Technische Universität Hamburg-Harburg
Arbeitsbereich Softwaresysteme

Diplomarbeit

zur Erlangung des Grades Diplom-Ingenieur

Data Abstraction Layer für den generischen Zugriff auf Infotainment Steuergeräte

im Studiengang Informatik - Ingenieurwesen

bei der Firma Berner & Mattner Systemtechnik GmbH
in Ottobrunn

Lars Brokmeier
Matrikelnummer: 16261

Fertig gestellt am: 30.06.2006

Betreuung:

Dr. Joachim W. Schmidt
Dipl. Inform. Rainer Marrone

Inhaltsverzeichnis

Erklärung

Ich versichere, die vorliegende Diplomarbeit selbstständig ohne fremde Hilfe und nur unter Benutzung der angegebenen Quellen angefertigt zu haben. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden. Alle Ausführungen der Arbeit, die wörtlich oder sinngemäß übernommen wurden, sind als solche kenntlich gemacht.

_____

Lars Brokmeier

Hamburg-Harburg im Juli 2006

Abbildungsverzeichnis

Abkürzungsverzeichnis

A2A	Application-to-Application
API	Application Programming Interface
B2B	Business-to-Business
CAN	Controller Area Network
CORBA	Common Object Request Broker Architecture
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
EAI	Enterprise Application Integration
EDA	Event-Driven Architecture
FOT	Fiber Optical Transceiver
HMI	Human Machine Interface
MDA	Model-Driven Architecture
MOST	Media Oriented Systems Transport
NIC	Netzwerk Interface Controller
P2P	Point-to-Point / Peer-to-Peer
POF	Plastical Optical Fiber
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
TCP	Transport Control Protocol
TTCAN	Time-triggered CAN
UDDI	Universal Description, Discovery and Integration
WSDL	Web Services Description Language

1 Einleitung

1.1 Motivation

Viele Stunden verbringt der Durchschnittsbürger jährlich im Auto, sei es auf dem Weg zur Arbeit, zum Einkaufen oder in den Urlaub. Zahlreiche Elektronikfunktionen im Automobil sorgen dabei für die notwendige Sicherheit und den gewünschten Komfort. Die verschiedenen Anwendungen softwaregesteuerter Elektronikkomponenten stellen dabei unterschiedliche Anforderungen an die verwendeten Bauteile und deren Kommunikation untereinander. Sicherheitskritische Anwendungen, wie etwa das Stabilitätsprogramm ESP, welches elektronisch Bremse, Motor und Getriebe miteinander koordiniert und den Fahrer in seiner Reaktion unterstützt, stellen hohe Anforderungen an die Zuverlässigkeit der Bauteile und deren Verbindungen, sowie an eine fehlertolerante Datenübertragung. Im Bereich der Multimediaanwendungen stehen dagegen eher Kosten- und Leistungsaspekte (z.B. hohe Übertragungsraten) im Vordergrund.

Das Automobil ist heutzutage eine mobile Kommunikationsplattform, in der „Home-Entertainment“ - Geräte wie Radio, CD/DVD, Video, Telefon/Handys oder Notebooks genutzt werden können. Auch komplexe Navigationssysteme mit Sprachsteuerung können durch entsprechenden Softwareeinsatz angesteuert werden. Diese Systeme existieren dabei nicht nur nebeneinander in einem Fahrzeug, sondern kommunizieren untereinander über definierte Protokolle. So kann zum Beispiel ein Telefon Informationen über einen eingehenden Anruf an das Radio (bzw. einen Verstärker) weiterleiten, um die aktuelle Wiedergabe zu unterbrechen und das Telefonat über die Lautsprecher zu empfangen.

Aufgrund der unterschiedlichen Anforderungen haben sich parallel mehrere Techniken, wie CAN, MOST oder FlexRay, als Standards etabliert.

Um Steuergeräte, wie sie im Automobil eingebaut werden, auch außerhalb des Fahrzeugs testen zu können, werden modellbasierte Simulationssysteme (wie MODENA von der Firma Berner & Mattner Systemtechnik) eingesetzt. Diese ermöglichen es, das Zusammenspiel existierender Steuergeräte zu überwachen und neue Hardware bereits bei der Entwicklung zu testen.

Aktuell wird die Anbindung einer speziellen Anwendung an eine bestimmte Hardware meist für jede Konfiguration individuell neu aufgesetzt. Das heißt, dass für eine bestimmte

Testumgebung jeweils eine neue, beziehungsweise neu angepaßte Schnittstelle entwickelt werden muss. Dadurch entsteht ein erheblicher Zeit und Kostenaufwand bei der Umsetzung geänderter Systemanforderungen. Daher wird eine flexible, einfach zu erweiternde Mittelschicht benötigt, welche die Integration verschiedener Anwendungen mit mehreren Hardwareschnittstellen ermöglicht.

1.2 Thema und Aufgabenstellung der Arbeit

In der Diplomarbeit geht es um die Entwicklung eines generischen Kommunikationsframeworks als Schnittstelle zwischen der Anwendungsschicht (MODENA / RAVENNA) und der Hardwareschicht (MOST / CAN / FlexRay).
Aktuell wird das Infotainment Testsystem MODENA bei Berner & Mattner eingesetzt, um Steuergeräte anhand von Simulationen bereits während der Spezifikationsphase testen zu können. Dabei können reale Steuersysteme, die zum Beispiel über ein MOST Bussystem kommunizieren, nach und nach eingebunden werden.

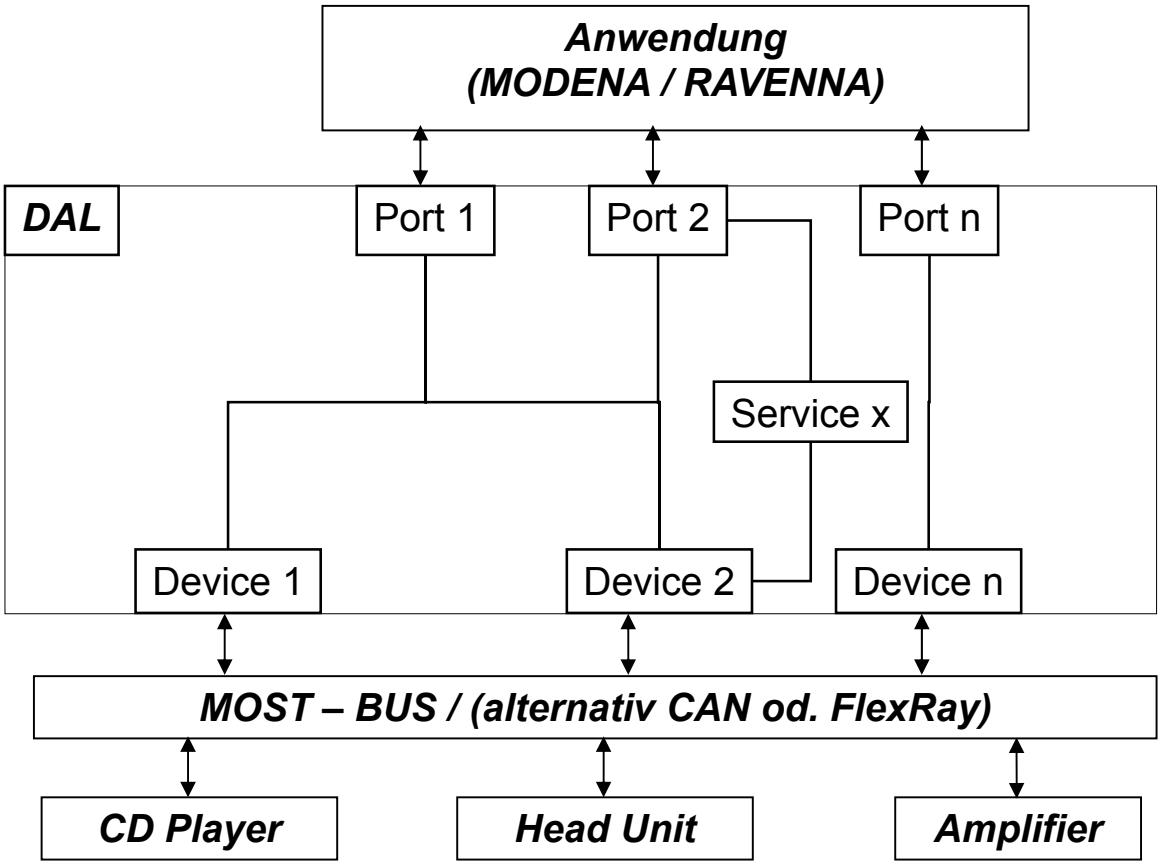


Abbildung 1: DAL Struktur

Das zu erstellende Kommunikationsframework soll einen generellen Zugriff auf eine Vielzahl von Steuergeräten unterschiedlicher Anbieter ermöglichen, welche an den verschiedenen Bussystemen angeschlossen sein können.
Hierzu wird es zunächst Teil der Aufgabe sein, sich in die Kommunikationsprotokolle heutiger Infotainmentsystemen einzuarbeiten, um die Anforderungen für eine generische Schnittstelle zwischen Anwendung und Steuergeräten zu spezifizieren.
Das Framework soll sowohl Anbindungen (Ports) an verschiedene Applikationen (vor

allem MODENA und RAVENNA), als auch Schnittstellen zu den unterschiedlichen Bussystemen (Devices) bereitstellen. Bei Berner & Mattner wird zum Beispiel der OptoLyzer von Oasis eingesetzt, um Nachrichten vom MOST Bus mitzulesen oder auf diesen zu schicken. Diese Geräte sollen (als Devices) in das Framework eingebunden werden. Über integrierte Services sollen zusätzliche Funktionen, wie etwa ein Nachrichtenfilter, realisiert werden. Die internen Verbindungspfade zwischen Ports und Devices (eventuell über Services) sollen dynamisch konfiguriert werden können. Die Klassen und Services im inneren der Abstraktionsschicht sollen ohne Änderung der übrigen Strukturen hinzugefügt werden können, um das Framework für zukünftige Hardware und Kommunikationsprotokolle (z.B. FlexRay) erweiterbar zu machen.
Die Funktionalität des erstellten Kommunikationsframeworks soll durch die exemplarische Implementierung spezieller Kanäle (insbesondere dem MOST Kontrollkanal) demonstriert und nachgewiesen werden.

1.3 Aufbau und Struktur der Arbeit

In Kapitel [2] der Arbeit wird zunächst eine Einführung in bestehende Technologien gegeben, welche für die weitere Bearbeitung wichtig sind.
Im ersten Teil dieses Kapitels [2.1] werden die Standards CAN [2.1.1], MOST [2.1.2] und FlexRay [2.1.3] beschrieben und für den Einsatz im Car-Infotainmentbereich klassifiziert. Hierbei sollen sowohl die vorhandenen Steuergeräte, als auch die eingesetzten Kommunikationsprotokolle untersucht werden.
Der zweite Teil des Kapitels [2.2] soll eine Einführung in die Modellierung mit der Unified Modelling Language (UML) geben, genauer gesagt in „Model Driven Architecture using executable UML“. Hier werden insbesondere Statecharts betrachtet, welche in erster Linie zur Modellierung und Codegenerierung bei dem im Rahmen dieser Diplomarbeit eingesetzten Testsystem MODENA verwendet werden.
Im dritten und letzten Teil dieses Kapitels [2.3] werden dann die von Berner & Mattner entwickelten Testsysteme MODENA [2.3.1] und RAVENNA [2.3.2] beschrieben.
Das nächste Kapitel [3] widmet sich dem entwickelten Kommunikationsframework – dem Data Abstraction Layer (DAL). Zunächst [3.1] werden die Anforderung an diese Mittelschicht und ihre Schnittstellen zur Applikation (MODENA) und zu den eingesetzten Bussystemen (insbesondere MOST) formuliert. Anschließend [3.2] werden zunächst verschiedenen Architekturen zur Erstellung eines Frameworks vorgestellt und auf ihren Einsatz für das zu erstellende Kommunkationsframeworks hin untersucht. Im letzten Teil dieses Kapitels [3.3] wird dann das konzeptuelle System Design der DAL vorgestellt.
Im vierten Kapitel schließlich wird die konkrete Systemarchitektur [4.1] und Implementierung [4.2] des Kommunikationsframeworks anhand einer Beispielkonfiguration angegeben.
Eine Zusammenfassung der erzielten Ergebnisse [5.1], sowie ein Ausblick für den Einsatz und die weitere Entwicklung des Frameworks [5.2] schließen die Arbeit ab.

1.4 Die Firma Berner & Mattner Systemtechnik

Die Berner & Mattner Systemtechnik GmbH (www.bms.de) konzentriert sich auf Dienstleistungen und Entwicklungswerkzeuge für die Software- und Systementwicklung in den Bereichen Automobil-Elektronik und Schienenverkehrssysteme.
Die Entwicklungswerkzeuge zur modellbasierten Software- und Systementwicklung verfolgen das Ziel komplexe Softwaresysteme strukturierter, effizienter und mit höherer Qualität herzustellen.

Berner & Mattner wurde 1979 in Ottobrunn bei München gegründet und besitzt inzwischen weitere Niederlassungen in Ingolstadt und Stuttgart. Die Firma beschäftigt insgesamt ca. 130 Mitarbeiter.
Zu den Kunden zählen namhafte Unternehmen aus der Automobil und Zulieferindustrie, wie etwa Audi, BMW, Daimler Chrysler oder Siemens.

2 Einführung in bestehende Technologien

2.1 Elektronik im Automobilbereich - Feldbusse

Durch die wachsende Anzahl elektronischer Komponenten (Steuergeräte, Sensoren und Aktoren) wurde die herkömmlich verwendete Verkabelung (die sogenannten “Kabelbäume”) zunehmend unwirtschaftlich und allein durch die Vielzahl der benötigten Kabel kaum mehr zu realisieren. Als Folge wurden spezielle Bussysteme für den Einsatz im Automobil entwickelt. Dabei waren neben den bereits aus der Informatik bekannten Auswahlfaktoren (wie Bandbreite, Störsicherheit oder die maximal mögliche Anzahl addressierbarer Knoten), sowie den kaufmännischen Kostenaspekten, zusätzlich automobilspezifische Kriterien (insbesondere die Elektromagnetische Verträglichkeit (EMV)) zu beachten.
Die SAE (Society of Automotive Engineering) entwickelte 2002 daraufhin eine Klassifizierung zur Unterscheidung der eingesetzten Systeme. Kriterien dabei waren unter anderem Kosten und Übertragungsgeschwindigkeiten. [RAN-02]

Class A: bis 10kbit/s, Knotenpreis US $ 4
Vertreter: LIN, TTP/A, J1850
(Klasse der Subbus-Systeme)

Class B: 10kbit/s bis 100kbit/s, Knotenpreis: US $ 5
Sicherheitsrelevante Applikationen mit Fehlertoleranz
Vertreter: Powertrain, x-by-wire, TTP/B, Byteflight, (TT-)CAN

Class C: 100kbit/s bis 1 Mbit/s, Knotenpreis: US $ 10
Verteile Echtzeitsysteme, Multimedia
Vertreter: MOST, D2B

Busse, die mehrere Klassen abdecken, werden mit dem entsprechenden Suffix „/x“ gekennzeichnet. Beispiel: “CAN/B” für Low Speed CAN und “CAN/C” für High Speed CAN.

Es existieren heute mehrere Techologien (Bussysteme, Protokolle) die sich im Laufe der letzten Jahre als Standards für die Integration elektrischer Geräte im Automobil etabliert haben.

In den folgenden Unterkapiteln werden drei dieser Technologien genauer vorgestellt, welche zur Vernetzung und Interaktion von (Infotainment-) Steuersystemen eingesetzt werden.

2.1.1 Controller Area Network (CAN)

Der Begriff Controller Area Network (CAN) bezeichnet ein von Bosch Anfang der achtziger Jahre entwickeltes echtzeitfähiges serielles Bussystem [BOS-01], welches 1994 international standardisiert wurde (ISO 11898). CAN wurde zunächst speziell für den schnellen seriellen Datenaustausch zwischen elektronischen Steuergeräten in Kraftfahrzeugen entwickelt. Dieser Standard hat sich darüber hinaus auch in der Automatisierungs- und Fertigungstechnik, beispielsweise als interner Bus von Werkzeugmaschinen, etabliert.

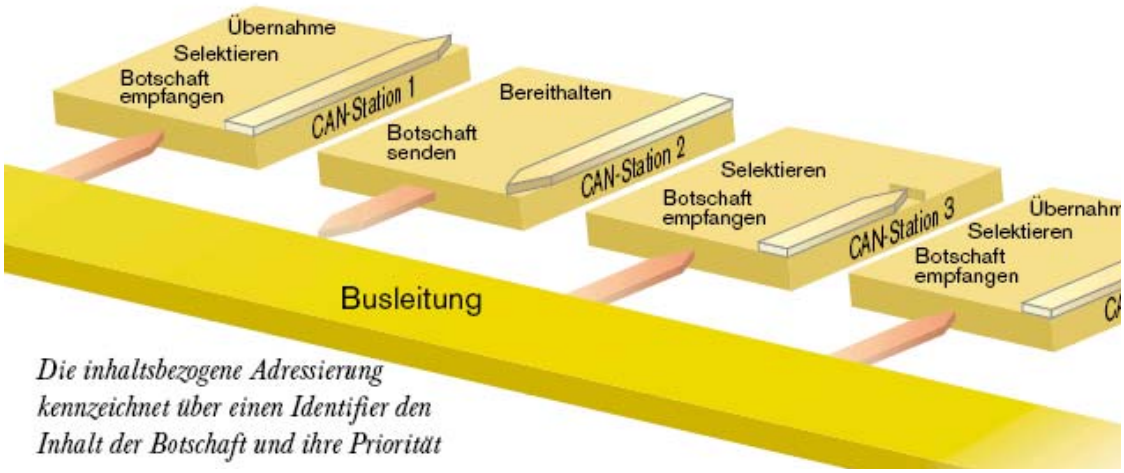


Abbildung 2: CAN-Busleitung mit Stationen (Sender und Empfänger) [BOS-02]

Der CAN-Datenbus sorgt für einen digitalen Datenaustausch zwischen Sensoren, Aktoren sowie Steuergeräten und gewährleistet, dass mehrere Steuergeräte die Informationen eines Sensors verarbeiten und ihre Aktoren entsprechend ansteuern können. Neben kurzen Leitungswegen ist ein besonderer Vorteil von CAN, dass bei Ausfall einer Komponente das übrige System weiterhin arbeitet und somit das Risiko eines Gesamtausfalls deutlich verringert wird. Auch altbekannte elektrische Systeme werden vermehrt über den CAN-Datenbus gesteuert und verbessern Sicherheit und Komfort.
Wegen der unterschiedlichen Wiederholrate der Signale und des entstehenden Datenvolumens werden CAN-Bus-Systeme in drei Kategorien untergliedert: Der CAN-Bus-Antrieb überträgt unter anderem die Signale des Motorsteuergerätes, der

Getriebesteuerung und der ABS/ESP-Einheit. Der CAN-Bus-Komfort wird beispielsweise für die Klimasteuerung eingesetzt und der CAN-Bus-Infotainment für die Signale von Multimediakomponenten wie etwa dem Autoradio.

Der CAN-Bus hat eine Linientopologie, bei der verdrillte Zweidrahtleitungen als Übertragungsmedium verwendet werden. Die maximale Ausdehnung eines CAN-Netzwerks (bei einer Übertragungsrate von bis zu 1 Mbit/s) liegt bei ca. 40 Metern.

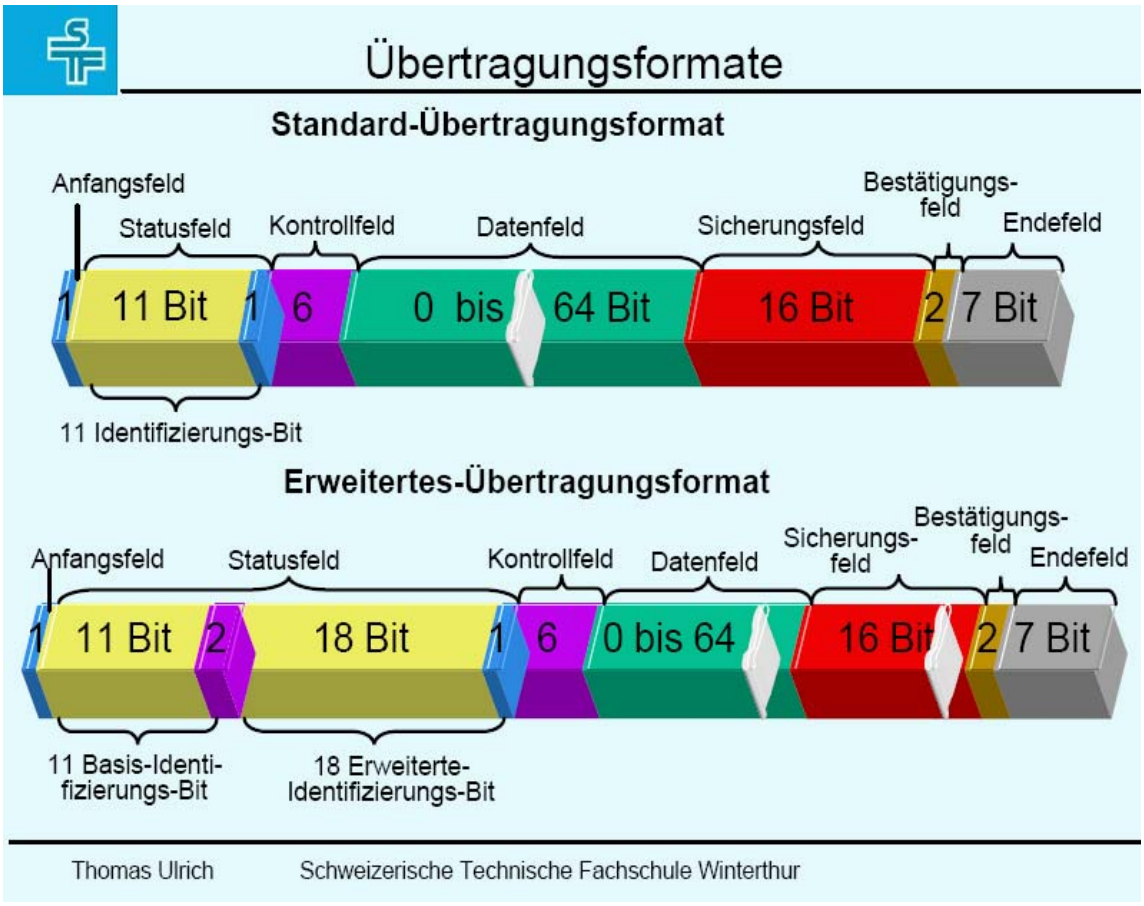


Abbildung 3: CAN-Übertragungsformate (DATA-FRAME) [ULR-01]

In der CAN-Spezifikation [INT-01] werden für den Nachrichtentransfer vier verschiedene Rahmentypen definiert. Neben dem DATA FRAME, in dem Nutzdaten vom Sender zum Empfänger übertragen werden, gibt es den REMOTE FRAME (zum Anfordern des entsprechenden DATA FRAMES), sowie einen ERROR FRAME (zur Übermittlung eines erkannten Fehlers) und einen OVERLOAD FRAME. Letzterer kann verwendet werden, um zwischen DATA und/oder REMOTE FRAMES eine zusätzliche Verzögerung einzufügen. DATA und REMOTE FRAMES sind identisch aufgebaut, außer dass es beim REMOTE FRAME kein Datenfeld gibt.

Auf dem Bus wird jeweils immer einer der beiden komplementären logischen Werte “dominant” (0) oder “recessive” (1) übertragen. Bei gleichzeitiger Übertragung beider Werte setzt sich immer “dominant” durch.

