Aspektorientierte Implementierung eines an AUTOSAR COM angelehnten Moduls für die CiAO-Betriebssystemfamilie

Studienarbeit
am Lehrstuhl für
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität Erlangen-Nürnberg

Christian Meier

Betreuer:
Dipl.-Inf. Wanja Hofer
Dr. Daniel Lohmann
Prof. Dr. Wolfgang Schröder-Preikschat

Zusammenfassung

Diese Ausarbeitung beschreibt die Analyse und Ansätze zur Implementierung eines zu AUTOSAR COM konformen Moduls für die CiAO-Betriebssystemfamilie.

Gegenwärtige, produktive Implementierungen, die sich strikt an die Spezifikation halten, erlauben nur eine sehr grobgranulare Konfiguration über die Wahl der Varianten, die in der Spezifikation von AUTOSAR COM vorgesehen sind.

Im Rahmen dieser Arbeit wurde gezeigt, dass unter Zuhilfenahme aspektorientierter Techniken eine wesentliche Verfeinerung der Konfigurierbarkeit erreicht werden kann. Hierfür wurde ein Prototyp implementiert, der auf einer TriCore-Plattform von Infineon evaluiert worden ist.

Inhaltsverzeichnis

1 Einführung und Begriffsbestimmung

1.1 Aspektorientierte Programmierung

Aspektorientierte Programmierung ist ein Entwurfsparadigma, bestimmte Merkmale mit Hilfe von Aspekten an mehreren Stellen zu modifizieren, ohne dass diese Stellen selbst modifiziert werden müssen. Mit aspektorientierter Programmierung ist es möglich, sogenannte querschneidende Belange kompakt zusammengefasst an einer oder an wenigen Stellen zu formulieren, anstatt alle Stellen, die von einem bestimmten Aspekt beeinflusst würden, direkt zu modifizieren. Dieses Muster ist eine konsequente Erweiterung des objektorientierten Programmierparadigmas und folgt dem Gedanken des „Separation of Concerns“, also der Trennung der Belange, den vermutlich Edsger Dijkstra zum ersten mal geprägt hatte. Hierzu ein Zitat aus [Dij74], in dem Dijkstra bereits 1974 in diesem Zusammenhang von Aspekten sprach:

> Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an **aspect** of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the **aspects**. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. ... It is what I sometimes have called "**the separation of concerns**", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focussing one's attention upon some **aspect**": it does not mean ignoring the other **aspects**, it is just doing justice to the fact that **from this aspect's point of view, the other is irrelevant**.

Wie er also hier erläutert, ist für ihn die Trennung der Belange die einzige Möglichkeit, diese Belange effizient darzustellen und nicht den Überblick zu verlieren. Er schlägt vor, sich immer auf einen bestimmten einzelne Aspekt zu konzentrieren.

Anhand des einfachen Beispiels in Listing 1.1 soll der Aspektgedanke erklärt werden. Es soll davon ausgegangen werden, dass eine beliebige Klasse `Example` existiert, die eine beliebige Funktionalität erfüllt. Verlangt ist nun, dass auf diese Klasse das Entwurfsmuster des Monitors angewendet wird.

Es ist unmöglich, ohne eine bekannte Definition der Objektmethoden für diese einen Monitor zu bauen. Es sei denn man verwendet aspektorientierte Techniken.

In dem Beispiel von Listing 1.1 ist ein Aspekt formuliert, der dafür sorgt, dass bei Ausführung einer beliebigen Methode der Klasse `Example` zuvor eine Sperrvariable gesetzt und direkt im Anschluss wieder gelöscht wird. Dies stellt folglich einen Monitor dar.

```
aspect Monitor {
  advice "Example" : slice class {
    MutexType mutex_;
  };

  advice execution ("% Example::% (...)") : around () {
    tjp->that ()->mutex_.lock ();
    tjp->proceed();
    tjp->that ()->mutex_.unlock ();
  }
};
```

Listing 1.1: Einfaches Beispiel zur aspektorientieten Programmierung

Diesen Aspekt kann man formulieren, ohne zu wissen, welche Funktionen es in der Klasse `Example` gibt. Auf der anderen Seite muss nirgends in `Example` ein Setzen der Sperrvariable manuell durchgeführt werden, genau genommen muss die Implementierung nicht wissen, dass über alle Methoden von `Example` ein Monitor gelegt wurde. Ob diese Klasse vielleicht einen Monitor voraussetzt, kann hiermit allerdings nicht ausgedrückt werden.

1.2 AUTOSAR

AUTOSAR steht für „AUTomotive Open System ARchitecture“ [AUT08h]. Die offizielle Homepage [AUT08k] beschreibt es als eine „offene und standardisierte automobile Softwarearchitektur, die von einem Konsortium bestehend aus Herstellern, Zulieferen und Softwareentwicklern, entwickelt wird“.

Dabei stellt AUTOSAR gewissermaßen den Nachfolger von OSEK dar. Es bringt neue Konzepte ein, die für eine bessere Wiederverwendbarkeit und Fehlerabschottung von Softwarekomponenten im automobilen Sektor sorgen sollen.

1.3 Verwendete Entwicklungsplattform

Zwar wird die CiAO Betriebssystemfamilie plattform übergreifend entwickelt, jedoch sind erste Treiber für einen Tricore 1796 Prozessor der Infineon AG vorgesehen, der auf einem Developmentboard mit CAN-Anbindung sitzt.

Weiterhin sind auch einige plattformunabhängige Tests geschrieben worden, die auch innerhalb eines hosted x86 Systems laufen können, um einfache Tests schnell durchführen zu können und auf weitestgehende Plattformunabhängigkeit zu testen.

MultiCAN Modul des TriCore1796 Das im TriCore1796 intern vorhandene MultiCAN Modul bietet vier voneinander unabhängige CAN-Bus-Anbindungen. Das Modul erlaubt weiterhin, jeweils die Anbindungen entweder nach außen zu führen, oder aber an einen internen CAN-Bus zu legen. Dies erlaubt das Testen von Hard- und Software, ohne eine zusätzliche äußere Beschaltung anzubringen.

Der Controller bietet eine flexible Vergabe von 16 teilbaren Interruptleitungen an Controller oder einzelne Hardware Message Objects.

2 Analyse des Kommunikationsstacks von AUTOSAR

Die Spezifikation von AUTOSAR COM ist als eigenständiges Dokument nicht einfach einzuordnen. Ist in OSEK COM noch der gesamte Kommunikationsstack grob spezifiziert, so spezifiziert AUTOSAR COM nur ein Modul in der Kommunikationshierarchie eines AUTOSAR-konformen Betriebssystems [OSE04, AUT08d]. Es gibt weitere Teile des Kommunikationsstacks, die in eigene Module ausgelagert wurden. Die in OSEK COM nicht genauer spezifizierte Schnittstelle zu hardwareabhängigen Softwareteilen wurde in AUTOSAR COM ebenfalls standardisiert. Die Spezifikation führt hierbei als Grund an, dass die einzelnen Softwarebestandteile zwischen einzelnen Produzenten auswechselbar gemacht werden sollten. Dies erleichtert zwar die Analyse bezüglich vorhandener Merkmale, jedoch macht es die Zusammenhänge auf den ersten Blick weniger übersichtlich, da die Menge an Dokumenten im Vergleich zu OSEK stark zugenommen hat. Dennoch sollte man bedenken, dass mit CiAO keine hundertprozentige Implementierung von AUTOSAR angestrebt werden soll, sondern eine möglichst konfigurierbare Betriebssystemfamilie, die auch gut mit der Konfiguration und dem Anwendungsgebiet skaliert. Daher können auch einzelne Merkmale nicht oder anders als in der Spezifikation beschrieben umgesetzt werden. Es sei weiterhin angemerkt, dass im Rahmen dieser Studienarbeit nicht alle analysierten Merkmale in der prototypischen Implementierung umgesetzt werden, da dies den zeitlichen Rahmen sprengen würde.

Zwar ist im Gegensatz zu seinem Vorgänger die Spezifikation des Kommunikationsstacks in AUTOSAR wesentlich modularer und genauer verfasst, doch baut sie auf der Spezifikation von OSEK COM auf und referenziert diese häufig.

In der folgenden Analyse werden zuerst die verwendeten Begriffe im Domänenlexikon kurz erklärt. Als nächstes wird die Schichtenarchitektur von AUTOSAR im Hinblick auf die Kommunikationsstrukturen genauer betrachtet. Schließlich wird das Modul, in dem AUTOSAR COM zu implementieren ist, merkmalbasiert dargestellt und darauf basierend eine Concern Impact Analyse durchgeführt.

2.1 Domänenlexikon

Es sei hier angemerkt, dass viele der folgenden Stichworte auch mit RX- und TX-Präfixen verwendet werden. Ist dies der Fall, so bezieht man sich nur auf den entsprechenden Richtungstyp, also RX für Empfang, und TX für Versand. Wird das Präfix weggelassen, so sind Erläuterungen für beide Richtungstypen gültig.

Run Time Environment (kurz: RTE) Steht in Autosar für die oberste Abstraktionsschicht, auf die Applikationen aufbauen. Wird in CiAO nicht implementiert.

Interaction Layer Der Interaction Layer ist die zentrale Schicht im Kommunikationsstack. In dieser sind die meisten kommunikationsrelevanten Module angesiedelt.

Network Layer Der Network Layer ist der von der Hardware abstrahierende Bereich im Kommunikationsstack. Er verbindet den Interaction Layer mit dem Link Layer.

{Can,Fr,Lin}If Bestandteil des Network Layers. Das {Can,FlexRay,Lin} Interface bildet eine gemeinsame Abstraktion für die vorhandenen {CAN,FlexRay,LIN}-Bus-Treiber.

Link Layer Der Link Layer beinhaltet alle hardwarespezifischen Netzwerktreiber. Er verbindet den Physical Layer mit dem Network Layer.

Diensterbringer Ein Diensterbringer bietet eine definierte Schnittstelle an, die zur Erfüllung seines Dienstes erforderlich ist.

Dienstnutzer Ein Dienstnutzer, im Folgenden auch Nutzer genannt, ist eine Entität, die auf die Schnittstellen eines Diensterbringers zugreift.

SDU Eine Service Data Unit besteht aus einer Bytesequenz an Nutzdaten, deren Länge bekannt ist.

PDU Eine Protocol Data Unit besteht aus einer SDU und Protokollinformationen in einem Header. Beim Kommunikationsstack von AUTOSAR besteht dieser nur aus einem Identifier.

{N,L}PDU Dies sind PDUs des Network und Link Layers.

IPDU Eine PDU des Interaction Layers. Eine IPDU besteht aus mindestens einem Signal. Mehrere Signale können sich überlappen oder völlig einschließen.

IPDU-Group Eine Gruppierung von IPDUs und IPDU-Groups, die getrennt aktiviert und deaktiviert werden kann.

RX-IPDU-Group Dies ist eine IPDU-Group, die aus mindestens einer RX-IPDU oder RX-IPDU-Group, die selbst keine RX-IPDU-Group mehr beinhaltet, besteht.

TX-IPDU-Group Dies ist eine IPDU-Group, die aus mindestens einer TX-IPDU oder TX-IPDU-Group, die selbst keine TX-IPDU-Group mehr beinhaltet, besteht.

Signal In OSEK COM noch "Message" genannt, handelt es sich bei den Signalen aus AUTOSAR COM um das gleiche Konzept nicht zusammengesetzter Datentypen. Ein Signal ist ein Serie von zusammenhängenden Bits innerhalb einer IPDU. Es kann als vorzeichenloser oder vorzeichenbehafteter Ganzzahltyp oder als Array von Bytes interpretiert werden.

Signal-Group Eine Menge von Signalen innerhalb einer IPDU, die bzgl. der Übertragung atomar behandelt werden müssen. Sie ist zur Abbildung von komplexen Datentypen gedacht.

Group-Signal Ein Group-Signal ist ein Signal, das Mitglied einer Signal-Group ist.

Shadow Buffer Ein Speicherbereich, in dem Daten aus bestimmten Gründen ein zweites mal gespeichert werden. In AUTOSAR werden Shadow Buffer zur Konsistenzwahrung von Signal-Groups herangezogen.

DM Deadline Monitoring überprüft, ob bestimmte Zeitschranken von Empfang und Versand von IPDUs eingehalten werden.

Byte Order Die Byte Order eines Signals gibt an, wie ein Signal innerhalb einer IPDU kodiert ist. Dabei bedeutet **Little Endian**, dass das niederwertigste Byte, **Big Endian**, dass das höchstwertige Byte zuerst gespeichert ist.

Filter Filter werden bei Empfang und Versand von Signalen zur Beeinflussung weiterer Aktionen herangezogen.

Indication ist das Anzeigen an einen Dienstnutzer, dass neue Daten für ein Signal oder eine PDU empfangen wurde.

Confirmation ist die Bestätigung für einen Dienstnutzer, dass eine Signal/PDU erfolgreich auf dem Bus versendet werden konnte.

Invalidation ist eine Möglichkeit, im Netzwerk bekannt zu geben, dass ein bestimmter Wert keine Gültigkeit hat, z.B. aufgrund eines defekten Sensors.

Transfer Property Diese Eigenschaft gibt für TX-Signale an, ob ein Update eines Signals in einer IPDU eine Übertragung dieser IPDU auslösen soll. Die tatsächliche Aktion hängt auch vom gewählten Transmission Mode der IPDU ab.

Transmission Mode Der Transmission Mode gibt das Übertragungsverhalten einer IPDU an. Dabei wird hauptsächlich zwischen periodischer und unmittelbarer Übertragung unterschieden. Für eine IPDU können zwei Transmission Modes konfiguriert werden, die je nach TMS ausgewählt werden.

Direct Transmission Mode, Direct TM Ein Transmission Mode, bei dem alle getriggerten Übertragungen direkt ausgeführt werden.

Periodic Transmission Mode, Periodic TM Ein Transmission Mode, bei dem alle von Dienstnutzern getriggerten Übertragungen ignoriert werden, und nur die internen, periodischen Tasks eine Übertragung mit eine konfiguierten Periodizität durchführen.

Transmission Mode Selection, TMS Dies ist das Verfahren, das auf den Filtern für die Senderichtung aufbaut und die anhand der TMC über die konfigurierten Transmission Modes einer IPDU entscheidet.

Transmission Mode Condition, TMC Die Bedingung, die über die Wahl der Transmission Modes in der TMS entscheidet. Sie bildet innerhalb einer IPDU ein Prädikat über alle Ergebnisse der für die assoziierten TX-Signale konfigurierten Filter.

2.2 Ein einfacher Anwendungsfall

Bevor nun auf die eigentliche Architektur eingegangen wird, wird hier anhand eines Beispiels beschrieben, was mit den von Applikationen bereitgestellten Daten gemacht werden soll.

2.2.1 Ausgangslage

Nehmen wir im Folgenden also an, dass in einem CAN-Bus-Netzwerk zwei Knoten existieren, einer mit einem Temperatursensor, der andere mit einem Aktuator, der die Geschwindigkeit eines Lüftermotors steuern und damit die Temperatur regeln kann. Weiterhin ist der Sensorknoten für den Empfang von Zeitsignalen verantwortlich, er verschickt also die aktuelle Tageszeit in der Form hh:mm:ss auf dem Bus. Es wird nun angenommen, dass die Temperatur ebenfalls ein mal pro Sekunde auf dem Bus benötigt wird.

