynamisch und erfordern einen robusten und flexiblen Mechanismus für eine lose gekoppelte Informationsversorgung. Für diese Interaktion eignet sich das Publish/Subscribe Kommunikationsmodell.

## 2.2 Strategische und taktische Überwachungssysteme

In den unterschiedlichsten Bereichen werden Überwachungssysteme eingesetzt, wo Katastrophen mit hoher Regelmäßigkeit auftreten. So gibt es beispielsweise in Japan ein gut funktio-

nierendes System zur Tsunamifrühwarnung [65]. Mit Hilfe von strategischen und taktischen Überwachungssystemen verfolgt man das Ziel, die Gefahren frühzeitig zu erkennen und die Betroffenen vor einer Gefahr rechtzeitig zu warnen. Die Basis bildet stets die Installation und Nutzung von Sensoren, die die benötigten Messdaten erfassen. Effektive Analysen und Vorhersagen sind nur dann möglich, wenn die Messdaten von vielen, örtlich weit entfernten Sensoren stammen und an einem zentralen Punkt zusammenlaufen. Die Messwerte der Sensoren werden bei solchen Überwachungssystemen nahe der Echtzeit auf Unregelmäßigkeiten und Besonderheiten geprüft. Dazu ist auch ein Vergleich mit historischen Daten nützlich. Nicht jeder auffällige Messwert muss eine Katastrophe ankündigen, aber durch das gleichzeitige Auftreten mehrerer auffälliger verschiedenartiger Messwerte kann zu einem ernstzunehmenden Hinweis führen. Besteht eine Gefahr muss diese Information schnellstmöglich weitergeleitet werden. Dazu wird schon beim Einrichten des Überwachungssystems festgelegt, wer wann welche Information erhält.

## 2.3 Das Sensorsystem zur Signalüberwachung der Plath GmbH

Die Firma Plath GmbH setzt strategische und taktische Überwachungssysteme ein, um Funksignale innerhalb eines definierten Frequenzbereichs zu erfassen, zu klassifizieren, zu orten und auszuwerten. Dafür bieten diese Systeme eine zeitlich lückenlose Überwachung breiter Frequenzbereiche mit zeitnaher Datenausgabe.
Die Leistungsfähigkeit des Systems wird unter anderem durch die Anzahl der Sensoren definiert. Durch den parallelen Einsatz mehrerer Sensoren sind Bandbreite und Aufdeckungswahrscheinlichkeit des Gesamtsystems skalierbar.
So ein Sensorsystem setzt sich aus zwei Teilsystemen zusammen. Das eine Teilsystem (Erfassungssystem) ist für die Datenerfassung bzw. Datenspeicherung und das andere Teilsystem (Auswertesystem) für die Auswertung der erfassten Messdaten zuständig.

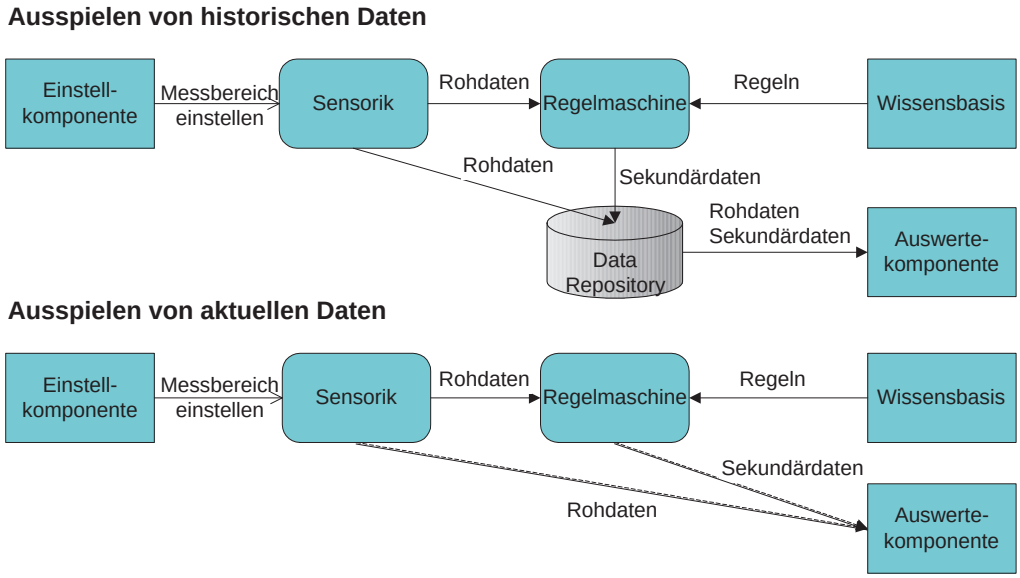


Abbildung 2.1: Modell des Sensorsystems zur Signalüberwachung der Firma Plath GmbH

Innerhalb des Erfassungssystems befinden sich Sensoren, die Messdaten in einem kontinuierlichen Eingangsdatenstrom erfassen. Das Auswertesystem besteht aus Auswertekomponenten,

die unabhängig voneinander Auswertungen durchführen.
Eine Einstellkomponente kann den Messbereich, den die Sensoren erfassen sollen, dynamisch einstellen. Die erfassten Daten werden anschließend verdichtet und abgespeichert. Diese Daten werden als Rohdaten bezeichnet. Die Regelmaschine führt zur gleichen Zeit einen Abgleich von Rohdaten mit den Regeln aus der Wissensbasis durch. Die daraus resultierenden Daten werden Sekundärdaten genannt. Es handelt sich also bei den Sekundärdaten um Daten, die nicht direkt erhoben wurden, sondern aus Rohdaten durch die Verarbeitungsschritte in der Regelmaschine hervorgehen. Sie werden über einen Ausgangsdatenstrom an die Auswertekomponenten direkt ausgespielt oder erst abgespeichert und danach an die Auswertekomponenten ausgespielt. Die Entscheidung hängt davon ab, ob die Auswertekomponenten an aktuellen oder an historischen Daten interessiert sind. Das Speichermedium für die Rohdaten und Sekundärdaten wird als Data Repository bezeichnet. Dieses Modell wird in Abbildung 2.1 illustriert.
Die Architektur des Sensorsystems zur Signalüberwachung der Plath GmbH lässt sich wie in der Abbildung 2.2 skizzieren.

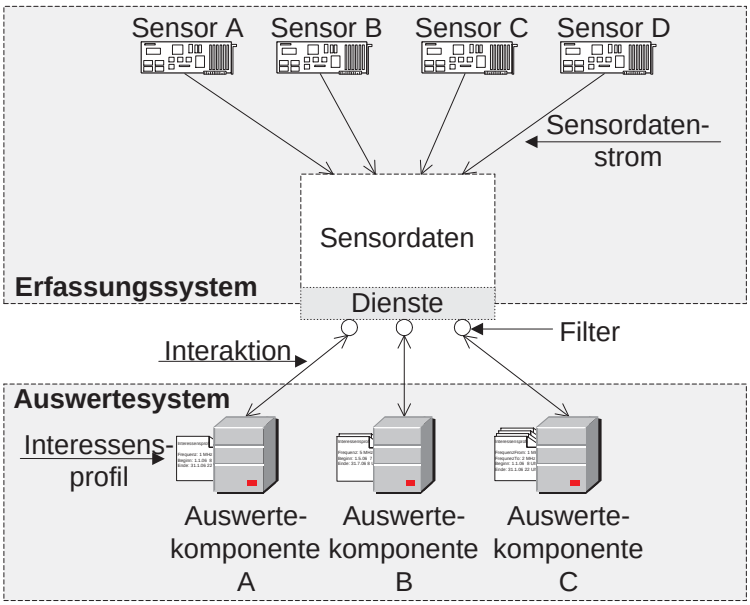


Abbildung 2.2: Sensorsystem zur Signalüberwachung

Im Erfassungssystem sind Rohdaten und Sekundärdaten vorhanden, die über Dienste an interessierte Auswertekomponenten ausgespielt werden. Das Interesse der Auswertekomponenten wird über Interessensprofile ausgedrückt, damit die individuellen gewünschten Daten herausgefiltert und an die Auswertekomponenten ausgespielt werden können.

### 2.3.1 Prozess der Sensordatenerfassung

Das Erfassungssystem ist physikalisch gesehen eine Datenbank, die den Auswertekomponenten Daten zur Verfügung stellt. Der Messbereich, den die Sensoren erfassen sollen, wird bei der Firma Plath GmbH durch so genannte Jobs eingestellt. Mit jedem Job wird der Zeitbereich vorgegeben, in dem die Sensoren hochfrequente Signale erfassen sollen. Der Zeitbereich setzt sich aus einem Anfangsdatum, aus einer Anfangszeit, aus einem Enddatum und aus einer Endzeit zusammen. Enddatum und Endzeit können optional vorgegeben werden. Jedem Job

sind ein oder mehrere Jobpositionen zugeordnet. Mit jeder Jobposition wird zum Beispiel angegeben auf welcher Frequenz oder im welchen Frequenzintervall Signale zu erwarten sind. Bei einer großen Anzahl an Jobs und mit einer begrenzten Anzahl an Sensoren ist es nicht immer möglich, alle Signale in unterschiedlichen Messbereichen zum gleichen Zeitpunkt zu erfassen. Daher wird jeder Job mit einer Priorität versehen, die angibt welcher Messbereich zuerst erfasst werden soll.

### 2.3.2 Ausspielen von Daten

Einer Auswertekomponente werden nur gewünschte Daten ausgespielt, wenn sie ihr Interesse mittels eines Interessensprofils ausdrückt.

**Definition 2.1** *Ein* **Interessensprofil** *stellt das Interessensgebiet einer Auswertekomponente dar. Eine Auswertekomponente kann mehrere Interessensprofile besitzen, die jeweils einen Namen, eine Beschreibung sowie inhaltliche Angaben zu ihrem Interessensgebiet beinhalten.*

**Definition 2.2** *Ein* **Interessensprofilschema** *ist ein Sprachmittel, die bei der Modellierung von Interessen zur Verfügung steht. Ein Interessensprofil ist einem Interessensprofilschema zugeordnet.*

In der Firma Plath GmbH gibt es ein eingeschränktes an die Sensoreinstellkomponente gebundenes technisches Konzept, das mittels eines Interessensprofils die Daten an die Auswertekomponenten ausspielt. Der Job definiert dabei nicht nur den Messbereich, den die Sensoren erfassen sollen, sondern ist zugleich ein Sprachmittel, das zur Modellierung von Interessen der Auswertekomponenten zur Verfügung steht.

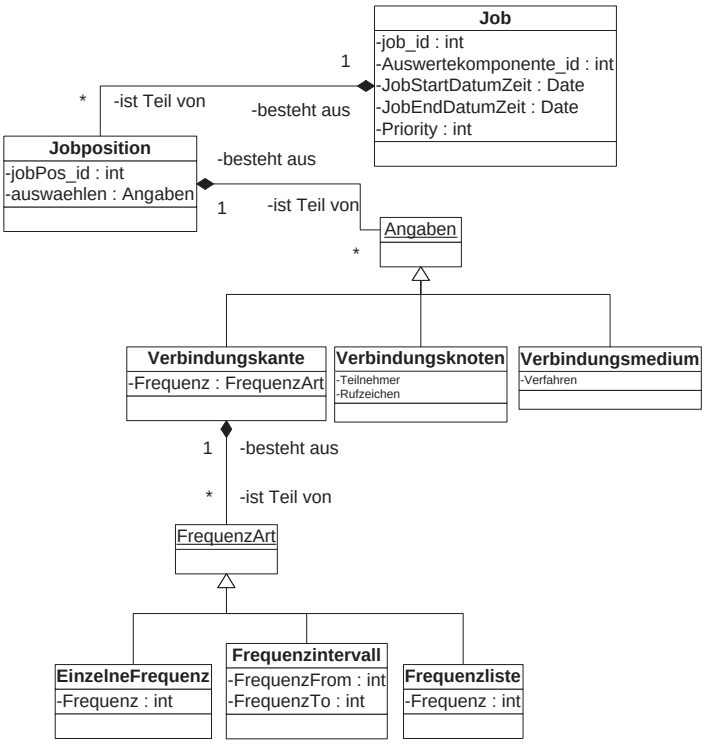


Abbildung 2.3: Ein Interessensprofilschema für Job

**Definition 2.3** *Ein* **Job** *definiert einerseits einen Messbereich, den ein oder mehrere Sensoren erfassen sollen und andererseits ist der Job ein Interessensprofilschema.*

Der Job besitzt eine vordefinierte Struktur. Daher können mit diesem Interessensprofilschema die Auswertekomponenten ihre individuellen Interessen nicht frei ausdrücken, z.B. können sie nur den Zeitbereich angeben. Ein Job besteht aus ein oder mehreren Jobpositionen. Sie definieren zusätzlich zum zugehörigen Job einige Parameter. Die Parameter, die in jeder Jobposition angegeben werden müssen, sind wahlweise eine Frequenz, ein Frequenzintervall, eine Frequenzliste oder eine Kommunikationsstruktur. Die Kommunikationsstruktur besitzt neben der Angabe von Frequenzen auch Angaben über das Verfahren, das die Kommunikationsteilnehmer verwenden, die Position, wo sich die Kommunikationsteilnehmer befinden, weitere zeitliche Einschränkungen und verwendete Rufzeichen zur Identifikation. Es können optional noch weitere Angaben wie eine weitere zeitliche Einschränkung gemacht werden, die nicht weiter erläutert werden, weil sie für die weiteren Konzepte nicht relevant sind.

Abbildung 2.3 gibt einen Überblick über den Aufbau des Interessensprofilschemas Job. Dabei sind jeder Jobposition eine Verbindungskante, ein Verbindungsknoten und ein Verbindungsmedium zugeordnet. Die Verbindungskante enthält die angegebene(n) Frequenz(en). Der Verbindungsknoten beinhaltet bei der Eingabe einer Kommunikationsstruktur die Teilnehmer, das Rufzeichen, das verwendete Verfahren und die Zeitangabe, wann die Teilnehmer kommunizieren. Bei Eingabe einer einzigen oder mehrerer Frequenzen, von Frequenzintervallen oder von Frequenzlisten können Verbindungsknoten, Verbindungsmedium und Zeitintervall optional angegeben werden.
Mit diesem Interessensprofilschema kann eine Auswertekomponente ihr Interessensprofil definieren.

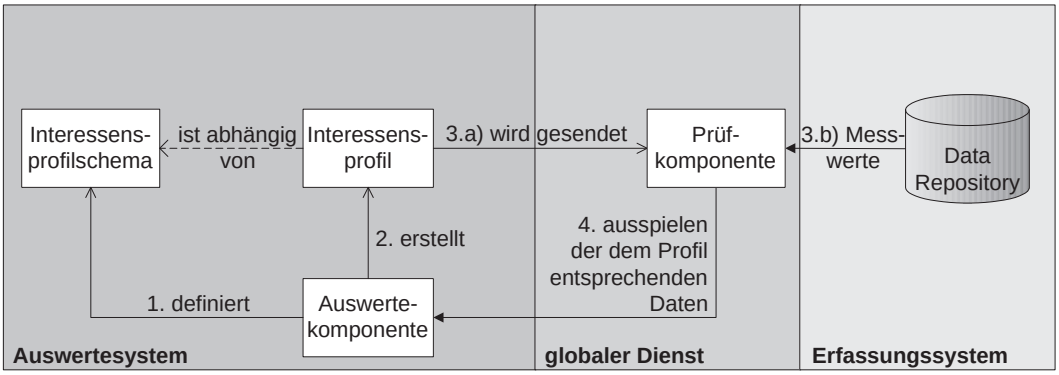


Abbildung 2.4: Ablauf eines Anfrageszenarios

Das Interessensprofil wird einem globalen Dienst weitergegeben, der das Interessensprofil mit den erfassten Daten z.B. aus dem Data Repository abgleicht (siehe Abbildung 2.4). Wenn die Daten zum Interessensprofil passen werden sie an die Auswertekomponente ausgespielt. Wie dabei die beteiligten Komponenten aus Abbildung 2.1 zusammen mit dem Job agieren, wird in dem Modell aus Abbildung 2.5 gezeigt. In der Abbildung wird nicht zwischen Jobs und Jobpositionen unterschieden.
Die Einstellkomponente entspricht in dieser Abbildung gleichzeitig der Auswertekomponente, die Jobs aufgibt. Ein Job hat insgesamt drei Funktionen zu erfüllen. Ein Job definiert das Interessensprofil einer Auswertekomponente, gibt die Bedingungen vor, welche Regeln aus

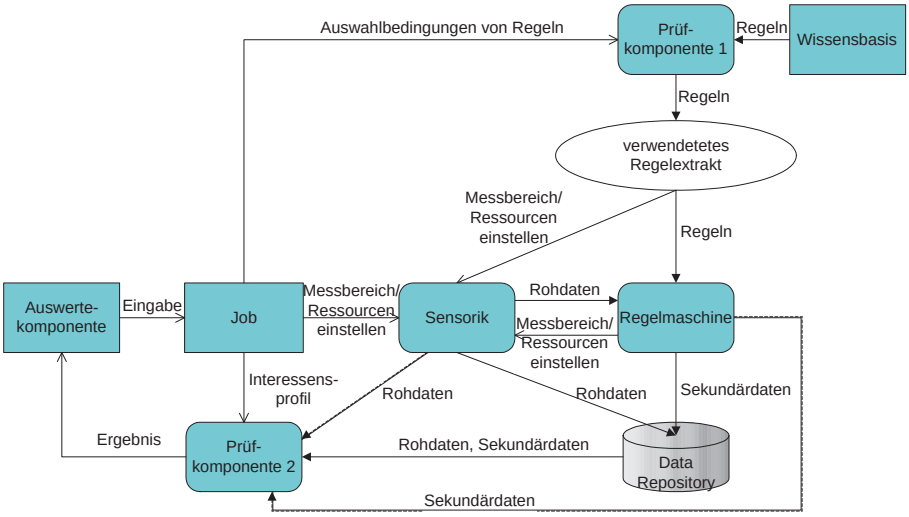


Abbildung 2.5: Modell des Sensorsystems zur Signalüberwachung der Firma Plath GmbH

der Wissensbasis verwendet werden sollen und stellt den Messbereich der Sensorik ein bzw. gibt die Ressourcen vor, mit denen die Sensorik die Signale erfassen soll. Die Regeln, die verwendet werden sollen, werden der Regelmaschine übergeben. Sie kann der Sensorik einen neuen Messbereich vorgeben bzw. gibt vor welche Ressourcen zur Sensordatenerfassung verwendet werden sollen. Daneben führt die Regelmaschine Verarbeitungsschritte mit der von der Sensorik gelieferten Rohdaten, mit den Regeln aus der Wissensbasis und mit den Einstellressourcen der Sensorik durch, und extrahiert daraus die Sekundärdaten. Je nachdem, ob sich die Auswertekomponente für historische oder aktuelle Daten interessiert, liefert die Regelmaschine die Sekundärdaten zusammen mit den Rohdaten aus der Sensorik in das Data Repository. Diese werden anschließend an die Prüfkomponente 2 weitergeleitet oder sie liefert die Sekundärdaten nahe der Echtzeit an die Prüfkomponente 2, die Rohdaten direkt von der Sensorik erhält. Die Prüfkomponente 2 hat die Aufgabe die Daten mit dem Interessensprofil der Auswertekomponente abzugleichen. Wenn die Daten zum Interessensprofil passen, werden diese an die Auswertekomponente ausgespielt, andernfalls werden sie verworfen.
Mit diesem Konzept sind nur einfache und vordefinierte Interessensprofile gegeben, sodass nur Rohdaten und keine Metainformationen als Ergebnis zurückgeliefert werden. Mit jedem Interessensprofil ist auch gleichzeitig die Anfrage *„Liefere mir alle erfassten Daten“* integriert. Damit erhalten die Auswertekomponenten alle erfassten Daten, die zu ihrem Interessensprofil passen. Damit jedoch Anfragen wie

- Gib mir alle Frequenzen aller Emissionen im Zeitintervall von 8 bis 10 Uhr!
- Gib mir alle Emissionen mit dem Verfahren X!
- Gibt es einen neuen Teilnehmer in einer Kommunikationsstruktur?
- Liefere mir alle Emissionen zur HF-Überwachung aus der Region Mexiko!

beantwortet werden können, wird ein neues Konzept mit einem losgelösten Interessensprofil oder mit einer komplexeren Programmierschnittstelle benötigt. Das folgende Kapitel über den Aufbau der Wissensbasis der Plath GmbH soll zunächst erläutern, welches Wissen bereit gestellt wird, um die oben genannten Anfragen mitunter beantworten zu können.

### 2.3.3 Wissensbasis

Im Allgemeinen lässt sich die Wissensbasis folgendermaßen definieren:

**Definition 2.4** *Eine* **Wissensbasis** *enthält eine Menge voneinander unabhängige Wissenseinheiten, deren Struktur durch die Wissenspräsentationsformalismen vorgegeben werden* [46].

Die Wissenspräsentationsformalismen sind vordefinierte Schablonen zur Repräsentation von Wissen. Es handelt sich z.B. um Regeln, die in der Wissensbasis gespeichert werden.

Bei der Firma Plath GmbH beinhaltet die Wissensbasis Regeln über den Aufbau bekannter Kommunikationsstrukturen. Diese Regeln können auf verschiedener Weise gebildet werden. Es kann sich bei den Regeln um z.B. abhängige Event-Condition-Action-Regeln handeln.

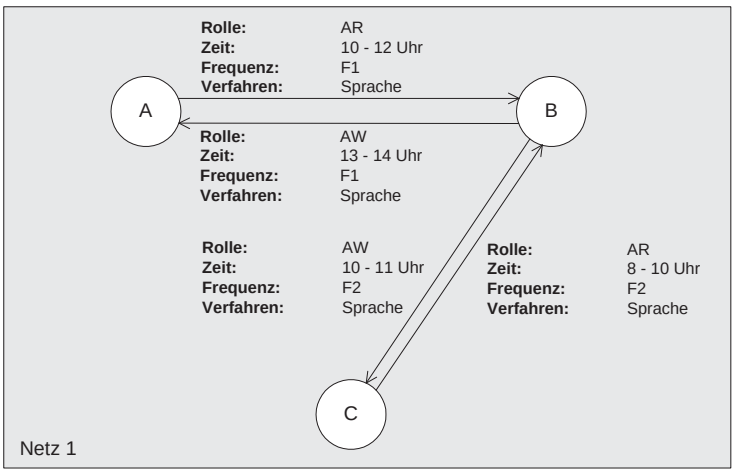


Abbildung 2.6: Kommunikationsstruktur der Teilnehmer A, B und C

Angenommen, es sei folgende Kommunikationsstruktur aus der Abbildung 2.6 bekannt:
Es gibt drei Kommunikationsteilnehmer aus den Ländern A, B und C. Von allen ist bekannt, auf welcher Frequenz, zu welcher Zeit, mit welchem Verfahren (Sprache, Morse, etc.) und in welcher Rolle (Anrufer, Antworter) sie untereinander kommunizieren. Diese Gruppe von Teilnehmern wird als Netz 1 bezeichnet. A ruft jeden Tag von 10:00 bis 12:00 Uhr B an. A besitzt dabei die Rolle des Anrufers (AR). A und B kommunizieren in Sprache auf der Frequenz F1. B antwortet A jeden Tag um 13:00 bis 14:00 Uhr und befindet sich in der Rolle des Antworters (AW). Wenn B jedoch nach C jeden Tag um 8:00 bis 10:00 Uhr auf der Frequenz F2 Signale sendet, trägt B die Rolle des Anrufers. C ist der Antworter und beide verwenden das Verfahren Sprache, um miteinander zu kommunizieren. A und C kommunizieren nicht miteinander oder es ist noch nicht bekannt.

Das Wissen über Kommunikationsstrukturen wird im Prinzip auf zweierlei Weisen erzeugt. Zum einen besitzen einige Auswertekomponenten Wissen über Kommunikationsstrukturen und definieren daraus Regeln, die sie in der Wissensbasis speichern. Zum anderen können solche Regeln aus der Extraktion der Rohdaten gewonnen und in die Wissensbasis gespeichert werden.

## 2.4 Daten- und Anfragemodell

Das Sensorsystem der Plath GmbH erfasst viele Daten im Sekundenbereich, die nach entsprechender Auswertung oder Filterung an Auswertekomponenten weitergeleitet werden sollen. Da es sich bei der Erfassung um eine Abfolge von Daten handelt, deren Ende nicht im Voraus abzusehen ist, wird von einem kontinuierlichen Datenstrom gesprochen. Datenströme können daher nicht als Ganzes, sondern nur fortlaufend verarbeitet werden. In Abbildung 2.7 wird ein beispielhaftes Anfrageszenario vorgestellt, das zeigt, wo Datenströme auftreten und wie mit ihnen ungegangen werden kann.

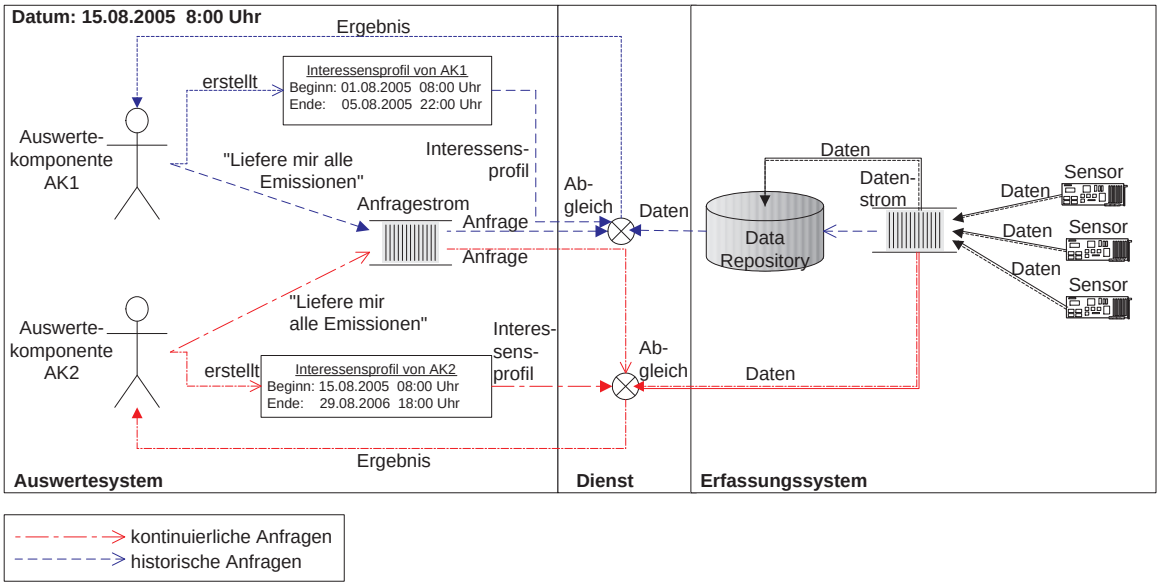


Abbildung 2.7: Anfrageszenario

Die Sensoren im Erfassungssystem erfassen kontinuierlich Signale und bilden durch die große Anzahl an erfassten Signalen im kurzen Zeitbereich einen kontinuierlichen Datenstrom. Dieser wird je nach Aufgabenstellung im Data Repository gespeichert oder direkt weiterverarbeitet. Die abgebildeten Auswertekomponenten AK1 und AK2 sind beide an den erfassten Emissionen interessiert. Beide besitzen unterschiedliche Interessensprofile, die vorgeben, in welchem Zeitbereich sie Daten generell erhalten wollen. Ihre Anfrage *„Liefere mir alle Emissionen“* ist dagegen gleich und wurde von beiden am 15.08.05 um 8:00 Uhr gestellt. Das Interessensprofil von AK1 zeigt das Interesse an historischen und das Interessensprofil von AK2 zeigt das Interesse an aktuellen Emissionen. In der Praxis werden in der Regel so um die Zehntausende von Anfragen gestellt, die in kürzester Zeit bearbeitet werden, daher bilden diese Anfragen einen Anfragestrom. Die Anfragen von AK1 und AK2 sind nur ein Teil, die diesen Anfragestrom bilden.

Die historische Anfrage von AK1 und ihr Interessensprofil werden über den Dienst mit den Daten aus dem Data Repository abgeglichen. AK1 enthält alle Emissionen, die am 01.08.2005 8:00 Uhr bis zum 05.08.2005 22:00 Uhr erfasst wurden. Diese Anfrage kann im Gegensatz zu AK2 als Ganzes (einmalig) verarbeitet werden, daher wird diese Anfrage auch als *Query on demand* bezeichnet.

Die Anfrage von AK2 und ihr Interessensprofil werden mit dem kontinuierlichen Datenstrom abgeglichen. Es wird dabei geprüft, ob die Emissionen zwischen den 15.08.2005 8:00 Uhr und den 15.08.2006 18:00 Uhr erfasst wurden. Wenn ja, werden diese Emissionen an AK2

ausgespielt. Da die eingehenden Datenströme praktisch unbegrenzt sind, sind die daraus berechneten Ergebnisse einer Verarbeitung von Datenströmen oft selbst wiederum Datenströme. Daher wird zwischen eingehenden und ausgehenden Datenströmen unterschieden. D.h. wenn der Abgleich der erfassten Emissionen aus den eingehenden Datenströmen dem Zeitintervall der AK2 entsprechen und die Anzahl an auszuspielenden Emissionen ganz groß ist, wird ein kontinuierlicher Ausgangsdatenstrom, der aus Emissionen im vorgegebenen Zeitintervall besteht, an die AK2 als Ergebnis weitergeleitet. Weil es sich bei AK2 um eine kontinuierliche Anfrage handelt, werden kontinuierlich aktuelle Emissionen ausgespielt, solange wie neue Emissionen erfasst werden und zum vorgegebenen Zeitintervall der AK2 passen.
Die Anforderung für den Dienst sind also die Verarbeitung der kontinuierlichen Anfragen, der historischen Anfragen und der Datenströme nahe der Echtzeit sowie das zeitnahe Ausspielen der Daten.

## 2.5 Kommunikationsmodelle

Es gibt zwei Akteure, die miteinander kommunizieren. Das Erfassungssystem, das Daten veröffentlicht und das Auswertesystem, das Daten empfängt. Die Daten werden entweder automatisch oder erst bei einer Anfrage an die Auswertekomponente ausgespielt.

Das Anfragebeispiel aus Abbildung 2.7 zeigt, dass AK2 an Emissionen im Zeitbereich vom 15.08.2005 8:00 Uhr bis zum 29.08.2006 18:00 Uhr interessiert ist. Zu der einmalig gestellten Anfrage sollten kontinuierlich Emissionen an AK2 ausgespielt werden, solange der 29.08.2006 18:00 Uhr nicht überschritten wurde. D.h. das Erfassungssystem soll in diesem Fall entsprechende Daten an AK2 ausspielen, sobald neue Sensordaten erfasst wurden und zur Anfrage von AK2 passen. Dieses Modell nennt sich Push-Modell.

Bei einer historischen Anfrage, die wie in Abbildung 2.7 von AK1 gestellt wurde, handelt es sich um eine einmalig gestellte Anfrage. D.h., AK1 erhält alle Emissionen die zwischen dem 01.08.2005 8:00 Uhr und 05.08.2005 22:00 Uhr erfasst wurden. Die Anfrage wird nur einmal ausgewertet. Dieses Modell bezeichnet man als Pull-Modell.

Bevor die Daten vom Erfassungssystem an die Auswertekomponente ausgespielt werden können, müssen die Auswertekomponenten sich mit ihrem Interessensprofil registrieren. So können die Daten mit dem Interessensprofil abgeglichen und an die Auswertekomponenten ausgespielt werden. Dieses Modell ist unter dem Namen Publish/Subscribe Kommunikationsmodell bekannt und wird in Kapitel 3.1.3 erläutert.

Das Erfassungssystem kann die Daten für die Auswertekomponenten unterschiedlich darstellen. Zum einen können die Daten einem bestimmten Thema zugeordnet werden und eine Auswertekomponente sucht sich ihre gewünschten Themen aus und abboniert diese. Zum anderen können die Daten dynamisch dargestellt werden. Bei einer Anfrage „Emission“ werden alle Emissionsdaten in einer Liste nach Datum sortiert dargestellt. Für die Beantwortung und Darstellung komplexerer Anfragen wie „Liefere mir alle Emissionen mit dem Verfahren X“ oder „Liefere mir alle Frequenzen zwischen 10 und 11 Uhr, die größer als 1MHz sind“ werden inhaltsbasierte Mechanismen eingesetzt, um die zusammengehörigen Daten zu finden und darzustellen.

## 2.6 Zusammenfassung

Sensorbasierte Systeme werden in verschiedenen Bereichen wie Katastrophenmanagement, Medizin und Militär eingesetzt. Dabei gilt es Daten vom Erfassungssystem zeitnah an das Auswertesystem auszuspielen, damit z.B. Analysen schnell vorgenommen werden können. Das Sensorsystem zur Signalüberwachung der Plath GmbH hat für die Anfrageverarbeitung ein an die Sensorik gebundenes eingeschränktes technisches Konzept entwickelt, mit dem sich keine komplexen Anfragen verarbeiten lassen.
Das zu entwickelnde Konzept soll daher kontinuierliche und historische Anfragen auch komplexer Natur verarbeiten können. Dazu muss ein inhaltsbasierter Mechanismus verwendet werden. Das ausgewählte Kommunikationsmodell sollte daher die Anforderungen Eingangsdatenströme, Ausgangsdatenströme und die unterschiedlichen Anfragetypen zu verwalten und Daten bei kontinuierlichen Anfragen nahe der Echtzeit an die Auswertekomponente auszuspielen. Im Kontext dazu soll erläutert werden, ob das ausgewählte Kommunikationsmodell, all diese Anforderungen erfüllen kann und welche Arten von Anfragen bzw. Interessensprofilen sich damit nahe der Echtzeit verarbeiten lassen.

# Kapitel 3

# Konzeptionelle und technologische Grundlagen

Ziel dieses Kapitel ist es, zum einen die Grundlagen für die Entwicklung eines Architekturkonzepts zur individuellen Sensordatenversorgung darzulegen und zum anderen aufzuzeigen, wie der bisherige Forschungsstand themenverwandter Arbeiten ist.

## 3.1 Grundlegende Begriffe

In diesem Abschnitt werden die wesentlichen Begriffe wie Echtzeit, Datenstrom, das Publish/Subscribe Paradigma, Dienst und die Service-orientierte Architektur definiert, um eine Grundlage für das Verständnis für die Problematik im Bereich des „sensor based computing“ zu bilden. Des Weiteren werden die Kerntechnologien und Sprachen der einzelnen Technologien vorgestellt und abschließend bewertet.

### 3.1.1 Echtzeitsysteme

Rechner wurden schon in den Anfängen für zeitkritische Anwendungen eingesetzt, wobei die Experten erst wenig Interesse an Echtzeitsystemen zeigten. Später nahm das Interesse soweit zu, sodass Echtzeitsysteme eine neue grundlegende technische und wirtschaftliche Bedeutung bekamen. Als Grundlage für diesen Wandel sorgten die Leitungssteigerung in der Hardware und die damit verbundene Kostenreduktion. Heute werden Echtzeitsysteme in vielen unterschiedlichen Bereichen wie Organisation, Verwaltung, Medizin, etc. eingesetzt. Grund dafür ist, das steigende Interesse an automatisierten Abläufen in Echtzeit [49].

**Definition 3.1 Echtzeit** *bedeutet, dass ein beabsichtigtes Resultat innerhalb eines spezifierten Zeitintervalls produziert worden sein muss. Echtzeitbedingungen sind erfüllt, wenn Wertebereich und Zeitbereich eingehalten worden sind* [46].

Je nachdem, ob die Forderung nach Ausführung in einem bestimmten Zeitintervall ist, unterscheidet man zwischen *harten* und *weichen* Echtzeitbedingungen. *Harte Echtzeitbedingungen* liegen dann vor, wenn eine verzögerte Ausführung zu einem Schaden führen kann. Jedoch nicht jede Verletzung der Echtzeit führt zu einer Katastrophe. Hier spricht man von *weichen Echtzeitbedingungen*. In der Praxis werden diese durchaus häufiger verwendet, weil in Anbetracht des Entwicklungstandes verteilter Programmierumgebungen meist nur die Einhaltung weicher Echtzeitbedingungen zu gewährleisten ist. Es gibt zu viele Verzögerungen durch das

Netzwerk, die Workstation, das Betriebssystem, das Laufzeitsystem und die zusätzlich integrierten Dienste [49, 68].
Die Programmiersprachen, die speziell für die Anforderungen von Echtzeitproblemen entwickelt worden sind u. a. PEARL (Process and Experiment Automation Realtime Language), Ada, C++ und die Echtzeiterweiterungen von Java (Java Real Time). Während sich die Entwickler von PEARL und Ada überwiegend auf die Merkmale der Echtzeitprogrammierung konzentrierten, entwickelten sich C++ und Java zu universell eingesetzten Programmiersprachen [53, 68].

### 3.1.2 Datenstrom

**Definition 3.2** *Daten, die kontinuierlich von aktiven Datenquellen (z.B. Sensoren) versendet werden, liegen in der Form eines* **kontinuierlichen Datenstromes** *vor* [16, 36, 51, 56].

Es gibt kontinuierliche Datenströme und nicht kontinuierliche Datenströme. Letztere werden als statische Daten bezeichnet. Statische Daten unterliegen keiner Zeitbedingung für das Eintreffen beim Empfänger, wie z.B. Texte und Bilder. Im Gegensatz dazu unterliegen kontinuierliche Datenströme wie Video-, Audio- oder Sensordatenströme strenger Zeitbedingungen für das Eintreffen beim Empfänger, weil beispielsweise die aufeinander folgenden Bilder bei Videoströmen in einer bestimmten Zeit übertragen werden müssen [36].
Da die eingehenden Datenströme praktisch unbegrenzt sind, sind die daraus berechneten Ergebnisse einer Verarbeitung von Datenströmen oft selbst wiederum Datenströme. Es wird zwischen eingehenden Datenströmen und ausgehenden Datenströmen unterschieden [64].

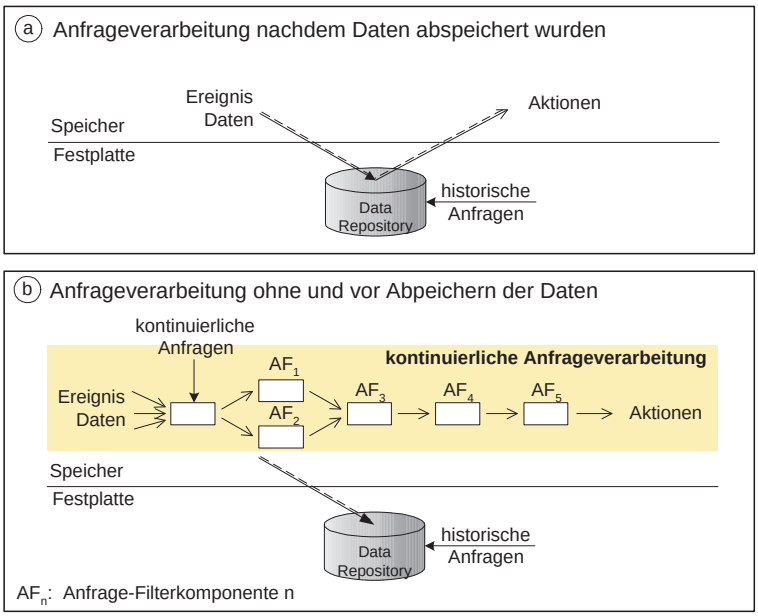


Abbildung 3.1: Anfrageverarbeitung bei Datenströmen

In Abbildung 3.1 werden zwei Methoden für die Verarbeitung von Datenströmen illustriert. Viele Systeme, die mit einem relationalen Datenbankmanagementsystem (RDBMS) ausgestattet sind, speichern die kontinuierlichen Datenströme ab, indexieren sie und verarbeiten danach erst die Anfragen (3.1a). Die Speicherung der Daten erhöht die Verarbeitungszeit. Um diese zu verkürzen und damit eine zeitnahe Anfrageverarbeitung zu gewährleisten, eignet sich

eine Methode, indem die Anfragen gleich nach dem Eintreffen von Datenströmen verarbeitet und je nach Anforderung die Daten anschließend abspeichert werden. AURORA [4] hat einen dynamisches Prozess für eine zeitnahe Anfrageverarbeitung über den kontinuierlichen Eingangsdatenstrom entwickelt (3.1b). Eine Anfrage wird nach dem *„Boxes and Arrows Paradigma“* verarbeitet. AURORA ist ein System bestehend aus Boxen und Pfeilen, welche die Datenströme manipulieren und steuern, um den Ausgabestrom mit Daten gemäß der Anfrage zu erzeugen. Zu jeder Anfrage gehört auch eine QoS-Spezifikation, welche angibt, welche Eigenschaften die Ausgabe auch im Überlastfall noch haben muss. Aurora kann kontinuierliche Anfragen und Ad-Hoc Anfragen verarbeiten. Kontinuierliche Anfragen leiten die Ströme durch die Boxen, in denen sie bis hin zum Ausgang verarbeitet werden. Alle Boxen führen jeweils eine Operation durch. Damit durchläuft eine Anfrage nicht alle Boxen, sondern nur die, die für die Anfrageverarbeitung benötigt werden. Nachdem alle erreichbaren Pfade durchlaufen wurden, werden die Daten aus dem Speicher entfernt, es findet keine Zwischenspeicherung statt. Verbindungspunkte *(connection points)* bieten die Möglichkeit, das Netzwerk dynamisch zu ändern. An sie lassen sich neue Boxen und Pfeile anschließen um Ad hoc Anfragen zu verarbeiten. Sie bieten auch die Möglichkeit, Daten vorübergehend persistent zu speichern, um den später gestellten Anfragen historische Daten zur Verfügung zu stellen.

