Entwicklung eines Telemetriesystems zur Erfassung von Prozessdaten im Feld

Ingenieurfakultät Bau Geo Umwelt
der Technischen Universität München
Lehrstuhl für Geoinformatik
Univ.-Prof. Dr. rer. nat. Thomas H. Kolbe

Masterarbeit

vorgelegt von:	Jacky Dezhi Fu
Studiengang:	Geodäsie und Geoinformation
Betreuer:	Dipl.-Geogr. Maximilian Sindram
	M.Sc. Thomas Machl
Bearbeitungszeitraum:	05.02.2016 - 30.09.2016

Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne unzulässige fremde Hilfe angefertigt habe. Die verwendeten Quellen sind im Literaturverzeichnis vollständig aufgeführt.

München, 24.08.2016

Jacky Dezhi Fu

Kurzfassung

In dieser Masterarbeit wird die Entwicklung eines Telemetriesystems zur Erfassung von landwirtschaftlichen Prozessdaten beschrieben und untersucht, wie diese mittels dem Framework *Sensor Web Enablement* (SWE) des *Open Geospatial Consortium* (OGC) über einen Webdienst, den *Sensor Observation Service* (SOS), verfügbar gemacht werden können. Es wird zum einen auf bereits vorhandene Telemetriesysteme kommerzieller Hersteller und deren Nachteile eingegangen, woraus die Motivation für die Entwicklung des vorliegenden Systems geschöpft wird. Zum anderen dient die Entwicklung und der Einsatz des Systems als Anwendungsfall eines mobilen Sensors für die SWE-Standards *Observations and Measurements* (O&M) und *Sensor Model Language* (SensorML). Es wird beschrieben, wie das Telemetriesystem als mobiles Sensorsystem mittels dieser Standards modelliert werden kann, um die erfassten Daten auf eine SOS-Instanz zu übertragen und somit Nutzern aus unterschiedlichen Bereichen zugänglich zu machen. In diesem Zusammenhang werden Möglichkeiten der Georeferenzierung eines solchen Systems geprüft. Auf Basis der Ergebnisse dieser Recherchen wird eine Modellierung des Systems mittels dieser Standards vorgenommen und in XML-Dokumenten realisiert. Einen weiteren wesentlichen Bestandteil dieser Arbeit bildet die Entwicklung eines Proxys, um TCP-basierte Datenübertragung an einen SOS zu ermöglichen. Die dazu notwendige Ausgestaltung der Prozesskette wird entworfen. Weiterhin wird in der Arbeit die Rolle des OGC-Standards *Moving Features* im Zusammenhang von SWE-basierten Daten diskutiert. Im Rahmen dessen werden Möglichkeiten aufgezeigt, wie Daten von jeweils einer Domäne in die andere überführt werden können. Nach diesen Arbeitsschritten folgt die Beschreibung der Zusammenstellung des Datenloggers mittels leicht erhältlicher, kostengünstiger Komponenten. Es werden die Komponenten zur Erfassung von CAN- und GPS-Daten sowie zur Übertragung dieser Daten über GSM vorgestellt. Daran knüpft die Beschreibung der Datenlogger-Prozesskette an. Im Anschluss werden die Arbeitsschritte auf Serverseite erläutert. In der darauf folgenden Systemevaluierung werden von einem Traktor erfasste CAN-Daten präsentiert. Es wird eine Möglichkeit aufgezeigt, wie das System ohne eine physikalische Abhängigkeit – also ohne ein Fahrzeug – erprobt und weiterentwickelt werden kann. Abschließend wird auf den Stand aktueller Client-Anwendungen zur Darstellung von Daten mobiler Sensoren eingegangen und noch vorhandene Einschränkungen sowohl dieser Anwendungen als auch des Telemetriesystems aufgezeigt.

Abstract

In this master's thesis, the development of a telemetry system for capturing agricultural process data is described, and it is examined how that data can be distributed via the *Sensor Observation Service* (SOS), using the framework *Sensor Web Enablement* (SWE) of the *Open Geospatial Consortium* (OGC). On the one hand, current commercial systems and their disadvantages are analysed in order to clarify the motivation behind this attempt at developing a new system. On the other hand, the development and deployment of the system serves as a use case of a mobile sensor for the SWE standards *Observations and Measurements* (O&M) and *Sensor Model Language* (SensorML). It is investigated how the telemetry system can be modelled as a mobile sensor system using these standards to transmit the captured data to an SOS instance and to make the data available to users of different fields. In this context, ways for georeferencing such a system are reviewed. Based on these research results, a system modelling using these standards is carried out, and corresponding XML documents are implemented. An additional essential part of this thesis is the development of a proxy, in order to enable TCP based data transfer to an SOS. Hence, the necessary design of the process chain is outlined. Furthermore, the role of the OGC standard *Moving Features* with regards to SWE based data is discussed in this work. Possibilities for transforming data from one domain to another and vice versa are demonstrated. After these steps, the assembly of the data logger using readily available low-cost components is described. Subsequently, the server-side working steps are explained. In the following system evaluation, CAN data captured from a farm tractor are presented. A possibility to further test and develop the system without a physical dependency, i. e. a vehicle, is shown. Finally, the status quo of current client applications for visualisation of mobile sensor data, as well as still existing restrictions of both those applications and the telemetry system, are looked into.

Inhaltsverzeichnis

1 Einleitung

1.1 Telemetriesysteme in der Landwirtschaft

Fahrzeuge werden gegenwärtig mit Sensoren und Datenloggern ausgestattet, um Informationen über den Fahrzeugzustand und -betrieb zu einem Zeitpunkt oder über einen Zeitraum zu erfassen, zu speichern und abzurufen. Solche Prozessdaten sind relevant für Bereiche wie etwa Dokumentation, Fernüberwachung, Flottenmanagement, Prozessoptimierung, Ressourcennutzung, Service Monitoring und Umweltschutz. Es wird an dieser Stelle darauf hingewiesen, dass in der Arbeit die Begriffe Prozessdaten und Fahrzeugdaten als synonym betrachtet werden. Die Gesamtheit aller an der zur Erfassung und drahtlosen Übertragung von solchen Daten beteiligten Komponenten wird im Begriff Telemetriesystem zusammengefasst.

Mittels eines solchen Systems kann über die ISOBUS-Schnittstelle (eigentlich ISO 11783, üblicherweise jedoch als ISOBUS bezeichnet) eines Traktors, die aus dem LBS (Landwirtschaftliches BUS-System) hervorging (siehe dazu Abb. 1.1), eine Vielzahl unterschiedlicher Daten des Fahrzeugs erfasst werden. Diese reichen von Kraftstoffverbrauch und Motordrehzahl über Hydraulikdrücke und Ventileinstellungen bis hin zu Einstellwerten des Pfluges und Füllstand des Spritztanks. Verfügt das System über GNSS-Empfänger (*Global Navigation Satellite System*), können die Fahrzeugdaten mit Positionsinformationen verknüpft werden. Dies gibt beispielsweise Aufschluss darüber, an welchen Orten, bei welchen Tätigkeiten und wie viel Kraftstoff verbraucht wird. Solche Informationen liefern Hinweise, an welchen Stellen Traktor und Geräte neu ausgerichtet werden müssen und die Fahrweise optimiert werden kann, um eine effiziente Nutzung des Kraftstoffs zu ermöglichen.

Die große Bedeutung von solchen Systemen wird durch entsprechende Produktsparten von Landmaschinenherstellern deutlich. Diese Systeme sind jedoch kostspielig und weisen eine eingeschränkte bis nicht vorhandene Erweiterbarkeit auf. Sind beispielsweise zusätzliche Funktionen erforderlich, z. B. die Georeferenzierung der Daten durch das GPS (*Global Positioning System*) oder die Übertragung der Daten mittels des GSM (*Global System for Mobile Communications*), ist die Anpassung eines bereits vorhandenen Systems oft nicht möglich und die Anschaffung eines neuen Geräts vonnöten. Die Verfügbarkeit der Daten ist ebenfalls an bestimmte Systeme gebunden. Zudem treten hinsichtlich weiterer Aspekte, wie z. B. dem zeitlich begrenzten Zugang und der verfügbaren Dateiformate der erfassten Daten, Einschränkungen auf. Schließlich sind diese Systeme oft mit Maschinen desselben Herstellers gekoppelt und funktionieren daher nicht oder nur eingeschränkt mit anderen Maschinen.

Aufgrund der o. g. Einschränkungen der proprietären Systeme scheint eine Nutzung von Telemetriesystemen und der zugehörigen Daten hauptsächlich den Landwirten und Herstellern vorbehalten. Der Zugang zu solchen Daten und die Nutzung durch nicht kommerzielle Anwender, wie z.B. aus der Wissenschaft, Interessenverbänden oder staatlichen Institutionen, werden wegen der genannten Eigenschaften der proprietären Systeme erschwert. Das aus solchen Daten potentielle Wissen über Prozesse, deren Produkte und Ergebnisse betrifft letztendlich uns alle. Daher ist diese Thematik für viele im Hinblick auf die Bereiche Forschung, Nachhaltigkeit, Umweltschutz, Verbraucherschutz u. a. von großem Belang. Daher besteht an Ansätzen und Entwicklungen, welche die o. g. Fahrzeugdaten in einer geeigneten Art und Weise für verschiedene Nutzergruppen bereitstellen, ein großes Interesse.

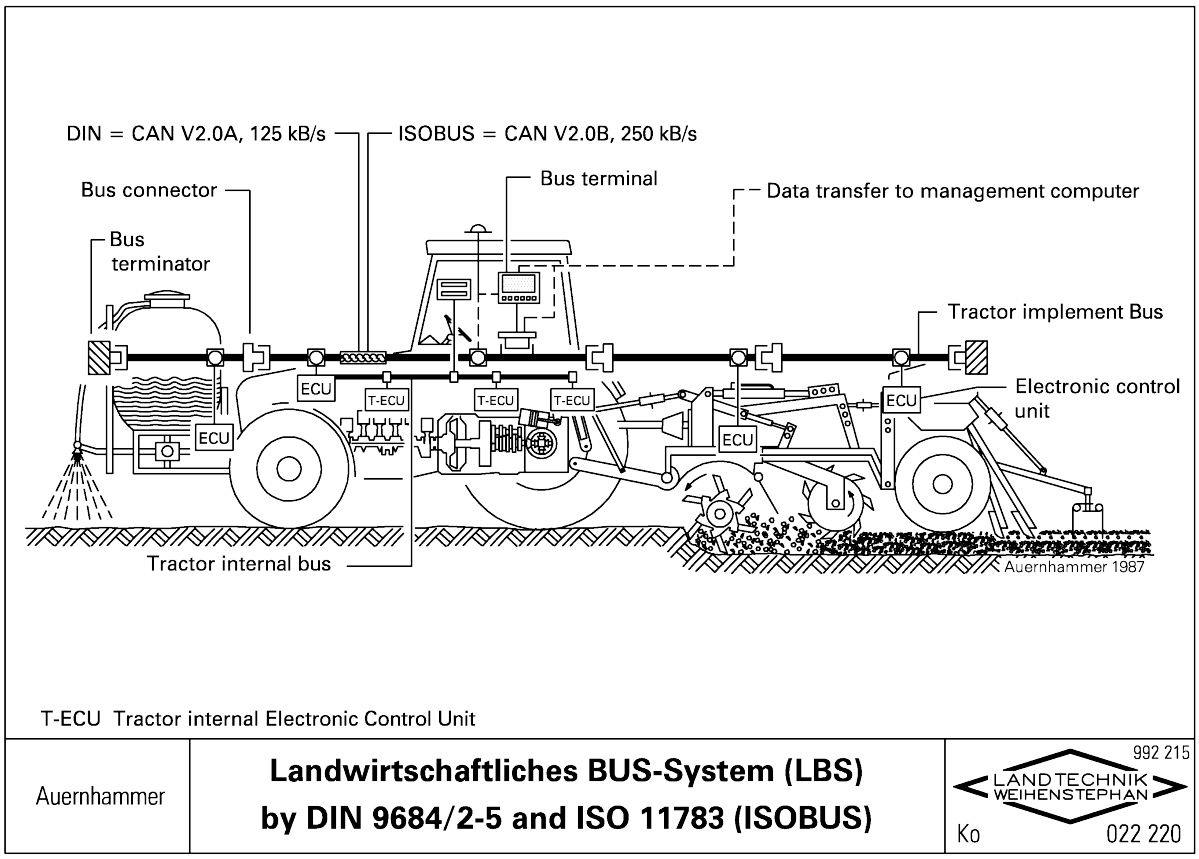


Abbildung 1.1: Landwirtschaftliches BUS-System aus [Auernhammer, 2002b]

1.2 Problemstellung

Wie ein Telemetriesystem in der Landwirtschaft für die Nutzung durch unterschiedliche Anwender entwickelt werden kann, ist die zentrale Fragestellung dieser Arbeit. Darauf basierende weitere Fragen betreffen, neben der geeigneten Systemzusammenstellung, vor allem das Ermöglichen der Datenbereitstellung im interoperablen Kontext. Gegenwärtig existieren Werkzeuge, Geräte und Methodiken, die einzelne Aufgaben wahrnehmen können, darunter Positionierung, Datenübertragung und Datenbereitstellung. Die Schwierigkeit besteht darin, diese Mittel geeignet zu verknüpfen. Denn, anders als die Komponenten und Software bei Herstellersystemen, wurden diese Mittel nicht im Vorfeld aufeinander abgestimmt und sind nicht grundsätzlich für einen gemeinsamen Einsatz vorgesehen. Es ist daher zu bestimmen, wie aktuelle, am Markt erhältliche Komponenten mit gegenwärtigen, wissenschaftlichen Methoden aus der Geoinformatik verknüpft werden müssen, um die zentrale Fragestellung zu beantworten. Zur Einbettung des Systems in ein interoperables Rahmenkonzept ist der aktuelle Stand der wissenschaftlichen Techniken in diesem Bereich zu eruieren. Die entsprechenden verfügbaren Mittel, wie Standards und Software, werden untersucht und auf das hier vorliegende Szenario eines mobilen Systems angewendet. Die Fragestellung hierbei ist die geeignete Modellierung für einen mobilen Sensor, der kontinuierlich Daten georeferenziert und diese an einen Server sendet. Die zur Verfügung stehenden wissenschaftlichen Methoden erlauben eine nahezu beliebige Modellierung von Sensoren und deren Beobachtungen. Dies ist sinnvoll, da damit viele Anwendungsszenarien in der Realität abgedeckt werden können. Diese Modellierung geeignet vorzunehmen, stellt jedoch eine konzeptionelle Herausforderung dar und beinhaltet die Beschreibung des Telemetriesystems mittels der in den Standards gegebenen Mittel.

1.3 Ziele

Ausgehend von den vorangegangenen Betrachtungen können für das Telemetriesystem folgende Ziele festgelegt werden: Das System soll aus kostengünstiger, gängiger sowie leicht erhältlicher Hardware bestehen, und der Betrieb soll mit freier, offener Software erfolgen. Es soll automatisch die Datenerfassung, Positionierung und Datenübertragung vornehmen können. Die Wahl des Rechners für das System fiel auf einen *Raspberry Pi*, einen Miniaturrechner, der die nötige Rechenleistung für die vorgesehene Datenerfassung aufweist, und mit einem Linux-Betriebssystem betrieben wird. Zudem werden weitere benötigte Komponenten an den Rechner angeschlossen und konfiguriert, sodass letztendlich ein kompaktes System entsteht und der Datenlogger Fahrzeugdaten erfassen, georeferenzieren und drahtlos auf einen Server übertragen kann. Die interoperable Bereitstellung der Daten soll durch die Einbindung des Systems und der Daten in einen SOS (*Sensor Observation Service*) [Bröring et al., 2012] auf einem stationären Rechner, der als Server fungiert, gewährleistet werden. Dafür ist die Vorverarbeitung der ankommenden Daten auf der Serverseite auszuarbeiten und umzusetzen. Für die Anbindung des Systems ist zu untersuchen, wie die Daten und das Telemetriesystem mit den im SWE (*Sensor Web Enablement*) beinhalteten Standards geeignet modelliert werden können. Voraussetzung dafür ist die Kodierung der Telemetriedaten und des Telemetriesystems mittels der Standards O&M (*Observations and Measurements*) [Cox, 2013] und SensorML (*Sensor Model Language*) [Botts und Robin, 2014] des Frameworks SWE des OGC (*Open Geospatial Consortium*). Ziel ist es hierbei, die Daten und das System für die Bereitstellung von Fahrzeugdaten geeignet mit den o. g. Standards zu modellieren und diese, während des laufenden Betriebs des Telemetriesystems, in einem SOS zu speichern. Die Wahl des SOS fiel auf die SOS-Implementierung des Unternehmens 52°North. Diese Ausprägung eines SOS gilt als eine der bekannten Referenzimplementierungen. Sind diese Ziele erreicht, wird das System zusammen mit dem SOS erprobt und die Ergebnisse werden ausgewertet. Anhand der Auswertung soll diskutiert werden, inwieweit der Ansatz eines Telemetriesystems im interoperablen Kontext gegenwärtig als erfolgreich eingeschätzt werden kann.

1.4 Forschungsgegenstand der Arbeit

Eine aus der Umsetzung der Ziele zu gewinnende Erkenntnis betrifft die Frage der Datenübertragung vom Datenlogger zum SOS. Das Senden von Daten an den SOS kann durch das HTTP (*Hytertext Transfer Protocol*) mittels der Methode POST geschehen. Der Datenlogger kommuniziert, aufgrund der zu erwartenden großen räumlichen Distanzen zwischen Fahrzeug und Server, über Mobilfunk mit dem Server. Jedoch ist die Unterstützung des HTTP insbesondere für günstige, kompakte Kommunikationsmodule nicht immer gegeben, wohl aber eine Unterstützung des TCP (*Transmission Control Protocol*). Es ist daher ein Weg zu finden, diese Kommunikationslücke zu überwinden, um einen möglichst großen Nutzerkreis der Sensor-Community für den Einsatz von Sensorplattformen im Framework des SWE zu erreichen.

Eine weitere Frage beschäftigt sich mit der Konzeption eines mobilen Sensorsystems innerhalb des SWE-Frameworks. Gerade durch die von den OGC-Standards gegebenen Möglichkeiten, Systeme, Sensoren und deren Beobachtungen beliebig modellieren zu können, ist eine geeignete Modellierung der Beobachtungen einer mobilen Sensorplattform zu finden. Beispielsweise erlauben die Standards sowohl die Georeferenzierung des Systems als auch der Beobachtungen. Im Falle einer stationären Sensorplattform findet die Georeferenzierung des Systems einmalig zum Zeitpunkt der Sensorregistrierung statt. Dies ist ausreichend, da sich die Koordinaten des Systems, nach Inbetriebnahme, nicht mehr ändern. Bei einer mobilen Sensorplattform ändert sich die Position ständig. Es ist daher herauszufinden, inwiefern sich eine Georeferenzierung des Systems von der Georeferenzierung der Beobachtungen im Kontext einer mobilen Sensorplattform unterscheidet und welche Vorgangsweise zu bevorzugen ist.

Eine andere Feststellung ist im Hinblick auf die Modellierung der Daten im gegenwärtigen Zustand aktueller SOS-Clients zu treffen. So ist es beispielsweise mit dem gegebenen JS-Client (JavaScript) von 52°North aktuell nicht möglich, Sensordaten mobiler Plattformen geeignet zu visualisieren. Diese Tatsache rührt von mehreren Umständen her: Im Kontext des SWE-Frameworks bleibt die Häufigkeit des Einsatzes von mobilen Sensoren hinter der von stationären noch zurück, und der JS-Client zielt bisher lediglich auf eine Visualisierung von Zeitreihen von stationären Sensoren. Anders als bei der Visualisierung von stationären Daten kämen mit jeder Messung durch die Sensoren eine neue Position und damit ein neuer Marker pro Messwert auf einer Karte hinzu. Nach einer Korrespondenz am 15.02.2016 mit 52°North-Entwickler Carsten Hollmann [Hollmann, 2016] haben sich jedoch Möglichkeiten herauskristallisiert, die Daten für eine künftige Version des JS-Clients geeignet zu modellieren.

Schließlich stellt sich auch die Frage, wie der Einsatz des OGC-Standards *Moving Features* [Asahara et al., 2015b] im Rahmen dieser Arbeit erfolgen kann. Ziel dieses Standards ist es, die Datenflut aus dynamischen Anwendungen unterschiedlichen Domänen interoperabel verfügbar zu machen. Die Notwendigkeit eines solchen Standards ergibt sich aus der zunehmenden Technologisierung bewegter Objekte. Daraus resultieren zunehmende Mengen an Bewegungsdaten in verschiedenen Anwendungsszenarien [Asahara, Hayashi et al., 2015]. Für enorme Datenmengen und zeitkritische Anwendungen können die Bewegungsdaten als CSV-Werte (*comma-separated values*) kodiert werden [Asahara et al., 2015a]. Gegenwärtig stellt dieser Standard lediglich

Schemata zur Speicherung und zum Austausch von Bewegungsdaten unterschiedlicher Herkunft zur Verfügung. Werkzeuge zur Generierung und Visualisierung von solch kodierten Daten existieren zum jetzigen Zeitpunkt nicht. Der Einsatz des Standards bietet sich daher an, um diesen an dem hier behandelten Anwendungsfall weiter zu erproben. Die Frage ist daher, wie eine Konvertierung von O&M-kodierten Beobachtungen und SensorML-kodierten mobilen Sensoren in das im *Moving Features* definierte Format aussehen kann und umgekehrt.

2 Stand der Technik und Forschung

2.1 Existierende Telemetriesysteme

Im Folgenden finden sich einige Beispiele von bereits existierenden kommerziellen Systemen von Fahrzeugherstellern. Sie sollen im Detail aufzeigen, welche Eigenschaften die Herstellersysteme aufweisen und eine Übersicht über die am Markt erhältlichen Produkte geben.

Mit „Variotronic“ [AGCO GmbH, Fendt-Marketing, 2015b] stellt der Hersteller für Landmaschinen Fendt ein System für Traktoren bereit, welches den Traktor und das Vorgewende steuert. Die Steuerung des Traktors kann wahlweise durch eine Spurführung unterstützt werden. Es existiert ein Terminal, von welchem die Schlepperbedienung und Gerätesteuerung getätigt werden kann. Dazu werden zwei unterschiedlich große Ausführungen des Terminals angeboten, welche unterschiedlichen Anforderungen entsprechen. Die voll ausgestattete Version ermöglicht die Erweiterung des Terminals um Funktionen wie Datenaufzeichnung, Datenübertragung, Auftrags-, Flotten- und Schlagkarteimanagement. Die Dokumentationsfunktion ermöglicht die Erstellung von Aufträgen und Übertragung dieser per Bluetooth oder Mobilfunk von einem PC auf das Terminal. Dieses Produktmerkmal liegt wiederum in zwei Versionen unterschiedlichen Funktionsumfangs vor. In der Basis-Version fehlt die Möglichkeit der Übertragung von Daten per Mobilfunk, zudem werden weder GNSS-Positionsdaten noch georeferenzierte Maschinendaten erfasst. Der Katalog an erfassten Maschinendaten ist explizit festgelegt, mögliche andere Arten von Maschinendaten werden nicht angeboten. Die Positions- und Maschinendaten stehen lediglich bei der erweiterten Version zur Verfügung, und werden per Mobilfunk auf einen vom Hersteller bereitgestellten Server geschickt. Die Daten können per Internetverbindung auf Endgeräte abgerufen und dargestellt werden. Die Telemetrie und Verarbeitung der gesendeten Daten übernimmt eine Software [AGCO GmbH, Fendt-Marketing, 2015a]. Diese steht in vollem Funktionsumfang nur für Landmaschinen im AGCO-Mutterkonzern (*Allis-Gleaner Corporation*) zur Verfügung. Landmaschinen anderer Hersteller können zwar mit dieser Software nachgerüstet werden, jedoch nur mit eingeschränkter Telemetriefunktion.

„Agri Doc“ ist das Telemetriesystem des Unternehmens Agri Con [Krieger, 2015]. Das System umfasst eine Software zur Datenerfassung und Verwaltung, sowie einen Datenlogger, der in drei verschiedenen Ausführungen vorliegt. Selbige unterscheiden sich in Leistungsumfang und Preis. Ein Datenlogger ist dabei nicht erweiterbar. Um eine neue Funktionalität zu erhalten, so muss ein neuer Datenlogger mit der gewünschten Funktion erworben werden. Die Nutzung der Software ist mit einer Lizenz verbunden, welche monatlich abgerechnet wird [Agri Con GmbH, 2015]. Die Nutzungsgebühr wird abhängig von der Anzahl der genutzten Datenlogger zugunsten des

Kunden ermäßigt. Die Daten werden per Mobilfunk auf Server externer Dienstleister übertragen und von Agri Con verwaltet. Das System arbeitet laut dem Hersteller herstellerunabhängig und für alle Maschinen. Kundenspezifische Erweiterungen hinsichtlich Auswertung und Darstellung der Daten sind, nach Absprache, möglich.

„JDLink“ stellt die Telemetrielösung des Unternehmens Deere & Company dar [John Deere GmbH & Co. KG, 2016]. Auch hier gibt es eine Basisausführung, welche laut Hersteller Auskunft über Maschinenstandort und Arbeitsfortschritt, Maschinenlaufzeiten, Diebstahl-/Missbrauchsschutz und Diebstahlalarme sowie Wartungsplanung gibt. Die damit verbundenen Informationen sind jedoch mit einer Maschine eines anderen Herstellers nicht alle verfügbar. Diese Ausführung kann erweitert werden, um drahtlose Datenübertragung und Bildschirmfernzugriff zu ermöglichen.

2.2 Relevante Arbeiten

Exemplarisch sollen im Folgenden wichtige Arbeiten erwähnt werden, welche die Bereiche Telemetrie in der Landwirtschaft und ihre effiziente praktikable Umsetzung diskutieren. *Precision Farming* ist das häufigste Stichwort, welches im Zusammenhang mit der Integration von Informationstechnologien in der Landwirtschaft auftritt.

Auernhammer diskutiert das Verständnis des *Precision Farming* und stellt die damit verbundenen Herausforderungen in Aussicht [Auernhammer, 2001]. Er merkt an, dass die Aufgabe des *Precision Farming* von vielen zu kurz aufgegriffen und oft lediglich als standortbezogene Bewirtschaftung verstanden wird. Primär schätzt Auernhammer die Erwartungshaltung von Landwirten so ein, dass durch *Precision Farming* entweder bei gleichem Ertrag die verwendete Menge an Düngemittel reduziert oder bei gleicher Düngemittelmenge ein höherer Ertrag erzielt werden sollte – der Erhalt und Schutz der Pflanzen sei nebensächlich. Er selbst hingegen sieht *Precision Farming* als Ausgangspunkt für eine neue Diskussion über agrikulturelle Bewirtschaftung und spricht *Precision Farming* eine entscheidende Rolle in dieser Hinsicht zu. Die Änderungen nämlich, die durch eine effektive Bewirtschaftung entstünden, sollten Auernhammer zufolge die Nutzung des Bodens und des Landes selbst derart gestalten und überarbeiten, dass bei effektiver Landwirtschaft der Erhalt der Umwelt, ihre Variabilität und ihre Heterogenität in den Vordergrund gerückt werde.

Dies erörtert er ebenso in einem Konferenzbeitrag und geht dabei im Zuge dessen auf *Precision Farming* und dessen Potential für die Landwirtschaft in der Zukunft ein [Auernhammer, 2009]. Probleme – aber auch erhoffte Vorteile – der Nutzung von *Precision Farming* gingen laut Auernhammer von verschiedenen Gruppen aus, nämlich den Wissenschaftlern, den Herstellern sowie den Landwirten. Insbesondere in letzterer Gruppe sieht Auernhammer, im Hinblick auf höhere Erträge und eine Verbesserung des gesamten Arbeitsprozesses, eine große Bereitschaft und gesteigertes Interesse an der Nutzung von *Precision-Farming*-Technologien. Gleichzeitig konstatiert Auernhammer einen verhaltenen Umsetzungswillen der Landwirte, die bisherigen, von Herstellern angebotenen, unterschiedlichen Systeme einzusetzen. Dies sei auf die hohen Investitionskosten monetärer und arbeitstechnischer Art zurückzuführen. Standardisierte Systeme

ermöglichten zum einen die flexible Verwendung der *Precision-Farming*-Lösungen für unterschiedliche Landmaschinen. Zum anderen würden dadurch Datenaustausch und Schnittstellen vereinfacht, wenn nicht sogar erst ermöglicht. Auernhammer führt weiter an, dass Patente und Konkurrenzdruck die Hersteller dazu veranlassten, ihre Systeme proprietär zu entwickeln und durch herstellerspezifische Erweiterungen zu ergänzen, um sich gegenüber anderen Wettbewerbern zu profilieren. Dies führte in der wissenschaftlichen Gemeinschaft dazu, dass gewisse Daten der Forschung verwehrt blieben bzw. nicht ohne weiteres zugänglich seien. Von Landwirten hingegen werde jedoch eine einfache Lösung bevorzugt, mit welcher sie sich auf ihre eigentliche Arbeit konzentrieren könnten.

Demmel und Auernhammer [Demmel und Auernhammer, 1998] berichten vom Einsatz von GPS-Technologie und Datenverarbeitung im Zusammenhang mit einer Erntemaschine von Zuckerrüben, dem sog. Zuckerrübenroder. Durch die Visualisierung von Fahrwegen konnten Informationen über Platzierung der Rübenmiete und des Erntevorgangs gewonnen und eine positive Korrelation der Schlaglänge mit der Ernteleistung quantitativ beobachtet werden. Zudem bestehe durch den Einsatz eines Telemetriesystems die Möglichkeit den Arbeitsprozess automatisiert zu erfassen. Diese Daten könnten in Analysen zur Optimierung des Prozesses einfließen und stellten wertvolle Informationen für das Flottenmanagement bereit. Entsprechende Datenverarbeitung und Datenverwaltung vorausgesetzt, sei es somit möglich, den Arbeitsprozess in der Landwirtschaft effektiv zu planen, zu koordinieren und zu überwachen. Weitere Ansätze sehen Demmel und Auernhammer in der Verbesserung der Sensorik und im Einsatz kostengünstiger Datenlogger.

Rothmund et al. setzen *Precision Farming* in Beziehung zur automatischen Prozessdatenerfassung und diskutieren Nutzung sowie Bedeutung für die Feldarbeit von *Precision Farming* [Rothmund et al., 2002]. Das darin vorgestellte System stellt kein Telemetriesystem dar, da die aufgezeichneten Daten auf ein Speichermedium zwischengespeichert wurden. Dieses Speichermedium wurde später in einem nachfolgenden Schritt ausgelesen und die Daten in einer Datenbank gespeichert sowie ausgewertet. Nichtsdestotrotz wurden Positionen sowie Maschinendaten des Traktors ermittelt und somit eine automatisierte Datenerfassung betrieben. Die automatisierte Datenerfassung erlaube auch bei großer Arbeitsbelastung, mit welcher die Datenmenge positiv korreliert, eine lückenlose Dokumentation der Daten. Letztere ist bei manueller Eingabe nicht anzutreffen. Zudem weise die automatisierte Datenerfassung eine mit herkömmlichen Schlagkarteisystemen nicht vergleichbare, hohe räumliche sowie zeitliche Auflösung auf. Weiterhin zeichneten sich die Daten, aufgrund der automatisierten Erfassung, durch Authentizität aus und dienten als vertrauenswürdige Basis für die Evaluierung verschiedener Interessensgruppen wie Betriebspersonal, Entscheidungsträger und die breite Gesellschaft.

Eine genaue Dokumentation und Transparenz des Arbeitsprozesses schaffe auf der einen Seite Vertrauen bei Verbrauchern, da entsprechende Informationen zur Verfügung stünden und eingesehen werden könnten [Rothmund et al., 2003]. Auf der anderen Seite sei nach Rothmund et al. eine Dokumentation Voraussetzung für die Gewährung von Subventionen durch staatliche

Institutionen. So wäre es möglich, beispielsweise mit der automatisierten Erfassung landwirtschaftlicher Daten und unter Ermöglichung des Zugangs für jedermann zu diesen, sich über den Herstellungsprozess in der Landwirtschaft in quantitativer Hinsicht und unter Zuhilfenahme von Visualisierungsmethoden zu informieren [Auernhammer, 2002a].

3 Theoretische Grundlagen

Zur Entwicklung des in den vorherigen Abschnitten beschriebenen Telemetriesystems ist eine Betrachtung konzeptioneller Natur vorwegzunehmen. Diese Betrachtung soll auf für das System notwendige Konzepte und Grundlagen eingehen, um die Umsetzung des Systems und die Zusammenstellung der relevanten Komponenten nachvollziehbar zu machen.

3.1 Sensor Web Enablement

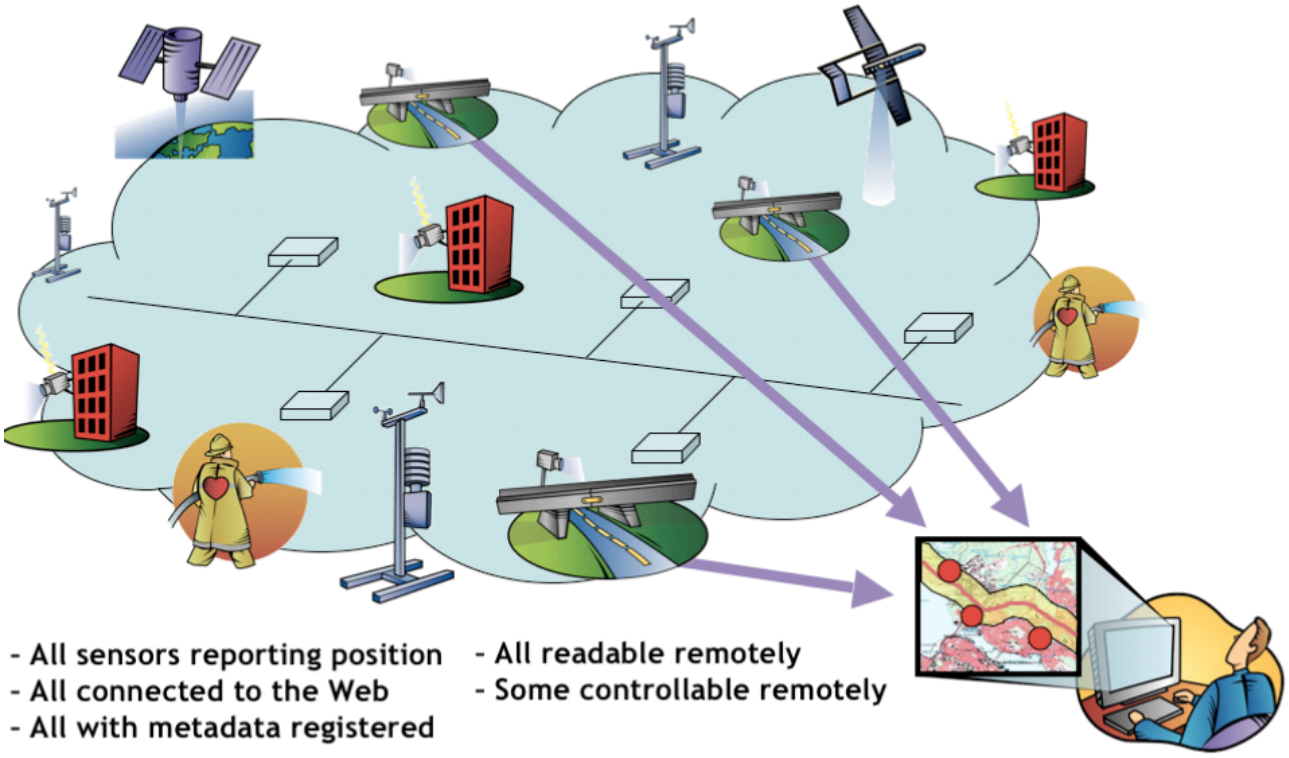


Abbildung 3.1: *Sensor Web* [Botts et al., 2008]

Das vorgesehene System sowie dessen erfasste Daten werden in das *Sensor Web* eingebunden (Abb. 3.1). Die OGC-Standards im Framework des SWE stellen Werkzeuge zur Konzeption und Realisierung eines *Sensor Web* und dessen Bestandteilen zur Verfügung [Bröring et al., 2011]. Nach Bröring et al. dient dieses interoperable Netzwerk als Plattform zur nahtlosen Sensorin-

tegration sowie standardisierten Kommunikation mit und zwischen unterschiedlichen Sensoren. Analog zu HTML-Seiten im *World Wide Web* finden sich somit im *Sensor Web* unterschiedliche Sensoren mit ihren Merkmalen, Erfassungsgrößen und zugehörigen Metadaten. In diesem können Sensoren registriert und Sensordaten abgefragt werden. Ein solch angereichertes *Sensor Web* kann unter Zuhilfenahme der SWE-Standards exploriert werden. Das SWE-Framework stellt standardisierte Werkzeuge zur Verfügung, um ein *Sensor Web* zu realisieren: Anwender können zum einen Webservices aufsuchen, um Sensorinformationen und Sensordaten abzufragen und neue Informationen einzufügen. Zum anderen können im *Sensor Web* weitere Dienste genutzt werden [Simonis und Echterhoff, 2011], um beispielsweise Sensoren aufzusuchen und diese zu beauftragen, bestimmte Messwerte zu liefern. Werden die gewünschten Messwerte von den Sensoren erfasst, werden die Nutzer dieser Sensoren benachrichtigt. Anhand dieser Gesichtspunkte wird die Relevanz der interoperablen Bereitstellung der generierten Prozessdaten des Telemetriesystems erkennbar. Durch diesen Schritt wird die Weiterverarbeitung und die Verwendung der Daten in anderen Anwendungsgebieten und Umgebungen erst ermöglicht. So wird die Bereitstellung georeferenzierter Prozessdaten für verschiedene Zielgruppen realisiert. Für die Abfrage, Speicherung und Bereitstellung von Prozessdaten sind hier drei Standards von besonderem Interesse. Diese werden in den folgenden Kapiteln näher erläutert.

3.2 Sensor Observation Service

Ein SOS stellt einen Webservice und Knoten im *Sensor Web* dar, der Informationen über Sensoren sowie deren Beobachtungen verfügbar macht. Zudem ist es möglich, neue Daten in einen SOS einzufügen; es wird dann von einem transaktionalen SOS gesprochen. Nach Botts et al. ist das Ziel des SOS-Konzepts, Sensoren und deren Beobachtungen in standardisierter Art und Weise verfügbar zu machen [Botts et al., 2008]. Dies soll für jegliche Sensoren gleich welcher Herkunft konsistent erfolgen. Voraussetzung dafür ist die Kodierung der Sensoren nach dem SensorML- und deren zugehörige Beobachtungen nach dem O&M-Standard. Ab der Version 2.0 lässt sich die Funktionalität des SOS in drei Gruppen einteilen: *Core*, *Extensions* und *Profiles*. Speziell für die vorliegende Arbeit relevant ist zum einen die Möglichkeit, Daten über die transaktionale Erweiterung *transactional extension* im SOS zu speichern. Zum anderen ist für die Speicherung und Abfrage georeferenzierter Beobachtungen das *Spatial Filtering Profile* des SOS 2.0 von Interesse.

3.2.1 SOS-Core

Laut dem SOS-Standard unterstützt jeder SOS folgende Operationen [Bröring et al., 2012]:

- *GetCapabilities* – stellt Metadaten und detaillierte Informationen über die im SOS verfügbaren Operationen bereit
- *DescribeSensor* – ermöglicht das Abfragen von Metadaten über die im SOS vorhandenen Sensoren

- *GetObservation* – stellt Beobachtungen der Sensoren zur Verfügung und ermöglicht es, diese nach räumlichen, zeitlichen und thematischen Kriterien zu filtern

Diese Operationen werden im Abschnitt *SOS Core requirements class* des SOS-Standard zusammengefasst. Die Datenabfrage durch einen SOS-Client wird im Standard durch ein Sequenzdiagramm (Abb. 3.2) dargestellt.

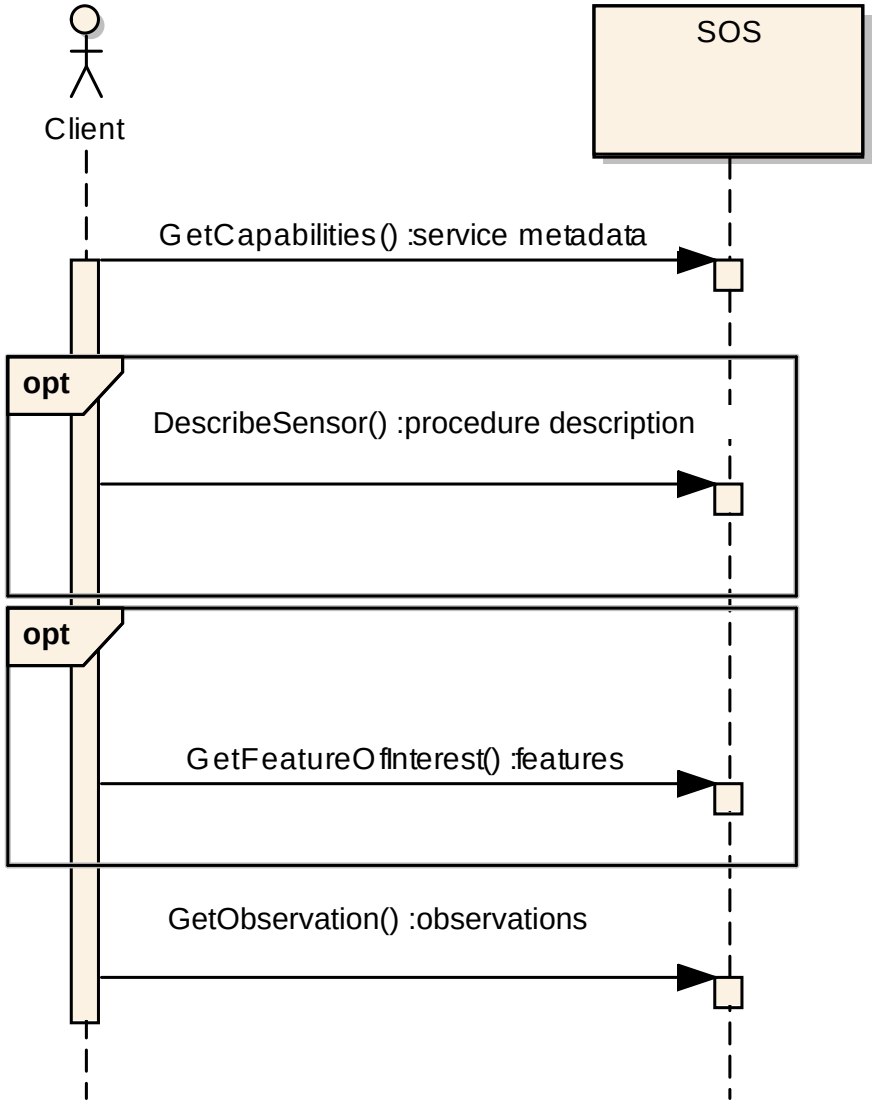


Abbildung 3.2: Ablauf der Datenabfrage im *Sensor Observation Service* nach [Bröring et al., 2012]

In Abb. 3.2 ist die Interaktion zwischen einer Client- und einer SOS-Instanz abgebildet. Die Kommunikation basiert auf dem Versenden von synchronen Nachrichten des Clients (von Client ausgehende Pfeile mit gefüllter Pfeilspitze), d. h. der Client wartet die Antwort des SOS ab, bevor der Client mit weiteren Aktionen fortfährt. Die so getätigten Anfragen werden

durch zurückgelieferte Antworten des SOS bestätigt (Die Antwort steht jeweils nach dem Doppelpunkt der Operationsbezeichnung in obiger Abbildung). Zunächst vergewissert sich der Client der Existenz verfügbarer Messdaten. Dazu führt er eine *GetCapabilities*-Anfrage aus. Wurde diese erfolgreich getätigt, werden Service-Metadaten vom SOS zurückgeliefert. Sofern diese Service-Metadaten Geoobjekte (Objekte mit räumlichen Bezug) enthalten, und diese außerdem eine Geometrie aufweisen, werden diese, auch *features of interest* (siehe 3.4.1) genannt, in der Eigenschaft *observedArea* festgehalten. Die räumliche Ausdehnung der *features of interest* stellt den minimal begrenzenden Raumabschnitt (*minimum bounding box*) dar, den diese beanspruchen. Im Anschluss erstellt der Client eine *GetObservation*-Anfrage und erhält, entsprechend spezifizierter Suchkriterien, die gewünschten enthaltenen Beobachtungen. Optional können zwischen diesen zwei Anfragen die Operationen *DescribeSensor* und *GetFeatureOfInterest* erfolgen. Dadurch werden die im SOS gespeicherten Sensoren (*procedure descriptions*) bzw. die *features of interest* abgefragt und vom SOS zurückgegeben. Dies ist sinnvoll, da durch diese Informationen das Aufstellen von Suchkriterien in einer *GetObservation*-Anfrage an die vorliegende Datenbasis angepasst werden kann. Das konzeptuelle Modell einer *GetObservation*-Anfrage findet sich im SOS-Standard [Bröring et al., 2012] und wird in Abb. 3.3 dargestellt.

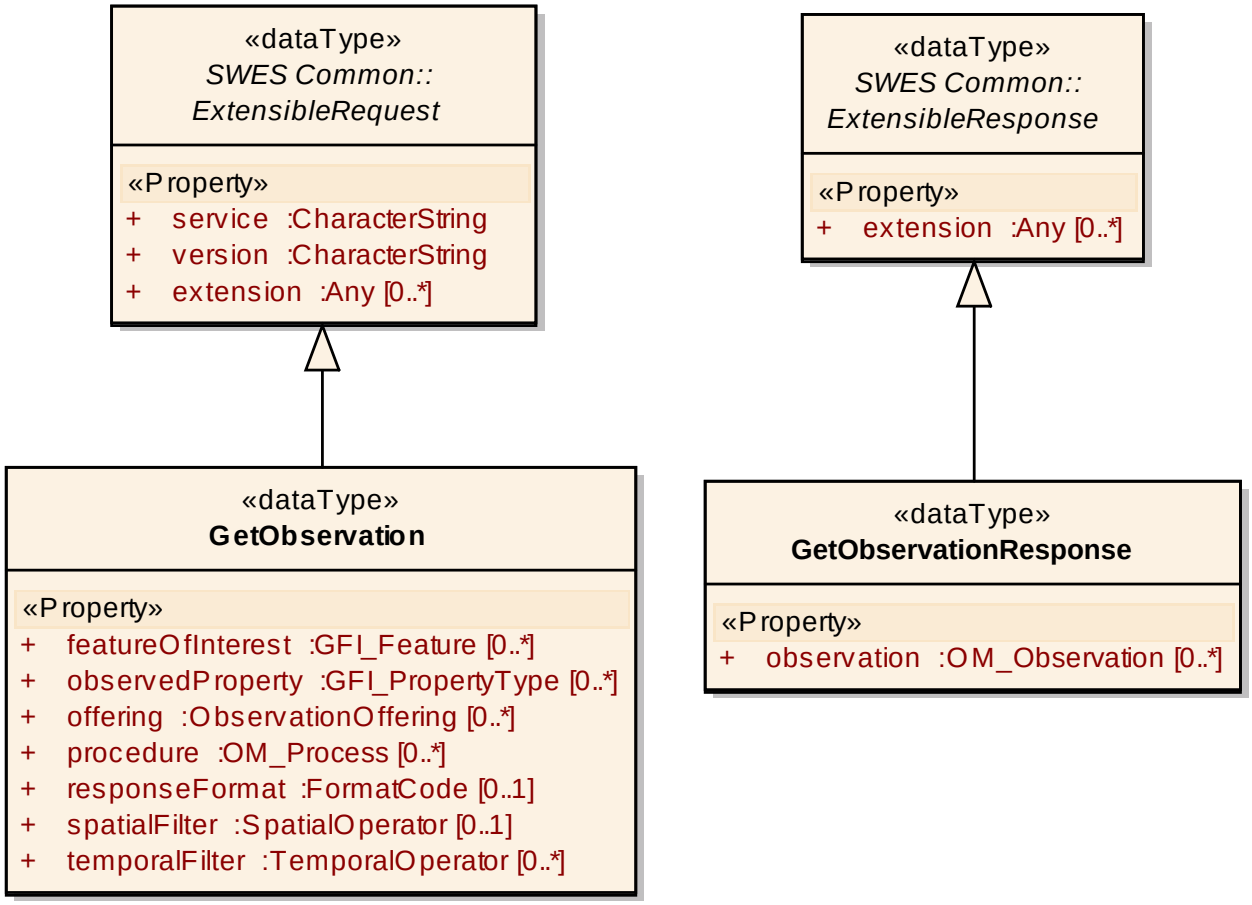


Abbildung 3.3: UML-Diagramm des *GetObservation*-Datentyps nach [Bröring et al., 2012]

Der *GetObservation*-Datentyp leitet sich vom Datentyp *SWES Common ExtensibleRequest* ab, welcher im *SWE Service Model* definiert ist [Echterhoff, 2011], und erbt dessen Attribute. Zusätzlich zu diesen kommen die Eigenschaften (*properties*) des *GetObservation*-Datentyps hinzu. Diese sind die Parameter, nach welchen Beobachtungsdaten vom SOS abgefragt werden können. Die Angabe der Parameter ist optional (Multiplizität 0..* bzw. 0..1). Wenn keine Parameter angegeben werden, so werden sämtliche Beobachtungen zurückgegeben. Das Filtern nach Beobachtungen geschieht durch die Angabe der gewünschten Eigenschaften in der *GetObservation*-Anfrage. Beispielsweise kann das räumliche Filtern einer Beobachtung durch die Angabe der Eigenschaft *spatialFilter* geschehen. Im *valueReference*-Element einer XML-*GetObservation*-Anfrage, muss dabei angegeben werden, nach welchem Kriterium das räumliche Filtern durchgeführt werden soll. Das *valueReference*-Element wird in [Vretanos, 2010] definiert. Darauf wird im Kapitel 3.2.4 näher eingegangen. Eine beispielhafte *GetObservation*-Anfrage formuliert als SOAP-Dokument findet sich im Anhang (siehe dazu B.1). Darin werden Prozedur (*procedure*), zu beobachtende Eigenschaft des *feature of interest* (*observedProperty*) sowie die logische Gruppierung von Beobachtungen einer Prozedur (*offering*) spezifiziert. Weitere Operationen werden in den unter *Extensions* zusammengefassten Kapiteln des SOS-Standards aufgeführt. Der Begriff *Extensions* wird dort als rein informativer Begriff verwendet [Bröring et al., 2012]. Er dient demnach zur besseren Übersicht und Strukturierung des SOS. Die Operationen werden nichtsdestotrotz formal in diesen *Extensions*-Kapiteln definiert [Bröring et al., 2012].

3.2.2 Transactional Extension

In der *Transactional Extension* des SOS-Standards werden die schreibenden Operationen, d. h. Hinzufügen sowie Entfernen von Sensoren und deren Beobachtungen, zusammengefasst [Bröring et al., 2012]. Hierbei handelt es sich um folgende Operationen:

- *InsertSensor* – erlaubt die Registrierung neuer Sensoren im SOS
- *DeleteSensor* – erlaubt die Entfernung von Sensoren sowie den damit verbundenen Beobachtungen
- *InsertObservation* – erlaubt das Hinzufügen neuer Beobachtungen von Sensoren im SOS

Bevor Beobachtungen eingefügt werden können, muss nach dem SOS-Standard ein Sensor registriert sein, welcher diese Beobachtungen durchführt [Bröring et al., 2012]. Daher ist die vorangehende Ausführung der *InsertSensor*-Operation obligatorisch für die Ausführung von *InsertObservation*-Operationen. Die Ausführung der *DeleteSensor*-Operation entfernt einen Sensor und die dazugehörigen Beobachtungen aus dem SOS. Das Hinzufügen von Daten durch eine SOS-Client-Anwendung wird im SOS-Standard mit einem Sequenzdiagramm veranschaulicht (siehe dazu Abb. 3.4). Auch hier findet eine Kommunikation basierend auf synchronen Nachrichten statt, welche vom Client (in der Abbildung als *Sensor Data Producer* bezeichnet) ausgehen. Der Aufruf einer *GetCapabilities*-Anfrage liefert Service-Metadaten zurück. Darunter fallen auch Informationen über bereits registrierte Sensoren und über die unterstützten Beobachtungstypen des SOS. Diese Informationen stehen im Abschnitt *Contents* respektive *InsertionCapabilities* des aus der Anfrage resultierenden *GetCapabilities*-Dokuments. Falls ein Sensor noch nicht im

SOS registriert wurde, so wird die *InsertSensor*-Operation ausgeführt, um diesen Sensor zu registrieren. Dabei wird die mit dem Sensor verbundene Beobachtungsgruppe (*assigned offering*) und ein Identifikator des Sensors (*identifier*) vom SOS zurückgegeben. Schließlich wird die *InsertObservation*-Operation ausgeführt.

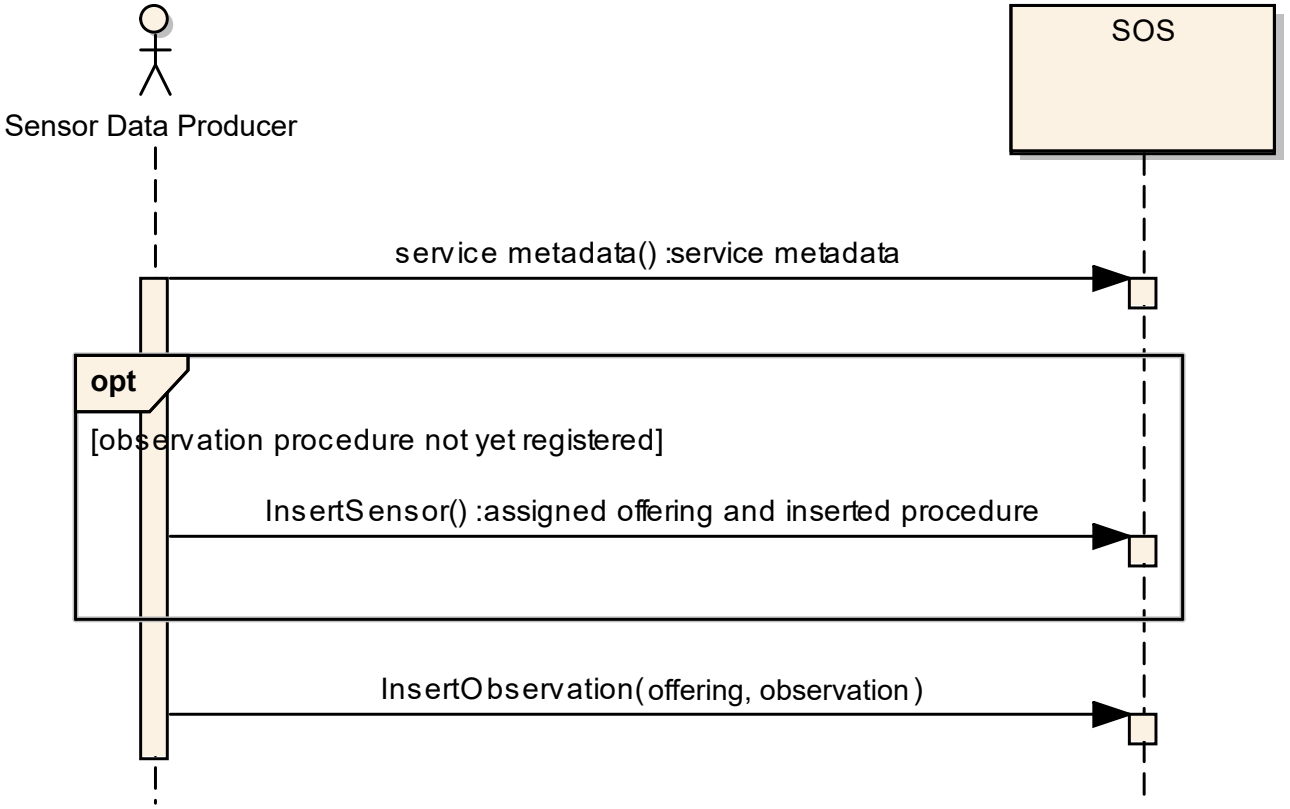


Abbildung 3.4: Ablauf des Einfügens von Daten im *Sensor Observation Service* (SOS) nach [Bröring et al., 2012]

Die zu tätigende *InsertObervation*-Operation enthält zum einen die Beobachtungsgruppe (Argument *offering*) sowie die Beobachtungen selbst (Argument *observation*) und gibt keinen Wert zurück. In einer *InsertObservation*-Operation können dabei mehrere Beobachtungen bzw. *offerings* übergeben werden [Bröring et al., 2012]. Mindestens eine Beobachtung und ein *offering* ist bei dieser Operation anzugeben. Dabei können mehrere Beobachtungen einem *offering*, umgekehrt eine Beobachtung mehreren *offering* zugeordnet werden. Das konzeptionelle UML-Modell einer *InsertObservation*-Anfrage im SOS-Standard zeigt diese Kardinalitätsbeziehungen auf (Abb. 3.5).

