CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Ein effizienter Ansatz zur Übersetzung zwischen Service-Protokollen durch die Modellierung von Verhaltensunterschieden

Dissertation zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

vorgelegt von

Dipl.-Inf. (FH) Claas Busemann

Gutachter:

Jun.-Prof. Dr. Daniela Nicklas
Prof. Dr. Ralf H. Reussner

Tag der Disputation: 22. November 2012

für Sandra

Danksagung

An dieser Stelle möchte ich mich bei meiner Doktormutter Frau Jun.-Prof. Dr. Daniela Nicklas bedanken, dass sie mir die Möglichkeit gegeben hat diese Promotion durchzuführen und mich während der vergangenen Jahre fachlich und methodisch unterstützt hat. Ohne ihr Vertrauen in meine Fähigkeiten wäre diese Arbeit nie zu einem Abschluss gekommen.

Darüber hinaus möchte ich mich bei Frau Prof. Dr. Susanne Boll und Herr Prof. Dr. Ralf H. Reussner, sowie bei Herr Prof. Dr. Andreas Winter bedanken. Durch ihre fachliche Unterstützung haben diese einen wertvollen Teil zu meiner Arbeit beigetragen.

Den Kollegen im OFFIS der Abteilungen Kooperierende Mobile Systeme und Informationssysteme möchte ich ebenfalls für die vielen fachlichen Diskussionen danken, die mir geholfen haben das Ziel nicht aus den Augen zu verlieren.

Mein persönlicher Dank gilt meiner Frau Sandra, ohne deren helfende Worte und unermüdliche Geduld diese Arbeit nicht möglich gewesen wäre.

Schaafheim, den 29. November 2012

Zusammenfassung

Das Internet erlaubt mittlerweile nicht nur den Zugriff auf verschiedene Dienste sondern auch auf Daten, die von unterschiedlichen Sensoren produziert werden. Diese Quellen verwenden dabei häufig unterschiedliche Protokolle zur Kommunikation. Die Integration von Sensor- und Service-Protokollen in Applikationen und Middleware-Systeme ist noch immer eine aufwändige und zeitintensive Aufgabe für Entwickler. Hauptgrund dafür ist, dass durch die heterogene Sensor- und Service-Welt eine Vielzahl unterschiedlicher Protokolle in teilweise unterschiedlichen Versionen existiert. Hinzu kommt, dass einige dieser Protokolle häufig verändert werden. Bestehende Sensoren oder Services können jedoch nicht immer aktualisiert werden, da sie zum Beispiel in geschlossenen Systemen laufen. Bei softwaretechnischen Lösungen wird versucht, dieses Problem mittels Design-Pattern, Datentransformationstechnologien oder manueller Programmierung zu lösen. Modellgetriebene Ansätze konzentrieren sich dagegen auf eine formale Beschreibung der Protokolle, um daraus automatisiert Konverter zu generieren. Modellgetriebene Ansätze versprechen zwar mehr Übersicht, Flexibilität und Wartbarkeit, erfordern aber auch eine entsprechend aufwändige Einarbeitung. Daher werden, gerade im Bereich der Middleware-Systeme, softwaretechnische Lösungen bevorzugt.

In dieser Arbeit wird ein neues Modellierungsverfahren vorgestellt, das speziell für die Konvertierung von Service-Protokollen entwickelt wurde. Dabei wurde darauf geachtet, dass das Modellierungsverfahren leicht erlernbar ist und der Modellierungsaufwand gering gehalten wird. Der Ansatz konzentriert sich auf die Modellierung von Protokollunterschieden und erlaubt die Beschreibung von Änderungen des Kommunikationsverhaltens mittels Mustern. Darüber hinaus können Zustände durch Entscheidungsbäume abgebildet werden. Die Transformation der Nachricht ist ebenfalls in das Modell integriert und kann mittels beliebiger Anfragesprachen realisiert werden. Diese erlauben die Transformation von Nachrichten mittels einer Anfrage, die auf die jeweilige Nachricht angewendet wird. Neben der Evaluation der Verarbeitungszeiten wird in der Arbeit auch der Aufwand des Modellierungsansatzes gegen eine softwaretechnische Lösung abgeglichen. Dazu wurde eine Benutzerstudie durchgeführt, in der die Teilnehmer eine Protokollkonvertierungsaufgabe entweder mit dem vorgestellten Modell oder mit einer konkreten Implementierung lösen mussten.

Abstract

Nowadays, the Internet not only allows access to different services but also to access data produced by different kinds of sensors. These sources of data usually use different protocols to communicate. For developers, the integration of such sensor and service protocols into applications is still a tedious and time-consuming task. This is caused by the increasing number of protocols that already exists in a variety of versions. Existing service protocols are often modified to support new features but sensors and services may be closed systems, so that it is impossible to update them. This problem can be solved by programming software that converts one protocol to another. Thereby, design-patterns, data transformation technologies or manual coding are used to integrate these protocols. Also, several model-driven solutions can be used that focus on the formal description of protocols to automatically generate converters. While model-driven approaches seem to be more flexible and easier to maintain, they also require a larger amount of effort for learning the basic skills. Due to this, those technologies are often not used, especially when it comes to middleware systems.

This thesis describes a model-driven approach that has been designed to convert service protocols into each other. The model was developed in a way so that it can be easily learned. The approach focuses on the modeling of the differences between protocols. The differences in the communication behavior are described using patterns, while the state of a service can be described using decision trees. The messages are converted using query technologies which have been integrated into the model. The evaluation contains an analysis of the processing times of the patterns as well as a comparison with other model-driven approaches. The approach is also compared to a software-driven approach using a user-study.

Inhalt

1 Einleitung

Die Anzahl der Dienste, die ihre Daten mittels unterschiedlicher Kommunikationsprotokolle übertragen, hat sich in den vergangenen Jahren stark erhöht. Diese Protokolle reichen von offenen Spezifikationen, die auf klar definierten Standards beruhen bis hin zu proprietären Systemen, zu denen in den meisten Fällen kaum Informationen verfügbar sind. Die Dienste selbst können sowohl statische Systeme sein, bei denen es keine Möglichkeit zur Änderung beziehungsweise Aktualisierung des Kommunikationsverhaltens gibt, wie auch Systeme bei denen eine Änderung des Kommunikationsprotokolls regelmäßig stattfindet. Das Nachrichtenformat der Protokolle reicht von einfachen textuellen Formen wie NMEA [Nat10] oder STOMP [The06] bis hin zu komplexen binären Protokollen, wie sie beispielsweise häufig in industriellen Feldbus Anwendungen (z.B. EtherCAT [BJR03])) verwendet werden.

1.1 Motivation und Ziele der Arbeit

Durch die hohe Anzahl an Diensten, die nicht nur unterschiedliche Kommunikationsprotokolle verwenden sondern diese auch über die Zeit ändern, müssen Applikationen immer flexibler werden wenn es um die Integration von Daten dieser Quellen geht. Ein beliebter Ansatz sind Middleware-Systeme wie beispielsweise SCAMPI [BKW+09] oder SStream [GLH09]. Dabei werden die Daten nicht von der Anwendungen entgegengenommen sondern von einer Middleware. Diese erlaubt durch Adapter und Wrapper die Integration unterschiedlicher Dienste, die auch physische Sensoren sein können [MR09]. Die Middleware nimmt die Daten entgegen und stellt sie in einem einheitlichem Format wieder zur Verfügung. Applikationen müssen so nur ein Protokoll und einen Kommunikationsweg implementieren. Einige dieser Systeme erlauben darüber hinaus die Vorverarbeitung der Daten innerhalb der Middleware. Dadurch wird ein Großteil der Komplexität in Bezug auf die Integration und Verarbeitung, in die Middleware verlagert und die Applikation entsprechend entlastet. Eine schematische Darstellung eine Middleware findet sich in Abb. 1.1.

Ein Großteil des Arbeitsaufwands bei der Entwicklung von Systemen, die mit einer Vielzahl von unterschiedlichen Datenquellen umgehen müssen, wird durch die Integration der Datenquellen erzeugt. Dies kann 35 bis 50 Prozent der gesamten System-Entwicklungskosten in Anspruch nehmen [Bro06, LP11] und wird durch die hohe Heterogenität der Quellen verursacht. Wird ein Kommunikationsprotokoll hinzugefügt, muss ein neuer Adapter oder Wrapper für das jeweilige Protokoll implementiert werden. Ändert sich diese Protokoll zu einem späterem Zeitpunkt muss sichergestellt werden, dass nicht nur alle Änderungen bei der Konvertierung der Daten, sondern auch bei der Konvertierung des

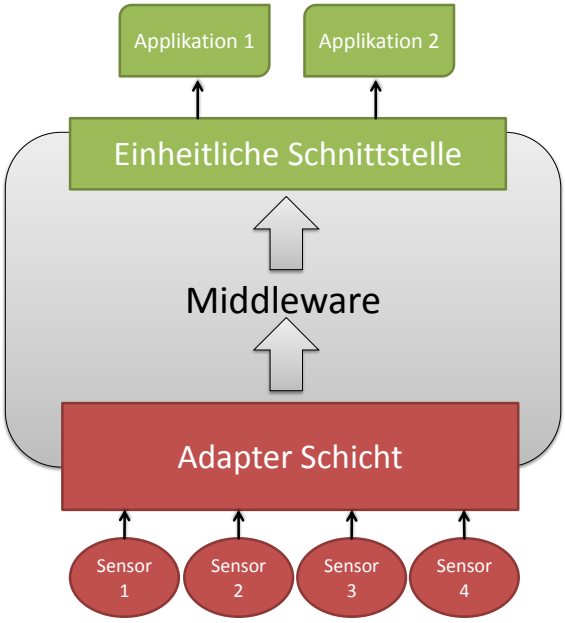


Abbildung 1.1: Schematische Darstellung einer Middleware

Kommunikationsverhaltens beachtet wurden. Eine Änderung des Kommunikationsverhaltens könnte beispielsweise eine zusätzliche Nachricht sein, die an die Quelle gesendet wird sobald Daten empfangen werden.

Momentan gibt es zwei übliche Ansätze um diesem Problem zu begegnen. Der erste Ansatz konzentriert sich auf die Transformation der Daten. Dazu wurden sogenannte Anfrage-Sprachen entwickelt (z.B. reguläre Ausdrücke, XSLT [Kay07] und XQuery [BCF+07]). Eine solche Anfrage kann verwendet werden um die Nachrichten eines Protokolls in die Nachrichten eines anderen Protokolls zu konvertieren. Der Vorteil dabei ist, dass diese Anfragen relativ einfach zu schreiben und lesen sind und somit einfach angepasst werden können. Allerdings sehen Anfrage-Technologien nicht vor, auch das Kommunikationsverhalten eines Protokolls zu konvertieren, da sie dazu gedacht sind, Dokumente zu konvertieren. Muss die Nachricht beispielsweise in ein neues Format übersetzt werden, aber zusätzlich noch eine weitere Nachricht an den Quelldienst gesendet werden (siehe Abb. 1.2), kann dies nicht mittels einer Anfrage-Sprache ausgedrückt werden. Veränderungen des Kommunikationsverhaltens werden deshalb häufig durch zusätzliche Implementierungen realisiert. Das führt wiederum dazu, dass die fallspezifische Implementierung in der Regel angepasst werden muss, wenn Änderungen an einem Protokoll durchgeführt werden.

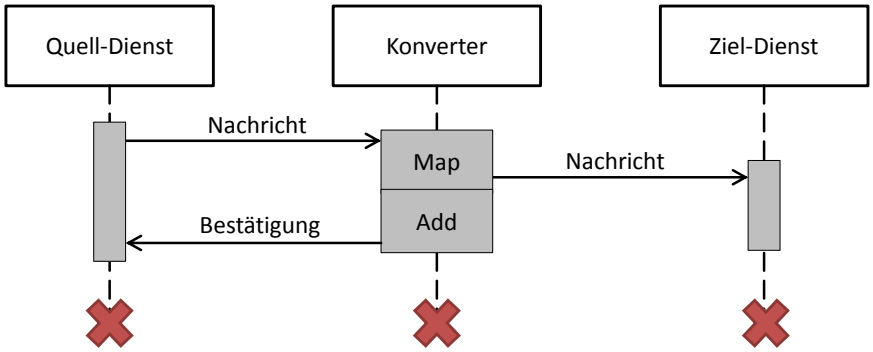


Abbildung 1.2: Beispiel eine Protokollkonvertierung

Der zweite Ansatz konzentriert sich auf die Modellierung der Protokolle selbst und der Unterschiede zwischen diesen. Dabei werden üblicherweise die verwendeten Protokolle mittels eines abstrakten Modells beschrieben, aus denen dann ein Konverter, üblicherweise in Form eines endlicher Zustandsautomat, erzeugt wird [YS97]. Davon unabhängig wird beschrieben, wie die Konvertierung der Datenrepräsentation realisiert werden soll. Aufbauend auf diesen Modellen kann dann ein spezifischer Adapter synthetisiert werden. Da sämtliche Informationen über das Verhalten der Protokolle, sowie die Unterschiede zwischen den Protokollen modelliert wurden, kann der Adapter nicht nur sehr flexibel angepasst werden, sondern auch formal auf Korrektheit geprüft werden. Dies ist gerade für sicherheitskritische Protokolle wichtig. Da der Ansatz die formale Modellierung der beteiligten Protokolle voraussetzt, ist dieser mit einem entsprechend hohen Beschreibungsaufwand verbunden. Daraus lässt sich die folgende Forschungsfrage ableiten, deren Beantwortung das Ziel dieser Arbeit ist:

Wie lässt sich der Aufwand beim Übersetzen zwischen Service-Protokollen für Entwickler verringern?

Das Ziel dieser Arbeit ist es somit, ein Modell zu entwickeln, das es Entwicklern ermöglicht, Konverter für Service-Protokolle zu definieren. Dabei soll der initiale Modellierungsaufwand vergleichbar mit dem einer konkreten Implementierung sein. Das im Rahmen dieser Arbeit vorgestellte Differential Behavior Model (DBM) erlaubt es dem Entwickler, die Unterschiede zwischen Service-Protokollen zu modellieren. Dabei wird, im Gegensatz zu den bereits erwähnten Ansätzen, auf eine formale Spezifikation der Protokolle verzichtet. Während die Datentransformation mittels Anfrage-Technologien realisiert wird, können Änderungen des Kommunikationsverhaltens mittels Mustern beschrieben werden.

Das DBM wurde mit dem Ziel entwickelt, eine möglichst einfache Form der Beschreibung von Unterschieden zwischen Protokollen zu realisieren.

1.2 Forschungsmethodik

Um die Forschungsfrage und die damit verbundenen Probleme zu lösen, wird in dieser Arbeit das Differential Behavior Model entwickelt. Dazu wird zunächst eine Analyse verschiedener Service-Protokolle durchgeführt. Dabei werden sowohl die Unterschiede zwischen verschiedenen Protokollen wie auch zwischen Protokollversionen betrachtet. Die Ergebnisse dieser Analyse werden verwendet um ein Modell zu entwickeln, das die Unterschiede zwischen Service-Protokollen mittels weniger, einfach zu verstehender Muster abbilden kann. Durch die Implementierung eines generischen Konverters, der in der Lage ist, dieses Modell zu interpretieren, wird die automatisierte Konvertierung der beteiligten Protokolle ermöglicht. Im Rahmen einer Evaluation wird geprüft, ob die durchschnittlichen Konvertierungszeiten, die von diesem Konverter benötigt wird, akzeptabel sind. Außerdem wird evaluiert, ob das System in einem realen Szenario eingesetzt werden kann. Darüber hinaus wird der Aufwand zum Beschreiben von Protokollunterschieden mit dem anderer Modellierungsverfahren verglichen. Der benötigte Initialaufwand zum Erlernen des Modellierungsverfahren wird im Rahmen einer Benutzerstudie ermittelt und mit dem einer konkreten Implementierung verglichen. Außerdem wird das Modell auf ein Protokoll aus einer anderen Domäne angewendet, um die Verwendbarkeit in anderen Anwendungsbereichen zu prüfen.

1.3 Problemstellung

Beim Erstellen eines Modells, auf dessen Grundlage Protokolle konvertiert werden können, müssen bestimmte Anforderungen beachtet werden. Diese wurden im Rahmen einer Protokollanalyse identifiziert, die in Kapitel 3 vorgestellt werden wird. Die Anforderungen werden im Folgenden näher erläutert.

Protokoll- und Nachrichten-Identifikation: Um Protokolle konvertieren zu können müssen diese zunächst eindeutig identifiziert werden. Dazu muss nicht nur das Protokoll selbst, sondern auch die Nachricht des jeweiligen Protokolls identifiziert werden. Dabei muss beachtet werden, dass unterschiedliche Nachrichten sich unter Umständen sehr ähnlich sein können. Dies kann gerade bei unterschiedlichen Versionen des gleichen Protokolls vorkommen. Das Modell muss dem Entwickler somit erlauben, mit simplen Mitteln Nachrichten und Protokolle anhand von beliebigen Merkmalen zu identifizieren.

Rekonstruktion fehlender Daten: Ein Hauptproblem beim Konvertieren von Protokollen sind fehlende Daten. Enthält beispielsweise eine Nachricht nicht

alle Informationen um konvertiert werden zu können, muss das Modell dem Entwickler die Möglichkeit bieten, diese zu rekonstruieren. Dies könnte beispielsweise durch das Senden einer Anfrage an einen Dienst geschehen oder durch das Auswerten von Nachrichten, die zuvor gesendet wurden. Dabei sollte das Modell unabhängig von spezifischen Kommunikationstechnologien sein.

Erzeugen zusätzlicher Nachrichten: Neben dem Rekonstruieren von fehlenden Daten muss das Modell darüber hinaus die Möglichkeit bieten, zusätzliche Nachrichten aus den eingehenden Nachrichten zu konstruieren. Diese können an den Quell- oder Zieldienst aber auch an dritte Quellen gesendet werden. Das Modell muss es dem Entwickler also erlauben, abhängig von eingehenden Nachrichten, nicht nur diese in eine andere Form zu konvertieren, sondern auch zusätzliche Nachrichten zu generieren.

Konsistenz: Ein weiterer wichtiger Punkt bei der Konvertierung von Protokollen ist die Konsistenzprüfung. Da die generierten Nachrichten auf sich ändernden eingehenden Daten beruhen, können unter Umständen fehlerhafte Nachrichten produziert werden. Daher muss das Modell alle nötigen Informationen enthalten, um zu prüfen, ob tatsächlich die korrekten Nachrichten produziert wurden.

Zeitliches Verhalten: Bei der Kommunikation mit Diensten und Sensoren kann es vorkommen, dass diese nicht immer erreichbar sind. Es ist ebenfalls möglich, dass ein Dienst Nachrichten erst nach einer bestimmten Zeit entgegen nimmt. Daher muss das Modell die Beschreibung von zeitlichem Verhalten, in Bezug auf das Absenden der produzierten Nachrichten, beinhalten.

Status Handhabung: Ein weiteres Problem ist die Handhabung des Status des Quell- oder Zieldienstes. Wenn ein Dienst zum Beispiel unterschiedliche Nachrichten, basierend auf seinem internen Status erwartet, muss das Modell die Möglichkeit bieten, die Konvertierung von Nachrichten an diese Bedingung zu knüpfen. Das Modell muss also in der Lage sein, die Ausführung von Verhaltensänderungen an für Service-Protokolle sinnvolle Bedingungen zu knüpfen.

Modellierungsaufwand: Der Aufwand zum Modellieren der Verhaltensänderungen soll mit dem Aufwand einer konkreten Implementierung vergleichbar sein. Dies gilt gerade für den Initialaufwand, der zum Erlernen des Modellierungsverfahrens benötigt wird. Hintergrund hierbei ist, dass komplexe Modellierungsverfahren häufig nicht eingesetzt werden, da die Einarbeitungszeit zu hoch erscheint auch wenn sie im späteren Entwicklungsprozess Aufwandseinsparungen bedeuten würden. Daher soll das Modell für Entwickler leicht zugänglich und einfach zu verstehen sein.

1.4 Aufbau der Arbeit

In Kapitel 2 werden verwandte Arbeiten vorgestellt. Daraufhin wird in Kapitel 3 die Analyse beschrieben, auf der das Differential Behavior Model beruht. In Kapitel 4 wird das DBM im Detail erklärt. Die entsprechende Implementierung findet sich in Kapitel 5. Die Evaluationen werden in Kapitel 6 beschrieben. Abschließend wird die Arbeit in Kapitel 7 zusammengefasst und ein Ausblick gegeben.

2 Verwandte Arbeiten

Verfahren zur Übersetzung zwischen Protokollen wurden bereits in der Mitte der 90er veröffentlicht [Liu96, TBD95]. Diese konzentrieren sich jedoch hauptsächlich auf die Übersetzung von Adressformaten und Datenstrukturen. Das Kommunikationsverhalten selbst, sowie Abhängigkeiten die durch die Übertragenen entstehen, wurden nicht beachtet. In diesem Kapitel werden Arbeiten vorgestellt, die eingesetzt werden um Service-Protokolle zu übersetzen beziehungsweise zu integrieren. Dabei wird der im Rahmen dieser Arbeit vorgestellte Ansatz gegen diese Arbeiten abgegrenzt.

2.1 Middleware-Systeme

Die Integration von Diensten und Sensoren wird häufig mittels Middleware-Systemen gehandhabt. Diese erlauben die Integration von unterschiedlichen Protokollen mittels Adapter-Schichten und Wrappern. Während ein Adapter als ein Treiber innerhalb der Middleware angesehen werden kann ist ein Wrapper eine Komponente, die außerhalb der Middleware läuft und die Daten in ein bereits bekanntes Protokoll übersetzt. Beispiele für solche Middleware-Systeme sind die Sensor.Network-Applikation [GPU10], die SenseWeb-Plattform [SNL+06], die GSN-Middleware [AHS07], die SStreamWare-Middleware [GRL+08], Oracle's SensorEdge-Plattform [Ora10], die SCAMPI-Middleware [BKW+09] oder die FireEagle-Plattform [Yah07].

Üblicherweise verfügen diese Systeme über ein eigenes Protokoll, das zur Beschreibung der Daten eingesetzt wird. Alle integrierten Protokolle werden intern in das jeweilige einheitliche Format übersetzt. Greift eine Applikation auf Daten der Middleware zu, braucht sie so nur ein Protokoll zu implementieren. Einige dieser Systeme, wie beispielsweise FireEagle, wurden speziell für bestimmte Anwendungszwecke entwickelt während andere Systeme, wie SStreamWare oder SCAMPI, versuchen, möglichst viele Anwendungen zu ermöglichen. Darüber hinaus erlauben einige dieser Systeme die Vorverarbeitung von Sensordaten innerhalb der Middleware (z.B. SCAMPI). So können Applikationen auf bereits vorverarbeitete Sensordaten zugreifen und werden entsprechend entlastet. Eine schematische Darstellung einer Middleware findet sich in Abb. 2.1.

Neben den Funktionen zur Integration und Verarbeitung von Sensordaten bieten Middleware-Systeme üblicherweise komplexe administrative Funktionen an. Es können beispielsweise Beschreibungen von Sensoren und Diensten angelegt und administriert werden, oder der Zugriff auf die Daten dieser Quellen verwaltet werden. Einige der genannten Systeme verfügen über Methoden, um den Zugriff auf Sensordaten auf Benutzer- oder Gruppen-Ebene zu regeln. Dabei kann dauerhafter oder temporärer Zugriff gewährt werden.

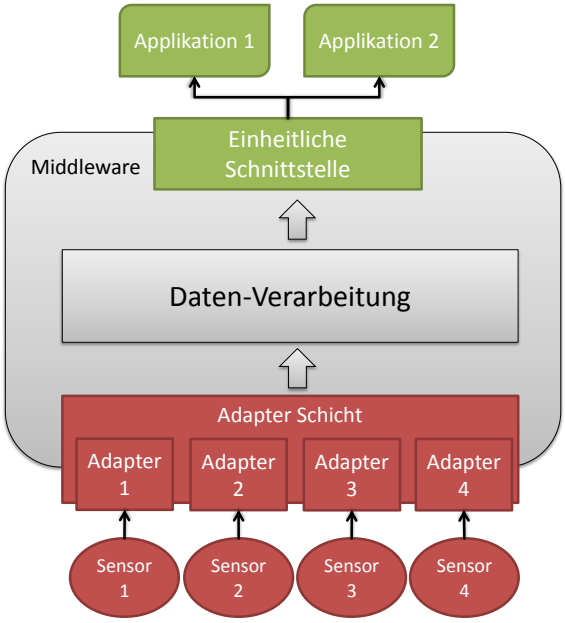


Abbildung 2.1: Schematische Darstellung eines Middleware-Systems

Da Systeme wie SenseWeb oder der Oracle Sensor Edge Server auf service-orientierten Architekturen (SOA) beruhen, können diese mit simplen Mitteln erweitert werden, so dass neue Schnittstellen und Funktionen mit relativ wenig Aufwand hinzugefügt werden können. Andere Systeme wie SStreamWare oder SCAMPI verfügen über ein integriertes Datenstrom-Management-System, das die effiziente Verarbeitung von Sensordaten, innerhalb der Middleware erlaubt [BKNB11].

Diese System integrieren also Protokolle, um die empfangenen Daten in aufbereiteter Form wieder zur Verfügung zu stellen. Dabei hat jede Middleware üblicherweise ihr eigenes einheitliches Format, in dem die Daten zur Verfügung gestellt werden. Bei der Entwicklung eines solchen Systems liegt ein Großteil des Aufwands in der Integration der Protokolle [BCG+05, Bro06]. Gerade wenn häufig neue Protokolle hinzukommen oder aber vorhandene Protokolle aktualisiert werden müssen, kann es zu komplexen Anpassungen in der Adapter-Schicht kommen. Wird das einheitliche Format, in dem die Daten wieder zur Verfügung gestellt werden geändert, muss gegebenenfalls jeder Adapter angepasst werden. Existiert ein Zugriffskontrollsystem, muss bei Änderungen der Dienste der Zugriff auch auf Adapter-Ebene beachtet werden [Fuc04].

Aus diesem Grund gibt es zahlreiche Bemühungen, die Entwicklung von Adapter und somit die Integration von Protokollen zu vereinfachen. In [BCG+05]

wird beispielsweise eine Methode vorgestellt, die Entwicklung von Adapter für Web-Dienste ermöglicht. Dabei werden beispielsweise SOA-basierte Technologien eingesetzt oder Muster spezifiziert, die falsche Zuordnungen erkennen lassen. In [DSW06] wird eine Algebra vorgestellt, die Verhaltensänderungen für Protokollschnittstellen beschreiben kann. Dabei wird ebenfalls eine grafische Entwicklungssprache vorgestellt, die die Verhaltensänderungen auf eine verständlichere Weise darstellen soll. In [KSPBC06] wird ein aspektorientiertes Framework vorgestellt, das über eine Taxonomie verschiedener Fehler in Schnittstellen-Beschreibungen verfügt. Mittels vordefinierter Templates kann so im Fehlerfall automatisiert die beste Lösung für das Konvertierungsproblem gefunden werden. In [PF04] wird darüber hinaus ein Analysewerkzeug vorgestellt, das prüft, ob eine Applikation mit einem Dienst kompatibel ist. Basierend auf dieser Analyse können dann semi-automatisiert Middleware-Komponenten generiert werden, die eine Konvertierung des Protokolls ermöglichen.

Ein etwas anderer Ansatz wird in [GMR+10] vorgestellt. Hierbei handelt es sich um ein Framework zur Protokollkonvertierung. Der Fokus liegt dabei auf der Kommunikation zwischen Sensoren und Web-Diensten. Daher wird den Sensoren ermöglicht, ihre Schnittstellen mittels eines komprimierten WSDL-Dokuments zu beschreiben. Mittels eines im Rahmen der Arbeit spezifizierten Suchalgorithmus können so Sensoren ähnlich wie Dienste gefunden werden. Das Protokollkonvertierungs-Framework wird eingesetzt, um eine Kommunikation zwischen den Sensoren und anderen IT-Systemen zu ermöglichen. Dabei werden die Kombinationen HTTP+SOAP [GHM+07b] und LTP+SOAP unterstützt. Bei der Protokollkonvertierung werden die Nachrichten zunächst in ein einheitliches Format transformiert. Die Header der Nachrichten werden transformiert, um eine Kommunikation zwischen Diensten und Sensoren zu ermöglichen. Dies ist eine reine Datentransformation und beruht auf statischen Regeln. Die Daten, die in den Nachrichten übertragen werden, werden mit zuvor spezifizierten Konvertern übersetzt. Dies enthält auch die Übersetzung des Kommunikationsverhaltens und wird auf Grundlage statischer Regeln übersetzt. Der Fokus wurde hier auf die Übersetzung des Kommunikationsverhalten von synchronen und asynchronen Protokollen gelegt. Somit erlaubt das Framework sowohl die Transformation der Daten wie auch des Kommunikationsverhaltens.

Die hier vorgestellten Middleware-Systeme beziehungsweise die entsprechenden Adapter-Technologien wurden mit dem Hintergrund entwickelt, die Integration von Sensoren und Diensten zu vereinfachen. Die Middleware-Systeme verlagern allerdings nur den Integrations- und Wartungsaufwand von der Applikation in die Middleware. Es existieren zwar unterschiedliche Ansätze um die Entwicklung und Wartung von Adapter zu vereinfachen, in den Implementierungen der jeweiligen Systeme wurden diese aber nicht verwendet. Das Framework zur Protokollkonvertierung, das in [GMR+10] vorgestellt wurde, verfügt bereits über Funktionen um die Unterschiede zwischen Protokollen zu

beschreiben. Allerdings beschränkt sich der Ansatz auf wenige Protokolle und hat einen sehr spezifischen Anwendungszweck. Die statischen Regeln zur Konvertierung des Kommunikationsverhaltens erlauben darüber hinaus keine Kommunikation mit anderen als den beteiligten Diensten oder die Beschreibung von Übertagungszeitpunkten um mit Diensten zu kommunizieren, die nicht immer erreichbar sind. Die verbleibenden Ansätze sind eher softwareorientiere Lösungen, die die Entwicklung von Adapter erleichtern sollen.

2.2 Anfrage-Sprachen

Anfrage-Sprachen sind eine beliebte Technologie, um Nachrichten von Service-Protokollen zu übersetzen [CB07, OS04]. Das hat unter anderem den Grund, dass in dieser Gruppe von Protokollen häufig XML zur Beschreibung der übertragenen Daten eingesetzt wird [GHM+07b, BB11, Cox11]. Da verschiedene Anfrage-Sprachen existieren, die komplexe Operationen auf XML-Dokumente ausführen können, um neue XML-Dokumente zu erzeugen, liegt der Schluss nahe, diese zum Übersetzen von Nachrichten XML-basierter Protokolle einzusetzen. In diesem Kapitel werden einige dieser Methoden aufgelistet. Da Service-Protokolle nicht zwangsläufig XML-basierte Protokolle verwendenden müssen, wurden entsprechend auch andere Anfrage-Sprachen aufgenommen.

2.2.1 Extensible Stylesheet Language Transformations

Bei Extensible Stylesheet Language Transformations (XSLT) [Kay07] handelt es sich um eine deklarative, XML-basierte Sprache, die verwendet werden kann, um XML-Dokumente in eine andere Form zu übersetzen. Sie wird vom World Wide Web Consortium (W3C) entwickelt. Die aktuelle Version ist 2.0 und wurde im Januar 2007 veröffentlicht. XSLT wird üblicherweise eingesetzt, um XML-Dokumente in XHTML-Dokumente zu übersetzen, so dass diese als Webseite angezeigt werden können. Da es aber auch in der Lage ist, XML-Dokumente von einem Schema in ein Anderes zu übersetzen, kann es ebenfalls zur Übersetzung von Nachrichten in Service-Protokollen eingesetzt werden.

Um ein Dokument zu übersetzen wird ein XSLT-Prozessor benötigt. Dies ist eine Software, der ein oder mehrere XML-Dokumente übergeben werden. Außerdem benötigt der Prozessor ein oder mehrere XSLT-Dokumente. Die XSLT-Dokumente beschreiben dabei wie die Quell-Dokumente verändert werden sollen. Der Prozessor generiert dann, entsprechend der Regeln in den XSLT-Dokumenten, ein oder mehrere neue Dokumente. Die erzeugten Dokumente müssen dabei nicht zwangsläufig im XML-Format sein, sondern können ein beliebiges Textformat haben. XSLT selbst ist vollständig Turing-Mächtig [Kep04].

```
<xsl:stylesheet
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="/Sensors/Values">
    <sensorValue>
      <temp>
        <xsl:value-of select="Temperature" />
      </temp>
    </sensorValue>
  </xsl:template>

</xsl:stylesheet>
```

Abbildung 2.2: XSLT-Beispiel

Ein Beispiel für ein XSLT-Dokument findet sich in Abb. 2.2. Es bewirkt, dass der Inhalt jedes *Temperature*-Elements, das im Pfad */Sensors/Values* gefunden wird, im neuen XML-Dokument im Pfad */sensorValue/temp* abgelegt wird.

2.2.2 XQuery

XQuery ist eine Anfrage-Sprache, die Daten aus XML-Dokumenten extrahieren und daraus neue Dokumente erzeugen kann [BCF+07]. Sie wurde ebenfalls vom W3C entwickelt. Dies geschah in enger Zusammenarbeit mit der Gruppe, die zur gleichen Zeit XSLT entwickelt hat. Daher verwenden beide Technologien die gleichen Basistypen. Während XSLT dazu gedacht ist, Daten möglichst komfortabel in eine neue Form zu bringen, konzentriert sich XQuery eher auf die Extraktion von Daten aus XML-Dokumenten. Beide Technologien haben aber den gleichen Anwendungszweck, die Übersetzung von XML-Dokumente in andere Formate.

```
<sensorValue>
{
        for /Sensors/Values/Temperature/$temp in doc("measurement.xml")
        return
        <temp>$temp</temp>
}
```

Abbildung 2.3: XQuery-Beispiel

XQuery wurde ursprünglich dazu entwickelt, Anfragen auf großen Mengen von XML-Dokumenten auszuführen. Daher muss nicht jedes Element eines XML-Dokuments eindeutig identifiziert werden. Der grundlegende Übersetzungsprozess ist der gleiche wie bei XSLT. Es werden eine oder mehrere XQuery-Anfragen auf ein oder mehrere XML-Dokumente angewendet. Das Ergebnis der

Anfrage kann ein oder mehrere neue Dokumente sein. XQuery ist ebenfalls Turing-Mächtig, ist aber einfacher zu erlernen als XSLT [Gra05]. Dies lässt sich grundsätzlich darauf zurückführen, dass XQuery Elemente aus Hochsprache, wie Schleifen und Abfragen, zur Verfügung stellt, so dass sie für Entwickler leichter zu verstehen und zu erlernen ist.

Ein Beispiel einer XQuery-Anfrage findet sich in Abb. 2.3. Hier werden, analog zum XSLT-Beispiel, alle Elemente des Pfads */Sensors/Values/Temperature/* identifiziert und deren Inhalt im neuen Dokument im Pfad */sensorValue/temp/* abgelegt.

2.2.3 Reguläre Ausdrücke

Die Grundlagen für reguläre Ausdrücke wurden bereits in der Mitte der 50er Jahre von Stephen Cole Kleene entwickelt [Kle56]. Diese wurden zu einer Anfrage-Sprache weiterentwickelt, die es Entwicklern erlaubt, komplexe Muster innerhalb von Texten zu identifizieren und diese mittels Templates in eine neue Form zu bringen [HU94]. Ein regulärer Ausdruck beschreibt ein Muster, das innerhalb eines Textes gesucht wird. Dabei kann der Entwickler so genannte Metazeichen verwenden, um beispielsweise unbekannte Zeichen zu beschreiben.

Ähnlich zu den bereits beschriebenen Anfrage-Sprachen ist auch hier ein Interpreter erforderlich. Dieser benötigt den regulären Ausdruck und den Text, in dem nach dem definierten Muster gesucht werden soll. Als Ergebnis liefert der Interpreter den übergebenen Text zurück. Allerdings wurde dieser, entsprechend des regulären Ausdrucks, in Segmente aufgeteilt. Mittels eines Templates kann dann ein neuer Text produziert werden. Dieses Template spezifiziert einen neuen Text mit Platzhaltern für die Segmente, die vom Interpreter zurück gegeben werden.

```
1 (\w*<Temperature>)(\w*)(</Temperature>\w*)
```

Abbildung 2.4: Beispiel eines regulären Ausdrucks

```
1 <sensorValue><temp>$2</temp></sensorValue>
```

Abbildung 2.5: Beispiel eines Templates für einen regulären Ausdruck

Ein Beispiel für einen regulären Ausdruck findet sich in Abb. 2.4. Analog zu den vorherigen Beispielen wird hier das Element *Temperature* identifiziert. Die Metazeichenfolge *\w** legt dabei fest, dass an der jeweiligen Stelle eine beliebige Folge von Buchstaben und Zahlen stehen kann. Ein Template, das die Nachricht

in das neue Format übersetzt, findet sich in Abb. 2.5. Hier wird der durch *$2* spezifizierte Platzhalter durch die mittels des regulären Ausdrucks gefundenen Daten ersetzt.

2.2.4 Anfrage Optimierung

Zusätzlich zu den vorgestellten Anfrage-Sprachen, existieren verschieden Ansätze um Anfragen zu optimieren. In [GGB+09] wird ein Algorithmus vorgestellt, der XSLT-Anfragen optimiert. Dazu wird die XSLT-Anfrage so angepasst, dass nur relevante Daten des XML-Dokuments überprüft werden. Außerdem werden Teile der XSLT-Anfrage, die keine Relevanz besitzen, ignoriert. In [GSLS+08] wird dagegen ein Verfahren vorgestellt, das existierende XSLT-Dokumente zuvor definierter Operatoren erweitert um die Übersetzung zu optimieren.

Darüber hinaus existieren diverse Ansätze, die die Anfragen selbst aus den Schemas der XML-Dokumente generieren [WC08]. Dazu werden, häufig für Datenbanken [LH10], Überschneidungen in Schema-Dokumenten gesucht [RB01, YMHF01, MBPF09] und ein entsprechendes Konzept zum Übersetzen der Nachricht abgeleitet. Konzepte, um die automatische Übersetzung mittels klar definierter Schemabeschreibungen zu vereinfachen, wurden in [BSSGM10, AK10] veröffentlicht. Da bei diesen Ansätzen die jeweiligen Anfragen automatisiert generiert werden, sind sie nicht mehr als Anfrage-Sprachen zu verstehen, sondern eher als modellbasierte Ansätze. Diese werden in Abschnitt 2.3 näher erläutert.

2.2.5 Abgrenzung

Die hier vorgestellten Anfrage-Sprachen sind in der Lage, mehrere Dokumente mittels einer beliebigen Anzahl von Anfragen zu konvertieren. Die erzeugten Nachrichten können dabei ein beliebiges Format haben. Somit lösen sie einige der in der Problemstellung definierten Probleme. Hinzu kommt, dass sie mit dem Fokus entwickelt wurden, leicht verstanden und erlernt zu werden. Allerdings wurden sie mit dem Ziel entwickelt, Dokumente in eine andere Form zu übersetzen und nicht, um Kommunikationsprotokolle zu konvertieren. So gibt es beispielsweise keine einfache Möglichkeit, fehlende Daten bei einem Dienst anzufragen, den Status eines Dienstes zu beachten oder das zeitliche Verhalten eines Zieldienstes zu beachten. Ein solches Verhalten müsste jeweils fallspezifisch implementiert werden. Damit würden wieder die gleichen Probleme erzeugt werden, wie sie bereits bei den Middleware-Systemen beschrieben wurden.

2.3 Modelle zur Protokollkonvertierung

Es existieren unterschiedliche Modellierungsansätze, um die Unterschiede zwischen Protokollen zu beschreiben und somit eine Konvertierung der Protokolle zu ermöglichen. Diese beruhen zum größten Teil auf dem Prinzip, das in Abb. 2.6 grafisch dargestellt wird. Zunächst wird eine formale Beschreibung des Quell- und Ziel-Protokolls benötigt. Diese muss vom Entwickler angefertigt werden. Sie beschreibt nicht nur das Kommunikationsverhalten sondern auch die Syntax und Semantik der Daten. Dies wird benötigt, da die Modelle üblicherweise mit einer abstrakten Beschreibung der Daten, die in den jeweiligen Nachrichten enthalten sind, arbeiten. Es wird also eine entsprechende Schnittstellen-Beschreibung benötigt, die die Extraktion dieser Daten aus den jeweiligen Nachrichten ermöglicht. Zusätzlich müssen die Unterschiede zwischen dem Kommunikationsverhalten der Protokolle beschrieben werden. Abhängig vom jeweiligen Ansatz geschieht dies unter Umständen bereits automatisiert. Darüber hinaus muss beschrieben werden, wie die Daten transformiert werden sollen. Das heißt, es muss beschrieben werden, wie aus den abstrakten Daten des Modells die Nachrichten des Ziel-Protokolls erzeugt werden können. Dies muss für jede Nachricht des Protokolls spezifiziert werden. Aus diesen Beschreibungen kann dann von Software-Werkzeugen (z.B. Cinderella SITE [FNO05]) ein spezieller Konverter synthetisiert werden. Üblicherweise wird der entsprechende Konverter mittels eines endlichen Zustandsautomaten [HU90] beschrieben [YS97, Nie93]. Gegebenenfalls muss die Beschreibung dieses Konverters aber noch angepasst werden. Dazu können grafische Beschreibungssprachen wie SDL [GGP03] eingesetzt werden.

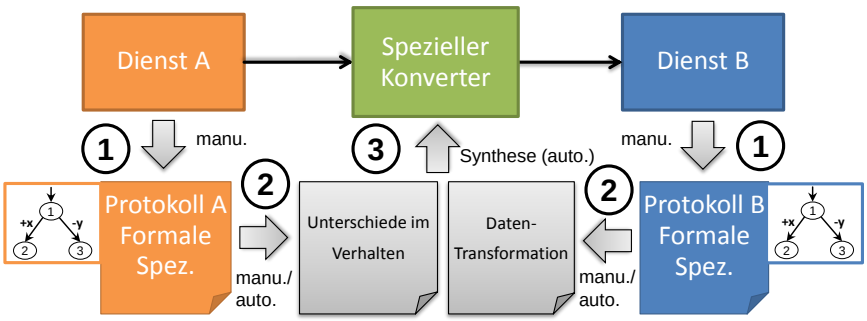


Abbildung 2.6: Schematische Darstellung des klassischen Modellierungsansatzes

Der Fokus dieser Modellierungsverfahren liegt auf der formalen Beschreibung der Protokolle. Speziell für die Beschreibung des Kommunikationsverhaltens existieren unterschiedliche Verfahren, um die Unterschiede zwischen den Pro-

tokollen automatisiert erkennen zu können, beziehungsweise die Beschreibung dieser Unterschiede zu vereinfachen. Im folgenden werden einige Interpretationen dieses Ansatzes weiter ausgeführt.

2.3.1 Model Matching

Wie bereits erwähnt, existieren verschiedene Interpretationen des oben beschrieben Ansatzes. Dabei bleibt der grundlegende Prozess der Modellierung der gleiche, lediglich die formelle Beschreibung von Quell- und Ziel-Protokoll unterscheidet sich.

In [Oku90] und [AM91] wird beispielsweise das Quell- und Ziel-Protokolls jeweils mittels eines endlichen Zustandsautomaten beschrieben. Darauf aufbauend wird automatisiert ein weiterer Zustandsautomat generiert, der als Konverter für beide Protokolle verwendet werden kann. Die Daten, mit denen die Automaten arbeiten, müssen dafür natürlich die entsprechend gleichen Namen haben.

In [PRSV98] wird eine auf regulären Ausdrücken basierende Sprache vorgestellt, die verwendet wird, um die Sprache des Ziel- und Quell-Protokolls zu beschreiben. Aus dieser kann ein Zustandsautomat für das jeweilige Protokoll generiert werden. Aus den so generierten Zustandsautomaten kann dann wiederum ein neuer Zustandsautomat generiert werden, der den Konverter repräsentiert. Eine grafische Repräsentation des Prozess finde sich in Abb. 2.7.