### 3.1.3 Publish/Subscribe Paradigma

Das Publish/Subscribe Paradigma [48] ist eine konzeptuelle Lösung, um eine Menge von Nutzern in einem bestimmten Zeitintervall mitzuteilen, wenn sich Daten ändern, für die sie sich interessieren, besonders wenn diese Mitteilung in einem langen Zeitabschnitt wiederholt werden soll. Die Nutzer registrieren sich beim Anbieter, dann erhalten die zu ihrer Anfrage die entsprechenden Daten gemäß der angebenden Regel. Die Regeln geben an, wann eine Benachrichtigung erfolgen soll. Es könnte also eine Benachrichtigung erfolgen, „wenn sich ein Wert ändert“ oder „wenn sich ein Wert mindestens x-mal geändert hat“, etc. Das minimiert den Rechenaufwand alle Nutzer zu benachrichtigen und minimiert die Busbandbreite, welche für alle Benachrichtungen benötigt wird.

Das Publish/Subscribe Paradigma ist ein asynchrones Kommunikationsmodell, bei dem sich die Nutzer für bestimmte Daten registrieren (subscribe) oder deregistrieren (unsubscribe) können. Der Anbieter macht den Nutzern die Daten zugänglich (publish). Wenn sich ein Nutzer registriert hat und die vorhandenen Daten beim Anbieter zum angegebenen Interesse eines Nutzers passen, wird der Nutzer benachrichtigt.
Der Prozess zum Verwalten und Ausspielen von Daten von einem Anbieter zu vielen Nutzern wird mit dem Publish/Subscribe Paradigma vereinfacht. Die Begründung liegt darin, weil sich die Anzahl an Nutzern zur Laufzeit einfach ändern kann und sich dieses Paradigma damit als sehr flexibel erweist. Daher wird dieses Paradigma in Bereichen eingesetzt, wo viele unbekannte Nutzer mit Informationen über Dienste versorgt werden wie in [14] und [25].

#### Interessensprofil

In einem Publish/Subscribe System können die Interessensprofile der Nutzer auf zwei Arten definiert werden [2].

**1. Themenbasierte Registrierung:** Ein Dienst stellt ein Datenkatalog bereit, welches die Daten von verschiedenen Anbietern beschreibt. Die Daten werden in diesem Katalog strukturiert als eine Ansammlung von Themen dargestellt. Zu jedem Thema kann es weitere Subthe-

men geben (hierarchische Strukturierung). Nutzer können nun sich für ein bestimmtes Thema oder Subthema registrieren. Der Nutzer kann sich aber auch für ein Thema T registrieren, das ein übergeordnetes Thema A in der Themenhierarchie besitzt. Damit erhält er zum Thema T alle zusammenhängenden Daten.
**Beispiel:** Ein Nutzer registriert sich für *Emission.neue Emission* und teilt so dem Dienst mit, dass er sich für alle Emissionen interessiert, die aktuell sind. Die Registrierung von *Emission.** hingegen würde dem Nutzer als Ergebnis alles zurückliefern, was mit Emission in Zusammenhang steht.

**2. Parameterbasierende Registrierung:** Die Nutzer registrieren ihr Interesse, indem sie neben dem Thema noch Parameterangaben machen können.
**Beispiel:** *Thema = neue Emission* AND *Frequenz = 1 MHz* AND *Qualität > 80*. Ein Nutzer gibt damit an, dass er alle neuen Emissionen, die mit 1MHz Frequenz gesendet wurden und eine Qualität von über 80 Prozent aufweisen, ausgespielt bekommen möchte.

Bei einer themenbasierten oder parametrisierten Registrierung führt die Beantwortung der Anfragen häufig zu unbefriedigenden oder auch unbrauchbaren Ergebnissen. Klassische Suchmaschinen wie Google versuchen über komplizierte Ranking-Funktionen die Gewichtung eines Schlagwortes herzuleiten. Dabei unterliegen sie folgenden Problemen

- Sie beziehen keine Synonyme, Unterbegriffe oder Übersetzungen mit ein. So werden relevante Informationen erst gar nicht angezeigt, weil gar nicht nach ihnen gesucht wird.
- Es werden Mehrdeutigkeiten nicht unterschieden. Wird der Begriff „Java“ in Google eingegeben, erhält man fast nur Seiten bzgl. der Programmiersprache Java. Das es hier auch um die Insel Java handeln kann, wird fast ausgeschlossen, weil die Ranking-Funktion eine niedrige Gewichtung für das Ergebnis „Insel Java“ berechnete.

Diese Probleme können mit einem inhaltsbasierten Mechanismus gelöst werden. Dafür gibt es zwei Ansätze:

1. Man kann die Anfragen verfeinern und präzisieren, also auf syntaktischer Ebene eine Verbesserung herbeiführen. Dafür braucht man die Daten nicht mit sehr vielen semantischen Anteilen zu versehen. Dieser Ansatz ist zwar einfach zu realisieren, aber die Anfragen können dabei sehr komplex werden oder sind sehr schwer zu formulieren.
2. Der zweite Ansatz ist die Verwendung einer Beschreibungslogik. Dabei wird Domänenwissen in Ontologien modelliert. Hier kann ein Nutzer seine Anfrage einfach formulieren und erhält ein korrektes Resultat. Nachteil dieser Methode ist es, dass die Anfragen nicht mehr effizient ausgewertet können. Beispielsweise ist der Aufwand bei einer ontologischen Anfragesprache $O(n^{\exp})$.

### 3.1.4 Dienst

**Definition 3.3** *Ein* **Dienst** *kapselt eine sinnvoll gruppierte Menge von Funktionalitäten und verbirgt sowohl Realisierungsdetails als auch benutzte „tiefer liegende“ Dienste (sog. „Basisdienste“). Ein Dienst wird verteilt und nebenläufig durch kommunizierende Prozesse erbracht* [46]*. Zu den kommunizierenden Prozessen gehören Diensterbringer und Dienstnutzer. Dienste werden aktiv vom Dienstnutzer über eine Service-Schnittstelle des Diensterbringers angefordert, wobei beide Parteien nicht statisch sind, d.h. sich können gegenüber Dritten jeweils auch die Rollen tauschen.*

Dienste unterstützen die Interaktion zwischen heterogenen Rechnerplattformen und sind daher eine architektonische Lösung zwei Interaktionspartner über eine gemeinsame Dienstschnittstelle zu verbinden. Verschiedenste Hersteller können ihre eigenen Produkte unabhängig von anderen Produkten entwickeln und eine Kommunikation über die gemeinsame Dienstschnittstelle ermöglichen.
Ein relativ einfaches Modell ist, alles wie eine Datei zu behandeln. Dieser Ansatz wurde in UNIX eingeführt und wird von vielen Diensten verfolgt. Dabei werden alle Ressourcen als Dateien behandelt. Da Dateien von mehreren Prozessen genutzt werden können, reduziert sich die Kommunikation auf einen einfachen Zugriff auf dieselbe Datei.
Häufig genutzte Modelle für die Kommunikation sind RPC (Remote Procedure Call, entfernter Prozeduraufruf), entfernte Objektaufrufe und nachrichtenorientierte Middleware (Message-Oriented Middleware, MOM).

### Remote Procedure Calls (RPC)

Verteilte Systeme basieren auf einem expliziten Nachrichtenaustausch zwischen den Prozessen. Die Prozeduren *send* und *receive* in den verteilten Systemen nicht transparent. Aus diesem Grund entwickelten *Birell und Nelson* ein Kommunikationsmodell, dass es den Anwendern erlaubt, Prozeduren aufzurufen, die sich auf anderen Systemen befinden. Diese Methode wird als *entfernter Prozeduraufruf* oder *Remote Procedure Call (RPC)* bezeichnet. Die entfernten Prozeduraufrufe sehen wie ein lokaler Prozeduraufruf aus.
Ein konventioneller Prozeduraufruf sieht so aus:

$$\text{Aufruf}(Datei, Eingangsparameter, Ausgangsparameter)$$

Dabei stellt *Datei* den Namen der Datei dar, der *Eingangsparameter* ist ein Array, in das Daten eingelesen werden und der *Ausgangsparameter* gibt die Datenstruktur an, in der die Ausgabewerte gespeichert werden sollen. Bevor der Aufruf vom Hauptprogramm ausgeführt wird, werden die Parameter vom Aufrufer auf dem Stack abgelegt. Nach der Ausführung wird der Rückgabewert in ein Register geschrieben, die Rückkehradresse entfernt und die Steuerung wieder dem Aufrufer übertragen. Der Aufrufer entfernt die Parameter vom Stack und stellt den Originalzustand her.
Ruft ein Dienstnutzer nun keine lokale, sondern eine entfernte Prozedur eines Dienstes auf, benutzt er konventionelle Protokolle, um eine Anfragenachricht über das Netzwerk an den Dienstanbieter zu senden. Die Anfrage identifiziert die aufzurufende Prozedur. Anschließend blockiert der Prozess auf der Dienstnutzerseite und wartet auf eine Antwort. Empfängt das entfernte Programm eine Anfrage ruft er die spezifierte Prozedur auf und sendet dem Dienstnutzer das Ergebnis zurück. Für die Interaktion wird eine zusätzliche Software benötigt. Diese Software nennt man *Kommunikationsstub* (kurz: Stub). Es gibt einen Stub für den Dienstnutzer und einen für den Dienstanbieter. Die Stubs werden passend zum existierenden Programm ausgelegt. Der Rest des Programms benutzt einfach Prozeduraufrufe. Zu dem verwendeten Tool muss der Dienstnutzer für die entfernten Prozeduren die entsprechenden Schnittstellendetails (den global eindeutige Bezeichner (ID)) spezifieren. Hierfür benötigt er die Interface Definition Language (IDL) des Tools. Das Tool liest die IDL-Spezifikationen ruft den IDL-Compiler auf, der die erforderlichen Stubs inklusive einer Header-Datei erzeugt. Die Header-Datei enthält die eindeutige ID, Typdefinitionen, Konstantendefinitionen und Funktionsprototypen. Sie sollte in den Dienstnutzer- und Dienstanbieter-Code aufgenommen werden. Der Dienstnutzer enthält die eigentlichen Prozeduren, die das Programm des Dienstnutzers aufruft. Der Dienstanbieter-Stub enthält die Prozeduren, die vom Laufzeitsystem auf

der Maschine des Dienstanbieters aufgerufen wird, wenn eine Nachricht eintrifft. Anschließend werden die zwei getrennten Programme kompiliert und verknüpft.
Damit ein Dienstnutzer mit einem Dienstanbieter kommunizieren kann, muss der Port des Dienstanbieters bekannt sein. Die Ports werden in einer entsprechenden Tabelle (z.B. DCE-Dämon (Distributes Computing Environment)-Dämon) verwaltet. Der Anbieter muss von seinem Betriebssystem ein Port anfordern und diesen beim DCE-Dämon registrieren. Anschließend registriert sich der Anbieter bei einem Verzeichnisdienst, indem er seine Netzwerkadresse bereitstellt. Nun kann ein Dienstnutzer den lokal bekannten Namen eines bestimmten Dienstes, diesen dem Verzeichnisdienst übergeben. Dann erhält der Dienstnutzer vom DCE-Dämon den Port, womit er nun ein RPC ausführen kann [57].

### Entfernte Objektaufrufe

Ein wichtiger Aspekt eines Objekts ist, dass die Interna von der Außenwelt verborgen bleiben, indem eine wohldefinierte Schnittstelle bereitgestellt wird. Da RPC fast Standard für die Kommunikation in verteilten Systemen wurde und hat man dieses Konzept auf entfernte Objekte übertragen. Verteilte Objekte unterstützen im Allgemeinen systemübergreifende Objektreferenzen, die frei zwischen den Prozessen auf unterschiedlichen Maschinen übertragen werden. Wenn ein Prozess eine Objektreferenz hält, muss er sich erst zu dem Objekt binden, bevor er eine seiner Methoden aufrufen kann. Beim Binden wird ein Proxy im Adressraum des Prozesses platziert, der eine Schnittstelle implementiert, der Methoden enthält, die der Prozess aufrufen kann. Nachdem ein Dienstnutzer zu einem Objekt gebunden wurde, kann er die Methoden des Objekts über den Proxy aufrufen. Es handelt sich um einen entfernten methodenbasierten Aufruf (RMI, Remote Method Invocation) und ist dem RPC sehr ähnlich. Für die Angabe der Schnittstellen des Objekts werden durch eine Schnittstellensprache, bei CORBA (Common Object Request Broker Architecture) mit der *Interface Definition Language* (IDL) oder beim Java RMI *Java*, verwendet, die das Erstellen der Stubs automatisch übernimmt. Beim dynamischen Methodenaufruf wird ein Dienst zur Laufzeit ausgewählt, welche Methode sie für ein entferntes Objekt auf folgender Weise aufruft:

$$\text{Aufruf}(Objekt, Methode, Eingangsparameter, Ausgangsparameter).$$

*Objekt* ist ein verteiltes Objekt, *Methode* gibt den Namen der auszurufenden Methode an, *Eingangsparameter* gibt die Datenstruktur an, die die Eingangsparameter für diese Methode enthält und *Ausgangsparameter* gibt die Datenstruktur an, in der die Ausgabewerte gespeichert werden sollen [57].

### Nachrichtenorientierte Middleware (Message-Oriented Middleware, MOM)

RPCs und RMIs basieren auf einer synchronen Kommunikation, durch die ein Nutzer blockiert wird, bis der Anbieter eine Antwort gesendet hat. Da sich eine synchrone Kommunikation nicht für alle Situationen eignet, können *nachrichtenorientierte Middleware* eingesetzt werden. Der Java Message Service (JMS) [20] ist ein nachrichtenorientierter Middleware-Dienst und bietet die Möglichkeit Nachrichten über eine direkte Verbindung (Point-to-Point) oder nach dem Publish/Subscribe Paradigma auszutauschen. JMS fungiert als Vermittler zwischen Nutzern und Anbietern, indem es vordefinierte Schnittstellen für beide Interaktionspartner anbietet. Mit diesen unterschiedlichen Schnittstellen können Dienstnutzer sich für den Empfang bestimmter Nachrichten registrieren und der Anbieter stellt seine Dienste zur Verfügung.

Der JMS stellt die Nachrichten zu, nimmt sie entgegen und stellt den Transport der Nachrichten zwischen den Systemen sicher. Es handelt sich um eine gerichtete und asynchrone Kommunikation. Ein Vorteil von nachrichtenorientierten Systemen ist die Unterstützung der Entkopplung von Systemen.

### 3.1.5 Service-orientierte Architektur

Ein Dienst ist das Kernelement einer Service-orientierten Architektur.

**Definition 3.4** *Eine* **Service-orientierte Architektur** *(SOA) ist ein Konzept für eine Systemarchitektur, in dem Funktionen in Form von wieder verwendbaren, voneinander unabhängigen und lose gekoppelten Services implementiert werden. Mit einer serviceorientierten Architektur werden i.d.R. die Gestaltungsziele der Geschäftsprozessorientierung, der Wandlungsfähigkeit (Flexibilität), der Wiederverwendbarkeit und der Unterstützung verteilter Softwaresysteme verbunden* [66].

Das Konzept einer SOA wurde in den 90er Jahren als Bestandteil der CORBA Architektur entwickelt und wurde erweitert und optimiert. Die Dienste sind in einer SOA lose gekoppelt. Viele SOAs bestehen aus drei Akteuren: Aus einem Dienstanbieter, einem Dienstnutzer und einem Verzeichnisdienst (siehe Abbildung 3.2). Der Dienstnutzer stellt eine Anfrage an einem Dienst und erhält vom Dienstanbieter eine Antwort. Oft werden für SOAs die weit verbreitete und standardisierte Web Services eingesetzt [1, 5, 15, 17, 30, 66].

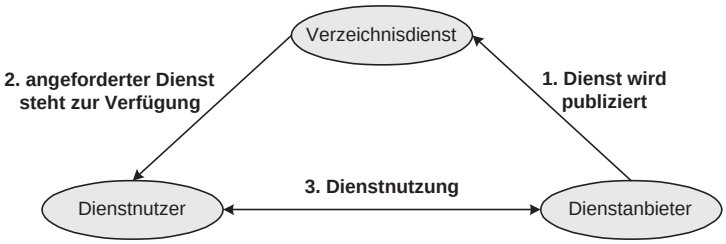


Abbildung 3.2: Beziehungsdreieck einer Service-orientierten Architektur

#### Standardisierte Web Services

Eine Web Service Architektur wird durch ein Kommunikationsprotokoll (SOAP), einem Beschreibungsprotokoll der Dienste (WSDL) und einen Namen- und Verzeichnisserver (UDDI) realisiert.

**SOAP** (ursprünglich Simple Object Access Protocol) [2] wurde 1999 entwickelt. Die erste Version (1.0) dieser Spezifikation basierte auf dem Transportprotokoll HTTP. Ab Version 1.2 können Daten auch über andere Transportprotokolle z.B. über SMTP (Simple Mail Transfer Protocol) übertragen werden. Die ursprüngliche Abkürzung für SOAP ist ab Version 1.2 keine offizielle Abkürzung mehr, weil es nicht nur dem Zugriff auf Objekte dient.
SOAP verwendet Nachrichten um Informationen auszutauschen, die ein so genanntes *Envelope* (deutsch: Briefumschlag) verwenden, welches einen *Header* und einen *Body* beinhaltet. Im Body stehen die Hauptinformation des Senders und im Header eventuell weitere nötige Informationen für eine Zwischenverarbeitung. Alle SOAP-Nachrichten müssen ein Body beinhalten, die Angabe des Headers ist optional.

SOAP definiert die Daten, wie die Daten in XML, in einer strukturierten Art und Weise zwischen zwei Interaktionspartnern ausgetauscht werden. Dazu wird folgendes spezifiert:

- Ein Nachrichtenformat für den einseitigen Informationsfluss (one-way Kommunikation). Diese beschreibt wie die Information in ein XML Dokument gepackt werden kann.

- Einige Vereinbarungen müssen getroffen werden, um SOAP mit dem RPC Interaktionsmuster zu implementieren. Damit wird einerseits definiert wie Nutzer entfernte Prozeduraufrufe ausführen können, wenn sie eine SOAP Nachricht senden. Andererseits wird definiert wie Dienste antworten können, wenn sie eine andere SOAP Nachricht zurück zum Nutzer senden.

- Einige Regeln, nach denen sich eine SOAP-Nachricht einhalten muss. Eine solche Regel könnte aussagen, dass der Grundaufbau einer SOAP-Nachricht erhalten bleiben muss. Alles was im Body steht wird an den Nutzer gesendet. Dabei werden z.B. pdf-Anhänge, die von einer SOAP-Nachricht nicht verstanden werden, in nativer Form an den Empfänger gesendet.

- Eine Beschreibung wie SOAP Nachrichten übermittelt werden.

SOAP ist ein auf XML basiertes Kommunikationsprotokoll, das programmiersprachen- und plattformunabhängig ist und die Semantik beim Austausch von Nachrichten ignoriert. Die Kommunikation ist einseitig gerichtet (one-way Kommunikation). Die Kommunikation zwischen zwei Interaktionspartnern muss in das SOAP Dokument kodiert werden und jedes Kommunikationsmuster, auch das Request/Response-Paradigma, muss in das unterliegende System implementiert werden. D.h. SOAP unterstützt lose gekoppelte Systeme, die einseitig gerichtet Nachrichten asynchron untereinander austauschen. Für einen Nachrichtenaustausch, der zu jeder Anfrage eine Antwort liefern (two-way Kommunikation) oder wo ein entfernter Prozeduraufruf erfolgen soll, benötigt SOAP ein unterliegendes Protokoll oder einen Dienst mit zusätzlichen Eigenschaften. Wenn beispielsweise ein entfernter Prozeduraufruf implementiert wurde, der einige Eingangsparameter verwendet und ein Ergebnis liefert und SOAP verwendet werden soll, müssen zuerst die Eingangsparameter und der Prozeduraufruf in eine SOAP Nachricht kodiert werden. Die Antwort der Prozedur wird ebenfalls kodiert, aber in eine andere SOAP Nachricht. Danach wird ein synchrones Transportprotokoll z.B. HTTP eingesetzt, um beide Nachrichten zu übertragen. Damit wird die erste Nachricht in einer HTTP Anfrage übermittelt, während die Ergebnisnachricht in einer HTTP Antwort zurückgegeben wird. SOAP beschäftigt sich mit einem solchen Austausch und beschreibt wie die Dokumente geschrieben und strukturiert sein müssen, damit eine Interaktion (z.B. RPC) ausgeführt wird und wie die Dokumente auf dem unterliegenden Protokoll (z.B. HTTP) abgebildet werden.
Die Verarbeitung einer SOAP-Nachricht kann bei der Verarbeitung sehr aufwändig sein, weil die Daten verpackt und auf der Gegenseitenseite von dem XML-Parser zerlegt und dann wieder entpackt werden. Aufgrund der schlechten Performance sollte SOAP in Kombination mit methodenbasierten Protokollen (z.B. RPC) für zeitkritische Anwendungen nicht eingesetzt werden.

**WSDL** (Web Services Description Language) [2] definiert einen plattform-, programmiersprachen- und protokollunabhängigen XML-Standard zur Beschreibung von Netzwerkdiensten (Web Services) zum Austausch von Daten. WSDL-Spezifikationen werden meistens durch einen konkreten und einen abstrakten Teil charakterisiert. Der konkrete Teil definiert

- die Protokollbindung. Sie spezifiert z.B. ob es eine RPC-Operation oder eine andere Operation ist. Für eine Verbindung wird meistens SOAP und HTTP verwendet, es können aber auch alternative Kommunikationsprotokolle verwendet werden.

- Ports. Die Ports kombinieren die Protokollbindung mit einer Netzwerkadresse (spezifiert über eine URI (Uniform Resource Identifier)), auf der die Implementation ausgeführt werden kann.

Der abstrakte Teil eines WSDL-Dokuments, enthält

- eine Porttyp-Definition, welche Schnittstellen definieren. Jeder Porttyp beinhaltet zugleich die bereitgestellten Operationen. Jede Operation definiert einen einfachen Austausch von Nachrichten.

- Informationen über die Nachrichten, die der Dienst sendet und empfängt. Jede Nachricht besteht aus einem oder mehreren Datenobjekten vordefinierter Typen.

- Informationen über den Datentypen, die der Dienst verwendet. Diese werden als XML Schema Definition (XSD) dargestellt. XML Schema ist eine Schemabeschreibung, die die Struktur von XML Dokumenten definiert.

**UDDI** (Universal Description, Discovery and Integration) [2, 21] ist eine standardisierte Schnittstelle, die für das Publizieren und Auffinden von Web Services entwickelt wurde. UDDI definiert Datenstrukturen und APIs (Application Programming Interfaces), um Dienstbeschreibungen in einem Verzeichnisdienst zu veröffentlichen und den Nutzern bei einer Anfrage an diesen Verzeichnisdienst veröffentlichte Dienstbeschreibungen zu liefern. Im Kontext von Web Services sind die UDDI APIs bereits in WSDL mit SOAP Bindungen spezifiert, sodass der Verzeichnisdienst selbst als Web Services agieren kann. Der UDDI Verzeichnisdienst beinhaltet elementare UDDI-Datenstrukturen, die in XML-Formate übersetzt werden: businessEntity-Struktur, publisherAssertion-Struktur, businessService-Struktur, bindingTemplate-Struktur und eine tModel-Struktur. Diese Strukturen lassen sich wie folgt beschreiben:

- Eine *businessEntity-Struktur* repräsentiert eine grundlegende Geschäftsinformation, die Kontaktinformationen (Kategorie, Beschreibung, Beziehung zu anderen Objekten, eindeutiger Schlüssel) beinhaltet.

- Eine *publisherAssertion-Struktur* schafft eine Beziehung zwischen zwei businessEntity-Strukturen.

- Eine *businessService-Struktur* fast eine Gruppe von ähnlichen angebotenen Web Services zusammen, d.h. eine businessEntity kann einen oder mehrere businessService-Strukturen referenzieren.

- Eine *bindingTemplate-Struktur* beschreibt, welche technische Informationen nötig sind, um bestimmte Web Services zu nutzen. Also, sie definiert die Adresse, wo der Web Service zusammen mit ausführlichen Informationen zur Verfügung gestellt wird und gibt die Referenzen zu den Dokumenten (technisches Modell) an, welche die Web Service Schnittstelle oder andere Diensteigenschaften beschreiben.

- Ein *tModel* dient zur eindeutigen Identifikation eines Dienstes und legt gleichzeitig fest, wie mit dem entsprechenden Web Service kommuniziert werden kann.

Über UDDI-APIs können Dienstanbieter ihre Dienste anbieten, Dienstnutzer nach Diensten suchen und Einträge vornehmen.

- Die *UDDI Inquiry API* enthält Operationen, um die Einträge von Diensten über die gegebene Datenstruktur zu finden. Diese API wird von einem UDDI Browser verwendet, um Informationen für den Dienstnutzer zur Laufzeit zu finden, um eine dynamische Bindung zu erzeugen.
- Die *UDDI Publisher API* richtet sich an den Dienstanbieter. Damit kann er seine Dienste anbieten, modifizieren und löschen.
- Die *Subscription API* ermöglicht die Überwachung von wechselnden Eintragungen.

Die Interaktion mit einem UDDI Eintrag erfolgt über den Austausch von XML Dokumenten, wenn typischerweise SOAP eingesetzt wird. UDDI spezifiert die möglichen Nachrichtenaustauschformate für die Anfrage und die Antwort. Diese Details sind dem Dienstanbieter und Nutzer transparent. Für den Dienstnutzer ist die wichtigste API die Inquiry API. Für die Suche in dem UDDI Verzeichnisdienst können Nutzer nur ein paar Anfragensprachen (z.B. SQL) verwenden. Grund dafür ist, dass UDDI in dieser Form einfach zu bedienen ist und ein Standard ist. Es wäre auch möglich gewesen, UDDI komplexer zu implementieren, sodass komplexere Anfragesprachen verarbeitet werden könnten. Aber lt. [2] zeigten Verzeichnisdienste mit Spezifikationen außerhalb des Standards, dass z.B. die Anbieter die Bedeutung der Elemente der UDDI Datenstruktur oft falsch interpretiert haben. Damit wurde das Finden nach Informationen immer schwieriger.
UDDI war ursprünglich zum Veröffentlichen von Diensten in einem globalen Netzwerk gedacht, hat sich aber nicht als ein globaler Verzeichnisdienst durchgesetzt. Der Grund liegt darin, dass bei vielen Unternehmen der Schwerpunkt der Verteilung der Dienste im firmeninternen Bereich liegt.

#### Einsatzszenario eines Enterprise Service Bus

Ein *Enterprise Service Bus* (ESB) [26, 38, 50] ist ein standardisiertes Muster für eine Middleware, die Dienste miteinander verbindet. Anders als bei vielen anderen verteilten Anwendungen, z.B. RPC, ermöglich das ESB Muster die Verbindung von Software auf unterschiedlichen Plattformen und in unterschiedlichen Programmiersprachen. Der ESB stellt wie die standardisierten Web Services eine Integrationsinfrastruktur für eine Service-orientierte Architektur zur Verfügung. Das Kernelement der Architektur ist ein zentraler Bus, der von der konkreten Implementierung von Softwarekomponenten abstrahiert. Der Bus wurde für einen hohen Durchsatz entwickelt und garantiert die Weiterleitung von Nachrichten von den Dienstanbietern zu den Dienstnutzern. Viele ESBs unterstützen XML (Extensible Markup Language) als Syntax des Datenaustauschs, wobei auch alternative Datenformate angeboten werden. Die Abbildung 3.3 zeigt einige typische Komponenten, die mit dem ESB verbunden werden können.

**Kommunikation** Die Dienste müssen mit jedem über das Netzwerk kommunizieren können. Dafür wird das in der Industrie dominierende JMS eingesetzt.

**Verbindung** Um Daten von einem Dienst zu nutzen, wird eine einfache Verbindung zu den Diensten benötigt. Diese Verbindung wird mit den öffentlichen Standards wie *J2EE Connector Architecture* (JCA), XML, JMS und Web Services ermöglicht.

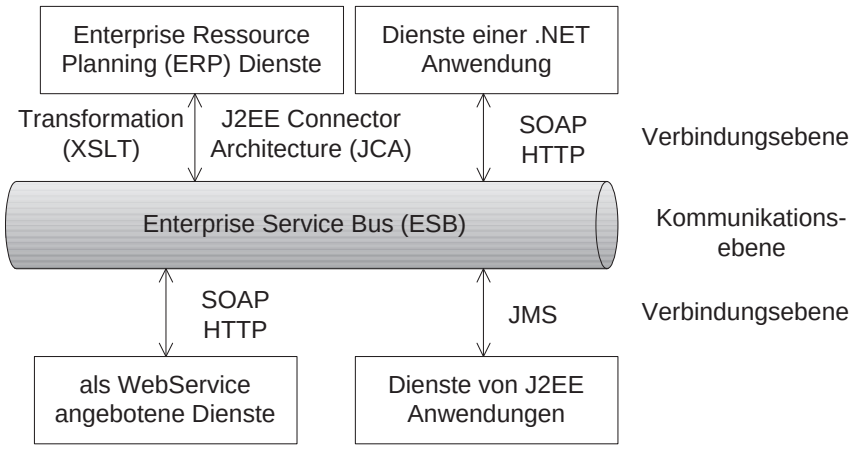


Abbildung 3.3: Enterprise Service Bus

**Transformation** *Enterprise Ressource Planning* (EPS) dient zur Transformation zwischen zwei verschiedenen Geschäftspartnern, die unterschiedliche Formate verwenden. *XSLT* (Extensible Stylesheet Language Transformation) ist eine regelbasierte Transformationssprache, die für die Transformation in das entsprechende Format eingesetzt wird.

**Interoperabilität** Programmiersprachen- und Plattformunabhängigkeit wird z.B. mit dem Einsatz von standardisierten Web Services erreicht.

### 3.1.6 Semantic Web

Web Services geben einen syntaktischen Rahmen vor, wie Dienste beschrieben (WSDL), gefunden (UDDI) und benutzt werden (SOAP). Um eine Automatisierung der Dienstnutzung zu ermöglichen, rückt der Aspekt der Semantik eines Dienstes in den Mittelpunkt. Bislang definieren die Spezifikationen hauptsächlich einen syntaktischen Rahmen. Bei der Beschreibung eines Dienstes mittels WSDL wird die Dienstnutzung nicht mit im WSDL-Dokument übertragen, d. h., dieses Dokument ist nicht maschineninterpretierbar. Um jedoch eine Semantik in einer maschinenlesbaren Form abzulegen und zugreifbar zu machen und ggf. auswerten zu können, wurden die Web Services um semantische Anteile erweitert.
Das Semantic Web erweitert das World Wide Web um maschinenlesbare Daten, welche die Semantik der Inhalte formal festlegt. Wenn die Daten mit Semantik angereichert wurden, können darauf Maschinen operieren und Zusammenhänge selbstständig erkennen. Das setzt voraus, dass die Anbieter von Daten diese semantisch beschreiben. Im nächsten Abschnitt werden Datenrepräsentationssprachen erläutert, die semantische Anteile besitzen.

#### Datenrepräsentation

Zu den gängigsten Datenrepräsentationssprachen gehören XML (Extensible Markup Language), RDF [41] (Ressource Description Framework), RDFS [42] (RDF Schema) und OWL [43] (Web Ontology Language).

**XML** (Extensible Markup Language) definiert syntaktische Regeln für den Aufbau der Datenstrukturen. Für den Datenaustausch ist es von Vorteil, wenn die Struktur von XML entweder mit dem vorgegebenen *Document Type Definition* (DTD) oder mit dem standardisierten *XML*

*Schema* definiert wird. DTD und XML Schema sind Schemabeschreibungen, die die Struktur von XML Dokumenten definieren. XML ist aus SGML (Standard Generalized Markup Language) entstanden. HTML (Hypertext Markup Language) ist ein Element von SGML, daher beinhalten XML-Dokumente Strukturelemente, die aus den HTML-Dokumenten bekannt sind. Ein XML-Dokument besitzt eine hierarchische Baumstruktur:

```
<?xml version="1.0"?>          <!-- Deklaration -->
<emissionen>
  <emission>
    <emission_id> 20 </emission_id>
    <job_id> 1 </job_id>
    <frequenz> 10000 </frequenz>
  </emission>
</emissionen>
```

Ein solches Dokument besteht aus Elementen, die sich aus einem passenden Paar aus Start-Tag (<emissionen>) und End-Tag ((</emissionen>)) zusammensetzten und aus Attributen, die sich innerhalb eines solchen Paares befinden. Diese liefern Zusatzinformationen über die Elemente (eine Art Meta-Information). In dem XML-Dokument handelt es sich um eine Emission, die dem ersten Job (job_id = 1) zugeordnet wurde. Es besteht aus Verarbeitungsanweisungen (<?xml version="1.0"?>) und aus Kommentaren (<!– Deklaration –>).
Um die Daten und Metadaten zwischen Applikationen auszutauschen, können Parser verwendet werden. Ein Parser ist ein Programm, das entscheidet, ob eine Eingabe zur Sprache einer bestimmten Grammatik gehört. Dennoch unterstützt XML nicht die Semantik von Daten, weil z.B. über Benennung der Tags keine eindeutige Bedeutung assoziiert wird. Der Ausdruck „*Die Emission 20 wird dem Job 1 zugeordnet*“ kann in XML folgendermaßen ausgedrückt werden:

```
<Emission id="20">
  <job_id> 1 </job_id>
</Emission>
```

```
<Job id="1">
  <wirdZugeordnet> 20 </wirdZugeordnet>
</Job>
```

Beide Formulierungen besitzen eine andere Syntax, repräsentieren aber die selbe Information. Es gibt für die Benennung der Elemente keine Standardisierung.

**RDF** (Resource Description Framework) ist ein Datenmodell, das auf *Statements* basiert. Ein Statement ist ein Subjekt-Prädikat-Objekt-Tripel bestehend aus Ressource, Eigenschaft und Wert. Werte können Ressourcen oder Literale sein. Literale sind atomare Werte. Eine Ressource kann als Objekt betrachtet werden. Ressorcen können Emissionen, Frequenzen, Teilnehmer und Ortungen sein. Jede Ressource besitzt eine URI *Uniform Resource Identifier*, die eine URL (Unified Resource Locator) oder ein anderer eindeutiger Bezeichner sein kann. Eigenschaften sind eine spezielle Art von Ressourcen, z.B. „empfangen von“, „name“, „dauer“, usw. Eigenschaften werden in RDF auch durch URIs identifiziert. Die Verknüpfung mehrerer Tripel lässt sich als gerichteter Graph mit Knoten- und Kantenbeschriftung verstehen, wobei Ressourcen und Werte eines Tripels Knoten und die Eigenschaften Kanten sind.

Die Besonderheit des RDF-Modells liegt zum einen darin, dass über die Ressourcen auch wiederum Aussagen getroffen werden können. Dadurch lassen sich die Ressourcen selbst mit RDF beschreiben und als Metadatenformat ablegen. Andere RDF-Angaben können diese Vokabulare durch Referenzierung weiterverwenden. Ein prominentes Beispiel dafür ist die Repräsentation von *Dublin Core* in RDF. Dublin Core ist ein Metadaten-Schema zur Beschreibung von Dokumenten. In seiner einfachen Version besteht der Dublin Core aus 15 optionalen Datenfeldern, die mehrfach vorkommen können und 30 Unterfeldern, mit denen sich speziellere Metadaten notieren lassen. In normalen, in HTML verfassten Webseiten können Dublin-Core-Metadaten mit dem allgemeinen Meta-Element im Dokumentkopf angegeben werden. Dazu wird das Präfix „DC“ verwendet.

**Beispiel:** `<meta name="DC.publisher" content="Plath">`

Das RDF Modell ist unabhängig von einer speziellen Darstellungsform. Am meisten verbreitet ist die Repräsentation in XML. Das folgende Beispiel zeigt, das die hierarchische Struktur von XML nicht immer von Vorteil ist. Denn in XML müsse alle zusammenhängenden Elemente ineinander verschachtelt sein im Gegensatz zu RDF.
Eine Webressource repräsentiert den Verlauf von Emissionen im Web. Jedes Element einer Emission besitzt eine entsprechende URI, die den Pfad der Webressource angibt, wenn eine neue Emission eingetroffen ist. Zu jeder Emission wird die jobID und die erfasste Frequenz angegeben. In der XML-Hierachie erhält man eine Tiefe von vier:

```
<?xml version="1.0"?>
<resource>
  <uri>http://plath.de/Emissionen.htm</uri>
  <verlauf>
    <neueEmission>
      <link>http://www.plath.de/Job/Emissionen.htm</link>
      <job_id> 1 </job_id>
      <frequenz> 10000 </frequenz>
    </neueEmission>
  </verlauf>
</resource>
```

In RDF/XML können man zwei separate XML Strukturen mit jeder URI (Uniform Resource Identifier) assoziiert werden. Mit der URI kann eine XML-Struktur mit einer anderen verknüpft werden, ohne dass die zweite Struktur direkt in die erste eingebettet werden muss.