2.2.2 Grobe Prozedurale Sichtweise

Der gemessene Temperaturwert und die aktuelle Zeit müssen als erstes in eine IPDU gepackt werden. Als nächstes wird die daraus resultierende IPDU der entsprechenden Netzwerkschnittstelle übergeben und dort als Busrahmen versendet.

Der Aktuatorknoten nimmt den Busrahmen an und übergibt die daraus resultierende IPDU dem Modul, das für die Extraktion des Temperaturwertes und der aktuellen Zeit aus der IPDU sorgt. Dieses Modul übergibt die Daten dann an die Applikation.

2.2.3 Verfeinerungen des Anwendungsfalls

Dieses Beispiel, wenn nicht anders erwähnt, wird als Standardanwendungsfall in der gesamten Analyse herangezogen. Anhand diesem werden dann die Vorteile des jeweils aktuell untersuchten Merkmals dargestellt. Die aktuellen Beispiele sind jeweils umrahmt.

2.3 Kommunikationsarchitektur - Schichtenarchitektur

Im folgenden Abschnitt soll ein grober Überblick über den gesamten Kommunikationsstack gegeben werden, um das zu implementierende Modul für AUTOSAR COM genauer einordnen zu können.

2.3.1 Applikationen

Die oberste Schicht in der Architektur von AUTOSAR stellen die Applikationen in Form von Komponenten dar [AUT08a]. Im Bezug zu externer Kommunikation stellt sich die Situation also wie in Abbildung 2.1 dar. Es soll erreicht werden, dass zwei Applikationen, die auf verschiedenen Netzwerkknoten arbeiten, Informationen austauschen können.

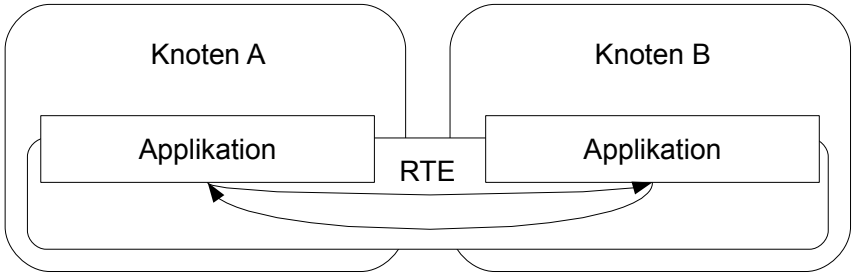

Abbildung 2.1: AUTOSAR Applikationsschicht mit RTE

2.3.2 RTE - Runtime Environment

Um das Ziel der Kommunikation über die Controllergrenzen hinweg zu ermöglichen, bietet das Runtime Environment die von Applikationen verlangten Dienste. Wie in Abbildung 2.1 zu sehen ist, können Applikationen über das RTE transparent auch mit Applikationen auf anderen Knoten in einem Netzwerk kommunizieren.

Dabei bildet das RTE eine Abstraktion bezüglich verschiedener Knoten [AUT08j]. In CiAO ist ein RTE nicht vorgesehen. Da aber wesentliche Merkmale, die in OSEK COM ursprünglich spezifiziert waren, in AUTOSAR COM in das Runtime Environment ausgelagert wurden, muss hier ein Kompromiss zwischen strikter Einhaltung der Spezifikation und Eigenschaften von CiAO getroffen werden.

2.3.3 IL - Interaction Layer

Der Interaction Layer stellt gewissermaßen die Zentrale der Kommunikationsarchitektur dar. In dieser Schicht befinden sich alle für die externe Kommunikation relevanten, plattformunabhängigen Module.

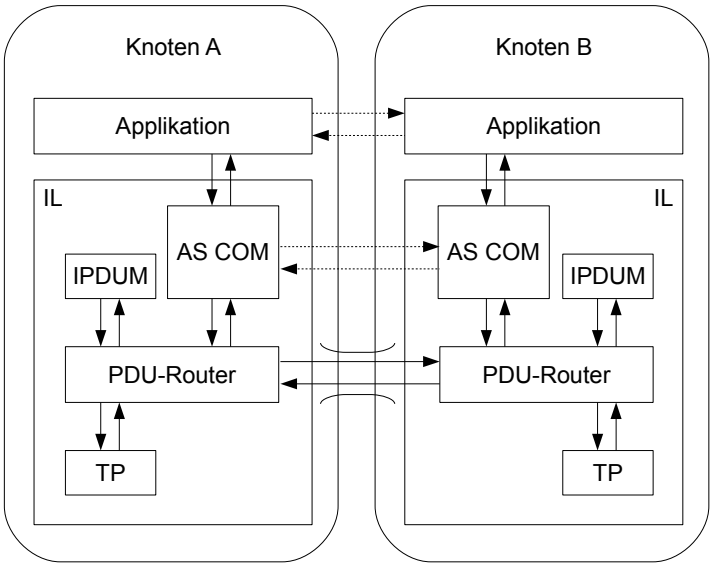

Abbildung 2.2: Interaction Layer

AS COM - AUTOSAR COM

AUTOSAR COM ist das Modul für die Konvertierung von Signalen und IPDUs. Hier werden alle Applikationsnutzdaten für den Versand auf einem Kommunikationsmedium vorbereitet, und empfangene Daten für Applikationen aufbereitet.

PDU-Router

Dieses Modul ist für die Zustellung einer PDU an die richtige Netzwerkschnittstelle, den IPDU-Multiplexer oder eines Transportmoduls zuständig. Dies wird anhand der Id der PDUs gemacht [AUT08i]. In Abbildung 2.2 ist zu sehen, dass bei dieser Betrachtungsweise der Abstraktionschichten die Kommunikation ausschließlich über einen gedachten Kanal zwischen den PDU-Routern verschiedener Knoten verläuft.

IPDUM - IPDU-Multiplexer

Sollen in einer IPDU Daten mit mehreren, verschiedenen Semantiken gefüllt werden, so muss die IPDU an den IPDU-Multiplexer weitergegeben werden. Dieser sorgt dann anhand eines Selektionsfeldes in der IPDU für die richtige Verarbeitung. Dabei wird zwischen Bereichen unterschieden, die immer den gleichen Informationstyp enthalten (statische Bereiche), und Bereichen, die je nach Inhalt des Selektorfeldes unterschiedliche Daten enthalten (dynamische Bereiche). Aus diesen Informationen generiert der IPDU-Multiplexer neue IPDUs mit anderer Id, die dann wieder dem PDU-Router zur weiteren Verarbeitung übergeben werden [AUT08f].

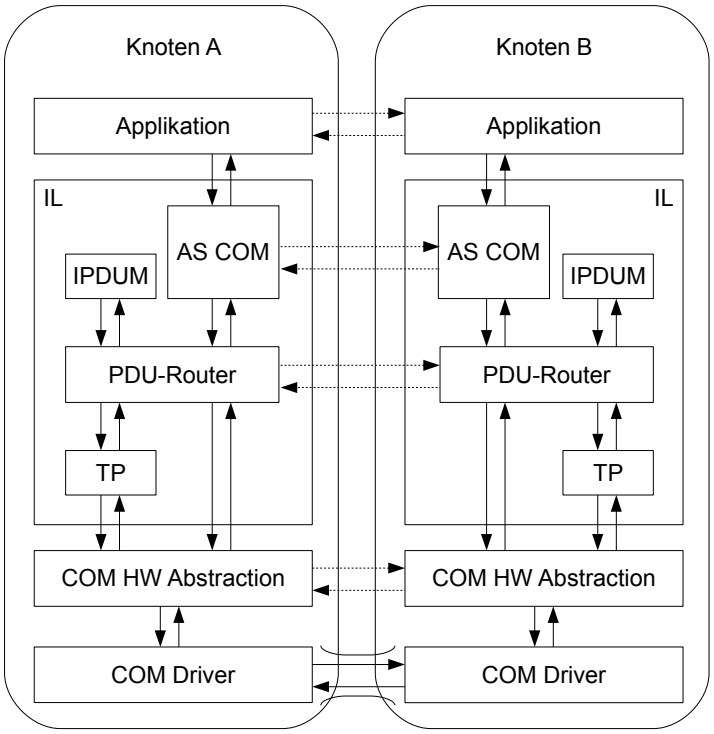

Abbildung 2.3: COM HW Abstraction/Driver

TP - Transport Layer Modul

Dieses Modul ist für die Segmentierung von IPDUs in NPDUs und deren Zusammensetzung zuständig, falls die zu übertragende Datenmenge die maximale Größe einer NPDU des jeweils zugrunde liegenden physikalischen Busses überschreiten würde. Wird für eine IPDU die Längenbeschränkung nicht überschritten, so kann die IPDU direkt an die konfigurierte Busschnittstelle als NPDU weitergeleitet werden [AUT08c, AUT08a].

2.3.4 Communication Hardware Abstraction/Driver

Der Interaction Layer gibt die Daten wie in Abbildung 2.3 dargestellt an die Communication Hardware Abstraction weiter. Die Communication Hardware Abstraction bietet eine hardwareunabhängige Schnittstelle für jeden Bustyp an. Sie besteht aus CAN-, FlexRay- und LIN-Schnittstelle.

In dieser Schicht wird von konkreten Hardwaretreibern abstrahiert. Einerseits bietet sie eine einheitliche Schnittstelle zu allen Treibern eines Bustyps an, andererseits ermöglicht sie es, nicht nur einen, sondern mehrere Treiber für den gleichen Bustyp und damit also auch verschiedene Hardware gleichzeitig in Betrieb zu halten. Dabei fasst die Schnittstelle alle physikalisch vorhandenen Netzwerkschnittstellen eines Bustyps, egal ob sie von

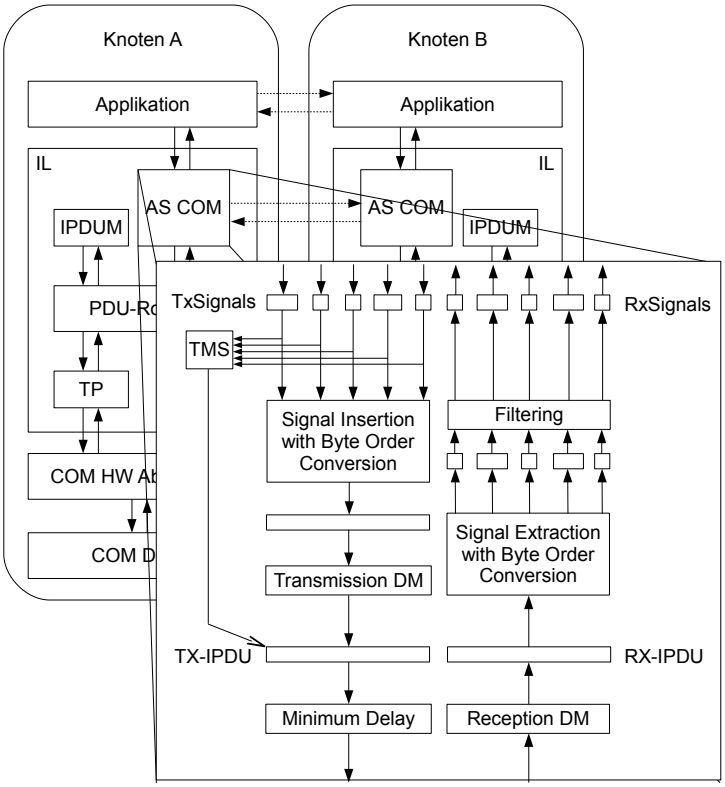

Abbildung 2.4: Von AUTOSAR COM zu bewerkstelligende Aufgaben

dem gleichen oder einem anderen Treiber bedient werden, in ein einheiltiches Schema zusammen [AUT08b, AUT08e, AUT08g].

In der untersten Schicht über der direkten Hardware sitzen die eigentlichen Hardwaretreiber. Sie sind abhängig von Plattform und Controllertyp. Die Schnittstelle ist ebenfalls standardisiert [AUT08a].

2.4 Merkmalmodellierung - AUTOSAR COM

Bevor nun auf die einzelnen Merkmale eingegangen wird, soll anhand von Abbildung 2.4 ein kurzer Überblick über die Aufgaben gegeben werden, die in AUTOSAR COM bewerkstelligt werden müssen.

Die zu sendenden Signale müssen zu einer IPDU zusammengefasst werden. Dazu werden die Signale von der Byte Order der Architektur auf die konfigurierte Byte Order des Signals konvertiert. Für die daraus resultierenden IPDUs muss je nach Konfiguration ein Deadline Monitoring für die Übertragung der IPDU gestartet werden, der erst bei erfolgter Übertragungsbestätigung zurückgesetzt wird. Zusätzlich dazu kann die *Transmission Mode Selection* (TMS) aus den Werten aller Signale, die zu einer IPDU gehören, und

einem pro Signal konfigurierten Filter ein Prädikat berechnen. Dieses bestimmt über die weitere Verarbeitung der IPDU. Als unterste Teilschicht in AUTOSAR COM gibt es noch den Minimum Delay Timer, der dafür sorgt, dass nicht zu viele Übertragungswünsche pro Zeitintervall einer IPDU an die nächste Schicht weiter gegeben werden.

Auf der Empfangsseite kann für eine IPDU eine Deadline definiert sein, die angibt, nach welcher Zeit seit dem letzten Empfang ein erneuter Empfang stattgefunden haben muss. Wird in dieser Zeit die jeweilige IPDU nicht empfangen, so wird die konfigurierte Fehlerbehandlung durchgeführt. Nach dem Empfang einer IPDU wird diese unter Berücksichtigung der Byte Order in die einzelnen Signale konvertiert. Auf den Ergebnissen wird je nach Konfiguration eine Filterung durchgeführt, die bestimmte nicht erwünschte oder nicht benötigte Daten herausfiltern kann.

Im folgenden sollen die einzelnen Merkmale beschrieben werden. Bei den Merkmalen in den Beschreibungen handelt es sich um globale Konfigurationen. Die Diagramme stellen also die mögliche Miteinbeziehung bestimmter Algorithmen und Datenstrukturen in den Kompiliervorgang dar. Wie die Merkmale für einzelne Objekte konfigurierbar sind, wird im Kapitel Implementierung erklärt.

2.4.1 Hauptmerkmale von AUTOSAR COM

RX und TX

AUTOSAR COM kann grob in zwei fast vollständig voneinander unabhängige Bereiche aufgeteilt werden, nämlich den Bereich des Sendens und den des Empfangens. Bis auf die Unterstützung für das *Signal Gateway*, das sowohl auf Empfang als auch auf Versand aufbaut und in dieser Arbeit nicht berücksichtigt wird, gibt es keine Abhängigkeiten. Diese Merkmale treten in den folgenden Merkmalmodellen mehrmals als *RX* und *TX* auf, und stehen für die jeweilige Unterstützung für Empfang und Versand des übergeordneten Merkmals.

Im Anwendungsfall:
Der Sensorknoten muss nur Versand, der Aktorknoten nur Empfang unterstützen.

Rx (DM)

Um den rechtzeitigen Empfang von IPDUs zu überprüfen, bietet AUTOSAR COM die Möglichkeit, für jede IPDU eine maximale Zwischenankunftszeit zu definieren. Wenn zwischen zwei Empfangsvorgängen zweier IPDUs mit der gleichen ID mehr als diese Zeitspanne vergangen ist, dann wird eine Fehlerbehandlung angestoßen.