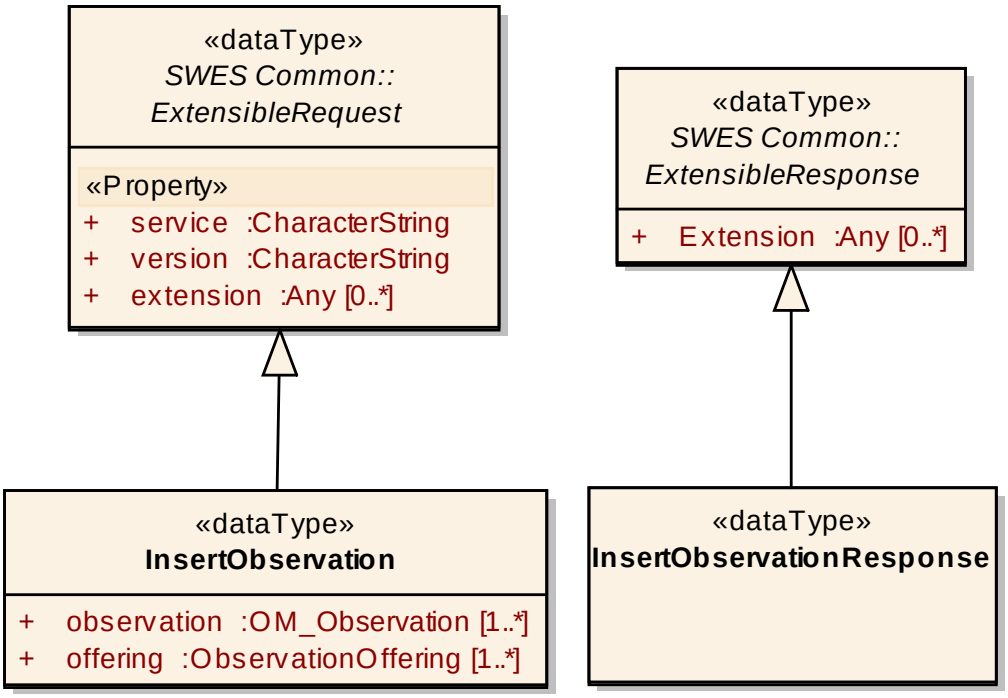


Abbildung 3.5: UML-Diagramm des *InsertObservation*-Datentyps nach [Bröring et al., 2012]

Der *InsertObservation*-Datentyp leitet sich, ebenso wie der *GetObservation*-Datentyp, vom Datentyp *SWES Common ExtensibleRequest* ab und erbt dessen Eigenschaften. Hinzu kommen die Eigenschaften *observation* sowie *offering*. Diese sind die Parameter der Operation. Es wird beim Hinzufügen von Daten die Angabe der Beobachtung sowie des entsprechenden *offering* verlangt (Multiplizität 1...*). Dabei ist zum einen die `1:n` Beziehung zwischen *procedures* und *offerings* zu beachten.[Bröring et al., 2012] Zum anderen besteht eine `n:m` Beziehung zwischen Beobachtungen und *offerings*. Normalerweise werden Beobachtungen in einer Gruppe zusammengefasst (`n:1`), da bei Nichtangabe eines *offerings* in der *InsertSensor*-Operation eine entsprechende Zuweisung zu einem *offering* automatisch durch den SOS vorgenommen wird.[1] Jedoch ist auch eine `1:n`-Beziehung zwischen Beobachtungen und *offering* erlaubt und in manchen Anwendungsfällen sinnvoll. Eine beispielhafte *InsertObservation*-Anfrage formuliert als SOAP-Dokument findet sich im Anhang (siehe B.3). In diesem Beispiel wird das Ergebnis (*result*) einer mittels einer Wassersäule (*procedure*) durchgeführten Höhenmessung (*observableProperty*) in einem Fluss (*feature of interest*) in Metern (uom, *unit of measurement*) angegeben. An dieser Stelle ist die mögliche Georeferenzierung der Beobachtung durch die Wahl eines *om:parameter*-Elements hervorzuheben. Auf diese Kodierung der räumlichen Referenzierung der Beobachtung wird in Kapitel 3.2.4 näher eingegangen.

[1] Lässt sich nach Ausführung einer *InsertObservation*-Operation und anschließenden Ausführung einer *GetObservation*-Operation im 52°North-SOS beobachten.

3.2.3 Binding Extension

Ein *binding* legt die Kommunikation zwischen SOS-Clients und Server fest [Bröring et al., 2012]. Die konzeptionellen UML-Modelle im SOS-Standard wurden dazu mittels XML implementiert und in XML-Schemata festgelegt. Die verschiedenen *bindings* stellen dabei konkrete Kodierungen dar, mit welchen die SOS-Operationen durch SensorML- und O&M-Dokumente repräsentiert werden. Im Referenzdokument des SOS [Bröring et al., 2012] werden *bindings* in KVP (*key-value pair*) und SOAP aufgeführt. Zusätzlich werden durch die 52°North-Implementierung XML-Dokumente (POX, *plain old XML*) unterstützt [52°North, 2016]. Es können somit unterschiedlich kodierte Dokumente dem SOS übergeben werden und von diesem ausgegeben werden. Neben diesen *bindings* unterstützt der SOS-Server von 52°North weitere Kodierungen [52°North, 2016].

3.2.4 Spatial Filtering Profile

Das *Spatial Filtering Profile* sieht vor, dass ausschließlich Beobachtungen mit Raumbezug durch einen SOS verfügbar gemacht werden [Bröring et al., 2012]. Auch das Hinzufügen neuer Beobachtungen, welche keinen Raumbezug aufweisen, wird dadurch verhindert. Damit wird die Filterung der Beobachtungen anhand von räumlichen Kriterien gewährleistet. Zudem erfährt die Eigenschaft *observedArea* des *ObservationOffering*-Datentyps eine Änderung. Anstelle der Repräsentation der räumlichen Ausdehnung der *features of interest* wird eine Repräsentation der räumlichen Ausdehnung der Beobachtungen vorgenommen. Die räumliche Ausdehnung, welche die Beobachtungen enthält (*observedArea*), stellt im *Capabilities*-Dokument einer *GetCapabilities*-Anfrage die minimal begrenzende Raumausdehnung (*minimum bounding box*) der Beobachtungen dar. Um die räumliche Ausdehnung von einzelnen Beobachtungen zu erhalten muss für die *valueReference* des räumlichen Filters der Wert `http://www.opengis.net/req/omxml/2.0/data/samplingGeometry` gewählt werden [Bröring et al., 2012]. Eine *GetObservation*-Anfrage mit räumlichem Filter kann dem Anhang (siehe B.2) entnommen werden. Soll stattdessen die Geometrie und Position des beobachteten Geoobjekts abgefragt werden, so wird für die *valueReference* der Wert `om:featureOfInterest/sams:SF_SpatialSamplingFeature/sams:shape` verwendet [Bröring et al., 2012].

3.3 Sensor Model Language

Der Standard SensorML ist ein Werkzeug zur Modellierung und Kodierung von Sensoren und anderen an der Beobachtung von Phänomenen der realen Welt beteiligten Komponenten [Botts und Robin, 2014]. Letztere werden in SensorML als Prozesse *processes* bezeichnet. Prozesse können demnach in rechnerische Prozesse wie mathematische Operationen sowie Funktionen und in physikalische Prozesse wie Sensoren, Detektoren und ganze Sensorsysteme unterschieden werden. Die Basis für die Modellierung der unterschiedlichen Prozesse findet sich in den *Core Concepts* von SensorML [Botts und Robin, 2014]. In diesen normativen Vorgaben sind die konzeptionellen Voraussetzungen für alle in SensorML vorkommenden und auf SensorML basierenden Modelle durch das sog. *Core Model* festgelegt. Nach diesem Modell verarbeitet ein Prozess Eingaben, entsprechend einer bestimmten Methodologie und nach festzulegenden Parametern, und generiert Ausgaben. Einem Prozess wird eine spezifische ID zugewiesen, durch welche dieser Prozess von anderen Prozessen unterschieden wird. Weiterhin trägt ein Prozess Metadaten, um die Identifizierung, Entdeckung und Beurteilung des Prozesses zu erleichtern. Dies ist für die Ausführung des Prozesses jedoch nicht zwingend erforderlich.

3.3.1 Sensormodell

Das grundlegende Prozessmodell wird im Abschnitt *Core Abstract Process* der SensorML festgelegt [Botts und Robin, 2014]. Es erfüllt die Mindestvoraussetzungen der *Core Concepts* und konkretisiert diese. Dazu werden verschiedene weitere Klassen implementiert. Zur Festlegung einer zu beobachteten Eigenschaft eines Prozesses wird jeweils eine Instanz der Klasse *ObservableProperty* herangezogen. Nach *Botts el al.* beschreibt diese eine physikalische Eigenschaft eines Phänomens (Temperatur, Druck, Länge etc.), welche beobachtet, oder auch erzeugt werden kann. Die Klasse *ObservableProperty* leitet sich von der Klasse *SWE Common AbstractSWEIdentifiable* ab [Botts und Robin, 2014]. Eine *ObservableProperty* besitzt ein *definition*-Attribut. Dieses Attribut referenziert eine beobachtbare Eigenschaft und verweist diese auf einen Eintrag in einem Lexikon oder einer Ontologie. Die Modellierung von Metadaten von Prozessen erfolgt in SensorML über die *DescribedObject*-Klasse (siehe dazu Abb. 3.6) vom Stereotypen *FeatureType*. Die Metadaten dienen zur raschen Identifizierung, Beurteilung oder Referenzierung von Prozessen. Zudem zeigen sie Beschränkungen der Prozesse auf und geben Versionierungsinformationen wieder. Alle in SensorML aufgeführten Klassen zur Modellierung von Prozessen basieren auf der *AbstractProcess*-Klasse. Jeder spezialisierte *Process* kann auf die *AbstractProcess*-Klasse zurückgeführt werden. Das Attribut *typeOf* hält diese Vererbungsinformation fest, indem es eine Instanz der Elternklasse hält (Multiplizität 0..1). Die Klasse *AbstractProcess* selbst leitet sich wiederum von der *DescribedObject*-Klasse ab. Zusätzlich zu den Metadaten kommen somit Attribute zur Beschreibung von Ein-, Ausgaben und Parametern hinzu. Numerische Werte oder physikalische Reize (d. h. beobachtbare Eigenschaften der Messumgebung) können solche Ein- und Ausgaben darstellen[Botts und Robin, 2014]. Parameter sind Botts et al. zufolge variabel oder konstant, sie können als Eingabe verstanden werden, welche fest gesetzt werden oder sich weniger häufig als Eingaben verändern. Daneben gibt es Attribute, um das zu beobachtenden

Phänomens der realen Welt, das sog. *feature of interest*, sowie Konfigurationsmöglichkeiten des Prozesses (*modes*) zu erfassen. Letzteres Attribut kann beispielsweise den Modus eines GNSS-Empfängers (kalter Modus, warmer Modus) repräsentieren, mit welchen der Empfänger eine Positionslösung bestimmen soll.

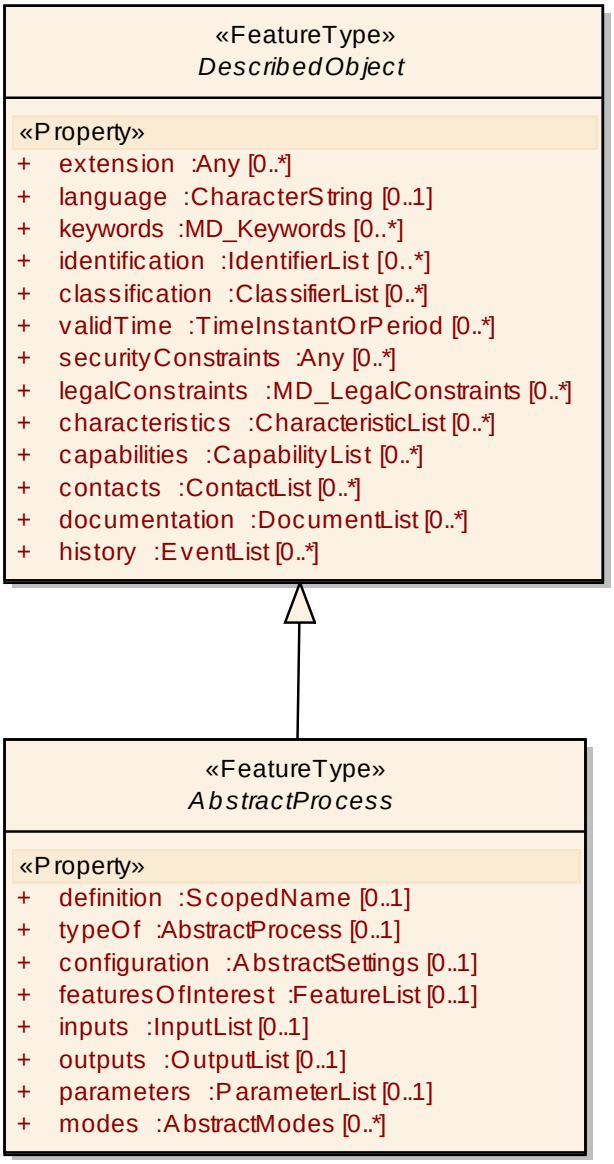


Abbildung 3.6: UML-Diagramm der *DescribedObject*- und *AbstractProcess*-Klassen nach [Botts und Robin, 2014]

3.3.2 Spezialisierte Prozesse

Alle in SensorML spezifizierten Klassen zur Modellierung von Prozessen erben von der Superklasse *AbstractProcess* [Botts und Robin, 2014]. Demnach kann in SensorML grundsätzlich zwischen physikalischen und nicht physikalischen Prozessen unterschieden werden. Zu der ersten Gruppe zählt die Basisklasse *PhysicalComponent* sowie die aggregierte Klasse *PhysicalSystem*. Letztere kann mehrere Instanzen der Klasse *PhysicalComponent* enthalten und beschreibt die Verbindung sowie die Interaktion zwischen diesen Komponenten [Botts und Robin, 2014]. Konkret lässt sich somit ein Hardwaresystem mit seinen einzelnen Komponenten und Verbindungen modellieren. Nicht physikalische Prozesse werden analog durch die Klassen *SimpleProcess* sowie *AggregateProcess* beschrieben [Botts und Robin, 2014]. Dadurch können der Datenfluss zwischen verschiedenen Softwaremodulen sowie die zugrundeliegenden Operationen, wie z.B. Filterung, mathematische Berechnungen, Konvertierungen etc., erläutert werden. Die Verkettung von Prozessen geschieht entweder durch die Klasse *AggregateProcess* bzw. *PhysicalSystem* [Botts und Robin, 2014]. Diese nehmen laut dem SensorML-Standard, anhand des Attributs *components*, Instanzen der spezialisierten Klassen von *AbstractProcess* in Listen auf. So können sowohl physikalische als auch nicht physikalische Komponenten miteinander in Beziehung gebracht werden. Ob dabei *SimpleProcess*-Instanzen oder *PhysicalComponent*-Instanzen herangezogen werden, ist unerheblich. Beide weisen die gleichen gemeinsamen Attribute auf. Die tatsächliche Unterteilung in die Kategorien physikalisch und nicht physikalisch geschieht konkret durch eine weitere Klasse: *AbstractPhysicalProcess* ist eine von *AbstractProcess* abgeleitete Klasse (siehe dazu Abb. 3.7). Diese weist zusätzlich Attribute auf, welche den Anschluss von Komponenten an andere Komponenten, Bezugssysteme sowie räumliche und zeitliche Daten über den physikalischen Prozess beschreiben. Daher können *PhysicalComponent*-Instanzen als erweiterte *SimpleProcess*-Instanzen mit der Möglichkeit der detaillierten Angabe von räumlichen und zeitlichen Attributen interpretiert werden. Die Klassen zur Modellierung von physikalischen Prozessen leiten sich von der Klasse *AbstractPhysicalProcess* ab, da hier nach der konzeptionellen Idee der physikalische Ort der Prozesse von Bedeutung ist. Klassen nicht physikalischer Prozesse hingegen sind Subklassen der Klasse *AbstractProcess* und tragen keine Ortsinformation. Dies ist einleuchtend, man stelle sich die Ausführung einer mathematischen Operation oder einer Funktion vor. Die Ausführung ist unabhängig vom Ausführungsort. Dagegen ist bei einem realen physikalischen Prozess, wie einem Sensor oder Detektor, von großer Bedeutung zu wissen, wo er sich befindet, da das Ergebnis der Prozessausführung vom Ort und auch von der Zeit abhängen kann.

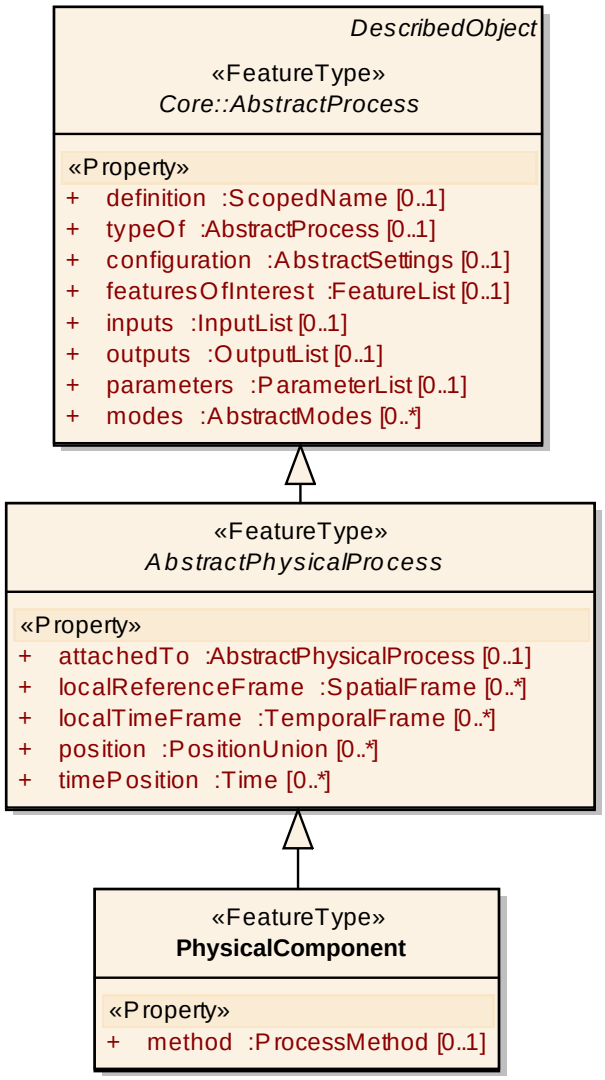


Abbildung 3.7: UML-Diagramm der Ableitung der *AbstractPhysicalProcess*-Klasse von der *AbstractProcess*-Klasse nach [Botts und Robin, 2014]

3.4 Observations and Measurements

Der Standard *Observations and Measurements* (O&M) stellt Modelle zur Beschreibung von Beobachtungen und Messungen zur Verfügung [Bröring et al., 2012]. Ziel des Standards ist es, Beobachtungen und Messungen jeglicher Art mit einem Framework beschreiben zu können. Damit soll der gegenseitige Austausch von Messergebnissen, sowohl innerhalb von als auch zwischen unterschiedlichen Fachgruppen, ermöglicht werden. Das O&M-Framework ist eng mit dem

SOS gekoppelt: Die Modellierung von Beobachtungen ist zwingend nach dem O&M-Standard durchzuführen, um den SOS zu nutzen [Bröring et al., 2012]. In den folgenden Abschnitten wird auf das Framework, die darin verwendete Terminologie und die Möglichkeit der parametrisierten Beobachtungen näher eingegangen.

3.4.1 Beobachtungsmodell

Abb. 3.8 bildet die Begriffe im O&M-Standard auf konkrete Objekte ab. Dargestellt wird eine Messung der Windgeschwindigkeit (*property*) eines Anemometers (*procedure*) an einem Gebäude (*feature of interest*). Die Messung lieferte den Wert 23 m/s (*value*, *unit of measurement*) und wurde am 16.09.2010, 13:45 Uhr erfasst.

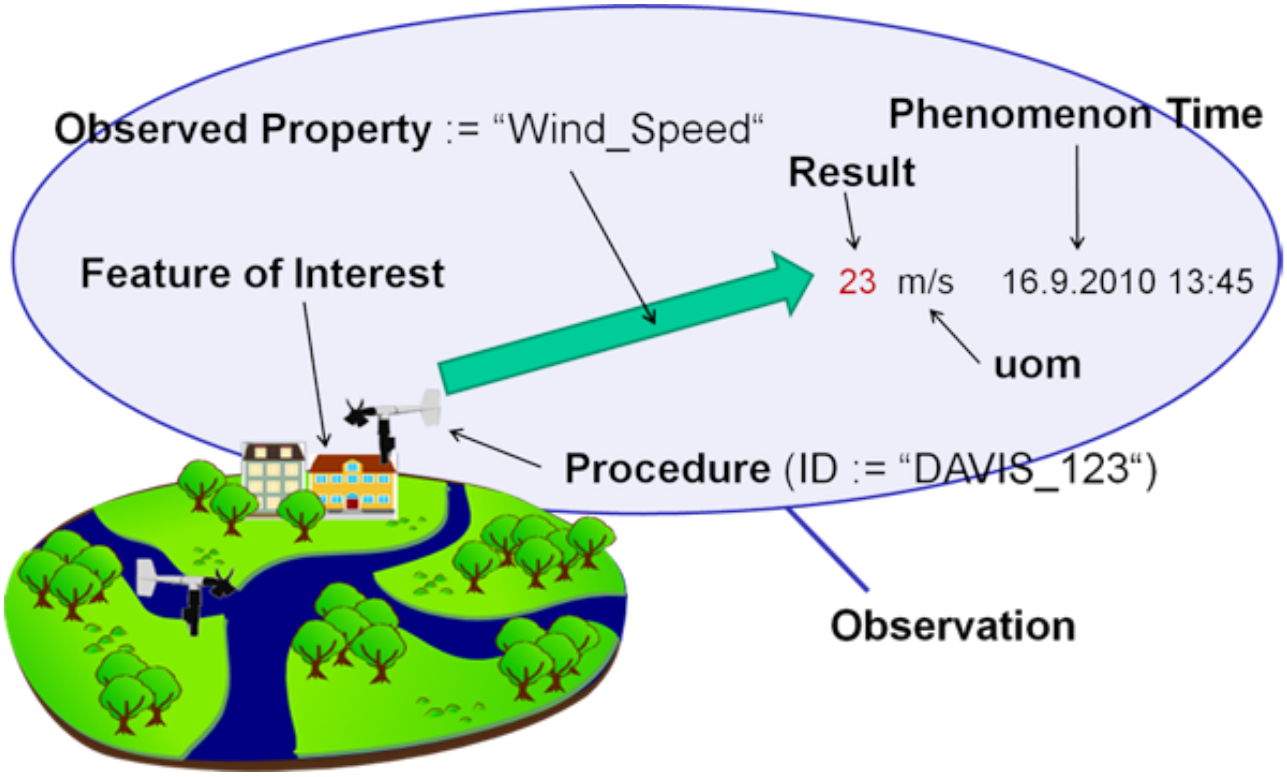


Abbildung 3.8: Terminologie im O&M-Standard, Abbildung aus http://www.ogcnetwork.net/sos_2_0/tutorial/om

Eine Beobachtung (*observation*) ist nach dem O&M-Standard eine Aktion mit einem Ergebnis (*result*), welches mit einem Wert (*value*) eine Eigenschaft (*property*) beschreibt. Die Eigenschaft und das Ergebnis sind einem *feature of interest* zugeordnet. Letzteres trägt einen Zeitstempel (*phenomenonTime*) und kann eine Einheit tragen (uom, *unit of measurement*). Das *feature of interest* beschreibt ein abstrahiertes Phänomen der realen Welt mit seinen zugehörigen Eigenschaften (*properties*). Dieses Phänomen möchte man in seinen Eigenschaften durch Be-

obachtungen bestimmen. Zur Durchführung einer Beobachtung ist eine Prozedur (*procedure*) notwendig. Eine Prozedur ist ein abstrakter Oberbegriff und kann einen Sensor, einen Beobachter, eine Computersimulation oder einen Algorithmus repräsentieren, analog zum Begriff *process* im SensorML-Standard. Eine Prozedur kann somit als Erzeuger von Beobachtungen verstanden werden. Das im O&M verwendete Modell einer Beobachtung ist konzeptionell definiert und wird in Abb. 3.9 dargestellt.

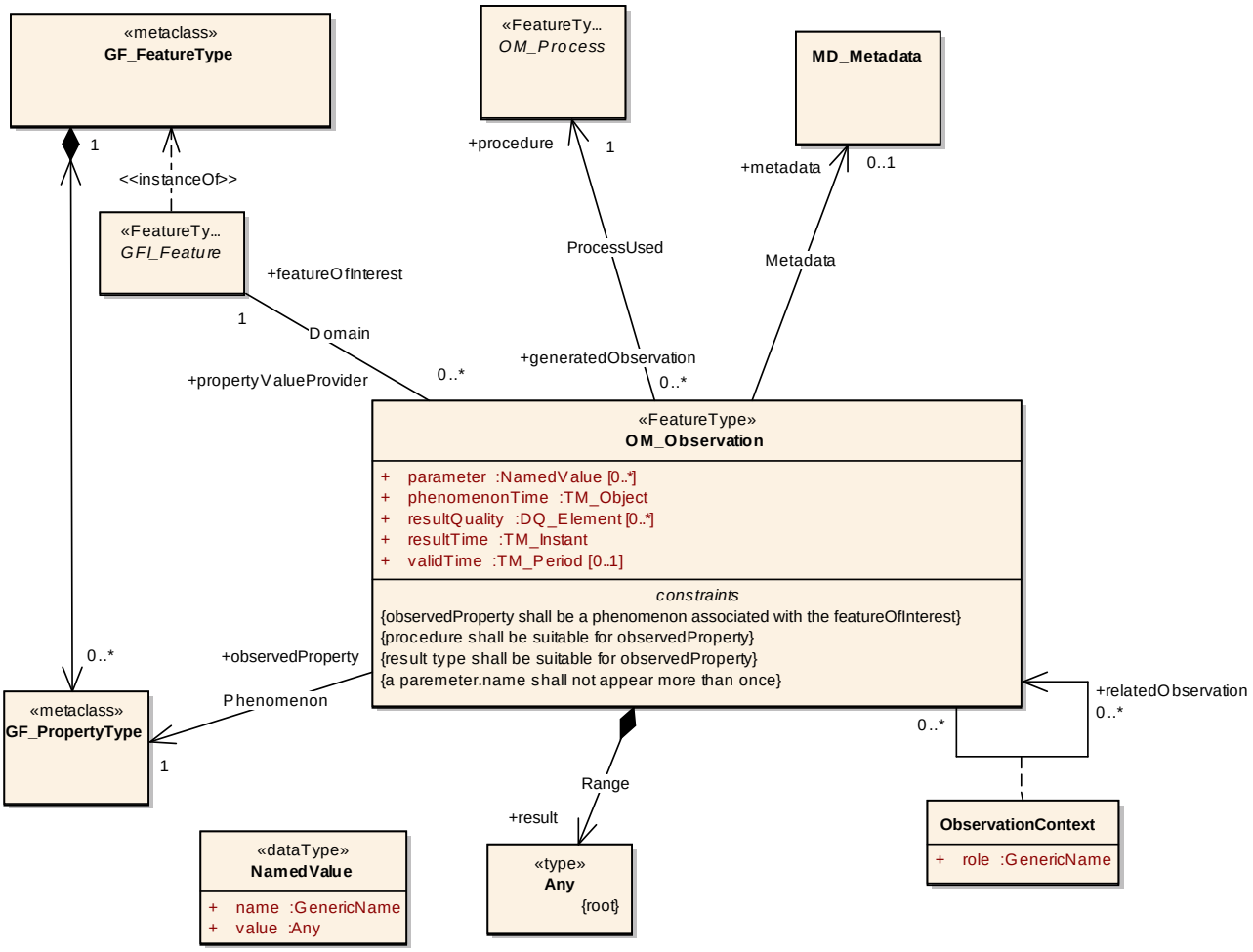


Abbildung 3.9: UML-Diagramm des Basistyps einer Beobachtung nach [Cox, 2013]

Instanzen der Klasse *OM_Observation* kommen in der Erfassung von Beobachtungen im SWE-Framework zum Einsatz. Eine Instanz bildet eine Beobachtung ab. Die Klasse weist Attribute und Assoziationen auf, die die notwendigen Informationen der Beobachtung zusammenfassen. Die *OM_Observation*-Klasse wird durch die Assoziation *processUsed* mit der Klasse *OM_Process* verbunden. Letztere abstrakte Klasse stellt die der Beobachtung zugrundeliegende Prozedur dar (gekennzeichnet durch die Rolle *procedure*). Eine Prozedur kann dabei keine bis beliebig viele Beobachtungen hervorbringen. Die Prozedur kann somit Beobachtungen generieren (siehe Rolle *generatedProcedure*). Liegt eine Beobachtung vor, so ist diese genau einer Prozedur zugeordnet.

Damit wird die Bedingung gewährleistet, dass keine Beobachtung existiert, welche nicht aus der Ausführung einer Prozedur resultiert. Durch die Komposition *Range* kann es ein Ergebnis nur geben, wenn es entsprechende Beobachtungen gibt. Das Ergebnis ist von einem beliebigen Typ (*Any*), da dessen Wert jede Eigenschaft eines *feature of interests* beschreiben kann. Im O&M-Framework eingesetzte Datentypen sind Spezialisierungen von Typen aus den Standards ISO 19103, 19108 und 19107 ([ISO, 2015a], [ISO, 2002], [ISO, 2003]) (Abb. 3.10).

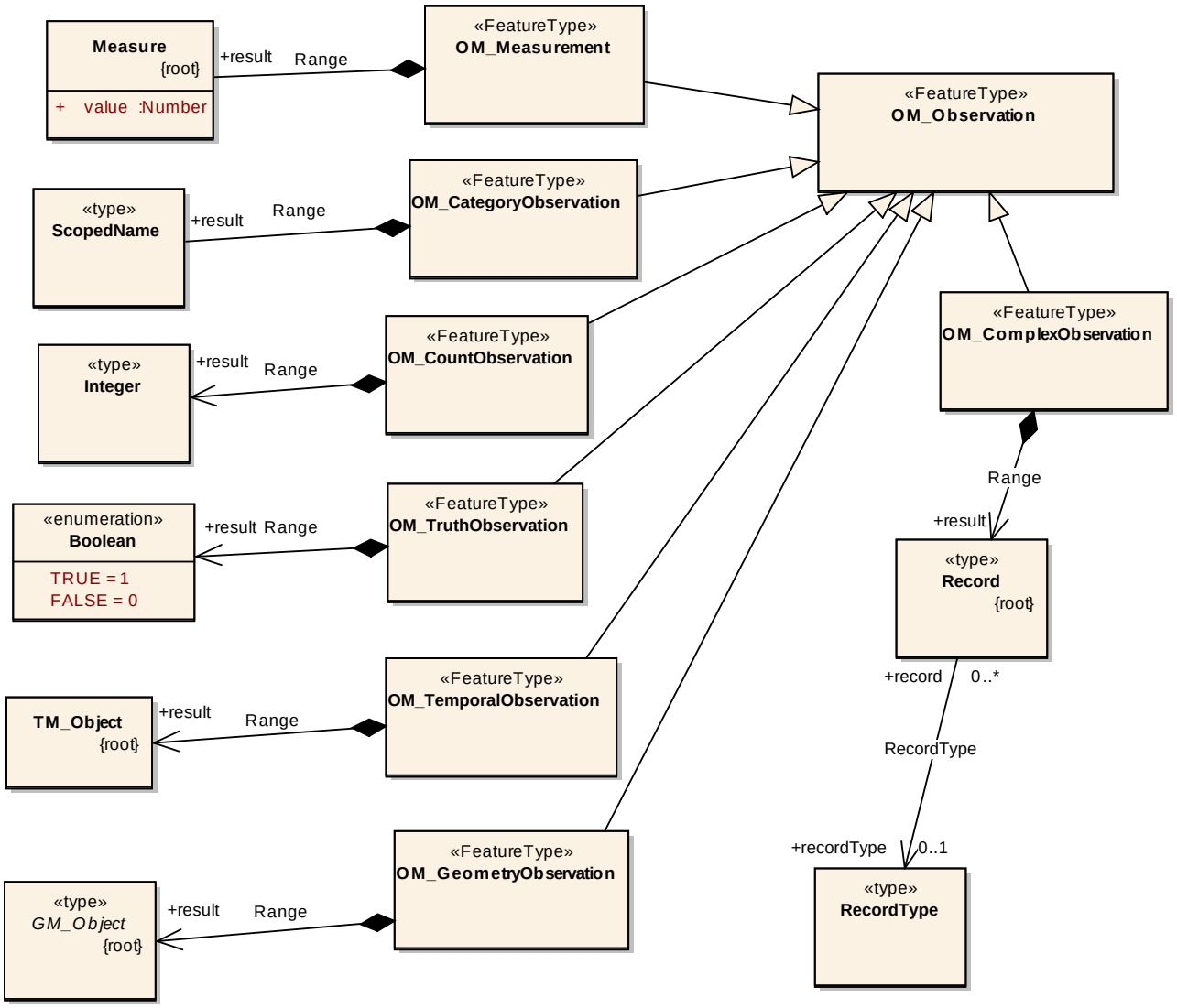


Abbildung 3.10: Spezialisierungen der Beobachtungen nach dem Ergebnistyp nach [Cox, 2013]

Die Assoziation *Phenomenon* verknüpft dazu die Klasse *OM_Observation* mit der Metaklasse *GF_PropertyType*. Letztere stellt eine Eigenschaft dar, deren Typ durch Beobachtungen zu bestimmen ist (siehe Rolle: *observedProperty*). Das Ergebnis einer Beobachtung liefert eine Schätzung des Wertes dieser beobachteten Eigenschaft. Jede Beobachtung ist mit genau einem *feature of interest* verknüpft. Dies wird durch die Assoziation *Domain* bewerkstelligt. Die

abstrakte Klasse *GFI_FeatureType* stellt das *feature of interest* einer Beobachtung dar. Eine *OM_Observation*-Instanz hingegen kann als Anbieter verstanden werden, welcher Werte für Eigenschaften des *feature of interest* bereitstellt (gekennzeichnet durch die Rolle *propertyValueProvider*). Jede Beobachtung benötigt zwingend zeitliche Informationen [Bröring et al., 2012]. Dazu gehört zum einen der Zeitpunkt, für welchen das Ergebnis der Beobachtung des *feature of interest* gilt. Der Zeitpunkt wird im Attribut *phenomenonTime* festgehalten. Zum anderen ist der Zeitpunkt, ab welchem das Ergebnis der Beobachtung verfügbar ist, im Attribut *resultTime* festzuhalten. Beide Zeitstempel sind in vielen Anwendungsfällen identisch [Bröring et al., 2012]. Sie können sich jedoch z. B. wegen nachträglichen Auswertungsprozeduren der Beobachtungen voneinander unterscheiden. Damit sind die notwendigen Voraussetzungen zur Modellierung von Beobachtungen geschaffen. Es kommen optionale Attribute und Assoziationen hinzu. Diese sind im Sinne des Beobachtungsmodells nicht zwingend. Insbesondere das *parameter*-Attribut ist für die vorgesehene Konzeption eines mobilen Sensors in dieser Arbeit von essentieller Bedeutung.

3.4.2 Parametrisierte Beobachtungen

Nach dem Beobachtungsmodell des O&M-Standards können Beobachtungen parametrisiert werden. Dies geschieht durch das Attribut *parameter*. Parameter enthalten spezifische Informationen über frei wählbare Ereignisse der Beobachtungen [Bröring et al., 2012]. Somit können diese Informationen über Beobachtungsumgebung, Einstellungswerte des Messinstruments oder Abtastparameter enthalten. Bei der Parametrisierung der Beobachtung darf jeder Parameter mit seiner Bezeichnung (*NamedValue*) nur einmal vorkommen (siehe dazu *constraints* der Klasse *OM_Observation*). Eine Bezeichnung besteht wiederum aus einem Namen-Wert-Paar. Dies wird durch die Attribute *name* und *value* ausgedrückt. Für die Beschreibung der Position einer Beobachtung ist diese Modellierungsmethode in der XML-Implementierung des O&M-Standards vorgesehen [Cox, 2011]. Eine solche räumliche Beobachtung (eine sog. *spatial observation*) weist demnach exakt einen Parameter zur Positionsbeschreibung auf. Das Attribut *name* dieses Parameters hat dabei den Wert `http://www.opengis.net/def/param-name/OGCOM/2.0/samplingGeometry`. Das Attribut *value* enthält ein Kindelement der Substitutionsgruppe der *gml:AbstractGeometry*, z. B. *gml:Point*, zur Beschreibung der Geometrie und Position der Beobachtung [Cox, 2011]. Metadaten der Beobachtung (Messgenauigkeit, Angaben zu Messbedingungen, Messreihe) können dadurch ebenfalls mittels weiterer Beobachtungsparameter eingeführt werden.

3.5 Moving Features

Der *Moving-Features*-Standard [Asahara et al., 2015b] ist das Ergebnis aus Implementierungsarbeiten des zuvor existierenden Standards ISO 19141 [ISO, 2008]. Letzterer ist ein Standard zur Modellierung von bewegten, festen (*solid*) Geoobjekten. Der Standard *Moving Features* stellt Kodierungen zur Modellierung dieser Objekte bereit. Ziel des Standards ist es einen effizienten und interoperablen Datenaustausch bewegter Objekte zu ermöglichen. Ausgenommen von der

Zielstellung des *Moving-Features*-Standards sind die Modellierung von deformierbaren Körpern sowie die Bereitstellung einer Webschnittstelle und Datenmodellen. Zur Klarstellung der anschließenden Erläuterungen sei erwähnt, dass der Ausdruck *Moving Features* bzw. seine Einzahl synonym mit sich bewegenden Geoobjekten verwendet wird, wenn nicht explizit abweichendes gesagt wird.

3.5.1 Foliation und Prismen

Die Bewegung und zeitliche Veränderung eines Objekts wird anhand eines *foliation and prisms*-Modell beschrieben (Abb. 3.11). Dieses Modell wird in der ISO 19141 [ISO, 2008] definiert und im Standard *Moving Features* mittels XML- und CSV-Kodierungen implementiert.

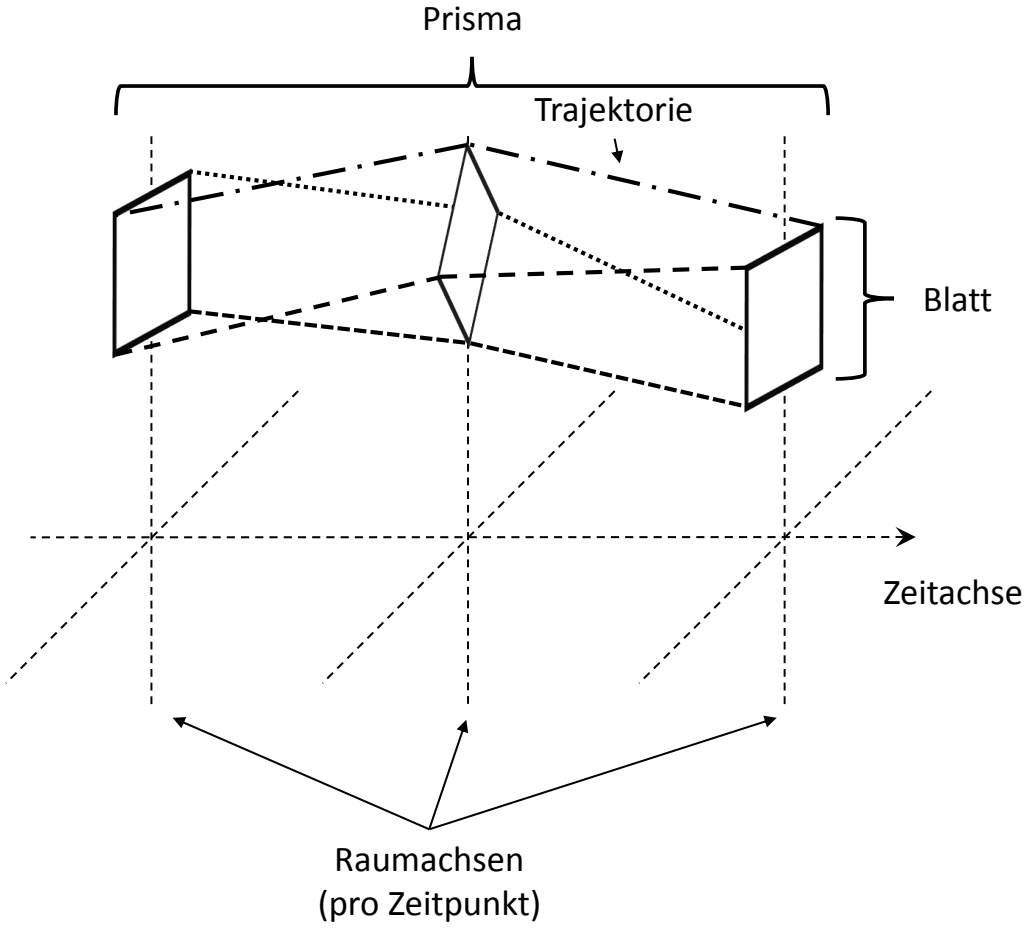


Abbildung 3.11: Foliation, Prismen und Trajektorien nach [Asahara et al., 2015b]

Anhand der Bewegung und Rotation eines Rechtecks wird dieses Bewegungsmodell erläutert. Zu drei unterschiedlichen Zeitpunkten lässt sich jeweils eine Repräsentation des Rechtecks finden. Jede Repräsentation zu einem Zeitpunkt stellt ein Blatt (*leaf*) dar. Während der Bewegung beschreiben die Eckpunkte des Rechtecks einen Pfad. Dieser wird als Trajektorie (*trajectory*) bezeichnet. Ein Prisma (*prism*) wird aus der Menge aller Punkte, welche sich in den Blättern und den Trajektorien befinden, gebildet. Die Gesamtmenge der Blätter stellt die „Belaubung“ (*foliation*) dar.

3.5.2 Implementierung des Bewegungsmodells

Zur Implementierung des in 3.5.1 genannten Modells stellt der *Moving-Features*-Standard sowohl eine XML-Kodierung [Asahara et al., 2015b] als auch eine CSV-Kodierung [Asahara et al., 2015a] von Bewegungsdaten bereit. *Moving-Features*-XML-Dokumente bestehen aus einem *Header*-Teil sowie einem *Body*-Teil. Da beim Parsen großer Datenmengen Speicherprobleme auftreten können, wurde die XML-Kodierung so gewählt, dass die Dokumente nicht vollständig verarbeitet werden müssen, um die essentiellen Informationen zu erhalten [Asahara et al., 2015b]. Lediglich die im *Header* enthaltenen Metadaten seien zur Interpretation der gespeicherten Bewegungsinformation notwendig. Im *Body* des Dokuments finden sich detaillierte Daten über die Bewegung. Die Struktur lässt sich an folgendem Schema veranschaulichen:

```
<mf:MovingFeatures
  gml:id="ID">
     <mf:sTBoundedBy>...</mf:sTBoundedBy>[1]
  <mf:member>...</mf:member>[0,..,n]
  <mf:Header>...</mf:Header>[0,1]
  <mf:Foliation>...</Foliation>[0,1]
</mf:MovingFeatures>
```

Listing 3.1: *Moving-Features*-XML-Struktur [Asahara et al., 2015b]

Ein *Moving-Features*-XML-Dokument besteht aus einem `mf:MovingFeatures`-Element. Das `mf:sTBoundedBy`-Kindelement enthält die räumlich-zeitliche Begrenzung der beschriebenen Trajektorien und wird für die Kodierung zwingend vorausgesetzt. Die *mf:member*-Elemente dienen zur Identifizierung und Beschreibung zusätzlicher Eigenschaften der sich bewegenden Geoobjekte (*Moving Features*). Das `mf:Header`-Element ist die entsprechende Implementierung des konzeptionellen *Header*-Teils. Darin werden sog. Attribute der *Moving Features* definiert, welche Zustände der *Moving Features* beschreiben. In Bezug auf die Forschungsfrage der Abbildung von Daten aus einem SOS in das Framework der *Moving Features* bietet sich diese Vergabe von Attributen zur Speicherung von Prozessdaten an. `VaryingAttrDefs`-Elemente listen die Attribute auf, `mf:AttrDef`-Elemente definieren sie im *Header*-Teil des XML-Dokuments. Die konkrete Angabe des Attributswerts geschieht innerhalb des `mf:foliation`-Elements. Der Attributswert, Zustand genannt, verändert sich entlang einer Trajektorie, d. h. einem Pfad innerhalb zweier Zeitpunkte, nicht [Asahara et al., 2015b]. Das bedeutet, eine Zustandsänderung impliziert immer eine neue Trajektorie. Ein solcher Zustand zwischen zwei Zeitpunkten kann anhand des folgenden Beispiels veranschaulicht werden: Eine Person bewegt sich im Schritttempo

von Zeitpunkt `A` bis Zeitpunkt `B`. Der exemplarisch eingenommene Zustand ist `gehend`. Am Zeitpunkt `B` angekommen, beginnt diese Person zu rennen und bleibt am Zeitpunkt `C` stehen. Der Zustand zwischen `B` und `C` ist damit `rennend`. Analog kann mit numerischen Zuständen gearbeitet werden, d.h. zwischen zwei Zeitpunkten hat beispielsweise ein Fahrzeug eine bestimmte Geschwindigkeit. Diese Eigenschaften der Attribute müssen bei der Abbildung von SWE-Daten auf *Moving Features* berücksichtigt werden und werden in Kapitel 4 diskutiert. Die auf CSV (*comma-separated values*) basierte Kodierung stellt eine im Vergleich zur XML-Kodierung vereinfachte Beschreibungsmethode zur Verfügung. Die CSV-Kodierung adressiert damit enorme Mengen an Bewegungsdaten, da weniger Ressourcen zur Speicherung und Verarbeitung in Anspruch genommen werden. Im Standard *Moving Features* besteht ein CSV-Dokument besteht, wie ein XML-Dokument auch, aus den Teilen *Header* und *Body*. Im *Header* dient eine Zeile zur Erläuterung von Metadaten. Eine solche Zeile beginnt mit einem @ -Zeichen sowie einem Metadatenbezeichner. Es folgen komma-separierte Werte zur Beschreibung von Metadaten. Beispielsweise erläutert der Bezeichner `stboundedby` die räumliche und zeitliche Ausdehnung der Daten. Er ist gegenwärtig der einzige im Standard definierte Bezeichner für Metadaten und hat folgende Struktur: `@stboundedby,[srid],[dim],[upper_left],[lower_right],[start_time],[end_time],[time encode]` Darin bedeuten die Werte:

1. `srid` = Bezeichnung des verwendeten Bezugssystems
2. `dim` = Räumliche Dimension der Daten, mögliche Werte:
   - 2D (standardmäßig)
   - 3D
3. `upper_left` = obere linke Ecke der räumlichen Begrenzung
4. `lower_right` = untere rechte Ecke der räumlichen Begrenzung der Trajektorie
5. `start_time` = Beginnzeitpunkt der Trajektorie nach dem ISO 8601-Format
6. `end_time` = Endzeitpunkt der Trajektorie nach dem ISO 8601-Format [ISO, 2004]
7. `time encode` = Zeiteinheit (standardmäßig Sekunden)

Der *Header*-Teil schließt stets mit dem Bezeichner `columns` ab. Dieser Bezeichner bildet mit seinen Werten die Bedeutung der im *Body*-Teil komma-separierten Werte ab und besitzt folgenden Aufbau:

```
@columns,mfidref,trajectory,[VaryingAttrDef(1) Name],[VaryingAttrDef(1) Type],...,
    VaryingAttrDef(n) Name],[VaryingAttrDef(n) Type]}
```

Listing 3.2: *Moving-Features*-CSV-*Body*-Aufbau nach [Asahara et al., 2015b]

Die nach `columns` aufgeführten Werte sind die Spaltenbezeichnungen der im *Body*-Teil aufgeführten Daten. Neben dem obligatorischem Bezeichner `mfidref` sowie der Beschreibung des zeitlich-räumlichen Zustands des bewegenden Objekts durch `trajectory` kommen beliebig viele Attributsdefinitionen (*VaryingAttrDef*) hinzu. Der Wert `trajectory` umfasst die im *Body*-Teil aufgeführten Zeit- und Rauminformationen. Diese sind komma-separierte Start- und Endzeitpunkte sowie räumliche Koordinaten der Objekttrajektorie. Die Koordinaten werden durch Leerzeichen voneinander getrennt. Attributsdefinitionen werden durch Paare, bestehend aus

einem Wert von `VaryingAttrDef Name` sowie einem Wert von `VaryingAttrDef Type`, gebildet. Ersterer trägt die Attributsbezeichnung, letzterer spezifiziert den Datentyp, mit welchem dieses Attribut beschrieben wird. Folgendes an Asahara et al. angelehntes, modifiziertes Beispiel soll die CSV-Kodierung von *Moving Features* erläutern [Asahara et al., 2015a]:

```
@stboundedby,urn:x-ogc:def:crs:EPSG:6.6:4326,2D,50.23 9.23,50.31 9.27,2012-01-17T12:33:40Z
    ,2012-01-17T12:37:50Z,sec
@columns,mfidref,trajectory,Gangart,xsd:token,gefundeneObjekte,xsd:integer
a,10,150,50.23 9.23 50.30 9.26,rennend,1
b,10,190,50.23 9.23 50.31 9.27,gehend,3
a,150,190,50.30 9.26 50.31 9.27,gehend,2
```

Listing 3.3: *Moving-Features*-CSV-Beispiel nach [Asahara et al., 2015a]

Im Beispiel werden die Trajektorien zweier Objekte mit den Bezeichnungen `a` und `b` beschrieben. Die Objekte beginnen ihre Bewegung zehn Sekunden (zweiter komma-separierter Wert) ab dem Startzeitpunkt, 12:33:40 Uhr, beide beginnen am selben Startort (Koordinaten 50,23 9,23) und beenden die Bewegung nach drei Minuten und zehn Sekunden am gleichen Zielort (Koordinaten 50,31 9,27). Das Attribut *Gangart* beschreibt das Tempo der Bewegung näher. Demnach vollzieht Objekt `a` seine Bewegung, in einem Zeitraum von zwei Minuten und 30 Sekunden, zunächst rennend. Objekt `b` hingegen geht über den gesamten Zeitraum und Weg hinweg. In den anschließenden letzten 40 Sekunden legt Objekt `a` im langsamen Tempo (gehend) den restlichen Weg zurück. Zusätzlich beschreibt das zweite Attribut die Anzahl der gefundenen Objekte. Beide Objekte haben nach Abschluss der Bewegung (letzter Wert der zweiten sowie dritten Zeile) drei Objekte gefunden.

3.5.3 Kodierung von Zustandsänderungen

Die Attribute in *Moving Features* beschreiben einen konstanten Zustand der Bewegung von einem Zeitpunkt zum nächsten [Asahara et al., 2015b]. In *Moving Features* ist daher eine neue Trajektorie immer mit einer Zustandsänderung verbunden. Zustandsänderungen beinhalten demnach Änderungen bezüglich Geschwindigkeit, Richtung, Existenz oder benutzerdefinierten Attributen des Objekts [Asahara et al., 2015b]. Abb. 3.12 aus dem *Moving-Features*-Standard veranschaulicht diesen Ansatz.

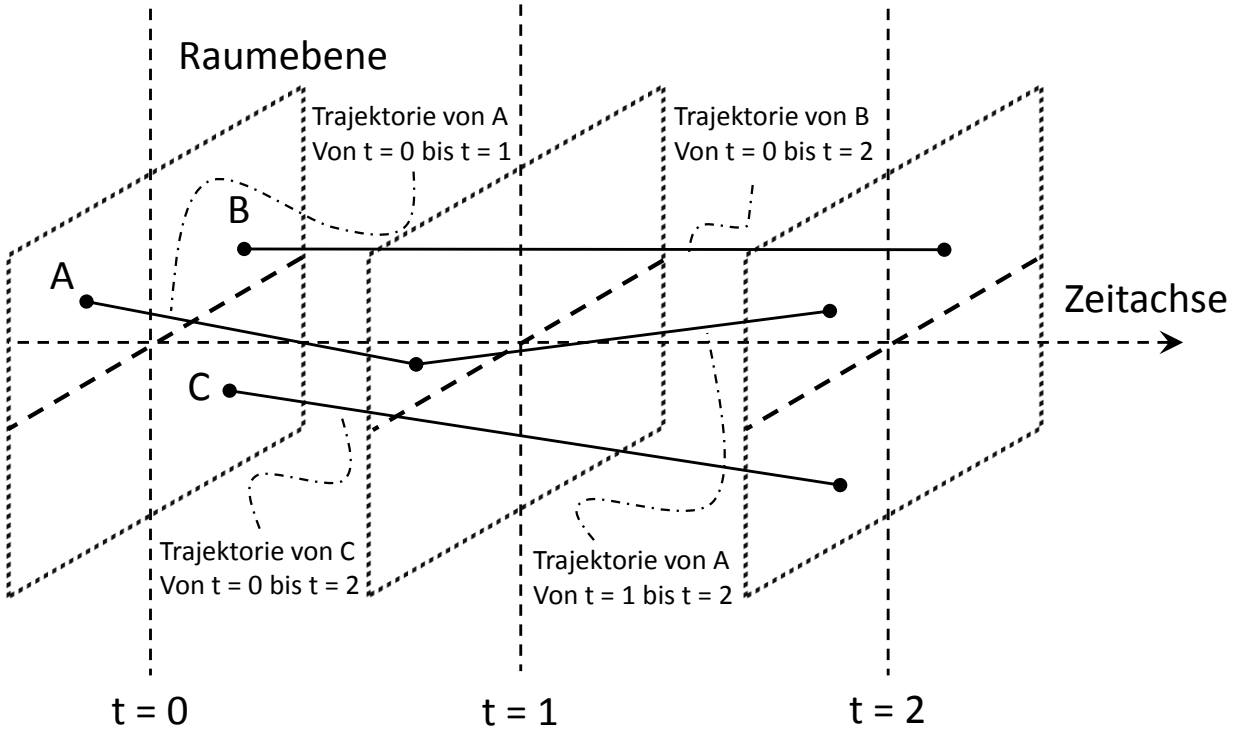


Abbildung 3.12: Trajektorienkodierung nach [Asahara et al., 2015b]

Dargestellt sind die Trajektorien dreier sich bewegender Punkte `A`, `B` und `C` zu drei Zeitpunkten `t=0, t=1, t=2`. Jede Trajektorie besitzt jeweils einen Beginn- und Endzeitpunkt. Die Bewegung beginnt für alle Punkte zum Zeitpunkt `t=0`. Zum Zeitpunkt `t=1` ändert sich der Zustand des Punkts A. Diese Änderung wird in der Kodierung durch Deklaration einer neuen Trajektorie festgehalten [Asahara et al., 2015b]. Die Bewegung des Punkts `A` wird somit zum Zeitpunkt `t=2` durch zwei Trajektorien beschrieben. Diese sind die Trajektorie vom Zeitpunkt `t=0` zum Zeitpunkt `t=1` sowie die Trajektorie vom Zeitpunkt `t=1` zum Zeitpunkt `t=2` Bei den Punkten `B` und `C` wird jeweils eine Trajektorie kodiert, nämlich vom Zeitpunkt `t=0` zum Zeitpunkt `t=2`, da zwischendurch keine Zustandsänderung vorgefunden hat. Die Anzahl der maximal zu kodierenden Trajektorien reduziert sich in diesem Beispiel von neun (drei Objekte x drei Zeitpunkte) auf vier Trajektorien. Daraus lassen sich für diese Trajektorien in der XML-Repräsentation die zeitliche Reihenfolge ableiten (Tabelle 3.1).

Tabelle 3.1: Schema der zeitlich sortierten Trajektorien

	Zeitraum der Trajektorie	
Moving Feature	Startzeitpunkt	Endzeitpunkt
A	t=0	t=1
B	t=0	t=2
C	t=0	t=2
A	t=1	t=2

Damit ist der Zustand zum jeweiligen Zeitpunkt der Bewegungen bekannt. Möglich ist auch eine sequentielle Sortierung einzelner Pfade nach der Zeit [Asahara et al., 2015b], welche in Tabelle 3.2 veranschaulicht wird.

Tabelle 3.2: Schema der sequentiell sortierten Trajektorien

	Zeitraum der Trajektorie	
Moving Feature	Startzeitpunkt	Endzeitpunkt
A	t=0	t=1
A	t=1	t=2
B	t=0	t=2
C	t=0	t=2

Liegen beispielsweise viele Zustandsänderungen des Punkt `A` vor, weist die sequentielle Sortierung einen Nachteil auf. Das Wissen über den Zustand der anderen Punkte `B` und `C` wird erst nach vollständigem Parsen der Trajektorien von `A` erschlossen.

3.6 Datenschnittstelle CAN

Der CAN-Bus (*Controller Area Network*) steht für eine Datenkommunikationstechnologie aus der Automobilbranche und wurde in der ISO-Norm 11898 [ISO, 2015b] festgehalten. Laut der CAN-Spezifikation von Bosch [Robert Bosch GmbH, 1991] entstand das System aus der Notwendigkeit, die vielen verschiedenen Datenübertragungswege zwischen Steuergeräten (ECU, *Electrical Control Unit*) in einem Fahrzeug und den damit verbundenen Verkabelungsaufwand sowie Kosten zu reduzieren. Stattdessen sollte der Datenaustausch seriell erfolgen, d.h. die zu übertragende Information wird nacheinander über die Datenleitungen gesendet. Laut der ISO

11898-1 werden Steuergeräte (ECU) im CAN-Standard durch Knoten (*nodes*) in einem Netzwerk dargestellt, welche miteinander kommunizieren. Dazu implementiert CAN die Schichten 1 und 2 des OSI-Modells (*Open Systems Interconnection*). Dieses Modell besteht aus insgesamt sieben Schichten und definiert die Funktionsweise von Netzwerken (Tabelle 3.3).

Tabelle 3.3: OSI-Modell nach [ISO und IEC, 1994]

OSI-Schicht		Einheit	Funktion
7	Anwendungen (*Application*)	Daten	Dienste, Nachrichtenformat, Mensch-Maschine-Schnittstelle
6	Darstellung (*Presentation*)	Daten	Repräsentation, Konvertierung, Kompression und Verschlüsselung
5	Kommunikation (*Session*)	Daten	Verwaltung, Organisation und Wiederherstellung von Kommunikationssitzungen
4	Transport (*Transport*)	Segmente	Austausch von Datensegmenten über Ende-zu-Ende-Verbindungen
3	Vermittlung (*Network*)	Pakete	Netzwerkadressierung, *Routing* und Datenflusskontrolle
2	Sicherung (*Data Link*)	Rahmen (Frames)	Fehlerdetektion, Kontrolle des Datenstroms der physikalischen Verbindung
1	Bitübertragung (*Physical*)	Bits	Senden und Empfangen von Bitströmen über physikalisches Medium

In der Bitübertragungsschicht *physical layer* definiert CAN die physikalische Umsetzung des Sendevorgangs von Bits und die elektrischen Eigenschaften des Bussystems. Damit legt diese Schicht die physikalische Verbindung zwischen zwei Knoten fest. Die Sicherungsschicht *data link layer* regelt den auf dem CAN-Bus stattfindenden Datenaustausch. Sie enthält die sog. *Frames*, welche Bits in Telegrammen zusammenfassen, sowie Informationen, um *Frames* und Fehler zu identifizieren. In den folgenden Beschreibungen wird auf die Grundlagen von CAN basierend auf den Spezifikationen in der ISO 11898-1, wenn nicht anders vermerkt, eingegangen.