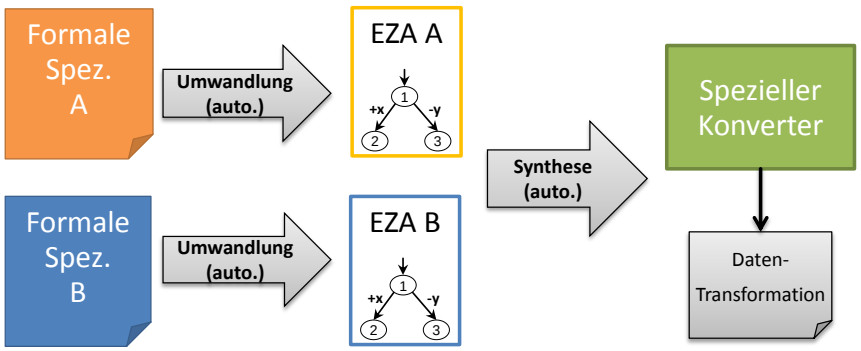


Abbildung 2.7: Schematische Darstellung der Erzeugung eines Konverters aus Regelsprachen

Darüber hinaus gibt es verschiedene softwareorientierte Ansätze [BBC05, KANK08, BBH+08, QPPS08]. Üblicherweise wird hier die Integration von Web-Diensten mittels modellbasierter Ansätze realisiert. Dazu wird grundlegend der

oben beschrieben Prozess eingesetzt. In [ZM09] wird beschrieben, wie UML-Diagramme verwendet werden können, um Ziel- und Quell-Protokoll formal zu spezifizieren. Mittels eines modellbasierten Ansatzes und vordefinierten Metamodellen, die zum Erkennen von Übersetzungsfehlern dienen, kann ein Konverter automatisiert erzeugt werden. Der Entwickler muss dazu das Kommunikationsverhalten des Ziel- und Quell-Protokoll jeweils mittels eines UML-Diagramms beschreiben. Die Unterschiede im Kommunikationsverhalten können über ein weiteres Metamodell spezifiziert werden.

2.3.2 Interface Matching

Bei diesem Ansatz wird aus den Schnittstellen-Beschreibungen der jeweiligen Protokolle die Beschreibung eines Konverters abgeleitet. Dies ist den bereits beschriebenen Modellierungverfahren relativ ähnlich, allerdings wird der Konverter bei diesem Verfahren bereits aus den Schnittstellen-Beschreibungen generiert und nicht aus den formalen Beschreibungen der Protokolle. Da die Schnittstellen-Beschreibungen aber stellenweise um Beschreibungen des Kommunikationsverhaltens erweitert werden, ist der Übergang zwischen den beiden Ansätzen fließend.

Die Protokolle werden als geschlossene Systeme betrachtet, die über eine entsprechend komplexe Schnittstellen-Beschreibung verfügen. Diese werden häufig mittels Regelsprachen [YS97] beschrieben, es existieren aber auch eher softwareorientierte Lösungen [SR02]. Dieser Ansatz wird auch auf web-basierte Service-Protokolle angewendet [BP06]. Dabei werden Beschreibungssprachen wie WSDL und BPEL zur Beschreibung der Schnittstellen verwendet.

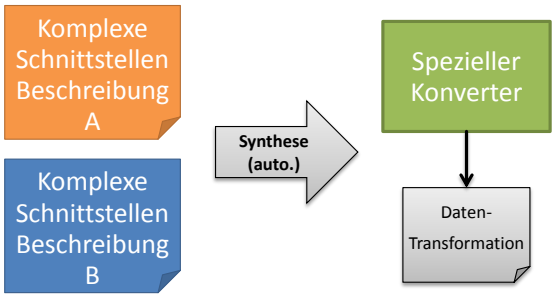


Abbildung 2.8: Schematische Darstellung der Erzeugung eines Konverters aus Beschreibung der Schnittstelle

Die Schnittstellen-Beschreibungen identifizieren sämtliche relevante Daten innerhalb der Nachrichten, spezifizieren ihren Typ und legen gegebenenfalls be-

reits erlaubte Reihenfolgen von Anfragen an den jeweiligen Dienst fest. Aufbauend auf diesen Beschreibungen können dann die Unterschiede zwischen zwei Protokollen beschrieben werden. Dazu müssen die Daten, die in beiden Protokollen gleich sind, entweder über die gleichen Namen verfügen oder aber vom Entwickler entsprechend zugeordnet werden. Da die Schnittstellen-Beschreibungen bereits Informationen über das Kommunikationsverhalten enthalten können, kann so das Kommunikationsverhalten automatisiert übersetzt werden. Das Ergebnis der Adapter-Synthese ist üblicherweise ein endlicher Zustandsautomat, der den speziellen Konverter darstellt (siehe Abb. 2.8).

Abhängig von der Komplexität der jeweiligen Beschreibungen kann eine Prüfung auf Korrektheit und Vollständigkeit durchgeführt werden. Da dies wiederum auf den entsprechenden Beschreibungen der Schnittstellen beruht, können Fehler im Modell zu entsprechenden Fehlern in der Übersetzung führen. In [PAHSV02] wird daher ein Ansatz vorgestellt, der mittels der Spieltheorie überprüft, ob unter zuvor definierten Voraussetzungen ein Konverter basierend auf einer Schnittstellen-Beschreibung erzeugt werden kann. Darüber hinaus erlaubt der in [NBM+07] vorgestellte Ansatz falsche Zuordnungen automatisiert zu erkennen und diese, basierend auf einem so genannten Mismatch Tree, zu korrigieren.

2.3.3 Message Sequence Charts

In [RTTZ04, Zve04] wird ein Ansatz vorgestellt, bei dem Sequenz-Diagramme verwendet werden, um das Quell- und Ziel-Protokoll formal zu beschreiben. Mittels Sequenz-Diagrammen können Abfolgen von Anfragen beschrieben werden. Der Ansatz verwendet sowohl reguläre Message Sequence Charts [LL92] als auch High Message Sequence Charts [MR97]. Dabei wird das gesamte Kommunikationsverhalten der beteiligten Protokolle mittels Sequenz-Diagramme beschrieben. Die Autoren konzentrieren sich in ihrer Arbeit auf sehr einfache Protokolle, das heißt es werden zwar Daten übertragen, die Konvertierung ist aber unabhängig von den übertragenen Daten.

Um eine Konvertierung zu ermöglichen, müssen die Nachrichten, die übersetzt werden sollen, über die gleichen Nachrichtennamen verfügen. Bei den Nachrichten selbst wird zwischen eingehenden, ausgehenden und geteilten Nachrichten unterschieden. Geteilte Nachrichten sind dabei Nachrichten, die von einem Prozess an einen anderen gesendet werden. Bei eingehenden und ausgehenden Nachrichten ist jeweils nur ein Prozess beteiligt.

Die von den Autoren veröffentlichte Implementierung erlaubt die Generierung eines endlichen Zustandsautomaten aus den so angelegten Diagrammen. Dieser repräsentiert einen spezifischen Konverter. Analog zu den anderen Ansätzen muss natürlich auch hier eine entsprechende Schnittstellen-Beschreibung

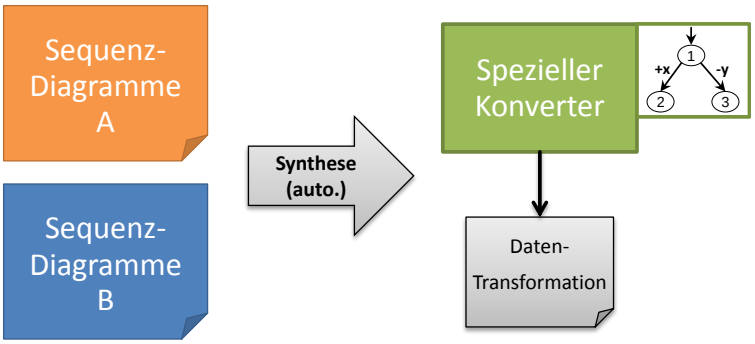


Abbildung 2.9: Schematische Darstellung der Erzeugung eines Konverters aus mehreren Sequenz-Diagrammen

vorliegen, die die Daten in den Nachrichten identifiziert. Außerdem muss eine entsprechende Komponente zur Datentransformation existieren. Dieser Ansatz bringt den Vorteil mit sich, dass bereits Methoden existieren [LKJ+05] um die Vollständigkeit und Konsistenz der Beschreibungen zu prüfen. In Abb. 2.9 findet sich eine grafische Repräsentation des Prozesses.

2.3.4 Tripel-Graph-Grammatik

Die Tripel-Graph-Grammatik (TGG) ist eine spezielle Form der Graphgrammatik und wurde bereits in der Mitte der Neunziger von Andy Schürr entwickelt [Sch94]. Der Ansatz verwendet drei Graphen. Der erste Graph (Mustergraph) beschreibt das Ausgangssystem, der zweite Graph (Ersetzungsgraph) das Zielsystem und der Dritte die Unterschiede zwischen den beiden Graphen. Der dritte Graph enthält also die Graphersetzungsregeln. Da die Produktionsregeln einer TGG monoton sind, müssen alle Knoten des Mustergraphen auch im Ersetzungsgraphen vorkommen. Das Löschen von Knoten ist somit nicht möglich. Dadurch müssen Transformationen aber nur in eine Richtung beschrieben werden, das heißt wird eine Änderung von Mustergraph zu Ersetungsgraph beschrieben, kann diese im Umkehrschluss auf den Ersetzungsgraph angewendet werden um den Mustergraph zu erhalten. Hinzu kommt, dass mittlerweile Erweiterungen existieren, die die Laufzeit des Ansatzes verbessern [KLKS10].

Die TGG könnte also eingesetzt werden, um Unterschiede im Kommunikationsverhalten von Protokollen zu beschreiben. Dabei würde der Mustergraph das Kommunikationsverhalten des Quell-Protokolls beschreiben, während der Ersetzungsgraph das Kommunikationsverhalten des Ziel-Protokolls beschreibt. Durch den dritten Graph würden dann die Unterschiede zwischen den Kom-

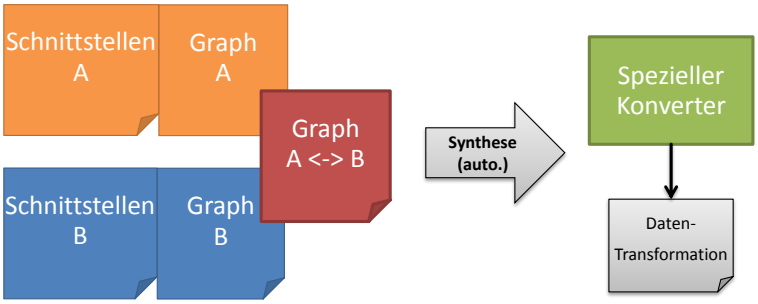


Abbildung 2.10: Schematische Darstellung der Erzeugung eines Konverters aus Tripel-Graph-Grammatik

munikationsverhalten der beiden Protokolle beschrieben werden. Dies müsste in diesem Fall nur in eine Richtung geschehen, da der Umkehrschluss sich automatisch ergibt. In Abb. 2.10 findet sich eine grafische Repräsentation des Prozesses.

Eine Schnittstellen-Beschreibung, die die relevanten Daten aus den Nachrichten gewinnt, müsste in diesem Fall für beide Protokolle angefertigt werden. Darüber hinaus müsste beschrieben werden, wie aus den erzeugten Daten die Nachrichten des Ziel-Protokolls erzeugt werden.

2.3.5 Abgrenzung

Die vorgestellten Ansätze folgen alle dem gleichen grundlegendem Prozess. Aus einer formalen Beschreibung des Ziel- und Quell-Protokolls wird ein spezifischer Konverter synthetisiert. Abhängig vom jeweiligen Ansatz muss dazu eine entsprechend komplexe Schnittstellenbeschreibung vorliegen. Da die Modelle mit abstrakten Daten arbeiten muss die Datentransformation separat beschrieben werden. Abhängig vom jeweiligen Ansatz müssen außerdem die Unterschiede im Kommunkationsverhalten beschrieben oder angepasst werden.

Die Ansätze unterscheiden sich lediglich in der Form der Modelle. Einige Ansätze, wie das Interface Matching, konzentrieren sich auf die Schnittstellen-Beschreibungen, während andere Ansätze wie die TGG sich auf das Beschreiben der Verhaltensunterschiede konzentrieren.

Die Ansätze selbst sind sehr mächtig und erlauben eine komplexe Beschreibung der Protokolle und ihrer Unterschiede. Allerdings sind viele der Modelle relativ abstrakt, so dass eine gewisse Einarbeitungszeit vorausgesetzt werden muss. Darüber hinaus können die Beschreibungen selbst, abhängig von dem jeweiligen Protokoll, sehr komplex und wartungsaufwändig werden. Häufig

existieren zwar Methoden um automatisiert Modelle aus anderen Beschreibungen zu erzeugen, allerdings setzen diese die Konsistenz der verwendeten Beschreibungen voraus, so dass die erzeugten Modelle im Zweifelsfall von einem Entwickler überprüft werden müssen.

2.4 Zusammenfassung

In diesem Kapitel wurden verschiedene Übersetzungsverfahren für Service-Protokolle vorgestellt. Während die Middleware-Systeme sehr beliebt sind, erzeugen sie durch ihre Adapter-Schichten einen hohen Integrations- und Wartungsaufwand. Anfrage-Sprachen erlauben dagegen eine relativ simple Übersetzung von Nachrichten, können das Kommunikationsverhalten aber nicht zufriedenstellend übersetzen. Die existierenden modellbasierten Ansätze erlauben dies zwar, abstrahieren die Protokoll-Beschreibungen aber so stark, dass der Intialaufwand zum Erlernen des jeweiligen Verfahrens aber auch der Wartungsaufwand relativ hoch sein kann.

3 Protokollanalyse und Szenarien

In diesem Kapitel wird eine Analyse verschiedener Protokolle vorgestellt, die die Grundlage für die Beantwortung der Forschungsfrage liefert. Das Ergebnis der Analyse sind unterschiedliche Konvertierungsszenarien, die ebenfalls in diesem Kapitel ausgeführt werden. Diese bilden wiederum die Grundlage des Modells, dass in Kapitel 4 beschrieben wird.

3.1 Übersicht

Der im Rahmen dieser Arbeit entwickelte Ansatz kann auf eine Gruppe von Protokollen mit ähnlichen Eigenschaften angewendet werden. Dies sind die so genannten Service-Protokolle. Um eine Konvertierung zwischen Service-Protokollen zu ermöglichen wurden verschiedene Protokolle dieser Gruppe analysiert. Die dazu angewandte Methodik wird in diesem Abschnitt beschrieben. Neben reinen Service-Protokollen wurden auch Anforderungen von Sensor-Protokollen aufgenommen. Dieser Abschnitt enthält ebenfalls die Definition dieser Protokoll-Gruppen und beschreibt, welche Anforderungen und Einschränkungen sich mit ihnen ergeben. In Kapitel 6 werden darüber hinaus Protokolle aus einer anderen Domäne evaluiert. Diese sind nicht Teil dieser Analyse, da die Analyse das Ziel hat Konvertierungsszenarien für die Domäne der Service-Protokolle zu identifizieren.

3.1.1 Service-Protokolle

Unter Service-Protokollen werden im Rahmen dieser Arbeit Protokolle verstanden, die den Zugriff auf einen Dienst mittels einer klar definierten Schnittstelle beschreiben. Diese Beschreibung enthält, neben der Beschreibung des Datenformats, auch die Beschreibung des Kommunikationsverhaltens. Die Beschreibung selbst kann in beliebigen Formaten vorliegen. Diese reichen von textuellen Beschreibungen wie OGC- oder RFC-Dokumente über simple Schnittstellenbeschreibungen wie WSDL bis hin zu komplexen Modellen wie die SDL-Beschreibung eines endlichen Automaten. Weiterhin wird davon ausgegangen, dass die Dienste, die diese Protokolle verwenden, entweder Informationen anbieten oder aber Funktionen zur Verfügung stellen.

Da Service-Protokolle üblicherweise auf der siebten Schicht des OSI-Modells [Int96] eingeordnet werden, wird die Handhabung von Sitzungen und der Transport der Nachrichten nicht betrachtet. Der Ansatz beschränkt sich somit auf das Analysieren und Konvertieren der übertragenen Nachricht und beachtet nicht die Verfahren, die zur Übermittlung eingesetzt werden. Es wird also davon aus-

gegangen, dass die Nachrichten des Protokolls an einen Konverter weitergereicht werden können, ohne den Kommunikationsprozess zu stören. Der Konverter übernimmt das Übersetzen der Nachrichten und des Kommunikationsverhaltens und gibt eine Liste neuer Nachrichten zurück. Für die Forschungsfrage bedeutet diese Einschränkung, dass eine Verringerung des Aufwands nur für diese Klasse von Protokollen versprochen werden kann.

Von den Diensten können beliebige Kommunikationstechnologien eingesetzt werden. Die Kommunikation kann sowohl synchron als auch asynchron erfolgen. Das Daten-Format der Nachricht unterliegt keinen Einschränkungen und kann beispielsweise textbasiert aber auch binär sein. Das Kommunikationsverhalten von Service-Protokollen basiert üblicherweise auf Anfrage/Antwort-Mustern. Allerdings können zwischen den einzelnen Anfragen beliebig komplexe Abhängigkeiten bestehen. Das heißt, dass eine Nachricht eine beliebig komplexe Reihe von Nachrichten, die abhängig vom Inhalt der jeweiligen Nachrichten ist, nach sich ziehen kann. Auch kann eine zu erwartende Nachricht abhängig vom internen Zustand des Quell- oder Zieldienstes sein. Beispiele für Service-Protokolle sind SOAP [GHM+07a], JSON-RPC [Cro06] oder REST [Roy00].

3.1.2 Sensor-Protokolle

Neben den Service-Protokollen wurden ebenfalls Sensor-Protokolle betrachtet. Hierbei wurden nur Protokolle betrachtet, die selbst auch in die Gruppe der Service-Protokolle eingeordnet werden können. Das heißt, es wurden nur Protokolle betrachtet die mittels klar definierter Schnittstellen den Zugriff auf Daten oder Funktionen erlauben. Die im vorherigen Schritt beschriebenen Einschränkungen treffen somit auch auf diese Klasse von Protokollen zu. Allerdings bringt der Umstand, dass das Protokoll auf einem Gerät mit beschränkten Ressourcen ausgeführt wird, weitere Anforderungen an die Konvertierung mit sich. Als Besonderheit werden hier zeitliche Bedingungen betrachtet. Diese werden bedingt durch die Tatsache, dass Sensoren nicht immer aktiv sein müssen. Somit kann das Übertragen einer Nachricht an zeitliche Bedingungen gebunden sein.

3.1.3 Methodik

Die zum Analysieren der verwendeten Service-Protokolle anwendete Methodik wird in diesem Abschnitt beschrieben. Zunächst wurden verschiedene Protokolle ausgewählt, die in die Gruppe der Service-Protokolle eingeordnet werden können. Diese sind das SCAI-Protokoll der SCAMPI-Middleware, das Sensor Web Enablement-Framework (SWE), das SOAP-Framework und das Protokoll

des flightradar24-Dienstes. Kriterien bei der Auswahl dieser Protokolle waren dabei:

- Service-Protokoll: Entsprechend der Einschränkungen aus 3.1.1
- Dokumentation: Qualität und Vollständigkeit
- Versionen: Vergleichbare Versionen eines Protokolls vorhanden
- Funktionsumfang: Protokolle haben ähnlichen Funktionsumfang
- Implementierung: Implementierung erlaubt Test mit Realsystem
- Popularität: Verwendung bekannter Protokolle

Bei der Auswahl musste entweder das *Versionen*-Kriterium oder das *Funktionsumfang*-Kriterium zutreffen. Außerdem wurden nur das SWE- und SOAP-Framework wegen ihrer Popularität gewählt. Das Flightradar-Protokoll wurde trotz der eingeschränkten Dokumentation aufgenommen. Hier wurde eine Ausnahme gemacht, da das Protokoll relativ simpel ist und bereits sehr gut über die Nachrichtenstruktur und Funktionsbeispiele dokumentiert ist. Die restlichen Kriterien treffen auf alle verwendeten Protokolle zu.

Protokoll	Dokumentation	Versionen	Funktionsumfang	Implementierung	Popularität
SCAI	ausführlich	2	-	ja	niedrig
SWE	ausführlich	-	ähnlich SCAI	ja	hoch
SOAP	ausführlich	2	-	ja	hoch
Flightradar	beschränkt	-	ähnlich SCAI	ja	mittel

Tabelle 3.1: Übersicht aller analysierten Protokolle

Um die Konvertierungsszenarien zu definieren, wurden die Protokolle auf ihre Unterschiede untereinander hin untersucht. Dabei wurden sowohl unterschiedliche Versionen des gleichen Protokolls betrachtet, als auch die Protokolle untereinander, wenn diese einen ähnlichen Satz an Funktionen bieten. Diese Analyse fand durch einen Vergleich der Informationen aus den jeweiligen Dokumentationen aber auch durch Tests mit den jeweiligen Implementierungen statt. Die Analyse konzentriert sich dabei auf die folgenden Kriterien:

- Strukturelle Unterschiede
- Unterschiede im Funktionsumfang
- Aus den Unterschieden resultierende Verhaltensänderungen

Es wurde darauf geachtet, dass ein im Rahmen der Protokoll-Spezifikationen akzeptables Kommunikationsverhalten erzeugt wird. Das bedeutet auch, dass wenn beispielsweise eine Nachricht empfangen wird, für die eine Konvertierung nicht möglich ist, eine entsprechende Fehlerbenachrichtigung an den Quelldienst gesendet wird, wenn dies nötig ist.

Sämtliche Unterschiede, die zwischen den Protokollen und Protokollversionen gefunden wurden, wurden tabellarisch festgehalten. Dabei wurde ebenfalls festgehalten, welche Änderungen nötig wären um eine Konvertierung zu ermöglichen. Bei komplexen Verhaltensänderungen wurden diese zusätzlich in Form von Sequenz-Diagrammen festgehalten. Die folgenden Arten der Konvertierung wurden dabei festgestellt und entsprechend festgehalten:

- Daten-Transformation: Änderungen des Daten-Formats
- Neue Nachricht: Hinzufügen zusätzlicher Nachrichten
- Anfragen senden: Rekonstruieren fehlender Daten
- Aufteilen: Erzeugen mehrerer Nachrichten
- Löschen: Entfernen einer Nachricht
- Speichern: Speichern einer Nachricht zur späteren Verwendung
- Kombinieren: Erzeugen einer neuen Nachricht aus mehreren Gespeicherten
- Bedingungen: Überprüfen welche Konvertierung ausgeführt werden muss

Die Tabelle hält dabei den Grund der jeweiligen Änderung fest, den Typ der Änderung, die Anzahl der betroffenen Funktionen, sowie die eben genannten Arten der Konvertierung.

Aus den gesammelten Informationen wurden Muster abgeleitet. Diese sollten die nötigen Konvertierungen möglichst einfach beschreiben. Die Muster sollten dabei nicht zwangsläufig atomar sein, sondern die häufigsten Konvertierungen so beschreiben, dass sie den Unterschied zwischen zwei Protokollen eindeutig beschreiben. Die Muster selbst sollten dabei alle betrachteten Szenarien abdecken. Die Protokollanalysen finden sich in Kapitel 3.2. Neben einer Übersicht über die jeweilige Analyse enthält dieses Kapitel jeweils die festgestellten Änderungen der Struktur und des Funktionsumfangs, beschreibt die daraus resultierenden Änderungen des Kommunikationsverhaltens und fasst die nötigen Schritte zur Konvertierung übersichtlich zusammen.

3.2 Protokollanalyse

Wie bereits erwähnt, wurde im Rahmen dieser Arbeit eine Analyse verschiedener Protokolle durchgeführt. Der Fokus lag dabei auf den im letzten Kapitel definierten Service-Protokollen. Bei der Analyse selbst wurden sowohl die Unterschiede in verschiedenen Versionen des gleichen Protokolls betrachtet als auch die Unterschiede zwischen Protokollen die für ähnliche Anwendungszwecke entwickelt wurden.

3.2.1 SCAI-Protokoll

Beim SCAI-Protokoll handelt es sich um ein Protokoll, das zur Übertragung von Sensordaten, zur Konfiguration und Administration von Sensoren, aber auch zur Administration von Middleware-Systemen verwendet werden kann [BKW+09, KBNB11]. Es wurde im Rahmen des SCAMPI-Projektes entwickelt und ist Teil der entsprechenden Sensor-Middleware. Der Teil des Protokolls, der auf Sensoren angewendet werden kann, wurde so konzipiert, dass er auch für Dienste aller Art funktioniert. Das Protokoll erlaubt, neben dem Administrieren der gespeicherten Sensorbeschreibungen auch die Beschreibung von Plänen zur Sensordatenverarbeitung. Es handelt sich beim SCAI-Protokoll somit um ein Service-Protokoll, dass Sensor-Funktionen zur Verfügung stellt.

SCAI verfügt über zwei Darstellungsformen: Eine XML-Darstellung (siehe Abb. 3.1), die alle verfügbaren Befehle hierarchisch anordnet und durch ein entsprechendes XML Schema für Typsicherheit sorgt, sowie eine Textdarstellung (siehe Abb. 3.2), die über den gleichen Funktionsumfang wie die XML-Darstellung verfügt, aber durch eine reduzierte Zeichenzahl die Nachrichtengröße extrem verringert [BB11].

Das Kommunikationsverhalten beschränkt sich auf Anfrage/Antwort-Muster. Allerdings können Abhängigkeiten zwischen diesen Mustern auftreten. Wird beispielsweise versucht, einen neuen Sensor zu registrieren, zu dem der entsprechende Sensortyp nicht existiert, wird eine entsprechende Fehler-Nachricht zurück gesendet. Daraufhin wird versucht, den entsprechenden Sensortyp zu registrieren, was wiederum wegen eines fehlenden Datenstromtyps fehlschlagen kann. Somit gibt es keine konkreten Abhängigkeiten zwischen den einzelnen Anfragen, allerdings kann mit einem entsprechenden Kommunikationsverhalten gerechnet werden, wenn ein Dienst beziehungsweise eine Applikation entsprechend der SCAI-Spezifikation implementiert wurde. Nachrichten können zudem asynchron übertragen werden. Anfrage und Antwort enthalten für jeden Befehl eine so genannte OperationID, die die Zuordnung erlaubt.

Das Datenformat, sowie das Kommunikationsverhalten sind durch die Dokumentation des SCAMPI-Projekts ausreichend beschrieben [BB11, KB10]. Au-

```
<SCAI xmlns='http://xml.offis.de/schema/SCAI-2.0'>
    <Payload>
        <ControlData>
            <SensorRegistryControl>
                <getSensor OperationID="4815162342">
                    <Sensor>
                        <NameReference>
                            <SensorName>Temp11</SensorName>
                            <SensorDomainName>SmartHome12</SensorDomainName>
                        </NameReference>
                    </Sensor>
                </getSensor>
            </SensorRegistryControl>
        </ControlData>
    </Payload>
</SCAI>
```

Abbildung 3.1: SCAI-Beispiel (XML)

```
rg_sensr rn=Temp11 dn=SmartHome12 id=4815162342;
```

Abbildung 3.2: SCAI-Beispiel (Text)

ßerdem erlaubt der SCAI-Stack, der Teil der SCAMPI-Middleware ist, aber als eigenständige Komponente in andere Projekte eingebunden werden kann, die Konsistenzprüfung von Nachrichten im SCAI-Format. Im Rahmen dieser Arbeit wurde das SCAI-Protokoll in der Version 1.0 und 2.0 analysiert. Version 1.0 ist die erste von den Entwicklern veröffentlichte Version des Protokolls während 2.0 die Version ist, die mit Abschluss des Projektes veröffentlicht wurde. In den folgenden Abschnitten werden diese Versionen auf ihre Unterschiede hin analysiert.

3.2.1.1 Übersicht

Das SCAI-Protokoll gliedert sich in verschiedene Funktionsgruppen. Diese sind die Gruppen *Identifikation*, *Sensordaten-Übertragung*, *Zugriffs-Administration*, sowie *Sensor-Administration* und *Middleware-Administration*. Außerdem ist ein gesonderter Funktionsblock für die Übertragung von Antworten auf Anfragen definiert. Innerhalb dieser Funktionsgruppen werden die jeweiligen Befehle definiert. Diese grundlegende Aufteilung wurde während der Änderungen am Protokoll beibehalten. Die Struktur der Nachrichten, also die Darstellung der Nachricht selbst, wurde dagegen an verschiedenen Stellen angepasst. Außerdem wurden diverse Funktionen abgewandelt, hinzugefügt, entfernt oder erweitert. Diese Änderungen werden in den folgenden Abschnitten beschrieben.

3.2.1.2 Änderungen der Struktur

Zwischen den beiden Versionen des SCAI-Protokolls wurden diverse strukturelle Änderungen durchgeführt. In der Gruppe *Identifikation* wurde beispielsweise das Element zur Identifikation einer Übertragung eine Ebene nach oben verschoben und umbenannt. Zusätzlich wurden verschiedene Elemente der umgebenden Struktur umbenannt. Darüber hinaus wurde die Anzahl, in der bestimmte Elemente vorkommen dürfen, geändert. Diese Änderungen wurden so durchgeführt, dass die Nachricht einer Version in die Nachricht einer anderen Version übersetzt werden können, ohne das beispielsweise eine Nachricht aufgeteilt werden muss. Bei einigen Funktionen kann es allerdings, abhängig von den übertragenen Daten, zu einem Verlust von Informationen kommen.

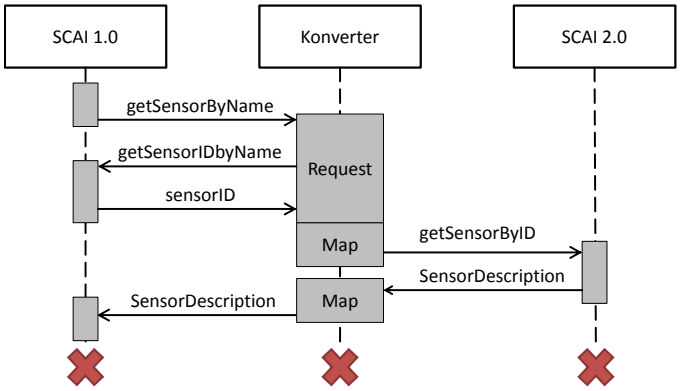


Abbildung 3.3: Prozess zur Konvertierung von SCAI-Nachrichten bei fehlender Sensor ID

Neben diesen simplen Änderungen wurde eine Änderungen des Referenzsystems für Sensoren durchgeführt. In Version 1.0 wurden Sensoren, genau wie alle anderen Objekte auch, mittels einer ID oder einem eindeutigen Namen identifiziert. In Version 2.0 geschieht die Identifikation von Sensoren allerdings mittels einer eindeutigen ID oder einer eindeutigen Kombination aus Sensorname und Domänenname. Diese strukturelle Änderung zieht komplexe Änderungen des Kommunikationsverhaltens nach sich. Wird in einer Nachricht des alten Formats ein Sensor mittels des Namen identifiziert, muss der Domänenname bei dem entsprechenden Quelldienst angefragt werden. Dazu muss eine Nachricht an diesen Dienst gesendet werden und die Antwort entsprechend analysiert werden. Erst dann ist eine Übersetzung möglich. Abb. 3.3 stellt die Konvertierung grafisch dar.

3.2.1.3 Änderungen des Funktionsumfangs

Neben den reinen strukturellen Änderungen wurden auch Änderungen am Funktionsumfang durchgeführt. Während in Version 1.0 noch ein Element zur Identifikation von Verschlüsselungsverfahren vorgesehen war, wurde dieses in der späteren Version entfernt. Bei einer Transformation zu einer älteren Version des Protokolls müsste diese Information also rekonstruiert werden. Dazu müsste wiederum mit dem Quelldienst kommuniziert werden.

In der Gruppe *Zugriffs-Administration* wurden drei neue Befehle hinzugefügt. Mittels dieser Funktionen kann der Zugriff von Benutzern auf Sensoren administriert werden. Da diese Befehle nicht in der älteren Version existieren, müsste die Nachricht beim Übersetzen verworfen werden. Außerdem sollte eine entsprechende Fehlernachricht an den Quelldienst gesendet werden. Diese muss die OperationID aus der ursprüngliche Nachricht enthalten.

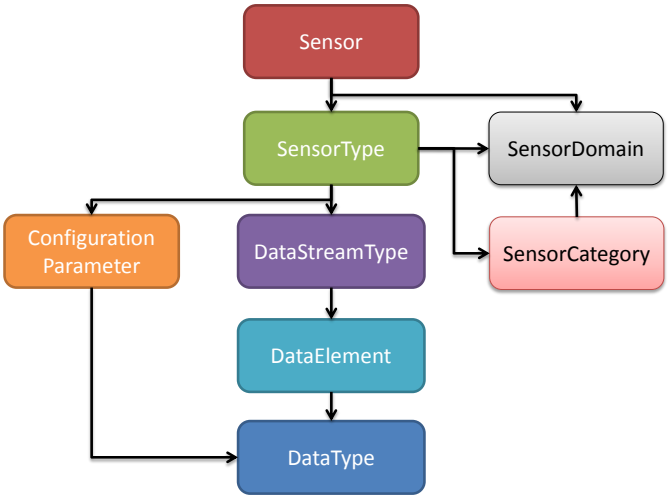


Abbildung 3.4: Schematische Darstellung des SCAMPI-Datenschemas

In der Gruppe *Middleware-Administration* befinden sich die komplexesten Befehle. Die meisten von ihnen werden dazu verwendet, Sensorbeschreibungen zu erzeugen. Diese Beschreibungen beruhen auf wiederverwendbaren Elementen, die aufeinander aufbauen. So wird ein Sensor mittels eines Sensortypes beschrieben. Dieser beruht auf einer Beschreibung des Datenstroms. Der Datenstrom wird wiederum durch Datenelemente beschrieben, die auf einem Datentyp beruhen. Hinzu kommen noch Beschreibungen für Konfigurationsparameter, Domänen und Kategorien. Eine vollständige Beschreibung des Schemas findet sich in Abb. 3.4. Für jedes Element existieren Befehle zum Erzeugen, Löschen, Aktualisieren, Anfragen und Auflisten. Hinzukommen spezielle Befehle, um

beispielsweise einen Sensor einer Kategorie hinzuzufügen. Theoretisch können all diese Befehle in eine neuere Version übersetzt werden. Allerdings wurden die zu Grunde liegenden Elemente verändert. Da einige dieser Änderungen optional sind, ist eine Konvertierung damit nicht ausgeschlossen. Gerade die Befehle zum Löschen, Anfragen und Auflisten bedürfen keiner Änderung. Die Befehle zum Erzeugen und zur Aktualisierung müssen allerdings gegebenenfalls angepasst werden.

In der neuen Version gibt es mehr Datentypen zur Auswahl. Wird ein neuerer Datentyp verwendet, ist eine Konvertierung in das alte Format nicht mehr möglich und ein entsprechendes Fehlerverhalten muss erzeugt werden. Außerdem können Sensortypen bereits während des Erzeugens Kategorien und Domänen zugeordnet werden. Die gleichen optionalen Zuordnungen wurden für Kategorien hinzugefügt. Die Beschreibung von Sensoren legt außerdem nun fest, dass eine Referenz auf eine Domäne und einen Sensortypen angegeben werden muss.

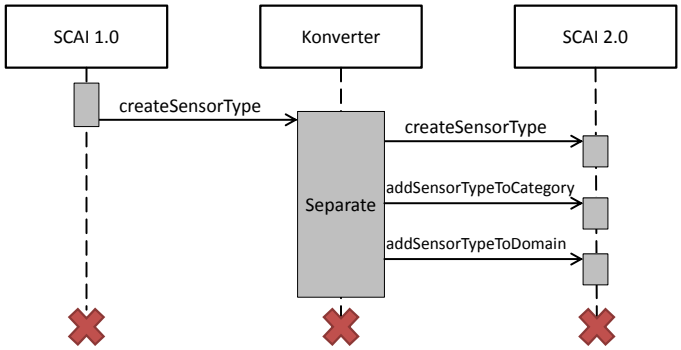


Abbildung 3.5: Prozess zur Konvertierung beim Erzeugen von SCAI-Sensortypen

Diese zusätzlichen Zuordnungen können gehandhabt werden, indem zunächst das Basiselement wie gewohnt erzeugt wird. Daraufhin können mittels weiterer Nachrichten die fehlenden Zuordnungen erzeugt werden. Dieser Prozess wird in Abb. 3.5 grafisch dargestellt. Bei den Befehlen zum Anfragen und Auflisten gilt es hier noch zusätzlich zu beachten, dass die entsprechenden Antworten natürlich die Elemente in ihrer neuen Form beinhalten müssen. Bei einer Übersetzung von Antworten des alten Formats in das neue müssen die fehlenden Informationen also durch entsprechende Anfragen rekonstruiert werden.

Ein weiteres Verhalten, das beachtet werden sollte, tritt auf, wenn eines der erwähnten Elemente nicht erzeugt werden kann. Wird die Nachricht, wie in Abb. 3.5 dargestellt, aufgeteilt und schlägt bereits der erste Befehl an den SCAI 2.0 Dienst fehl, gibt es keinen Grund mehr, die restlichen Befehle zu senden. Die folgenden Nachrichten würden zwar kein fehlerhaftes Verhalten im Dienst er-

zeugen, allerdings würde jede dieser Nachrichten vom SCAI 2.0 Dienst mit einer entsprechenden Fehlernachricht beantwortet werden. Diese müssten wiederum entsprechend vom Konverter gehandhabt werden. Somit sollte nach dem Senden des ersten Befehls die empfangene Antwort auf ihren Inhalt geprüft werden und im Fehlerfall auf das Senden der Folgenachrichten verzichten werden.

Neben diesen zusätzlichen Referenzen wurden beim Erzeugen einer Domäne die möglichen Referenzen auf Sensortypen und Kategorien entfernt. Das bedeutet, dass in diesem Fall die Nachricht des alten Protokolls mehr Informationen enthält als die des neuen. Hier kann aber das gleiche Konvertierungsverhalten angewendet werden wie oben beschrieben, nur entsprechend in die andere Richtung.

In der Gruppe *Middleware-Administration* wurden außerdem noch diverse Funktionen zum Erzeugen von Verarbeitungsplänen für Sensordaten hinzugefügt. Da diese Funktionen nicht in Version 1.0 existieren, ist eine Übersetzung nicht möglich. Tritt eine entsprechende Nachricht auf, muss diese entfernt werden und eine Fehlernachricht an den Quelldienst gesendet werden.

3.2.1.4 Zusammenfassung

Eine Übersicht der Unterschiede zwischen den beiden Protokollversionen findet sich in Tab. 3.2. Außerdem enthält die Tabelle den Typ der Änderungen, sowie die Funktionen und die Anzahl der betroffenen Funktionen. Darüber hinaus wurde festgehalten, welche Änderungen nötig sind, um eine Konvertierung durchzuführen. Dabei wurde sowohl die Konvertierung von der älteren Version des Protokolls in die Neuere, als auch die umgekehrte Konvertierung betrachtet.

Die Tabelle zeigt, dass viele strukturelle Änderungen gemacht wurden, die aber relativ einfach gehandhabt werden können. Die Anpassung der Sensor-Referenzen und der zu Grunde liegenden Datentypen bedarf aufwändigerer Methoden zur Anpassung. Da die neuen Funktionen nicht übersetzt werden können, muss hier mit relativ komplexen Mitteln ein entsprechendes Fehlerverhalten erzeugt werden.

3.2.2 SCAI zu SWE-Konvertierung

In diesem Abschnitt werden die Unterschiede zwischen dem SCAI-Protokoll, das bereits im vorherigen Abschnitt beschrieben wurde, und dem Sensor Web Enablement (SWE) Framework untersucht. Dazu wurden das SCAI-Protokoll in der Version 2.0 und das SWE-Framework in der Version 1.0.0 verwendet. Diese Protokolle wurden ausgewählt, da beide über einen ähnlichen Satz an Funktionen verfügen [Fun10].

Beschreibung	Typ	Betroffen	Transf.	Neue Nachricht	Anfrage	Aufteilen	Löschen	Speichern	Kombinieren	Prüfung
Aufbau	Struktur	Alle	ja	nein	nein	nein	nein	nein	nein	nein
Sensor Referenz	Struktur	11	ja	nein	ja	nein	nein	nein	nein	nein
Encrytion Elem.	Struktur	Alle	ja	nein	ja	nein	nein	nein	nein	nein
Sensor-Zugriff	Neue F.	3	nein	ja	nein	nein	ja	nein	nein	nein
DataType	Geänderte F.	3	nein	ja	nein	nein	ja	nein	nein	ja
SensorType	Geänderte F.	3	ja	nein	ja	ja	nein	nein	nein	ja
SensorCategory	Geänderte F.	3	ja	nein	ja	ja	nein	nein	nein	ja
Sensor	Geänderte F.	3	ja	nein	ja	ja	nein	nein	nein	ja
SensorDomain	Geänderte F.	3	ja	nein	ja	ja	nein	nein	nein	ja
Operators	Neue F.	25	nein	ja	nein	nein	ja	nein	nein	nein

Tabelle 3.2: Übersicht aller Änderungen innerhalb des Protokolls

3.2.2.1 Sensor Web Enablement

Mit Sensor Web Enablement (SWE) entwickelt das Open Geospatial Consortium, Inc. (OGC) einen Standard, um die Kommunikation von Sensoren und Sensorsystemen über das Web zu ermöglichen, um einen Austausch von Sensordaten zu erlauben[BPRD07]. Dazu wird eine einheitliche Beschreibungssprache für Sensoren definiert. Zusätzlich werden Kommunikationswege, Schnittstellen und Kommunikationsverhalten für den Zugriff auf Sensordaten definiert. Übertragene Daten werden mittels der Beschreibungssprache XML formatiert. Eine Beispiel-Nachricht findet sich in Abb. 3.6. Von der OGC wird dabei keine konkrete Implementierung zur Verfügung gestellt, sondern eine umfassende Beschreibung des Standards. Durch die weite Verbreitung des Standards existieren mittlerweile verschiedene Implementierungen, wie beispielsweise die SWE-Implementierung der SCAMPI-Middleware [FBK+11] oder die Implementierung der 52°North Initiative [BFJ10, 52°12]. SWE konzentriert sich dabei auf Prozesse zum Finden von Sensoren und Sensordaten, zum Feststellen von Sensorfunktionalitäten sowie auf die Übertragung von Sensordaten. Darüber hinaus soll der Standard den geo-basierten Zugriff auf Sensordaten ermöglichen. Das heißt, das System soll unabhängig vom eigentlichen Sensor den Zugriff auf Sensordaten eines bestimmten geografischen Gebiets ermöglichen. Darüber hinaus wird das Erzeugen von Alarmen, basierend auf der Auswertung von Sensordaten, ermöglicht. Somit stellt das SWE-Framework ein für Sensoren konzipiertes Service-Protokoll dar.

```
<ns:InsertObservation service="SOS" version="1.0.0"
                      xmlns:gml="http://www.opengis.net/gml"
                      xmlns:swe="http://www.opengis.net/swe/1.0.1"
                      xmlns:om="http://www.opengis.net/om/1.0"
                      xmlns:ns="http://www.opengis.net/sos/1.0">
    <ns:AssignedSensorId>Weather.Station1234</ns:AssignedSensorId>
    <om:Observation>
        <om:samplingTime>
            <gml:TimeInstant>
                <gml:timePosition>2010-09-19T16:08:31.443+00:00</gml:timePosition>
            </gml:TimeInstant>
        </om:samplingTime>
        <om:procedure/>
        <om:observedProperty/>
        <om:featureOfInterest/>
        <om:result>
            <swe:SimpleDataRecord>
                <swe:field name="Wind/windChill">
                    <swe:Text>
                        <swe:value>4</swe:value>
                    </swe:Text>
                </swe:field>
                <swe:field name="Wind/windDirection">
                    <swe:Text>
                        <swe:value>180</swe:value>
                    </swe:Text>
                </swe:field>
                <swe:field name="Wind/windSpeed">
                    <swe:Text>
                        <swe:value>14.48</swe:value>
                    </swe:Text>
                </swe:field>
            </swe:SimpleDataRecord>
        </om:result>
    </om:Observation>
</ns:InsertObservation>
```

Abbildung 3.6: SWE-Beispiel

3.2.2.2 Übersicht

SWE ist in 7 Unterprojekte aufgeteilt. Da in dieser Analyse die Überschneidungen zum SCAI-Protokoll betrachtet werden, wurden nur 3 der Unterprojekte einbezogen. Die verbleibenden 4 Projekte beschreiben lediglich Features, die im SCAI-Protokoll nicht vorhanden sind [Fun10]. Diese werden darüber hinaus in einer sehr abstrakten Form beschrieben, so dass ein direkter Vergleich nicht möglich ist. Die Teile des SWE-Frameworks, die verglichen werden können, sind das *Observations & Measurements Schema (O&M)* [Cox11], die *Sensor Model Language (SensorML)* [BR07] und der *Sensor Observations Service (SOS)* [BSE12]. Dabei beschreibt O&M wie Sensordaten übertragen werden, SensorML wie Sensoren beschrieben werden und SOS wie mit der Middleware kommuniziert wird.