Im Anwendungsfall:
Soll der Aktorknoten überprüfen, dass der Temperaturwert nach jeweils spätenstens 1.5 Sekunden auf den Bus gegeben worden ist, so muss für die entsprechende IPDU Reception DM mit 1.5 sec konfiguriert werden.

Tx (DM)

Da ein Merkmal des AUTOSAR-Kommunikationsstacks ist, dass viele Schnittstellen asynchronen Charakter haben, ist dies bei den unteren Netzwerkschichten, wie z.B. dem

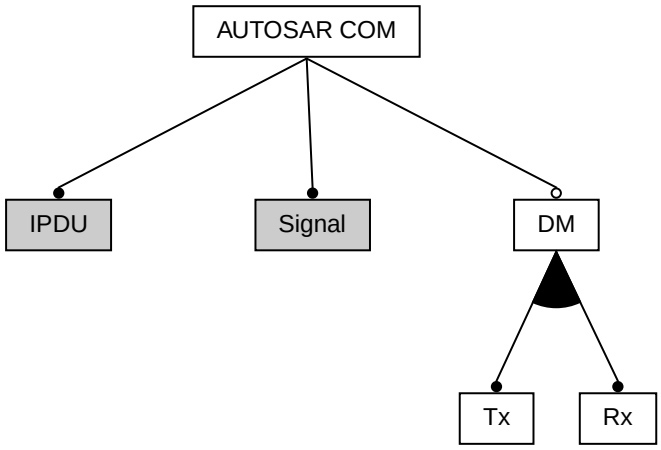

Abbildung 2.5: AUTOSAR COM Konzeptknoten und oberste Ebenen

CanIf-Modul auch der Fall. Im einzelnen bedeutet dies, dass ein erfolgreicher Aufruf von `CanIf::Transmit()` noch nicht bedeutet, dass die Übertragung erfolgreich abgeschlossen wurde. Es bedeutet hingegen nur, dass der Übertragungswunsch erfolgreich angenommen wurde. Um sicherzustellen, dass die jeweilige IPDU übertragen worden ist, bieten die unteren Schichten eine Callbackschnittstelle zur Bestätigung einer erfolgten Übertragung an.

Im Anwendungsfall:
Soll der Sensorknoten überprüfen, dass der Temperaturwert nach spätestens 1.5 Sekunden auf den Bus gegeben worden ist, so muss für die entsprechende IPDU Transmission DM mit 1.5 sec konfiguriert werden.

Das *Transmission DM* überwacht nun die Übertragung, indem es die für eine IPDU konfigurierte Zeitspanne als Timeout setzt. Wird innerhalb dieses Zeitraums keine Bestätigung an AUTOSAR COM weitergegeben, dann wird eine Fehlerbehandlung angestoßen.

Dieses Merkmal kann nur aktiviert werden, wenn für die untere Schicht das Merkmal der Versandbestätigung auch aktiviert wurde. Wenn dies nicht der Fall wäre, würde niemals eine Versandbestätigung zu AUTOSAR COM gelangen.

2.4.2 Untermodell für das Merkmal *IPDU*

IPDU

Unter diesem verpflichtenden Merkmal werden alle weiteren Merkmale zusammengefasst, die in direktem Zusammenhang mit IPDUs stehen.

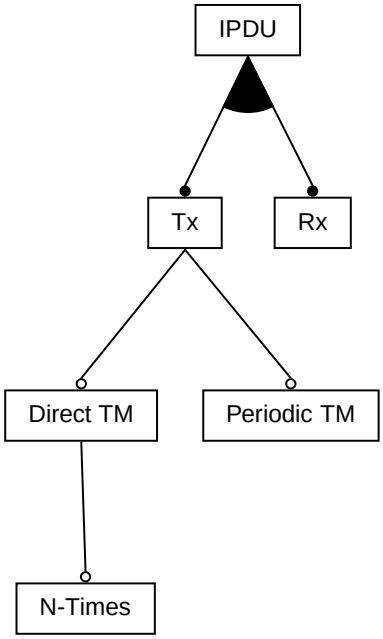

Abbildung 2.6: Zu IPDU untergeordnete Merkmale

Periodic TM

Der *Periodic Transmission Mode* sorgt für eine rein periodische Übertragung einer IPDU, unabhängig von der konfigurierten *Transfer Property* der Signals.

Im Anwendungsfall:
Soll auf dem Bus unabhängig von der Applikation die IPDU vom Sensorknoten genau einmal pro Sekunde übertragen werden, so muss für die IPDU der *Periodic Transmission Mode* konfiguriert werden.

Dies hat zur Folge, dass die IPDU genau einmal pro Sekunde auf dem Bus verschickt wird, auch wenn die Applikation den Datenpuffer von AUTOSAR COM öfters oder weniger oft aktualisiert hat.

Direct TM

Der *Direct Transmission Mode* sorgt für eine ausschließlich direkte Übertragung, d.h. bei jedem von der Applikation generierten Übertragungswunsch eines Signals, das mit *Triggered Transfer Property* konfiguriert wurde, wird ein Übertragungswunsch an den nächsten Verarbeitungsschritt weitergegeben.

Im Anwendungsfall:

Soll jedesmal bei Erneuerung des Zeitsignals eine IPDU mit den neuesten Daten verschickt werden, so muss die IPDU mit dem *Direct TM* konfiguriert, und das Zeitsignal mit *Triggered Transfer Property* konfiguriert werden.

N-Times Direct TM

Der *N-Times TM* ist eine Erweiterung des *Direct TMs* und erlaubt es, bei einmaligem Sendewunsch eines Dienstnutzers für ein Signal den Wunsch N-mal für die entsprechende IPDU an den entsprechenden Diensterbringer weiterzuleiten.

(Mixed TM)

Der *Mixed TM* repräsentiert zwar kein eigenes Merkmal, soll aber hier in diesem Zusammenhang trozdem erwähnt werden. Sobald global *Periodic TM* und *Direct TM* konfiguriert wurden, so steht der Konfiguration einer einzelnen IPDU auch die Möglichkeit des *Mixed TM* zur Verfügung, bei dem sowohl direkte, als auch periodische Übertragungen erlaubt sind.

2.4.3 Untermodell für das Merkmal *Signal*

Signal

In dem Merkmalmodell aus Abbildung 2.7 sind die zu *Signal* untergeordneten Merkmale zusammengefasst.

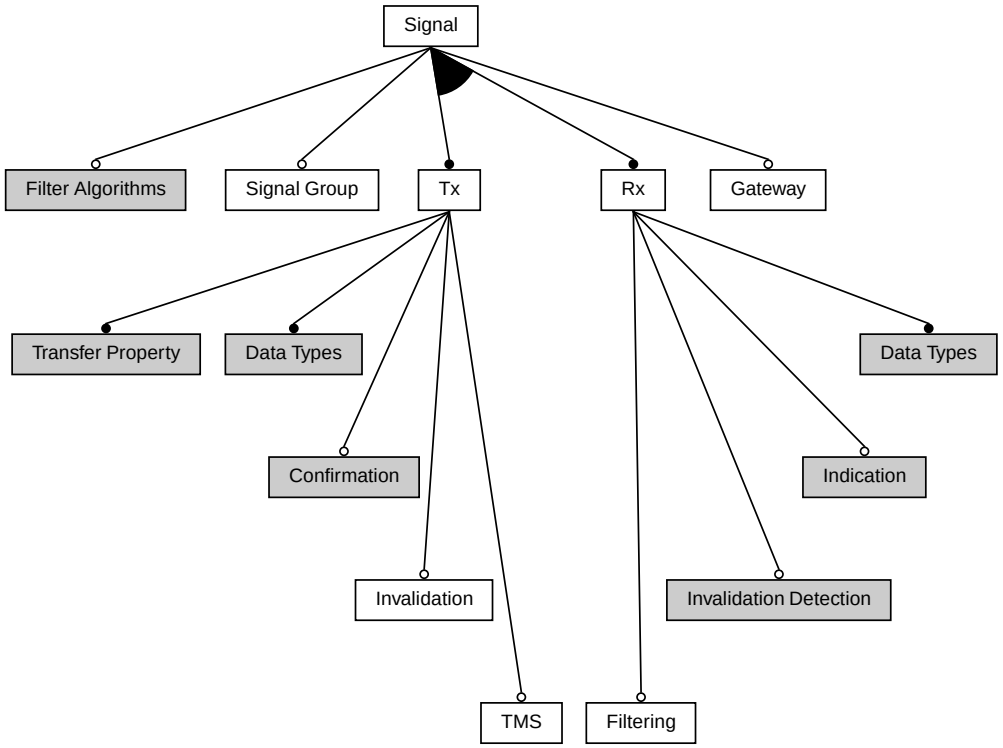

Abbildung 2.7: Zu Signal untergeordnete Merkmale

Signal Gateway

Das *Signal Gateway* ist für die weitere Verteilung von über den PDU-Router erhaltenen Signalen (in Form von IPDUs) an andere Netzwerke zuständig. Signale können also hier von bestimmten IPDUs ausgepackt und in anderer Kombination in andere IPDUs eingepackt werden.

Signal-Group

Signal-Groups sind ein optionales Merkmal. Ist dieses Merkmal aktiviert, so steht dem System eine Möglichkeit zur Verfügung, verschiedene einzelne Signale innerhalb einer IPDU zusammen als einzelnes atomares Objekt zu behandeln. Diese dann Group-Signals genannten Signale werden intern in einem Shadow Buffer zwischengespeichert. Erst wenn die Applikation die Signal-Group freigibt, werden die Daten intern übernommen.

Im Anwendungsfall:

Man stelle sich eine auf dem Sensorknoten vorhandene Instanz `Time` der Struktur

```
struct TimeType {
  UInt hr_;
  UInt min_;
  UInt sec_;
};
```

vor. Es ist für die Übertragung der Zeitdaten nicht möglich, drei einfache Signale zu konfigurieren, die innerhalb einer IPDU mit konfiguriertem *Periodic TM* liegen, da die folgenden Operationen von einer Applikation nicht atomar ausgeführt werden.

```
AsCom::SendSignal{TimeHrSignal,  &Time.hr_);
AsCom::SendSignal{TimeMinSignal, &Time.min_);
AsCom::SendSignal{TimeSecSignal, &Time.sec_);
```

Stattdessen muss eine Signal-Group definiert werden, die aus den drei Signalen besteht. Im folgenden wird angenommen, dass die drei Signale in der Konfiguration in die Signal-Group `TimeGroup` gelegt wurden. Dann ergeben sich diese Aufrufe:

```
AsCom::UpdateShadowSignal(TimeHrSignal,  &Time.hr_);
AsCom::UpdateShadowSignal(TimeMinSignal, &Time.min_);
AsCom::UpdateShadowSignal(TimeSecSignal, &Time.sec_);
AsCom::SendSignalGroup(TimeGroup);
```

Nach dem Update der einzelnen Shadow Buffer wird mit dem Aufruf der Funktion `SendSignalGroup()` die Signal-Group an sich freigegeben, d.h. die Daten der gesamten `TimeGroup` werden in die entsprechend für `TimeGroup` konfigurierte IPDU übernommen.

Filtering

Filtering ermöglicht es, beim Empfangsvorgang bestimmter Werte eines Signals die weitere Verarbeitung nicht durchzuführen. Dies heißt, dass der neu empfangene Wert verworfen wird. Hierfür stehen die Filteralgorithmen, die unter dem Merkmal *Filter Algorithms* konfiguriert worden sind, zur Verfügung.

Im Anwendungsfall:

Angenommen, der Aktuator soll nur Temperaturwerte kleiner als 0x30 und größer als 0x40 weiterverarbeiten. Dazu muss das Temperatursignal mit dem Filter `NewIsOutside` mit `min=0x30` und `max=0x40` konfiguriert werden. Schwanken die Werte nun ausschließlich innerhalb dieser Werte, dann wird niemals die eigentliche Behandlung durchgeführt.

TMS

Die *TMS* ermöglicht es, beim Senden in Abhängigkeit der neu zu sendenden Werte eine Umschaltung zwischen zwei verschiedenen Transmission Modes zu machen. Hierzu

können die gleichen Filteralgorithmen herangezogen werden, wie sie auch bei Empfangsfiltern benutzt werden. Bei Konfiguration eines TX-Signals müssen zwei Transmission Modes konfiguriert werden, einen für den Fall, dass die TMS `true` liefert, einen für den Fall `false`.

Im Anwendungsfall:

Angenommen, der Sensor soll bei extremer Temperatur das Signal öfters auf den Bus legen, dann können zwei *Transmission Modes* für die entsprechende IPDU konfiguriert werden. Beide *TMs* werden auf *Periodic* gestellt, der eine mit einer Periode von einer Sekunde, der andere mit einer Periode von 0,5 Sekunden. Die Entscheidung, welcher *TM* genutzt werden soll, kann durch einen *NewIsWithin*-Filter mit `min=0x20` und `max=0x50` erledigt werden.

Invalidation

Das *Invalidation*-Merkmal ermöglicht es einem Dienstnutzer, AUTOSAR COM mitzuteilen, dass ein Signalwert ungültig ist, z.B. zur Anzeige, dass ein Sensor defekt ist. Dazu muss für jedes Signal, das invalidiert werden kann, auch eine spezieller Wert konfiguriert werden, der die Ungültigkeit der Daten kodiert. Dieser Wert sollte, um Überschneidungen zu vermeiden, außerhalb des normalen Wertebereichs des Signals liegen.

Im Anwendungsfall:

Soll der Temperatursensor mitteilen können, dass die die Temperaturdaten ungültig sind, so muss das Temperatursignal mit *Invalidation* konfiguriert werden. So kann man den Temperaturbereich für ein 8-bit-Signal von 0x00 bis 0x80 legen und 0xee als ungültigen Wert definieren.
Wird nun von der Applkation `InvalidateSignal()` auf das Temperatursignal aufgerufen, so wird der interne Puffer mit dem Wert 0xee ersetzt.

2.4.4 Untermodell für das Merkmal *Filter Algorithms*

Filter Algorithms

Das Merkmal *Filter Algorithms* stellt Algorithmen für *Filtering* und *TMS* zur Verfügung, die eine Entscheidung über die weitere Verwendung der Daten eines Signals bei Empfang und für die Auswahl des Transmission Modes bei Versand liefern. Für die Entscheidung stehen dem Filter verschiedene Daten zur Verfügung, die sich je nach Kategorie unterscheiden.

Gibt der Filter `true` zurück, so bedeutet dies, dass der Wert akzeptiert, also nicht verworfen werden soll. Die im Folgenden konfigurierbaren Parameter können für jedes Signal unterschiedlich gewählt werden.

Simple

Die trivialen Filteralgorithmen sind unter dem Merkmal *Simple* zusammengefasst. Da diese immer den gleichen Wert liefern, benötigen die Filter keine Daten, auf denen sie ihre Entscheidung treffen.

Always liefert immer `true`.