3.6.1 Signalübertragung

Die Signalübertragung bei CAN basiert auf einer Übertragung von Spannungsdifferenzen [Wiesinger, 2015]. Bei der Übertragung eines Signals sendet ein Steuergerät zusätzlich ein Referenzsignal entgegengesetzter Polarität aus. Beide Signale tragen aufgrund von Umgebungseinflüssen jeweils ein ähnliches Störsignal mit sich. Am Empfänger angekommen, wird die Differenz dieser Signale gebildet. Die Störsignale werden dabei, da sie das gleiche Vorzeichen aufweisen, weitgehend eliminiert. Zur Umsetzung der Spannungsdifferenzbildung werden im Falle von Kupferleitungen zwei verdrillte Adern, *CAN High* und *CAN Low*, eingesetzt. Durch die Verdrillung wird die Störanfälligkeit gegen elektromagnetische Störungen von außen verringert. Mittels der Spannungsdifferenzen werden die zum Datenaustausch genutzten Bits repräsentiert. Eine logische 1 ist gleichbedeutend mit einer Spannungsdifferenz von 0 Volt, dabei liegen beide Leitungen auf 2,5 V (Ruhezustand) [Robert Bosch GmbH, 1991]. Hingegen bedeutet eine Spannungsdifferenz von 2 Volt eine logische 0. Die Differenz resultiert aus der *CAN-High*-Spannung von ca. 3,5 V und der *CAN-Low*-Spannung von 1,5 V.

3.6.2 Netzwerkkommunikation

Mittels eines CAN-Bussystems kommunizieren mehrere Steuergeräte gleichberechtigt miteinander (*Multi-Master*-Prinzip) [Robert Bosch GmbH, 1991]. Ein Teilnehmer dieses Netzwerkes benötigt nach dem CAN-Standard eine CAN-Schnittstelle. Diese besteht aus einem CAN-Controller und einem CAN-Transceiver. Der CAN-Controller regelt den Datenaustausch zwischen Steuergerät und CAN-Schnittstelle. Er bereitet sie entsprechend auf und gibt diese an den CAN-Transceiver respektive an das Steuergerät weiter. Der CAN-Transceiver ist für die physikalische Übersetzung der Datenbotschaften vom und für den CAN-Controller zuständig. Wenn ein Sender eine Datenbotschaft (*Frame*) übermittelt, erhält jeder Knoten in diesem Netzwerk diese Datenbotschaft (sog. *Broadcasting*). Anhand der Bezeichner (*identifier*) der Datenbotschaften können die Empfängerknoten entscheiden, ob der darin enthaltene Inhalt für sie relevant ist oder nicht. Die *identifier* sind Bitfolgen definierter Länge und legen das Format der *Frames* fest:

- *Base Frame Format*, auch *CAN 2.0A*, bestehend aus 11 Bits
- *Extended Frame Format*, auch *CAN 2.0B*, bestehend aus 29 Bits

Bei gleichzeitiger Nutzung des CAN-Bus durch zwei oder mehr Knoten wird durch die *identifier* der *Frames* eine Kollisionsauflösung erreicht: Durch Konjunktion (logisches *UND*) überschreibt eine logische 0 (dominantes Bit) in der Bitfolge eines *Frames* die logische 1 (rezessives Bit) eines anderen *Frames*. Die Nachricht eines Knotens wird daher anhand der überschreibenden Bits bevorzugt (sog. bitweise Arbitrierung). Dies bedeutet auch, dass eine Datenbotschaft von maximal einem Sender übermittelt werden darf, da ansonsten die Kollisionsauflösung nicht funktionieren kann.

3.6.3 Frame-Aufbau

Die zur Kommunikation genutzten *Frames* lassen sich im CAN-Standard in vier unterschiedliche Typen unterteilen [Robert Bosch GmbH, 1991]:

- *Data Frame* – enthält die Nutzdaten
- *Remote Frame* – fordert Daten-Frames eines anderen Knotens an
- *Error Frame* – signalisiert allen Knoten einen erkannten Übertragungsfehler
- *Overload Frame* – führt Zwangspause zwischen einem *Data-Frame* und *Remote-Frame* ein

Ein *Frame* besteht aus einer definierten Abfolge von Bits. Die in einem solchen *Frame* vorkommenden Bits werden wiederum in genormten Gruppen zusammengefasst. Der *Data Frame* enthält die eigentlichen Nutzdaten (Tabelle 3.4).

Tabelle 3.4: *Base Frame Format* mit 11-Bit-*identifier*

Feldname	Anzahl Bits
Start Of Frame (SOF)	1
identifier	11
Remote Transmission Request (RTR)	1
Identifier Extension (IDE)	1
reserviertes Bit	1
Data Length Code (DLC)	4
Data Frame	0 - 64
Cyclic Redundancy Check (CRC)	15
CRC-Trennzeichen	1
ACK-Slot (*Acknowledge*)	1
ACK-Trennzeichen	1
End Of Frame (EOF)	7

Die Bitabfolge eines solchen *Frames* beginnt mit dem Startbit SOF. Dieses ist mit einem dominanten Bit (logische 0) versehen, um durch den eingeleiteten Flankenwechsel von logisch 1 (rezessiv) auf logisch 0 die Knoten im Netzwerk auf sich aufmerksam zu machen. Dem folgt ein *identifier*. Je niedriger die Bitfolge des *identifier* desto höher zu priorisieren ist die Datenbotschaft, da die dominanten Bits die rezessiven überschreiben. Durch den Wert des anschließenden RTR-Bits wird der *Frame*-Typ festgelegt. Handelt es sich dabei um ein dominantes Bit, so liegt ein *Data Frame* ansonsten ein *Remote Frame* vor. Empfängt ein Steuergerät einen *Remote Frame* mit einem *identifier* seiner eigenen Datenbotschaften, so sendet er ein entsprechendes *Data Frame* an die anderen Knoten im Netzwerk. Steuergeräte erkennen durch den Empfang eines solchen *Remote*

Frame, dass ihr eigener *Data Frame* angefordert wird. Es folgt ein IDE-Bit zur Unterscheidung des *identifier*-Formats. Ist dieses dominant, so liegt ein 11-Bit-*identifier* (*Base Frame Format*) vor. Andernfalls wird mit einem rezessiven Bit eine 29-Bit-Datenbotschaft gesendet. An das IDE-Bit knüpft ein dominantes Bit (sog. *reserved Bit*), für welches eine zukünftige Funktion durch den CAN-Standard vorgesehen ist. Die darauffolgende Bitfolge DLC wird aus vier Bits gebildet und gibt die Anzahl der Nutzbytes an. Die Nutzdaten des *Data Frame* selbst werden im bis zu acht Byte großen *Data Field* angegeben. Die Integrität der Nutzdaten wird durch einen CRC geprüft. Diese Überprüfung endet mit einem CRC-Trennzeichen (rezessives Bit). Das Bit im ACK-Slot ist rezessiv und wird von den Empfängern des *Frames* dominant überschrieben, falls diese mittels des CRC den korrekten Empfang des *Frames* quittieren. Auch hier wird ein rezessives Bit zur Trennung des ACK-Slots von den anschließenden Bits angefügt. Der *Data Frame* wird mit dem EOF, bestehend aus sieben rezessiven Bits, abgeschlossen. Ähnlich ist die Struktur des *Data Frame* mit einem 29-Bit-*identifier* (Tabelle 3.5).

Tabelle 3.5: *Extended Frame Format* mit 29-Bit-*identifier*

Feldname	Anzahl Bits
SOF	1
identifier A	11
Substitute Remote Request (SRR)	1
IDE	1
identifier B	18
RTR	1
reserviertes Bit r1	1
reserviertes Bit r0	1
DLC	4
Data Frame	0 - 64
CRC	15
CRC-Trennzeichen	1
ACK-Slot	1
ACK-Trennzeichen	1
EOF	7

An der Stelle des RTR-Bit im *Base Frame Format* tritt ein rezessives SRR-Bit. Falls zwei Datenbotschaften unterschiedlichen Formats mit demselben *identifier* empfangen werden, hat der *Frame* im *Base Frame Format* durch das dominante RTR-Bit Vorrang. Dies bedeutet, dass ein *Frame* im *Base Frame Format* stets einem *Frame* im *Extended Frame Format* vorzuziehen ist. Auch das IDE-Bit ist hier rezessiv. Daran knüpft ein zusätzliches Feld mit einem 18-Bit-*identifier* an. Es folgt das RTR-Bit sowie ein zusätzliches, dominantes, reserviertes Bit mit der Bezeichnung *r1*.

3.6.4 ISOBUS

Auf Basis von CAN wurden weitere Technologien und Anwendungen zur Datenkommunikation zwischen Steuergeräten entwickelt. Das LBS basiert auf dem CAN-Bussystem und wurde unter der Leitung von Hermann Auernhammer für den Einsatz in Landmaschinen entwickelt [Auernhammer, 1983]. Auch dieses System wurde genormt [DIN, 1997], konnte sich jedoch nicht durchsetzen. Die Ergebnisse des LBS-Standards sowie des Standards J1939 [SAE, 2013] aus dem Automobilbereich flossen in die Entwicklung des ISOBUS-Systems ein [Rothmund und Wodok, 2010] [ISO, 2007]. ISOBUS steht für ein System, welches physikalische Anschlüsse, Datenaustausch von Steuergeräten sowie Mensch-Maschine-Schnittstellen für Traktoren und Anbaugeräte in der Landwirtschaft zur Verfügung stellt. Die Norm ISO 11783 standardisiert diese Komponenten des Systems. Die Datenübertragung basiert auf CAN und erfolgt mit einer Übertragungsgeschwindigkeit von 250 kbit/s [ISO, 2012].

Im Folgenden wird auf den modifizierten CAN-Frame im ISOBUS-System eingegangen, der im dritten Teil der ISO 11783 definiert wird [ISO, 2014]. Im Unterschied zum CAN-Bus werden die Teilnehmer des Netzwerks ebenfalls adressiert, um zu verhindern, dass mehrere Steuergeräte den gleichen *identifier* benutzen. Ein ISOBUS-*Frame* im *Extended Frame Format* besteht analog zum CAN-Frame aus einem 29-Bit-*identifier* sowie einem 64-Bit-Datenfeld. Dieser *Frame* wird als PDU (*protocol data unit*) bezeichnet. Gespeichert werden die Daten auf mehreren Seiten, d. h. eine ISOBUS-Nachricht kann sich auf mehrere CAN-*Frames* erstrecken. Eine so modifizierte CAN-Nachricht besteht aus sieben definierten Feldern unterschiedlicher Länge:

- *Priority*
- *Extended Data Page* (EDP)
- *Data Page* (DP)
- *PDU Format* (PF)
- *PDU Specific* (PS)
- *Source Address* (SA)
- Datenfeld

Die darin enthaltenen Informationen stellen sog. Parameter dar und werden in Parametergruppen eingeteilt. Eine Parametergruppe lässt sich anhand einer PGN (*Parameter Group Number*) identifizieren. Die PGN wird aus den ersten fünf der o. g. Felder gebildet. Mittels einer PGN werden die Nachrichten identifiziert, da bestimmte Nachrichtentypen einer PGN zugeordnet sind. Das *Priority*-Feld gibt die Priorität der Nachricht mittels drei Bits an. Die höchste Priorität einer

Nachricht hat den Wert 000_2 (dezimal 0) und die geringste Priorität den Wert 111_2 (dezimal 7). Das EDP-Bit gibt in Kombination mit dem DP-Bit die Seite der Nachricht an. Es sind vier verschiedene Seiten möglich, wobei gegenwärtig nur zwei von der ISO 11783 genutzt werden (Tabelle 3.6).

Tabelle 3.6: EDP-DP-Kombinationen

EDP-Bitwert	DP-Bitwert	Beschreibung/Funktion
0	0	PGN Seite 0
0	1	PGN Seite 1
1	0	reserviert durch ISO 11783, zukünftige Verwendung vorgesehen
1	1	Wird genutzt durch die ISO 15765-3

Das PF-Feld gibt an, welches Nachrichtenformat verwendet wird. Ist der hier vorliegende Wert kleiner als 240, so liegt ein PDU1-Format vor. Der Wert im PS-Feld gibt dann eine bestimmte Zieladresse (*Destination Address*) an, an welche die Nachricht bestimmt ist. Ist der Wert im PF-Feld hingegen im Bereich von 240 bis 255, so wird das PDU2-Format verwendet. Der Wert im PS-Feld spezifiziert dann eine *Broadcast*-Nachricht mit erweiterten Informationen, welche an alle Teilnehmer des Netzwerks gerichtet ist. Das PS-Feld trägt dann die Bezeichnung GE (*Group Extension*). Im SA-Feld befindet sich die Adresse des Nachrichtensenders. Tabelle 3.7 stellt die in ISO 11783 veränderten Felder dem CAN-*identifier* gegenüber.

Tabelle 3.7: Vergleich der *Extended-Frame*-Kontrollfelder zwischen CAN-Bus und ISOBUS

CAN-Bus-Feld	ISOBUS-Feld	Anzahl Bits
identifier A	3 Priorität-Bits, 1 EDP-Bit, 1 DP-Bit, 6 PF-Bits	11
identifier B	2 PF-Bits, 8 PS-Bits, 8 SA-Bits	18

Alle anderen Felder aus CAN werden durch die ISO 11783 unverändert übernommen.

3.7 Modemkommunikation mittels AT-Befehlssatz

Die Positionsbestimmung zur räumlichen Referenzierung der Prozessdaten und die Übertragung von Daten wird durch den Einsatz eines GSM-Moduls mit integriertem GNSS-Empfänger realisiert. Die zusammengefügten Komponenten eines solchen Moduls sind auf einer Platine montiert und können als Kernstück eines Mobiltelefons betrachtet werden. Es lassen sich damit im Wesentlichen Anrufe tätigen sowie empfangen, Kurznachrichten versenden, Daten über das Mobilfunknetz übertragen und, falls die entsprechende Funktion vorhanden ist, die Position über GPS bestimmen. Insbesondere die Datenübertragungsfunktion ist im Kontext des Telemetriesystems essentiell und unter gegebener Netzabdeckung, durch die erhöhte Reichweite, vorteilhafter als ähnliche Techniken (WLAN, Bluetooth). Die Konfiguration und Steuerung dieser Module erfolgt in vielen Fällen durch den sog. AT-Befehlssatz. Dieser Befehlssatz wird durch die International Telecommunications Union standardisiert und legt die Konfiguration und Kommunikation von Modems fest [ITU-T, 2003]. Diese Befehle erlauben die verschiedenen Funktionen eines GSM-Moduls zu nutzen und weisen ein spezifisches Format auf. Sie beginnen immer mit der Zeichenfolge `AT`, welche für sich allein auch einen Befehl darstellt (`AT` steht dabei für ***At****tention*). Hiermit wird der gegenwärtige Status des Modems abgefragt. Eine entsprechende Antwort durch die serielle Schnittstelle wird zurückgegeben. Andere Befehle basieren darauf, dass nach dem `AT` weitere Zeichen folgen.

- `ATA` – Nehme Anruf entgegen (A : Accept).
- `ATD110` – Wähle die Notrufnummer 110 (D : Dial).

Neben diesen grundlegenden Befehlen existieren spezifische Befehle, welche durch vorangestelltes „+“ und weitere Zeichen gekennzeichnet sind. Diese ermöglichen erweiterte Funktionen des Modems:

- `AT+CPIN=1058` – Gebe PIN zur Aktivierung der SIM-Karte ein.
- `AT+CGPSPWR=1` – Schalte GPS-Modul ein.
- `AT+CGPSSTATUS=?` – Liegt eine GPS-Lösung vor?
- `AT+CGPSINF=0` – Fordere Position im NMEA-Format an

Das Zeichen unmittelbar nach „+“ kennzeichnet eine Gruppe von Befehlen, die zu einer spezifischen Erweiterung zu zählen ist. Erweiterungen sind zum Teil herstellerabhängig und sind daher, anders als die grundsätzlichen Befehle, nicht auf jedem GSM-Modul verfügbar. Zuweilen bestehen zwischen verschiedenen Modellen von GSM-Modulen des gleichen Herstellers Unterschiede im erweiterten Befehlssatz hinsichtlich Umfang, Syntax und Semantik. Zur Nutzung dieser Module sind daher die entsprechenden Handbücher zu studieren. Dieser Umstand ist insbesondere im Rahmen eines Anwendungsszenarios der Nutzung verschiedener Module zu berücksichtigen.

4 Mapping zwischen SWE und Moving Features

Ausgehend von einem Framework wird untersucht, ob die darin enthaltenen Modelle auch in irgendeiner Art und Weise im anderen Framework erfasst werden und umgekehrt. Es ist zunächst festzustellen, dass im *Moving-Features*-Framework Sensoren sowie Sensordaten keinen expliziten Schwerpunkt bilden. Vielmehr handelt es sich bei *Moving Features* um eine Kodierungsmethode für den effizienten Datenaustausch von bewegten Geoobjekten. Diese Eigenschaft steht im Gegensatz zum umfassenden Fundus an Werkzeugen zur Modellierung von Sensoren sowie Beobachtungen im SWE-Framework. Im Hinblick auf mobile Sensoren rücken diese beiden Bereiche jedoch näher heran. Eine mögliche Speicherung von Sensordaten im *Moving-Features*-Framework lässt sich durch die in diesem Standard vorkommenden Attribute realisieren. Letztere stellen, mit einem festen Set an Datentypen, ein Werkzeug zur Modellierung beliebiger Zustände eines Geoobjekts dar. Es lässt sich sagen, dass auf der einen Seite bei *Moving Features* die Positionen von Geoobjekten bzw. deren Trajektorien den Fokus bilden. Attribute geben darüber hinaus wertvolle Informationen über den Zustand von sich bewegenden Geoobjekten. Auf der anderen Seite liegt der Schwerpunkt im SWE-Framework auf Sensoren sowie deren Daten. Bei mobilen Sensoren tritt zusätzlich die räumliche Komponente dazu, die Position des sich bewegenden Sensors wird relevant. In den folgenden Abschnitten wird daher auf mögliche Abbildungsregeln zwischen diesen Frameworks eingegangen. Gefundene Abbildungsregeln bedürfen einer konkreten Umsetzung, beispielsweise mittels XML-Manipulationswerkzeugen, welche jedoch in dieser Arbeit nicht weiter behandelt werden. XML-Elemente aus *Moving Features* werden durch den Namensraum *mf* gekennzeichnet und lassen sich somit von XML-Elementen aus dem SWE-Framework unterscheiden.

4.1 XML-basierte Abbildung

Ausgehend von der Struktur sowie Semantik eines *Moving-Features*-XML-Dokuments wird untersucht, ob SWE-Dokumente in dieses Format konvertiert und welche Daten dabei aus SWE extrahiert werden können. Das erste Element eines solchen XML-Dokuments, das *mf:stBoundedBy*-Element, enthält den Zeitraum und die räumliche Ausdehnung der Geoobjekte. Diese Informationen lassen sich von SWE-Dokumenten nach *Moving Features* übertragen.

Konkret befinden sich solche Daten in einem *GetCapabilitiesResponse*-Dokument. Darin wird der Beobachtungszeitraum eines Sensors anhand des *sos:phenomenonTime*-Elements angegeben. Zu beachten ist die Tatsache, dass eine weitere Zeitangabe durch das Element *sos:resultTime* existiert. Diese gibt die Zeit an, nach welcher die Beobachtung, nach vollständiger Prozessierung, endgültig vorliegt. Abhängig vom Beobachtungsmodell ist diese Zeitangabe mit der Zeit der erstmaligen Erfassung der Beobachtung identisch, Abweichungen davon sind jedoch bei einer Abbildung vom SWE-Framework nach *Moving Features* zu berücksichtigen.

Die räumliche Ausdehnung eines Geoobjekts im *Moving-Features*-Dokument wird in in den Kindelementen *gml:lowerCorner* sowie *gml:upperCorner* angegeben. Diese Elemente spezifizieren die untere rechte Ecke bzw. obere linke Ecke des begrenzenden Raumes, d. h. die *minimum bounding box* des *Moving Features*. Analog wird im SWE-Framework die räumliche Ausdehnung aller Beobachtungen eines Sensors im *GetCapabilitiesResponse*-Dokument im Abschnitt *Contents* mittels des Elements *observedArea* angegeben. Hier haben die Kindelemente *gml:lowerCorner* und *gml:upperCorner* jedoch eine andere Definition: Diese spezifizieren die untere linke Ecke respektive die obere rechte Ecke der *minimum bounding box*.

Die *mf:member*-Elemente enthalten Metadaten der *Moving Features*. Das SWE-Framework unterstützt ebenso die Modellierung von Metadaten eines Geoobjekts. Eine Abbildung solcher Informationen ist mittels des *feature of interest* in SWE möglich. Die genaue Vorgehensweise bei einer solchen Abbildung hängt vom gewählten Konzept des bewegten Objekts ab: Im Falle des Telemetriesystems ist das landwirtschaftliche Fahrzeug, in welchem sich der Datenlogger befindet und *in situ* Daten erfasst, das sog. *feature of interest*. Es liegt daher nahe, das Fahrzeug als *Moving Feature* aufzufassen und mittels der *mf:member*-Elemente zu kodieren. Eine solche Abbildung muss jedoch nicht zwangsläufig vorgenommen werden. Häufig stellt das *feature of interest* ein statisches Objekt dar, dessen Eigenschaften durch ein bewegtes Sensorsystem aus der Ferne (*remote*) beobachtet werden. Dies ist beispielsweise die Erdoberfläche, von welcher Luftbilder aus einer luftgestützten Sensorplattform gemacht werden. In einem solchen Fall ist von einer Abbildung des *feature of interest* (hier Erdoberfläche) auf ein *Moving Feature* abzusehen. Vielmehr müssen die Metadaten des bewegten Sensors, beispielsweise Sensorbezeichnung, Seriennnummer etc., zur Abbildung auf *Moving Features* herangezogen werden. Der Einfachheit halber, und bezogen auf den in dieser Arbeit behandelten Fall, erfolgt die Abbildung auf ein *Moving Feature* mittels der in einem *feature of interest* gespeicherten Metadaten mit dem *gml:identifier*-Element. Die im *Header*-Abschnitt in einem *Moving-Features*-Dokument definierten Attribute eines solchen Geoobjekts können als Möglichkeit zur konzeptionelle Abbildung von Sensordaten angesehen werden.

Messwerte eines Sensors werden als Zustände des bewegten Geoobjekts aufgefasst. Das Element zur Definition von Attributen *mf:attrDef* enthält die XML-Attribute *name* sowie *type*, welche die Bezeichnung respektive den Datentyp des *Moving-Feature*-Attributs repräsentieren. Die hier vorliegende Bezeichnung *name* korrespondiert mit der Bezeichnung *observedProperty* im SWE-Framework. Der Datentyp kann aus dem XML-Attribut *xsi:type* des *om:result*-Elements übernommen werden. Zu beachten bei der Definition eines *Moving-Feature*-Attributs ist die Tatsache, dass es ohne Einheit angegeben wird. Um Sensorbeobachtungen trotzdem in ein

Moving-Features-Dokument einfließen zu lassen, kann dies indirekt durch die Beschreibung des *Moving-Features*-Attributs erfolgen. Eine solche textuelle Beschreibung wird durch das *Moving-Features*-Element *mf:AttrAnnotation* vorgenommen. Dieses dient zur textuellen Beschreibung der Attribute.

Den Abschluss eines *Moving-Features*-Dokuments stellt das *mf:foliation*-Element dar. Dieses Element beinhaltet die Trajektorien der bewegten Geoobjekte sowie konkrete Werte der im *Header*-Abschnitt definierten Attribute. Start- und Endzeitpunkte der Trajektorien in *Moving Features* können aus den Zeitstempeln der *sos:phenomenonTime*-Elementen aus einem *GetObservationResponse*-Dokument gebildet werden. Letzteres enthält ebenso die Positionen der Beobachtungen. Diese werden bei räumlichen Beobachtungen im *om:parameter*-Element angegeben.

4.2 Beobachtungsabbildung durch Attribute

Ein Attributswert im *Body*-Teil eines *Moving Features*-Dokuments gilt innerhalb einer Trajektorie und zwei Zeitpunkten. Die zwei Zeitpunkte spezifizieren Start- und Endzeitpunkt der Trajektorie. Beim hier vorliegenden Datenmodell des Telemetriesystems jedoch, besitzt ein Messwert genau einen Zeitstempel. Aus diesem Grund ist eine einfache Abbildung von georeferenzierten Messwerten des Telemetriesystems auf die hier vorliegenden Attribute nicht ohne Weiteres möglich. Abbildung 4.1 veranschaulicht den Gültigkeitsbereich der *Moving-Features*-Attribute:

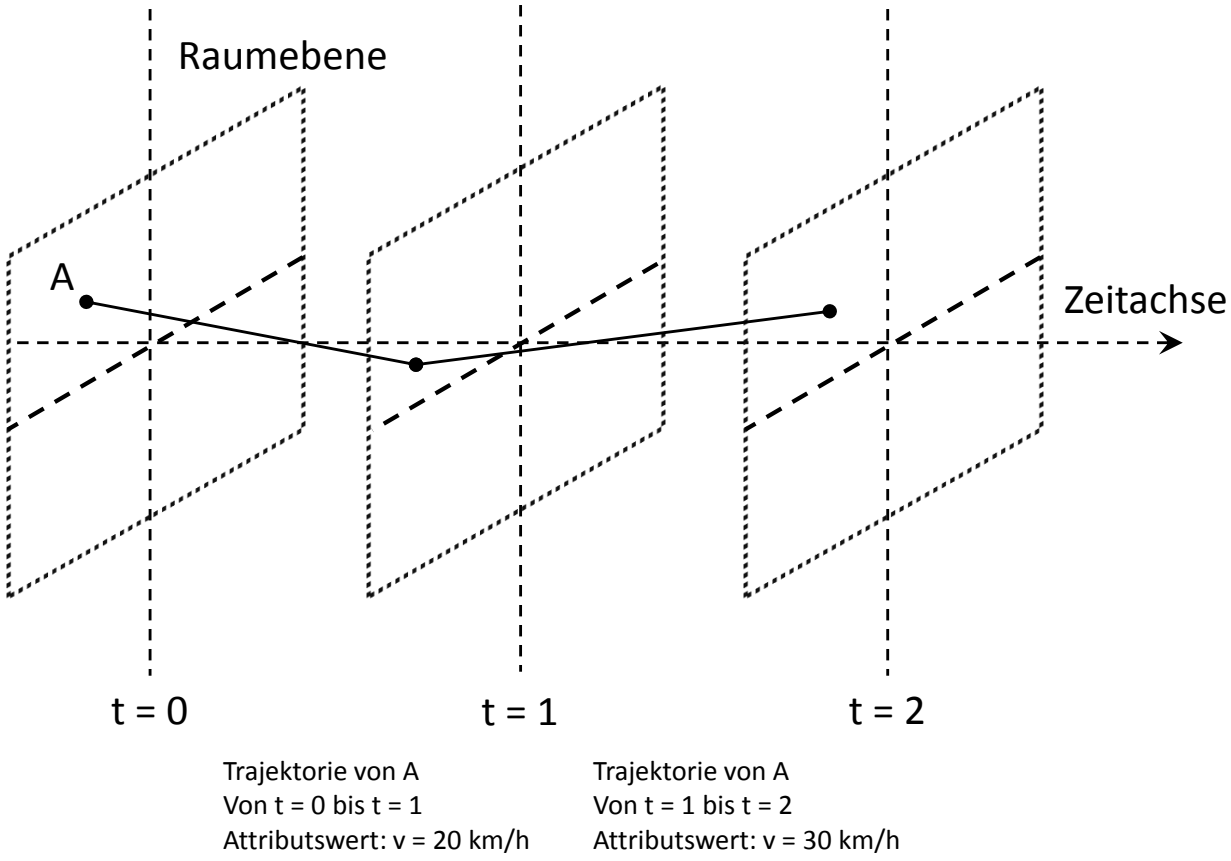


Abbildung 4.1: Attribute in *MovingFeatures*, hier beispielhaft ein Geschwindigkeitsattribut

In dieser Abbildung sind zwei Trajektorien des bewegten Objekts A dargestellt. Von einem Zeitpunkt zum nächsten Zeitpunkt hat das Objekt einen konstanten Zustand. In diesem Fall ist es das Attribut Geschwindigkeit (Bezeichnung v), der Attributswert beschreibt somit eine konstante Geschwindigkeit. Eine neue Trajektorie (zum Zeitpunkt t = 1) wird erstellt, da sich Objekt A nun mit einer höheren Geschwindigkeit fortbewegt. Bei diesem Beispiel wird auch ersichtlich, dass unklar ist, wie die Geschwindigkeitsänderung zustande kommt. Die im Beispiel vorgenommene Angabe des Geschwindigkeitsattributs ist syntaktisch in *Moving Features* korrekt. Dass jedoch

das Objekt A von einem Zeitpunkt auf den anderen Zeitpunkt eine Geschwindigkeitssteigerung von 10 km/h erfährt, ist physikalisch nicht möglich, da die dazu nötige Beschleunigung nicht berücksichtigt wird. Tatsächlich gibt der Standard in einem Beispiel an [Asahara et al., 2015b], Werteintervalle vorzuziehen (siehe Listing 4.1).

```
<mf:AttrDef name="tachometerAnzeige" type="xsd:string">
  <mf:AttrAnnotation>
   Dieses Attribut zeigt die Anzeige eines Fahrzeugtachometers.
   Langsam: weniger oder gleich 20 km/h
   Mittel: mehr als 20 km/h und weniger oder gleich 100 km/h
   Schnell: mehr als 100 km/h
  </mf:AttrAnnotation>
</mf:AttrDef>
```

Listing 4.1: Beispiel für ein *Moving Features*-Geschwindigkeitsattribut nach [Asahara et al., 2015b]

Diese Modellierung ist jedoch nicht verpflichtend, sie wird im Referenzdokument nebenläufig erwähnt. Eine solche Definition würde die in der Abbildung 4.1 gegebenen numerischen Attributswerte durch zwei textuelle Attributswerte der Trajektorien ersetzen (Langsam bzw. Mittel). Die im Rahmen der Arbeit vorliegenden Messungen (z. B. Geschwindigkeit aus GPS-Informationen) erfahren eine andere Darstellung (siehe Abb. 4.2).

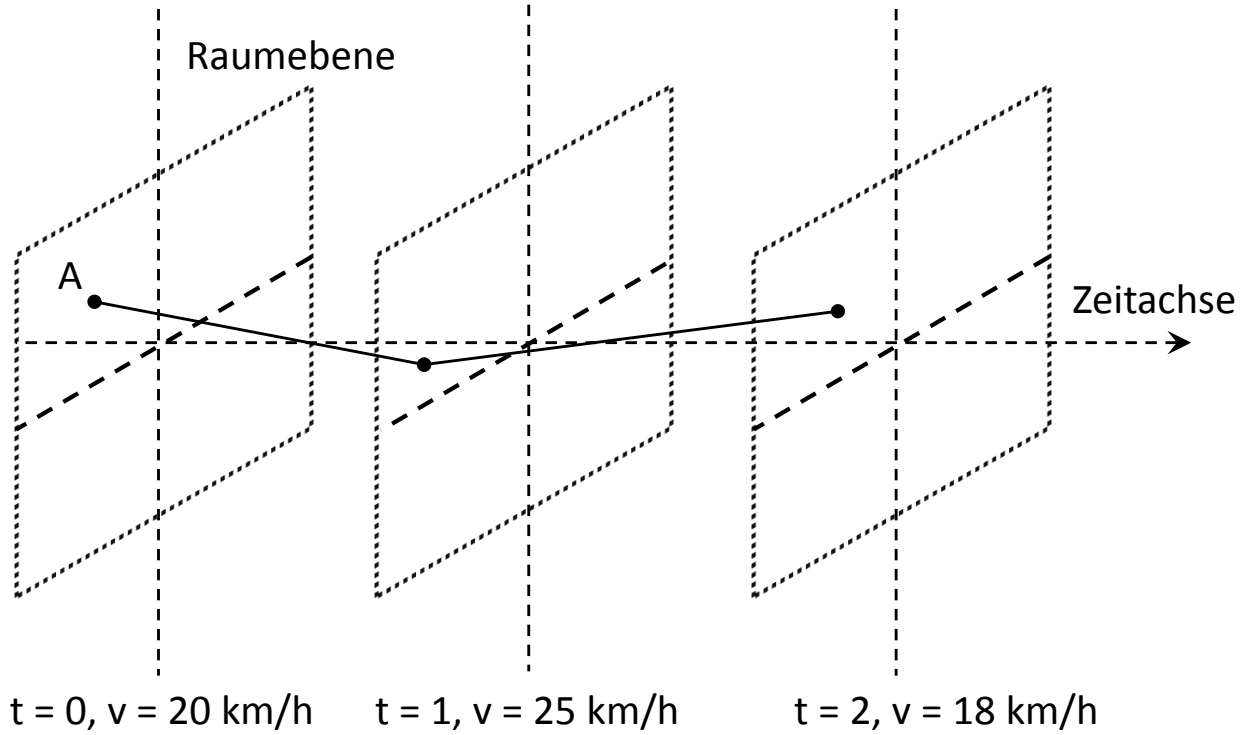


Abbildung 4.2: Beobachtete Geschwindigkeit eines Objekts A zu Zeitpunkten t

In der obigen Abbildung hat das bewegte Objekt A zu jedem Zeitpunkt eine Momentangeschwindigkeit. Um diese Messwerte in Zustände eines *Moving Features* abbilden zu können, wurde hier der Ansatz der **Zustandsänderungsrepräsentation** durch den Attributswert konzipiert: Der Attributswert repräsentiert in diesem Fall die relative Änderung des Zustands vom Endzeitpunkt zum Zustand des Startzeitpunkts (siehe Abb. 4.3).

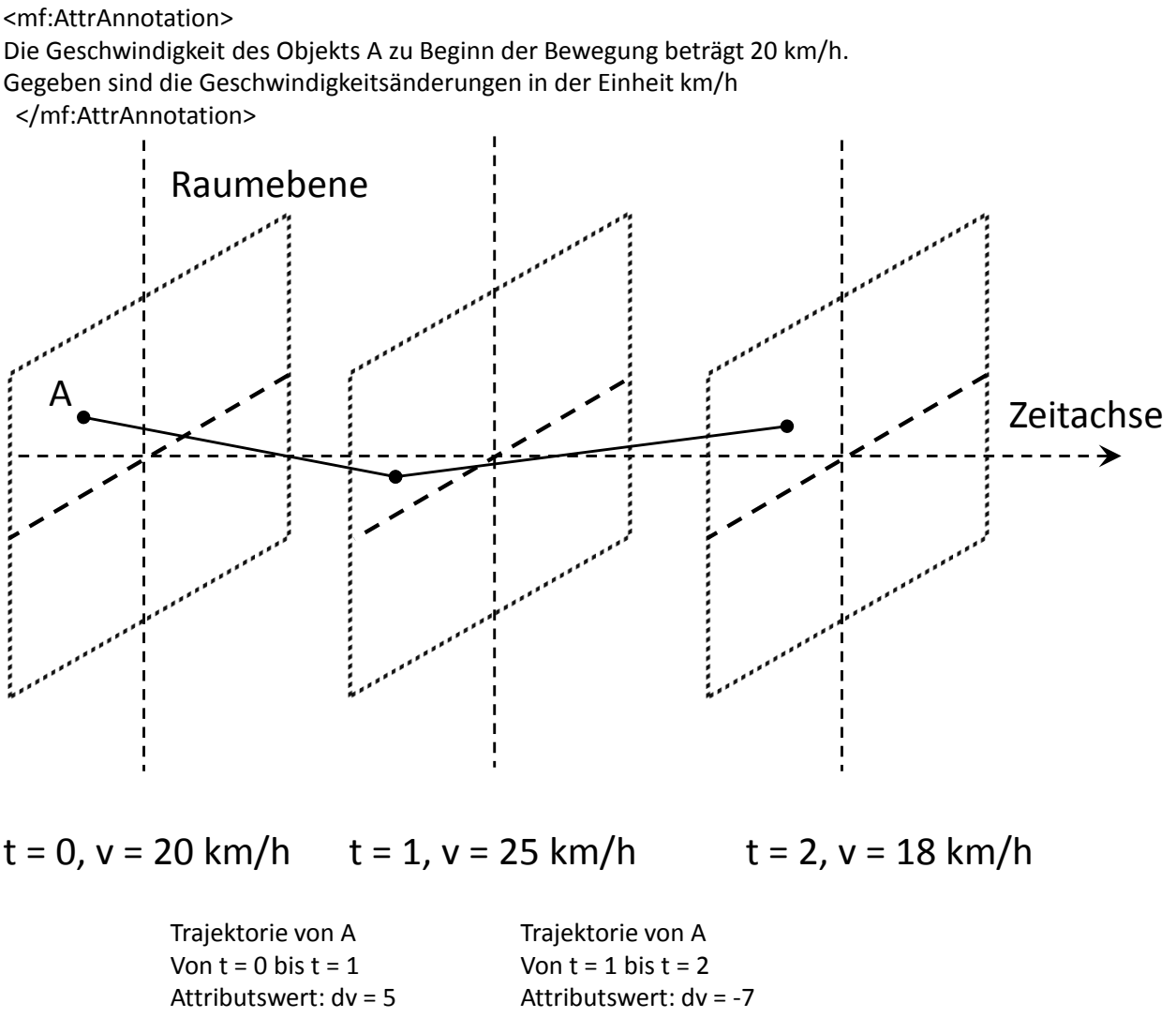


Abbildung 4.3: Abbildung von Momentangeschwindigkeiten in Geschwindigkeitsänderungen

In der Abbildung sind die absoluten Beobachtungen der Geschwindigkeit des Objekts A gegeben (Werte mit Einheit km/h). Wenn diese Daten (sie kommen beispielsweise in einem *GetObservationResponse*-Dokument vor) in ein *Moving-Features*-Dokument überführt werden sollen, wird der erste Beobachtungswert in die Attributsannotation geschrieben und kann als Additionskonstante angesehen werden. Alle anderen Beobachtungen werden relativ in Bezug auf die erste Messung als Attributswerte der Trajektorien gesetzt. Somit werden die Messwerte den Zeitpunkten einer

Trajektorie zugeordnet, gleichzeitig wird die Bedingung der *Moving-Features*-Attribute eingehalten: Innerhalb einer Trajektorie ist der durch das Attribut beschriebene Zustand konstant. Wenn umgekehrt die absoluten Beobachtungswerte abgeleitet werden sollen, um diese beispielsweise in ein *InsertObservation*-Dokument einzufügen, wird der erste Attributswert mit dem Wert in der Attributsannotation addiert. Alle übrigen Werte leiten sich aus den relativen Änderungswerten ab.

4.3 Variierende Zustände

Die CSV-Kodierung von *Moving Features* weist im Vergleich zur XML-Kodierung Unterschiede auf. Diese Unterschiede erschweren die Datentransformation mittels der in 4.2 vorgeschlagenen Abbildungsregeln. So existieren hier keine Mittel zur Annotation von Attributen. Damit bleibt die Möglichkeit verwehrt, absolute Anfangswerte von Beobachtungen aus dem SWE-Framework in der Attributsannotation zu speichern. Formal wäre die Definition eines benutzerdefinierten Metadatenbezeichners denkbar. Jedoch gibt es keine Angaben zur Unterstützung solcher Bezeichner im *Moving-Features*-Standard. Explizit sind in der CSV-Kodierung von *Moving Features* lediglich die Bezeichner *stboundedby* sowie *columns* zur Beschreibung von Zeit-Raumbegrenzung respektive der Trajektoriedaten definiert [Asahara et al., 2015a]. Zur Abbildung von Beobachtungsdaten aus dem SWE-Framework kann die Verwendung von zwei Attributen mit gleichem Datentyp und gleicher Semantik herangezogen werden. Die Bezeichnungen der Attribute hingegen unterscheiden sich. Diese können für das erste Attribut in „Startzustand“ und für das zweite Attribut in „Endzustand“ vorgenommen werden. Damit repräsentiert der Wert des ersten Attributs den Messwert zum Startzeitpunkt der Trajektorie. Analog gilt der Wert des zweiten Attributs als Messwert zum Endzeitpunkt der Trajektorie. So kann beispielsweise die momentane Fahrzeuggeschwindigkeit zu jeweils einem Zeitpunkt durch zwei Zeitpunkte einer Trajektorie angegeben werden (siehe Listing 4.2).

```
@stboundedby,urn:x-ogc:def:crs:EPSG:6.6:4326,2D,50.23 9.23,50.31 9.27,2012-01-17T12:33:40Z
    ,2012-01-17T12:37:50Z,sec
@columns,mfidref,trajectory,tachometerAnzeigeStart,xsd:integer,tachometerAnzeigeEnde,
    xsd:integer
a,10,150,50.23 9.23 50.30 9.26,20,30
...
```

Listing 4.2: *Moving-Features*-CSV-Beispiel mit variierenden Zuständen

Dadurch ist der Zustand innerhalb einer Trajektorie nicht konstant, sondern variiert. Diese Vorgehensweise lässt sich bei den Abbildungsregeln der XML-Kodierung ebenso vornehmen. Jedoch steht dieser hier konzipierte Transformationsansatz der **Variierenden Zustände** im Widerspruch zum eigentlichen konzeptionellen Modell von Attributen in *Moving Features*. Nimmt man das o. g. genannte Beispiel der Fahrzeuggeschwindigkeit, so ist die Angabe von zwei Attributen mit Geschwindigkeitsangaben mehrdeutig: Das Fahrzeug weist innerhalb der

Trajektorie zwei konstante Geschwindigkeiten auf, da ein Attribut im Sinne des *Moving-Features*-Frameworks einen konstanten Zustand innerhalb einer Trajektorie beschreibt. Um den Ansatz der Varrienden Zustände innerhalb des Standards *Moving Features* zu realisieren, müsste selbiger um dieses Konzept erweitert werden.

5 Systemkonzeption

In den folgenden Kapiteln wird auf das Konzept des zu entwickelnden Systems eingegangen. Es wird der aktuelle Stand zur Modellierung mobiler Sensoren im Rahmen des SWE untersucht. Davon wird abgeleitet, wie die Georeferenzierung der Prozessdaten und die entsprechende Kodierung erfolgen soll. Eine konzeptionelle Beschreibung des zu entwickelnden Telemetriesystems untersucht die Beschaffenheit eines Telemetriesystems, welches Daten über eine TCP-Verbindung an einen SOS sendet. Dabei werden die Daten zunächst in einer eigens dafür erstellten Datenbank gespeichert. Es werden jedoch lediglich Beobachtungen an den SOS gesendet, welche einen Raumbezug aufweisen, um die Unterstützung von mobilen Sensoren durch den SOS zu erproben. Alle übrigen Daten werden nicht weiter berücksichtigt, es ist jedoch grundsätzlich möglich, auch diese in den SOS zu überführen.

5.1 Modellierung mobiler Sensoren im SWE-Framework

Im SWE-Framework existieren unterschiedliche Möglichkeiten zur Georeferenzierung von Sensoren (Abb. 5.1).

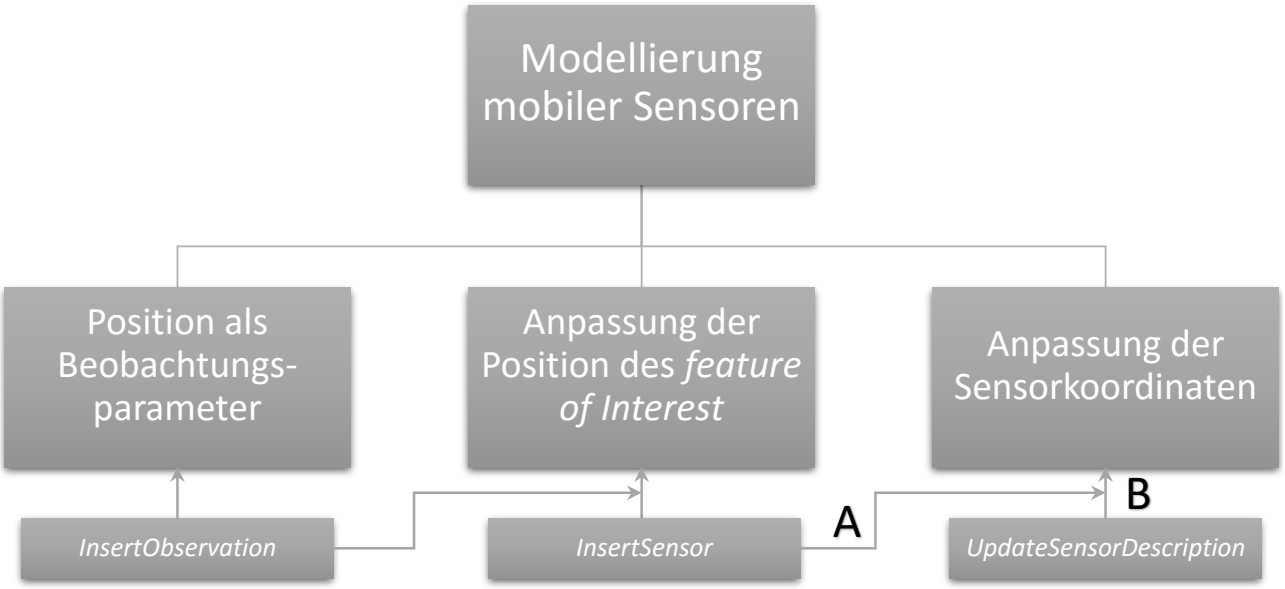


Abbildung 5.1: Modellierung mobiler Sensoren im SWE-Framework

Zum einen kann die Sensorposition durch eine *InsertSensor*-Operation übergeben werden (in der Abbildung durch den Großbuchstaben A gekennzeichnet). Solange der Sensor seine registrierte Position nicht verändert, ist dieser Schritt für die Georeferenzierung des Sensors ausreichend. Verändert sich die Position, so kann mit der Operation *UpdateSensorDescription* eine modifizierte Beschreibung des Sensors übergeben werden (in der obigen Abbildung durch den Großbuchstaben B gekennzeichnet). Das hier vorliegende Telemetriesystem verändert seine Position laufend. Es liegt daher nahe, kontinuierlich die Position in der Sensorbeschreibung mit der entsprechenden *UpdateSensorDescription*-Operation an die tatsächliche Position anzupassen. Zudem erfasst der Datenlogger in regelmäßigen Intervallen Prozessdaten und überträgt diese an den SOS mittels *InsertObservation*-Operationen. Somit müssten mit einer Beobachtung zwei SOS-Operationen durchgeführt werden. Das SWE-Framework stellt mit der Parametrisierung von Beobachtungen (siehe Kapitel 3.4.2) Möglichkeiten zur Verfügung, Beobachtungsdaten mit Zusatzinformationen zu versehen. Ein solcher Parameter kann der Beobachtung, durch Angabe einer georeferenzierten Geometrie, eine Position zuordnen. Die Parametrisierung wird im Zuge einer *InsertObservation*-Operation vorgenommen. Jede parametrisierte Beobachtung weist demnach, neben dem Messwert der zu beobachtenden Größe, ein *om:parameter*-Element auf, welches die geometrische Eigenschaft der Beobachtung enthält. Das Telemetriesystem wird in diesem Falle indirekt durch die Beobachtungen georeferenziert. Eine dedizierte Georeferenzierung des Sensors mittels der Sensorbeschreibung entfällt. Im SWE-Framework besteht auch die Möglichkeit, das *feature of interest* zu georeferenzieren. Dies ist sowohl mittels einer *InsertSensor*-Operation als auch mittels einer *InsertObservation*-Operation möglich (in Abb. 5.1 mit Pfeilen sowohl vom *InsertSensor*- als auch vom *InsertObservation*-Block dargestellt). Je nach Anwendungsszenario ist eine geeignete Modellierung vorzunehmen: Wird beispielsweise ein Fahrzeug als *feature of interest* von einer luftgestützten Sensoreinheit erfasst, so unterscheiden sich die Sensorkoordinaten von den Fahrzeugkoordinaten. In einem solchen Fall ist eine Georeferenzierung sowohl für den Sensor als auch für das zu beobachtende Objekt vorzunehmen. Im hier vorliegenden Fall befindet sich der Sensor auf dem Fahrzeug selbst. Werden Position und Orientierung des Datenloggers nicht in Bezug auf den Fahrzeugkörper berücksichtigt, so kann die Position für beide Objekte als die gleiche angenommen werden. Dann repräsentiert das *feature of interest* lediglich das zu erfassende Objekt. Die Position des Geoobjekts wird daher indirekt durch parametrisierte Beobachtungen wiedergegeben. Die Positionsangabe wird dabei pro Beobachtungstyp – beispielsweise sind das bei Drehzahl und Kraftstoffverbrauch zwei Typen – mit je einem *om:parameter*-Element (mit räumlichen Informationen) vorgenommen. Das heißt wiederum, pro *OM_Observation*-Instanz wird somit ein Parameter für die Georeferenzierung der Beobachtung verwendet. Das Listing 5.1 veranschaulicht diese Parametrisierung anhand einer Drehzahl-Beobachtung in einem *InsertObservation*-Dokument.

```
...
<om:parameter>
  <om:NamedValue>
    <om:name
      xlink:href="http://www.opengis.net/def/param-name/OGC-OM/2.0/samplingGeometry" />
    <om:value xsi:type="gml:GeometryPropertyType">
      <gml:Point gml:id="RPMPositionID">
        <gml:description>description</gml:description>
```

```
      <gml:identifier codeSpace="">ObservationPositionIdentifier
      </gml:identifier>
        <gml:name>hereIam</gml:name>
        <gml:pos srsName="http://www.opengis.net/def/crs/EPSG/0/4326">42.45 11.98</gml:pos>
    </gml:Point>
    </om:value>
  </om:NamedValue>
</om:parameter>
<om:observedProperty
  xlink:href="http://www.52north.org/test/observableProperty/engineRPM" />
<om:featureOfInterest
  xlink:href="http://www.52north.org/test/featureOfInterest/Vehicle" />
<om:result xsi:type="xs:integer">2400</om:result>
...
```

Listing 5.1: Ausschnitt einer Beobachtung der Motordrehzahl mit einem räumlichen Parameter

In diesem Ausschnitt eines *InsertObservation*-Dokuments werden Parameter, Beobachtungsgröße, das *feature of interest* sowie das Messergebnis (Beobachtungsergebnis) dargestellt. Koordinaten und Messwert werden als Textwerte der entsprechenden XML-Elemente angegeben und Beobachtungsgröße sowie das *feature of interest* werden mittels *xlink*-Attributen referenziert. Basierend auf diesem Modellierungsansatz wird ein *InsertObservation*-XML-Dokument mit parametrisierten räumlichen Beobachtungen entworfen. Im Rahmen des implementierten *Workflow* wird dieses Dokument für jede neue Beobachtung des Telemetriesystems an den entsprechenden Stellen modifiziert und einer SOS-Instanz übergeben. Das vollständige Dokument befindet sich in Anhang B.5. Die Freiheit in der Modellierung der Rauminformation eines Sensors und dessen Beobachtungen im SWE-Framework stellt somit hohe Anforderungen an Anwendungen, welche solche Daten verarbeiten. Um allen Modellierungsansätzen gerecht werden zu können, ist eine hohe Flexibilität der verarbeitenden Anwendungen (Web-Clients, GIS-Systeme etc.) notwendig. Im Rahmen dieser Arbeit wird der Ansatz der parametrisierten Beobachtungen für die Georeferenzierung der Sensoren verfolgt, da dieser von 52°North im Hinblick auf zukünftige Entwicklungen des SOS sowie zugehöriger Clients genutzt wird. Zurzeit unterstützt der JavaScript-Client von 52°North lediglich die Visualisierung von Zeitreihen stationärer Sensoren. Dabei wird gegenwärtig zur Positionierung der Sensorstation die Geometrie des beobachteten *feature of interest* herangezogen. Die in der *InsertSensor*-Operation mitgegebene Sensorposition wird nicht berücksichtigt. Konsekutive Änderungen der Geometrie des *feature of interest* werden ebenfalls nicht berücksichtigt. In Zukunft soll die Visualisierung von Beobachtungsdaten eines mobilen Sensors durch parametrisierte Beobachtungen ermöglicht werden.

5.2 SOS-Kommunikation bei TCP-basierter Datenübertragung

Die in Kapitel 1.4 genannten Einschränkungen bzgl. der Kommunikation zwischen Datenlogger und SOS legen die Einrichtung einer intermediären Kommunikationsschicht nahe. Daher bildet die Entwicklung eines Proxys einen wesentlichen Bestandteil dieser Arbeit (siehe Abb. 5.2). Ein solcher Proxy wartet auf Verbindungsanfragen und erlaubt die Herstellung einer TCP-Verbindung. Auch die Weitergabe der Rohdaten an eine Datenbank, das Abrufen georeferenzierter Daten, Datenverarbeitung und Kodierung gehören zu den Aufgaben dieser Zwischenschicht. Somit wird eine indirekte Datenübertragung vom Datenlogger zum SOS ermöglicht. Die auf dem Server gespeicherten Rohdaten werden in SOS-konforme Dokumente restrukturiert und über HTTP-POST auf den SOS übertragen. Dabei wird der Kommunikationskanal des Datenloggers entlastet, da über die TCP-Verbindung vergleichsweise geringe Datenmengen übertragen werden. Dieser zu entwickelnde Proxy und der SOS befinden sich, im Rahmen dieser Arbeit, auf dem selben Server. Diese Tatsache schließt jedoch den Einsatz zweier separierter Rechner nicht aus. Im Folgenden wird Server synonym mit Proxy verwendet, wenn nicht anders angegeben.

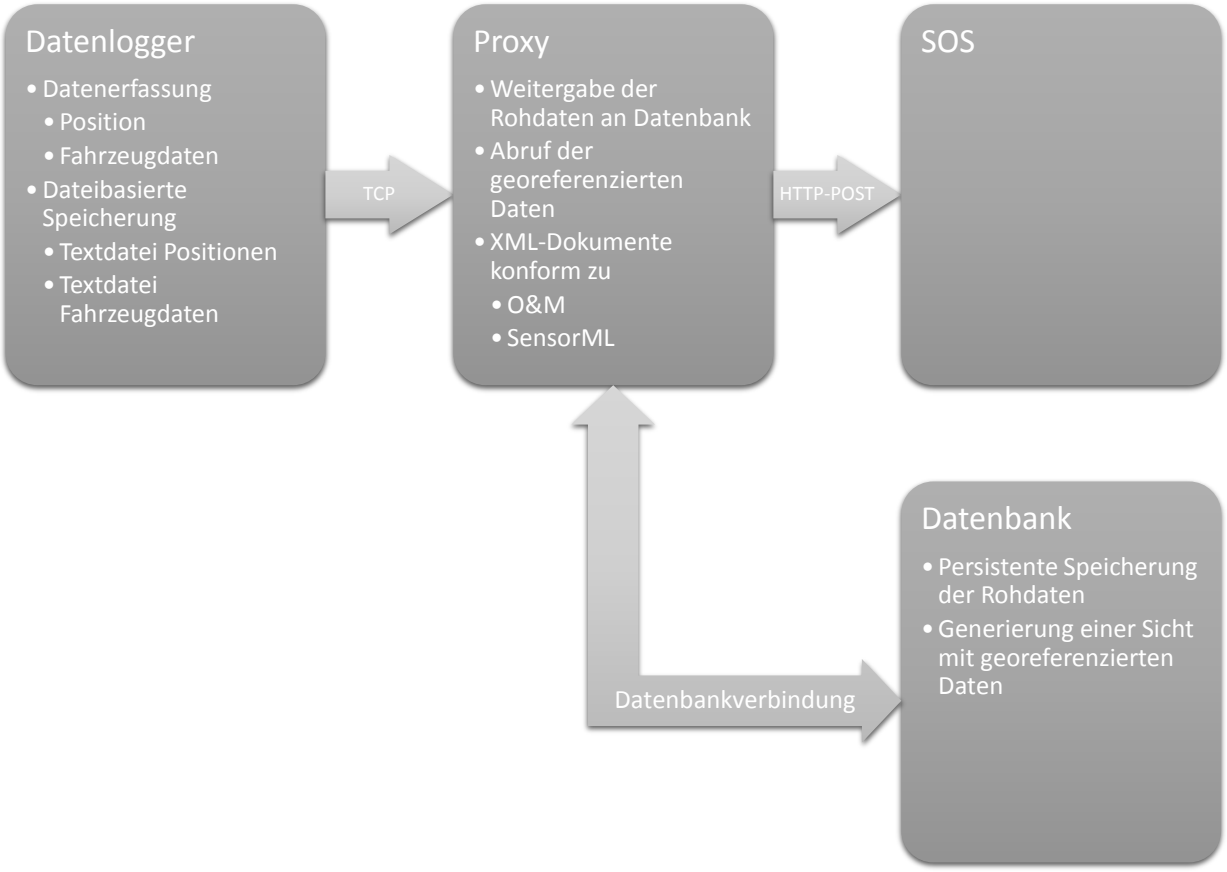


Abbildung 5.2: SOS-Kommunikation bei TCP-basierter Datenübertragung

5.3 Georeferenzierung der Prozessdaten

Zur Georeferenzierung von Prozessdaten muss ein Schlüssel gefunden werden, mit welchem Positions- und Fahrzeugdaten identifiziert und verknüpft werden können. Dies ist nötig, da die Daten mit unterschiedlicher zeitlicher Auflösung und durch unterschiedliche Programme erfasst werden. Für diese Arbeit wurde dazu eine solche Kennung basierend auf einem Zeitstempel der lokalen Zeit des Datenloggers erstellt. Dieser gibt die Zeit in Sekunden, inkl. Nachkommastellen, in Bezug auf das Referenzdatum 01.01.1970, 00:00:00 Uhr an. Solche Kennzahlen werden sowohl an die Positionsdaten als auch an die Fahrzeugdaten zum Zeitpunkt ihrer Generierung vergeben. Da die erneute Inbetriebnahme des Datenloggers sowie ein Ausfall, beispielsweise durch äußere Einwirkungen, die lokale Uhrzeit zurücksetzt, wird ein weiteres Identifikationsmerkmal herangezogen, um Mehrdeutigkeiten aufzulösen. Dazu wird zusätzlich zur lokalen Uhrzeit mit jeder Inbetriebnahme des Systems eine Zählervariable inkrementiert. Das Listing 5.2 veranschaulicht diese Schlüsselvergabe.

```
...
6_1468514638.834464,Datentyp a,Datenwert,Dateneinheit
6_1468514638.835041,Datentyp b,Datenwert,Dateneinheit
...
```

Listing 5.2: Kennung von Daten aus beispielhaftem Ausschnitt

Der Zähler, im vorliegenden Fall trägt dieser den Wert 6, wird an den lokalen Zeitstempel vorangestellt. Im Beispiel ist das System zum Zeitpunkt dieser Messung zum sechsten Mal gestartet, beide Messungen fanden am Donnerstag 14.07.2016, 18:43:58 Uhr (lokale Rechnerzeit) statt. Somit können die Daten des Systems eindeutig identifiziert werden. Eine solche Verknüpfung von Daten, welche in Textdateien gespeichert werden, ist mit erheblichen Schwierigkeiten verbunden. Beispielsweise kann es zu Zugriffskonflikten wegen gleichzeitig stattfindender Schreib- und Leseprozesse kommen. Daher werden die Daten zuerst an den Server übertragen und dort in einer Datenbank gespeichert. Die Verknüpfung der Daten geschieht in der Datenbank und diese werden in einer Datenbanksicht aggregiert. Dazu wird die Differenz der Zeitstempel zwischen Prozessdaten und Positionsdaten berechnet, welche die gleiche Zählervariable gemein haben. Die geringste Differenz kennzeichnet dann zum annähernd gleichen Zeitpunkt erfasste Prozess- bzw. Positionsdaten.