Die Abbildung auf der letzten Seite zeigt den inneren Aufbau eines DATA FRAMES, einmal für das Standardformat (Spezifikation 2.0A) und einmal für das erweiterte Format (Spezifikation 2.0B).

Die zu übertragenden Nachrichten werden über eine Nachrichtenkennung (Identifizierung) eindeutig gekennzeichnet. Im Gegensatz zur Teilnehmeradressierung wird bei CAN dabei kein Steuergerät, sondern die Nachricht selbst adressiert. Dadurch steht eine Nachricht grundsätzlich jedem Busteilnehmer zum Empfang zur Verfügung. Ob eine Nachricht angenommen wird, hängt einzig von der Entscheidung der einzelnen Steuergeräte ab (empfängerselektives System). Diese Entscheidung wird über Nachrichtenfilter durchgeführt.
Im Standardformat umfasst die Identifizierung 11 Bits. Somit können in einem CAN-Netzwerk 2048 verschiedene Nachrichten voneinander unterschieden werden. Bei dem erweiterten Format stehen weitere 18 Bits für die Nachrichtenadresse zur Verfügung. Damit können über 536 Millionen CAN-Botschaften spezifiziert werden.
Das letzte Bit des Statusfeldes entscheidet darüber, ob es sich bei dem Rahmen um ein DATA oder ein REMOTE FRAME handelt. Im Kontrollfeld wird die Anzahl der folgenden Datenbytes angegeben. Bei einem REMOTE FRAME gilt diese Größe für das Datenfeld des angefragten DATA FRAMES. Das Sicherungsfeld (CRC Feld) dient dazu Übertragungsstörungen zu erkennen und im Bestätigungsfeld signalisieren die Empfänger einer Nachricht dem Sender, dass sie die Nachricht korrekt empfangen haben. Das Ende eines Rahmens wird durch sieben “recessive“ Bits gekennzeichnet.
CAN basiert auf einer Multi-Master-Architektur, dass heißt alle angeschlossenen Steuergeräte sind gleichberechtigt und können aus eigener Kraft das gemeinsame Übertragungsmedium benutzen. Somit ist jedes Steuergerät in der Lage, bei Bedarf (zum Beispiel dem Eintreten eines Ereignisses) eine entsprechende CAN-Nachricht abzuschicken. Dabei kann ein Gerät nur senden, wenn der Bus frei ist (CSMA/CD-Prinzip). Der Zugriff auf den CAN-Bus wird über Prioritäten gesteuert. Wenn mehrere Steuergeräte gleichzeitig versuchen eine Nachricht auf den Bus zu schicken, setzt sich die Nachricht mit der höchsten Priorität (niedrigster Identifier) durch. Die Identifizierung der Nachricht enthält dabei jeweils auch die Knotenposition des Steuergerätes, damit auch

eine Sendeentscheidung getroffen werden kann falls mehrere Geräte die gleiche Nachricht senden wollen. Eine Nachricht mit der ID 0x700 wird demnach von einem Steuergerät mit Knotennummer 3 als 0x703 gesendet.

Mit TTCAN (Time-Triggered CAN) wurde zudem eine Erweiterung zur zeitgesteuerten Kommunikation entwickelt, die auf dem Standard CAN-Protokoll aufsetzt. Diese ermöglicht es Nachrichten nicht nur ereignisgesteuert über das CAN-Netzwerk zu schicken, sondern auch zu vorher festgelegten Sendezeitpunkten. Ein solches Kommunikationssystem eignet sich insbesondere für Anwendungen, in denen ein Grossteil des Nachrichtenverkehrs periodischer Natur ist. Die Synchronisation der einzelnen Steuergeräte übernimmt das TTCAN-Protokoll durch das Senden von Referenznachrichten, bei deren Empfang in jedem angeschlossenen Gerät ein Zähler neu gestartet wird. Eine Kombination von zeit- und ereignisgesteuerten Nachrichten lässt sich hiermit ebenfalls realisieren. Zum Beispiel können die Referenznachrichten, welchen eine zeitlich festgelegte Abfolge von Nachrichten folgt, auch durch ein Ereignis ausgelöst werden.

2.1.2 Media Oriented Systems Transport (MOST)

Die MOST-Cooperation [INT-02] wurde 1998 von BMW, Daimler Chrysler, Becker Radio und OASIS Silicon Systems mit der Zielsetzung gegründet, die MOST Technologie zu standardisieren. 1999 entstand mit dem „MOST Specification Framework“ die Version 1.1 des MOST-Busses und damit die Grundlage für die aktuell verwendeten Bussysteme auf MOST-Basis. Inzwischen haben sich mehr als 70 Fahrzeughersteller und Hersteller von Multimediakomponenten der MOST-Cooperation angeschlossen.
Die MOST-Spezifikation [MOS-05] schreibt zwar keine bestimmte Vernetzungstopologie vor, allerdings wird vorwiegend eine Ringtopologie verwendet (wie in der folgenden Abbildung gezeigt).

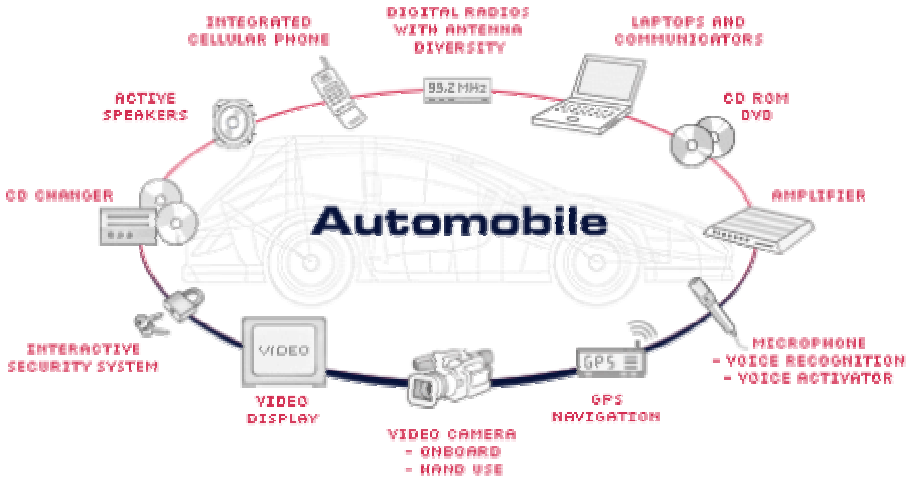


Abbildung 4: MOST Ringtopologie [INT-02]

Die Vorteile dieser Vernetzung liegen bei den relativ niedrigen Kosten aufgrund nur weniger benötigter Verbindungsleitungen. Außerdem können Nachrichten so leicht allen angeschlossen Geräten zur Verfügung gestellt werden. Der größte Nachteil einer Ringtopologie ist, dass der Ausfall eines Knotens zum Ausfall des gesamten Netzwerkes führt. Um dies zu vermeiden kann jeder MOST Transceiver (FOT) in einen sogenannten Bypass-Mode geschaltet werden, in dem er für das übrige Netz unsichtbar wird. Ein ausgefallenes Gerät schaltet seinen Transceiver automatisch in diesen Mode um, damit der Ring nicht unterbrochen wird.
Über den MOST-Bus können maximal 64 Knoten adressiert werden, bei einer Bandbreite von 24,8 Mbps (3,1 Mbyte/s). Als Übertragungsmedium werden Lichtwellenleiter, sogenannte Plastical Optical Fiber (POF), eingesetzt. Der größte Vorteil dieser

Informationsübertragung liegt darin, dass sie resistent ist gegenüber elektromagnetischen Störungen, welche im Automobil verstärkt auftreten.
Der Hauptnachteil der beschränkten maximalen Entfernungen von etwa 250m spielt dagegen hier keine Rolle.
Die MOST Spezifikation verwendet den Begriff “Frame” für einen kompletten Kommunikationszyklus, der immer eine feste Länge von 512 Bits hat. Dies entspricht einem Zeitinterval von 22,67 µs bei einer üblicherweise verwendeten Frequenz von 44,1 kHz (= Samplingrate bei CDs). Jeweils 16 solcher Frames werden zu einem Block zusammengefasst, somit wird pro Block genau eine komplette Kontrollnachricht übertragen.

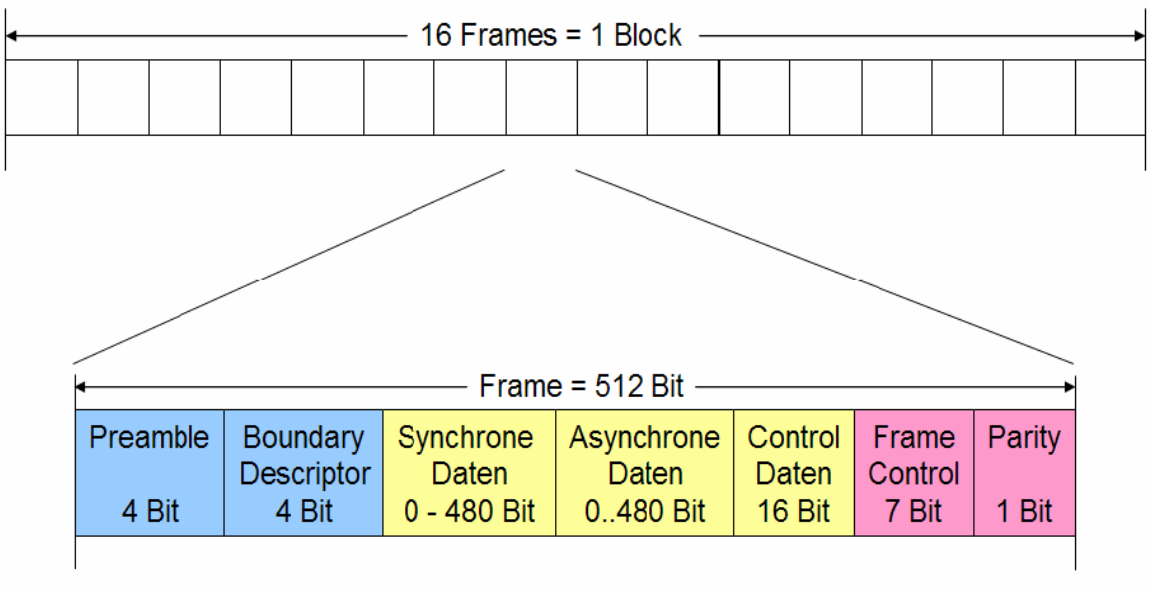


Abbildung 5: MOST Kommunikations-Frame [DOH-02 Seite 13]

Die Preamble zu Beginn eines jeden Frames dient zur Synchronisation zwischen Master und Slave Knoten. Über den Boundary Descriptor kann die Länge der nachfolgenden Datenbereiche für synchrone und asynchrone Daten angegeben werden. Die beiden Felder zusammen belegen immer 60 Bytes, wobei die Grenze dynamisch zur Laufzeit verschoben werden kann, um die Frames dem aktuellen Verkehr anzupassen. Diese Verschiebung findet immer in 4 Byte Schritten (sogenannten Quadlets) statt.
Der synchrone Datenkanal wird zur Übertragung von Realzeit Daten wie zum Beispiel Audio- und Video-Streams oder auch Sensordaten verwendet. Die Zuteilung der einzelnen Zeitschlitze erfolgt mittels Time Division Multiplexing (TDM), wobei die jeweils benötigte Bandbreite von den einzelnen Anwendungen angefordert werden muss (channel allocation). Die maximale Anzahl von 60 Bytes pro Frame entspricht 15 Audiokanälen

(stereo) in CD Qualität. Die Kanalzuteilung wird auf Applikationsebene vom Connection Manager abgewickelt.
Im asynchronen Kanal können größere Nachrichtenpakete, zum Beispiel Bilder und Grafiken für Navigationssysteme, übertragen werden. Die maximale Größe eines Paketes beträgt 48 oder 1014 Bytes je nach verwendeter Verbindungsschicht. Das bedeutet, dass asynchrone Nachrichten auch über mehrere Frames verteilt sein können. Der Zugriff auf den asynchronen Kanal wird über ein Token-Ring Verfahren realisiert, bei dem die Knoten nacheinander jeweils ein Paket senden dürfen. Die Nachrichten enthalten neben den eigentlichen Daten die Sender- und Empfängeradresse der jeweiligen Knoten, die Länge des Datenfeldes, sowie CRC Prüfsummen.
Der Kontrollkanal schließlich dient zur (asynchronen) Übermittlung von Kommunikationsnachrichten zwischen den einzelnen Knoten.

Arbitration	Target	Source	Type	Data	CRC	Status	Reserved
4 Bytes	2 Bytes	2 Bytes	1 Byte	17 Bytes	2 Bytes	2 Bytes	2 Bytes

Abbildung 6: Control Data Frame

Der Zugriff auf diesen Kanal wird wie bei CAN über das CSMA Verfahren realisiert. Der Typ gibt an, ob es sich um eine (normale) anwendungsbezogene Nachricht oder um eine Systemnachricht (etwa zur Resourcenverwaltung) handelt.
Die letzten acht Bits eines Frames (Frame Control und Parity, Abbildung 5) dienen zur Fehlererkennung.
Die Abbildung auf der folgenden Seite zeigt den Schichtenaufbau eines MOST Devices. Der MOST Network Interface Controller (NIC) wickelt die reale Kommunikation zwischen den verschiedenen Geräten über das Netzwerk ab.
Aus Anwendungssicht enthält ein MOST Device mehrere Komponenten, so genannte Function Blocks. Diese repräsentieren spezielle Anwendungen innerhalb des jeweiligen Gerätes, zum Beispiel für die Steuerung eines CD Players (FBlock AudioDiskPlayer). Innerhalb eines solchen Function Blocks werden mehrere Funktionen bereitgestellt, auf die über ein Function Interface von außen zugegriffen werden kann. Der spezielle Function Block NetBlock, der in jedem Device vorhanden ist, stellt Funktionen zur Verfügung die das gesamte Device betreffen (etwa die Adresse eines Knotens). Der NetworkMaster Block, der in jedem MOST Netzwerk genau einmal (im Masterknoten) vorkommen muss, verwaltet die logischen und physikalischen Adressen aller Knoten des Netzwerkes und ist somit für die eindeutige Addressierbarkeit verantwortlich.

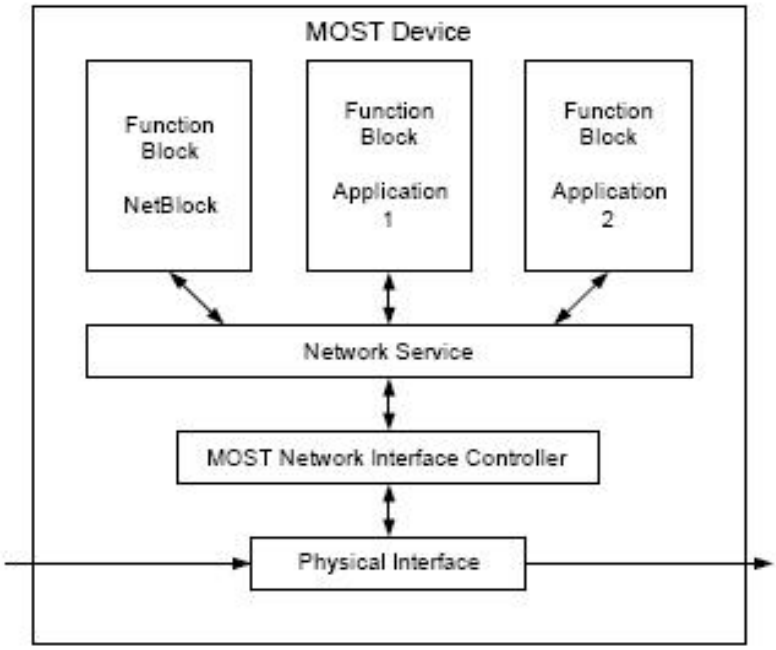


Abbildung 7: MOST Device [MOS-05, Seite 23]

Bei den Funktionen wird zwischen Methoden und Properties / Events unterschieden. Methoden können gestartet werden und führen zu einem Resultat. Ein Beispiel hierfür wäre die Zuweisung (Allocation) eines Kanals zu einem Sender- oder Empfängerknoten. Properties dagegen sind Zustandsvariablen deren Werte (Status) durch festgelegte Operationen gelesen und/oder geschrieben werden können (etwa die aktuelle Position einer CD). Damit Änderungen von Properties nicht regelmäßig abgefragt werden müssen, kann sich ein Device in die Notifikationsmatrix eines Function Blocks eintragen lassen, über die Geräte automatisch über geänderte Werte der entsprechenden Properties informiert werden. Hierzu verschickt das betroffene Device ein Event mit dem neuen Status seines geänderten Properties an alle eingetragenen Knoten.

Die mittlere Schicht in Abbildung 7 (Network Service) stellt eine Softwareschnittstelle zum MOST Netzwerk zur Verfügung. Hier werden die verschiedenen Transportlayer realisiert, in denen die Nachrichten aus dem Transceiver für die Applikation entsprechend aufbereitet werden.

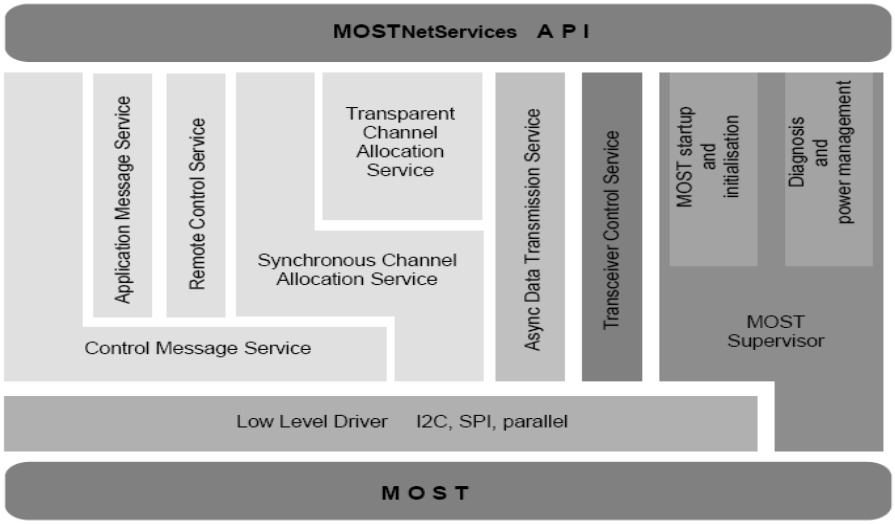


Abbildung 8: MOST Net Services – Basic Layer [MOS-06]

Die NetServices bestehen ihrerseits wiederum aus zwei Schichten. Der Basic Layer (Abbildung 8) stellt neben low-level Treibern zum direkten Zugriff auf den MOST Bus spezielle Services für den Zugriff auf die einzelnen MOST Datenkanäle zur Verfügung.
Die Control Messages Services (CMS) etwa dienen zum Senden und Empfangen von Kontrollnachrichten und realisieren zudem die Zwischenspeicherung und Fehlerbehandlung der Nachrichten. Der Application Message Service (AMS) wiederum setzt auf CMS auf und ermöglicht die Segmentierung von Kontrollnachrichten auf mehrere Frames.
Die zweite Schicht, der Application Socket Layer, beeinhaltet unter anderem den MOST Command Interpreter zur Interpretation der empfangenen Nachrichten. Außerdem enthält sie den Function Block NetBlock, der in allen MOST Steuergeräten vorhanden sein muss, sowie ein dezentrales Deviceregister und die Notification Services zur automatischen Benachrichtigung eingetragener Steuergeräte.
Zusätzlich zu den beiden oben beschriebenen Schichten wird noch das MOST High Protocol zu den NetServices gezählt. Dieses verbindungsorientierte Protokoll kann zur Verwaltung des asynchronen Kanals verwendet werden und benutzt einige Mechanismen von TCP.

2.1.3 FlexRay

Um die gewachsenen Anforderungen an deterministische Echtzeit-Anwendungen im Automobilbereich (z.B. X-By-Wire) auch zukünftig erfüllen zu können, wurde 1999 das FlexRay – Konsurtium [INT-03] von BMW, Daimler Chrysler, Motorola und Philips gegründet, dem sich inzwischen weitere Automobil- und Zuliefererunternehmen angeschlossen haben.
FlexRay Knoten kommunizieren über zwei physikalisch getrennte (optische oder elektrische) Kanäle mit einer Datenübertragungsgeschwindigkeit von jeweils 10 Mbits/s. Die beiden Kanäle werden hauptsächlich für die redundante und daher fehlertolerante Nachrichtenübertragung verwendet. Es können aber auch unterschiedliche Nachrichten übertragen werden, in diesem Fall verdoppelt sich der Datendurchsatz.

Zur Verbindung der Steuergeräte wurde eine Topologie festgelegt, die aus Punkt-zu-Punkt Verbindungen mit aktiven Sternen besteht (Abbildung 8). Dabei dürfen maximal drei Sterne auf dem Weg zwischen zwei beliebigen Knoten liegen. Die Vorteile dieser Vernetzung liegen in den minimierten Reflexionen durch ideale Bus-Abschlüsse, sowie der flexiblen Erweiterbarkeit und möglichen Abkopplung fehlerhafter Zweige.

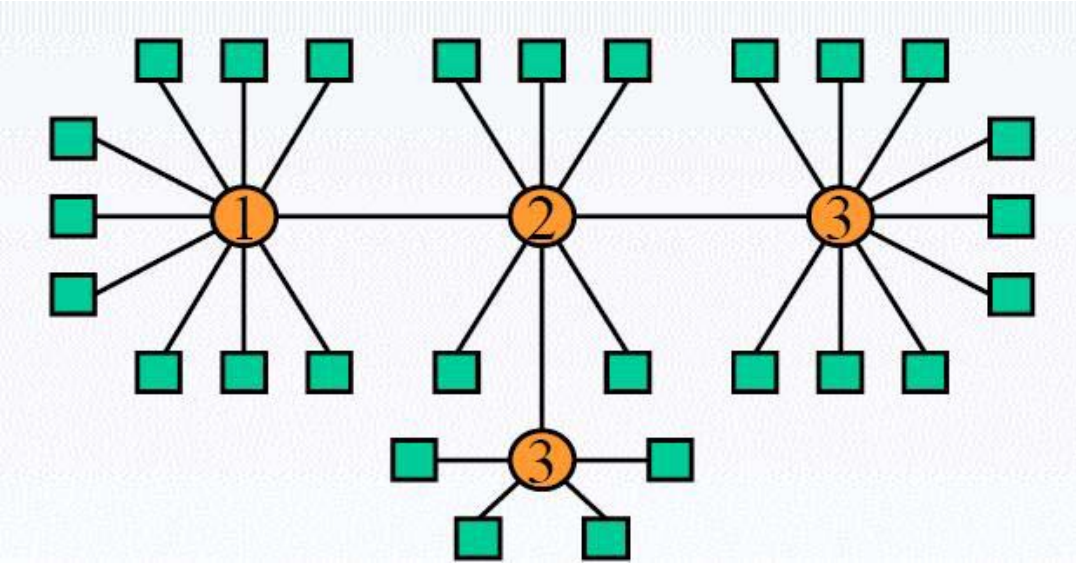


Abbildung 9: FlexRay Verbindungen [FLE-03, Seite 23]

Die aktiven Sterne sind für die Weiterleitung der Nachrichten, sowie die Verwaltung der verschiedenen Knotenmodi (Normal, Standby, Sleep) zuständig. Außerdem werden hier

zusätzlich Methoden zur Fehlereindämmung, etwa die Abkapselung gestörter Punkt-zu-Punkt Verbindungen realisiert (Bus Guardian).

Zukünftige Fahrwerksanwendungen werden vermehrt als verteilte Regelsysteme ausgeführt. Solche Systeme verlangen meist eine zyklische, zeitsynchrone Übertragung. Andere nicht ständig benötigte Daten, zum Beispiel Diagnosedaten, lassen sich dagegen vorteilhafter asynchron übertragen.