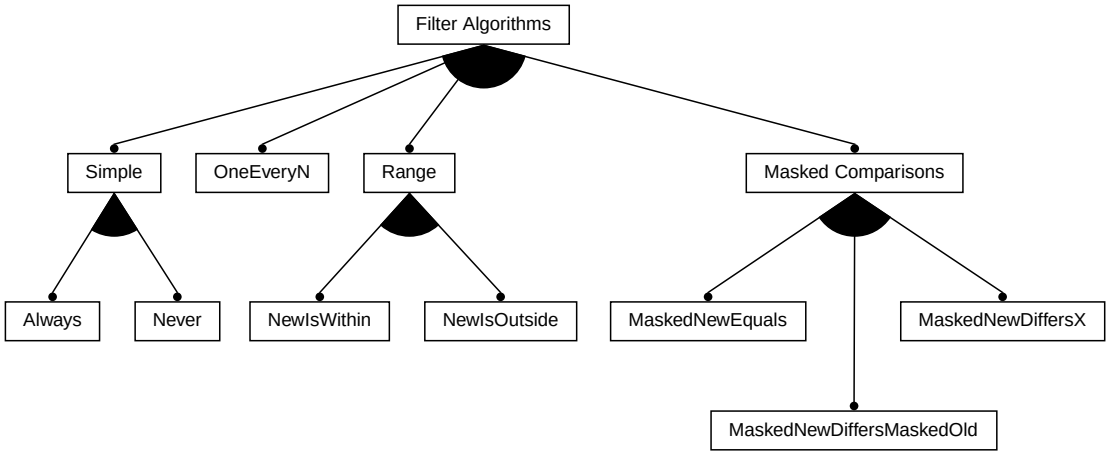

Abbildung 2.8: Für Signale zur Verfügung stehende Filteralgorithmen

Never liefert immer `false`.

Masked Comparisons

Der Filterkategorie *Masked Comparisons* steht der letzte nicht verworfene und der zu filternde Wert, eine konfigurierte Maske und ein konfigurierter Vergleichswert zur Verfügung.

MaskedNewEqualsX liefert genau dann `true`, wenn der neue Wert nach Anwendung einer konfigurierten Maske einem konfigurierten Wert entspricht.

MaskedNewDiffersX liefert genau dann `true`, wenn der neue Wert nach Anwendung einer konfigurierten Maske einem konfigurierten Wert nicht entspricht.

MaskedNewDiffersMaskedOld liefert genau dann `true`, wenn nach Anwendung einer konfigurierten Maske auf alten und neuen Wert sich die Resultate unterscheiden.

Range

Der Filterkategorie *Range* steht der zu filternde Wert und ein Wertebereich, kodiert in einem Minimal- und einem Maximalwert zur Verfügung.

NewIsWithin liefert genau dann `true`, wenn der neue Wert innerhalb eines für ein Signal konfigurierten Wertebereichs liegt.

NewIsOutside liefert genau dann `true`, wenn der neue Wert außerhalb eines für ein Signal konfigurierten Wertebereichs liegt.

OneEveryN

Als Konfiguration steht dem Filter *OneEveryN* eine ganzzahlige Periode N und ein ganzzahliger Offset O zur Verfügung. Nach jeweils N Aufrufen des Filters für ein bestimmtes Signal liefert dieser `true`, sonst `false`. Ist ein Offset O ungleich 0 konfiguriert, so liefert der Filter initial O-mal `false` und beginnt dann mit der periodischen Filterung des jeweiligen Signals, ansonsten gibt der erste Filtervorgang `true` zurück.

Im Anwendungsfall:

Soll der Aktuator nur jede fünfte Aktualisierung des Temperatursignals umsetzen, so muss ein *OneEveryN*-Filter mit Periode 5 auf das Temperatursignal konfiguriert werden.

2.4.5 Untermodell für die Merkmale *Confirmation* und *Indication*

Confirmation/ Indication

Confirmation und *Indication* sind Merkmale, die es Dienstnutzern von AUTOSAR COM erlauben, benachrichtigt zu werden, wenn neue Werte bestimmter Signale eingetroffen sind, bzw. die Übertragung bestätigt worden ist.

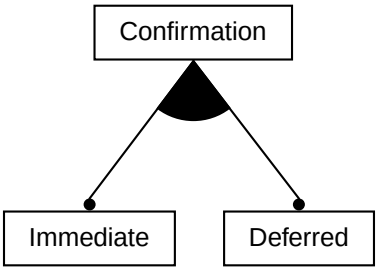

Abbildung 2.9: Versandbestätigung

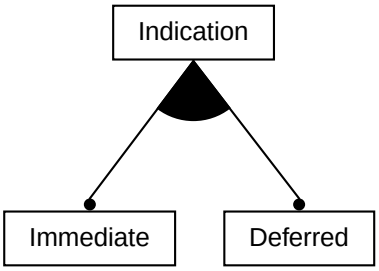

Abbildung 2.10: Empfangsanzeige

Immediate

Die *Immediate Confirmation/Indication* verursacht eine direkte Bearbeitung, sobald AUTOSAR COM Notiz von dem erfolgreichen Abschluss der jeweiligen Aktion bekommen hat.

Im Anwendungsfall:

Soll für ein zeitkritisches Signal eine Benachrichtigung möglichst zeitnah erfolgen, so muss für dieses Signal die *Immediate Confirmation/Indication* gewählt werden.

Deferred

Bei *Deferred*-Benachrichtigung wird bei Erkennung des erfolgreichen Abschlusses der jeweiligen Aktion nur ein internes Flag gesetzt. Die eigentliche Benachrichtigung wird erst bei der nächsten periodischen Aktion, deren Periode jeweils für Sende- und Empfangsrichtung global konfiguriert wird, durchgeführt.

Im Anwendungsfall:

Sollen die Ausführungszeiten der von AUTOSAR COM bereitgestellten und möglicherweise innerhalb eines Interruptkontexts laufenden Methoden `RxIndication()` und `TxConfirmation()` begrenzt werden, so muss für alle Signale eine *Deferred Confirmation/Indication* konfiguriert werden.

2.4.6 Untermodell für das Merkmal *Data Types*

Die Datentypen definieren verschiedene Möglichkeiten, die aus einer IPDU extrahierten oder die von Dienstnutzern erhaltenen Daten zu interpretieren. Dabei muss bei der internen Verarbeitung auf Vorzeichen und Byte Order geachtet werden.

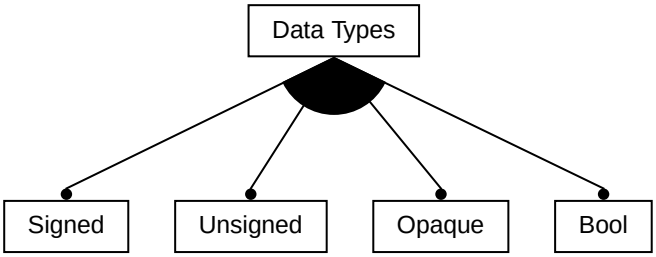

Abbildung 2.11: Von AUTOSAR COM unterstützte Datentypen

Signed

Dieses Merkmal steht für die Unterstützung vorzeichenbehafteter Datentypen.

Im Anwendungsfall:

Soll auf dem Aktorknoten das Temperatursignal beispielsweise als vorzeichenbehaftete 8-bit-Ganzzahl definiert sein, und sie belegt in der IPDU 5 Bit, dann muss das Vorzeichen negativer Werte an mehrere Stellen in den Empfangspuffer geschrieben werden.

`Signal innerhalb der IPDU:`	`11000`	`01000`
`Wert des Signals:`	`-8`	`8`
`Signal im Empfangspuffer:`	`11111000`	`00001000`

Weiterhin wird durch dieses Merkmal auch die Byte Order Conversion aktiviert, d.h. dass die eingestellte Netzwerkdarstellung in die Architekturdarstellung umgewandelt wird.

Unsigned

Die `Unsigned`-Datentypen sorgen ebenfalls für eine automatische Byte Order Konvertierung, allerdings werden die Bitfolgen als vorzeichenlose Zahlenwerte interpretiert.

Im Anwendungsfall:

Wird das Temperatursignal vorzeichenlos übertragen, so muss *Unsigned* als Datentyp angegeben werden.

`Signal innerhalb der IPDU:`	`11000`	`01000`
`Wert des Signals:`	`24`	`8`
`Signal im Empfangspuffer:`	`00011000`	`00001000`

Opaque

Der *Opaque*-Datentyp sorgt für die unveränderte Weiterleitung der empfangenen/zu sendenden Daten. Dabei wird keine Konvertierung der Byte Order oder Erweiterung eines Vorzeichens vorgenommen. Dieser Datentyp ist auf Signale beschränkt, deren Position in der SDU auf Bytegrenzen konfiguriert sind.

Bool

Der *Bool*-Datentyp steht für eine Repräsentation eines Wahrheitswertes, der nur `true` und `false` darstellen kann.

2.4.7 Untermodell für das Merkmal *Transfer Properties*

Transfer Property

Die *Transfer Property* definiert die verschiedenen Möglichkeiten auf den Erhalt eines Übertragungswunsches von einem Dienstnutzer für ein Signal zu reagieren.

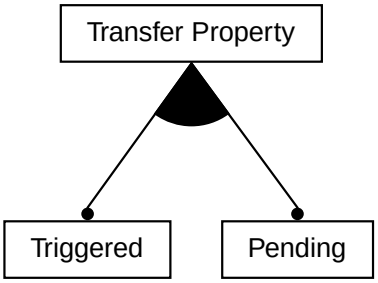

Abbildung 2.12: Untermodell zu den verfügbaren *Transfer Property*

Triggered TP

Die *Triggered Transfer Property* sorgt je nach Konfiguration des *Transmission Modes* der IPDU, zu der das Signal gehört, für ein unterschiedliches Verhalten. Ist der *Direct/N-Times TM* oder der *Mixed TM* konfiguriert, so wird der Übertragungswunsch direkt verarbeitet. Ist der *Periodic TM* konfiguriert, so werden zwar die neuen Daten in den Sendepuffer übernommen, aber es findet keine Weiterleitung des Übertragungswunsches statt.

Im Anwendungsfall:
Immer dann, wenn das Zeitsignal aktualisiert wird, soll dies in einer Übertragung auf dem Bus resultieren. Dazu wird das Zeitsignal mit der *Triggered Transfer Property* konfiguriert.

Pending TP

Die *Pending Transfer Property* sorgt unabhängig von der Konfiguration des *Transmission Modes* der IPDU, zu der das Signal gehört, für die Ignorierung des Übertragungswunsches. Einzig der interne Sendepuffer wird mit den neuen Daten gefüllt. Diese Einstellung ist nur sinnvoll, wenn noch mindestens ein Signal in der IPDU liegt, das mit *Triggered Transfer Property* konfiguriert wurde, die IPDU *Mixed TM* oder *Periodic TM* besitzt, oder die Daten von einer Netzwerkschnittelle selbst abgeholt werden.

Im Anwendungsfall:

Der Temperaturwert soll nur dann übertragen werden, wenn das Zeitsignal erneuert worden ist. Dazu muss das Temperatursignal mit der *Pending Transfer Property* konfiguriert werden.

2.4.8 Untermodell für das Merkmal *Invalidation Detection*

Ist bei einem Sender ein Signal so konfiguriert, dass es seine *Invalidation* unterstützt, so kann auf der Empfängerseite für den dann erhaltenen Wert, der die Ungültigkeit kodiert, ein gesondertes Verhalten aktiviert werden.

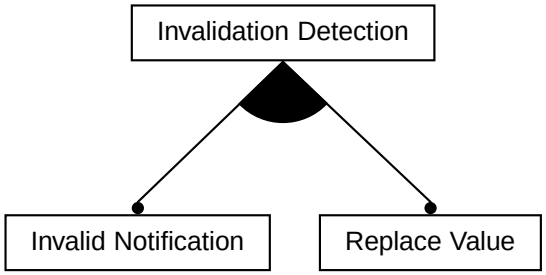

Abbildung 2.13: Möglichkeiten zur Invalidierung von Signalwerten

Invalid Notifcation

Soll eine gesonderte Benachrichtigung bei Erhalt eines ungültigen Wertes erfolgen, so muss *Invalid Notification* gewählt werden. Die normale Empfangsbenachrichtigung wird nicht vollzogen.

Im Anwendungsfall:

Ist der Aktuatorknoten für die IPDU, die das Temperatursignal enthält, mit *Invalid Notification* konfiguriert, so wird bei Erhalt des ungültigen Wertes eine gesonderte Benachrichtigung gestartet.

Replace Value

Soll die Benachrichtigung auch bei ungültigen Werten über die normale Empfangsbenachrichtigung ablaufen, so muss *Replace Value* konfiguriert werden. Bei diesem Verfahren wird der Empfangspuffer des Signals auf den initialen Wert gesetzt.

Im Anwendungsfall:

Ist der Aktuatorknoten für die IPDU, die das Temperatursignal enthält, mit *Replace Value* konfiguriert, und der initiale Wert des Signals ist auf 0xff konfiguriert, so wird bei Erhalt des ungültigen Wertes der interne Puffer mit 0xff ersetzt. Ein anschließendes Auslesen des Empfangspuffers durch die Anwendung liefert 0xff.

2.5 Concern Impact Analysis

In Rahmen einer Concern Impact Analyse sind die Auswirkungen bestimmter Merkmale auf die verschiedenen Elemente des Gesamtsystems ermittelt worden. Die Ergebnisse sind in Tabelle 2.1 aufgelistet. Dabei wurden Auswirkungen auf folgende Stellen analysiert:

Schnittstelle Diese Auswirkungen beschreiben die Änderung der nach außen sichtbaren Schnittstelle von AUTOSAR COM.

Systemressourcen Unter den Auswirkungen auf Systemressourcen werden die Auswirkungen zusammengefasst, die Ressourcen des Betriebssystems benötigen.

Konfiguration Die Auswirkungen auf die Konfiguration bedeuten eine Änderung der Container, in der die Konfiguration gespeichert ist.

Laufzeitcontainer Mit dem Laufzeitcontainer sind die Strukturen gemeint, in denen sich zur Laufzeit ändernde Attribute abgelegt werden.

Algorithmen Mit den Algorithmen ist der Code gemeint, der auf der Konfiguration und den Laufzeitattributen arbeitet.

Im Folgenden werden einige der analysierten Merkmale aus Tabelle 2.1 exemplarisch genauer erläutert.

2.5.1 (RX/TX) DM

Je nach Konfiguration kann *Deadline Monitoring* für den Versand und/oder Empfang konfiguriert sein.

Auswirkung auf die Schnittstelle Für das Deadline Monitoring wird eine Schnittstelle zum Deaktivieren und Aktivieren angeboten, wenn dies gewünscht wird.

Auswirkung auf die Systemressourcen Für das *Deadline Monitoring* werden ein oder zwei Tasks benötigt, die die Methoden `MainFunctionTx` für *Rx DM* und `MainFunctionRx` für *Tx DM* von AUTOSAR COM streng periodisch aufrufen.

Auswirkung auf die Konfiguration Es muss gespeichert werden, ob für die jeweilige IPDU *Deadline Monitoring* durchgeführt werden soll. Weiterhin muss jeweils eine Periode und eine Phase gespeichert werden. Es muss konfiguriert werden, was beim Ablaufen der Deadline passieren soll.

Auswirkung auf die Laufzeitcontainer Das Merkmal *DM* benötigt zur Abbildung der konfigurierten Periode einen Laufzeitzähler, der je nach systemweiter maximaler Periode aller Signale verschieden groß gewählt werden kann.

Auswirkung auf die Algorithmen In den periodisch aufgerufenen Funktionen wird für alle mit *DM* konfigurierten IPDUs die zeitliche Überprüfung durchgeführt. Bei entsprechendem Erreichen eines Zeitlimits wird eine Fehlerbehandlung durchgeführt. Dazu wird ein Countdown benötigt.