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:pstcn="http://plath.de/postcon/elemente/1.0/"
  xmlns:base="http://www.plath.de/Job/Emissionen.htm"
  <pstcn:Resource rdf:about="Emissionen.htm">
    <pstcn:verlauf>
      <rdf:Seq>
        <rdf:_3 rdf:resource="http://www.plath.de/Emissionen.htm" />
      </rdf:Seq>
```

```
    </pstcn:verlauf>
  </pstcn:Resource>
  <pstcn:neueEmission rdf:about="http://www.plath.de/Emissionen.htm">
    <pstcn:neueEmissionType>Hinzufuegen</pstcn:neueEmissionType>
    <pstcn:job_id>1</pstcn:job_id>
    <pstcn:frequenz>10000</pstcn:frequenz>
  </pstcn:neueEmission>
</rdf:RDF>
```

RDF/XML besitzt zwar eine schwierigere Lesbarkeit für den Menschen, aber in einem automatisierten Prozess ist diese Struktur vorteilhafter, z.B. lassen sich viel einfacher Anfragen formulieren. Wird eine Anfrage nach der Frequenz einer neuen Emission gefragt, muss in XML die gesamte Struktur aller Elemente zur Verfügung gestellt werden, die das Element „Frequenz“ beinhaltet, um sicherzustellen, dass es sich um den korrekten Wert handelt. In RDF/XML hingegen muss nur das Tripel der Spezifikation angegeben werden und nach einem Tripel mit dem Muster einer spezifischen URI gesucht werden, um den spezifischen Wert zu finden. Die Anfrage nach der Frequenz lautet dann „*<http://www.plath.de/Emissionen.htm> pstcn:frequenz*“.
In RDF wird zwischen dem Datenmodell (RDF-Graph) und der Syntax (Serialisierung) unterschieden. Daher muss RDF nicht auf XML basieren, sondern kann seine eigene Terminologie in einer Schema-Sprache definieren. Diese Sprache nennt sich *RDF Schema* (RDFS) [41, 55]. In RDFS kann ein Vokabular definiert werden, das spezifiziert welche Eigenschaften auf welche Objekten zutreffen und welche Werte sie beinhalten können und beschreibt die Beziehungen zwischen den Objekten.
**Beispiel:** *Eine Jobposition ist eine Subklasse von einem Job.*
Dieser Satz bedeutet, dass jede Jobposition einem Job zugeordnet ist. Die Bedeutung wird durch „*ist eine Subklasse von*“ ausgedrückt und wird nicht durch eine Applikation interpretiert, sondern wird von RDF Softwareprozessen übernommen. Durch die Fixierung der Semantik von bestimmten Bestandteilen ermöglicht RDF/RDFS bestimmte Domänen zu modellieren.
Es seien folgende XML Elemente gegeben:

```
<Job> 12 </Job>

<Jobposition id="30">
  <Job_id> 15 </Job_id>
</Jobposition>

<Emission id="20">
  <gehörtZuJob> 1 </gehörtZuJob>
</Emission>
```

Um alle Jobs zu erhalten, kann man mit einem Parser z.B. XPath den Ausdruck *//Job* eingeben und erhält als Ergebnis *<Job> 12 </Job>*. Zwar ist dieses Ergebnis aus XML Sicht richtig, aber semantisch unbefriedigend. Die *Jobposition 30* und die *Emission 20* beinhalten ebenfalls Jobs und sollten daher auch in der Antwort vorkommen. Mit diesem Beispiel wurde gezeigt, dass hier XML und RDF/XML nicht ausreichend sind. Das semantisches Modell der bestimmten Domänen kann in RDFS repräsentiert werden.

Neben RDFS existieren auch viele weitere Ontologien-Beschreibungssprachen wie die *Web Ontology Language* (OWL). Die Ausdrucksmächtigkeit von OWL ist größer als RDF und macht es möglich, die beabsichtigte Bedeutung der modellierenden Elemente und der Datenwerte genau zu definieren, die in den unterschiedlichen Quellen verwendet werden und beseitigt die Unzulänglichkeiten von RDFS [55]. Für OWL gibt es drei Ebenen: OWL Lite, OWL DL und OWL Full.

- *OWL Lite* kann (Un-)Gleichheiten und einfache Bedingungen für eine Hierarchie ausdrücken
- *OWL DL* unterstützt die maximale Ausdrucksmächtigkeit, die komplett berechenbar und entscheidbar ist (alle Bedingungen werden zu einem endlichen Zeitpunkt garantiert berechnet)
- *OWL Full* erlaubt die maximale Ausdrucksmächtigkeit und syntaktische Freiheit, wobei nicht garantiert wird, dass alle Bedingungen berechenbar sind bzw. zu einem endlichen Zeitpunkt berechenbar sind.

Es gilt:

$$\text{RDFS} \subset \text{OWL Lite} \subset \text{OWL DL} \subset \text{OWL FULL}.$$

In OWL wird der Ausdruck *Emission* wie folgt definiert:

$$\text{Emission} \equiv \text{EmissionID} \sqcap \text{Frequenzintervall} \sqcap \text{Zeitintervall}$$

Eine Emission setzt sich auch einer EmissionID, einem Frequenzintervall und einem Zeitintervall zusammen.

Eine Gleichheit kann in RDFS nicht ausgedrückt werden [55]. In OWL Lite wird die Gleichheit „*Morse und Current Write sind jeweils ein Verfahren*“ wie im folgenden Beispiel ausgedrückt:

$$\text{Verfahren}(\text{Morse}, \text{Current Write})$$

In OWL DL kann im Gegensatz zu QWL Lite ausgedrückt werden, dass zwei Klassen z.B. *Ortungsposition* und *Sperrfrequenzen* disjunkt sind.

$$\text{DisjunkteKlassen}(\text{Ortungsposition}, \text{Sperrfrequenzen})$$

OWL Full wird eingesetzt, wenn zwei Ontologien, z.B. eine als Klasse und die andere als Instanz modelliert, von unterschiedlichen Geschäftspartnern in ein vorhandenes Modell integriert werden soll.
**Beispiel:** „747“ ist eine *Instanz* der Klasse von allen Flugzeugtypen, die von Boeing hergestellt wurden oder „747“ ist eine *Subklasse* der Klasse von allen Flugzeugen, die von Boeing hergestellt wurden. Die Anfrage„*Sind Düsenflugzeuge eine Instanz dieser Subklasse?* kann zu einer langen bzw. auch zu keiner Anfrageauswertung führen, obwohl beide Sichten richtig sind. In OWL Full ist es möglich, dass eine Klasse und eine Instanz die gleichen Ausdrücke verwenden.

Die Abbildung 3.4 gibt einen Überblick über die Einordnung der Datenrepräsentationssprachen hinsichtlich der Komplexität der Auswertung und Ausdrucksmächtigkeit.
Die bisherige Entwicklung der Internet-Technologie definierten zu den Spezifikationen hauptsächlich einen syntaktischen Rahmen. Die Begründung liegt in der Historie. Die Informationsdarstellung richtete sich direkt an den menschlichen Nutzer, der die Informationen in seinem

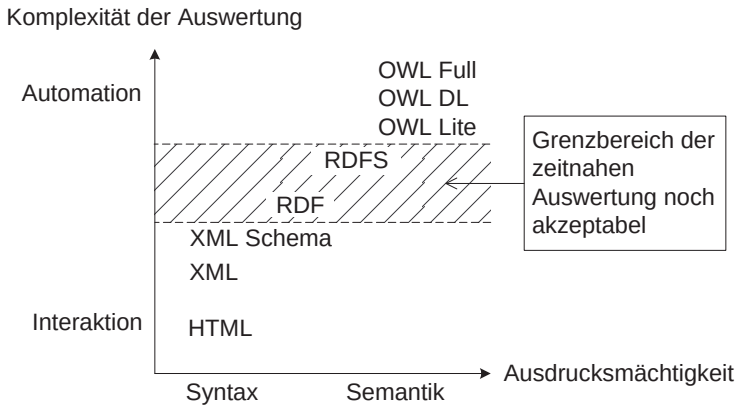


Abbildung 3.4: Darstellung der Komplexität der Anfragesprachen und der Ausdruckmächtigkeit

Kopf verarbeitete. Daher musste HTML lediglich Daten und Angaben über deren Repräsentation übermitteln. Dazu waren syntaktische Angaben völlig ausreichend. Erst mit der Entwicklung von Web Services stieg die Nachfrage nach einer Automatisierung der Dienstnutzung und damit der Bedarf nach Semantik. Der Nachteil, der bei der steigenden Ausdrucksmächtigkeit entsteht, ist die komplexe Anfrageauswertung bzw. die Unentscheidbarkeit für ein Ergebnis. Der schraffierte Bereich zeigt, wo sich die Grenze für eine zeitnahe Anfrageauswertung angenommen wird. Entweder ist eine zeitnahe Auswertung beim Einsatz von RDF nicht mehr möglich oder erst wenn OWL zum Einsatz kommt.

### Anfragesprachen

Für jede Datenrepräsentationssprache wurde eine entsprechende Anfragesprache entwickelt. Dazu gehören die weit verbreitete und bekannte SQL-Anfragesprache, die ausdruckkräftige auf XML-basierende Anfragesprache XQuery [32], die ontologischen Anfragesprachen RDQL [44], RQL [27] und noch nicht standardisierte Anfragesprache OWL-QL.

*SQL* (Structured Query Language) ist die Standardsprache für die Datendefinition und Datenmanipulation in relationalen Datenbanksystemen. SQL ist relational vollständig und ist sogar mächtiger als die Relationenalgebra, aber SQL ist nicht Turing-vollständig, d.h., es gibt berechenbare Funktionen, die nicht in SQL ausdrückbar sind [28].
**Beispiel:** Die Anfrage *„Liefere alle Daten zu den Emissionen, die eine Frequenz größer als 1MHz besitzen“* wird in SQL so ausgedrückt:

```
SELECT *
FROM Emissionen
WHERE Frequenz > 1000000
```

Hinweis: Der Anfragende muss hier wissen, dass alle Werte in der Datenbank in Hz abgespeichert werden. Die Angabe 1 MHz wird also als die Zahl 1000000 abgebildet.
Wird eine Anfrage gestellt, die eine Berechnung einer Rekursion erfordert, kann der SQL3- bzw. SQL:1999-Standard verwendet werden. Damit können zwar Rekursionen berechnet werden; trotz dessen ist SQL nicht Turing-berechenbar, sondern es wird damit lediglich ein Spe-

zialfall abgedeckt.

In den letzten Jahren haben sich XML-Dokumente nicht nur im Datenaustausch, sondern auch als Ablage von Daten so weit etabliert, dass es die Grundlage für eine Vielzahl geschäftskritischer Anwendungen bildet. *XQuery* (XML Query Language) hat sich als mächtiges und zugleich komplexes Werkzeug zur Abfrage von XML-Datenbeständen entwickelt [32]. XQuery benutzt eine an SQL und C angelehnte Syntax und verwendet den Parser XPath sowie XML Schema für sein Datenmodell und seine Funktionsbibliothek. X-Query ist Turing-berechenbar und ist damit eine sehr mächtige Anfragesprache.
**Beispiel:** Eine XQuery Anfrage soll als Ergebnis eine Liste von Emissionen liefern, die in der xml-Datei Emissionen.xml vorhanden sind und eine Frequenz größer als 1000000 Hz haben. Man beachte, dass die Angabe Hz nicht explizit in der Anfrage vorkommt. Der Anfragende muss hier wissen, dass die Frequenzen in Hz abgespeichert werden.

```
for $jobPos in document("Emissionen.xml")//Jobpositionen
where $jobPos/Frequenz > xs:integer(1000000)
return
   <Jobpositionen>{$jobPos}</Jobpositionen>
```

Auch hier gilt der Hinweis, wie bei der SQL-Anfrage. 1MHz wird hier als Integer 1000000 abgebildet.
Der Vorteil, dass XQuery sehr stark an XML gebunden ist, und damit die Anfragen nach XML Daten einfach ist. Diese starke Bindung ist aber auch gleichzeitig ein Nachteil von XQuery. Mit nicht strukturierten Daten, die nicht mit XML präsentiert werden, kann XQuery nämlich so gut wie nichts anfangen. Daher hängt die Verwendung XQuery von der Repräsentationssprache der Daten ab.

Da sich RDF-Aussagen in einer Syntax mitunter auf viele verschiedene Arten ausdrücken lassen, ist es sinnvoll zur Verarbeitung von RDF-Daten einen RDF-Anfragesprache zu verwenden, der auch die Validierung gegen ein RDF-Schema vornehmen kann. Zu den RDF-Anfragesprachen gehören *RDQL* (RDF Data Query Language) und *RQL* (RDF Query Language), wobei die Standardisierung dieser Sprachen noch nicht sehr weit fortgeschritten ist. Beide Anfragesprachen erinnern der Form nach sehr stark an SQL.
**Beispiel:** Eine RDQL Anfrage soll das Ergebnis liefern, wie der Name eines Jobs ist, der die Emission20 beinhaltet.

```
SELECT
  ?Name
WHERE
  (?res, <plath:Job>, ?job),
  (?res, <dc:Name>, ?Name)
AND
 ?job eq Emission20
USING
  rdf for <http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
  dc for <http://purl.org/dc/elements/1.1/>
  plath for <http://www.plath.de/plathdomain/elemente/>
```

Für OWL wurden die Anfragesprache *OWL Query Language* (OWL-QL) entwickelt, welche noch nicht standardisiert ist. OWL-QL ist eine formale Sprache und spezifiert genau die se-

mantischen Beziehungen einer Anfrage und der Wissensbasis, um eine Antwort zu generieren. Anders als bei Datenbankanfragesprachen läuft bei OWL-QL die Anfrage und Beantwortung der Anfrage automatisiert ab. Unter Verwendung von OWL-QL kann es zu einer Anfrage einen Satz von Antworten geben, die sehr groß sein können, sodass die Verarbeitung zeitlich sehr aufwändig sein kann.
**Beispiel:** Die OWL-DL Anfrage *„Welcher Job besitzt die Emission 20“* sieht so aus:

```
(Besitzer ?Job ?e) (type ?e Emission) (hatID ?e 20)
must-bind ?Job don't-bind ?e
```

In der Anfrage wird eine bzw. keine Bindung (must-bind, don't-bind) an Variablen angegeben und haben folgende Bedeutung:

- *Must-bind* Variablen erfordern zwingend eine Bindung an ein Antwortelement. Es muss mindestens eine must-bind Variable in der Anfrage enthalten sein [18].
- Don't-bind Variable erfordert keine Bindung der Variablen. Die Antwort liefert die Ergebnismenge von bekannten Instanzen der Wissensbasis [18]. Es ist bekannt, dass zu jeder Emission ein Job gehört.

In Abbildung 3.5 werden ausgewählte Anfragesprachen bzgl. ihrer Mächtigkeit graphisch dargestellt. Die Mächtigkeit steigt nach außen hin.

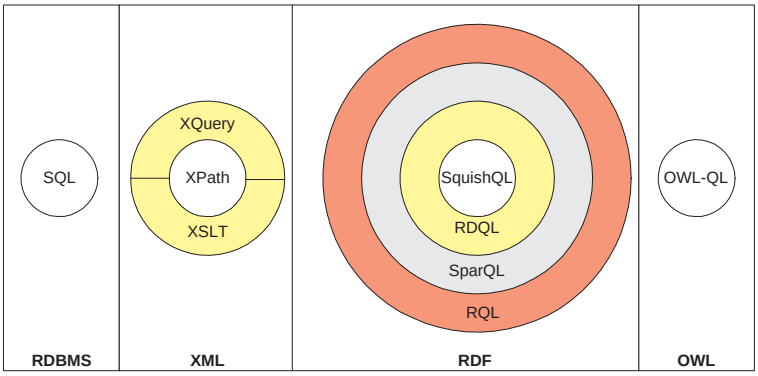


Abbildung 3.5: Mächtigkeiten ausgewählter Anfragesprachen

SQL eignet sich besonders für die Anfrage von relationalen Datenbeständen.
XPath ist ein Parser aus dem u.a. XQuery und XSLT (Extensible Stylesheet Language Transformation) entstanden sind.
Für die Abfrage von RDF-Datenbeständen werden zur Zeit sehr viele Anfragesprachen entwickelt. In der Abbildung sind nur vier Anfragesprachen dargestellt. SquishQL ist eine der ersten Anfragesprachen, die für die Abfrage von RDF Datenbeständen entwickelt wurde. RDQL ist eine mächtigere Anfragesprache als SquishQL. Obwohl die Syntax der beiden Sprachen variert, das Konzept ist sehr ähnlich. SparQL ist eine Erweiterung von RDQL. RQL ist die ausdrucksstärkste Anfragesprache für RDF. Sie erlaubt z.B. die Anfrage nach einer transitiven Hülle, welche die anderen Sprachen nicht abbilden können. Für OWL gibt es noch keine standardisierte Anfragesprachen. OWL-QL wird sehr wahrscheinlich eine standardisierte und sehr ausdrucksmächtige Anfragesprache für Ontologien werden.

### 3.1.7 Modellierungssprachen und Dienste für geographische Entitäten

Neben der Sprachen des Semantic Webs wurden speziell Spartial-temporale Sprachen vom Open Geospatial Consortium[1] (OGC) entwickelt. OGC wurde 1994 gegründet und ist eine gemeinnützige Organisation, die den Dienstnutzern und den Entwicklern diverse Spezifikationen, die auf XML basieren, frei zur Verfügung stellt. Zu den spezifierten Datenrepräsentationssprachen gehören GML (Geography Markup Language), SensorML (Sensor Model Language) und O&M (Observations and Measurements). Zu den spezifierten Anfragesprachen zählen Filter Encoding, SOS (Sensor Observation Service), SCS (Sensor Collection Service) und Active Sensor Collection Service (ASCS). Diese Sprachen werden in diesem Abschnitt kurz erläutert.

#### Geography Markup Language (GML)

Geography Markup Language (GML) ist ein Datenformat zum Austausch raumbezogener Objekte (Features). GML basiert auf XML und ist durch standardisierte Schemata (*.xsd) festgelegt. GML erlaubt die Übermittlung von Objekten mit Attributen, Relationen und Geometrien im Bereich der Geodaten. GML wird vom Open Geospatial Consortium festgelegt. Inzwischen liegt GML in der Version 3.1.1 vor und ist frei verfügbar unter http://www.opengis.net/gml/.

#### Sensor Model Language (SensorML)

Die Sensor Model Language (SensorML) Spezifikation definiert ein XML Schema, das die geometrischen, dynamischen und beobachtbaren Charakteristika eines Sensors als technisches Gerät definiert [7]. Mit SensorML sollen allgemeine Sensorinformationen zur Unterstützung der Datenermittlung bereitgestellt werden, die Verarbeitung und Analyse von Sensormessungen werden unterstützt und die genaue Lokation eines beobachteten Wertes ebenfalls [47].

#### Observations & Measurements (O&M)

Observation & Measurement [11] baut auf GML 3.0 auf und beschreibt die Messdaten von Sensoren. Eine detaillierte Beschreibung und Analyse dieser Datenrepräsentationssprache erfolgt in Abschnitt 4.4.4.

#### Filter Encoding

Die Filter Encoding Implementation Spezifikation [60] definiert ein XML-basiertes Encoding für Filteranfragen und ist Teil der Web Feature Service (WFS) Implementation Spezifikation. Die Filterausdrücke basieren auf der Common Query Language (CQL) von OpenGIS.
Durch die Filterausdrücke, die in Abbildung 3.6 dargestellt sind, kann die Ergebnismenge eingeschränkt werden. Ein Filter kann aus räumlichen `spatialOps`, logischen `logicOps` oder Vergleichs-Operatoren `comparsionOps` bestehen. Filterdefinitionen können über den Parameter `_Id` angegeben werden.

#### Sensor Collection Service (SCS)

Das Sensor Collection Service (SCS) ist eine Spezifikation, die Echtzeitdaten- oder Archivdaten, die beobachtet wurden, bereitstellt. Die vier Funktionen für ein SCS sind `GetCapabilities`, `DescribePlatform`, `DescribeSensor` und `GetObservation`. Die Antworten werden in Form eines O&M-Dokuments kodiert.

[1] registriertes Markenzeichen ist OpenGIS

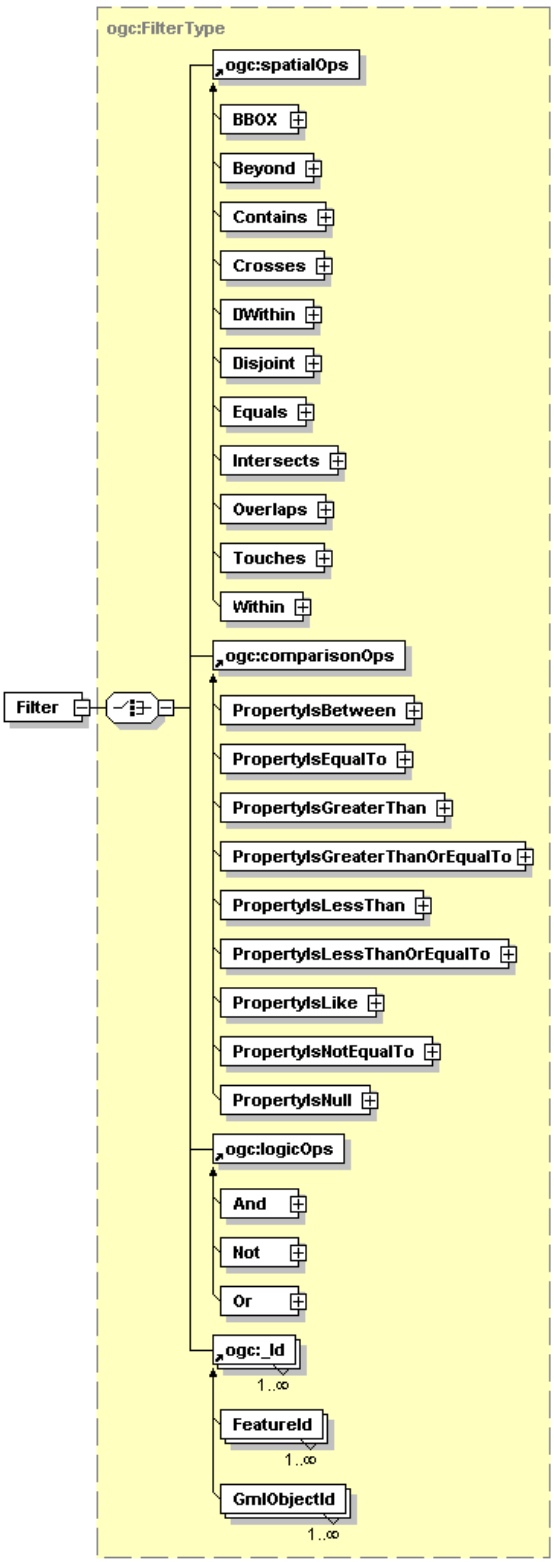


Abbildung 3.6: Schematische Darstellung des Filters von Filter Encoding

Die `GetCapabilities`Funktion ist der allgemeine Mechanismus, bei dem sich ein Dienst selbst beschreibt. Die Antwort einer `GetCapabilities`-Funktion liefert Metadaten über den angebotenen Dienst. Die Funktionen `DescribePlatform` und `DescribeSensor` liefern Metadaten zum Sensor bzw. zur Plattform. Mit der `GetObservation`-Funktion können Anfragen gestellt werden, die Sensorsammlungen mittels einem Pull-Modell zurückliefert.

#### Active Sensor Collection Service (ASCS)

Der Active Sensor Collection Service (ASCS) [61] ist eine Erweiterung des SCS um das Push-Modell. Dafür stehen zusätzlich die Funktionen `SubscribeObservation` und `UnsubscribeObservation` zur Verfügung, damit ein Dienstnutzer seine Anfrage registrieren kann. Wenn die angegebene Bedingung erfüllt ist, erhält der Nutzer solange Daten bis er sein Abonnement beendet. Der ASCS ist kein Standard.

#### Sensor Observation Service (SOS)

Sensor Observation Service ist ein Nachfolger vom SCS [12]. Er stellt im Gegensatz zum SCS eine Möglichkeit dar, um räumlich und zeitlich bezogene Sensordaten standardisiert abzufragen. Dazu können die Ergebnisse laut [62] auch über ein Push-Modell[2] geliefert werden, was bislang nur über ein Pull-Modell realisiert wurde. Deegree definiert zur Zeit die Spezifikation von *SOS deegree*, welche noch nicht veröffentlicht ist.

In nächsten Abschnitt werden alle Technologien und Sprachen mit ihren Vor- und Nachteilen tabellarisch aufgelistet und kurz erläutert.

### 3.1.8 Überblick und Vergleich aller Technologien und Sprachen

| Themengebiet | Technologie/ Sprache | Vorteile | Nachteile |
|---|---|---|---|
| Datenstrom | AURORA | dynamischer Prozess, zeitnahe Anfrageverarbeitung, Verarbeitung von Datenströmen | — |
| Publish/ Subscribe | themenbasiert | einfache Anfrageauswertung | teils schlechte oder unbrauchbare Ergebnisse |
| | inhaltsbasiert (syntaktisch) | einfache Anfrageverarbeitung | Anfragen müssen präzise ausgedrückt werden |
| | inhaltsbasiert (semantisch) | Anfragen einfach ausdrücken | kann zur uneffizienten oder zu keiner Auswertung der Anfragen führen |

Tabelle 3.2: Überblick vorgestellter Technologien (1)

In vielen Bereichen nimmt die Bedeutung von Echtzeitsystemen immer mehr zu. Es sollen möglichst viele Daten in Echtzeit verarbeitet werden, die aus kontinuierlichen Strömen stammen, um die Menschen vor Katastrophen schützen zu können. Die Datenströme sollen dabei

[2] neueste Version vom SOS, dessen Spezifikation noch nicht existiert

| Themengebiet | Technologie/ Sprache | Vorteile | Nachteile |
|---|---|---|---|
| Dienst | RPC | Verwendung konventioneller Protokolle | Nutzer muss Schnittstellendetails spezifieren, enge Kopplung, synchrone Kommunikation |
| | Entfernte Objektaufrufe | wohldefinierte, transparente Schnittstelle | enge Kopplung, synchrone Kommunikation |
| | Nachrichtenorientierte Middleware | lose Kopplung, asynchrone Kommunikation, Publish/Subscribe Paradigma | — |
| Serviceorientierte Architektur | Standardisierte Web Services | SOAP: plattform- und programmiersprachenunabhängig | SOAP: Eignet sich nicht für zeitkritische Anwendungen |
| | | WSDL: Beschreibung der Dienste, plattform- und programmiersprachenunabhängig | — |
| | | UDDI: standardisierte Schnittstelle, um Dienste zu veröffentlichen und zu finden | Veröffentlichung von Diensten im globalen Netzwerk |
| | ESB | Abstraktion konkreter Softwareimplementierungen, zentraler Bus | — |

Tabelle 3.3: Überblick vorgestellter Technologien (2)

effizient verarbeitet werden. Effizient bedeutet nicht nur, dass Datenströme in Echtzeit verarbeitet werden, sondern auch dass Nutzer nur Daten erhalten, für die sie sich auch interessieren. Dafür eignet sich besonders das Publish/Subscribe Paradigma, mit dem sich die Nutzer ihr Interessensprofil definieren und registrieren können. Der dynamische Dienstmechanismus wählt einen Dienst aus, welcher am besten die registrierten Anforderungen des Nutzers erfüllen kann. Dienste können mit Web Services realisiert werden. Die Syntax von Web Services basiert auf XML und findet eine große Akzeptanz und Verbreitung, weil viele Dokumente in XML gespeichert und XML als Datenaustauschformat eingesetzt wird. Der standardisierte Web Service besteht aus SOAP, WSDL und UDDI. Die drei Komponenten sollten jedoch als Möglichkeit einer technischen Lösung betrachtet werden, weil z.B. SOAP in Kombination mit methodenbasierten Protokollen eine schlechte Performance besitzt. Es gibt Web Services, die sich aus integrierten Web Services von Drittanbietern zusammensetzen (Bsp. siehe [69]). Mit dem steigenden Interesse Daten maschinell interpretieren zu können, rückt der Begriff des Semantic Webs in den Vordergrund. Die Anbieter fügen zu ihren anzubietenden Daten noch Semantik hinzu und stellen sie für den Nutzer zur Abfrage zur Verfügung. Für den Datenaustausch und die Abfrage wurden unterschiedliche Sprachen vorgestellt. Es zeigte sich, dass sich XML besonders gut für die Datenpräsentation eignet, weil es in sehr vielen Geschäftsprozessen eingesetzt wird. Vor allem können zeitvariante und geospartiale Messdaten in den auf XML basierenden Sprachen GML, SensorML und O&M repräsentiert werden. Die Anfragesprache XQuery kann komplexe Anfragen der Nutzer bearbeiten, die in XML vorliegen. Wird dabei die

| Sprache | Eigenschaft | Vorteile | Nachteile |
|---|---|---|---|
| XML | syntaktische Regeln, zusammenhängende Elemente sind verschachtelt | Zusatzinformation über die Elemente befinden sich innerhalb eines Tag-Paares | noch nicht praxistauglich [32], keine Unterstützung der Semantik, Finden von Informationen kann zeitaufwändig werden |
| GML | syntaktische Regeln, zusammenhängende Elemente sind verschachtelt | Zusatzinformation über die Elemente befinden sich innerhalb eines Tag-Paares, selbst erstellte Modelle werden unterstützt | Darstellung von Informationen beschränkt sich nur auf den geographischen Bereich |
| SensorML | syntaktische Regeln, zusammenhängende Elemente sind verschachtelt | Zusatzinformation über die Elemente befinden sich innerhalb eines Tag-Paares, selbst erstellte Modelle werden unterstützt, Beschreibung der physikalischen Eigenschaften eines Sensors (Metainformation) | Beschreibung nur von physikalischen Bauteilen |
| O&M | syntaktische Regeln, zusammenhängende Elemente sind verschachtelt | Zusatzinformation über die Elemente befinden sich innerhalb eines Tag-Paares, selbst erstellte Modelle werden unterstützt, Darstellung der Messwerte eines Sensors | — |

Tabelle 3.4: Übersicht vorgestellter Datenrepräsentationssprachen (1)

Anforderung Echtzeit zu erfüllen berücksichtigt, zeigte das XML Beispiel, dass mit wachsender Verschachtelungstiefe, Performance bei der Auswertung verloren geht. Hier erweist sich RDF/XML effizienter, hat aber auch einen höheren Implementierungsaufwand. Ontologien sollten bei der Darstellung der Daten und der Abfrage erst zum Einsatz kommen, wenn die semantischen Beziehungen genau spezifiert werden sollen und keine Echtzeitanforderungen zu erfüllen sind. Denn bei der Anfrageverarbeitung ist mit einem erhöhten Rechenaufwand aufgrund der Komplexität zu rechnen.

### 3.1.9 Zusammenfassung

In dem ersten Abschnitt dieses Kapitels wurden unterschiedliche Technologien vorgestellt und die grundlegenden Begriffe Echtzeit, Datenstrom, Dienst und Service-orientierte Architektur definiert. Das Datenstrom-Management System AURORA wurde vorgestellt, dass Anfragen in einer dynamischer Weise mit einem kontinuierlichen Eingangsdatenstrom abgleichen kann. Das Publish/Subscribe-Paradigma ermöglicht die Registrierung von Interessensprofilen und liefert zu den Anfragen der Auswertekomponenten individuelle Daten. Die Anfragen können dabei auf syntaktischer oder semantischer Ebene ausgewertet werden. Die drei Modelle, die eine Dienstschnittstelle zwischen den Dienstanbietern und -nutzern zur Verfügung stellen, sind

| Sprache | Eigenschaft | Vorteile | Nachteile |
|---|---|---|---|
| RDF | Datenmodell, definieren eines Vokabulars | z.B. Dublin Core bietet standardisiertes Vokabular für RDF, Trennung von Datenmodell und Syntax | — |
| RDFS | Schemasprache für RDF | Definition einer eigenen Terminologie | (Un-)Gleichheiten können nicht ausgedrückt werden |
| OWL | Beschreibungslogik | Automatisierung | Zu komplexe Modelle können nicht garantiert berechnet werden |

Tabelle 3.5: Übersicht vorgestellter Datenrepräsentationssprachen (2)

entfernte Prozeduraufrufe (RPC), entfernte Objektaufrufe und nachrichtenorientierte Middleware. Bei RPC und bei den entfernten Objektaufrufen läuft die Kommunikation synchron und methodenbasiert ab und fordert eine enge Kopplung von Systemen. Die nachrichtenorientierte Middleware hingegen verwendet Nachrichten zum Austausch von Informationen, wobei eine asynchrone Kommunikation stattfindet und die Systeme lose gekoppelt sind. Dienste sind das Kernelement einer Service-orientierten Architektur, für die standardisierte Web Services oder Enterprise Service Bus (ESB) eingesetzt werden können. Der standardisierte Web Service besteht aus einem Kommunikationsprotokoll (SOAP), einer Beschreibungssprache für die angebotenen Dienste (WSDL) und aus einem Verzeichnisdienst (UDDI), der für das Veröffentlichen und Finden von Web Services verwendet wird. Der EBS hingegen besteht aus keiner konkreten Softwareimplementierung. Dem zentralen Bus können wahlfrei einzelne Komponenten hinzugefügt oder entfernt werden. Eine automatische Dienstnutzung kann im Bereich des Semantik Webs erreicht werden. Dafür bieten sich unterschiedliche Datenrepräsentationssprachen mit ihren entsprechenden Anfragesprachen an, die um semantische Anteile erweitert wurden. Mit diesen Sprachen lässt sich eine inhaltsbasierte, aber auch komplexere Anfrageverarbeitung durchführen.

## 3.2 Themenverwandte Arbeiten

In diesem Abschnitt wird der aktuelle wissenschaftliche Stand zu Publish/Subscribe Systemen, Datenstrom-Management Systemen und Echtzeitsystemen vorgestellt. Dazu wird kurz erläutert welche Datenaustauschformate gewählt wurden, um die Anforderungen an die jeweiligen Systeme erfüllen zu können. Abschließend gibt es einen tabellarischen Überblick über alle Systeme, die ähnliche Anforderungen wie die des Plath Sensorsystems erfüllen sowie eine kurze Zusammenfassung.

### 3.2.1 Existierende Publish/Subscribe Systeme

Viele Arbeiten [13, 14, 25, 39, 59, 63] haben ein Publish/Subscribe Paradigma für ihre Systeme eingesetzt. Alle Systeme haben dabei die Anforderung zu erfüllen, Daten von Anbietern (Publisher) in einem Broker Daten zu veröffentlichen. Ein Nutzer (Subscriber) registriert sich in diesem System für eine Kategorie von Daten und der Broker benachrichtigt den Nutzer, wenn die Daten zu seiner Registrierung passen. Im Folgenden werden einige Publish/Subscribe Systeme vorgestellt.

| Sprache | Eigenschaft | Vorteile | Nachteile |
|---|---|---|---|
| SQL | Anfragesprache auf relationale Datenbanken | relational vollständig | nicht Turing-vollständig |
| XQuery | Abfragesprache der XML-Datenbestände | Turing-vollständig | Abfragen nur von XML-Dokumenten |
| Filter Encoding | Abfrage der GML-Datenbestände | Filtern der Anfragen | Nur reine Filteranfragen möglich |
| SOS | Abfrage von SensorML- O&M-Datenbeständen | Mehr als reine Filteranfragen möglich, Push- oder Pull-Modell | noch veröffentlichte Spezifikation, erlaubt inhaltlich falsche Anfragen |
| SCS | Abfrage von SensorML- O&M-Datenbeständen | Mehr als reine Filteranfragen möglich | nur Pull-Modell |
| RDQL, RQL | Abfrage der RDF-Datenbestände | Liefern von inhaltlich richtigen Abfragen | ggf. lange Berechnungszeit, Abfragen von Netzstrukturen nicht möglich |
| OWL-QL | Abfrage der OWL-Datenbestände | automatisierte Anfrageverarbeitung | noch nicht standardisiert, komplexe Anfrageverarbeitung, Abfrage nur nach Rollen oder Instanzen |

Tabelle 3.6: Übersicht vorgestellter Anfragesprachen

#### Themenbasierte Publish/Subscribe Systeme

In den ersten Generationen wurden themenbezogene (topic-based) Publish/Subscribe Systeme [13] eingesetzt, um Anfragen der Nutzer zu beantworten. Nutzer registrieren sich für eine spezifische Kategorie von Daten. Im Broker steht eine Liste mit Themeneinträgen zur Verfügung und der Nutzer registriert sich für alle Themen, zu dem er Daten ausgespielt haben möchte. Wenn ein Anbieter ein neues Thema veröffentlichen möchte, teilt er allen Brokern das neue Thema mit. Der Name des Themas wird so gewählt, dass er seinen Inhalts möglichst kurz beschreibt.
**Beispiel:** Es wird das Thema „Emission“ veröffentlicht. Ein Nutzer registriert sich für dieses Thema und erhält alle Daten, die diesem Thema zugeordnet wurden.
Der Nutzer muss sich jedoch nicht bedingt für alle Arten von Emissionen interessieren, sondern interessiert sich z.B. nur für Emissionen, die mit dem Verfahren X entstanden. Weil die Emissionen und Verfahren in zwei unterschiedlichen Tabellen[3] gespeichert sind, kann keine einfache Filteranfrage[4] vorgenommen werden. Es wäre sehr umständlich und uneffektiv, wenn der Nutzer sich für das Thema „Emission“ und das Thema „Verfahren X“ registrie-

[3]Zur Erläuterung des Problems bei themenbasierte Systemen wird hier angenommen, dass Emissionen und das Verfahren in separaten Tabellen gespeichert sind und keine Beziehung über eine zusätzliche Tabelle besitzen. In der Praxis gibt es jedoch eine weitere Tabelle, die eine Beziehung zwischen Emissionen und Verfahren abbildet. Dieser Hinweis ist wichtig, weil es sonst zu Widersprüchen zu den Beispielen in Kapitel 4 kommen könnte

[4]Alle Emissionen werden nach dem Verfahren X gefiltert.

ren müsste. Anschließend müsse der Nutzer auch noch selbst nachsehen, wie die erhaltenden Daten zusammenhängen. Solche Auswertungen können automatisch mit inhaltsbasierten Publish/Subscribe Systemen vorgenommen werden.

### Inhaltsbasierte Publish/Subscribe Systeme

Das in den letzten Jahren entstandene inhaltsbasierte Publish/Subscribe Paradigma bietet die geforderte Flexibilität komplexe Anfragen zu verarbeiten und erlaubt den Nutzern seine Interessen über eine beliebige Anfrage über den Inhalt von Daten auszudrücken. Dadurch entfällt die Gruppierung der Daten und es können unterschiedliche, voneinander unabhängige Nutzer individuelle Anfragen stellen wie „Liefere mir alle Emissionen mit dem Verfahren X“ oder „Liefere mir alle Frequenzen zwischen 10 und 11 Uhr, die größer als 1MHz sind“. Die Umsetzung eines solchen Paradigmas führt zu hohem Design- und Implementierungsaufwand, da das Finden der zugehörigen Ergebnisse von komplexen Anfragen schwieriger wird als bei dem traditionellem topic-based Systemen. Dennoch gibt es eine Reihe von umgesetzten Publish/Subscribe Systemen, die als nächstes vorgestellt werden.

*ONYX* [14], das XML Publish/Subscribe System aus [59] und das Publish/Subscribe System für den mobilen Einsatz aus [25] sind inhaltsbasierte Publish/Subscribe Systeme und besitzen folgende Architektur:

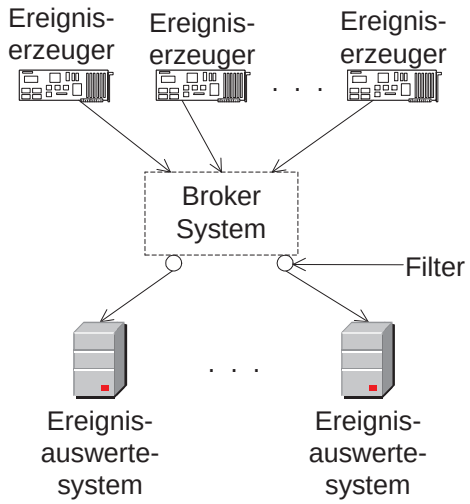


Abbildung 3.7: Ein Publish/Subscribe System [25]

Der Aufbau des Publish/Subscribe Systems ist dem Sensorsystem der Plath GmbH aus der Abbildung 2.2 sehr ähnlich. Das Broker System speichert alle Registrierungen und veröffentlicht alle Ereignisse. Daneben gleicht das Broker System alle Ereignisse mit den Registrierungen ab. Passt ein Ereignis zu einer Registration, wird dieses Ereignis an das Ereignisauswertesystem ausgespielt, das wiederum Analysen mit den gelieferten Ereignissen vornehmen kann. Dieser Filtervorgang wird in der Abbildung durch die kleinen Kreise widergespiegelt.
Das Brokersystem in ONYX erlaubt eine n:n Datenweiterleitung von den Anbietern zu den Nutzern. Daher wird das Brokersystem in $n$ Datenbroker (kurz: Broker) unterteilt. Fast alle Broker verarbeiten Daten und Anfragen, während ein anderer Broker einen Registrierungsdienst anbietet. Dieser Überblick ist in Abbildung 3.8 illustriert.

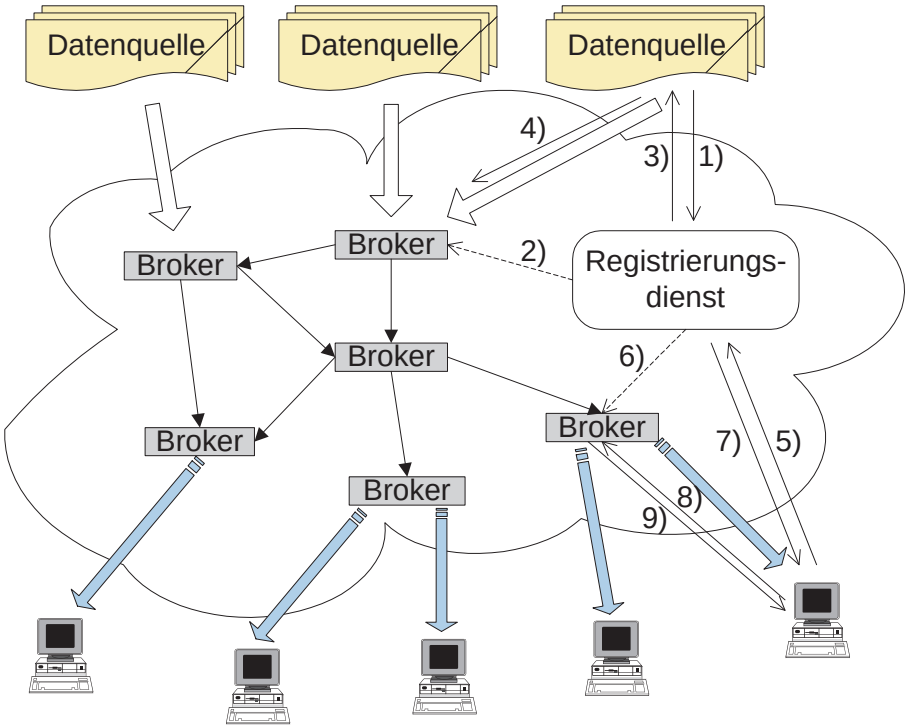


Abbildung 3.8: Architektur von ONYX [14]

**Datenerfassung**
1. Eine Datenquelle wird registriert, indem die Datenquelle den Registrierungsdienst kontaktiert.
2. Der Registrierungsdienst vergibt einer Datenquelle eine ID und sucht einen geeigneten Broker für die Datenquelle aus. Die Wahl eines Brokers basiert dabei auf seinem topologischen Abstand zur Datenquelle, der verfügbaren Bandbreite und dem erwarteten Datenvolumen der Quelle.
3. Anschließend wird der Datenquelle die ID und die Adresse des Brokers gesendet.
4. Nach der Registrierung kann die Datenquelle nun die erfassten Daten mit Hinzufügen ihrer ID veröffentlichen.

**Datenausspielung**
5. Ein Nutzer sendet sein Interessensprofil und seine Adresse an den Registrierungsdienst.
6. Dieser vergibt dem Interessensprofil des Nutzers eine ID sucht den passenden Broker basierend auf seiner Lokation und/oder dem Inhalt des Interessensprofil. Nach der Registrierung sendet der Dienst das Interessensprofil und die entsprechenden Informationen an den ausgewählten Broker.
7. Der Nutzer erhält von dem Broker die Profil ID und die Broker Adresse. Die Daten werden nun an alle Nutzer mit dem gleichen Profil vom entsprechenden Broker ausgespielt.
8. Der Nutzer führt ein Update seines Interessensprofil direkt beim Broker durch.
9. Der Broker überprüft, ob Daten zum Interessensprofil passen oder nicht und spielt sie entsprechend aus. Deswegen wird ein Update des Interessensprofils nicht über den Registrierungsdienst vorgenommen, sondern direkt über den Broker.

Mit ONYX können historische und kontinuierliche Anfragen unterschiedlicher Komplexität verarbeitet werden. Testergebnisse zeigen, dass bei steigender Komplexität der Anfragen die Performance und die Verarbeitungszeit darunter leiden. Komprimiert man jedoch das XML Format, so lässt sich dieses Ergebnis als brauchbar bezeichnen [14].
Bessere Ergebnisse erzielt das **O**ntology-based **P**ublish/**S**ubscribe (OPS) System [63] in Hinblick auf die Verarbeitungszeit von komplexen Anfragen und einer großen Anzahl von Anfragen. Das mit Semantic Web Technologien entwickeltes und in Java umgesetztes Publish/Subscribe System besteht aus einem Konzeptmodell und einem Ereignismodell. Das Konzeptmodell spezifiert die Beziehungen zwischen den Ereignissen und die zugehörigen Bedingungen. Es wird mit DAML+OIL definiert. Das Ereignismodell organisiert die Daten nach dem Konzeptmodell und wird mit RDF definiert. Jede Registration wird als Graph repräsentiert. Der Graph kann mit den RDF Anfragesprachen SquishQL, RDQL und RQL ausgedrückt werden.

### 3.2.2 Datenstrom-Management Systeme

Viele haben sich mit der Problematik der Datenstromauswertung in Sensornetzwerken beschäftigt. [6] und [10] haben Datenstrom-Management Systeme entwickelt, die speziell für historische Daten entwickelt worden sind. Beide verwenden das Push-Modell, um historische Daten an die Nutzer auszuspielen. Da Nutzer nicht nur an historischen Daten, sondern immer mehr an aktuellen Daten interessiert sind, benötigt man andere Datenstrom-Management Systeme. *Fjords* (**F**ramework in **J**ava for **O**perators on **R**emote **D**ata **S**tream) [34] ist ein System, das ebenfalls speziell für die Verarbeitung von Sensordatenströmen entwickelt wurde. Fjords kombinieren das Push- und das Pull-Modell. Dies entspricht der Kombination von Datenströmen und statischen Daten. Das konventionelle Pull-Modell wurde speziell für statische Daten entwickelt und eignet sich nicht für Daten aus kontinuierlichen Strömen. Fjords kombinieren daher beide Modelle, indem Queues eingesetzt werden, die als Push- oder Pull-Modell agieren können. Damit können gemischte Anfragetypen verarbeitet werden, die auch unterschiedlicher Komplexität sein können.
Wenn ein Nutzer eine Anfrage „Create Fjord: Join[historische Emission,neue Emission] with filter A“ stellt, wird diese einem Kontroller übergeben. Der Kontroller instanziert dann die entsprechenden Operatoren (hier: Join), die für die Anfrageverarbeitung benötigt werden. Die Operatoren beinhalten die standardisierten Datenbankmodule wie join, select, etc. Die historischen Emissionen aus der Datenbank werden einer Queue übergeben und generieren historische Emissions-Tupel. Diese werden über ein Pull-Modell an eine weitere Queue übergeben, die die nächste Operation durchführen soll. Die aktuellen Emissionen aus dem kontinuierlichen Sensordatenstrom liefern aktuelle Emissions-Tupel und werden über einen Push-Modell einer Queue übergeben die die nächste Operation durchführen soll. Da historische und aktuelle Emissionen vereinigt werden sollen, befinden sich beide Tupel in der selben Queue. Dort werden sie vereinigt und anschließend werden diese Daten einer anderen Queue übergeben. Diese Queue gleicht die Daten mit dem Filter A ab und spielt die relevanten Daten an den Nutzer aus.

### 3.2.3 Existierende Echtzeitsysteme

*ROADNet* [45] kann Sensordaten in kontinuierlichen Strömen in Echtzeit aufrufen und ausfindig machen. Die ROADNet Architektur ist in 3.9 abgebildet.
Sensoren erfassen Sensordaten und werden einem *Object Ring Buffer* (ORB) zugeordnet (1). Der ORB ist dafür zuständig, die Sensordaten zu verarbeiten und zu verwalten (2). Die bei der

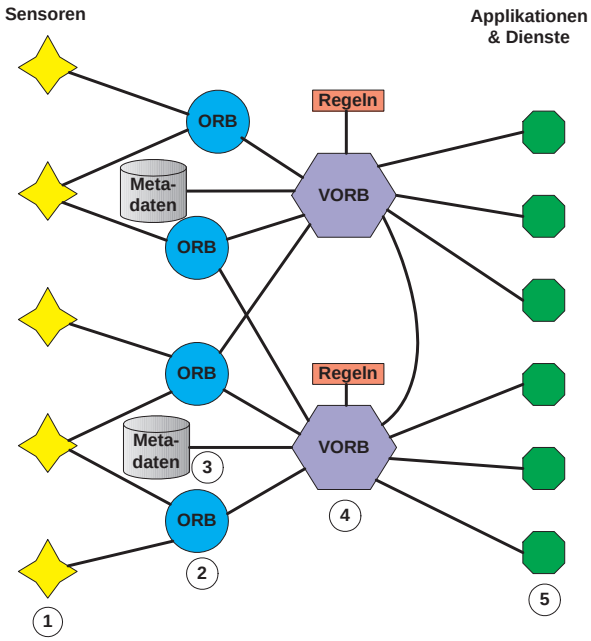


Abbildung 3.9: ROADNet Architektur [45]

Verarbeitung resultierenden Metadaten werden in einem so genannten Echtzeit Datenbankschema abgespeichert (3). Im *Virtual Object Ring Buffer* (VORB) werden alle Echtzeitdaten von den ORB zusammengefügt, indem zu jedem Datenstrom der logische Name gespeichert wird (4). Anschließend werden sie Attribut-basiert visualisiert und den Nutzern über angebotene Schnittstellen zugänglich gemacht (5). Der Nutzer sucht sich eine Schnittstelle aus und stellt damit eine technische Anfrage an einem VORB. Optional kann der Nutzer jedem VORB Regeln vorgeben. Dieser spielt die resultierenden Datenströmen an die entsprechenden Nutzer aus und berücksichtigt dabei die vorhandenen Regeln.
Mit diesem System lassen sich in Echtzeit kontinuierliche Sensordaten verarbeiten und an eine große Anzahl an Nutzern ausspielen. Vorteil dieser Architektur ist es, dass schon gleich nach der Erfassung von Sensordatenströmen diese abstrahiert werden, indem systematische, beschreibende und strukturierte Metadaten-Eigenschaften in einem Katalog definiert und in der Datenbank abgespeichert werden. So kann jeder Nutzer nach seinem Interesse Daten in diesem Katalog finden. In Hinblick auf zu entwickelndes Architekturkonzept für das Sensorsystem der Plath GmbH geht es nicht nur darum passende Daten zeitnah auszuspielen, sondern den Nutzern entsprechend darzustellen und mit dem Einsatz von Interessensprofilen eine individuelle Sensordatenversorgung zeitnah zu ermöglichen.

WISE (**W**eb-based **I**ntelligent **S**ensor **E**xplorer) [39] ist ein Web-Service Framework, mit dem kontinuierliche Datenströme in Echtzeit veröffentlicht, gefunden und analysiert werden. Die Architektur von WISE entspricht der ONYX-Architektur aus der Abbildung 3.8. Für die Realisierung von WISE wurden standardisierte Web Services eingesetzt, weil die Charakteristiken wie Interoperabilität, Entkopplung und Verfügbarkeit von der WISE Umgebung gefordert werden. Ein Anbieter veröffentlicht seine Daten, indem er die relevanten Metadaten für seine Sensoren vorgibt. Ein Beschreibungsdienst (WSDL) generiert aus den zur Verfügung gestellten Daten oder auch Diensten automatisch ein WSDL-Dokument. Dieses WSDL-Dokument

wird anschließend in einem Verzeichnisdienst (UDDI) veröffentlicht. Ein Nutzer kann nun über seinen lokalen Browser nach relevanten Daten suchen. Die Software verbindet sich dann zum UDDI-Eintrag und schaut nach, ob entsprechende Dienste für die Beantwortung der Anfrage vorhanden sind. Diese Dienste werden im Browser angezeigt. Der Nutzer sucht sich dann seinen gewünschten Dienst aus und klickt auf ihn. Der Dienst verbindet sich nun mit dem Anbieter und verwendet dazu ein *Sensor Stream Control Protocol* (SSCP). Die kontinuierlichen Datenströme werden über ein Sensor Stream Transport Protocol (SSTP) dem Browser in Echtzeit übergeben. Die Daten werden anschließend von diesem Browser visualisiert.
Wie komplex eine Anfrage sein kann und ob es sich bei den kontinuierlichen Datenströmen um eine große Menge an Daten handelt, wird in [39] nicht geklärt.

### 3.2.4 Verwendete Datenaustauschformate

SQL hat sich als deklarative und standardisierte Abfragesprache für relationale Datenbanken durchgesetzt. Einige neuere Systeme oder Applikationen setzen SQL nicht mehr ein, weil SQL z.B. keine Sequenz von Daten effizient manipulieren kann.
**Beispiel** (aus [52]): Ein Wetterüberwachungssystem möchte Informationen über meteorologische Ereignisse über einen bestimmten Zeitintervall erhalten. Die Frage lautet: *Für welche Vulkanausbrüche war die Stärke von den meisten neuen Erdbeben größer als 7.0 auf der Richterskala?* Diese Anfrage kann nur schwierig in eine relationale Anfragesprache wie SQL ausgedrückt werden, da sie eine ineffiziente Bewertung der Daten vornehmen würde. Daher wurden SQL-ähnliche Anfragesprachen wie Sub-Query [52] und CQL [3] entwickelt. Dies ist ein Grund, warum SQL nicht eingesetzt wird. Ein weiterer Grund ist, dass sich XML als Standardaustauschformat für Daten durchgesetzt hat, wurde die relationale Anfragesprache durch ausdrucksstärkere Anfragesprachen, die auf XML basieren wie XQuery, abgelöst. Mit XML kann das Veröffentlichen von Daten von vielen Anbietern ermöglicht werden, die zugleich eine flexible Dokumentenstruktur besitzen. Zu den Millionen von Anfragen (historische und kontinuierliche Anfragen) werden Daten mit einer akzeptablen Verzögerung nach einem inhaltsbasierten Mechanismus ausgespielt.
Die Anfragesprache richtet sich meist nach der Darstellung der Daten. Werden Daten in XML dargestellt, verwendet man XQuery. Bei einer Darstellung der Daten als Ontologien, wählten die Forscher und Entwickler ontologische Anfragesprachen (z.B. [63]), weil sie für die entsprechende Darstellung optimiert sind.

### 3.2.5 Überblick über alle Arbeiten

In diesem Abschnitt wird noch ein Überblick über die vorgestellten Arbeiten in Form einer Matrix gegeben. Alle relevanten Kommunikationsmodelle wie Publish/Subscribe, Push und Pull, die für das Sensorsystem der Plath GmbH in Betracht kommen und die zu erfüllenden Anforderungen wie Echtzeit, Datenstromverarbeitung, große Datenmengen, inhaltsbasierte Datenweiterleitung zur Beantwortung von komplexen Anfragen sowie die zwei Anfragetypen historische und kontinuierliche Anfragen werden tabellarisch dargestellt. Ein System sollte im Idealfall ein von den drei Kommunikationsmodellen verwenden und alle Anforderungen erfüllen.

Aus den Tabelleneinträgen ist zu entnehmen, dass es kein System gibt, das alle Anforderungen des Plath Systems erfüllen kann.

| | Publish/ Subscribe | Pull | Push | Echtzeit | Datenstrom |
|---|---|---|---|---|---|
| Ein generisches Notifikations-Framework [58] | X | | | | |
| Streaming Queries over Streaming Data [10] | X | | | | X |
| Fjording the Stream [34] | | X | X | | X |
| Implementing A Scalable XML Publish/Subscribe System [59] | X | | | | |
| A Web Services Environment for Internet-Scale Sensor Computing [39] | | | | X | X |
| Towards an Internet-Scale XML Dissemination Service [14] | X | | X | | X |
| Publish/subscribe in a Mobile Environment [25] | X | | | | |
| Data Stream Management for Historical XML Data [6] | | | X | | |
| Accessing Sensor Data Using Meta Data [45] | | | | X | |
| A Semantic-aware Publish/Subscribe System with RDF Patterns [63] | X | | | | |

Tabelle 3.7: Überblick vorgestellter Arbeiten in Bezug der zu erfüllenden Anforderungen

### 3.2.6 Zusammenfassung

Es wurden verschiedene Publish/Subscribe Systeme, Datenstrom-Management Systeme und Echtzeitsysteme vorgestellt. Bei den Publish/Subscribe Systemen können historische und kontinuierliche Anfragetypen unterschiedlicher Komplexität verarbeitet werden. Bei diesen Systemen ergab sich eine schlechte Performance bei der Verarbeitung von komplexen Anfragen, die auf die Verwendung von XML als Datenaustauschformat zurückzuführen ist. Das OPS System zeigte, dass die Verarbeitungszeit von komplexen Anfragen besser war. Es ist hier aber zu berücksichtigen, dass es sich um nicht so viele veröffentlichten Daten handelte. All diese Systeme arbeiteten mit statischen Daten und es ist nicht geklärt wie die Resultate bei Datenströmen aussehen. Einige Forscher haben Systeme entwickelt, die Datenströme verarbeiten können. Mit diesen Systemen können zwar komplexe Anfragen verarbeitet werden, aber die Nutzer werden dabei nicht individuell mit Sensordaten versorgt. ROADNet und WISE haben sich mit der Problematik der Datenversorgung in Echtzeit beschäftigt. ROADNet geht dabei eher auf die Echtzeitverarbeitung im Bereich des Erfassungssystems ein, wogegen sich WISE auf das Veröffentlichen, Suchen und Analysieren von Daten in Echtzeit beschäftigt.
Die Ideen und Architekturen der Systeme WISE, ONYX und OPS könnten die Grundlage für diese Arbeit bilden. Diese Systeme könnten sich in ihren Funktionalitäten ergänzen und damit alle Anforderungen des Plath Systems erfüllen. Die Architektur von ONYX bietet eine Architektur mit lose gekoppelten Brokern an, die als Dienste agieren könnten. Die Dienstrealisierung kann mit Web Services umgesetzt werden. Dabei sollte wegen der schlechten Perfor-

| | großes Datenvolumen | inhaltsbasiert | hist. Anfrage | kont. Anfrage |
|---|---|---|---|---|
| Ein generisches Notifikations-Framework [58] | | X | X | X |
| Streaming Queries over Streaming Data [10] | X | | X | X |
| Fjording the Stream [34] | X | | X | X |
| Implementing A Scalable XML Publish/Subscribe System [59] | X | X | X | X |
| A Web Services Environment for Internet-Scale Sensor Computing [39] | | X | X | X |
| Towards an Internet-Scale XML Dissemination Service [14] | X | X | X | X |
| Publish/subscribe in a Mobile Environment [25] | | X | X | X |
| Data Stream Management for Historical XML Data [6] | X | | X | |
| Accessing Sensor Data Using Meta Data [45] | X | | X | X |
| A Semantic-arware Publish/Subscribe System with RDF Patterns [63] | | X | X | X |

Tabelle 3.8: Überblick vorgestellter Arbeiten in Bezug der zu erfüllenden Anforderungen

mance nicht unbedingt SOAP als Kommunikationsprotokoll eingesetzt werden, sondern etwas adäquates. Die Daten könnten als Ontologien dargestellt werden, wobei damit ebenfalls die Performance bei der Anfrageverarbeitung darunter leiden kann. Welche Kombination oder Ergänzungen vorgenommen werden, ist speziell für die Anforderungen und über die Analyse Use Cases des Sensorsystems der Plath GmbH abzuschätzen.

# Kapitel 4

# Konzeptionelles Modell

In diesem Kapitel wird unter der Verwendung der in Kapitel 2 eingeführten Terminologie und auf Grundlage der Charakteristika der in Kapitel 3 betrachteten Systeme eine Übersicht über die Anforderungen an die zu modellierende Architektur gegeben, die als Basis für die Systemarchitektur und der prototypischen Realisierung in den Kapiteln 5 und 6 dient.

## 4.1 Anfrageszenario

Die Darstellung des Anfrageszenarios in Abbildung 4.1 motiviert die Diskussion von Anforderungen an ein Interessensprofil, an die Datenmodellierung und an die Leistung eines Dienstes.

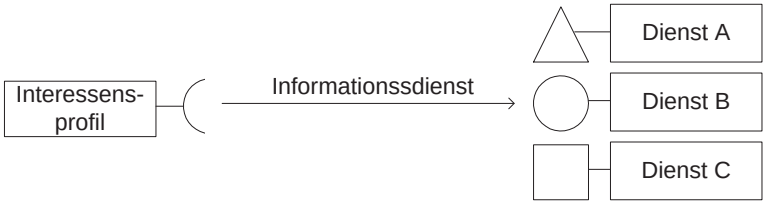


Abbildung 4.1: Anfrageszenario

Ein Interessensprofil kann u.a. hinsichtlich der Freiheitsgrade auf unterschiedliche Weise modelliert werden. In Abschnitt 4.3.1 werden die unterschiedlichen Modellierungen eines Interessensprofils erläutert. Über einen Informationsdienst wird festgestellt, welche Daten zur Abfrage zur Verfügung stehen und welcher Dienst passende Ergebnisse liefern kann. Wird ein passender Dienst gefunden, werden die Daten an die entsprechenden Auswertesysteme ausgespielt. In der Abbildung 4.1 symbolisieren Dreieck, Kreis und Quadrat die Schnittstelle eines Dienstes, welche nur zu einem bestimmten Interessensprofil passt. Das Interessensprofil mit dem Halbkreis passt nur zum Dienst mit der Kreis-Schnittstelle (Dienst B). Die Zuordnung eines Dienstes erfolgt über einen Informationsdienst.

## 4.2 Domänenmodelle

**Definition 4.1** *Mengen von atomaren Werten, denen jeweils eine bestimmte Bedeutung unterstellt ist, nennt man* **Domänen** [46].

Die Modellierung von Sensordaten basiert auf wohlbekannten physikalischen Domänen *Zeit* und *Frequenz*. Anwendungsspezifische Domänen der Firma Plath GmbH sind zum Beispiel

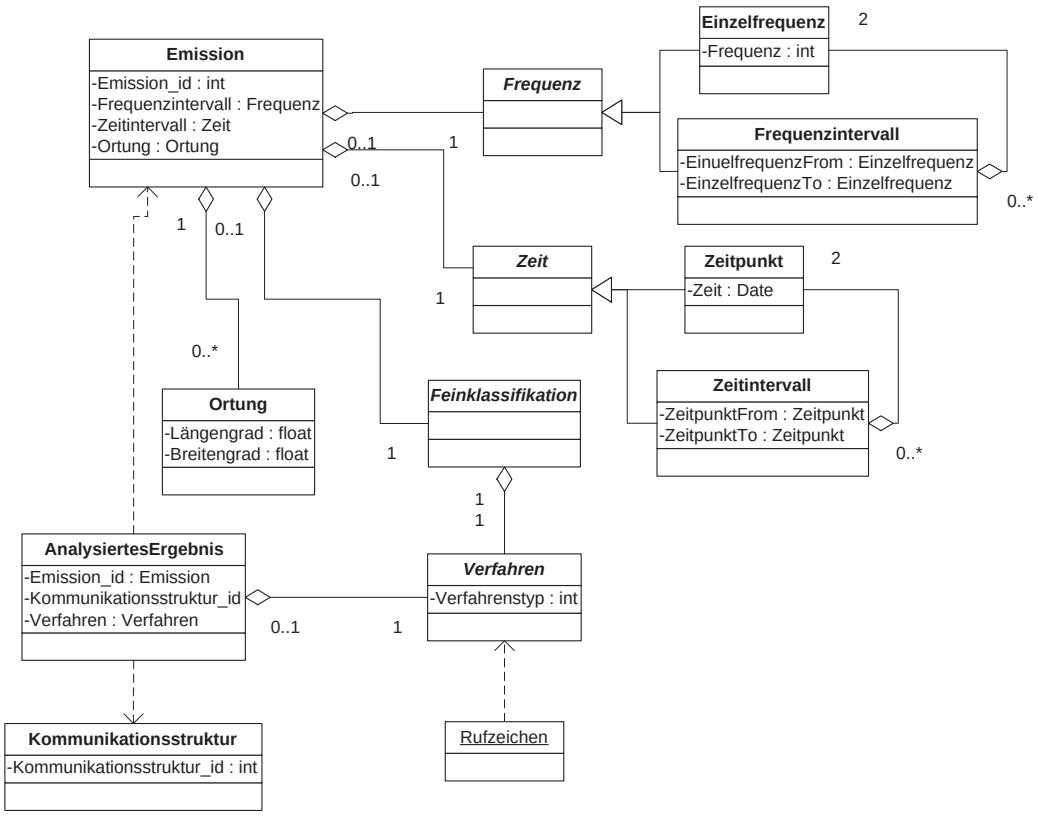


Abbildung 4.2: Konzeptionelles Modell der Rohdaten

*Rohdaten* und das *Interessensprofil*. In diesem Abschnitt werden zwei konzeptionelle Modelle vorgestellt. In Abbildung 4.2 werden die für die Arbeit relevanten Domänen als UML-Diagramm und als XML-Schema modelliert. Das XML-Schema wird für die Überprüfung der Mächtigkeiten einzelner Abfragesprachen in Abschnitt 4.4 benötigt. In Abschnitt 4.2.2 wird das konzeptionelle Modell des Interessensprofils nur als XML-Schema dargestellt, welches das Interesse der Auswertekomponenten widerspiegelt.

### 4.2.1 Konzeptionelles Modell der Rohdaten

**Definition 4.2** *Ein* **Konzept** *beschreibt, wie die Wirklichkeit*[1] *sein soll, es ist also ein Sollkonzept. Ein Sollkonzept ist eine vereinfachte Vorstellung darüber, wie ein bestimmter Ausschnitt der Wirklichkeit aussehen soll bzw. wie er aussieht, wenn diese Vorstellung verwirklicht wird. Konzept entsteht in der Regel auf Grundlage eines Modells* [46].

**Definition 4.3** *Ein* **Modell** *ist ein Abbild der Wirklichkeit. Ein Modell ist immer eine vereinfachte Vorstellung darüber, wie ein bestimmter Ausschnitt der Wirklichkeit tatsächlich beschaffen ist* [46].

Rohdaten setzen sich aus **Emissionen**, **analysierten Ergebnissen** und **Kommunikationsstrukturen** zusammen. Eine Emission ist ein Konzept, das sich aus den Konzepten **Zeitintervall**, **Frequenzintervall** und **Ortung** zusammensetzt. *Zeit* wird verfeinert in **Zeit-**

[1] Die Wirklichkeit ist die individuelle und sozial konstruierte Sichtweise des Betrachters auf ein System [46]

**punkt** und **Zeitintervall** und *Frequenz* wird verfeinert in **Einzelfrequenz** und **Frequenzintervall**. Das Zeitintervall gibt an, wann ein Signal aktiv war. Das Frequenzintervall gibt an, auf welcher Frequenz ein Signal gesendet wurde und die Ortung gibt an, aus welchem Ort. Einer Emission können keine bis endlich vielen Ortungen zugeordnet sein und noch weitere feinklassifizierte Parameter. Dazu gehört die Domäne **Verfahren**. Beim Verfahren kann es sich um *Sprache* oder einer *Datenübertragung* (z. B. Morse) handeln. Aus dem Inhalt eines Verfahrens lässt sich das Rufzeichen des jeweiligen Senders ermitteln. Das Rufzeichen ist also die Kennung eines Signals. Jede **Ortung** wird durch einen *Längengrad* und *Breitengrad* ausgedrückt.
Die analysierten Ergebnisse stellen die Beziehungen zwischen Emissionen, den verwendeten Verfahren und der zugehörigen Kommunikationsstruktur dar. Jede Kommunikationsstruktur besitzt den eindeutigen Bezeichner Kommunikationsstruktur_id.
Dieses UML-Diagramm ist auch als XML-Schema modelliert worden (siehe Abbildung 4.3), weil dieses Schema für die Datenmodellierung benötigt wird.

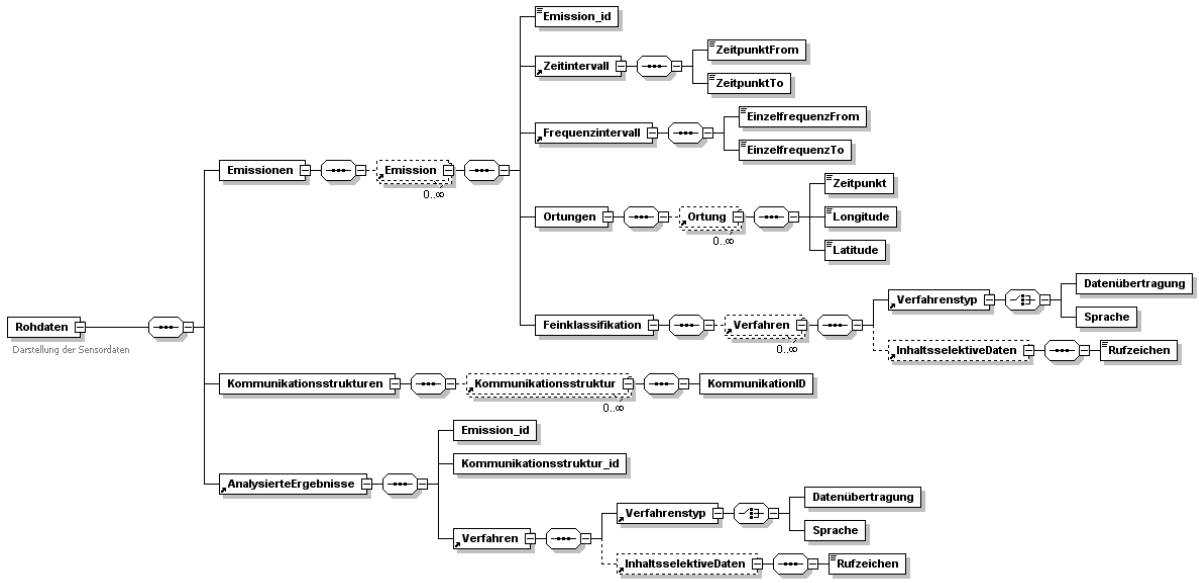


Abbildung 4.3: Konzeptionelles Modell der Rohdaten

**Erläuterung zur XML Schemanotation** Ein Element ist der gebräuchlichste Knotentyp. Er besteht aus einem Start-Tag und einem End-Tag und kann Attribute und weitere Elemente enthalten. Die Attribute in UML werden in dem XML Schema meist auch als Elemente modelliert. Wann ein Attribut und wann ein Element zu verwenden ist, kann aus dem Buch [54] entnommen werden. Mit der Schema-Komponente **Sequenz** legt man für die innerhalb des von ihr aufgespannten Containers deklarierten Elemente eine Reihenfolgenbeziehung fest. Die einzelnen Elemente müssen genau in der Reihenfolge, in der sie innerhalb von der Sequenz erscheinen, auch in einer XML Instanz auftreten [54]. Die Schema-Komponente **Auswahl** definiert eine Auswahl von Elementen, die alternativ als Inhaltsmodell für ein Eltern-Element auftreten können. Wird in der XML-Darstellung z.B. ein Element schraffiert dargestellt, handelt es sich um ein optionales Element. Die Kardinalität $0..\infty$ sagt aus, dass ein Element sich

0 bis $\infty$ wiederholen kann.

**Abbildung UML → XML** Eine Klasse in UML sowie auch die meisten Attribute werden in XML Schema als Element modelliert. Teilweise werden die Attribute auch als Attribute modelliert. Die Kardinalität (0..$\infty$ oder (1..$\infty$)) eines Elementes bildet eine Assoziation ab. Die Abbildung spezifischer Klassen abgeleitet von einer Hauptklasse wird durch Angabe einer Sequenz definiert.

### 4.2.2 Konzeptionelles Modell des Interessensprofils

Ein Interessensprofil erlaubt Anfragen nach signalselektiven, raumselektiven, inhaltsselektiven und nach netzorientierten Kriterien (siehe Abbildung 4.4).

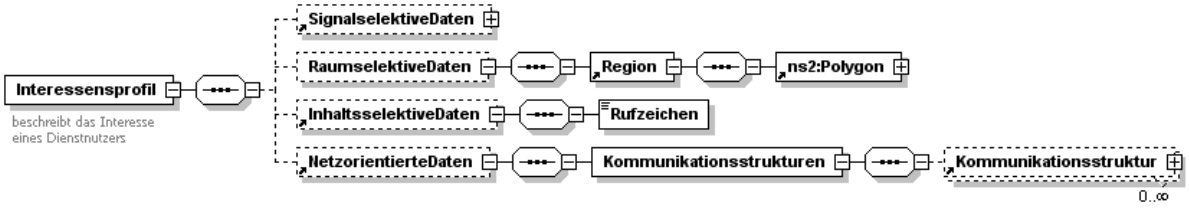


Abbildung 4.4: Interessensprofil

Raumselektive Kriterien beinhalten die Angabe geographischer Rohdaten. Diese werden in geospartialen Polygonen abgebildet. Über inhaltsselektive Kriterien werden die Rohdaten nach ihrem Inhalt gefiltert, z. B. nach Rufzeichen (Kennung des Senders). Wird nach einem Rufzeichen gefiltert, muss das Verfahren einer Emission angegeben sein. Über das netzorientierte Kriterium werden die Rohdaten nach der bestehenden Kommunikationsstruktur gefiltert. Wie sich eine solche Struktur zusammensetzt, wurde in Kapitel 2.3.3 bereits erläutert.
Ein Interessensprofil über signalselektive Kriterien werden Emissionen nach einer Zeitangabe, einer Frequenzangabe und/oder nach einem Verfahren gefiltert (siehe Abbildung 4.3).
Die Sender werden als Teilnehmer bezeichnet. Ein Teilnehmer wird durch eine Frequenzangabe, eine Zeitangabe und einer Ortung beschrieben. Optional wird die Rolle (Anrufer oder Antworter) des Teilnehmers und das Rufzeichen angegeben.
Eine **Zeitangabe** wird im Modell als ein *unscharfer Zeitpunkt*, ein *Zeitintervall* oder als *unscharfes Zeitintervall* modelliert. Im Modell wird eine Zeitpunktangabe mit einem Toleranzwert addiert bzw. subtrahiert. Daraus resultiert ein Zeitintervall. Dieses Zeitintervall wird als *UnscharferZeitpunkt* bezeichnet. Es gilt:

$$\text{Unscharfer Zeitpunkt} \in [\text{ZeitpunktFrom,ZeitpunktTo}]$$

Das Zeitintervall wird durch zwei Zeitpunkte *ZeitpunktFrom* und *ZeitpunktTo* gebildet. Es gilt:

$$\text{Zeitintervall} \in [\text{ZeitpunktFrom,ZeitpunktTo}]$$

Ein unscharfes Zeitintervall wird im Modell durch eine Startzeit (*ZeitBegonnenAb*) oder durch eine Endzeit (*ZeitBeendetZu*), die jeweils als Zeitpunkte modelliert werden. Es gilt:

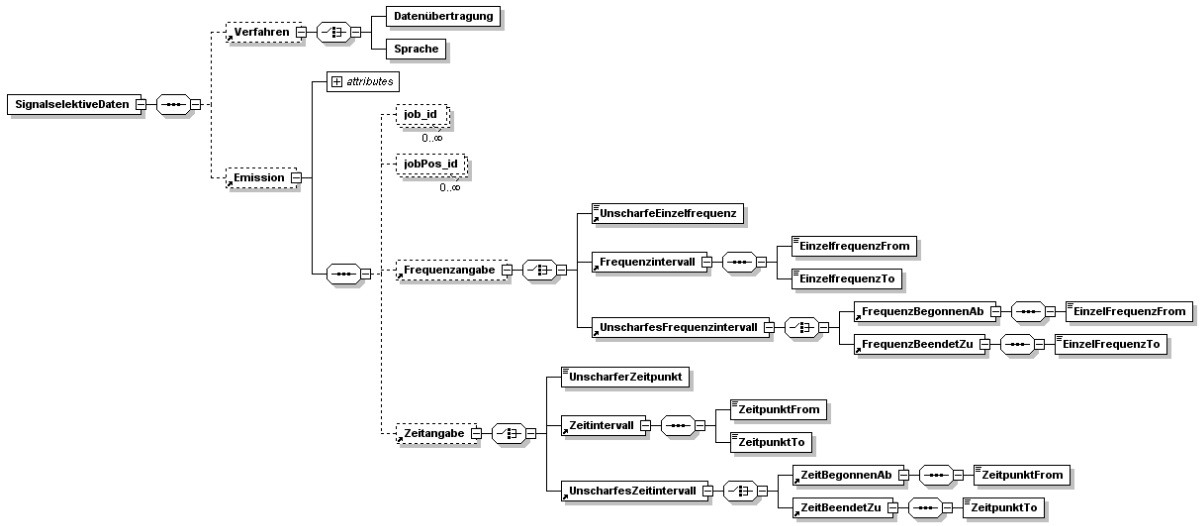


Abbildung 4.5: Signalselektives Kriterium des Interessensprofils

$$\begin{aligned} \text{Unscharfes Zeitintervall} &\in \text{(ZeitpunktFrom,ZeitpunktTo] oder} \\ \text{Unscharfes Zeitintervall} &\in \text{[ZeitpunktFrom,ZeitpunktTo)} \end{aligned}$$

Eine **Frequenzangabe** wird im Modell als ein *unscharfe Einzelfrequenz*, ein *Frequenzintervall* oder als *unscharfes Frequenzintervall* modelliert. Im Modell wird eine Einzelfrequenz mit einem Toleranzwert addiert bzw. subtrahiert, sodass sich ein Frequenzintervall ergibt. Dieses Frequenzintervall wird als *UnscharfeEinzelfrequenz* bezeichnet. Es gilt:

$$\text{Unscharfe Einzelfrequenz} \in \text{[EinzelfrequenzFrom,EinzelfrequenzTo]}$$

Das Frequenzintervall wird durch zwei Einzelfrequenzen gebildet. Es gilt:

$$\text{Frequenzintervall} \in \text{[EinzelfrequenzFrom,EinzelfrequenzTo]}$$

Ein unscharfes Frequenzintervall wird im Modell durch eine minimale Einzelfrequenz (*FrequenzBegonnenAb*) oder durch eine maximale Einzelfrequenz (*FrequenzBeendetZu*) modelliert. Es gilt:

$$\begin{aligned} \text{Unscharfes Frequenzintervall} &\in \text{(EinzelfrequenzFrom,EinzelfrequenzTo] oder} \\ \text{Unscharfes Frequenzintervall} &\in \text{[EinzelfrequenzFrom,EinzelfrequenzTo)} \end{aligned}$$

Ein Teilnehmer wird einer Kommunikationsstruktur zugeordnet. Ein Teilnehmer setzt sich aus einer Frequenzangabe, einer Zeitangabe und einer Region zusammen. Optional kann die Rolle und das Rufzeichen eines Teilnehmers angegeben sein.

In Abbildung 2.7 wurden die Unterschiede zwischen kontinuierlichen und historischen Interessensprofilen erläutert. Zu kontinuierlichen Anfragen werden kontinuierlich Daten ausgespielt. Historischen Anfragen werden dagegen einmalig ausgewertet. In der Praxis interessieren sich Dienstnutzer auch regelmäßig für historische Daten, d.h. sie wollen regelmäßig darüber informiert werden, z.B. welche Emissionen in der letzten Woche erfasst worden sind. Das Interessensprofil unterscheidet daher nicht zwischen historischen und kontinuierlichen Anfragen. Historische Anfragen werden wie kontinuierliche Anfragen behandelt.

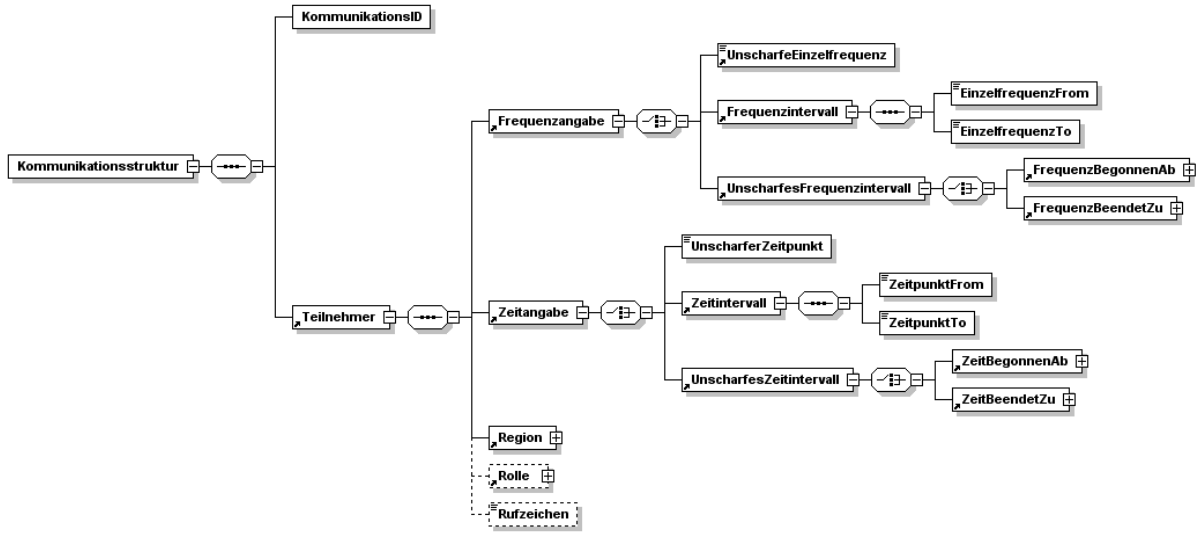


Abbildung 4.6: Netzorientiertes Kriterium des Interessensprofils

## 4.3 Dienstbeschreibung

Ein Interessensprofil kann auf verschiedenster Weise ausgedrückt werden. In diesem Abschnitt werden drei mögliche Arten ein Interessensprofil auszudrücken erläutert hinsichtlich gegebener bzw. nicht gegebener Freiheitsgrade. Dazu wird erläutert, welche Datenrepräsentation benötigt wird und was ein Dienst leisten muss.

### 4.3.1 Klassifikation von Interessensprofilen

In Abbildung 4.7 werden drei Typen von Interessensprofilen definiert: Ein Interessensprofil kann

- Typ 1: nicht
- Typ 2: teilweise
- Typ 3: vollkommen

frei definiert werden. Die Typen unterscheiden sich bei der gegebenen Anzahl an Freiheitsgraden.
Für die unterschiedlichen Typen werden unterschiedliche Anforderungen an einen Informationsdienst und an die Leistung eines Dienstes gestellt, die im Folgenden erläutert werden.

**Beispiel 4.1** *Typ 1*

Ein Dienstnutzer kann sein Interessensprofil nicht frei definieren.

**Anforderungen an den Informationsdienst:**
Ein Informationsdienst stellt die Informationen über die vorhandenen Dienste bereit.
**Beispiel:** Ein Informationsdienst liefert über den Aufruf einer Methode die Information, dass folgende Interessensprofile existieren:

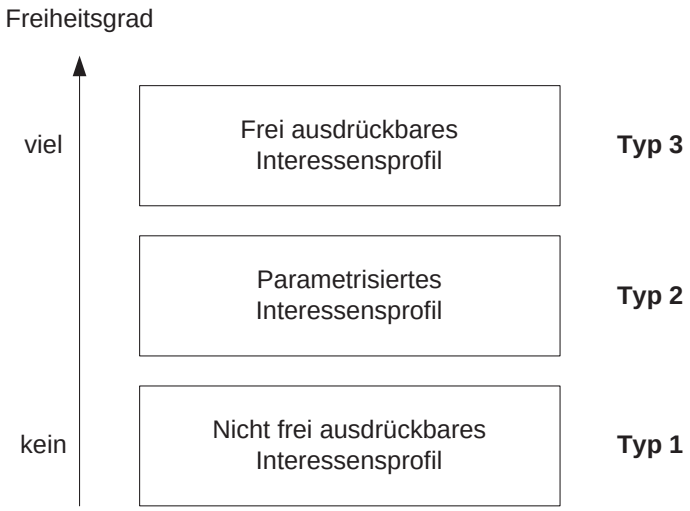


Abbildung 4.7: Freiheitsgrade unterschiedlicher Anfragetypen

1. Alle Frequenzen zu allen Emissionen im Zeitintervall von 8:00 bis 10:00 Uhr,
2. Alle Emissionen mit dem Verfahren Sprache.

Der Dienst spielt zu den entsprechenden Interessensprofilen Daten aus dem Data Repository an den Dienstnutzer aus.

**Vorteile:**

- Die Dienstnutzer müssen keine Kenntnis über das Sensordatenmodell des Diensterbringers besitzen, weil das Interessensprofil im Dienst gekapselt ist.
- Zeitnahe Auswertung kann garantiert werden, weil die Komplexität des Interessensprofil vom Diensterbringer bestimmt wird.

**Nachteile:**

- Interessen aller Dienstnutzer werden nicht abgedeckt, weil kein eigenes Interessensprofil definiert werden kann.

**Beispiel 4.2** *Typ 2*

Ein Dienstnutzer kann sein Interessensprofil teilweise definieren. Es wird ihm ermöglicht über eine vorgegebene feste Struktur Parameter auswählen.

**Anforderungen an den Informationsdienst:**
Ein Informationsdienst stellt die Informationen über die vorhandenen Dienste bereit.
**Beispiel:** Ein Informationsdienst liefert über den Aufruf einer Methode, der man Parameter übergibt, die Information, dass folgende Interessensprofile definiert werden können:

- Dienst A: Alle Frequenzen zu allen Emissionen im Zeitintervall von *X:Zeitpunkt* bis *Y:Zeitpunkt* Uhr,

- Dienst B: Alle Emissionen mit den Verfahren(Verfahrenstyp). Ein Verfahrenstyp kann z.B. Sprache oder Datenübertragung sein.

Ein Dienst liefert hier die Funktionalität Zeitintervalle oder Arten von Verfahren über Parameter auszuwählen.

**Vorteile:**

- Die Dienstnutzer müssen keine Kenntnis über das Sensordatenmodell des Diensterbringers besitzen, weil das Interessensprofil im Dienst gekapselt ist.
- Zeitnahe Auswertung kann garantiert werden, weil die Komplexität des Interessensprofil vom Diensterbringer bestimmt wird.
- Interessensprofile können flexibler ausgedrückt werden im Vergleich zu Typ 1.

**Nachteile:**

- Interessen aller Dienstnutzer werden nicht abgedeckt, weil kein eigenes Interessensprofil definiert werden kann.

**Beispiel 4.3** *Typ 3*

Ein Dienstnutzer kann ein eigenes Interessensprofil frei definieren.

**Anforderungen an den Informationsdienst:**
Ein Informationsdienst stellt die Informationen wie diese Daten repräsentiert werden und welche Anfragesprache ein solcher Dienst unterstützt über die vorhandenen Dienste bereit.
Ein eigenes Interessensprofil wird durch eine existierende Anfragesprache oder durch eine Framesprache definiert, die vom Diensterbringer vorgegeben wird. Im Folgenden wird jeweils ein Beispiel für die Definition eines Interessensprofils gegeben und damit gezeigt, welche unterschiedlichen Möglichkeiten und den unterschiedlichen Anforderungen es gibt sein Interessensprofil frei zu definieren. Auf die Framesprache wird nur kurz eingegangen. Das entsprechende Beispiel soll an dieser Stelle einen kleinen Einblick in eine Modellsprache geben und aufzeigen, dass ein Dienst dabei mehr leisten muss als die Verwendung einer existierenden Anfragesprache.

**Interessensprofil mittels Anfragesprache definieren**

Mit einer Anfragesprache z.B. einen Subset von SQL kann das eigene Interessensprofil so formuliert werden:

```
SELECT
FROM
WHERE
```

Hier wird im Gegensatz zu Typ 2 nicht nur erlaubt den `WHERE`-Teil frei zu definieren, sondern auch den `FROM`- und `SELECT`-Teil. Welche Probleme durch eine freie Formulierung seines Interessensprofils auftreten können, wird an folgenden zwei Beispielen erläutert:

**Beispiel 4.4** *„Gibt es einen neuen Teilnehmer in einer bekannten Kommunikationsstruktur?“*

Die o.g. Anfrage könnte ein Dienstnutzer in SQL so ausdrücken:

```
SELECT e.Frequenz, e.Einzelfrequenz, e.Stopzeit, ar.Verfahren, e.Ortung,
       e.Rufzeichen, k.Kommunikationsstruktur_id
FROM Emission e, Kommunikationsstruktur k, AnalysiertesErgebnis ar
WHERE EXISTS
      (SELECT k.*
       FROM Kommunikationsstruktur k
       WHERE e.Frequenz = k.Frequenz AND
             ar.EmissionID = e.EmissionID AND
             ar.Verfahren = k.Verfahren AND
             e.Ortung != k.Ortung);
```

**Probleme:**

- Eine geospartiale Domäne kann in SQL nicht ausgedrückt werden.

**Leistung eines Dienstes:**
Ein Dienst muss den Subset von SQL auswerten.

**Beispiel 4.5** *„Liefere mir alle Emissionen zur HF-Überwachung aus der Region Mexiko!“*

**Probleme:**

- Die Domänen *Region* und *HF* sind in SQL kein Begriff.

**Leistung eines Dienstes:**
Um die Anfrage nach einer Region zu beantworten, muss der Dienst einen anderen Dienst heranziehen. Dieser Dienst bildet eine Region in ein geographisches Polygon und den HF-Begriff in ein Frequenzintervall von $10kHz$ bis $30MHz$ ab.

**Vorteile:**

- Dienstnutzer kann sein Interessensprofil im Rahmen der Sensordatenstruktur des Diensterbringers und mit der unterstützenden Anfragesprache, die vom Diensterbringer vorgegeben wird, modelliert werden.

**Nachteile:**

- Dienstnutzer muss Kenntnis über die interne Rohdatenrepräsentation des Diensterbringers haben.
- Zeitnahe Auswertung einer Anfrage des Dienstnutzers kann nicht garantiert werden, weil die Komplexität des Interessensprofil nicht vom Diensterbringer bestimmt wird.

#### Interessensprofil mittels Modellsprache definieren

Möglich ist es auch, das Interessensprofil mittels einer Framesprache zu definieren. Framesprachen wie CKML (Structure-based Configuration Language) [22] oder eine Anwendung, die mittels einer Framesprache beschrieben wird, werden in [23], [24] und [37] beschrieben. Ein Ausschnitt einer Framesprache kann folgendermaßen aussehen:

```
BNachfolgeEmissionVonA
Teile:          Emission A mit Frequenzbereich [1..30MHz]
                           mit Dauer > 2 sek
                Emission B
Bedingung:      A.Ende vor B.Beginn
                B.EinzelfrequenzFrom != A.EinzelfrequenzFrom
                B.EinzelfrequenzTo != A.EinzelfrequenzTo
Eigenschaften:  IstNeuerZeitpunkt.Beginn = B.Beginn
                IstNeuerZeitpunkt.Ende = B.Ende
```

Ein Dienstnutzer definiert ein eigenes Modell für **BNachfolgeEmissionVonA**. Dieses Modell steht für eine Menge von Emissionspaaren A und B (Teile), die die folgenden Bedigungen erfüllen müssen: A muss geendet haben, bevor B beginnt und deren Einzelfrequenzen müssen unterschiedlich sein. Die Eigenschaften des Aggregates werden aus den Eigenschaften der Teile berechnet. Mathematisch lässt sich dieses Modell so ausdrücken:

$$
\begin{aligned}
\forall A : \exists B \text{ mit } & \text{A.Ende vor B.Beginn} \wedge \\
& \text{B.EinzelfrequenzFrom} \neq \text{A.EinzelfrequenzFrom} \wedge \\
& \text{B.EinzelfrequenzTo} \neq \text{A.EinzelfrequenzTo} \\
& \Rightarrow \text{IstNeuerZeitpunkt.Beginn} = \text{B.Beginn} \wedge \\
& \quad \text{IstNeuerZeitpunkt.Ende} = \text{B.Ende}
\end{aligned}
$$

**Leistung eines Dienstes:**
Ein generischer Dienst muss vorher vereinbaren, dass eine Modellsprache zum Einsatz kommt und führt die Auswertung des Interessensprofils durch.

**Vorteile:**

- Ein Dienstnutzer kann sein Interessensprofil frei definieren.

**Nachteile:**

- Ein Dienstnutzer muss Kenntnis über die interne Datenrepräsentation haben.
- Zeitnahe Auswertung der Anfrage kann nicht garantiert werden, weil die Komplexität des Interessensprofil nicht vom Dienstnutzer definiert wird.

**Zusammenfassung**

Es wurden drei verschiedene Typen vorgestellt, die Angeben wie ein Interessensprofil definiert werden kann. Jeder Typ zeichnet sich durch den vorgegebenen Freiheitsgrad aus um ein Interessensprofil zu definieren. Je größer der Freiheitsgrad, desto weniger kann eine zeitnahe Auswertung gewährleistet werden. Zum einen liegt es an der großen Datenmenge, die kontinuierlich verarbeitet und gegen die Anfragen abgeglichen wird und zum anderen an der Komplexität der Interessensprofile. Die beiden Anfragen

- Gib mir alle Frequenzen aller Emissionen im Zeitintervall von 8 bis 10 Uhr!
- Gib mir alle Emissionen mit dem Verfahren X!

besitzen die Komplexität $O(n)$. Die Anfrage

- Gibt es einen neuen Teilnehmer in einer Kommunikationsstruktur?

besitzt die Komplexität $O(n^2)$ oder im besseren Fall $O(n \log n)$. Die Anfrage

- Liefere mir alle Emissionen zur HF-Überwachung aus der Region Mexiko!

besitzt die Komplexität $O(n)$, wobei die Auswertung des Prädikats *Region* Mexiko komplex ist.
Eine zeitnahe Auswertung kann bei Anfragen mit der Komplexität $O(n)$ oder ggf. $O(n \log n)$ erfolgen. Für komplexere Anfragen müssten Einschränkungen bzgl. der Auswertung vorgenommen werden.

## 4.4 Analyse vorgestellter Datenrepräsentations- und Anfragesprachen

In diesem Abschnitt werden die vorgestellten Sprachen in Kapitel 3 bezüglich der Anforderungen analysiert und deren Mächtigkeit bestimmt. Die Analyse der Sprachen wird als Paar vorgenommen, d.h. es wird eine Datenrepräsentationssprache mit der zugehörigen Anfragesprache analysiert, weil diese Sprachen aufeinander abgestimmt sind und damit die Anfragesprache die Vorteile einer Datenrepräsentationssprache nutzt.
Für die Wahl einer externen Datenrepräsentationssprache sollte eine Sprache ausgewählt werden, welche die eigentliche Struktur der internen Datenrepräsentation kapselt, weil die interne Struktur auf der einen Seite nicht gerne veröffentlicht wird und auf der anderen Seite, die interne Struktur von vielen Dienstnutzern nicht verstanden wird.
Für die Wahl einer Anfragesprache muss analysiert werden, welche Sprachen den Anforderungen aus Abschnitt 4.4.1 gerecht werden.

### 4.4.1 Anforderungen

Zu den allgemeinen Anforderungen gehören

- eine zeitnahe Auswertung und Ausspielung von Sensordaten
- bei großem Datenvolumen,
- aus kontinuierlichen Datenströmen unter
- der Verwendung des Publish/Subscribe Paradigmas.

Aus den Anfragenbeispielen ergeben sich folgende Anforderungen: Filterung bzw. Auswertung nach

- Zeitpunkten,
- Zeitintervallen,
- Zeitreihen,
- Frequenzintervallen,
- geographischen Daten (Längen- und Breitengrade),

- Netzstruktur (Beziehungen der Sensordaten untereinander)
- Kommunikationsstruktur (Zuordnung von Mengen, wobei die Kanten Eigenschaften besitzen) und
- Mengen („Liegt ein Punkt im Polygon X?“).

Die Unterschiede zwischen Netzstruktur und Kommunikationsstruktur werden in dem Abschnitt 5.2.2 genauer erläutert.
All diese Anforderungen sollten die auszuwählenden Sprachen erfüllen.

### 4.4.2 Relationale Tupel (R-Tupel) und SQL

Die relational vorliegenden Rohdaten werden aus der objekrelationalen Abbildung eines Objektmodells gewonnen. SQL eignet sich nur für die relationale Darstellung, daher wird nur eine Analyse der relationalen Darstellungsform durchgeführt. Die Repräsentation der Sensordaten in relationalen Tupeln kapselt nicht die interne Datenstruktur und ist für die verschiedenen Dienstnutzer nicht verständlich, weil sie nicht eindeutig ist.
SQL ist eine relational vollständige Anfragesprache, daher können prinzipiell alle Anfragen beantwortet werden, sofern die Daten in Tupeln vorliegen. Für eine Punkt-In-Polygon-Anfrage[2] werden Tupel mit den entsprechenden Einträgen für Regionen o.ä. benötigt, damit diese Anfrage beantwortet werden kann. Relationale Tupel eignen sich jedoch nicht für die Repräsentation von geospartialen Daten. Die Anfragen lassen sich in Echtzeit beantworten, sofern die Anfragen nicht komplex sind (also kleiner als $O(n^2)$ und keine große Datenmenge vorhanden ist.

### 4.4.3 XML, GML und XQuery

Mittels XML Schema und GML können die Daten in einer externen Datenstruktur auf einer standardisierten Weise gekapselt und dargestellt werden. Dienstnutzer können die externe Datenrepräsentation abfragen und daraufhin ihr Interessensprofil definieren.
XQuery ist eine Turing-vollständige auf XML basierende Anfragesprache. Damit können Zeitanfragen, Frequenzintervalle, Geodaten und Unschärfe abgefragt werden, aber keine Netzstrukturen, weil diese Sprache auf Bäumen operiert und man für die Beantwortung netzorientierter Anfragen Graphen benötigt [33].

### 4.4.4 SensorML, O&M, FilterEncoding, SOS und SCS

Es gibt unterschiedliche geospartiale Sprachen, die zurzeit noch in der Entwicklung sind und voraussichtlich in kürzester Zeit neue Funktionalitäten liefern - speziell die geospartialtemporale Anfragesprachen. An dieser Stelle werden nur die Anfragesprachen Filter Encoding, SCS und teilweise der ASCS analysiert, weil der SOS noch nicht existiert und der ASCS nicht standardisiert ist. Beim ASCS wird die Erweiterung des SCS um das Push-Modell analysiert.

#### SensorML und O&M

Die Sprachen SensorML und O&M repräsentieren Sensor- und Messdaten in einer standardisierten Form.
Das O&M Modell besteht aus mehreren auf einander bezogenen konzeptionellen Modellen, dazu zählen u.a. das *Observation Model* und das *Phenomenon Model* (siehe Abbildung 4.8).

[2] Z.B.: „Liefere alle Emission aus der Region Mexiko von 10 bis 11 Uhr?“

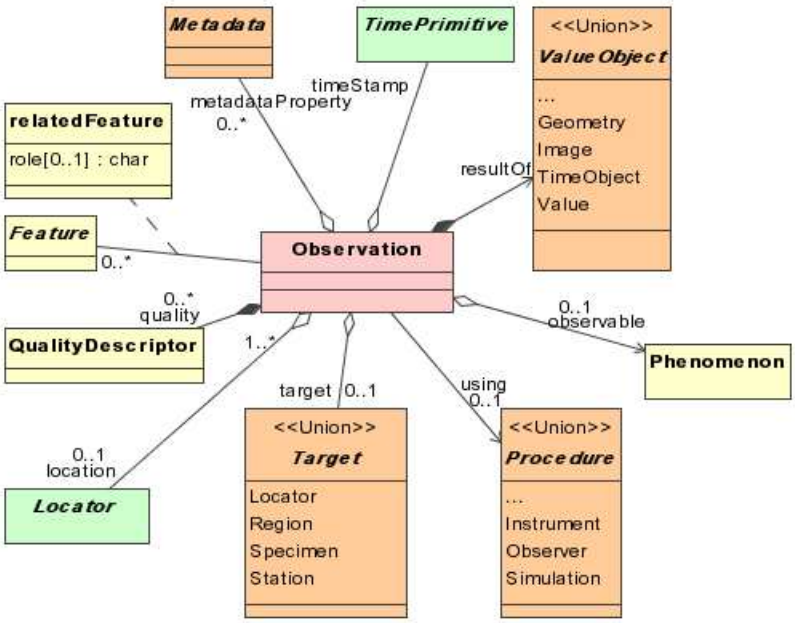


Abbildung 4.8: Observations & Measurements Modell in UML-Notation [11]

**Observation Model** Eine `Observation` ist ein Element, welches die Rohdaten aus dem konzeptionellen Modell abbildet und wird als Feature bezeichnet. Es besitzt die Eigenschaft `resultOf`, dessen Wert ein Phänomen beschreibt und ist vom abstrakten Datentyp `Value Object`. Mit welchen konkreten Datentypen dieser bei einer Instanziierung ersetzt werden kann, wird im *Value Model* beschrieben. Ein `Measurement` ist eine spezielle Beobachtung, die einen Sensor nutzt und die Messgröße in einen skalaren Wert übersetzt.
Eine Observation hat einen `timeStamp`, wenn das Observation Feature ein Ereignis beschreibt und hat eine `using` Eigenschaft, dessen Wert eine Beschreibung einer Prozedur (`Procedure`) ist.
Eine Observations können zu anderen Features, z.B. zu einer Observation Collection aggregiert werden, die sich wiederum wie eine Observation behandeln lassen. Die Assoziation mit anderen Features verwenden eine `relatedFeature` Assoziationsklasse mit einem Rollen Attribut `role`. Verwandte Features haben eine feste Rolle `target`. Eine Observation kann auch `quality` Indikatoren besitzen, die mit ihnen assoziiert werden. Der räumliche Bezug wird mit `location` hergestellt [11].

**Phenomenon Model** Das Phänomenen Modell (Phenomenon Model) ist ein Teil der GML Spezifikation (`phenomenon.xsd`). Ein Phänomen besteht grundlegend aus einem Namen und einer Beschreibung wie im folgenden Beispiel gezeigt:

```
<om:Phenomenon gml:id="Emission">
<gml:description>Besteht aus einem Frequenz- und einem
 Zeitintervall</gml:description>
```

Falls eine Definition von einer Quelle verfügbar ist, kann sie über eine URI referenziert werden:

```
<om:Phenomenon gml:id="Emission>
<gml:description xlink:href="http://www.plath.de/Emission/>
```

Darüber hinaus können erweiterte Phänomentypen wie `ParameterisedPhenomenon`, `CompositePhenomenon` usw. detaillierte Informationen zu den einzelnen Phänomenen geben, die in [11] beschrieben sind.

Zeitpunkte können mit dem *timestamp* und Zeitintervalle können über die Eigenschaft `validTime` einer `Observation` repräsentiert werden. Zeitreihen und Ortungen lassen sich über die Eigenschaft `boundedBy` einer `Observation` abbilden, sodass das Tracking eines Signals verfolgt werden kann. Netzstrukturen können durch Beziehungen der Observations untereinander abgebildet werden. Kommunikationsstrukturen können nicht repräsentiert werden. Unschärfe geographischer Daten und von Zeitdaten wird auch unterstützt (z.B. über die Eigenschaft `BoundingBox`). Frequenzintervalle lassen sich nicht darstellen.

**Filter Encoding**
Filter Encoding kann nur einfache Filteranfragen vornehmen. Es wurde in den o.g. Anfragebeispielen gezeigt, dass sich nicht alle Anfragen durch einfache Filteranfragen beantworten lassen. Es müssen auch Vergleiche vorgenommen werden. Filter Encoding kann keine Vergleiche vornehmen und basiert auf dem Pull-Modell, daher kommt diese Anfragesprache für diese Arbeit nicht in Betracht.

**SCS**
Wie bereits in Kapitel 3 erwähnt, besitzt der SCS die vier Funktionen `GetCapabilities`, `DescribePlatform`, `DescribeSensor` und `GetObservation`.
Die `GetCapabilities` Funktion ist der Standard einer GetCapabilities Funktion, die vom OWSCommon.getCababilities() definiert wurde. Es ist eine HTTP POST Schema Spezifikation, die folgenden Aufbau besitzt:

```
<xsd:complexType name="GetCapabilitiesType" id="GetCapabilitiesType">
 <xsd:annotation>
 <xsd:documentation xml:lang="en">
 Requests the capability profile for the service.
 </xsd:documentation>
 </xsd:annotation>
 <xsd:sequence>
 <xsd:element name="timeStamp" type="xsd:dateTime" minOccurs="0"/>
 <xsd:choice minOccurs="0">
 <xsd:element name="view" type="ows:idListType" maxOccurs="unbounded"/>
 </xsd:choice>
 </xsd:sequence>
 <xsd:attribute name="service" type="xsd:string" use="optional"/>
 <xsd:attribute name="version" type="xsd:string" use="optional"
   default="0.7.1"/>
</xsd:complexType>
```

Die `GetCapabilities` Funktion liefert Metadaten eines Dienstes.

Eine `GetObservation` Funktion ist in Abbildung 4.9 illustriert, die eine oder mehrere `Query`-Elemente beinhaltet, welche die Observations vom SCS einschränkt.

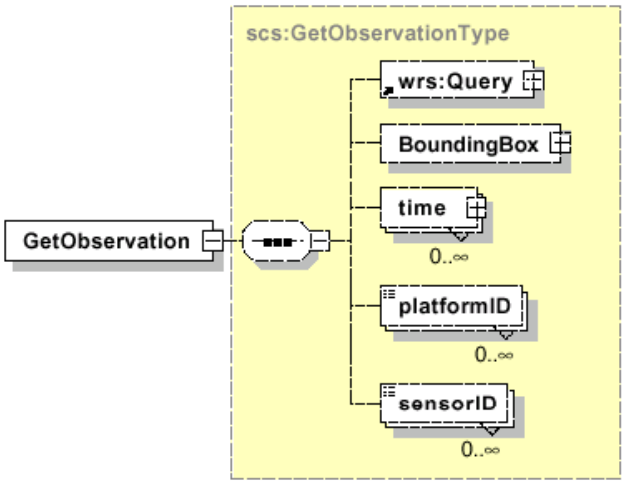


Abbildung 4.9: Schematische Darstellung von `GetObservation`

Die `GetObservation` Funktion spezifiert einen anzugebenden Parameter (`BoundingBox`) und vier optionale Parameter (`wrs:Query`, `time`, `platformID`, `sensorID`). Das Ergebnis dieser Anfrage wird in dem XML-Format zurückgeliefert.

**wrs:Query** Der optionale Parameter `wrs:Query` erlaubt dem Nutzer die Anfrage nach spezifischen Eigenschaften, welche vom Sensor erfasst und durch den SCS veröffentlicht werden. Die Anfrageschnittstelle für den SCS wurde von der Web Registry Service Implementation Spezification importiert, weil es die erweiterte OGC Anfragesyntax dahingehend unterstützt, eine multi-type oder eine „join"-Anfrage zu spezifieren, die eine Bindung zwischen zwei registrierten Objekten bildet kann. Der folgende XML-Code illustriert einen Ausschnitt von `wrs:Query`. In diesem Beispiel fragt ein Nutzer die <HF-Datenbank> nach einer Frequenz die größer als 1000000 ist. Die Information, die diese Anfrage unterstützt beinhaltet die Eigenschaft <Frequenz>.

```
<?xml version="1.0" encoding="UTF-8"?>
<GetRecord xmlns="http://www.opengis.net/wrs"
           xmlns:ogc="http://www.opengis.net/ogc" outputFormat="XML">
  <Query typeName="Hochfrequenz Messwerte">
    <ogc:PropertyName>HF-Datenbank</ogc:PropertyName>
    <QueryConstraint>
      <ogc:Filter>
        <ogc:PropertyIsGreaterThan>
          <ogc:PropertyName>Frequenz</ogc:PropertyName>
          <ogc:Literal>1000000</ogc:Literal>
        </ogc:PropertyIsGreaterThan>
      </ogc:Filter>
    </QueryConstraint>
  </Query>
</GetRecord>
```

Für eine solche Anfrage gibt es bislang keine standardisierte Spezifikation, sondern es besteht lediglich nur diese Idee.

**BoundingBox** Der `BoundingBox`-Parameter definiert, welcher Raumausschnitt für die Messinformationen abgefragt werden soll. Es können Angaben in der Form $min_x$, $min_y$, $max_x$ und $max_y$ gemacht werden. Der `BoundingBox`-Parameter wurde vom gml:EnvelopeType übernommen und hat die optionalen Attribute `gml:id`, `id` und `srsName`. Ein BoundingBox Code Fragment einer `GetObservation` Anfrage kann so aussehen:

```
<BoundingBox srsName="Mexiko">
  <gml:coord>
    <gml:X>-77.06245743559009</gml:X>
    <gml:Y>38.76401735867015</gml:Y>
  </gml:coord>
  <gml:coord>
    <gml:X>-77.00824254832564</gml:X>
    <gml:Y>38.82208357672388</gml:Y>
  </gml:coord>
</BoundingBox>
```

**time** Der `time`-Parameter schränkt die Anfrage hinsichtlich bestimmter Zeitpunkte oder Zeitintervalle ein.

**platformID** Die `platformID` schränkt die Ausgabe von Messwerten bestimmter Sensoren ein, die in der `platformID` angegeben sind.

**sensorID** Alle Sensoren, die in `sensorID` angeben sind, liefern spezifische Ergebnisse, wie die Qualität eines Signals, zurück.

Der SCS ermöglicht mit der Funktion `GetObservation` die Abfrage nach raumzeitvarianten Sensordaten, die gemäß des Pull-Modells an den Nutzer ausgespielt werden. D.h. wenn ein Nutzer sich nur für einen auffälligen Messwert interessiert, muss dieser in periodischen Abständen (z.B. jede 5 Sekunden) seine Anfrage (`GetObservation`) an den Dienst stellen, ob ein auffälliger Messwert aufgetreten ist (siehe Abbildung 4.10), was zu einer erhöhten Netzlast führen kann.

**ASCS** Der ASCS besitzt zusätzlich zum SCS die Funktionen `SubscribeObservation` und `UnsubscribeObservation`. Mit `SubscribeObservation` kann ein Dienstnutzer seine Anfrage registrieren. So wird ein Dienstnutzer nur informiert, wenn ein auffälliger Messwert erfasst wurde. Damit wird der Dienstnutzer automatisch informiert (siehe Abbildung 4.11), wenn ein auffälliger Messwert erfasst wurde.
Aus den Kapiteln 2 und 3 ist zu entnehmen, dass ein Nutzer zum einen kontinuierlich Daten erhalten möchte ohne eine bestimmte Bedingung vorzugeben und zusätzlich auch informiert werden möchte, wenn ein auffälliger Messwert auftritt. Es wird also eine Kombination der beiden Modelle benötigt, welche bislang noch nicht existiert.

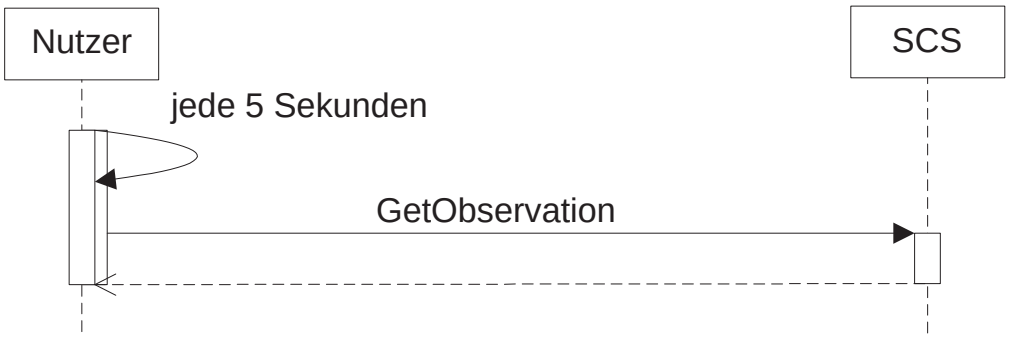


Abbildung 4.10: Interaktion basierend auf dem Pull-Modell

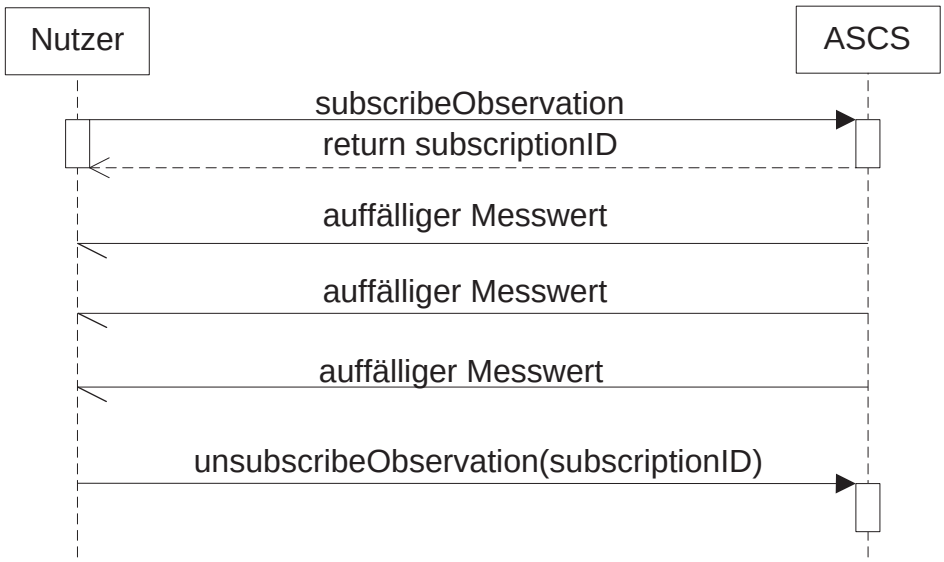


Abbildung 4.11: Interaktion basierend auf dem Push-Modell

### 4.4.5 RDF, RDQL und SparQL

In RDF werden die Daten nicht auf technischer Ebene, sondern auf ontologischer Ebene repräsentiert. Es können Daten eindeutig beschrieben werden und bieten den Dienstnutzern komplexe Anfragen zu stellen, zu denen automatisierte Antworten geliefert werden können. Für RDF gibt es standardisierte Spezifikationen, die angeben, wie Zeiten und Geodaten dargestellt werden. Frequenzintervalle können mit RDF nicht dargestellt werden.
Eine Punkt-In-Polygon- oder netzorientierte Anfrage kann keine Anfragesprache für RDF lösen. Das OPS System [63] zeigte, dass Filteranfragen zu den zehn modellierten Klassen mit je zwei Eigenschaften noch zeitnah ausgewertet werden können. Werden die Klassen abstrakter, wie es für Netzstrukturen oder geographische Angaben nötig ist, und besitzen sie mehr als nur zwei Eigenschaften, kann man sich vorstellen, dass es Probleme mit der zeitnahen Auswertung geben wird.

## 4.5 Zusammenfassung

In der Tabelle 4.1 wird dargestellt, welche Sprachen[3] die Anforderungen aus Abschnitt 4.4.1 erfüllen können. Der Eintrag „ ja“ bedeutet, dass die Anforderung von der Datenrepräsentations- und Anfragesprache erfüllt wird, „nein“ bedeutet das Gegenteil und die Einträge, die mit einem „?“ versehen sind, wurden nicht überprüft.

| | R-Tupel/ SQL | XML/GML XQuery | SensorML/ O&M/SCS | RDF/ RDQL | RDF/ SparQL | OWL/ OWL-QL |
|---|---|---|---|---|---|---|
| Mengenorientiert | ja | ja | ja | ? | ? | nein |
| Frequenzintervall | ja | ja | nein | nein | nein | nein |
| Zeitpunkt | ja | ja | ja | ja | ja | ? |
| Zeitintervall | ja | ja | ja | ja | ja | ? |
| Geodaten (Region) | nein | ja | ja | nein | nein | nein |
| Unschärfe | ja | ja | ja | ? | ? | nein |
| Netzorientiert | nein | nein | ja | nein | nein | nein |
| Kommunikationsstruktur | nein | nein | nein | nein | nein | nein |
| Echtzeit bei gr. Datenvolumen | nein | nein | ? | nein | nein | nein |
| Datenströme | ja [32] | ja [32] | nein | ja [67] | ja | nein |

Tabelle 4.1: Mächtigkeiten der Datenrepräsentations- und Anfragesprachen

Für die Repräsentation der Daten wird ein Datenmodell benötigt, dass die interne Datenstruktur kapselt und dem Nutzer einen Überblick gibt, welche Daten zur Verfügung stehen. Ein solcher Überblick soll ein Dienst auf Anfrage über eine Dienstbeschreibung schaffen. Dafür eignet sich keine relationale Darstellung der Daten, weil sie nicht auf einem Standard beruht und die dargestellten Daten von Nutzern unterschiedlich interpretiert werden können. Das Datenmodell soll Mengen, Zeitpunkte, Zeit- und Frequenzintervalle, Geodaten, Unschärfe, Netze und Kommunikationsstrukturen abbilden können und die entsprechende Anfragesprache soll die Anfragen bei einer großen Anzahl an Daten aus kontinuierlichen Datenströmen

[3] außer Filter Encoding und SOS

zeitnah auswerten. Die meisten Anforderungen werden von den ontologischen Sprachen nicht erfüllt. Die GML-, O&M- und SensorML-Schemata stellen die Daten in einer standardisierten Form dar. Diese Schemata sind erweiterbar und bieten den Vorteil, dass die dazu passende Abfragesprache (z.B. der Active Sensor Collection Service (ASCS)), nicht für jedes neue Modell entsprechend angepasst werden muss im Gegensatz zu XQuery.
Für die Darstellung von Frequenzintervallen und Kommunikationsstrukturen müssen die geographischen Datenrepräsentationssprachen erweitert werden. Die Anfragesprache muss auf dem Interessensprofil basieren und einen Dienst oder Dienste zur Anfrage zur Verfügung stellen. Die Ergebnisse der Anfragen sollen als Push- und/oder als Pull-Modell an die Dienstnuzter ausgespielt werden. Die existierenden geographischen Anfragesprachen unterstützen keine Kombination der beiden Modelle, sodass auch hier Ergänzungen notwendig sind.

# Kapitel 5

# Systemarchitektur

In diesem Kapitel wird eine Systemarchitektur vorgestellt, die die Anforderungen aus Abschnitt 4.4.1 erfüllt. Zunächst wird ein Überblick über die Konzeption einer Architektur geschaffen. Anschließend werden das Daten- und das Anfragemodell erläutert. Darauf aufbauend wird der Entwurf der Architektur vorgestellt, welcher eine Basis für die prototypische Implementierung in Kapitel 6 ist.

## 5.1 Überblick

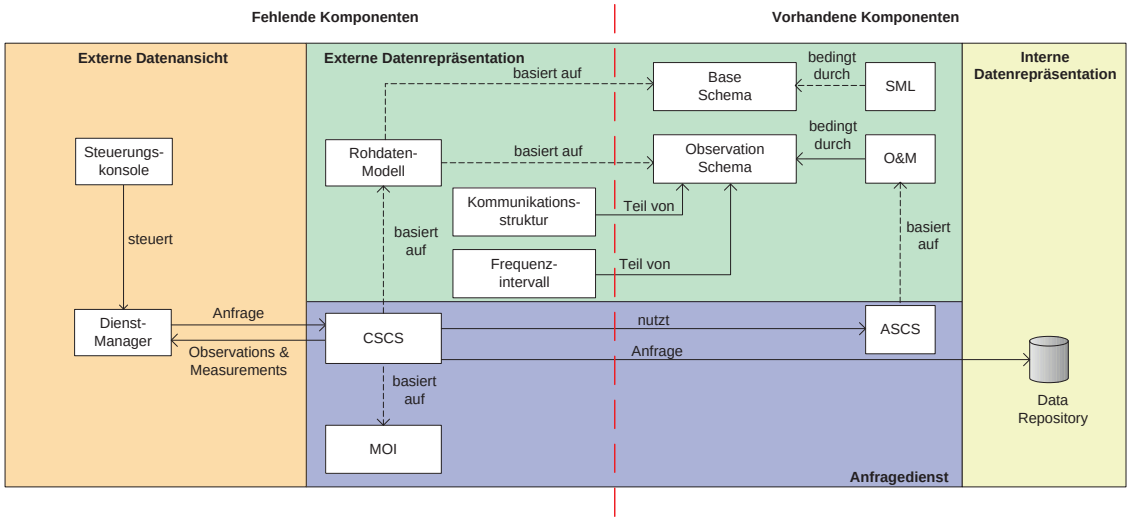


Abbildung 5.1: Architekturkonzept

Dieser Abschnitt schafft einen Überblick über das zu entwerfende Architekturkonzept. Das konzeptionelle Modell in Abbildung 5.1 zeigt welche Komponenten benötigt werden, um alle Anforderungen erfüllen zu können. Als Basis für das konzeptionelle Modell werden SensorML, O&M und der ASCS genutzt.
Die Komponenten, die hinzugefügt werden müssen, sind eine Steuerungskonsole, ein Dienstmanager, ein **C***ontinuous* **S***ensor* **C***ollection* **S***ervice (CSCS)*, ein **Mo***dell des* **I***nteressensprofils (MOI)*, ein Rohdatenmodell und die Ergänzungen der vorhandenen Datenmodelle um die Darstellung von Frequenzintervallen und Kommunikationsstrukturen.

**Steuerungskonsole**
Die Steuerungskonsole bietet dem Nutzer verschiedene Funktionalitäten an. Diese werden über den Dienstmanager gesteuert.

**Dienstmanager**
Der Nutzer kann über den Dienstmanager neue Dienste veröffentlichen (publish) oder existierende Dienste nutzen. Die Dienste können unterschiedlichen Klassen zugeordnet sein, die in Abschnitt 4.3.1 erläutert wurden.

**CSCS**
Der Continuous Sensor Collection Service ist ein Anfragedienst, der die Anfragen eines Dienstnutzers bearbeitet und die Daten gemäß des Push-Modells, des Pull-Modells oder gemäß der Kombination der beiden Modelle ausspielt. Dieser Dienst ist eine Erweiterung des ASCS, die benötigt wird, da der ASCS die Kombination der beiden Modelle nicht unterstützt. Der CSCS basiert auf dem MOI aus Abschnitt 4.4 sowie auf dem Rohdatenmodell aus Abschnitt 4.2. Zur Beantwortung der Anfragen werden die Daten aus dem Data Repository geholt.

**Rohdatenmodell**
Das Rohdatenmodell nutzt die Spezifikation von O&M (`Observation.xsd`) inklusive der Erweiterung um die Darstellung von Frequenzintervallen und Kommunikationsstrukturen und SensorML (`Base.xsd`).

Diese Komponenten existieren nicht und werden zur Erfüllung gegebener Anforderungn modelliert.

**Architekturkonzept**
Die Architektur basiert auf dem Publish/Subscribe Paradigma, die eine schnell bedarfsgerechte (inhaltsbasierte) und relevante Sensordatenversorgung unterstützt. Die Steuerungskonsole gibt den Nutzer die Möglichkeit verschiedene Dienste zu nutzen, die vom Diensterbringer veröffentlicht wurden. Die Steuerungskonsole steuert einen Dienstmanager an, der entsprechend den Angaben in der Steuerungskonsole die Veröffentlichung von neuen Diensten, die Registrierung von Anfragen und die Abfrage vorhandener Dienste vornimmt.
Ein Informationsdienst liefert auf Anfrage die Metadaten angebotener Dienste und der Anfragedienst stellt Funktionen für die Anfrage eines Informationsdienstes bzw. Registrierung der Anfragen zur Verfügung. Die Funktionen erlauben die Abfrage nach Rohdaten gemäß eines Push- oder einem Pull-Modells oder der Kombination der beiden Modelle. Der CSCS basiert auf dem Rohdatenmodell aus Kapitel 4.2 und bildet den MOI der Nutzer ab. Die Funktionalitäten unterstützt der ASCS nicht und wird dahingehend erweitert. Die Anfragen sind vom Typ 2, da ein Dienstnutzer nur parametrisierte Anfragen stellen kann.
Die Rohdaten aus dem Data Repository werden mittels des Rohdatenmodells extern repräsentiert. Es beschreibt geographische Daten, Zeitpunkte, Zeitintervalle, Zeitreihen, Frequenzintervalle, Verfahren, Rufzeichen, Netze und Kommunikationsstrukturen. Da die Anfragestruktur von der der internen Datenrepräsentation abweicht, wird ein Wrapper benötigt, der eine Sprachtransformation von der externen in die interne Datenstruktur durchführt, damit die Daten aus dem Data Repository abgefragt werden können. Die Daten liegen als relationale Tupel in dem Data Repository vor, sodass eine Transformation von XML nach SQL eingesetzt wird. Die Ergebnisse in SQL liegen in Form eines *ResultSets* vor und werden an den

Anfragedienst weitergeleitet, der den Nutzern situationsbezogene Sensordaten ausspielt. Die beschriebene Architektur wird in Abbildung 5.2 illustriert.

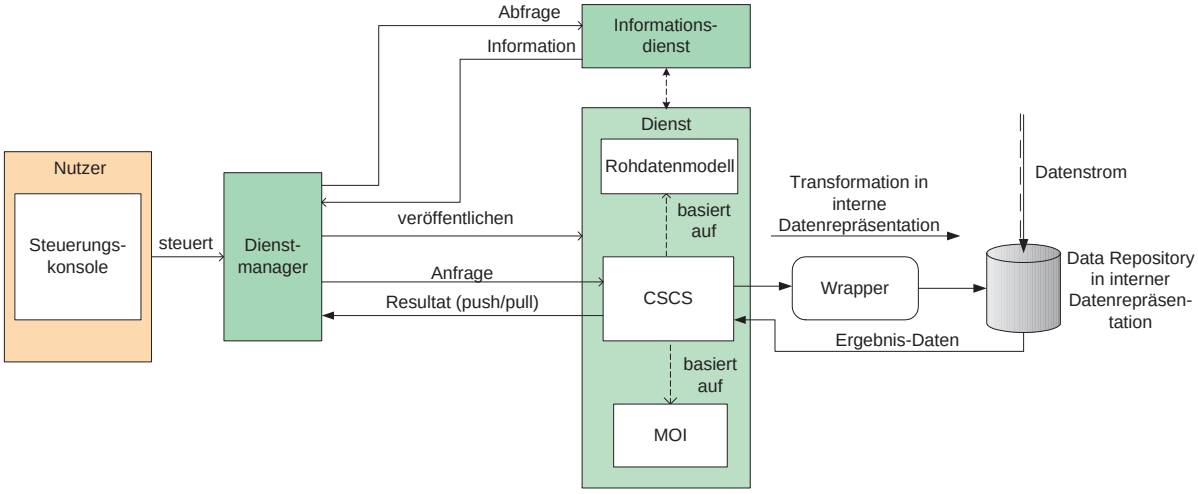


Abbildung 5.2: Konzeption einer Publish/Subscribe Architektur

## 5.2 Das Rohdatenmodell

Das Rohdatenmodell dient zur Repräsentation der Sensordaten. Es bildet das konzeptionelle Modell der Rohdaten aus Abschnitt 4.2.1 ab. Die Rohdaten bilden eine Menge von Elementen, deren Beziehung in einer hierarchischen Struktur abgebildet wird. Die Spezifikation des Datemodells (SchemaSensor.xsd) befindet sich im Anhang A.

### 5.2.1 Beschreibung des Sensors

Das Data Repository wird als ein Element mit der Bezeichnung *Sensor* modelliert, weil dieses Data Repository die Messwerte der Sensoren beinhaltet und Metadaten besitzt. Die Metadaten beschreiben den Sensor mit `sml:description`, `sml:characteristics` und `sml:contact` (siehe Abbildung 5.3). Die Menge der Messdaten werden mit `ObservationCollection` abgebildet.
Die Spezifikation basiert auf SensorML und O&M, die jeweils auf die Spezifikation GML 3.1.1 aufbauen[1].

### 5.2.2 Repräsentation der Messwerte

`ObservationCollection` beinhaltet eine Menge von Messwerten. Jede `ObservationCollection` ist einem bis unendlich vielen `ObservationMember` zugeordnet, welches eine weitere Menge von Messwerten abbildet. Jeder `ObservationMember` wird weiteren `ObservationCollections` zugeordnet, welches eine Eigenschaft beschreibt, welche zu einer Feature-Instanz gehört `boundedBy`. Die von eins bis unendlich anzugebenden `ObservationMembers` besitzen eine `Observation` (siehe Abbildung 5.4).

[1]O&M basiert auf GML 3.0, daher wurde diese Spezifikation nachmodelliert, sodass sie auch mit GML 3.1.1 harmonisiert

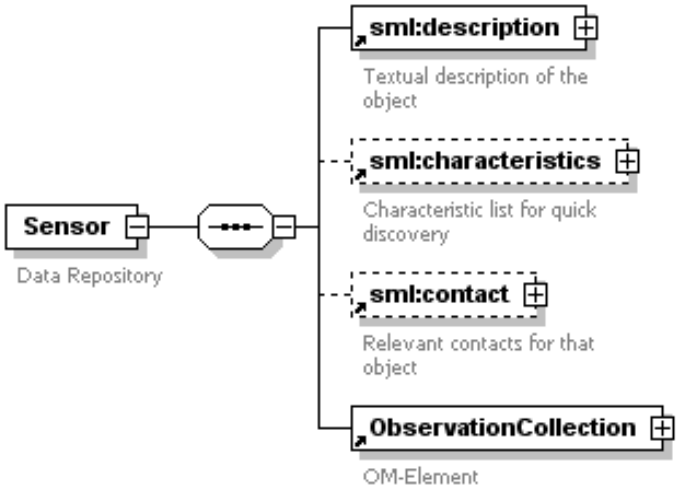


Abbildung 5.3: Schematische Darstellung des Data Repositories

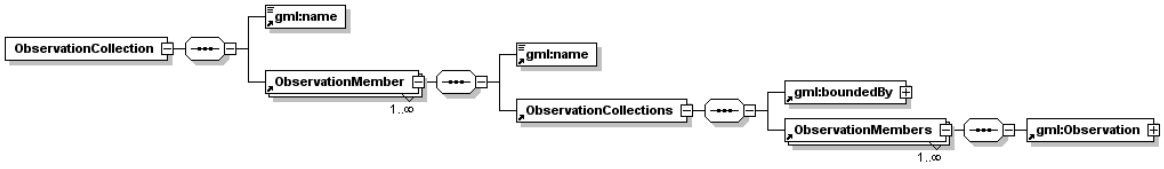


Abbildung 5.4: Schematische Darstellung von `ObservationCollection`

Jede Observation ist ein Element, welches die Rohdaten aus dem konzeptionellen Modell abbildet.

### Geodaten und Zeitreihe

In der GML Version 3.0 ist der `timeStamp` ein Element von `Observation` (siehe Abbildung 4.8). In der GML Version 3.1.1 ist das Element `timeStamp` durch `validTime` ersetzt worden, welches mehr Darstellungsmöglichkeiten von Zeiten mit oder ohne geographischen Angaben bereitstellt.
Das Element `boundedBy` bildet ein Rechteck mit einer Zeitangabe ab. Damit werden die Ortungspunkte bzw. -rechtecke, die einen Ort oder eine Region beschreiben, abgebildet sowie die zugehörige erfasste Zeit. Eine `Observation` ist eine Menge von `Observations`, sodass sich eine Zeitreihe aus der verschachtelte Struktur ergibt.

### Zeitintervall

Wie in Abschnitt 5.2.2 beschrieben, gibt es unterschiedliche Darstellungsmöglichkeiten einer Zeit. Das Rohdatenmodell bildet neben der Darstellung von Zeitreihen auch Zeitintervalle in folgender Form ab:

```
<gml:validTime>
  <gml:TimePeriod>
    <gml:relatedTime relativePosition="Begins"></gml:relatedTime>
    <gml:relatedTime relativePosition="Ends"></gml:relatedTime>
    <gml:beginPosition>2005-12-02T08:00:00</gml:beginPosition>
    <gml:endPosition>2005-12-02T08:02:00</gml:endPosition>
  </gml:TimePeriod>
</gml:validTime>
```

Ein Zeitintervall wird durch zwei Zeitpunkte abgebildet. Die Startzeit (`relativePosition`="`Begins`" und `beginPosition`) ist ein Element von `gml:TimePeriod`, welches zur Menge `gml:validTime` gehört. Die Endzeit (`relativePosition`="`Ends`" und `endPosition`) ist auch ein Element von `gml:TimePeriod`.

### Frequenzintervall

Das Frequenzintervall wird als positives Integerintervall abgebildet (siehe Abbildung 5.5).

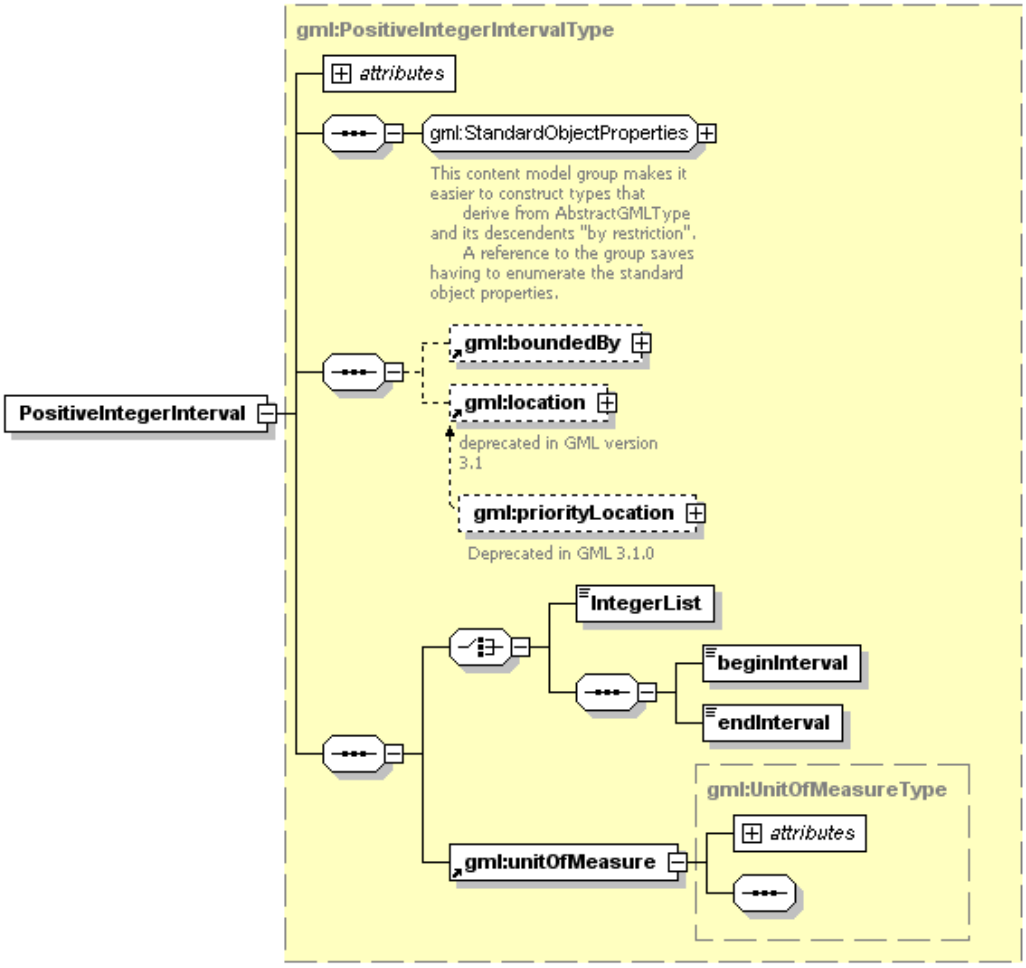


Abbildung 5.5: Schemadarstellung eines positiven Integerintervalls

Wie die meisten Features wird auch dieses Modell durch standardisierte Eigenschaften (*StandardObjectProperties*) beschrieben. Ortungsangaben können optional mit `boundedBy` oder `location` angegeben werden. Einem positiven Integerintervall werden eine Integerliste oder eine Intervallangabe und die Einheit der Integer angegeben.
Das vorhandene Datenmodell bietet eine solche Darstellung nicht an, daher wurde das GML-Schema *gmlbase.xsd* um dieses Modell erweitert wurde.

### Verfahren und Rufzeichen

Darstellung durch Symbole Das Symbolelement `symbol` ist ein Element eines Features und beschreibt graphische Attribute graphischer Objekte ohne eine partikuläre oder implizite Bedeutung zu besitzen. Es kann die Beschreibung einer Linie, eines Kreises, eines Polygons oder etwas komplexeres sein. Für das Datenmodell wurde dieses Element dazu verwendet die Verfahrensart darzustellen. Jedem Symbol kann man über das Attribut `symbolType` den Typ des Symbols z.B. Sprache angeben.

```
<gml:symbol symbolType="Sprache"></gml:symbol>
```

In dem Schema *defaultStyle.xsd* unter `SymbolTypeEnumeration` wurden die Symbolnamen „Sprache“ und „Datenübertragung“ hinzugefügt.

```
<simpleType name="SymbolTypeEnumeration">
  <annotation>
   <documentation>
      Used to specify the type of the symbol used.
    </documentation>
  </annotation>
 <restriction base="string">
 <enumeration value="Sprache"/>
 <enumeration value="Datenübertragung"/>
 </restriction>
</simpleType>
```

### Netz- und Kommunikationsstruktur

Charakteristika von Netzen sind:

- **Topologie**: Darunter versteht man die räumliche Anordnung der Elemente [46].
- **Dynamik**: *Statische Netze* (z.B. Gitter) verbinden die Module nach einem festen Muster. Jeder Verbindungskanal liegt genau zwischen zwei Modulen. In *dynamischen Netzen* können zu verschiedenen Zeitpunkten unterschiedliche Verbindungsmuster realisiert sein [46].

GML bietet für die Repräsentation von Topologien das entsprechende Schema *Topology.xsd* an. Statische Netze lassen sich in GML mit dem Schema *grid.xsd* abbilden. Bei der Firma Plath GmbH können zu verschiedenen Zeitpunkten unterschiedliche Verbindungsmuster auftreten. Es wird hier also eine Abbildung von *dynamischen Netzen* benötigt, die durch Referenzen einzelner `Observations` dargestellt wird.
Die Referenzen einzelner Observation bilden jedoch nur eine Instanz eines Graphen ab, was den Unterschied zur Kommunikationsstruktur ausmacht. Denn eine Kommunikationsstruktur

| Emission | Beziehung zu |
|---|---|
| $E1$ | $E7$, $E11$ |
| $E2$ | — |
| $E3$ | — |
| $E4$ | — |
| $E5$ | — |
| $E6$ | — |
| $E7$ | $E1$, $E11$ |
| $E8$ | — |
| $E9$ | — |
| $E10$ | — |
| $E11$ | $E1$ |
| $E12$ | — |
| $E13$ | — |
| $E14$ | — |
| $E15$ | — |
| $E16$ | — |
| $E17$ | — |
| $E18$ | — |

Tabelle 5.1: Emissionsliste

bildet das gesamte Netz ab. Der Unterschied zwischen einer Netzstruktur und einer Kommunikationsstruktur wird anhand des folgenden Beispiels noch mal erläutert:
In Abbildung 5.6 werden achtzehn erfasste Emissionen ($E1$ bis $E18$) abgebildet. Diese Emis-

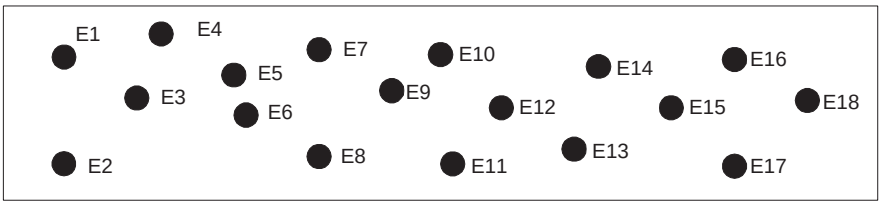


Abbildung 5.6: Emissionen

sionen haben eine Beziehung zu anderen Emissionen, welche in einer Emissionsliste aufgelistet werden:
Emission 1 hat eine Beziehung zu Emission 7 und Emission 11. Emission 7 hat eine Beziehung zu Emission 1 und Emission 11. Emission 11 hat eine Beziehung zu Emission 1. Die restlichen Emissionen haben keine Beziehungen zu anderen Emissionen.
Das Ergebnis einer Anfrage nach gegebenen Netzstrukturen könnte so eine Emissionsliste sein. Die Emissionsliste bildet nur eine Instanz eines Graphen ab. Ziel ist jedoch herauszufinden, welcher gerichteter Graph die erfassten Emissionen abbilden. Bei einer Kommunikationsstruktur werden verschiedene Emissionen einer Menge zugeorndet. Eine Menge ist ein Teilnehmer, der Signale sendet. Diese Mengen stehen zueinander in Beziehung. **Beispiel**:
Bodenstation 1 sendet Funksignale an Bodenstation 2. Bodenstation 2 antwortet Bodenstation 1. Es entsteht also eine Menge von Emissionen aus Bodenstation 1 und eine Menge von Emissionen von Bodenstation 2. Beide Mengen stehen zueinander in Beziehung. Analog gilt

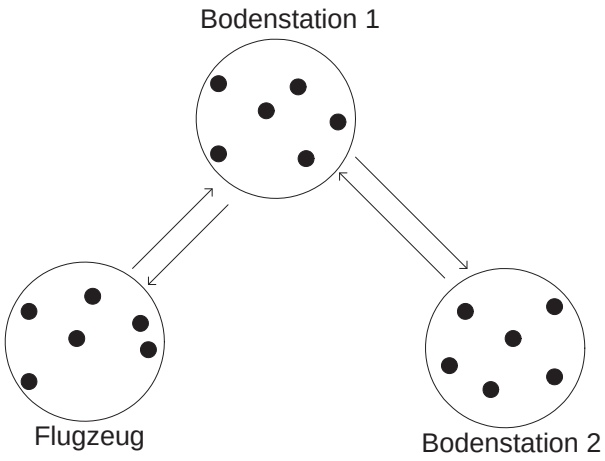


Abbildung 5.7: Kommunikationsstruktur

dieses Szenario, wenn Bodenstation 1 Signale an ein Flugzeug sendet und dieser der Bodenstation antwortet.

Eine Netzstruktur ist ein gerichteter Graph.

**Definition 5.1** *Ein* **Graph** *$G$ ist ein geordnetes Paar $(V, E)$ aus einer Menge $V$ von* **Knoten** *und einer Menge $E$ von* **Kanten**. *Wird jedem Element von $E$ ein geordnetes Paar zugeordnet, dann spricht man von einem* **gerichteten Graphen** [8]. *In der graphischen Darstellung erscheinen Knoten der Graphen als Kreise und die gerichteten Kanten als Pfeile.*

Eine Kommunikationsstruktur ist ebenfalls ein gerichteter Graph , wobei die Knoten Mengen sind, die Elemente besitzen.

**Definition 5.2** *Eine* **Menge** *$A$ ist eine Zusammenfassung bestimmter, wohlunterschiedener Objekte $a$ unserer Anschauung oder unseres Denkens zu einem Ganzen. Diese Objekte heißen* **Elemente** *der Menge* [8].

**Definition 5.3** *Ein* **Kartesisches Produkt zweier Mengen** *$A \times B$ ist durch $A \times B = (a, b)|a \in A \wedge b \in B$ definiert* [8].

**Definition 5.4** *Die Elemente $(a, b)$ von $A \times B$ heißen* **geordnete Paare** *und sind charakterisiert durch $(a, b) = (c, d) \Leftrightarrow a = c \wedge b = d$* [8].

Eine Relation in einer Kommunikationsstruktur lässt sich auf folgende Weise definieren:

**Definition 5.5 Relationen** *beschreiben Beziehungen zwischen den Elementen einer oder verschiedener Mengen. Die Kommunikationsstruktur mit $n$ Mengen ist eine $n$-stellige Relation $R$. $R$ ist zwischen den Mengen $A_1, ...A_n$ ist eine Teilmenge des kartesischen Produkts dieser Mengen, d.h. $R \subseteq A_1 \times ... \times A_n$. Sind die Mengen $A_i, i = 1, ..., n$, sämtlich gleich der Menge $A$, so wird $R \subseteq A^n$ und heißt $n$-stellige Relation in der Menge $A$* [8].

Für die Beziehung der Elemente einer Netzstruktur gilt diese Relationsdefinition analog.

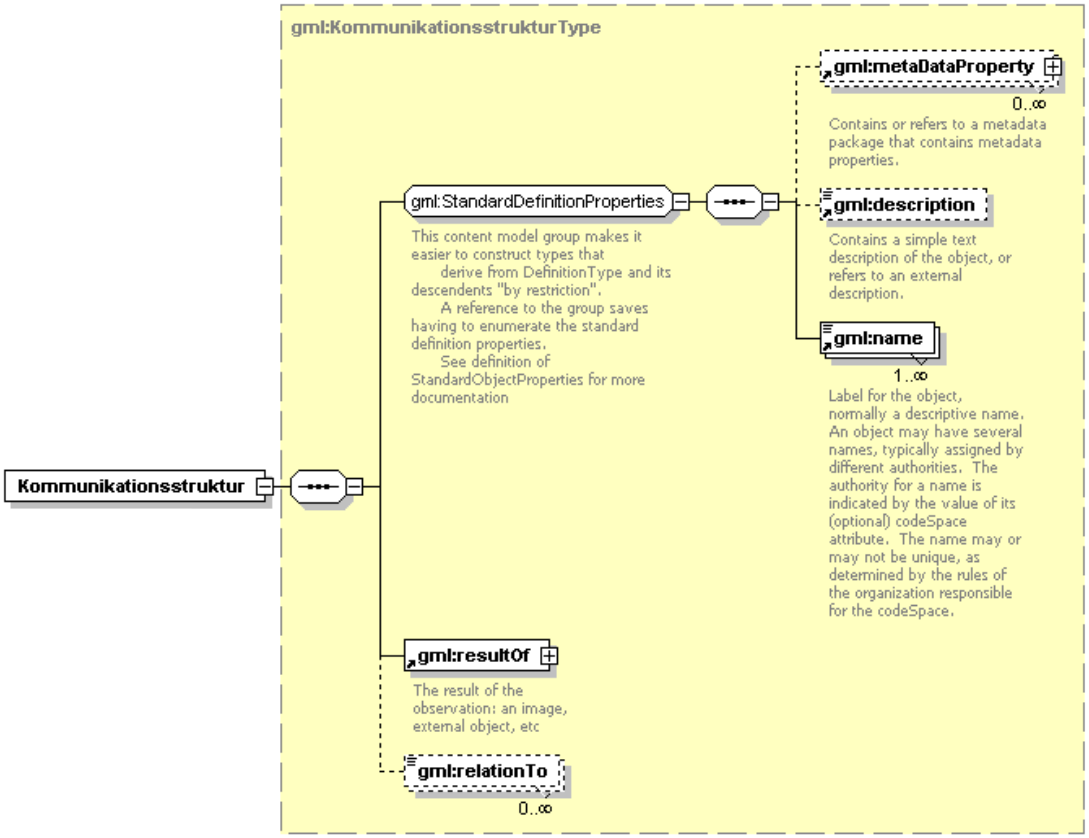


Abbildung 5.8: Schemadarstellung einer Kommunikationsstruktur

Für das Datenmodell wurde eine Kommunikationsstruktur modelliert und in das GML-Schema *gmlbase.xsd* integriert. Das Schema der Kommunikationsstruktur ist in Abbildung 5.8 illustriert.

Die Kommunikationsstruktur wird durch standardisierte Eigenschaften (*StandardObjectProperties*) beschrieben. Die Messergebnisse werden über das Element (`resultOf`) z.B. für Zeit, Frequenzintervalle und Ortung angegeben. Die Beziehungen werden durch Kanten abgebildet, die durch ein bestimmtes Verfahren beschrieben werden.

## 5.3 Das Anfragemodell

Das Anfragemodell stellt verschiedene Dienstfunktionen für den Nutzer zur Verfügung, um zu Erfahren welche Dienste zur Verfügung stehen, um sein Interessensprofil zu definieren und um seine Anfragen zu stellen bzw. zu registrieren (subscribe). Das Abfragemodell basiert auf dem konzeptionellen Modell des Interessensprofils (MOI) aus Kapitel 4.2.2 und wird als *Continuous Sensor Collection Service (CSCS)* bezeichnet. Der CSCS spielt die Sensordaten kontinuierlich an die Nutzer als Push- und/oder Pull-Modell aus. Im nächsten Abschnitt werden die erforderlichen Dienstfunktionen erläutert. Im Anschluss daran wird der Entwurf der Architektur beschrieben.

### 5.3.1 Dienstfunktionen

Im nun folgenden Abschnitt werden die verschiedenen Funktionen beschrieben, die in dem Bereich der Anfrageauswertung relevant sind.

Sei im Folgenden $\mathcal{F}$ die Menge aller Funktionen, $\mathcal{D}$ die Menge aller Dienste, $\mathcal{A}$ die Menge aller Sensor- und Messdaten, $\mathcal{M}$ die Menge aller Metadaten und *Präd* die Menge aller Prädikate.

Zur Erfüllung der Anforderungen werden folgende Funktionen benötigt:

$$\mathcal{F}_1 : \mathcal{D} \rightarrow \mathcal{M}_{\mathcal{D}}$$

$\mathcal{F}_1$ erzeugt Metadaten über die angebotenen Dienste. So erfährt ein Nutzer, welche Daten zur Abfrage zur Verfügung stehen.

$$\mathcal{F}_2 : \mathcal{D} \times Präd \rightarrow \mathcal{A}$$

$\mathcal{F}_2$ liefert die Menge aller Messdaten eines Dienstes, die das Prädikat erfüllen. D.h. Es werden dem Nutzer nur die Messdaten ausgespielt, die eine vorgegebene Bedingung erfüllen. Das Ausspielen der Daten erfolgt einmalig mit

$$\mathcal{F}_{2a} : \mathcal{D} \times Präd \overset{Push}{\rightarrow} \mathcal{A}$$

oder kontinuierlich mit

$$\mathcal{F}_{2b} : \mathcal{D} \times Präd \overset{Pull}{\rightarrow} \mathcal{A}.$$

Der Push-Modell Algorithmus:

```
DIENSTNUTZER                          DIENSTERBRINGER
subscribeQuery(IP);
                                      subscriptionID = subscribeQuery(IP);
                                      return subscriptionID;
                                      while( subscriptionID exists )
                                        if( match(Data,Query) )
                                          push(ResultSet);

unsubscribeQuery(subscriptionID);
```

Ein Dienstnutzer registriert sich für eine Anfrage mit `subscribeQuery` und übergibt mit dieser Funktion sein Interessensprofil (IP). Der Diensterbringer erhält diese Anfrage und weist dieser Anfrage eine `subscriptionID` zu und sendet sie an den Dienstnutzer. Solange die Anfrage registriert ist, wird mit der Funktion *match* überprüft, ob die kontinuierlichen Eingangsdaten zu der Anfrage passen (match(Data,Query)). Wenn ja, werden die Ergebnisse *ResultSet* an die Dienstnutzer ausgespielt. Ist der Nutzer nicht mehr an seine Anfrage interessiert, kann er seine Registrierung mit der Funktion `unsubscribeQuery` aufheben.

Der Pull-Modell Algorithmus:

```
DIENSTNUTZER          DIENSTERBRINGER
sendQuery(IP);
                      if( match(Data,Query) )
```