Um ein deterministisches Verhalten zu ermöglichen, ist bei FlexRay das Übertragungsschema in Zyklen organisiert. Jeder Kommunikationszyklus gliedert sich in ein statisches und ein dynamisches Segment von konfigurierbarer Länge (Abbildung 9). Innerhalb des statischen Segments können jedem Netzknoten bestimmte Zeitfenster (Timeslots) zugeteilt werden, in denen die Nachrichten des Teilnehmers zu festen Zeitpunkten übertragen werden. Die definierten Sendezeiten der zugewiesenen Zeitfenster garantieren eine deterministische Übertragung der Daten. Im dynamischen Segment erfolgt die Zuteilung der vorhandenen Bandbreite prioritätsgesteuert, damit Nachrichten mit hoher Priorität garantiert innerhalb eines Kommunikationszyklus übertragen werden können, während sich für Nachrichten mit niedriger Priorität die Sendung verzögern kann.

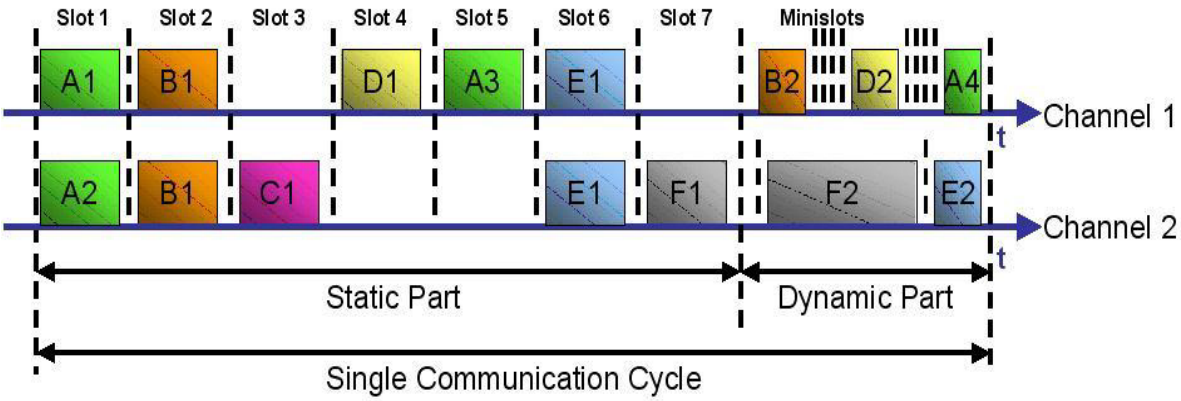


Abbildung 10: FlexRay Kommunikationszyklus [INT-04]

Echtzeit-relevante und zeitkritische Nachrichten werden vorzugsweise im statischen Segment übertragen. Das dynamische Segment ist dagegen besser geeignet für die Übertragung von Daten mit geringeren Echtzeit-Anforderungen. Da diese Daten nicht in jedem Kommunikationszyklus übertragen werden müssen, können die Netzknoten die zur Verfügung stehende Bandbreite im dynamischen Segment gemeinsam nutzen. Aufgrund

der flexibel festlegbaren Grenze zwischen statischem und dynamischem Segment ist auch ein rein statischer bzw. rein dynamischer Betrieb möglich.

Indicators	Frame ID	Length	Header CRC	Cycle Count	Data	Frame CRC
3 Bits	12 Bits	7 Bits	11 Bits	6 Bits	0 - 254 Bytes	24 Bits

Abbildung 11: FlexRay Frame

2.2 Modellierung mit UML Diagrammen (executable UML)

Die Unified Modeling Language (UML) [UML-20] ist ein Standart der Object Management Group, in dem die Notation und Semantik von Diagrammen zur Visualisierung, Konstruktion und Dokumentation für die Geschäftsprozessmodellierung und die objektorientierte Softwareentwicklung definiert werden.
In der aktuellen Version 2.0 (Einführung 2004) werden insgesamt 13 verschiedene Diagrammtypen zur Darstellung von Objekten (Klassen) und ihren Beziehungen unterschieden. Die einzelnen Diagrammtypen lassen sich wie folgt den zwei Hauptkategorien Struktur- und Verhaltensdiagramme zuordnen.

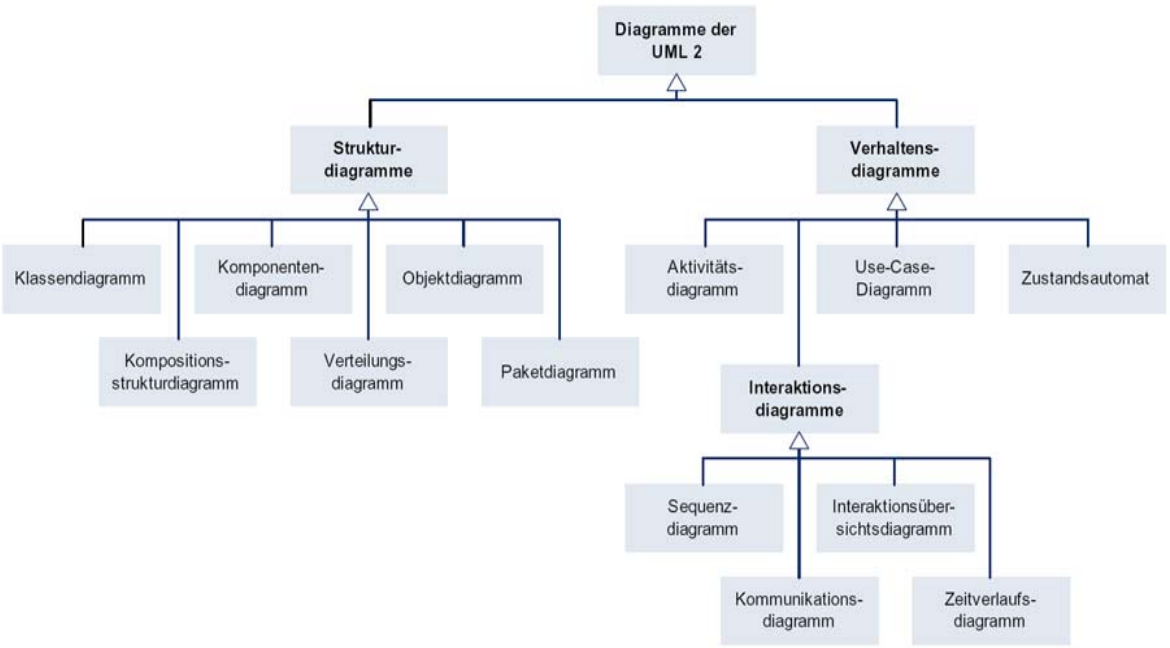


Abbildung 12: UML Diagrammtypen [INT-05]

In diesem Kapitel sollen UML Zustandsdiagramme (Statecharts) genauer beschrieben werden, welche bei den von Berner & Mattner entwickelten Testsystemen (siehe Kapitel 2.3) zur Verhaltensmodellierung von Automobil-Steuergeräten eingesetzt werden. Darüber hinaus werden im weiteren Verlauf dieser Arbeit UML Klassen- und Sequenzdiagramme für die Konzeptionierung des zu entwickelnden Kommunikationsframeworks verwendet, welche daher hier ebenfalls zunächst kurz erläutert werden.

Klassendiagramme sind statische Strukturdiagramme, in denen die im System vorkommenden Objekte und deren Beziehung zueinander dargestellt werden können. Gleichartige Objekte (gekennzeichnet durch gemeinsame Attribute und Funktionalitäten)

werden hierbei zu Klassen zusammengefasst. Außerdem können durch Generalisierung beziehungsweise Vererbung hierarchische Beziehungen zwischen den einzelnen Klassen abgebildet werden.

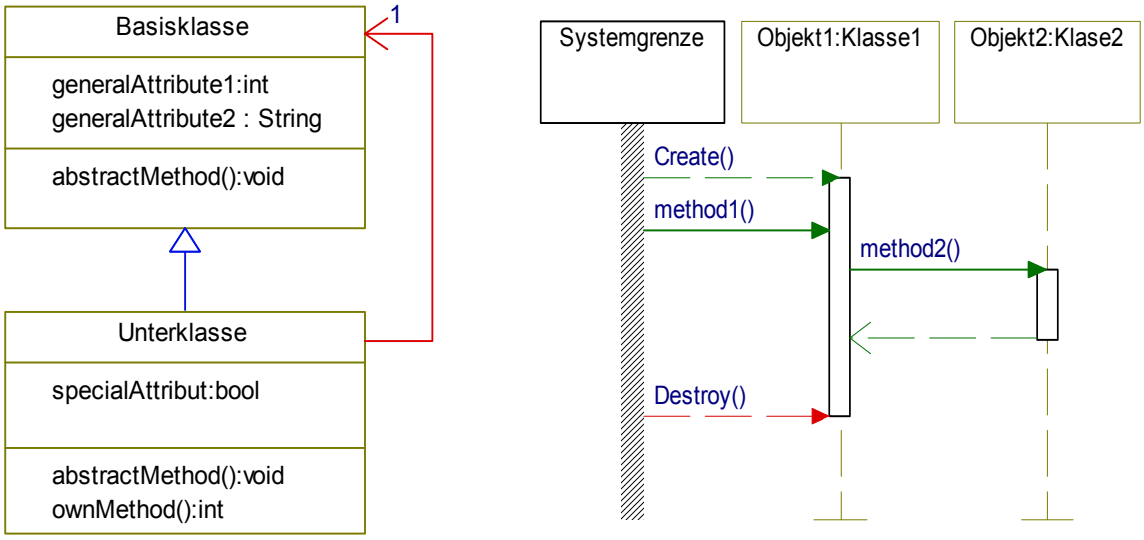


Abbildung 13: UML Klassen- und Sequenzdiagramm

Im linken Teil der obigen Abbildung ist ein beispielhaftes Klassendiagramm dargestellt. Es besteht aus einer (abstrakten) Basisklasse, die zwei generelle Attribute bereitstellt und eine abstrakte Methode definiert. Die davon abgeleitete Unterklasse muss die geerbte Methode implementieren und kann darüber hinaus weitere Attribute und Methoden enthalten. Eine Instanz der Unterklasse steht in Beziehung zu einem anderen Objekt, welches ebenfalls von der Basisklasse abgeleitet ist.
Der rechte Teil der Abbildung zeigt ein Sequenzdiagramm, welches die zeitliche Abfolge von Interaktionen zwischen Objekten beschreibt. Zusätzlich können Systemgrenzen und externe Aufrufe modelliert werden. Objekte besitzen eine (senkrechte) Lebenslinie und können zur Laufzeit erzeugt und wieder vernichtet werden. Die Kommunikation zwischen einzelnen Objekten geschieht durch Methodenaufrufe, welche in zeitlicher Abfolge von oben nach unten notiert werden.

Zustandsdiagramme (Statecharts) bilden das diskrete Verhalten einer Instanz einer Klasse ab. Mit ihnen können alle möglichen Zustande die ein Objekt im Verlaufe seiner Existenz einnehmen kann, sowie die Übergänge zwischen den einzelnen Zuständen modelliert werden. Daher lassen sich mit Hilfe von Statecharts besonders gut spezielle Testfälle (einzelne Use Cases) aufstellen und Ist-Soll Vergleiche anfertigen.
UML Zustandsdiagramme bestehen aus Zuständen, welche ihrerseits wiederum

vollständige Zustandsautomaten repräsentieren können und Transitionen. Außerdem gibt es Symbole für ausgewiesene Start und Endzustände. Die folgende Abbildung zeigt ein beispielhaftes Zustandsdiagramm für eine Ampelsteuerung.

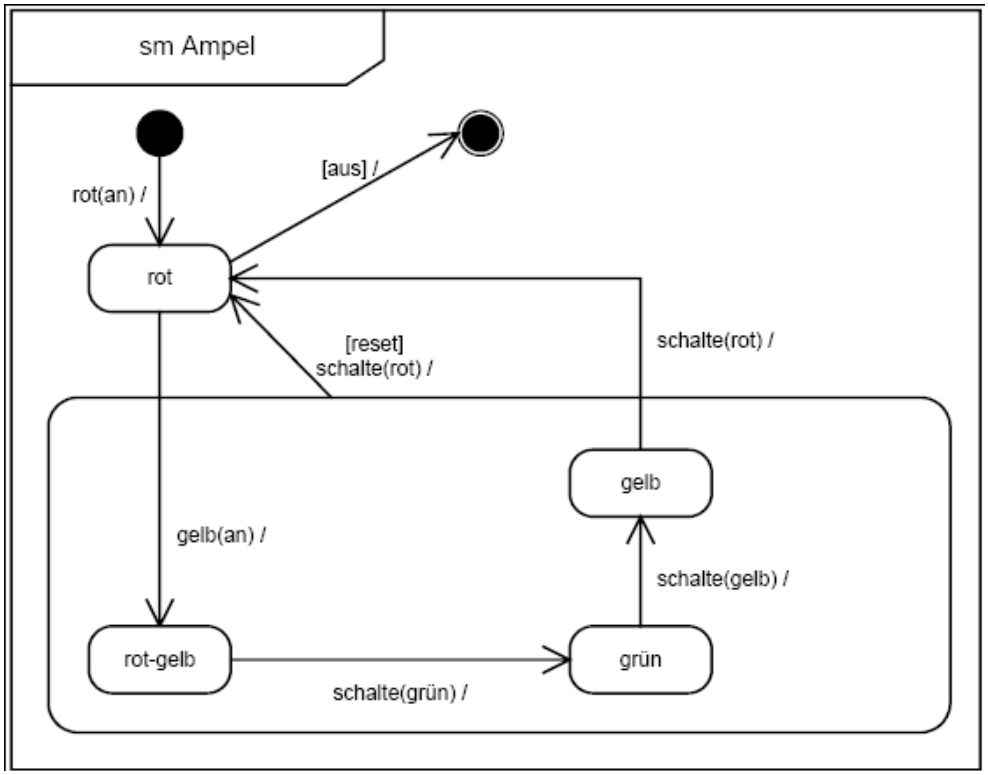


Abbildung 14: UML Zustandsdiagramm [INT-06, S.85]

Zustandsübergänge werden durch Eintritt eines Events (Trigger) ausgelöst und können zusätzlich an eine Schaltbedingung (Guard) gekoppelt sein. Außerdem kann beim Schalten einer Transition eine Aktivität ausgelöst werden. In Abhängigkeit der Schaltbedingung kann über ein Entscheidungsglied auch zwischen mehreren verschiedenen Ausgangstransitionen eine passende ausgewählt werden. Durch Gabelungen beziehungsweise Vereinigungen können zudem Transitionen auf mehrere parallele Zustände verweisen und später wieder zusammengefügt werden. Außerdem können auch Zustände ihrerseits Aktivitäten enthalten, welche zu unterschiedlichen Zeitpunkten (z.B. Eintritt / Verlassen des Zustands) ausgelöst werden.
Mit Hilfe von speziellen Entwicklungsumgebungen können die erstellten UML Diagramme direkt in ausführbaren Code übersetzt werden. Im Rahmen der Diplomarbeit wird hierzu die C++ Version von I-Logix Rhapsody (Version 6.0) [INT-11] eingesetzt.
Ein weiterer Grund für den Einsatz von UML Statecharts für die modellbasierten Testsysteme bei Berner & Mattner liegt in der relativ intuitiven und einfachen Verständlichkeit der Diagramme.

2.3 Testsysteme bei Berner & Mattner

2.3.1 MODENA

MODENA ist ein modellbasiertes Testsystem für Infotainment-Steuergeräte. Mit Hilfe von UML-Statecharts modelliert der Anwender das Sollverhalten für die Kommunikation der Steuergeräte untereinander. Nach der Erstellung von Testmodellen in UML oder C++ ermöglicht MODENA, die erstellten Modelle schon während der Spezifikationsphase anhand von Simulationen zu testen. Nach und nach werden die realen Steuergeräte in das Testsystem eingebunden und mit Hilfe der dann bereits vorhandenen und verifizierten Modelle getestet.

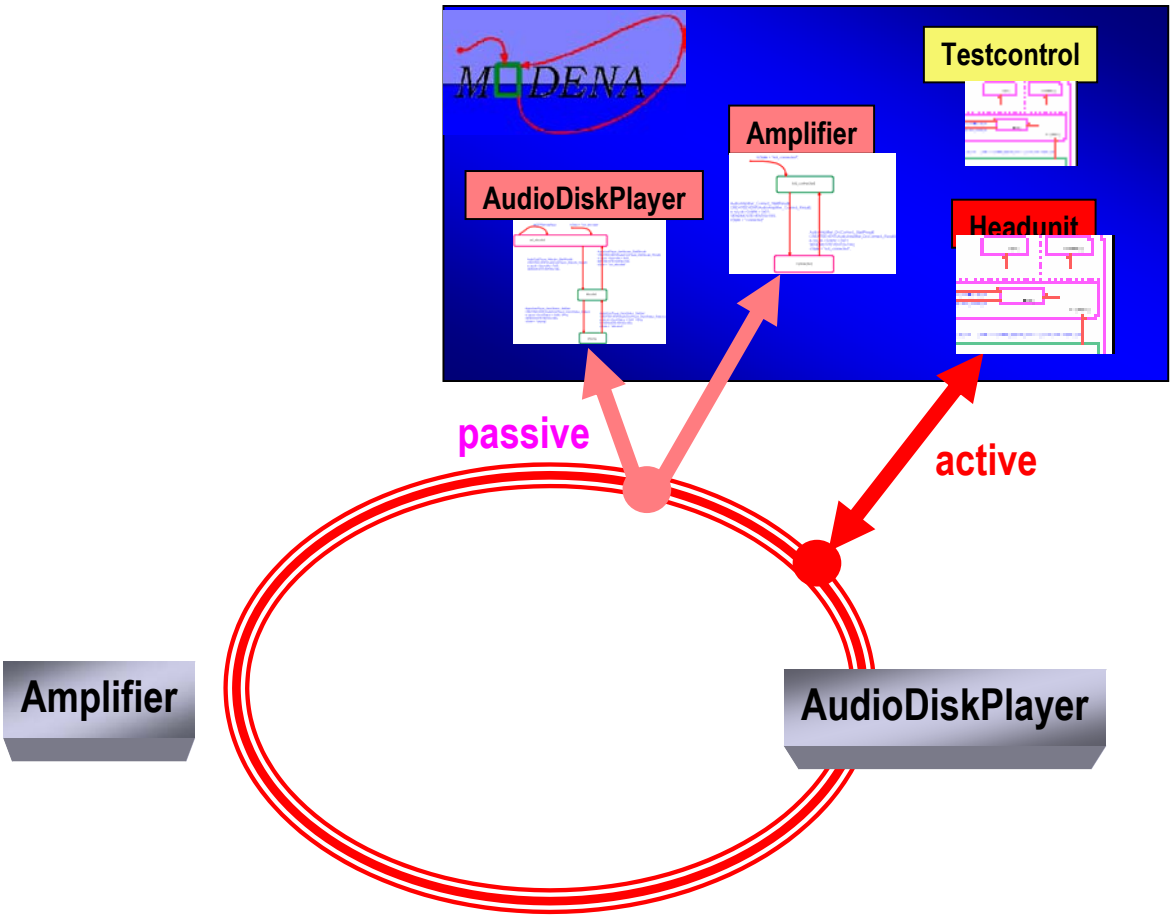


Abbildung 15: Modellbasiertes Testsystem MODENA [B&M-05]

Physikalisch nicht vorhandene Steuergeräte können dabei emuliert werden (Restbussimulation). Damit unterstützt MODENA einen durchgängigen Verifikationsprozess von der Spezifikation bis zum Integrationstest.

Bei MODENA wird zwischen Kommunikations- und Testmodellen unterschieden. Kommunikationsmodelle repräsentieren direkt ein zu testendes Steuergerät und modellieren dessen mögliche Zustände, sowie die Übergänge zwischen ihnen. Abbildung 16 zeigt ein solches Modell einer Kontrolleinheit (HeadUnit).

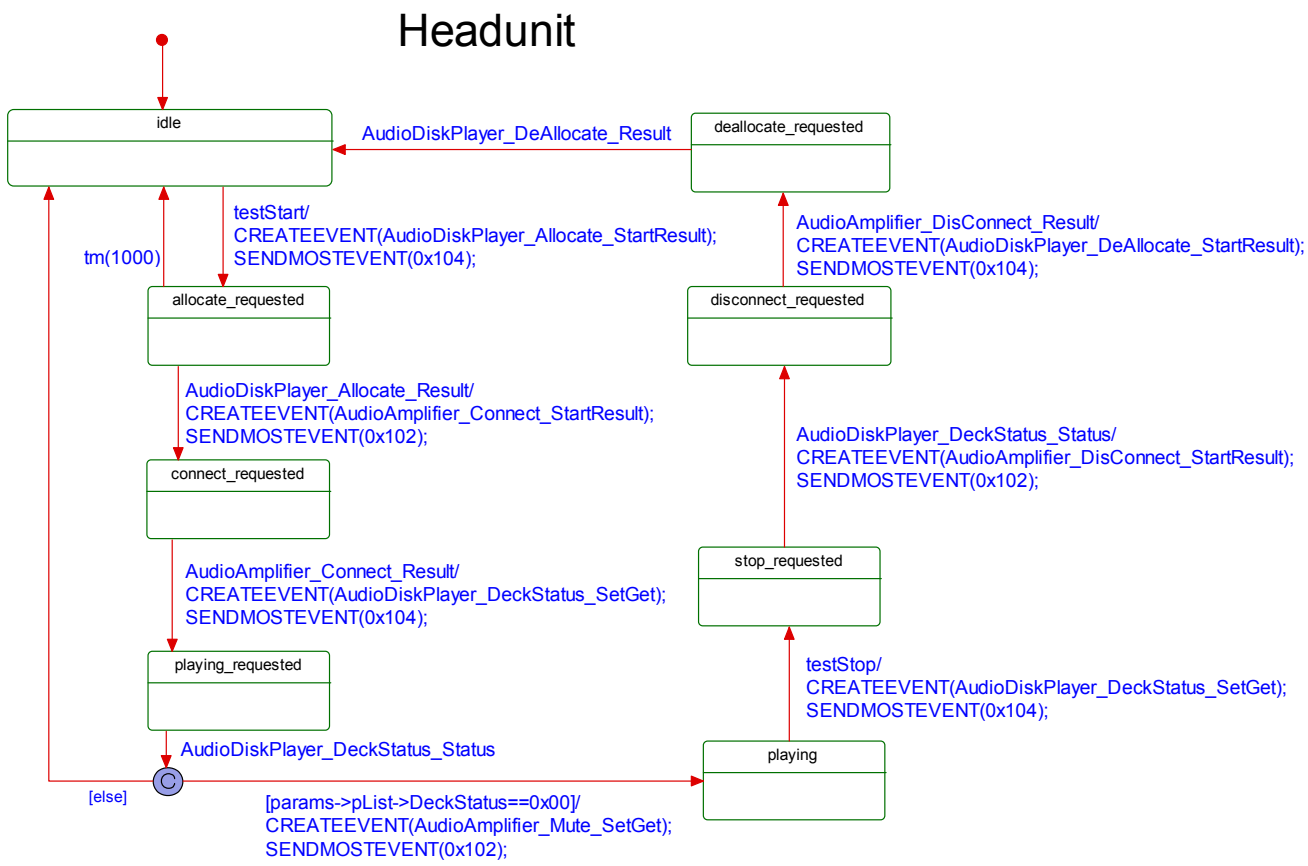


Abbildung 16: MODENA Kommunikationsmodel (HeadUnit)

In diesem Modell sind die benötigten Vorgänge zum Abspielen einer CD aus Sicht der Kontrolleinheit dargestellt. Der Testlauf wird zunächst von einem Testmodell (wird weiter unten noch genauer erläutert) gestartet. Die von diesem Testmodell geschaltete Transition (**testStart**) aus dem Ursprungszustand heraus erzeugt daraufhin ein Event zum zuweisen synchroner Kanäle für den CD Spieler (**AudioDiskPlayer_Allocate_StartResult**), welches über die verwendete Busschnittstelle an das entsprechende Gerät (logische Adress 0x104) gesendet wird. Anschließend wartet das Modell auf die Antwort des Steuergerätes mit der Bestätigung, dass die Zuweisung erfolgreich war. Fall diese Bestätigung nach einer vorgegebenen Zeit nicht eingetroffen ist, wird der Testlauf beendet. Beim Erhalt der Nachricht schaltet die nächste Transaktion und fordert den Verstärker per Event auf sich mit den vom CD Spieler belegten Kanälen zu verbinden. Wenn auch von diesem eine Bestätigung empfangen wurde

(**AudioAmplifier_Connect_Result**) geht das Modell in den Zustand (**playing requested**) über und schickt dem CD Spieler eine Aufforderung die CD abzuspielen (**AudioDiskPlayer_DeckStatus_SetGet**). Über eine Verzweigung wird geprüft, ob dieser Auftrag erfolgreich erledigt wurde und schließlich der Verstärker angewiesen die Lautstärke einzuschalten. Nun befindet sich der CD Spieler im Zustand **playing**. Konnte die CD nicht gestartet werden, so wird wiederum der Testlauf abgebrochen (else Verzweigung). Soll der Test anschließend wieder beendet werden, muss vom Testmodell das Event **testStop** erzeugt werden. Daraufhin wird die CD angehalten, sowie die Kanalbelegung der beiden angeschlossenen Steuergeräte (CD Spieler und Verstärker) wieder freigegeben (**DisConnect** bzw. **DeAllocate**).
Die beiden genannten Events zum starten und beenden des Testlaufes werden von der Anwendung beziehungsweise direkt vom Benutzer mit Hilfe der zweiten Kategorie von Modellen bei MODENA, den Testmodellen, erzeugt.