Bei Empfang einer IPDU muss der Countdown der IPDU zurückgestellt werden.

Bei Versand muss entweder nach erfolgreich bearbeitetem Erhalt eines Übertragungswunsches von einem Dienstnutzer oder nach Generierung eines periodischen Übertragungswunsches ein Countdown gestartet werden. Dieser muss bei Erhalt der Versandbestätigung zurückgesetzt werden. Laufen die Countdowns ab, wird die konfigurierte Fehlerbehandlung durchgeführt.

2.5.2 Direct TM/Periodic TM

Direct TM und *Periodic TM* hat Einfluss auf die sendeseitge Verarbeitung bzgl. Übertragungswünschen von Dienstnutzern. Dabei ist auch eine Kombination von Direct und Periodic TM möglich (*Mixed TM*). Wenn weder *Direct TM* noch *Periodic TM* für eine IPDU konfiguriert wird, so wird der *Transmission Mode* als *None* bezeichnet.

Auswirkung auf die Systemressourcen Für den *Periodic TM* wird ein Task benötigt, der die Funktion `MainFunctionTx` von AUTOSAR COM streng periodisch aufruft. In dieser Funktion wird für alle IPDUs mit *Periodic/Mixed TM* bei entsprechendem Erreichen eines Zeitlimits ein periodischer Übertragungswunsch getriggert.

Auswirkung auf die Konfiguration Diese Merkmale benötigen zur Konfiguration der jeweiligen IPDU einen Speicherplatz zur Kodierung des *Transmission Modes*. Zur Unterscheidung der beiden Modi werden 2 bit benötigt, da für eine IPDU auch gleichzeitig beide oder kein *Transmission Mode* konfiguriert sein kann.

Auswirkung auf die Laufzeitcontainer Das Merkmal *Periodic TM* benötigt zur Abbildung der konfigurierten Periode einen Laufzeitzähler, der je nach systemweiter maximaler Periode aller Signale verschieden groß gewählt werden kann.

Auswirkung auf die Algorithmen Bei Auslösung eines Übertragungswunsches einer IPDU wird überprüft, ob diese durch einen periodischen Übertragungswunsch oder durch einen direkten Übertragungswunsch von einem Dienstnutzer verursacht wurde. Dann wird überprüft, ob für die jeweilige Herkunft des Übertragungswunsches auch der entsprechende *Transmission Mode* konfiguriert worden ist. Ist dies der Fall, so wird der Übertragungswunsch weitergeleitet, ansonsten wird er verworfen.

2.5.3 Triggered/Pending Transfer Property

Die für ein TX-Signal konfigurierbare *Transfer Property* hat Einfluss auf die Generierung von Übertragungswünschen der assoziierten TX-IPDU.

Auswirkung auf die Konfiguration Dieses Merkmal benötigt zur Konfiguration des jeweiligen TX-Signals ein Bit pro TX-Signal zur Speicherung der *Transfer Property*, außer es wird nur eine Transfer Property benötigt.

Auswirkung auf die Algorithmen Bei Erhalt von Daten vom Dienstnutzer wird anhand der konfigurierten *Transfer Property* entschieden, ob ein Übertragungswunsch getriggert (*Triggered Transfer Property*) werden soll oder nicht (*Pending Transfer Property*).

2.5.4 Data Types

Die unter dem Merkmal *Data Types* zu findenden Konfigurationsparameter beeinflussen hauptsächlich die Extraktions- und Einfügealgorithmen von Signalen aus den und in die assoziierten IPDUs für Empfang und Versand.

Auswirkung auf die Konfiguration Wenn für für die Menge aller Signale mehr als ein Datentyp vorgesehen ist, werden Konfigurationsparamater für Vorzeicheninterpretation, INTEGER, OPAQUE bzw. BOOLEAN Datentyp und Länge des Signals benötigt.

Es ist anzumerken, dass beim Versand ohne *TMS* nur die Länge des Signals benötigt wird, da beim Einfügen des Signals in die IPDU nichts vergleichbares zur Vorzeichenbestimmung und -erweiterung beim Empfang gemacht wird. Daher spielt der Typ keine Rolle.

Auswirkung auf die Algorithmen Beim Emfang einer IPDU muss der entsprechende Algorithmus für die Extraktion der Daten aus der SDU selektiert und ausgeführt werden.

	AUTOSAR COM Schnittstelle	TX-IPDU Konfiguration	TX-IPDU Laufzeitcontainer	TX-IPDU Algorithmen	RX-IPDU Konfiguration	RX-IPDU Laufzeitcontainer	RX-IPDU Algorithmen	TX-Signal Konfiguration	TX-Signal Laufzeitcontainer	TX-Signal Algorithmen	RX-Signal Konfiguration	RX-Signal Laufzeitcontainer	RX-Signal Algorithmen	Systemressource Task
AUTOSAR COM	●	-	-	-	-	-	-	-	-	-	-	-	-	-
DM	-	-	-	-	-	-	-	-	-	-	-	-	-	●
RX DM	●	-	-	-	●	●	●	-	-	-	-	-	-	-
TX DM	●	●	●	●	-	-	-	-	-	●	-	-	-	-
IPDU	-	-	-	-	-	-	-	-	-	-	-	-	-	-
TX	-	●	●	●	-	-	-	-	-	-	-	-	-	-
Direct TM	-	●	-	●	-	-	-	-	-	-	-	-	-	-
Periodic TM	-	●	●	●	-	-	-	-	-	-	-	-	-	●
Minimum Delay Timer	-	●	●	●	-	-	-	-	-	-	-	-	-	●
RX	-	-	-	-	●	●	●	-	-	-	-	-	-	-
Signal	-	-	-	-	-	-	-	-	-	-	-	-	-	-
RX	●	-	-	-	-	-	-	-	-	-	-	-	-	-
Data Types	-	-	-	-	-	-	-	-	-	-	●[a]	-	●	-
All	-	-	-	-	-	-	-	-	-	-	-	-	●	-
Indication	-	-	-	-	-	-	-	-	-	-	●	-	●	-
Invalid. Detection	-	-	-	-	-	-	-	-	-	-	-	-	●	-
Nofication	-	-	-	-	-	-	-	-	-	-	●	-	●	-
Replace Value	-	-	-	-	-	-	-	-	-	-	●	-	●	-
Filtering	-	-	-	-	-	-	-	-	-	-	●[b]	●	●	-
All w/o OneEveryN	-	-	-	-	-	-	-	-	-	-	●	-	●	-
OneEveryN	-	-	-	-	-	-	-	-	-	-	●	●	●	-
TX	●	-	-	-	-	-	-	-	-	-	-	-	-	-
Transfer Property	-	-	-	-	-	-	-	●[c]	-	-	-	-	-	-
Triggered TP	-	-	-	-	-	-	-	-	-	●	-	-	-	-
Pending TP[d]	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Data Types	-	-	-	-	-	-	-	●[a]	-	●	-	-	-	-
All	-	-	-	-	-	-	-	-	-	●	-	-	-	-
Invalidation	●	-	-	●	-	-	-	●	-	●	-	-	-	-
Confirmation	-	-	-	●	-	-	-	●	-	●	-	-	-	-
Immediate	-	-	-	-	-	-	-	-	-	●	-	-	-	-
Deferred	-	-	-	-	-	-	-	-	●	●	-	-	-	●
TMS	-	●	-	-	-	-	-	●	●	●	-	-	-	-

[a] Nicht nötig, wenn nur ein Datentyp (aber nicht OPAQUE oder BOOLEAN) verarbeitet werden soll
[b] Nicht nötig, wenn nur ein Filteralgorithmus benötigt wird
[c] Nicht nötig, wenn nur eins von Triggered/Pending TP genutzt wird
[d] Das Verhalten ist in der Grundkonfiguration bereits enthalten

Tabelle 2.1: Auswirkungen einzelner Merkmale

3 Implementierung

Aufgrund von zum Teil fehlenden Use Cases und zeitlichen Einschränkungen werden einige Merkmale im Rahmen dieser Studienarbeit nicht implementiert. Dabei handelt es sich hauptsächlich um *TMS*, *N-Times Direct Transmission Mode* und das *Signal-Gateway*.

3.1 Allgemeine Verwaltungsstrukturen

Bevor auf die AUTOSAR COM spezifischen Module eingegangen wird, soll hier die bei der Implementierung neu entstandene Verwaltungsstruktur `Bitset` kurz dargestellt werden, da diese auch in anderen Subsystemen genutzt werden könnte.

3.1.1 Bitset

Das Bitset erlaubt eine beliebige, positive Anzahl an Bits. Jedes Bit kann gesetzt und gelöscht werden. Es kann nach dem ersten/letzten gesetzten/gelöschten bit gesucht werden. Die Implementieung greift dabei auf die Funktionen von `namespace bitmanip` zurück.

```
template <unsigned long long NUM_BITS>
class Bitset {
public:
  typedef ... IndexType;

  void Set (IndexType BitIndex);
  void Clear (IndexType BitIndex);
  Bool Test (IndexType BitIndex);
  IndexType Highest1Bit ();
  IndexType Highest0Bit ();
  IndexType Lowest1Bit ();
  IndexType Lowest0Bit ();
  ...
};
```

Listing 3.1: Die Klasse Bitset

Passt die Anzahl an Bits nicht mehr in einen primären, vorzeichenlosen Ganzzahltyp, so wird ein Array von Elementen vom Typ `UInt`, die also so groß wie die Register der Architektur sind, angelegt. Weiterhin ist darauf geachtet worden, dass maximal eine einfache Implementierung und eine Arrayimplementierung im System instanziiert ist, da die relevanten Zugriffsfunktionen in eine Basisklasse, die selbst kein Template ist, ausgelagert worden sind.

3.1.2 Erweiterung der Templates des Typensystems

Die bereits vorhandenen Templates zur plattformunabhängigen Bestimmung von Integer Typen wurden mit folgenden Templates erweitert:

Überprüfung der Existenz eines Typs eines bestimmter Größe und Vorzeichenbehaftung

```
template <unsigned int BYTES, bool SIGNED> struct int_t_exists;
```

Dabei gilt `int_t_exists<BYTES,SIGNED>::RES == true` genau dann, wenn der Compiler einen nativen Datentyp besitzt der `BYTES` Bytes hat und je nach `SIGNED`, mit oder ohne Vorzeichen sein soll.

Ermittlung eines vorzeichenlosen Typs gleicher Größe

```
template <typename INT_TYPE> struct int_t_to_unsigned;
```

Der Ergebnistyp `int_t_to_unsigned<INT_TYPE>::T` bestimmt abhängig von `INT_TYPE`, einen gleich großen vorzeichenlosen Typ.

Ermittlung eines vorzeichenbehafteten Typs gleicher Größe

```
template <typename INT_TYPE> struct int_t_to_signed;
```

Der Ergebnistyp `int_t_to_signed<INT_TYPE>::T` bestimmt abhängig von `INT_TYPE`, einen gleich großen vorzeichenbehafteten Typ.

3.2 Indizierung für mehrfach instanzierte Objekte gleichen Typs

Viele Typen, u.a. `TxIpduType`, `TxSignalType`, `RxSignalType` und `RxIpduType` sind zur Laufzeit mehrfach instanziiert. Um eine eindeutige Identifikation zugewährleisten, und trotzdem Speicherplatz zu sparen, sind alle Objekte eines Typs innerhalb der definierenden Klasse als statisches Array zusammengefasst. Die Größe des jeweiligen Arrays bestimmt die zur Laufzeit maximal verfügbare Anzahl an Objekten dieses Typs und wird über die pure::variants Konfiguration eingestellt. Für jeden der Typen, die diesem Schema folgen, existiert auch ein entsprechender Id-Typ, also gibt es z.B. für `TxSignalType` es ein `TxSignalIdType`. Dieser Typ ist eine nicht negative Ganzzahl, deren Größe beim Kompilieren durch die Typentemplates so gewählt wird, dass die Anzahl der vorhandenen Instanzen ohne Überlauf in diesem Typ gespeichert werden kann. Hierbei wird die optimierte Variante `uint_t_value_opt` des Typensystems benutzt, so dass bei Architekturen mit größerer Bitbreite kein Codeoverhead entsteht. Für manche Idtypen stehen auch noch die jeweils nicht optimierten Varianten, beispielsweise `TxSignalIdStorageType` zur Verfügung, die mit `uint_t_value` erstellt wurden. Diese werden beispielsweise in größeren Tabellen zum platzsparenderen Ablegen der Daten benötigt.

Weiterhin ist es auch möglich, die Id einer gegebenen Instanz herauszufinden. Dies erfolgt bei bekanntem Objektpointer durch die Berechnung des Offsets innerhalb des Arrays mit Hilfe der Methode `Id()`.

Aus diesem Schema ergibt sich das Beispiel für `RxSignalType` in Listing 3.2.

```
// pure::variants Konfiguration der maximal zur Verfügung
// stehenden Instanzen vom Typ RxSignalType
#define cfOS_COM_ASCOM_NUM_RX_SIGNALS 123

// Automatisches Setzen des Indextyps
typedef int_t_value_opt<cfOS_COM_ASCOM_RX_SIGNAL_MULTIPLICITY>
    RxSignalIdType;

class RxSignalType {
  ...
  static RxSignalType RxSignals_[cfOS_COM_ASCOM_RX_SIGNALS_MULTIPLICITY];

public:
  static RxSignalType& Inst (RxSignalIdType Id) { return RxSignals_[Id]; }
  RxSignalIdType Id () { return this - RxSignals_; }
};
```

Listing 3.2: Instanziierungsschema anhand der Klasse `RxSignalType`

3.3 Pufferverwaltung bestimmter Klassen

In vielen Klassen (u.a. `TxIpduType`, `RxSignalType`, `RxSignalGroupType`) müssen Nutzdaten zwischengespeichert werden. Dafür werden Puffer benötigt, deren Größe zwar statisch bekannt ist, deren Größe sich allerdings nicht zwingend vom jeweiligen Typ ableiten lassen kann. Beispielsweise kann es mehrere Instanzen vom Typ `TxIpduType` geben, von den eine 1 Byte und die andere 8 Byte Nutzdaten verwaltet.

Dies in Templates aufzulösen ist keine Alternative, da alle diese Objekte zur Indizierung in einem Array abgelegt werden sollen und damit alle vom gleichen Typ sein müssen. Eine Ablage von Referenzen einer Basisklasse steht ebenfalls nicht zur Debatte, da dies einen Mechanismus zum indexspezifischen Casten erfordern würde, also z.B. virtuelle Funktionen oder Funktionspointer. Diese Techniken sollen weitestgehend vermieden werden, so dass keine unnötigen indirekten Aufrufe gemacht werden müssen, und Inlining weitestgehend ermöglicht bleibt.

Zur Lösung dieses Problems gibt es verschiedene Ansätze, die alle je nach Nutzerkonfiguration mehr oder weniger gut bzgl. Ausführungzeit und Speichernutzung sind.

Jedes Objekt, dass einen Puffer benutzt, bestimmt die Adresse des Puffers nie direkt, sondern immer über die Methode `GetBuffer()`. Diese Methode wird für jeden Ausgangstyp (`TxIpduType`, `RxSignalType`, u.a.) per introduction advice hinzugefügt.