5.4 Datenfluss des Datenloggers

Vorgesehen für den Datenlogger sind, neben einer Recheneinheit, Komponenten zur Erfassung von Position und Fahrzeugdaten und Übertragung dieser an einen Server. Dabei können auch Kombinationen davon, beispielsweise ein GPS-GSM-Modul, verwendet werden (wie in dieser Arbeit vorgesehen), was jedoch die grundsätzliche Vorgehensweise nicht beeinflusst. Sämtliche Komponenten werden sowohl in der Hardware als auch in der Software konfiguriert. Die Datenerfassungskomponenten speichern ihre Beobachtungen lokal auf dem Datenlogger ab. Dies

geschieht dateibasiert, d. h. für jede Komponente wird jeweils eine Textdatei, welche die Daten aufnimmt, eingerichtet. Die Datenübertragungskomponente greift auf den Inhalt der Textdateien zu und bereitet diese für die Datenübertragung an den Server vor. Bei der vorgesehenen Datenübertragungskomponente geschieht dies befehlsbasiert, d. h. alle Aktionen (Verbindungsherstellung, Datenübermittlung etc.) werden als Antwort auf den jeweiligen AT-Befehl (in 3.7 behandelt) ausgeführt. In den folgenden Kapiteln wird genauer auf den konzeptionellen *Workflow* des Datenloggers eingegangen.

5.4.1 Datenerfassung durch Komponenten

Abb. 5.3 veranschaulicht die Datenerfassung sowie lokale Speicherung der Daten.

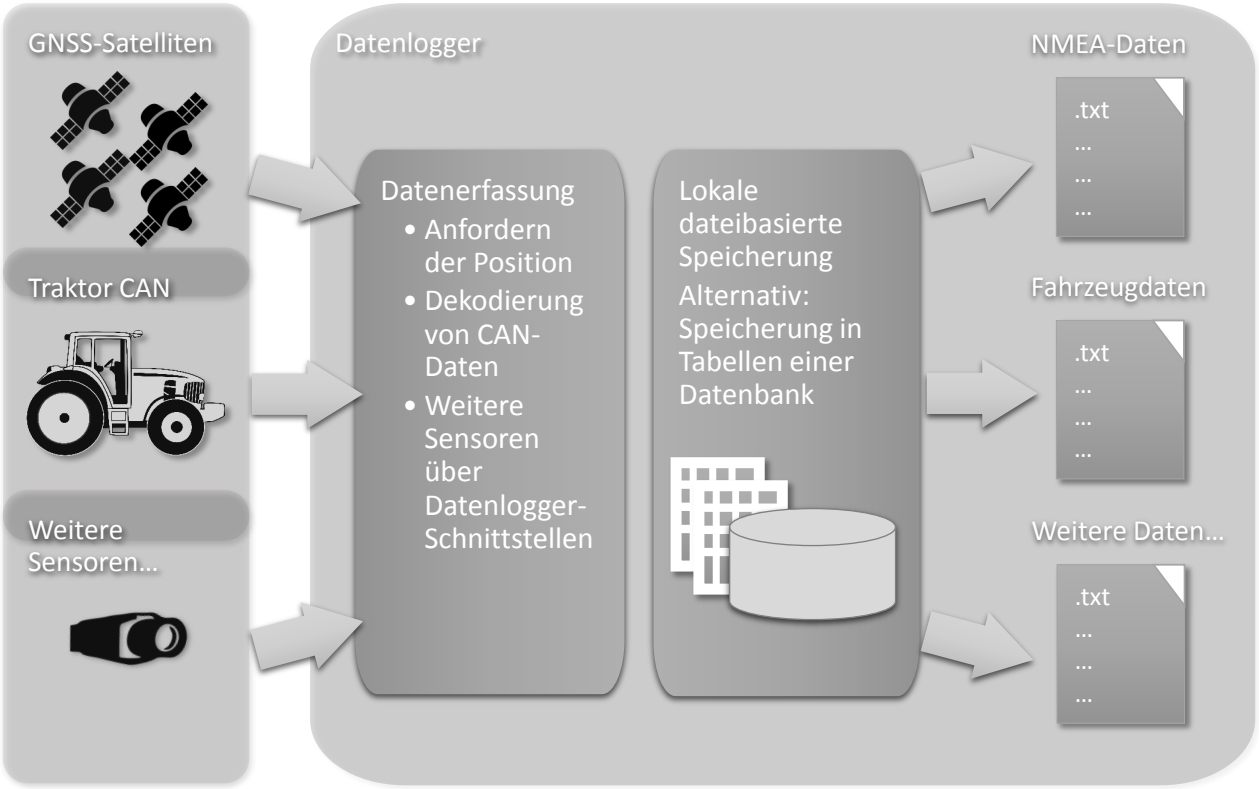


Abbildung 5.3: Datenerfassung und lokale Speicherung durch den Datenlogger

Der Datenlogger weist eine Hardware-Komponente zur Erfassung von Prozessdaten auf. Sobald der Datenlogger mit einem Fahrzeug über die CAN-Schnittstelle verbunden wird, werden über diese Schnittstelle laufend Daten erfasst. Die Erfassung beinhaltet das Initialisieren und Ansprechen der Schnittstelle sowie Abgreifen der Schnittstellenausgabe. Zudem müssen die Daten dekodiert werden, da CAN-Daten in keinem für Menschen lesbaren Format geliefert werden können (sondern binär, hexadezimal etc.). Die dekodierten Daten werden in einer Textdatei

Zeile für Zeile abgespeichert. Eine weitere Komponente ist für die Erfassung der Position des Datenloggers zuständig. Diese Aufgabe übernimmt ein GNSS-Empfänger. Sobald dieser eine Position (geographische Breite, Länge) erfasst hat, werden die Koordinaten, Zeitstempel und weitere NMEA-Daten (Geschwindigkeit, Anzahl Satelliten etc.) ebenso Zeile für Zeile in einer Textdatei abgespeichert. Der zukünftige Einsatz weiterer Komponenten ist ebenfalls möglich. Analog würden dazu Schnittstellen des Datenloggers konfiguriert und genutzt. Weitere Textdateien würden angelegt, um die entsprechenden Messdaten zu speichern. Alternativ wäre in einem solchen erweiterten Fall der Einsatz einer auf Datenlogger-Seite befindlichen Datenbank denkbar.

5.4.2 Datenübertragung durch GSM-Komponente

Zur Datenübertragung auf den Server besitzt der Datenlogger eine Übertragungskomponente. Diese ermöglicht die drahtlose Datenübermittlung an den Proxy über eine TCP-Verbindung. Aufgrund der weiten Distanzen im vorgesehenen Arbeitsfeld fiel die Wahl der drahtlosen Übertragungstechnologie auf Mobilfunk. Realisiert wird eine solche Komponente durch ein GSM-Modul. Zur Übertragung der Daten werden diese aus dem lokalen Speicher, d. h. aus den Textdateien, ausgelesen (Abb. 5.4).

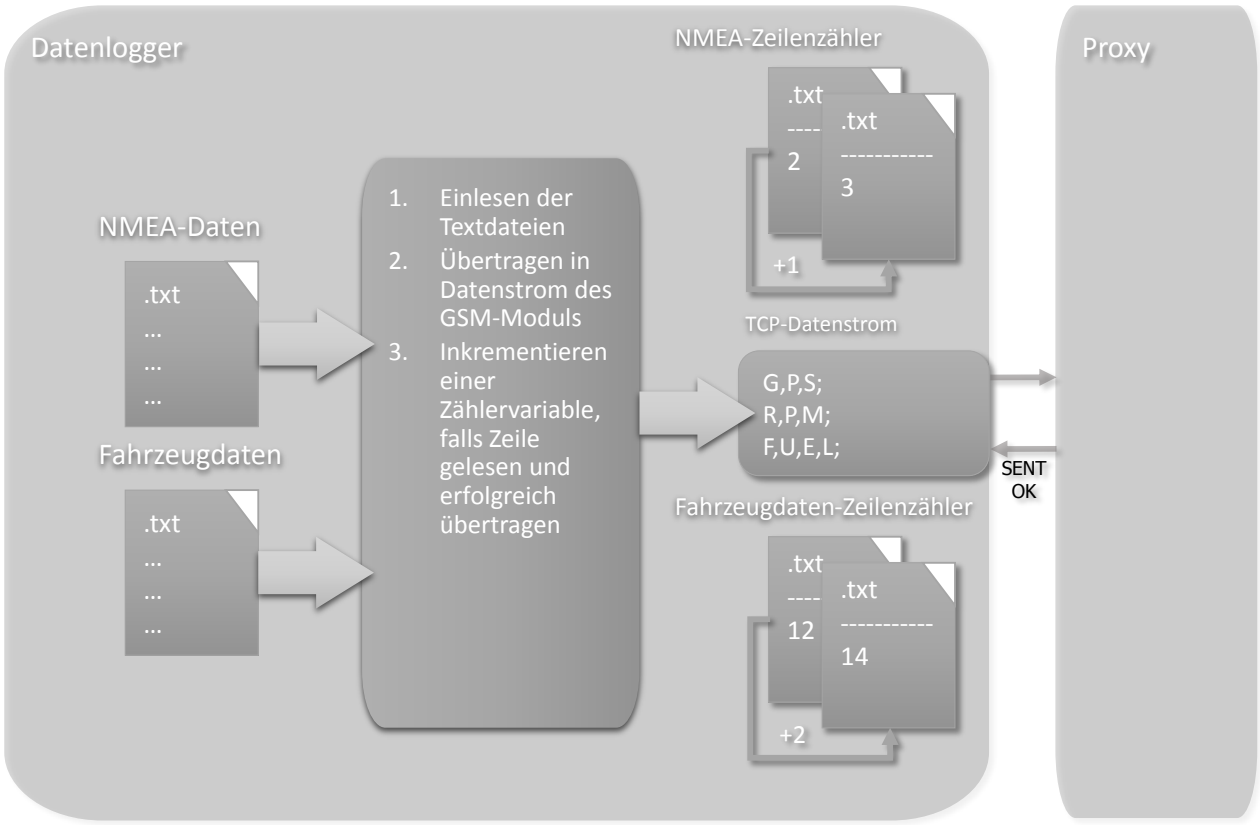


Abbildung 5.4: Datenabruf und Festhalten der gelesenen Zeilen

Da die zu übermittelnde Datenmenge beschränkt ist (bedingt durch die technische Limitierung der maximal zu sendenden Datenmenge im Übertragungskanal des GSM-Moduls), werden die Daten iterativ und ausschnittsweise ausgelesen. Durch einen Lesezähler werden die bereits eingelesenen Datenzeilen festgehalten. Wenn die ausgelesenen Daten erfolgreich an den Server gesendet wurden (in Abb. 5.4 mit Serverantwort *SENT OK* dargestellt), wird der Lesezähler um die Anzahl der gesendeten Datenzeilen inkrementiert. Es wird dadurch festgehalten, welche Daten übertragen wurden. Auch dieser Zähler wird in einer Textdatei gespeichert, um bei einem erneuten Start des Systems (beispielsweise nach einem Stromausfall) wieder dort zu beginnen, wo zuletzt aufgehört wurde. Für jede Datenerfassungskomponente wird ein solcher Zähler konfiguriert. Im vorliegenden Fall werden somit zwei Zähler realisiert, welche jeweils den Stand der Datei mit den Positionsdaten und der Datei mit den Fahrzeugdaten überwachen. Im Schema in der Abb. 5.4 sind die vorgesehenen Merkmale zur Unterscheidung der Datenzeilen zu erkennen. Die einzelnen Werte einer Datenzeile, d. h. eines Datentyps (Position, Drehzahl, Kraftstoffverbrauch), werden durch ein Komma separiert (sinnbildlich durch Buchstabenkürzel dargestellt, die zusammen die Datentypen kennzeichnen). Eine vollständige Zeile einer Beobachtung wird durch ein Semikolon von den anderen Zeilen getrennt.

5.4.3 TCP-Verbindung zwischen Datenlogger und Proxy

Bei der Übertragung wird eine TCP-Verbindung zwischen einem Client und einem Server mittels Sockets aufgebaut. Diese Sockets stellen Datenströme zur Verfügung, über welchen die Datenübertragung vom Telemetriesystem zum Proxy ermöglicht wird. Die Adresse des Proxys (dazu gehört Host und Port) ist im Datenlogger hinterlegt. Diese Informationen werden bei der Verbindungsherstellung übergeben. Dazu wird in einem vorangehenden Schritt eine Verbindung zwischen GSM-Modul und Mobilfunkbetreiber aufgebaut. Die Anmeldung beim Mobilfunkbetreiber erfolgt über einen Zugangspunkt, einen Benutzernamen und ein Passwort. Diese Daten variieren von Betreiber zu Betreiber und sind im Vorfeld einzuholen. Wurde dieser Schritt erfolgreich ausgeführt, wird eine GPRS-Verbindung (*General Packet Radio Service*) initiiert. Dem Modul wird nach erfolgtem Verbindungsaufbau durch den Mobilfunkbetreiber eine IP-Adresse zugewiesen. Der Server besitzt ebenfalls eine IP-Adresse, da dieser, solange er betrieben wird, permanent mit dem Internet verbunden ist. Beide Seiten befinden sich somit nun im Internet. Unter Verwendung der Adresse des Servers und des Ports baut der Datenlogger eine TCP-Verbindung zum Server auf.

5.5 Datenfluss des Servers

Der gesamte Prozess auf der Serverseite besteht grundsätzlich aus zwei Teilschritten. Abb. 5.5 verdeutlicht diesen Prozess.

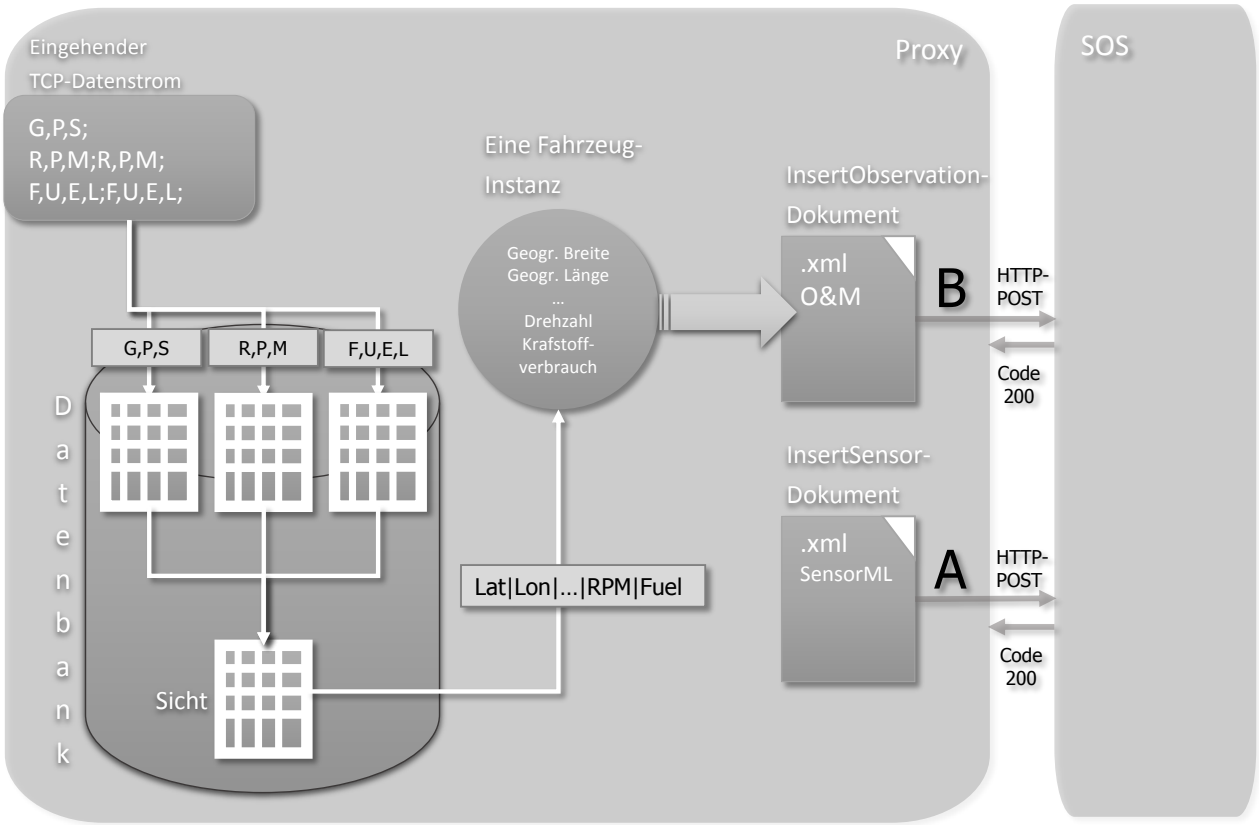


Abbildung 5.5: Workflow auf Serverseite

Beim ersten Teilschritt werden die vom Datenlogger ankommenden Daten in einer eigens dafür angelegten Datenbank abgespeichert. Die Daten werden in Abb. 5.5 als durch Semikola getrennte Sätze dargestellt. Die in einem Satz enthaltenen Werte werden durch Kommata getrennt. Programme und ein Datenbankmodul, welche zu implementieren sind, nehmen die Daten vom Datenstrom des TCP-Sockets entgegen und übernehmen die Verbindung sowie Kommunikation mit der Datenbank. Auf Datenbankseite wird die Minimalkonfiguration verwendet, lesende sowie schreibende Operationen (Abfrage, Datenbanktabellen, Datenbanksicht erstellen) werden durch Programme vorgenommen. Zur Aufnahme der Telemetriedaten in der Datenbank wird dazu für jede Fahrzeugdatenkategorie eine Tabelle generiert. Hinzu kommt eine Tabelle, welche die Positionsdaten aufnimmt. Im vorliegenden beispielhaften Szenario, Erfassung von Motordrehzahl, Kraftstoffverbrauch und Position, werden somit drei Tabellen erstellt. Aus den Daten dieser

Tabellen wird eine Datenbanksicht erstellt. In diese Sicht werden Positionsdaten mit angefügten Fahrzeugdaten aggregiert. Die Verknüpfung geschieht auf Datenbankebene, dazu wird eine SQL-Anweisung formuliert, welche die Kennung der erfassten Daten in die Verknüpfung mit einfließen lässt. In der Datenbanksicht befinden sich dann die georeferenzierten Fahrzeugdaten.

Beim zweiten Teilschritt werden die in der Datenbanksicht enthaltenen Daten auf den SOS übertragen. Dazu werden diese zunächst aus der Sicht abgerufen. Das Abrufen der Daten geschieht auszugsweise, da bei einer großen Anzahl an vorhandenen Daten selbige nicht vollständig in den Arbeitsspeicher geladen werden können. Wurden die Daten extrahiert, werden diese in Objekte abgebildet. Ein solches Objekt hält mit seinen Attributen eine Datenzeile aus der Sicht fest. Zu bemerken ist die absichtliche Darstellung einer einzelnen abgerufenen Zeile aus der Datenbanksicht und die nachfolgende Instanziierung eines einzelnen Objektes. Da in Abb. 5.5 lediglich die Speicherung einer Position dargestellt wird, wird auch nur diese, mit Fahrzeugdaten versehene, Position weiterverarbeitet. Prozessdaten, welche im Verknüpfungsprozess keiner Position zugeordnet wurden, werden für die weiteren Schritte nicht berücksichtigt. Im Anschluss erfolgt die Abbildung eines Objekts in ein O&M-Dokument. Das Dokument wird entsprechend den O&M-Spezifikationen kodiert und in XML implementiert. Die Erzeugung dieses Dokuments geschieht im Vorfeld und mit Beispielwerten. Diese werden durch Objektwerte ersetzt, sobald ein Objekt vorliegt. Im Anschluss wird das Dokument über einen zu implementierenden HTTP-Client per HTTP-POST auf den SOS übertragen. Dieser quittiert die erfolgreiche Anfrage durch den Server-Statuscode 200. Bevor Beobachtungen in den SOS eingefügt werden können, erfolgt in einem Vorschritt das Registrieren des Telemetriesystems mittels eines *InsertSensor*-Dokuments und der gleichnamigen Operation. Erst wenn diese getätigt wurde, können Beobachtungen eingefügt werden. Diese vorgegebene Reihenfolge wird durch die Buchstaben A und B in der Abb. 5.5 gekennzeichnet. In den folgenden Kapiteln wird genauer auf die einzelnen Schritte im *Workflow* des Servers eingegangen.

5.5.1 Serverseitige persistente Datenspeicherung

Werden Daten vom Datenlogger übertragen, so werden diese in einer Datenbank auf dem Server gespeichert. Dazu sind zunächst eine Datenbank einzurichten sowie notwendige Tabellen zu entwerfen, in welchen die Daten gespeichert werden können. Die dazu notwendigen SQL-*CREATE-TABLE*-Anweisungen werden im Programmcode des Proxys als *Strings* hinterlegt. Der Übersichtlichkeit halber könnten diese in externen Dateien gespeichert werden, um SQL-Semantik und Programmcode voneinander zu trennen. Zum Zeitpunkt der Programmausführung würde diese dann in das Programm geladen. Basierend auf den eingerichteten Tabellen sind SQL-*INSERT*-Anweisungen zu formulieren. Eine Datenzeile der Telemetriedaten wird in eine solche Anweisung überführt. Dies kann stapelbasiert erfolgen, da dadurch, bei den zu erwartenden Datenmengen, die ineffiziente, sequentielle Ausführung einzelner *INSERT*-Anweisungen verhindert wird. Zusätzlich muss eine SQL-Anweisung erarbeitet werden, welche die gespeicherten Daten aus den Einzeltabellen zusammenführt und zeitlich miteinander verknüpft. Die verknüpften

Daten werden in einer Datenbanksicht abgebildet. Sie hat die gleiche Struktur wie die Tabelle der Positionsdaten, jedoch wird sie um weitere Spalten für die Fahrzeugdaten erweitert. Abschließend ist ein Datenbankmodul zu entwickeln, mit welchen die o. g. SQL-Anweisungen softwareseitig umgesetzt werden können.

5.5.2 Abruf der Daten und SOS-konforme Kodierung

Liegen Daten in der Datenbanksicht vor, werden diese mit dem o. g. Datenbankmodul abgerufen. Dies wird durch eine SQL-*SELECT*-Anweisung realisiert. Die Anweisung wird um *LIMIT*- und *OFFSET*-Klauseln erweitert, um lediglich Ausschnitte aus der Datenbanksicht abzufragen. Wurden die Daten extrahiert und auf den SOS übertragen, wird der Ausschnitt angepasst und beim nächsten Aufruf der Datenbanksicht berücksichtigt. Zunächst werden die extrahierten Daten in den Programmspeicher geladen. Dazu wird jede Zeile der Sicht auf Instanzvariablen eines Objekts abgebildet. Die Attribute dieses Objekts entsprechen jeweils einem Element der Zeile. Sie werden in XML-Dokumente, welche nach den Spezifikationen des SWE-Standards kodiert wurden, überführt. Dazu können entsprechend beschaffene, statische Dokumente herangezogen werden, deren Inhalt an den entsprechenden Stellen verändert wird. Bei diesen Dokumenten handelt es sich um *InsertObservation*-XML-Dokumente. Ein Objekt wird wiederum in ein Dokument überführt. Jedes Dokument besitzt mehrere *OM_Observation*-Elemente. Jeweils eines dieser Elemente stellt einen Prozessdatentyp dar. Dies bedeutet, dass beispielsweise bei einer Übertragung von Drehzahl- und Kraftstoffverbrauchsdaten zwei *OM_Observation*-Elemente für diese Datentypen in einem XML-Dokument vorliegen. Für eine solche Kodierung der Prozessdaten in XML sind daher Werkzeuge zur XML-Manipulation heranzuziehen.

5.5.3 SOS-Datenübermittlung durch HTTP-POST

Die Prozessdaten liegen nach der Kodierung durch den Proxy in einem Format vor, welches der SOS unterstützt. Diesem Schritt folgt nun die Übertragung der Daten zum SOS. Voraussetzung dafür ist die Anwendung der HTTP-Methode POST durch den Proxy. Wird diese implementiert, so lassen sich unter Verwendung der transaktionalen Schnittstelle die SOS-Operationen *InsertSensor* und *InsertObservation* ausführen. Erstere Operation wird, falls der Sensor noch nicht registriert wurde, vor dem Einfügen von Beobachtungsdaten ausgeführt. Bei dem in der Masterarbeit vorliegenden Fall geschieht dies durch ein statisches Dokument, welches an das Telemetriesystem angepasst wurde. Kommt es künftig zu einer Anwendung mehrerer Telemetriesysteme, so ist dieser erweiterte Fall bei der Ausgestaltung des *InsertSensor*-Dokuments zu berücksichtigen. Ob der Sensor registriert ist, wird mittels eines *GetCapabilitiesResponse*-Dokuments überprüft, das nach einer *GetCapabilities*-Anfrage vom SOS zurückgeliefert wird. Nach der erfolgreichen Registrierung erfolgt mit jedem neuen Eintrag in der Datenbanksicht die Ausführung der *InsertObservation*-Operation und damit die Speicherung der Daten im SOS.

6 Systementwicklung

6.1 Systemarchitektur

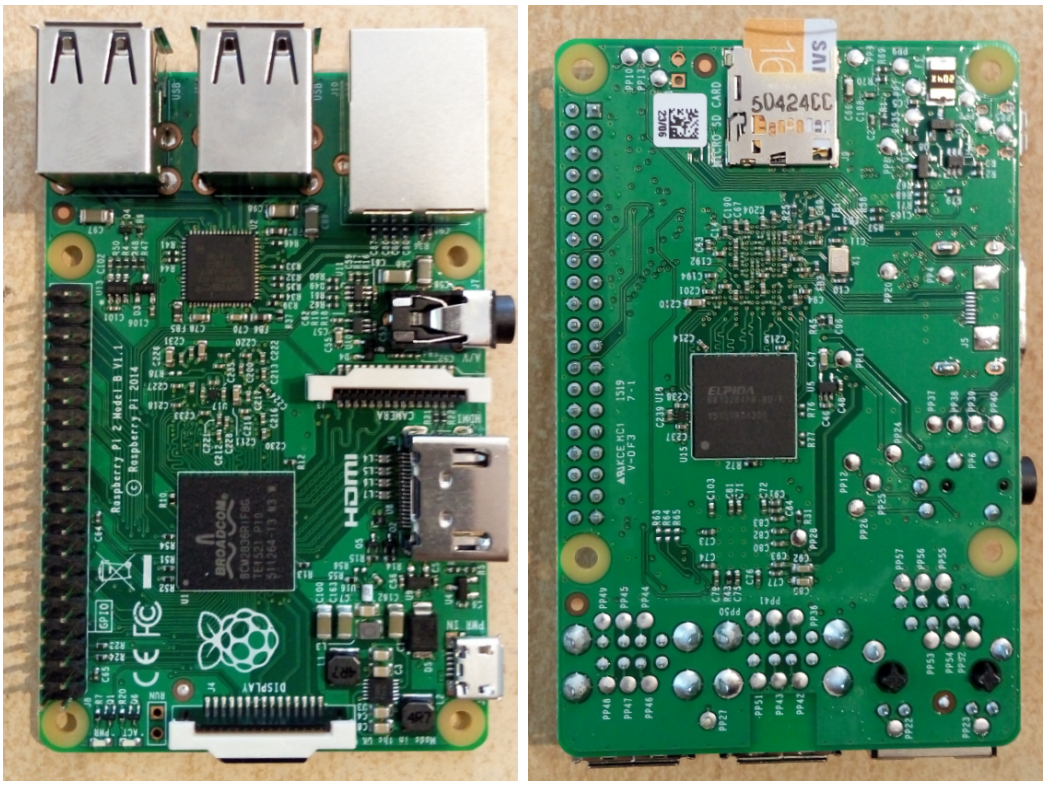
Die Umsetzung dieses Telemetriesystems soll mit kostengünstigen, leicht verfügbaren Komponenten erfolgen. Kernstück eines solchen Ansatzes ist ein Einplatinencomputer, an welchen unterschiedliche Sensoren angeschlossen werden können. Zusammenfassend ein Überblick über die verwendeten Komponenten und deren aufgerundete Preise:

1. Rechner
    - Raspberry Pi 2 **42 €**
    - SD-Karte **12 €**
2. GPS-GSM-Modul
    - SIM908-Modul **54 €**
    - Erweiterungsplatine **12 €**
    - GPS-Antenne **8 €**
    - GSM-Antenne **7 €**
3. CAN-Modul **40 €**
4. Netzteil **15 €**
5. USV **30 €**

Die Kosten des Datenloggers belaufen sich somit auf **220 €**. In den folgenden Abschnitten werden die einzelnen Komponenten näher untersucht.

6.1.1 Recheneinheit

Zentral ist die Rolle der Recheneinheit. Für das vorgesehene Telemetriesystem und die Platzierung im Fahrzeug sollte die Recheneinheit möglichst kompakt sein. Gleichzeitig sollte diese über ausreichende Rechenleistung verfügen, um die gewünschten Aufgaben der Datenerfassung und Übertragung wahrnehmen zu können. Diese Aufgaben übernimmt ein *Raspberry Pi* (Abb. 6.1).



(a) *Raspberry-Pi*-Vorderseite **(b)** *Raspberry-Pi*-Rückseite

Abbildung 6.1: *Raspberry Pi*

Raspberry Pi ist die Bezeichnung für eine Reihe von Computern, welche in etwa die Länge und Breite einer Kreditkarte aufweisen. Alle Komponenten eines Rechners sind auf einer einzigen Platine untergebracht. Aufgrund seiner kleinen Dimensionen und seinen geringen Anschaffungskosten, eignet sich der *Raspberry Pi* für verschiedene, insbesondere mobile, Anwedungsszenarien. Gleichzeitig unterscheidet sich die Handhabung des *Raspberry Pi* nicht von einem üblichen Computer. Ohne großen Aufwand kann, mit einem auf einer SD-Karte installierten Betriebssystem, der Raspberry Pi genutzt werden. Bei dem im Rahmen dieser Arbeit verwendeten *Raspberry Pi* handelt es sich dabei um die zweite Generation, nämlich den *Raspberry Pi 2 Model B* [Raspberry Pi Foundation, 2016]. Der Rechner ermöglicht das Verbinden verschiedener Komponenten durch die GPIO-Schnittstelle (*General Purpose Input Output*). Abb. 6.2 zeigt die Pins des GPIO auf dem Board nach der physikalischen Nummerierung. Die Nummerierung erfolgt dabei von links oben (oben bedeutet, der Pin, der sich am nächsten zur SD-Karte befindet, ist oben und trägt die Nummer 1) nach rechts unten. Die Pins dienen als Anschlussmöglichkeit und Datenaustauschschnittstelle für die vorgesehenen Sensoren. Durch den GPIO kann der *Raspberry Pi* um zusätzliche Komponenten erweitert werden. Beispielsweise können Sensoren nach dem HAT-Prinzip (*Hardware Attached on Top*) auf dem *Raspberry Pi* aufgesteckt werden.

Durch den GPIO weist der *Raspberry Pi* weitere Schnittstellen auf, die eine Anbindung von Komponenten ermöglichen. Diese sind die Schnittstellen I²C (*Inter-Integrated Circuit*, kurz gesprochen „I squared C“) und SPI (*Serial Peripheral Interface*), welche im Rahmen der Arbeit ebenfalls genutzt werden.

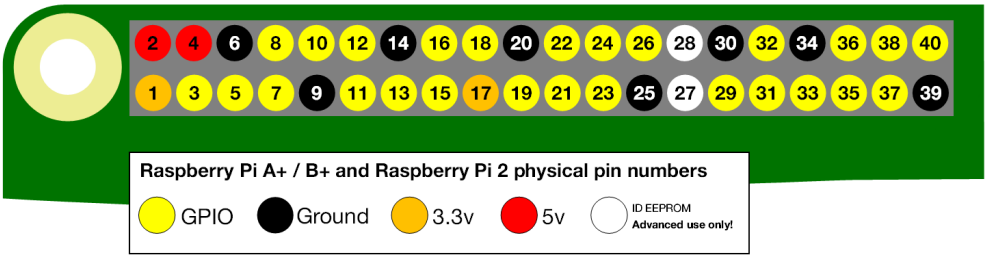
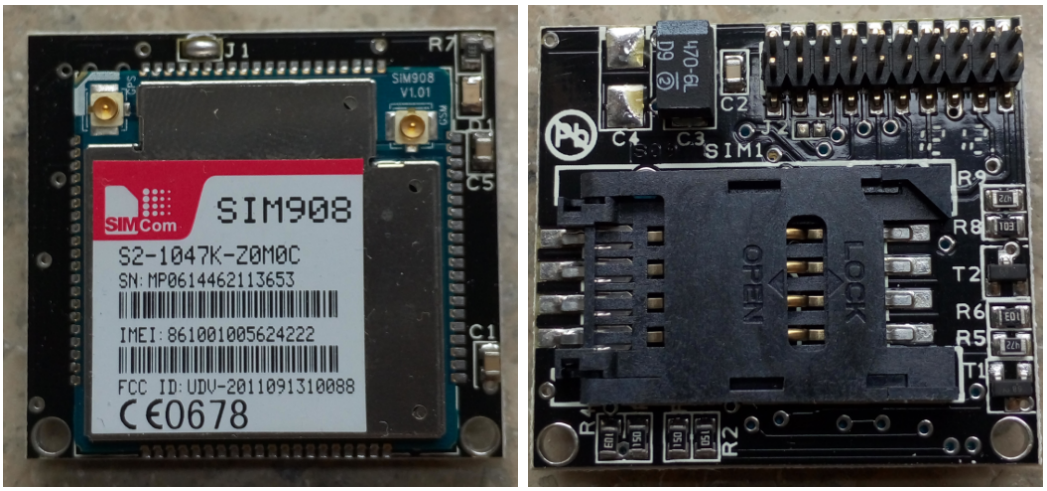


Abbildung 6.2: GPIO-Pin Belegung. Abbildung aus: https://www.raspberrypi.org/documentation/usage/gpio-plus-and-raspi2/README.md

6.1.2 GPS-GSM-Modul

Für die Positionsbestimmung und die Übertragung der Daten per GSM wird eine integrierte Lösung in Form eines GPS-GSM-Moduls verwendet: Das SIM908-Modul (Abb. 6.3) des Unternehmens SIMCOM ist ein GPS-GSM-Modul, welches sowohl Positionsbestimmung als auch die Datenübertragung per Mobilfunk ermöglicht [ElectroDragon, 2013].



(a) SIM908-Modul mit Antennenbuchsen **(b)** SIM908-Modul-Rückseite mit SIM-Kartenhalter

Abbildung 6.3: SIM908-Modul

Über eine Erweiterungsplatine, auf welcher das SIM908-Modul verbaut ist, wird die Verbindung zum Rechner hergestellt. Diese Erweiterungsplatine verbindet die Pins des Moduls mit den entsprechenden GPIO-Pins des Rechners. Die Kommunikation vom Rechner zum Modul wird dabei über die serielle Schnittstelle des GPIO realisiert. Konkret werden AT-Befehle an das Modul gesendet, um gewünschte Aktionen auszulösen. Das SIM908-Modul weist zwei U.FL-Stecker für GPS und GSM auf. Entsprechende Antennen werden mit U.FL zu SMA-Adapterkabeln (*SubMiniature version A*) an das Modul angeschlossen. Zur Herstellung von Mobilfunkverbindungen ist zudem eine handelsübliche SIM-Karte notwendig. Diese wird auf der Rückseite in den vorgesehenen Kartenhalter eingeführt.

6.1.3 CAN-Modul

Ein CAN-Modul dient zur Erfassung von Fahrzeugdaten mittels CAN-Schnittstelle. Das RSP-PiCAN2 des Unternehmens *SK Pang Electronics* [SK Pang Electronics Ltd., 2016a] stellt ein solches Modul dar und wird nach dem HAT-Prinzip mit dem Rechner verbunden (Abb. 6.4).

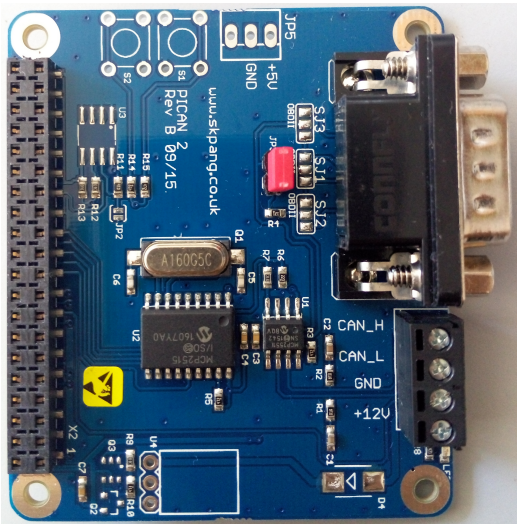


Abbildung 6.4: PiCAN2-Modul

Die Datenverbindung selbst wird durch das SPI des Rechners realisiert. Dieses Modul unterstützt den *CAN2.0B*-Standard und kann somit sowohl *Frames* mit 11-Bit-*identifier* als auch 29-Bit-*identifier* verarbeiten. Das Modul beinhaltet den CAN-Controller MCP2515 und CAN-Transceiver den MCP2551. Ein Widerstand von 120 Ohm ist auf dem Modul integriert und ist durch die nachträgliche Anbringung eines Jumpers verwendungsbereit. Neben einem Schraubterminal für *CAN High* und *CAN Low* kann eine 9-Pin-D-Sub-Anschlussbuchse (*D-Subminiature*) zur Verbindungsherstellung genutzt werden.

6.1.4 USV

Eine USV (Unterbrechungsfreie Stromversorgung) stellt den unterbrechungsfreien Betrieb des Systems sicher bzw. schützt, durch ordnungsgemäßes Herunterfahren des Rechners, bei einem Ausfall der Stromversorgung vor Datenverlust oder Beschädigung des Systems. Die Platine CW2 Pi UPS +, welche am GPIO des Rechers angeschlossen wird, übernimmt diese Aufgabe. Durch die I²C-Schnittstelle kommuniziert die USV mit dem *Raspberry Pi*. Durch den Anschluss der USV an das Stromnetz (in Abb. 6.5 mit *USB-Netzteil* beschriftet) wird das komplette System mit Strom versorgt.

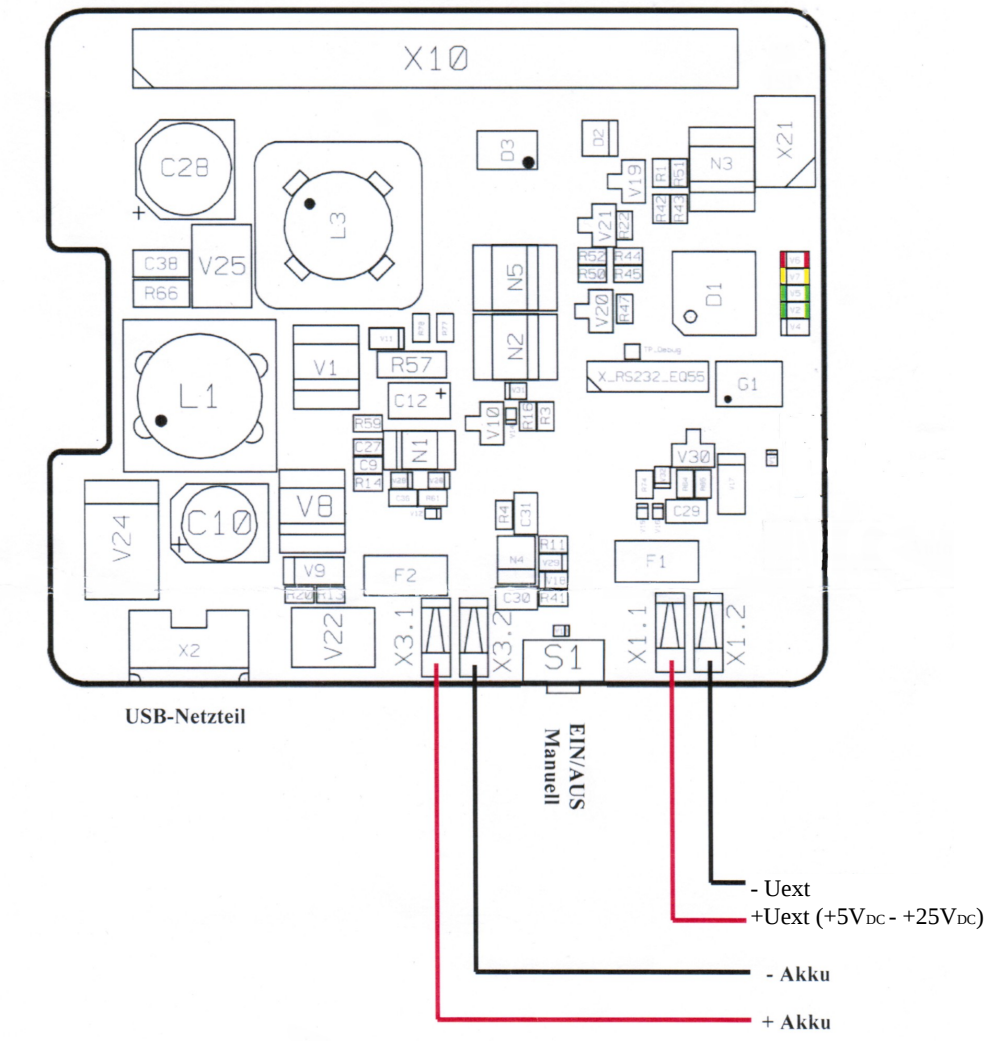
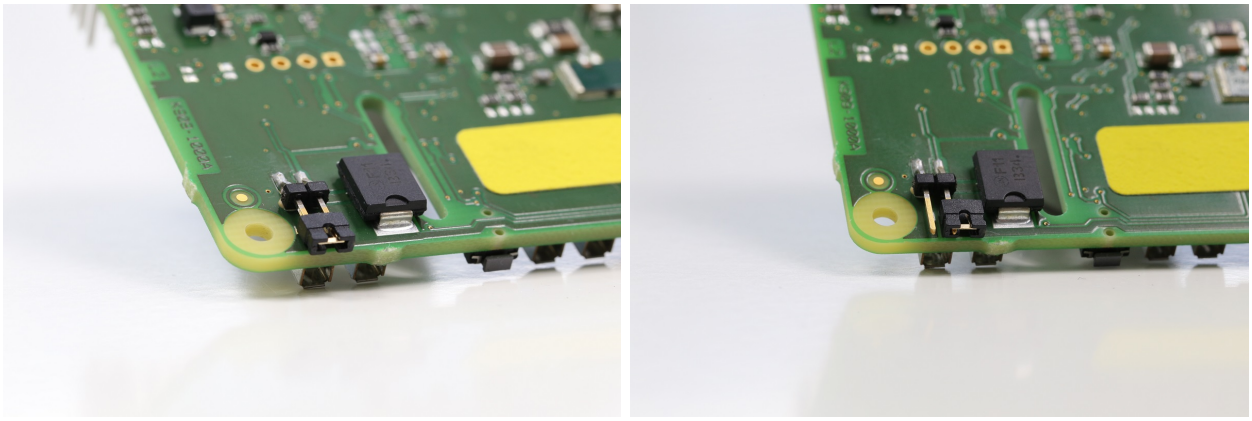


Abbildung 6.5: Anschlussbelegung der USV [CW2. GmbH & Co. KG, 2015]

Die USV besitzt zudem auf der einen Seite einen Anschluss, bei welchem entweder ein Lithium-Polymer-Akku bzw. ein Lithium-Ionen-Akku angeschlossen wird (in der Abb. 6.5 mit *+ Akku - Akku* benannt). Dieser Akku übernimmt die Notstromversorgung des Systems nach einem Betriebsausfall und ermöglicht das Herunterfahren des Systems. Auf der anderen Seite existiert eine weitere Anschlussmöglichkeit (*Uext*) für eine Stromquelle, welche zwischen 5 und 24 Volt liefert. Ein Batteriepack oder eine Solarzelle kann eine solche Stromquelle dar darstellen. Zusätzlich weist die USV einen Taster auf, mit welchem sich das System manuell ein- und ausschalten lässt. Durch einen *Jumper* kann einerseits eine rein manuelle Inbetriebnahme konfiguriert werden (Abb. 6.6).



(a) *Jumper*-Konfiguration für manuellen Start **(b)** *Jumper*-Konfiguration für automatischen Start

Abbildung 6.6: *Jumper*-Konfigurationen der USV aus [CW2. GmbH & Co. KG, 2015]

Dabei wird der Rechner lediglich gestartet, wenn der Taster betätigt wird. Hier wurde diese Konfiguration vorgenommen. Andererseits besteht die Möglichkeit eines automatischen Starts. Das System fährt in diesem Falle hoch, sobald eine Spannung an den primären Eingängen liegt. Die Funktionsweise des Tasters kann durch die *Jumper*-Konfiguration eingestellt werden.

6.2 Entwicklung des Datenloggers

Der Datenlogger umfasst den Rechner sowie die Komponenten der Datenerfassung, Positionsbestimmung und Datenübertragung. Wenn das USV-Modul mit Strom versorgt wird und der Power-Schalter betätigt wird, startet das System. Es lädt dabei die zur Datenerfassung notwendigen Programme hoch.

Zur Erfassung der Prozessdaten wird das CAN-Modul mit einem Softwarepaket namens *SocketCAN* [Volkswagen Research, 2016] angesprochen. Diese von Volkswagen Research initiierte Werkzeugsammlung stellt Mittel zur programmgesteuerten Kommunikation mit CAN-Modulen und Realisierung der Module als Netzwerkgeräte bereit. Ein in der Sprache Python geschriebenes

Programm (siehe dazu in Anhang C) nutzt die Funktionen von *SocketCAN* zur Speicherung der Prozessdaten und speichert diese in einer Textdatei ab. Vor dem Übertragungsschritt liegen die Daten somit als komma-separierte Werte in dieser Datei vor. Zur Steuerung des GPS-GSM-Moduls werden AT-Befehle mit der frei verfügbaren C++-Bibliothek *arduPi* von libelium [Libelium, 2016b] implementiert. Diese stellt Funktionen zur Verfügung, mit welchen die Steuerung der GPIO des *Raspberry Pi* ermöglicht wird und Eingaben über die serielle Schnittstelle getätigt werden können. Die ermittelten Positionen sowie UTC-Zeitstempel werden ebenso lokal in einer Textdatei gespeichert. Konkret liegt nach einem abgeschlossenen Erfassungsschritt eine Zeile von Daten mit diesen Informationen vor. Diese Datenzeilen werden ebenfalls auf der SD-Karte des Datenloggers in einer Textdatei zwischengespeichert.

Vor der Übertragung werden die Datenzeilen aus den unterschiedlichen Textdateien gelesen. Lag ein Verbindungsabbruch vor, so entfällt der Übertragungsschritt. Der Datenlogger erfasst dann weiterhin Daten und speichert diese in den Textdateien. Im Anschluss wird ein erneuter Versuch der Verbindungsherstellung unternommen. Konnte eine Verbindung hergestellt werden, so werden alle bis zu diesem Zeitpunkt nicht übertragenen Datenzeilen dem Server übermittelt. Die Übertragung selbst erfolgt über eine TCP-Verbindung. Bei der Einrichtung und Herstellung dieser Verbindung kommen ebenfalls implementierte AT-Befehle zum Einsatz. Da das SIM908-Modul eine Datenübertragung über das GSM-Netz tätigt, fordert dieses eine IP-Adresse vom Telekommunikationsdienstleister an. Ist diese dem Gerät zugewiesen, so stellt das Modul eine TCP-Verbindung über Sockets zum Server her.

Der Server selbst ist ein vorkonfigurierter Rechner. Dieser wurde unter einer festen Adresse[1] eingerichtet. Die Datenübertragung über HTTP-POST ist theoretisch mit dem SIM908-Modul möglich, die Bedienungsanleitung verweist auf diese Funktionalität [ElectroDragon, 2015]. Die eigene Erprobung der HTTP-POST-AT-Befehle, bei der versucht wurde, *InsertObservation*-XML-Dokumente mit Prozessdaten zu übertragen, brachte jedoch unbefriedigende Ergebnisse hervor. Dies ist wahrscheinlich auf eine mangelnde Umsetzung der HTTP-Unterstützung des Moduls und auf die Größe der XML-Dokumente zurückzuführen. Durch die Verbindung über TCP ist eine Datenübertragung von Rohdaten möglich. Zudem kann serverseitig eine größere Bandbreite zur Übertragung der SWE-kodierten Daten an den SOS genutzt werden, als es die GPRS-Verbindung des GSM-Moduls erlaubt. Es bestand daher in der Systemkonzeption das Interesse, diesen Ansatz weiterzuverfolgen und die Übertragung der Daten an den SOS serverseitig zu regeln.

6.2.1 Konfiguration der Recheneinheit

Zur Inbetriebnahme eines *Raspberry Pi* muss eine SD-Karte mit enthaltenem Betriebssystem in den SD-Slot eingeführt werden. Die Installation des Linux-Betriebssystems *Raspbian* übernimmt die Installationsroutine NOOBS (NOOBS). Diese wird als zip-Datei auf der Website der *Raspberry Pi Foundation* zur Verfügung gestellt. Zur Installation muss diese Datei auf eine formatierte SD-Karte kopiert werden. Sobald diese SD-Karte in den entsprechenden Slot

[1] http://gpslog.ddns.net:8080

am *Raspberry Pi* eingeführt wurde und der Rechner gestartet wird, startet das Installationsprogramm. Mittels eines angeschlossenen Bildschirms folgt der Nutzer den Anweisungen des Programms. Nach Befolgung und Ausführung aller notwendigen Installationsschritte ist das Betriebssystem installiert.

Standardmäßig ist die serielle Schnittstelle des *Raspberry Pi* durch eine Login-Konsole belegt. Damit die Kommunikation zwischen *Raspberry Pi* und dem GPS-GSM-Modul stattfinden kann, muss die serielle Login-Konsole abgeschaltet werden: Dazu muss die Datei `/boot/cmdline.txt` verändert werden. Das Entfernen der Einträge

```
console=ttyAMA0,115200 kgdboc=ttyAMA0,115200
```

schaltet die Login-Konsole ab. Danach kann die serielle Schnittstelle zur Kommunikation mit dem GPS-GSM-Modul verwendet werden.

Bash-Skripte (*Bourne-again shell*) starten die nach dem Systemstart benötigten Programme des Datenlogers. Diese befinden sich im Pfad `/etc/init.d`. Es werden dabei zusätzliche Start-Stopp-Skripte des *Init*-Systems genutzt, welche die Bash-Skripte ansteuern. Dazu müssen für die Bash-Skripte schreibende Zugriffsrechte vorliegen, damit diese von den Skripten des *Init*-Systems beim Hochfahren des Systems ausgeführt werden können.

Das CAN-Modul ist über die SPI-Schnittstelle mit dem *Raspberry Pi* verbunden. Zur Nutzung des Moduls muss diese Schnittstelle zuerst aktiviert werden. Dazu müssen in der Datei `/boot/config.txt` die Einträge

```
dtparam=spi=on
dtoverlay=mcp2515-can0-overlay,oscillator=16000000,interrupt=25
dtoverlay=spi-bcm2835-overlay
```

ergänzt bzw. auskommentiert werden. Bei Nutzung des *Raspberry Pi Kernel Jessie* ab der Version 4.4 ist der mittlere Eintrag zu ändern [SK Pang Electronics Ltd., 2016b]:

```
dtoverlay=mcp2515-can0,oscillator=16000000,interrupt=25
```

Nach einem Neustart des Systems sollte das CAN-Modul mittels des Befehls *ifconfig* bei den Netzwerkgeräten als `can0` aufgeführt sein. Um die Kommunikation mit anderen Teilnehmern eines CAN-Netzwerkes vorzubereiten, muss die Datenrate des CAN-Moduls festgelegt werden: `sudo /sbin/ip link set can0 up type can bitrate <BITRATE>`. Der Wert in eckigen Klammern ist durch eine Datenrate in Bits pro Sekunden (beispielsweise 250000 für 250 kb/s) zu ersetzen. Die Festlegung der Datenrate kann bei jedem Start automatisch durch das System vorgenommen werden. Dazu ist es notwendig, folgenden Eintrag

```
auto can0
    iface can0 inet manual
    pre-up /sbin/ip link set $IFACE type can bitrate 250000
    up /sbin/ifconfig $IFACE up
```

in der *interfaces*-Datei im Pfad `/etc/network/interfaces` einzufügen.

Das USV-Modul kommuniziert über die I²C-Schnittstelle mit dem *Raspberry Pi*. Diese ist standardmäßig deaktiviert und muss daher manuell eingeschaltet werden. Zunächst müssen dafür in der Datei `/etc/modules` die I²C-Kernelmodule eingetragen werden:

```
i2c-bcm2708
i2c-dev
```

Damit werden die Kernelmodule beim Starten des Systems geladen. Zudem sollte überprüft werden, ob in der Konfigurationsdatei des *Raspberry Pi* die entsprechenden schnittstellenspezifischen Parameter aktiviert sind. Dazu müssen in der Datei `/boot/config.txt` die Einträge

```
dtparam=i2c1=on
dtparam=i2c_arm=on
```

ergänzt bzw. auskommentiert werden. Schließlich sollten Einträge in der *module blacklist* `/etc/modprobe.d/raspi-blacklist.conf` überprüft werden. Diese können bei manchen *Raspbian*-Versionen standardmäßig eingetragen sein und somit zu Problemen führen. Deshalb muss der Eintrag

```
blacklist i2c-bcm2708
```

in der *module blacklist* für die korrekte Funktionsweise des I²C kommentiert bzw. entfernt werden. Nach einem Neustart des Systems kann die I²C-Schnittstelle mit dem I²C-Toolkit überprüft werden:

```
sudo apt-get install i2c-tools
sudo i2cdetect -y 1
```

i2cdetect sollte hier die Adresse 18 als belegt ausweisen. Ist dies der Fall so ist das USV-Modul angeschlossen. Zur Inbetriebnahme des Moduls bedarf es des auf der Herstellerwebsite befindlichen *debian package PiUPS monitor*. Der *PiUPS monitor* überwacht und loggt den Zustand des USV-Moduls und führt bei Stromausfall die notwendigen Schritte zum sicheren Herunterfahren des Telemetriesystems durch. Konfiguriert wird das USV-Modul durch eine von *PiUPS monitor* zu diesem Zweck eingerichtete Konfigurationsdatei. Diese befindet sich nach der Installation im Pfad:

```
/etc/piupsmon/piupsmon.conf
```

In dieser Konfigurationsdatei lassen sich Einstellungen vornehmen, wie bei einem Stromausfall vorzugehen ist. Darin lässt sich die restliche Betriebszeit des Systems sowie des USV-Moduls konfigurieren. Zudem besteht die Möglichkeit, ein benutzerdefiniertes Skript anzugeben, welches im Falle der kontrollierten Abschaltung ausgeführt wird. Weiterhin ist es möglich die Semantik der aufgezeichneten Nachrichten des USV-Moduls festzulegen. Schließlich kann die Beschreibung der Lognachrichten (numerisch oder mittels zusätzlicher textueller Beschreibung) eingestellt werden. Die aufgezeichneten Nachrichten werden in der Datei `/var/log/piupsmon.log` gespeichert.