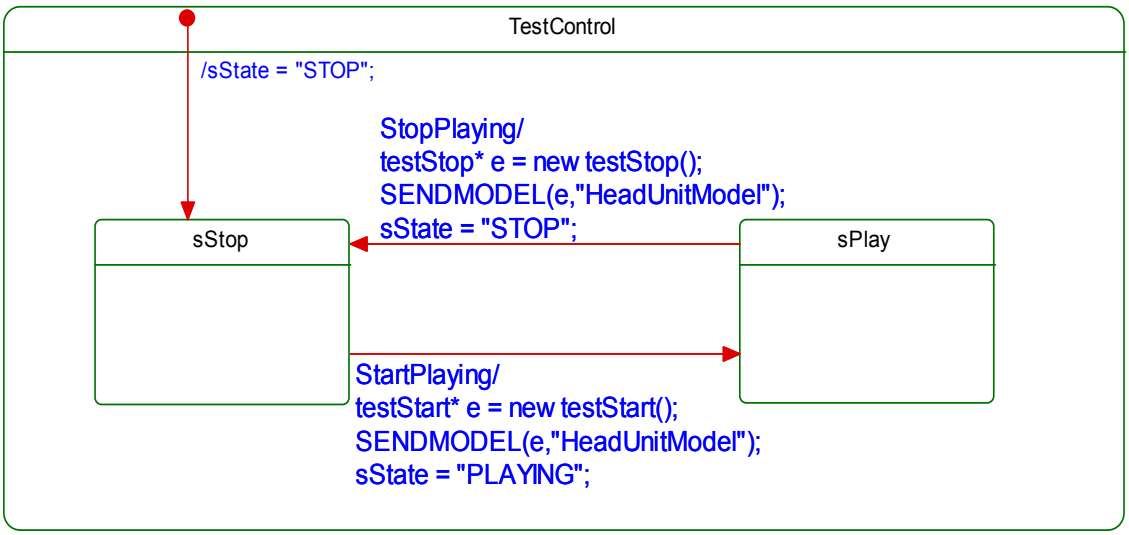


Abbildung 17: MODENA Testmodel

Ein solches Testmodell kommuniziert mit den Kommunikationsmodellen. Hierzu gibt es wiederum zwei Möglichkeiten. Zum einen kann ein Testmodell Events erzeugen und an andere Modelle senden (**SENDMODEL**), zum anderen können auch die aktuellen Zustände der Modelle abgefragt werden. Für den zweiten Ansatz muss jedes Modell eine Zustandsvariable (**sState**) besitzen, die beim Betreten eines jeden Zustandes des Modells aktualisiert wird.

2.3.2 RAVENNA

RAVENNA ist ein modellbasiertes Spezifikationssystem für Elektronik-Komponenten im Infotainment-Umfeld. Mit Hilfe leistungsstarker Werkzeuge werden Anforderungen systematisch erfasst und durch symbolische UML-Spezifikationen ergänzt. Die erstellten Modelle lassen sich in RAVENNA simulieren und durch typische Infotainment-Vorserienhardware wie Displays und Bedienelemente sehr einfach ergänzen. RAVENNA unterstützt zusätzlich die Integration von Sprachausgabe und Spracherkennung, sowie die Ankopplung von Bussystemen wie CAN zu den Funktionsmodellen.

Im Gegensatz zu MODENA, wo der Schwerpunkt auf dem Testen existierender Steuergeräte und deren Kommunikation (auch bereits während der Entwicklung) liegt, setzt RAVENNA bereits einen Schritt eher bei der Spezifikation an.
Traditionelle, textuelle Spezifikationen stoßen heutzutage wegen der stark angestiegenen Funtionsvielfalt der Systeme oft an ihre Grenzen und werden zudem meist sehr unübersichtlich. Insbesondere die immer kürzeren Entwicklungszyklen erfoldern eine Möglichkeit zur Frühzeitigen Validierung der Konzepte. Durch simulierte Prototypen ermöglicht es RAVENNA, den Entscheidungsprozeß transparenter zu gestalten.

Im Rahmen dieser Arbeit wird zunächst nur die Anbindung von MODENA an die Infotainment Busse realisiert. Allerdings soll das Konzept über eine einheitliche Schnittstelle eine spätere Anbindung weiterer Anwendungen (wie RAVENNA) ermöglichen.

3 Design eines Kommunikationsframeworks als Daten-Abstraktionsschicht

3.1 Anforderungsanalyse

Ziel dieser Arbeit ist die Konzeptionierung und Entwicklung einer Abstraktionsschicht zur generischen Datenkommunikation zwischen den in Kapitel 2.1 erläuterten Bustechnologien auf der einen Seite und modellbasierten Modellierungsumgebungen (insbesondere MODENA – siehe Kapitel 2.3.1) auf der anderen Seite. Hierzu müssen zunächst die speziellen Anforderungen an eine solche Mittelschicht aus Anwendungs- sowie Hardwaresicht untersucht werden.

3.1.1 Nachrichtenprotokolle

Hardwareseitig sind hier die bereits in Kapitel 2 erläuterten Transportprotokolle, insbesondere die verwendeten Nachrichtenformate und Systemmeldungen genauer zu betrachten. Aus diesen Nachrichten soll später ein geeignetes Datenformat für die interne Weiterleitung und Verarbeitung innerhalb des Kommunikationsframeworks hergeleitet werden.
Bei MOST sollen insbesondere der Kontrollkanal und der asynchrone Kanal von der Abstraktionsschicht unterstützt werden, außerdem sollen CAN Nachrichten verarbeitet werden können. Darüber hinaus soll das verwendete Datenformat später einfach an weitere Nachrichtenprotokolle (etwa der synchrone MOST Kanal oder FlexRay) angepasst werden können.
In der MOST Spezifikation [MOS-05] wurde der Aufbau von Kontrollnachrichten und asynchronen Datenpaketen wie folgt festgelegt.

Byte	Task
0-3	Arbitration
4-5	Target address
6-7	Own address (Source address)
8	Message type
9-25	Data area
26-27	CRC
28-29	Transmission status
30-31	Reserved

Data area of MOST Network Interface Controller = 17 Byte

16 Bit DeviceID	8 Bit FBlock ID	8 Bit Inst. ID	12 Bit Fkt ID	4 Bit OP Type	4 Bit Tel ID	4 Bit Tel Len	8 Bit Data 0	8 Bit Data 1	...	8 Bit Data 11

Abbildung 18: MOST Kontrollnachricht und MOST Telegramm [MOS-05,S.117 + S.177]

Neben den Adressen vom sendenden und empfangenen Steuergerät und dem Typ der übermittelten Nachricht (Normal oder System), stellt das Datenfeld (auch MOST Telegram genannt) den wichtigsten Teil einer Kontrollnachricht dar. In diesem wird die prinzipielle Protokollstruktur zum Aufruf spezieller Funktionen eines Steuergerätes versendet:

DeviceID.FBlockID.InstID.Fkt.ID.OPType.Length(Data)

Die Device ID kennzeichnet bei empfangenen Nachrichten die logische Adresse des Senders und bei zu sendenden Nachrichten die Adresse des Empfängers. Diese muss nicht von der Anwendung interpretiert werden, sondern kann auf unterer Ebene von den Network Services eingetragen werden. Die möglichen Werte für Funktionsblöcke und ihre enthaltenen Funktionen, sowie die entsprechenden Parameter können in speziellen Funktionskatalogen nachgeschlagen werden. Hierbei gibt es sowohl allgemein festgelegte als auch herstellerspezifische Funktionen. Die möglichen Operationstypen sind für alle Funktionen fest in der Spezifikation vorgegeben.

OPType	For Properties	For Methods
Commands:		
0	Set	Start
1	Get	Abort
2	SetGet	StartResult
3	Increment	Reserved
4	Decrement	Reserved
5	GetInterface	GetInterface
6	Locked for definitions	StartResultAck
7	Locked for definitions	AbortAck
8	Locked for definitions	StartAck
Reports:		
9	ErrorAck	ErrorAck
A	Locked for definitions	ProcessingAck
B	Reserved	Processing
C	Status	Result
D	Locked for definitions	ResultAck
E	Interface	Interface
F	Error	Error

Abbildung 19: MOST OPTypes [MOS-05,S.44]

Dabei müssen einzelne Funktionen nicht alle Operationstypen implementieren. Fehlermeldungen werden von den jeweiligen Steuergeräten versendet, wenn die Funktionen nicht ausgeführt werden konnten (zum Beispiel weil falsche Parameter übergeben wurden).
Die Telegramm ID (s. Abbildung 19) gibt an, ob es sich um ein einzelnes Telegramm oder um eine segmentierte Übertragung einer längeren Nachricht handelt. Mit TelLen wird die Anzahl der verwendeten Datenbytes angegeben.
Nachrichtenpakete, welche über den asynchronen Kanal übertragen werden können, haben im Grunde denselben Aufbau wie Kontrollnachrichten. Allerdings können in ihnen größere Datenmengen übertragen werden. Außerdem gibt es beim asynchronen Kanal keinen Acknowledge - Mechanismus und dementsprechend auch keine automatische Übertragungswiederholung auf unterster Ebene.
Neben den beschriebenen Nachrichten treten beim MOST – Bus noch weitere Ereignisse (Events) auf, welche Änderungen am Systemzustand repräsentieren. So sendet bei der üblicherweise verwendeten optischen Datenübertragung ein ausgewiesenes Steuergerät (der Network Master) Licht an seinem Ausgang und generiert die Übertragungsrahmen. Alle anderen Einheiten im Ring müssen sich zunächst mit diesem Gerät synchronisieren (light & lock) damit eine Kommunikation stattfinden kann. Im Laufe des Betriebs kann es vorkommen, dass diese Synchronisation kurzzeitig verloren geht oder dass der Lichtimpuls unterbrochen wird. Dies passiert zum Beispiel wenn ein Steuergerät aus dem Ring entfernt wird. Außerdem besitzt jedes Steuergerät im Ring eine physikalische und eine logische Adresse, sowie eine Gruppenadresse, welche sich dynamisch ändern können. Über all diese Zustandsänderungen am System soll die angeschlossene Anwendung informiert werden. Darüber hinaus können die Events auch anwenderseitig generiert werden, zum Beispiel um Störungen bei der Kommunikation zu simulieren.

CAN Nachrichten sind im Vergleich zu MOST Nachrichten einfacher aufgebaut. Sie bestehen aus einer Identifizierung (entweder 11 oder 29 Bits), welche bereits eindeutig die jeweilige Nachricht, sowie den sendenden Knoten kennzeichnet. Daneben enthalten sie bis zu 8 Bytes an Nutzdaten, eine entsprechende Längenangabe und Sicherungs- und Bestätigungsfelder (siehe Kapitel 2.1.1).

Für jede zu unterstützende Protokollnachricht beziehungsweise für jedes Ereignis wird es ein eigenes Datenobjekt geben, um die jeweiligen Attribute zu übermitteln.

3.1.2 existierende Hardwareschnittstellen zur Busanbindung

Um die im vorherigen Kapitel beschrieben Nachrichten und Signale von den entsprechenden Übertragungsmedien abzuhören und dem Computer für weitere Untersuchungen zur Verfügung zu stellen, existieren bereits mehrere Hardwareschnittstellen am Markt. Bei Berner & Mattner werden insbesondere zwei dieser Geräte verwendet, welche im Folgenden kurz erläutert werden sollen.

Für den MOST Bus wird zum einen die so genannte Optolyzer Interface Box von Oasis [INT-08], im weiteren Verlauf dieser Arbeit kurz als Optolyzer bezeichnet, und zum anderen das VN2600 MOST Interface der Firma Vector Informatik [INT-09] eingesetzt. Für den CAN Bus existieren bei Berner & Mattner ebenfalls mehrere Hardwareschnittstellen (zum Beispiel die CANCardXL von Vector Informatik).

Beim Optolyzer handelt es sich um einen MOST-Netzwerkadapter, der mit einem Windows-PC über die serielle Schnittstelle (RS232 Interface) verbunden wird. Die Portnummer der Schnittstelle dient auch zur Unterscheidung mehrerer angeschlossener Geräte. Der MOST Transceiverchip wandelt die empfangenen Lichtimpulse in elektrische Impulse um (und umgekehrt). Über eine S/PDIF-Schnittstelle oder jeweils zwei Audio Ein- und Ausgänge können außerdem Audiodaten auf den synchronen Kanal des MOST-Bus eingespeist beziehungsweise der Kanal abgehört werden. Der Zugriff auf die Optolyzer Hardware erfolgt über die bereits in Kapitel 2.1.2 vorgestellte NetServices API. In dieser Schnittstelle sind Callback Funktionen definiert über die eine angebundene Anwendung über das Eintreffen einer neuer Nachricht beziehungsweise eines Systemevents informiert wird. Darüber hinaus gibt es Methoden zum senden von Kontrollnachrichten und Datenpaketen, sowie zum auslesen und ändern von Registereinträgen der Hardware. Der Optolyzer kann dabei wahlweise in verschiedenen Modi betrieben werden. Als Master oder Slave repräsentiert er ein reales Steuergerät und nimmt aktiv an der Kommunikation auf dem Bus teil. Das heißt, er wird von anderen Geräten erkannt und angesprochen und kann auch selbst Nachrichten an die anderen Kommunikationsteilnehmer verschicken. In diesem Fall müssen in den dafür vorgesehenen Registern des Optolyzers die benötigten Adressen eines MOST Steuergerätes eingetragen werden. Als Master übernimmt er zudem die Generierung der Übertragungsrahmen. Im passiven Spymode dagegen hängt der Optolzer nur als unsichtbarer Beobachter im Ring. So kann er weder von anderen Steuergeräten erkannt werden, noch selbst eigene Nachrichten versenden. Dieser Modus

ermöglicht es daher die Kommunikation mit zu protokollieren, ohne die Systemkonfiguration zu beeinflussen.

Der VN2600 funktioniert im Prinzip ähnlich wie der Optolyzer. Allerdings erlaubt er aufgrund der Anbindung über die USB Schnittstelle des PCs höhere Übertragungsgeschwindigkeiten und ermöglicht es neben dem Kontrollkanal auch auf den asynchronen Kanal zuzugreifen. Außerdem kann die VN2600 Interface Hardware parallel im Spy und im Knotenmodus (Master / Slave) betrieben werden. Somit ist es (bei eben dieser Konfiguration) möglich zwei Optolyzer durch einen VN2600 zu ersetzen. Der Zugriff auf die Hardware erfolgt beim VN2600 über eine spezielle mitgelieferte Schnittstelle, welche es der Anwendung unter anderem ermöglicht den internen Nachrichtenpuffer auszulesen.

3.1.3 weitere Anforderungen und Übersicht

Zusätzlich zu den oben beschriebenen Anforderungen aus Sicht der anzubindenden Hardware sollten im Rahmen dieser Arbeit weitere Kriterien auf ihre Erfüllbarkeit hin untersucht werden.
Um eine spätere einfache Erweiterbarkeit des Kommunikationsframeworks zu ermöglichen, galt es besonders einen modularen Aufbau und eine einheitliche Schnittstelle für den internen Datentransport in Betracht zu ziehen. Außerdem sollte die Mittelschicht möglichst unabhängig von der verwendeten Hard- und Software (insbesondere dem Betriebssystem) nutzbar sein.

Anforderung	Umsetzung	Bemerkungen / Probleme
Nachrichtenprotokolle		
MOST Systemevents (Light & Lock, Address Change)	konkrete Datenobjekte abgeleitet von einer SystemEvent Klasse	Devices wandeln Events in Datenobjekte um
MOST Kontrollnachrichten (senden und empfangen)	konkretes Datenobjekt abgeleitet von einer MOST Message Klasse	Verarbeitung von Bestätigungen (Acknowledgement) und Prüfsummen hardwareabhängig

asynchrone Pakete (MOST)	konkretes Datenobjekt abgeleitet von einer MOST Message Klasse	eingeschränkte Testmöglichkeit aufgrund der Einfachheit der verwendeten Testumgebung
CAN Nachrichten	konkretes Datenobjekt abgeleitet von einer CAN Message Klasse	nur konzeptionell ausgearbeitet
Hardwareschnittstellen		
VN2600 (Vektor Informatik)	Anbindung der Hardwareschnittstelle über mitgelieferte API, Blockklasse zur Verarbeitung der Daten	mitgelieferte Geräte-Treiber nur für Windows
OptoLyzer (Oasis)	Anbindung der Hardwareschnittstelle über mitgelieferte API oder direkt über RS232	nur konzeptionell ausgearbeitet, mitgelieferte Geräte-Treiber nur für Windows
zusätzliche Anforderungen		
Erweiterbarkeit	Modulare Mittelschicht, Service Orientierte Architektur	Nachweis über unterschiedliche Testkonfigurationen
Portabilität (BS unabhängig)	C++ Bibliothek	Verzicht auf softwareabhängige Datentypen (Rhapsody) und Befehle (WIN API)

Abbildung 20: Anforderungen und Realisierungen

3.2 Framework – Architekturen

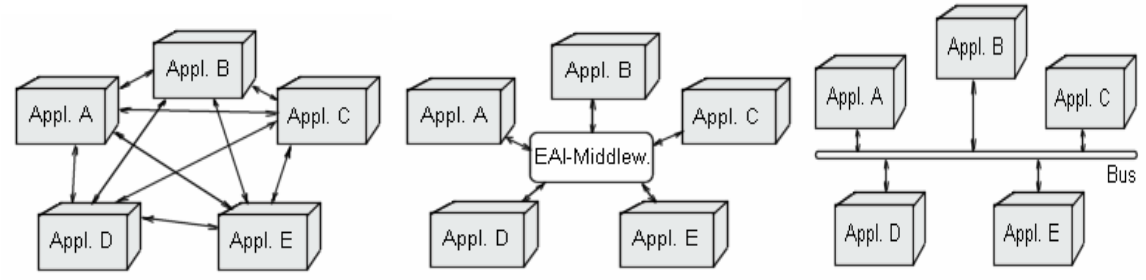
Im folgenden Kapitel werden alternative Architekturansätze für die Entwicklung eines Frameworks beschrieben und auf ihre Anwendbarkeit für das zu erstellende Kommunikationsframework hin untersucht.

3.2.1 Enterprise Application Integration

In der Literatur lassen sich jede Menge teilweise recht unterschiedliche Definitionen von EAI finden, bei denen oft jeweils nur auf spezielle Eigenschaften eingegangen wird. Die folgende Begriffsdefinition stammt aus der Wikipedia Online Enzyklopädie [WIK-06]:

„Enterprise Application Integration (EAI), auch Unternehmensanwendungsintegration (UAI), ist ein Konzept zur unternehmensweiten Integration der Geschäftsfunktionen entlang der Wertschöpfungskette, die über verschiedene Applikationen auf unterschiedlichen Plattformen verteilt sind, und die im Sinne der Daten- und Geschäftsprozessintegration verbunden werden können."
„EAI umfasst die Planung, die Methoden und die Software, um heterogene, autonome Anwendungssysteme - ggf. unter Einbeziehung von externen Anwendungssystemen - prozessorientiert zu integrieren."

Es handelt sich hierbei also um einen Ansatz, um (verteilte) Anwendungen, welche unabhängig voneinander und zum Teil mit verschiedenen Technologien entwickelt wurden Interagieren zu lassen.



.

Abbildung 21: EAI – Integrationstopologien [INT-10]

Die obige Abbildung zeigt die herkömmlichen Punkt zu Punkt (P2P) Kommunikationen zwischen den verschiedenen Abbildungen ohne den Einsatz einer EAI Software, sowie

zwei alternative Integrationstopologien. Die Nachteile der Punkt zu Punkt Lösung liegen vor allem in der geringen Flexibilität insbesondere beim Austausch einzelner Anwendungen und der damit verbundenen hohen Folgekosten durch die vielen benötigten Kommunikationswege. Durch die Integration der Anwendungen über eine EAI Middleware oder einen gemeinsamen Bus können einzelne Systeme relativ einfach ersetzt werden und somit die Folgekosten reduziert werden. Die verschiedenen Anwendungen werden über so genante Adaptoren angebunden. Dabei kann die Mittelschicht beziehungsweise der Bus bereits eine Prozesslogik bereitstellen, um die entstehenden Daten eines bestimmten Geschäftsprozesses in richtiger Reihenfolge an die jeweiligen Anwendungen zu leiten.

Man unterscheidet je nach Einsatzgebiet zwischen unterschiedlichen Integrationsarten. So können Anwendungen unternehmensintern (A2A) oder auch unternehmensübergreifend (B2B) verbunden werden. Außerdem können mehrere Anwendungen mit einer gemeinsamen Benutzeroberfläche versehen werden (P2S), um einen einheitlichen Zugriff für den Anwender zu realisieren.

Das Prinzip der Anwendungsintegration findet bei dem Kommunikationsframework dieser Arbeit Anwendung, indem mehrere verschiedene existierende Anwendungen (MODENA / RAVENNA / etc.) auf einheitliche Weise mit den ebenfalls bereits existierenden Hardwareanbindungen interagieren sollen.

3.2.2 Service Oriented Architecture

„SOA ist ein System-Architekturkonzept, das unternehmensweit eingesetzt werden kann. Eine SOA ist ein Konzept für eine Systemarchitektur, in dem Funktionen in Form von wieder verwendbaren, technisch voneinander unabhängigen und fachlich lose gekoppelten Services implementiert werden. Services können unabhängig von zugrunde liegenden Implementierungen über Schnittstellen aufgerufen werden, deren Spezifikationen öffentlich und damit vertrauenswürdig sein können. Serviceinteraktion findet über eine dafür vorgesehene Kommunikationsinfrastruktur statt. Mit einer serviceorientierten Architektur werden in der Regel die Gestaltungsziele der Geschäftsprozessorientierung, der Wandlungsfähigkeit (Flexibilität), der Wiederverwendbarkeit und der Unterstützung verteilter Softwaresysteme verbunden.“ [WIK-06].