Es gibt verschiedene Slices, die den Puffer auf unterschiedliche Art zur Verfügung stellen:

3.3.1 DirectFixedSizeBuffer

Sind die benötigten Puffergrößen für alle Objekte eines Typs ähnlich, so bietet es sich an, alle Objekte mit einem Puffer der Größe max(Länge Puffer i) auszustatten. Dieses Verfahren wird dann ineffizient, wenn nur wenige Puffer groß sein müssen und der Rest sehr klein.

3.3.2 IndirectRealPointerBuffer

Bei großen Variationen der Puffergröße sollte für jedes Objekt ein Pufferbereich mit entsprechender Größe zur Verfügung gestellt werden. Dazu wird der Puffer nicht mehr in der entsprechenden Klasse, sondern für jedes Objekt als externes Array abgelegt. In einem Objekt wird nur noch ein Pointer abgelegt, der auf den jeweiligen Puffer verweist. Dieses Verfahren wird dann ineffizient, wenn es sehr viele Objekte mit Puffern gibt, deren Größe kleiner ist als die eines Pointers.

3.3.3 IndirectSmallPointerBuffer

Um das Problem von IndirectRealPointerBuffer zu umgehen, kann man den Pointer mit einem Integer mit weniger Bits nachbilden. Dieser Integer wird als Index in einen großen, zusammenhängenden Puffer benutzt. Dieses Verfahren ist nur dann gut, wenn der zusammenhängende Pufferbereich so klein ist, dass der überhaupt mit weniger Bits indiziert werden kann.

Wie man sehen kann, hängt also die Optimalität dieser Verfahren stark von den konfigurierten Größen der Signale und IPDUs ab.

3.4 Überblick über die wichtigsten internen Typen von AUTOSAR COM

Die folgenden Typen sind bis auf das oben erklärte Instanziierungsschema leer und werden durch Aspekte beeinflusst.

RxIpduConfigType In diese Klasse werden alle Konfigurationsparameter eingefügt, die für RX-IPDUs relevant sind.

RxIpduType In dieser Klasse werden alle zur Laufzeit veränderlichen Parameter abgelegt, die für die Verarbeitung von RX-IPDUs benötigt werden. Weiterhin erbt sie von `RxIpduConfigType`.

TxIpduConfigType In diese Klasse werden alle Konfigurationsparameter eingefügt, die für TX-IPDUs relevant sind.

TxIpduType In dieser Klasse werden alle zur Laufzeit veränderlichen Parameter abgelegt, die für die Verarbeitung von TX-IPDUs benötigt werden. Weiterhin erbt sie von `TxIpduConfigType`.

RxSignalConfigType In diese Klasse werden alle Konfigurationsparameter eingefügt, die für RX-Signale relevant sind.

RxSignalType In dieser Klasse werden alle zur Laufzeit veränderlichen Parameter abgelegt, die für die Verarbeitung von RX-Signalen benötigt werden. Weiterhin erbt sie von `RxSignalConfigType`.

TxSignalConfigType In diese Klasse werden alle Konfigurationsparameter eingefügt, die für TX-Signale relevant sind.

TxSignalType In dieser Klasse werden alle zur Laufzeit veränderlichen Parameter abgelegt, die für die Verarbeitung von TX-Signalen benötigt werden. Weiterhin erbt sie von `TxSignalConfigType`.

3.5 Umsetzung der Merkmale

3.5.1 Unterstützung für Signale in RxIpduType

Damit RX-Signale Daten aus RX-IPDUs extrahieren können, muss den RX-Signalen auch mitgeteilt werden, dass neue Daten vorliegen. Dies wird durch einen Advice auf `RxIpduType::RxIndication()` ermögicht.

```
advice execution ("% os::com::ascom::RxIpduType::RxIndication (...)")
    && args (SduPtr)
    : before (UInt8 const * SduPtr) {

  for (RxSignalIdType ActId = tjp->that ()->SignalIdMin ();
       ActId <= tjp->that()->SignalIdMax (); ++ActId) {
    os::com::ascom::RxSignalType::Inst (ActId).RxIndication (SduPtr);
  }
}
```

Listing 3.3: Advice für die Unterstützung von RX-Signalen in einer RX-IPDU

Der Advice sorgt dafür, dass alle Signale, die mit der aktuellen IPDU assoziiert sind, eine Empfangsbenachrichtigung erhalten.

3.5.2 Extraktion der Signale aus einer SDU

Bei der Extraktion der Daten gibt es viele verschiedene Konfigurationsparameter zu beachten. Neben der Byte Order bezüglich Architektur und dem jeweiligen Signal innerhalb der SDU stellen die verschiedenen Möglichkeiten Byte- und Bitpositionen, die Interpretation der extrahierten Daten und deren Filterung eine große Menge an Konfigurationsparametern zur Verfügung, bei denen es sich nur um die Verarbeitung der Nutzdaten handelt.

Um sich an die verwendeten Algorithmen anzunähern wird zuerst der verwendete Grundalgorithmus und seine Varianten erklärt. Im weiteren Verlauf wird auf ausgewählte Verfeinerungen genauer eingegangen.

Grundalgorithmus Um eine allgemeine Implementierung für verschiedene Plattformen zu erhalten, wird unser Grundalgorithmus bytebasiert arbeiten. Spezialimplementierungen für bestimmte Architekturen können aber zu einem späteren Zeitpunkt nachgeholt werden.

Für unsere erste Betrachtung gehen wir davon aus, dass alle Signale innerhalb der SDU auf Bytegrenzen ausgerichtet sind.

```
void CopyFromSduToSignal (UInt8 const * SduPtr) {
  RxByteIndexType LESignalIndex = 0;
  SduPtr = & SduPtr [LsByteIndex ()];

  for (RxByteIndexType StillToRead = SduBytesToRead ();
       StillToRead;
       StillToRead--) {
    NewValue ()[SignalByteMapping (LESignalIndex)] = *SduPtr;
    SduPtr += ByteOrder ();
    ++LESignalIndex;
  }
}
```

Listing 3.4: Extraktion bytebegrenzter Signale

Folgende Methoden treten in Listing 3.4 zusätzlich auf:

LsByteIndex() Diese Methode liest aus der Konfiguration die Position des niederwertigsten Bytes des Signals innerhalb der SDU aus.

MsByteIndex() Diese Methode liest komplementär zu `LsByteIndex()` die Position des höchstertigen Bytes des Signals innerhalb der SDU aus.

ByteOrder() Liefert die Bytereihenfolge des aktuellen Signals innerhalb der SDU. Dabei steht 1 für Little Endian und -1 für Big Endian.

NewValue() Liefert je nachdem, ob Filter aktiviert worden sind, einen temporären Zwischenspeicher oder den endgültigen Empfangspuffer.

SduBytesToRead() Bestimmt die Anzahl an Bytes, die für dieses Signal aus der IPDU gelesen werden müssen.

SignalByteMapping() Die Methode bestimmt anhand der Byte Order der Architektur und der Typkonfiguration des Signals, an welche Stelle innerhalb des Empfangspuffers ein Byte mit einem bestimmten Byteindex in Least Significant Byte Order geschrieben werden muss. Auf Architekturen mit Little Endian Byte Order ist dies immer der übergebene `LEIndex` selbst.

```
template<typename INT_TYPE>
INT_TYPE
extract_low (INT_TYPE Value, UInt NumBits) {
  Value <<= sizeof(INT_TYPE)*__CHAR_BIT__ - NumBits;
  Value >>= sizeof(INT_TYPE)*__CHAR_BIT__ - NumBits;
  return Value;
}
```

Listing 3.5: Korrekturverfahren für die nicht genutzen Bits des Signalpuffers

Der Algorithmus iteriert über die Bytes eines Signals innerhalb der SDU von LSB zu MSB und kopiert die Daten in den Zielpuffer.

Was hier noch nicht bedacht wurde, ist, dass ein Signal innerhalb der SDU zwar auf Bytegrenzen ausgerichtet sein kann, allerdings mit weniger Bits in der SDU als auf der Architektur konfiguriert wurde. So kann es beispielsweise vorkommen, dass ein Integer mit 16 Bit auf der Architektur nur 8 Bit innerhalb der SDU erhalten hat. Dies erledigt eine anschließende Korrektur.

Korrektur bezüglich der Bitlänge Wenn nun die Implementierung aus Listing 3.4 ohne weitere Maßnahmen angewendet wird, dann werden eventuell einige Bits von in der SDU nicht belegten oder zu anderen Signalen gehörenden Bits in den internen Puffer (oder den tempöraren Puffer) kopiert. Um dies zu verhindern, wird ein Korrekturalgorithmus ausgeführt. Dieser `Extend` Algorithmus benötigt die exakte Länge des Signals in Bits innerhalb der SDU als Parameter. Da es für jede Kombination von Vorzeicheninterpretation und Länge des Signals eine andere Ausprägung des Algorithmus gibt, wird der Algorithmus mit einem in einer Jumptable gespeicherten Funktionszeiger aufgerufen. Nur der Index innerhalb dieser Tabelle muss in der Konfiguration zusätzlich gespeichert sein.

Da die aus der SDU kopierten Daten immer beim Bitindex 0 anfangen, kann das Korrekturverfahren auch davon ausgehen. Die Korrektur muss nur bei Signalen durchgeführt werden, die nicht den Datentyp *Opaque* konfiguriert haben, da diese immer eine Folge ganzer Bytes repräsentieren. Die Funktion für den Datentyp *Opaque* bleibt folglich leer.

In Listing 3.5 ist das Template zur entsprechenden Korrektur dargestellt. Die erste Shiftoperation sorgt für das Ignorieren aller höherwertigen Bits von Value, die nicht zum Signal gehören. Der zweite Shift sorgt abhängig von INT_TYPE für eine Vorzeichenerweiterung oder Erweiterung mit Nullen.

Nicht an Bytegrenzen orientierte Signale Der obige Grundalgorithmus aus Listing 3.4 funktioniert für nicht auf Bytegrenzen orientierte Signale nicht, denn dort müssen die Daten zusätzlich noch geshiftet werden. Um dies zu erreichen, werden im erweiterten Algorithmus je zwei Bytes des Signals innerhalb der SDU benötigt, um ein Byte im Zielpuffer zu schreiben. In Listing 3.6 ist das erweiterte Verfahren dargstellt. Dabei bleibt der Grundgedanke, von LSByte zu MSByte innerhalb der SDU zu iterieren, erhalten. Lediglich das Zusammensetzen der Daten unterscheidet sich.

```
CopyFromSduToSignal (UInt8 const * SduPtr) {
  RxByteIndexType LESduIndex = LsByteIndex ();
  RxByteIndexType LESignalIndex = 0;

  // The buffer must take at least two bytes
  UInt16Opt Buffer = SduPtr[LESduIndex]

  while(LESduIndex != MsByteIndex ()) {
    // Adjust position
    LESduIndex += ByteOrder ();

    // We load the new Data
    Buffer |= ((UInt16Opt) (SduPtr[LESduIndex])) << 8;

    // Save the new data
    NewValue ()[SignalByteMapping (LESignalIndex)]
      = Buffer >> LsBitIndex ();

    // Shift the buffer
    Buffer = Buffer >> 8;

    // Adjust position
    LESignalIndex++;
  }

  if(MsBitIndex () >= LsBitIndex ()) {
    // We must write one more byte
    NewValue ()[SignalByteMapping (LESignalIndex)]
      = Buffer >> LsBitIndex ();
  }
}
```

Listing 3.6: Extraktion von nicht an Bytegrenzen ausgerichteten Signalen

In Listing 3.6 werden zwei neue Methoden aufgerufen. Diese haben folgende Funktionalität.

LsBitIndex() Die Methode bestimmt aus der Konfiguration den Bitoffset des niederwertigsten Bits innerhalb des niederwertigsten Bytes des Signals innerhalb der SDU.

MsBitIndex() Analog zu `LsBitIndex()` bestimmt `MsBitIndex()` den Bitoffset des höchstwertigsten Bits innerhalb des höchstwertigen Bytes des Signals innerhalb der SDU.

Der Puffer `Buffer`, der mindestens zwei Bytes fassen kann, wird mit dem LSB initialisiert. Dann wird über das Signal innerhalb der SDU iteriert. Dabei wird ein Byte immer aus dem um `LsBitIndex()` nach rechts geshifteten Puffer gelesen und in den Zielpuffer geschrieben. Ist der MsBitIndex größer als der LsBitIndex, dann muss am Schluss noch ein Schreibvorgang nachgeholt werden, da noch benötigte aber nicht geschriebene Bits im Zwischenpuffer liegen. Der oben beschriebene Korrekturalgorithmus kann hier ohne Änderung übernommen werden.

Feingranulare Konfigurierbarkeit Um die Algorithmen für jeden Anwendungsfall automatisiert optimieren lassen zu können, gibt es für alle Konfigurationsparameter jeweils verschiedene Aspekte, die Methoden zum Lesen der Konfiguration beeinflussen. Ist zum Beispiel vorgegeben, dass das Position des MSByte aller Signale aller IPDUs im Kommunikationssystem immer konstant ist, so muss dies einerseits in der Konfiguration nicht mehr gespeichert werden. Damit fällt auch das Lesen aus der Konfiguration weg. In Listing 3.7 und 3.8 sind beispielhaft die Slices für konstante und variable MSByte Position gegenübergestellt.

```
slice struct {
  RxByteIndexType
  LsByteIndex () const {
    return dOS_COM_ASCOM_RX_SIGNAL_MAX_MS_BYTE_INDEX;
  }
};
```

Listing 3.7: Slice für konstanten MsByteIndex

Im Fall einer konstanten MSByte Position reicht es den konstanten Wert direkt in `MsByteIndex()` zurückzugeben. Ist der Parameter hingegen variabel bzgl. verschiedener Signale, so muss der Konfiguration eine zusätzliches Attribut hinzugefügt werden. Dies wird in Listing 3.8 unter Zuhilfenahme eines aus der Konfiguration generierten Makros bewerkstelligt, dem man (als vorzeichenloser Parameter) nur den maximal zu erwartenden Wert übergibt. Das Makro wandelt dann je nach Konfiguration bzgl. des zu verwendenden RAMs und der zeitlichen Anforderungen das so deklarierte Attribut in ein Mitglied eines Bitfeldes, oder in ein normales Attribut. Das Bitfeld braucht weniger Speicher, es ist aber etwas aufwändiger, die Daten zu extrahieren.

```
slice struct {
  RxByteIndexType
  MsByteIndex () const {
    return MsByteIndex_;
  }

  RX_SIGNAL_CONFIG_TYPE_FIELD_MEMBER(MsByteIndex_,
          dOS_COM_ASCOM_RX_SIGNAL_MAX_MS_BYTE_INDEX);
};
```

Listing 3.8: Slice für variablen MsByteIndex

3.5.3 Tick Support

Für jedes zeitbezogene Merkmal wird eine Zeitquelle benötigt. In AUTOSAR COM dienen die zwei Funktionen `MainFunctionTx` und `MainFunctionRx` für diesen Zweck. Diese müssen von außen streng periodisch aufgerufen werden.