6.2.2 Prozessdatenerfassung und Dekodierung

Sind die in Kapitel 6.2.1 genannten Voraussetzungen für das CAN-Modul erfüllt, können Fahrzeugdaten vom Datenlogger erfasst werden. Das Python-Programm *readDecodeCAN.py* (siehe Anhang C.1) implementiert diese Datenerfassung, dekodiert die CAN-Nachrichten und speichert die dekodierten Werte in einer Textdatei. Auf dem Datenlogger befindet sich das Programm im Pfad:

```
/rbpi/readDecodeCAN.py
```

Gestartet wird das Programm durch das *init*-System *systemd*. Damit wird mit jeder Inbetriebnahme das Programm automatisch ausgeführt. Hierzu wurde eine *Unit*-Datei angelegt, welche unter dem Pfad

```
/lib/systemd/system/myscript.service
```

zu finden ist. Eine solche *Unit*-Datei erlaubt die automatische Ausführung von Programmen zum Systemstart im Betriebssystem *Raspbian* des Datenloggers. Zusätzlich müssen schreibende Zugriffsrechte (mit dem Befehl *chmod*) vergeben werden und die Befehle

```
sudo systemctl daemon-reload
sudo systemctl enable myscript.service
```

im Terminal ausgeführt werden. Für die Dekodierung wurden CAN-*identifier*, *Low Byte*, *High Byte* und Skalierungsfaktoren für zwei Fahrzeugdatentypen im Programmcode hinterlegt (siehe Tabelle 6.1).

Tabelle 6.1: Übersicht über erfasste CAN-Daten und deren *identifier*

Datentyp	Einheit	Bezeichner
Motordrehzahl (RPM, *Revolutions per minute*)	1/min	0CF004F0
Kraftstoffverbrauch	l/h	0CFEF2F0

Diese Dekodierungsinformationen wurden von Prof. Noack von der Hochschule Weihenstephan-Triesdorf zur Verfügung gestellt (E-Mail-Korrespondenz zwischen Betreuer Thomas Machl und Prof. Noack vom 19.07.2016). Der *identifier* ist eine achtstellige, hexadezimale Kennung und ist einer Fahrzeugdatenkategorie zugeordnet. Die Informationen *Low Byte* sowie *High Byte* stellen die Indices der Fahrzeugdaten im *Data Field* der CAN-Nachricht dar. Skalierungsfaktoren sind an die dekodierten *Bytes* zur Korrektur anzubringen. Nach ordnungsgemäßer Konfiguration des CAN-Moduls können CAN-Daten mit der Funktion *candump* aus der Werkzeugsammlung *SocketCAN* abgegriffen werden. Eine Ausgabe von *candump* enthält verschiedene Datenzeilen und weist eine Struktur auf, welche in Listing 6.1 dargestellt wird.

```
...
(1468514638.833893) can0 0CFE48F0 [8] 00 00 00 00 00 00 01 D5
(1468514638.834464) can0 0CF004F0 [8] FF FF FF 38 23 FF FF FF
(1468514638.835041) can0 0CFEF2F0 [8] 4E 00 FF FF FF FF FF FF
(1468514638.835608) can0 0CFE45F0 [8] F2 CF 80 44 7F FF FF FF
...
```

Listing 6.1: Ausschnitt aus einer *candump*-Bildschirmausgabe

Der oben angegebene Ausschnitt der Ausgabe von *candump* wurde mit den Argumenten *-t a* erzeugt. Dadurch werden neben den CAN-Botschaften absolute (Argumentwert *a*) Zeitstempel (Befehlsargument *-t*), basierend auf der lokalen Zeit des Datenloggers im Linux-Terminal ausgegeben. Das erste Element einer solchen Zeile repräsentiert somit die Anzahl der Sekunden, die seit der Referenzepoche, dem 01.01.1970, 00:00:00 Uhr, vergangen sind. Die erste dargestellte Zeile trägt daher den Zeitstempel: Donnerstag, der 14.07.2016, 18:43:58 Uhr. Das zweite Element stellt die von *SocketCAN* für das CAN-Modul vergebene Bezeichnung dar. Daran knüpft die achtstellige, hexadezimale Kennung (CAN-*identifier*) der Nachricht an. Mittels der von Prof. Noack gegebenen *identifier* können die oben dargestellte zweite und dritte Zeile identifiziert werden. Das dritte Element in einer *candump*-Zeile stellt den DLC dar. Dieser gibt wieder, wieviele *Bytes* im darauffolgenden *Data Field* vorhanden sind. Nach den Informationen von Prof. Noack sind für die Kodierung der tatsächlich vom CAN-Netzwerk ausgegebenen Drehzahlwerte die Doppelziffern an vierter (*Low Byte 4*) und fünfter Stelle (*High Byte 5*) im *Data Field* relevant. Analog werden für die Kodierung der Kraftstoffwerte die Doppelziffern an der ersten (*Low Byte 1*) und zweiten Stelle *High Byte 2* verwendet. Die Ziffern 38 (hexadezimal 38) und 23 (hexadezimal 2300) der zweiten Zeile im oben angegebenen Beispiel stellen die dezimalen Werte 56 und 8960 dar. Diese werden addiert und durch den Skalierungsfaktor von 8 dividiert, was eine Drehzahl von 1127 pro Minute ergibt. Analog lässt sich aus den Angaben des *Data Field* der dritten Zeile ein Kraftstoffwert von 3,9 (4E, dezimal 78, dividiert durch den Skalierungsfaktor 20) Liter pro Stunde ableiten.

Das Python-Programm *readDecodeCAN.py* führt diese Prozedur aus und ruft *candump* in einem Unterprozess aus. Dazu wird der für den Aufruf von *candump* notwendige *Shell*-Befehl als *String* dem Konstruktor der Klasse *Popen* des Python-Moduls *subprocess* übergeben. Die so erzeugte *Popen*-Instanz leitet die Ausgabe des *Shell*-Befehls in eine Instanzvariable um, welche mittels der Methode *readline* Zeile für Zeile ausgelesen werden kann. Auf eine solche Zeile werden Funktionen zur Zeichenmanipulation angewendet. Mittels eines Python-Idioms, der sog. *List Comprehension*, wird eine Liste erzeugt, welche eine CAN-Datenbotschaft repräsentiert. Für die Dekodierung notwendig sind der CAN-*identifier* sowie das Datenfeld des CAN-*Frames*. Durch die Python-Listentraversierung, das sog. *Slicing*, wird der durch *candump* zugeordnete Zeitstempel, der CAN-*identifier* sowie das CAN-*Data-Field* des CAN-*Frames* der Liste entnommen. Das *Data Field* sowie die Informationen *Low Byte*, *High Byte* und Skalierungsfaktor werden der Funktion *decodeCanMessage()* übergeben. In dieser Funktion wurde der oben erläuterte Auswerteprozess implementiert. Rückgabewert ist der tatsächliche Fahrzeugdatenwert. Mittels der Funktion *buildSentence()* werden die Informationen lokaler Zeitstempel, textuelle Beschreibung des Fahrzeugdatentyps (R für Drehzahl, F für Kraftstoffverbrauch) sowie Fahrzeugdatenwert

in einer komma-separierten Zeile aggregiert. Hinzu kommt die Zählervariable des Systems, welche an erste Stelle gesetzt wird und mittels eines Unterstrichs vom Zeitstempel getrennt wird. Eine solche Zeile wird in einer Textdatei (gewählte Bezeichnung *CANData.txt*) für die spätere Datenübertragung durch das GPS-GSM-Modul abgespeichert.

Letztendlich weisen die zu übertragenden Daten in der Textdatei folgende Struktur auf:

```
...
6_1469700933.807518,R,1129
6_1469700933.808641,F,4.2
6_1469700933.905687,R,1128
6_1469700933.906809,F,4.0
...
```

Listing 6.2: Dekodierter, restrukturierter Ausschnitt der CAN-Daten aus *CANData.txt*

Die automatische Inkrementierung der Zählervariablen wird ebenso mittels eines Python-Programms und einer *Unit*-Datei des *init*-Systems *systemd* realisiert. Das Python-Programm befindet sich im Pfad

`/rbpi/startCounter.py`

und die *Unit*-Datei ist im Pfad

`/lib/systemd/system/myscript.service`

abgelegt.

6.2.3 Programmierung des GPS-GSM-Moduls

Der Start und der laufende Betrieb des GPS-GSM-Moduls wird durch verschiedene modifizierte C++-Programme (siehe dazu in Anhang A.1, A.2 und A.3) und Bash-Skripte realisiert. Die C++-Programme binden die *arduPi*-Bibliothek ein, um eine Kommunikation mit der seriellen Schnittstelle zu ermöglichen, vorausgesetzt die serielle Schnittstelle steht frei (siehe dazu Kapitel 6.2.1). Die zur Kommunikation mit dem Modul erforderlichen AT-Befehle werden direkt durch Eingabe entsprechender Zeichenketten in die serielle Schnittstelle übergeben. Alle im Rahmen der Arbeit verwendeten AT-Befehle werden aus der Bedienungsanleitung des GPS-GSM-Moduls entnommen (siehe dazu [ElectroDragon, 2015]). Dies erfolgt durch die `print`-Methode der Klasse *Serial* aus der *arduPi*-Bibliothek. AT-Befehle sind in diesen Programmen implementiert. Jedes Programm ist dabei jeweils für eine Aufgabe zuständig:

1. *closeGSM* – Schaltet das SIM908-Modul ab
2. *gsmIgnition* – Startet das SIM908-Modul
3. *gpsTcp* – Führt die Positionierung sowie Übertragung der georeferenzierten Prozessdaten über GSM aus

Die Kompilierung der Programme wird durch die *Compiler-Suite* GCC (*GNU Compiler Collection*) und unter Einbeziehung der *arduPi*-Bibliothek durchgeführt (siehe Listing 6.3).

```
g++ -lrt -lpthread /rbpi/GSMOff.cpp /rbpi/arduPi.o -o /rbpi/OffGSM
g++ -lrt -lpthread /rbpi/GSMIgnition.cpp /rbpi/arduPi.o -o /rbpi/OnGSM
g++ -lrt -lpthread -std=c++11 /rbpi/gpsTcp.cpp /rbpi/arduPi.o -o /rbpi/gpsTcp
```

Listing 6.3: Terminalbefehle zur Kompilierung der GPS-GSM-Programme

Wird das Telemetriesystem eingeschaltet, fährt das Betriebssystem hoch und führt die genannten Programme aus. Dabei werden die ersten zwei Programme einmalig ausgeführt. Diese stammen vom Vertreiber des Moduls und der Erweiterungsplatine (siehe dazu [Landoni, 2013]). Das Programm *gpsTcp* wird initialisiert und wird ausgeführt, solange das Telemetriesystem eingeschaltet ist. Das Programm basiert auf einer Vorlage des Unternehmens Libelium [Libelium, 2016a]. Alle zur Initialisierung von GPS und GSM benötigten Funktionen werden dabei in der Funktion *setup()* zusammengefasst. Dies beinhaltet auch die Initialisierung der in Kapitel 5.4.2 genannten Textdateien. Funktionen, welche während des laufenden Betriebes immer wieder aufgerufen werden müssen, beispielsweise die Abfrage nach der aktuellen Position, werden in der Funktion *loop()* zusammengefasst. Somit ergibt sich folgender Ablauf des Programms *gpsTcp* (Listing 6.4).

```
...
int main (){
    setup();
    while(1){
        loop();
    }
    return (0);
}
```

Listing 6.4: *main()*-Funktion im Programm *gpsTcp*

Nach Ausführung der Funktion *setup()* wird in einer ununterbrochenen Schleife die Funktion *loop()* ausgeführt. In der Funktion *setup()* werden, neben der Initialisierung der zur Speicherung der gelesenen Daten notwendigen Textdateien und der Anmeldung des GSM-Moduls beim Mobilfunkanbieter, die Funktionen *buildTcp()* sowie *start_GPS()* ausgeführt. Erstere führt die Verbindungsherstellung zum Server über einen TCP-Socket durch, letztere aktiviert das GPS-Modul. Die Funktion *loop()* beinhaltet die Ausführung von vier Funktionen:

1. *reset_GPS()* – Startet GPS-Modul erneut, falls keine Positionslösung gefunden werden kann
2. *get_GPS()* – Erfragt vom GPS-Modul einen NMEA-Datensatz und weist diesen einer globalen Variable zu
3. *writeGPS2file()* – Schreibt den Wert der globalen Variable in die Textdatei *gpslog.txt*
4. *sendData()* – Liest Positions- und CAN-Daten aus den Textdateien *gpslog.txt* und *CAN-Data.txt*

Die für die Ausführung dieser Funktionen notwendige Eingabe von AT-Befehlen erfolgt durch die Methode *print()* der Klasse *Serial*. Eingebettet ist diese Methode wiederum in der Funktion *sendATcommand()* bzw. in ihrer Erweiterung *sendATcommand2()*. Die Funktion *sendATcommand()* dient dazu, die nach Eingabe eines AT-Befehls vom GSM-Modul zurückgelieferte Antwort zu

überprüfen und eine *Timeout*-Funktionalität zu implementieren. Sie nimmt als erstes Argument den AT-Befehl als Zeichenkette (`ATcommand`) an. Weiterhin werden eine (bei *sendATcommand()*) oder mehrere (bei *sendATcommand2()*) Prüfzeichenketten übergeben, welche in einer *if*-Klausel mit der zurückgelieferten Antwort verglichen werden. Fällt der Vergleich der Antwort mit der Prüfzeichenkette negativ aus, so werden Werte ungleich 0 zurückgeliefert. Das letzte zu übergebene Argument der Funktion *sendATcommand()* ist das Argument *timeout*. Dieses stellt den Zeitraum in Millisekunden dar, nach dessen Überschreitung die Funktion *sendATcommand()* die Kommunikation mit der seriellen Schnittstelle abbricht. Basierend auf dieser Funktion wird die Kommunikation mit dem Modul getätigt. In den folgenden Kapiteln wird detailliert auf die implementierten Programme zur Kommunikation mit dem GPS-GSM-Modul eingegangen.

6.2.4 Inbetriebnahme des GPS-GSM-Moduls

Im Folgenden wird auf die Inbetriebnahme des Moduls anhand des C++-Programms *gsmIgnition* (siehe dazu A.1) eingegangen. Zu Beginn der Inbetriebnahme des Systems muss die Baudrate, mit welcher der Rechner mit dem Modul kommuniziert, festgelegt werden. Die Baudrate besitzt die Einheit bps (*bits per second*) und stellt die reziproke Zeit dar, welche benötigt wird, um ein Bit zu senden. Festgesetzt wird die Baudrate auf 115200 bps durch die *begin*-Methode der *Serial*-Klasse. Das Herunterfahren und Starten des Moduls basiert auf dem programmgesteuerten Heraufsetzen und Absetzen der an einem Pin befindlichen Spannung. Dazu werden in einem vorangehenden Schritt die GPIO-Pins Nummer 8 (mit der Bezeichnung *transmit data pin* oder TXD) und 9 (Masse) durch die Methode *pinMode* auf einen Zustand niedriger Impedanz (durch das Argument *OUTPUT*) gesetzt. Dies ist notwendig, um ausreichend Strom für das Modul bereitzustellen. Es wird anschließend durch Senden der Zeichenkette *AT* an die serielle Schnittstelle überprüft, ob das Modul auf Eingaben von außen antwortet. Die Überprüfung geschieht mithilfe der Funktion *available()*, diese liest den *Buffer* der seriellen Schnittstelle aus und liefert die Anzahl der im *Buffer* enthaltenen *Bytes*. Liefert diese keine Antwort, d. h. die Anzahl der zurückgegebenen *Bytes* beträgt 0, wird die Funktion *resetModule()* aufgerufen. Diese legt an den Pin 9 des *Raspberry Pi* eine Spannung für 500 ms an und setzt diese wieder herab. Dadurch wird das Modul in den Ausgangszustand zurückgesetzt. Anschließend wird erneut der Inhalt der seriellen Schnittstelle durch eine AT-Abfrage überprüft. Falls wieder keine Antwort vom Modul zurückgeliefert wird, wird das Modul durch die Funktion *switchModuleOn()* aktiviert. Dabei wird eine Spannung an den Pin 8 für 2 Sekunden angelegt und abschließend wieder herabgesetzt. Die verwendete Prozedur unterscheidet sich von der Abschaltung des Systems mittels des Programms *closeGSM* (siehe A.2) lediglich in einer Fallunterscheidung, welche den ausgeschalteten Zustand des Systems berücksichtigt. Das Modul reagiert danach durch Aufleuchten einer am Modul befindlichen Diode. Diese leuchtet für etwa 2 Sekunden dauerhaft, anschließend blinkt sie etwa mit einer Frequenz von 2 Hz. In diesem Zustand verbleibt die Diode bis eine Verbindung mit dem GSM-Netzwerk aufgebaut wurde. Liegt eine solche Verbindung vor, wird die Frequenz, mit der die Diode blinkt, auf etwas weniger als 1 Hz reduziert. Damit ist das GPS-GSM-Modul betriebsbereit.

6.2.5 Speicherung der Position durch das GPS-GSM-Modul

Zur Initialisierung der GPS-Funktionalität des Moduls wird die Funktion *start_GPS()* im Programm *gpsTcp* verwendet. Diese führt die zur Bestimmung einer Positionslösung notwendigen AT-Befehle aus. Mittels der Funktion *sendATcommand()* wird zunächst der AT-Befehl `AT+CGPSPWR=0` ausgeführt. Dieser schaltet das GPS-Modul aus, falls es bereits aktiviert ist, um den Ausgangszustand herzustellen. Der Befehl `AT+CGPSPWR=1` startet das Modul. Im Anschluss erfolgt der Befehl `AT+GPSRST=0`. Dies veranlasst die Suche nach Satelliten im sog. „kalten Modus“. Bei freier Sicht zum Himmel benötigt die Bestimmung einer Positionslösung ungefähr 40 Sekunden. Dabei werden keine Vorinformationen, wie z. B. Almanachdaten, genutzt, welche aus einer früheren Positionslösung stammen. Im Anschluss wird in einer Schleife mittels des AT-Befehls `AT+CGPSSTATUS?` alle fünf Sekunden der Status dieser Positionsbestimmung überprüft. Gibt das Modul die Antwort `2D Fix` bzw. `3D Fix` zurück, so ist eine Positionslösung bestimmt. Wenn diese Positionslösung im Laufe des Systembetriebs verloren geht (beispielsweise die Sicht zu Satelliten wird verhindert), wird das GPS-Modul erneut gestartet und eine Positionslösung im „warmen Modus“ ermittelt. Da bereits in der Vergangenheit eine Positionslösung bestimmt wurde, können Vorinformationen genutzt werden. Die Bestimmung einer Positionslösung im warmen Modus benötigt ungefähr fünf Sekunden. Die Funktion *get_GPS()* führt den zur Abfrage der aktuellen Position benötigten AT-Befehl aus. Solange eine *Fix*-Lösung vorliegt, wird mit dem Befehl `AT+CGPSINF=0` eine NMEA-Zeichenkette mit der aktuellen Position abgefragt. Eine solche komma-separierte Zeichenkette weist folgendes Format auf:

```
mode,longitude,latitude,altitude,UTC,TTFF,num,speed,course
```

Listing 6.5: Struktur der von AT+CGPSINF=0 gelieferten NMEA-Zeichenkette

Darin bedeutet

- `mode` – Typ der Nachricht, mögliche Werte 0 oder Zweierpotenz bis zum Exponent 7
- `longitude` – Geographische Länge in ganzen Grad und dezimalen Minuten
- `latitude` – Geographische Breite in ganzen Grad und dezimalen Minuten
- `altitude` – Höhe in Metern, bezogen auf WGS84-Ellipsoid
- `UTC` – *Universal Time Coordinated* mit aneinandergereihten Zeichen für Datum und Uhrzeit
- `TTFF` – *Time To First Fix*, Zeit in Sekunden bis zur Bestimmung der Positionslösung
- `speed` – Geschwindigkeit in Metern pro Sekunde
- `course` – Kurs in Grad in Bezug auf geographisch Nord

Diese zurückgegebene Zeichenkette wird in der im Programmquelltext *gpsTcp.cpp* global initialisierten Variable *Basic_str* gespeichert. Eine beispielhafte Zeile lässt sich aus dem Listing 6.6 entnehmen.

```
...
0,1134.076920,4808.975868,614.856750,20160728210207.000,66,9,0.000000,0.000000
0,1134.078142,4808.971464,614.989380,20160728210220.000,66,10,0.000000,0.000000
0,1134.059431,4808.991553,633.804443,20160728210233.000,66,8,0.000000,0.000000
0,1134.051550,4809.002311,642.888184,20160728210246.000,66,8,0.000000,0.000000
0,1134.044958,4809.010132,649.596069,20160728210259.000,66,8,0.000000,0.000000
...
```

Listing 6.6: Ausschnitt der Textdatei *gpslog.txt*, welche die NMEA-Datensätze enthält

Der Wert der Variablen *Basic_str* wird mittels der Funktion *write2GPSFile()* in die Textdatei *gpslog.txt* geschrieben. Zusätzlich wird in einer anderen Textdatei (mit der Bezeichnung *writeLinePointerFile.txt*) anhand einer Zählervariable festgehalten, wieviele Datensätze bereits geschrieben wurden. Dies dient, dazu den Lesezähler zu begrenzen und die Datei *gpslog.txt* nicht über die vorhandenen Datenzeilen hinaus zu lesen. Zu bemerken ist die Tatsache, dass diese Zeichenkette keinen gebräuchlichen NMEA-Datensatz darstellt. Vielmehr ist diese eine Kombination von Datenwerten aus verschiedenen NMEA-Datensätzen. Gebräuchliche NMEA-Datenzeilen können durch die Festlegung des Wertes des Befehls `AT+CGPSINF` (in diesem Fall beträgt der Wert 0) abgefragt werden.

6.2.6 Verbindungsherstellung mittels Sockets

Die Funktion *buildTcp()* leitet eine Verbindungsherstellung über TCP ein. Für die Verbindungsherstellung notwendig sind Informationen über den Mobilfunkbetreiber sowie Daten über den Server, an den die gespeicherten Prozessdaten gesendet werden sollen. Mobilfunkdaten beinhalten Angaben zu Zugangspunkt, Benutzernamen und Passworts. Diese Daten unterscheiden sich innerhalb verschiedener Mobilfunkbetreiber und sind im Vorfeld einzuholen. Serveradressdaten stellen die IP-Adresse des Servers sowie den Port des Sockets dar. Alle diese Informationen befinden sich fest kodiert im Programm *gpsTcp*. Der Vorgang der Verbindungsherstellung basiert auf einer Abfolge von Abfragen mittels des AT-Befehls `CIPSTATUS` und der zurückgelieferten Antworten. Diese Antworten geben den Status des Verbindungsaufbaus wieder. Zu Beginn der Verbindungsherstellung wird die aktuelle Systemzeit festgehalten. Der Versuch eine Verbindung herzustellen wird für einen Zeitraum von bis zu zehn Sekunden unternommen. Ist die Differenz zwischen der dann aktuellen und der festgehaltenen Zeit größer als zehn Sekunden, so wird die Verbindungsherstellung unterbrochen. Dies soll weitere Abfragen der aktuellen Position ermöglichen, da ansonsten die serielle Schnittstelle, aufgrund ununterbrochener Versuche der Verbindungsherstellung zum Server, nicht zur Verfügung steht. Eine Verbindung wird durch den Befehl `AT+CIPMUX=0` initialisiert. Anschließend wird der Status der Verbindung auf die zu erwartende Antwort geprüft. Wird `IP INITIAL` zurückgeliefert, kann im nächsten Schritt eine GPRS-Verbindung mittels der hinterlegten Mobilfunkdaten angefordert werden. Dabei werden diese Daten dem AT-Befehl `AT+CSTT` übergeben. Der Verbindungsstatus sollte nach diesem Schritt `START` lauten. Ist dies der Fall folgt der Befehl `AT+CIICR`, mit welchem eine drahtlose GPRS-Verbindung aufgebaut wird. Die erfolgreiche Verbindungsherstellung wird durch den Statuswert `GPRSACT` bestätigt. Es wird im darauffolgenden Schritt eine IP-Adresse

vom Mobilfunkanbieter angefordert. Diese wird durch den Befehl `AT+CIFSR` realisiert. Wurde dem GSM-Modul eine IP-Adresse zugewiesen (der Statuswert der Verbindung lautet dann `IP STATUS`), kann eine socketbasierte Verbindung hergestellt werden. Dazu werden die IP-Adresse des Servers sowie dessen Port dem AT-Befehl `AT+CIPSTART` übergeben. Wird dieser Aufruf vom Modul mit der Antwort `CONNECT OK` quittiert, ist die Verbindung mit dem Server hergestellt.

6.2.7 Datenübertragung an Server

Wurde eine Verbindung mit dem Server erfolgreich hergestellt, kann die Datenübertragung erfolgen. Die dafür notwendigen AT-Befehle sind in der Funktion *sendData()* zusammengefasst. Hierzu werden die in den zwei Textdateien abgespeicherten NMEA- und CAN-Datensätze in zwei Schleifen ausgelesen und in einer Variablen gespeichert. Bei dieser Variablen handelt es sich um einen *stringstream*, der als Container für Zeichenketten fungiert. Nach Einlesen der Zeilen aus den Textdateien werden diese mittels des «-Operators dem *stringstream* übergeben. Hinter jeder vollständigen Zeile wird ein Semikolon eingeschoben. Dieses dient als Trennzeichen, um abgeschlossene Zeilen voneinander zu diskriminieren. Wurden die Daten in das Programm geladen, erfolgt die Ausführung des AT-Befehls `AT+CIPSEND`. Das zurückgelieferte Zeichen `>` indiziert die von der seriellen Schnittstelle erwartete Dateneingabe. Die eingelesenen Datensätze können nun dem Modul zur Datenübertragung übergeben werden. Ist dies nicht der Fall, liegt ein Verbindungsfehler (Server nicht erreichbar, kein Netz etc.) vor. Es wird dazu die Funktion *buildTcp()* aufgerufen, um die Verbindung wiederherzustellen.

6.3 Serverseitige Prozessierung

Die Verarbeitung der Daten am Server und die Übertragung der verarbeiteten Daten auf den SOS wurde mittels Java-Klassen (siehe dazu in Anhang D Java-Proxy-Quellcode) in einem Eclipse-Projekt realisiert. Bei erstmaliger Erwähnung einer Klasse wird auf den entsprechenden Quellcode im Anhang hingewiesen. Verwaltet werden das Projekt und genutzte *dependencies* (externe Softwaremodule) durch das *Build-Management*-Tool *Maven*. Einstiegspunkt des Proxys ist die statische *main*-Methode der gleichnamigen Klasse (siehe D.1). Dort werden jeweils zwei Instanzen generiert:

- *Server*-Instanz (siehe D.2) – Verwaltet die TCP-Verbindung zum Datenlogger und schreibt die ankommenden Prozess- und Positionsdaten in eine Datenbank
- *SOSFeeder*-Instanz (siehe D.3) – Verarbeitet Prozessdaten aus einer Datenbanksicht und übermittelt sie einer laufenden SOS-Instanz mittels HTTP-POST

Am Server angekommen werden die Daten in einer eigens dafür angelegten Postgresql-Datenbank gespeichert. Dort werden die Daten, nach ihrer Semantik, in entsprechenden Tabellen aggregiert (Tabelle 6.2).

Tabelle 6.2: Tabellen zur Aufnahme der Rohdaten

Tabellenbezeichnung	Aufgenommene Datentypen	Anzahl Tabellenspalten
nmeadata	NMEA-Werte	11
rpmdata	Motordrehzahl	4
fuelconsumptiondata	Kraftstoffverbrauch	4

Die Anzahl der Tabellenspalten der Tabelle *nmeadata* richtet sich nach der Anzahl der komma-separarierten Werte der *NMEA*-Daten (siehe Listing 6.6). Analog dazu richtet sich die Anzahl der Tabellenspalten der beiden anderen Tabellen nach der Anzahl der komma-separierten Werte der CAN-Daten (siehe Listing 6.2). Alle Tabellen haben eine ID-Spalte, welche die eingehenden Datenzeilen nummeriert. Die eindeutige Kennung Datenlogger-Daten wurde in zwei Tabellenspalten, nämlich *start_Counter* und *local_time*, getrennt. Diese Spalten tragen somit die Informationen

- Anzahl der Systemstarts des Datenloggers
- Lokaler Zeitstempel des Rechners

Sie bilden zusammen einen eindeutigen Schlüssel für die Beobachtungen. Eine Datenbanksicht enthält auf der Basis dieses Schlüssels zeitlich korrespondierende Zeilen der drei Tabellen. Ziel ist es diejenigen Zeilen in den Tabellen mit CAN-Daten zu identifizieren, welche annähernd den gleichen Zeitstempel haben, wie den einer NMEA-Datenzeile (Abb. 6.7).

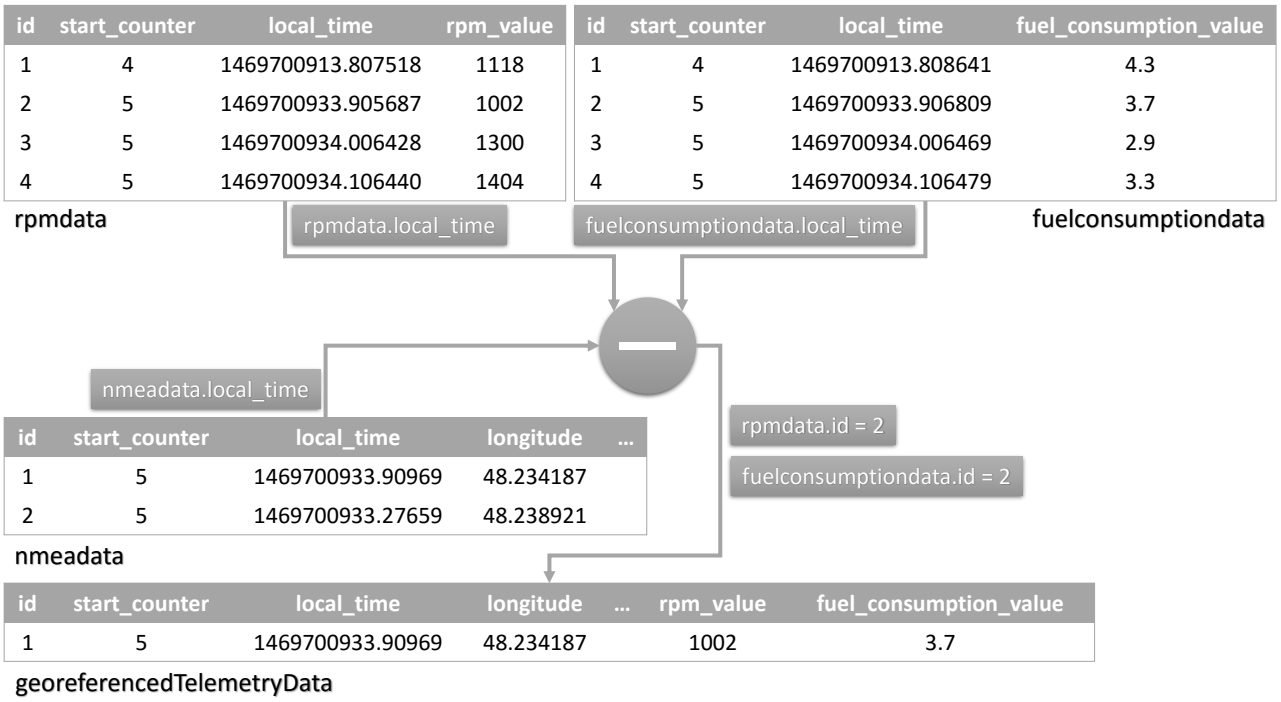


Abbildung 6.7: Schema der Georeferenzierung der Telemetriedaten

Unter der zusätzlichen Voraussetzung, dass dabei die Werte in der Spalte *start_Counter* dieser zu vergleichenden Zeilen übereinstimmen, wird die absolute Differenz zwischen dem lokalen Zeitstempel einer Zeile aus der Tabelle *nmeadata* mit allen Zeitstempeln der beiden anderen Tabellen berechnet. In der Abb. 6.7 wird diese Differenzbildung (dargestellt in der Abbildung durch Differenzoperator) für die erste Zeile der *nmeadata*-Tabelle demonstriert. Die Tabellenbezeichnungen befinden sich unter den Tabellen (schwarze Beschriftung). Da der Wert der Spalte *start_counter* in der ersten Zeile der Tabellen *rpmdata* und *fuelconsumptiondata* einen anderen Wert trägt (4) als der korrespondierende Wert in der erste Zeilen der Tabelle *nmeadata* (5), werden jeweils die ersten Zeilen beider CAN-Datentabellen bei der Differenzbildung ignoriert. Von den übrigen Zeilen der CAN-Datentabellen stellt jeweils die zweite Zeile diejenige Zeile dar, welche die geringste zeitliche Abweichung zu der ersten Zeile der *nmeadata*-Tabelle aufweist. Diese Zeilen mit der kleinsten Differenz werden mithilfe ihrer Zeilen-ID referenziert (in der obigen Abbildung durch die Blöcke $rmpdata.id = 2$ bzw. $fuelconsumptiondata.id = 2$ dargestellt). Die IDs werden genutzt, um die entsprechenden Zeilen der CAN-Datentabellen an die Positionsdaten anzufügen. Die in Abb. 6.7 beschriebene Prozedur wird für jede Zeile der Tabelle *nmeadata* durchgeführt. Die Datenbanksicht weist die gleiche Struktur wie die Tabelle *nmeadata* auf, wird jedoch zusätzlich um zwei Spalten (Drehzahl-, Kraftstoffverbrauchswerte) erweitert. Die georeferenzierten Daten aus dieser Sicht werden durch den *SOSFeeder* stückweise und laufend abgefragt. Nach der Extraktion durchlaufen diese ein *Parsing*, und werden in Attribute eines Objekts abgebildet. Eine solche Instanz enthält eine Position mit Zeitstempel sowie entsprechende Prozessdaten. Diese Instanz wird in ein XML-Dokument mit vordefinierter Struktur überführt.

Die Struktur wurde entsprechend der Spezifikationen des O&M-Standards vorgenommen. Die Registrierung des Telemetriesystems wird mittels eines zuvor erstellten SensorML-Dokuments durchgeführt. Dieses XML-Dokument wird über HTTP-POST an eine laufende SOS-Instanz übergeben. Der Grund für die Verwendung dieser transaktionalen Schnittstelle des SOS liegt in der Anpassungsfähigkeit dieser Schnittstelle: Die direkte Übertragung der Daten in die Datenbank des SOS mittels SQL ist verhältnismäßig aufwendig vorzunehmen und versagt bei einer Änderung des Datenmodells in der Datenbank. Eine solche Änderung tritt z. B. im Rahmen eines Updates der SOS-Implementierung auf. Zudem bleibt diese Möglichkeit verwehrt, wenn sich der SOS nicht auf demselben Rechner befindet. Der mit der Verwendung von direkten SQL-Statements verbundene Wartungsaufwand wird durch Verwendung der transaktionalen Schnittstelle weitgehend vermieden. In den folgenden Kapiteln wird detailliert auf die serverseitige Verarbeitung der Prozessdaten eingegangen.

6.3.1 Verbindungsherstellung

Während des Serverbetriebs werden eingehende Verbindungsanfragen des Datenloggers erwartet. Eine TCP-basierte Verbindung wird serverseitig durch die Klasse *ServerSocket* realisiert und mittels einem 1-Parameter-Konstruktor instanziiert. Das dabei zu übergebende Argument stellt die Port-Nummer dar, mit dem der Server erreicht werden kann. Der Port des Servers ist auch auf Client-Seite fest kodiert. Ist ein *ServerSocket*-Objekt instanziiert, wird die Methode *accept()* ausgeführt. Diese Methode liefert, bei erfolgreicher Verbindungsherstellung, eine Instanz der Klasse *Socket* zurück. Mittels dieser Instanz wird das Einlesen der Daten ermöglicht. Die Klasse *Server* (siehe in Anhang D.2) beinhaltet dazu ein *ServerSocket* als Attribut und führt die oben beschriebene Prozedur mittels der Methode *listen()* aus. Am Einstiegspunkt des Programmes wird eine Instanz dieser Klasse mit Übergabe der erreichbaren Port-Nummmer erzeugt. Der zugehörige Konstruktor führt daraufhin die Methode *listen()* aus. Während der gesamten Betriebszeit des Servers, „lauscht“ der Server am angegebenen Port, um die Herstellung von neuen Verbindungen zu ermöglichen. Ist die Verbindungsherstellung erfolgreich, wird im letzten Schritt der Methode *listen()* der 2-Parameter-Konstruktor der Klasse *ServerThread* aufgerufen. Diese Klasse (siehe in Anhang D.4) implementiert die *Runnable*-Schnittstelle und überschreibt deren *run()*-Methode. Dadurch wird die Ausführung des in der Methode *run* enthaltenen Programmcodes in einem separaten *Thread* ermöglicht. Dies ist Voraussetzung für die parallel stattfindende Verarbeitung der Daten, ohne das „Lauschen“ auf weitere Verbindungsanfragen durch die Klasse *Server* zu unterbrechen. Der *ServerThread*-Konstruktor nimmt die aktuelle *Server*-Instanz sowie die erzeugte *Socket*-Instanz an. Erstere Instanz wird benötigt, um die Verbindung, wenn erforderlich, zu beenden. Letztere Instanz ist dafür zuständig, die eingehenden Daten des Datenloggers zu empfangen und zu verarbeiten. Gestartet wird der Code der Klasse *ServerThread* im Konstruktor durch die Instanziierung eines *Thread*-Objekts und dem anschließenden Aufruf der zugehörigen statischen Methode *start()*. In der *run*-Methode werden, nach erfolgreichem Aufruf der *start-*

Methode, die Methoden *processData()* und *removeConnection()* für die vorgesehenen Schritte Datenverarbeitung und Verbindungsbeendung getätigt. Die Methode *processData()* umfasst die Datenentgegennahme und Speicherung der Daten in der Datenbank. Auf diese Schritte wird in den nächsten Kapiteln eingegangen.

6.3.2 Rekonstruktion der textuellen Daten

Die Methode *processData()* nimmt zunächst den eingehenden Datenstrom des Datenloggers als Argument an und instanziiert ein Objekt der Klasse *DataInputStream*, um die für das Einlesen benötigten Methoden aufzurufen. Nach diesem initialen Schritt beginnt das Einlesen der Daten. Dies geschieht mittels des vom Socket übergebenen Datenstroms. Dazu werden in einer Schleife *Bytes* des Eingangsdatenstroms, mittels der zugehörigen Methode *read()*, ausgelesen. Diese liest *byte* für *byte* und gibt sie als Zahl im Bereich von 0 bis 255 zurück. Solange die von dieser Methode zurückgegebene Zahl größer als -1 ist, liegen Daten vor und der Einleseprozess wird fortgesetzt. Für die weitere Verarbeitung ist dabei der Verlust des ersten Zeichens des übermittelten Datenstroms durch den initialen Aufruf der *read()*-Methode in der Schleifenbedingung zu beachten. Der Verlust des ersten Zeichens ist geeignet zu berücksichtigen. Im Rahmen der Arbeit findet die Berücksichtigung auf Client-Seite statt: Hierzu wird den zu übertragenden Daten ein Blindzeichen vorangestellt, um die restlichen Zeichen vor einem Verlust durch den serverseitigen Leseprozess zu bewahren. Damit die Daten gespeichert werden können, wird im nächsten Schritt ein *array* alloziert, welches Daten vom Datentyp *byte* aufnehmen kann. Die Daten werden mittels der überladenen Methode *read()* eingelesen. Diese nimmt als Argument ein *byte array* an und speichert den gelesenen *byte*-Wert in diesem *array*. Sind die gelesenen Werte im *array* gespeichert, erfolgt eine Konvertierung in textuelle Information. Mittels des überladenen Konstruktors der *String*-Klasse kann eine Instanz der Klasse *String* mithilfe des *byte array* erzeugt werden. Dieser *String* enthält die einzelnen Datensätze, welche durch Semikola getrennt sind.

6.3.3 Speicherung in der Datenbank

Die Klasse *DataSourceConnectorFactory* (siehe D.9) stellt Mittel zur programmgesteuerten Verbindungsherstellung mit Datenbanken bereit. Sie wurde mit fest kodierten Verbindungsparametern implementiert und weist statische *Getter*-Methoden auf, welche *DataSource*-Instanzen zurückliefern. Die Klasse *DataSourceConnector* (siehe D.8) nutzt eine solche *DataSource*-Instanz zur Verbindung und Kommunikation mit der eigens für den Zweck der Telemetriedatenspeicherung anlegten Postgresql-Datenbank zur Verfügung. Ihre Methoden nutzen die Funktionen der JDBC-API, einer Programmierschnittstelle zur Implementierung von Datenbank-basierten Anwendungen. Die Datenbank trägt die Bezeichnung *telemetrySystemDatabase* und enthält drei Tabellen, welche die komma-separierten Werte der drei unterschiedlichen Datensätze (NMEA,

Drehzahl, Kraftstoffverbrauch) aufnehmen. Zum Einfügen der Daten in diese Tabellen werden in der Methode *insertValues()* drei *PreparedStatement*-Objekte instanziiert. Diese beinhalten parametrisierte SQL-*INSERT*-Anweisungen, welche an den vorgesehenen Stellen die kommaseparierten Werte einer Datenzeile aufnehmen. (Listing 6.7).

```
...
public int[] insertValues(String linesOfData) {
    // SQL parameterized Insert-statements
    String sqlForInsertNMEA = ("INSERT INTO nmeadata VALUES (Default, ?, ?,
       ?, ?, ?, ?, ?, ?, ?, ?)");
    String sqlForInsertRPM = ("INSERT INTO rpmdata VALUES (Default, ?, ?, ?)
       ");
    String sqlForInsertFuelConsumption = ("INSERT INTO fuelconsumptiondata
       VALUES (Default, ?, ?, ?)");
...
```

Listing 6.7: Ausschnitt der *insertValues*-Methode

Der im vorangehenden Schritt rekonstruierte *String* wird als Argument der Methode *insertValues()* eines *DataSourceConnector*-Objekts übergeben. Zur Einpflegung der einzelnen Datenwerte wird dieser *String* in einem ersten Schritt an den Stellen der Semikola in die einzelnen Datenzeilen getrennt. Ein Element des resultierenden *String array* stellt eine solche Telemetrie-Datenzeile dar. Die Werte in einer solchen Zeile werden durch Kommata getrennt. In einer Schleife werden für jede Datenzeile die Werte extrahiert und als Parameter des entsprechenden *PreparedStatement*-Objekts festgesetzt. Das Festsetzen geschieht bei JDBC-*PreparedStatements* mithilfe von *Setter*-Methoden, deren Signatur mit dem Datentyp der Zieltabellenspalte übereinstimmen muss: Beispielsweise muss die Methode *setInt()* zur Festlegung eines *Integer*-Wertes in der SQL-*Insert*-Anweisung aufgerufen werden, wenn die vorgesehene Spalte der Zieltabelle für *Integer*-Werte definiert ist. Die Fallunterscheidung bei dieser Zuordnung, welche Datenzeile mit welchem *PreparedStatement*-Objekt korrespondiert, basiert zum einen auf der Länge dieser Datenzeilen: Besteht die Datenzeile aus elf Werten, so handelt es sich dabei um einen NMEA-Satz. Andernfalls handelt es sich um Zeilen, bestehend aus vier Werten, welche Fahrzeugdaten beinhalten. Zum anderen wird bei der Unterscheidung, welcher Fahrzeugdatentyp vorliegt, das dritte Element des *array* einer Datenzeile untersucht. Dieses Element enthält einen *String* (R für Drehzahl, F für Kraftstoffverbrauch), welcher Auskunft über den Fahrzeugdatentyp gibt. Nachdem die Parameter der *PreparedStatement*-SQL-Anweisungen mit konkreten Werten festgelegt wurden, werden diese einem Stapel (*Batch*) hinzugefügt. Für jede *PreparedStatement*-Instanz wird ein Stapel bereitgestellt. Sind die Datenzeilen abgearbeitet, werden die im Stapel enthaltenen Anweisungen mittels der Methode *executeBatch()* abgearbeitet. Die Stapelverarbeitung ist, verglichen mit der sequentiellen Ausführung einzelner SQL-Anweisungen, effizienter, da für den Stapel nur einmal der Weg zur Datenbank aufgenommen werden muss. Die Methode *insertValues()* gibt nach Abschluss der o. g. Schritte die Anzahl der eingefügten Zeilen zurück. Diese Information kann zur Validierung der erfolgreichen Einfüge-Operationen herangezogen werden und wurde mittels eines implementierten *JUnit*-Tests geprüft.

6.3.4 Datenabruf aus Datenbanksicht

Während die Klasse *ServerThread* die Telemetriedaten des Datenloggers empfängt und in der Datenbank speichert, ruft die Klasse *SOSFeeder* (siehe D.3) die in der Datenbanksicht *georeferencedTelemetryData* georeferenzierten Telemetriedaten ab. Dies geschieht kontinuierlich über die gesamte Betriebszeit des Proxy-Programms. Die Definition dieser Sicht enthält eine SQL-*SELECT*-Abfrage, welche in einer weiteren *SELECT*-Abfrage eingebettet ist. Die innere Anweisung ordnet jeder Zeile der Tabelle *nmeadata* mögliche Fahrzeugdatenwerte sowie deren absoluten Zeitunterschied bezogen, auf die lokale Zeit des Datenloggers, zu (Listing 6.8).

```
SELECT DISTINCT ON(nmeadata.id)
  nmeadata.id as nmeadata_id,
  rpmdata.id as rpmdata_id,
  fuelconsumptiondata.id as fuelconsumption_id,
  nmeadata.start_counter as nmeadata_start_counter,
  rpmdata.start_counter as rpmdata_start_counter,
  fuelconsumptiondata.start_counter as fuelconsumptiondata_start_counter,
  abs(nmeadata.local_time-rpmdata.local_time_identity) as abs_time_difference_rpmdata,
  abs(nmeadata.local_time-fuelconsumptiondata.local_time_identity) as
      abs_time_difference_fuelconsumptiondata
FROM
  public.nmeadata,
  public.rpmdata,
  public.fuelconsumptiondata
  WHERE
  nmeadata.start_counter = rpmdata.start_counter AND rpmdata .start_counter =
      fuelconsumptiondata.start_counter
ORDER BY
  nmeadata_id, abs_time_difference_rpmdata, abs_time_difference_fuelconsumptiondata ASC
```

Listing 6.8: Anweisung zur zeitlichen Zuordnung der Prozessdaten zu den Positionsdaten

Das resultierende Ergebnis dieser Anweisung wird nach der ID der NMEA-Datenzeile sowie nach den Zeitunterschieden aufsteigend geordnet und enthält die entsprechende ID der jeweiligen Prozessdaten aus den Tabellen *rpmdata* und *fuelconsumptiondata*. In der äußeren SQL-*SELECT*-Anweisung wird die *nmeadata* anhand der IDs der inneren Tabelle, welche hier mit *inner_table* bezeichnet wird, um die korrespondierenden Fahrzeugdaten ergänzt (Listing 6.9). Die resultierende Tabelle wird in der Datenbanksicht gespeichert.

```
SELECT
  nmeadata.id,
  nmeadata.longitude,
  nmeadata.latitude,
  nmeadata.altitude,
  nmeadata.utc,
  nmeadata.time_to_first_fix,
  nmeadata.number_of_satellites_in_view,
  nmeadata.speed_over_ground,
  nmeadata.course_over_ground,
  rpmdata.rpm_value,
```

```
  fuelconsumptiondata.fuel_consumption_value
FROM
 nmeadata, rpmdata, fuelconsumptiondata, inner_table
WHERE (nmeadata.id = nmeadata_id AND rpmdata.id = inner_table.rpmdata_id AND
    fuelconsumptiondata.id = inner_table.fuelconsumption_id) AND (
    abs_time_difference_rpmdata < 0.5 AND abs_time_difference_fuelconsumptiondata < 0.5)
```

Listing 6.9: Erweiterung der *nmeadata*-Tabelle um Fahrzeugdatenwerte

Es sei darauf hingewiesen, dass eine zusätzliche Bedingung eingeführt wurde: Die Zeitdifferenz zwischen einer Position und einem Fahrzeugdatenwert soll weniger als eine halbe Sekunde betragen (zu sehen in der *WHERE*-Klausel). Dies ist notwendig, da ansonsten, basierend auf dem aktuellen Inhalt der Datenbanktabellen, die Zeilen mit der kleinsten Differenz verknüpft werden, diese aber nicht unbedingt zeitlich nah beieinander liegen müssen. Im Proxy werden durch die Methode *getVehicleData()* der Klasse *DataSourceConnector* die Daten aus der Datenbanksicht abgerufen. Dazu ruft die Methode *getVehicleData()* die Methode *selectGeoreferencedTelemetryDataView()* auf. In Letzterer ist eine *SELECT*-Anweisung mit *LIMIT*- und *OFFSET*-Klauseln implementiert (Listing 6.10).

```
SELECT * FROM georeferencedTelemetryData ORDER BY georeferencedTelemetryData.id LIMIT %d
    OFFSET %d
```

Listing 6.10: SELECT-Anweisung zur Abfrage der Sichtdaten

Diese extrahiert eine Untermenge aus sämtlichen Zeilen der Datenbanksicht. Dabei werden maximal zehn Einträge (*LIMIT*-Argumentwert ist 10) eingelesen. Diese Anweisung wird der Methode *executeQuery()* eines *Statement*-Objekt übergeben (Listing 6.11).

```
...
try (ResultSet rs = stmt.executeQuery(sql)) {
 while (rs.next()) {
   double longitude = rs.getDouble("longitude");
   double latitude = rs.getDouble("latitude");
   // double altitude = rs.getDouble("altitude");
   String timestamp = rs.getString("utc");
   // int ttff = rs.getInt("time_to_first_fix");
   short numberOfSatellitesInView = rs.getShort("
      number_of_satellites_in_view");
   // double speedOverGround = rs.getDouble("speed_over_ground");
   // double courseOverGround = rs.getDouble("course_over_ground");
   short rpmValue = rs.getShort("rpm_value");
   double fuelConsumptionValue = rs.getDouble("fuel_consumption_value");
   vehicleList.add(
     new Vehicle(longitude, latitude, timestamp, numberOfSatellitesInView,
        rpmValue, fuelConsumptionValue));
  }
}
...
```

Listing 6.11: Ausschnitt aus der Methode *selectGeoreferencedTelemetryDataView()*, Zeilenwerte der Sicht werden in Variablen abgebildet, diese wiederum in einem *Vehicle*-Objekt zusammengefasst

Die Methode führt die SQL-Anweisung durch und gibt eine Instanz der Klasse *ResultSet* zurück. Mittels *Getter*-Methoden des *ResultSet*-Objekts wird eine Zeile der Datenbanksicht in den Programmspeicher geladen: Diese werden in primitive Datentypen bzw. *String*-Objekte abgebildet. Die extrahierten Werte wiederum werden im Anschluss dem Konstruktor der Klasse *Vehicle* übergeben. Für jede Datenzeile wird so eine *Vehicle*-Instanz erzeugt. Die Instanzen werden nach und nach einer *ArrayList* übergeben, welche abschließend von der Methode *getVehicleData()* zurückgegeben wird. Eine solche *ArrayList* enthält somit für jede Zeile der Datenbanksicht ein entsprechendes *Vehicle*-Objekt. Die *OFFSET*-Klausel der zugrundeliegenden *SELECT*-Anweisung beginnt mit dem Wert 0 (lies ab erste Zeile der Datenbanksicht) und wird um die entsprechende Anzahl der eingelesenen Zeilen (konkret ist das die Größe der *ArrayList*) inkrementiert. Im gezeigten Ausschnitt sind kommentierte Anweisungen zu sehen. Diese extrahierten weitere Daten, falls sie auskommentiert wären, aus der Datenbanksicht. In jenem Fall müssten jedoch entsprechend Refaktorierungen des Programmcodes sowie Umstrukturierungen in den statischen SWE-Dokumenten vorgenommen werden, um diese zusätzlichen Daten in den SOS mit einfließen zu lassen.

6.3.5 Abbildung auf Instanzen

Vor der Übertragung der Daten auf den SOS werden diese in Instanzen der Klasse *Vehicle* (siehe in Anhang D.5) gespeichert. Dieser Zwischenschritt vor der Konvertierung der Daten in das O&M-Format ist hilfreich, um die Schnittstelle einheitlich zu halten: Der Übergang von Objektdaten in XML-strukturierte Daten erfolgt stets auf die gleiche Art und Weise. Dies trifft auch auf Änderungen in Bezug auf das Dateiformat der vom Datenlogger übermittelten Daten oder in Bezug auf Komponenten des Telemetriesystems zu. Die Instanzvariablen der Klasse *Vehicle* nehmen die aus der Datenbanksicht enthaltenen Informationen auf (Listing 6.12).

```
public class Vehicle {
  private double longitude;
  private double latitude;
  private String timestamp;
  private int numberOfSatellitesTracked;
  private int engineRPM;
  private double fuelConsumption;

  // Constructor and methods
  ...

}
```

Listing 6.12: Klassendefinition der Klasse *Vehicle*

Die Klasse umfasst Attribute zur Speicherung der geographischen Koordinaten, des Zeitstempels, der Anzahl der sichtbaren Satelliten zum Zeitpunkt der Positionsbestimmung und der fachbezogenen Prozessdaten. Entsprechende Instanzmethoden übernehmen die spezifische Verarbeitung dieser Werte, um das Format der Daten aus der Datenbanksicht in das Format von O&M zu konvertieren. Auf die Anpassungen wird im Folgenden eingegangen. Die Positionsdaten des

Datenloggers liegen im sog. „*DDMM.mmm*“-Format vor. Von rechts nach links gelesen bilden die Ziffern (*mmm*) vor dem Komma sowie die zwei Ziffern (*MM*) nach dem Komma dezimale Minuten. Die darauffolgenden Ziffern (*DD*) repräsentieren den geographischen Grad der jeweiligen Koordinate. Für den SOS werden dezimale Gradangaben, d. h. im Format *DD.ddd* benötigt. Die Konvertierung übernimmt die Methode *convert2DecimalDegrees()*. Diese nimmt als Argument den textuellen Wert einer geographischen Koordinate entgegen. Eine solche Zeichenkette stellt eine Fließkommazahl dar und wird mit der Methode *parseDouble()* der Wrapper-Klasse *Double* durch sog. *unboxing* in den primitiven Datentyp *double* umgewandelt. Um die Minuten von den Gradangaben zu trennen, wird eine Rest-Division, mittels des Modulo-Operators, mit 100 durchgeführt. Nach diesem Schritt werden die dezimalen Minuten (*MM.mmm*) durch Division mit 60 in Dezimalgrad umgerechnet. Die Addition dieses Ergebnisses zu den Ziffern der Gradangabe (*DD*) ergibt die geographische Koordinate in Dezimalgrad. Die Ziffern der Gradangabe resultieren aus der Division der Eingangszahl mit 100 und dem Weglassen der Nachkommastellen. Das Weglassen von Nachkommastellen einer Fließkommazahl wird durch *explicit casting* bzw. *narrowing conversion* vom Datentyp *double* auf den Datentyp *int* ermöglicht.

Der Zeitstempel der Prozessdaten erfordert ebenso eine Umwandlung. Er liegt im Format

```
YYYYMMDDHHmmss.s
```

vor. Für die Speicherung im SOS benötigt er das Format

```
YYYY-MM-DDTHH:mm:ss.sTZD
```

Letzteres Format stammt aus dem Datum- und Zeitstandard ISO:8601 [ISO, 2004] und wird in folgender Auflistung erläutert:

- YYYY = vierstelliges Jahr
- MM = zweistelliger Monat (01=Januar, usw.)
- DD = zweistelliger Tag des Monats (01 bis 31)
- hh = zwei Ziffern für Stunden (00 bis 23)
- mm = zwei Ziffern für Minuten (00 bis 59)
- ss = zwei Ziffern für Sekunden (00 bis 59)
- s = eine oder mehrere Ziffern zur Repräsentation von Dezimalbrüchen der Sekunden
- TZD = Bezeichner der Zeitzone, mögliche Werte sind Z für GMT (*Greenwich Meridian Time*) und konkrete Versätze +hh:mm oder -hh:mm
- T = Zeichen zur Trennung von Datum- und Zeitangabe

Die Methode *convertTimeStamp()* überführt das Zeitformat der Prozessdaten in das für den SOS benötigte Format. Sie separiert die zeitlichen Entitäten durch entsprechende Trennzeichen und fügt einen Zeitzonenbezeichner (in diesem Fall UTC-Zeit) hinzu. Alle weiteren Daten, welche keine Datenverarbeitung erfahren, werden – je nach Datentyp – mit entsprechenden Methoden von Wrapper-Klassen umgewandelt, um die textuellen Werte in vorgesehene Attribute der Klasse *Vehicle* zu speichern.