Im Rahmen der Analyse wurden die Befehle zum Übertragen von Sensordaten, die Befehle zum Erzeugen neuer Sensorbeschreibungen inklusive der Beschreibung des Sensortyps und des Datenstromtyps, sowie die entsprechenden Befehle zum Anfragen von Sensorbeschreibungen betrachtet. Außerdem wurden die Antworten auf diese Anfragen betrachtet. Es existieren zwar in beiden Protokollen Befehle, die nicht übersetzt werden können und somit mit einem entsprechendem Fehlerverhalten versehen werden müssten. Da diese Befehle allerdings nur zusätzliche Funktionen bieten, aber nicht zum Betrieb des jeweiligen Systems nötig sind, wurde entschieden, sie nicht in die Analyse aufzunehmen.

3.2.2.3 Unterschiede zwischen den Protokollen

Wie zu erwarten war, sind die XML-Strukturen der beiden Protokolle sehr unterschiedlich. Somit muss für jede Funktion des Protokolls eine Transformation der Daten durchgeführt werden. In den Befehlen zum Anfragen von Sensordaten beider Protokolle sind allerdings alle nötigen Informationen enthalten, so dass eine Konvertierung von einem Nachrichtenformat in das Andere problemlos möglich ist.

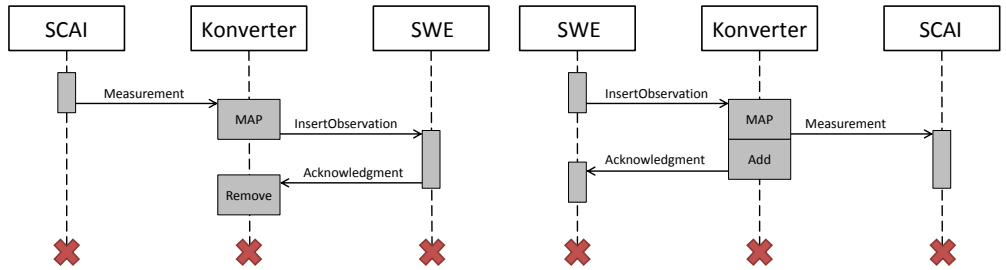


Abbildung 3.7: Prozess zur Konvertierung zwischen SCAI und SWE beim Übertragen von Sensordaten

Bei der Übertragung von Sensordaten ist dies nicht möglich, da es einen grundlegenden Unterschied gibt: Das SCAI-Protokoll sendet im Gegensatz zu SWE keine Bestätigungsnachricht zurück, sobald die Daten empfangen wurden. Bei einer SCAI zu SWE Konvertierung muss also die Bestätigungsnachricht entfernt werden, während bei einer Konvertierung in die andere Richtung eine zusätzliche Nachricht hinzugefügt werden muss. Die Konvertierungen werden in Abb. 3.7 grafisch dargestellt.

Beim Registrieren von neuen Sensoren muss das Kommunikationsverhalten ebenfalls angepasst werden, da das Verhalten der einzelnen Protokolle sich bereits voneinander unterscheidet. Das SCAI-Protokoll sendet mehrerer Nachrichten um einen Sensor zu registrieren. Üblicherweise wird zunächst ein Datenstromtyp, dann ein Sensortyp und dann der Sensor selbst registriert. Das SWE

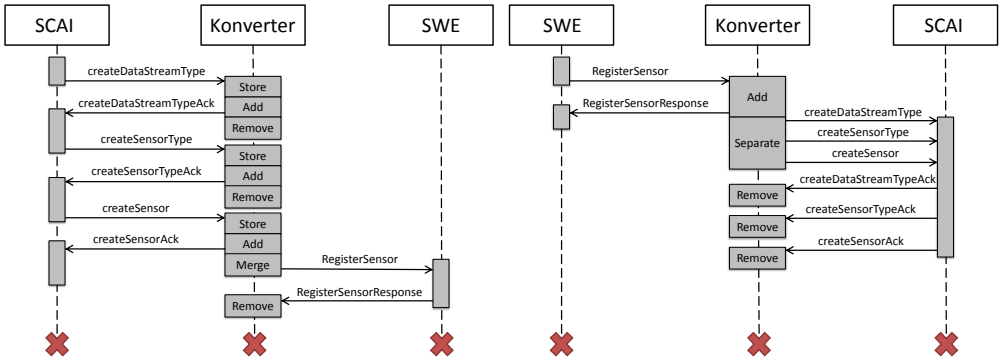


Abbildung 3.8: Prozess zur Konvertierung zwischen SCAI und SWE beim Erzeugen und Aktualisieren von Sensoren

sendet die gleichen Informationen in einer Nachricht. Jede der Nachrichten wird mit einer Bestätigung beantwortet. Allerdings können die Bestätigungsnachrichten nicht übersetzt werden, da sie nicht die nötigen Informationen enthalten.

Bei einer SCAI zu SWE-Konvertierung müssen somit die übermittelten Nachrichten gesammelt und zwischengespeichert werden, bis die Nachricht zum Erzeugen des Sensors übertragen wird. Die Bestätigungen für die einzelnen Nachrichten müssen dabei vom Konverter erzeugt werden. Wurden alle nötigen Nachrichten vom Konverter gespeichert, kann eine neue Nachricht erzeugt werden und an den SWE-Dienst gesendet werden. Die Antwort des SWE-Dienstes wird entfernt, da bereits eine entsprechende Bestätigung erzeugt wurde und die SWE-Nachricht nicht übersetzt werden kann. Bei einer SWE zu SCAI-Konvertierung muss zunächst die empfangende Nachricht mit einer Bestätigung beantwortet werden. Danach wird die Nachricht in die entsprechenden Nachrichten des SCAI-Protokolls aufgeteilt und an den SCAI-Dienst gesendet. Die Bestätigungsnachrichten des SCAI-Dienstes werden auch hier entfernt.

Ein ähnliches Verfahren muss angewendet werden, wenn andere Teile der Sensorbeschreibung, wie beispielsweise der Sensortyp, erzeugt werden. Außerdem muss das gleiche Verfahren zur Konvertierung angewendet werden, wenn eine Aktualisierung vorhandener Sensorbeschreibungen durchgeführt wird. Die Konvertierungen werden in Abb. 3.8 grafisch dargestellt.

Es treten ebenfalls Probleme auf wenn Information über einen Sensor angefragt werden. Bei beiden Protokollen wird dazu eine Anfrage-Nachricht an den entsprechenden Dienst gesendet. Die Antwort enthält die jeweilige Beschreibung des Sensors. Bei einer Übersetzung des SCAI-Protokolls in das SWE-Protokoll können die Nachrichten transformiert werden, ohne dass eine Anpassung des Kommunikationsverhaltens nötig ist. Wird allerdings das SWE-Protokoll in das

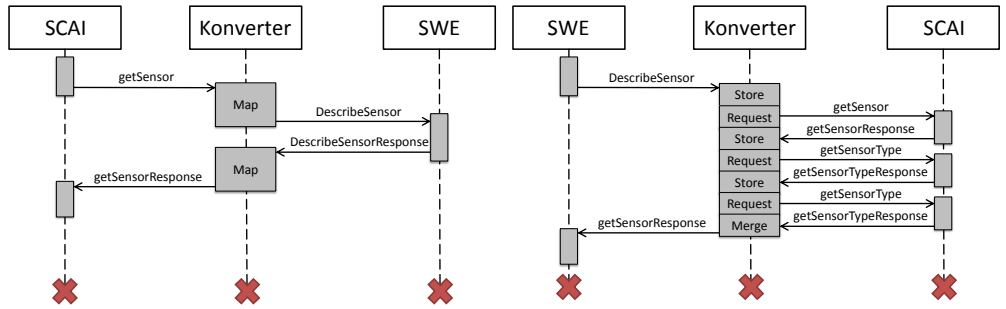


Abbildung 3.9: Prozess zur Konvertierung zwischen SCAI und SWE beim Anfragen von Sensoren

SCAI-Protokoll übersetzt, enthält die Antwort nicht alle nötigen Informationen. Das heißt, der Konverter muss die fehlenden Daten aktiv beim SCAI-Dienst anfragen. Hat er alle fehlenden Daten empfangen, können die gesammelten Nachrichten zu einer neuen Nachricht vereint werden.

Das gleiche Verfahren zur Konvertierung muss angewendet werden, wenn eine eine Auflistung aller Sensoren angefragt wird, allerdings für jeden Sensor der Liste. Dieses Verfahren muss auch bei der Anfrage anderer Elemente der Sensorbeschreibung, wie beispielsweise bei einem Sensortype, angewendet werden. Die Konvertierungen werden in Abb. 3.9 grafisch dargestellt.

3.2.2.4 Zusammenfassung

Eine Übersicht der Unterschiede zwischen den beiden Protokollen finden sich in Tab. 3.3. Außerdem enthält die Tabelle den Typ der Änderungen und die Anzahl der betroffenen Funktionen. Darüber hinaus wurde festgehalten, welche Änderungen nötig sind, um eine Konvertierung durchzuführen. Dabei wurde die Konvertierung in beide Richtungen betrachtet.

Die Tabelle zeigt, dass trotz der Unterschiede in Struktur und Verhalten der beiden Protokolle, einen Übersetzung für die Kernfunktionen möglich ist. Die Komplexität bei der Übersetzung der einzelnen Nachrichten ist dabei sehr unterschiedlich. Besonders hoher Aufwand wird erzeugt wenn Daten in unterschiedlich vielen Nachrichten übertragen werden, da hier unter Umständen Daten rekonstruiert werden müssen.

3.2.3 SOAP-Framework

SOAP ist ein Framework zur Übertragung von Nachrichten für Web-Dienste und wurde vom World Wide Web Consortium (W3C) spezifiziert. Nachrichten wer-

Beschreibung	Typ	Betroffen	Transf.	Neue Nachricht	Anfrage	Aufteilen	Löschen	Speichern	Kombinieren	Prüfung
Daten Anfrage	Struktur	2	ja	nein	nein	nein	nein	nein	nein	nein
Daten Übertragen	F. Untersch.	2	ja	ja	nein	nein	ja	nein	nein	nein
Beschr. Erzeugen	F. Untersch.	6	nein	ja	nein	ja	ja	ja	ja	nein
Beschr. Aktualisiern	F. Untersch.	6	nein	ja	nein	ja	ja	ja	ja	nein
Beschr. Anfragen	F. Untersch.	6	ja	nein	ja	nein	nein	ja	ja	nein
Beschr. Auflisten	F. Untersch.	6	ja	nein	ja	nein	nein	ja	ja	nein

Tabelle 3.3: Übersicht aller Änderungen zwischen den Protokollen

den mittels der Beschreibungssprache XML formatiert. In Abb. 3.10 findet sich eine Beispiel-Nachricht. Der Standard beschreibt mittels eines Prozess-Modells, wie Nachrichten übermittelt werden. Außerdem wird beschrieben, wie SOAP-Nachrichten mittels verschiedener Transport-Protokolle verwendet werden können. Darüber hinaus wird beschrieben, wie Daten, die nicht mittels XML beschrieben werden, in SOAP-Nachrichten übertragen werden können [WCL+08].

Die Version 1.0 des SOAP-Frameworks wurde von Microsoft und Userland entwickelt. Sie stellt die erste Basisversion des Protokolls dar. Allerdings gibt es neben einem Internet-Draft [BKL+99] keine offizielle, vollständige Dokumentation dieser Version. Die Version 1.1 [BEK+00] des Protokolls wurde an das W3C übergeben. Diese veröffentlichten es mit einer entsprechend umfangreichen Dokumentation. Die W3C entwickelte aus dieser Version SOAP 1.2 [GHM+07a, GHM+07b], das ebenfalls über eine entsprechend umfangreiche Dokumentation verfügt. Daher wurden im Rahmen dieser Analyse nur die Unterschiede zwischen den Versionen 1.1 und 1.2 betrachtet. Die Unterschiede zwischen diesen beiden Versionen wurden unter anderem bereits formal in [Had07] herausgearbeitet.

Wie bereits erwähnt, handelt es sich bei SOAP um ein Framework, das die Kommunikation mit Web-Diensten ermöglicht. Das Framework selbst ist unabhängig vom Inhalt der Nachrichten. Allerdings beschreibt es sowohl wie Nachrichten formatiert werden müssen, als auch was beim Verarbeiten von Nachrichten beachtet werden muss. Dabei wird auch beschrieben, welches Verhalten im Fehlerfall ausgeführt werden muss. Da das SOAP-Framework mittlerweile sehr verbreitet ist, existieren viele unterschiedliche Implementierungen, die zu Testzwecken verwendet werden können. Im Rahmen der Analyse wurde die Implementierung von Apache CXF verwendet [The12]. Das SOAP-Framework stellt somit ein Service-Protokoll dar, das die grundlegenden Funktionen zum Übertragen von Daten definiert.

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
    <soap:Header>
        <f:Test xmlns:f="http://schemas.cbusemann.de/"/>
    </soap:Header>
    <soap:Body>
        <soap:Fault>
            <soap:Code>
                <soap:Value>soap:Receiver</soap:Value>
                <soap:Subcode>
                    <soap:Value>f:Memory</soap:Value>
                </soap:Subcode>
            </soap:Code>
            <soap:Reason>
                <soap:Text>Out of memory</soap:Text>
            </soap:Reason>
            <soap:Role>ultimateReceiver</soap:Role>
            <soap:Detail>
                <soap:Error/>
            </soap:Detail>
        </soap:Fault>
    </soap:Body>
</soap:Envelope>
```

Abbildung 3.10: SOAP-Beispiel

3.2.3.1 Übersicht

Da es sich bei SOAP um ein Framework handelt, dass von Diensten eingesetzt wird, um Nachrichten zu übertragen, verfügt es nur über sehr wenige Nachrichtentypen. Diese Nachrichtentypen werden von den entsprechenden Web-Diensten verwendet, um ihre eigenen Protokolle zu definieren. Dazu können so genannte Message Exchange Pattern (MEP) eingesetzt werden. Bei dem MEP one way werden lediglich Daten übertragen. Im Fehlerfall wird eine entsprechende Nachricht zurück gesendet. Beim Request/Response-MEP wird eine Nachricht übermittelt und eine Antwort erwartet. Tritt ein Fehler auf, muss eine entsprechende Fehlernachricht übertragen werden. Beim Response-MEP wird die Anfrage mittels eines anderen Protokolls gesendet und nur die Antwort mittels des SOAP-Protokolls übertragen. Tritt ein Fehler auf, wird auch hier eine entsprechende Fehlernachricht gesendet. Außerdem können so genannte Long-Running Conversatinal MEPs aus einer Abfolge von Request/Response-MEPs erzeugt werden.

Das SOAP-Framework spezifiziert, dass jeder Dienst eine Rolle hat. Ein SOAP Sender sendet eine Nachricht, ein SOAP Intermediary leitet die Nachricht weiter und ein SOAP Receiver empfängt eine Nachricht. In der jeweiligen Nachricht können Rahmenbedingungen im Header spezifiziert werden, die ein teilnehmender Dienst mit seiner jeweiligen Rolle erfüllen muss. Dazu wird das so genannte *mustUnderstand*-Attribute gesetzt. Der entsprechende Dienst muss dann

in der Lage sein, das spezifizierte Element des Headers zu verstehen. Andernfalls produziert er eine Fehlernachricht.

3.2.3.2 Unterschiede zwischen den Versionen

Zwischen Version 1.1 und 1.2 existieren verschiedene strukturelle Unterschiede. Beispielsweise wurde das *root*-Element entfernt, verschiedene Elemente, wie *actor*, *Client* und *Server* wurden umbenannt, die Fehlerstruktur wurde hierarchisch angeordnet und ist auf ein Element beschränkt, der Array-Typ wurde geändert und die Darstellung des Referenz-Elementes wurde angepasst. All diese Änderungen können durch eine Transformation der Daten gehandhabt werden.

Hinzu kommen verschiedene Änderungen, die nicht übersetzt werden können. Zum Beispiel sind keine Folge-Elemente nach dem *Body-Element* mehr erlaubt, es wurden die Rollen *None* und *Ultimate Receiver* hinzugefügt, es wurde ein *Upgrade-Element* hinzugefügt, das *encodingStyle*-Attribute wurde angepasst und es wurde ein *Misunderstood*-Header hinzugefügt. In diesen Fällen muss ein entsprechendes Fehlerverhalten erzeugt werden. Dazu wird eine entsprechende Fehlernachricht mit einem *VersionMismatch* erzeugt. Dass heißt, in diesen Fällen müsste die jeweilige Nachricht entfernt werden und eine entsprechende Fehlernachricht erzeugt werden.

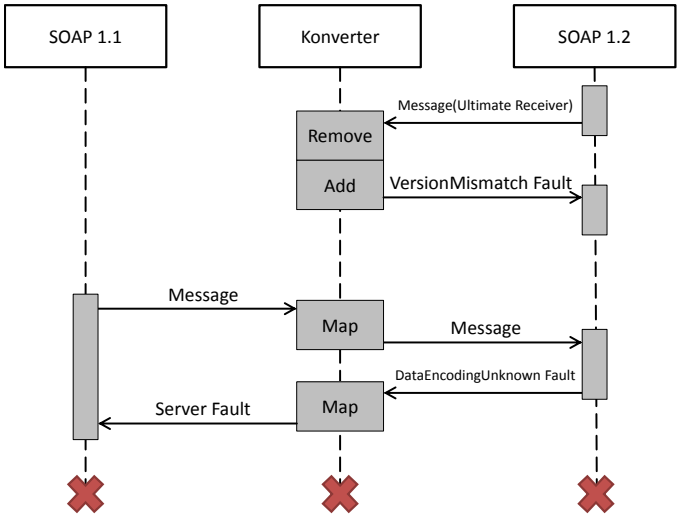


Abbildung 3.11: Prozess zur Konvertierung von SOAP-Fehlernachrichten

Außerdem wurden in Version 1.2 neue Fehlertypen hinzugefügt und die existierenden klarer spezifiziert. Somit lassen sich Fehler nicht zwangsläufig übersetzen. An dieser Stelle einen *VersionMismatch*-Fehler zu erzeugen, könnte allerdings wiederum zu einer Fehlinterpretation auf Sender-Seite führen. Daher

muss in diesem Fall die Fehlernachricht in eine *Server* Fehlermeldung übersetzt werden. Das Verhalten bei Fehlern wurde in Abb. 3.11 grafisch dargestellt.

Da das Kommunikationsverhalten auf Anfrage/Antwort-Mustern beruht und die Übertragung unabhängig von der eigentlichen Applikation ist, sind keine komplexeren Änderungen nötig. Dies könnte sich allerdings ändern, wenn das Protokoll, das das SOAP-Framework verwendet, Abhängigkeiten zwischen einzelnen Nachrichten hat. In diesem Fall könnten weitere Anpassungen des Kommunikationsverhaltens nötig sein.

3.2.3.3 Zusammenfassung

Eine Übersicht der Unterschiede zwischen den beiden Protokollversionen findet sich in Tab. 3.4. Außerdem enthält die Tabelle den Grund und Typ der Änderungen. Da es sich bei SOAP um ein Framework handelt, das selbst keine Funktionen zur Verfügung stellt, wurde die Anzahl der Funktionen nicht mit aufgeführt. Darüber hinaus wurde festgehalten, welche Änderungen nötig sind um eine Konvertierung durchzuführen. Dabei wurde sowohl die Konvertierung von der älteren Version des Protokolls in die Neuere, als auch die umgekehrte Konvertierung betrachtet.

Beschreibung	Typ	Transf.	Neue Nachricht	Anfrage	Aufteilen	Löschen	Speichern	Kombinieren	Prüfung
Folge-Element entfernt	Geändert	nein	ja	nein	nein	ja	nein	nein	nein
Actor umbenannt	Struktur	ja	nein	nein	nein	nein	nein	nein	nein
Neue Rollen	Geändert	nein	ja	nein	nein	ja	nein	nein	nein
Neuer Fault-Code	Geändert	nein	ja	nein	nein	ja	nein	nein	nein
Fault-Codes umbenannt	Struktur	ja	nein	nein	nein	nein	nein	nein	nein
Fault Struktur geändert	Struktur	ja	nein	nein	nein	nein	nein	nein	nein
Misunderstood Header	Geändert	nein	ja	nein	nein	ja	nein	nein	nein
Neues Upgrade-Element	Geändert	nein	ja	nein	nein	ja	nein	nein	nein
EncodingStyle geändert	Geändert	nein	ja	nein	nein	ja	nein	nein	nein
In-Place Referenzen erlaubt	Struktur	ja	nein	nein	nein	nein	nein	nein	nein
Referenz anders dargestellt	Struktur	ja	nein	nein	nein	nein	nein	nein	nein
Array-Typ geändert	Struktur	ja	nein	nein	nein	nein	nein	nein	nein
root-Attribut entfernt	Struktur	ja	nein	nein	nein	nein	nein	nein	nein
Neue Fehlertypen	Geändert	nein	ja	nein	nein	ja	nein	nein	nein

Tabelle 3.4: Übersicht aller Änderungen innerhalb des Protokolls

Die Tabelle zeigt, dass auch wenn viele Änderungen gemacht wurden, die Übersetzung zwischen den einzelnen Versionen relativ einfach gehandhabt werden kann. Dies hat den Grund, dass es sich bei SOAP um ein Framework handelt, das selbst keine konkreten Funktionen zur Verfügung stellt. Sowohl geänderte Strukturen sowie unterschiedliche Fehlerhandhabung ließen sich in ein akzeptables Verhalten übersetzen.

3.2.4 Flightradar24 zu SCAI-Konvertierung

In dieser Analyse wurden die Unterschiede zwischen dem Protokoll des Web-Dienstes flightradar.24 (im Folgenden Flightradar-Protokoll genannt) und dem SCAI-Protokoll in der Version 2.0 untersucht. Das Flightradar-Protokoll stellt Informationen über Flugzeuge und deren Positionen zur Verfügung. Da der flightradar.24 frei über das Internet zugänglich ist, existiert eine Implementierung, die zu Testzwecken verwendet werden kann. Allerdings existiert keinerlei Dokumentation über das Protokoll selbst. Da die Daten aber in einem übersichtlichen JSON-Format übertragen werden, das Protokoll nicht sonderlich komplex ist und der Inhalt, sowie die Funktionsweise des Dienstes über die entsprechende Website dokumentiert sind [Fli12], konnte aus diesen Informationen eine genügende Beschreibung des Protokolls abgleitet werden. Eine Beispiel-Nachricht des Protokolls findet sich in Abb. 3.12. Das hier beschreibende Analyseszenario wurde im Rahmen dieser Arbeit, in leicht abgewandelter Form auch für die Evaluation verwendet. Ausführliche Information über die Funktionsweise des Flightradar-Protokolls finden sich daher in Kapitel 6.3.

3.2.4.1 Übersicht

Das Flightradar-Protokoll ist, wie bereits erwähnt, relativ simpel. Es verfügt über vier verschiedene Nachrichtentypen. Die Nachricht *getPositionList* wird verwendet um eine Liste aller gesichteten Flugzeuge und ihrer Positionen anzufragen. Diese Nachricht bleibt immer gleich. Die Antwort auf diese Anfrage ist eine Nachricht, die die Liste aller gesichteten Flugzeuge inklusive ihrer GPS-Positionen enthält (*PositionList*). Dies sind üblicherweise zwischen 1500 und 1800 Flugzeuge. Darüber hinaus kann mittels der Nachricht *getPlaneData* Informationen über ein in der Nachricht spezifiziertes Flugzeug angefragt werden. Dazu muss die Nachricht die Identifikationsnummer des entsprechenden Flugzeuges enthalten. Die Antwort auf diese Anfrage ist eine Nachricht, die eine detaillierte Beschreibung des Flugzeuges enthält (*PlaneData*). Diese Nachricht enthält unter anderem auch den Flugzeugtyp.

Da das SCAI-Protokoll über wesentlich mehr Funktionen als das Flightradar-Protokoll verfügt, ist eine Konvertierung von SCAI zu Flightradar nicht sinnvoll. Allerdings existieren im SCAI-Protokoll alle Nachrichten des Flightradar-