Das Ziel beim Einsatz einer service- oder auch dienstorientierten Architektur ist es, die heterogene Systemlandschaft effizient an Änderungen im Geschäftsprozess anpassen zu können. Hierzu werden komplexe Geschäftsprozesse in Abfolgen einzelner Funktionen (Dienste) unterteilt, welche von einem Service (oder genauer einem Service Provider) bereitgestellt werden. Service Consumer stellen entsprechende Anfragen (Requests) an die Service Provider und erhalten das Ergebnis der Verarbeitung als Antwort. Man kann dabei zwischen synchroner und asynchroner Kommunikation zwischen den beiden Partnern unterscheiden. Die Kommunikation erfolgt dabei über definierte Schnittstellen unabhängig von der konkreten Implementierung der einzelnen Services.
Daher können verschiedene Services auch in unterschiedlichen Programmiersprachen und auf unterschiedlichen Systemen realisiert werden, wodurch sich eine SOA gut zur Anwendungsintegration (siehe vorheriges Kapitel) eignet.
Ein weiteres Ziel einer Service Orientierten Architektur ist die Kapselung von Daten durch Dienste, die exklusive Lese- und Schreibrechte darauf besitzen. Die somit erzielte Modularität führt zu geringen Redundanzen und einer höheren Flexibilität der Systeme.
Zur Verwaltung der Services können die Provider ihre Dienste in einen speziellen Katalog (Service Registry) eintragen, in dem die Service Consumer nach bestimmten Diensten suchen können. Der allgemeine Ablauf einer solchen Dienstanfrage ist in der folgenden Abbildung dargestellt.

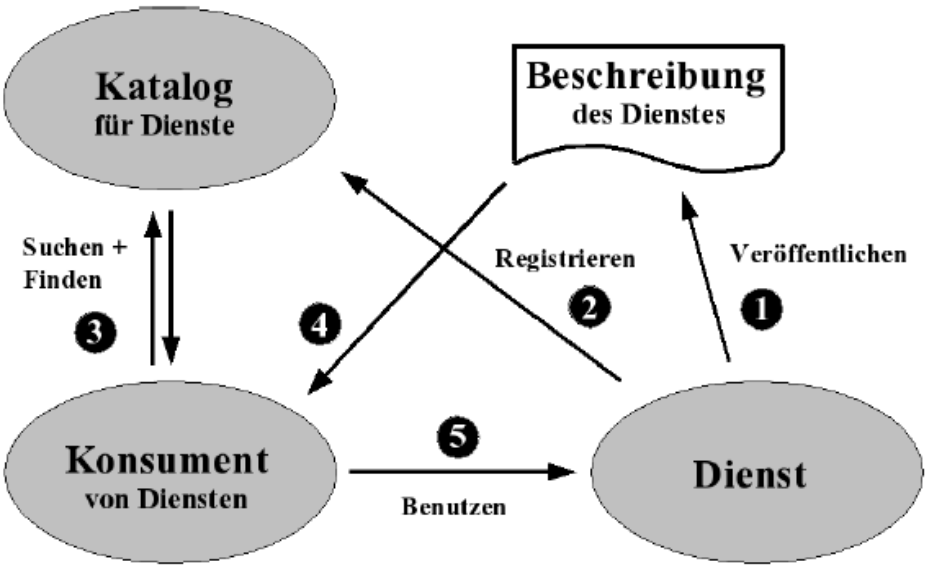


Abbildung 22: SOA – Benutzung von Services [INT-11]

Häufig wird Service Orientierte Architektur mit Web Services gleichgesetzt, wobei diese lediglich eine mögliche Realisierung von SOA darstellen. (Welche allerdings sehr oft eingesetzt wird.)

„Ein **Web Service** ist eine Software-Anwendung, die mit einem Uniform Resource Identifier (URI) eindeutig identifizierbar ist und deren Schnittstellen als XML-Artefakte definiert, beschrieben und gefunden werden können. Ein Web Service unterstützt die direkte Interaktion mit anderen Software-Agenten unter Verwendung XML-basierter Nachrichten durch den Austausch über internetbasierte Protokolle.“ [WIK-06]

Ein Web Service repräsentiert dabei einen Service Provider, der einen bestimmten Dienst über das Internet bereitstellt. In der auf XML basierenden Web Services Description Language (WSDL) werden Dienste (besser gesagt der Zugriff darauf) plattform- und programmiersprachenunabhängig beschrieben. Diese Beschreibungsdokumente werden in einem Verzeichnisdienst (UDDI) hinterlegt, in dem der Service Consumer nach bestimmten Web Services suchen kann. Die eigentliche Kommunikation zwischen Consumer und Provider wird über ein standardisiertes Protokoll (SOAP) realisiert.

Neben Web Services werden aber auch noch weitere dienstorientierte Technologien, wie etwa CORBA oder DCOM für den Aufbau einer Service Orientierten Architektur eingesetzt.
Eine wichtige Entscheidung beim Entwurf einer Service Orientierten Architektur betrifft die Granularität der identifizierten Services. Fein-granulare Services führen zu einer hohen Wiederverwendbarkeit, stellen aber nur einen geringen Funktionsumfang zur Verfügung. Außerdem kann die Aufteilung eines Prozesses auf viele einzelne Services zu einem erheblichen Kommunikationsaufwand führen, was insbesondere in größeren Netzen einen merklichen Performanzverlust im Gegensatz zu wenigen, größeren Services zur Folge hat.

Für das in dieser Arbeit zu entwickelnde Kommunikationsframework wird hauptsächlich der Ansatz verfolgt, Funktionalitäten in Form von Services zu realisieren, die unabhängig von ihrer konkreten Implementierung über eine einheitliche Schnittstelle angesprochen werden können. Allerdings liegt der Fokus zunächst nicht, wie bei SOA üblich, auf Webanwendungen und dem entfernten Zugriff über ein Netzwerk. Wobei man sich später gegebenenfalls darüber Gedanken machen könnte, das Kommunikationsframework auch übers Netz von mehreren verteilten Anwendungen aus nutzbar zu machen (hierfür werden im Konzept auch bereits Ports für eine Socket Anbindungen angedacht.

3.2.3 Event-Driven Architecture

„Eine der großen Herausforderungen für Unternehmen besteht darin, Entscheidungen immer schneller zu treffen, Änderungen, Risiken und Chancen auf den jeweiligen Zielmärkten sowie in der Wertschöpfungskette und in den innerbetrieblichen Prozessen schneller wahrzunehmen und darauf zu reagieren.“ [INT-12, S.3]

Jede dieser Änderungen tritt in Form eines Ereignisses auf, zum Beispiel eine Auftragserteilung oder die Bestätigung des Eingangs einer neuen Nachricht.
Ereignisse können geplant (etwa eine Empfangsbestätigung einer Bestellung) oder ungeplant (wie beispielsweise eine Fehlermeldung eines Steuergerätes).
Das Modell einer ereignisgesteuerten Architektur basiert auf der Verwendung von Ereignissen als Auslöser einer Zustellung von Nachrichten, über welche ein oder mehrere Empfängern über das Ereignis informiert werden. Die Empfänger verarbeiten die Nachrichten und reagieren damit auf die Ereignisse. Eine Ereignisgesteuerte Architektur weist im Allgemeinen folgende Eigenschaften auf: INT-12, S.4]

- **Asynchron**: EDA unterstützt in erster Linie asynchrone Interaktionen, bei denen Informationen gesendet werden, ohne dass eine unmittelbare Antwort erwartet wird oder während des Wartens auf eine Antwort eine funktionsfähige Verbindung zwischen den beiden Systemen aufrechterhalten werden muss.
- **Publish/Subscribe**: EDA unterstützt in erster Linie "many-to-many"-Interaktionen, bei denen Systeme Informationen über ein bestimmtes Ereignis im Netzwerk veröffentlichen, sodass eine Vielzahl anderer Systeme (die den Empfang solcher Nachrichten abonniert haben und dafür autorisiert sind) diese Informationen empfangen und darauf entsprechend reagieren kann.
- **Entkoppelt:** EDA ermöglicht Interaktionen zwischen Systemen, bei denen der Verfasser einer Nachricht nicht weiß, wer die Abonnenten sind, und umgekehrt. Die Interaktion umfasst ausschließlich die gesendeten und empfangenen Informationen und nicht eine Beziehung zwischen den beiden Systemen.

Das Hauptziel für den Einsatz einer ereignisgesteuerten Architektur ist es, Unternehmen dabei zu unterstützen geplante und ungeplante Ereignisse zu erkennen und zu verarbeiten.
In der Praxis verwendet man heute meistens eine Kombination von service- und ereignisorientierter Architektur, um eine Echtzeit Unternehmensstruktur zu realisieren.

Das zu entwickelnde Kommunikationsframework muss Ereignisse, die von einer angeschlossenen Hardwareschnittstelle empfangen werden verarbeiten können. Dabei muss allerdings in Abhängigkeit von der jeweils verwendeten Hardware unterschieden werden, wie diese Events von der Schnittstelle weitergereicht werden. Die empfangenen Ereignisse werden dabei direkt auf Deviceebene verarbeitet und anschließend entsprechende interne Datenobjekte an die weiteren Blöcke gesendet.
Die Echtzeitfähigkeit des Systems, also die Möglichkeit bestimmte Reaktions- und Antwortzeiten zu garantieren, wird zunächst nicht explizit sichergestellt.

3.2.4 Model-Driven Architecture

„Der Begriff Model Driven Architecture (MDA) (wortwörtlich auf deutsch „modellgetriebene Architektur“) bezeichnet einen Ansatz zur Entwicklung von IT-Lösungen, der auf einer klaren Trennung von Funktionalität und Technik beruht.“ [WIK-06]

MDA ist ein Standard der Object Management Group (OMG), der davon ausgeht, dass für die Konstruktion eines Softwaresystems ein einziges Modell zu unscharf und überladen ist und daher das Gesamtmodell in mehrere Schichten unterteilt:

- Computation Independent Model (CIM): umgangssprachliche Beschreibungen
- Platform Independent Model (PIM): Modellierung der Geschäftsprozesse
- Platform Specific Model (PSM): Architekturmodell
- Codemodell

„Die Trennung der Modelle (*separation of concerns*) stellt eine inhaltliche Erweiterung des UML-Standards [UML-20] dar.“
„Ein Modell ist eine abstrakte Repräsentation von Struktur, Funktion oder Verhalten eines Systems.“ [WIK-06]

Neben der inhaltlichen Trennung der Modelle werden auch die Transformationen zwischen den Modellen beziehungsweise zwischen Modell und Code definiert. Dabei wird von jeweils von abstrakteren zu konkretren Modellen transformiert.

Das vordergründige Ziel der MDA ist es, die Softwareentwicklung durch Automation zu beschleunigen. Dabei wird jedoch keine hundertprozentige Automatisierung angestrebt,

sondern eine flexiblere Lösung in Abhängigkeit der jeweiligen Problemstellung.
Um diese Automation zu ermöglichen, müssen die erstellten Modelle formal eindeutig sein, was durch die Verwendung einer formal eindeutigen Modellierungssprache sichergestellt wird. MDA-Modelle werden in der Regel in UML definiert, was aber nicht zwingend vorgeschrieben ist. Darüber hinaus stellen Modelle eine höhere Abstraktionsebene als generierter Code dar und ermöglichen dadurch eine einfachere Verständlichkeit und bessere Handhabbarkeit. Hierauf zielt auch die Trennung von fachlichen und technischen Aspekten ab, welche eine Trennung der Verantwortlichkeiten ermöglicht. Außerdem soll die Interoperabilität und Austauschbarkeit durch die Verwendung standardisierter Modelle verbessert werden.

Insbesondere bei kleineren, weniger komplexen Anwendungen wird allerdings der zusätzliche Aufwand, bedingt durch den hohen Abstraktionsgrad, als Nachteil im Gegensatz zu herkömmlichen Entwicklungstechniken angesehen.

Im Zuge der Codegenerierung entsteht in der Regel kein fertiges Programm, sondern lediglich ein Codegerüst, welches durch manuelle Erweiterungen mit spezieller Funktionalität gefüllt werden muss. Grundsätzliche Änderungen an der entwickelten Software sollten bei MDA immer über eine Anpassung der entsprechenden Modelle und anschließender Neugenerierung und nicht durch direkte Änderung des fertigen Codes erfolgen. Hierdurch können Änderungen meist systematischer und nachvollziehbarer durchgeführt werden und darüber hinaus wird ein mögliches Auseinanderlaufen von Code und Modellen verhindert.

Das Kommunikationsframework dieser Arbeit wird mit Hilfe von UML Diagrammen (insbesondere Klassendiagramme) in der Entwicklungsumgebung Rhapsody für C++ entworfen. Aus den erstellten Modellen, welche die verwendeten Bausteine und deren Beziehungen untereinander abbilden, wird automatisch ein C++ Codegerüst zur weiteren Bearbeitung erstellt.
Änderungen und Erweiterungen des Systems können folglich bequem auf der Modellebene durchgeführt werden, ohne den konkreten Code anfassen zu müssen.

3.3 Entwurf

In diesem Kapitel wird die Vorgehensweise beim Entwurf des Kommunikationsframeworks beschrieben. Um die zuvor erläuteten Anforderung zu erfüllen, wurden im Laufe der Designphase mehrere Architekturmuster verwendet, welche hier ebenfalls kurz erläutert werden sollen. Für genauere Beschreibungen der einzelnen Architekturmuster wird auf die entsprechende Fachliteratur, zum Beispiel [GAM-96] oder [SHA-96] verwiesen.

3.3.1 Grundkonzept und Erweiterbarkeit des Systems

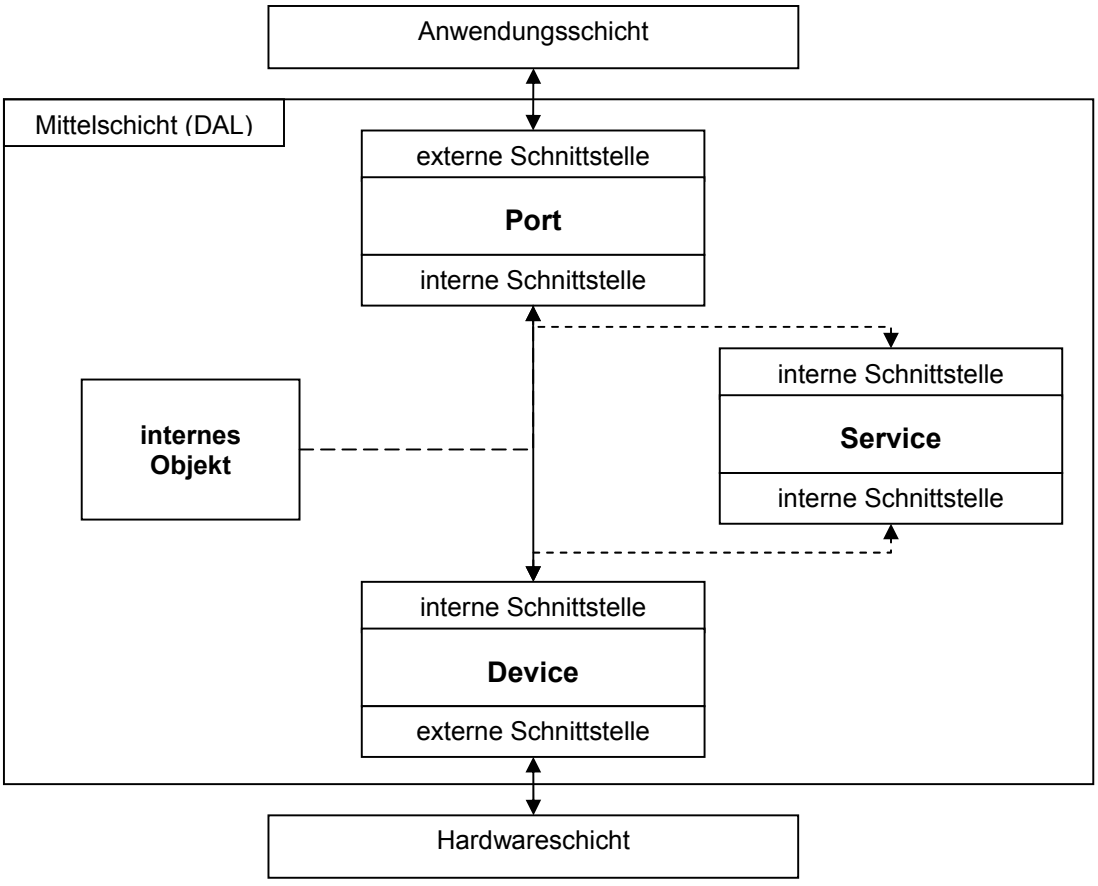


Abbildung 23: Grundstruktur der DAL

Die obige Abbildung zeigt die Grundstruktur der zu entwickelnden Mittelschicht als Bindeglied zwischen der Anwendungs- und der Hardwareschicht. Das Framework wird aus drei Arten von Bausteinen bestehen. Ports werden eine externe Schnittstelle zur Anwendung bereitstellen, während über Devices die Außenverbindung zur

Hardwareebene realisiert werden soll. Deren Schnittstelle wird dabei in Abhängigkeit der konkret verwendeten Hardwareschnittstelle variieren. In Form von Services sollen darüber hinaus zusätzliche interne Verarbeitungen in die Mittelschicht integriert werden können. Im Inneren der Mittelschicht sollen die verschiedenen Bausteine über eine einheitliche Schnittstelle miteinander kommunizieren können. Als Kommunikationsfluss werden interne Datenobjekte zwischen den Bausteinen ausgetauscht werden, welche von jedem Bausteintyp interpretiert werden können. Außerdem sollen alle Bausteine parallel und unabhängig voneinander arbeiten können. Die einheitliche Schnittstelle wird es später ermöglichen Bausteine beliebig miteinander zu kombinieren und nachträglich weitere Bausteine hinzuzufügen beziehungsweise wieder zu entfernen. Somit kann das Gesamtsystem leicht erweiterbar gemacht werden.
Diese Struktur ist dem „Pipes and Filters“ Architekturmuster [SWA-05] nachempfunden.

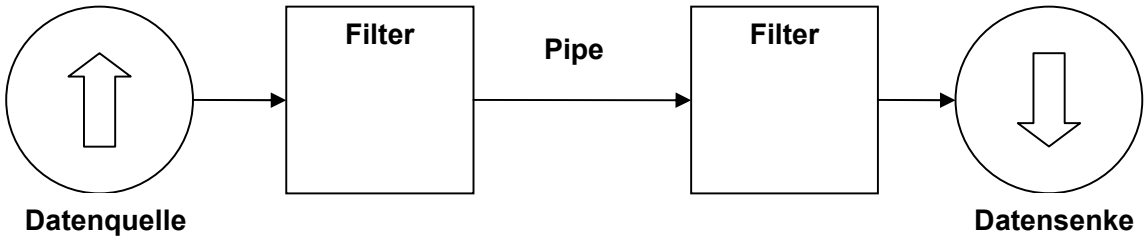


Abbildung 24: Pipes and Filters (Architekturmuster)

Eine solche Architektur eignet sich insbesondere für die Verarbeitung von Datenströmen, die in einzelne diskrete Einheiten unterteilt werden können. Die einzelnen Dateneinheiten sind dabei unabhängig voneinander und lassen sich daher auch völlig getrennt und parallel verarbeiten. Das „Pipes and Filters“ - Muster teilt die Aufgaben des Gesamtsystems dazu in mehrere Verarbeitungsstufen, die durch den Datenfluss miteinander verbunden sind

Ein solches System besteht im Grunde aus zwei Komponenten: Filter, die eine in sich abgeschlossene Verarbeitungseinheit darstellen und Pipes, über die der Datentransfer zwischen den Filtern realisiert wird. Außerdem gibt es zwei spezielle Arten von Filtern, Datenquellen und Datensenken. Über diese werden Daten in die Filterkette eingebracht beziehungsweise aus ihr abgezogen. Darüber hinaus unterscheidet man zwischen aktiven und passiven Filtern. Aktive Filter steuern den Datenfluss, indem sie Daten aus ihrer Eingangspipeline lesen und nach der Verarbeitung wieder auf ihre Ausgangspipeline

schreiben. Passive Filter dagegen werden vom Datenfluss auf den Pipes gesteuert. Ein Pipe implementiert zudem üblicherweise einen Datenpuffer.

Beim entwickelten Kommunikationsframework werden die drei Bausteintypen (Ports, Devices und Services) als unabhängige Filter realisiert. Da es möglich sein soll einen Block mit theoretisch beliebig vielen anderen Blöcken zu verbinden, werden die Datenpuffer nicht von den Pipes sondern direkt in den Filtern bereitgestellt. Der Vorteil dabei ist, dass für jeden Baustein nur ein Puffer benötigt wird, welcher Datenobjekte von verschiedenen Vorgängerknoten enthalten kann. Hierbei muss man sich später eventuell Gedanken über unterschiedliche Prioritäten der Datenobjekte machen. Dies soll im Rahmen dieser Arbeit allerdings nicht untersucht werden. Als eventueller Nachteil ist außerdem zu bedenken, dass die Puffer bei diesem Ansatz möglicherweise größer dimensioniert werden müssen, als es bei einzelnen Puffern für jede Verbindung der Fall wäre. Als Pipes dienen einfache Methodenaufrufe der Form „**Buffer.add(Object)**“. Dazu muss jeder Block seine aktuell verbundenen Nachfolger beziehungsweise deren Puffer kennen.
Durch die universellen Kombinationsmöglichkeiten beliebiger Blöcke wäre es darüber hinaus möglich Schleifen in die Filterketten einzubauen.

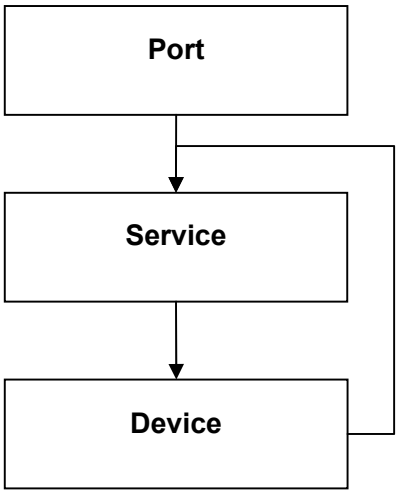


Abbildung 25: Filterkette mit Schleife

Die obige Abbildung zeigt eine solche Filterkette. Dieses Problem kann teilweise dadurch gelöst werden, dass Ports und Devices intern zwei getrennte Verarbeitungsprozesse ausführen. Der eine Prozess (bzw. Thread) verarbeitet Eingänge von der externen Schnittstelle und leitet sie an die nachfolgenden Blöcke weiter, während der andere Datenobjekte aus dem internen Datenspeicher (interne Schnittstelle) verarbeitet und über

die externe Schnittstelle nach außen transportiert. Allerdings bleibt immer noch die Möglichkeit mehrere Services zu einer Endlosschleife zu verschachteln. Eine automatische Überprüfung der DAL Strukturen soll allerdings hier nicht betrachtet werden, stattdessen wird die Verantwortung für den Aufbau einer sinnvollen und schleifenfreien Filterkette dem Anwender übertragen.
Als Datenquellen bzw. Senken werden externe Komponenten aufgefasst, die nicht direkt Bestandteil des Kommunikationsframeworks sind. Dies können zum einen die in Kapitel 3.1.2 vorgestellten Hardwareschnittstellen und zum anderen Schnittstellenkomponenten der Anwendung sein.
Als Beispiel für eine solche Schnittstellenkomponente zur Anbindung der Anwendung wird im späteren Verlauf der Arbeit eine einfache Beobachterkomponente (Observer Pattern, [GAM-96, S.]) entwickelt werden.