6.3.6 Marshalling

Im *Marshalling* werden die Attributswerte einer *Vehicle*-Instanz in einem O&M-Dokument abgebildet. Dieses Dokument wurde im Vorfeld manuell erstellt und findet sich im Anhang (siehe dazu in Anhang B.5). Die Erstellung dieses *InsertObservation*-Dokuments richtete sich nach dem ebenfalls in einem vorangehenden Schritt erstellten *InsertSensor*-Dokument. In letzterem wurde entsprechend dem SensorML-Standard das Telemetriesystem modelliert und in eine XML-Struktur überführt (siehe dazu in Anhang B.6). Ein *Insert-Observation*-Dokument beginnt mit einem *sos:InsertObservation*-Element. Kindelemente sind *sos:offering* sowie *sos:observation*. Letzteres enthält *om:OM_Observation*-Elemente, welche die Beobachtungen des Telemetriesystems repräsentieren. Ein *om:OM_Observation*-Element stellt jeweils einen Typ der Prozessdaten dar. Die Werte der Instanzvariablen eines *Vehicle*-Objekts werden im O&M-Dokument auf die XML-Attributswerte oder als Text des entsprechenden XML-Elements gesetzt. Um die Daten eines *Vehicle*-Objekts in das O&M-Dokument einzupflegen, wird die Software-Bibliothek JDOM [Hunter und McLaughlin, 2015] verwendet. Diese erlaubt durch Bereitstellung von Java-Klassen die Manipulation von XML-Dokumenten. XML-Elemente sowie deren Eigenschaften und Attribute werden auf spezifische JDOM-Klassen abgebildet. Die selbst implementierte Klasse *XMLParser* (siehe in Anhang D.6) übernimmt Aufgaben in Bezug auf das *Marshalling*. Sie nutzt dabei die von JDOM zur Verfügung gestellten Klassen und Methoden. Die Methode *readXML()* dieser Klasse liest das im selben Eclipse-Projekt befindliche O&M-Dokument mit der Bezeichnung *InsertObservation.xml* ein. Dies geschieht durch die Methode *build()* einer *SAXBuilder*-Instanz. Die Methode nimmt eine *File*-Instanz als Argument an und konstruiert einen JDOM-spezifischen Baum des O&M-Dokuments. Nach erfolgreichem Einlesen des Dokuments gibt die *build()*-Methode eine *Document*-Instanz zurück. Diese Instanz besitzt Methoden, mit welchen der Zugriff auf die Elemente des XML-Dokuments ermöglicht wird. So wird nach dem Einlesen das Wurzelelement *sos:InsertObservation* als Instanzvariable der Klasse *XML-Parser* gesetzt. Basierend auf der Struktur des O&M-Dokuments werden die XML-Elemente in entsprechende Elemente des JDOM-Baumes abgebildet. Die Methode *marshal()* übernimmt diese Aufgabe. Sie nimmt als Argument eine *Vehicle*-Instanz an und speichert diese in der entsprechenden Instanzvariable der Klasse *XMLParser*. Im nächsten Schritt werden die Elemente *om:offering* sowie *om:observation* durch Anwendung der Methode *getChildren()* des Wurzelelements zugänglich gemacht. In einer Schleife werden für jedes *om:OM_Observation*-Element die Kindelemente aufgerufen, deren Werte mittels der *Vehicle*-Instanzvariablen geändert werden. Tabelle 6.3 zeigt die zu verändernden Elementwerte der Beobachtungen auf.

Tabelle 6.3: Eigenschaften und Inhalte im *InsertObservation*-Dokument, welche dynamisch verändert werden

Beobachtungseigenschaft	O&M-Elementinhalt
Bezeichnung	Inhalt des *gml:identifier*-Elements
Beobachtungszeit	Inhalt des *gml:timePosition*-Elements
Beobachtungsort	Inhalt des *gml:pos*-Elements
Beobachtungswert	Inhalt des *om:result*-Elements

Das *gml:id*-Attribut des *om:OM_Observation*-Elements wird in der SOS-Implementierung von 52°North nicht berücksichtigt. Wurden die Werte der Instanzvariablen des *Vehicle*-Objekts in das O&M-Dokument überführt, wird die zugehörige *Document*-Instanz von der Methode *marshal()* zurückgegeben. Diese Instanz stellt das fertige O&M-Dokument dar, welches im nächsten Schritt der serverseitigen Prozessierung auf den SOS übertragen wird. In einem solchen Dokument liegen zwei *OM_Observation*-Instanzen vor, welche Drehzahl respektive Kraftstoffverbrauch zu einem Zeitpunkt und einer Position angeben (siehe dazu das beispielhafte Dokument in Anhang B.5).

6.3.7 Datenübertragung mittels HTTP-POST

Liegen die Prozessdaten im O&M-Format vor, werden diese über die transaktionale Schnittstelle des SOS selbigem übermittelt. Die Schnittstelle nimmt SWE-basierte Dokumente an und regelt die Speicherung der Werte in der zugrundeliegenden Postgresql-Datenbank. Zur Nutzung dieser Übertragungsmethode ist die Anwendung der HTTP-POST-Methode vorgesehen. Damit fungiert der Proxy als Webclient-Anwendung. In diesem Fall führt er periodisch HTTP-POST-Anfragen aus, um die O&M-kodierten Daten an den SOS zu senden. Zwei Umstände sind bei der im Rahmen dieser Arbeit vorgenommenen Konfiguration des SOS von 52°North hierbei zu beachten: Auf der einen Seite wurde beim SOS die Aktivierung des *transactional profile* vorgenommen und die zugehörigen SOS-Operationen wurden *InsertSensor* sowie *InsertObservation* explizit freigeschalten. Auf der anderen Seite ist die Ausführung dieser Operationen durch die Aktivierung der *Transactional-Security*-Einstellung des SOS auf dem Rechner, auf dem sich der SOS befindet, beschränkt. Dies stellt für die hier beschriebene Systementwicklung kein Problem dar, da sich der Proxy lokal auf dem selben Rechner befindet. Eine Änderung dieser Konstellation erfordert jedoch Maßnahmen zur Zugriffsregelung. Beispielsweise ist der Einsatz von *transactional authorization tokens* bzw. einem WSS (*Web Security Service*) denkbar. Auf den Aspekt der Zugriffsregelung sei jedoch nur hingewiesen, da sie keine essentiellen Komponenten für den hier vorliegenden *Workflow* der Datenübertragung an den SOS darstellen. Realisiert wird die HTTP-POST-Methode mittels der Bibliothek *Apache HttpClient* [The Apache Software Foundation, 2016]. Diese wird in der Klasse *SOSClient* (siehe in Anhang D.7) eingebunden. *SOSClient* bedient sich der in *Apache HttpClient* enthaltenen Klassen und Methoden, um HTTP-POST-Anfragen zu generieren. Grundsätzlich ist dazu die Angabe des Hosts, d. h. der Adresse des SOS, sowie des Inhalts und des *Internet Media Type*, in diesem Fall XML, der zu übergebenden Nachricht, notwendig. Die Methode HTTP-POST wird durch die Methode *postDocument()* der Klasse *SOSClient* implementiert. Zunächst wird darin der JDOM-Baum in eine Zeichenkette mittels der Methode *output()* eines *XMLOutputter*-Objekts umgewandelt und in einem *String*-Objekt gespeichert. Zur Initialisierung der HTTP-POST-Anfrage wird eine Instanz der Klasse *CloseableHttpClient* mittels der statischen Methoden der Klasse *HttpClientBuilder* erzeugt. Die Adresse des SOS wird als *String* dem 1-Parameter-Konstruktor der Klasse *HttpPost* übergeben. Zur Festsetzung der HTTP-POST-Nachricht, sowie des *Internet Media Type* wird der 2-Parameter Konstruktor der Klasse *StringEntity* aufgerufen. Diese *StringEntity*-Instanz wird mittels der *Setter*-Methode *setEntity()* der *HttpPost*-Instanz übergeben. Damit ist die Vorberei-

tung zur Übermittlung der HTTP-POST-Anfrage an den SOS abgeschlossen. Zur Ausführung der Anfrage wird die *HttpPost*-Instanz der Methode *execute()* des *CloseableHttpClient*-Objekts übergeben. Diese Methode gibt eine Instanz der Klasse *CloseableHttpResponse* zurück. Bei erfolgreicher Übermittlung der Anfrage enthält die zurückgegebene Instanz die Antwort des SOS, welche mittels der Methode *getEntity()* erhalten werden kann. Zur Ausgabe der Nachricht kann eine solche *StringEntity*-Instanz mittels der statischen Methode *toString()* der Klasse *EntityUtils* als *String*-Objekt zurückgegeben werden. Abschließend wird die Anfrage durch Aufrufen der statischen Methode *consume()* der Klasse *EntityUtils* geschlossen, d. h. es wird gewährleistet, dass der gesamte Inhalt der Serverantwort verarbeitet wurde und der Datenstrom geschlossen wird. Zusätzlich wird der interne Zustand des *HttpPost*-Objekts durch Aufrufen der Methode *reset()* zurückgesetzt, um diese für weitere Anfragen zur Verfügung zu stellen. Damit wird die Ausführung von SOS-Operationen zur Übermittlung der in den *Vehicle*-Objekten gespeicherten Daten ermöglicht. Die Klasse *SOSClient* enthält weitere Methoden, welche sich der *postDocument()*-Methode bedienen. Diese Methoden tragen die gleichen Bezeichnungen wie die vorgesehenen zu nutzenden SOS-Operationen. Beispielsweise werden durch die Methode *InsertObservation()* Prozessdaten dem SOS übergeben. Das im vorherigen Schritt erstellte O&M-Dokument wird dieser Methode übergeben. Diese wiederum übergibt den Inhalt des O&M-Dokuments der Methode *postDocument()*, um die letztendliche Datenübertragung an den SOS durchzuführen. Bei Einsetzen des *InsertObservation*-Dokuments muss in einem vorherigen Schritt überprüft werden, ob der Sensor bereits im SOS gespeichert ist. Dazu wird im Konstruktor der Klasse *SOSClient* eine *GetCapabilities*-Anfrage mittels der gleichnamigen Methode ausgeführt. Das dazu genutzte Dokument findet sich im Anhang (siehe dazu in Anhang B.7). Die vom SOS zurückgelieferte Antwort (in Form eines *GetCapabilitiesResponse*-Dokuments) wird mittels XPath (XML *Path Language*) nach dem Bezeichner des Telemetriesystems durchsucht. Wurde dieser Bezeichner gefunden, wird wie gewohnt fortgefahren. Ansonsten liegt noch keine Registrierung des Telemetriesystems vor. Zur Registrierung des Telemetriesystems wird daher die *InsertSensor()*-Methode der Klasse *SOSClient* durchgeführt, welche das ebenfalls im Vorfeld erstellte *InsertSensor*-Dokument (siehe dazu in Anhang B.6) dem SOS übergibt.

7 Systemevaluierung

7.1 Erprobung des CAN-Moduls

Am 21. Juli 2016 erfolgte die zum damaligen Zeitpunkt erste Erprobung der aktuellen Version des Systems. Dazu wurde im Vorfeld ein Treffen mit dem Betreuer Thomas Machl und Prof. Noack von der Hochschule Weihenstephan-Triesdorf vereinbart. Bei diesem Treffen wurde das System mit einem Kabel, welches von einem Mitarbeiter Prof. Noacks angefertigt wurde, mit einem Traktor verbunden. Der Traktor befand sich auf dem Hochschulgelände und wies für die Verbindung eine ISOBUS-Buchse auf. Zu diesem Zeitpunkt war das System in der Lage, Fahrzeugdaten durch die CAN-Schnittstelle zu erfassen. Bestätigt wurde die Funktionalität durch Vergleich der Bildschirmausgabe (realisiert durch einen angeschlossenen Laptop) des Datenloggers mit dem Armaturenbrett des Traktors. Die angezeigten Werte für Motordrehzahl und Kraftstoffverbrauch stimmten mit den vom Datenlogger gemessenen Daten überein. Zusätzlich wurden Daten des Traktors mit dem Programm *candump* aus der Werkzeugsammlung *SocketCAN* in einer Logdatei[1] abgelegt. In den Abb. 7.1 und 7.2 werden diese Daten dargestellt. In dieser Logdatei befinden sich 5461 Zeilen an Fahrzeugdaten unterschiedlicher Kategorien. Mittels eines Python-Skripts (siehe in Anhang C.2) wurden die darin enthaltenen Werte (jeweils 327 Zeilen) für Motordrehzahl und Kraftstoffverbrauch extrahiert und dekodiert. Die Darstellungen wurden mittels eines weiteren Python-Programms (siehe in Anhang C.3) generiert und basieren auf der Ausgabe des Programms *readDecodeCANFromFile.py*. Letzteres erzeugt aus einer *candump*-Logdatei eine Textdatei mit dekodierten CAN-Daten. Auf den Abbildungen sind Aufzeichnungen von Fahrzeugdaten zu sehen. Die gesamte Messdauer beträgt 32,603 Sekunden. Dabei zeigen beide Kurven ähnliche Verläufe, wobei die Drehzahlwerte einen glatteren Verlauf aufweisen. Die Verläufe lassen sich anhand der durchgeführten Aktionen im Traktor erläutern. Nachdem der Motor gezündet wurde, wurde im Anschluss das Gaspedal betätigt. Kurz darauf wurde der Druck auf das Gaspedal gelockert und der Motor abgeschaltet. Der Traktor befand sich über die gesamte Messung hinweg im Standbetrieb. Dies hat den Hintergrund, dass zum Zeitpunkt der Messung das System noch nicht so weit entwickelt war, dass Fahrzeugdaten mit Positionen verknüpft werden konnten. Durch die Speicherung einer Logdatei und die Nutzung einer virtuellen CAN-Schnittstelle kann der *Workflow* des Systems ungeachtet dessen weiterentwickelt und getestet werden.

[1] Zu finden im Pfad `\rbpi\candump-2016-07-14_184358.log` auf dem Datenlogger.

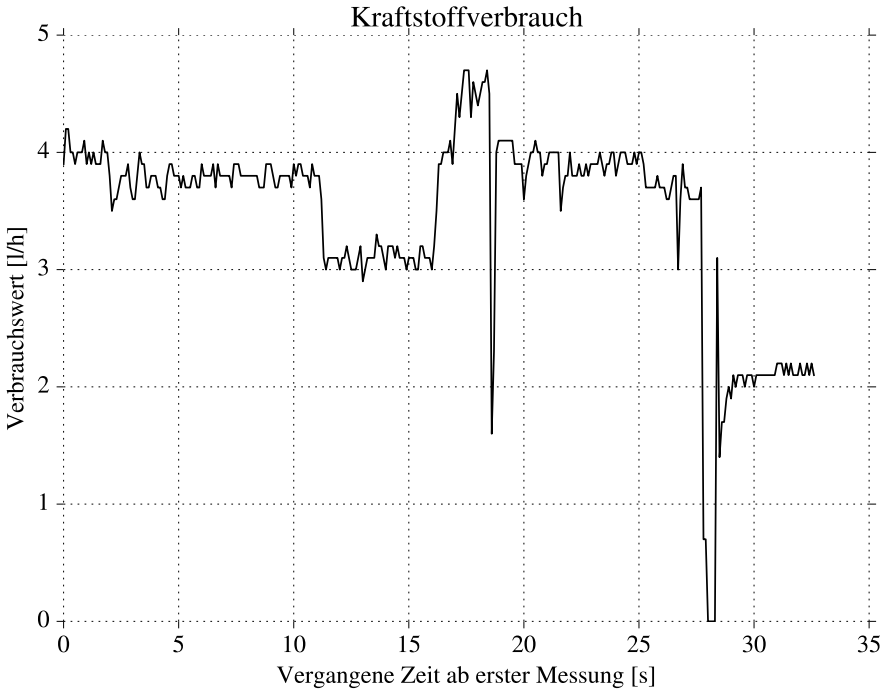


Abbildung 7.1: Vom Datenlogger gemessener Kraftstoffverbrauch

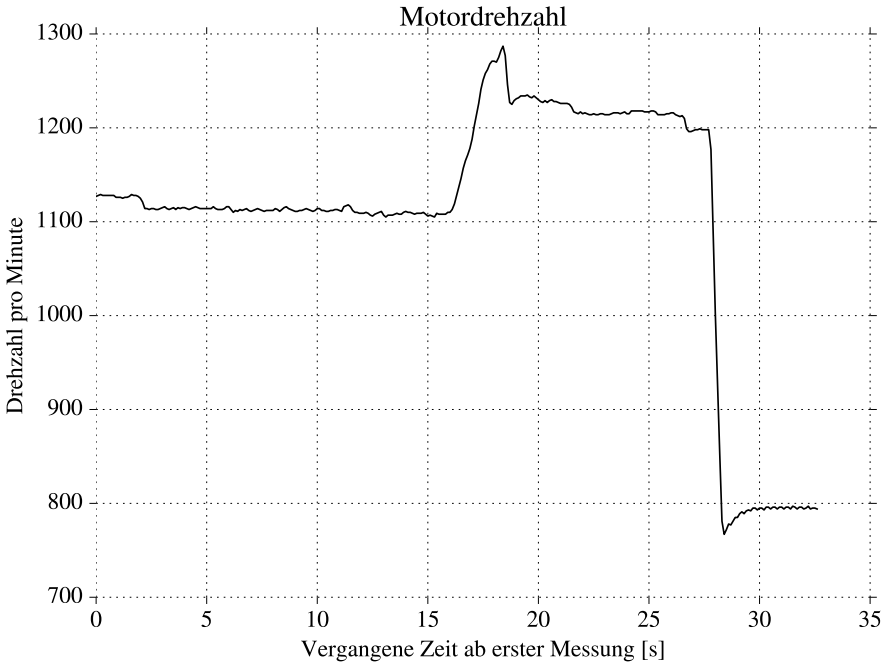


Abbildung 7.2: Vom Datenlogger gemessene Drehzahl

7.2 Erprobung des Systems durch virtuelle CAN-Schnittstelle

Mittels des Programms *canplayer* ist es möglich eine von *candump* erzeugte Datei als Eingabe einer virtuellen CAN-Schnittstelle zu nutzen. Damit werden die vom Traktor aufgezeichneten Daten wieder „abgespielt“ und können für Testzwecke genutzt werden. Ein beispielhafter Aufruf im Linux-Terminal sieht wie folgt aus (Listing 7.1).

```
canplayer -l i -I candump-2016-07-14_184358.log
```

Listing 7.1: Terminalbefehl für *canplayer* zur virtuellen CAN-Datenerzeugung

Damit wird der Inhalt (Befehlsargument -I) der o. g. Logdatei ununterbrochen (Befehlsargument -l sowie Argumentwert i) als Ausgabe einer virtuellen CAN-Schnittstelle genutzt. An welche Schnittstelle die Daten gesendet werden, wird anhand des Inhalts festgelegt: Ist in einer *candump*-Datei als Wert der zweiten Spalte (vgl. dazu die in Kapitel 6.2.2 angegebene Textdateistruktur) der Wert *vcan0* angegeben, so sendet *canplayer* einen kontinuierlichen Datenstrom an den virtuellen CAN-Bus, welcher mit dem Programm *readDecodeCAN* abgegriffen werden kann. Zur programmgesteuerten Auswahl, *vcan0* (virtuell) oder *can0* (real), ist die betreffende Zeile im Quellcode von *readDecodeCAN* zu kommentieren bzw. auszukommentieren. Durch diese Schnittstelle werden kontinuierliche Daten aggregiert und können an den Proxy gesendet werden. Die obligatorische Verbindung mit einem Traktor entfällt hierbei. Für Testzwecke kann somit der *Workflow* des Datenloggers – ohne diesen mit einem Fahrzeug zu verbinden – erprobt werden. Der Datenlogger sammelt dabei reale Positionsdaten. Diese Daten werden zusammen mit den virtuellen CAN-Daten an den Server gesendet und verarbeitet.

7.3 Datenabruf aus dem SOS

An Ende der entwickelten Prozesskette werden georeferenzierte Daten im SOS gespeichert. Der Abruf dieser Daten kann auf unterschiedliche Art und Weise durchgeführt werden. Zum einen existieren verschiedene Client-Anwendungen, welche im Rahmen der Arbeit getestet wurden. Auf diese soll in Kapitel 7.3.1 eingegangen werden. Zur Einsicht sowie zur Abrufung der Daten, und somit zur Kontrolle des *Workflow* im SOS, ist es möglich, den in einer SOS-Instanz vorhandenen Test-Client zu nutzen.[2] In diesem wird ein Eingabefeld bereitgestellt, in welches O&M- und SensorML-kodierte Dokumente eingefügt werden können. Ein Ausgabefeld gibt die Antwort des SOS zurück. Damit konnten die erstellten *InsertObservation*- sowie *InsertSensor*-XML-Dokumente geprüft werden. Zum anderen wurde, aufgrund der in jenen Anwendungen fehlenden bzw. wenig ausgereiften Möglichkeiten zur Darstellung von mobilen Sensoren, ein eigener prototypischer Client entwickelt. Diesen Aspekten widmen sich die folgenden Abschnitte.

[2] Eine beispielhafte Demo-Instanz wurde unter der Adresse http://gpslog.ddns.net:8080/52n-sos-webapp/client auf dem in der Arbeit verwendeten Server eingerichtet.

7.3.1 Einschränkungen vorhandener SOS-Clients

Es existieren verschiedene Client-Anwendungen zur Visualisierung von SOS-Daten. 52°North stellt in ihrer Download-Rubrik zwei Clients zur Verfügung. Daneben existieren verschiedene weitere Anwendungen, welche aus verschiedenen Forschungsarbeiten resultierten. Diese stehen ebenfalls bei 52°North zum Download bereit. Auf letztere, prototypische Anwendungen wird jedoch nicht näher eingegangen, da ihre Entwicklung nicht weiter vorangetrieben wird.

Eine verfügbare Anwendung ist der 52°North-Sensor-Web-JS-Client in der Version 1.0. In einer Korrespondenz vom 04.07.2016 mit 52°North-Entwickler Henning Bredel [Bredel et al., 2016] konnten nähere Informationen zu diesem Client gewonnen werden. Der Client basiert laut Bredel auf JavaScript-Technologien und beansprucht vergleichsweise wenig Rechenleistung, da die meiste Datenverarbeitung auf dem Server stattfindet (daher auch die Bezeichnung Thin-Client). Zwischen dem Client und dem 52°North-SOS befindet sich eine „leichtgewichtige“ Zwischenschicht, welche die Kommunikation mit Client und dem SOS übernimmt. Dadurch würde Bredel zufolge die Performance des Clients gesteigert sowie die Entwicklung am Client erleichtert. Durch die Parametrisierung von Beobachtungen werden Sensoren indirekt durch ihre Beobachtungen georeferenziert (siehe dazu in Kapitel 3.4.2). Diese Methodik wird bei mobilen Sensoren von 52°North verfolgt. Der Client unterstützt gegenwärtig dieses Merkmal nicht; zwar werden Messwerte dargestellt, die Positionierung der Sensorstation wird jedoch anhand der Geometrie des *feature of interest* vorgenommen. Weder parametrisierte Beobachtungen, noch Stationskoordinaten, welche bei einer *InsertSensor*-Operation dem SOS übergeben werden, spielen in der Setzung des Markers auf der Kartenansicht eine Rolle. Folglich hat die Modifizierung der Position des Sensors, durch die Operation *UpdateSensorDescription*, auch keinen Einfluss auf die Darstellung. Wenn eine neue Beobachtung im SOS gespeichert wird, beeinflussen veränderte Koordinaten eines *feature of interest* die Positionierung des Markers nicht, es wird auch kein neuer Marker erstellt. Wird ein neues *feature of interest* bei der *InsertObservation*-Operation eingeführt, d. h. der Wert des `gml:identifier`-Elements wird geändert, erscheint ein neuer Marker auf der Karte. Mit dem neuen Marker ist eine neue Zeitreihe verbunden, d. h. konsekutive Messwerte werden nicht in einem Diagramm gezeigt. Vielmehr ist die Einführung eines neuen *feature of interest* mit der Einführung einer neuen Sensorstation verbunden. In diesem Fall listet der Client zwei Stationen mit demselben Namen auf. Dies ist auf die Kopplung der *procedure* mit dem *feature of interest* im momentanen Zustand des JavaScript-Clients zurückzuführen: Ein Marker, welcher die Sensorstation repräsentiert, wird in die Karte eingefügt, sobald eine Beobachtung mit einem *feature of interest* eingefügt wird. Die Bezeichnung der Station wird aus der Sensorbeschreibung entnommen. Diese Eigenschaften des Clients schließen seinen Einsatz zur Darstellung mobiler Sensoren leider aus.

Eine andere Anwendung ist der SensorWebClient. Er liegt in der Version 3.1.0 vor und erlaubt den Zugriff auf Daten von Zeitreihen, welche im SOS gespeichert sind. Implementiert wurde der Client mit dem Werkzeug GWT (*Google Web Toolkit*) [Bredel et al., 2016]. Laut der offiziellen Beschreibung sind die Merkmale des Clients die Auflistung von Sensorstationen und die Auswahl dieser auf einer Karte. Weiterhin ist der Beschreibung zufolge möglich, im SOS befindliche Daten als Zeitreihen in Diagrammen darzustellen. Mehrere Zeitreihen können demnach miteinander

kombiniert werden. Bei der Nutzung des Clients wurden jedoch Probleme festgestellt. Angezeigt werden in der Liste der Sensorstationen drei beispielhafte Sensorstationen, welche nicht in der eigenen SOS-Instanz gespeichert sind. Die im SOS registrierten Stationen hingegen werden im Client nicht aufgeführt. Zudem lädt die Karte in der Anwendung ununterbrochen, bis nach einem Zeitraum von etwa fünf Minuten der Ladevorgang abgebrochen wird. Nach einer Korrespondenz mit 52°North-Entwickler Henning Bredel [Bredel et al., 2016] könnten diese Schwierigkeiten auf folgende Umstände zurückgeführt werden: Die momentane Inkompatibilität zwischen dem 52°North-SOS-Server und dem Client läge darin begründet, dass das Basisverbindungsmodul des SOS derzeit keine mobile Sensordaten unterstütze. Bredel zufolge stellt dieser Client momentan nicht den Schwerpunkt der Entwicklungsarbeiten dar. Dies gilt nach 52°North-Entwickler Jan Schulte auch für die vorliegende Version des 52°North-Sensor-Web-JS-Client. Dieser unterstützt keine parametrisierten Beobachtungen und zurzeit lägen keine Pläne für die weitere Ausarbeitung dieses Clients vor (Korrespondenz vom 08.07.2016 [Bredel et al., 2016]).

Auch die bekannten Geoinformationssysteme ArcGIS sowie QGIS besitzen Plug-Ins, mit denen die Kommunikation zu einem SOS ermöglicht wird. Jedoch können parametrisierte Beobachtungen zum Stand dieser Arbeit noch nicht durch diese Geoinformationssysteme verarbeitet werden. Dies ist nachvollziehbar, vergegenwärtigt man sich der Tatsache, dass selbst die genannten Clients von 52°North, dem Unternehmen, das eine führende Rolle in der Konzeption und Implementierung des SOS einnimmt, keine parametrisierten räumlichen Beobachtungen darstellen können.

Laut Schulte konzentrieren sich die Arbeiten bei 52°North zurzeit auf den Nachfolger Sensor-Web-Thin-Client, auch *Helgoland* genannt, des im ersten Abschnitt genannten Sensor-Web-JS-Client [Bredel et al., 2016]. Dieser wurde im Rahmen dieser Arbeit getestet, jedoch ist die Benutzung sporadisch mit Problemen verbunden, da sich der Client sowie die entsprechende SOS-Instanz in der frühen Testphase befinden. Es konnten jedoch erfolgreich Funktionen getestet werden, welche im Folgenden beschrieben werden. In dem Client wird die Visualisierung mobiler Sensordaten durch eine Trajektoriendarstellung realisiert. Die Positionsinformationen im *om:parameter*-Element einer parametrisierten Beobachtung werden auf einer Karte dargestellt. Punkte werden miteinander, gemäß ihrer zeitlichen Abfolge, verbunden. Fährt man mit dem Mauszeiger über einen Punkt, so werden Beobachtungswert, Messseinheit und Zeitpunkt der Messung (ohne Sekunden) in einem *Popup* visualisiert. Ein unter der Kartendarstellung positioniertes Diagramm stellt die Messwerte der zugehörigen Trajektorienpunkte dar. Auf der Ordinate lassen sich die Messwerte ablesen. Auf der Abszisse werden die Entfernungen (in km) der auf die erste Messung folgenden Messungen dargestellt. Diese Trajektorien wurde mit der sich in der Entwicklung befindenden Version 4.4.0 des 52°North-SOS getestet. Diese unterscheidet sich von der offiziellen Version 4.3.7 des SOS, da dadurch laut 52°North-Entwickler Carsten Hollmann (Korrespondenz vom 08.07.2016 [Bredel et al., 2016]) Anpassungen

[3] https://52north.org/delivery/SensorWeb/SOS/52n-sensorweb-sos-bundle-4.4.0-Extended-REST-API-Helgoland.zip

[4] http://gpslog.ddns.net:8080/52n-sos-webapp/static/client/helgoland/

an das Datenmodell des SOS durchgeführt wurden, die die Trajektoriendarstellung und die Unterstützung von räumlichen Beobachtungen ermöglichen. Die Beta-Version dieses SOS und des *Helgoland*-Clients können zum gegenwärtigen Zeitpunkt der Arbeit heruntergeladen werden. [3] Eine solche Instanz wurde auf dem in der Arbeit verwendeten Server eingerichtet. [4]

7.3.2 Prototypischer JS-Client

Da die offiziellen Clients die Visualisierung von georeferenzierten Beobachtungen momentan nicht unterstützen, wurde im Rahmen dieser Arbeit ein prototypischer Client unter Zuhilfenahme der JS-Bibliothek *jQuery* [The jQuery Foundation, 2016] sowie der *Leaflet*-API [Agafonkin, 2015] zur Visualisierung der Prozessdaten entwickelt (siehe Abb. 7.3, 7.4). Dieser Client sendet periodisch eine *GetObservation*-Anfrage per HTTP-POST an den SOS und verarbeitet das zurückgelieferte Ergebnisdokument. Für jede Beobachtung, d. h. *om:OM_Observation*-Instanz, wird ein Marker auf der Karte generiert. Diese werden nach der Beobachtungsgröße in verschiedene *Layer* getrennt, welche zu- und abgeschaltet werden können. Durch Klick auf einen Marker wird die entsprechende Beobachtung ausgewählt. Es erscheint ein auf CSS3 (*Cascading Style Sheets Level 3*) basierendes *Popup* mit Informationen über Prozedur, *feature of interest*, Beobachtungsgröße, Messwert, Beobachtungszeit, Position sowie, wenn vorhanden, Einheit. Dieser Marker wird für jede *om:OM_Observation*-Instanz eines *GetObservationResponse*-Dokuments erzeugt. Zu beachten ist bei dieser Anwendung, dass die Visualisierung allein für die Daten des vorliegenden Telemetriesystems erfolgt: Bei der genannten *GetObservation*-Anfrage, welche fest kodiert im Quellcode des Clients hinterlegt ist, werden lediglich Beobachtungen des hier verwendeten Telemetriesystems abgerufen. Darauf basiert die Verarbeitung des *GetObservationResponse*-Dokuments, welches alle Beobachtungen des Telemetriesystems abruft. Weichen die Dokumente welche dem SOS übermittelt werden von dem hier beschriebenen Format ab, kann eine Visualisierung von SOS-Daten nicht garantiert werden. Auch die *ObservableProperties*, d. h. die vom System erfassten Beobachtungsgrößen, wurden fest kodiert hinterlegt. Die bis dato entwickelte Version weist eine einfache, grundlegende, und somit wenig ausgereifte Struktur auf, die dennoch unkompliziert die Telemetriedaten visualisieren kann. Ein Ausbau dieses Clients kann vorgenommen werden und wird sehr begrüßt. Dieser steht zusammen mit dem Code des in Kapitel 6.3 genannten Java-Proxys in einem *bitbucket repository*[5] zur Verfügung und wurde unter einer Webadresse eingerichtet[6]. Jedoch ist langfristig gesehen eine neue Version des 52°North-JS-Clients, welcher parametrisierte Beobachtungen visualisieren kann, für die Darstellung von SOS-Daten vorzuziehen. Dieser antizipierte Client wird verglichen mit der hier prototypisch entwickelten Anwendung weit ausgereifter sein.

[5] https://bitbucket.org/jd_fu/telemetrieserver.git
[6] http://gpslog.ddns.net:8080/sosClient

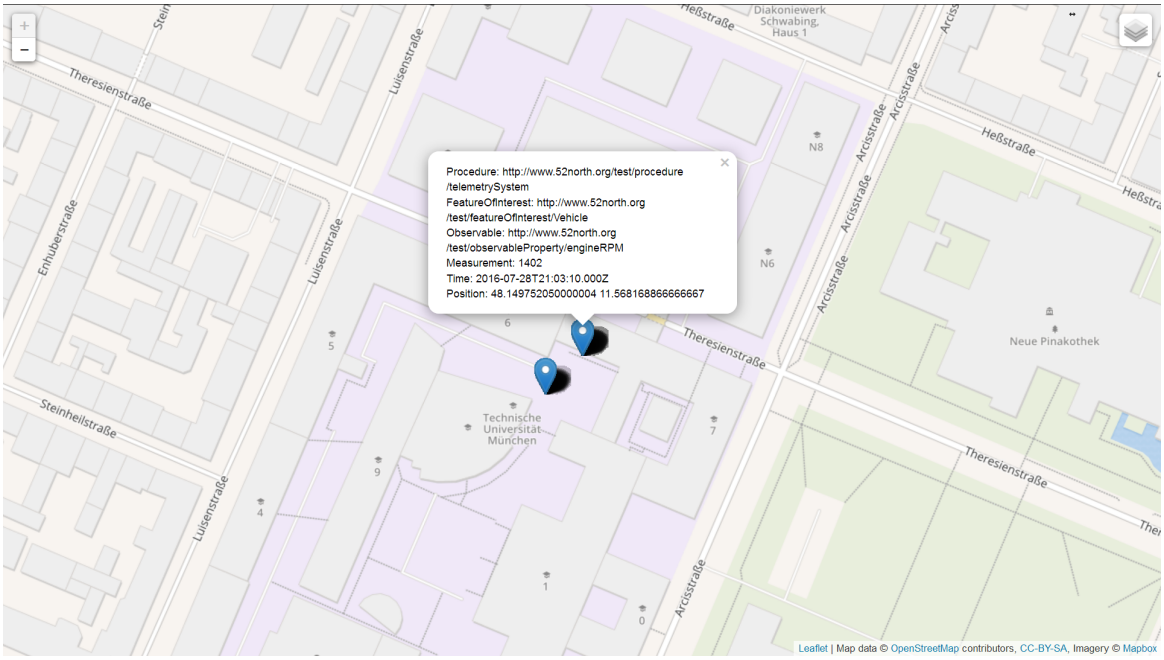


Abbildung 7.3: Von SOS-Instanz abgerufene virtuelle Drehzahl-Beobachtung

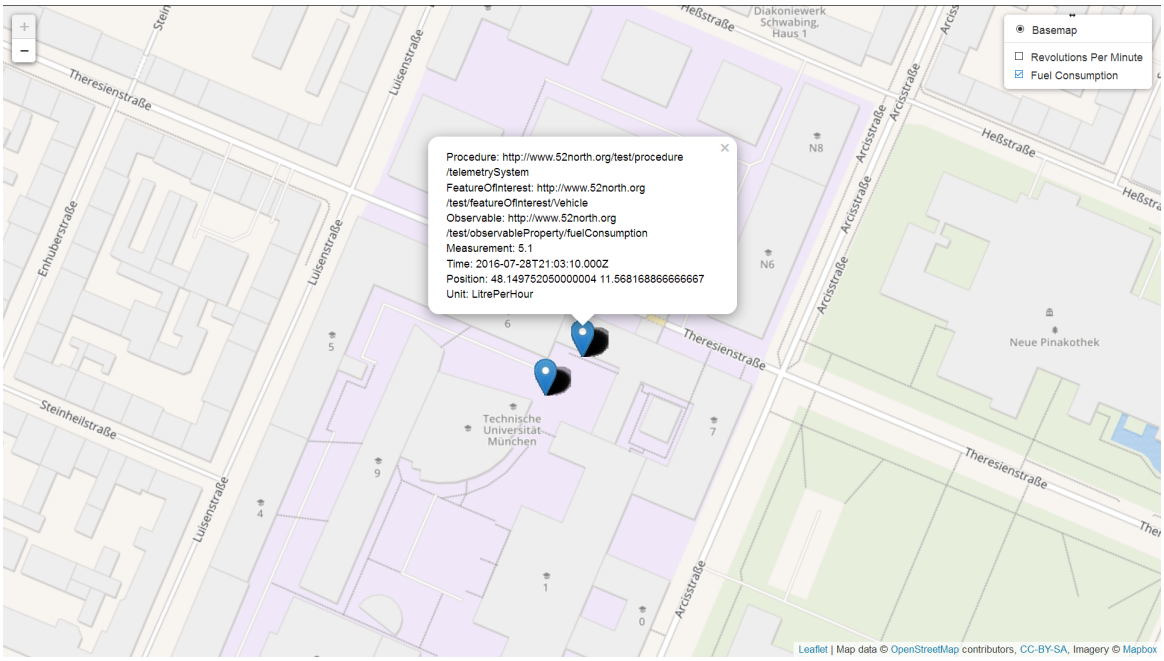


Abbildung 7.4: Von SOS-Instanz abgerufene virtuelle Beobachtung über den Kraftstoffverbrauch

7.4 Einschränkungen des Systems

Es wurde gezeigt, dass sowohl CAN-Daten durch das System erfasst werden, als auch die Verarbeitung der Daten auf Serverseite bis hin zum Abruf dieser aus einer SOS-Instanz funktioniert. Jedoch zeigen sich Einschränkungen des Systems. Es werden Herausforderungen und Schwachstellen aufgezeigt, welche sich bei der Entwicklung und der Nutzung des Systems ergaben. In Abschnitt 8.2 wird auf mögliche Verbesserungsvorschläge für die in den folgenden Abschnitten aufgeführten Gesichtspunkte eingegangen.

7.4.1 Aktualität der Daten

CAN-Daten werden nach Inbetriebnahme des System wesentlich früher als GPS-Daten, mit einer verglichen hohen Messfrequenz erfasst und in einer Textdatei gespeichert. Im Gegensatz dazu benötigt die GPS-GSM-Komponente deutlich länger, um operational tätig zu sein. Die Datenübertragung und Positionsbestimmung sind miteinander gekoppelt, da lediglich eine serielle Schnittstelle vorliegt. Über diese Schnittstelle wird sowohl die Position erfragt als auch die Übertragung initiiert. Es kann somit keine Datenübertragung stattfinden, sobald eine Positionslösung zu bestimmen ist. So vergehen ab der Inbetriebnahme mehr als 40 Sekunden (bei freier Sicht zum Himmel) bis erste NMEA-Datensätze erfasst werden und die Datenübertragung von CAN-Daten beginnen kann. In dieser Zeit werden vom CAN-Modul knapp 1000 erfasste Messungen (in 40 Sekunden liegen jeweils 400 Messungen pro CAN-Datentyp vor) im lokalen Speicher abgelegt. Wenn das GPS-GSM-Modul nun eine Position bestimmt und diese überträgt, ist es noch beschäftigt, Daten aus der Vergangenheit (der letzten 40 Sekunden) Zeile für Zeile zu verschicken. Sobald das Modul eine TCP-Verbindung hergestellt hat, kann es pro Sekunde zwanzig CAN-Messungen (siehe dazu 7.4.2) aus dem lokalen Speicher lesen und versenden. Zur Georeferenzierung auf Serverseite ist es jedoch notwendig, dass Positions- und Fahrzeugdaten keinen großen zeitlichen Abstand aufweisen. Das GPS-GSM-Modul kommt dieser Aufgabe nicht nach, es kann schwerlich zu einer Georeferenzierung auf Serverseite kommen, da zum Zeitpunkt der Übertragung von NMEA-Daten zeitlich veraltete CAN-Daten verschickt werden. Im oben beschriebenen Fall kommen entsprechende Daten mit vierzig Sekunden Verspätung an. Diese Situation wird verschärft, wenn Verbindungsabbrüche auftreten, da dann das GPS-GSM-Modul sich noch mehr im „Rückstand“ befände. Dieses Aktualitätsproblem bezieht sich lediglich auf die initiale zeitliche Abweichung zwischen CAN und GPS. Es ergeben sich also in dieser Hinsicht keine Probleme bei einer späteren Auswertung der Daten (beispielsweise nach einer Laufzeit von einer Stunde), da dann ausreichend Positionsdaten vorliegen, welche zeitlich mit CAN-Daten verknüpft werden können.

7.4.2 Durchsatzrate

Die vorliegende TCP-Übertragungsfunktion nimmt Daten auf und schickt die Daten an den Server. Die Kapazität des Datenstroms, bis zu dem Punkt, an dem die Daten abgeschickt werden müssen, umfasst knapp 1300 Zeichen. Diese Kapazitätsbeschränkung konnte mit dem AT-Befehl `AT+CIPSEND?` abgefragt werden. Dies ist für die Übertragung von NMEA-Datensätzen mehr als ausreichend, welche knapp 100 Zeichen (je nach tatsächlicher Ausprägung) pro Zeile ausmachen, d. h. die Anzahl der ausgelesenen Daten aus dem lokalen Speicher kann bis zu 13 Zeilen pro Sendung betragen. Im Gegensatz dazu werden CAN-Daten mit einer Messfrequenz von 10 Hz pro Sekunde erfasst. Eine so in einer Textdatei gespeicherte Zeile besteht aus knapp 26 Zeichen. Im Rahmen der Arbeit wurden zwei Kategorien (Drehzahl und Kraftstoffverbrauch) aus den CAN-Daten dekodiert. Das wiederum ergibt 260 Zeichen pro Sekunde (zehn Zeilen), für jeden dekodierten Datentyp sind das 520 Zeichen. In dieser Arbeit werden daher max. 20 Zeilen (1020 Zeichen) aus der *CANData.txt* durch das Programm *gpcTcp* ausgelesen. Hinzu kommen 100 Zeichen pro NMEA-Datenzeile, daher werden maximal zwei Zeilen aus der *gpslog.txt*-Textdatei ausgelesen. Dies ergibt insgesamt eine Datenmenge von 1220 Zeichen pro Sendung, dies ist knapp unter der Kapazitätsgrenze. Kämen jedoch zukünftig mehr Daten (beispielsweise durch weitere CAN-Datentypen) hinzu, so kommt es zu einem Engpass im Datenstrom.

8 Fazit und Ausblick

8.1 Fazit

Im Rahmen dieser Arbeit konnte ein Telemetriesystem und serverseitige Software entwickelt werden, wodurch es möglich ist, Prozessdaten im Feld zu erfassen und auf einen SOS zu übertragen und verfügbar zu machen. Der entwickelte Datenlogger wurde aus kostengünstigen, leicht verfügbaren Komponenten zusammengestellt und nutzt Open-Source-basierte sowie selbst entwickelte Programme. Sowohl Hardware als auch Software können jederzeit verändert und erweitert werden (im Rahmen der gegebenen Möglichkeiten des Rechners), um zusätzliche Funktionalität zu realisieren. Das SWE-Framework wurde in der konzeptionellen Ausgestaltung des Telemetriesystems untersucht. Schwerpunkt der Arbeit war die Modellierung von mobilen Sensoren. Die konzeptionelle Untersuchung hat zunächst gezeigt, dass eine Georeferenzierung der Sensorposition bzw. des *feature of interest* im vorliegenden Fall des Telemetriesystems nicht stattfindet, da diese Aspekte nicht von der zukünftigen *Client*-Anwendung *Helgoland* von 52°North für die Modellierung von mobilen Sensoren vorgesehen ist. Stattdessen wird die Georeferenzierung der Prozessdaten durch parametrisierte räumliche Beobachtungen vorgenommen. Aus diesem Resultat heraus wurden O&M-basierte XML-Dokumente erstellt, welche Beobachtungen mittels Parameter georeferenzieren. Diese Beobachtungen konnten erfolgreich einem SOS übergeben werden. In diesem Zusammenhang wurde die Registrierung des Telemetriesystems mit einem SensorML-kodierten Dokument durchgeführt. Nach der Modellierung wurde der *Workflow* des Telemetriesystems zur Datenerfassung und Übertragung auf einen SOS konzipiert und implementiert, was die Grundlage dafür bietet die erfassten Daten vielen verschiedenen Nutzern bereit zu stellen. Die Diskrepanz zwischen der Unterstützung des TCP des GPS-GSM-Moduls und der Unterstützung des HTTP des SOS, wurde durch die Entwicklung einer Zwischenschicht, eines Proxys, überwunden. Zudem wurde untersucht, wie eine Abbildung von SWE auf *Moving Features* und umgekehrt aussehen kann. Dazu wurden die Konzepte der Zustandsänderungsrepräsentation sowie der Variierenden Zustände erarbeitet. Letzteres Konzept bedarf einer Einpflegung in den Standard *Moving Features*, um SWE-Daten in *Moving Features* zu konvertieren. Mittels der Zustandsänderungsrepräsentation jedoch ist es möglich, eine Abbildung von *Moving-Features*-Daten auf das SWE-Framework vorzunehmen und umgekehrt. Es konnte festgestellt werden, dass Sensoren im Sinne des SWE nur bedingt in *Moving Features* abgebildet werden können. Dies kann durch die Attributsannotation in *Moving Features* geschehen. Das erarbeitete Konzept des Telemetriesystem-*Workflow* erläutert die Vorgehensweise und die Teilschritte zur Erfassung von Prozessdaten und zur Distribution dieser durch einen SOS. Es sieht vor, Daten des Datenloggers über eine TCP-Verbindung auf

einen Server zu übertragen, in einer Datenbank zu aggregieren und mittels einer Kennung zu synchronisieren, zu verarbeiten, in *InsertObservation*-Dokumente zu überführen, und schließlich von diesem Server aus diese Daten, mittels HTTP-POST und *InsertObservation*-Operationen, dem SOS zu übergeben. Die im Systemkonzept erläuterten Schritte wurden implementiert. Dazu wurde zunächst die Erfassung von CAN- und Positionsdaten, sowie die lokale, textdateibasierte Speicherung dieser Daten, realisiert. Für die Georeferenzierung wurde ein Schlüssel erarbeitet, welcher aus einer Kombination aus der Anzahl der Systemstarts des Datenloggers und der lokalen Zeit der Datenlogger-Recheneinheit gebildet wird. Damit wurde die Aufgabe der Georeferenzierung der Prozessdaten, welche gemäß den am Anfang formulierten Zielen (siehe in Kapitel 1.3) auf dem Datenlogger stattfinden sollte, auf den Server verlagert und erfüllt. Dieser Schlüssel wurde den Beobachtungen zur eindeutigen Identifizierung zugewiesen. Im Anschluss wurde für den Fall von Ausfällen des Datenloggers und der zusätzlichen lokalen Datenhaltung das zeilenbasierte Auslesen von in Textdateien gespeicherten Daten umgesetzt. Auf Serverseite wurden Programme und ein Datenbankmodul entwickelt, welche die Rohdaten des Datenloggers vom eingehenden Datenstrom des serverseitigen TCP-Socket entgegennehmen und in einer Datenbank ablegen. Auf Datenbankseite wurden SQL-Anweisungen erarbeitet, welche die zur Speicherung der Daten notwendigen Tabellen erzeugen und zeitlich, mithilfe des auf Datenlogger-Seite erzeugten Schlüssel, verknüpfen. Die verknüpften Daten können in einer Datenbanksicht aggregiert werden. Diese georeferenzierten Fahrzeugdaten können weiterverarbeitet werden, um in das für den SOS notwendige O&M-Format überführt zu werden. Die so erzeugten *InsertObservation*-Dokumente können mittels einem entwickelten HTTP-Client dem SOS durch die implementierte Methode HTTP-POST übergeben werden. Die im SOS gespeicherten Daten können so durch Client-Anwendungen abgerufen werden, womit Prozessdaten des Telemetriesystems interoperabel verfügbar sind. Dies wurde durch den Test-Client der SOS-Instanz und eine eigene entwickelte Client-Anwendung überprüft.

An dieser Stelle sollte eingeräumt werden, dass lediglich eine isolierte Prüfung der Ergebnisse durchgeführt werden konnte. Insbesondere die Funktionalität der *SOSFeeder*-Klasse, die darin verwendeten Klassen und damit letztendlich die Übertragung von georeferenzierten Daten, wurde mittels Testdaten (wie sie vom Datenlogger geliefert werden können) überprüft – entsprechende *JUnit*-Tests wurden geschrieben und befinden sich ebenfalls im Eclipse-Projekt des Java-Proxys. Wegen der in den Abschnitten in Kapitel 7.4 genannten Probleme, kam es effektiv zu keiner Datenaggregation durch einen vollständigen Testlauf mittels realer oder virtueller CAN-Daten. Ungeachtet dessen konnten alle in der Einleitung genannten Fragestellungen beantwortet und die aufgestellten Ziele erfüllt werden.

8.2 Ausblick

Nach der Entwicklung des Systems stehen verschiedene weitere Entwicklungsmöglichkeiten offen. Im Rahmen der Arbeit wurde das *InsertSensor*-Dokument, welches zur Registrierung des Telemetriesystems am SOS notwendig ist, serverseitig erstellt und dem SOS übergeben. Im Falle des Einsatzes mehrerer Telemetriesysteme wäre die dynamische Implementierung der *InsertSen-*

sor-Operation vonnöten. So würde es notwendig sein, dass unterschiedliche Telemetriesysteme eindeutige Kennungen (z. B. durch Geräte-, Fahrer-ID) aufweisen und diese bei der Registrierung dem SOS übergeben. Dazu wären entsprechende *InsertSensor*-Dokumente für jeden Datenlogger anzulegen. Alternativ könnte ein am Server befindliches *InsertSensor*-Dokument dynamisch durch das Telemetriesystem geändert werden. In diesem Fall wäre das modifizierte Dokument dem SOS zu übergeben.

Die Voraussetzungen für die Anmeldung mehrerer Telemetriesysteme am Proxy sind gegeben. Das entsprechende Programm wurde mit *Threads* realisiert, somit ist der Proxy ausgelegt, die Anmeldung mehrerer Systeme zu ermöglichen. Falls dies geschieht, müsste der weitere serverseitige *Workflow* auf notwendige Veränderungen überprüft werden. Es wäre dann auszuarbeiten, wie mehrere Telemetriesysteme auf die eingerichtete Datenbank zugreifen. Im vorliegenden Fall greift ein Programm auf alle Daten eines Datenloggers in der Datenbank zu. Bei mehreren Datenloggern müsste erarbeitet werden, wie das auf die Datenbank zugreifende Programm entscheidet, welche Daten welchem Datenlogger zugeordnet sind. Beispielsweise könnte in einem solchen Fall der im Rahmen der Arbeit erarbeitete Schlüssel der Beobachtungen um eine Datenlogger-Kennung erweitert werden. Somit könnten die Beobachtungen zugeordnet und voneinander unterschieden werden.

Weiterhin könnte der Wechsel von dem hier verwendeten auf AT-Befehlen basierendem GSM-Modul zu einer Datenübertragungskomponente, welche eine permanente Internetverbindung ermöglicht, bzw. die Einrichtung einer weiteren seriellen Schnittstelle, diskutiert werden. In Frage käme z. B. ein 3G-Modul bzw. ein Mikrocontroller mit mehreren seriellen Schnittstellen, um die in dieser Arbeit aufgetretenen Einschränkungen aufzuheben. Dies würde die Einschränkungen des Systems, welche im Kapitel 7.4 erläutert wurden, adressieren. Dazu zählen zum einen die Existenz lediglich einer seriellen Schnittstelle, welche nur eine sequentielle Positionsermittlung und anschließende Datenübertragung erlaubt. Dies verhindert eine parallele Ausführung der Positionsermittlung und Datenübertragung und zudem ist dadurch die CAN-Datenübertragung an die Positionsbestimmung gekoppelt. Durch Einführung mehrerer serieller Schnittstellen könnte letztendliche eine kontinuierliche, parallele Positionsermittlung und Datenübertragung ermöglicht werden. Es könnte eine der Messrate von CAN-Daten entsprechende Übertragungsrate erzielt werden. Zum anderen könnten die Einschränkungen in der maximal übertragbaren Datenmenge über TCP aufgehoben werden. Somit könnte eine bessere Durchsatzrate der Daten erzielt werden. Bei einer stabilen, permanenten Datenübertragung wäre es auch denkbar, sämtliche im Rahmen der Arbeit auf dem Proxy durchgeführten Arbeitsschritte direkt auf dem Datenlogger auszuführen. Der Datenlogger könnte somit selbst die erfassten Daten in einer Datenbank abspeichern, weiterverarbeiten, zeitlich verknüpfen, in SOS-konforme Dokumente überführen und diese über HTTP-POST der SOS-Instanz übergeben.

Das Interesse an Sensoren wird, aufgrund der niedrigen Kosten, der hohen Verfügbarkeit und der vielen potentiellen Anwendungsmöglichkeiten, vermutlich weiter steigen. Die Datenerfassung von Sensoren einem möglichst großen Kreis zugänglich zu machen, ist notwendige Voraussetzung, um aus Daten Wissen zu generieren und Entscheidungen zu ermöglichen. Dies wird auch in Standardisierungsarbeiten des OGC ersichtlich: Am 27.07.2016 wurde der *SensorThings*-Standard vom OGC veröffentlicht [Liang et al., 2016]. In diesem Standard wird ein dem

SOS entgegenstehender Ansatz der Datenabfrage verfolgt. Er besteht aus zwei Teilen, dem *Sensing* und dem *Tasking*. Letzterer Teil befindet sich noch in der Ausarbeitungsphase und wird in einem getrennten Dokument veröffentlicht. Ziel dieses Standards ist es ebenfalls den standardisierten Zugang zu Sensordaten zu ermöglichen. Es werden jedoch effiziente, auf die ressourcenbeschränkten Sensoren zugeschnittene Mittel und Methoden bereitgestellt. Diese sollen ermöglichen, die Sensoren selbst standardisiert zu verwalten, nach Beobachtungen abzufragen bzw. zu beauftragen Beobachtungen zu erzeugen, also im Allgemeinen den Zugang zu Sensoren und Daten zu vereinfachen.

In diesem Sinne ist ein Ende von weiteren Forschungsarbeiten, auch in Anbetracht technologischer Neuerungen in der Sensortechnik, auf absehbare Zeit nicht zu erwarten. Die vorliegende Arbeit demonstrierte, dass die Erfassung von Fahrzeugdaten aus der Landwirtschaft und die Bereitstellung dieser Daten durch einen SOS möglich sind. Die hier gemachten Erfahrungen sollen unterschiedliche Nutzer und Anwender ermuntern, auf Basis dieses Systems und der zur Verfügung gestellten Daten, weiterführende und anknüpfende Arbeiten aufzunehmen.

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Programmlistings

Abkürzungsverzeichnis

ACK *Acknowledge*

AGCO *Allis-Gleaner Corporation*

API *Application Programming Interface*

AT ***AT****tention*

Bash *Bourne-again shell*

bps *bits per second*

CAN *Controller Area Network*

CRC *Cyclic Redundancy Check*

CSS3 *Cascading Style Sheets Level 3*

CRC *Cyclic Redundancy Check*

CSV *comma-separated values*

DA *Destination Address*

DLC *Data Length Code*

DP *Data Page*

D-Sub *D-Subminiature*

ECU *Electrical Control Unit*

EDP *Extended Data Page*

EOF *End Of Frame*

GCC *GNU Compiler Collection*

GE *Group Extension*

GIS Geographisches Informationssystem

GMT *Greenwich Meridian Time*

GNSS *Global Navigation Satellite System*

GPIO *General Purpose Input Output*

GPRS *General Packet Radio Service*

GPS *Global Positioning System*

GSM *Global System for Mobile Communications*

GWT *Google Web Toolkit*

HAT *Hardware Attached on Top*

HTML *Hypertext Markup Language*

HTTP *Hytertext Transfer Protocol*

I²C *Inter-Integrated Circuit*

IDE *Identifier Extension*

IP *Internet Protocol*

ISO *International Organization for Standardization*

JS JavaScript

KVP *key-value pair*

LBS Landwirtschaftliches BUS-System

NMEA *National Marine Electronics Association*

NOOBS *New Out Of Box Software*

OGC *Open Geospatial Consortium*

O&M *Observations and Measurements*

OSI *Open Systems Interconnection*

PDU *protocol data unit*

PF *PDU Format*

PGN *Parameter Group Number*

PIN Persönliche Identifikationsnummer

POX *plain old XML*

PS *PDU Specific*

RPM *Revolutions per minute*

RTR *Remote Transmission Request*

SA *Source Address*

SensorML *Sensor Model Language*

SIM *subscriber identity module*

SMA *SubMiniature version A*

SOF *Start Of Frame*

SOS *Sensor Observation Service*

SPI *Serial Peripheral Interface*

SQL *Structured Query Language*

SRR *Substitute Remote Request*

SWE *Sensor Web Enablement*

TCP *Transmission Control Protocol*

UML *Unified Modeling Language*

uom *unit of measurement*

USV Unterbrechungsfreie Stromversorgung

UTC *Universal Time Coordinated*

WLAN *Wireless Local Area Network*

WSS *Web Security Service*

XML *Extensible Markup Language*

XPath XML *Path Language*

Literaturverzeichnis

52°North (2016). *SOS 4.x Documentation*. URL: https://wiki.52north.org/bin/view/SensorWeb/SensorObservationServiceIVDocumentation (Letzter Zugriff am 24. 08. 2016).

Agafonkin, V. (2015). *Documentation - Leaflet - a JavaScript library for interactive maps*. URL: http://www.jdom.org/docs/apidocs/ (Letzter Zugriff am 24. 08. 2016).

AGCO GmbH, Fendt-Marketing (2015a). *Dokumentation & Telemetrie*. Marktoberdorf, Deutschland: AGCO GmbH. URL: http://www.fendt.com/de/11206.asp (Letzter Zugriff am 24. 08. 2016).

— (2015b). *Fendt Variotronic*. Marktoberdorf, Deutschland: AGCO GmbH. URL: http://www.fendt.com/de/assets/article/4168/35642_Variotronic_03-2015_DE_TD.pdf (Letzter Zugriff am 24. 08. 2016).

Agri Con GmbH (2015). *Agri Doc*. Ostrau, Deutschland: Agri Con GmbH. URL: http://www.agricon-shop.de/datenmanagement/agri-doc.html (Letzter Zugriff am 24. 08. 2016).

Asahara, A., H. Hayashi et al. (2015). „International standard “OGC Moving Features” to address “4Vs” on locational BigData“. In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, S. 1958–1966. DOI: 10.1109/BigData.2015.7363975.

Asahara, A. et al. (2015a). *OGC Moving Features Encoding Extension: Simple Comma Separated Values (CSV)*. OGC® Implementation Standard OGC 14-084r2. URL: http://docs.opengeospatial.org/is/14-084r2/14-084r2.html (Letzter Zugriff am 24. 08. 2016).

— (2015b). *OGC Moving Features Encoding Part I: XML Core. OGC Implementation Standard*. OGC® Implementation Standard OGC 14-083r2. URL: http://docs.opengeospatial.org/is/14-083r2/14-083r2.html (Letzter Zugriff am 24. 08. 2016).

Auernhammer, H. (1983). „Die elektronische Schnittstelle Schlepper - Gerät“. In: *Landwirtschaftliches BUS-System - LBS. KTBL-Arbeitspapier* 196.

— (2001). „Precision farming—the environmental challenge“. In: *Computers and electronics in agriculture* 30.1, S. 31–43.