```
    pull(ResultSet);
  else
    emptySet();
```

Ein Dienstnutzer stellt seine Anfrage mit `sendQuery(IP)` und übergibt mit dieser Funktion sein Interessensprofil (IP). Mit der Funktion `match` wird überprüft, ob die Daten im Data Repository zu der Anfrage passen (match(Data,Query)). Wenn ja, werden die Ergebnisse *ResultSet* an die Dienstnutzer ausgespielt. Andernfalls wird der Dienstnutzer informiert, dass keine passenden Daten vorhanden sind.

$\mathcal{F}_3$ liefert alle Messwerte, die aus verschiedenen Diensten stammen. Es gibt Anfragen, für die erst mit dem Einsatz zweier Dienste beantwortet werden kann. Für solche Anfragen muss also gelten:

$$\mathcal{F}_3 : \mathcal{D}_1 \times \mathcal{D}_2 \rightarrow \mathcal{A}.$$

Der Continuous Sensor Collection Service bieten solche Funktionen an, die nun vorgestellt werden.

| Funktion | Beschreibung |
|---|---|
| GetCapabilities | Beschreibung des Dienstes |
| DescribePlatform | Anfrage von Plattform-Metadaten |
| DescribeSensor | Abfrage von Sensor-Metadaten |
| SubscribeObservation | Registrierung raumzeitvarianter Messdaten |
| UnsubscribeObservation | Registrierung löschen |
| GetObservation | Abfrage raumzeitvarianter Messdaten |
| SubscribeContinuousEvent | Abfrage und Registrierung von Messdaten |
| UnsubscribeContinuousEvent | Abfrage und Registrierung von Messdaten löschen |

Tabelle 5.2: Dienstfunktionen des CSCS

### GetCapabilities

Der CSCS stellt die `GetCapabilities` Operation zur Verfügung, um Metadaten über Dienste zu liefern. Dabei werden die in der Tabelle 5.3 aufgelisteten Parameter übergeben. Ob die Angabe eines Parameters optional oder notwendig ist, erkennt man an den Buchstaben O für optional und R für notwendig (engl.: required).

| Parameter | Beschreibung |
|---|---|
| REQUEST=GetCapabilities (R) | Name der Anfrage |
| SERVICE (R) | Bezeichnung des Servicetyps |
| VERSION (O) | Version des Dienstes |

Tabelle 5.3: Parameter der `GetCapabilites` Operation

Für GetCapabilities gilt also:

**Definition 5.6** *getCapabilities:* $\mathcal{D} \rightarrow \mathcal{M}_{\mathcal{D}}$.

### SubscribeObservation

Die `SubscribeObservation` Funktion ermöglicht die Registrierung von raumzeitvarianten Messdaten. In der Tabelle 5.4 sind dessen Überparameter aufgelistet.

| Parameter | Beschreibung |
|---|---|
| SERVICE (R) | Bezeichnung des Servicetyps |
| VERSION (O) | Version des Dienstes |
| BoundingBox (R) | Raumausschnitt der Messdaten |
| platformID (O) | Einschränkung der Messdaten bzgl. der Plattform |
| sensorID (O) | Einschränkung der Messdaten bzgl. des Sensors |
| ogc:Filter (O) | Messdatenfilterung nach vorgegebenen Bedingungen |
| CallbackServerURL (R) | Adresse, an der die Daten ausgespielt werden |
| Period OR MaxAllowedDelay (R) | Zeit- oder ereignisbasierte Messdaten |

Tabelle 5.4: Parameter der `SubscribeObservation` Operation

Es gilt:

**Definition 5.7** *subscribeObservation:* $\mathcal{D} \times Präd \overset{push}{\rightarrow} \mathcal{A}$.

### UnsubscribeObservation

Die `UnsubscribeObservation` Funktion löscht eine mit der Funktion `SubscribeObservation` erstellte Registrierung.

| Parameter | Beschreibung |
|---|---|
| SERVICE (R) | Bezeichnung des Servicetyps |
| VERSION (O) | Version des Dienstes |
| SubscriptionID (R) | Angabe, welche Registrierung gelöscht werden soll |

Tabelle 5.5: Parameter der `UnsubscribeObservation` Funktion

### GetObservation

Die `GetObservation` Funktion erlaubt eine Abfrage nach raumzeitvarianten Messdaten. Die Übergabeparameter sind in der Tabelle 5.6 enthalten.

| Parameter | Beschreibung |
|---|---|
| SERVICE (R) | Bezeichnung des Servicetyps |
| VERSION (O) | Version des Dienstes |
| BoundingBox (R) | Raumausschnitt der Messdaten |
| time (O) | Einschränkung der Messdaten bzgl. Zeit |
| platformID (O) | Einschränkung der Messdaten bzgl. der Plattform |
| sensorID (O) | Einschränkung der Messdaten bzgl. des Sensors |
| ogc:Filter (O) | Messdatenfilterung nach vorgegebenen Bedingungen |

Tabelle 5.6: Parameter der `GetObservation` Operation

**Definition 5.8** *getObservation:* $\mathcal{D} \times Präd \overset{Pull}{\rightarrow} \mathcal{A}$.

### DescribePlatform

Die optionale `DescribePlatform` Operation erlaubt die Abfrage nach plattformspezifischen Metadaten. Als Parameter wird eine Liste von `TypeName`s, welche die Plattformidentifikatoren enthält, der Service und die Version übergeben. Der Parameter `outputFormat` kann optional angegeben werden.

| Parameter | Beschreibung |
|---|---|
| TypeName (R) | Plattformspezifische Metadaten |
| SERVICE (R) | Bezeichnung des Servicetyps |
| VERSION (O) | Version des Dienstes |
| outputFormat (O) | Ausgabeformat der Metadaten |

Tabelle 5.7: Parameter der `DescribePlatform` Operation

**Definition 5.9** *describePlatform:* $\mathcal{D} \rightarrow \mathcal{M}_{\mathcal{P}}$.

### DescribeSensor

Die optionale `DescribeSensor` Operation erlaubt die Abfrage nach sensorspezifischen Metadaten. Als Parameter wird eine Liste von `TypeName`s, welche die Sensoridentifikatoren enthält, der Service und die Version übergeben. Der Parameter `outputFormat` kann optional angegeben werden.

| Parameter | Beschreibung |
|---|---|
| TypeName (R) | Sensorspezifische Metadaten |
| SERVICE (R) | Bezeichnung des Servicetyps |
| VERSION (O) | Version des Dienstes |
| outputFormat (O) | Ausgabeformat der Metadaten |

Tabelle 5.8: Parameter der `DescribeSensor` Operation

**Definition 5.10** *describeSensor:* $\mathcal{D} \rightarrow \mathcal{M}_{\mathcal{S}}$.

Mit den ASCS Funktionen können keine zeit- und ereignisbasierten Sensordaten kontinuierlich ausgespielt werden. Dafür wird eine neue Funktion benötigt, der das Push- und das Pull-Modell miteinander kombiniert. Eine solche Funktion stellt der CSCS zur Verfügung.

### SubscribeContinuousEvent

Die `SubscribeContinuousEvent` Funktion liefert raumzeitvariante Messdaten gemäß des Push- und des Pull-Modells. In der Abbildung 5.9 wird das Schema von `SubscribeContinuousEvent` illustriert, welches die Parameter `BoundingBox`, `platformID`, `sensorID`, `ogc:Filter`, `CallbackServerURL`, `Period` oder `MaxAllowedDelay` und `QueryReturn` besitzt.
Die Parameter Service (required) und Version (optional) werden auch angegeben, welche jedoch nicht in der Abbildung zu sehen sind. Über den `BoundingBox` Parameter wird der Raumausschnitt der Messdaten definiert. Die Parameter `platformID` und `sensorID` sind optional und schränken die Anfrage bzgl. der Plattform und des Sensors ein. Sollen die Messdaten erst

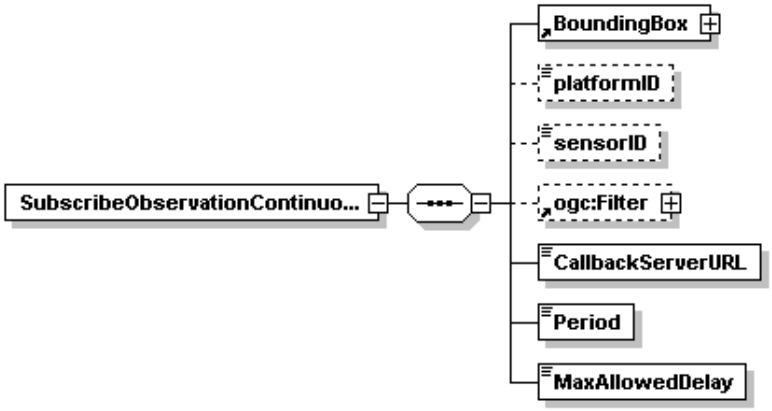


Abbildung 5.9: Schematische Darstellung von `SubscribeContinuousEvent`

| Parameter | Beschreibung |
|---|---|
| SERVICE (R) | Bezeichnung des Servicetyps |
| VERSION (O) | Version des Dienstes |
| BoundingBox (R) | Raumausschnitt der Messdaten |
| platformID (O) | Einschränkung der Messdaten bzgl. der Plattform |
| sensorID (O) | Einschränkung der Messdaten bzgl. des Sensors |
| ogc:Filter (O) | Messdatenfilterung nach vorgegebenen Bedingungen |
| CallbackServerURL (R) | Adresse, an der die Daten ausgespielt werden |
| Period (R) | Zeitbasierte Messdaten |
| MaxAllowedDelay (R) | Ereignisbasierte Messdaten |

Tabelle 5.9: Parameter der `SubscribeContinuousEvent` Operation

nach einer erfüllten Bedingung ausgespielt werden, kann diese über den `ogc:Filter` Parameter angegeben werden, welcher im Abschnitt 3.1.7 kurz erläutert wurde.

Mit diesem Filter können räumliche, logische oder Vergleich-Anfragen gestellt werden. Gegenüber die eigene Definition von Filtern (`wrs:Query`), wie es in der Spezifikation des konventionellen SCS vorgeschlagen wird (siehe Abschnitt 4.4.4), bietet dieser Filter den Vorteil, einem Standard des OpenGIS zu folgen.
Mit der Spezifikation der Funktion können damit alle Beispielanfragen beantwortet werden.

Über den `CallbackServerURL` Parameter wird die Adresse des Web Services angegeben, an den die registrierten Messdaten versandt werden. Mit der Operation `SubscribeObservation` aus [61] kann man nur eine Auswahl zwischen zeitbasierten (`Period`) und ereignisbasierten (`MaxAllowedDelay`) Subskriptionen treffen. Es wird jedoch gefordert, dass ein Nutzer einen kontinuierlichen Ausgangsdatenstrom erhält, der sich sowohl aus historischen als auch aus aktuellen Daten zusammensetzen kann und wenn dieser kontinuierliche Datenstrom einen auffälligen Messwert beinhaltet. Diese Operation ermöglicht dies durch die Registrierung zeit- **und** ereignisbasierter Sensordaten.

**Definition 5.11** *SubscribeContinuousEvent:* $\mathcal{D} \times \mathit{Präd} \rightarrow \mathcal{A}$.

**UnsubscribeContinuousEvent**

Die `UnsubscribeContinuousEvent` Funktion löscht eine Registrierung, die mit der Operation `SubscribeContinuousEvent` erstellt wurde. Die `UnsubscribeContinuousEvent` Operation besitzt den Parameter `SubscriptionID`, der dann beim Aufruf dieser Operation gelöscht wird sowie die Parameter Service und Version.

| Parameter | Beschreibung |
|---|---|
| SERVICE (R) | Bezeichnung des Servicetyps |
| VERSION (O) | Version des Dienstes |
| SubscriptionID (R) | Angabe, welche Registrierung gelöscht werden soll |

Tabelle 5.10: Parameter der `UnsubscribeContinuousEvent` Operation

## 5.4 Wrapper Spezifikation

Die externe Repräsentation der Rohdaten und das Interessensprofil basieren auf XML und die Rohdaten stehen in der internen Datenrepräsentation als relationale Tupeln zur Verfügung. Für die Abfrage der Rohdaten aus dem Data Repository wird daher ein Wrapper eingesetzt, welcher einfach aus einem XML-Dokument SQL-Anfragen generiert, die auf der relationalen Datenbank ausgeführt werden. Die Spezifikation eines XML2SQL Wrappers wird ein *Application Programming Interface (API)* zur Verarbeitung und evtl. Erzeugung von XML-Dokumenten eingesetzt.
Es gibt zwei Parsertypen mit denen man auf XML Daten zugreifen kann: Der SAX (Simple API for XML)- und der DOM (Document Object Model )-basierende Parser [35]. Ein *SAX-Parser* liest ein XML Dokument ein und erzeugt Events an den relevanten Stellen z.B. der Beginn eines XML-Tags. Die dokumentenspezifische Anwendungslogik befindet sich in dem Event-Handler, der vom Parser eingebunden wird und arbeitet die erzeugten Events ab. Ein *DOM-Parser* erzeugt aus einem XML Dokument einen DOM-Baum im Speicher. Die Repräsentation des XML Dokumentes als Baum eignet sich bei XML, weil deren Struktur ebenfalls ein Baum ist. Elemente in einer hierarchischen Struktur lassen sich einfach abfragen oder manipulieren, hat jedoch bei einem wahlfreien Zugriff den Nachteil viel Speicher zu verbrauchen. Das DOM-Modell ist sprachen- und plattformabhängig und bietet bei einer plattformunabhängigen Sprachen wie z.B. Java nicht die volle Funktionalität [lt. [19]].
SAX und DOM sind *Low-Level-APIs*, d.h. sie operieren sehr nach an der Maschinensprache auf einem niedrigen Level. Das XML Data Binding ist eine *High Level API* und schafft die direkte Verbindung zwischen Komponenten eines XML Dokumentes und den Objekten einer Programmiersprache. Ein Objekt enthält alle Daten eines XML Dokumentes, mit denen nun ein SQL Statement gebildet werden kann, um die entsprechenden Daten aus dem Data Repository zu erhalten.

## 5.5 Architekturentwurf

In den drei Abbildungen 5.10, 5.11 und 5.12 werden Use Cases dargestellt in einem teilweise detailierten Aktivitätsdiagramm. Sie stellen alle Dienstfunktionen vor, die für eine Anfragebeantwortung zum Einsatz kommen.
Eine zentrale Anforderung an einem Dienst ist, dass ein Nutzer sich für Daten registrieren kann, die Registrierung gespeichert wird, die Daten an den Nutzer kontinuierlich ausgespielt

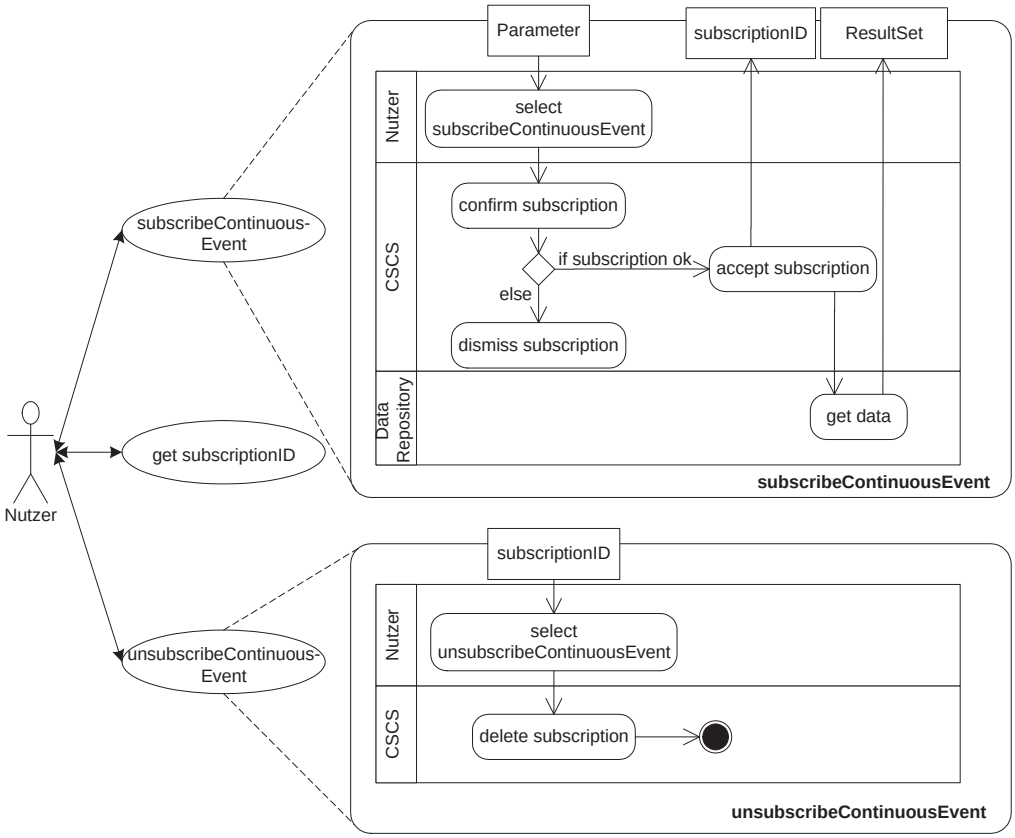


Abbildung 5.10: Use Case für die Registrierung für zeit- und ereignisbasierte Messdaten in einem detaillierten Aktivitätsdiagramm

werden, die dabei einer vom Nutzer definierten Bedingung unterliegen. Hierfür wird eine Registrierungsschnittstelle bereitgestellt, die einen Vertrag zwischen Nutzer und dem Dienst bildet. Der Dienst liefert an den Nutzer solange kontinuierlich entsprechende Daten bis er seine Registrierung aufhebt (unsubscribe). Dabei werden vier verschiedene Anfragearten unterschieden.

- Der Nutzer kann zeit- und ereignisbasierte Daten erhalten. Da werden Daten kontinuierlich an den Nutzer versandt und wenn ein Ereignis auftritt, wird der Nutzer über die Funktion `notify` seitens des Diensterbringers darüber informiert (Abbildung 5.10).

- Der Nutzer erhält nur zeitbasierte Daten. D.h. Daten werden periodisch an den Nutzer ausgespielt (GetObservation in Abbildung 5.12).

- Der Nutzer erhält nur ereignisbasierte Daten. D.h. Daten werden nur versandt, wenn ein bestimmtes Ereignis aufgetreten ist (Abbildung 5.11).

- Der Nutzer erhält Metadaten über den Dienst, den Sensor oder die Plattform (Abbildung 5.12).

Detailliertere Beschreibung der Abbildungen:

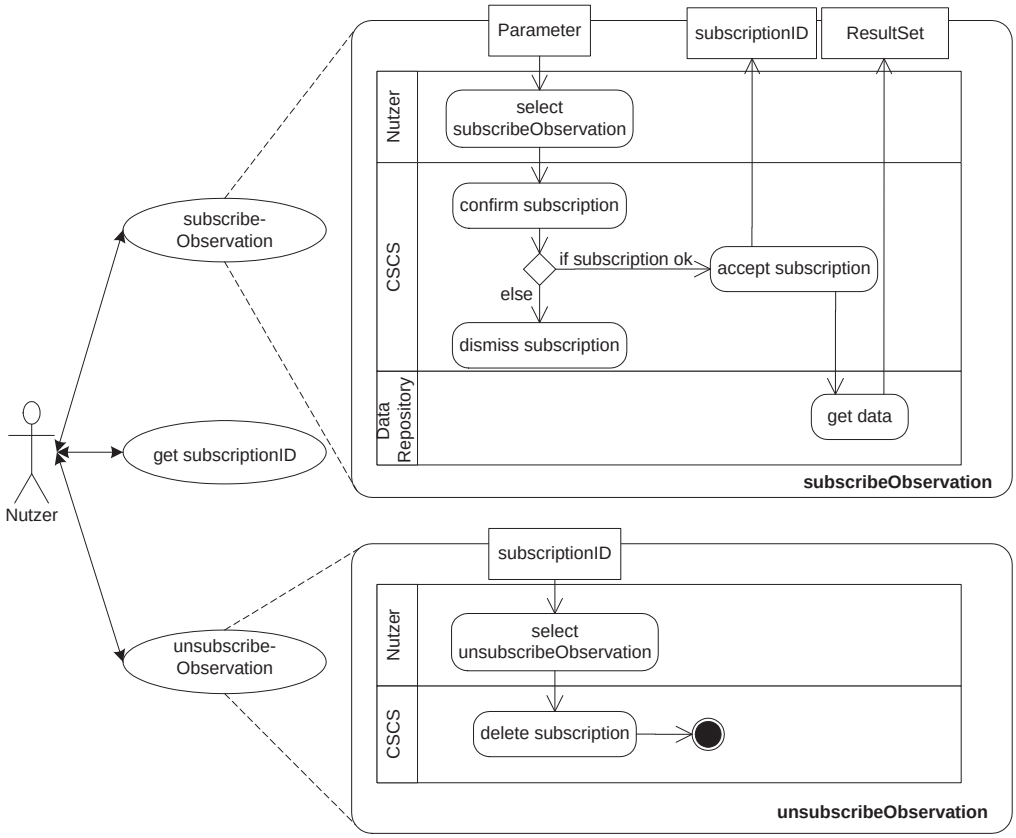


Abbildung 5.11: Use Case für die Registrierung für zeit- oder ereignisbasierte Messdaten in einem detaillierten Aktivitätsdiagramm

**Beschreibung von Abbildung 5.10** Ein Nutzer kann sich für zeit- und ereignisbasierte Messdaten mit der Funktion `subscribeContinuousEvent` registrieren. Dieser Funktion übergibt der Nutzer die erforderlichen Parameter, die in Abschnitt 5.3.1 erläutert wurden. Anschließend überprüft der CSCS, ob die Registrierung zulässig ist (confirm subscription). Wenn ja, wird die Registrierung akzeptiert und der Nutzer erhält eine *SubscriptionID*, andernfalls schlägt die Registrierung fehl (dismiss subscription). Das Data Repository spielt an den Dienstnutzer Messdaten (ResultSet) kontinuierlich aus, wenn diese zu der registrierten Anfrage passen. Wird ein auffälliger Messwert vom Sensor erfasst wird der Nutzer auch darüber informiert.
Interessiert sich der Nutzer nicht mehr für seine registrierte Anfrage, kann er die mit der Funktion `unsubscribeContinuousEvent` löschen. Dieser Funktion muss jedoch der Parameter *SubscriptionID* übergeben werden. Der CSCS löscht dann die entsprechende Registrierung.

**Beschreibung von Abbildung 5.11** Ein Nutzer kann sich für zeit- oder ereignisbasierte Messdaten mit der Funktion `subscribeObservation` registrieren. Dieser Funktion übergibt der Nutzer die erforderlichen Parameter, die in Abschnitt 5.3.1 erläutert wurden. Anschließend überprüft der CSCS, ob die Registrierung zulässig ist (confirm subscription). Wenn ja, wird die Registrierung akzeptiert und der Nutzer erhält eine *SubscriptionID*, andernfalls schlägt die Registrierung fehl (dismiss subscription). Das Data Repository spielt an den Dienstnutzer Messdaten (ResultSet) kontinuierlich aus, wenn diese zu der registrierten Anfrage passen oder

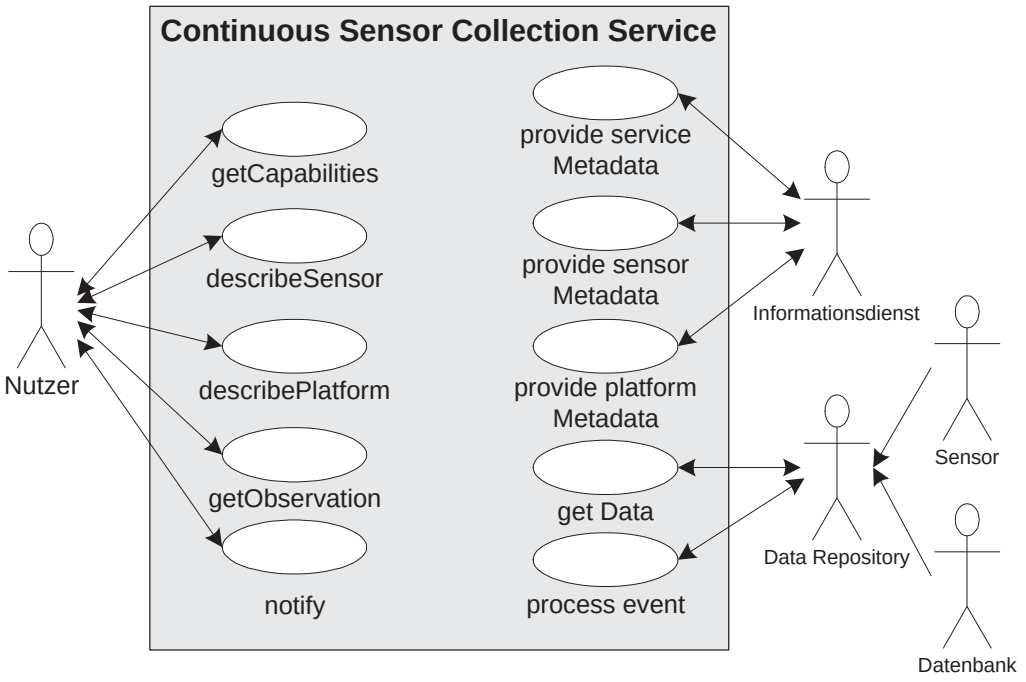


Abbildung 5.12: Weitere Use Cases seitens des Nutzers, des Informationsdienstes und des Data Repositories

der Nutzer wird nur über einen auffälligen Messwert informiert.
Interessiert sich der Nutzer nicht mehr für seine registrierte Anfrage, kann er die mit der Funktion `unsubscribeObservation` löschen. Dieser Funktion muss jedoch der Parameter *SubscriptionID* übergeben werden. Der CSCS löscht dann die entsprechende Registrierung.

**Beschreibung von Abbildung 5.12** Es stehen folgende Funktionen zur Verfügung:

| Funktion | Beschreibung |
|---|---|
| `getCapabilities`: | Der Informationsdienst liefert dem Nutzer Metadaten über den Dienst. |
| `describeSensor`: | Der Informationsdienst liefert dem Nutzer Metadaten über den Sensor. |
| `describePlatform`: | Der Informationsdienst liefert dem Nutzer Metadaten über die Plattform. |
| `getObservation`: | Der Nutzer stellt damit seine Anfrage. Die Anfrage wird einmal ausgewertet und die entsprechenden Daten werden an den Nutzer ausgespielt. |
| `notify`: | Der Dienst teilt dem Nutzer mit, wenn zu seiner Anfrage passende Messdaten erfasst worden sind. |
| `provide service Metadata`: | Der Informationsdienst stellt dem Nutzer Metadaten über den Dienst bereit. |
| `provide sensor Metadata`: | Der Informationsdienst stellt dem Nutzer Metadaten über den Sensor bereit. |
| `provide platform Metadata`: | Der Informationsdienst stellt dem Nutzer Metadaten über die Plattform bereit. |

| Funktion | Beschreibung |
|---|---|
| `process event`: | Wenn neue Messdaten erfasst wurden, werden die registrierten Nutzer informiert. |

Das Sequenzdiagramm in Abbildung 5.13 gibt die Interaktion zwischen einem Dienst und einem Nutzer an.

1. Der Nutzer stellt eine `GetCapabilities` Anfrage. Als Antwort erhält er eine Beschreibung des angebotenen Dienstes zurück.
2. Der Nutzer stellt eine `DescribeSensor` Anfrage. Als Antwort erhält er eine detaillierte Beschreibung, also Metadaten einzelner Sensoren.
3. Der Nutzer stellt eine `DescribePlatform` Anfrage. Als Antwort erhält er eine detaillierte Beschreibung, also Metadaten einzelner Plattformen.
4. Die Ergebnisse zu den bisherigen Anfragen bilden eine Grundlage für die Abfrage nach den Messdaten mittel `SubscribeObservation`.
   (a) Der CSCS prüft, ob die gewünschten Messwerte zur Verfügung stehen und liefert bei einer erfolgreichen Registrierung eine *subscriptionID* an, weil es sich um eine asynchrone Kommunikation handelt.
5. Die Messwerte werden kontinuierlich mit den registrierten Anfragen abgeglichen. Wenn ein Messwert zu der Beantwortung einer Anfrage passt, wird der Nutzer informiert bzw. die entsprechenden Daten werden ihm übermittelt (`notify`).
6. Wenn der Nutzer nicht mehr an seiner Anfrage interessiert ist, kann er diese über `UnsubscribeObservation` abbestellen.
   (a) Die *subscriptionID* zu der abbestellten Anfrage wird entfernt.
7. Möchte ein Nutzer zu seiner Anfrage eine einmalige Antwort erhalten, kann er diese mit `GetObservation` stellen.
8. Ein Nutzer erhält über `SubscribeContinuousEvent` kontinuierlich Messdaten zu seiner einmal gestellten Anfrage ausgespielt.
   (a) Der CSCS prüft, ob die gewünschten Messwerte zur Verfügung stehen und liefert bei einer erfolgreichen Registrierung eine *subscriptionID* an.
9. Wenn der Nutzer nicht mehr an seiner Anfrage interessiert ist, kann er diese über `UnsubscribeContinuousEvent` abbestellen.
   (a) Die *subscriptionID* zu der abbestellten Anfrage wird entfernt.

Die Komponenten, die für eine solche Interaktion benötigt werden, sind

- eine Steuerungskonsole,
- ein Dienstmanager,
- ein Dienst bestehend aus Rohdaten- und Anfragemodell (Continous Sensor Collection Service),

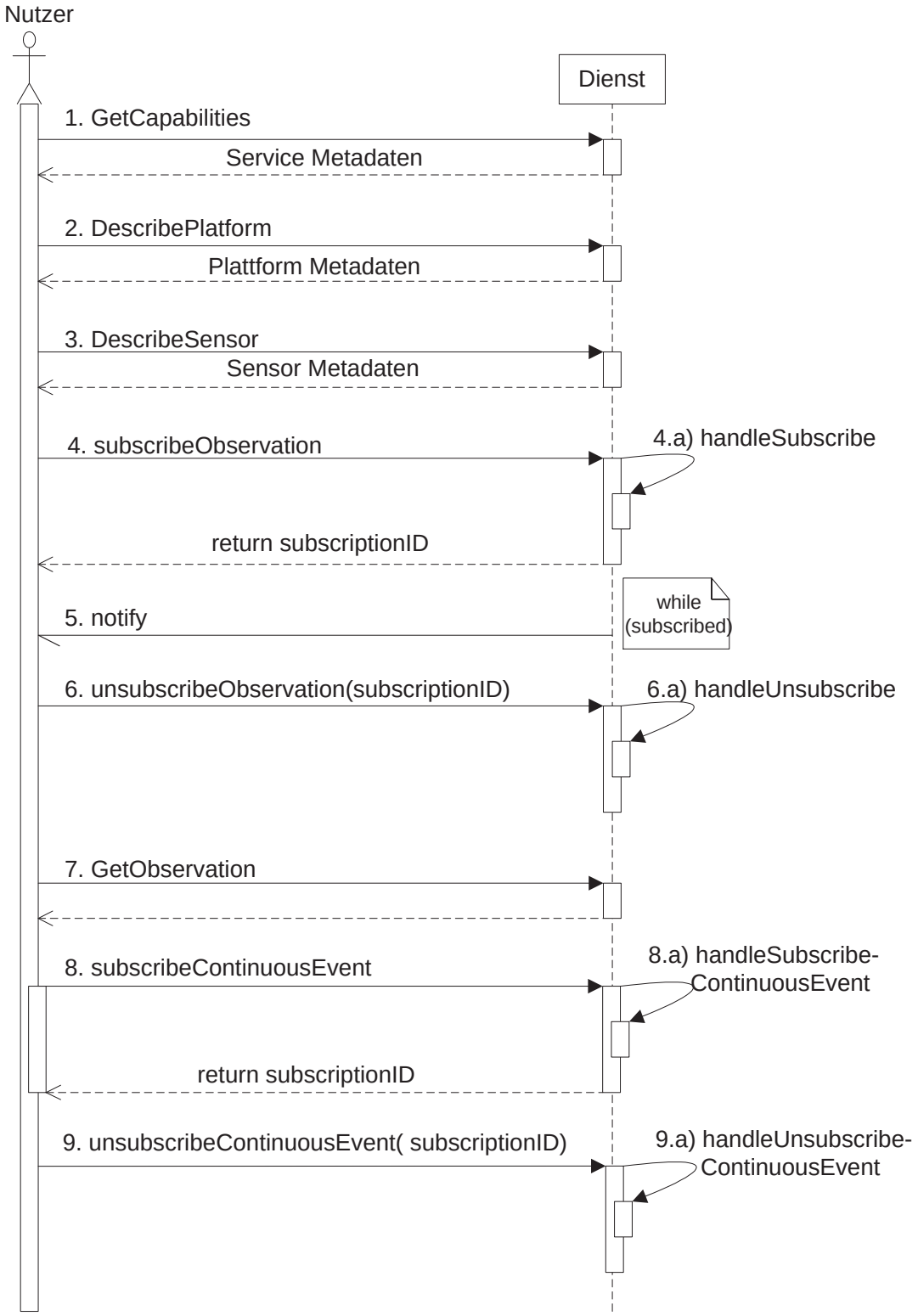


Abbildung 5.13: Interaktion zwischen Nutzer und einem Dienst

- ein Wrapper,
- ein Data Repository und
- eine Datenbank, in der die Registrierungen gespeichert werden.

Die Steuerungskonsole bietet dem Nutzer verschiedene Funktionalitäten an. Diese werden über den Dienstmanager gesteuert. Der Continuous Sensor Collection Service ist ein Anfragedienst, der Anfragen der Nutzer entgegen nimmt, sich um die Registrierung der Anfragen kümmerst und die Veröffentlichung neuer Dienste unterstützt. Der CSCS setzt sich aus einem Rohdatenmodell und einem Anfragemodell zusammen. Der CSCS basiert auf dem Rohdatenmodell, das die Daten aus dem Data Repository im XML Format repräsentiert. Das Abfragemodell selbst basiert ebenfalls auf XML. Aus diesem Grund wird der Wrapper XML2SQL eingesetzt, um eine Transformation von XML nach SQL durchzuführen, um die entsprechenden Daten aus dem Data Repository zu holen. All diese Komponenten bilden eine Architektur, die auf dem Publish/Subscribe Paradigma basiert. Diese Architektur bietet den Nutzern die Funktionalität Anfragen zu stellen, die auf dem Inhalt der Daten greifen. Es handelt sich also um eine inhaltsbasierte Publish/Subscribe Architektur. Die individuelle Sensordatenversorgung ist damit gegeben, dass die Nutzer mit den vorhandenen Funktionen ihre individuellen Anfragen definieren können und nur Daten erhalten, für die sie sich auch interessieren. Die Daten werden entweder einmalig an den Nutzer ausgespielt oder können auf Wunsch auch kontinuierlich an sie ausgespielt werden. Diese Architektur wird aufgrund ihrer Funktionalitäten als *Continuous and Individual Sensordata Supply Architecture (CISSA)* bezeichnet und ist in Abbildung 5.14 illustriert.

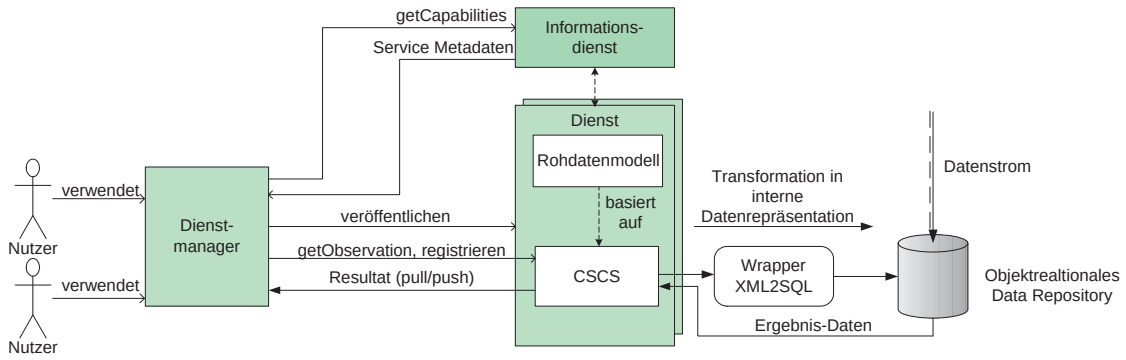


Abbildung 5.14: Continuous and Individual Sensordata Supply Architecture (CISSA)

Zum Austausch der Nachrichten[2] muss ein geeignetes Nachrichtenformat ausgewählt werden, welches beide Interaktionspartner verstehen. Diese Nachrichten werden dann über ein Transportprotokoll (TCP/IP) über das Netzwerk synchron oder asynchron ausgetauscht.

## 5.6 Zusammenfassung

In diesem Kapitel wird die inhaltsbasierte Publish/Subscribe Architektur CISSA vorgestellt, die alle Anforderungen erfüllt und damit eine individuelle Sensordatenversorgung unterstützt. Ein Nutzer kann über den Dienstmanager einen Dienst veröffentlichen. Anfragen an einem veröffentlichten Dienst werden über die Steuerungskonsole gestellt und registiert. Der Nutzer

[2]Dienstfunktionen oder Daten

kann entweder kontinuierliche oder historische Daten erhalten, die gemäß des Push-, Pull- oder als kombiniertes Push/Pull-Modell ausgespielt werden. Der CSCS stellt dafür die entsprechenden Dienstfunktionen zur Verfügung und bildet das MOI der Nutzer ab. Diese Funktionalitäten unterstützen andere Dienste nicht. Das Rohdatenmodell bildet die Rohdaten aus dem Data Repository in einem auf XML basierenden Format ab. Damit kann der Nutzer Metadaten über den Dienst, den Sensor oder über die Plattform erhalten. Das Rohdatenmodell kann positive Integerintervalle und Kommunikationsstrukturen abbilden, die andere Datenmodelle nicht abbilden können. Der Wrapper führt eine Sprachtransformation von XML nach SQL aus, um die Messdaten aus dem Data Repository zu erhalten, die als relationale Tupel vorliegen.

# Kapitel 6

# Prototypische Implementierung und Proof of Concept

Mit dem Architekturentwurf aus Kapitel 5 wurde eine Grundlage für die Implementierung geschaffen. Inwieweit die spezifizierten Erweiterungen realisierbar sind, soll durch eine prototypische Implementierung nachgewiesen werden.
In Abschnitt 6.1.1 wird die Schichtenarchitektur des Prototypen vorgestellt. Im Anschluss wird auf die bei der Umsetzung des Prototyps verwandten Techniken eingegangen (6.1.2), bevor in Abschnitt 6.1.4 die Realisierung skizziert wird. Abschließend wird der Entwurf der Architektur in Abschnitt 6.2 geprüft.

## 6.1 Prototypische Implementierung

**Definition 6.1** *Ein* **Prototyp** *ist eine spezielle Ausprägung eines ablauffähigen Softwaresystems. Er realisiert ausgewählte Aspekte des Zielsystems im Anwendungsbereich. Prototypen lassen sich wie folgt klassifizieren: Ein* **Demonstrationsprototyp** *zeigt nur die prinzipiellen Einsatzmöglichkeiten, meist nur die möglichen Handhabungsformen des zukünftigen Systems.* **Funktionelle Prototypen** *modellieren in der Regel Ausschnitte der Benutzungsschnittstelle und Teile der Funktionalität* [46].

### 6.1.1 Schichtenarchitektur

Die funktionale oder logische Gliederung des Systems dienen als Vorlage der Planung für eine Implementierung. Denn aus dieser Gliederung ergeben sich die Einheiten, die zusammenhängend entwickelt werden können. Darüber hinaus kann mit einem Paketdiadramm für ein Softwaresystem eine Schichtenarchitektur realisieren. Die einzelnen Schichten werden in UML durch Pakete definiert. Diese können weitere, nach funktionalen Gesichtspunkten gegliederte Pakete enthalten.

In Abbildung 6.1 ist eine Schichtenarchitektur des Systems dargestellt. Es besteht aus einer Anwendungs-, einer Schnittstellen-, einer Geschäftslogik-, einer Adapter- und einer Datenhaltungsschicht.

**Anwendungsschicht**
Die Anwendungsschicht besteht einer Steuerungskonsole. Darüber wird dem Dienstnutzer ermöglicht veröffentlichte Dienste zu nutzen.

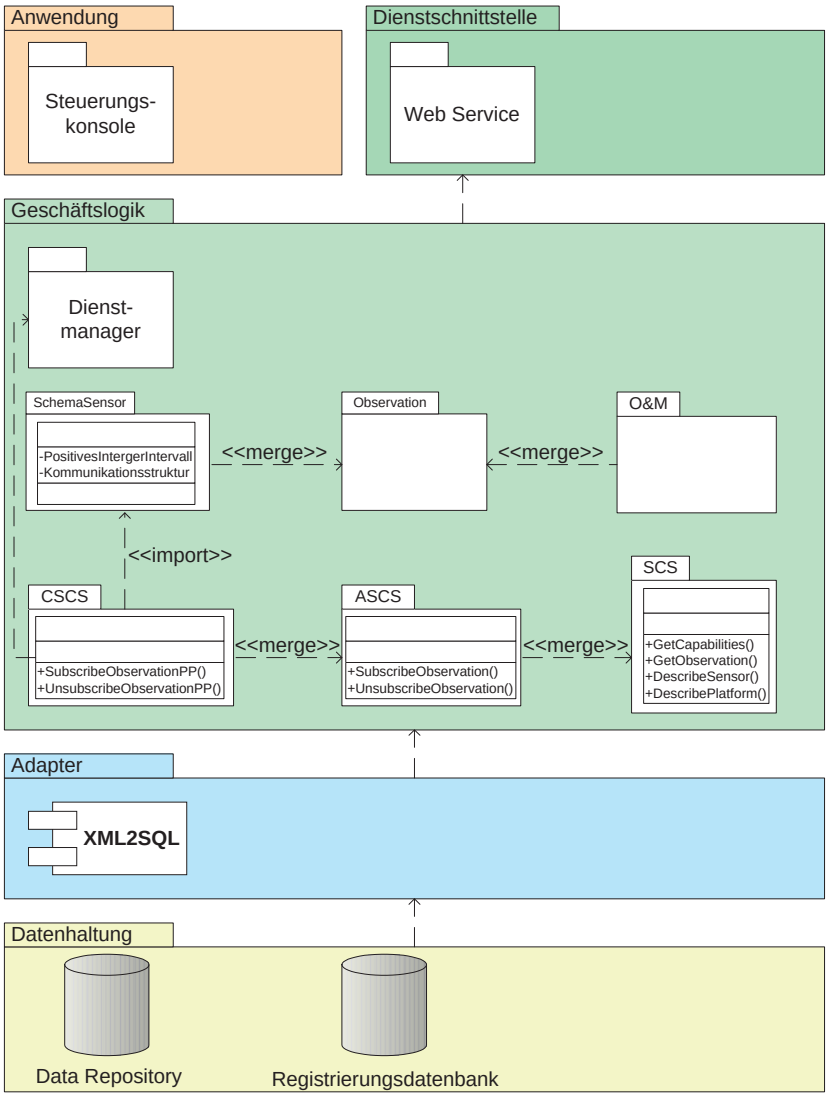


Abbildung 6.1: Schichtenarchitektur

**Schnittstellenschicht**
Die Schnittstellenschicht stellt die Schnittstellen bereit, die für eine Interaktion zwischen Dienstnutzer und Diensterbringer benötigt werden. Die Interaktion wird mittels einem Web Service realisiert, der eine Web Service Schnittstelle bereit stellt.

**Geschäftslogikschicht**
Die Geschäftslogikschicht ist die Hauptkomponente der Architektur. Sie stellt Dienste zur Verfügung, die der Dienstnutzer verwenden kann. Die Geschäftslogik unterteilt sich in das Rohdaten- und Anfragemodell aus den Abschnitten 5.2 und 5.3 und beinhaltet einen Dienstmanager.

**Adapterschicht**
Die Adapterschicht enthält alle Adapter, die für die Bearbeitung der Anfragen benötigt werden. Für die Sprachtransformation von XML nach SQL wird ein XML2SQL-Wrapper einge-

setzt.

**Datenhaltungsschicht**
Die Datenhaltungsschicht beinhaltet alle Speichermedien, die zur Datenhaltung von Rohdaten und Registrierungsdaten eingesetzt werden. Die Rohdaten liegen als relationale Tupel in dem Data Repository vor und die Registrierungsdaten werden in einer Datenbank gespeichert.

### 6.1.2 Eingesetzte Technologien

Die Technologien, die für Realisierung dieser Arbeit eingesetzt wurden, werden im Fortlaufenden kurz erläutert.

#### Java

Java ist eine objektorientierte Programmiersprache, die sich durch einige zentrale Eigenschaften auszeichnet und ist eine plattformunabhängige Laufzeitumgebung. Diese machen sie universell einsetzbar und robust. Sie unterstützt heterogene Systemumgebungen, weil der Java-Compiler Programmcode für eine virtuelle Maschine, den so genannten *Bytecode* erzeugt und nicht für eine spezielle Plattform wie in den meisten anderen Programmiersprachen. Da der Bytecode von jeder virtuellen Maschine ausgeführt werden kann, muss nicht auf einer speziellen Plattform für eine speziellen Plattform entwickelt werden.

**Web Services** Der Aufbau und die Funktionsweise eines Web Services wurde in Abschnitt 3.1.5 erläutert. Die Interaktion der Kommunikationspartner erfolgt per HTTP POST.

**Java Database Connectivity (JDBC)** JDBC (inoffizielle Abkürzung für *Java Database Connectivity*) bezeichnet einen Satz von Klassen und Methoden, um relationale Datenbanksysteme von Java zu nutzen. Die Spezifikation von JDBC wurde im Juni 1996 festgelegt. Mit der JDBC-API und den JDBC-Treibern wird eine wirksame Abstraktion von Datenbanken erreicht, sodass durch die einheitliche Programmierschnittstelle die Funktionen verschiedener Datenbanken in gleicher Weise genutzt werden können.

### 6.1.3 Eingesetzte Werkzeuge

Um die genannten Technologien zu verwenden, gibt es verschiedenste Werkzeuge. In diesem Abschnitt wird vorgestellt, welche Werkzeuge zum Einsatz kamen.

#### Eclipse

Das Java-Framework Eclipse 3.1 wurde zur Realisierung des Prototypen eingesetzt. Es stellt eine leistungsfähige GUI-Umgebung bereit.

#### Tomcat

Apache Tomcat stellt eine Umgebung zur Ausführung von Java-Code auf Webservern bereit, die im Rahmen des Jakarta-Projekts der Apache Software Foundation entwickelt wird. Es handelt sich um einen in Java geschriebenen Servletcontainer, der auch JavaServer Pages in Servlets übersetzen und ausführen kann. Er besitzt dazu einen HTTP-Server, der meist zur Entwicklung eingesetzt wird, während in Produktion zumeist ein Apache Web-Server zum

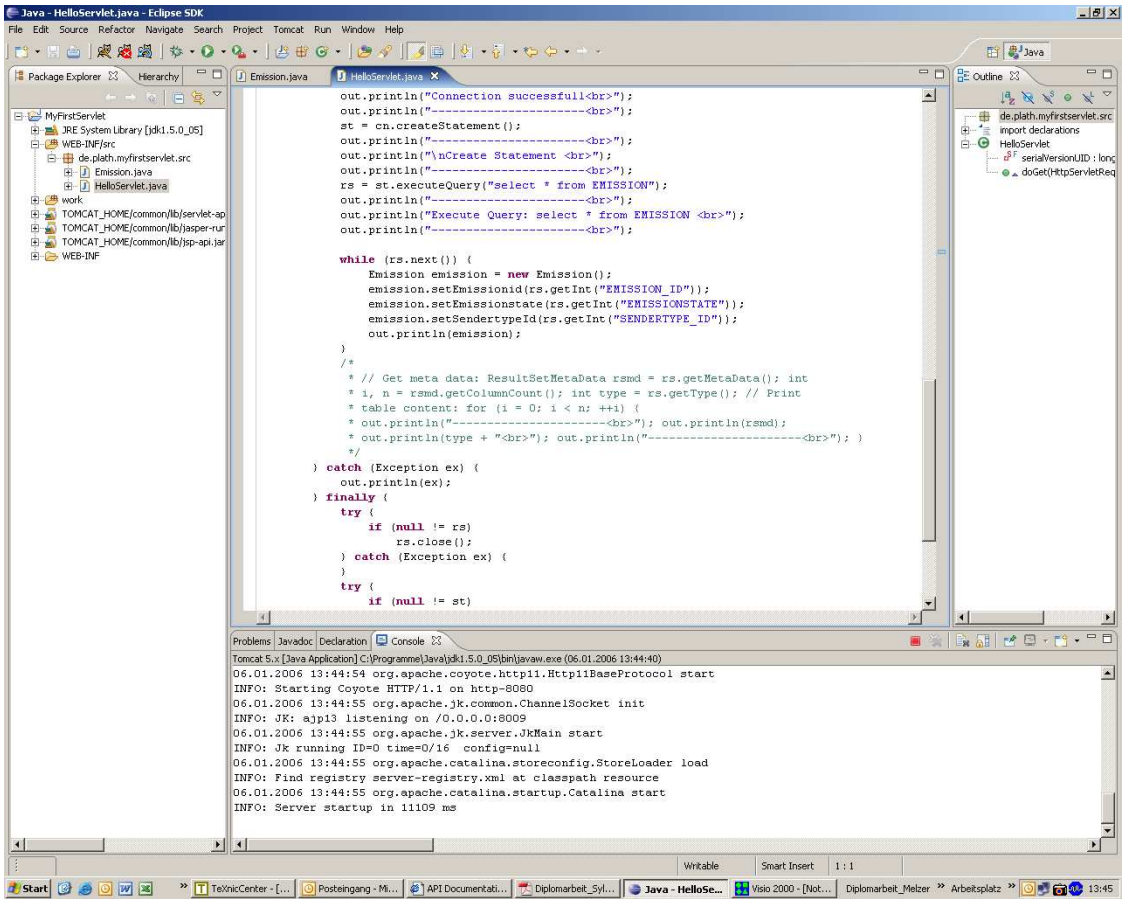


Abbildung 6.2: Eclipse 3.1

Einsatz kommt. Dazu wird in den Apache ein Plugin eingebunden, das nur Anfragen für dynamische Inhalte an den Tomcat weiterleitet. Für diese Arbeit wurde die Version 5.5.14 verwendet.

### JDOM

JDOM ist eine XML-Darstellung in der Programmiersprache Java, sie wurde von Jason Hunter und Brett McLaughlin begründet und ist inzwischen eine implementierte API zur Arbeit mit XML in Java. JDOM ist ein leichtgewichtiges und schnelles API, das relativ sparsam mit Speicher umgeht. Ähnlich wie beim DOM wird ein XML-Dokument als Baum im Hauptspeicher repräsentiert, jedoch wurde JDOM speziell für Java entwickelt. Deswegen wird eine spezifische Java-Klassen zur Repräsentation eines XML-Knoten verwendet. JDOM kann DOM-Dokumente und SAX-Events verarbeiten und entprechenden Output erzeugen. JDOM wird in dieser Arbeit als Xml2Sql-Parser eingesetzt.

### Datenbanken

Die Messdaten liegen in einer Oracle 9i Datenbank vor.
Die Subskriptionen werden in der Oracle 10g Datenbank gespeichert.

### 6.1.4 Realisierung

Für die Realisierung dieser Architektur sollten vorhandene Funktionaliäten von deegree verwendet werden. Bislang war es nur möglich Metadaten mit der `GetCapabilities()` aufzurufen. Die Funktionen des SCS, ASCS oder des SOS sind nicht standardisiert, nicht verfügbar oder funktionieren nicht, sodass für die prototypische Implementierung alle notwendigen Funktionalitäten von neuem implementiert werden mussten, um das entwickelte Konzept evaluieren zu können.

Die Klassen des Continuous Sensor Collection Service (CSCS) sind in Abbildung 6.3 illustriert.

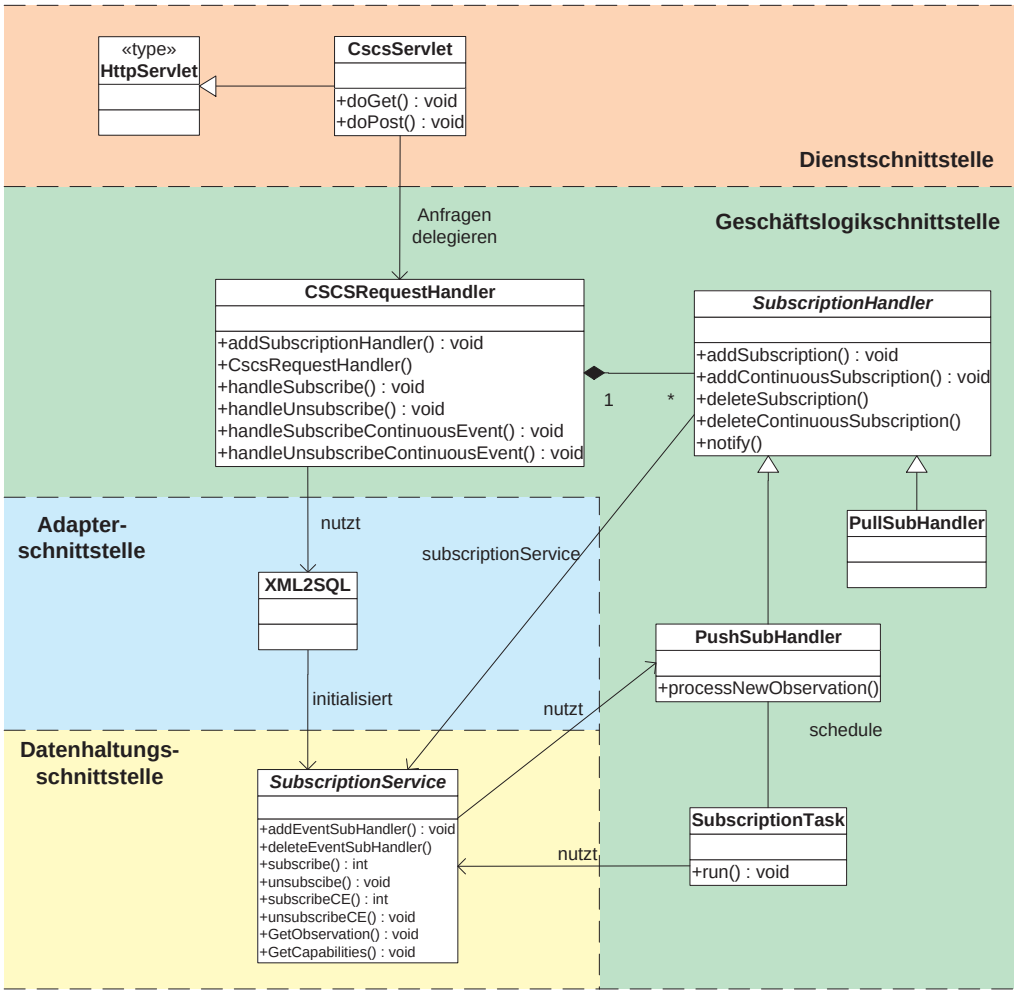


Abbildung 6.3: Klassendisgramm des CSCS

**Klassenbeschreibungen**

---

**KLASSE** CscsServlet;
**KURZBESCHREIBUNG**

Die Klasse CscsServlet ist von Http Servlet abgeleitet. Ein Dienstnutzer kann über die Steuerungskonsole seine Anfragen stellen, Dienste veröffentlichen oder seine Anfragen registrieren.
**OBERKLASSEN**
HttpServlet
**UNTERKLASSEN**
keine
**BEZIEHUNGEN**
CscsRequestHandler, Steuerungskonsole
**METHODEN/ DIENSTE**
void init(); Initialisiert alle Parameter
void doGet(); Überprüfung, welcher Dienst ausgeführt werden soll
void doPost(); Überprüfung, welcher Dienst ausgeführt werden soll
**ERGÄNZENDE BESCHREIBUNGEN**
Klassendiagramm
Alle Funktionen werden vom CscsRequestHandler delegiert.

---

---

**KLASSE** CscsRequestHandler;
**KURZBESCHREIBUNG**
Die Klasse CscsRequestHandler ist die Steuerungsklasse des Continuous Sensor Collection Service. Sie verwaltet die Funktionen für die Anfrageauswertung und -registrierung.
**OBERKLASSEN**
keine
**UNTERKLASSEN**
keine
**BEZIEHUNGEN**
SubscriptionHandler, SubscriptionService, Xml2Sql
**METHODEN/ DIENSTE**
void addSubscriptionHandler(); Übergibt dem SubscriptionHandler eine neue Anfrage, die registriert werden soll.
void CscsRequestHandler(); Übergibt dem SubscriptionService eine neue Anfrage, die einmal ausgewertet werden soll.
void handleSubscribe(); Übergibt dem SubscriptionHandler eine registrierte Anfrage, die ausgewertet werden soll. (Push-Modell)
void handleSubscribeContinuousEvent(); Übergibt dem SubscriptionHandler eine registrierte Anfrage, die ausgewertet werden soll. (Push- und Pull-Modell)
void handleUnsubscribe(subscriptionID); Übergibt dem SubscriptionHandler eine subscriptionID. Die zugehörige Anfrage wird aus der Registrierungsdatenbank gelöscht.
void handleUnsubscribeContinuousEvent(subscriptionID); Übergibt dem SubscriptionHandler eine subscriptionID. Die zugehörige Anfrage wird aus der Registrierungsdatenbank gelöscht.
**ERGÄNZENDE BESCHREIBUNGEN**
Klassendiagramm

---

---

**KLASSE** SubscriptionHandler;

**KURZBESCHREIBUNG**
Die Klasse SubscriptionHandler ist der Handler aller Funktionen des Continuous Sensor Collection Service.
**OBERKLASSEN**
keine
**UNTERKLASSEN**
PushSubHandler, PullSubHandler
**BEZIEHUNGEN**
CscsRequestHandler, SubscriptionService
**METHODEN/ DIENSTE**
void addSubscription(); Speichert die zu registrierende Anfrage (Push-Modell).
void addContinuousSubscription(); Speichert die zu registrierende Anfrage (Push- und Pull-Modell).
void deleteSubscription(subscriptionID); Löscht eine vorhandene Registrierung.
void deleteContinuousSubscription(subscriptionID); Löscht eine vorhandene Registrierung.
void notify(); Informiert den Dienstnutzer, wenn Messdaten zu seiner Anfrage passen.
**ERGÄNZENDE BESCHREIBUNGEN**
Klassendiagramm
Der SubscriptionHandler trifft die Entscheidung, ob eine Anfrage gemäß eines Push- oder/ und als Pull-Modell ausgewertet wird.

---

---

**KLASSE** PullSubHandler;
**KURZBESCHREIBUNG**
Die Klasse PullSubHandler wertet Anfragen gemäß des Pull-Modells aus.
**OBERKLASSEN**
SubscriptionHandler
**UNTERKLASSEN**
keine
**BEZIEHUNGEN**
keine
**ERGÄNZENDE BESCHREIBUNGEN**
Klassendiagramm

---

---

**KLASSE** PushSubHandler;
**KURZBESCHREIBUNG**
Die Klasse PushSubHandler wertet Anfragen gemäß des Push-Modells aus.
**OBERKLASSEN**
SubscriptionHandler
**UNTERKLASSEN**
keine
**BEZIEHUNGEN**
SubscriptionService, SubscriptionTask
**METHODEN/ DIENSTE**
void processNewObservation(); Führt die registrierten Anfragen aus.

**ERGÄNZENDE BESCHREIBUNGEN**
Klassendiagramm

---

---

**KLASSE** Xml2Sql;
**KURZBESCHREIBUNG**
Die Klasse Xml2Sql wandelt XML-Anfragen in SQL-Anfragen um.
**OBERKLASSEN**
keine
**UNTERKLASSEN**
keine
**BEZIEHUNGEN**
SubscriptionService, CscsRequestHandler
**ERGÄNZENDE BESCHREIBUNGEN**
Klassendiagramm

---

---

**KLASSE** SubscriptionService;
**KURZBESCHREIBUNG**
Die Klasse SubscriptionService stellt die Verbindungen zu der Registrierungsdatenbank und dem Data Repository her.
**OBERKLASSEN**
keine
**UNTERKLASSEN**
keine
**BEZIEHUNGEN**
Xml2Sql, SubscriptionHandler, PushSubHandler, SubscriptionTask
**METHODEN/ DIENSTE**
void addEventSubHandler(); Speichert alle Anfragen, die mehrmals ausgewertet werden sollen.
void subscribe(); Registriert die Anfrage (Push-Modell).
void unsubscribe(); Löscht die registrierte Anfrage (Push-Modell).
void subscribeCE(); Registriert die Anfrage (Push- und Pull-Modell).
void unsubscribeCE(); Löscht die registrierte Anfrage (Push- und Pull-Modell).
void getObservation(); Wertet eine Anfrage einmal aus.
void getCapabilities(); Liefert Metadaten zum ausgewählten Dienst.
**ERGÄNZENDE BESCHREIBUNGEN**
Klassendiagramm

---

**Erläutertung** Die Klasse **CscsServlet** ist von der Klasse **HttpServlet** abgeleitet, welcher ein Dienstnutzer über die Steuerungskonsole nutzen kann, und wird vom Servlet Container geladen. Der **CscsServlet** delegiert die Anfragen an den **CscsRequestHandler**, der die Aufgabe hat, die Anfragen zu bearbeiten und die entsprechenden Daten an den Dienstnutzer auszuspielen. Der **CscsRequestHandler** verwaltet ein bis mehrere **SubscriptionHandler**.

Die Klasse **SubscriptionHandler** ist die Hauptklasse der beiden Klassen **PushSubHandler** und **PullSubHandler**.
Bei einer `SubscribeContinuousEvent` oder `SubscribeObservation` Anfrage werden die Funktionen `handleSubscribeContinuousEvent` bzw. `handleSubscribe` der Klasse **CscsRequestHandler** aufgerufen sowie die `addSubscription` Funktion der abstrakten Klasse **SubscriptionHandler**. Über eine JDBC-Schnittstelle kann der **CscsRequestHandler** entsprechende Daten laden, wobei vorher die Funktion `Xml2Sql` aufgerufen wird, um die XML Anfragen in SQL umzuwandeln. Anschließend erhält der entsprechende Dienstnutzer über die Funktion `notify` die geforderten Sensordaten. Solange ein eingehender Datenstrom existiert, wird die Funktion `processNewObservation` der Klasse **PushSubHandler** aufgerufen. Diese Funktion überprüft, ob die neuen Messwerte zur registrierten Anfrage passen und diese ggf. ausgespielt werden müssen. Zeitbasierte Registrierungen werden über den **PullSubHandler** verarbeitet.

## 6.2 Proof of Concept

Es hat sich bei der prototypischen Implementierung gezeigt, dass eine Erweiterung des ACSC bzw. SCS in einer so kurzen Zeit nicht möglich war. Es lag auf der einen Seite daran, dass die Spezifikationen auf unterschiedlichen Versionen basierten, die von OpenGIS und deegree frei zur Verfügung gestellt wurden und auf der anderen Seite daran, dass eigene Ergänzungen und Erweiterungen vorgenommen wurden, die nicht auf dem OpenGIS-Standard basieren.
Die neu entwickelte Spezifikation des Continuous Sensor Collection Service (CSCS) wird in dieser Arbeit geliefert (siehe Anhang A). Es basiert auf dem Modell des Interessensprofils (MOI) sowie auf dem Rohdatenmodell. Beide Modelle wurden ebenfalls in dieser Arbeit nach den Anforderungen in Abschnitt 4.4.1 spezifiert. Durch die Erweiterung des ASCS um die Kombination des Push- und des Pull-Modells bietet er die Möglichkeit, die Dienstnutzer kontinuierlich mit Sensordaten zu versorgen und zugleich auch potentielle Abweichungen zu erkennen.

Einige Dienstfunktionen der erstellten Spezifikation des Continuous Sensor Collection Services wurden prototypisch umgesetzt, sodass gezeigt werden konnte, dass die au den Use Cases (siehe Abschnitt 5.5) abgeleiteten Anforderungen gerecht wird. Für die Überprüfung des Konzepts wurden vier Anfragebeispiele gewählt, die zeigen inwiefern die Anforderungen erfüllt werden.

**Beispiel 6.1** *Anfragebeispiel 1: „Liefere alle Metadaten zum Continuous Sensor Collection Service!“*

Diese Metadaten-Anfrage kann der Dienstnutzer über die Steuerungskonsole mit der von deegree bereitgestellten Dienstfunktion `getCapabilities` vornehmen. Der Dienst liefert die entsprechenden Metadaten und zeigt sie in der Steuerungskonsole an. Der folgende Ausschnitt zeigt das Ergebnis einer solchen Anfrage.