Um die zeitliche Information innerhalb von AUTOSAR COM weiterzuleiten, wird den Klassen `RxIpduType` und `TxIpduType` jeweils, wenn benötigt, eine Methode `Tick()` und eine statische Methode `TickAll()` hinzufügt.

```
static void
TickAll () {
  // We only tick the TxIpdus of the current configuration
  for (TxIpduIdType i = 0; i < TxIpduType::Multiplicity (); ++i) {
    TxIpdus_[i].Tick ();
  }
}

void
Tick () { }
```

Listing 3.9: In TxIpduType eingeführte Methoden zum Tick Support

In Listing 3.9 und 3.10 ist dies beispielhaft für `TxIpduType` dargestellt. `TickAll()` ruft dabei für alle IPDUs der aktuellen Konfiguration die nichtstatische Methode `Tick()` auf. Schließlich wird der Aufruf von `TickAll()` an die Ausführung von `MainFunctionTx()` gebunden.

```
advice execution ("% os::com::ascom::AsCom::MainFunctionTx (...)")
    : before () {
  // We send a tick to all txipdus
  os::com::ascom::TxIpduType::TickAll ();
}
```

Listing 3.10: TxTick Support

3.5.4 Periodic Transmission Mode

Um für das Merkmal des *Periodic TM* den zeitlichen Bezug zu erhalten, wird der Aufruf von `TxIpduType::PeriodicTMTick()` mit einem Advice wie in Listing 3.11 in die Ausführung von `TxIpduType::Tick()` gewoben.

```
advice execution ("% os::com::ascom::TxIpduType::Tick (...)")
    : after () {
  tjp->that ()->PeriodicTMTick ();
}
```

Listing 3.11: Ausschnitt aus dem PeriodicTM Aspekt

Bei erfolgtem Tick wird wie in Listing 3.12 zu sehen überprüft, ob der Countdown für die periodische Übertragung läuft. Wenn ja, dann wird dieser pro Tick um 1 verringert, bis er auf 0 abgesunken ist. In diesem Moment wird der Countdown wieder aufgezogen und ein Übertragungswunsch mit dem Aufruf der Methode `TxIpduType::TxRequest()` eingeleitet.

Zum Aufziehen des Countdowns wird der Konfigurationsparameter `TimePeriodFactor` in den Countdown kopiert. Interessant ist dabei, dass ein Countdownwert von 0 gleichzeitig auch einen nicht laufenden Countdown kodiert, d.h. wenn der Wert 0 konfiguriert wurde werden keine periodischen Übertragungsünsche gestartet. Ist er verschieden von 0, wird immer gezählt. Somit muss nirgends gespeichert werden, ob der *Periodic TM* für die aktuelle TX-IPDU aktiviert worden ist.

```
void
PeriodicTMTick () {
  if (PeriodicTMRunning ()) {
    if (--PeriodicTMCounter_ == 0) {
      TxRequest ();
      PeriodicTMWindup ();
    }
  }
}

void
PeriodicTMWindup () {
  PeriodicTMCounter_ = PeriodicTMTimePeriodFactor ();
}
```

Listing 3.12: Periodic TM Slice für TxIpduType

Erweiterung für initialen Offset Die Spezifikation sieht pro IPDU auch einen Konfigurationsparameter für einen initialen Offset vor, mit dem die periodische Übertragung beim Start der IPDU verzögert wird. Dies wird durch Modifikation der Initialisierung der Countdowns erreicht.

3.5.5 Transmission Deadline Monitoring

Beim Stoppen einer IPDU, die einen laufenden Countdown für das *Transmission DM* haben, muss der Countdown abgebrochen werden und sofort eine Benachrichtigung der fehlerhaften Übertragung gestartet werden. Dies kann direkt in den Advice in Listing 3.13 übernommen werden.

```
advice execution ("% os::com::ascom::TxIpduType::Stop (...)")
    : before () {
  if (tjp->that ()->DMRunning ()) {
    tjp->that ()->DMCancel ();
    tjp->that ()->DMEvent ();
  }
}

advice execution ("% os::com::ascom::TxIpduType::TxRequestPeriodic (...)")
    : after () {
  tjp->that ()->DMWindup ();
}

advice
    execution ("% os::com::ascom::TxSignalType::ProcessSendSignal (...)")
    && args (TxIpdu, SignalDataPtr)
    : after (os::com::ascom::TxIpduType& TxIpdu,
             const void * SignalDataPtr) {
  if (*tjp->result() == os::com::ascom::E_OK) {
    TxIpdu.DMWindup ();
  }
}
```

Listing 3.13: Ausschnitt der Advices zum Tranmission Deadline Monitoring

Zu bestimmten Zeitpunkten muss der Countdown aufgezogen werden. Dies ist genau dann der Fall, wenn entweder ein periodischer Übertragungswunsch generiert wurde, oder wenn ein Sendewunsch eines Dienstnutzers erfolgreich angenommen wurde. Man beachte, dass bei gestoppter TX-IPDU implizit kein `DMWindup()` aufgerufen wird, da weder periodische Requests erfolgen, noch `ProcessSendSignal()` jemals bei gestoppter TX-IPDU ein `E_OK` zurückgibt.

Es ist wichtig, bei der Umsetzung darauf zu achten, dass ein bereits aufgezogener Countdown ein erneutes Aufziehen ignoriert, da ja bereits auf eine Bestätigung gewartet wird.

Die von dem Aspekt benötigten Methoden in `TxIpduType` werden durch einen Slice hinzugefügt, der in Ausschnitten in Listing 3.14 dargestellt ist:

DMRunning() Testet, ob der Countdown gerade läuft.

DMEffectiveTimeoutFactor() Diese Methode gibt hier nur den konfigurierten Timeout-Factor zurück. Diese Methode wird von anderen Aspekten noch beeinflusst.

DMWindup() Diese Methode speichert den konfigurierten Timeout im Counter, falls dieser noch nicht läuft.

DMTick() Bei laufendem Countdown wird pro Tick um 1 heruntergezählt. Bei 0 wird eine Benachrichtigung gestartet. Bei nicht laufendem Countdown wird keine Aktion durchgeführt.

```
Bool DMRunning () { return DMCounter_; }

TxDMCounterType DMEffectiveTimeoutFactor ()
                  { return DMTimeoutFactor (); }

void
DMWindup () {
  if (!DMRunning ()) DMCounter_ = DMEffectiveTimeoutFactor ();
}

void DMTick () {
  if (DMRunning ()) {
    DMCounter_--;
    if (!DMCounter_) {
      DMIndication ();
    }
  }
}
```

Listing 3.14: Ausschnitt des Slice für TxIpduType zum Tranmission DM

Es bleibt anzumerken, dass die bei Erstellung der Konfiguration ein Timeoutparameter, der verschieden von 1 ist, um 1 erhöht werden muss, damit die Timeouts zum richtigen Zeitpunkt ablaufen. Da nun aber periodische Übertragungen eigentlich einen Tick zu lange zählen, muss bei einem Tick zuerst ein eventuell zu erledigendes `TxRequestPeriodic()` und damit `DMWindup()` aufgerufen werden, und erst anschließend noch bei der Bearbeitung des gleichen Ticks `DMTick()`, damit der Countdown gleich um einen Tick verringert wird, um den er zu hoch gestartet wurde.

Dadurch wird vermieden, dass man zur Laufzeit bei jedem `DMWindup()` erst noch entscheiden muss, ob der Windup durch einen periodischen oder direkten Übertragungswunsch generiert wurde.

Wenn für eine IPDU kein TimeoutFactor konfiguriert wurde, dann wird der Konfigurationsparameter mit 0 initialisiert. In diesem Fall wird ebenfalls bei erfolgtem Übertragungswunsch `DMWindup()` aufgerufen. Wird nun bei nicht laufendem Countdown der Konfigurationsparameter in den Countdown kopiert, so bleibt dieser gestoppt. Hierfür ist keine zusätzliche Fallunterscheidung nötig.

Erweiterung für initialen Timeout Um beim Startup ein anderes als das periodische Verhalten zu ermöglichen, kann in der Konfiguration ein abweichender TimoutFactor für den ersten Lauf nach Initialisierung angegeben werden. Dies wird hier durch eine

hier nicht genauer erläuterte Modifikation der Methode `DMEffectiveTimeoutFactor()` erreicht.

3.5.6 Minimum Delay Timer

Das Merkmal *Minimum Delay Timer* ist ebenfalls als Aspekt implementiert. Dieser Aspekt webt um die Funktion `TxIpduType::TxRequest()` mit höchster Priorität die Abfrage eines Countdowns ein, der unter bestimmten Bedingungen aufgezogen wurde (Listing 3.15). Ist der Countdown 0, so ist eine Übertragung erlaubt, anderenfalls wird vermerkt, dass für die entsprechende IPDU ein Übertragungswunsch (noch) nicht ausgeführt werden konnte.

```
advice execution ("% os::com::ascom::TxIpduType::TxRequest (...)")
    : around () {
  if (tjp->that ()->MDRunning ()) {
    // If the countdown is running we will defer the transmission
    tjp->that ()->MDDeferTransmission ();

    // Application gets E_OK
    *tjp->result () = AS::Std::E_OK;
  } else {
    // We can directly request transmission
    tjp->proceed ();

    // If TxRequest was successful we must start the countdown
    if (*tjp->result () == AS::Std::E_OK) {
      tjp->that ()->MDWindup ();
    }
  }
}

advice execution ("% os::com::ascom::TxIpduType::Tick (...)")
    : before () {
  tjp->that ()->MDTick ();
}

advice execution ("% os::com::ascom::TxIpduType::TxConfirmation (...)")
    : after () {
  tjp->that ()->MDWindup ();
}
```

Listing 3.15: Ausschnitt des Aspekts zum Minimum Delay Timer

Um die zeitliche Abhängigkeit einzuführen, wird dem periodischen Tick der TX-IPDU ein `MDTick()`-Aufruf hinzugefügt, der einen laufenden Countdown um eins verringert und beim Ablaufen des Countdowns überprüft, ob ein Übertragungswunsch verzögert wurde. Ist dies der Fall, so wird im `MDTick()` ein weiterer Übertragungswunsch gestartet.

Der Countdown wird immer dann aufgezogen, wenn eine Bestätigung für die Übertragung einer bestimmten IPDU erhalten wurde. Wichtig ist hierbei im Gegensatz zum

```
...
void
MDTick () {
  if (MDRunning ()) {
    if (--MDCounter_ == 0) {
      // Check whether we have a deferred Transmission request
      if (MDDeferredTransmissions_.TestAndClear (Id ())) {
        TxRequest ();
      }
    }
  }
}
```

Listing 3.16: Umsetzung der Methode `MDTick()`

Countdown des *Periodic TM*, dass der Countdown auch aufgezogen werden kann, wenn er noch läuft, damit auch zu spät empfangene Empfangsbestätigungen eine Verzögerung verursachen.

Die Spezifikation erwähnt in diesem Zusammenhang, dass eine fehlerhafte Applikation durch mehrmalige, direkt aufeinanderfolgende Sendewünsche innerhalb des Zeitfensters bis zur empfangenen Bestätigung eine drastische Unterschreitung der*Minimum Delay Time* zur Folge haben kann.

Weiterhin sagt die Spezifikation nichts Konkretes zu dem Fall, dass für eine IPDU keine Bestätigung möglich ist (keine Unterstützung von der Netzwerkschicht), aber dennoch eine Minimum Delay Time konfiguriert wurde.

Als Lösung bietet sich an, zusätzlich zum Aufziehen des Countdowns bei erhaltener Übertragungsbestätigung diesen auch direkt nach erfolgreicher Weitergabe eines Übertragungswunsches an die nächste Schicht zu starten. Auch durch diesen zusätzlichen Mechanismus kann keine Garantie gegeben werden, dass die mininalen Zeitabstände zwischen zwei Übertragungen auf dem Bus unter allen Umständen eingehalten werden, aber es kann garantiert werden, dass der durchschnittliche zeitliche Abstand zweier Übertragungen immer größer oder gleich der konfigurierten *Minimum Delay Time* ist.

Dies ist offentsichtlich, da Autosar COM für die entsprechende IPDU durch den beschriebenen Mechanismus nur Übertragungswünsche mit Mindestabstand der konfigurierten *Minumum Delay Time* generieren kann.

Es bleibt anzumerken, dass eine zeitlich Überwachung mit vollständiger Garantie (mit Ausnahme eines Hardwaredefekts) für zeitliche Abstände eigentlich nur auf der untersten Schicht gemacht werden kann, da „unterhalb“ dieser Schicht keine weitere Pufferung erfolgt. Dies ist in AUTOSAR COM für die CAN-Infrastrukur aber nicht vorgesehen.

4 Evaluation

Die folgende Evaluierung soll einen kleinen Einblick in die Skalierbarkeit und deren gegenwärtige Grenzen geben und aufzeigen, an welchen Stellen noch Verbesserungsbedarf besteht.

4.1 Vorstellung der evaluierten Elemente

Als kurze Einführung sollen hier die evaluierten Methoden kurz vorgestellt werden. Dabei wurden nur Methoden erwähnt, die nicht von vornerein immer `inline` sind. In den gezeigten Größentabellen werden sowohl die Größen der Konfigurations- und Laufzeitcontainer dargestellt, als auch die Codegrößen der Methoden.

AsCom::Init Die Methode konfiguriert und initialisiert das gesamte AsCom-Subsystem. Dabei werden die einzelnen Initialisierungsmethoden der Objekte aufgerufen.

...::Init Die `...::Init`-Methode liegt in den meisten Containern in zwei Varianten vor. Eine davon erhält keinen Parameter, die andere Variante erhält einen Zeiger auf den entsprechenden Konfigurationscontainer. In den Tabellen ist die erste Variante mit (), die zweite mit (*) gekennzeichnet.

...::Start Die Start-Methoden der Container sorgen dafür, dass die Container Daten der Applikation verarbeiten können. Dabei kann der Aufrufer auch eine Reinitialisierung der internen Puffer und Countdowns veranlassen, was intern durch den Aufruf der parameterlosen Init-Methode erledigt wird.

TxSignalType::SendSignal Dies ist die sendeseitig über die AS-Klasse nach außen sichtbare Methode zum Senden von Signalen.

TxIpduType::TxRequest Wird immer dann aufgerufen, wenn intern ein Sendewunsch einer IPDU generiert werden soll.

TxIpduType::CopyFromSignalToSdu Diese Methode kopiert den Inhalt des Signals an die konfigurierte Stelle innerhalb zugeordneten IPDU.

TxIpduType::TickAll Die Methode erledigt alle periodischen Aufgaben, die für eine Instanz der Klasse `TxIpduType` anfallen.

RxIpduType::RxIndication Die Methode wird von unteren Schichten aufgerufen, um anzuzeigen, dass neue Daten erhalten worden sind. Hier wird intern die Verteilung der Daten an die Signale angestoßen.

RxIpduType::ReceiveSignal Dies ist die empfangsseitig über die AS-Klasse nach außen sichtbare Methode zum Empfang von Signalen.

4.2 Beschreibung der Konfigurationen

Da eine komplette Evaluation aller Merkmalkombinationen nicht möglich ist, wird hier nur ein kleiner Ausschnitt aller möglichen Merkmalkombinationen evaluiert.

Dabei soll vornehmlich Augenmerk auf die grundsätzlichen Merkmale des *Transmission Modes* und der *Transfer Property* gelegt werden.

Eine weitere Variation der evaluierten Konfigurationen wird durch das Erlauben von Signalen erreicht, die innerhalb der IPDU kleiner sind als isoliert im Signalpuffer.