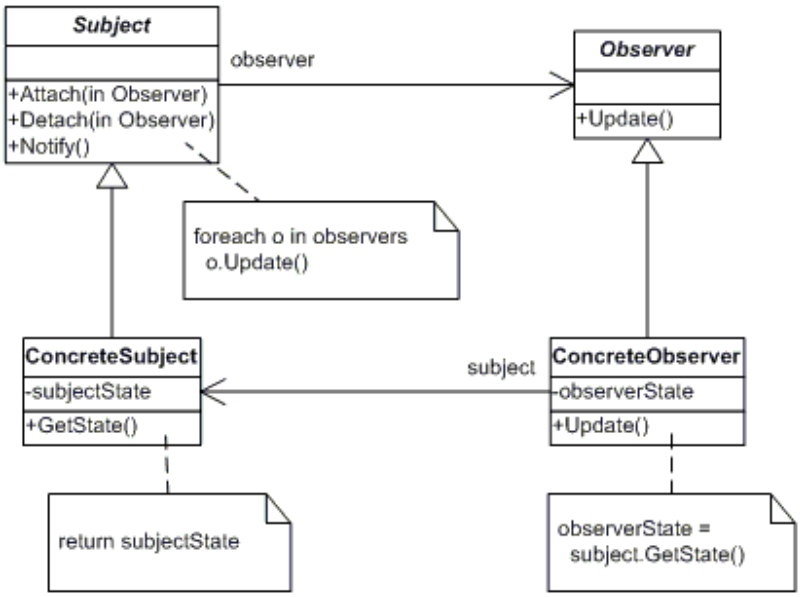


Abbildung 26: Observer Pattern [INT-10]

Das Beobachtermuster besteht aus Subjekten, welche einen Zustand bereithalten und Beobachtern, die sich für Änderungen von Zuständen eines oder mehrerer Subjekte interessieren. Um über Zustandsänderungen informiert zu werden muss sich ein konkreter Beobachter bei einem Subjekt registrieren. Hierzu stellt die abstrakte Oberklasse Subjekt Methoden zum an- bzw. abmelden zur Verfügung. Außerdem wird eine Liste aller registrierten Beobachterkomponenten verwaltet und eine Methode zu deren Benachrichtigung bereitgestellt. Dabei kennen die Subjekte keine konkreten Beobachterklassen. Die einzelnen Beobachter speichern jeweils eine Kopie des Zustands

der beobachteten Subjekte. Ziel einer solchen Architektur ist es mehrere unabhängige Komponenten konsistent zu halten.

Bei dem in dieser Arbeit zu entwickelnden Kommunikationsframeworks sollen sich Schnittstellenkomponenten einer Anwendung bei einem oder mehreren Ports registrieren können, um vom Bus empfangene Nachrichten weitergeleitet zu bekommen. Dabei speichert ein Port keinen Zustand, sondern leitet einfach alle eingehenden Datenobjekte an die registrierten Beobachterkomponenten weiter. Die Beobachter müssen daher das interne Datenformat des Frameworks kennen und interpretieren können. Eine weitere Schnittstellenkomponente wird später benötigt, um von der Anwendung erzeugte Events in das Datenformat der DAL zu übersetzen und an die Ports zu senden.
Für die Vernetzung der Basiskomponenten des Kommunikationsframeworks kann das Kompositum Strukturmuster (Composite Pattern) angewandt werden. Mit diesem Muster wird eine baumartige Hierarchie von kombinierten Objekten dargestellt. Die Komponente definiert eine gemeinsame Schnittstelle für einfache und zusammengesetzte Objekte und kann zudem ein Standardverhalten implementieren. Ein Kompositum setzt sich wiederum aus mehreren Komponenten (einfache und kombinierte) zusammen, die häufig als seine Kinder bezeichnet werden.

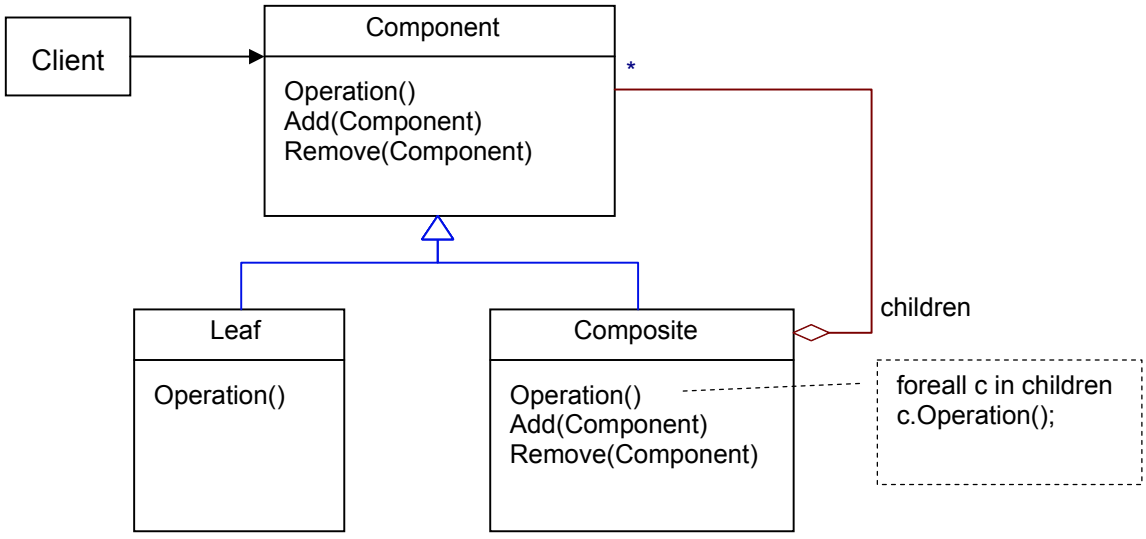


Abbildung 27: Composite Pattern (Eigene Darstellung nach [GAM-96, S. 241])

Eine externe Klasse (der Client) ruft die von der Schnittstelle definierte Operation auf, ohne wissen zu müssen ob es sich bei der konkreten Zielklasse um ein einzelnes Blatt oder eine komplexere Komponente handelt. Die Operation eines Kompositums ruft dabei jeweils die entsprechende Operation aller Kindelemente auf.

Für das Kommunikationsframework wird das Kompositum Muster verwendet, um zu ermöglichen, dass die Basisblöcke beliebig miteinander verbunden werden können. Hierzu wird eine einheitliche Schnittstelle definiert, die alle Blöcke implementieren müssen. Die hierarchischen Beziehungen sind hierbei allerdings keine wirklichen Teil-Ganzes Relationen, da sich ein einzelner Block nicht aus mehreren Bausteinen zusammensetzen soll, sondern nur zu einer Reihe von anderen (von der Schnittstelle her gleichartigen) Blöcken verbunden sein soll. Konkret wird es eine Methode **sendObject** geben, mit der Datenobjekte zwischen beliebigen Blöcken transferiert werden können.
Reine Blätter die keine Verbindungen zu anderen Blöcken haben wird es dabei zunächst nicht explizit geben, da jeder Block theoretisch mit beliebigen weiteren Blöcken kommunizieren können soll. Allerdings könnte man sich auch interne Services vorstellen, die zwar Datenobjekte erhalten aber nicht weiterleiten sollen. Zum Beispiel würde es Sinn machen eine Filterkette fürs reine „Hochreichen“ der Daten zu konfigurieren und gewisse Verarbeitungen (etwa Protokollierungen) welche die eigentlichen Daten nicht verändern parallel dazu in einer zweiten Filterkette durchzuführen. Um ein einzelnes Datenobjekt hierbei nicht mehrfach an die Anwendung bzw. die Hardware zu schicken, könnte diese zweite Kette durch einen Block ohne Ausgabemöglichkeit beendet werden. Die folgende Abbildung zeigt Komponente und Kompositum des Kommunikationsframeworks.

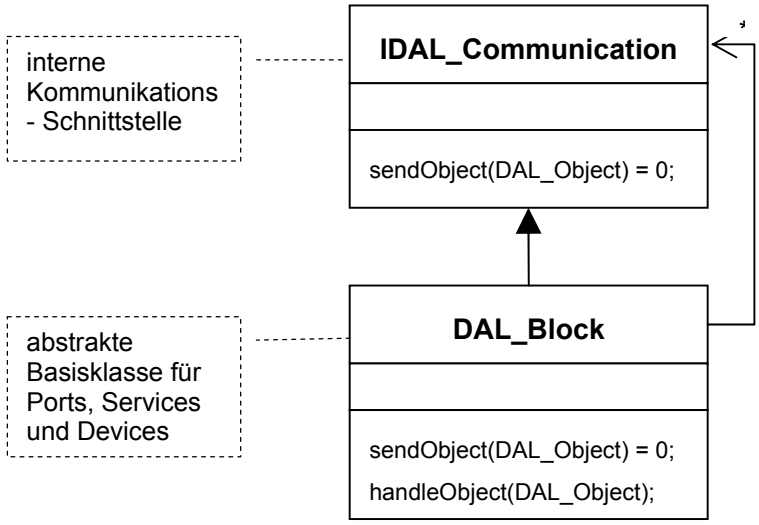


Abbildung 28: Anwendung des Kompositum Musters

Das Interface **IDAL_Communication** definiert eine einheitliche interne Kommunikationsschnittstelle und ermöglicht somit die universelle Vernetzung beliebiger Blöcke. Die abstrakte Basisklasse **DAL_Block** verwaltet eine Liste von Komponenten, die alle diese Schnittstelle implementieren und später die internen Datenobjekte weitergeleitet

bekommen. Von ihr müssen alle zukünftig zu implementierenden konkreten Blöcke abgeleitet werden.

Zur Konfiguration des Kommunikationsframeworks soll eine Administratorklasse verwendet werden, welche nach dem Singleton Pattern konzipiert ist. Dieses Erzeugungsmuster stellt sicher, dass es von einer Klasse nur genau eine Instanz geben kann und erlaubt einen globalen Zugriff auf dieses Objekt. Eine solche Klasse besitzt nur einen privaten Konstruktor, der nicht von außerhalb aufgerufen werden kann. Über eine statische Klassenmethode (getInstance) können andere Objekte auf die Singletoninstanz zugreifen. Beim ersten Aufruf legt diese Methode die Instanz an und liefert später immer dieses eine Objekt zurück. Die Administratorklasse wird darüber hinaus weitere (öffentliche) Methoden erhalten, mit deren Hilfe die Blöcke des Kommunikationsframeworks erzeugt und verbunden werden können ohne direkt auf spezielle Blöcke zugreifen zu müssen.

3.3.2 Blockklassen

Im Folgenden werden Klassendiagramme für die Blocktypen Devices und Ports entworfen, um einen Überblick über die möglichen Bausteine des Kommunikationsframeworks zu vermitteln. Im Laufe dieser Arbeit werden allerdings nicht alle Blöcke, sondern nur eine konkrete Beispielkonfiguration implementiert werden.

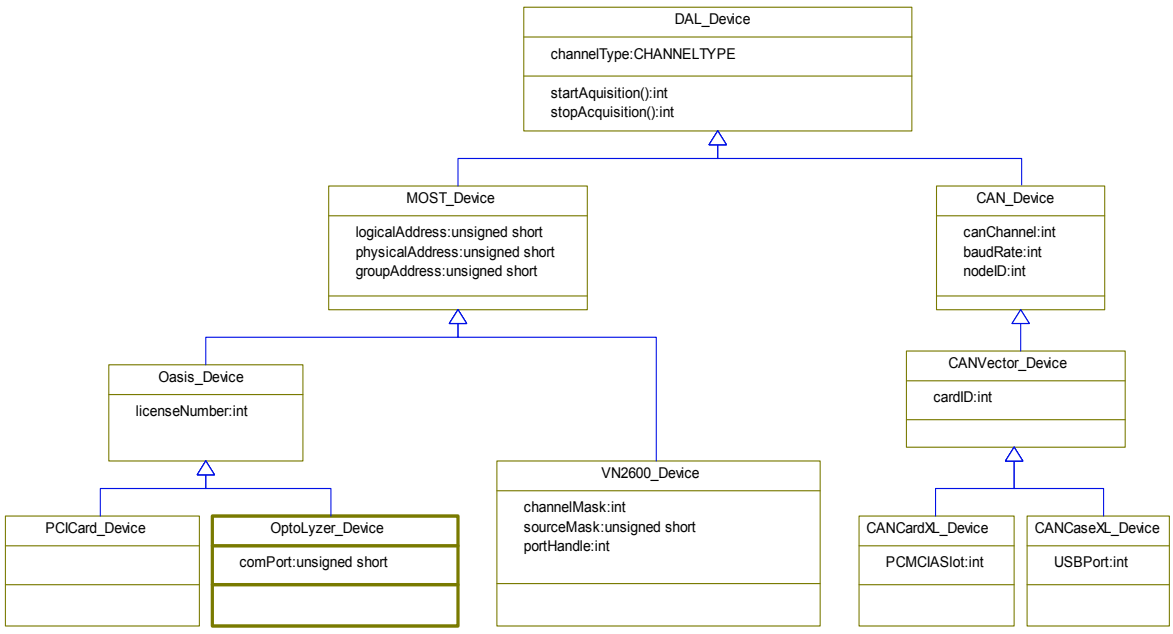


Abbildung 29: DAL Device Klassenhierarchie

Das folgende Klassendiagramm enthält die Schnittstellengeräte zur Hardwareanbindungen, die aktuell bei Berner & Mattner vorhanden sind und eingesetzt werden. Zunächst lassen sich diese Geräte in zwei Kategorien (MOST und CAN Devices) unterteilen. Für beide Typen werden Methoden zum starten und beenden der Buskommunikation benötigt, welche bereits in der allgemeinen Basisklasse **DAL_Device** deklariert werden. Die konkreten Implementierungen müssen in den speziellen Geräteklassen angegeben werden. Alle MOST Devices haben eine Reihe von Adressen (logisch, physikalisch und Gruppenadresse), welche von der abstrakten Klasse **MOST_Device** bereitgestellt werden. Im speziellen sollen Hardwaregeräte der Firmen Oasis und Vector Informatik (siehe Kapitel 3.1.2) unterstützt werden. Diese haben unterschiedliche Softwareschnittstellen, die von den zugehörigen Deviceblöcken unterstützt werden müssen. Außerdem sind zum Teil spezifische Geräteeinstellungen (zum Beispiel der comPort eines Optolyzers oder die sourceMask des VN2600 Interfaces) zu speichern. Bei den CAN Geräten werden zu Zeit nur zwei verschiedene Kartentypen von Vector Informatik eingesetzt, welche sich in ihrer Anschlussart unterscheiden. Auch hier werden allgemeine Attribute wie die Datenrate in der gemeinsamen Oberklasse (**CAN_Device**) verwaltet.

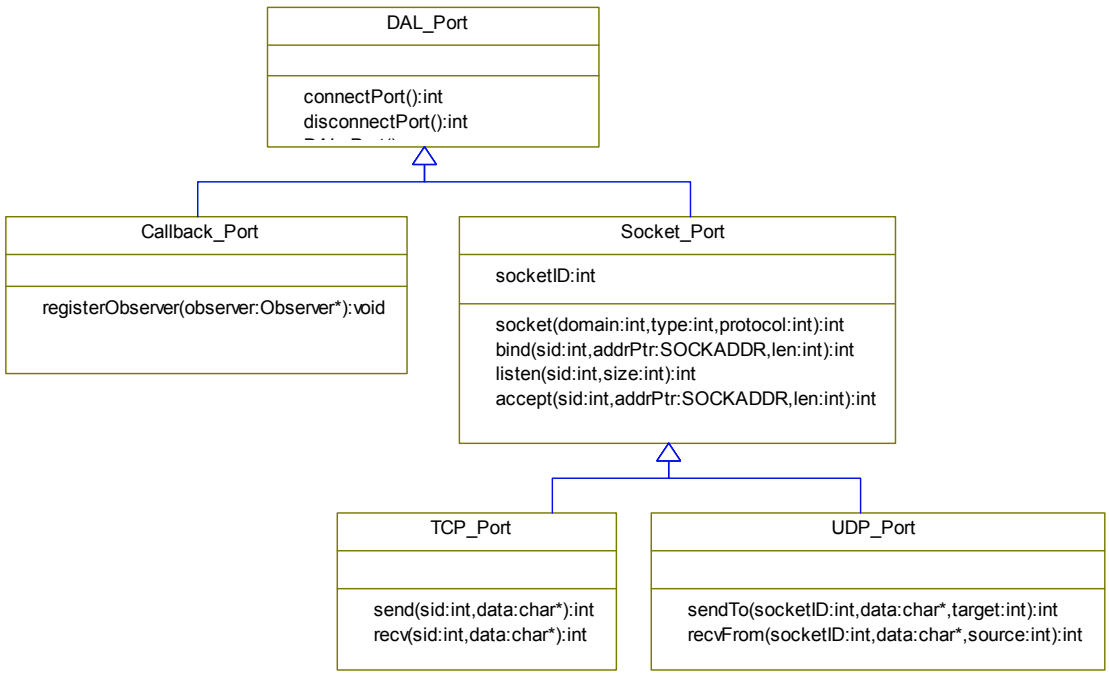


Abbildung 30: DAL Port Klassenhierarchie

Bei den Ports sollen zunächst ebenfalls zwei verschiedene Kategorien untersucht werden. Zum einen einfache Callback Ports, bei denen sich eine Beobachterkomponente einer

Anwendung registrieren kann, um Daten aus dem Kommunikationsframework weitergereicht zu bekommen. Diese Klasse soll später als Teil der Beispielkonfiguration implementiert werden. Daneben könnte es aber auch etwas komplexere Socket Ports geben, welche als Serverkomponenten fungieren mit denen sich Clientkomponenten einer Anwendung verbinden können.
Die allgemeine (abstrakte) Basisklasse **DAL_Port** definiert Methoden, über die man sich von außen mit einem beliebigen Port verbinden kann ohne über die konkrete Portklasse Bescheid zu wissen. Die speziellen Unterklassen müssen diese Methoden dann jeweils implementieren. Während es bei einem Callback Port nur eine weitere Methode (**registerObserver**) gibt, sind für den Verbindungsaufbau und die Kommunikation über eine Socket Verbindung mehrere Methoden nötig. Bei ihnen kann man darüber hinaus noch zwischen verbindungsorientierter (TCP) und paketorientierter (UDP) Kommunikation unterscheiden.
Beide vorgestellten Klassendiagramme können in Zukunft beliebig erweitert werden, damit das Kommunikationsframework auch weitere Möglichkeiten der Anbindung von Softwarekomponenten und Hardwareschnittstellen zur Verfügung stellen kann.
Die hier bisher fehlenden Serviceklassen werden im nächsten Kapitel anhand der bereits erwähnten Beispielkonfiguration erläutert.

3.3.3 Kommunikationsfluss

Anhand zweier UML Sequenzdiagramme soll der Kommunikationsfluss durch das Framework als Bindeglied zwischen Anwendungs- und Hardwareebene verdeutlicht werden.
Im ersten Sequenzdiagramm ist der Weg einer Nachricht oder eines Event vom Bus zur Anwendung dargestellt. Die Hardwareschnittstelle (zum Beispiel der VN2600 von Vector) zeichnet empfangene Nachrichten auf und stellt diese dem zugehörigen Deviceobjekt zur Verfügung. Hierbei gibt es grundsätzlich zwei Möglichkeiten der Kommunikation. Entweder die Nachrichten werden automatisch an registrierte Objekte weitergereicht, die diese dann eventuell vor der weiteren Verarbeitung zwischenspeichern müssen, oder das Deviceobjekt muss sich die Informationen selbst aktiv aus einem Empfangspuffer der Hardwareschnittstelle abholen. Beim VN2600 wird die zweite Variante verwendet, das heißt das zu entwickelnde Device muss neue Nachrichten erfragen.
Die im Diagramm mit MessageReceived bezeichnete Funktion ist daher keine Methode vom Device, welche von außen aufgerufen werden kann sondern beschreibt nur ganz

allgemein den Transport der Daten zwischen Hardwareinterface und Kommunikationsframework.

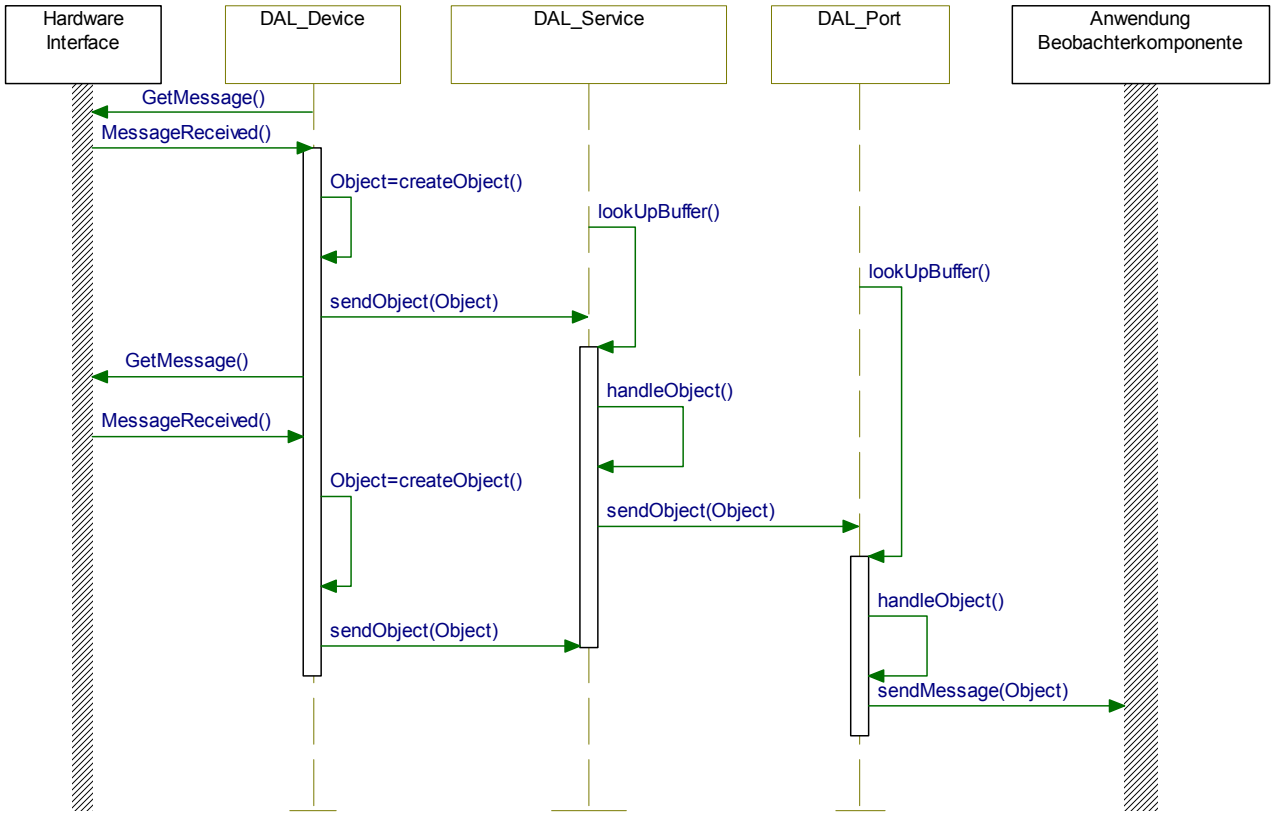


Abbildung 31: Kommunikationsfluss (Bus → Anwendung)

Ein Device muss nun zunächst ein Datenobjekt aus dem erhaltenen Event erzeugen. Hierzu sind später spezielle Mechanismen zu implementieren, da die einzelnen Events je nach verwendeter Hardwareschnittstelle variieren können. Hierauf soll an dieser Stelle aber erstmal nicht weiter eingegangen werden. Die erzeugten Datenobjekte werden anschließend über die bereits genannte interne Schnittstellenfunktion **sendObject** an die nächsten Blöcke weitergeleitet. Der weitere Ablauf ist im Grunde für alle Blocktypen identisch. Jeder Block schaut, ob Datenobjekte in seinem Eingangspuffer liegen und verarbeitet diese in der Methode **handleObject**. Am Ende der Bearbeitung sendet er die Objekte wiederum an die nachfolgenden Blöcke seiner Filterkette weiter. Vom Port schließlich werden die Datenobjekte der Anwendung zur Verfügung gestellt, die diese eventuell erneut in ein eigenes internes Format umwandelt.
Das zweite Sequenzdiagramm zeigt den umgekehrten Weg von der Anwendung über das Kommunikationsframework zur Hardware.