```
{
        "icao":"BAW484",
        "iata":"BA484",
        "from":"London, Heathrow (LHR)",
        "to":"Barcelona, El Prat (BCN)",
        "via":"",
        "via2":"",
        "comment":"",
        "airline":"British Airways",
        "aircraft":"Airbus A319-131 (A319)",
        "reg":"G-EUPH",
        "img":"",
        "positions":[
                ["48.9391098022461","0.476965665817261","37025"],
                ["49.0282","0.4412","37025"],
                ["49.133056640625","0.398958653211594","37025"],
                ["49.2520332336426","0.350695163011551","37000"],
                ["49.3773651123047","0.299729555845261","37000"],
                ["49.530029296875","0.237149998545647","36350"],
                ["49.6167259216309","0.201391264796257","35500"],
                ["49.7966003417969","0.126900002360344","33425"],
                ["49.8751373291016","0.0943234115839005","32600"],
                ["49.9880676269531","0.0466196164488792","31050"],
                ["50.115","-0.0143","29700"],
                ["50.2328453063965","-0.100955449044704","28325"],
                ["50.3672790527344","-0.203680738806725","26600"],
        ]
}
```

Abbildung 3.12: Flightradar-Beispiel

Protokolls, so dass eine Konvertierung in diese Richtung möglich ist. Dabei werden Flugzeuge als Sensoren interpretiert und Flugzeugtypen als Sensortypen. Das heißt, die Nachrichten *getPositionList* und *getPlaneData* werden vom Konverter aus den entsprechenden SCAI-Nachrichten erzeugt, während die Nachricht *PositionList* und *PlaneData* vom Konverter in das SCAI Format übersetzt werden.

3.2.4.2 Unterschiede zwischen den Protokollen

Das Erzeugen der *getPositionList* Nachricht aus einer *getMeasurements* Nachricht kann durch eine reine Transformation der Daten geschehen und bedarf keiner Anpassung des Kommunikationsverhaltens. Das Gleiche gilt für die Nachricht *getPlaneData*. Bei der Übersetzung der Antwort gehen allerdings Informationen verloren, da die Nachricht *PlaneData* mehr Informationen enthält, als in der Sensorbeschreibung gespeichert werden kann. Da der entsprechende SCAMPI-Service die zusätzlichen Informationen nicht interpretieren kann, braucht das Kommunikationsverhalten aber nicht angepasst werden.

Die Übersetzung der Nachricht *PositionList* in ein SCAI-*Measurement* ist theoretisch ebenfalls möglich. Dabei wird jeder Eintrag der Liste in eine *Measurement-*

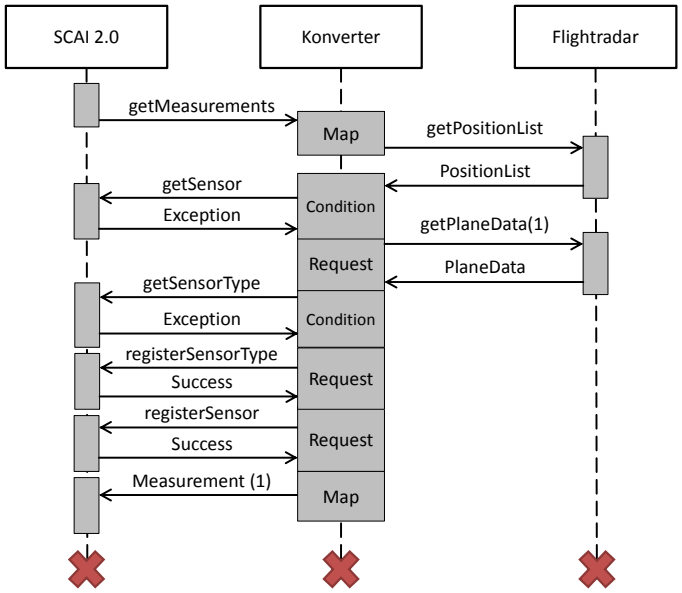


Abbildung 3.13: Prozess zur Konvertierung von Flightradar24 Daten in das SCAI Format

Nachricht des SCAI-Service übersetzt. Allerdings müssen für die erfolgreiche Übertragung der entsprechenden Sensoren, in diesem Fall die Flugzeuge, bereits registriert sein. Es muss also für jedes Flugzeug der Liste geprüft werden, ob der entsprechende Sensor beim SCAI-Dienst bereits registriert ist. Ist dies nicht der Fall sollte dieses mittels einer Anfrage erledigt werden. Die Sensor-Registrierung kann allerdings nur durchgeführt werden, wenn der entsprechende Sensortyp bereits registriert ist. Dazu muss dieser zunächst beim Flightradar-Dienst angefragt werden. Ist der Sensortyp nicht registriert, muss auch dieser zunächst mittels einer Anfrage beim SCAI-Dienst registriert werden. Wurden alle Anfragen erfolgreich gesendet, kann die Nachricht übersetzt und gesendet werden. Tritt allerdings ein Fehler auf, wie beispielsweise eine fehlgeschlagene Sensor-Registrierung, darf der entsprechende Eintrag der Flugzeugliste nicht übersetzt werden und die Nachricht muss entfernt werden. Abb. 3.13 zeigt den Prozess, der durchlaufen wird, wenn Sensor und Sensortyp noch nicht registriert sind.

3.2.4.3 Zusammenfassung

Eine Übersicht der Unterschiede zwischen den beiden Protokollen findet sich in Tab. 3.5. Außerdem enthält die Tabelle den Typ der Änderungen, sowie die Funktionen und die Anzahl der betroffenen Funktionen. Darüber hinaus wurde

festgehalten, welche Änderungen nötig sind, um eine Konvertierung durchzuführen.

Beschreibung	Typ	Betroffen	Transf.	Neue Nachricht	Anfrage	Aufteilen	Löschen	Speichern	Kombinieren	Prüfung
Liste Anfragen	Struktur	1	ja	nein	nein	nein	nein	nein	nein	nein
Flugzeug. Anfragen	Struktur	1	ja	nein	nein	nein	nein	nein	nein	nein
Flugzeug Info	Struktur	1	ja	nein	nein	nein	nein	nein	nein	nein
Flugzeugliste	Struktur	1	ja	nein	ja	nein	ja	ja	nein	ja

Tabelle 3.5: Übersicht aller Änderungen zwischen den Protokollen

Die Tabelle zeigt, dass die meisten Funktionen des Flightradar-Protokolls relativ einfach übersetzt werden können. Die einzige Ausnahme bildet die Übertragung der Sensordaten. Hier muss mit aufwändigen Mitteln geprüft werden, ob eine Übersetzung bereits sinnvoll ist und gegebenenfalls Beschreibungen beim SCAMPI-Dienst angelegt werden.

3.3 Szenarien

Die folgenden Konvertierungsszenarien sind das Ergebnis der Protokollanalyse, die im vorherigen Abschnitt beschrieben wurde. Sie stellen wiederkehrende Muster bei der Übersetzung von Service-Protokollen dar. Auf der Grundlage dieser Szenarien wird in Kapitel 4 das Differential Behavior Model definiert.

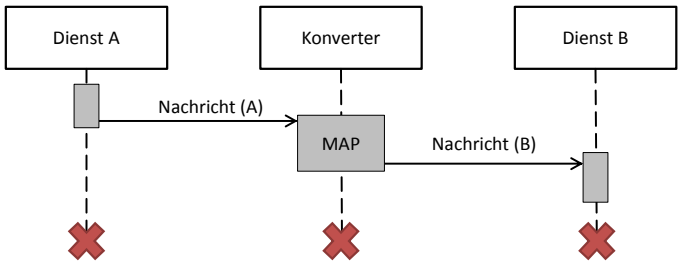


Abbildung 3.14: Konvertierungsszenario: Übersetzen von Nachrichten

Das erste Szenario besteht aus zwei Diensten, die unterschiedliche Protokolle verwenden. Da von Dienst A Nachrichten an Dienst B gesendet werden, muss eine Übersetzung des Nachrichten-Formats durchgeführt werden. Ein grafische Repräsentation des Szenarios findet sich in Abb. 3.14. Sollte von Dienst B Nach-

richten an Dienst A gesendet werden, muss die Konvertierung in die andere Richtung vorgenommen werden. Eine Konvertierung des Kommunikationsverhaltens ist in diesem Szenario nicht nötig.

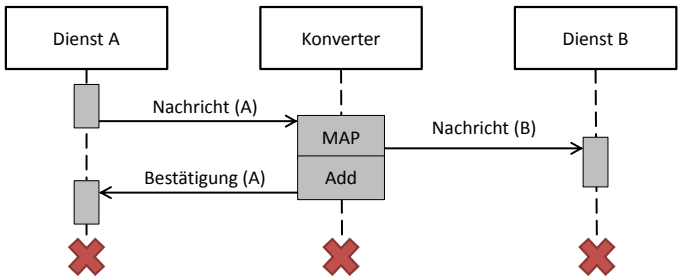


Abbildung 3.15: Konvertierungsszenario: Hinzufügen von Nachrichten

Im zweiten Szenario muss ebenfalls eine Nachricht des Dienstes A übersetzt werden, allerdings erwartet Dienst A eine Bestätigung auf jede Nachricht die er sendet. Diese Bestätigung wird nicht von Dienst B gesendet. Es muss also eine zusätzliche Nachricht generiert werden. Die fehlende Nachricht könnte natürlich auch an einen anderen Dienst gesendet werden müssen. Entscheidend ist hier, dass im Kommunikationsprozess Nachrichten fehlen können, die zusätzlich vom Konverter erzeugt werden müssen. Ein grafische Repräsentation des Szenarios findet sich in Abb. 3.15.

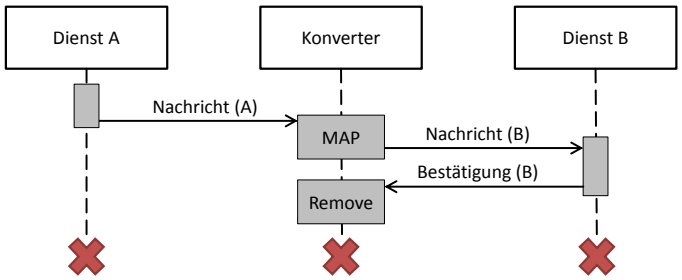


Abbildung 3.16: Konvertierungsszenario: Entfernen von Nachrichten

In Szenario 3 wird von Dienst B für jede empfangene Nachricht eine Bestätigung gesendet. Diese Bestätigungsnachrichten führen bei Dienst A allerdings zu fehlerhaftem Verhalten. Daher müssen diese vom Kommunikationsprozess entfernt werden. Auch in diesem Szenario kann die Kommunikation, und somit die Konvertierung, in die andere Richtung stattfinden. Der entscheidende Fakt in diesem Szenario ist, dass Nachrichten existieren können, die vom Kom-

munikationsprozess entfernt werden müssen. Ein grafische Repräsentation des Szenarios findet sich in Abb. 3.16.

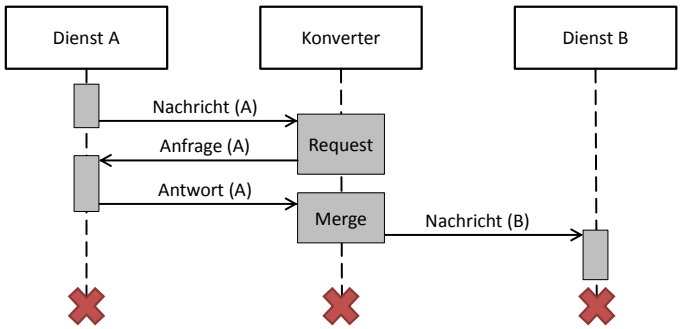


Abbildung 3.17: Konvertierungsszenario: Anfragen von Nachrichten

In Szenario 4 müssen die Nachrichten von Dienst A in die von Dienst B übersetzt werden. Allerdings sind nicht alle Daten vorhanden, um die Konvertierung durchzuführen. Daher muss eine Anfrage an Dienst A gesendet werden, um die fehlenden Daten zu erhalten. Wurden alle Nachrichten eingesammelt, werden diese zu einer neuen Nachricht kombiniert. Bei diesem Szenario kann natürlich auch die Kommunikation mit dem Zieldienst oder einem dritten Dienst nötig sein. Auch kann es nötig sein, dass mehrere Anfragen gesendet werden müssen, um die fehlenden Daten zu erhalten. Entscheidend ist hier, dass der Konverter in der Lage sein muss, fehlende Daten aktiv anfragen zu können. Ein grafische Repräsentation des Szenarios findet sich in Abb. 3.17.

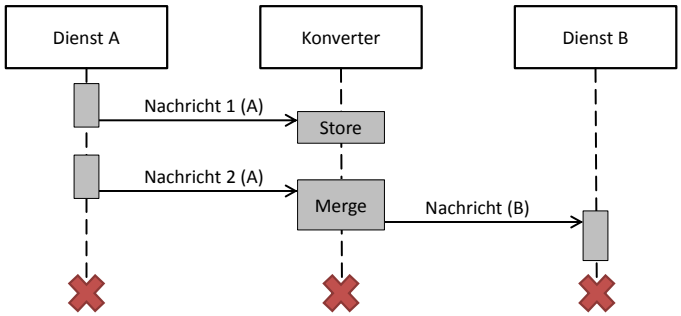


Abbildung 3.18: Konvertierungsszenario: Speichern von Nachrichten

Szenario 5 ist eine abgewandelte Variante des letzten Szenarios. In diesem Szenario werden die fehlenden Daten nicht aktiv angefragt. Statt dessen wird gewartet, bis alle nötigen Nachrichten im Rahmen anderer Kommunikationsprozesse übertragen werden. Diese Nachrichten werden, unabhängig von ihrem

eigenen Prozess, zwischengespeichert. Wurden alle Nachrichten zwischengespeichert, werden diese zu einer neuen Nachricht kombiniert. Dies ist unabhängig von der Anzahl der Nachrichten, die zwischengespeichert werden müssen. Darüber hinaus sollten gespeicherte Nachrichten nach einer bestimmten Zeit verfallen, um die Verwendung alter Daten zu vermeiden. Außerdem müssen die gespeicherten Nachrichten nicht zwangsläufig zum Erzeugen einer neuen Nachricht eingesetzt werden, sondern könnten zum Beispiel auch zum Erzeugen von Anfragen, wie in Szenario 4 beschrieben, verwendet werden. Entscheidend ist hier, dass der Konverter in der Lage sein muss, Nachrichten zwischenzuspeichern um sie später für andere Konvertierungen zu verwenden. Ein grafische Repräsentation des Szenarios findet sich in Abb. 3.18.

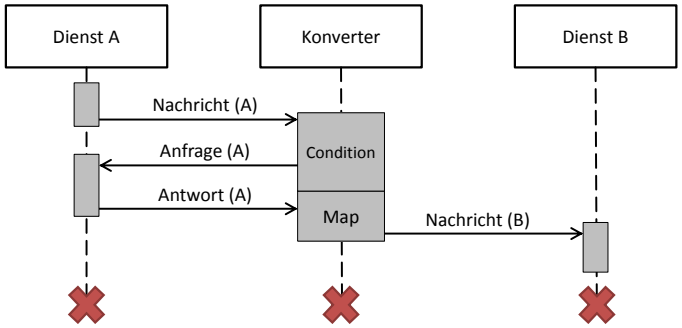


Abbildung 3.19: Konvertierungsszenario: Prüfen von Bedingungen

In Szenario 6 kann die Übersetzung der Nachricht nur durchgeführt werden, wenn einer der Dienste einen bestimmten Zustand hat. Um diesen Zustand festzustellen, muss eine Nachricht an den jeweiligen Dienst gesendet werden. Die Antwort auf diese Nachricht wird dann analysiert. Basierend auf dem Ergebnis der Analyse wird dann eine spezifische Konvertierung durchgeführt. Bei diesem Szenario sind auch andere Bedingungen denkbar, an die die Konvertierung der Nachricht geknüpft wird. Beispiele hierfür sind der Zustand eines anderen Dienstes, aber auch ob vor dem Erhalt der Nachricht eine andere Nachricht übertragen wurde. Kern des Szenarios ist es also, die Konvertierung einer Nachricht abhängig von klar definierten Bedingungen zu machen. Ein grafische Repräsentation des Szenarios findet sich in Abb. 3.19.

Die vorgestellten Szenarien stellen die grundlegenden Konvertierungsszenarien dar, die im Rahmen der Analyse festgestellt wurden. Die Szenarien selbst können in beliebigen Variationen und Kombinationen auftreten. Das heißt, das Empfangen einer Nachricht kann im Konverter verschiedene der beschriebenen Konvertierungsszenarien nach sich ziehen.

3.4 Zusammenfassung

Im Rahmen dieser Analyse wurden die Unterschiede zwischen verschiedenen Typen von Protokollen betrachtet. Während SCAI sich sehr stark auf die Beschreibung von Sensoren und deren Daten konzentriert, liegt der Fokus von SWE eher auf den gemessen Sensordaten, das heißt, diese werden unabhängig vom eigentlichen Sensor beschrieben. SOAP dagegen ist ein Framework, das von Web-Diensten verwendet wird um Daten zu übertragen. Dabei ist SOAP, im Gegensatz zu den vorherigen Protokollen vollkommen unabhängig von dem Inhalt seiner Nachrichten. Das Flightradar-Protokoll dagegen ist ein ein simples Web-Protokoll, das für einen spezifischen Anwendungszweck entwickelt wurde.

Die Analyse hat dabei nicht nur die Unterschiede zwischen einzelnen Versionen dieser Protokolle betrachtet, sondern auch Unterschiede zwischen den Protokollen selbst. Dabei fällt auf, dass zwischen Protokollversionen relativ viele strukturelle Änderungen auftreten, die sich entweder relativ simpel übersetzen lassen oder aber keine Übersetzung erlauben, da sie eine Funktion zur Verfügung stellen, die in einer älteren Version nicht existiert. Bei den Unterschieden zwischen verschiedenen Protokollen mussten dagegen häufiger fehlende Daten rekonstruiert werden und komplexe Bedingungen geprüft werden, um eine Übersetzung zu ermöglichen. Die Ergebnisse dieser Studie wurden verwendet, um das Modell aus Kapitel 4 zu definieren.

4 Differential Behavior Model

Dieses Kapitel beschreibt den zentralen Lösungsansatz der Arbeit. Neben dem grundlegenden Konzept wird beschrieben, wie die Transformation der Daten und des Kommunikationsverhaltens gehandhabt wird. Darüber hinaus wird das System zur Protokoll- und Nachrichtenidentitifikation erläutert.

4.1 Grundlegendes Konzept

Klassische Modellierungsverfahren konzentrieren sich üblicherweise auf die Modellierung der Protokolle selbst und generieren daraus automatisiert eine Komponente, die die Unterschiede zwischen beiden Protokollen beschreibt. Allerdings erfordern diese Verfahren häufig eine aufwendige Einarbeitung, da die Protokolle und ihre Unterschiede mittels eines abstrakten Modells beschrieben werden. Aufbauend auf den Ergebnissen der Analyse aus Kapitel 3 wurde daher ein Modell entwickelt, dass sich dem Prozess einer Implementierung annähert und somit den Aufwand zum Übersetzen zwischen Service-Protokoll verringert. Das Differential Behavior Model (DBM) beschränkt sich dabei [BN11b, BN11a] auf die Beschreibung der Protokollunterschiede. Das heißt, während üblicherweise eine vollständige Beschreibung des Quell- und Ziel-Protokolls angefertigt werden muss, muss beim DBM lediglich beschrieben werden was die Unterschiede zwischen den beiden Protokollen sind. Dies geschieht in einer Form, die sich dem Prozess einer Implementierung annähert, so dass das Konzept für Entwickler einfach zu verstehen ist. Abb. 4.1 stellt das Konzept des DBM grafisch dar.

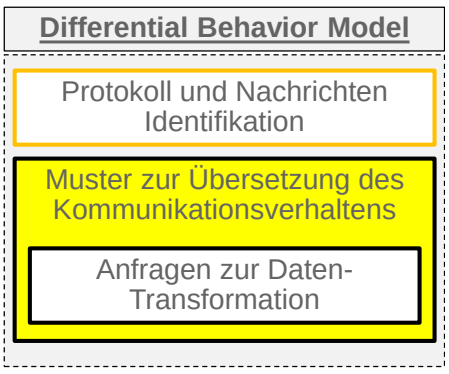


Abbildung 4.1: Schematische Darstellung des DBM-Konzepts

Während die Protokoll- und Nachrichten-Identifikation mittels eines leichtgewichtigen Sub-Modells realisiert wird, wird das Kommunikationsverhalten mittels eines Satz von Mustern übersetzt. Diese Muster wurden mit dem Hintergrund entwickelt, einen leicht verständlichen Weg für Entwickler zu bieten, um Änderungen im Kommunikationsverhalten zu beschreiben. Die Datentransformation selbst wird mittels Anfrage realisiert, die bereits Teil der jeweiligen Muster sind. Die Muster lassen sich zu komplexen Sequenzen kombinieren und mittels Bedingungen in ihrer Ausführung einschränken. Die folgenden Abschnitte beschreiben die einzelnen Elemente des Konzepts im Detail.

4.2 Datentransformation

Dass DBM wurde so entwickelt, dass beliebige Technologien zur Datentransformation eingebunden werden können. Da eine der Rahmenbedingungen des Modells ist, dass es von Entwicklern einfach zu verstehen beziehungsweise zu erlernen ist, wurde der Fokus auf Anfrage-Technologien gelegt. Diese haben den Vorteil, dass sie bereits mit dem Ziel entwickelt wurden, eine einfach zu erlernende Technologie zur Datentransformation zu realisieren.

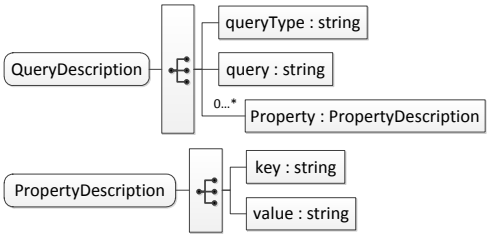


Abbildung 4.2: Schematische Darstellung der Anfrage-Darstellung

Das DBM verfügt über eine generische Beschreibung, um Datentransformationen mittels Anfrage-Sprachen zu beschreiben. Eine grafische Repräsentation davon findet sich in Abb. 4.2. Dazu definiert das Modell, dass ein *queryType* angegeben werden muss. Dieser identifiziert die jeweilige Anfrage-Technologie, die zum Transformierten der Daten eingesetzt werden soll. Die Anfrage selbst wird mittels des Elements *query* spezifiziert. Zusätzlich erlaubt das Modell die Definition beliebig vieler *Property*-Elemente. Diese enthalten *key*- und *value*-Elemente und können verwendet werden, um die entsprechende Anfrage zu konfigurieren.

Das Modell erlaubt somit die Verwendung von Anfrage-Technologien, wie beispielsweise XQuery, XSLT oder reguläre Ausdrücke (siehe Abschnitt 2.2). Prinzipiell erlaubt dieses Konzept aber auch die Verwendung von anderen Technologien. Dazu muss in der später vorgestellten Implementierung (siehe Kapitel

5) lediglich der entsprechende Code integriert werden. Im DBM kann dieser dann über die Elemente *queryType* und *query* identifiziert werden.

4.3 Protokoll- und Nachrichtenidentifikation

Um die Nachrichten eines Service-Protokolls zu konvertieren, muss zunächst das Protokoll und die Nachricht innerhalb des Protokolls identifiziert werden. Dazu definiert das DBM ein Sub-Modell, das so genannte Protocol Identification Model (PIM). Dieses Modell identifiziert Protokolle und Nachrichten durch die Beschreibung wiederkehrender Elemente innerhalb der Nachrichten. Zur Identifikation dieser Elemente werden, genau wie zur Datentransformation, Anfrage-Technologien eingesetzt. Dies können Technologien wie XPath oder reguläre Ausdrücke sein. Dabei gilt zu beachten, dass im Gegensatz zur Datentransformation in diesem Fall keine neue Nachricht erzeugt werden soll sondern geprüft wird, ob ein Element in einem Dokument vorhanden ist. Da diese ebenfalls mit der generischen Anfrage-Beschreibung aus Abb. 4.2 dargestellt werden kann, wird diese hier wiederverwendet. Die Entscheidung ob eine neue Nachricht erzeugt wird oder eine Prüfung stattfindet, ist somit abhängig von dem verwendeten *queryType*. Eine grafische Repräsentation des PIM findet sich in Abb. 4.3.

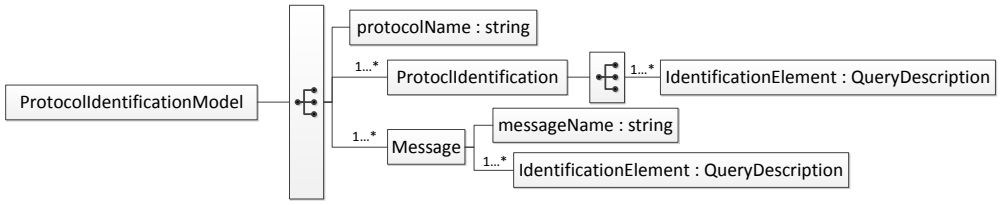


Abbildung 4.3: Schematische Darstellung des PIM

Das PIM spezifiziert, dass jedes Protokoll eindeutig mittels eine Namens identifiziert wird. Dazu wird der *protocolName* verwendet. Die Identifikation des Protokolls wird mittels des *ProtocolIdentification* Elementes realisiert. Innerhalb eines *ProtocolIdentification*-Elements können beliebig viele *IdentificationElement*-Elemente hinzugefügt werden. Diese beruhen auf dem Type *QueryDescription*, der bereits in Abschnitt 4.2 beschrieben wurde. Werden alle Elemente eines *ProtocolIdentification*-Elementes gefunden, gilt das Protokoll als identifiziert. Das Modell erlaubt die Spezifikation mehrerer *ProtocolIdentification*-Elemente. Um ein Protokoll zu identifizieren müssen dabei nur bei einem *ProtocolIdentification*-Element alle *IdentificationElement*-Elemente gefunden werden.

Die Nachrichten des jeweiligen Protokolls werden auf ähnliche Weise identifiziert. Dazu wird für jede Nachricht ein *Message*-Element spezifiziert. Jeder

Nachricht wird mittels des *messageName*-Elements ein innerhalb des Protokolls eindeutiger Name zugeordnet. Nachrichten werden, genau wie das Protokoll, mit einer beliebigen Anzahl von Anfragen identifiziert. Die Anfragen werden entsprechend über *IdentificationElement*-Elemente spezifiziert.

4.4 Transformation des Kommunikationsverhaltens

Das DBM erlaubt die Beschreibung von Unterschieden des Kommunikationsverhaltens zwischen zwei Protokolle. Eine grafische Darstellung des DBM finde sich in Abb. 4.4.

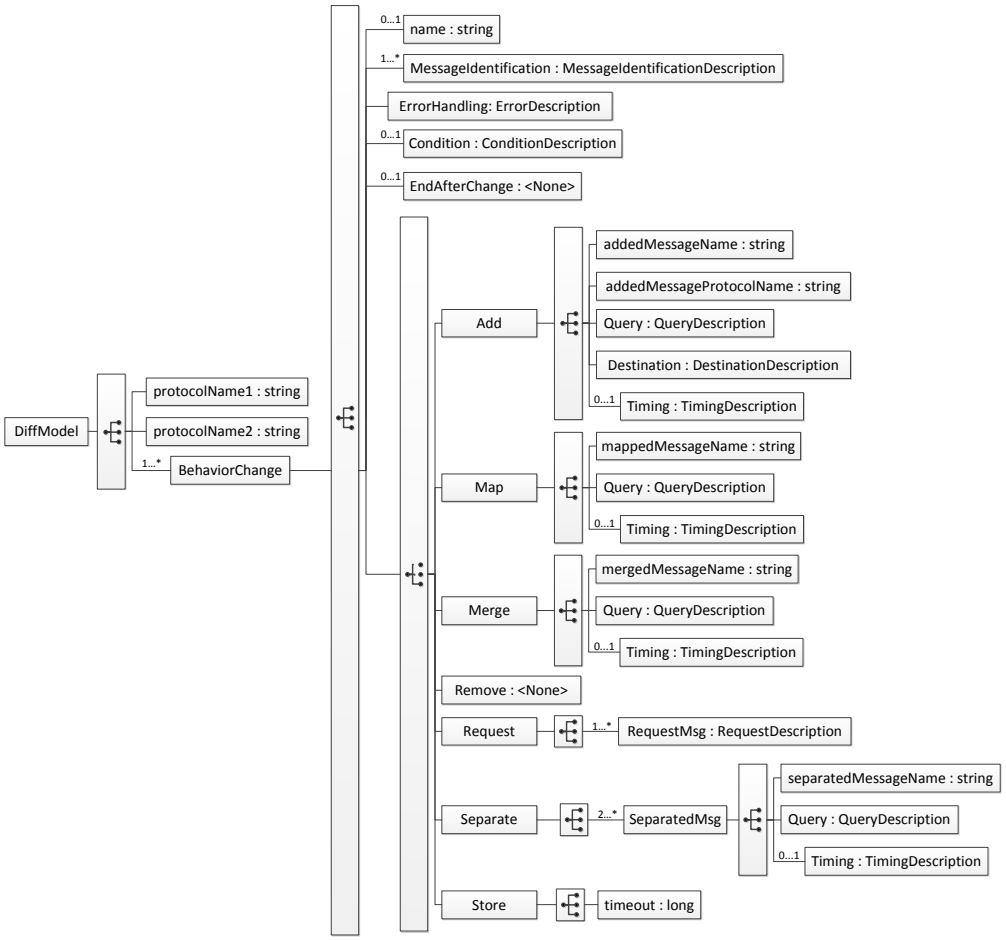


Abbildung 4.4: Schematische Darstellung des DBM

Zunächst werden die beteiligten Protokolle mittels Angabe von *protocolName1* und *protocolName2* identifiziert. Diese Namen beziehen sich auf die Namen

der Protokolle, die mittels des PIM spezifiziert wurden (siehe Abschnitt 4.3). Eine Änderung des Kommunikationsverhaltens wird jeweils mittels *BehaviorChange*-Elements beschrieben. Jedes *BehaviorChange*-Element beschreibt genau eine Verhaltensänderung, die aber auf unterschiedliche Nachrichten angewendet werden kann. Die Nachrichten, auf die die Verhaltensänderung angewendet werden soll, werden jeweils mittels eines *MessageIdentification*-Elements spezifiziert. Eine grafische Repräsentation des Typs dieses Elements findet sich in Abb. 4.5.

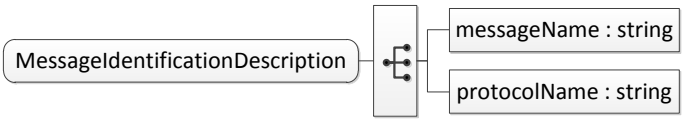


Abbildung 4.5: Schematische Darstellung des MessageIdentificationDescription-Typs

Mittels der Elemente *messageName* und *protocolName* werden der Name der Nachricht und des Protokolls spezifiziert. Diese Namen entsprechen den Namen, die im PIM angegeben wurden. Nach den *MessageIdentifcation* Elementen folgen die teilweise optionalen Elemente *ErrorHandling*, *Condition* und *EndAfterChange*. Diese spezifizieren Rahmenbedingungen der Verhaltensänderung und werden in den folgenden Abschnitten näher erläutert. Bei der Beschreibung der Unterschiede im Kommunikationsverhalten wird auf die vollständige Beschreibung des Quell- und Ziel-Protokolls verzichtet. Statt dessen stellt das DBM einen Satz von Mustern zur Verfügung, die Änderungen des Kommunikationsverhaltens beschreiben. Diese Muster beruhen auf der Analyse aus Abschnitt 3. Im Folgenden werden die einzelnen Muster im Detail erklärt:

Add: Aus der eingehenden Nachricht wird eine weitere Nachricht erzeugt. Die eingehende Nachricht wird ohne Änderungen weitergeleitet. Mittels der Elemente *addedMessageName* und *addedMessageProtocolName* wird der Name der Nachricht und des Protokolls der neuen Nachricht angeben. Durch das *Query*-Element wird die Anfrage spezifiziert, mittels derer die neue Nachricht erzeugt werden soll. Das *Destination*-Element legt das Ziel der neuen Nachricht fest. Diese wird über den *DestinationDescription*-Typ aus Abb. 4.6 spezifiziert. Der Entwickler kann als Ziel entweder den Ziel- oder Quelldienst mittels des *Source* oder *Destination*-Elementes wählen. Darüber hinaus kann ein weiterer Dienst mittels des *Other*-Elementes gewählt werden. In diesem Fall kann der Service mittels des *address*-Elements angegeben werden. Mittels des Elements *methodName* wird die Kommunikationstechnologie identifiziert. Eine Konfiguration kann über die *Property*-Elemente stattfinden.

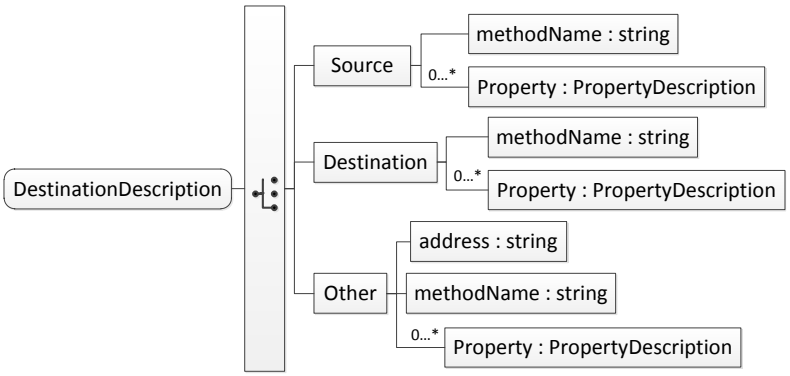


Abbildung 4.6: Schematische Darstellung des DestinationDescription-Typs

Das *Timing*-Element kann verwendet werden, um den Übertragungszeitpunkt der neuen Nachricht zu definieren. Eine ausführliche Beschreibung dieses Elements kann in Abschnitt 4.5 gefunden. Daher wird in den folgenden Beschreibungen nicht näher auf dieses Element eingegangen werden.

Map: Dieses Muster transformiert die Daten-Repräsentation der eingehenden Nachricht. Die transformierte Nachricht wird an den Zieldienst weitergeleitet. Der Name der transformierten Nachricht muss mittels des *mappedMessageName* angeben werden. Auf die Angabe des neuen Protokollnamens kann verzichtet werden, da sich dieser aus dem Kontext ergibt. Die Datentransformation wird mittels des *Query*-Elements spezifiziert.

Merge: Mittels dieses Musters werden mehrere Nachrichten zu einer neuen Nachricht zusammengefasst. Es muss auf alle Nachrichten angewendet werden, die vereinigt werden sollen. Das Element *mergedMessageName* gibt den Namen der vereinten Nachricht an. Das *Query*-Element definiert die Anfrage zur Vereinigung. Das Speichern der benötigten Nachrichten muss zuvor entweder mittels des *Store*- oder des *Request*-Musters realisiert werden. Diese werden in diesem Abschnitt noch näher erklärt.

Remove: Das Muster bewirkt, dass die eingehende Nachricht entfernt wird. Da die Nachricht nicht weiter bearbeitet wird, sind keine zusätzlichen Informationen nötig.

Request: Dieses Muster wird verwendet, um zusätzliche Nachrichten bei einem beliebigem Dienst anzufragen. Genau wie beim Add-Muster wird die eingehende Nachricht nicht entfernt und ohne Änderungen weitergeleitet. Innerhalb eines Musters können beliebig viele Anfrage-Nachrichten mittels jeweils eines *RequestMsg*-Elements definiert werden. Die Antwort auf eine Anfrage-Nachricht wird im Konverter gespeichert und kann bei der Generierung einer

weiteren Anfrage-Nachricht verwendet werden. Ein grafische Repräsentation des *RequestDescription*-Typs findet sich in Abb. 4.7.

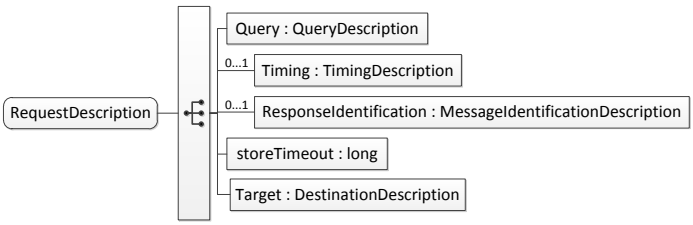


Abbildung 4.7: Schematische Darstellung des Request-Typs

Das *Query*-Element definiert die Anfrage, mittels der eine Anfrage-Nachricht aus den eingehenden Nachrichten generiert wird. Wird ein *ResponseIdentificaion*-Element hinzugefügt, muss die Antwort auf die Anfrage-Nachricht den über dieses Element definierten Protokoll- und Nachrichtennamen haben. Mittels des *storeTimeout*-Elements wird angegeben, wie lange die Antwort im Konverter gespeichert werden soll. Das *Target*-Element gibt, analog zum Add-Muster, das Ziel der Anfrage-Nachricht an.

Theoretisch könnte das Request-Muster auch durch eine Kombination der Muster *Add*, *Store* und *Merge* realisiert werden. Das Muster bietet allerdings eine komfortablere Möglichkeit Nachrichten einzusammeln, die nur kurzzeitig benötigt werden. Darüber hinaus müssen die generierten Anfragen nicht zwangsläufigerweise mittels eines PIM spezifiziert worden sein.

Separate: Das Muster spezifiziert, dass die eingehende Nachricht in zwei oder mehr Nachrichten aufgeteilt wird. Jede neue Nachricht muss mittels eines *SeparatedMsg*-Elements spezifiziert werden. Für jedes dieser Elemente muss mittels des Elementes *separatedMessageName* der Name der neuen Nachricht angegeben werden. Das *Query*-Element spezifiziert die Anfrage mit der die neue Nachricht aus der Alten erzeugt wird.

Store: Eine Kopie der Nachricht wird im Konverter gespeichert. Die eingehende Nachricht wird ohne Änderungen weitergeleitet. Das *timeout*-Element definiert, nach welcher Zeit die Nachricht im Konverter verworfen wird.

Diese Muster bilden die Grundlage für die Beschreibung von Unterschieden im Kommunikationsverhalten. Da mehrere Muster auf die gleiche Nachricht angewendet werden können, können diese zu komplexen Verhaltensänderungen im Protokoll führen. Werden mehrere Verhaltensänderungen für eine Nachricht definiert, werden diese sequentiell angewendet, sobald die entsprechende Nachricht empfangen wird. Dadurch ist es beispielsweise möglich, eine eingehende Nachricht zunächst zu speichern, eine Bestätigungsnachricht an den Quelldienst

zu senden, weitere Nachrichten beim Quelldienst anzufragen und aus allen gesammelten Nachrichten eine neue Nachricht zu produzieren.

4.5 Übertragungszeitpunkt

Bei der Kommunikation mit Sensoren und Diensten kann es vorkommen, dass diese nur unter bestimmten Voraussetzungen Nachrichten entgegen nehmen. Gerade im Bereich der Wireless Sensor Networks (WSN) sind Sensoren häufig nur zu bestimmten Zeitpunkten aktiv, um die Batterien nicht unnötig zu belasten. Da bei dem vorgestellten Ansatz nicht nur Nachrichten an solche Dienste gesendet werden, sondern auch fehlende Daten aktiv abgefragt werden, verfügen die Muster Add, Map, Merge, Request und Separate über das Element *Timing*.

Dadurch kann im Modell festgelegt werden, zu welchem Zeitpunkt mit dem entsprechenden Dienst kommuniziert werden kann. Wie Abb. 4.8 zeigt erlaubt das DBM die Wahl zwischen der Festlegung eines *Delay*, der Definition von einem oder mehrerer fester Zeitfenster mittels des *TimeOfDay*-Elements oder der Definition mittels einer generischen Beschreibung. Wird ein *Delay* verwendet wird mittels des *min*- und *max*-Elementes der Sendezeitpunkt nach Erhalt der Nachricht definiert. Beim *TimeOfDay* wird der Sendezeitpunkt mittels des *time*-Elements definiert, während die Größe des Zeitfensters mittels des *windowSize*-Elements festgelegt wird. Der generische Ansatz verwendet die gleiche Beschreibung aus *type* und *Property* wie bei der Datentransformation in Abschnitt 4.2.

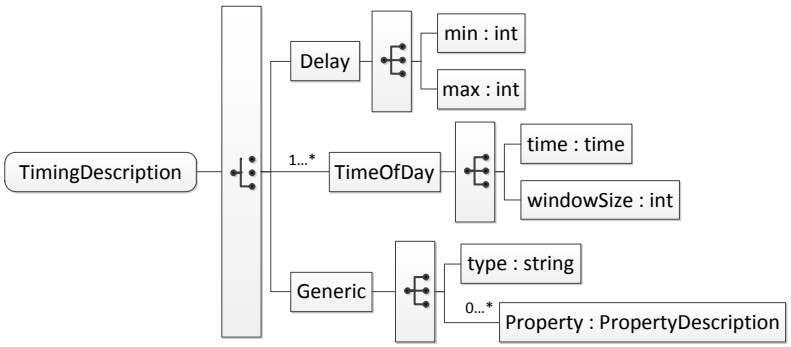


Abbildung 4.8: Schematische Darstellung des Timing-Elements

4.6 Fehlerbehandlung

Da die Datentransformation auf Anfragen beruht, die auf variable Nachrichten angewendet werden, kann es theoretisch vorkommen, dass eine Transformation zu fehlerhaften Nachrichten führt. Daher muss für jede Verhaltensänderung definiert werden, was im Fehlerfall passieren soll. Abb. 4.9 zeigt ein grafische Repräsentation des *ErrorDescription*-Typs.

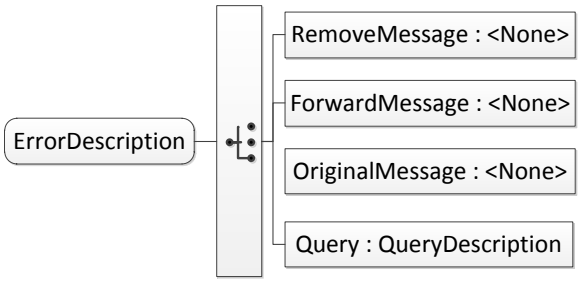


Abbildung 4.9: Schematische Darstellung des ErrorDescription-Typs

Wird das *RemoveMessage*-Element ausgewählt, wird die Originalnachricht und jede eventuell produzierte Nachricht entfernt und nichts weitergeleitet. Wird das *ForwardMessage*-Element ausgewählt, wird die fehlerhaft produzierte Nachricht weitergeleitet. Wird das *OriginalMessage*-Element ausgewählt wird die fehlerhaft produzierte Nachricht entfernt und die original Nachricht weitergeleitet. Wird das *Query*-Element ausgewählt, wird die entsprechende Anfrage verwendet um eine neue Nachricht, wie beispielsweise eine Fehlermeldung, zu produzieren. Darüber hinaus können mittels der Bedingung (siehe Abschnitt 4.7) *PreviousError* beliebige Muster nach dem Auftreten eines Fehlers ausgeführt werden.

4.7 Bedingungen

Die im Abschnitt 4.4 beschriebenen Muster erlauben die Definition von Änderungen des Kommunikationsverhaltens. Das *Condition*-Element erlaubt es dem Entwickler, die Ausführung dieser Muster an Bedingungen zu knüpfen. Eine grafische Darstellung des *ConditionDescription*-Typs findet sich in Abb. 4.10.

Jede Bedingung kann ein Name über das *name*-Element zugewiesen bekommen. Mittels des *ConditionType*-Elements wird festgelegt, ob erwartet wird, dass die Bedingung wahr oder falsch ist. Wird das *If*-Element ausgewählt, muss die Bedingung erfüllt werden, während beim *IfNot*-Element die Bedingung nicht erfüllt sein darf. Das *Condition*-Element erlaubt die Auswahl zwischen fünf verschiedenen Typen von Bedingungen. Diese werden mittels des *ConditionPredicate*-Elements ausgewählt und sind wie folgt definiert:

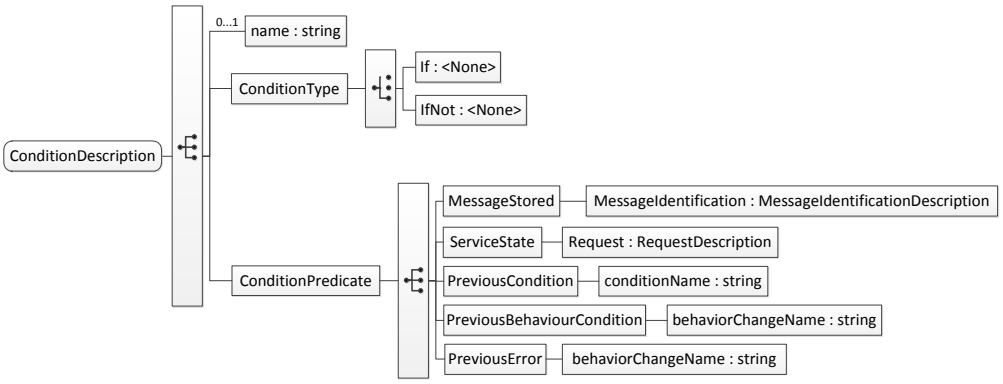


Abbildung 4.10: Schematische Darstellung des ConditionDescription-Typs

MessageStored: Mit dieser Bedingung kann überprüft werden, ob eine bestimmte Nachricht zuvor mittels des Store- oder Request-Muster gespeichert wurde. Dazu muss die Nachricht mittels des *MessageIdentification*-Elements identifiziert werden.

ServiceState: Die Bedingung wird verwendet, um den Zustand eines Dienstes abzufragen. Dazu wird eine Nachricht an den jeweiligen Dienst gesendet und die Antwort analysiert. Die Anfrage wird mittels der gleichen Beschreibung realisiert, die bereits für das *Request*-Muster verwendet wurde. Dabei muss das *ResponseIdentification*-Element zwingend angegeben werden, damit überprüft werden kann, ob die empfangene Nachricht der erwarteten Nachricht entspricht.

PreviousCondition: Mittels dieser Bedingung wird geprüft, ob eine vorherige Bedingung erfüllt wurde. Dazu muss mittels des *conditionName*-Elements der Name der vorherigen Bedingung angegeben werden. Dieses Element referenziert auf das *name*-Element einer vorherigen Bedingung.

PreviousBehaviorChange: Die Bedingung überprüft, ob eine bestimmte Verhaltensänderung durchgeführt wurde. Dies wird zusätzlich zur *PreviousCondition*-Bedingung benötigt, da nicht jede Verhaltensänderung notwendigerweise eine Bedingung haben muss. Das *behaviorChangeName*-Element der Bedingung referenziert dabei das *name*-Element eines anderen *BehaviorChange*-Elementes.

PreviousError: Die Bedingung überprüft, ob beim Ausführen einer vorherigen Verhaltensänderung ein Fehler aufgetreten ist. Dazu muss die Verhaltensänderung mittels des *behaviorChangeName*-Elements spezifiziert werden. Dieses referenziert das *name*-Element eines anderen *BehaviorChange*-Elementes.

Diese Bedingungen werden verwendet, um einen Entscheidungsbaum abzubilden. Während die Zustände die Knoten darstellen, repräsentieren die Bedingungen die Kanten des Baums. Zyklen können somit nicht beschrieben werden.

Ein Blatt des Baums entspricht einer Verhaltensänderung. Das DBM sieht vor, dass der Entscheidungsbaum bei der Prüfung mittels einer Tiefensuche durchlaufen wird. Ist eine Bedingung nicht wahr, wird der darunter liegende Teilbaum ignoriert. Wird ein Blatt erreicht, muss die entsprechende Verhaltensänderung durchgeführt werden. Wurde im DBM für diese Änderung das Element *EndAfterChange* gesetzt, wird der Prozess an dieser Stelle abgebrochen. Ansonsten wird der Baum weiter nach Verhaltensänderungen durchsucht. Ein Beispiel für einen solchen Entscheidungsbaum findet sich in Abb. 4.11.

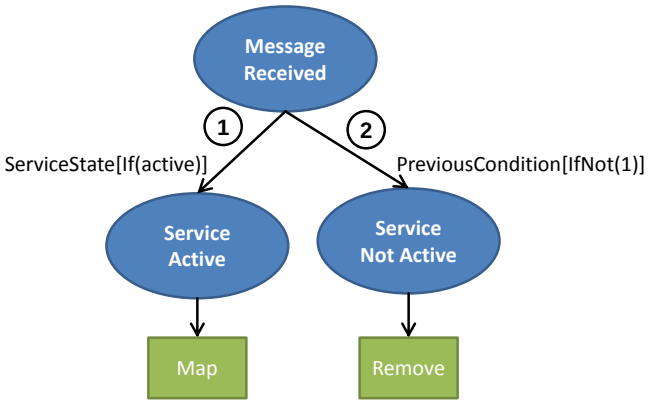


Abbildung 4.11: Beispiel eines DBM Entscheidungsbaums

Das System startet im Zustand *Message Received*. Entsprechend der Tiefensuche wird in Schritt 1 geprüft, ob ein bestimmter Dienst aktiv ist. Dazu wird mittels einer ServiceState-Bedingung geprüft, ob der Dienst die Nachricht *active* zurück liefert. Ist dies der Fall, befindet sich das System in dem Zustand *Service Active* und die Verhaltensänderung Map wird ausgeführt. An dieser Stelle könnte die Tiefensuche mittels des *EndAfterChange*-Elements abgebrochen werden. Falls der Dienst nicht aktiv ist, wird dies in Schritt 2 festgestellt. Hier wird mittels einer PreviousCondition geprüft, ob die Bedingung in Schritt 1 fehlgeschlagen ist. Ist dies der Fall, befindet sich das System im Zustand *Service Not Active* und die Verhaltensänderung Remove wird ausgeführt.

4.8 Vollständigkeit

Das DBM wurde so entwickelt, dass es in der Domäne der Service-Protokolle angewendet werden kann. Dazu verwendet es einen Satz von Mustern, die nicht atomar sind. Das Request-Muster kann beispielsweise durch keine Kombination des Add-, Store- und Merge-Musters abgebildet werden. Da das DBM aber mit dem Hintergrund entwickelt wurde, einen leicht zu verstehenden Weg

zur Übersetzung von Protokollen zu bieten, ist eine atomare Beschreibung der Muster nicht sinnvoll. Im Rahmen dieser Arbeit wurde allerdings nur gezeigt, dass die vorgestellten Muster für die im Rahmen den Analyse beschriebenen Protokolle funktioniert. Da bei anderen Service-Protokollen unvorhergesehene Probleme auftreten können, soll an dieser Stelle auf die Vollständigkeit des DBM eingegangen werden.

Da die Datentransformation nicht Teil des DBM ist, hängt die Vollständigkeit des Ansatzes von der verwendeten Anfrage-Sprache ab. Wird eine turingvollständige Sprache wie XQuery [BCF+07] verwendet, kann bereits das Map-Muster verwendet werden, um beliebige Operationen auf einer eingehenden Nachricht auszuführen. Theoretisch könnte somit jedes der beschriebenen Muster, sowie auch jedes andere, mittels des Map-Musters abgebildet werden.

Das DBM erfüllte aber auch selbst bereits Kriterien der Turing-Vollständigkeit [Tur37, Tur38]. So lassen sich mittels der Bedingungen und des *Timing*-Elements bereits zeitliche Abfolgen beschreiben sowie zeitdiskrete Systeme realisieren. Durch das Store-Muster können Daten gespeichert werden auf die von anderen Mustern aus zugegriffen werden kann. Da beliebige Anfrage-Sprachen auf die Nachrichten ausgeführt werden können, kann somit jede berechenbare Funktion auf eingehende Daten angewendet werden.

Allerdings ist das DBM nicht in der Lage, Schleifen abzubilden. Sollte beispielsweise eine Operation wiederholt ausgeführt werden bis sich der Zustand eines externen Dienstes ändert, könnte diese nicht abgebildet werden ohne die Anzahl der Wiederholungen fest vorzugeben. Diese Einschränkung wird durch die Form der Bedingungen (siehe Abschnitt 4.7) verursacht. Da diese als Entscheidungsbäume interpretiert werden, sind Wiederholungen nicht möglich. Das Ausführen von Schleifen würde ermöglicht werden, wenn die Bedingungen als Graphen interpretiert werden würden. Das Modell müsste dazu nicht angepasst werden, sondern lediglich die Implementierung. Während die Verhaltensänderungen des DBM momentan als sequentielle Liste interpretiert werden, müssten diese von der Implementierung wiederholt durchlaufen werden bis alle vorhandenen Bedingungen erfüllt sind. Allerdings führt dies zu einem höheren Grad der Komplexität bei der Beschreibung von Verhaltensänderungen. Hinzu kommt, dass blockierende Operationen im DBM beschrieben werden könnten, was wiederum eine starken Einfluss auf die Ausführungszeiten hätte.

Somit existieren verschiedene Möglichkeiten, gegebenenfalls fehlende Muster zu erzeugen. Die Verwendung des Map-Musters in Verbindung mit einer turing-mächtigen Anfrage-Sprache ist dabei eine relativ einfache Lösung, solange die Transformation nicht in Abhängigkeit zu anderen Bedingungen steht. Sollte es nötig sein Schleifen mittels des DBM abzubilden, kann diese durch eine Anpassung der Implementierung realisiert werden.

4.9 Zusammenfassung

Dieses Kapitel beschreibt das DBM im Detail. Neben dem grundlegendem Konzept werden die Techniken zur Datentransformation, Identifikation von Protokollen und Nachrichten sowie die Methoden zur Transformation des Kommunikationsverhaltens beschrieben. Darüber hinaus wird gezeigt, dass das Konzept vollständig ist und somit theoretisch auch in anderen Anwendungsbereichen eingesetzt werden kann.

Im Gegensatz zu den in Kapitel 2.2 vorgestellten Anfrage-Technologien, erlaubt das DBM nicht nur die Konvertierung der Daten sondern auch des Kommunikationsverhaltens. Während die in Kapitel 2.1 beschriebenen Middleware-Systeme eine spezifische Implementierung für jedes zu übersetzende Protokoll benötigen, kann das DBM verwendet werden um mittels einer Beschreibung der Unterschiede zwischen den Protokollen eine Übersetzung zu ermöglichen. Das DBM ist als Alternative zu einer konkreten manuellen Implementierung zu sehen. Daher wurde in Kapitel 6.5 im Rahmen einer Studie evaluiert, ob der Initialaufwand zum Erlernen des DBM mit dem einer konkreten Implementierung vergleichbar ist.

Das DBM unterscheidet sich von den in Kapitel 2.3 beschriebenen Modellierungsansätzen in erster Linie, indem es sich nicht auf die formale Spezifikation der Protokolle fokussiert, sondern lediglich auf die Beschreibung der Unterschiede zwischen den Protokollen. Dabei verwendet es kein, wie bei den vorgestellten Ansätzen, abstraktes sondern ein an den Entwicklungsprozess von Protokoll-Konvertern angepasstes Modell. Ähnlich wie bei den vorgestellten Ansätzen verfügt auch das DBM über ein Sub-Modell zur Schnittstellenbeschreibung. Dies ist aber lediglich ein sehr leichtgewichtiges Modell, das zum Identifizieren von Protokollen und Nachrichten dient. Im Gegensatz zu den vorgestellten Modellierungsverfahren ist die Datentransformation bereits Teil des DBM und braucht nur für die tatsächlich auftretenden Fälle und nicht für das gesamte Protokoll beschrieben werden. Da sich das DBM auf die Modellierung von Verhaltensunterschieden konzentriert, kann es darüber hinaus nicht zu Inkonsistenzen durch ungleiche Nachrichten- oder Parameter-Namen, wie sie bei anderen Verfahren auftreten können, kommen.

5 Implementierung

Diese Kapitel beschreibt die konkrete Implementierung des im Rahmen der Arbeit entwickelten generischen Konverters. Dieser erlaubt die Konvertierung von Protokollen auf Basis des Modells, das in Kapitel 4 beschrieben wurde. In Abschnitt 5.1 wird zunächst auf den allgemeinen Konvertierungsprozess eingegangen, während in Kapitel 5.2 die konkrete Umsetzung beschrieben wird. Abschnitt 5.4 fasst das Kapitel zusammen.

5.1 Konvertierungsprozess

Der allgemeine Konvertierungsprozess ist in Abb. 5.1 grafisch dargestellt. Wird eine Nachricht empfangen, wird zunächst das Protokoll der Nachricht identifiziert. Dazu wird ein passendes Protocol Identification Model (PIM), wie in Abschnitt 4.3 beschrieben, in der Bibliothek des Konverters gesucht. Wird ein entsprechendes PIM gefunden, wird dieses verwendet um die Nachricht zu identifizieren. Konnten Protokoll und Nachricht identifiziert werden, wird in der Bibliothek nach einem Differential Behavior Modell (DBM), wie in Abschnitt 4.4 beschrieben, gesucht. Wird ein entsprechendes DBM gefunden, werden die nötigen Verhaltensänderungen identifiziert. Die Liste dieser Verhaltensänderungen wird daraufhin sequentiell vom Konverter abgearbeitet. Sind die Verhaltensänderungen an Bedingungen geknüpft, werden diese vor einer etwaigen Ausführung geprüft. Die Anfrage-Technologien, die unter Umständen zur Konvertierung der Nachrichten eingesetzt werden müssen, werden über die Modelle identifiziert. Am Ende des Konvertierungsprozesses hat der Konverter eine beliebige Anzahl von Nachrichten erzeugt. Mittel des PIM des Ziel-Protokolls und des DBM wird dann geprüft, ob der Konverter die zu erwartenden Nachrichten erzeugt hat. Ist dies der Fall, werden die entsprechenden Nachrichten weitergeleitet. Andernfalls wird eine entsprechende Fehlerbehandlung, wie in Abschnitt 4.6 beschrieben, ausgeführt.

Dieser Prozess erlaubt die Konvertierung beliebiger Protokolle, unabhängig von der Darstellung ihrer Daten sowie der Technologien, die zur Übersetzung der Nachricht und des Kommunikationsverhaltens eingesetzt werden. Er enthält außerdem bereits die nötigen Methoden zur Konsistenzprüfung und Fehlerbehandlung. Der Entwickler selbst muss lediglich die Modelle spezifizieren und falls noch nicht vorhanden, die nötigen Anfrage- und Kommunikationstechnologien implementieren.

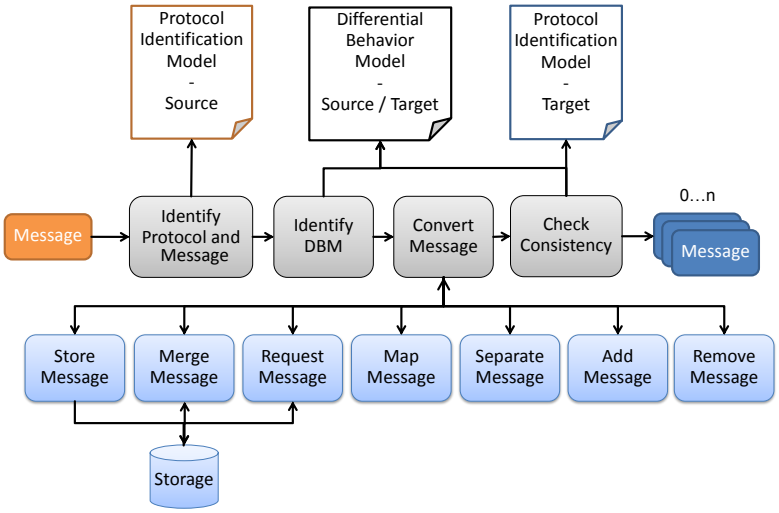


Abbildung 5.1: Schematische Darstellung des Konvertierungsprozesses

5.2 Umsetzung

Um aus der abstrakten Modellbeschreibung eine konkrete Implementierung zu generieren, wurden die Modelle als XML-Schema beschrieben (siehe Anhang A.2 und A.3). Dies hat den Vorteil, dass es Entwickler ermöglicht eine Protokollkonvertierung, die auf dem DBM beruht, mittels der Beschreibungssprache XML zu realisieren. Das erzeugte XML-Dokumente kann dann mittels des erwähnten XML-Schemas auf seine Konsistenz geprüft werden. Außerdem existieren bereits eine Vielzahl von Werkzeugen, die diesen Prozess grafisch unterstützen.

Mittels des Apache XMLBeans-Projekts wurde aus dem XML-Schema entsprechende Java Klassen erzeugt, die innerhalb des Konverters verwendet werden. Diese Klassen bilden das XML-Schema und somit auch das DBM mittels ihrer Objektstruktur ab. Darüber hinaus erlauben die erzeugten Klassen das Analysieren von, auf besagtem XML-Schema basierenden, XML-Dokumenten. Als Resultat wird eine Objektstruktur erzeugt, die dem DBM entspricht und alle vom Entwickler spezifizierten Informationen enthält.

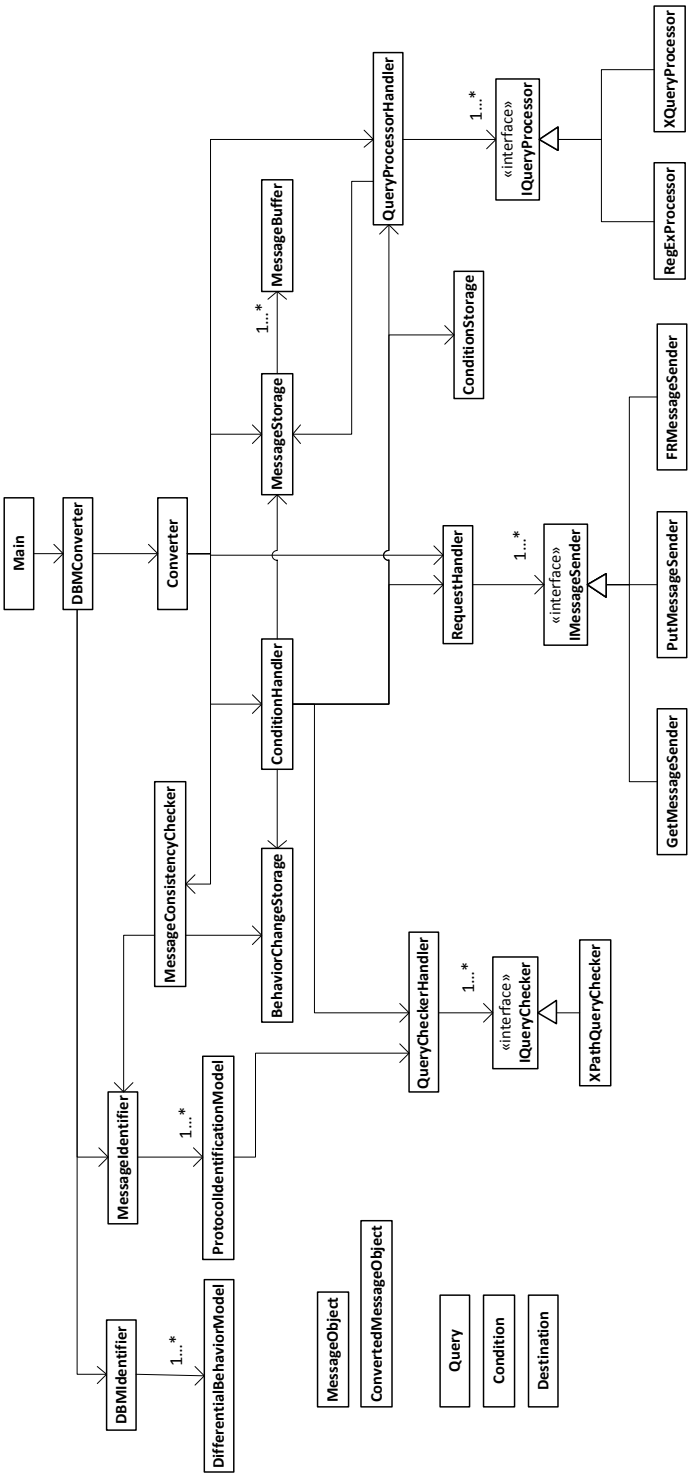


Abbildung 5.2: Klassendiagramm des Konverters

Die Implementierung selbst beruht auf dem Klassendiagramm, das in Abb. 5.2 dargestellt wird. Aus Platzgründen enthält das Klassendiagramm lediglich die entsprechenden Klassennamen. Ein vollständiges Diagramm findet sich in Anhang A.1. Die Implementierung beruht auf der Programmiersprache Java und verwendet die mittels XMLBeans erzeugten Klassen um das vom Entwickler spezifizierte DBM zu analysieren. Da diese Klassen lediglich das DBM abbilden, aber nicht zum Konvertierungsprozess beitragen, wurden sie nicht in das Klassendiagramm aufgenommen. Die einzigen Ausnahmen bilden die Klassen *Condition*, *Query* und *Destination*. Diese werden während der Konvertierung an andere Klassen übergeben und wurden daher mit in das Diagramm aufgenommen.

Die Hauptklasse des Projekts ist *DBMConverter*. Es wird davon ausgegangen, dass sie von einer anderen Klasse (hier *Main*) implementiert wird. Dem DBMConverter können Nachrichten zur Konvertierung übergeben werden. Die Rückgabe ist eine Liste aller erzeugten Nachrichten. Diese sind vom Type *ConvertedMessageObject* und enthalten neben der neuen Nachricht auch Informationen über den Zeitpunkt, zu dem die Nachricht gesendet werden soll.

Wird eine Nachricht übergeben, wird die Klasse *MessageIdentifier* verwendet, um Protokoll- und Nachrichtennamen zu identifizieren. Diese verfügt über eine Liste aller bekannten PIM und werden jeweils mittels der Klasse *ProtocolIdentificationModel* im Speicher gehalten. Jede dieser Klassen hat außerdem Zugriff auf die Klasse *QueryCheckerHandler* und kann somit wiederum verschiedene Anfrage-Technologien mittels der Schnittstelle *IQueryChecker* implementieren. Somit kann die QueryCheckerHandler Klasse verwendet werden, um mittels der spezifizierten Anfrage-Technologien festzustellen, ob die im PIM definierten Bedingungen zutreffen. Konnte die Nachricht identifiziert werden, wird vom MessageIdentifier ein Objekt vom Typ *MessageObject* zurückgegeben. Dieses enthält neben der Nachricht selbst auch den eindeutigen Namen der Nachricht und des Protokolls.

Wurden Protokoll und Nachricht identifiziert, wird vom DBMConverter die Klasse *DBMIdentifier* verwendet, um das benötigt Differential Behavior Model zu identifizieren. Alle DBMs werden jeweils mittels der Klasse *DifferentialBehaviorModel* im Speicher gehalten. Wird ein passendes DBM identifiziert, wird vom DBMIdentifier eine Referenz auf das entsprechende DifferentialBehaviorModel Objekt im MessageObject Objekt gespeichert.

Nach der erfolgreichen Identifizierung des DBM übergibt der DBMConverter das MessageObject an die Klasse *Converter*. Diese führt die eigentliche Konvertierung der Nachricht durch. Dazu identifiziert sie zunächst die nötigen Muster, die angewendet werden müssen. Hierfür wird das im MessageObject referenzierte DBM analysiert. Die Converter Klasse verfügt über alle Methoden, die zur Ausführung der Muster, wie in Abschnitt 4.4 spezifiziert, nötig sind. Die Con-

verter Klasse identifiziert somit die nötigen Konvertierungen und führt diese entsprechend sequentiell durch.

Sollte es nötig sein, eine Nachricht zwischenzuspeichern oder auf eine gespeicherte Nachricht zuzugreifen, wird dies vom Converter mittels der Klasse *MessageStorage* realisiert. Diese Klasse legt jeweils, mittels einer Instanz der Klasse *MessageBuffer*, die entsprechende Nachricht im Speicher ab und erlaubt die Identifikation mittels ihres Nachrichten- und Protokollnamen.

Sollte eine Datentransformation nötig sein, wird die Klasse *QueryProcessorHandler* verwendet. Diese verfügt über eine Liste aller, mittels der Schnittstelle *IQueryProcessor* implementierten Anfrage-Technologien, die zur Konvertierung verwendet werden können. Außerdem kann sie auf gespeicherte Nachrichten über die MessageStorage Klasse zugreifen. Dies ist beispielsweise nötig, wenn mehrere Nachrichten mittels des Merge-Musters kombiniert werden.

Müssen zum Konvertieren der Nachricht Anfragen an einen Dienst gestellt werden, wird dies über die Klasse *RequestHandler* realisiert. Diese ermöglicht die Kommunikation mittels der Schnittstelle *IMessageSender* implementierten Kommunikationstechnologien. Hier können sowohl allgemeine Kommunikationstechnologien, wie REST-Anfragen, implementiert werden aber auch spezialisierte Technologien, die nur für bestimmte Dienste funktionieren.

Sind Muster an Bedingungen geknüpft, werden diese von der Klasse *ConditionHandler* behandelt. Da diese Klasse komplexe Bedingungen prüfen muss, die auf gespeicherten Nachrichten, dem Zustand externer Dienste und dem Inhalt von Nachrichten beruhen können, hat die Klasse Zugriff auf die bereits erwähnten Klassen QueryCheckerHandler, RequestHandler, MessageStorage und QueryProcessorHandler. Außerdem kann der ConditionHandler mittels der Klassen *BehaviorChangeStorage* und *ConditionStorage* prüfen, ob bestimmte Verhaltensänderungen, beziehungsweise Bedingungen, zuvor erfolgreich ausgeführt wurden.

Wurden alle im DBM beschriebenen Konvertierungen durchgeführt, verwendet der Converter die Klasse MessageConsistencyChecker um zu prüfen ob Fehler bei der Konvertierung aufgetreten sind. Dazu werden die bereits beschriebenen Klassen MessageIdentifier und *BehaviorChangeStorage* verwendet. Wird ein Fehler festgestellt, wird die im DBM spezifizierte Fehlerbehandlung vom Converter durchgeführt. Wurden alle Konvertierungen durchgeführt, wird vom Converter eine Liste der erzeugten Nachrichten, jeweils mittels eines ConvertedMessageObject, an den DBMConverter zurückgegeben. Dieser gibt die Liste der neuen Nachrichten an die eigentliche Applikation zurück.

5.3 Anwendungsbeispiele

Um ein besseres Verständnis für das DBM zu ermöglichen, werden an dieser Stelle Anwendungsbeispiele für das DBM aufgezeigt. Dazu werden die Funktionen des DBM an Nachrichten aus den Protokollen, die in Kapitel 3 analysiert wurden, demonstriert. Es geht bei diesen Beispielen weniger darum, eine konkrete Übersetzung zwischen den beteiligten Protokollen zu erreichen, als die Funktionsweise des DBM an realen Beispiele zu verdeutlichen. Die Anfragen wurden aus Platzgründen nicht in die Beispiele aufgenommen.

5.3.1 Protokollidentifikation

Damit der Konverter in der Lage ist Nachrichten der beiden Protokolle zu identifizieren, müssen diese zunächst mittels des PIM beschrieben werden. Dazu müssen Elemente innerhalb der Nachrichten spezifiziert werden, mittels derer das Protokoll eindeutig identifiziert werden kann. Dies soll hier an Hand einer SCAI-Nachricht (Version 2.0) verdeutlicht werden.