— (2002a). „Automatische Betriebsdatenerfassung im Ackerbau und seine Nutzanwendung“. In: *"'Ackerbau der Zukunft"'*. Landtechnik-Schrift 14. Landtechnik Weihenstephan. Deggendorf, Deutschland: Georg Wendl, S. 45–58.

— (2002b). „Landwirtschaftliches BUS-System (LBS) by DIN 9684/2-5 and ISO 11783 (ISOBUS)“. In: TU München 2009: AgTecCollection: Institut für Landtechnik TUM / Zeichenbüro. URL: http://mediatum.ub.tum.de/?id=733783 (Letzter Zugriff am 24. 01. 2016).

— (2009). „Precision Farming - Concepts, Expectations, Results, Constraints - or “Idea only” versus “Farming of tomorrow” ?“ en. In: *ECPA / EFITA / ECPLF - JIAC Conference Cross-Theme Session „Commonalities of Technological Innovation Adoption“*. Wageningen, Niederlande, S. 25.

Botts, M. und A. Robin (2014). *OGC® SensorML: Model and XML Encoding Standard*. OGC Encoding Standard OGC 12-000. URL: https://portal.opengeospatial.org/files/?artifact_id=55939 (Letzter Zugriff am 24. 08. 2016).

Botts, M. et al. (2008). „OGC® Sensor Web Enablement: Overview and High Level Architecture". In: *GeoSensor Networks: Second International Conference, GSN 2006, Boston, MA, USA, October 1-3, 2006, Revised Selected and Invited Papers*. Hrsg. von S. Nittel, A. Labrinidis und A. Stefanidis. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 175–190. ISBN: 978-3-540-79996-2. DOI: 10.1007/978-3-540-79996-2_10. URL: http://dx.doi.org/10.1007/978-3-540-79996-2_10 (Letzter Zugriff am 24. 08. 2016).

Bredel, H. et al. (2016). *Spatial Observation support for SOS clients*. URL: http://sensorweb.forum.52north.org/Spatial%5C-Observation%5C-support%5C-for%5C-SOS%5C-clients%5C-td4028563.html (Letzter Zugriff am 24. 08. 2016).

Bröring, A., C. Stasch und J. Echterhoff (2012). *OGC Sensor Observation Service Interface Standard*. OpenGIS® Implementation Standard OGC 12-006. URL: https://portal.opengeospatial.org/files/?artifact_id=47599 (Letzter Zugriff am 24. 08. 2016).

Bröring, A. et al. (2011). „New Generation Sensor Web Enablement". In: *Sensors* 11.3, S. 2652–2699. ISSN: 1424-8220. DOI: 10.3390/s110302652.

Cox, S. (2011). *Observations and Measurements - XML Implementation*. OpenGIS® Implementation standard OGC 10-025r1. URL: http://portal.opengeospatial.org/files/?artifact_id=41510 (Letzter Zugriff am 24. 08. 2016).

— (2013). *OGC Abstract Specification Geographic information – Observations and measurements*. OGC® Abstract Specification OGC 10-004r3. URL: http://portal.opengeospatial.org/files/?artifact_id=41579 (Letzter Zugriff am 24. 08. 2016).

CW2. GmbH & Co. KG (2015). *Dokumentation CW2. PiUSV+*. URL: http://www.piusv.de/support/piusvdoku.pdf (Letzter Zugriff am 24. 08. 2016).

Demmel, M. und H. Auernhammer (1998). „Automatisierte Prozessdatenerfassung". In: *LANDTECHNIK – Agricultural Engineering* 53.3, S. 144–145. ISSN: 0023-8082. URL: https://www.landtechnik-online.eu/ojs-2.4.5/index.php/landtechnik/article/view/1998-3-144-145 (Letzter Zugriff am 24. 08. 2016).

DIN (1997). *Landmaschinen und Traktoren - Schnittstellen zur Signalübertragung - Part 1: Punkt-zu-Punkt-Verbindung*. Norm DIN 9684-1:1997-02. Deutsches Institut für Normung.

Echterhoff, J. (2011). *OpenGIS® SWE Service Model Implementation Standard*. OpenGIS® Standard OGC 09-001. URL: http://portal.opengeospatial.org/files/?artifact_id=38476 (Letzter Zugriff am 24. 08. 2016).

ElectroDragon (2013). *SIM908*. URL: http://www.electrodragon.com/w/images/1/14/Sim908.pdf (Letzter Zugriff am 24. 08. 2016).

— (2015). *SIM908_AT Command Manual_V1.02*. URL: http://www.electrodragon.com/w/File:SIM908_AT_Command_Manual_V1.02.pdf (Letzter Zugriff am 24. 08. 2016).

Hollmann, C. (2016). *Display of mobile sensor positions in JS thin client*. URL: http://sensorweb.forum.52north.org/Display%5C-of%5C-mobile%5C-sensor%5C-positions%5C-in%5C-JS%5C-thin%5C-client%5C-td4028357.html#a4028368 (Letzter Zugriff am 24. 08. 2016).

Hunter, J. und B. McLaughlin (2015). *JDOM v2.0.6 API Specification*. URL: http://www.jdom.org/docs/apidocs/ (Letzter Zugriff am 24. 08. 2016).

ISO (2002). *Geographic information – Temporal schema*. Standard ISO 19108:2002. Genf, Schweiz: International Organization for Standardization.

— (2003). *Geographic information – Spatial schema*. Standard ISO 19107:2003. Genf, Schweiz: International Organization for Standardization.

— (2004). *Data elements and interchange formats – Information interchange – Representation of dates and times*. Standard ISO 8601:2004. Genf, Schweiz: International Organization for Standardization.

— (2007). *Tractors and machinery for agriculture and forestry – Serial control and communications data network – Part 1: General standard for mobile data communications*. Standard ISO 11783-1:2007. Genf, Schweiz: International Organization for Standardization.

— (2008). *Geographic information – Schema for moving features*. Standard ISO 19141:2008. Genf, Schweiz: International Organization for Standardization.

— (2012). *Tractors and machinery for agriculture and forestry – Serial control and communications data network – Part 2: Physical layer*. Standard ISO 11783-1:2007. Genf, Schweiz: International Organization for Standardization.

— (2014). *Tractors and machinery for agriculture and forestry – Serial control and communications data network – Part 3: Data link layer*. Standard ISO 11783-3:2014. Genf, Schweiz: International Organization for Standardization.

— (2015a). *Geographic information – Conceptual schema language*. Standard ISO 19103:2015. Genf, Schweiz: International Organization for Standardization.

— (2015b). *Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*. Standard ISO 11898-1:2015. Genf, Schweiz: International Organization for Standardization.

ISO und IEC (1994). *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. Standard ISO/IEC 7498-1:1994. Genf, Schweiz: International Organization for Standardization, International Electrotechnical Commission.

ITU-T (2003). *Serial Asynchronous Automatic Dialling and Control*. Standard Recommendation V.250. International Telecommunication Union.

John Deere GmbH & Co. KG (2016). *AMS Broschüre*. John Deere GmbH & Co. KG. 76646 Bruchsal. URL: https://www.deere.de/de_DE/docs/html/brochures/publication.html?id=016a19e6#38 (Letzter Zugriff am 24. 08. 2016).

Krieger, A. (2015). *Agridoc Die Telematiklösung von Agricon*. Agri Con GmbH. Ostrau, Deutschland. URL: https://issuu.com/hotwar/docs/agridoc-broschuere?e=1351148/12877991 (Letzter Zugriff am 24. 08. 2016).

Landoni, B. (2013). *A GSM/GPRS & GPS Expansion Shield for Raspberry Pi*. URL: http://www.open-electronics.org/a-gsmgprs-gps-expansion-shield-for-raspberry-pi/ (Letzter Zugriff am 24. 08. 2016).

Liang, S., C.-Y. Huang und T. Khalafbeigi (2016). *OGC SensorThings API Part 1: Sensing*. OGC Standard OGC 15-078r6. URL: http://docs.opengeospatial.org/is/15-078r6/15-078r6.html (Letzter Zugriff am 24. 08. 2016).

Libelium (2016a). *Geolocation Tracker (GPRS + GPS) with SIM908 over Arduino and Raspberry Pi*. URL: https://www.cooking-hacks.com/documentation/tutorials/geolocation-tracker-gprs-gps-geoposition-sim908-arduino-raspberry-pi/ (Letzter Zugriff am 24. 08. 2016).

Libelium (2016b). *Raspberry Pi to Arduino Shields Connection Bridge*. URL: https://www.cooking-hacks.com/documentation/tutorials/raspberry-pi-to-arduino-shields-connection-bridge%5C#step3 (Letzter Zugriff am 24. 08. 2016).

Raspberry Pi Foundation (2016). *Raspberry Pi 2 Model B*. URL: https://www.raspberrypi.org/products/raspberry-pi-2-model-b/ (Letzter Zugriff am 24. 08. 2016).

Robert Bosch GmbH (1991). *CAN Specification*. Spezifikation 2.0. Stuttgart, Deutschland: Bosch. URL: http://www.bosch-semiconductors.de/media/ubk_semiconductors/pdf_1/canliteratur/can2spec.pdf (Letzter Zugriff am 24. 08. 2016).

Rothmund, M., M. Demmel und H. Auernhammer (2002). „Nutzung von Informationen aus der automatischen Prozessdatenerfassung". In: *LANDTECHNIK – Agricultural Engineering* 57.3, S. 148–149.

— (2003). „Methods and Services of Data Processing for Data Logged by Automatic Process Data Acquisition Systems". en. In: *XXX CIOSTA – CIGR V Conference*. Turin, Italien, S. 17.

Rothmund, M. und M. Wodok (2010). „ISOBUS - Eine systematische Betrachtung der Norm ISO 11783". In: *Precision Agriculture Reloaded - Informationsgestützte Landwirtschaft. Referate der 30. GIL Jahrestagung, 24.-25. Februar 2010, Stuttgart*, S. 163–166.

SAE (2013). *Serial Control and Communications Heavy Duty Vehicle Network - Top Level Document*. Standard. Society of Automotive Engineers.

Simonis, I. und J. Echterhoff (2011). *OGC Sensor Planning Service Implementation Standard*. OpenGIS® Standard OGC 09-000. URL: http://portal.opengeospatial.org/files/?artifact_id=38478 (Letzter Zugriff am 24. 08. 2016).

SK Pang Electronics Ltd. (2016a). *PiCAN 2 USER GUIDE*. Rev B V1.1. URL: http://skpang.co.uk/catalog/images/raspberrypi/pi_2/PICAN2UGB.pdf (Letzter Zugriff am 24. 08. 2016).

— (2016b). *Software installation for Jessie 2016-05-10 Kernel 4.4*. Jessie 2016-05-10 Kernel 4.4. URL: http://skpang.co.uk/catalog/images/raspberrypi/pi_2/PICAN2_jessie_2016-05-10.pdf (Letzter Zugriff am 24. 08. 2016).

The Apache Software Foundation (2016). *Apache HttpClient 4.5.2 API*. URL: https://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/ (Letzter Zugriff am 24. 08. 2016).

The jQuery Foundation (2016). *jQuery API Documentation*. URL: http://www.jdom.org/docs/apidocs/ (Letzter Zugriff am 24. 08. 2016).

Volkswagen Research (2016). *SocketCAN userspace utilities and tools*. URL: https://github.com/linux-can/can-utils (Letzter Zugriff am 24. 08. 2016).

Vretanos, P. (2010). *OpenGIS Filter Encoding 2.0 Encoding Standard*. OpenGIS® Implementation Standard OGC 09-026r1. URL: https://portal.opengeospatial.org/files/?artifact_id=39968 (Letzter Zugriff am 24. 08. 2016).

Wiesinger, J. (2015). *Der CAN-Bus - Grundlagen von Automobil Bussystemen*. URL: http://www.kfztech.de/kfztechnik/elo/can/can_grundlagen_1.htm (Letzter Zugriff am 24. 08. 2016).

Anhänge

A C++-Programme zur Steuerung des GPS-GSM-Moduls

```
/*
 *  GSMIgnition - program to activate SIM908 module
 */
//Include ArduPi library
#include "arduPi.h"
int resetModulePin = 9;
int onModulePin = 8;
void switchModuleOn() {
  digitalWrite(onModulePin, HIGH);
  delay(2000);
  digitalWrite(onModulePin, LOW);
}
void resetModule() {
  digitalWrite(resetModulePin, HIGH);
  delay(500);
  digitalWrite(resetModulePin, LOW);
  delay(100);
}
int main() {
  Serial.begin(115200);
  delay(2000);
  pinMode(resetModulePin, OUTPUT);
  pinMode(onModulePin, OUTPUT);
  Serial.flush();
  printf("zero\n");
  Serial.print("AT");
  delay(1000);
  if (Serial.available() == 0) {
    printf("one\n");
    resetModule();
    delay(2000);
  }
  Serial.print("AT");
  delay(1000);
  if (Serial.available() == 0) {
    printf("two\n");
    switchModuleOn();
  }
```

```
  return (0);
}
```

Listing A.1: Aktivierung des SIM908-Moduls [Landoni, 2013]

```
/*
 *  GSMOff - program to deactivate SIM908 module
 */
//Include ArduPi library
#include "arduPi.h"
int resetModulePin = 9;
int onModulePin = 8;
void switchModuleOff() {
  digitalWrite(onModulePin, HIGH);
  delay(2000);
  digitalWrite(onModulePin, LOW);
}
void resetModule() {
  digitalWrite(resetModulePin, HIGH);
  delay(500);
  digitalWrite(resetModulePin, LOW);
  delay(100);
}
int main() {
  Serial.begin(115200);
  delay(2000);
  pinMode(resetModulePin, OUTPUT);
  pinMode(onModulePin, OUTPUT);
  Serial.flush();
  printf("zero\n");
  Serial.print("AT");
  delay(1000);
  if (Serial.available() == 0) {
    printf("one\n");
    resetModule();
    delay(2000);
  }
  Serial.print("AT");
  delay(1000);
  // the following line is different from GSMIgnition.cpp source code
  if (Serial.available() > 0) {
    printf("two\n");
    switchModuleOff();
  }
  return (0);
}
```

Listing A.2: Deaktivierung des SIM908-Moduls [Landoni, 2013]

```
/*
 *  GPRS+GPS Quadband Module (SIM908)
 *  Copyright (C) Libelium Comunicaciones Distribuidas S.L.
 *  http://www.libelium.com
 *  This program is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *  You should have received a copy of the GNU General Public License
 *  along with this program.  If not, see http://www.gnu.org/licenses/.
 *  Version:           2.0.1
 *  Design:            David Gascón
 *  Implementation:    Alejandro Gallego & Marcos Martinez
 *  Modification:      Dezhi Fu
 *  This program gets GPS-data from the module and sends the data to
 *  the dedicated server, which is specified in the variables section.
 */
// include libraries
#include <iostream>
#include <sstream>
#include <fstream>
#include <string>
#include <limits>
#include <cstdio>
#include <chrono>
#include "arduPi.h"
using namespace std;
// function declaration
int8_t sendATcommand2(const char* ATcommand, const char* expected_answer1,
    const char* expected_answer2, unsigned int timeout);
int8_t sendATcommand(const char* ATcommand, const char* expected_answer,
    unsigned int timeout);
void power_on();
int8_t start_GPS();
int8_t reset_GPS();
int8_t get_GPS();
int8_t writeGPS2file();
fstream& GotoLine(fstream& file, unsigned int num);
int8_t buildTcp();
int8_t sendData();
int8_t recordCounter(const char *filename, int recordData);
int readCounter(const char *filename);
bool fexists(const char *filename);
int countNumberOfLines(const char *filename);
// variable declaration
int8_t answer;
char aux_str[100];
int counter;
```

```
int systemCounter;
int CANLinePointer;
int writeLinePointer;
int readLinePointer;
long previous;
char gpsRecordHistory[] = "/rbpi/writeLinePointerFile.txt";
char systemCounterLogFile[] = "/rbpi/counterVariable.txt";
char gpsSentHistory[] = "/rbpi/readLinePointerFile.txt";
char canSentHistory[] = "/rbpi/canreadLinePointerFile.txt";
// put mobile service provider information here
char pin[] = "*******";
char apn[] = "internet.eplus.de";
char user_name[] = "eplus";
char password[] = "gprs";
// host ip adress (server dns for this purpose is configured)
char IP_address[] = "gpslog.ddns.net";
// tcp server port
char port[] = "2345";
// NMEA sentence
char Basic_str[100];
double localTime;
char sendString[2048] = "";
// prerequisite steps
void setup() {
  // Is there already gps data saved on the machine?
  if (fexists(gpsRecordHistory) == false) {
    // if not then initiate counter and save it to file
    writeLinePointer = 1;
    recordCounter(gpsRecordHistory, writeLinePointer);
  } else {
    writeLinePointer = readCounter(gpsRecordHistory);
    cout << "NMEA lines written: " << writeLinePointer << endl;
  }
  // Is there already gps data saved and not sent to server?
  if (fexists(gpsSentHistory) == false) {
    // if not then initiate counter and save it to file
    readLinePointer = 1;
    recordCounter(gpsSentHistory, readLinePointer);
  } else {
    readLinePointer = readCounter(gpsSentHistory);
    cout << "NMEA lines read: " << readLinePointer << endl;
  }
  // Is there can data saved?
  if (fexists(canSentHistory) == false) {
    // if not then initiate counter and save it to file
    CANLinePointer = 1;
    recordCounter(canSentHistory, CANLinePointer);
  } else {
    CANLinePointer = readCounter(canSentHistory);
    cout << "CAN lines read: " << CANLinePointer << endl;
  }
  // get system counter variable (for each boot the counter is incremented)
```

```
  systemCounter = readCounter(systemCounterLogFile);
  cout << "System counter:  " << systemCounter << endl;
  // Delete position file when 1000 positions are logged and sent to save SD
      card memory
  if (readLinePointer == 1000 && readLinePointer == writeLinePointer) {
    const int result = remove("/rbpi/gpslog.txt");
    readLinePointer = 0;
    writeLinePointer = 0;
  }
  Serial.begin(115200);
  printf("Starting...\n");
  // enter PIN of SIM card if needed
  //snprintf(aux_str, sizeof(aux_str), "AT+CPIN=%s", pin);
  sendATcommand2("AT+CPIN?", "OK", "ERROR", 2000);
  printf("Connecting to the cellular network...\n");
  // register to network, CREG results are:
  //   0 - unsolicited result code network registration disabled
  //   1 - registered, home network
  //   5 - registered, roaming
  while (sendATcommand2("AT+CREG?", "+CREG: 0,1", "+CREG: 0,5", 1000) == 0)
    ;
  cout << "Connect to server..." << endl;
  buildTcp();
  start_GPS();
}
void loop() {
  // Check if GPS is fixed
  cout << "Checking GPS Status... " << endl;
  if (sendATcommand("AT+CGPSSTATUS?", "2D Fix", 2000)
      || sendATcommand("AT+CGPSSTATUS?", "3D Fix", 2000) == 1) {
    // gps fixed
  }
  //start GPS using warm start
  else {
    cout << "Reset GPS using warm start..." << endl;
    reset_GPS();
  }
  get_GPS();
  writeGPS2file();
  sendData();
}
bool fexists(const char *filename) {
  ifstream ifile(filename);
  return ifile.good();
}
// function to send AT commands using one expected answer
int8_t sendATcommand(const char* ATcommand, const char* expected_answer,
    unsigned int timeout) {
  uint8_t x = 0, answer = 0;
  char response[100];
  unsigned long previous;
  memset(response, '\0', 100);    // Initialize the string
```

```
  delay(100);
  while (Serial.available() > 0)
    Serial.read();    // Clean the input buffer
  Serial.println(ATcommand);    // Send the AT command
  x = 0;
  previous = millis();
  // this loop waits for the answer
  do {
    if (Serial.available() != 0) {
      // if there are data in the UART input buffer, reads it and checks for
          the asnwer
      response[x] = Serial.read();
      printf("%c", response[x]);
      x++;
      // check if the desired answer  is in the response of the module
      if (strstr(response, expected_answer) != NULL) {
        printf("\n");
        answer = 1;
      }
    }
  }
  // Waits for the answer with time out
  while ((answer == 0) && ((millis() - previous) < timeout));
  return answer;
}
// function to send AT commands using two expected answers
int8_t sendATcommand2(const char* ATcommand, const char* expected_answer1,
    const char* expected_answer2, unsigned int timeout) {
  uint8_t x = 0, answer = 0;
  char response[100];
  unsigned long previous;
  memset(response, '\0', 100);    // Initialize the string
  delay(100);
  while (Serial.available() > 0)
    Serial.read();    // Clean the input buffer
  Serial.println(ATcommand);    // Send the AT command
  x = 0;
  previous = millis();
  // this loop waits for the answer
  do {
    // if there are data in the UART input buffer, reads it and checks for
       the asnwer
    if (Serial.available() != 0) {
      response[x] = Serial.read();
      printf("%c", response[x]);
      x++;
      // check if the desired answer 1  is in the response of the module
      if (strstr(response, expected_answer1) != NULL) {
        printf("\n");
        answer = 1;
      }
      // check if the desired answer 2 is in the response of the module
```

```
      else if (strstr(response, expected_answer2) != NULL) {
        printf("\n");
        answer = 2;
      }
    }
  }
  // Waits for the answer with time out
  while ((answer == 0) && ((millis() - previous) < timeout));
  return answer;
}
// function to start GPS module
int8_t start_GPS() {
  unsigned long previous;
  unsigned long timeout = 120000;
  previous = millis();
  // closes the GPS
  sendATcommand("AT+CGPSPWR=0", "OK", 2000);
  // starts the GPS
  sendATcommand("AT+CGPSPWR=1", "OK", 2000);
  cout << "Search for GPS satellites using cold start..." << endl;
  // cold start
  sendATcommand("AT+CGPSRST=0", "OK", 2000);
  // waits for fix GPS
  while ((sendATcommand("AT+CGPSSTATUS?", "2D Fix", 5000)
      || sendATcommand("AT+CGPSSTATUS?", "3D Fix", 5000)) == 0)
    ;
  if ((millis() - previous) < timeout) {
    printf("Time to fix GPS: %lu ms\n", (millis() - previous));
    return 1;
  } else {
    printf("%lu minutes timeout reached.", timeout / 1000 * 60);
    printf("Time to fix GPS: %lu ms\n", (millis() - previous));
    return 0;
  }
}
// reset GPS module using warm start
int8_t reset_GPS() {
  unsigned long previous;
  unsigned long timeout = 120000;
  previous = millis();
  // closes the GPS
  sendATcommand("AT+CGPSPWR=0", "OK", 2000);
  // starts the GPS
  sendATcommand("AT+CGPSPWR=1", "OK", 2000);
  // warm start
  sendATcommand("AT+CGPSRST=1", "OK", 2000);
  // waits for fix GPS
  while ((sendATcommand("AT+CGPSSTATUS?", "2D Fix", 5000)
      || sendATcommand("AT+CGPSSTATUS?", "3D Fix", 5000)) == 0)
    ;
  if ((millis() - previous) < timeout) {
    printf("Time to fix GPS: %lu ms\n", (millis() - previous));
```

```
    return 1;
  } else {
    printf("%lu minutes timeout reached.\n", timeout / 1000 * 60);
    printf("Time to fix GPS: %lu ms\n", (millis() - previous));
    return 0;
  }
}
// function to record counter variable in a specified file
int8_t recordCounter(const char *filename, int recordData) {
  ofstream fout;
  fout.open(filename);
  fout << recordData;
  fout.close();
  return 0;
}
// count number of lines in a file
int countNumberOfLines(const char *filename) {
  int number_of_lines = 0;
  std::string line;
  std::ifstream myfile(filename);
  while (std::getline(myfile, line))
    ++number_of_lines;
  return number_of_lines;
}
// function to read a counter value from file
int readCounter(const char *filename) {
  int counter;
  fstream file(filename);
  GotoLine(file, 1);
  string str;
  file >> str;
  stringstream convert(str);
  if (!(convert >> counter)) {
    counter = 0;
  }
  return counter;
}
// function to write GPS NMEA data to a file
int8_t writeGPS2file() {
  ofstream outfile;
  outfile.open("/rbpi/gpslog.txt", std::ios_base::app);
  outfile << systemCounter << "_" << fixed << localTime;
  outfile << "," << Basic_str << "\n";
  cout << "Position written to line " << writeLinePointer << " in datafile."
      << endl;
  writeLinePointer++;
  recordCounter(gpsRecordHistory, writeLinePointer);
  return 0;
}
// function to send data to a server using TCP
int8_t sendData() {
  memset(sendString, '\0', 2048);    // Initialize the string
```

```
fstream file("/rbpi/gpslog.txt");  // read NMEA data from file
int numberOfLinesInCANFile = countNumberOfLines("/rbpi/CANData.txt");
// if number of can data lines exceeded 500000 delete file to save memory
if (numberOfLinesInCANFile >= 500000) {
  const int result = remove("/rbpi/CANData.txt");
  CANLinePointer = 1;
}
fstream canfile("/rbpi/CANData.txt"); // read CAN data from file
stringstream ss; // establish string stream, push read data to stream
int i = readLinePointer;
int j = CANLinePointer;
if (i < writeLinePointer || j < numberOfLinesInCANFile) {
  ss << systemCounter;
}
while (i < readLinePointer + 2) {
  if (i == writeLinePointer) {
    break;
  }
  GotoLine(file, i);
  string str;
  file >> str;
  ss << str << ";";
  cout << "NMEA Data on line " << i << " in datafile read." << endl;
  i++;
}
while (j < CANLinePointer + 20) {
  if (j == numberOfLinesInCANFile) {
    break;
  }
  GotoLine(canfile, j);
  string str;
  canfile >> str;
  ss << str << ";";
  cout << "CAN data on line " << j << " in datafile read." << endl;
  j++;
}
string s = ss.str(); // build string from string stream
strncpy(sendString, s.c_str(), sizeof(sendString));
cout << "Length of to be sent data: " << s.size() << endl;
cout << "Data to be sent: " << s << endl;
sendATcommand("AT+CIPSEND?", "OK", 1000);
// send data according to length of String variable "s"
sprintf(aux_str, "AT+CIPSEND=%d", strlen(sendString));
cout << '\n' << "Sending data..." << endl;
// Sends GPS data to the TCP socket
if (sendATcommand2(aux_str, ">", "ERROR", 30000) == 1) {
  sendATcommand2(sendString, "SEND OK", "ERROR", 30000);
  cout << endl;
  readLinePointer = i;
  CANLinePointer = j;
  recordCounter(gpsSentHistory, readLinePointer);
  recordCounter(canSentHistory, CANLinePointer);
```

```
  } else { //restore connection calling buildTcp
    cout
        << "Sending data failed. Save the position data until connection is
           restored."
        << endl;
    cout << "Restoring TCP connection..." << endl;
    buildTcp();
  }
  return 0;
}
// function to build TCP socket connection
int8_t buildTcp() {
  long previous;
  previous = millis();
  int delayFactor = 1;
  while ((millis() - previous) < 10000) {
// Select Single-connection mode
    if (sendATcommand2("AT+CIPMUX=0", "OK", "ERROR", 1000 / delayFactor) ==
       1) {
      // Waits for status IP INITIAL
      while (sendATcommand("AT+CIPSTATUS", "INITIAL", 500 / delayFactor) ==
         0)
        ;
      //delay(500);
      // set the AT-command for using GPRS Connection
      snprintf(aux_str, sizeof(aux_str), "AT+CSTT=\"%s\",\"%s\",\"%s\"", apn
          ,
          user_name, password);
      // Sets the APN, user name and password
      if (sendATcommand2(aux_str, "OK", "ERROR", 30000 / delayFactor) == 1)
         {
        // Waits for status IP START
        while (sendATcommand("AT+CIPSTATUS", "START", 500 / delayFactor) ==
           0)
          ;
        //delay(500);
        // Brings Up Wireless Connection
        if (sendATcommand2("AT+CIICR", "OK", "ERROR", 30000 / delayFactor)
            == 1) {
          // Waits for status IP GPRSACT
          while (sendATcommand("AT+CIPSTATUS", "GPRSACT", 500 / delayFactor)
              == 0)
            ;
          //delay(500);
          // Gets Local IP Address
          if (sendATcommand2("AT+CIFSR", ".", "ERROR", 1000 / delayFactor)
              == 1) {
            // Waits for status IP STATUS
            while (sendATcommand2("AT+CIPSTATUS", "IP STATUS", "",
                5000 / delayFactor) == 0)
              ;
            //delay(500);
```

```
            printf("Opening TCP\n");
            snprintf(aux_str, sizeof(aux_str),
                "AT+CIPSTART=\"TCP\",\"%s\",\"%s\"", IP_address, port);
            if (sendATcommand2(aux_str, "CONNECT OK", "CONNECT FAIL",
                5000 / delayFactor) == 1) {
              printf("Connected\n");
              break;
            } else {
              printf("Error opening the connection\n");
              // Closes the socket
              sendATcommand2("AT+CIPCLOSE", "CLOSE OK", "ERROR",
                  1000 / delayFactor);
            }
          } else {
            printf("Error getting the IP address\n");
          }
        } else {
          printf("Error bring up wireless connection\n");
        }
      } else {
        printf("Error setting the APN\n");
      }
    } else {
      printf("Error setting the single connection\n");
      sendATcommand2("AT+CIPSHUT", "OK", "ERROR", 10000 / delayFactor);
    }
  }
}
// function to read line in a file using counter variable
std::fstream& GotoLine(std::fstream& file, unsigned int num) {
  file.seekg(std::ios::beg);
  for (int i = 0; i < num - 1; ++i) {
    file.ignore(std::numeric_limits < std::streamsize > ::max(), '\n');
  }
  return file;
}
// function to get NMEA sentence
int8_t get_GPS() {
  int8_t counter, answer;
  long previous;
  // First get the NMEA string
  // Clean the input buffer
  while (Serial.available() > 0)
    Serial.read();
  // request Basic string
  cout << "Requesting GPS position..." << endl;
  sendATcommand("AT+CGPSINF=0", "AT+CGPSINF=0\r\n\r\n", 1000);
  cout << "GPS Position retrieved." << endl;
  counter = 0;
  answer = 0;
  memset(Basic_str, '\0', 100);    // Initialize the string
  previous = millis();
```

```
  // this loop waits for the NMEA string
  do {
    if (Serial.available() != 0) {
      Basic_str[counter] = Serial.read();
      counter++;
      // check if the desired answer is in the response of the module
      if (strstr(Basic_str, "OK") != NULL) {
        answer = 1;
      }
    }
    // Waits for the answer with time out
  } while ((answer == 0) && ((millis() - previous) < 1000));
  Basic_str[counter - 3] = '\0';
  localTime = std::chrono::duration_cast<std::chrono::duration<double>>(
      std::chrono::system_clock::now().time_since_epoch()).count();
  cout << "Local time timestamp of position: " << fixed << localTime << endl
     ;
  return answer;
}
int main() {
  setup();
  while (1) {
    loop();
  }
  return (0);
}
```

Listing A.3: Positionierung und Datenübertragung durch das SIM908-Modul

B XML-Beispiele

```
<!--============================================================================
Example SOAP Request GetObservation operation which contains filters for procedure, offering
    , and observedProperty. The response contains observations for the observed property,
    procedure and the offering.
============================================================================
--><soap12:Envelope xsi:schemaLocation="http://www.w3.org/2003/05/soap-envelope http://www.
    w3.org/2003/05/soap-envelope/soap-envelope.xsd http://www.opengis.net/sos/2.0 http://
    schemas.opengis.net/sos/2.0/sos.xsd"><soap12:Header><wsa:To>http://www.ogc.org/SOS</
    wsa:To><wsa:Action>http://www.opengis.net/def/serviceOperation/sos/core/2.0/
    GetObservation</wsa:Action><wsa:ReplyTo><wsa:Address>http://www.w3.org/2005/08/
    addressing/anonymous</wsa:Address></wsa:ReplyTo><wsa:MessageID>http://my.client.com/uid/
    msg-0010</wsa:MessageID></soap12:Header><soap12:Body><sos:GetObservation service="SOS"
    version="2.0.0"><!--identifier of a procedure-->
<sos:procedure>http://www.my_namespace.org/sensors/Water_Gage_1</sos:procedure><!--
    identifier of an offering-->
<sos:offering>http://www.my_namespace.org/water_gage_1_observations</sos:offering><!--
    identifier of an observed property-->
<sos:observedProperty>http://sweet.jpl.nasa.gov/2.0/hydroSurface.owl#WaterHeight</
    sos:observedProperty></sos:GetObservation></soap12:Body></soap12:Envelope>
\end{appendices}
```

Listing B.1: *GetObservation*-SOAP-Dokument mit Angaben zu *procedure*, *feature of interest* und *offering* Bröring et al., 2012

```
<?xml version="1.0" encoding="UTF-8"?>
<!--============================================================================
Example Request for GetObservation operation which contains filters for offering,
    observedProperty, procedure and a spatial filter according to Spatial Filtering Profile
    of SOS. The response contains observations for the observed property which are executed
    by the procedure defined in this request and match the spatial filter defined in this
    request.
============================================================================-->
<soap12:Envelope xmlns:soap12="http://www.w3.org/2003/05/soap-envelope" xsi:schemaLocation="
    http://www.w3.org/2003/05/soap-envelope http://www.w3.org/2003/05/soap-envelope/soap-
    envelope.xsd http://www.opengis.net/sos/2.0 http://schemas.opengis.net/sos/2.0/sos.xsd"
    xmlns:sos="http://www.opengis.net/sos/2.0" xmlns:wsa="http://www.w3.org/2005/08/
    addressing" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:swe="http://www.
    opengis.net/swe/2.0" xmlns:swes="http://www.opengis.net/swes/2.0" xmlns:fes="http://www.
    opengis.net/fes/2.0" xmlns:gml="http://www.opengis.net/gml/3.2" xmlns:ogc="http://www.
    opengis.net/ogc" xmlns:om="http://www.opengis.net/om/1.0">
  <soap12:Header>
    <wsa:To>http://www.ogc.org/SOS</wsa:To>
```

```
    <wsa:Action>http://www.opengis.net/def/serviceOperation/sos/core/2.0/GetObservation</
        wsa:Action>
    <wsa:ReplyTo>
      <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
    </wsa:ReplyTo>
    <wsa:MessageID>http://my.client.com/uid/msg-0010</wsa:MessageID>
  </soap12:Header>
  <soap12:Body>
<sos:GetObservation service="SOS" version="2.0.0">
<!--identifier of a procedure-->
  <sos:procedure>http://www.my_namespace.org/sensors/Water_Gage_1</sos:procedure>
  <!--identifier of an offering-->
  <sos:offering>http://www.my_namespace.org/water_gage_1_observations</sos:offering>
  <!--identifier of an observed property-->
  <sos:observedProperty>http://sweet.jpl.nasa.gov/2.0/hydroSurface.owl#WaterHeight</
      sos:observedProperty>
<!--the observations returned shall match the spatial filter defined in this request (the
    spatial property defined in the ValueReference element must be within the passed polygon
    )-->
<sos:spatialFilter>
  <fes:Within>
  <fes:ValueReference>http://www.opengis.net/req/omxml/2.0/data/samplingGeometry</
      fes:ValueReference>
     <gml:Polygon gml:id="Muenster" srsName="http://www.opengis.net/def/crs/EPSG/0/4326">
       <gml:exterior>
         <gml:LinearRing>
           <gml:posList>
             52.90 7.52 52.92 7.51 52.96 7.54 52.90 7.52
           </gml:posList>
         </gml:LinearRing>
       </gml:exterior>
     </gml:Polygon>
  </fes:Within>
</sos:spatialFilter>
</sos:GetObservation>
  </soap12:Body>
</soap12:Envelope>
```

Listing B.2: *GetObservation*-SOAP-Dokument mit räumlichem Filter [Bröring et al., 2012]

```
<!--============================================================================
Example Request for InsertObservation operation which contains the identifier of the
    observation for which the observation shall be inserted as well as the observation which
     shall be inserted. The response contains the assigned observation ID.
=============================================================================
--><soap12:Envelope xsi:schemaLocation="http://www.w3.org/2003/05/soap-envelope http://www.
    w3.org/2003/05/soap-envelope/soap-envelope.xsd http://www.opengis.net/sos/2.0 http://
    schemas.opengis.net/sos/2.0/sos.xsd"><soap12:Header><wsa:To>http://www.ogc.org/SOS</
    wsa:To><wsa:Action>http://www.opengis.net/def/serviceOperation/sos/obsInsertion/2.0/
    InsertObservation</wsa:Action><wsa:ReplyTo><wsa:Address>http://www.w3.org/2005/08/
    addressing/anonymous</wsa:Address></wsa:ReplyTo><wsa:MessageID>http://my.client.com/uid/
    msg-0010</wsa:MessageID></soap12:Header><soap12:Body><sos:InsertObservation service="SOS
    " version="2.0.0"><!--identifier of offering for which the observation shall be inserted
```

```
--><sos:offering>http://www.my_namespace.org/water_gage_2_observations</sos:offering><!--
    observation which shall be inserted-->
<sos:observation><om:OM_Observation gml:id="obsTest1"><om:type xlink:href="http://www.
    opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement"/><om:phenomenonTime><
    gml:TimeInstant gml:id="phenomenonTime"><gml:timePosition>2010-08-31T17:45:15.000+00:00<
    /gml:timePosition></gml:TimeInstant></om:phenomenonTime><om:resultTime xlink:href="#
    phenomenonTime"/><!-- link to DescribeSensor operation of SOS which is providing the
    sensor description
--><om:procedure xlink:href="http://www.my_namespace.org/sensors/Water_Gage_2"/><!--
    parameter containing samplingPoint as defined in SOS 2.0 Extension - Data Encoding
    Restriction
--><om:parameter><om:NamedValue><om:name xlink:href="http://www.opengis.net/req/omxml/2.0/
    data/samplingGeometry"/><om:value><gml:Point gml:id="SamplingPoint2"><gml:pos srsName="
    http://www.opengis.net/def/crs/EPSG/0/4326">54.9 10.52</gml:pos></gml:Point></om:value><
    /om:NamedValue></om:parameter><!-- a notional URN identifying the observed property -->
<om:observedProperty xlink:href="http://sweet.jpl.nasa.gov/2.0/hydroSurface.owl#WaterHeight"
    /><!-- a notional WFS call identifying the object regarding which the observation was
    made
--><om:featureOfInterest xlink:href="http://wfs.example.org?request=getFeature&featureid=
    river1"/><om:result xsi:type="gml:MeasureType" uom="urn:ogc:def:uom:OGC:m">0.28</
    om:result></om:OM_Observation></sos:observation></sos:InsertObservation></soap12:Body></
    soap12:Envelope>
```

Listing B.3: *InsertObservation*-SOAP-Dokument [Bröring et al., 2012]

```
<?xml version="1.0" encoding="UTF-8"?>
<mf:MovingFeatures
  xmlns:mf="http://schemas.opengis.net/mf-core/1.0"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:gml="http://www.opengis.net/gml/3.2"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.opengis.net/mf-core/1.0
  moving_features_gml_app.xsd"
  gml:id="MFC_0002">
    <mf:sTBoundedBy offset="sec">
        <gml:EnvelopeWithTimePeriod srsName="urn:x-ogc:def:crs:EPSG:6.6:4326">
            <gml:lowerCorner>150 0</gml:lowerCorner>
            <gml:upperCorner>250 50</gml:upperCorner>
            <gml:beginPosition>2013-05-01T10:33:41Z
            </gml:beginPosition>
            <gml:endPosition>2013-05-01T10:37:41Z </gml:endPosition>
        </gml:EnvelopeWithTimePeriod>
    </mf:sTBoundedBy>
    <mf:member xlink:href="#pointerToGmlIdOfMovingFeature_a"/>
    <mf:member xlink:href="#pointerToGmlIdOfMovingFeature_b"/>
    <mf:member>
        <mf:MovingFeature gml:id="a">
            <gml:description>Nissan Sentra - License plate ABC 123. Five passengers.</
                gml:description>
            <gml:name>NissanA</gml:name>
        </mf:MovingFeature>
    </mf:member>
```

```
    <mf:member>
        <mf:MovingFeature gml:id="b">
            <gml:description>Nishiki 21 speed racer</gml:description>
            <gml:name>BicycleB</gml:name>
        </mf:MovingFeature>
    </mf:member>
    <mf:header>
        <mf:VaryingAttrDefs>
            <mf:attrDef name="direction" type="xsd:double">
                <mf:AttrAnnotation>
              The direction of the
  vehicle: the attribute value is the angle between the north direction and the
  vehicle direction. Unit of angle is radian.
                For example, North is
  0[rad], East is 1.57[rad] and West is -1.57[rad].
                </mf:AttrAnnotation>
            </mf:attrDef>
        </mf:VaryingAttrDefs>
    </mf:header>
    <mf:foliation>
        <mf:LinearTrajectory gml:id="LT0001" mfIdRef="a" start="0" end="50">
            <gml:posList>161 5 172 5</gml:posList>
            <mf:Attr>0.0</mf:Attr>
        </mf:LinearTrajectory>
        <mf:LinearTrajectory gml:id="LT0002" mfIdRef="b" start="0" end="50">
            <gml:posList>158 20 158 50 159 60</gml:posList>
            <mf:Attr>1.57</mf:Attr>
        </mf:LinearTrajectory>
        <mf:LinearTrajectory gml:id="LT0003" mfIdRef="a" start="50" end="100">
            <gml:posList>172 5 172 1</gml:posList>
            <mf:Attr>-1.57</mf:Attr>
        </mf:LinearTrajectory>
        <mf:LinearTrajectory gml:id="LT0004" mfIdRef="b" start="50" end="100">
            <gml:posList>159 60 166 50</gml:posList>
            <mf:Attr>0</mf:Attr>
        </mf:LinearTrajectory>
    </mf:foliation>
</mf:MovingFeatures>
```

Listing B.4: *Moving Features*-XML-Dokument von Fahrzeugen [Asahara et al., 2015b]

```
<?xml version="1.0" encoding="UTF-8"?>
<sos:InsertObservation service="SOS" version="2.0.0"
  xmlns:sos="http://www.opengis.net/sos/2.0" xmlns:swes="http://www.opengis.net/swes/2.0"
  xmlns:swe="http://www.opengis.net/swe/2.0" xmlns:sml="http://www.opengis.net/sensorML
      /1.0.1"
  xmlns:gml="http://www.opengis.net/gml/3.2" xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:om="http://www.opengis.net/om/2.0" xmlns:sams="http://www.opengis.net/
      samplingSpatial/2.0"
  xmlns:sf="http://www.opengis.net/sampling/2.0" xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengis.net/sos/2.0 http://schemas.opengis.net/sos/2.0/sos.
      xsd http://www.opengis.net/samplingSpatial/2.0 http://schemas.opengis.net/
      samplingSpatial/2.0/spatialSamplingFeature.xsd">
  <!-- multiple offerings are possible -->
  <sos:offering>http://www.52north.org/test/offering/telemetrySystemOffering</sos:offering>
  <sos:observation>
    <om:OM_Observation gml:id="countObservation">
      <gml:identifier codeSpace="">RPMIdentifier
      </gml:identifier>
      <om:type
        xlink:href="http://www.opengis.net/def/observationType/OGC-OM/2.0/
            OM_CountObservation" />
      <om:phenomenonTime>
        <gml:TimeInstant gml:id="phenomenonTime">
          <gml:timePosition>2012-11-19T17:47:15.000+00:00</gml:timePosition>
        </gml:TimeInstant>
      </om:phenomenonTime>
      <om:resultTime xlink:href="#phenomenonTime" />
      <om:procedure xlink:href="http://www.52north.org/test/procedure/telemetrySystem" />
      <om:parameter>
        <om:NamedValue>
          <om:name
            xlink:href="http://www.opengis.net/def/param-name/OGC-OM/2.0/samplingGeometry" />
          <om:value xsi:type="gml:GeometryPropertyType">
            <gml:Point gml:id="RPMPositionID">
              <gml:description>description</gml:description>
              <gml:identifier codeSpace="">ObservationPositionIdentifier
              </gml:identifier>
              <gml:name>hereIam</gml:name>
              <gml:pos srsName="http://www.opengis.net/def/crs/EPSG/0/4326">42.45 11.98</
                  gml:pos>
            </gml:Point>
          </om:value>
        </om:NamedValue>
      </om:parameter>
      <om:parameter>
        <om:NamedValue>
          <om:name xlink:href="NumberOfSatellitesInView" />
          <om:value xsi:type="xs:integer">9</om:value>
        </om:NamedValue>
      </om:parameter>
      <om:observedProperty
```

```
      xlink:href="http://www.52north.org/test/observableProperty/engineRPM" />
    <om:featureOfInterest
      xlink:href="http://www.52north.org/test/featureOfInterest/Vehicle" />
    <om:result xsi:type="xs:integer">2400</om:result>
  </om:OM_Observation>
 </sos:observation>
 <sos:observation>
  <om:OM_Observation gml:id="measurement">
    <gml:identifier codeSpace="">FuelConsumptionIdentifier
    </gml:identifier>
    <om:type
      xlink:href="http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement" />
    <om:phenomenonTime xlink:href="#phenomenonTime" />
    <om:resultTime xlink:href="#phenomenonTime" />
    <om:procedure xlink:href="http://www.52north.org/test/procedure/telemetrySystem" />
    <om:parameter>
      <om:NamedValue>
        <om:name
          xlink:href="http://www.opengis.net/def/param-name/OGC-OM/2.0/samplingGeometry" />
        <om:value xsi:type="gml:GeometryPropertyType">
          <gml:Point gml:id="FuelConsumptionPositionID">
            <gml:description>description</gml:description>
            <gml:identifier codeSpace="">hereIdentifier
            </gml:identifier>
            <gml:name>hereIam</gml:name>
            <gml:pos srsName="http://www.opengis.net/def/crs/EPSG/0/4326">42.45 11.98</
                gml:pos>
          </gml:Point>
        </om:value>
      </om:NamedValue>
    </om:parameter>
    <om:parameter>
      <om:NamedValue>
        <om:name xlink:href="NumberOfSatellitesInView" />
        <om:value xsi:type="xs:integer">9</om:value>
      </om:NamedValue>
    </om:parameter>
    <om:observedProperty
      xlink:href="http://www.52north.org/test/observableProperty/fuelConsumption" />
    <om:featureOfInterest
      xlink:href="http://www.52north.org/test/featureOfInterest/Vehicle" />
    <om:result xsi:type="gml:MeasureType" uom="LitrePerHour">12.24</om:result>
  </om:OM_Observation>
 </sos:observation>
</sos:InsertObservation>
```

Listing B.5: *InsertObservation*-Dokument zur Speicherung der Prozessdaten im SOS

```
<?xml version="1.0" encoding="UTF-8"?>
<swes:InsertSensor service="SOS" version="2.0.0"
  xmlns:swes="http://www.opengis.net/swes/2.0" xmlns:sos="http://www.opengis.net/sos/2.0"
  xmlns:swe="http://www.opengis.net/swe/2.0" xmlns:sml="http://www.opengis.net/sensorml/2.0"
  xmlns:gml="http://www.opengis.net/gml/3.2" xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:gco="http://www.isotc211.org
      /2005/gco"
  xmlns:gmd="http://www.isotc211.org/2005/gmd"
  xsi:schemaLocation="http://www.opengis.net/sos/2.0 http://schemas.opengis.net/sos/2.0/
      sosInsertSensor.xsd http://www.opengis.net/swes/2.0 http://schemas.opengis.net/swes
      /2.0/swes.xsd">
  <swes:procedureDescriptionFormat>http://www.opengis.net/sensorml/2.0</
      swes:procedureDescriptionFormat>
  <swes:procedureDescription>
    <sml:PhysicalSystem gml:id="Telemetrysystem">
      <!-- ================================================ -->
      <!-- System Description -->
      <!-- ================================================ -->
      <gml:description>Telemetrysystem based on Raspberry Pi 2 with
        PICAN2-module and SIM908 GPS/GSM-module
      </gml:description>
      <!--Unique identifier -->
      <gml:identifier codeSpace="uniqueID">http://www.52north.org/test/procedure/
          telemetrySystem</gml:identifier>
      <sml:identification>
        <sml:IdentifierList>
          <sml:identifier>
            <sml:Term definition="urn:ogc:def:identifier:OGC:1.0:longName">
              <sml:label>longName</sml:label>
              <sml:value>Telemetrysystem based on Raspberry Pi 2 with
                PICAN2-module and SIM908 GPS/GSM-module
              </sml:value>
            </sml:Term>
          </sml:identifier>
          <sml:identifier>
            <sml:Term definition="urn:ogc:def:identifier:OGC:1.0:shortName">
              <sml:label>shortName</sml:label>
              <sml:value>Raspberry Pi Telemetrysystem</sml:value>
            </sml:Term>
          </sml:identifier>
        </sml:IdentifierList>
      </sml:identification>
      <sml:capabilities name="offerings">
        <sml:CapabilityList>
          <!-- Special capabilities used to specify offerings. -->
          <!-- Parsed and removed during InsertSensor/UpdateSensorDescription,
            added during DescribeSensor. -->
          <!-- Offering is generated if not specified. -->
          <sml:capability name="offeringID">
            <swe:Text definition="urn:ogc:def:identifier:OGC:offeringID">
              <swe:label>offeringID</swe:label>
```

```
        <swe:value>http://www.52north.org/test/offering/telemetrySystemOffering</
            swe:value>
      </swe:Text>
    </sml:capability>
  </sml:CapabilityList>
</sml:capabilities>
<sml:capabilities name="metadata">
  <sml:CapabilityList>
    <!-- status indicates, whether sensor is insitu (true) or remote (false) -->
    <sml:capability name="insitu">
      <swe:Boolean definition="insitu">
        <swe:value>true</swe:value>
      </swe:Boolean>
    </sml:capability>
    <!-- status indicates, whether sensor is mobile (true) or fixed/stationary
      (false) -->
    <sml:capability name="mobile">
      <swe:Boolean definition="mobile">
        <swe:value>true</swe:value>
      </swe:Boolean>
    </sml:capability>
  </sml:CapabilityList>
</sml:capabilities>
<sml:featuresOfInterest>
  <sml:FeatureList
    definition="http://www.opengis.net/def/featureOfInterest/identifier">
    <swe:label>featuresOfInterest</swe:label>
    <sml:feature
      xlink:href="http://www.52north.org/test/featureOfInterest/Vehicle" />
  </sml:FeatureList>
</sml:featuresOfInterest>
<!-- ================================================ -->
<!-- Inputs = Observed Properties -->
<!-- ================================================ -->
<sml:inputs>
  <sml:InputList>
    <sml:input name="engineRPM">
      <sml:ObservableProperty
        definition="https://en.wikipedia.org/wiki/Revolutions_per_minute" />
    </sml:input>
    <sml:input name="fuelConsumption">
      <sml:ObservableProperty
        definition="https://en.wikipedia.org/wiki/Fuel_economy_in_automobiles" />
    </sml:input>
  </sml:InputList>
</sml:inputs>
<!-- ================================================ -->
<!-- Outputs = Quantities -->
<!-- ================================================ -->
<sml:outputs>
  <sml:OutputList>
    <sml:output name="engineRPM">
```

```
      <swe:Count definition="https://en.wikipedia.org/wiki/Revolutions_per_minute">
      </swe:Count>
    </sml:output>
    <sml:output name="fuelConsumptionLiterPer100Km">
      <swe:Quantity
        definition="https://en.wikipedia.org/wiki/Fuel_economy_in_automobiles">
        <swe:uom code="LitrePerHour" />
      </swe:Quantity>
    </sml:output>
  </sml:OutputList>
</sml:outputs>
<!-- ================================================ -->
<!-- System Components -->
<!-- ================================================ -->
<sml:components>
  <sml:ComponentList>
    <sml:component name="RaspberryPi2ModelB"
      xlink:title="urn:tumgis:components:computingUnit"
      xlink:href="https://www.raspberrypi.org/products/raspberry-pi-2-model-b/" />
    <sml:component name="SIM908"
      xlink:title="urn:tumgis:components:gsmgpsmodule"
      xlink:href="http://www.simcom.eu/media/files/SIM908_Specification_V1106.pdf" />
    <sml:component name="PICAN2"
      xlink:title="urn:tumgis:components:canbusmodule"
      xlink:href="http://skpang.co.uk/catalog/images/raspberrypi/pi_2/PICAN2UGB.pdf" />
    <sml:component name="CW2PiUPS" xlink:title="urn:tumgis:components:ups"
      xlink:href="http://www.piusv.de/" />
  </sml:ComponentList>
</sml:components>
<!-- ================================================ -->
<!-- Connections between components and system output -->
<!-- ================================================ -->
<sml:connections>
  <sml:ConnectionList>
    <!-- connection between can module fuel consumption output and system's
      fuel consumption output -->
    <sml:connection>
      <sml:Link>
        <sml:source ref="components/can/outputs/fuelConsumption" />
        <sml:destination ref="outputs/telemetrysystem/fuelConsumption" />
      </sml:Link>
    </sml:connection>
    <!-- connection between can module rpm output and system's rpm output -->
    <sml:connection>
      <sml:Link>
        <sml:source ref="components/can/outputs/engineRPM" />
        <sml:destination ref="outputs/telemetrysystem/engineRPM" />
      </sml:Link>
    </sml:connection>
  </sml:ConnectionList>
</sml:connections>
<!-- <sml:attachedTo xlink:href="http://www.52north.org/test/procedure/1"
```

```
        xlink:title="http://www.52north.org/test/procedure/1"/> -->
    </sml:PhysicalSystem>
  </swes:procedureDescription>
  <!-- multiple values possible -->
  <swes:observableProperty>http://www.52north.org/test/observableProperty/engineRPM</
      swes:observableProperty>
  <swes:observableProperty>http://www.52north.org/test/observableProperty/fuelConsumption</
      swes:observableProperty>
  <swes:metadata>
    <sos:SosInsertionMetadata>
      <sos:observationType>http://www.opengis.net/def/observationType/OGC-OM/2.0/
          OM_CountObservation</sos:observationType>
      <sos:observationType>http://www.opengis.net/def/observationType/OGC-OM/2.0/
          OM_Measurement</sos:observationType>
      <sos:featureOfInterestType>http://www.opengis.net/def/samplingFeatureType/OGC-OM/2.0/
          SF_SamplingPoint</sos:featureOfInterestType>
    </sos:SosInsertionMetadata>
  </swes:metadata>
</swes:InsertSensor>
```

Listing B.6: *InsertSensor*-Dokument zur Speicherung des Telemetriesystems im SOS

```
<?xml version="1.0" encoding="UTF-8"?>
<sos:GetCapabilities service="SOS"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:sos="http://www.opengis.net/
      sos/2.0"
  xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:swe="http://www.opengis.net/swe/2.0"
  xsi:schemaLocation="http://www.opengis.net/sos/2.0 http://schemas.opengis.net/sos/2.0/
      sosGetCapabilities.xsd">
  <ows:AcceptVersions>
    <ows:Version>2.0.0</ows:Version>
  </ows:AcceptVersions>
  <ows:Sections>
    <ows:Section>Contents</ows:Section>
  </ows:Sections>
</sos:GetCapabilities>
```

Listing B.7: SOS-*GetCapabilities*-Dokument

C Python-Programme

```
from subprocess import Popen, PIPE, STDOUT
import time

""" readDecodeCAN.py is a program which executes the candump command of
the SocketCAN Utility toolbox for reading CAN-based Frames.
The candump command dumps a file of CAN frames into a file.
That file is processed by this Python program.
There are currently two vehicle data identifier frames identified.
Using those and low byte high byte information as well as scaling factors,
we can decode the hexadecimal-based message format.
More CAN metadata can be identified, and implemented using the here chosen
   approach,
if desirable. A conjunction of current local time of the data logger and a
   counter variable,
resembling the number of system boots, is used to uniquely identify the
   measured CAN frame.
The decoded messages are saved in a separate text file, which is processed
   continuously by the
C++ gpsTcp.cpp program in order to transmit the data to the server.

Author: Dezhi Fu

"""
# CAN identifiers
RPMIdentifier = '0CF004F0'
fuelConsumptionIdentifier = '0CFEF2F0'
# Low Byte High Byte
RPMLow = 3
RPMHigh = 4
ConsumptionLow = 0
ConsumptionHigh = 1
# Scaling factors
RPMScalingFactor = 8
ConsumptionScalingFactor = 20
# CounterVariable
global counterVariable
# open counter variable file to assign the value to our global variable
with open("/rbpi/counterVariable.txt","r") as file:
  counterVariable = file.readline()
# functions
def decodeCanMessage(DataField,LowByte,HighByte,ScalingFactor):
  """ decodeCanMessage accepts 4 parameters in order to decode a hexadecimal
      CAN frame data field string into a number.
```

```
  It returns the decoded data value.
  DataField: The hexadecimal string to decode. It is converted using int(
     argument, 16) to an integer.
  LowByte: The position of the hexadecimal string of low data field byte.
  HighByte: The position of the hexadecimal string of high data field byte.
  ScalingFactor: The reciprocal conversion factor we need to apply on the
     measured value.
  """
  DataValueLow = (int("".join(DataField[LowByte]),16))
  DataValueHigh = (int("".join(DataField[HighByte])+'00',16))
  DataValue = (DataValueLow+DataValueHigh)/ScalingFactor
  return DataValue

def buildSentence(identifier,localTimeTimeStamp,DataValue):
  """ buildSentence accepts 3 parameters in order to construct a telemetry
     data sentence.
  Those are the hexadecimal identifier of the CAN frame, the local time
     timestamp, and the measured value.
  It returns a data sentence, which contains unique identifier, data value,
     observed property, and unit.
  identifier: Hexadecimal identifier of CAN frame.
  localTimeTimeStamp: The number of seconds, including after dot values,
     counted from a reference time.
  DataValue: The converted data value from the CAN frame.
  """
  if (identifier == RPMIdentifier):
    return str(counterVariable) + "_" + localTimeTimeStamp+",R," + str(int(
       float(str(DataValue))))
  elif (identifier == fuelConsumptionIdentifier):
    return str(counterVariable) + "_" + localTimeTimeStamp+",F," + str(
       DataValue)
def file_len(fname):
  """ file_len accepts 1 parameter, which represents the path of a file.
  It calculated the number of lines in the file and returns it.
  fname: The path to the file.
  """
  with open(fname) as f:
    return len(f.readlines())
# execute candump to get virtual CANBUS data, comment this line to unselect
   virtual can
cmd = 'candump -t a vcan0'
# comment out following line to choose CAN interface
#cmd = 'candump -t a can0'
# redirect terminal output to subprocess object p
p = Popen(cmd, shell=True, stdin=PIPE, stdout=PIPE, stderr=STDOUT)
while True:
  # read each incoming line
  output = p.stdout.readline()
  # decode string
  output = output.decode("utf-8")
  # strip whitespace
  output = output.strip(" ")
```

```
  # get one CAN Frame and save to list
  ListOfOneCANFrame = [x for x in output.split(' ') if x]
  # extract identifier
  identifier = [x for x in ListOfOneCANFrame if x==RPMIdentifier or x==
     fuelConsumptionIdentifier]
  identifier = ''.join(identifier)
  # extract localtime timestamp
  localTimeTimeStamp = ListOfOneCANFrame[0][1:len(ListOfOneCANFrame[0])-1]
  # get data field of CAN frame
  DataField = ListOfOneCANFrame[4:]
  # decode message according to identifier
  if (identifier == RPMIdentifier):
    DataValue = decodeCanMessage(DataField,RPMLow,RPMHigh,RPMScalingFactor)
    # build data sentence and write to file
    DataSentence = buildSentence(identifier,localTimeTimeStamp,DataValue)
    with open('/rbpi/CANData.txt','a') as file:
      file.write(DataSentence+'\n')
  elif (identifier == fuelConsumptionIdentifier):
    DataValue = decodeCanMessage(DataField,ConsumptionLow,ConsumptionHigh,
       ConsumptionScalingFactor)
    # build data sentence and write to file
    DataSentence = buildSentence(identifier,localTimeTimeStamp,DataValue)
    with open('/rbpi/CANData.txt','a') as file:
      file.write(DataSentence+'\n')
```