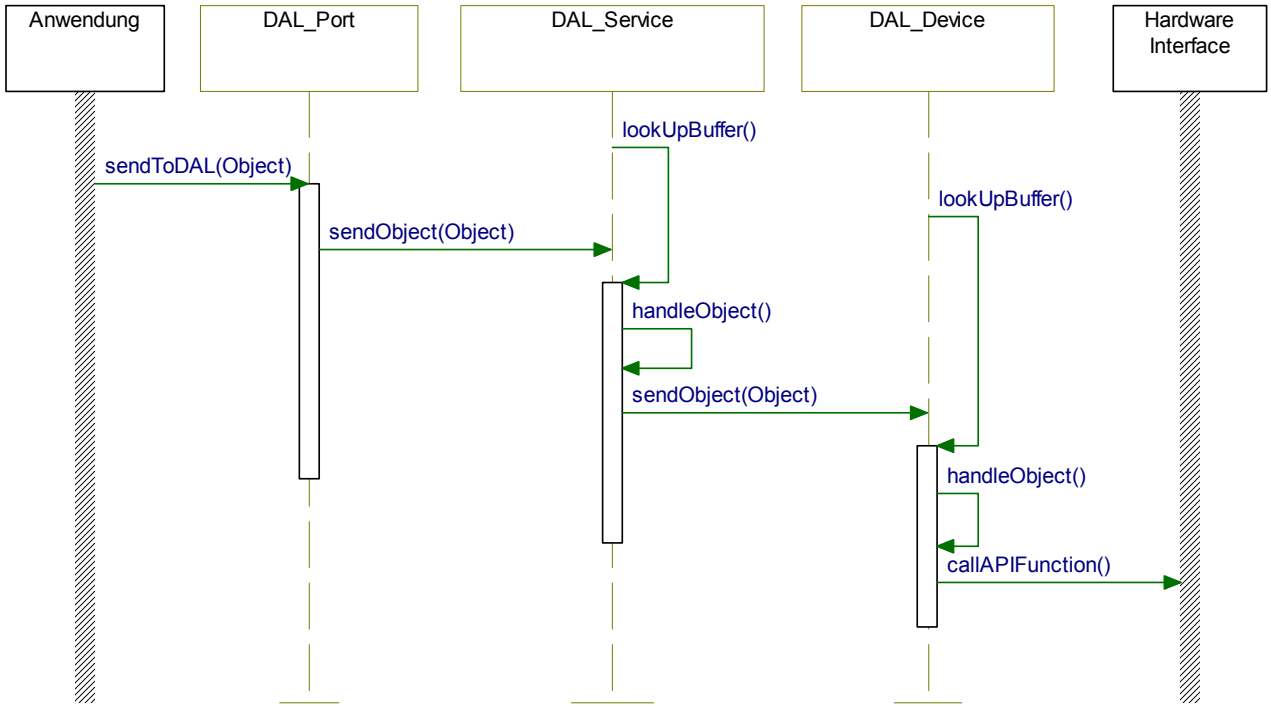


Abbildung 32: Kommunikationsfluss (Anwendung → Bus)

Hier werden die Datenobjekte zunächst über die Methode **sendToDAL** an einen Port übermittelt. Das bedeutet, die Anwendung muss bereits eine Komponente bereitstellen die aus internen Events für den Port verständliche Objekte generiert. Der Port führt selbst keine Operation auf den Datenobjekten aus, sondern sendet sie direkt an die mit ihm verbundenen Blöcke weiter. Anschließend werden die Objekte wie oben für die andere Richtung beschrieben durch das Framework transportiert bis sie bei einem Device angekommen sind. Hier werden die Objekte (in der **handleObject** Methode) interpretiert und die entsprechenden API Funktionen der Hardwareschnittstelle aufgerufen um die Nachricht auf den Bus zu bringen.
Bei beiden Richtungen ist zu beachten, dass jeder Block in einem eigenen Prozess (bzw. Thread) laufen soll um parallele Verarbeitungen zu ermöglichen. Deshalb gibt es in der Praxis keine rein sequenzielle Kommunikationsabfolge, wie es aus den Diagrammen abzulesen wäre. Stattdessen treffen durchaus mehrere neue Datenobjekte bei einem Block ein während dieser noch mit der Bearbeitung des aktuellen Objektes beschäftigt sein kann. Aufgrund dieser Tatsache wird später im Einsatz noch zu untersuchen sein, in wieweit die Datenpuffer der Blöcke dimensioniert werden müssen, um möglichst keine Objekte zu verlieren.

Die folgende Abbildung zeigt die möglichen Datenobjekte die durch das

Kommunikationsframework transportiert werden können.

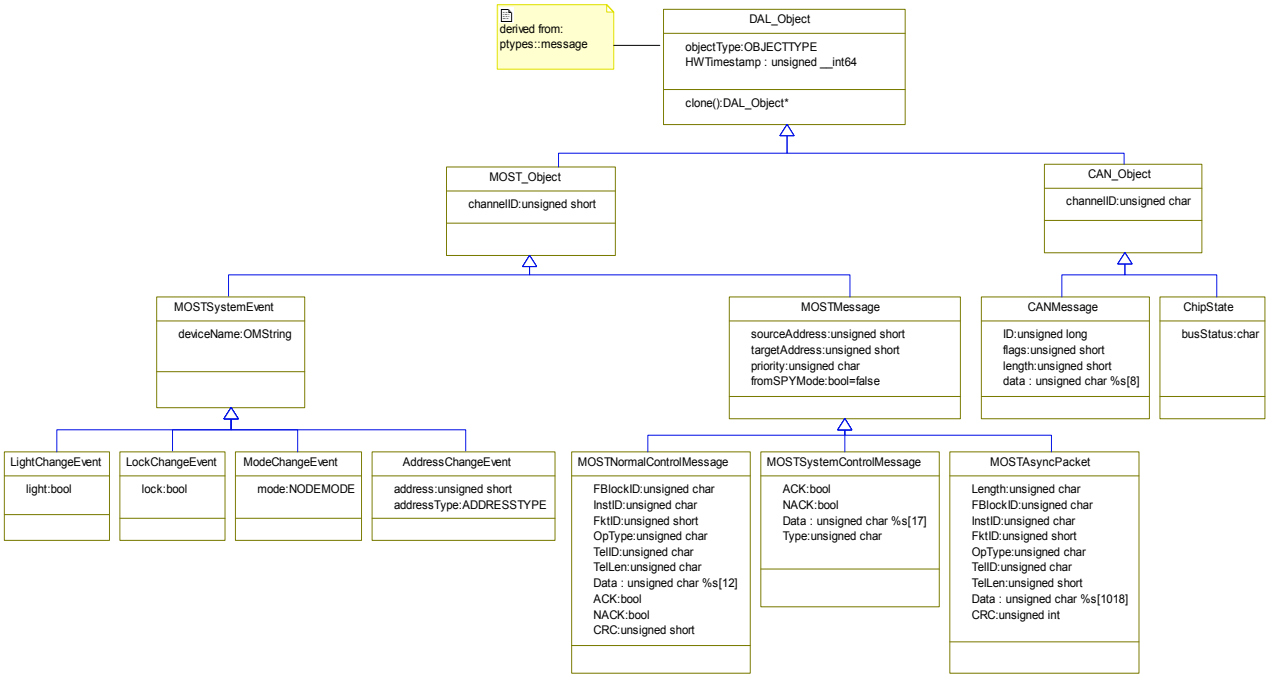


Abbildung 33: Datenobjekte

Die abstrakte Basisklasse **DAL_Object** wird von ptypes::message abgeleitet, damit die Datenobjekte von den verwendeten Nachrichtenpuffern (ptypes::msgqueue – siehe nächstes Kapitel) aufgenommen werden können. Der Objekt Type dient zur Identifizierung spezieller Objekte. Daneben wird für alle Datenobjekte die Nummer des Kanals gespeichert auf dem die Nachricht empfangen wurde bzw. für den sie gedacht ist (unterschiedliche Größen bei MOST und bei CAN Objekten) und es wird ein Empfangs-Zeitstempel eingetragen.

Nun können Datenobjekte zunächst in die beiden Kategorien MOST und CAN Objekte unterteilt werden, wobei letztere in dieser Arbeit nicht implementiert werden sollen. Hier gibt es ein Objekt für normale CAN Nachrichten welches deren eindeutige Kennzeichnung und die übertragenen Daten aufnehmen kann. Außerdem treten spezielle Events auf, die über den aktuellen Busstatus informieren.

MOST Objekte können wiederum in zwei Unterkategorien (**MOSTSystemEvent** und **MOSTMessage**) eingeteilt werden. System Events können verschiedene Statusmeldungen darstellen, bei denen jeweils ein bestimmter Zustand (z.B. Licht am Eingang / Ausgang des Steuergerätes) angegeben wird. Es werden vier verschiedene Datenobjekte definiert (**LightChangeEvent**, **LockChangeEvent**, **ModeChangeEvent** und **AddressChangeEvent**), über welche eine Anwendung über Zustände und Eigenschaften einer Hardwareschnittstelle informiert werden oder diese selbst ändern kann. Darüber

hinaus wird in den Objekten jeweils der Name des Devices gespeichert von dem ein Event beobachtet wurde bzw. an das die Nachricht gerichtet ist.
Die Klasse MOST Messages definiert Attribute für die Adressen der Quell- und Zielgeräte, sowie eine Priorität. Außerdem wird hier eine Variable bereitgestellt, um zu hinterlegen, ob eine Nachricht von einem Knoten oder über den so genannten Spymode empfangen wurde. Hier müssen wiederum drei verschiedene Nachrichtentypen untersucht werden. Zunächst soll das Hauptaugenmerk auf den **MOSTNormalControlMessages** liegen. In diesem Objekt werden die einzelnen Bestandteile einer MOST Kontrollnachricht übertragen. Daneben werden noch Objekte für System Kontrollnachrichten und asynchrone Datenpakete definiert. Diese Nachrichten unterscheiden sich in der Anzahl und Größe ihrer Attribute. Der genaue Aufbau der einzelnen MOST Nachrichten wurde bereits in Kapitel 3.1.1 erläutert (siehe auch [MOS-05]).
Die MOST Datenobjekte sollen alle im Verlauf dieser Arbeit implementiert und getestet werden.

4 Realisierung des Kommunikationsframeworks

Aufbauend auf dem Design des Kommunikationsframeworks, welches im vorherigen Kapitel [3] erläutert wurde, soll in diesem Kapitel nun die konkrete Architektur und Umsetzung der im Laufe dieser Arbeit implementierten Blockklassen und Datenobjekte aufgezeigt werden.

4.1 Architektur

Das Kommunikationsframework besteht allgemein aus den drei Blocktypen Ports, Services und Devices, welche beliebig miteinander verschaltet werden können. Eine Administrator Klasse dient zur dynamischen Erzeugung und Verlinkung der benötigten Verarbeitungsblöcke.

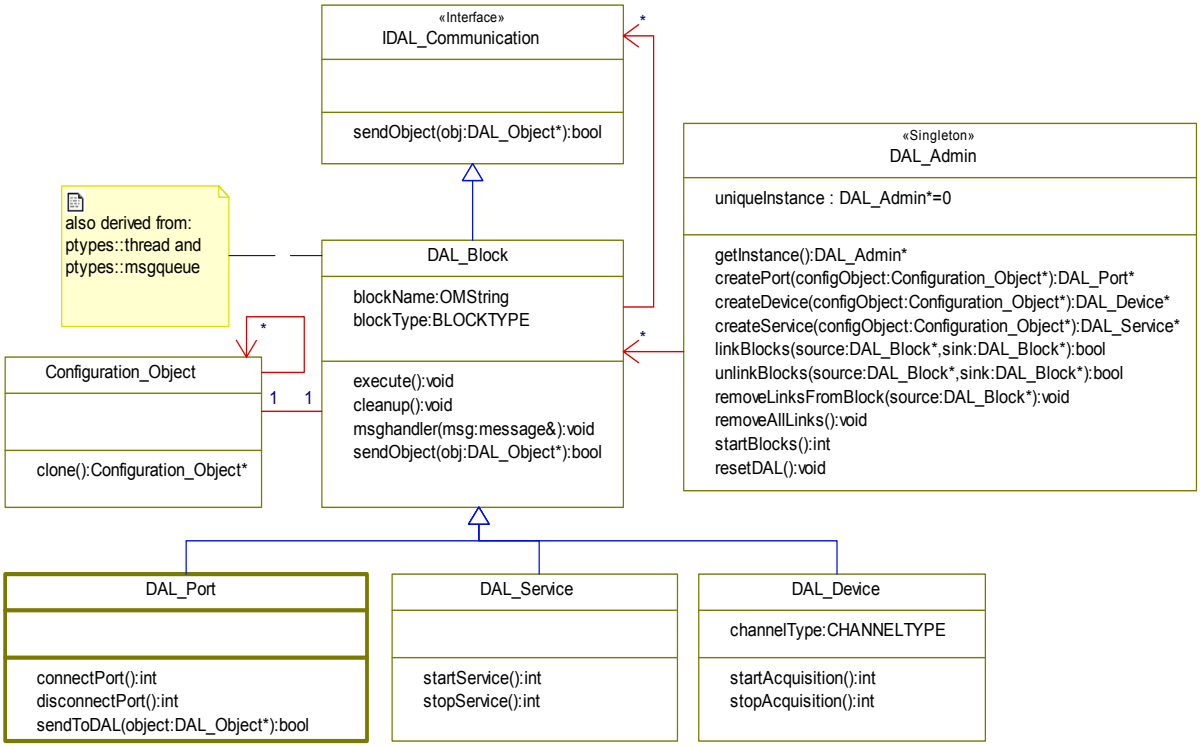


Abbildung 34: Kommunikationsframework (Schnittstellen und Basisklassen)

Um die flexiblen Kombinationsmöglichkeiten aller Blöcke untereinander realisieren zu können, wird eine einheitliche Schnittstelle verwendet. Das Interface **IDAL_Communication** (siehe Abbildung 34) definiert dazu eine rein virtuelle Methode **sendObject** über die Datenobjekte (**DAL_Objects** – siehe weiter unten) zwischen

beliebigen Blöcken weitergeleitet werden können.
Die drei verschiedenen Blocktypen des Frameworks werden alle von der gemeinsamen Basisklasse **DAL_Block** abgeleitet. Diese abstrakte Klasse implementiert die von **IDAL_Communication** geerbte Methode **sendObject** und stellt damit eine universelle Kommunikationsmöglichkeit zwischen den Blöcken zur Verfügung.
Jeder Block kann mit beliebig vielen weiteren Blöcken verbunden sein, welche wiederum alle die gemeinsame Schnittstelle **IDAL_Communication** realisieren müssen. Die Klasse DAL_Block leitet sich zudem von den Klassen **thread** und **msgqueue** der PTypes Bibliotheken (C++ Portable Types Library [INT-11]) ab. Hierdurch befindet sich jeder Block des Frameworks später in einem eigenen Thread und kann parallel zu den anderen Blöcken Datenobjekte verarbeiten. Aufgrund dieser Parallelität wird auch in jedem Block ein Nachrichtenpuffer (die Message Queue) benötigt, um Datenobjekte zwischenspeichern zu können.
Die Methoden **execute** und **cleanup** müssen aufgrund der Vererbung von ptypes::thread implementiert werden. **Execute** startet eine Endlosschleife (*msgqueue::run()*), in der das jeweils aktuelle Datenobjekt aus dem Puffer gelesen und zur weiteren Bearbeitung an die Methode **msghandler** weitergereicht wird. Die beiden Attribute **blockName** und **blockType** dienen zur eindeutigen Identifizierung spezieller Blöcke.
Zur Konfiguration des Kommunikationsframeworks dient die Singletonklasse **DAL_Admin.** Diese stellt Methoden zur Erzeugung und Verknüpfung von Blöcken zur Verfügung. Die Methoden **createDevice**, **createPort** und **createService** verwenden Konfigurationsobjekte die der Anwender anlegen muss, um die entsprechenden konkreten Blöcke zu erstellen. Um die Informationen dieser Objekte (siehe Abbildung 33) in den erzeugten Blöcken nicht komplett neu speichern zu müssen, wird für jeden Block ein Verweis auf das entsprechende Konfigurationsobjekt (und umgekehrt) hinterlegt.
Die drei Methoden zum setzen und löschen von Verbindungen (**link/unlinkBlocks**, **removeLinksFromBlock** und **removeAllLinks**) des Administrators verändern die Liste der verbundenen Blöcke (bzw. Kommunikationsschnittstellen) der jeweiligen Blöcke. Der Administrator speichert sich dabei eine Liste aller Blöcke die er angelegt hat. Die Methode **startBlocks** dient sowohl zum starten der jeweiligen Block - Threads, als auch zur Initialisierung der vorhandenen Blöcke. Hierzu werden die speziellen Methoden **connectPort**, **startService** und **startAcquisition** der konkreten Blockklassen aufgerufen, welche jeweils von den entsprechenden Basisklassen **DAL_Port**, **DAL_Service** und **DAL_Device** abgeleitet worden sind.

Zum Testen des Kommunikationsframeworks wurden im Verlauf dieser Arbeit vier

konkrete Blöcke definiert und implementiert, die im folgenden Klassendiagramm abgebildet sind.

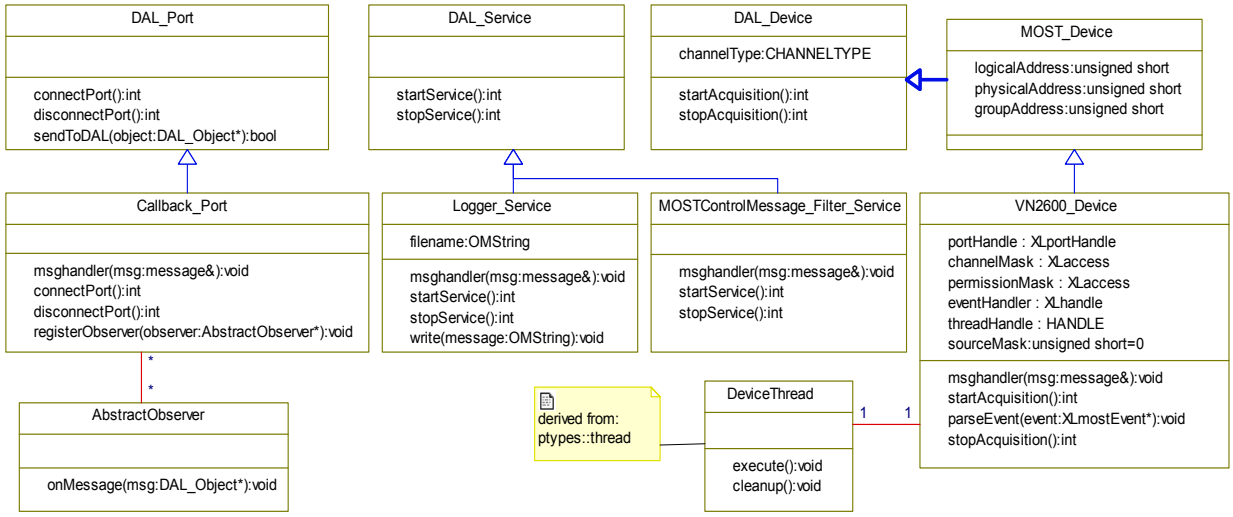


Abbildung 35: konkrete Blockklassen

Das **VN2600_Device** repräsentiert das gleichnamige MOST Hardware Interface der Firma Vector Informatik. Diese Blockklasse wird wie bereits im letzten Kapitel erläutert nicht direkt von der allgemeinen Basisklasse DAL_Device, sondern von der zusätzlichen (ebenfalls abstrakten) Klasse MOST_Device abgeleitet. Dieser Block implementiert die bereits mehrfach genannte Methode **msghandler**, welche hier interne Datenobjekte verarbeitet und entsprechende Nachrichten an die Hardware sendet. Hierzu werden spezifische API Funktionen der Hardwareschnittstelle aufgerufen, die dem Device durch die Einbindung einer entsprechenden externen Bibliothek bekannt gemacht werden müssen. Um parallel zu der Verarbeitung interner Datenobjekten Events vom Bus an die höher liegenden Blöcke des Kommunikationsframeworks weiterreichen zu können, besitzt das Device einen zusätzlichen Thread (Klasse **DeviceThread**), der wiederum von ptypes::thread abgeleitet ist. In diesem Thread wird zyklisch der Empfangspuffer der Vector Hardware ausgelesen. Die so empfangenen Nachrichten werden direkt an die Methode **parseEvent** weitergeleitet, wo sie auf das interne Datenformat (DAL_Objects, siehe Abbildung 33) abgebildet und an die mit dem VN2600_Device verbundenen Blöcke (in unserer Testkonfiguration entweder direkt ein Callback_Port oder beliebige Services) weitergeleitet. Bevor Nachrichten mit der Hardware ausgetauscht werden können muss das VN2600_Device über die Methode **startAcquisition** mit einem Kanal der VN2600 Hardwareschnittstelle verbunden werden. Hierbei werden im Device einige Variablen initialisiert.

Über einen **Callback_Port** werden die empfangenen Datenobjekte (vom VN2600_Device bzw. von Services) an die Anwendung (in unserem Fall MODENA) weitergeleitet. Dies wird wiederum durch die Methode **msghandler** realisiert. Diese ruft in einer Schleife eine **onMessage** Methode der beim Port registrierten Beobachter auf, welcher das jeweils aktuelle Datenobjekt als Parameter übergeben wird. Für die benötigten Beobachterklassen wird innerhalb des Kommunikationsframeworks nur eine abstrakte Basisklasse (AbstractObserver) definiert, die ausschließlich diese eine Methode enthält. Um Datenobjekte aus der DAL zu erhalten muss die darüber liegende Anwendung eine konkrete Beobachterkomponente erzeugen, die von dieser Oberklasse abgeleitet ist und sie über die Methode **registerObserver** beim Port registrieren. Außerdem sollen von der Anwendung erzeugte Events in das Framework eingebracht werden können, welche zuvor anwenderseitig bereits auf das interne Datenformat abgebildet werden müssen. Die entstandenen Datenobjekte werden nun vom Port direkt an die verbundenen Blöcke weitergeleitet. Da die Anwendung im Fall eines einfachen Callback Ports mit dem Senden eines weiteren Objekts warten soll bis die Weiterleitung des letzten Events bestätigt wurde ist hier im Gegensatz zum VN2600_Device kein weiterer Thread und auch kein zusätzlicher Puffer nötig. Die beiden geerbten Methoden **connectPort** und **disconnectPort** enthalten bei dieser einfachen Portklasse keine weiteren Funktionalitäten, da kein expliziter Verbindungsauf- bzw. abbau realisiert werden muss.