```
<ProtocolModel xmlns='http://cbusemann.de/schema/protocolModel'>
    <protocolName>EX1</protocolName>
    <ProtocolIdentification>
        <IdentificationElement>
            <queryType>XPath</queryType>
            <query>/EX1</query>
            <Property>
                <key>defaultNamespace</key>
                <value>http://xml.cbusemann.de/schema/example1</value>
            </Property>
            <Property>
                <key>minOccurNumber</key>
                <value>1</value>
            </Property>
        </IdentificationElement>
    </ProtocolIdentification>
    <Message>
            ...
    </Message>
    ...
</ProtocolModel>
```

Abbildung 5.3: Beispiel-Protokollidentifikation SCAI-Nachrichten

Da jede SCAI-Nachricht mit dem Element *SCAI* beginnt und den eindeutigen Namespace *http://xml.offis.de/schema/SCAI-2.0* verwendet, kann dieses Element verwendet werden um das Protokoll zu identifizieren. Abb. 5.3 zeigt das entsprechende PIM. Zunächst wird mittels des *protocolName*-Elements festgelegt, dass dem Protokoll der Name *SCAI2.0* zugewiesen wird. Mittels des *queryType* Elements wird angegeben, dass eine XPath-Anfrage verwendet wird um die

Prüfung durchzuführen. Das *query*-Element spezifiziert dabei den Pfad, nach dem gesucht werden soll. In diesem Fall ist dies ein *SCAI*-Element, das alle anderen Elemente umgibt. Der Namespace des Elements wird mittels des *Property*-Elements *defaultNamespace* definiert. Darüber hinaus wird die *Property*-Elements *minOccurNumber* festgelegt, wie oft das *SCAI*-Element mindestens auftreten muss.

Eine Nachricht wird somit als *SCAI2.0*-Nachricht identifiziert, wenn sie mindestens ein *SCAI*-Element enthält, das alle anderen Elemente umgibt und den Namespace *http://xml.offis.de/schema/SCAI-2.0* verwendet. Sollte es nötig sein, können an dieser Stelle weitere *IdentificationElement*-Elemente hinzugefügt werden, um andere Elemente zu beschreiben, die innerhalb der Nachricht vorkommen müssen. Dazu können natürlich auch andere Anfrage-Sprachen als XPath eingesetzt werden. Außerdem können weitere *ProtocolIdentification*-Elemente hinzugefügt werden. Werden bei der Überprüfung alle *IdentificationElement*-Elemente eines *ProtocolIdentification*-Elements gefunden, gilt das Protokoll als identifiziert.

5.3.2 Nachrichtenidentifikation

Um eine Nachricht innerhalb des Protokolls zu identifizieren, müssen dem PIM die entsprechenden *Message*-Elemente hinzugefügt werden. Jedes *Message*-Element identifiziert genau eine Nachricht. Der Aufbau ist dabei dem der Nachrichtenidentifikation sehr ähnlich. Abb. 5.4 zeigt ein *Message*-Element, dass verwendet werden kann, um die *Mesaurement*-Nachricht (siehe Abb. 5.5) des SCAI-Protokolls zu identifizieren.

```
1  <Message>
2      <messageName>Measurement</messageName>
3      <IdentificationElement>
4          <queryType>XPath</queryType>
5          <query>/SCAI/Payload/Measurements</query>
6          <Property>
7              <key>defaultNamespace</key>
8              <value>http://xml.offis.de/schema/SCAI-2.0</value>
9          </Property>
10         <Property>
11             <key>minOccurNumber</key>
12             <value>1</value>
13         </Property>
14     </IdentificationElement>
15 </Message>
```

Abbildung 5.4: Message-Element für SCAI-Measurement-Nachrichten

Mittels des *messageName*-Elements wird der Nachricht zunächst der eindeutige Name *Measurement* zugewiesen. Genau wie bei der Protokollidentifikation wird

hier eine XPath-Anfrage verwendet, um ein Element der Nachricht zu identifizieren. Allerdings wird in diesem Fall nach dem Element *Mesurements* gesucht, das sich in dem Pfad */SCAI/Payload* befinden muss. Wird das entsprechende Element gefunden, gilt die Nachricht als identifiziert. Auch hier können natürlich weitere *IdentificationElement*-Elemente hinzugefügt werden, um die Identifikation der Nachricht weiter einzuschränken.

5.3.3 Anlegen des DBM

Um eine Nachricht zu konvertieren muss diese zunächst mittels des PIM identifiziert werden. Daraufhin kann ein DBM angelegt werden, das die Unterschiede zwischen diesem und einem anderem Protokoll beschreibt.

```
<SCAI xmlns="http://xml.offis.de/schema/SCAI-2.0">
    <Payload>
        <Measurements>
            <DataStream>
                <timeStamp>2010-09-19T16:08:31.443+00:00</timeStamp>
                <SensorName>Multisensor13</SensorName>
                <SensorDomainName>City17</SensorDomainName>
                <DataStreamElement path="Temp"  >
                    <Data>27</Data>
                </DataStreamElement>
                <DataStreamElement path="Motion"  >
                    <Data>5</Data>
                </DataStreamElement>
            </DataStream>
        </Measurements>
    </Payload>
</SCAI>
```

Abbildung 5.5: Beispiel einer SCAI-Measurement-Nachrichten

Ein Beispiel einer SCAI-Measurement-Nachricht findet sich in Abb. 5.5. Da diese Nachricht alle im PIM spezifizierten Kriterien erfüllt, wird sie vom Konverter als *Measurement*-Nachricht des *SCAI2.0*-Protokolls identifiziert. Um eine Konvertierung dieser Nachricht durchzuführen kann das DBM aus Abb. 5.6 verwendet werden.

Hier werden zunächst die beteiligten Protokolle mittels der Elemente *protocolName1* und *protocolName2* identifiziert. Diese Namen referenzieren die Protokollnamen, die mittels des PIM spezifiziert wurden. Um Verhaltensänderungen zu beschreiben, muss jeweils ein entsprechendes *BehaviorChange*-Element hinzugefügt werden. Diese werden in den folgenden Abschnitten an Hand von Bespielen beschrieben.

```
<DiffModel xmlns='http://cbusemann.de/schema/diffModel'>
    <protocolName1>SCAI2.0</protocolName1>
    <protocolName2>SOS1.0</protocolName2>
    <BehaviorChange>
            ...
    </BehaviorChange>
</DiffModel>
```

Abbildung 5.6: DBM-Beispiel

5.3.4 Übersetzen einer Nachricht

Um eine Nachricht in ein anderes Format zu bringen, muss dem DBM ein *BehaviorChange*-Element, das ein *Add*-Element enthält, hinzugefügt werden. Um die Nachricht aus Abb. 5.5 in eine SWE-Nachricht zu konvertieren, kann das *BehaviorChange*-Element aus Abb. 5.7 verwendet werden.

```
<BehaviorChange>
    <name>MapExample</name>
    <MessageIdentification>
        <messageName>Measurement</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <Map>
        <mappedMessageName>InsertObservation</mappedMessageName>
        <Query>
            <queryType>XQuery</queryType>
            <query><![CDATA[ ... ]]></query>
        </Query>
    </Map>
</BehaviorChange>
```

Abbildung 5.7: Map-Beispiel

Der Verhaltensänderung wird hier zunächst mittels des *name*-Elements der eindeutige Name *MapExample* zugewiesen. Dieser kann beispielsweise in Bedingungen verwendet werden, um zu prüfen ob die Verhaltensänderung ausgeführt wurde. Danach wir mittels des *MessageIdentification*-Elements definiert, dass die Verhaltensänderung auf die Nachricht *Measurement* des *SCAI2.0*-Protokolls angewendet werden soll. Um die Verhaltensänderung auf mehrere Nachrichten auszuführen, können hier beliebig viele *MessageIdentification*-Element hinzugefügt werden. Danach wird mittels des *Map*-Elements festgelegt, dass die Nachricht in ein anderes Format übersetzt werden soll. Das *mappedMessageName*-Element gibt dabei an, das die Nachricht *InsertObservation* erzeugt werden soll. In diesem Fall wird die Transformation mittels einer *XQuery*-Anfrage durchge-

führt. Das Ergebnis der Konvertierung ist eine neue Nachricht, die mittels der entsprechenden Anfrage erzeugt wurde.

5.3.5 Hinzufügen einer Nachricht

Um eine neue Nachricht zu erzeugen, muss dem DBM ein *BehaviorChange*-Element, das ein *Map*-Element enthält, hinzugefügt werden. Um beim Eintreffen einer *getSensor*-Nachricht des SCAI-Protokolls eine Bestätigung an den Quelldienst zu senden kann das Beispiel aus Abb. 5.8 verwendet werden.

```
<BehaviorChange>
    <name>AddExample</name>
    <MessageIdentification>
        <messageName>createSensor</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <Add>
        <addedMessageName>Acknowledgment</addedMessageName>
        <addedMessageProtocolName>SCAI2.0</addedMessageProtocolName>
        <Query>
            <queryType>XQuery</queryType>
            <query><![CDATA[ ... ]]></query>
        </Query>
        <Destination>
            <Source/>
        </Destination>
    </Add>
</BehaviorChange>
```

Abbildung 5.8: Add-Beispiel

Hier wird mittels des *Add*-Elements spezifiziert, dass eine neue Nachricht hinzugefügt werden soll. Mittels des *MessageIdentification*-Elements wird festgelegt, dass die neue Nachricht hinzugefügt werden soll, wenn die Nachricht *createSensor* empfangen wird. Das Element *addedMessageName* legt fest, das die neue Nachricht den Namen *Acknowledgment* haben muss. Das Element *addedMessageProtocolName* legt entsprechend fest, das der Protokollname der neuen Nachricht *SCAI2.0* sein muss. Mittels des *Query*-Elements wird die Anfrage spezifiziert, die verwendet wird um die neue Nachricht zu erzeugen. In diesem Fall wird eine *XQuery*-Anfrage verwendet. Mittels des *Destination*-Elements wird das Ziel der Nachricht angegeben. In diesem Fall wird die Nachricht an den Quelldienst zurück gesendet. Das Ergebnis der Konvertierung sind zwei Nachrichten: Die Original-Nachricht und die vom Konverter erzeugte Nachricht. Werden keine weiteren Verhaltensänderungen auf die Original-Nachricht angewendet, wird diese an den Zieldienst gesendet während die neue Nachricht an den Quelldienst gesendet wird.

5.3.6 Entfernen einer Nachricht

Um eine Nachricht zu entfernen muss ein *BehaviorChange*-Element, das ein *Remove*-Element enthält, hinzugefügt werden. Um die *InsertObservationResponse*-Nachricht des SWE-Protokolls zu entfernen kann das Beispiel aus Abb. 5.9 verwendet werden.

```
<BehaviorChange>
    <name>RemoveExample</name>
    <MessageIdentification>
        <messageName>InsertObservationResponse</messageName>
        <protocolName>SOS1.0</protocolName>
    </MessageIdentification>
    <Remove/>
</BehaviorChange>
```

Abbildung 5.9: Remove-Beispiel

In diesem Fall wird die Nachricht *InsertObservationResponse* des Protokolls *SOS1.0* lediglich mittels eines *MessageIdentification*-Elements identifiziert. Das *Remove*-Element selbst bedarf keiner weiteren Konfiguration. Wird eine Nachricht entfernt, können keine weiteren Verhaltensänderungen auf sie angewendet werden. Das heißt, das Entfernen sollte immer nach allen anderen nötigen Verhaltensänderungen auf eine Nachricht angewendet werden.

5.3.7 Speichern einer Nachricht

Um eine Nachricht im Konverter zu speichern muss ein *BehaviorChange*-Element, das ein *Store*-Element enthält, hinzugefügt werden. Um die *createSensor-*, *createDataStreamType-* und *createSensorType*-Nachricht des SCAI-Protokolls zu speichern kann das Beispiel aus Abb. 5.10 verwendet werden.

Hier werden die Nachrichten, die gespeichert werden sollen, jeweils mittels eines *MessageIdentification*-Elements identifiziert. Das *Store*-Element erlaubt die Spezifikation einer Ablaufzeit mittels des *timeout*-Elements. Wird eine Nachricht gespeichert wird sie nach Ablauf dieser Zeit (hier 60 Sekunden) wieder gelöscht. Dies verhindert nicht nur die langfristige Speicherung alter Daten sondern auch, dass Konflikte durch alte Nachrichten entstehen. Wird keine weitere Operation ausgeführt, wird die eingehende Nachricht nach dem Speichern an den Zieldienst gesendet.

```
<BehaviorChange>
    <name>StoreExample</name>
    <MessageIdentification>
        <messageName>createSensor</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <MessageIdentification>
        <messageName>createDataStreamType</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <MessageIdentification>
        <messageName>createSensorType</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <Store>
        <timeout>60</timeout>
    </Store>
</BehaviorChange>
```

Abbildung 5.10: Store-Beispiel

5.3.8 Anfragen von Nachrichten

Um eine Nachricht anzufragen muss ein *BehaviorChange*-Element, das ein *Request*-Element enthält, hinzugefügt werden. Um die fehlenden Daten der *getSensorResponse*-Nachricht des SCAI-Protokolls bei einer Übersetzung in das SWE-Protokoll anzufragen kann das Beispiel aus Abb. 5.10 verwendet werden.

Mittels des *MessageIdentification*-Elements wird hier festgelegt, das die Anfrage gesendet wird, wenn die Nachricht *getSensorResponse* empfangen wird. Innerhalb des *Request*-Elements können beliebig viele Nachrichten angefragt werden. Dazu muss jede Anfrage mittels eines *RequestMsg*-Elements spezifiziert werden. Diese enthält ein *Ouery*-Element das verwendet wird, um aus der eingehenden und allen gespeicherten Nachrichten eine Anfrage-Nachricht zu generieren. In diesem Fall wird dazu eine XQuery-Anfrage verwendet. Analog zum Speichern von Daten wird mittels des *storeTimeout*-Elements eine Zeitspanne angegeben werden, nach der die gespeicherte Nachricht gelöscht wird (hier 60 Sekunden). Mittels des *Target*-Elements wird das Ziel der Anfrage angegeben. In diesem Fall wird die Anfrage an den Zieldienst gesendet. Die Antwort auf die Anfrage wird im Konverter gespeichert. Wird mehr als eine Anfrage gesendet, kann die Antwort auf die ersten Anfrage verwendet werden um eine weitere Anfrage zu generieren. Wird keine weitere Verhaltensänderung ausgeführt, wird die eingehende Nachricht nach dem Senden der Anfragen an den Zieldienst gesendet.

```
<BehaviorChange>
    <name>RequestExample</name>
    <MessageIdentification>
        <messageName>getSensorResponse</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <Request>
        <RequestMsg>
            <Query>
                <queryType>XQuery</queryType>
                <query><![CDATA[ ... ]]></query>
            </Query>
            <storeTimeout>60</storeTimeout>
            <Target>
                <Destiantion/>
            </Target>
        </RequestMsg>
    </Request>
</BehaviorChange>
```

Abbildung 5.11: Request-Beispiel

5.3.9 Aufteilen einer Nachricht

Um eine Nachricht aufzuteilen muss ein *BehaviorChange*-Element, das ein *Seperate*-Element enthält, hinzugefügt werden. Um die *RegisterSensor*-Nachricht des SWE-Protokolls in die Nachrichten *createDataStreamType* und *createSensorType* des SCAI-Protokoll aufzuteilen, kann das Beispiel aus Abb. 5.12 verwendet werden.

Mittels des *MessageIdentification*-Elements wird hier festgelegt, das die Nachricht *RegisterSensor* des Protokolls *SCAI2.0* aufgeteilt werden soll. Für jede neue Nachricht wird innerhalb des *Seperated*-Elements ein *SeperatedMsg*-Element hinzugefügt. In diesem Fall wurde dies jeweils für die beiden zu erzeugen Nachrichten gemacht. Jedes der *SeperatedMsg*-Elemente enthält ein *seperatedMessageName*-Element, das den Namen der neuen Nachricht angibt. Außerdem wird jeweils mittels eines *Query*-Elements die Anfrage spezifiziert, die verwendet wird um aus der eingehenden Nachricht die neue zu generieren. In diesem Beispiel werden dazu XQuery-Anfragen verwendet. Das Ergebnis dieser Konvertierung sind die zwei neuen Nachrichten.

5.3.10 Vereinen einer Nachricht

Um eine Nachricht aus mehreren eingehenden Nachrichten zu erzeugen, müssen diese zunächst, wie oben beschrieben, im Konverter gespeichert werden. Zudem muss ein *BehaviorChange*-Element, das ein *Merge*-Element enthält, hinzugefügt werden. Um die SCAI-Nachrichten *createSensor*, *createDataStreamType*

```
<BehaviorChange>
    <name>SeperateExample</name>
    <MessageIdentification>
        <messageName>RegisterSensor</messageName>
        <protocolName>SOS1.0</protocolName>
    </MessageIdentification>
    <Seperate>
        <SeperatedMsg>
            <seperatedMessageName>createDataStreamType</seperatedMessageName>
            <Query>
                <queryType>XQuery</queryType>
                <query><![CDATA[ ... ]]></query>
            </Query>
        </SeperatedMsg>
        <SeperatedMsg>
            <seperatedMessageName>createSensorType</seperatedMessageName>
            <Query>
                <queryType>XQuery</queryType>
                <query><![CDATA[ ... ]]></query>
            </Query>
        </SeperatedMsg>
    </Seperate>
</BehaviorChange>
```

Abbildung 5.12: Seperate-Beispiel

und *createSensorType* zu der SWE-Nachricht *RegisterSensor* zu vereinen, kann das Beispiel aus Abb. 5.13 verwendet werden.

Hier werden zunächst alle Nachrichten, die benötigt werden, um die neue Nachricht zu erzeugen, jeweils mittels eines *MessageIdentification*-Elements identifiziert. Da diese bereits durch das vorherigen Beispiel im Konverter gespeichert werden, können die Nachrichten in beliebiger Reihenfolge empfangen werden. Wird eine dieser Nachrichten empfangen, wird diese zunächst gespeichert. Daraufhin wird geprüft, ob alle Nachrichten zur Vereinigung vorhanden sind. Ist dies der Fall, wird die Anfrage, die über das *Query*-Element spezifiziert wurde, verwendet, um die neue Nachricht zu erzeugen. Der Name der erzeugten Nachricht muss *RegisterSensor* lauten, da dieser mittels des Elements *mergedMessageName*-Elements angegeben wurde. Das Ergebnis dieser Konvertierung ist eine neue Nachricht, die aus mehreren eingehenden Nachrichten erzeugt wurde.

5.3.11 Übertragungszeitpunkt festlegen

Um den Übertragungszeitpunkt einer Nachricht festzulegen, kann das *Timing*-Element verwendet werden. Diese kann den Elementen *Add*, *Map*, *Merge*, und *Seperate* hinzugefügt werden. Abb. 5.14 zeigt, wie der Übertragungszeitpunkt auf eine bestimmte Zeit nach dem Empfang der Nachricht gesetzt werden kann.

```
<BehaviorChange>
    <name>MergeExample</name>
    <MessageIdentification>
        <messageName>createSensor</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <MessageIdentification>
        <messageName>createDataStreamType</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <MessageIdentification>
        <messageName>createSensorType</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <Merge>
        <mergedMessageName>RegisterSensor</mergedMessageName>
        <Query>
            <queryType>XQuery</queryType>
            <query><![CDATA[ ... ]]></query>
        </Query>
    </Merge>
</BehaviorChange>
```

Abbildung 5.13: Merge-Beispiel

```
<Timing>
    <Delay>
        <min>1</min>
        <max>5</max>
    </Delay>
</Timing>
```

Abbildung 5.14: Timing-Beispiel (Delay)

Hier wird mittels des *min*-Elements festgelegt, dass die konvertierte Nachricht frühestens 1 Sekunde nach dem Empfang der eingehenden Nachricht gesendet werden darf. Das *max*-Element legt fest, dass die konvertierte Nachricht mindestens 5 Sekunden nach dem Empfang der eingehenden Nachricht gesendet worden sein muss. Der Konverter berechnet diese Zeitpunkte und gibt sie zu jeder konvertierten Nachricht, die von ihm zurückgegeben wird, an.

Eine weitere Möglichkeit, um den Übertragungszeitpunkt einer Nachricht festzulegen, bietet das *TimeOfDay*-Element des *Timing*-Elements. Ein Beispiel dazu findet sich in Abb. 5.15. Hier wird mittels des *time*-Elements die Uhrzeit festgelegt, zu der die Nachricht übertragen werden soll. Das Element *windowSize* gibt dabei die Zeit an, nach der die Nachricht gesendet worden sein muss. In diesem Fall sind dies 5 Sekunden. Das *Timing*-Element erlaubt die Angabe beliebig vieler *TimeOfDay*-Elemente.

```
<Timing>
    <TimeOfDay>
        <time>23:42:05</time>
        <windowSize>5</windowSize>
    </TimeOfDay>
</Timing>
```

Abbildung 5.15: Timing-Beispiel (TimeOfDay)

5.3.12 Kombinieren von Verhaltensänderungen

Um mehrere Verhaltensänderungen auf die gleiche Nachricht anzuwenden, muss lediglich das jeweilige *MessageIdentification*-Element innerhalb mehrerer *BehaviorChange*-Elemente verwendet werden. Ein Beispiel dazu findet sich in Abb. 5.16.

```
<BehaviorChange>
    <MessageIdentification>
        <messageName>ExampleMsg</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <Add>
        ...
    </Add>
</BehaviorChange>
<BehaviorChange>
    <MessageIdentification>
        <messageName>ExampleMsg</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <Store>
        <timeout>60</timeout>
    </Store>
</BehaviorChange>
<BehaviorChange>
    <name>EX1Remove</name>
    <MessageIdentification>
        <messageName>ExampleMsg</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <Remove/>
</BehaviorChange>
```

Abbildung 5.16: Beispiel von kombinierten Verhaltensänderungen

Hier wird beim Erhalt der Nachricht *ExampleMsg* zunächst das Add-Muster verwendet, um eine zusätzliche Nachricht zu erzeugen. Daraufhin wird die eingehende Nachricht mittels des Store-Musters im Konverter für 60 Sekunden gespeichert. Abschließend wird die Nachricht vom Kommunikationsprozess entfernt. Das Ergebnis dieser Konvertierung ist eine neue Nachricht, die mit-

tels des Add-Musters erzeugt wurde. Die gespeicherte Kopie der eingehenden Nachricht kann im Anschluss von anderen Verhaltensänderungen verwendet werden. Beim Anlegen von Verhaltensänderungen gilt es zu beachten, das diese immer sequentiell vom Konverter abgearbeitet werden. Das heißt, würde das Remove-Muster am Anfang der Liste stehen, könnten die folgenden Operationen nicht mehr ausgeführt werden.

5.3.13 Fehlerbehandlung

Da bei der Konvertierung der Nachrichten Anfragen auf Nachrichten angewendet werden, deren Inhalt sich ändern kann, können bei der Konvertierung einer Nachricht theoretisch Fehler auftreten. Diese werden spätestens bei der Konsistenzprüfung der Nachricht durch den Konverter festgestellt. Tritt ein solcher Fall auf, verhält sich der Konverter so, als wenn die entsprechende Verhaltensänderung nicht im DBM spezifiziert worden wäre. Der Entwickler hat aber die Möglichkeit das Verhalten im Fehlerfall zu beeinflussen. Dazu kann jedem *BehaviorChange*-Element ein *ErrorHandling*-Element hinzugefügt werden. Die möglichen Fehlerbehandlungen sollen hier anhand von Beispielen demonstriert werden.

```
<BehaviorChange>
    <MessageIdentification>
        <messageName>ExampleMsg</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <ErrorHandling>
        <RemoveMessage/>
    </ErrorHandling>
    <Map>
        ...
    </Map>
</BehaviorChange>
```

Abbildung 5.17: Beispiel eine Fehlerbehandlung (RemoveMessage)

In Abb. 5.17 findet sich ein Beispiel, bei dem im Fehlerfall die Nachricht vom Kommunikationsprozess entfernt wird. Das heißt, es wird weder die fehlerhaft erzeugte Nachricht, noch die Original-Nachricht weitergeleitet. Außerdem wird der Konvertierungsprozess nach dieser Verhaltensänderung abgebrochen.

In Abb. 5.18 findet sich ein Beispiel, bei dem im Fehlerfall die fehlerhaft erzeugte Nachricht an ihr jeweiliges Ziel weitergeleitet wird. Dies kann allerdings nur durchgeführt werden, wenn von der jeweiligen Anfrage tatsächlich eine Nachricht produziert wurde. Ist dies nicht der Fall, wird die Verhaltensänderung ignoriert. Sollten nach dieser Verhaltensänderung weitere folgen, werden diese wie gewohnt ausgeführt.

```
<BehaviorChange>
    <MessageIdentification>
        <messageName>ExampleMsg</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <ErrorHandling>
        <ForwardMessage/>
    </ErrorHandling>
    <Map>
        ...
    </Map>
</BehaviorChange>
```

Abbildung 5.18: Beispiel eine Fehlerbehandlung (ForwardMessage)

```
<BehaviorChange>
    <MessageIdentification>
        <messageName>ExampleMsg</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <ErrorHandling>
        <OriginalMessage/>
    </ErrorHandling>
    <Map>
        ...
    </Map>
</BehaviorChange>
```

Abbildung 5.19: Beispiel eine Fehlerbehandlung (OriginalMessage)

In Abb. 5.19 findet sich ein Beispiel, bei dem im Fehlerfall die Original-Nachricht anstelle der fehlerhaft erzeugten Nachricht an das jeweilige Ziel weitergeleitet wird. Sollten weitere Verhaltensänderungen nach dieser folgen, werden diese wie gewohnt ausgeführt.

In Abb. 5.20 findet sich ein Beispiel, bei dem im Fehlerfall die mittels des *Query*-Elements definierte Anfrage verwendet wird, um eine alternative Nachricht zu produzieren. Diese könnte beispielsweise eine Nachricht an den jeweiligen Dienst sein, die diesen informiert, dass ein Fehler aufgetreten ist. Sollte bei dieser Anfrage ebenfalls ein Fehler auftreten, wird die entsprechende Verhaltensänderung ignoriert. Sollten weitere Verhaltensänderungen folgen, werden diese in jedem Fall wie gewohnt ausgeführt.

5.3.14 Prüfen von Bedingungen

Um die Ausführung einer Verhaltensänderung an eine Bedingung zu knüpfen, muss dem jeweiligen *BehaviorChange*-Element ein *Condition*-Element hinzuge-

```
<BehaviorChange>
                <name>ErrorExample</name>
    <MessageIdentification>
        <messageName>ExampleMsg</messageName>
        <protocolName>SCAI2.0</protocolName>
    </MessageIdentification>
    <ErrorHandling>
            <Query>
                <queryType>XQuery</queryType>
                <query><![CDATA[ ... ]]></query>
            </Query>
        </ErrorHandling>
    <Map>
        ...
    </Map>
</BehaviorChange>
```

Abbildung 5.20: Beispiel eine Fehlerbehandlung (Query)

fügt werden. Dabei stehen dem Entwickler verschiedene Bedingungen zur Verfügung.

Wird eine Bedingung hinzugefügt, kann zunächst mittels des Elements *name* ein Name für die Bedingung vergeben werden. Soll zu einem späteren Zeitpunkt geprüft werden, ob diese Bedingung wahr war, kann dazu dieser Name verwendet werden. Danach muss mittels des *ConditionType*-Elements festgelegt werden, ob die Bedingung erfüllt werden muss oder nicht. Dazu muss entweder das *if*-Element oder das *ifNot*-Element hinzugefügt werden. Darauf folgt die Definition der Bedingung selbst, mittels des spezifischen *Condition*-Elements.

```
    <Condition>
        <name>MSExample</name>
        <ConditionType>
            <ifNot/>
        </ConditionType>
        <ConditionPredicate>
            <MessageStored>
                <MessageIdentification>
                    <messageName>createSensor</messageName>
                    <protocolName>SCAI2.0</protocolName>
                </MessageIdentification>
            </MessageStored>
        </ConditionPredicate>
    </Condition>
    <Map>
        ...
    </Map>
```

Abbildung 5.21: Beispiel eine Bedingung (MessageStored)

In Abb. 5.21 findet sich ein Beispiel für eine *MessageStored*-Bedingung. Diese wird verwendet, um zu prüfen, ob eine bestimmte Nachricht vom Konverter gespeichert wurde. Da das *ifNot*-Element gesetzt wurde, ist diese Bedingung wahr, wenn die Nachricht nicht gespeichert wurde. In der Bedingung selbst wird mittels eines *MessageIdentification*-Elements spezifiziert, das nach der Nachricht *createSensor* des Protokolls *SCAI2.0* gesucht wird.

```
1  <Condition>
2      <name>PCExample</name>
3      <ConditionType>
4          <if/>
5      </ConditionType>
6      <ConditionPredicate>
7          <PreviousCondition>
8              <condtionName>MSExample</condtionName>
9          </PreviousCondition>
10     </ConditionPredicate>
11 </Condition>
12 <Map>
13     ...
14 </Map>
```

Abbildung 5.22: Beispiel eine Bedingung (PreviousCondition)

In Abb. 5.22 findet sich ein Beispiel für eine *PreviousCondition*-Bedingung. Mittels dieser kann geprüft werden, ob eine vorherige Bedingung wahr war. Die vorherige Bedingung wird dabei mittels des *condtionName*-Elementes angegeben. Dabei muss die Bedingung selbst während des aktuellen Konvertierungsprozesses ausgeführt worden sein. In diesem Beispiel wird geprüft ob die Bedingung *MSExample* zuvor erfolgreich geprüft wurde.

```
1  <Condition>
2      <name>PBCExample</name>
3      <ConditionType>
4          <if/>
5      </ConditionType>
6      <ConditionPredicate>
7          <PreviousBehaviorChange>
8              <behaviorChangeName>MergeExample</behaviorChangeName>
9          </PreviousBehaviorChange>
10     </ConditionPredicate>
11 </Condition>
12 <Map>
13     ...
14 </Map>
```

Abbildung 5.23: Beispiel eine Bedingung (PreviousBehaviorChange)

In Abb. 5.23 findet sich ein Beispiel für eine *PreviousBehaviorChange*-Bedingung. Diese funktioniert ähnlich wie eine *PreviousCondition*-Bedingung, nur dass in diesem Fall geprüft wird, ob eine bestimmte Verhaltensänderung ausgeführt wurde. Die Verhaltensänderung wird dabei mittels des *behaviorChangeName*-Elements identifiziert. In diesem Fall wird also geprüft, ob die Verhaltensänderung *MergeExample* zuvor erfolgreich ausgeführt wurde.

```
    </Condition>
        <name>ConditionSS</name>
        <ConditionType>
            <if/>
        </ConditionType>
        <ConditionPredicate>
            <ServiceState>
                <Request>
                    <Query>
                        <queryType>XQuery</queryType>
                        <query><![CDATA[ ... ]]></query>
                    </Query>
                    <ResponseIdentification>
                        <messageName>getSensorResponse</messageName>
                        <protocolName>SCAI2.0</protocolName>
                    </ResponseIdentification>
                    <storeTimeout>60</storeTimeout>
                    <Target>
                        <Source/>
                    </Target>
                </Request>
            </ServiceState>
        </ConditionPredicate>
    </Condition>
    <Map>
        ...
    </Map>
```

Abbildung 5.24: Beispiel eine Bedingung (ServiceState)

In Abb. 5.24 findet sich ein Beispiel einer *ServiceState*-Bedingung. Diese wird verwendet, um die Ausführung einer Verhaltensänderung an den Zustand eines Dienstes zu koppeln. Dazu wird eine Nachricht an den jeweiligen Dienst gesendet und die Antwort analysiert. Die Nachricht wird dabei mittels einer Anfrage, die über das *Query*-Element spezifiziert wird, erzeugt. In diesem Fall wird eine XQuery verwendet um die Nachricht zu erzeugen. Die Antwort auf diese Nachricht muss dem mittels des *ResponseIdentification*-Elements definierten Nachrichtennamen und Protokollnamen entsprechen. In diesem Fall wird also erwartet, dass auf die Anfrage-Nachricht, die Nachricht *getSensorResponse* des SCAI-Protokolls zurück gesendet wird. Das Ziel der Anfrage wird mittels des *Target*-Elements spezifiziert. In diesem Fall wird die Anfrage-Nachricht also an den Quelldienst gesendet.

```
<Condition>
    <name>ConditionPE</name>
    <ConditionType>
        <if/>
    </ConditionType>
    <ConditionPredicate>
        <PreviousError>
            <behaviorChangeName>ErrorExample</behaviorChangeName>
        </PreviousError>
    </ConditionPredicate>
</Condition>
<Map>
    ...
</Map>
```

Abbildung 5.25: Beispiel eine Bedingung (PreviousError)

In Abb. 5.25 findet sich ein Beispiel für eine *PreviousError*-Bedingung. Mittels dieser Bedingung wird geprüft, ob zuvor bei der Ausführung einer Verhaltensänderung ein Fehler aufgetreten ist. Dazu wird die jeweilige Verhaltensänderung mittels des *behaviorChangeName*-Elements referenziert. In diesem Fall wird also geprüft, ob bei der Verhaltensänderung *ErrorExample* ein Fehler aufgetreten ist.