Listing C.1: *readDecodeCAN.py*-Python-Programm zur Erfassung, Dekodierung und Speicherung der CAN-Daten

```
import time

""" readDecodeCANFromFile.py is a program which reads a candump dumpfile of
the SocketCAN Utility toolbox for reading CAN-based Frames.
The file is given after using candump command.
That file is processed by this Python program.
There are currently two vehicle data identifier frames identified.
Using those and low byte high byte information as well as scaling factors,
we can decode the hexadecimal-based message format.
More CAN metadata can be identified, and implemented using the here chosen
   approach,
if desirable.
Author: Dezhi Fu
"""
# identifiers
RPMIdentifier = '0CF004F0'
fuelConsumptionIdentifier = '0CFEF2F0'
# Low Byte High Byte
RPMLow = 3
RPMHigh = 4
ConsumptionLow = 0
ConsumptionHigh = 1
# Scaling factor
RPMScalingFactor = 8
ConsumptionScalingFactor = 20
```

```
# functions
def decodeCanMessage(DataField,LowByte,HighByte,ScalingFactor):
  """ decodeCanMessage accepts 4 parameters in order to decode a hexadecimal
      CAN frame data field string into a number.
  It returns the decoded data value.
  DataField: The hexadecimal string to decode. It is converted using int(
     argument, 16) to an integer.
  LowByte: The position of the hexadecimal string of low data field byte.
  HighByte: The position of the hexadecimal string of high data field byte.
  ScalingFactor: The reciprocal conversion factor we need to apply on the
     measured value.
  """
  DataValueLow = (int("".join(DataField[LowByte]),16))
  DataValueHigh = (int("".join(DataField[HighByte])+'00',16))
  DataValue = (DataValueLow+DataValueHigh)/ScalingFactor
  return DataValue

def buildSentence(identifier,localTimeTimeStamp,DataValue):
   """ buildSentence accepts 3 parameters in order to construct a telemetry
       data sentence.
  Those are the hexadecimal identifier of the CAN frame, the local time
     timestamp, and the measured value.
  It returns a data sentence, which contains unique identifier, data value,
     observed property, and unit.
  identifier: Hexadecimal identifier of CAN frame.
  localTimeTimeStamp: The number of seconds, including after dot values,
     counted from a reference time.
  DataValue: The converted data value from the CAN frame.
  """
  if (identifier == RPMIdentifier):
    return str(counterVariable) + "_" + localTimeTimeStamp+",R," + str(int(
       float(str(DataValue))))
  elif (identifier == fuelConsumptionIdentifier):
    return str(counterVariable) + "_" + localTimeTimeStamp+",F," + str(
       DataValue)
def file_len(fname):
   """ file_len accepts 1 parameter, which represents the path of a file.
  It calculates the number of lines in the file and returns it.
  fname: The path to the file.
  """
  with open(fname) as f:
    return len(f.readlines())
def get_file(fname):
   """ get_file accepts 1 parameter, which represents the path of a file.
  It returns the content of a textfile as a list.
  fname: The path to the file.
  """
  with open(fname,"r") as f:
    return f.readlines()
def encrypt(string, length):
   """ encrypt accepts 2 parameters, which represent the to be processed
       string
```

```
  and the number of characters we want to skip over.
  It returns the string with separating whitespaces, which are inserted
     after "length" characters.
  string: The to be processed string.
  length: The number of characters to skip over.
  """
  return ' '.join(string[i:i+length] for i in range(0,len(string),length))
# get all lines from file
output = get_file("/home/jfu/Desktop/candump-2016-07-14_184358.log")
# for each line
for line in output:
  # get one CAN Frame and save to list
  ListOfOneCANFrame = [x for x in line.split(' ') if x]
  # extract identifier
  identifier = ""
  if (ListOfOneCANFrame[2][0:8]==RPMIdentifier or ListOfOneCANFrame[2][0:8]
     == fuelConsumptionIdentifier):
    identifier = ListOfOneCANFrame[2][0:8]
  # extract localtime timestamp
  localTimeTimeStamp = ListOfOneCANFrame[0][1:len(ListOfOneCANFrame[0])-1]
 # print(localTimeTimeStamp)
  # get data field of CAN frame
  DataField = ListOfOneCANFrame[2][9::]
  # encrypt it
  DataField = encrypt(DataField,2)
  # split the result to get twin digits
  DataField = DataField.split(" ")
  # decode message according to identifier
  if (identifier == RPMIdentifier):
    DataValue = decodeCanMessage(DataField,RPMLow,RPMHigh,RPMScalingFactor)
    DataSentence = buildSentence(identifier,localTimeTimeStamp,DataValue)
    with open('/home/jfu/Desktop/CANData.txt','a') as file:
      file.write(DataSentence+'\n')
  elif (identifier == fuelConsumptionIdentifier):
    DataValue = decodeCanMessage(DataField,ConsumptionLow,ConsumptionHigh,
       ConsumptionScalingFactor)
    DataSentence = buildSentence(identifier,localTimeTimeStamp,DataValue)
    with open('/home/jfu/Desktop/CANData.txt','a') as file:
      file.write(DataSentence+'\n')
```

Listing C.2: *readDecodeCANFromFile.py*-Python-Programm zur Dekodierung und Speicherung von CAN-Daten aus einer *candump*-Logdatei

```
# -*- coding: utf-8 -*-
"""
canPlot is a program to visualize CAN data from a text file, which
is produced by the program readDecodeCANFromFile.py

"""
import numpy
import time
from datetime import datetime
from matplotlib import rcParams
import matplotlib.pyplot as plt
# load data
data = numpy.loadtxt("/home/jfu/Desktop/CANData.txt",usecols=(0,2,),
    delimiter=",")
# odd rows are rpm data
rpm = data[::2]
# even rows are fuel consumption data
fuelconsumption = data[1::2]

# setup plot and adjust configuration
# DIN A5 size
figure(figsize=(8.27,5.83))
# math type font
matplotlib.rcParams['mathtext.fontset'] = 'stix'
matplotlib.rcParams['font.family'] = 'STIXGeneral'
# outward-oriented axes ticks
rcParams['xtick.direction'] = 'out'
rcParams['ytick.direction'] = 'out'
# more parameters
params = {
    'axes.labelsize': 14,
    'text.fontsize': 14,
    'xtick.labelsize': 14,
    'ytick.labelsize': 14,
    'text.usetex': False,
    }
# commit parameters
rcParams.update(params)
# no frame
axes(frameon=0)
# light grid
grid()
# generate plot
plt.plot(fuelconsumption[:,0]-fuelconsumption[0,0],fuelconsumption[:,1],
    color='k',linewidth=1, linestyle='-')
plt.xlabel('Vergangene Zeit ab erster Messung [s]')
plt.ylabel('Verbrauchswert [l/h]')
plt.title('Kraftstoffverbrauch')
# save as pdf
savefig('/home/jfu/Desktop/Kraftstoffverbrauch.pdf',#This is simple
    recommendation for publication plots
             dpi=1000,
```

```
            # Plot will occupy a maximum of available space
            bbox_inches='tight')
# again the same stuff for the rpm plot
# setup plot and adjust configuration
# DIN A5 size
figure(figsize=(8.27,5.83))
# math type font
matplotlib.rcParams['mathtext.fontset'] = 'stix'
matplotlib.rcParams['font.family'] = 'STIXGeneral'
# outward-oriented axes ticks
rcParams['xtick.direction'] = 'out'
rcParams['ytick.direction'] = 'out'
# more parameters
params = {
   'axes.labelsize': 14,
   'text.fontsize': 14,
   'xtick.labelsize': 14,
   'ytick.labelsize': 14,
   'text.usetex': False,
   }
# commit parameters
rcParams.update(params)
# no frame
axes(frameon=0)
# light grid
grid()
# generate plot
plt.plot(rpm[:,0]-rpm[0,0],rpm[:,1],color='k',linewidth=1, linestyle='-')
plt.xlabel('Vergangene Zeit ab erster Messung [s]')
plt.ylabel('Drehzahl pro Minute')
plt.title('Motordrehzahl')
savefig('/home/jfu/Desktop/RPM.pdf',#This is simple recommendation for
   publication plots
            dpi=1000,
            # Plot will occupy a maximum of available space
            bbox_inches='tight')
```

Listing C.3: *canPlot.py*-Python-Programm zur Visualisierung von CAN-Daten aus einer von *readDecodeCANFromFile* erzeugten Textdatei

```
global counter
with open("/rbpi/counterVariable.txt","r") as file:
  counter=file.read()
counter = int(counter)+1
with open("/rbpi/counterVariable.txt","w") as file:
  file.write(str(counter))
```

Listing C.4: *startCounter.py*-Python-Programm zur Aufzeichnung eines Zählers, welcher bei jedem Systemstart inkrementiert wird

D Java-Proxy-Quellcode

```
package main;

import classes.SOSFeeder;
import classes.Server;

/**
 * Entry point of data logger program on server side. We wait for incoming
 * socket-based connection attempt on port 2345. Furthermore we initiate the
 * SOSFeeder.
 *
 * @author Dezhi Fu
 */
public class Main {
  public static void main(String[] args) {
    // Initiate thread which deals with submitting georeferenced telemetry
       data
    // to SOS instance
    new SOSFeeder();
    // start Server instance
    new Server(2345);
  }
}
```

Listing D.1: *Main*-Klasse - Einstiegspunkt des Java-Proxys

```
package classes;

import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Enumeration;
import java.util.Hashtable;

/**
 * A class to establish socket-based connections using java.net package.
 *
 * @author Dezhi Fu
 */
public class Server {
  private ServerSocket serverSocket;
  private Hashtable<Socket, DataOutputStream> outputStreams = new Hashtable<
     Socket, DataOutputStream>();
```

```
/**
 * Constructor which calls the listen method on a specified port.
 *
 * @param port
 *          the port on which the server can be reached
 * @author Dezhi Fu
 */

public Server(int port) {

  // We listen to the port for incoming connection requests
  try {
    listen(port);
  } catch (IOException e) {
    e.printStackTrace();
  }
}

/**
 * Using ServerSocket instance this method wait for incoming connection
 * attempts. It returns a Socket instance which can be used to send data
    in
 * and read data out.
 *
 * @param port
 *          the port on which the Server instance listens to incoming
 *          connection attempts
 * @author Dezhi Fu
 */
private void listen(int port) throws IOException {
  serverSocket = new ServerSocket(port);
  System.out.println("Listen to port: " + serverSocket.getLocalPort());
  // Listen to the port and accept incoming connection requests
  while (true) {
    // return a Socket instance, if connection was established
    Socket s = serverSocket.accept();
    System.out.println("Connection established to socket " + s);
    // Generate data stream to send data to client (possible feature)
    DataOutputStream dout = new DataOutputStream(s.getOutputStream());
    outputStreams.put(s, dout);
    // Pass the socket to a thread which deals with data saving to
       database
    new ServerThread(this, s);
  }
}

/**
 *
 * Sends a message to all connected clients.
 *
 * @param message
```

```
     *          The message as A String which is to be sent to the clients.
     */

    void sendToAll(String message) {
      // synchronuous message sending
      synchronized (outputStreams) {
        // For each client
        for (Enumeration<DataOutputStream> e = getOutputStreams(); e.
            hasMoreElements();) {
          // select communication channel to client
          DataOutputStream dout = (DataOutputStream) e.nextElement();
          // send message
          try {
            dout.writeUTF(message);
          } catch (IOException ie) {
            ie.printStackTrace();
          }
        }
      }
    }

    /**
     *
     * Removes client connection.
     *
     * @param s
     *          The TCP socket of the client
     */
    void removeConnection(Socket s) {
      synchronized (outputStreams) {
        System.out.println("Remove connection from socket " + s);
        // remove socket
        outputStreams.remove(s);
        // close connection
        try {
          s.close();
        } catch (IOException ie) {
          System.out.println("Error while closing socket " + s);
          ie.printStackTrace();
        }
      }
    }

    // Getter
    Enumeration<DataOutputStream> getOutputStreams() {
      return outputStreams.elements();
    }

}
```

Listing D.2: *Server*-Klasse zur Herstellung einer auf Sockets basierenden Verbindung

```
package classes;

import java.util.List;
import org.jdom2.Document;

/**
 * A class that handles fetching data from database view to submit to SOS
 * instance using threads.
 *
 * @author Dezhi Fu
 */
public class SOSFeeder implements Runnable {

  /**
   * Constructor which starts the thread.
   *
   */
  public SOSFeeder() {
    Thread thread = new Thread(this, "New SOSFeeder thread");
//    thread.start();
    System.out.println(thread.getName() + " started");
  }

  /**
   * Fetch the data and submit to SOS instance.
   *
   */

  @Override
  public void run() {
    submitGeoreferencedTelemetryData();
  }

  /**
   * Telemetry data are fetched using the getVehicleData()-method to further
   * pass them to XML-documents, which are encoded correspondingly to
   * O&M-Format. Finally those documents are submitted to SOS using HTTP-
      POST.
   *
   */

  public void submitGeoreferencedTelemetryData() {
    // Prepare the SOSClient instance in order to execute HTTP POST
    SOSClient sos = new SOSClient();
    // Prepare database connection
    DataSourceConnector dbc = new DataSourceConnector(
        DataSourceConnectorFactory.getPostgresqlDataSource());
    dbc.createTables();
    dbc.createView();
    while (true) {
      // fetch vehicle data
      List<Vehicle> vehicleData = dbc.getVehicleData();
```

```
      System.out.println(vehicleData.size());
      // if there is any data
      if (vehicleData.size() != 0) {
        for (Vehicle v : vehicleData) {
          // parse to O&M
          XMLParser parser = new XMLParser();
          Document observationsAndMeasurementsDocument = parser.readXML("/
              resources/InsertObservation.xml").marshal(v);
          // submit to SOS instance
          sos.InsertObservation(observationsAndMeasurementsDocument);
        }
      }
    }
  }
}
```

Listing D.3: *SOSFeeder*-Klasse zur *Thread*-basierten Abfrage und Übertragung der georefenzierten Prozessdaten

```
package classes;

import java.io.DataInputStream;
import java.io.EOFException;
import java.io.IOException;
import java.net.Socket;
import java.net.SocketTimeoutException;

/**
 * * A class that handles incoming data of the data logger using threads.
 *
 * @author Dezhi Fu
 */
public class ServerThread implements Runnable {
  Server server;
  Socket socket;

  /**
   * Constructor which starts the thread and accepts a Server instance as
      well
   * as a Socket instance. The first object is used for reading the data the
   * last object is used for terminating the connection the second for
      reading
   * the data.
   */

  public ServerThread(Server server, Socket socket) {
    this.server = server;
    this.socket = socket;
    // start thread
    Thread thread = new Thread(this, "New server thread");
    thread.start();
    System.out.println(thread.getName());
  }
```

```
/**
 * Process the data and remove the connection, when no further incoming
    data.
 */
@Override
public void run() {
  try {
    // process incoming data stream.
    processData(new DataInputStream(socket.getInputStream()));
  } catch (EOFException ie) {
    ie.printStackTrace();
  } catch (IOException ie) {
    ie.printStackTrace();
  } finally {
    // Connection will be closed
    server.removeConnection(socket);
  }
}

/**
 * Incoming data from socket is processed and delivered to SOS using this
 * method. Each incoming data sentence, i.e. semicolon separated values is
 * passed to DataSourceConnector instance to save the values in a
    postgresql
 * database.
 *
 * @param din
 *          the input datastream as a DataInputStream instance, which is
    used
 *          to read the incoming data.
 */

private void processData(DataInputStream din) {
  // Prepare database connection and database structure
  DataSourceConnector dbc = new DataSourceConnector(
     DataSourceConnectorFactory.getPostgresqlDataSource());
  dbc.createTables();
  dbc.createView();
  // Read data from socket input stream
  try {
    while (din.read() >= 0) {
      // allocate space for data input
      byte[] vehicleDataString = new byte[2048];
      // read the data
      try {
        din.read(vehicleDataString);
        // if timeout reached
      } catch (SocketTimeoutException e) {
        System.out.println("No incoming data from client. Closing socket
           ...");
        break;
```

```
        } catch (IOException ioe) {
          ioe.printStackTrace();
        }
        // convert input from telemetry system to String
        String data = new String(vehicleDataString);
        System.out.println(data);
        // insert values into telemetrySystemDatabase
        dbc.insertValues(data);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Listing D.4: *ServerThread*-Klasse zur *Thread*-basierten Verarbeitung der eingehenden Fahrzeugdaten

```
package classes;

/**
 * A class to save the position and process data of a data logger in a
    vehicle.
 * A corresponding instance represents the position with a timestamp and
 * corresponding process data (e.g. fuel consumption).
 *
 * @author Dezhi Fu
 */
public class Vehicle {
  private double longitude;
  private double latitude;
  private String timestamp;
  private int numberOfSatellitesTracked;
  private int engineRPM;
  private double fuelConsumption;

  public Vehicle(double longitude, double latitude, String timestamp, int
     numberOfSatellitesInView, int rpmValue,
      double fuelConsumptionValue) {
    setLongitude(convert2DecimalDegree(longitude));
    setLatitude(convert2DecimalDegree(latitude));
    setTimestamp(convertTimeStamp(timestamp));
    setNumberOfSatellitesViewed(numberOfSatellitesInView);
    setEngineRPM(rpmValue);
    setFuelConsumption(fuelConsumptionValue);
  }

  public Vehicle() {
  }

  /**
   * Performs the conversion of a coordinate from degree decimal minutes to
   * decimal degrees, i.e. from DDMM.mmm notation to DD.dddddd notation.
   *
```

```
   * @param coordinate
   *          the input coordinate as a String, which is in degree decimal
   *          minutes, to convert to.
   * @return the coordinate as a double type in decimal degrees notation
   */
  private double convert2DecimalDegree(Double coordinate) {
    double deg = 0;
    double min = 0;
    deg = coordinate;
    // the two digits before and all digits after the dot correspond to
       decimal
    // minutes
    min += deg % 100;
    // the other digits before the dot correspond to our degrees
    deg = deg / 100;
    // convert minutes to decimal degrees and add that to the whole degrees
    deg = (int) deg + min / 60;
    return deg;
  }

  /**
   * Performs the conversion of a timestamp of the data logger to date time
   * expression as specified in ISO 8601
   *
   * @param dateTimeString
   *          the input timestamp as a String, which is in the data logger's
   *          format, to convert to ISO 8601 notation.
   * @return the timestamp as a String, which is in ISO 8601 notation,
   *         containing separation characters and time zone designator
   */

  private String convertTimeStamp(String dateTimeString) {
    String date = dateTimeString.substring(0, 8);
    String timeStamp = dateTimeString.substring(8);
    date = date.substring(0, 4) + "-" + insertSymbol(date.substring(4), "-")
       ;
    String subTime = timeStamp.split("\\.")[1];
    String time = insertSymbol(timeStamp.split("\\.")[0], ":");
    return date + "T" + time + "." + subTime + "+00:00";
  }

  /**
   * Method to insert a user-defined character in a String
   *
   * @param data
   *          the String in which a character has to be modified
   * @param symbol
   *          the character which we want to insert into the String
   * @return the new String with characters inserted
   */
  private String insertSymbol(String data, String symbol) {
    return data.replaceAll("..(?!$)", "$0" + symbol);
```

```
}

public boolean hasSameData(Vehicle vehicle) {
  return this.longitude == vehicle.getLongitude() && this.latitude ==
      vehicle.getLatitude()
       && this.timestamp.equals(vehicle.getTimestamp())
       && this.numberOfSatellitesTracked == vehicle.
          getNumberOfSatellitesTracked()
       && this.engineRPM == vehicle.getEngineRPM() && this.fuelConsumption
          == vehicle.getFuelConsumption();
}

// Getters and Setters
public double getLatitude() {
  return latitude;
}

public void setLatitude(double latitude) {
  this.latitude = latitude;
}

public double getLongitude() {
  return longitude;
}

public void setLongitude(double longitude) {
  this.longitude = longitude;
}

public String getTimestamp() {
  return timestamp;
}

public void setTimestamp(String timestamp) {
  this.timestamp = timestamp;
}

public int getNumberOfSatellitesTracked() {
  return numberOfSatellitesTracked;
}

public void setNumberOfSatellitesViewed(int numberOfSatellitesTracked) {
  this.numberOfSatellitesTracked = numberOfSatellitesTracked;
}

public double getFuelConsumption() {
  return fuelConsumption;
}

public void setFuelConsumption(double fuelConsumption) {
  this.fuelConsumption = fuelConsumption;
}
```

```
  public int getEngineRPM() {
    return engineRPM;
  }

  public void setEngineRPM(int engineRPM) {
    this.engineRPM = engineRPM;
  }
}
```

Listing D.5: *Vehicle*-Klasse zur Verarbeitung und Zwischenspeicherung der Rohdaten

```
package classes;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.StringReader;
import java.util.List;

import org.jdom2.Attribute;
import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.JDOMException;
import org.jdom2.Namespace;
import org.jdom2.filter.Filters;
import org.jdom2.input.SAXBuilder;
import org.jdom2.output.Format;
import org.jdom2.output.XMLOutputter;
import org.jdom2.xpath.XPathExpression;
import org.jdom2.xpath.XPathFactory;

/**
 * This class is using JDOM in order to pass the instance variables values
    of a
 * Vehicle instance to O&M-Documents. It further deals with XML processing.
 *
 * @author Dezhi Fu
 */

public class XMLParser {

  private Document xmlDocument;
  private Element root;
  private Element offering;
  private Element observation;
  private String xmlPath;
  private Vehicle vehicleData;
  private Namespace sos;
  private Namespace gml;
  private Namespace om;
  private Namespace xlink;
```

```
private Namespace xs;
private Namespace xsi;
private Namespace swes;
private String identifier;

/**
 * Constructor which initializes Namespaces needed for O&M, SensorML
 * processing.
 */

public XMLParser() {
  sos = Namespace.getNamespace("sos", "http://www.opengis.net/sos/2.0");
  gml = Namespace.getNamespace("gml", "http://www.opengis.net/gml/3.2");
  om = Namespace.getNamespace("om", "http://www.opengis.net/om/2.0");
  xlink = Namespace.getNamespace("xlink", "http://www.w3.org/1999/xlink");
  xs = Namespace.getNamespace("xs", "http://www.w3.org/2001/XMLSchema");
  xsi = Namespace.getNamespace("xsi", "http://www.w3.org/2001/XMLSchema-
     instance");
  swes = Namespace.getNamespace("swes", "http://www.opengis.net/swes/2.0")
     ;
}

/**
 * This method accepts the path to a xml file and generates from the xml
 * document a JDOM-based tree. The document is saved as an instance
    variable
 * of the parser instance, you can get it using the getter method
 * getXMLDocument()
 *
 * @param xmlPath
 *          the path to the to be processed XML file
 * @return the instance itself
 */

public XMLParser readXML(String xmlPath) {
  // set the path and generate file instance, which represents the XML
  // document
  setXMLPath(xmlPath);
  File xmlFile = new File(getClass().getResource(getXMLPath()).getPath());
  // Prepare construction of JDOM tree
  SAXBuilder builder = new SAXBuilder();

  try {
    // returns XML Document as a JDOM tree
    xmlDocument = (Document) builder.build(xmlFile);
    // set the root element and the tree
    setRoot(xmlDocument.getRootElement());
    setXmlDocument(xmlDocument);
  } catch (IOException io) {
    System.out.println(io.getMessage());
  } catch (JDOMException jdomex) {
    System.out.println(jdomex.getMessage());
```

```
    }
    return this;
  }

  /**
   * This method accepts String representation of a XML document and returns
       the
   * JDOM-based representation of that document.
   *
   * @param xmlDocumentString
   *          the String representation of a XML document.
   * @return the JDOM based representation of that document
   */

  public Document buildDOM(String xmlDocumentString) {
    SAXBuilder builder = new SAXBuilder();
    try {
      return (Document) builder.build(new StringReader(xmlDocumentString));
    } catch (JDOMException | IOException e) {
      e.printStackTrace();
    }
    return null;
  }

  /**
   * This method accepts String representation of a XML document and returns
       the
   * file representation, i.e. the actual document in XML format.
   *
   * @param xmlString
   *          the String representation of a XML document.
   * @param outputPath
   *          the output path to which the file is written
   */

  public void writeXML(String xmlString, String outputPath) {

    XMLOutputter xmlOutputter = new XMLOutputter(Format.getPrettyFormat());

    try {
      xmlOutputter.output(buildDOM(xmlString), new FileOutputStream(
          outputPath));
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  /**
   * After reading a O&M document and setting the root element of that
      document,
```

```
 * this method accepts Vehicle objects and parse the values of the
    instance
 * variables to O&M documents.
 *
 * @param vehicle
 *          the vehicle instance which contains process data to parse to O
    &M
 *          files.
 * @return the O&M document which contains process data
 */

public Document marshal(Vehicle vehicle) {
  setVehicleData(vehicle);
  // identifier for the observations
  setIdentifier(getVehicleData().getTimestamp().replaceAll("[^0-9]", ""));
  // get offerings and observations in the O&M document
  List<Element> observations = root.getChildren();

  // skip first node (which is offering) for the other nodes, which are
     the
  // observations, insert observation properties
  for (Element observation : observations) {
    if (observation == observations.get(0))
      continue;
    insertObservationID(observation);
    insertObservationTime(observation);
    insertObservationParameters(observation);
    insertObservationResult(observation);
  }
  return xmlDocument;
}

/**
 * This method accepts O&M elements and modifies the observationID using
    the
 * instance variable identifier.
 *
 * @param observation
 *          the O&M observation element which we want to modify
 */

private void insertObservationID(Element observation) {
  // get the ID of the observation and attach a unique identifier to it
  Element observationID = observation.getChild("OM_Observation", om).
     getChild("identifier", gml);
  observationID.setText(observationID.getText() + getIdentifier());
}

/**
 * This method accepts O&M elements and modifies the phenomenon time.
 *
 * @param observation
```

```
     *          the O&M phenomenon time element which we want to modify
     */

    private void insertObservationTime(Element observation) {
      // if there is a concrete time given, i.e. if there is a corresponding
      // child, modify it accordingly
      if (!observation.getChild("OM_Observation", om).getChild("phenomenonTime
          ", om).getChildren().isEmpty()) {
        Element phenomenonTime = observation.getChild("OM_Observation", om).
            getChild("phenomenonTime", om);
        // set the time of the element according to timestamp of vehicle
            instance
        phenomenonTime.getChild("TimeInstant", gml).getChild("timePosition",
            gml)
            .setText(getVehicleData().getTimestamp());
      }
    }

    /**
     * This method accepts O&M elements and modifies all parameters. In this
     * implementation if you add new parameters to the InsertObservation
        document
     * you have to extend the switch-case-construct
     *
     * @param observation
     *          the O&M element which we want to modify
     */

    private void insertObservationParameters(Element observation) {
      // pick all parameters including position and quality parameters
      List<Element> observationParameters = observation.getChild("
          OM_Observation", om).getChildren("parameter", om);
      for (Element observationParameter : observationParameters) {
        Attribute parameterName = observationParameter.getChild("NamedValue",
            om).getChild("name", om)
            .getAttribute("href", xlink);
        switch (parameterName.getValue()) {
        case "NumberOfSatellitesInView":
          observationParameter.getChild("NamedValue", om).getChild("value", om
              )
              .setText(String.valueOf(vehicleData.getNumberOfSatellitesTracked
                  ()));
          break;
        case "http://www.opengis.net/def/param-name/OGC-OM/2.0/
            samplingGeometry":
          observationParameter.getChild("NamedValue", om).getChild("value", om
              ).getChild("Point", gml)
              .getChild("pos", gml).setText(vehicleData.getLatitude() + " " +
                  vehicleData.getLongitude());
          break;
        }
      }
```

```
}

/**
 * This method accepts O&M elements and modifies the observation result.
 *
 * @param observation
 *          the O&M result element which we want to modify
 */
private void insertObservationResult(Element observation) {
  Attribute observedPropertyXLink = observation.getChild("OM_Observation",
      om).getChild("observedProperty", om)
      .getAttribute("href", xlink);
  Element observationResult = observation.getChild("OM_Observation", om).
     getChild("result", om);
  // set the observation results
  if (observedPropertyXLink.getValue().contains("engineRPM")) {
    observationResult.setText(String.valueOf(vehicleData.getEngineRPM()));
  } else if (observedPropertyXLink.getValue().contains("fuelConsumption"))
      {
    observationResult.setText(String.valueOf(vehicleData.
       getFuelConsumption()));
  }
}

/**
 * This method uses XPath to find a element in a JDOM-based representation
     of
 * a XML document.
 *
 * @param xPath
 *          XPath expression to search for an element
 * @param ns
 *          Namespace which is associated with the elements
 * @param doc
 *          XML document which contains the desired elements
 * @return A list containing the found elements
 */

public List<Element> getElementsByXpath(String xPath, Namespace ns,
   Document doc) {
  // use the default implementation
  XPathFactory xFactory = XPathFactory.instance();
  // select all links
  XPathExpression<Element> expr = xFactory.compile(xPath, Filters.element
     (), null, ns);
  List<Element> searchedElements = expr.evaluate(doc);
  return searchedElements;
}

// Getter, Setter
/**
```

```
 * This method returns the Document representation of the XML document of
    the
 * current parser instance.
 */

public Document getXmlDocument() {
  return xmlDocument;
}

public void setXmlDocument(Document xmlDocument) {
  this.xmlDocument = xmlDocument;
}

public Element getOffering() {
  return offering;
}

public void setOffering(Element offering) {
  this.offering = offering;
}

public Element getObservation() {
  return observation;
}

public void setObservation(Element observation) {
  this.observation = observation;
}

public String getXMLPath() {
  return xmlPath;
}

public void setXMLPath(String xmlPath) {
  this.xmlPath = xmlPath;
}

public Element getRoot() {
  return root;
}

public void setRoot(Element root) {
  this.root = root;
}

public Vehicle getVehicleData() {
  return vehicleData;
}

public void setVehicleData(Vehicle vehicle) {
  this.vehicleData = vehicle;
}
```

```
  public Namespace getSwes() {
    return swes;
  }

  public void setSwes(Namespace swes) {
    this.swes = swes;
  }

  public Namespace getSos() {
    return sos;
  }

  public void setSos(Namespace sos) {
    this.sos = sos;
  }

  public String getIdentifier() {
    return identifier;
  }

  public void setIdentifier(String identifier) {
    this.identifier = identifier;
  }
}
```

Listing D.6: *XMLParser*-Klasse Verarbeitung von XML-Dokumenten

```
package classes;

import java.io.IOException;
import java.util.List;

import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.ContentType;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.util.EntityUtils;
import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.output.XMLOutputter;

/**
 * A class to submit O&M as well as SensorML documents to an SOS instance.
 *
 * @author Dezhi Fu
 */

public class SOSClient {
  private String GetCapabilitiesResponse;
```

```
private String getObservationResponse;
// save the GetObservation response to Tomcat webapps directory to feed
   the JS
// client
private static final String responseStandardOutputPath = "C:\\Program
   Files\\Apache Software Foundation\\Tomcat 9.0\\webapps\\MySOSClient\\
   GetObservationResponse.xml";
private static final String sosInstanceAddress = "http://localhost:8080/52
   n-sos-webapp/service";

/**
 * Constructor which inserts the telemetry system SensorML description if
    the
 * GetCapabilities response does not indicate any telemetry system
 * registration.
 */
public SOSClient() {
  if (!observationProcedureRegistered(GetCapabilities()))
    InsertSensor();
}

/**
 * This method applies InsertObservation operation of SOS standard using a
 * static dynamically modified O&M document and HTTP POST.
 *
 * @param observationsAndMeasurementsDocument
 *          a JDOM-based representation of the O&M document.
 * @return the instance itself
 */

public SOSClient InsertObservation(Document
   observationsAndMeasurementsDocument) {
  postDocument(observationsAndMeasurementsDocument);
  return this;
}

/**
 * This method applies GetCapabilities operation of SOS standard using a
 * static dynamically modified O&M document and HTTP POST. It further
    saves
 * the response to an instance variable and returns it.
 */

public String GetCapabilities() {
  XMLParser parser = new XMLParser();
  parser.readXML("/resources/GetCapabilities.xml");
  setGetCapabilitiesResponse(postDocument(parser.getXmlDocument()));
  return getGetCapabilitiesResponse();
}

/**
```

```
 * This method applies GetObservation operation of SOS standard using a
    static
 * dynamically modified O&M document and HTTP POST. It further saves the
 * response to an instance variable.
 *
 * @return the instance itself
 */

public SOSClient GetObservation() {
  XMLParser parser = new XMLParser();
  parser.readXML("/resources/GetObservation.xml");
  setGetObservationResponse(postDocument(parser.getXmlDocument()));
  return this;
}

/**
 * This method saves the SOS response and writes it to a XML document.
 *
 * @return the instance itself
 */

public SOSClient saveResponse(String serverResponse, String outputPath) {
  XMLParser parser = new XMLParser();
  parser.writeXML(serverResponse, outputPath);
  return this;
}

/**
 * This method applies InsertSensor operation of SOS standard using a
    static
 * SensorML document and HTTP POST.
 *
 * @return the instance itself
 */

public SOSClient InsertSensor() {
  XMLParser parser = new XMLParser();
  parser.readXML("/resources/InsertSensor.xml");
  postDocument(parser.getXmlDocument());
  return this;
}

/**
 * This method executes HTTP POST method using Apache HttpClient API. It
 * accepts a XML document coming from JDOM based representation and
    submits
 * the data to a SOS using hard coded address.
 *
 * @param doc
 *          JDOM based representation of XML document which shall be
    submitted
 *          to server/SOS.
```

```
   * @return the server response
   */

  public String postDocument(Document doc) {
    // Get the string representation of Document instance
    String xml = new XMLOutputter().outputString(doc);
    // set HTTP request body content and type of the content
    StringEntity entity = new StringEntity(xml, ContentType.create("
        application/xml"));
    // build client
    CloseableHttpClient httpClient = HttpClientBuilder.create().build();
    // specify server address and generate POST request
    HttpPost post = new HttpPost(sosInstanceAddress);
    // attach message to request
    post.setEntity(entity);
    // initialize response and result of the server
    CloseableHttpResponse response = null;
    String result = null;
    // execute HTTP POST
    try {
      // To get server response print out getStatusLine()-method of response
          ,
      // but here we just execute the HTTP-POST
      response = httpClient.execute(post);
      result = EntityUtils.toString(response.getEntity());
      System.out.println(result);
      // final steps
      if (entity != null)
        EntityUtils.consume(entity);
      post.reset();
    } catch (ClientProtocolException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        response.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
    return result;
  }

  /**
   * This method checks if the telemetry system is already registered and
      known
   * by the SOS using XPath.
   *
   * @param response
   *          of the server which has to be analyzed.
```

```
   * @return a boolean which specifies whether the procedure is registered
      or
   *         not.
   */

  public boolean observationProcedureRegistered(String response) {
    XMLParser parser = new XMLParser();
    List<Element> procedures;
    try {
      procedures = parser.getElementsByXpath(".//swes:procedure", parser.
          getSwes(), parser.buildDOM(response));
    } catch (NullPointerException e) {
      return false;
    }
    for (Element procedure : procedures) {
      if (procedure.getText().compareTo("http://www.52north.org/test/
          procedure/telemetrySystem") == 0) {
        System.out.println(procedure.getText() + " is already registered at
            SOS.");
        return true;
      }
    }
    return false;
  }

  // Getters and Setters
  public String getGetObservationResponse() {
    return getObservationResponse;
  }

  public void setGetObservationResponse(String getObservationResponse) {
    this.getObservationResponse = getObservationResponse;
  }

  public static String getResponseStandardOutputPath() {
    return responseStandardOutputPath;
  }

  public String getGetCapabilitiesResponse() {
    return GetCapabilitiesResponse;
  }

  public void setGetCapabilitiesResponse(String getCapabilitiesResponse) {
    GetCapabilitiesResponse = getCapabilitiesResponse;
  }
}
```

Listing D.7: *SOSClient*-Klasse zur Erzeugung von HTTP-POST-Anfragen

```
package classes;

import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import com.zaxxer.hikari.HikariDataSource;

/**
 * Class for manipulating and querying telemetry data using plain SQL
 * statements. The statements are hard coded in the source code to minimize
 * security risk potential and to ensure comprehensibility. If further
 * statements are needed there is the possibility to outsource the SQL
 * statements in external resources to keep code clean.
 *
 * @author Dezhi Fu
 */
public class DataSourceConnector {
  private static HikariDataSource dataSource;
  // create table sql statements
  private static final String sqlCreateNMEABasicTable = "CREATE TABLE IF NOT
      EXISTS nmeadata("
      + "id BIGSERIAL NOT NULL PRIMARY KEY," + "start_counter INTEGER NOT
         NULL, "
      + "local_time DOUBLE PRECISION NOT NULL," + "longitude DOUBLE
         PRECISION NOT NULL,"
      + "latitude DOUBLE PRECISION NOT NULL," + "altitude DOUBLE PRECISION
         NOT NULL," + "utc TEXT NOT NULL,"
      + "time_to_first_fix INTEGER NOT NULL," + "
         number_of_satellites_in_view SMALLINT NOT NULL,"
      + "speed_over_ground DOUBLE PRECISION NOT NULL," + "course_over_ground
          DOUBLE PRECISION NOT NULL);";
  private static final String sqlCreateRPMTable = "CREATE TABLE IF NOT
     EXISTS rpmdata("
      + "id BIGSERIAL NOT NULL PRIMARY KEY," + "start_counter INTEGER NOT
         NULL, "
      + "local_time_identity DOUBLE PRECISION NOT NULL ," + "rpm_value
         SMALLINT NOT NULL);";
  private static final String sqlCreateFuelConsumptionTable = "CREATE TABLE
     IF NOT EXISTS fuelconsumptiondata("
      + "id BIGSERIAL NOT NULL PRIMARY KEY," + "start_counter INTEGER NOT
         NULL, "
      + "local_time_identity DOUBLE PRECISION NOT NULL ," + "
         fuel_consumption_value DOUBLE PRECISION NOT NULL);";
  // drop table sql statements
  private static final String sqlDropNMEABasicTable = "DROP TABLE IF EXISTS
     nmeadata CASCADE;";
```

```
private static final String sqlDropRPMTable = "DROP TABLE IF EXISTS
    rpmdata CASCADE;";
private static final String sqlDropFuelConsumptionTable = "DROP TABLE IF
    EXISTS fuelconsumptiondata CASCADE;";
// create view sql statement
private static final String sqlCreateView = "CREATE OR REPLACE VIEW
    georeferencedTelemetryData AS "
     + "SELECT nmeadata.id," + "nmeadata.longitude," + "nmeadata.latitude,"
         + "nmeadata.altitude," + "nmeadata.utc,"
     + "nmeadata.time_to_first_fix," + "nmeadata.
        number_of_satellites_in_view," + "nmeadata.speed_over_ground,"
     + "nmeadata.course_over_ground," + "rpmdata.rpm_value," + "
        fuelconsumptiondata.fuel_consumption_value FROM "
     + "nmeadata, rpmdata, fuelconsumptiondata," + "(SELECT DISTINCT ON (
        nmeadata.id) nmeadata.id as nmeadata_id,"
     + "rpmdata.id as rpmdata_id," + "fuelconsumptiondata.id as
        fuelconsumption_id,"
     + "nmeadata.start_counter as nmeadata_start_counter," + "rpmdata.
        start_counter as rpmdata_start_counter,"
     + "fuelconsumptiondata.start_counter as
        fuelconsumptiondata_start_counter,"
     + "abs(nmeadata.local_time-rpmdata.local_time_identity) as
        abs_time_difference_rpmdata,"
     + "abs(nmeadata.local_time-fuelconsumptiondata.local_time_identity) as
         abs_time_difference_fuelconsumptiondata "
     + "FROM public.nmeadata, public.rpmdata, public.fuelconsumptiondata "
     + "WHERE nmeadata.start_counter = rpmdata.start_counter "
     + "AND rpmdata .start_counter = fuelconsumptiondata.start_counter "
     + "ORDER BY nmeadata_id, abs_time_difference_rpmdata,
        abs_time_difference_fuelconsumptiondata ASC)inner_table "
     + "WHERE (nmeadata.id = nmeadata_id AND rpmdata.id = inner_table.
        rpmdata_id "
     + "AND fuelconsumptiondata.id = inner_table.fuelconsumption_id) " + "
        AND (abs_time_difference_rpmdata < 0.5 "
     + "AND abs_time_difference_fuelconsumptiondata < 0.5)";
// select paginated view sql statement
private static String sqlSelectViewSublist = "SELECT * FROM
    georeferencedTelemetryData ORDER BY georeferencedTelemetryData.id LIMIT
     %d OFFSET %d";
// offset view rows
private static int offsetRows = 0;

/**
 * Assign a HikariDataSource to static DataSource.
 *
 * @param ds
 *          a HikariConfig instance
 */
public DataSourceConnector(HikariDataSource ds) {
  dataSource = ds;
}
```

```
/**
 * Call the close()-method of the DataSource-instance to shutdown the
 * DataSource and its associated pool.
 */
public void closeDataSource() {
  dataSource.close();
}

/**
 * Drop all tables and linked dependencies (e. g. views).
 *
 */
public void dropTables() {
  dropTable(sqlDropNMEABasicTable);
  dropTable(sqlDropRPMTable);
  dropTable(sqlDropFuelConsumptionTable);
}

/**
 * Drop single database table using the static DROP TABLE SQL statements
    of
 * this class.
 *
 * @param sqlStatement
 *          One of the DROP SQL statements which are hard coded in the
    class
 *          definition.
 */
private void dropTable(final String sqlStatement) {
  try (Connection conn = dataSource.getConnection()) {
    // Drop existing table
    try (Statement stmt = conn.createStatement()) {
      stmt.executeUpdate(sqlStatement);
    }
  } catch (SQLException e) {
    e.printStackTrace();
  }
}

/**
 * Create view of the georeferenced telemetry data using the static CREATE
 * VIEW SQL statements of this class.
 */
public void createView() {
  try (Connection conn = dataSource.getConnection()) {
    // Drop existing table
    try (Statement stmt = conn.createStatement()) {
      stmt.executeUpdate(sqlCreateView);
    }
  } catch (SQLException e) {
    e.printStackTrace();
  }
```

```
}

/**
 * Get a List of vehicle instances using the
 * selectGeoreferencedTelemetryDataView()-method.
 */
public List<Vehicle> getVehicleData() {
  return selectGeoreferencedTelemetryDataView();
}

/**
 * Return a list of vehicle instances. First execute a SELECT SQL
    Statement to
 * get the VIEW of georeferenced telemetry data. Use pagination technique
    to
 * ensure selecting a subset of the total result set. The number of
    retrieved
 * rows is set by the value of the 'limitNumberOfRows' variable.
 *
 */
private List<Vehicle> selectGeoreferencedTelemetryDataView() {
  int limitNumberOfRows = 10;
  String sql = String.format(sqlSelectViewSublist, limitNumberOfRows,
     offsetRows);
  List<Vehicle> vehicleList = new ArrayList<Vehicle>();
  try (Connection conn = dataSource.getConnection()) {
    // enable scrolling of rows, which are sensitive to changed data
    Statement stmt = createStatement(conn, ResultSet.TYPE_SCROLL_SENSITIVE
       , ResultSet.CONCUR_READ_ONLY);
    // give the database 'hints' how many rows should be returned
    stmt.setFetchSize(limitNumberOfRows);
    stmt.setMaxRows(limitNumberOfRows);
    try (ResultSet rs = stmt.executeQuery(sql)) {
      while (rs.next()) {
        double longitude = rs.getDouble("longitude");
        double latitude = rs.getDouble("latitude");
        // double altitude = rs.getDouble("altitude");
        String timestamp = rs.getString("utc");
        // int ttff = rs.getInt("time_to_first_fix");
        short numberOfSatellitesInView = rs.getShort("
           number_of_satellites_in_view");
        // double speedOverGround = rs.getDouble("speed_over_ground");
        // double courseOverGround = rs.getDouble("course_over_ground");
        short rpmValue = rs.getShort("rpm_value");
        double fuelConsumptionValue = rs.getDouble("fuel_consumption_value
           ");
        vehicleList.add(
            new Vehicle(longitude, latitude, timestamp,
               numberOfSatellitesInView, rpmValue, fuelConsumptionValue));
      }
    } catch (SQLException e) {
      e.printStackTrace();
```

```
      }
    } catch (SQLException e1) {
      e1.printStackTrace();
    }
    // adjust offset parameter to ensure reading rows from new offset on
    offsetRows = offsetRows + vehicleList.size();
    return vehicleList;
  }

  /**
   * Create all tables used in the telemetry system workchain.
   *
   */
  public void createTables() {
    createTable(sqlCreateNMEABasicTable);
    createTable(sqlCreateRPMTable);
    createTable(sqlCreateFuelConsumptionTable);
  }

  /**
   * Create single database table using the static CREATE TABLE SQL
      statements
   * of this class.
   *
   * @param sqlStatement
   *          One of the CREATE SQL statements which are hard coded in the
      class
   *          definition.
   */
  private void createTable(String createTableStatement) {
    try (Connection conn = dataSource.getConnection()) {
      // Create tables
      try (PreparedStatement pstmt = conn.prepareStatement(
         createTableStatement)) {
        try {
          pstmt.executeUpdate();
        } catch (SQLException e) {
          e.printStackTrace();
        }
      } catch (SQLException e1) {
        e1.printStackTrace();
      }
    } catch (SQLException e2) {
      e2.printStackTrace();
    }
  }

  /**
   * Insert values into database using a String of incoming data
   *
   * @param linesOfData
```

```
     *          A concatenated String of semicolon-separated sentences
        containing
     *          telemetry data. The values in one sentence are separated by
     *          commata.
     * @return A int array containing the number of inserted rows for each of
        the
     *         three data groups, i. e. NMEA-sentences, RPM-sentences, and
        Fuel
     *         Consumption-sentences.
     */
    public int[] insertValues(String linesOfData) {
      // SQL parameterized Insert-statements
      String sqlForInsertNMEA = ("INSERT INTO nmeadata VALUES (Default, ?, ?,
         ?, ?, ?, ?, ?, ?, ?, ?)");
      String sqlForInsertRPM = ("INSERT INTO rpmdata VALUES (Default, ?, ?, ?)
         ");
      String sqlForInsertFuelConsumption = ("INSERT INTO fuelconsumptiondata
         VALUES (Default, ?, ?, ?)");
      // strip whitespace and parse data using delimiter semicolon
      String[] separatedSentences = linesOfData.trim().split(";");
      System.out.println();
      System.out.println("Number of incoming telemetry system data sentences:
         " + separatedSentences.length);
      // prepare PreparedStatement-instances
      PreparedStatement psNMEA = null;
      PreparedStatement psRPM = null;
      PreparedStatement psFuelConsumption = null;
      // hold batch constant to limit the number of inserts
      final int batchSize = 1000;
      int batchCount = 0;
      // count the number of inserts
      int insertNMEACounter = 0;
      int insertRPMCounter = 0;
      int insertFuelConsumptionCounter = 0;
      // get database connection
      try (Connection conn = dataSource.getConnection()) {
        // create SQL statements
        psNMEA = createPreparedStatement(conn, sqlForInsertNMEA);
        psRPM = createPreparedStatement(conn, sqlForInsertRPM);
        psFuelConsumption = createPreparedStatement(conn,
           sqlForInsertFuelConsumption);
        // for each semicolon-separated sentence we're going to parse them
        // appropriately
        for (int i = 0; i < separatedSentences.length; i++) {
          System.out.println("--------------------------------");
          System.out.println("Parsing sentence " + (i + 1) + " from " +
             separatedSentences.length + ":");
          System.out.println();
          System.out.println(separatedSentences[i]);
          // separate values in one sentence, '_'-delimiter is used to split
             the
```

```
// local time identity which consists of a start counter and the
   local
// timestamp of the measurement data logger (e. g. 6_1469695189
   .109694)
String[] separatedSentence = separatedSentences[i].split(",|_");
int startCounter = Integer.parseInt(separatedSentence[0]);
double localTimestamp = Double.parseDouble(separatedSentence[1]);
// if we have a NMEA sentence (contains 11 values), parse it
   accordingly
// and add to our batch

if (separatedSentence.length == 11) {
  double longitude = Double.parseDouble(separatedSentence[3]);
  double latitude = Double.parseDouble(separatedSentence[4]);
  double altitude = Double.parseDouble(separatedSentence[5]);
  String timestamp = separatedSentence[6];
  int ttff = Integer.parseInt(separatedSentence[7]);
  short numberOfSatellitesInView = Short.parseShort(
     separatedSentence[8]);
  double speedOverGround = Double.parseDouble(separatedSentence[9]);
  double courseOverGround = Double.parseDouble(separatedSentence
     [10]);
  psNMEA.setInt(1, startCounter);
  psNMEA.setDouble(2, localTimestamp);
  psNMEA.setDouble(3, longitude);
  psNMEA.setDouble(4, latitude);
  psNMEA.setDouble(5, altitude);
  psNMEA.setString(6, timestamp);
  psNMEA.setInt(7, ttff);
  psNMEA.setShort(8, numberOfSatellitesInView);
  psNMEA.setDouble(9, speedOverGround);
  psNMEA.setDouble(10, courseOverGround);
  psNMEA.addBatch();
  // otherwise the sentence is a processed CAN-sentence (contains 5
  // values)
} else if (separatedSentence.length == 4) {

  String vehicleDataType = separatedSentence[2];
  String vehicleData = separatedSentence[3];
  // depending on the data type (e. g. RPM, fuel consumption etc.)
     we
  // process the sentence accordingly
  switch (vehicleDataType) {
  case "R":
    short rpm = (short) Double.parseDouble(vehicleData);
    psRPM.setInt(1, startCounter);
    psRPM.setDouble(2, localTimestamp);
    psRPM.setShort(3, rpm);
    psRPM.addBatch();
    break;
  case "F":
    double fuelConsumption = Double.parseDouble(vehicleData);
```

```
            psFuelConsumption.setInt(1, startCounter);
            psFuelConsumption.setDouble(2, localTimestamp);
            psFuelConsumption.setDouble(3, fuelConsumption);
            psFuelConsumption.addBatch();
            break;
          }
        }
        // safe batch within batch to limit inserts according to batch size
        if (++batchCount > batchSize) {
          if (psNMEA != null) {
            int[] results = psNMEA.executeBatch();
            insertNMEACounter = insertNMEACounter + results.length;
          }
          if (psRPM != null) {
            int[] results = psRPM.executeBatch();
            insertRPMCounter = insertRPMCounter + results.length;
          }
          if (psFuelConsumption != null) {
            int[] results = psFuelConsumption.executeBatch();
            insertFuelConsumptionCounter = insertFuelConsumptionCounter +
               results.length;
          }
          // reset batchCount
          batchCount = 0;
        }
      }
      // execute each batch
      if (psNMEA != null) {
        int[] results = psNMEA.executeBatch();
        insertNMEACounter = results.length;
      }
      if (psRPM != null) {
        int[] results = psRPM.executeBatch();
        insertRPMCounter = results.length;
      }
      if (psFuelConsumption != null) {
        int[] results = psFuelConsumption.executeBatch();
        insertFuelConsumptionCounter = results.length;
      }
    } catch (SQLException e) {
      e.printStackTrace();
      e.getNextException().printStackTrace();
    }
    int[] insertedRowsPerColumn = { insertNMEACounter, insertRPMCounter,
       insertFuelConsumptionCounter };
    return insertedRowsPerColumn;
  }

  /**
   * Returns a boolean which indicates the existence of a view.
   *
   * @param viewName
```

```
     *          The name of the view as a string
     * @return a boolean stating the desired view does exist
     *
     */
    public boolean viewExists(String viewName) {
      DatabaseMetaData meta = null;
      try (Connection conn = dataSource.getConnection()) {
        meta = conn.getMetaData();
      } catch (SQLException e) {
        e.printStackTrace();
      }
      try (ResultSet res = meta.getTables(null, null, viewName, new String[] {
          "VIEW" })) {
        try {
          if (res.next()) {
            // Table exists
            return true;
          } else {
            // Table does not exist
            return false;
          }
        } catch (SQLException e1) {
          e1.printStackTrace();
        }
      } catch (SQLException e) {
        e.printStackTrace();
      }
      return false;
    }

    /**
     * Returns a boolean which indicates the existence of a table.
     *
     * @param tableName
     *          The name of the table as a string
     * @return a boolean stating the desired table does exist
     *
     */
    public boolean tableExists(String tableName) {
      DatabaseMetaData meta = null;
      try (Connection conn = dataSource.getConnection()) {
        meta = conn.getMetaData();
      } catch (SQLException e) {
        e.printStackTrace();
      }
      try (ResultSet res = meta.getTables(null, null, tableName, new String[]
         { "TABLE" })) {
        try {
          if (res.next()) {
            // Table exists
            return true;
          } else {
```

```
            // Table does not exist
            return false;
          }
        } catch (SQLException e1) {
          e1.printStackTrace();
        }
      } catch (SQLException e) {
        e.printStackTrace();
      }
      return false;
    }

    /**
     * Returns a Statement instance for use in a try-resources-section.
     *
     * @param conn
     *          A Connection instance to a data source.
     * @param resultSetType
     *          A int constant to set the type of the result set
     * @param resultSetConcurrency
     *          A int constant to set the concurrency mode of the result set
     * @return A Statement instance
     *
     */
    private Statement createStatement(Connection conn, int resultSetType, int
        resultSetConcurrency) throws SQLException {
      Statement stmt = conn.createStatement(resultSetType,
          resultSetConcurrency);
      return stmt;
    }

    /**
     * Returns a PreparedStatement instance for use in a try-resources-section
        .
     *
     * @param conn
     *          A Connection instance to a data source.
     * @param sql
     *          A String which containts the parameterized SQL Statement
     *
     */
    private PreparedStatement createPreparedStatement(Connection conn, String
        sql) throws SQLException {
      PreparedStatement ps = conn.prepareStatement(sql);
      return ps;
    }
}
```

Listing D.8: *DataSourceConnector*-Klasse zur Kommunikation mit Datenbanken

```
package classes;

import java.util.Properties;
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

/**
 * * A class to connect to databases using DataSource instances. Using the
 * provided methods one can connect to different databases.
 *
 * @author Dezhi Fu
 */
public class DataSourceConnectorFactory {
  private static final String POSTGRES_DATA_SOURCE_CLASS_NAME = "org.
     postgresql.ds.PGPoolingDataSource";
  private static final String USER = "postgres";
  private static final String PASSWORD = "p0stgr3s";

  /** Don't let anyone instantiate this class */
  private DataSourceConnectorFactory() {
  }

  /**
   * This method creates a connection to the 'telemetrySystemDatabase' which
       was
   * preconfigured using pgAdmin. Here we save the telemetry data of our
   * datalogger.
   */
  public static HikariDataSource getPostgresqlDataSource() {
    final String DB_NAME = "telemetrySystemDatabase";
    Properties props = new Properties();
    props.setProperty("dataSourceClassName", POSTGRES_DATA_SOURCE_CLASS_NAME
        );
    props.setProperty("dataSource.user", USER);
    props.setProperty("dataSource.password", PASSWORD);
    props.setProperty("dataSource.databaseName", DB_NAME);
    props.setProperty("dataSource.portNumber", "1890");
    return new HikariDataSource(new HikariConfig(props));
  }

  /**
   * This method creates a connection to the 'testDatabase' which was
   * preconfigured using pgAdmin. Here we save test data. The test data is
       hard
   * coded as class variables of the class 'DataSourceConnector' and is used
       in
   * JUnit tests to check the functionality of the 'DataSourceConnector'
       class.
   */
  public static HikariDataSource getTestPostgresqlDataSource() {
    final String DB_NAME = "testDatabase";
    Properties props = new Properties();
```

```
    props.setProperty("dataSourceClassName", POSTGRES_DATA_SOURCE_CLASS_NAME
        );
    props.setProperty("dataSource.user", USER);
    props.setProperty("dataSource.password", PASSWORD);
    props.setProperty("dataSource.databaseName", DB_NAME);
    props.setProperty("dataSource.portNumber", "1890");
    return new HikariDataSource(new HikariConfig(props));
  }
}
```

Listing D.9: *DataSourceConnectorFactory*-Klasse zur Auswahl von DataSourceConnector-Instanzen mit festgelegten Datenbankparametern