Weiterhin wurden die in der Analysephase ermittelten Abhängigkeiten der Nutzung der Laufzeit- und Konfigurationscontainer im Rahmen der Entwicklung des Prototypen durch zusätzliche Attributierung der Merkmale weiter verfeinert. Daher wurde in der Evaluation auch darauf Wert gelegt, einige unterschiedliche Konstellationen mit einer Gegenüberstellung von konstanten und variablen Parametern zu vollziehen. Dabei wurden IPDU- und Signallängen, Byteindizes, und Signalalignment evaluiert.

Um die unterschiedlichen Größen der benutzten Strukturen deutlich zeigen zu können, wurden die Varianten zur Evaluierung alle so konfiguriert, dass die Strukturen keine Bitfelder einhalten, sondern immer normale Attribute.

Konfigurationsnr.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Transmission Mode															
None	●	●	●	●	●	-	-	-	-	-	●	●	●	●	●
Direct	-	-	-	-	-	●	●	●	●	●	●	●	●	●	●
Periodic	-	-	-	-	-	-	-	-	-	-	●	●	●	●	●
Mixed	-	-	-	-	-	-	-	-	-	-	●	●	●	●	●
Transfer Property															
Pending	●	●	●	●	●	-	-	-	-	-	●	●	●	●	●
Triggered	-	-	-	-	-	●	●	●	●	●	●	●	●	●	●
Signallänge in IPDU ≤ echte Signallänge	-	-	-	-	-	-	-	-	-	-	●	●	●	●	●
Var. Ipdulänge	-	●	●	●	●	-	●	●	●	●	-	●	●	●	●
Var. Byteindizes	-	-	●	●	●	-	-	●	●	●	-	-	●	●	●
Var. Signallänge	-	-	-	●	●	-	-	-	●	●	-	-	-	●	●
Signale ohne Bytealignment	-	-	-	-	●	-	-	-	-	●	-	-	-	-	●

Tabelle 4.1: Evaluierte Teilkonfigurationen

Konfigurationsaufteilung Die Konfigurationsaufteilung ist in Tabelle 4.1 dargestellt. Die Konfigurationen 1 bis 5 wurden ausschließlich mit dem *None Transmission Mode*

und der *Pending Transfer Property* evaluiert, d.h. die verarbeiteten Daten müssen in diesen Fällen von einer Netzwerkschnittstelle selbst abgeholt werden.

Konfiguration 6 bis 10 übermittelt alle Signale immer mit der *Triggered Transfer Property* und dem *Direct Transmission Mode*, d.h. alle Signalübertragungen resultieren sofort in einem Übertragungswunsch der IPDU.

Bei Konfiguration 11 bis 15 ist sowohl *Direct Transmission Mode* und *Triggered Transfer Property* frei wählbar und es ist zusätzlich erlaubt, dass die konfigurierte Länge der Signale innerhalb der IPDU auch kleiner sein kann, als die Länge des Signals in der Applikation. In den Konfigurationen 1 bis 10 müssen diese Längen übereinstimmen.

Die Konfigurationen 1, 6 und 11 erlauben keine variable Konfiguration von IPDU und Signallängen und Start- und Endpositionen der Signale innerhalb der IPDU. Die Konfigurationen 2, 7 und 12 erlauben bereits, die Ipdulänge variabel zu halten. 3, 8 und 13 fügen die Möglichkeit variabler Bytepositionen der Signale innerhalb der IPDU hinzu. Die Konfigurationen 4, 9 und 14 erlauben darüberhinaus, die Signallänge zwischen verschiedenen Signale zu variieren. Die Konfigurationen 5, 10 und 15 erlauben schließlich, die Signale frei innerhalb der IPDU auch an Stellen zu positionieren, die nicht an Bytegrenzen anliegen.

4.3 Größenvergleich

Im folgenden werden einige der gemessenen Effekte diskutiert und erläutert. Dazu werden die vorher präsentierten Konfigurationsbeispiele miteinander verglichen.

4.3.1 Variation der Konfigurationsdaten

Die Variation der Menge an Konfigurationsdaten äußert sich unter anderem darin, dass die meisten Konfigurationscontainer wesentliche Unterschiede in ihrer Größe zeigen. Es ist nun beispielsweise so, dass die Information, wie lange eine IPDU ist, nur dann im Konfigurationscontainer gespeichert werden muss, wenn die Attribute der Variantenkonfiguration eine unterschiedliche untere und obere Grenze für diesen Wert festlegen. Ist die untere und obere Grenze gleich, so muss für diesen Parameter keine zusätzliche Information in jedem einzelnen Konfigurationscontainer abgelegt werden (siehe `...ConfigType` in Tabelle 4.2 und 4.3). Vielmehr reicht es in diesem Fall, die konstante Länge im Programmcode direkt abzulegen.

Diese Tatsache führt weiterhin dazu, dass das intern immer benötigte Kopieren der Konfiguration in die Laufzeitcontainer innerhalb der Methode `AsCom::Init` weniger aufwändig ist und damit der Initialisierungscode kleiner ausfällt.

4.3.2 Konfigurations- vs. Laufzeitcontainer

Bei der Gegenüberstellung von Konfigurationscontainer und Laufzeitcontainer lässt sich ein unterschiedliches Verhalten für die sendeseitige und empfangseitige Verarbeitung feststellen.

Konfigurationsnr.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
AsCom::Init(*)	1a0	1a2	1b6	1d2	1d4	1a0	1a2	1b8	1d4	1d6	1ac	1a8	1ba	1d6	1d6
TxIpduConfigType	2	4	4	4	4	2	4	4	4	4	8	8	8	8	8
TxIpduType	6	c	c	c	c	6	c	c	c	c	e	12	12	12	12
Init(*)	6	a	a	a	a	6	a	a	a	a	16	16	16	16	16
Start	a6	a8	ac	ac	ae	a6	a8	ac	ac	ae	be	be	be	c0	c0
TxRequest	-	-	-	-	-	5e	60	60	60	60	5e	62	62	62	62
TickAll	-	-	-	-	-	-	-	-	-	-	8e	90	90	90	90
RxIpduConfigType	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
RxIpduType	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Init()	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Start	80	80	86	9e	9e	80	80	86	9e	9e	80	80	86	9e	9e
RxIndication	5e	5e	60	62	62	5e	5e	60	62	62	5e	5e	60	62	62

Tabelle 4.2: Struktur- und Codegrößen von IPDUs und Initialisierung

In der sendeseitigen Verarbeitung unterscheiden sich die Größen der Signalcontainer (`TxSignalConfigType` für den Konfigurations- und `TxSignalType` für den Laufzeitcontainer) nicht. Das rührt daher, dass sämtliche in dieser Evaluation beurteilten Konfigurationen keinen signalbasierten Puffer oder anderweitige Laufzeitparameter benötigen. Im Gegensatz dazu werden in den IPDU-Containern der sendeseitigen Verarbeitung Puffer zur Zwischenspeicherung der SDU benötigt. Daher ist der Laufzeitcontainer `TxIpduType` größer als der Konfigurationscontainer `TxIpduConfigType`. Innerhalb der Empfangsver-

Konfigurationsnr.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
TxSignalConfigType	1	1	4	4	6	1	1	4	4	6	2	2	4	6	8
TxSignalType	1	1	4	4	6	1	1	4	4	6	2	2	4	6	8
Init()	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
SendSignal	5c	5c	6c	6c	4c	98	98	aa	aa	92	ae	ae	c2	c2	ac
CopyFromSignalToSdu	-	-	-	-	ce	-	-	-	-	ce	-	-	-	-	ce
RxSignalConfigType	1	1	2	4	6	1	1	2	4	6	1	1	2	4	6
RxSignalType	4	4	6	c	e	4	4	6	c	e	4	4	6	c	e
ReceiveSignal	c	c	e	18	18	c	c	e	18	18	c	c	e	18	18
RxIndication	12	12	20	20	5e	12	12	20	20	5e	1a	1a	4a	4a	94

Tabelle 4.3: Struktur- und Codegrößen von Signale

arbeitung zeigt sich das gegenteilige Bild. Während die RX-Signale jeweils einen eigenen Signalpuffer benötigen und damit Instanzen von `RxSignalType` größer sind, als die von `RxSignalConfigType`, haben die RX-IPDUs keine eigenen Puffer.

Bei der Betrachtung der Größe der Empfangscontainer in Tabelle 4.2 fällt weiterhin auf, dass diese bei den präsentierten Konfigurationen gleich bleiben. Dies demonstriert, dass die sich hauptsächlich auf die Sendeseite auswirkenden Merkmale die Containergröße der Empfangsseite nicht beeinflussen.

4.3.3 Variabilität von Transfer Property und Transmission Mode

In den Konfigurationen 1 bis 5 ist die *Transfer Property* konstant auf *Pending* eingestellt. Dieser Umstand führt dazu, dass innerhalb von `TxSignalType::SendSignal` die Methode `TxIpduType::TxRequest` niemals aufgerufen wird. Daher findet man für die Konfigurationen 1 bis 5 auch keine Implementierung von `TxIpduType::TxRequest` im Programmcode.

Mit der Erweiterung in den Konfigurationen 6 bis 10 wird `TxIpduType::TxRequest` immer aufgerufen, da die *Transfer Property* konstant auf *Triggered* eingestellt ist und alle IPDUs konstant mit einem *Direct Transmission Mode* konfiguriert sind.

Die letzte Erweiterung in den Konfigurationen 11 bis 15 bringt eine variable Einstellung von *Transfer Property* und *Direct Transmission Mode* mit sich.

Die beschriebenen Erweiterungen spiegeln sich wie erwartet in den entsprechenden Codegrößen in Tabelle 4.3 wieder. Während anfänglich `TxSignalType::SendSignal` noch relativ klein ausfällt, wird sie durch die erste Erweiterung bereits deutlich vergrößert. Die zweite Erweiterung, die den vollen Funktionsumfang bietet, resultiert schließlich im größten Programmcode.

4.3.4 Bytealignment der Signale

Im Vergleich der Konfigurationen von 4, 9, 14 zu 5, 10, 15 ist ein erheblicher Unterschied in der Summe der Codegrößen der Methoden `SendSignal` und `CopyFromSignalToSdu` von der Klasse `TxSignalType` und `RxSignalType::RxIndication` zu beobachten.

Der Einfüge- bzw. Auslesevorgang in bzw. aus IPDUs braucht im Fall von Signalen ohne Bytealignment sowohl wesentlich mehr Programmcode, als auch zwei zusätzliche Parameter im Konfigurationscontainer zum Speichern der Bitpositionen innerhalb des Start- und Endbytes des Signals in der IPDU.

Es sei angemerkt, dass die zusätzliche Methode `CopyFromSignalToSdu` eigentlich in jeder Konfiguration vorhanden ist und durch `SendSignal` aufgerufen wird. Diese Methode wird lediglich in den meisten Fällen durch Inlining in den Aufrufer integriert. Diese Tatsache erklärt auch den Grund, weshalb - isoliert betrachtet - `SendSignal` bei diesem Vergleich kleiner wird.

4.3.5 Grenzen der derzeitigen Implementierung

Bei der Evaluation ist aufgefallen, dass die derzeitigen `...::Start`-Methoden noch optimiert werden könnten, und das Inlining der zwei Varianten der `...::Init`-Methoden innerhalb dieser Methoden noch effizienter eingesetzt werden könnte.

Ein Review des Programmcodes hat außerdem ergeben, dass für eine große Anzahl an IPDUs, die *Deadline Monitoring* benutzen, ein eigener Scheduler für die Verwaltung der Deadlines sinnvoll wäre, da die Benutzung von Countdowns innerhalb jeder einzelnen IPDU für eine große Anzahl von IPDUs zunehmend ineffizient wird.

5 Fazit

Das aspektorientierte Programmierparadigma ist ein weiterer Schritt, in immer komplexeren Softwareumgebungen den Überblick zu behalten. Die querschneidenden Belange lassen sich auf einfache Art und Weise formulieren.

Die konsequente Benutzung von aspektorientierten Techniken verlangt allerdings auch, dass bestimmte Vorgehensweisen immer eingehalten werden. So muss man, um sich alle Varianten von Manipulation einer Klasse zu ermöglichen, beispielsweise immer Get und Set-Methoden benutzen, anstatt direkt auf die Attribute zuzugreifen. Auch wenn dies keinen Runtimeoverhead erzeugt (Inlining), so muss dieses Schema auch innerhalb der Klasse durchweg eingehalten werden. Um über den Zugriff auf Objektattribute mehr Kontrolle zu haben, wäre die Einführung des in AspectJ bereits vorhandenen set()-Advice, ein angenehme Alternative.

Es fällt weiterhin auf, dass die eigentlich durch die Aspektorientierung gewünschten Seiteneffekte bei zu umfassend geschriebenen Pointcuts leicht auch weitere, ursprünglich nicht eingeplante Seiteneffekte erzeugen können.

Eine Idee, weiteren Programmieraufwand zu sparen, wäre es, templatisierte Aspekte und Slices zu implementieren. Damit könnte man einige Aspekte bzw. Slices, die beispielsweise bis auf Typenangaben gleich sind in einem Aspekt/Slice darstellen. Ob dies auf einfache Art und Weise in C++ Code transformiert werden kann, soll und muss an anderer Stelle geklärt werden.

Literaturverzeichnis

[AUT08a] AUTOSAR Layered Software Architecture (version 2.2.1). Auxiliary Document, AUTOSAR GbR, Feb 2008. http://www.autosar.org.

[AUT08b] AUTOSAR Specification of CAN Interface (version 3.0.2). Standard, AUTOSAR GbR, Feb 2008. http://www.autosar.org.

[AUT08c] AUTOSAR Specification of CAN Transport Layer (version 2.2.1). Standard, AUTOSAR GbR, Feb 2008. http://www.autosar.org.

[AUT08d] AUTOSAR Specification of Communication (version 3.0.2). Standard, AUTOSAR GbR, Feb 2008. http://www.autosar.org.

[AUT08e] AUTOSAR Specification of FlexRay interface (version 3.0.2). Standard, AUTOSAR GbR, Feb 2008. http://www.autosar.org.

[AUT08f] AUTOSAR Specification of I-PDU Multiplexer (version 1.2.2). Standard, AUTOSAR GbR, Feb 2008. http://www.autosar.org.

[AUT08g] AUTOSAR Specification of LIN interface (version 2.0.1). Standard, AUTOSAR GbR, Feb 2008. http://www.autosar.org.

[AUT08h] AUTOSAR Specification of Operating System (version 3.0.2). Standard, AUTOSAR GbR, Jun 2008. http://www.autosar.org.

[AUT08i] AUTOSAR Specification of PDU Router (version 2.2.2). Standard, AUTOSAR GbR, Feb 2008. http://www.autosar.org.

[AUT08j] AUTOSAR Specification of RTE (version 2.0.1). Standard, AUTOSAR GbR, Feb 2008. http://www.autosar.org.

[AUT08k] AUTOSAR GbR. About AUTOSAR. Homepage of AUTOSAR, 2008. http://www.autosar.org.

[Dij74] Edsger W. Dijkstra. On the role of scientific thought. http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF, published as [Dij82], August 1974.

[Dij82] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[OSE04] OSEK/VDX Communication (version 3.0.3). Standard, OSEK, Jul 2004. http://www.osek-vdx.org.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Studienarbeit selbst und ohne fremde Hilfe verfasst und nur die im Literaturverzeichnis angegebenen Quellen zugrunde gelegt habe.

Nürnberg, 11. September 2008

Christian Meier