```
<Condition>
  <ConditionType>
      <if/>
  </ConditionType>
  <ConditionPredicate>
      <MessageStored>
          <MessageIdentification>
              <messageName>createSensor</messageName>
              <protocolName>SCAI2.0</protocolName>
          </MessageIdentification>
      </MessageStored>
  </ConditionPredicate>
</Condition>
<Condition>
    <ConditionType>
        <ifNot/>
    </ConditionType>
    <ConditionPredicate>
        <PreviousCondition>
            <behaviorChangeName>SCAMPIServerOffline</behaviorChangeName>
        </PreviousCondition>
    </ConditionPredicate>
</Condition>
```

Abbildung 5.26: Beispiel eine Bedingung (PreviousError)

Einer Verhaltensänderung können beliebig viel verschiedene Bedingungen angefügt werden. In Abb. 5.26 findet sich ein Beispiel für eine Verhaltensänderung,

bei der zunächst geprüft wird, ob die Nachricht *createSensor* des SCAI-Protokolls gespeichert wurde. Ist dies der Fall, wird geprüft ob eine vorherige Bedingung *SCAMPIServerOffline* unwahr war. Ist dies auch der Fall, wird die Verhaltensänderung ausgeführt. Daraufhin können selbstverständlich weitere Verhaltensänderungen mittels weiterer *BehaviorChange*-Elemente angefügt werden.

5.4 Zusammenfassung

In diesem Kapitel wurde eine Implementierung vorgestellt, die die beschriebenen Modelle verwendet, um Protokollkonvertierungen durchzuführen. Der implementierte Konverter ist dabei unabhängig vom Format der zu konvertierenden Nachrichten. Durch entsprechende Schnittstellen können neue Anfrage-Technologien und Kommunikationstechnologien auf einfache Weise hinzugefügt werden. Darüber hinaus wurden verschiedene Anwendungbeispiele vorgestellt, die mittels des PIM und DBM modelliert wurden, und mittels des hier beschriebenen Konverters ausgeführt werden können. Dabei wurde die Übersetzung von Nachrichten beschrieben, die im Rahmen der Analyse in Kapitel 3 evaluiert wurden. Sämtliche Evaluationen aus Kapitel 6 wurden mit dieser Implementierung durchgeführt.

6 Evaluation

In diesem Kapitel werden verschiedene Aspekte des zuvor beschriebenen Ansatzes evaluiert. Dabei wird sowohl auf die Verarbeitungszeiten der Implementierung eingegangen als auch auf die Zeiten die zum Modellieren benötigt werden. Das Modell wird dabei mit anderen gängigen Ansätzen verglichen.

6.1 Evaluationsumgebung

Alle Tests wurden auf einem “Thomas Krenn 2HE Intel Single-CPU RM217” Server durchgeführt. Dieser verfügt über eine “Intel Dual Core E3110” CPU mit einem Takt von 3,00GHz. Dem System stehen 4096 MB DDR2 RAM zur Verfügung. Als Betriebssystem wird Debina Linux (Lenny) verwendet. Darüber hinaus wird das “Java(TM) SE Runtime Environment” in der Version 1.6.0_31-b05 verwendet.

6.2 Evaluation der Verarbeitungszeit

Die durchschnittliche Zeit, die benötigt wird, um ein bestimmtes Muster auszuführen, wurde bestimmt, indem das DBM auf die bereits in Kapitel 3 erwähnten Protokolle SCAI und SWE angewendet wurde. Dazu wurden verschiedene Nachrichten der Protokolle mit zufälligen Daten gefüllt und in das jeweils andere Protokoll konvertiert. Dabei wurde die Zeit gemessen, die benötigt wurde, um das jeweilige Muster auszuführen. Die Zeit, die beim Request-Muster benötigt wird, um mit dem Server zu kommunizieren, wurde dabei außer acht gelassen, da diese unabhängig vom Konzept und der Implementierung variieren kann. Abb. 6.1 zeigt die Ergebnisse der ersten Messungen. In diesem Fall wurden XQuery-Anfragen zur Datentransformation eingesetzt. Die Identifikation der Nachrichten wurde mittels XPath-Ausdrücken realisiert.

Abb. 6.1 ist zu entnehmen, dass das Remove- und Store-Muster mit unter 1 ms am schnellsten ausgeführt werden können. Das hat den Grund, dass für diese Muster keine Anfrage zur Datentransformation ausgeführt werden, sondern lediglich die entsprechende Anfrage zur Identifikation. Das Map- und das Add-Muster benötigen zwischen 2 und 3,5 ms zur Verarbeitung. Der zusätzliche Aufwand wird hier durch die Anfragen zur Datentransformation verursacht. Es wäre zu erwarten, dass das Add-Muster schneller ausgeführt wird, als das Map-Muster, da beim Map lediglich die Nachricht in ein neues Format gebracht wird, während beim Add-Muster eine neue Nachricht erzeugt wird und die Originalnachricht weitergeleitet wird. Der Effekt lässt sich allerdings durch die Tatsache erklären, dass die Verarbeitungszeit auch abhängig von der Komplexität der jeweiligen Anfrage sowie der Komplexität der Nachrichten selbst ist. Die

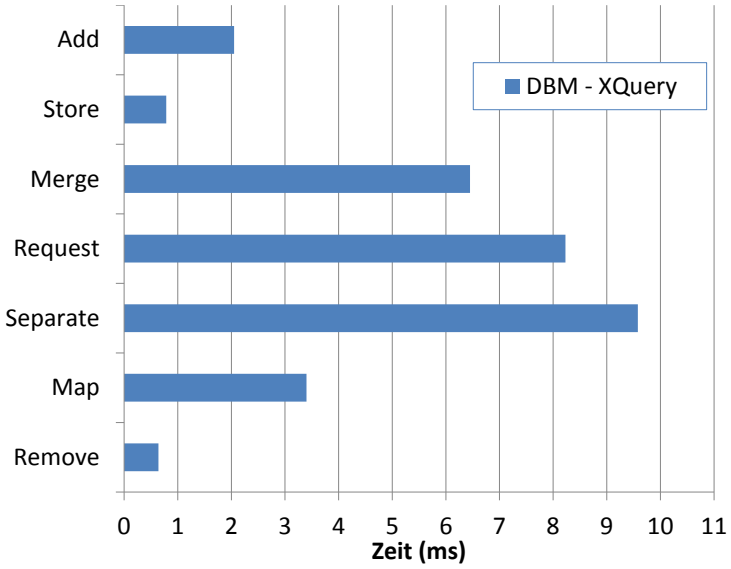


Abbildung 6.1: Verarbeitungszeit der Muster beim XQuery basierten Ansatz

relativ hohe Verarbeitungszeit von ca. 6,5ms beim Merge-Muster lässt sich durch den Umstand erklären, dass hier drei relativ komplexe Nachrichten mittels einer ebenfalls komplexen Anfrage zu einer neuen Nachricht zusammengefasst werden. Beim Request-Muster wurden aus jeder eingehenden Nachricht zwei Anfragen an einen externen Dienst generiert. Die zweite Anfrage enthielt dabei sowohl Daten aus der eingehenden Nachricht als auch aus der Antwortnachricht auf die erste Anfrage. Dies erklärt die hohe Verarbeitungszeit von ca. 8,2 ms. Beim Separate-Muster wurden drei Nachrichten aus der eingehenden Nachricht erzeugt, was bedeutet, dass drei Anfragen auf die eingehende Nachricht angewendet wurden. Dadurch ergibt sich die hohe Verarbeitungszeit von ca. 9,5 ms.

Da die meiste Zeit für das Ausführen der XQuery-Anfragen verwendet wird, wurde eine statische Implementierung als Anfrage-Sprache in das DBM integriert. Dazu wurden für jede String-Operation die entweder eine Nachricht identifiziert oder transformiert ein Anfrage spezifiziert. Diese Anfrage identifiziert lediglich die jeweilige String-Operation und muss daher nicht interpretiert werden. Zur Evaluation wurden dabei drei Optimierungsansätze gewählt. Beim ersten Optimierungsansatz wird die Nachrichtentransformation durch String-Operation gehandhabt, während die Nachrichtenidentifikation mittels XQuery-Anfragen realisiert wird. Beim zweiten Optimierungsansatz wird die Nachrichtenidentifikation durch String-Operationen realisiert während die Nachrichtentransformation mittels XQuery-Anfragen realisiert wird. Beim dritten Optimie-

rungsanssatz wird sowohl die Nachrichtenidentifikation, als auch die Transformation mittels String-Operationen realisiert.

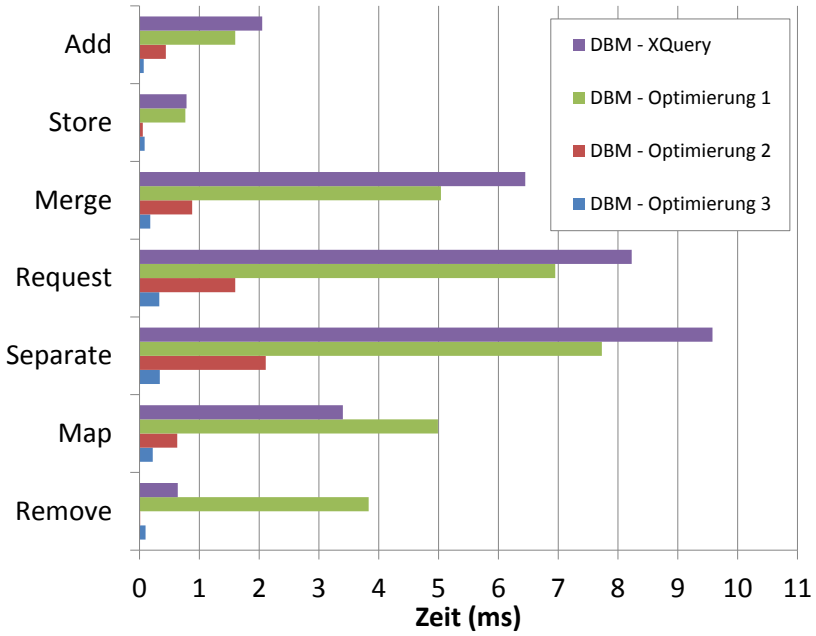


Abbildung 6.2: Verarbeitungszeit der Muster aller Ansätze

Wie in Abb. 6.2 zu sehen ist, bringt Optimierungsansatz 1 nur geringfügige Vorteile. Beim Remove- und Map-Muster ist sogar eine Verschlechterung gegenüber dem XQuery-Ansatz zu beobachten. Optimierungsansatz 2 dagegen zeigt bereits eine wesentliche Verbesserung. Bei diesem Ansatz liegt die Verarbeitungszeit immer unter 2,2 ms. Das lässt sich durch die Tatsache erklären, das bei der Identifikation einer Nachricht unter Umständen mehrere Anfragen ausgeführt werden müssen. Dies führt zu entsprechend hohen Verarbeitungszeiten. Ein einfacher Weg zur Optimierung der Verarbeitungszeiten ist also eine optimierte Implementierung der Nachrichtenidentifikation. Da die Nachrichtenidentifikation über das PIM gehandhabt wird, kann dies ohne Änderungen am DBM realisiert werden. Der dritte Optimierungsansatz senkt die Verarbeitungszeit erneut. In dieser Variante sinkt die Verarbeitungszeit unter 1 ms.

Diese Evaluation zeigt, dass das DBM beziehungsweise der DBM-Konverter in der Lage ist, die Konvertierung von Nachrichten in akzeptabler Zeit durchzuführen. Für Web-Services sind Verarbeitungszeiten von 10 ms nicht sonderlich viel, da allein die Kommunikation mit dem Web-Service selbst mehrere 100 ms in Anspruch nehmen kann. Bei der Kommunikation mit Sensoren können 10 ms bereits ein Problem sein. Mittels der beschriebenen Ansätze kann für diese Fälle die Verarbeitungszeit aber soweit gesenkt werden, dass sie bis unter 1 ms pro

Muster sinken. Somit kann das DBM in den vorgestellten Szenarien verwendet werden.

6.3 Evaluation der Status-Handhabung

Die Status-Handhabung wurde mittels eines realen Anwendungsszenarios getestet. Dabei sollte festgestellt werden, ob das DBM bei Einsatz von Bedingungen wie in Kapitel 4.7 beschrieben, innerhalb akzeptabler Zeit alle Nachrichten konvertieren kann.

Der Zieldienst ist eine SCAMPI-Middleware, die mittels des SCAI-Protokolls kommuniziert (siehe Kapitel 3). Als Quelldienst wird der Web-Service fightradar24.com verwendet. Dieser erlaubt Benutzern, Live-Informationen über die Position von Flugzeuge abzufragen. Der Dienst selbst kennt nur vier Nachrichtentypen. Mittels der Nachricht *getPositionList* kann eine Liste aller bekannten Flugzeuge und ihrer Positionen angefragt werden. Diese Anfrage wird mit der Nachricht *PositionList* beantwortet. Mittels der Nachricht *getPlaneData* können zusätzliche Informationen über ein Flugzeug abgefragt werden. Diese werden mittels der Nachricht *PlaneData* übertragen. Da der FlightRadar-Dienst nicht in der Lage ist, seine Daten direkt an einen anderen Dienst beziehungsweise an den Konverter zu senden, wurde ein Programm implementiert (Poller), das kontinuierlich die Nachricht PositionList beim Quelldienst abruft. Dieses Programm erzeugt dann aus der PositionList jeweils eine Nachricht für jedes Flugzeug und übermittelt diese an den Konverter. Im Idealfall kann dieser die Nachricht direkt in das SCAI Format übersetzen und somit ein *Measurement* an die SCAMPI-Middleware senden. Dieser Prozess wird Abb. 6.3 schematisch dargestellt.

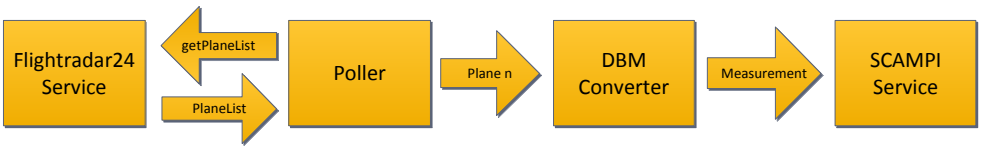


Abbildung 6.3: Schematische Darstellung des Testszenarios

Der Prozess kann allerdings nur wie oben beschrieben durchgeführt werden, wenn der entsprechende Sensor bereits bekannt ist. Andernfalls muss der Sensor zuvor bei der SCAMPI-Middleware registriert werden. Dies ist wiederum nur möglich, wenn zuvor der entsprechende Sensortyp registriert wurde. In dem beschriebenen Szenario ist das Flugzeug der Sensor und das Flugzeugmodell der Sensortyp. Ist der Sensor nicht registriert, muss mittels der Nachricht *getPlaneData* die Nachricht *PlaneData* beim Quelldienst angefragt werden. Diese kann dann verwendet werden, um zu prüfen ob der Sensortyp bereits regis-

triert wurde. Falls dies nicht der Fall ist, muss der Sensortyp und der Sensor registriert werden, bevor die Nachricht konvertiert werden kann. Diese Bedingungen können problemlos mittels des DBM abgebildet werden. Abb. 6.4 zeigt den entsprechenden Entscheidungsbaum.

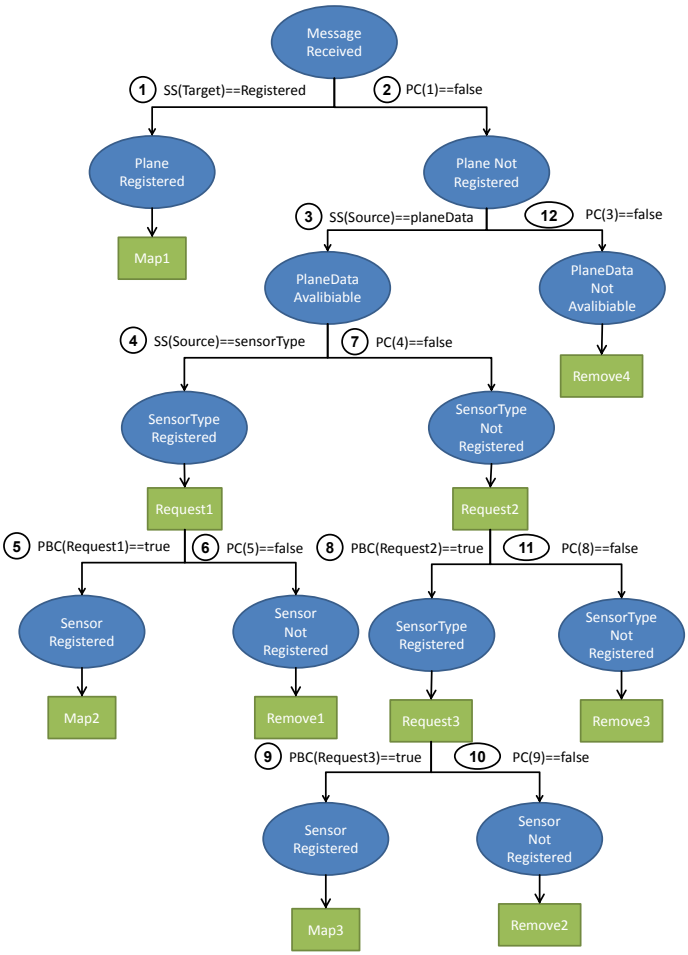


Abbildung 6.4: Entscheidungsbaum des Testszenarios

Der Ausgangszustand ist, dass eine Nachricht empfangen wurde (*Message Received*). In Schritt 1 wird geprüft, ob das Flugzeug bereits in der SCAMPI-Middleware registriert ist. Dazu wird eine ServiceState (SS)-Bedingung verwendet. Ist dies der Fall, wird die Nachricht mittels der Verhaltensänderung *Map1* transformiert und an den Zieldienst gesendet. Andernfalls wird in Schritt 2 mittels einer PreviousCondition (PC)-Bedingung festgestellt, dass Bedingung 1 fehlgeschlagen ist. Das System wäre somit in dem Zustand *PlaneNotRegistered*. In diesem Fall muss zunächst geprüft werden, ob beim Flightradar-Service

zusätzliche Daten über das Flugzeug abgerufen werden können, da diese zum Registrieren des Sensors und des Sensortypen in der SCAMPI-Middleware benötigt werden. Dies wird in Schritt 3 mittels einer ServiceState-Bedingung realisiert. Ist dies der Fall, ist das System in dem Zustand *PlaneDataAvalibiable*. Nun muss zunächst geprüft werden, ob der entsprechende Sensortyp bereits registriert ist. Dies wird in Schritt 4 mittels einer ServiceState-Bedingung geprüft. Ist dies der Fall, befindet sich das System in dem Zustand *SensorTypeRegistered*. Daher muss nur der Sensor registriert werden, was mittels der Verhaltensänderung *Request1* realisiert wird. In Schritt 5 wird geprüft, ob der Sensor tatsächlich registriert werden konnte, indem mittels einer PreviousBehaviorChange (PBC)-Bedingung geprüft wird, ob Request1 erfolgreich ausgeführt wurde. Ist dies der Fall, wird die Nachricht mittels der Verhaltensänderung *Map2* transformiert und an den Zieldienst gesendet. Falls der Sensor nicht registriert werden konnte, wird dies mittels einer PreviousCondition-Bedingung in Schritt 6 festgestellt. Das System wäre dann im Zustand *SensorNotRegistered* und die Nachricht würde mittels der Verhaltensänderung *Remove1* entfernt werden. Falls der Sensortyp nicht registriert ist, wird dies mit einer PreviousCondition-Bedingung in Schritt 7 festgestellt. Mittels der Verhaltensänderung *Request2* wird daraufhin versucht, den Sensortypen zu registrieren. Ist dies erfolgreich wird dies in Schritt 8 festgestellt. Daraufhin wird in Schritt 9 und 10 versucht, den Sensor zu registrieren. Dazu wird der gleiche Prozess wie in Schritt 5 und 6 angewendet. Falls der Sensortyp nicht registriert werden kann, wird dies in Schritt 11 festgestellt und die Nachricht entfernt. In dem Fall, dass bereits zu Beginn keine Daten über das Flugzeug abgerufen werden konnte, wird die Nachricht in Schritt 12 entfernt.

Für die Evaluation werden drei Möglichkeiten dieses Szenarios betrachtet. Im ersten Fall wurde das Flugzeug bereits registriert und die Nachricht wird mittels *Map1* übersetzt. Im zweiten Fall wurde der Sensortyp bereits registriert, der Sensor allerdings noch nicht. Somit würde die Nachricht mittels der *Map2* Verhaltensänderung übersetzt werden. Im dritten Fall sind Sensor und Sensortyp noch nicht registriert und die Verhaltensänderung *Map3* wird ausgeführt. Es wird davon ausgegangen, dass die Anfragen nie fehlschlagen. Die Verhaltensänderungen *Remove1* bis *Remove4* werden also nie ausgeführt. Die zeitaufwändigste Variante ist *Map3*, da die meisten Bedingungen geprüft werden müssen. Die schnellste Variante sollte *Map1* sein, da lediglich eine Bedingung geprüft werden muss.

Während der Evaluation werden dem Konverter 2 Minuten Zeit gegeben, um alle Nachrichten zu übersetzen. Dies sind, abhängig von der Anzahl der gesichteten Flugzeuge, zwischen 1500 und 1800 Nachrichten. Sobald die zwei Minuten vergangen sind, werden alle noch nicht konvertierten Nachrichten verworfen und mit einem aktuellen Datensatz begonnen. Zu Beginn des Tests ist die SCAMPI Datenbank leer, so dass auf jeden Fall neue Sensoren und Sensortypen

registriert werden müssen. Ziel ist es, festzustellen ob und nach welcher Zeit der Punkt erreicht wird, ab dem alle Nachrichten übersetzt werden können.

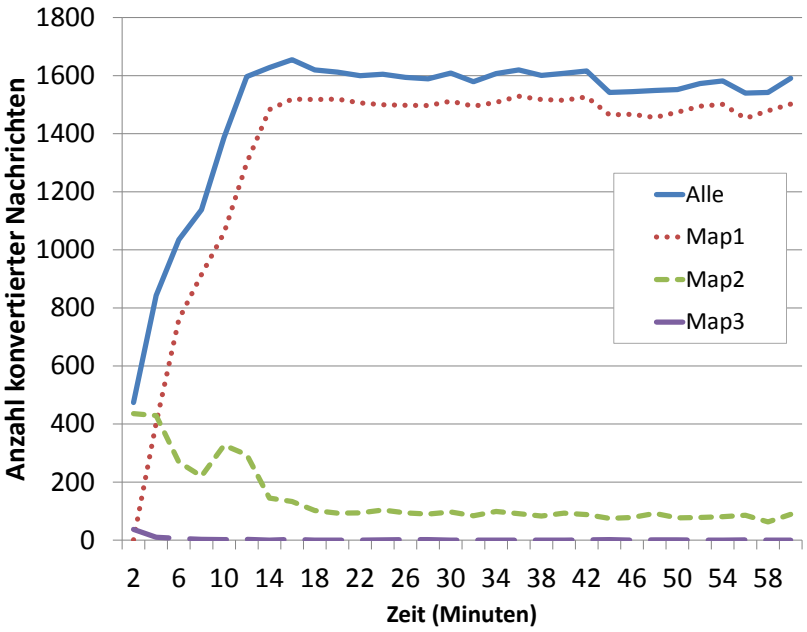


Abbildung 6.5: Ergebnisse der Evaluation der Status-Handhabung

Das Ergebnis des Tests findet sich in Abb. 6.5. Die Grafik zeigt die Anzahl von Nachrichten, die zum jeweiligen Zeitpunkt konvertiert werden konnten. Während die durchgezogene Linie alle Nachrichten darstellt, stellen die gestrichelten Linien jeweils die Nachrichten dar, die mittels der Verhaltensänderungen Map1, Map2 und Map3 konvertiert wurden. Zu Beginn des Tests konnten nur ca. 450 Nachrichten übersetzt werden. Dies lässt sich durch die Tatsache erklären, dass die SCAMPI Datenbank zu diesem Zeitpunkt leer war und entsprechend viele Sensoren registriert werden mussten. Innerhalb der ersten 10 Minuten steigt die Anzahl der konvertierten Nachrichten konstant an. Nach ca. 12 Minuten ist der Punkt erreicht, ab dem alle Nachrichten konvertiert werden können. Die Anzahl der zu registrierenden Sensoren liegt nach ca 15. Minuten relativ konstant bei ungefähr 100. Die Anzahl der zu registrierenden Sensortypen liegt bereits 5 Minuten nach dem Start des Test bei unter 10. Dieser Test wurde zu verschieden Uhrzeiten, mit einer maximalen Länge von bis zu 5 Stunden durchgeführt. Dabei wurde kein signifikant anderes Verhalten festgestellt. Der Test zeigt also, dass eine Konvertierung aller Nachrichten in dem beschriebenen realen Szenario, trotz der Anwendung komplexer Bedingungen, nach einer Startzeit von wenigen Minuten problemlos möglich ist.

6.4 Aufwand im Vergleich zu anderen Modellierungsverfahren

Der Aufwand, den die einzelnen Modellierungsansätze verursachen, lässt sich nur schwer gegeneinander abschätzen. Dies hat den Grund, dass der Modellierungsaufwand abhängig von der Komplexität der Protokolle, den übertragenen Daten, von der Anzahl der beteiligten Protokolle und von der Art der Protokolle sein kann. Da bestimmte Ansätze weniger oder mehr Aufwand unter bestimmten Voraussetzungen bedeuten, kann ein Vergleich der Aufwände für verschiedene Modellierungsverfahren nur argumentativ stattfinden. Daher werden in diesem Kapitel zunächst die Aufwände der Ansätze aus Kapitel 2 mit dem Modellierungsaufwand des DBM verglichen. Da das DBM mit dem Fokus entwickelt wurde, Service-Protokolle möglichst einfach zu konvertieren, konzentriert sich der Vergleich auf folgende Punkte:

- Komplexität der Schnittstellenbeschreibung
- Aufwand, um Änderungen des Kommunikationsverhaltens zu beschreiben
- Aufwand zur Beschreibung der Datentransformation
- Wartbarkeit

Da ein beliebter Ansatz die statische Implementierung von Konvertern durch Adapter und Wrapper ist, wird im Kapitel 6.5 eine Studie vorgestellt, in der der Modllierungsaufwand des DBM mit dem einer konkreten Implementierung verglichen wird.

6.4.1 Middleware-Systeme

Neben den klassischen Modellierungsverfahren gibt es Middleware-Systeme, die bereits Methoden und Schnittstellen zur Verfügung stellen, um die Konvertierung von Protokollen zu beschreiben. Von diesen Ansätzen kommt das Protocol Conversion Framework [GMR+10] dem in dieser Arbeit vorgestellten Ansatz am nächsten. Dabei handelt es sich um ein Framework zur Übersetzung von Protokollen, mit dem Ziel Sensoren in die Web-Service Welt zu integrieren. Während die Übersetzung des Kommunikationsverhaltens auf einem Satz von Regeln beruht, muss die Übersetzung des Kommunikationsverhaltens implementiert werden.

Da die Übersetzung der Protokolle in diesen Systemen auf statischen Implementierungen beruht, führen Änderungen der Protokolle zu einem entsprechend hohen Aufwand bei der Wartung. Die Identifikation der Protokolle und Nachrichten beruht ebenfalls auf einer statischen Implementierung, so dass auf eine komplexe Schnittstellenbeschreibung verzichtet werden kann. Änderungen der

Protokolle können aber auch hier zu aufwändigen Änderungen in der Implementierung führen. Der Aufwand zum Programmieren der Transformation der Daten und des Kommunikationsverhaltens ist abhängig von der Komplexität der Protokolle.

6.4.2 Anfrage-Sprachen

Anfrage-Sprachen sind relativ einfach zu schreiben und zu verstehen. Gerade XQuery [BCF+07] erlaubt dem Entwickler, mit relativ einfachen Möglichkeiten Daten aus komplexen Nachrichten zu extrahieren und diese in ein neues Format zu bringen. Allerdings können Änderungen im Kommunikationsverhalten eines Protokolls nicht zufriedenstellend abgebildet werden, so dass zusätzliche Implementierungen benötigt werden.

Auch wenn die Transformation der Daten relativ simpel beschrieben werden kann, kommt es bei komplexen Protokollen zu einer entsprechend schlechten Wartbarkeit, da hier im Zweifelsfall der spezifische Code und die Anfrage angepasst werden müssen. Da die Verhaltensänderungen programmiert werden müssen, kann dies bei entsprechend komplexen Protokollen zu einem hohen Aufwand führen.

6.4.3 Model Matching

Bei diesem Ansatz muss der Entwickler zunächst das Quell- und Ziel-Protokoll mittels eines entsprechenden Models beschreiben [YS97, Oku90, AM91]. Für die Transformation der Protokolle müssen komplexe Schnittstellenbeschreibungen der Daten vorliegen. Diese müssen gegebenenfalls noch um spezifischen Code erweitert werden. Aus den Beschreibungen der beiden Modelle kann dann ein endlicher Automat abgeleitet werden, der den Konverter der beiden Protokolle repräsentiert.

Die Komplexität der zu definierenden Modelle ist stark abhängig von der Komplexität der jeweiligen Protokolle und dem jeweiligen Modellierungsverfahren. Während die Definition von Modellen für relativ simple Protokolle einfach zu handhaben ist, können die Beschreibungen sehr schnell komplex werden, sobald ein gewisses Maß an Abhängigkeiten zwischen den Nachrichten der Protokolle auftritt. Auch Ausnahmen, wie beispielsweise das Verhalten im Fehlerfall, kann sehr schnell zu sehr komplexen Modellen führen. Die Wartbarkeit dieser Systeme ist somit ebenfalls relativ komplex, da im Zweifelsfall nicht nur das Modell angepasst werden muss, sondern auch die Schnittstellenbeschreibungen. Da die Datentransformation immer für das gesamte Protokoll beschrieben werden muss, kann auch dies zu erheblichen Aufwand führen.

6.4.4 Interface Matching

Beim Interface Matching muss der Entwickler zunächst eine Beschreibung der Schnittstellen für jedes beteiligte Protokoll definieren [YS97, NBM+07]. Diese Beschreibungen können, abhängig von der Komplexität der Konvertierung, von simplen Identifizierungen der Nachrichten und ihrer Elemente bis hin zu regulären Ausdrücken, die eine Sprache beschreiben, reichen. Im nächsten Schritt muss der Entwickler festlegen, wie die Inhalte eines Protokolls auf die Nachrichten des Ziel-Protokolls zugeordnet werden. Die Zuordnung wird üblicherweise über eine modellspezifische Sprache gesteuert. Aus der Beschreibung der Schnittstellen und der Beschreibung der Zuordnung kann im nächsten Schritt automatisiert ein endlicher Zustandsautomat generiert werden, insofern dieser existiert. Dieser ist in der Lage, das Kommunikationsverhalten der über die Schnittstellenbeschreibungen definierten Protokolle zu konvertieren. Die Datentransformation kann gegebenenfalls bereits aus den Schnittstellenbeschreibungen generiert werden, bedarf im Zweifelsfall aber zusätzlicher Implementierungen.

Der Vorteil dieses Ansatzes ist, dass die Modellierung des Kommunikationsverhaltens nur indirekt über die Zuordnung der Daten aus den einzelnen Nachrichten geschieht. Dies bietet dem Entwickler eine wesentlich einfachere Möglichkeit, Daten eines Protokolls auf Nachrichten eines Anderen abzubilden. Die Schnittstellenbeschreibungen sowie die Beschreibung der Zuordnung kann allerdings sehr schnell sehr komplex werden, gerade wenn mehrere Protokolle verwendet werden. Die Wartbarkeit dieser Systeme kann durch die komplexen Beschreibungen sehr aufwändig sein.

6.4.5 Message Sequence Charts

Bei diesem Ansatz wird zunächst das Ziel- und Quell-Protokoll beschrieben. In diesem Fall beschreibt der Entwickler das Kommunikationsverhalten mittels Message Sequence Charts (MSC) [RTTZ04]. Dies sind Ablaufdiagramme, die das Kommunikationsverhalten des jeweiligen Protokolls beschreiben. In einem automatisierten Prozess werden dann Daten innerhalb der einzelnen Nachrichten identifiziert und ein endlicher Zustandsautomat erzeugt, der den Konverter repräsentiert. Zur Transformation der Daten ist eine entsprechende Schnittstellenbeschreibung notwendig.

Die Spezifikation der Verhaltensänderungen sollte von einem Entwickler mit akzeptablen Aufwand möglich sein, da Sequenz-Diagramme häufig bei der Entwicklung von Protokollen verwendet werden. Bei Änderungen des Kommunikationsverhaltens müsste lediglich das entsprechende Diagramm angepasst werden. Beim automatisierten Erzeugen des Konverters kann es zu Interpretationsfehlern kommen. Wird beispielsweise das gleiche Attribut in zwei Nachrich-

ten übertragen, die in unterschiedlichen Kontexten verwendet werden können, kann nicht zwangsläufig das richtige Verhalten erzeugt werden. In diesem Fall müsste der Entwickler die entsprechende Zuordnung selber durchführen. Das führt zu einer entsprechend durchschnittlichen Wartbarkeit.

6.4.6 Triple Graph Grammatiken

Bei diesem Ansatz muss der Entwickler zunächst das Kommunikationsverhalten des Quell- und Ziel-Protokolls mittels eines Graphen abbilden [Sch94]. Darauf aufbauend wird ein weiterer Graph definiert, der die Änderungen zwischen den beiden vorherigen Graphen definiert. Dies hat unter anderem den Vorteil, dass Umkehrschlüsse automatisch gezogen werden können.

Es kann davon ausgegangen werden, dass die Beschreibung der Protokolle mittels Graphen verhältnismäßig einfach ist, da lediglich das Kommunikationsverhalten des jeweiligen Protokolls abgebildet werden muss. Allerdings müssen Änderungen des Kommunikationsverhaltens konkret beschrieben werden. Die Beschreibung der Unterschiede kann komplexer werden, hängt aber stark von der Komplexität der Protokolle ab. Die Datentransformation geschieht unabhängig auf Grundlage von Schnittstellenbeschreibungen und kann somit, abhängig vom Protokoll, relativ komplex werden. Änderungen am Protokoll können relativ komplexe Änderungen am Modell nach sich ziehen, da diese unter Umständen in die Beschreibung der Unterschiede aufgenommen werden müssen.

6.4.7 Differential Behavior Model

Beim DBM muss der Entwickler die Protokolle zunächst mittels des PIM beschreiben. Im Vergleich zu den vorherigen Ansätzen bedeutet dies nur einen sehr geringen Aufwand, da hier lediglich Nachrichten und Protokolle über wiederkehrende Muster identifiziert werden. Da die Identifikation mittels Anfragen realisiert wurde, ist dies auch von unerfahrenen Entwicklern problemlos zu realisieren. Änderungen des Kommunikationsverhaltens können mit einem simplen Satz von Mustern abgebildet werden. Diese wurde bereits mit dem Ziel entwickelt, dass sie für Entwickler einfach zu verstehen sind. Sollten Änderungen des Kommunikationsverhaltens nur unter bestimmten Bedingungen nötig sein, kann dies mit einem Entscheidungsbaum abgebildet werden. Dadurch können auch komplexe Verhaltensänderungen mit wenig Aufwand beschrieben werden. Sollten Änderungen an einem Protokoll auftreten, muss das entsprechende DBM angepasst werden. Wurden Änderungen an der Datenrepräsentation durchgeführt, muss die entsprechende Anfrage angepasst werden. Bei Änderungen des Kommunikationsverhaltens muss überprüft werden, ob die entsprechenden Muster noch das gewünschte Verhalten erzeugen. Abhängig von der

Komplexität der Protokolle und der Änderungen kann dies auch hier komplexe Änderungen nach sich ziehen. Da die Beschreibung der Verhaltensänderungen auf Muster beruht, die relativ einfach zu verstehen sind, ist allerdings davon auszugehen, dass die Änderungen mit weniger Aufwand durchgeführt werden könnte als beispielsweise die Anpassung einer abstrakten Graphenstruktur. Ein entsprechendes Modellierungswerkzeug, dass den Entwickler bei der Modellierung unterstützt, kann den Wartungsaufwand weiter senken.

6.4.8 Vergleich der Aufwände

In den vorherigen Abschnitten wurden die Aufwände für verschiedene Ansätze zur Modellierung von Protokollkonvertern vorgestellt. Die Grafik in Abb. 6.6 stellt diese einander gegenüber.

	Wartung	**Schnittstellen Beschreibung**	**Aufwand für Verhaltens-änderungen**	**Aufwand zur Daten-transformation**
Model Matching	Mittel bis Komplex	Komplex	Komplex	Komplex
Interface Matching	Mittel bis Komplex	Komplex	Komplex	Mittel bis Komplex
Message Sequence Charts	Mittel	Komplex	Mittel	Komplex
Tripel Graph Grammatiken	Mittel	Komplex	Niedrig	Komplex
Middleware-Systeme	Mittel	Einfach	Mittel bis Komplex	Mittel
Anfrage Sprachen	Mittel bis Komplex	Einfach	Mittel bis Komplex	Mittel
Differential Behavior Model	Einfach	Einfach	Niedrig	Mittel

Abbildung 6.6: Vergleich der verschiedenen Konvertierungsansätze

Die Modelle der klassischen Ansätze können nur relativ aufwändig gewartet werden. Bei den Tripel Graph Grammatiken und den Message Sequence Charts wird der Aufwand durch die einfachere Beschreibungsweise reduziert. Unter der Voraussetzung, dass ein unterstützendes Modellierungswerkzeug existiert, kann das DBM relativ einfach gewartet werden. Bedingt durch die Tatsache, dass die klassischen Ansätze eine entsprechend komplexe Schnittstellenbeschreibung voraussetzen um von den eigentlichen Nachrichten auf die Daten des Modells zu kommen, ist der Aufwand bei diesen Ansätzen entsprechend hoch. Die Middleware-Systeme benötigen, da sie nur bestimmte Protokolle unterstützen,

nur eine sehr einfache Schnittstellenbeschreibung. Das PIM des DBM wurde bereits mit dem Fokus entwickelt, eine einfache Identifikation von Protokollen und Nachrichten zu ermöglichen. Somit ist die Schnittstellenbeschreibung entsprechend simpel. Der Aufwand zum Beschreiben von Verhaltensänderungen ist bei den klassischen Ansätzen relativ hoch. Dies hat den Grund, dass üblicherweise versucht wird, das Protokoll möglichst vollständig zu beschreiben. Bedingt durch die Tatsache, dass die Verhaltensänderungen nur in eine Richtung beschrieben werden müssen, ist der Aufwand bei den Tripel Graph Grammatiken entsprechend niedriger. Da das DBM mit dem Fokus entwickelt wurde, Verhaltensänderungen auf einfache Weise zu beschreiben, ist der Aufwand hier ebenfalls entsprechend niedrig. Der Aufwand zur Datentransformation ist bei den klassischen Ansätzen hoch, da diese voraussetzen, dass das gesamte Protokoll übersetzt wird. Bei den verbleibenden Ansätzen muss lediglich eine Übersetzung für die nötigen Fälle beschrieben werden. Daher ist der Aufwand hier entsprechend niedriger.

6.5 Aufwand im Vergleich zu manueller Implementierung

In diesem Kapitel wird der Modellierungsaufwand des DBM mit dem einer Implementierung verglichen. Dazu wurde im Rahmen einer Studie der Aufwand für jeden dieser Ansätze gemessen. Teilnehmer waren Personen mit allgemeiner Programmiererfahrung in der Programmiersprache Java. Dies waren sowohl Informatik-Studenten, ausgebildete Fachinformatiker wie auch Hochschulabsolventen. Die Teilnehmer wurden in zwei Gruppen, Modellierer und Programmierer, eingeteilt. Die Teilnehmerzahl dieser Studie betrug 18 Personen, jeweils 9 in jeder Gruppe. Die folgenden Abschnitte beschreiben die Studie, sowie deren Ergebnisse im Detail.

6.5.1 Ablauf

Die Zuteilung der Teilnehmer geschah mittels eines Einteilungsfragebogen. Beim Ausfüllen des Einteilungsfragebogen mussten die Teilnehmer angeben, wie gut sie sich selbst in der Programmiersprache Java, sowie der Modellierungssprache XML einschätzen. Dies sind die Basistechnologien, die zum Lösen der Aufgaben benötigt werden. Entsprechend der Ergebnisse des Einteilungsfragebogen, wurden die Teilnehmer gleich auf die beiden Gruppen verteilt. Ziel bei der Verteilung war es somit, gleich viele erfahrene und unerfahrene Teilnehmer in jeder Gruppe zu haben.

Der Prüfungsteil der Studie wurde, unabhängig von der jeweiligen Gruppe, in drei Phasen unterteilt. Zunächst wurde eine Einarbeitung durchgeführt, um dem Teilnehmer ein Grundverständins für die Problemstellung zu schaffen. Dar-

aufhin wurden die ersten, relativ simplen Aufgaben bearbeitet. Danach wurden komplexere Aufgaben, die auf die vorherigen Aufgaben aufbauen, bearbeitet. Im Folgenden werden diese drei Phasen genauer beschrieben:

Phase 1 - Einarbeitung: In dieser Phase soll sich der Teilnehmer mit dem grundlegendem Konzept sowie der Entwicklungsumgebung vertraut machen. Dazu wird von ihm eine Dokumentation des jeweiligen Ansatzes durchgearbeitet. Außerdem hat er bereits die Möglichkeit, sich mit der Entwicklungsumgebung vertraut zu machen. Im Anschluss werden etwaige Fragen mit dem Betreuer geklärt. Daraufhin wird eine Einführungsaufgabe bearbeitet. Diese dient noch zur Einarbeitung und ist noch nicht Teil der Evaluation. Unklarheiten können daher auch während der Bearbeitung mit dem Betreuer diskutiert werden. Am Ende der Einarbeitung sollte der Teilnehmer das grundlegende Problem verstanden haben und in der Lage sein, die Aufgaben aus den folgenden Phasen selbstständig zu lösen.

Phase 2 - Simple Aufgaben: In dieser Phase müssen die Teilnehmer konkrete Probleme mit dem jeweiligem Ansatz lösen. Dabei werden die Unterschiede zwischen zwei fiktiven Protokollen beschrieben. Die Aufgaben orientieren sich dabei an realen Problemen, die in Kapitel 3 beschrieben wurden. Der Test selbst ist in drei Teilaufgaben unterteilt. Jede Teilaufgabe kann mittels eines Musters aus dem DBM, beziehungsweise mit einer Kombination von Mustern gelöst werden. Die Programmierer können die Aufgabe durch simple Vergleichsoperationen und Methodenaufrufe lösen.

Phase 3 - Komplexe Aufgaben: Diese Aufgabe baut auf die der vorherigen Phase auf. Es werden die selben fiktiven Protokolle behandelt, allerdings wird davon ausgegangen, dass sich das Ziel-Protokoll geändert hat. Somit muss jede der vorherigen Lösungen angepasst werden. Die DBM-Gruppe kann die Aufgaben durch Hinzufügen zusätzlicher Muster und Bedingungen lösen, während die Programmierer ihren vorherigen Code anpassen müssen.

Beim Bearbeiten der einzelnen Aufgaben wurde die Zeit gemessen, die zum Lösen der Teilaufgabe benötigt wurde. Nach jeder Aufgabe durfte eine Pause eingelegt werden, die nicht in die Gesamtzeit einfließt. Das Lesen und Verstehen der Aufgabe wurde ebenfalls nicht miteinbezogen.

Im Anschluss an diese Phasen wurde mit dem jeweiligen Teilnehmer eine kurze Evaluation durchgeführt. Zunächst wurde ein Usability-Test nach John Brooke [Joh96] durchgeführt, der Aufschluss über die Benutzbarkeit des jeweiligen Ansatzes geben soll. Dazu wurde dem Teilnehmer 10 Aussagen vorgelegt, die er mit 1 bis 5 bewerten musste. Eine ausführliche Beschreibung dieses Tests findet sich in der entsprechenden Auswertung.

Danach wurde ein kurzes Interview durchgeführt, das die allgemeinen Vor- und Nachteile des jeweiligen Ansatzes herausstellen soll. Dabei wurden dem

Teilnehmer drei Fragen gestellt. Diese Fragen dienen dazu, die Aufgaben beziehungsweise Funktionen des jeweiligen Ansatzes zu identifizieren, die dem Teilnehmer die meisten Schwierigkeiten bereitet haben. Dadurch sollen sowohl Fehler im Testverfahren wie auch im Modellierungsverfahren selbst gefunden werden.

6.5.2 Rahmenbedingungen

Um die beiden Ansätze miteinander vergleichen zu können, wurden für alle Teilnehmer die gleichen Rahmenbedingungen definiert. Daher wurde der Test mit jedem Teilnehmer am gleichen Rechner und im gleichen Raum durchgeführt. Von allen Teilnehmern, unabhängig von ihrer Gruppe, wurden die gleichen Aufgaben bearbeitet. Ebenfalls wurde bei der Hilfestellung darauf geachtet, dass keine Gruppe, beziehungsweise kein Teilnehmer, bevorzugt wird. Zudem wurde die Entwicklungsumgebung fest vorgegeben. Beide Gruppen mussten die Aufgaben ohne Testfälle lösen.

Die einzigen Unterschiede zwischen den Gruppen waren somit der Ansatz mit dem das Problem gelöst wurde, die Entwicklungsumgebung und das einführende Dokument aus Phase 1. Dieses Dokument unterschied sich zwischen den Gruppen allerdings nur in den Punkten, in denen auf die Entwicklungsumgebung eingegangen wird. Es herrschten somit die gleichen Bedingungen für alle Teilnehmer, unabhängig von ihrer jeweiligen Gruppe.

Die Modellierer lösten die Aufgaben mittels des DBM. Dazu mussten sie ein XML-Dokument erzeugen, dass die Änderungen zwischen den Protokollen, entsprechend der Beschreibung aus Kapitel 4, entspricht. Als Entwicklungsumgebung stand ihnen der grafische XML Editor Serna zur Verfügung. Mittels eines XML-Schemas wurde dabei der Rahmen des DBM vorgegeben, so dass sich der Aufwand auf die reine Modellierung des Problems reduziert.

Die Programmierer lösten die Aufgaben durch eine konkrete Implementierung mittels der Programmiersprache Java. Als Entwicklungsumgebung stand ihnen NetBeans zur Verfügung. Außerdem erhielten die Entwickler eine Klassenstruktur, die bereits alle benötigten Klassen zur Kommunikation, Nachrichtenidentifikation und Datentransformation zur Verfügung stellte. Der Aufwand reduzierte sich somit auf die reine Implementierung der Übersetzung des Kommunikationsverhaltens. Somit war die Lösung der Aufgaben durch simple Vergleichsoperationen und Mehtodenaufrufe möglich.

Da die Komplexität der Datentransformation abhängig vom jeweiligen Anwendungsfall und der verwendeten Programmiersprache ist wird diese im Test nicht beachtet. Daher wurden Anfragen vorgegeben, die dann in beiden Ansätzen verwendet werden können.

6.5.3 Auswertung

Bei der Auswertung der Studie wurden verschiedene Aspekte evaluiert. Dieser Abschnitt enthält die entsprechenden Ergebnisse dieser Auswertungen.

6.5.3.1 Bearbeitungszeit

Um festzustellen, ob das DBM eine ähnliche Bearbeitungszeit verspricht wie die einer klassischen Implementierung wurde beim Lösen der Aufgaben die Bearbeitungszeit für jede Teilaufgabe gemessen. Die Zeit, die zum Lesen und Verstehen der Aufgabe benötigt wurde, wurde dabei nicht miteinbezogen. Im Rahmen der Auswertung wurde die Gesamtzeit, die jeweils zum Lösen der Teilaufgaben aus Phase 2 und Phase 3 benötigt wurde für jeden Teilnehmer berechnet. Daraufhin wurde für jede Gruppe die durchschnittliche Bearbeitungszeit innerhalb der jeweiligen Phasen berechnet. Die Ergebnisse finden sich in Abb. 6.7.

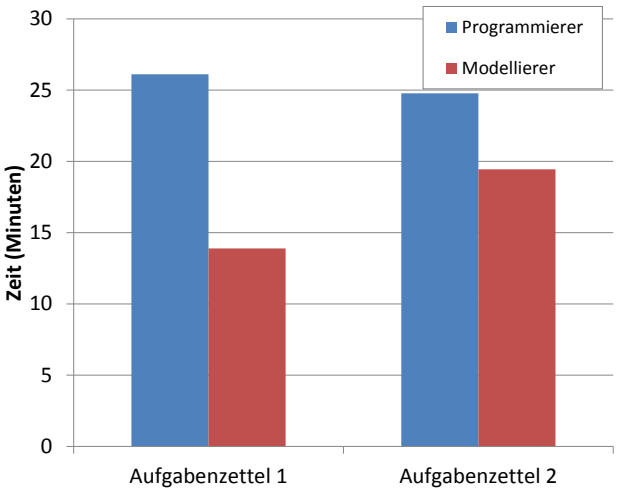


Abbildung 6.7: Bearbeitungszeiten

Das Diagramm zeigt, dass die Programmierer für alle Teilaufgaben in Aufgabenzettel 1 (Phase 2) durchschnittlich ca. 26 Minuten gebraucht haben, während die Modellierer mit ca. 14 Minuten wesentlich schneller waren. Für Aufgabenzettel 2 (Phase 3) benötigten die Programmierer ca. 24 Minuten, um alle Aufgaben zu lösen, während die Modellierer ca. 19 Minuten benötigten. Das Ergebnis zeigt also, dass das DBM eine geringere Einarbeitungszeit bietet, wenn es um die Bearbeitung simpler Ausgaben geht. Darüber hinaus zeigt das Ergebnis, dass der Aufwand beim Modellierern steigt, sobald vorhandene Modelle angepasst werden. Bei den Programmierern bleibt der Aufwand dagegen relativ konstant.

Trotz des gestiegenen Aufwands in Phase 3 lag die Gesamtbearbeitungszeit aber noch immer unter der einer Implementierung.

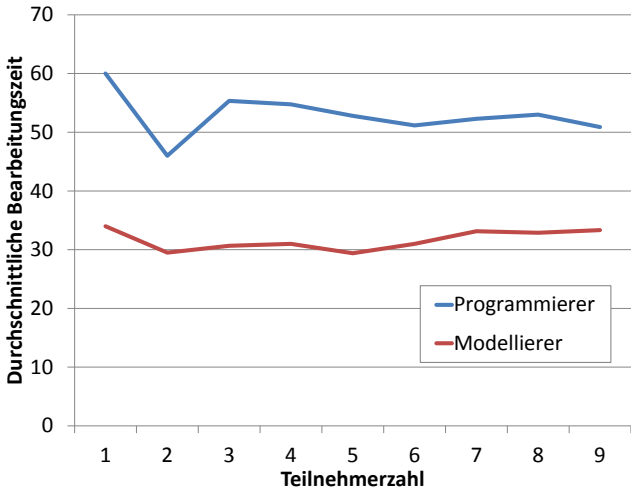


Abbildung 6.8: Durchschnittliche Bearbeitungszeiten

Um festzustellen, ob die Testgruppe groß genug ist um verlässliche Ergebnisse zu produzieren, wurde außerdem die durchschnittliche Bearbeitungszeit über eine ansteigende Anzahl von Teilnehmern berechnet. Das Ergebnis dazu findet sich in Abb. 6.8. Die Grafik zweigt, das bereits nach ca. 5 bis 6 Teilnehmer keine großartigen Änderungen in der durchschnittlichen Bearbeitungszeit mehr auftreten.

k	$P(X \leq k)$
0	0
1	0
2	0
3	0.0001
4	0.0014
5	0.0117
6	0.0672
7	0.2599
8	0.6496
9	1

Tabelle 6.1: Kumuliale Binomialverteilung für n = 9 und p = 0,89

Um festzustellen, ob dieses Ergebnis auch bei größeren Testgruppen zu erwarten ist, wurde ein Hypothesentest [Rin08] durchgeführt. Dafür wurde die Hypothese: *„90% der Modellierer können Aufgabenzettel 1 in weniger als 15 Minuten*

lösen“ aufgestellt. Daraus ergibt sich folgende Nullhypothese und alternative Hypothese:
$H_0 : p > 0.89$
$H_1 : p \leq 0.89$

Der Annahme- und Ablehnungsbereich sehen somit folgendermaßen aus:
$A : \{8, 9\}$
$\overline{A} : \{0, 1, ..., 7\}$

Mittels der Tabelle 6.1 kann die kumulierte Binomialverteilung bestimmt werden. Daraus ergibt sich mittels folgender Formel ein Signifikanzniveau von 74%.
$P(X > 7) = 1 - (X \leq 7) = 1 - 0,2599 = 0,7401$

Das heißt, unter der Annahme, dass 90% der Teilnehmer schneller als 15 Minuten sind, kommt es bei der angegebenen Befragung mit einer Wahrscheinlichkeit von ca. 74% zu einem solchen Ergebnis und damit zur Annahme Nullhypothese. Geht man von einem Fehler von 5% aus, kann mittels folgender Formel und Tabelle 6.1 der entsprechende Annahme- und Ablehnungsbereich bestimmt werden:
$P(X > k) \geq 0,95$
$P(X \leq k - 1) < 0,05$
$k = 5 - 1 = 4$

$A : \{5, 9\}$
$\overline{A} : \{0, 1, ..., 4\}$

Das heißt, die Hypothese kann angenommen werden, wenn in der Evaluation mindestens 5 Modellierer den ersten Aufagbenzettel in weniger als 15 Minuten lösen konnten. Da dies von 6 Modellierern erreicht wurde, ist die Hypothese somit wahr. Da für die Programmierer die gleichen Rahmenbedingungen verwendet wurden, kann die Hypothesenanalyse auch auf ihre Ergebnisse angewendet werden. Das heißt, die Hypothese *„90% der Programmier können Aufgabenzettel 1 nicht innerhalb von 20 Minuten lösen“* ist wahr, wenn mindestens 5 Programmierer diese Bedingung in der Evaluation erfüllt haben. Auch dies war in der Studie der Fall.

6.5.3.2 Usability

Wie bereits erwähnt, wurde im Rahmen der Evaluation ein Usability-Test nach John Brook durchgeführt. Dieser gibt Aufschluss über die Bedienbarkeit eines Systems. Der Test wurde im Anschluss an Phase 3 durchgeführt. Bei diesem Test muss der Teilnehmer 10 Aussagen mit Punkten von 1 bis 5 bewerten. In der Auswertung ergibt dies Punkte zwischen 0 und 100, wobei 100 das beste Ergebnis ist. Während der Erklärung des Tests wurden die Teilnehmer darauf aufmerksam gemacht, dass nicht die jeweilige Entwicklungsumgebung bewertet werden soll,

sondern der Ansatz selbst. Das durchschnittliche Ergebnis der beiden Gruppen findet sich in Abb. 6.9.

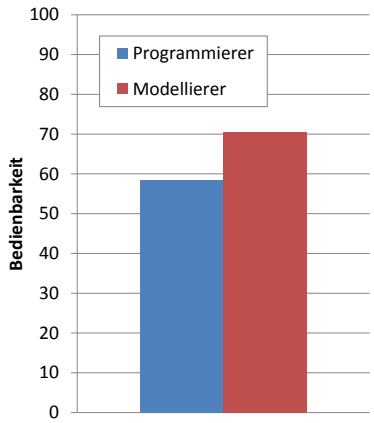


Abbildung 6.9: Bedienbarkeit

Während die Programmierer den Implementierungsansatz durchschnittlich mit ca. 58 bewertet haben, haben die Modellierer ihren Ansatz mit durchschnittlich 70 bewertet. Analog zur Evaluation der Bearbeitungszeit wurde auch hier der Durchschnittswert über eine ansteigende Anzahl von Teilnehmern berechnet, um festzustellen ob die Teilnehmerzahl repräsentativ ist. Wie in Abb. 6.10 zu erkennen, ändert sich der Durchschnittswert ebenfalls nach ca. 5 bis 6 Teilnehmern nicht mehr wesentlich. Somit verspricht das DBM eine vergleichbare bis höhere Bedienbarkeit als der Implementierungsansatz. Es gilt zu beachten, dass die Teilnehmer natürlich unter Umständen subjektiv eine Bewertung der Entwicklungsumgebung mit haben einfließen lassen. Im Interview wurde allerdings festgestellt, dass der grafische XML-Editor als unzureichend eingestuft wurde. Sollte also die Entwicklungsumgebung subjektiv in das Ergebnis eingeflossen sein, hat dies das Ergebnis höchstens zum Schlechteren verfälscht.

6.5.3.3 Fehlerquote

Neben der Bearbeitungszeit und Bedienbarkeit des DBM wurde ihm Rahmen der Benutzerstudie auch die Fehlerquote der Teilnehmer analysiert. Da weder den Modellierern noch den Programmierern Testfälle zur Verfügung standen, sind diese Ergebnisse nicht auf ein reales Szenario übertragbar. Allerdings können die Ergebnisse als Maßstab für die Komplexität des jeweiligen Ansatzes betrachtet werden. Außerdem können die Ergebnisse verwendet werden, um festzustellen ob in beiden Gruppen gleiche Rahmenbedingungen herrschten. Zur Analyse der Fehlerquote wurden alle Lösungen der Teilnehmer auf Korrektheit geprüft, das heißt, es wurde überprüft, ob die Lösung zu einer korrekten Übersetzung der

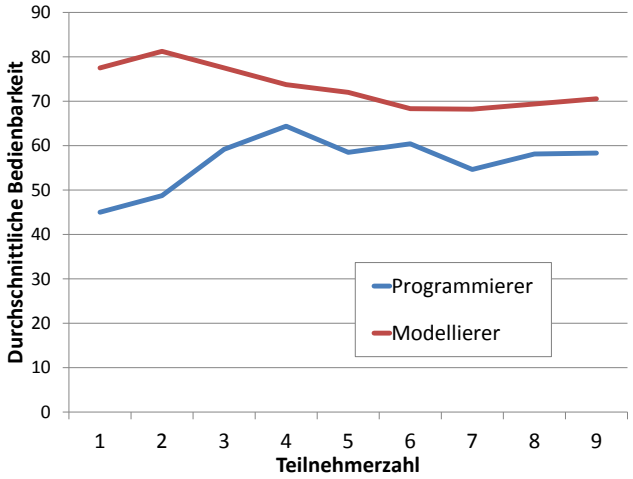


Abbildung 6.10: Durchschnittliche Bedienbarkeit

Protokolle führt. Außerdem wurde die Anzahl der Fehler innerhalb der Lösungen gezählt. Fehler, die von Aufgabenzettel 1 in Aufagbenzettel 2 übernommen wurden, wurden dabei nicht doppelt gezählt.

In Abb. 6.11 findet sich die durchschnittliche Zahl der gelösten Aufgaben. Die Gruppen haben zwischen 2,5 und 2,7 Aufgaben vollständig richtig gelöst. Die Anzahl der gemachten Fehler findet sich in Abb. 6.12. Die Fehlerquote beider Gruppen liegt zwischen 4,5 und 5,5 Fehlern. Die sehr ähnlichen Ergebnisse bestätigten, dass beide Gruppen unter den gleichen Rahmenbedingungen gearbeitet haben und die Ansätze eine vergleichbare Komplexität besitzen.

6.5.3.4 Interviews

Im Rahmen der Interviews wurden den Teilnehmern die folgenden drei Fragen gestellt:

- Welche Aufgabe fanden Sie warum am schwierigsten?
- Haben Sie Verbesserungsvorschläge für das System?
- Welche Probleme würden Sie bei einer sehr komplexen Aufgabe erwarten?

Die erste Frage wurde gestellt um die komplexeste Aufgabe zu identifizieren. In beiden Gruppen wurde üblicherweise eine der Aufgaben aus Aufgabenzettel 2 genannt. Dies hatte zumeist den Grund, dass die zuvor gelöste Aufgabe angepasst werden musste. Die zweite Frage wurde gestellt, um allgemeine Schwä-

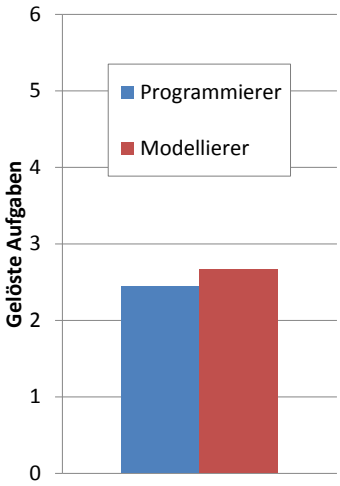


Abbildung 6.11: Gelöste Aufgaben

chen des jeweiligen Ansatzes zu finden. Während die Programmierer allgemeine Verbesserungen der umgebenden Klassenstruktur vorschlugen, wurde von den Modellierern häufig ein komplexeres Modllierungswerkzeug vorgeschlagen, das den Modellierungsprozess besser unterstützt. Die letzte Frage wurde gestellt, um mögliche Schwächen des jeweiligen Ansatzes, in Bezug auf eine weitaus komplexere Aufgabe, zu finden. Hier wurden aus beiden Gruppen ähnliche Antworten gegeben. Zum größten Teil wurde hier auf komplexe Sequenzen von Nachrichten sowie die allgemeine Wartbarkeit des jeweiligen Ansatzes eingegangen.

6.5.3.5 Fazit

Die Benutzerstudie hat gezeigt, dass die Bearbeitungszeit zur Konvertierungen von Protokollen mittels des DBM unter der einer konkreten Implementierung liegt. Zwar erhöht sich der Aufwand, wenn vorhandene Modelle angepasst werden müssen, der Gesamtaufwand liegt aber unter dem einer Implementierung. Der Usability-Test bestätigt, dass die Bedienbarkeit besser ist, als der einer Implementierung. Die Analyse der Fehlerquoten bestätigt, das die Komplexität des Ansatzes mit dem einer Implementierung vergleichbar ist. Die Interviews legen den Schluss nahe, dass durch ein unterstützendes Modellierungswerkzeug der Aufwand weiter reduziert werden kann.

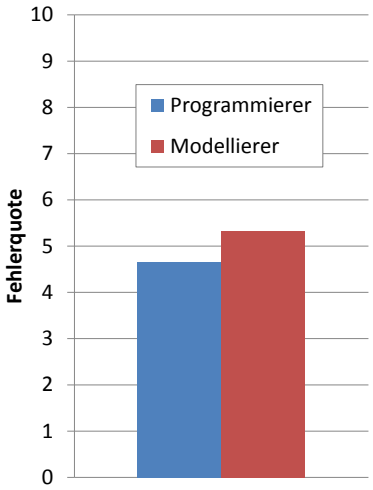


Abbildung 6.12: Fehlerquote

6.6 Anwendung in anderen Domänen

In diesem Abschnitt wird die Verwendbarkeit des vorgestellten Ansatz auf Protokolle einer anderen Domäne als der der Service-Protokolle getestet. Für die Evaluation wurden das Extensible Messaging and Presence Protocol (XMPP) [SA11, STSA09] sowie das Internet Relay Chat (IRC) Protokoll [OR93] ausgewählt. Bei beiden Protokollen handelt es sich um Instant Messaging Protokolle. Ihr Anwendungszweck ist die Übertragung von Textnachrichten zwischen zwei oder mehr Teilnehmern. Die Protokolle sind also keine Service-Protokolle, da sie keine Informationen oder Funktionen über ihre Schnittstellen anbieten. Vielmehr handelt es sich um Kommunikationsprotokolle.

Die erste Version des IRC-Protokolls wurde bereits 1988 entwickelt und verwendet eine textbasierte Formatierung, um Daten zu beschreiben. Dabei setzt das Protokoll einen Server voraus, der die Verbindungen aller teilnehmender Clients verwaltet. Die Kommunikation findet ausschließlich zwischen Server und Client statt. Um an einer Kommunikation teilzunehmen, muss der Client sich zunächst beim Server authentifizieren. Dabei muss, abhängig von der Konfiguration des Servers, nicht zwangsläufig ein Passwort angegeben werden. Ist ein Client mit einem Server verbunden, muss dieser einen *Channel* wählen. Alle Teilnehmer eines Channels sind in der Lage, Nachrichten miteinander auszutauschen. Darüber hinaus können Nachrichten direkt von einem Teilnehmer an einen Anderen gesendet werden. Ist ein Client mit einem Server verbunden, sendet der Server zyklisch eine *Ping*-Nachricht an den Client. Diese enthält eine ID, die vom Client mittels einer *Pong*-Nachricht zurück gesendet werden muss.

Geschieht dies nicht innerhalb einer definierten Zeit, wird die Verbindung mit dem Client unterbrochen.

XMPP ist aus dem Jabber-Protokoll entstanden und wurde in der ersten Version 1998 entwickelt. Nachrichten werden mittels XML formatiert. Ähnlich wie beim IRC-Protokoll, basiert es auf einer Client/Server Architektur. Ein Client muss sich zunächst mit Benutzernamen und Passwort beim Server anmelden. Danach ist der Client in der Lage, Nachrichten an andere Teilnehmer zu senden. Dazu muss er deren ID kennen. Im Gegensatz zum IRC-Protokoll verfügt das XMPP-Protokoll über komplexe Verfahren zur Verschlüsselung, Authentifikation und zum Handhaben von Sitzungen. Zur Kommunikation wird ein *Long HTTP-Request* verwendet. Das heißt, die gesamte Kommunikation zwischen Server und Client wird in einem einzigen XML-Dokument übertragen, das im Rahmen des Kommunikationsprzesses stetig erweitert wird. Erst bei Abschluss der Kommunikation wird das Ende des Dokuments übertragen.

6.6.1 Übersetzen den Authentifizierung

Rein theoretisch könnten die Authentifizierungsnachrichten der Protokolle in das jeweils andere übersetzt werden, da sie ähnliche Daten übertragen. Die Authentifizierung findet in beiden Protokollen mittels Benutzername und Passwort statt, allerdings gibt es grundlegende Unterschiede in beiden Protokollen. Zum einen kann eine Authentifizierung im IRC-Protokoll ohne Passwort stattfinden, XMPP erlaubt dieses aber nicht. Eine Konvertierung wäre in diesem Fall also nicht möglich. Außerdem werden in XMPP die Authentifizierungsdaten standardmäßig verschlüsselt. Eine Konvertierung wäre also nur möglich, wenn die Daten während der Konvertierung entsprechend entschlüsselt werden. Abhängig vom Verschlüsselungsverfahren wären hierfür wiederum zusätzliche Informationen, wie die Schlüssel die zur Verschlüsselung verwendet wurden, nötig. Hinzu kommt, dass die Verfahren zum Handhaben von Sitzungen sich grundsätzlich voneinander unterscheiden. Während IRC Ping-Nachrichten einsetzt, wird die Sitzung in XMPP mittels HTTP-Request gehandhabt. Sollte ein Konverter die Nachrichten übersetzen, müsste dieser das Handhaben der Sitzungen für beide Protokolle entsprechend implementieren.

6.6.2 Übersetzen der Nachrichten

Auch wenn das Übersetzen des Authentifizierungsprozesses nur mit komplexen Mitteln realisiert werden kann, können XMPP-Nachrichten in das IRC-Protokoll übersetzt werden. Eine Übersetzung in die andere Richtung funktioniert allerdings nicht. Dies hat den Grund, dass eine XMPP-Nachricht (siehe Abb. 6.13) mehr Informationen benötigt als in einer IRC-Nachricht übertragen werden.