Mit diesen beiden Blöcken kann bereits ein erster Systemtest durchgeführt werden. Dabei werden die Nachrichten zunächst ohne weitere Verarbeitung auf direktem Weg zwischen Port und Device transferiert. In einem weiteren Schritt sollen dann zwei einfache Services implementiert werden, welche zwischen den VN2600_Device und den Callback_Port geschaltet werden können. Mit Hilfe dieser Services soll gezeigt werden, dass sich das entwickelte Kommunikationsframework relativ leicht erweitern lässt, um zusätzliche Verarbeitungen der empfangenen Nachrichten und Events zu realisieren. Als erstes wird ein **Logger_Service** entworfen, der alle eingehenden Informationen in eine vom Nutzer festzulegende Ausgabedatei schreibt und die Datenobjekte anschließend unverändert weiterreicht. Die Methoden **startService** und **stopService** können hierbei zum öffnen und schließen des Ausgabestroms zum Beginn bzw. zum Ende der Datenaufzeichnung verwendet werden.
Als zweiter Service soll anschließend noch ein einfacher Filter implementiert werden, der MOST Kontrollnachrichten zunächst nach einem einzelnen Kriterium (etwa der Adresse des Absenders) weiterleitet oder verwirft (**MOSTControlMessage_Filter_Service**). Bei beiden Services wird die eigentliche Verarbeitung der Datenobjekte wiederum in der

msghandler Methode durchgeführt. Anschließend werden die Objekte wie bei jedem Block des Kommunikationsframeworks über die Methode **sendObject** an die Eingangspuffer der nachfolgenden Blöcke weitergereicht.

Nachdem die zu erstellenden Blockklassen definiert wurden, muss man sich als nächstes Gedanken über die entsprechenden Konfigurationsobjekte machen die benötigt werden, um die oben beschriebenen Blöcke mit Hilfe des Administrators zu erzeugen.
Konfigurationsobjekte sollen vom Benutzer aus der jeweiligen Anwendung heraus angelegt werden, um die gewünschten Instanzen der Blocktypen zu initialisieren. Dafür müssen spezifische Startparameter festgelegt werden, die ein Anwender eingeben muss um den entsprechenden Verarbeitungsblock zu erhalten.
Jedes Konfigurationsobjekt besitzt zunächst einen Objecttyp, welcher in den create Methoden des Administrators dazu benutzt wird den passenden Block zu instanziieren und den Namen des zu erzeugenden Blockes, welcher später zur Identifizierung und Unterscheidung verwendet wird.

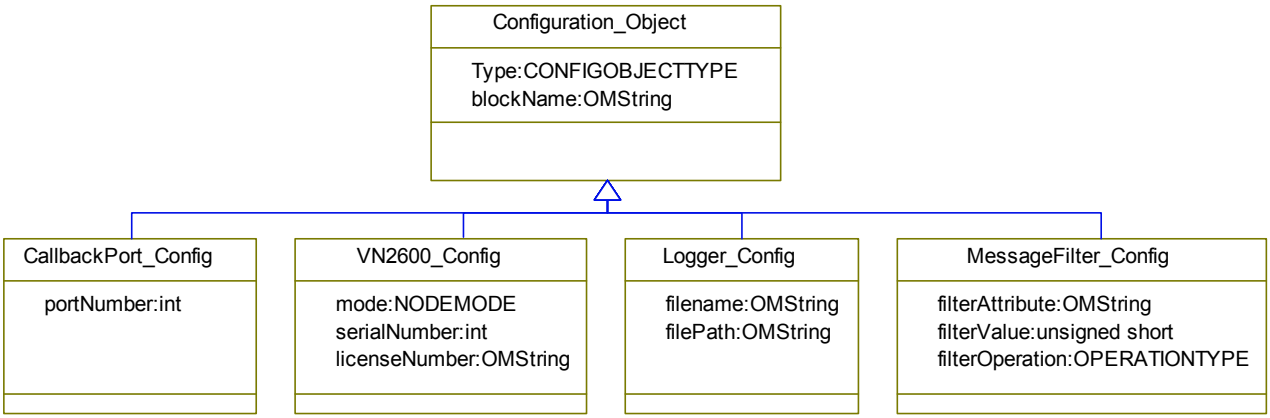


Abbildung 36: Konfigurationsobjekte

Ein Callback Port benötigt eigentlich keine Startparameter und wird lediglich mit einer Portnummer instanziiert. Beim VN2600 Device muss der Benutzer neben einer Hardware - Identifizierungsnummer und einer optionalen Lizenz den gewünschten Betriebsmodus des Gerätes angeben. Um eine VN2600 Hardwareschnittstelle parallel in zwei Modi (Master oder Slave + Spymode) zu betreiben, müssen demnach zwei Konfigurationsobjekte angelegt werden. Über einen gemeinsamen Blocknamen erkennt der Administrator in diesem Fall, dass es sich um ein einziges Gerät handelt und demnach auch nur eine Instanz des VN2600 Devices erstellt werden soll. Für den Logger Service muss der Name und der Pfad der zu verwendenden Ausgabedatei eingetragen werden und beim Nachrichtenfilter muss der Benutzer angeben, nach welchem Attribut (z.B.

Senderadresse) die Kontrollnachrichten gefiltert werden sollen. Neben dem zu überprüfenden Attributnamen muss noch ein Referenzwert und eine Vergleichsoperation (gleich, ungleich, größer, …) angegeben werden.
Wenn der Administrator aus einem Konfigurationsobjekt das entsprechende Blockobjekt erzeugt, wird ein bidirektionaler Verweis zwischen beiden Objekten angelegt.
Zunächst sollen die oben definierten Attribute der Konfigurationsobjekte lediglich für die Instanziierung der Blockklassen dienen. Später könnte man sich dann Gedanken darüber machen, ob es teilweise sinnvoll wäre Konfigurationsparameter dynamisch zur Laufzeit ändern zu können. Als Beispiel hierfür soll im Verlaufe der Systemtest der Betriebsmodus des VN2600 Devices durch versenden eines speziellen Events umgeschaltet werden.

4.2 Implementierung

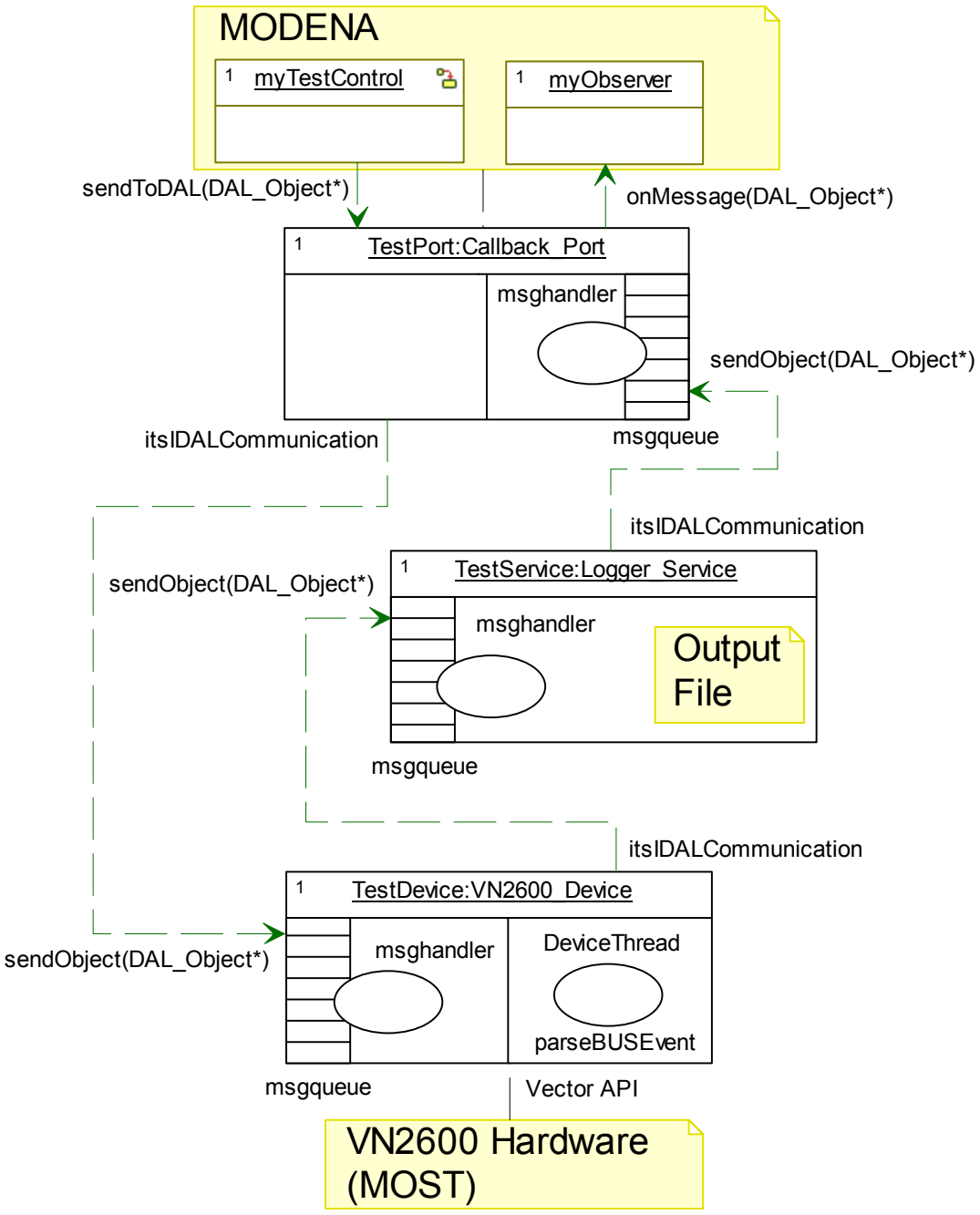


Abbildung 37: beispielhafter Datenfluss

In der obigen Abbildung wird der Datenfluss des Kommunikationsframeworks anhand der im Folgenden beschriebenen Beispielkonfiguration dargestellt. Die verwendeten konkreten Blockklassen für Port, Service und Device wurden im Laufe dieser Arbeit komplett implementiert. Für die Interaktion mit MODENA wurde zunächst eine eigene Testklasse (**myTestControl**), sowie eine Beobachterklasse erstellt, um die DAL unabhängig von

MODENA testen zu können. In einem weiteren Schritt wurde MODENA dann angepasst, um direkt diese Funktion zu übernehmen und damit mit der DAL verbunden werden zu können.

Eine Instanz der Klasse Callback_Port nimmt Datenobjekte von der Anwendung entgegen und sendet diese an das nachfolgende VN2600_Device (**itsIDALCommunication**) weiter (**sendObject**). Hierzu muss die Anwendung die Methode sendToDAL des Ports aufrufen und ihm ein Datenobjekt als Parameter übergeben. Dort werden die Datenobjekte zunächst im Eingangspuffer (**msgqueue**) abgelegt. Die Methode **msghandler** holt sich das aktuelle Datum aus diesem Puffer, interpretiert es und sendet die entsprechenden Befehle an die Hardwareschnittstelle. Parallel dazu werden Nachrichten aus dem Empfangspuffer des VN2600 abgefragt. Hierzu wird der interne DeviceThread verwendet, der alle Events an die Methode **parseEvent** des VN2600 Devices weitergibt. Dort werden entsprechende Datenobjekte generiert und wiederum an die nachfolgenden Blöcke weitergeleitet. In der obigen Beispielkonfiguration also an die Eingangsschnittstelle des Logger Services. Dieser verarbeitet die Objekte aus seinem Eingangspuffer indem er die Inhalte in eine Ausgabedatei schreibt und die unveränderten Datenobjekte an den nächsten Block (in diesem Fall wieder den Callback_Port) sendet. Von hier werden die empfangenen Objekte in der **msghandler** Methode an eine Beobachterkomponente der Anwendung (MODENA) weitergeleitet, die sich zuvor beim Port registriert haben muss.

Für die Konstruktion der DAL (besser gesagt ihrer Modelle) wurde wie bereits erwähnt die Entwicklungsumgebung Rhapsody für C++ von I-Logix verwendet. Diese erzeugt automatisch ein Codegerüst aus den erstellten Modellen, welches bereits die modellierten Beziehungen zwischen den einzelnen Klassen realisiert. Bei der Codegenerierung muss darauf geachtet werden, dass keine speziellen Datentypen oder Ausdrücke von Rhapsody verwendet werden, da der erzeugte Code sonst von der Entwicklungsumgebung abhängig werden würde. Rhapsody stellt zum Beispiel spezielle Datentypen für Strings oder Listen (OMCollection) zur Verfügung, welche standardmäßig durch die Einbindung entsprechender Bibliotheken hinzugelinkt werden. Damit das erstellte Kommunikationsframework später nicht auf diese speziellen Bibliotheken angewiesen ist, müssen diese Rhapsodytypen durch Standardtypen (std::list) ersetzt werden. Die somit verwendeten Standardbibliotheken sind unabhängig von der Entwicklungsumgebung und der Plattform auf der sie erstellt wurden einsetzbar.

Zusätzlich zu den automatisch generierten Klassen und Abhängigkeiten, müssen die erhaltenen Codedateien mit entsprechender Funktionalität gefüllt werden.
Der folgende Auszug aus dem Code des Logger Services zeigt die von allen Blockklassen verwendete Methode msghandler, über die Datenobjekte aus dem Eingangspuffer eines Bausteins gelesen und verarbeitet werden. Diese Methode wird von der eingebundenen externen Klasse msgqueue der PTypes Library [INT-14], welche zur Verwaltung von Puffern eingesetzt wurde, vererbt.

```
void Logger_Service::msghandler(message& msg) {

    std::stringstream stream;
    std::string message = "";

    switch (msg.id)
    {
        case DAL_MSG:
        {
          // create log message
          int objectType = ((DAL_Object&)msg).getObjectType();
          switch(objectType)
          {
              case LIGHT_CHANGE_EVENT:
              {
                if (((LightChangeEvent&)msg).getLight())
                {
                    message = "MOST Light On";
                }
                else
                {
                    message = "MOST Light Off";
                }
                break;
              }
              …

          } // switch object Type
          write(message);
          break;
        } // case DAL_MSG
        default : defhandler(msg);
    }
}
```

Abbildung 38: Beispielcode – msghandler Methode

Zunächst wird über eine switch Anweisung der Typ der erhaltenen Nachricht überprüft. Dabei kennen die implementierten Blöcke explizit lediglich einen Nachrichtentyp (DAL_MSG) welcher ein internes Datenobjekt kennzeichnet. Über die default Verzweigung am Ende der Anweisung werden alle anderen Nachrichten an einen bereits von der Oberklasse msgqueue realisierten Handler weitergereicht. Dieser wird vor allem dazu

verwendet, die Abarbeitung von Pufferelementen, welche in einer Endlosschleife abläuft, zu beenden. Hierfür kann eine spezielle Systemnachricht (MSG_QUIT) an eine Pufferklasse versendet werden. Innerhalb der Verzweigung für Datenobjekte der DAL wird nun über eine weitere switch Anweisung der Typ des Objektes abgefragt und das Datenobjekt entsprechend verarbeitet. Im obigen Beispiel des Logger Services wird zum Beispiel bei einem empfangenen Light Change Event eine Meldung (Light On / Off) gespeichert und über eine weitere Methode (write – nicht genauer angegeben) ausgegeben. Für jeden Objekttype der von dem aktuellen Block unterstützt werden soll, muss eine Verzweigung implementiert werden. An dieser Stelle müssten vorhandene Blöcke auch später erweitert werden, um weitere Datenobjekte verarbeiten zu können.

5 Zusammenfassung und Ausblick

5.1 Ergebnisse

Im Rahmen dieser Diplomarbeit wurde ein Konzept für ein generisches Kommunikationsframework zur flexiblen Anbindung von Infotainment Steuergeräten entworfen und Anhand einer konkreten Beispielkonfiguration umgesetzt und verifiziert.

Für die Entwicklung der DAL wurden zunächst verschiedene Architekturansätze untersucht, von denen einige Punkte in das eigene Konzept eingeflossen sind. So ist das entstandene Kommunikationsframework etwa einer Service – Orientierten Architektur nachempfunden, in der unabhängige Dienste (Ports, Services und Devices) miteinander verbunden werden können und über eine einheitliche Schnittstelle Datenobjekte austauschen. Dabei werden zunächst allerdings keine verteilten Dienste, wie eigentlich bei SOA üblich betrachtet (siehe SOA und Web Services), sondern die einzelnen Bausteine eines zentralen Systems. Außerdem gibt es Dienste (insbesondere die Devices), die auf geplante oder zufällige Ereignisse (vom Bus) regieren müssen. Ein weiterer wichtiger Ansatz bei der Entwicklung des entstandenen Kommunikationsframeworks war die Modell betriebene Architektur (MDA). Die DAL wurde in Form von UML Diagrammen (insbesondere Klassendiagramme) in der Entwicklungsumgebung Rhapsody für C++ modelliert, mit deren Hilfe automatisch ein Codegerüst zur weiteren Bearbeitung entstand.

Die Ziele der Diplomarbeit, insbesondere die Erstellung eines schlüssigen Konzepts für eine Daten Abstraktionsschicht wurden erreicht. Durch die in Kapitel 4 beschriebene Realisierung einer Beispielkonfiguration wurde die Umsetzbarkeit des Konzepts nachgewiesen. Hierzu wurde eine spezielle Hardwareschnittstelle, nämlich der VN2600 der Firma Vector Informatik, sowie die von Berner & Mattner entwickelte Modellierungssoftware MODENA über das Framework verbunden. Der Versand sowie der Empfang von MOST NormalControlMessages über die DAL wurde komplett realisiert und anhand eines vorhandenen Testmodells getestet. Hierbei wurden bereits erste Fehler- und Performanztests durchgeführt.

Die Erweiterbarkeit des Systems durch hinzufügen neuer Bausteine (insbesondere Devices und Services) wurde anhand eines einfachen Logger Services, welcher zwischen Port- und Deviceschnittstellen gelinkt werden kann, nachgewiesen. Hierbei müssen neue Blöcke von den bestehenden Basisklassen abgeleitet werden und mindestens die dort

deklarierten Attribute und Methoden implementieren. Da die Kommunikation zwischen den Blöcken über eine fest vorgegebene einheitliche Schnittstelle realisiert ist, können neue Bausteine die diese Schnittstelle einhalten ohne Änderung des Systemkonzepts in das Framework eingebunden werden.

5.2 Ausblick

Da im Rahmen dieser Diplomarbeit der Schwerpunkt auf der Entwicklung eines Konzepts und nicht in der reinen Implementierung lag, ist noch keine direkt voll einsatzfähige, fertige Software entstanden. Um das Framework mit MODENA für weitere Szenarien außer der zum Nachweis der Umsetzbarbarkeit realisierten Beispielkonfiguration einsetzen zu können, müssen weitere Implementierungen durchgeführt werden. Insbesondere die Unterstützung weiterer Nachrichten und Events neben den bereits umgesetzten MOST Kontrollnachrichten, müsste in einem weiteren Schritt realisiert werden. Die benötigten Datenobjekte sind im entwickelten Konzept bereits vorgesehen und werden teilweise auch schon von den speziellen Blockklassen interpretiert.

Weiteren zukünftigen Entwicklungsspielraum bietet die Unterstützung beziehungsweise Anbindung weiterer Hardwareschnittstellen und Anwendungen, sowie zusätzlicher Services zur internen Verarbeitung der Datenobjekte. Hier ist zum Beispiel die bereits in dieser Arbeit erwähnte Optolyzer Schnittstelle der Firma Oasis zu nennen, welche ähnlich des VN2600 MOST Signale verarbeitet und über die serielle Schnittstelle angesprochen werden kann. Als zusätzlicher Service könnte zum Beispiel der „Advanced Message Service (AMS)“ von MOST implementiert werden, welcher die Segmentierung längerer Nachrichten übernimmt.
Aus Sicht der Anwendung könnte es in Zukunft neben den Callback Ports, bei denen sich eine Anwendung registrieren muss, um Daten hochgereicht zu bekommen weitere Anbindungen geben (etwa eine entfernte Kommunikation über eine Socket-Verbindung).

Literaturverzeichnis

Bücher und Zeitschriften:

BMS-05	MODENA Präsentation	Berner & Mattner, Dezember 2005
DOH-02	Bussysteme im Automobil CAN, FlexRay und MOST	Th. Dohmke, TU Berlin März 2002
ENG-02	CAN-Bus: Feldbusse im Überblick	Horst Engels, Franzis 2002
GAM-96	Entwurfsmuster – Elemente wieder verwendbarer objektorientierter Software	E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1996 (Deutsche Übersetzung von D. Riehle)
MOS-06	Schulung MOST-Bus	Markus Freutsmiedl, Berner & Mattner, Februar 2006
MÜL-04	Infotainment, Telematik im Fahrzeug: Trends für die Serienentwicklung	Ch. Müller-Bagehl, Export-Verlag 2004
RAN-02	Workshop „Bussysteme im Automobil“	Michael Randt, ECT 2002 Augsburg, Kapitel 5
SHA-96	Software Architecture - Perspectives on an Emerging Discipline	M. Shaw, D. Garlan, Prentice-Hall, 1996
ULR-01	Informationstagung Landtechnik – CAN – Bussysteme	Th. Ulrich, Schweizer Technische Fachschule Winterthur, Oktober 2001

Internet:

BOS-01	CAN (Bosch)	http://www.semiconductors.bosch.de/de/20/can/index.asp
BOS-02	CAN – das Netzwerk für die Elektronik im Kraftfahrzeug	Robert Bosch GmbH, Stuttgart 1999 http://www.semiconductors.bosch.de/pdf/can.pdf

FLE-03	FlexRay International Workshop, Detroit 03/2003	http://www.flexray.de/products/automotive%20application%20requirements.pdf
INT-01	CAN Specification 2.0B, Bosch	http://www.semiconductors.bosch.de/pdf/can2spec.pdf
INT-02	MOST Cooperation	http://www.mostcooperation.com/index.php
INT-03	FlexRay	http://www.flexray.com
INT-04	TZ Mikroelektronik – FlexRay Einführung	http://www.tzm.de/FlexRay/FlexRay_Introduction.html
INT-05	UML - Softwarepraktikums Projekt, Uni Magdeburg	http://ivs.cs.uni-magdeburg.de/~lother/Teaching/Slides/UML2SP/index1.html
INT-06	Vorlesung Software-Engineering, TUHH, UML 2.0	http://www.sts.tu-harburg.de/~r.f.moeller/lectures/se-ss-05/07-Spezifikation-UML-Teil-3.pdf
INT-07	I-Logix Rhapsody	http://www.ilogix.com/homepage.aspx
INT-08	OASIS Silicon Systems	http://www.oasis.de/eng/content/view/2/25/
INT-09	Vector Informatik	http://www.vector-informatik.de/deutsch/
INT-10	Enterprise Application Integration - EAI	http://www.torsten-horn.de/techdocs/eai.htm#KomponentenEAILösung
INT-11	SOA, A. Kowallik, SS2004	http://www.ibr.cs.tu-bs.de/courses/ss04/svs/Folien_SOA.pdf
INT-12	SOA und EDA, TIBCO 2005	http://www.competence-site.de/eaisysteme.nsf/959A27B2A7FD8C7BC12570890033BF33/$File/echtzeit-geschäftsprozesse%20durch%20soa%20und%20eda_whitepaper.pdf
INT-13	Codeproject - Patterns	http://www.codeproject.com/useritems/applyingpatterns.asp
INT-14	C++ Portable Types Library	http://www.melikyan.com/ptypes/
MAG-01	Hanser Automotive 09-10/2005	http://www.hanser-automotive.de/fileadmin/heftarchiv/2004/7883.pdf

MOS-05	MOST-Specification (Rev 2.4)	http://www.mostcooperation.com/downloads/Specifications/MOST%20Specifications/MOSTSpecification.pdf
SWA-05	Software Architectures – TU Hamburg – Harburg	http://www.sts.tu-harburg.de/teaching/ss-05/SWArch/material/SWArch-4-PipesandFilter.pdf
UML-20	UML	http://www.uml.org