```
<?xml version="1.0" encoding="UTF-8"?>
<Sensor xmlns:sch="http://www.ascc.net/xml/schematron"
        xmlns:gml="http://www.opengis.net/gml"
        xmlns:ns1="http://www.opengis.net/om"
        xmlns:sml="http://www.opengis.net/sensorML"
        xmlns:swe="http://www.opengis.net/swe"
```

```
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:smil20="http://www.w3.org/2001/SMIL20/"
        xmlns:smil20lang="http://www.w3.org/2001/SMIL20/Language"
        xmlns:ism="urn:us:gov:ic:ism:v2"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="F:\GMLSML\SchemaSensor.xsd">
  <sml:description>
    <sml:Discussion>
      Hochfrequenzdaten stehen im Data Repository zur Abfrage zur Verfügung
    </sml:Discussion>
  </sml:description>
  <sml:characteristics>
    <sml:PropertyList>
      <sml:property>
        <swe:Category>Hochfrequenzdaten</swe:Category>
      </sml:property>
    </sml:PropertyList>
  </sml:characteristics>
  <sml:contact role="www.plath.de">
    <sml:contact>
      <sml:Person>
        <sml:surname>Sylvia</sml:surname>
        <sml:name>Melzer</sml:name>
        <sml:userID>12345</sml:userID>
        <sml:affiliation>Plath GmbH</sml:affiliation>
        <sml:phoneNumber>348</sml:phoneNumber>
        <sml:email>sylvia.melzer@plath.de</sml:email>
      </sml:Person>
    </sml:contact>
  </sml:contact>
  <ObservationCollection>...</ObservationCollection>
</Sensor>
```

**Beispiel 6.2** *Anfragebeispiel 2: „Liefere die letzten 10000 Emissionen mit der Frequenz größer als 1000000!“*

Diese Anfrage kann der Dienstnutzer mit der Dienstfunktion `getObservation` stellen. Hierzu wird eine Anfrage in XML formuliert, wie sie in Abschnitt 5.3.1 vorgestellt wurde. Diese Anfrage wird vom CSCS einmalig bearbeitet. Dazu wird ein XML2SQL Wrapper eingesetzt, um eine Sprachtransformation von XML in SQL vorzunehmen, weil die Daten in interner Datenrepräsentation relational vorliegen.
Es erfolgt eine zeitnahe Auswertung sowie eine zeitnahe Ausspielung an den Dienstnutzer. Die Daten werden in der Steuerungskonsole angezeigt.

**Beispiel 6.3** *Anfragebeispiel 3: „Liefere kontinuierlich alle Emissionen aus der Region Mexiko!“*

Für das kontinuierliche Ausspielen der Daten steht die Dienstfunktion `subscribeObservation` bzw. `subscribeContinuousEvent` zur Verfügung. Für die Auswertung des Prädikats *Region*

wird der geospartiale Dienst der Oracle 10g Datenbank genutzt. Dieser wertet nicht nur dieses Prädikat aus, sondern führt auch die Berechnung Punkt-in-Polygon durch. Diese Anfrage wurde kontinuierlich ausgewertet und die entsprechenden Ergebnisse werden in der Steuerungskonsole angezeigt. Die zeitnahe Auswertung dieser Anfrage kann nicht evaluiert werden, weil zum Zeitpunkt des Tests keine kontinuierlichen Datenströme zur Verfügung standen.

**Beispiel 6.4** *Anfragebeispiel 4: „Liefere alle neuen Teilnehmer in einer bekannten Kommunikationsstruktur!“*

Die Auswertung einer Kommunikationsstruktur konnte nicht vorgenommen werden, weil es bislang noch keinen Dienst gibt, der eine solche Auswertung unterstützt. Daher kann dieser Anfragetyp nicht evaluiert werden.
Es wurde gezeigt, dass durch die Strukturierung der Klassen bei der prototypischen Implementierung eine Entkopplung der Systeme bzw. Komponenten erreicht wurde. Wird beispielsweise eine Datenbank ausgetauscht, muss nur eine Anpassung der entsprechenden Klasse (**SubscriptionService**) vorgenommen werden, weil alle Datenbankverbindungen in dieser Klasse gekapselt sind.
Mit dem Prototyp wurde ein inhaltsbasiertes System geschaffen, welches durch den Zugriff auf den Inhalt der Daten eine individuelle Sensordatenversorgung unterstützt.
Die Interaktion zwischen Dienstnutzer und Diensterbringer erfolgte per HTTP Post. Es wurde dieses Protokoll gewählt, weil der konventionelle Sensor Collection Service dieses Protokoll verwendet und für die ersten Tests zur Prüfung der generellen Funktionsweise ausreichend war.
Der Prototyp zeigte, dass nur Daten zeitnah ausgespielt werden konnten, wenn die Anfragen einer Filterbedingung unterlagen, sodass keine Übermittlung der gesamten Datenmenge erfolgte. Für die Übermittlung aller Daten ergaben Tests, dass das HTTP Protokoll nicht ausreichend ist und hier nach einer anderen Möglichkeit geschaut werden muss.

# Kapitel 7

# Zusammenfassung und Ausblick

In diesem Kapitel wird die Diplomarbeit zusammengefasst. Der Ausblick soll eine Grundlage für neue Ideen schaffen und Weiterentwicklungsmöglichkeiten aufzeigen.

## 7.1 Zusammenfassung

In dieser Arbeit wurden die Grundlagen strategischer und taktischer Überwachungssysteme und die generelle Funktionsweise des Sensorsystems der Firma Plath GmbH erläutert. Für dieses Sensorsystem wurde ein adäquater Architekturvorschlag gemacht, welche die zentralen Aspekte Erfassung von Sensordaten in kontinuierlichen Strömen, die kontinuierliche Verwaltung und Verarbeitung großer Datenmengen nahe der Echtzeit und die zeitnahe Ausspielung von Messdaten an nachfolgende Dienstnutzer adressiert. Diese Aspekte haben großen Einfluss auf die zu entwerfende Architektur.
Es wurden die technologischen Grundlagen, Sprachen und der aktuelle technische Stand vorgestellt und erläutert, welche für das Konzept einer Architektur von wesentlicher Bedeutung sind.
Die themenverwandten Arbeiten zeigten in welchen Bereichen stark geforscht wird und dass nicht alle Anforderungen wie die kontinuierliche Sensordatenverarbeitung nahe der Echtzeit mit einer großen Anzahl an Daten mit einem inhaltsbasierten Mechanismus erfüllt werden können. Für die dynamische Interaktion der Systeme bot sich das inhaltsbasierte Publish/Subscribe Paradigma an, das eine Entkopplung der Systeme unterstützt sowie eine individuelle Sensordatenversorgung. Zum erfolgreichen Austausch von Informationen in lose gekoppelten Systemen wird eine technische Datenrepräsentationssprache eingesetzt, die die interne Dateninfrastruktur kapselt und auf einer standardisierten Weise repräsentiert, sodass Dienstnutzer die Informationen nicht unterschiedlich interpretieren.
Die Analyse der existierenden Sprachen ergab, dass sie einerseits nicht mächtig genug sind, die Anfragen zu beantworten und zum anderen, dass die Datenrepräsentationssprache nicht standardisiert ist. Des Weiteren ergab sich dass nicht alle Datenrepräsentationssprachen Mengen, Zeitpunkte, Zeit- und Frequenzintervalle, Geodaten, Unschärfe, Netze und Kommunikationsstrukturen abbilden können und daher ergänzt werden müssen. Für die Wahl einer Anfragesprache war es wichtig, dass sie die zum einen das Interessensprofil der Dienstnutzer abbilden, die Beispielanfragen beantworten und die Daten an die Dienstnutzer zeitnah ausspielen kann. Die analysierten Anfragesprachen zeigten, dass sie nicht alle Anforderungen wie in Abschnitt 4.4 erläutert, erfüllen.
Auf der Basis der in der Analyse gewonnenen Erkenntnissen wurde eine Continuous and Indi-

vidual Sensordata Supply Architecture (CISSA) entworfen, die eine inhaltsbasierte und individuelle Sensordatenversorgung unterstützt. Die Architektur basiert auf dem Publish/Subscribe Paradigma, welches eine inhaltsbasierte und individuelle Sensordatenversorgung unterstützt. Sie besteht aus einer Steuerungskonsole, einem Dienstmanager, einem Informationsdienst, einem Continuous Sensor Collection Service (CSCS), welcher auf dem Rohdatenmodell und dem Modell des Interessensprofils (MOI) basiert und einem Data Repository. Das Rohdatenmodell besitzt die Ergänzungen des vorhandenen Datenmodells und kann positive Frequenzintervalle und Kommunikationsstrukturen repräsentieren. Der CSCS ist ein Anfragedienst, der die Anfragen eines Dienstnutzers bearbeitet und die Daten gemäß des Push-Modells, des Pull-Modells oder gemäß der Kombination der beiden Modelle ausspielt. Dieses Modell ist eine Erweiterung des Active Sensor Collection Service, welches die Kombination der beiden Modelle nicht unterstützt.
Über die Steuerungskonsole können Dienstnutzer eigene Interessensprofile und Dienste definieren und einsetzen. Der CSCS stellt Dienstfunktionen zur Verfügung um die vom Dienstnutzer gewünschten Funktionen zu erfüllen. Diese Arbeit liefert die konzeptionellen Modelle und Spezifikationen der genannten Komponenten.
Dieses Konzept wurde prototypisch implementiert und geprüft.

## 7.2 Ausblick

Für die Spezifikation des Rohdatenmodells und des Continuous Sensor Collection Service wurden die neuesten Versionen von SensorML [7], O&M [11] und SCS [12] eingesetzt. Diese Sprachen basieren auf unterschiedlichen GML Versionen, sodass in dieser Arbeit die Spezifikationen an die GML Version 3.1.1. angepasst wurden. Diese Anpassung sollte vom OGC vorgenommen werden, um einen einheitlichen Standard verwenden zu können.

Die Dienstfunktionen zur Registrierung der Anfragen basieren auf dem Filterkonzept des OGCs. Für die Beantwortung von komplexeren Anfragen, z.B. die eine Verbindung zweier Relationen mit einem gemeinsamen Wertebereich berechnen, empfielt es sich die Verwendung eines Verbundoperators (engl. join). OGC entwickelt zurzeit eine „ join“-Anfragespezifikation, welche die Mächtigkeit einer Anfragesprache erhöht.

Die prototypische Implementierung hat gezeigt, dass die Spezifikation des CSCS realisierbar ist. Eine endgültige Validierung kann jedoch nur durch Durchführung umfassender Laufzeittests erfolgen, die im Rahmen einer zeitlich begrenzten Diplomarbeit nicht erbracht werden kann. Erst durch eine detaillierte Analyse bzgl. des Echtzeitverhaltens mit Anfragen unterschiedlicher Komplexität kann eine konkrete Aussage über die konzipierte Architektur gemacht werden.

# Glossar

**Anfrage** Es handelt sich um eine fachliche Anfrage, die das Interesse der Auswertekomponente in Form einer Fragestellung widerspiegelt.

**API** Application Programming Interface.

**Auswertekomponente** Eine Auswertekomponente ist ein Teil eines Auswertesystems, der die Sensordaten verarbeitet, sein Interessensprofil einem Dienst registriert und Anfragen stellt.

**Auswertesystem** Im Auswertesystem besteht aus unterschiedlichen Auswertekomponenten. Es werden an dieses Teilsystem die Sensordaten ausgespielt.

**Azimut** Winkel zwischen der Meridianebene und dem Schnittpunkt des Vertikalkreises eines Gestirns mit dem Horizont.

**CORBA** Common Object Request Broker Architecture.

**Emission** Eine Emission ist ein Konzept, das sich aus den Domänen Zeitintervall, Frequenzintervall und Ortung zusammensetzt.

**Erfassungssystem** Im Erfassungssystem werden Sensordaten erfasst und verteilt.

**gerichtete Kommunikation** Bei einer gerichteten Kommunikation werden die Daten gebündelt entlang einer gewünschten Strecke gesendet.

**GIS** Geografisches Informationssystem.

**GML** Geography Markup Language.

**HTTP** Hypertext Transfer Protocol.

**Interessensprofil** Ein Interessensprofil stellt das Interessensgebiet einer Auswertekomponente dar. Eine Auswertekomponente kann mehrere Interessensprofile besitzen, die jeweils einen Namen, eine Beschreibung sowie inhaltliche Angaben zu ihrem Interessensgebiet beinhalten.

**Interessensprofilschema** Ein Interessensprofilschema ist ein Sprachmittel, die bei der Modellierung von Interessen zur Verfügung steht. Ein Interessensprofil ist einem Interessensprofilschema zugeordnet.

**JDBC** Java Database Connectivity.

**Job** Ein Job definiert einerseits einen Messbereich, den ein oder mehrere Sensoren erfassen sollen. Andererseits dient der Job dazu die Interssensprofile der Auswertesysteme auszudrücken.

**Meridianebene** Die Meridianebene eines Punktes auf der Erdoberfläche ist jene Vertikalebene, die durch Lotrichtung und Parallele zur Erdachse aufgespannt wird.

**OGC** Open Geospatial Consortium.

**O und M** Observations and Measurements.

**OWL** Web Ontology Language.

**Parser** Ein Parser ist ein Programm, das entscheidet, ob eine Eingabe zur Sprache einer bestimmten Grammatik gehört.

**Publisher** Das Erfassungssystem, das die Sensordaten für die Auswertekomponenten zur Verfügung stellt.

**Query on demand** Einmalig gestellte Anfrage.

**RDF** Resource Description Framework.

**RDQL** RDF Data Query Language.

**relational vollständig** Wenn jeder Term der relationalen Algebra in der Anfragesprache umgesetzt werden kann, heißt sie relational vollständig.

**RPC** Remote Procedure Protocol.

**RQL** RDF Query Language.

**Rohdaten** Es handelt sich um Sensordaten, die in unverarbeitender Form in der Datenbank (Erfassungssystem) zur Verfügung stehen.

**Sensoren** messen bestimmte Größen und Eigenschaften.

**SensorML** Sensor Model Language.

**SOA** Service-orientierte Architektur

**SOAP** Ursprünglich als Abkürzung für *Simple Object Access Protocol* verwendet. Mit Erscheinen der SOAP Spezifikation 1.2 wurde die Verwendung als eigenständiger Begriff verankert.

**Sperrfrequenzen** Sperrfrequenzen beihalten die Frequenzen, die nicht erfasst werden, weil sie im Allgemeinen uninteressant sind.

**SQL** Structured Query Language

**Subscriber** Eine Auswertekomponente, die ihr Interesse registriert.

**Technische Anfrage** Eine technische Anfrage ist eine (existierende) Anfragesprache.

**UDDI** Universal Description Discovery and Integration.

**URI** Universal Resource Identifier

**URL** Uniform Resource Locator

**Verdichtete Daten** Es handelt sich um Sensordaten, die in verarbeiteter Form zur Verfügung stehen.

**WSDL** Web Service Description Language.

**XML** Extensible Markup Language.

**XQuery** XML Query Language.

**XSLT** Extensible Stylesheet Language Transformation.

# Literaturverzeichnis


[1] Stephan Aier and Marten Schönherr. *Enterprise Application Integration. Serviceorientierung und nachhaltige Architekturen*. Gito, 1 edition, 2004.

[2] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services Concepts, Architecture and Applications*. Springer, 1 edition, 2004.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. Cql: A language for continuous queries over streams and relations. In *DBPL*, pages 1–19, 2003.

[4] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[5] Douglas K. Barry. *Web Services and Service-Oriented Architectures. The savvy manager's guide. Your road map to emerging IT*. Morgan Kaufmann, Amsterdam, 1 edition, 2004.

[6] Sujoe Bose and Leonidas Fegaras. Data stream management for historical xml data. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 239–250, New York, NY, USA, 2004. ACM Press.

[7] Mike Botts. Opengis sensor model language (sensorml) implementation specification - version 1.0. Technical report, Open Geospatial Consortium Inc., October 2005.

[8] I.N. Bronstein, K.A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 5 edition, 2000.

[9] Ahmet Bulut and Ambuj K. Singh. A unified framework for monitoring data streams in real time. In *ICDE*, pages 44–55, 2005.

[10] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data, 2002.

[11] Simon Cox. Observations and measurements - version 0.9.2. Technical report, Open Geospatial Consortium Inc., February 2003.

[12] Simon Cox. Sensor collection service - ogc 03-022r3, version 0.9.0. Technical report, Open Geospatial Consortium Inc., January 2003.

[13] Ajoy Kumar Datta, Maria Gradinariu, Michel Raynal, and Gwendal Simon. Anonymous publish/subscribe in p2p networks. In *IPDPS*, page 74, 2003.

[14] Y. Diao, S. Rizvi, and M. Franklin. *Towards an Internet-Scale XML Dissemination Service*, 2004.

[15] Wolfgang Dostal and Mario Jeckle. Semantik, odem einer service-orientierten architektur. *JAVAspektrum*, pages 53–56, January 2004.

[16] Thomas Eiter and Leonid Libkin, editors. *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*, volume 3363 of *Lecture Notes in Computer Science*. Springer, 2005.

[17] Thomas Erl. *Service-Oriented Architecture. A field guide to integrating XML and Web services*. Upper Saddle River, Prentice Hall, 1 edition, 2004.

[18] Jan Galinski. *Einsatz von Web-Services im Semantic Web am Beispiel der RACER Engine und OWL-QL*. PhD thesis, Technische Universität Hamburg-Harburg, 2004.

[19] Peter Gerstbach. Xml data binding. Master's thesis, Technische Universität Wien, 2004.

[20] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. *Java Message Service*, 2002.

[21] Manfred Hein and Henner Zeller. *Java Web Services – Entwicklung plattenübergreifender Dienste mit J2EE, XML und SOAP*. Addison-Wesley, 2003.

[22] L. Hotz. *A Structure-based Configuration Language*, pages 137–166. AKA Verlag, December 2005.

[23] Lothar Hotz and Bernd Neumann. Scene Interpretation as a Configuration Task. *KI-Zeitung*, 3:59–65, 2005.

[24] Lothar Hotz and Bernd Neumann. Scene Interpretation as a Configuration Task. Technical Report Bericht 262, Universität Hamburg, 2005. http://medoc.informatik.uni-hamburg.de/Dienst/UI/2.0/Describe/ncstrl.uhamburg_cs%2fB-262-05?abstract=hotz.

[25] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile environment. In *MobiDe '01: Proceedings of the 2nd ACM international workshop on Data engineering for wireless and mobile access*, pages 27–34, New York, NY, USA, 2001. ACM Press.

[26] Fausto Ibarra. The enterprise service bus: Building enterprise soa. WWW-Seite, September 2004. http://dev2dev.bea.com/pub/a/2004/12/soa_ibarra.html (Zugriff am 12.10.05).

[27] Gregory Karvounarakis, Vassilis Christophides, Sofia Alexaki, Dimitris Plexousakis, and Michel Scholl. *RQL: A Declarative Query Language for RDF**, 2002.

[28] Alfons Kemper and André Eickler. *Datenbanksysteme: Eine Einführung*. Oldenburg Wissenschaftsverlag GmbH, 5 edition, 2004.

[29] Alfons Kemper, Erhard Rahm, Bernhard Seeger, and Gerhard Weikum. Dynamische informationsfusion. WWW-Seite, September 2004. http://www-db.in.tum.de/research/GI-Workshop-DynamischeInfoFusion/GI-Workshop.html (Zugriff am 27.05.05).

[30] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA*. Upper Saddle River, Prentice Hall, 1 edition, 2004.

[31] Paul Krill. Streambase eye real-time sharing apps. WWW-Seite, January 2005. http://www.infoworld.com/article/05/01/07/HNstreambasestone_1.html (Zugriff am 24.07.05).

[32] Wolfgang Lehner and Harald Schöning. *XQuery Grundlagen und fortgeschrittene Methoden*. dpunkt.verlag GmbH, 1 edition, 2004.

[33] Ulf Leser. A query language for biological networks. Technical report, Department for Computer Science, Humboldt-Universität zu Berlin, 2005.

[34] Sam Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, 2002.

[35] McLaughlin and Brett. *Java and XML Data Binding*. OReilly and Associates, 1 edition, 2002.

[36] Christoph Nasko. *Speicher- und Auslesestrategien für Datenströme in Video-Servern und deren Analyse mittels Simulation*. PhD thesis, Technische Universität Hamburg-Harburg, 2000.

[37] Bernd Neumann and R. Möller. On Scene Interpretation with Description Logics. Technical Report Bericht 257, Universität Hamburg, 2004.

[38] ORACLE. Oracle application server 10g esb. WWW-Seite, July 2005. http://www.oracle.com/technology/products/integration/esb/pdf/ds_esb_v10_1_2.pdf (Zugriff am 19.10.05).

[39] Rui Peng, Kien A. Hua, and Georgiana L. Hamza-Lup. A web services environment for internet-scale sensor computing. In *IEEE SCC*, pages 101–108, 2004.

[40] Roland Piquepaille. Streaming a Database in Real Time. WWW-Seite, January 2005. http://www.primidi.com/2005/01/21.html (Zugriff am 24.07.05).

[41] Shelley Powers. *Practical RDF*. O'Reilly & Associates, Inc., 1 edition, 2003.

[42] Shelley Powers. *Practical RDF*, chapter 5. O'Reilly & Associates, Inc., 2003.

[43] Shelley Powers. *Practical RDF*, chapter 12. O'Reilly & Associates, Inc., 2003.

[44] Shelley Powers. *Practical RDF*, chapter 8. O'Reilly & Associates, Inc., 2003.

[45] Arcot Rajasekar, Sifang Lu, Reagan Moore, Frank Vernon, John Orcutt, and Kent Lindquist. Accessing sensor data using meta data: a virtual object ring buffer framework. In *DMSN '05: Proceedings of the 2nd international workshop on Data management for sensor networks*, pages 35–42, New York, NY, USA, 2005. ACM Press.

[46] Peter Rechenberg and Gustav Pomberger. *Informatik-Handbuch*. Carl Hanser Verlag, 1 edition, 1997.

[47] Daniel Rock. Sfb 627 - nexus, sensordatenmodellierung in nexus, stupro-seminar smartroom, version 1.02. Technical report, Universität Stuttgart, 2004.

[48] William A. Ruh, Francis X. Maginnis, and William J. Brown. *Enterprise Application Integration*. John Wiley & Sons, INC., 1 edition, 2001.

[49] Helmut Rzehak. *Echtzeitsysteme und Fuzzy Control*. Verlag Vieweg, 1 edition, 1994.

[50] Atul Saini. Demystifying the Enterprise Service Bus. WWW-Seite, September 2003. http://bijonline.com/PDF/Sep03Saini.pdf (Zugriff am 19.10.05).

[51] Bernhard Seeger. Datenströme. *Datenbank-Spektrum*, pages 30–33, September 2004.

[52] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 430–441. ACM Press, 1994.

[53] Alan C. Shaw. *Real-Time Systems and Software*. John Wiley & Sons, INC., 1 edition, 2001.

[54] Marco Skulschus and Marcus Wiederstein. *XML Schema*. Galileo Computing, 200.

[55] Heiner Stuckenschmidt and Frank van Harmelen. *Information Sharing on the Semantic Web*. Springer Verlag, 1 edition, 2005.

[56] Stream Based Systems. Frequently asked questions. WWW-Seite, October 2005. http://www.streambase.com/www/products/faqs.html (Zugriff am 10.10.05).

[57] Andrew S. Tanenbaum and Marten van Steen. *Verteilte Systeme – Grundlagen und Paradigmen*. Prentice-Hall, International, 2003.

[58] Karsten Thurow. Ein generisches Notifiktations-Framework. Master's thesis, Technische Universität Hamburg-Harburg, 2000.

[59] Feng Tian, Berthold Reinwald, Hamid Pirahesh, Tobias Mayr, and Jussi Myllymaki. Implementing a scalable xml publish/subscribe system using relational database systems. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 479–490, New York, NY, USA, 2004. ACM Press.

[60] Panagiotis A. Vretanos. Sensor web als grundlage für hochwasserfrühwarnsysteme. Technical report, Open Geospatial Consortium Inc., May 2005.

[61] Alexander C. Walkowski. *Active Sensor Collection Service - Weiterentwicklung des OpenGIS SCS anhand eines use case aus dem Hochwassermanagement* -. PhD thesis, Westfälische Wilhelms-Universität Münster, 2004.

[62] Alexander W. Walkowski. Opengis® filter encoding implementation specification - version 1.1.0. Technical report, Institut für Geoinformatik, Westfälische Wilhelms-Universität Münster, 2005.

[63] Jinling Wang, Beihong Jin, Jing Li, and Danhua Shao. A semantic-aware publish/subscribe system with rdf patterns. In *COMPSAC*, pages 141–14, 2004.

[64] Wikipedia. Datenstrom. WWW-Seite, September 2005. http://de.wikipedia.org/wiki/Datenstrom (Zugriff am 17.09.05).

[65] Wikipedia. Frühwarnsystem. WWW-Seite, July 2005. http://de.wikipedia.org/wiki/Fr%C3%BChwarnsystem (Zugriff am 18.07.05).

[66] Wikipedia. Service-orientierte architektur. WWW-Seite, September 2005. http://de.wikipedia.org/wiki/Service_Oriented_Architecture (Zugriff am 15.09.05).

[67] Gregory Williams. Life on marsrdf::query. WWW-Seite, November 2005. http://kasei.us/code/rdf-query/ (Zugriff am 21.11.05).

[68] Dieter Zöbel and Wolfgang Albrecht. *Echtzeitsysteme: Grundlagen und Techniken*. Internat. Thomson Publisher, 1 edition, 1995.

[69] Liang-Jie Zhang, editor. *Web Services, European Conference, ECOWS 2004, Erfurt, Germany, September 27-30, 2004, Proceedings*, volume 3250 of *Lecture Notes in Computer Science*. Springer, 2004.

# Anhang A

## Rohdatenschema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:gml="http://www.opengis.net/gml"
           xmlns:sml="http://www.opengis.net/sensorML"
           xmlns:ns1="http://www.opengis.net/om"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
  <xs:import namespace="http://www.opengis.net/gml"
             schemaLocation="F:\GMLSML\GML\3.1.1\base\gml.xsd"/>
  <xs:import namespace="http://www.opengis.net/sensorML"
             schemaLocation="F:\GMLSML\SML\base.xsd"/>
  <xs:element name="Sensor">
    <xs:annotation>
      <xs:documentation>Data Repository</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="sml:description"/>
        <xs:element ref="sml:characteristics" minOccurs="0"/>
        <xs:element ref="sml:contact" minOccurs="0"/>
        <xs:element ref="ObservationCollection"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="ObservationCollections">
    <xs:annotation>
      <xs:documentation>OM-Element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="gml:boundedBy"/>
        <xs:element ref="ObservationMembers" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="ObservationMembers">
```

```
    <xs:annotation>
      <xs:documentation>OM-Element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="gml:Observation"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="ObservationArray">
    <xs:annotation>
      <xs:documentation>OM-Element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="gml:boundedBy"/>
        <xs:element ref="ObservationMembers"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="ObservationMember">
    <xs:annotation>
      <xs:documentation>OM-Element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="gml:name"/>
        <xs:element ref="ObservationCollections"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="ObservationCollection">
    <xs:annotation>
      <xs:documentation>OM-Element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="gml:name"/>
        <xs:element ref="ObservationMember" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="FeatureCollection">
    <xs:annotation>
      <xs:documentation>OM-Element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
```

```
        <xs:element ref="gml:FeatureCollection"/>
        <xs:element ref="FeatureMembers"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="FeatureMembers">
    <xs:annotation>
      <xs:documentation>OM-Element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Station" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Station">
    <xs:annotation>
      <xs:documentation>OM-Element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="gml:description"/>
        <xs:element ref="gml:name"/>
        <xs:element ref="gml:location"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

## Continuous Sensor Collection Service Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:sml="http://www.opengis.net/sensorML"
           xmlns:gml="http://www.opengis.net/gml"
           xmlns:sim="http://www.opengis.net/sim"
           xmlns:ogc="http://www.opengis.net/ogc"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:ns1="http://www.opengis.net/scs"
           targetNamespace="http://www.opengis.net/scs"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
  <!-- ********************************************************************* -->
  <!-- * I N C L U D E S & I M P OR T S * -->
  <!-- ********************************************************************* -->
  <!-- import gml 3.0 -->
  <xs:import namespace="http://www.opengis.net/gml"
    schemaLocation="H:/Diplomarbeit/GMLSML/GML/3.1.1/base/gml.xsd"/>
  <!-- import sensorML-->
```

```
<xs:import namespace="http://www.opengis.net/sensorML"
  schemaLocation="H:/Diplomarbeit/GMLSML/SML/base.xsd"/>
<!-- import FilterEncoding -->
<xs:import namespace="http://www.opengis.net/ogc"
  schemaLocation="H:/Diplomarbeit/GMLSML/Filter/1.1.0/filter.xsd"/>
<!-- ********************************************************************* -->
<!-- * R E Q U E S T M E S S A G E S * -->
<!-- ********************************************************************* -->
<!--GetCapabilities-->
<xs:element name="GetCapabilities" type="scs:GetCapabilitiesType"/>
<!--DescribePlatform-->
<xs:element name="DescribePlatform" type="scs:DescribePlatformType"/>
<!--DescibeSensor-->
<xs:element name="DescribeSensor" type="scs:DescribeSensorType"/>
<!--GetObservation-->
<xs:element name="GetObservation" type="scs:GetObservationType"/>
<!--SubscribeObservation-->
<xs:element name="SubscribeObservation" type="scs:SubscribeObservationType"/>
<!--UnsubscribeObservation-->
<xs:element name="UnsubscribeObservation"
  type="scs:UnsubscribeObservationType"/>
<!--SubscribeContinuousEvent-->
<xs:element name="SubscribeContinuousEvent"
    type="scs:SubscribeContinuousEventType"/>
<!--UnsubscribeContinuousEvent-->
<xs:element name="UnsubscribeContinuousEvent"
    type="scs:UnsubscribeContinuousEventType"/>
<!-- ********************************************************************* -->
<!-- * R E S P O N S E S * -->
<!-- ********************************************************************* -->
<!--DescribePlatformResponse-->
<xs:element name="DescribePlatformResponse"
  type="scs:DescribePlatformResponseType"/>
<!--DescribeSensorResponse-->
<xs:element name="DescribeSensorResponse"
  type="scs:DescribeSensorResponseType"/>
<!--GetObservationResponse-->
<xs:element name="GetObservationResponse"
  type="scs:GetObservationResponseType"/>
<!--SubscribeObservationResponse-->
<xs:element name="SubscribeObservationResponse"
  type="scs:SubscribeObservationResponseType"/>
<!--UnsubscribeObservationResponse-->
<xs:element name="UnsubscribeObservationResponse"
  type="scs:UnsubscribeObservationResponseType"/>
<!--SubscribeContinuousEventResponse-->
<xs:element name="SubscribeContinuousEventResponse"
  type="scs:SubscribeContinuousEventResponseType"/>
```

```
  <!--UnsubscribeContinuousEventResponse-->
  <xs:element name="UnsubscribeContinuousEventResponse"
    type="scs:UnsubscribeContinuousEventResponseType"/>
  <!-- ********************************************************************* -->
  <!-- * T Y P E S * -->
  <!-- ********************************************************************* -->
  <!-- ======================================== -->
  <!-- Message Types -->
  <!-- ======================================== -->
  <!-- GetCapabilitiesType-->
<xs:complexType name="GetCapabilitiesType">
<xs:attribute name="version" type="xs:string" use="optional"
    default="0.9.1"/>
<xs:attribute name="service" type="xs:string" use="required"
    fixed="SCS"/>
</xs:complexType>
<!-- DescribePlatformType -->
<xs:complexType name="DescribePlatformType">
    <xs:sequence>
      <xs:element name="TypeName" type="xs:string" minOccurs="0"
                  maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="version" type="xs:string" use="optional"
    default="0.9.1"/>
    <xs:attribute name="service" type="xs:string" use="required"
                  fixed="SCS"/>
    <xs:attribute name="outputFormat" type="xs:string" use="optional"
                  fixed="SensorML"/>
  </xs:complexType>
  <!-- DescribeSensorType -->
  <xs:complexType name="DescribeSensorType">
    <xs:sequence>
      <xs:element name="TypeName" type="xs:string" minOccurs="0"
                  maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="version" type="xs:string" use="optional"
                  default="0.9.1"/>
    <xs:attribute name="service" type="xs:string" use="required"
                  fixed="SCS"/>
    <xs:attribute name="outputFormat" type="xs:string" use="optional"
                  fixed="SensorML"/>
  </xs:complexType>
  <!--GetObservationType-->
  <xs:complexType name="GetObservationType">
    <xs:sequence>
      <xs:element name="BoundingBox" type="gml:EnvelopeType"/>
      <xs:element name="time" type="scs:ObservationTimeType"
                  minOccurs="0"
```

```
                maxOccurs="unbounded"/>
    <xs:element name="platformID" type="xs:string" minOccurs="0"
                maxOccurs="unbounded"/>
    <xs:element name="sensorID" type="xs:string" minOccurs="0"
                maxOccurs="unbounded"/>
    <xs:element ref="ogc:Filter" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="version" type="xs:string" use="optional"
                default="0.9.1"/>
  <xs:attribute name="service" type="xs:string" use="required"
                fixed="SCS"/>
  <xs:attribute name="outputFormat" type="scs:OutputFormatType"
                use="optional" default="OM"/>
</xs:complexType>
<!--SubscribeObservationType-->
<xs:complexType name="SubscribeObservationType">
  <xs:annotation>
    <xs:documentation>specifies the subscription condition:
      Period - specifies the period for time based subscriptions in minutes
      MaxAllowedDelay - specifies the maximum allowed delay for event based
                        subscriptions in minutes
     </xs:documentation>
   </xs:annotation>
   <xs:sequence>
   <xs:element name="BoundingBox" type="gml:EnvelopeType"/>
   <xs:element name="platformID" type="xs:string" minOccurs="0"
               maxOccurs="unbounded"/>
   <xs:element name="sensorID" type="xs:string" minOccurs="0"
               maxOccurs="unbounded"/>
   <xs:element ref="ogc:Filter" minOccurs="0"/>
   <xs:element name="CallbackServerURL" type="xs:anyURI"/>
   <xs:choice>
     <xs:element name="Period" type="xs:double"/>
     <xs:element name="MaxAllowedDelay" type="xs:double"/>
   </xs:choice>
 </xs:sequence>
 <xs:attribute name="version" type="xs:string" use="optional"
               default="0.9.1"/>
 <xs:attribute name="service" type="xs:string" use="required"
               fixed="SCS"/>
 <xs:attribute name="outputFormat" type="scs:OutputFormatType"
               use="optional" default="OM"/>
</xs:complexType>
<!--UnsubscribeObservationType-->
<xs:complexType name="UnsubscribeObservationType">
  <xs:annotation>
    <xs:documentation>
      specifies the subscriptionID for which the user wants to unsubscribe
```

```
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="SubscriptionID" type="xs:unsignedLong"/>
  </xs:sequence>
  <xs:attribute name="version" type="xs:string" use="optional"
            default="0.9.1"/>
  <xs:attribute name="service" type="xs:string" use="required"
            fixed="SCS"/>
</xs:complexType>
<!--SubscribeContinuousEventType-->
<xs:complexType name="SubscribeContinuousEventType">
  <xs:annotation>
    <xs:documentation>specifies the subscription condition:
      Period - specifies the period for time based subscriptions in minutes
      MaxAllowedDelay - specifies the maximum allowed delay for event based
                        subscriptions in minutes
     </xs:documentation>
   </xs:annotation>
   <xs:sequence>
   <xs:element name="BoundingBox" type="gml:EnvelopeType"/>
   <xs:element name="platformID" type="xs:string" minOccurs="0"
               maxOccurs="unbounded"/>
   <xs:element name="sensorID" type="xs:string" minOccurs="0"
               maxOccurs="unbounded"/>
   <xs:element ref="ogc:Filter" minOccurs="0"/>
   <xs:element name="CallbackServerURL" type="xs:anyURI"/>
   <xs:element name="Period" type="xs:double"/>
   <xs:element name="MaxAllowedDelay" type="xs:double"/>
 </xs:sequence>
 <xs:attribute name="version" type="xs:string" use="optional"
               default="0.9.1"/>
 <xs:attribute name="service" type="xs:string" use="required"
               fixed="SCS"/>
 <xs:attribute name="outputFormat" type="scs:OutputFormatType"
               use="optional" default="OM"/>
</xs:complexType>
<!--UnsubscribeContinuousEventType-->
<xs:complexType name="UnsubscribeContinuousEventType">
  <xs:annotation>
    <xs:documentation>
      specifies the subscriptionID for which the user wants to unsubscribe
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="SubscriptionID" type="xs:unsignedLong"/>
  </xs:sequence>
  <xs:attribute name="version" type="xs:string" use="optional"
```

```
                default="0.9.1"/>
  <xs:attribute name="service" type="xs:string" use="required"
                fixed="SCS"/>
</xs:complexType>
<!-- ========================================== -->
<!-- Response Types -->
<!-- ========================================== -->
<!-- GetCapabilitiesResponseType-->
<xs:complexType name="GetCapabilitiesResponseType">
  <xs:sequence>
    <xs:element ref="sim:OGC_Capabilities"/>
  </xs:sequence>
</xs:complexType>
<!--DescribePlatformResponseType-->
<xs:complexType name="DescribePlatformResponseType">
  <xs:sequence>
    <xs:element ref="sml:Platform" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<!--DescribePlatformSensorType-->
<xs:complexType name="DescribeSensorResponseType">
  <xs:sequence>
    <xs:element ref="sml:Sensor"/>
  </xs:sequence>
</xs:complexType>
<!--GetObservationResponseType-->
<xs:complexType name="GetObservationResponseType">
  <xs:sequence>
    <xs:element ref="om:RichObservation"/>
  </xs:sequence>
</xs:complexType>
<!--SubscribeContinuousEventResponseType -->
<xs:complexType name="SubscribeContinuousEventResponseType">
  <xs:annotation>
    <xs:documentation>
      unique subscription id, provided by the CSCS
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="SubscriptionID" type="xs:unsignedLong"/>
    <xs:element name="OutputFormat" type="scs:OutputFormatType"
                default="OM"/>
  </xs:sequence>
</xs:complexType>
<!--UnsubscribeContinuousEventResponseType-->
<xs:complexType name="UnsubscribeContinuousEventResponseType">
  <xs:sequence>
    <xs:element ref="scs:_UnsubscribeContinuousEventRespContent"/>
```

```
    </xs:sequence>
  </xs:complexType>
  <!--SubscribeResponseType -->
  <xs:complexType name="SubscribeContinuousEventResponseType">
    <xs:annotation>
      <xs:documentation>
        unique subscription id, provided by the CSCS
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="SubscriptionID" type="xs:unsignedLong"/>
      <xs:element name="OutputFormat" type="scs:OutputFormatType"
                  default="OM"/>
    </xs:sequence>
  </xs:complexType>
  <!--UnsubscribeContinuousEventResponseType-->
  <xs:complexType name="UnsubscribeContinuousEventResponseType">
    <xs:sequence>
      <xs:element ref="scs:_UnsubscribeContinuousEventRespContent"/>
    </xs:sequence>
  </xs:complexType>
  <!-- ========================================== -->
  <!-- Further Types & Elements -->
  <!-- ========================================== -->
  <!--ObservationTime-->
  <xs:complexType name="ObservationTimeType">
    <xs:choice>
      <xs:element ref="gml:TimeInstant"/>
      <xs:element ref="gml:TimePeriod"/>
    </xs:choice>
  </xs:complexType>
  <!-- OutputFormat-->
  <xs:simpleType name="OutputFormatType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="OM"/>
      <xs:enumeration value="GeoTIFF"/>
      <xs:enumeration value="JPEG"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="_UnsubscribeObRespContentType"
                  abstract="true"/>
  <xs:element name="_UnsubscribeObRespContent"
              type="scs:UnsubscribeObRespContentType"
              abstract="true"/>
  <xs:element name="SimpleUSORespContent" type="xs:string"
              substitutionGroup="scs:UnsubscribeObRespContent"/>
</xs:schema>
```