```
<stream:stream
    to='xmpp.com'
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'
    version='1.0'>
    <message from='xmppuser@xmpp.com' to='ircuser@irc' xml:lang='en'>
        <body>Hello! I am not a Bot! I am a Converter!</body>
    </message>
</stream:stream>
```

Abbildung 6.13: XMPP Beispiel-Nachricht

Das wichtigste Element ist dabei die Absenderidentifikation, die in der XMPP-Nachricht mittels des *from*-Attributes im *Message*-Element übertragen wird. Eine IRC-Nachricht enthält diese Information nicht. Darüber hinaus erlaubt der IRC-Server nicht, diese Information mittels einer Nachricht festzustellen.

```
USER xmppuser - - -
NICK xmppuser
JOIN #irc
PRIVMSG ircuser :Hello! I am not a Bot! I am a Converter!
```

Abbildung 6.14: IRC Beispiel-Nachricht

Wie bereits erwähnt, kann eine XMPP-Nachricht allerdings in eine Nachricht des IRC-Protokolls übersetzt werden. Die Absenderidentifikation kann zum IRC-User konvertiert werden. Das *to*-Attribut kann verwendet werden um, den Empfänger der Nachricht, sowie den Channel des IRC-Protokolls zu identifizieren. Die Nachricht selbst kann dem *body*-Element entnommen werden. So könnte nicht nur die Nachricht selbst übersetzt werden, sondern im gleichen Schritt die Nachrichten zum Verbinden mit dem IRC-Server, zum Authentifizieren und zum Beitreten des Channels erzeugt werden. Da das Verbinden mit dem IRC Server nur zu Beginn der Kommunikation geschehen muss, muss mittels einer Bedingung geprüft werden, ob bereits eine Verbindung zum Server besteht. Dazu kann der *PING*-Befehl des IRC-Protokolls verwendet werden. Das übersetzte Kommunikationsverhalten des XMPP-Beispiels findet sich in Abb. 6.14.

6.6.3 Zusammenfassung

Der Test hat gezeigt, dass eine grundlegende Übersetzung auch in anderen Domänen, als der der Service-Protokolle möglich ist. Da vom DBM aber das Handhaben von Sitzungen und das Entschlüsseln von Daten nicht unterstützt wird, kann eine Konvertierung nur für bestimmte Anwendungsfälle durchge-

führt werden. Für die komplexeren Fälle ist eine Konvertierung zwar möglich, allerdings würde dies eine entsprechend komplexe Implementierung in Komponenten des Konverters, beziehungsweise in der Applikation, die den Konverter implementiert, voraussetzen. Bei einer Änderung des Protokolls müssten diese Implementierungen gegebenenfalls entsprechend angepasst werden, wodurch der Vorteil des DBM verloren gehen würde. Das gleiche gilt für das Verhalten im Fehlerfall. Dieses wurde in der Analyse nur relativ oberflächlich betrachtet. Würde aber ein Fehler auftreten, zieht dies unter Umständen nicht nur eine entsprechende Fehlernachricht nach sich, sondern beeinflusst auch die Sitzung selbst. Um ein solches Verhalten mit DBM abzubilden, müssten wiederum entsprechend komplexe Bedingungen definiert werden.

6.7 Zusammenfassung

In diesem Kapitel wurde das DBM evaluiert. Im ersten Abschnitt wurde die Verarbeitungszeit der einzelnen Muster gemessen. Dabei wurde festgestellt, dass für die beschriebenen Szenarien die Verarbeitung innerhalb akzeptabler Zeit durchgeführt werden kann. Im nächsten Abschnitt wurde evaluiert, ob die Muster innerhalb eines realen Szenarios ausgeführt werden können, wenn gleichzeitig Bedingungen angewendet werden. In dem evaluierten Szenario konnte das Ziel bereits nach wenigen Minuten erreicht werden. Daraufhin wurde der Ansatz, in Bezug auf den Modellierungsaufwand argumentativ gegen andere Ansätze abgegrenzt. Das Ergebnis zeigt, dass unter den definierten Rahmenbedingungen das DBM am vielversprechendsten ist. Darüber hinaus wurde der Modellierungswaufwand mit dem einer Implementierung verglichen. Dazu wurde eine Benutzerstudie durchgeführt. Die Ergebnisse dieser Studie zeigen, dass das DBM eine geringere Einarbeitungszeit und Bearbeitungszeit bietet. Die Benutzbarkeit des DBM ist ebenfalls höher, als die einer klassischen Implementierung. Abschließend wurde der Ansatz auf Protokolle einer anderen Domäne angewendet. Die Ergebnisse zeigen, dass eine Konvertierung grundsätzlich möglich ist, auch wenn eine vollständige Konvertierung nur mit relativ komplexen Mittel realisiert werden kann.

7 Zusammenfassung und Ausblick

Dieses Kapitel fasst die geleistete Arbeit zusammen, beschreibt, welchen Einschränkungen sie unterliegt und gibt eine Ausblick auf mögliche Erweiterungen.

7.1 Zusammenfassung

In dieser Arbeit wurde ein Modell entwickelt, dass die Beschreibung von Unterschieden zwischen Service-Protokollen erlaubt. Wie in Kapitel 2 beschrieben, existieren zwar verschiedene Ansätze, die die Beschreibung von Protokollunterschieden erlauben, allerdings haben diese häufig einen hohen Beschreibungs- und Wartungsaufwand. Darüber hinaus abstrahieren andere Verfahren die Beschreibungen so sehr, dass bereits der Initialaufwand zum Erlernen der Verfahren sehr hoch erscheint. Die Alternative zu Modellierungsverfahren sind statische Implementierungen. Diese haben ebenfalls einen hohen Wartungsaufwand. In der Arbeit werden verschiedene dieser Verfahren analysiert und die Unterschiede zum vorgestellten Ansatz klar dargestellt. Der Hauptunterschied ist dabei, dass das DBM sich auf die Beschreibung der Unterschiede zwischen Protokollen beschränkt.

Der vorgestellte Ansatz ist aus einer Analyse der Unterschiede zwischen verschiedenen Service-Protokollen sowie verschiedenen Versionen von Service-Protokollen entstanden. Diese Analyse wird in Kapitel 3 ausgeführt. Dabei wurden die Unterschiede zwischen dem SCAI-Protokoll in zwei Versionen, dem SCAI-Protokoll und dem SWE-Framework, dem SOAP-Framework in zwei Versionen und dem SCAI-Protokoll und dem Flightradar-Protokoll untersucht. Aufbauend auf diesen Unterschieden wurden Konvertierungsmuster entworfen, die eine Übersetzung zwischen den Protokollen beziehungsweise Protokollversionen ermöglichen.

Diese Muster werden in Kapitel 4 zu einem Modell zusammengefasst. Dabei wird im Detail beschrieben, wie die einzelnen Muster parametrisiert und miteinander kombiniert werden können. Darüber hinaus wird die Handhabung von zeitlichen Einschränkungen beim Senden von Nachrichten beschrieben. Außerdem enthält das Modell Funktionen, die die Ausführung eines Musters an Bedingungen knüpfen können. Dadurch kann die Konvertierung einer Nachricht beispielsweise vom internen Zustand des Ziel- oder Quelldienstes abhängig gemacht werden. Da die Muster und somit auch das Modell aus einer Analyse bestimmter Protokolle entstanden ist, wird in diesem Kapitel auch beschrieben, unter welchen Voraussetzungen das Model als vollständig anzusehen ist.

Die Implementierung eines generischen Konverters, der das DBM interpretieren und somit eine Konvertierung der beschriebenen Protokolle durchführen

kann, wird in Kapitel 5 beschrieben. Dabei wird sowohl das allgemeine Konzept, die Architektur sowie der Konvertierungsprozess selbst beschrieben.

In Kapitel 6 werden verschiedene Evaluationen des Ansatzes durchgeführt. Dafür wurde die soeben erwähnte Implementierung eingesetzt. Zunächst wurden die Ausführungszeiten der einzelnen Muster gemessen. Für die beschriebenen Szenarien ergab diese zufriedenstellende Resultate. Darüber hinaus wurde der Modellierungsaufwand, der benötigt wird, um Protokollunterschiede mittels des DBM zu beschreiben, mit dem anderer Modellierungsverfahren verglichen. Dabei stellte sich heraus, dass das DBM, im Gegensatz zu den anderen Ansätzen, mit relativ niedrigem Aufwand eingesetzt werden kann. Außerdem wurde der Initialaufwand zum Erlernen des Ansatzes mit dem Durchführen eine konkreten Implementierung verglichen. Dazu wurde eine Benutzerstudie durchgeführt. Das DBM schnitt dabei in allen Auswertungen besser als die Implementierung ab.

Außerdem wurde im Rahmen einer Evaluation versucht, das DBM in einer anderen Domäne als der der Service-Protokolle anzuwenden. Dazu wurde eine Übersetzung für die Instant Messaging Protokolle IRC und XMPP durchgeführt. Eine Übersetzung war dabei nur für bestimmte Nachrichten möglich. Problematisch war die Handhabung von Sitzungen, sowie Konvertierung von verschlüsselten Daten. Dabei ist eine Konvertierung theoretisch möglich, allerdings müssten dafür dem generischen Konverter statische Implementierungen hinzugefügt werden. Das würde wiederum dem Konzept des DBM widersprechen.

7.2 Ausblick

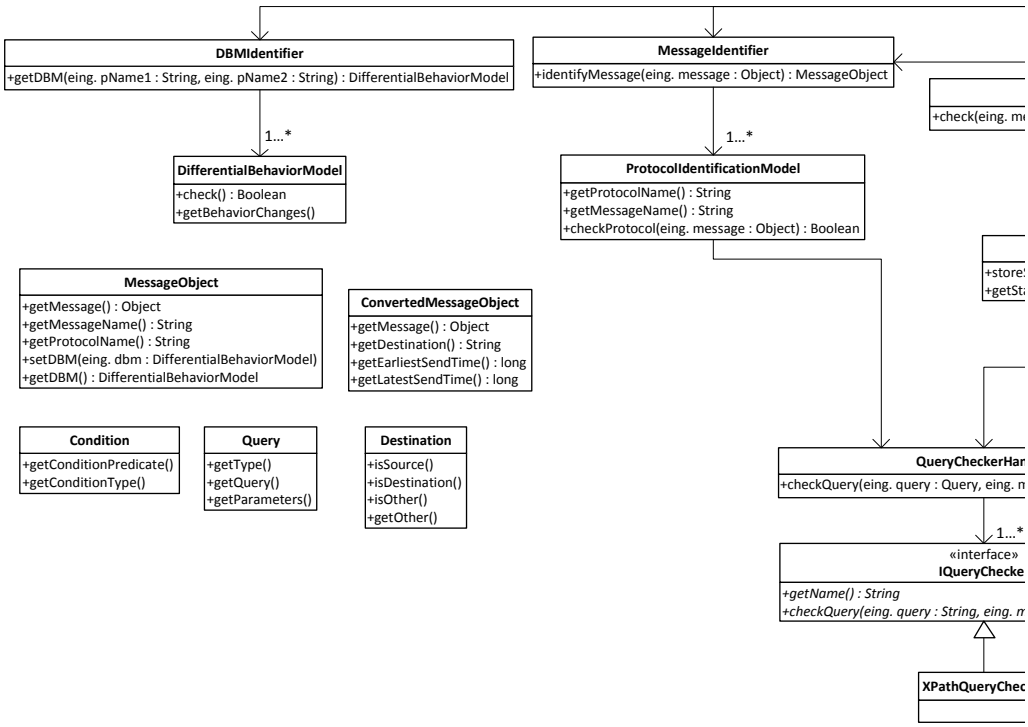
Der vorgestellte Ansatz löst die in der Motivation beschriebenen Problemstellungen. Mittels des DBM können mit wenig Aufwand für den Entwickler Service-Protokolle konvertiert werden. Dies wird durch die Evaluation bestätigt. Der Ansatz könnte jedoch erweitert werden, um die Konvertierung verschlüsselter Daten zu ermöglichen. Dazu müssten Beschreibungen der Schlüssel, die zum ent- und verschlüsseln benötigt werden, hinzugefügt werden. Auch könnten Mechanismen zum Anfragen dieser Schlüssel eingeführt werden.

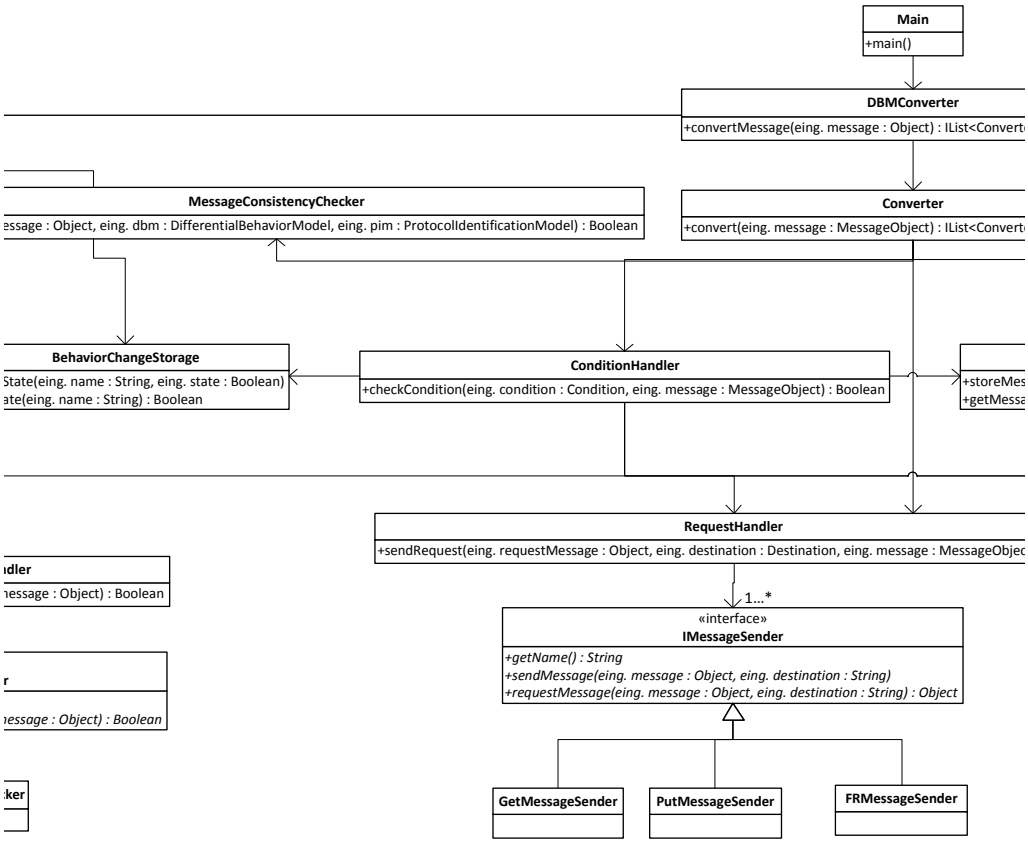
Darüber hinaus könnte das Konzept um einen Mechanismus zum Handhaben von Sitzungen erweitert werden. Diese wurde für die Service-Protokolle zwar nicht benötigt, allerdings könnten mit einem solchen Mechanismus weitere Klassen von Protokollen übersetzt werden. Dazu müssten Standardverfahren zum Handhaben von Sitzungen definiert werden. Diese können den Beschreibungen der Protokolle hinzugefügt werden. Durch eine entsprechende Erweiterung der Implementierung könnten so mit relativ simplen Mitteln verschiedene Arten von Sitzungen gehandhabt werden und somit eine vereinfachte Übersetzung der Protokolle ermöglicht werden.

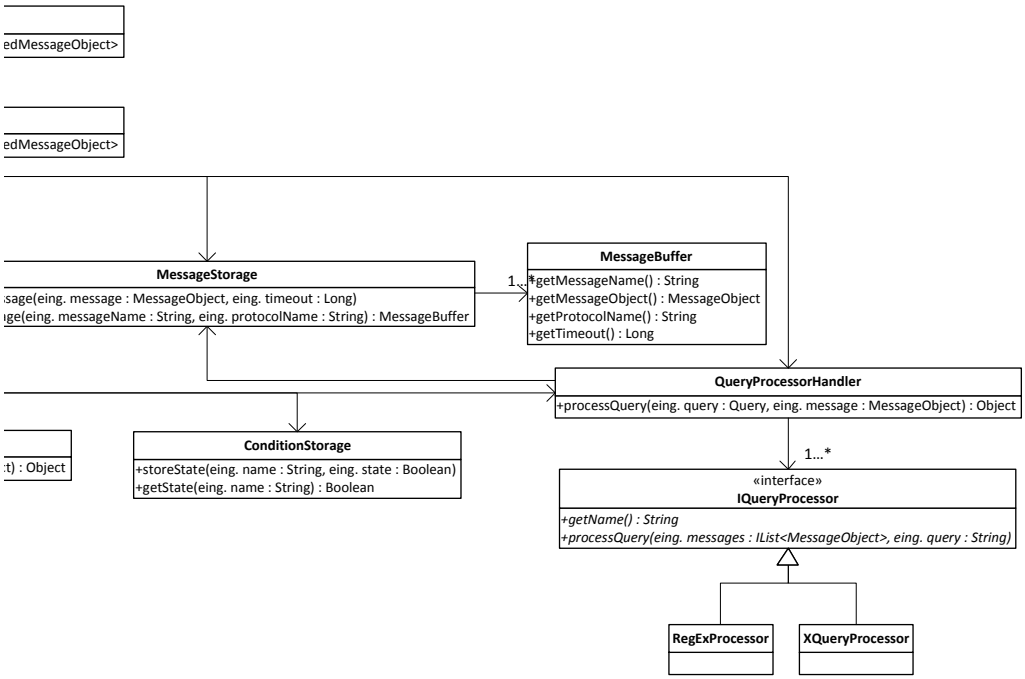
Die Benutzerstudie zeigt zwar, dass die Benutzbarkeit des DBMs besser ist als die einer Implementierung, den Interviews ist allerdings zu entnehmen, dass es noch Möglichkeiten zur Optimierung gibt. Diese könnte durch die Entwicklung eines grafischen Modellierungswerkzeugs realisiert werden, das den momentan verwendeten XML-Editor ablöst. Dieses Werkzeug könnte nicht nur die Auswirkungen der Muster und Bedingungen klarer darstellen, sondern auch den Entwickler beim Beschreiben von Protokollunterschieden unterstützen. Dies könnte beispielsweise durch das automatische Vorschlagen von häufig verwendeten Kombinationen von Mustern geschehen.

A Anhang

A.1 Vollständiges Klassendiagramm




Main
+main()
DBMConverter
+convertMessage(eing. message : Object) : IList<Convert
MessageConsistencyChecker
essage : Object, eing. dbm : DifferentialBehaviorModel, eing. pim : ProtocolIdentificationModel) : Boolean
Converter
+convert(eing. message : MessageObject) : IList<Convert
BehaviorChangeStorage
State(eing. name : String, eing. state : Boolean)
ate(eing. name : String) : Boolean
ConditionHandler
+checkCondition(eing. condition : Condition, eing. message : MessageObject) : Boolean
+storeMes
+getMessa
RequestHandler
+sendRequest(eing. requestMessage : Object, eing. destination : Destination, eing. message : MessageObjec
dler
essage : Object) : Boolean
1...*
«interface»
IMessageSender
+getName() : String
+sendMessage(eing. message : Object, eing. destination : String)
+requestMessage(eing. message : Object, eing. destination : String) : Object
r
essage : Object) : Boolean
ker
GetMessageSender
PutMessageSender
FRMessageSender


edMessageObject>
edMessageObject>
MessageStorage
sage(eing. message : MessageObject, eing. timeout : Long)
ge(eing. messageName : String, eing. protocolName : String) : MessageBuffer
1
MessageBuffer
*getMessageName() : String
+getMessageObject() : MessageObject
+getProtocolName() : String
+getTimeout() : Long
QueryProcessorHandler
+processQuery(eing. query : Query, eing. message : MessageObject) : Object
t) : Object
ConditionStorage
+storeState(eing. name : String, eing. state : Boolean)
+getState(eing. name : String) : Boolean
1...*
«interface»
IQueryProcessor
+getName() : String
+processQuery(eing. messages : IList<MessageObject>, eing. query : String)
RegExProcessor
XQueryProcessor

A.2 DBM XSD-Schema

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4              targetNamespace="http://cbusemann.de/schema/diffModel"
5              elementFormDefault="qualified"
6              xmlns:tns="http://cbusemann.de/schema/diffModel">
7
8   <xsd:element name="DiffModel">
9    <xsd:complexType>
10    <xsd:sequence>
11     <xsd:element name="protocolName1" type="xsd:string"/>
12     <xsd:element name="protocolName2" type="xsd:string"/>
13     <xsd:element name="BehaviorChange" minOccurs="1" maxOccurs="unbounded">
14      <xsd:complexType>
15       <xsd:sequence>
16        <xsd:element name="name" minOccurs="0" type="xsd:string"/>
17        <xsd:element name="MessageIdentification" maxOccurs="unbounded"
18                     type="tns:MessageIdentificationDescription"></xsd:element>
19        <xsd:element name="ErrorHandling" minOccurs="0"
20                     type="tns:ErrorDescription"></xsd:element>
21        <xsd:element name="Condition" minOccurs="0"
22                     type="tns:ConditionDescription" maxOccurs="unbounded"/>
23        <xsd:element name="EndAfterChange" minOccurs="0"/>
24        <xsd:choice>
25         <xsd:element name="Add">
26          <xsd:complexType>
27           <xsd:sequence>
28            <xsd:element name="addedMessageName" type="xsd:string"/>
29            <xsd:element name="addedMessageProtocolName" type="xsd:string"/>
30            <xsd:element name="Query" type="tns:QueryDescription"/>
31            <xsd:element name="Destination" minOccurs="1"
32                         type="tns:DestinationDescription"></xsd:element>
33            <xsd:element name="Timing" minOccurs="0"
34                         type="tns:TimingDescription"/>
35           </xsd:sequence>
36          </xsd:complexType>
37         </xsd:element>
38         <xsd:element name="Map">
39          <xsd:complexType>
40           <xsd:sequence>
41            <xsd:element name="mappedMessageName" type="xsd:string"/>
42            <xsd:element name="Query" type="tns:QueryDescription"/>
43            <xsd:element name="Timing" type="tns:TimingDescription"
44                         minOccurs="0"/>
45           </xsd:sequence>
46          </xsd:complexType>
47         </xsd:element>
48         <xsd:element name="Merge">
49          <xsd:complexType>
50           <xsd:sequence>
51            <xsd:element name="mergedMessageName" type="xsd:string"/>
52            <xsd:element name="Query" type="tns:QueryDescription"/>
53            <xsd:element name="Timing" minOccurs="0"
54                         type="tns:TimingDescription"/>
55           </xsd:sequence>
56          </xsd:complexType>
57         </xsd:element>
58         <xsd:element name="Remove">
59          <xsd:complexType></xsd:complexType>
```

```
60          </xsd:element>
61          <xsd:element name="Request">
62           <xsd:complexType>
63            <xsd:sequence>
64             <xsd:element name="RequestMsg" maxOccurs="unbounded"
65                          type="tns:RequestDescription"></xsd:element>
66            </xsd:sequence>
67           </xsd:complexType>
68          </xsd:element>
69          <xsd:element name="Seperate">
70           <xsd:complexType>
71            <xsd:sequence>
72             <xsd:element name="SeperatedMsg" minOccurs="2" maxOccurs="unbounded">
73              <xsd:complexType>
74               <xsd:sequence>
75                <xsd:element name="seperatedMessageName" type="xsd:string"/>
76                <xsd:element name="Query" type="tns:QueryDescription"/>
77                <xsd:element name="Timing" minOccurs="0"
78                             type="tns:TimingDescription"/>
79               </xsd:sequence>
80              </xsd:complexType>
81             </xsd:element>
82            </xsd:sequence>
83           </xsd:complexType>
84          </xsd:element>
85          <xsd:element name="Store">
86           <xsd:complexType>
87            <xsd:sequence>
88             <xsd:element name="timeout" type="xsd:long" minOccurs="1"/>
89            </xsd:sequence>
90           </xsd:complexType>
91          </xsd:element>
92         </xsd:choice>
93        </xsd:sequence>
94       </xsd:complexType>
95      </xsd:element>
96     </xsd:sequence>
97    </xsd:complexType>
98   </xsd:element>
99   <xsd:complexType name="QueryDescription">
100   <xsd:sequence>
101    <xsd:element name="queryType" type="xsd:string" minOccurs="1"/>
102    <xsd:element name="query" minOccurs="1" maxOccurs="1" type="xsd:string"/>
103    <xsd:element name="Property" type="tns:PropertyDescription" minOccurs="0"
104                 maxOccurs="unbounded"/>
105   </xsd:sequence>
106  </xsd:complexType>
107  <xsd:complexType name="TimingDescription">
108   <xsd:choice>
109    <xsd:element name="Delay">
110     <xsd:complexType>
111      <xsd:sequence>
112       <xsd:element name="min" type="xsd:int"/>
113       <xsd:element name="max" type="xsd:int"/>
114      </xsd:sequence>
115     </xsd:complexType>
116    </xsd:element>
117    <xsd:element name="TimeOfDay" maxOccurs="unbounded">
118     <xsd:complexType>
119      <xsd:sequence>
120       <xsd:element name="time" type="xsd:time"/>
```

```
121       <xsd:element name="windowSize" type="xsd:int"/>
122      </xsd:sequence>
123     </xsd:complexType>
124    </xsd:element>
125    <xsd:element name="Generic">
126     <xsd:complexType>
127      <xsd:sequence>
128       <xsd:element name="type" type="xsd:string"/>
129       <xsd:element name="Property" type="tns:PropertyDescription"
130                    maxOccurs="unbounded" minOccurs="0"/>
131      </xsd:sequence>
132     </xsd:complexType>
133    </xsd:element>
134   </xsd:choice>
135  </xsd:complexType>
136  <xsd:complexType name="PropertyDescription">
137   <xsd:sequence>
138    <xsd:element name="key" type="xsd:string"/>
139    <xsd:element name="value" type="xsd:string"/>
140   </xsd:sequence>
141  </xsd:complexType>
142  <xsd:complexType name="ConditionDescription">
143   <xsd:sequence>
144    <xsd:element name="name" minOccurs="0" type="xsd:string"/>
145    <xsd:element name="ConditionType">
146     <xsd:complexType>
147      <xsd:choice>
148       <xsd:element name="if"/>
149       <xsd:element name="ifNot"/>
150      </xsd:choice>
151     </xsd:complexType>
152    </xsd:element>
153    <xsd:element name="ConditionPredicate">
154     <xsd:complexType>
155      <xsd:choice>
156       <xsd:element name="MessageStored">
157        <xsd:complexType>
158         <xsd:sequence>
159          <xsd:element name="MessageIdentification"
160                       type="tns:MessageIdentificationDescription"/>
161         </xsd:sequence>
162        </xsd:complexType>
163       </xsd:element>
164       <xsd:element name="ServiceState">
165        <xsd:complexType>
166         <xsd:sequence>
167          <xsd:element name="Request" type="tns:RequestDescription" />
168         </xsd:sequence>
169        </xsd:complexType>
170       </xsd:element>
171       <xsd:element name="PreviousCondition">
172        <xsd:complexType>
173         <xsd:sequence>
174          <xsd:element name="condtionName" type="xsd:string"/>
175         </xsd:sequence>
176        </xsd:complexType>
177       </xsd:element>
178       <xsd:element name="PreviousBehaviorChange">
179        <xsd:complexType>
180         <xsd:sequence>
181          <xsd:element name="behaviorChangeName" type="xsd:string"/>
```

```
182         </xsd:sequence>
183        </xsd:complexType>
184       </xsd:element>
185       <xsd:element name="PreviousError">
186        <xsd:complexType>
187         <xsd:sequence>
188          <xsd:element name="behaviorChangeName" type="xsd:string"/>
189         </xsd:sequence>
190        </xsd:complexType>
191       </xsd:element>
192      </xsd:choice>
193     </xsd:complexType>
194    </xsd:element>
195   </xsd:sequence>
196  </xsd:complexType>
197  <xsd:complexType name="RequestDescription">
198   <xsd:sequence>
199    <xsd:element name="Query" type="tns:QueryDescription"/>
200    <xsd:element name="Timing" minOccurs="0" type="tns:TimingDescription"/>
201    <xsd:element name="ResponseIdentification"
202                 type="tns:MessageIdentificationDescription" minOccurs="0"/>
203    <xsd:element name="storeTimeout" type="xsd:long" minOccurs="1"/>
204    <xsd:element name="Target" type="tns:DestinationDescription"></xsd:element>
205   </xsd:sequence>
206  </xsd:complexType>
207  <xsd:complexType name="MessageIdentificationDescription">
208   <xsd:sequence>
209    <xsd:element name="messageName" type="xsd:string"/>
210    <xsd:element name="protocolName" type="xsd:string"/>
211   </xsd:sequence>
212  </xsd:complexType>
213  <xsd:complexType name="DestinationDescription">
214   <xsd:choice>
215    <xsd:element name="Source">
216     <xsd:complexType>
217      <xsd:sequence>
218       <xsd:element name="methodName" type="xsd:string"/>
219       <xsd:element name="Property" type="tns:PropertyDescription"
220                    minOccurs="0" maxOccurs="unbounded"/>
221      </xsd:sequence>
222     </xsd:complexType>
223    </xsd:element>
224    <xsd:element name="Destination">
225     <xsd:complexType>
226      <xsd:sequence>
227       <xsd:element name="methodName" type="xsd:string"/>
228       <xsd:element name="Property" type="tns:PropertyDescription"
229                    minOccurs="0" maxOccurs="unbounded"/>
230      </xsd:sequence>
231     </xsd:complexType>
232    </xsd:element>
233    <xsd:element name="Other">
234     <xsd:complexType>
235      <xsd:sequence>
236       <xsd:element name="address" type="xsd:string"/>
237       <xsd:element name="methodName" type="xsd:string"/>
238       <xsd:element name="Property" type="tns:PropertyDescription"
239                    minOccurs="0" maxOccurs="unbounded"/>
240      </xsd:sequence>
241     </xsd:complexType>
242    </xsd:element>
```

```
243   </xsd:choice>
244  </xsd:complexType>
245  <xsd:complexType name="ErrorDescription">
246   <xsd:choice>
247    <xsd:element name="RemoveMessage"/>
248    <xsd:element name="ForwardMessage"/>
249    <xsd:element name="OriginalMessage"/>
250    <xsd:element name="Query" type="tns:QueryDescription"/>
251   </xsd:choice>
252  </xsd:complexType>
253 </xsd:schema>
```

Listing A.1: XML-Schema des DBM

A.3 PIM XSD-Schema

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4              targetNamespace="http://offis.de/schema/protocolModel"
5              xmlns:tns="http://offis.de//schema/messagesSchema"
6              elementFormDefault="qualified"
7              xmlns:ns="http://offis.de/schema/protocolModel">
8
9   <xsd:complexType name="IdentificationElementDescription">
10   <xsd:sequence>
11    <xsd:element name="queryType" type="xsd:string" minOccurs="1"/>
12    <xsd:element name="query" minOccurs="1" maxOccurs="1" type="xsd:string"/>
13    <xsd:element name="Property" type="ns:PropertyDescription" minOccurs="0"
14                 maxOccurs="unbounded"/>
15   </xsd:sequence>
16  </xsd:complexType>
17  <xsd:complexType name="PropertyDescription">
18   <xsd:sequence>
19    <xsd:element name="key" type="xsd:string"/>
20    <xsd:element name="value" type="xsd:string"/>
21   </xsd:sequence>
22  </xsd:complexType>
23  <xsd:element name="ProtocolModel">
24   <xsd:complexType>
25    <xsd:sequence>
26     <xsd:element name="protocolName" type="xsd:string"/>
27     <xsd:element name="ProtocolIdentification" maxOccurs="unbounded">
28      <xsd:complexType>
29       <xsd:sequence>
30        <xsd:element name="IdentificationElement"
31                     type="ns:IdentificationElementDescription"
32                     maxOccurs="unbounded"/>
33       </xsd:sequence>
34      </xsd:complexType>
35     </xsd:element>
36     <xsd:element name="Message" maxOccurs="unbounded">
37      <xsd:complexType>
38       <xsd:sequence>
39        <xsd:element name="messageName" type="xsd:string"/>
40        <xsd:element name="IdentificationElement"
41                     type="ns:IdentificationElementDescription"
42                     maxOccurs="unbounded"/>
```

```
43          </xsd:sequence>
44        </xsd:complexType>
45      </xsd:element>
46    </xsd:sequence>
47  </xsd:complexType>
48 </xsd:element>
49 </xsd:schema>
```

Listing A.2: XML-Schema des PIM

Glossar

Nachfolgend sind noch einmal wesentliche Begriffe dieser Arbeit zusammengefasst und erläutert. Eine ausführliche Erklärung findet sich jeweils in den einführenden Abschnitten sowie der jeweils darin angegebenen Literatur. Das im Folgenden im Rahmen der Erläuterung verwendete Symbol ~ bezieht sich jeweils auf den im Einzelnen vorgestellten Begriff, das Symbol ↑ verweist auf einen ebenfalls innerhalb dieses Glossars erklärten Begriff.

Adapter Ein ~ ist eine Software-Komponente, die die Kommunikation von Diensten mit anderen Systeme erlaubt. Er ist in der Lage, das Protokoll des ↑Quelldienstes zu sprechen und dieses in ein einheitliches Format zu übersetzen.

Adapter-Synthese Die ~ ist ein Prozess, bei dem eine abstrakte Beschreibung des Quell- und ↑Ziel-Protokolls sowie eine ↑Schnittstellen-Beschreibung der beteiligten ↑Protokolle verwendet wird, um automatisiert einen ↑Konverter zu erzeugen.

Anfrage-Sprache Eine ~ ist eine Sprache, mittels der die Transformation von Daten in ein anderes Format beschrieben werden kann. Die dabei zu formulierende Anfrage ist dabei normalerweise einfach zu lesen.

Anfrage/Antwort-Muster Das ~ ist ein Muster, das bei der Kommunikation mit Diensten auftritt. Dabei wird eine Nachricht an den ↑Zieldienst gesendet, die vom diesem zwingend mit einer Antwortnachricht beantwortet wird.

Client Ein ~ ist ein Teilnehmer an einer Kommunikation. Im Gegensatz zum ↑Server stellt er selbst keine Funktionen zur Verfügung. Er baut allerdings eine Kommunikation zu diesem auf, um die Kommunikation mit anderen Clients zu ermöglichen.

Dienst Ein ~ ist eine Applikation, die von einem ↑Server zur Verfügung gestellt wird. Der Dienst kann dabei Funktionen wie den Zugriff auf Daten, das Ausführen von Funktionen oder das Speichern von Daten auf dem Server ermöglichen.

Differential Behavior Model Das ~ ist ein Modell, dass die Unterschiede zwischen ↑Service-Protokollen beschreibt. Mittels eines generischen ↑Konverters kann das Differential Behavior Model interpretiert und somit Protokolle automatisiert übersetzt werden.

Endlicher Zustandsautomat Ein ~ ist ein Modell, das verwendet wird, um ein Verhalten zu beschreiben. Dabei werden Zustände, Zustandsübergänge und Aktionen beschrieben. Die Menge der Zustände muss endlich sein.

Entscheidungsbaum Ein ~ wird verwendet, um die Zustände eines Systems, sowie die Bedingungen, durch die diese Zustände erreicht werden können, zu beschreiben. Ein ~ wird dabei mittels einer Baumstruktur dargestellt. Die Wurzel beschreibt dabei den Ausgangszustand, die Blätter die Zustände, die erreicht werden können und die Kanten die Bedingungen unter denen die Zustände erreicht werden können.

Framework Ein ~ ist ein Programmiergerüst, das in der objektorientierten Softwareentwicklung verwendet wird. Ein ~ stellt dabei den grundlegenden Rahmen zur Verfügung, mittels dessen andere Applikationen implementiert werden können.

Hypothesentest Ein ~ ist ein statistischer Test, der auf Grundlage einer vorhandenen Beobachtung eine begründete Entscheidung erlaubt, ob eine Hypothese zutrifft oder nicht.

Instant Messaging Protokolle Die ~ beschreiben eine Klasse von Protokollen. Diese erlauben die Übertragung von Textnachrichten zwischen zwei oder mehr Quellen.

Konverter Ein ~ ist (in diesem Fall) eine Komponente, die ein ↑Service-Protokoll in ein anderes übersetzt. Dabei werden sowohl die Daten, als auch das ↑Kommunikationsverhalten übersetzt.

Kommunikationsverhalten Das ~ ist das Verhalten eines Dienstes, das mittels des ↑Protokolls beschrieben wird. Wird zum Beispiel nach dem Empfangen einer Nachricht, abhängig von deren Inhalt, eine bestimmte Nachricht zurück gesendet, ist dies Teil des Kommunikationsverhaltens.

Message Exchange Pattern Ein ~ ist eine klar definierte Abfolge von Nachrichten, die im Rahmen einer Kommunikation übertragen werden. Sie können verwendet werden, um das ↑Kommunikationsverhalten zu beschreiben.

Middleware Eine ~ ist ein System, das zwischen der Applikation und den Datenquellen (z.B. ↑Sensoren und ↑Dienste) vermittelt. Die Middelware abstrahiert dabei die Komplexität der Datenquellen, so dass die Applikation entsprechend entlastet wird.

Muster Im Rahmen dieser Arbeit wird ein ~ als ein wiederkehrendes Verhalten beim Übersetzen des ↑Kommunikationsverhaltens eines Protokolls verstanden.

Poller Ein ~ ist eine Applikation, die kontinuierlich Daten bei einer Quelle (z.B. ↑Sensoren und ↑Dienste) anfragt und diese nach Erhalt weiterleitet.

Quelldienst Ein ~ ist ein ↑Dienst, der im Rahmen einer Kommunikation eine Nachricht an einen anderen ↑Dienst sendet.

Regulärer Ausdruck Ein ~ ist eine Zeichenkette, die die Beschreibung von Mengen mittels einer fest definierten Sprache erlaubt.

Schnittstellen-Beschreibung Eine ~ ist eine abstrakte Beschreibung der Schnittstelle eines ↑Dienstes. Abhängig von der Art der ~ beschreibt diese neben den Daten auch das ↑Kommunikationsverhalten.

Sensor Ein ~ ist eine physikalische Einheit, die ein Phänomen durch die Messung physikalischer oder chemischer Eigenschaft wahrnimmt. Die Ergebnisse der Messung werden vom Sensor anderen Quellen zur Verfügung gestellt oder an diese weitergeleitet.

Server Ein ~ ist eine Applikation, die einem ↑Client Daten oder Funktionen über eine definierte Schnittstelle anbietet.

Service-Protokolle ~ beschreiben eine Klasse von Protokollen. Diese erlauben den Zugriff auf die Funktionen eines ↑Dienstes mittels einer klar definierten ↑Schnittstellen-Beschreibung.

Sequenz-Diagramm Ein ~ ist ein Verhaltensdiagramm. Dabei wird üblicherweise ein Weg durch einen ↑Entscheidungsbaum dargestellt.

Template Ein ~ ist eine Schablone, die verwendet werden kann, um Daten in eine neue Form zu bringen.

Tiefensuche Die ~ ist ein Suchalgorithmus, der mittels einer klar definierten Folge alle Knoten eines ↑Entscheidungsbaums durchlaufen kann.

Tripel-Graph-Grammatik Die ~ ist eine spezielle Art der Graphgrammatik, die die Beschreibung von Unterschieden zwischen Graphen erlaubt.

Turing-Mächtig Als ~ wird die allgemeine Programmierbarkeit eines Systems beschrieben. Ist ein System ~ , kann sie alle Funktionen einer Turingmaschine abbilden und somit eine logische Programmiersprache abbilden.

Verhaltensänderung Als ~ wird im Rahmen dieser Arbeit eine Änderung des ↑Kommunikationsverhaltens bezeichnet, die vom ↑Konverter vorgenommen wurde.

Web-Dienst Ein ~ ist ein ↑Dienst, auf den über World Wide Web zugegriffen werden kann.

Wrapper Ein ~ ist eine Applikation, die die Daten und das Kommunikationsverhalten übersetzt, so dass sie von einer anderen Quelle verstanden werden können.

XML-Schema Ein ~ ist eine formale Beschreibung der Syntax eines bestimmten XML-Dokumentes.

Zieldienst Ein ~ ist ein ↑Dienst, der von einem ↑Quelldienst im Rahmen seines ↑Kommunikationsverhaltens eine Nachricht empfängt.

Abkürzungen

BPEL	WS-Business Process Execution Language
DBM	Differential Behavior Model
GPS	Global Positioning System
IRC	Internet Relay Chat
MEP	Message Exchange Pattern
NMEA	National Marine Electronics Association
O&M	Observations and Measurements
OGC	Open Geospatial Consortium
OSI	Open Systems Interconnection
PBC	ProviousBehaviorChange
PC	ProviousCondition
PIM	Protocol Identification Model
REST	Representational State Transfer
RFC	Request for Comments
SCAI	Sensor Configuration, Aggregation and Interchange
SCAMPI	Sensor Configuration and Aggregation Middleware for Multi Platform Interchange
SensorML	Sensor Model Language
SS	ServiceState
STOMP	Streaming Text Oriented Messaging Protocol
SDL	Specification and Description Language
SOA	Service-oriented architecture
SOAP	ursprüngl. Simple Object Access Protocol
SOS	Sensor Observation Service
SWE	Sensor Web Enablement
TGG	Tripel-Graph-Grammatik
UML	Unified Modeling Language
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
WSN	Wireless Sensor Network
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XSLT	Extensible Stylesheet Language Transformations

Abbildungen

Literatur

[52°12] 52°North Initiative: *Packages - Sensor Web*, 2012. `http://52north.org/downloads/sensor-web`

[AHS07] Aberer, Karl ; Hauswirth, Manfred ; Salehi, Ali: Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks. In: Becker, Christian (Hrsg.) ; Jensen, Christian S. (Hrsg.) ; Su, Jianwen (Hrsg.) ; Nicklas, Daniela (Hrsg.): *MDM*, IEEE, 2007. – ISBN 1–4244–1240–4, S. 198–205

[AK10] Athanasopoulos, M. ; Kontogiannis, K.: Identification of REST-like Resources from Legacy Service Descriptions. In: *Reverse Engineering (WCRE), 2010 17th Working Conference on*, 2010. – ISSN 1095–1350, S. 215 –219

[AM91] Akella, Janaki ; McMillan, Kenneth L.: Synthesizing Converters Between Finite State Protocols. In: *ICCD*, IEEE Computer Society, 1991. – ISBN 0–8186–2270–9, S. 410–413

[BB11] Busemann, Claas ; Behrensen, Stefan: Requirements and Final Specification of the SCAI-Protocol / OFFIS - Institute for Information Technology. 2011. – Forschungsbericht

[BBC05] Bracciali, Andrea ; Brogi, Antonio ; Canal, Carlos: A formal approach to component adaptation. In: *Journal of Systems and Software* 74 (2005), Nr. 1, S. 45–54

[BBH+08] Böhm, Matthias ; Bittner, Jürgen ; Habich, Dirk ; Lehner, Wolfgang ; Wloka, Uwe: Model-Driven Generation of Dynamic Adapters for Integration Platforms. In: *1st International Workshop on Model Driven Interoperability for Sustainable Information Systems*, 2008

[BCF+07] Boag, Scott ; Chamberlin, Don ; Fernández, Mary F. ; Florescu, Daniela ; Robie, Jonathan ; Siméon, Jérôme: XQuery 1.0: An XML Query Language / W3C. 2007. – Forschungsbericht

[BCG+05] Benatallah, Boualem ; Casati, Fabio ; Grigori, Daniela ; Nezhad, Hamid R. M. ; Toumani, Farouk: Developing Adapters for Web Services Integration. In: Pastor, Oscar (Hrsg.) ; Cunha, Joao F. (Hrsg.): *CAiSE* Bd. 3520, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–26095–1, S. 415–429

[BEK+00] Box, Don ; Ehnebuske, David ; Kakivaya, Gopal ; Layman, Andrew ; Mendelsohn, Noah ; Nielsen, Henrik F. ; Tahtte, Satish ; Winer,

Dave: Simple Object Access Protocol (SOAP) 1.1 / World Wide Web Consortium (W3C). 2000. – Forschungsbericht

[BFJ10] Bröring, Arne ; Foerster, Theodor ; Jirka, Simon: Interaction patterns for bridging the gap between sensor networks and the Sensor Web. In: *P*erCom Workshops[DBL10], S. 732–737

[BJR03] Buettner, H. ; Janssen, D. ; Rostan, M.: EtherCAT: The Ethernet Fieldbus, PC Control Magazine 3: 14 to 19. 2003. – Forschungsbericht

[BKL+99] Box, Don ; Kakivaya, Gopal ; Layman, Andrew ; Tahtte, Satish ; Winer, Dave: Simple Object Access Protocol / Microsoft Corporation and Userland Sodtware. 1999. – Forschungsbericht

[BKNB11] Busemann, Claas ; Kuka, Christian ; Nicklas, Daniela ; Boll, Susanne: Flexible and Efficient Sensor Data Processing - A Hybrid Approach. In: Härder, Theo (Hrsg.) ; Lehner, Wolfgang (Hrsg.) ; Mitschang, Bernhard (Hrsg.) ; Schöning, Harald (Hrsg.) ; Schwarz, Holger (Hrsg.): *BTW* Bd. 180, GI, 2011 (LNI). – ISBN 978–3–88579–274–1, S. 123–134

[BKW+09] Busemann, Claas ; Kuka, Christian ; Westermann, Utz ; Boll, Susanne ; Nicklas, Daniela: SCAMPI - Sensor Configuration and Aggregation Middleware for Multi Platform Interchange. In: *Informatik 2009: Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 28.9.-2.10.2009, Lübeck, Proceedings*, 2009, S. 2084–2097

[BN11a] Busemann, Claas ; Nicklas, Daniela: Combining Sensor Systems: A Differential Behavior Model. In: *International Transactions on Systems Science and Applications, Vol. 7, No. 1/2*, 2011, S. 83–95

[BN11b] Busemann, Claas ; Nicklas, Daniela: Converting Conversation Protocols Using an XML Based Differential Behavioral Model. In: *DEXA*, 2011

[BP06] Brogi, Antonio ; Popescu, Razvan: Automated Generation of BPEL Adapters. In: [DL06], S. 27–39

[BPRD07] Botts, Mike ; Percivall, George ; Reed, Carl ; Davidson, John: OGC® Sensor Web Enablement: Overview And High Level Architecture. / Open Geospatial Consortium, Inc. 2007. – Forschungsbericht

[BR07] Botts, Mike ; Robin, Alexandre: OpenGIS® Sensor Model Language (SensorML) Implementation Specification / Open Geospatial Consortium, Inc. 2007. – Forschungsbericht

[Bro06] Brodie, Michael L.: Integration in A Service-Oriented World: The Big Picture. In: *I-ESA*, 2006

[BSE12] Bröring, Arne ; Stasch, Christoph ; Echterhoff, Johannes: OGC® Sensor Observation Service Interface Standard / Open Geospatial Consortium, Inc. 2012. – Forschungsbericht

[BSSGM10] Bittencourt, R.A. ; Souza Santos, G. Jansen d. ; Guerrero, D.D.S. ; Murphy, G.C.: Improving Automated Mapping in Reflexion Models Using Information Retrieval Techniques. In: *Reverse Engineering (WCRE), 2010 17th Working Conference on*, 2010. – ISSN 1095–1350, S. 163 –172

[CB07] Chu, Xingchen ; Buyya, Rajkumar: Service Oriented Sensor Web. In: Mahalik, Nitaigour P. (Hrsg.): *Sensor Networks and Configuration*. Springer Berlin Heidelberg, 2007. – ISBN 978–3–540–37366–7, S. 51–74

[Cox11] Cox, Simon: Observations and Measurements - XML Implementation / Open Geospatial Consortium, Inc. 2011. – Forschungsbericht

[Cro06] Crockford, D.: RFC: 4627, The application/json Media Type for JavaScript Object Notation (JSON) / Network Working Group. 2006. – Forschungsbericht

[DBL10] *Eigth Annual IEEE International Conference on Pervasive Computing and Communications, PerCom 2010, March 29 - April 2, 2010, Mannheim, Germany, Workshop Proceedings*. IEEE, 2010

[DL06] Dan, Asit (Hrsg.) ; Lamersdorf, Winfried (Hrsg.): *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*. Bd. *4294*. Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–68147–7

[DSW06] Dumas, Marlon ; Spork, Murray ; Wang, Kenneth: Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In: Dustdar, Schahram (Hrsg.) ; Fiadeiro, José Luiz (Hrsg.) ; Sheth, Amit P. (Hrsg.): *Business Process Management* Bd. 4102, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–38901–6, S. 65–80

[FBK+11] Funk, Alexander ; Busemann, Claas ; Kuka, Christian ; Boll, Susanne ; Nicklas, Daniela: Open Sensor Platforms: The Sensor Web

Enablement Framework and Beyond. In: *6. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS)*, 2011

[Fli12] Flightradar24.com: *How Flightradar24 works*, 2012. `http://www.flightradar24.com/about.php`

[FNO05] Fischer, Joachim ; Neumann, Toby ; Olsen, Anders: SDL Code Generation for Open Systems. In: Prinz, Andreas (Hrsg.) ; Reed, Rick (Hrsg.) ; Reed, Jeanne (Hrsg.): *SDL Forum* Bd. 3530, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–26612–7, S. 313–322

[Fuc04] Fuchs, Matthew: Adapting Web Services in a Heterogeneous Environment. In: *ICWS*, IEEE Computer Society, 2004, S. 656–

[Fun10] Funk, Alexander: *Sensoren im Netz: Integration des "Sensor Web Enablement" in eine Middleware*, Carl von Ossietzky Universität Oldenburg, Diplomarbeit, 2010

[GGB+09] Groppe, Sven ; Groppe, Jinghua ; Böttcher, Stefan ; Wycisk, Thomas ; Gruenwald, Le: Optimizing the execution of XSLT stylesheets for querying transformed XML data. In: *Knowl. Inf. Syst.* 18 (2009), Nr. 3, S. 331–391

[GGP03] Glässer, U. ; Gotzhein, R. ; Prinz, A.: The formal semantics of SDL-2000: Status and perspectives. In: *Computer Networks* 42 (2003), Nr. 3, S. 343 – 358. – ISSN 1389–1286. – ITU-T System Design Languages (SDL)

[GHM+07a] Gudgin, Martin ; Hadley, Marc ; Mendelsohn, Noah ; Moreau, Jean-Jacques ; Nielson, Henrik F. ; Karmarkar, Anish ; Lafon, Yves: SOAP Version 1.2 Part 1: Messaging Framework (Second Edition) / World Wide Web Consortium (W3C). 2007. – Forschungsbericht

[GHM+07b] Gudgin, Martin ; Hadley, Marc ; Mendelsohn, Noah ; Moreau, Jean-Jacques ; Nielson, Henrik F. ; Karmarkar, Anish ; Lafon, Yves: SOAP Version 1.2 Part 2: Adjuncts (Second Edition) / World Wide Web Consortium (W3C). 2007. – Forschungsbericht

[GLH09] Gürgen, Levent ; Labbé, Cyril ; Honiden, Shinichi: Opérations d'administration pour SStreaMWare. In: Menga, David (Hrsg.) ; Sedes, Florence (Hrsg.): *UbiMob* Bd. 394, ACM, 2009 (ACM International Conference Proceeding Series). – ISBN 978–1–60558–622–9, S. 41–44

[GMR+10] Glombitza, Nils ; Mietz, Richard ; Romer, Kay ; Fischer, Stefan ; Pfisterer, Dennis: Self-Description and Protocol Conversion for

a Web of Things. In: *Sensor Networks, Ubiquitous, and Trustworthy Computing, International Conference on* 0 (2010), S. 229–236. ISBN 978–0–7695–4049–8

[GPU10] Gupta, Vipul ; Poursohi, Arshan ; Udupi, Poornaprajna: Sensor.Network: An open data exchange for the web of things. In: *P*erCom Workshops[DBL10], S. 753–755

[Gra05] Graaumans, Joris Petrus M.: *Usability of XML Query Languages*, Utrecht University, Diss., 2005

[GRL+08] Gurgen, Levent ; Roncancio, Claudia ; Labbé, Cyril ; Bottaro, André ; Olive, Vincent: SStreaMWare: a service oriented middleware for heterogeneous sensor data management. In: *ICPS '08: Proceedings of the 5th international conference on Pervasive services*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–135–4, S. 121–130

[GSLS+08] García-Sánchez, Pablo ; Laredo, Juan Luís J. ; Sevilla, Juan P. ; Castillo, Pedro A. ; Guervós, Juan Julián M.: Improved evolutionary generation of XSLT stylesheets. In: *CoRR* abs/0803.1926 (2008)

[Had07] Hadley, Marc: What'sNew in SOAP 1.2 / Sun Microsystems. 2007. – Forschungsbericht

[HU90] Hopcroft, J.E. ; Ullman, J.D.: *Introduction To Automata Theroy, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990

[HU94] Hopcroft, John E. ; Ullman:, Jeffry D.: *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison Wesley, Bonn, 1994

[Int96] Internationalen Organisation für Normung: ISO 7498-1, Information technology - Open Systems Interconnection - Basic Reference Model: The basic model / ISO. 1996. – Forschungsbericht

[Joh96] John Brooke: SUS - A quick and dirty usability scale / Redhatch Consulting Ltd. 1996. – Forschungsbericht

[KANK08] Kenzi, Adil ; Asri, Bouchra E. ; Nassar, Mahmoud ; Kriouile, Abdelaziz: The Multiview Service: A New Concept for the Development of Adaptable Service Oriented Systems. In: Lee, Jonathan (Hrsg.) ; Liang, Deron (Hrsg.) ; Cheng, Y. C. (Hrsg.): *SOSE*, IEEE Computer Society, 2008, S. 38–43

[Kay07] Kay, Michael: XSL Transformations (XSLT) Version 2.0 / W3C. 2007. – Forschungsbericht

[KB10] Kuka, Christian ; Busemann, Claas: Sensormanagement / OFFIS - Institute for Information Technology. 2010. – Forschungsbericht

[KBNB11] Kuka, Christian ; Busemann, Claas ; Nicklas, Daniela ; Boll, Susanne: Mashups for Community Aware Sensor Processing with SCAMPI. In: *BTW Workshops*, 2011, S. 52–57

[Kep04] Kepser, Stephan: A Simple Proof for the Turing-Completeness of XSLT and XQuery. In: *Extreme Markup Languages®*, 2004

[Kle56] Kleene, Stephen C.: Representation of events in nerve nets and finite automata. In: *Automata Studies, Princeton University Press. NJ* (1956), S. 3–41

[KLKS10] Klar, Felix ; Lauder, Marius ; Königs, Alexander ; Schürr, Andy: Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In: Engels, Gregor (Hrsg.) ; Lewerentz, Claus (Hrsg.) ; Schäfer, Wilhelm (Hrsg.) ; Schürr, Andy (Hrsg.) ; Westfechtel, Bernhard (Hrsg.): *Graph Transformations and Model-Driven Engineering* Bd. 5765, Springer, 2010 (Lecture Notes in Computer Science). – ISBN 978–3–642–17321–9, S. 141–174

[KSPBC06] Kongdenfha, Woralak ; Saint-Paul, Régis ; Benatallah, Boualem ; Casati, Fabio: An Aspect-Oriented Framework for Service Adaptation. In: [DL06], S. 15–26

[LH10] Lemaitre, Jonathan ; Hainaut, Jean-Luc: Transformation-Based Framework for the Evaluation and Improvement of Database Schemas. In: *Advanced Information Systems Engineering, 21st International Conference, CAiSE 2009, Amsterdam, The Netherlands, June 8-12, 2009. Proceedings*, 2010, S. 317–331

[Liu96] Liu, Ming T.: Network Interconnection and Protocol Conversion. In: *Advances in Computers* 42 (1996), S. 119–239

[LKJ+05] Letichevsky, Alexander A. ; Kapitonova, Julia V. ; Jr., A. A. L. ; Volkov, Vladislav A. ; Baranov, Sergey ; Weigert, Thomas: Basic protocols, message sequence charts, and the verification of requirements specifications. In: *Computer Networks* 49 (2005), Nr. 5, S. 661–675

[LL92] Ladkin, Peter B. ; Leue, Stefan: On the Semantics of Message Sequence Charts. In: König, Hartmut (Hrsg.): *FBT*, K. G. Saur Verlag, 1992. – ISBN 3–598–22409–5, S. 88–104

[LP11] Lempert, Sebastian ; Pflaum, Alexander: Towards a Reference Architecture for an Integration Platform for Diverse Smart Object Technologies. In: *MMS*, 2011

[MBPF09] Mao, Lu ; Belhajjame, Khalid ; Paton, Norman W. ; Fernandes, Alvaro A. A.: Defining and Using Schematic Correspondences for Automatically Generating Schema Mappings. In: *Advanced Information Systems Engineering, 21st International Conference, CAiSE 2009, Amsterdam, The Netherlands, June 8-12, 2009. Proceedings*, 2009, S. 79–93

[MR97] Mauw, Sjouke ; Reniers, Michel A.: High-level message sequence charts. In: Cavalli, Ana R. (Hrsg.) ; Sarma, Amardeo (Hrsg.): *SDL Forum*, Elsevier, 1997, S. 291–306

[MR09] McCann, Dónall ; Roantree, Mark: A Query Service for Raw Sensor Data. In: *Smart Sensing and Context, 4th European Conference, EuroSSC 2009, Guildford, UK, September 16-18, 2009. Proceedings*, 2009, S. 38–50

[Nat10] National Marine Electronics Association: NMEA 0183 Standard. 2010. – Forschungsbericht

[NBM+07] Nezhad, Hamid R. M. ; Benatallah, Boualem ; Martens, Axel ; Curbera, Francisco ; Casati, Fabio: Semi-automated adaptation of service interactions. In: Williamson, Carey L. (Hrsg.) ; Zurko, Mary E. (Hrsg.) ; Patel-Schneider, Peter F. (Hrsg.) ; Shenoy, Prashant J. (Hrsg.): *WWW*, ACM, 2007. – ISBN 978–1–59593–654–7, S. 993–1002

[Nie93] Nierstrasz, Oscar: Regular Types for Active Objects. In: Babitsky, Timlynn (Hrsg.) ; Salmons, Jim (Hrsg.): *OOPSLA*, ACM, 1993. – ISBN 0–89791–587–9, S. 1–15

[Oku90] Okumura, Kaoru: Generation of Proper Adapters and Converters From A Formal Service Specification. In: *INFOCOM*, 1990, S. 564–571

[OR93] Oikarinen, J. ; Reed, D.: RFC 1459: Internet Relay Chat Protocol / Network Working Group. 1993. – Forschungsbericht

[Ora10] Oracle Corporation: *Oracle Sensor Edge Server*, 2010. `http://www.oracle.com/technology/products/\sensor_edge_server/index.html`

[OS04] Onose, Nicola ; Simeon, Jerome: XQuery at your web service. In: *Proceedings of the 13th international conference on World Wide Web*.

New York, NY, USA : ACM, 2004 (WWW '04). – ISBN 1–58113–844–X, 603–611

[PAHSV02] Passerone, Roberto ; Alfaro, Luca de ; Henzinger, Thomas A. ; Sangiovanni-Vincentelli, Alberto L.: Convertibility verification and converter synthesis: two faces of the same coin. In: Pileggi, Lawrence T. (Hrsg.) ; Kuehlmann, Andreas (Hrsg.): *ICCAD*, ACM, 2002. – ISBN 0–7803–7607–2, S. 132–139

[PF04] Ponnekanti, Shankar ; Fox, Armando: Interoperability Among Independently Evolving Web Services. In: Jacobsen, Hans-Arno (Hrsg.): *Middleware* Bd. 3231, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–23428–4, S. 331–351

[PRSV98] Passerone, Roberto ; Rowson, James A. ; Sangiovanni-Vincentelli, Alberto L.: Automatic Synthesis of Interfaces Between Incompatible Protocols. In: *DAC*, 1998, S. 8–13

[QPPS08] Quartel, Dick A. C. ; Pokraev, Stanislav ; Pessoa, Rodrigo M. ; Sinderen, Marten van: Model-Driven Development of a Mediation Service. In: *EDOC*, IEEE Computer Society, 2008. – ISBN 978–0–7695–3373–5, S. 117–126

[RB01] Rahm, Erhard ; Bernstein, Philip A.: A survey of approaches to automatic schema matching. In: *VLDB J.* 10 (2001), Nr. 4, S. 334–350

[Rin08] Rinne, Horst: *Taschenbuch der Statistik. 4.* Harri Deutsch, Frankfurt am Main, 2008

[Roy00] Roy Thomas Fielding: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Diss., 2000

[RTTZ04] Roychoudhury, Abhik ; Thiagarajan, P. S. ; Tran, Tuan-Anh ; Zvereva, Vera A.: Automatic Generation of Protocol Converters from Scenario-Based Specifications. In: *RTSS*, IEEE Computer Society, 2004. – ISBN 0–7695–2247–5, S. 447–458

[SA11] Saint-Andre, Peter: RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core / Internet Engineering Task Force (IETF). 2011. – Forschungsbericht

[Sch94] Schürr, Andy: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, Ernst W. (Hrsg.) ; Schmidt, Gunther (Hrsg.) ; Tinhofer, Gottfried (Hrsg.): *WG* Bd. 903, Springer, 1994 (Lecture Notes in Computer Science). – ISBN 3–540–59071–4, S. 151–163

[SNL+06] Santanche, Andre ; Nath, Suman ; Liu, Jie ; Priyantha, Bodhi ; Zhao, Feng: SenseWeb: Browsing the Physical World in Real Time. In: *Demo Abstract, ACM/IEEE IPSN 2006, Nashville, TN*, April 2006

[SR02] Schmidt, Heinz W. ; Reussner, Ralf: Generating Adapters for Concurrent Component Protocol Synchronisation. In: Jacobs, Bart (Hrsg.) ; Rensink, Arend (Hrsg.): *FMOODS* Bd. 209, Kluwer, 2002 (IFIP Conference Proceedings). – ISBN 0–7923–7683–8, S. 213–229

[STSA09] Smith, Kevin ; Tronçon, Remko ; Saint-Andre, Peter: *XMPP: The Definitive Guide: Building Real-Time Applications with Jabber Technologies*. O'Reilly Media, 2009

[TBD95] Tao, Z.P. ; Bochman, G.v. ; Dssouli, R.: An efficient method for protocol conversion. In: *Computer Communications and Networks, International Conference on* 0 (1995), S. 0040. ISBN 0–8186–7180–7

[The06] The Codehaus: Streaming Text Orientated Messaging Protocol v1.0. 2006. – Forschungsbericht

[The12] The Apache Software Foundation: *Apache CXF - CXF User's Guide*, 2012. `http://cxf.apache.org/docs/index.html`

[Tur37] Turing, Alan: On Computable Numbers, with an Application to the Entscheidungsproblem. In: *London Mathematical Society. Bd. s2-42, Nr. 1*, 1937, S. 230–265

[Tur38] Turing, Alan: On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction. In: *London Mathematical Society. Bd. s2-43, Nr. 1*, 1938, S. 544–546

[WC08] Wang, Xinwei ; Cao, Chunjing: Mining Association Rules from Complex and Irregular XML Documents Using XSLT and Xquery. In: *ALPIT 2008, Proceedings of The Seventh International Conference on Advanced Language Processing and Web Information Technology, Dalian University of Technology, Liaoning, China, 23-25 July 2008*, 2008, S. 314–319

[WCL+08] Weerawarana, Sanjiva ; Cubera, Francisco ; Leymann, Fran ; Storey, Tony ; Ferguson, Donald F.: *Web Service Platform Architecture*. Prentice Hall, 2008

[Yah07] Yahoo! Inc: *FireEagle: Centralized management of user location*, 2007. `http://fireeagle.yahoo.net/developer`

[YMHF01] Yan, Ling-Ling ; Miller, Renée J. ; Haas, Laura M. ; Fagin, Ronald: Data-Driven Understanding and Refinement of Schema Mappings. In: *SIGMOD Conference*, 2001, S. 485–496

[YS97] Yellin, Daniel M. ; Storm, Rober E.: Protocol Specifications and Component Adaptors. In: *ACM Transactions on Programming Languages and Systems*, 1997

[ZM09] Zahia, MAROUF ; Malki, Mimoun: Model Driven Adapter generation for Web services. In: *JEESI*, 2009

[Zve04] Zvereva, Vera A.: Converter between incompatible protocols specified using MSC diagrams / National Univ. of Singapore. 2004. – Forschungsbericht

Index

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe. Außerdem versichere ich, dass ich den Inhalt dieser Arbeit nicht schon für eine Diplomarbeit oder Ähnliches verwendet habe. Darüber hinaus versichere ich, dass ich die Regeln guter wissenschaftlicher Praxis entsprechend der DFG-Richtlinien eingehalten habe. Zudem versichere ich, dass ich keine kommerziellen Beratungs- oder Vermittlungsdienste in Anspruch genommen habe.

Schaafheim, den 29. November 2012

Dipl.-Inf. (FH) Claas Busemann

Claas Busemann

Lebenslauf

Akademische Ausbildung

1999–2001 **Fachoberschule Leer**, *Hochschulreife*.

2001–2006 **Fachhochschule Oldenburg Ostfriesland Wilhelmshaven**, *Diplom der Informatik*.

2009–2012 **Carl von Ossietzky Universität Oldenburg**, *Promotion zur Erlangung des Grades eines Doktors der Ingenieurwissenschaften*.

Beruflicher Werdegang

2005–2006 **Volkswagen AG**, *Emden*, Werkstudent, Entwicklung von Web-Anwendungen für den Centers of Competence e.V..

2006–2012 **OFFIS e.V.**, *Oldenburg*, Wissenschaftlicher Mitarbeiter, Entwicklung von serviceorientierten Systemen zur Kommunikation mit Diensten und Sensoren.

seit 2012 **AGT International**, *Software Developer*, Entwicklung von Sicherheitsanwendungen.

Schaafheim, 26. November 2012

Im Herrngarten 12a – 64850 Schaafheim
✉ *claas.busmann@gmx.de*