Diplomarbeit an der Technischen Universität Kaiserslautern
Fachbereich Informatik

Entwicklung und Integration von QoS-Mikroprotokollen zur Steuerung eines Fluggerätes über WLAN

Christian Webel
Juni 2004

AG
Vernetzte Systeme

Erklärung

Ich erkläre hiermit, dass ich die Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen verwendet habe.

Kaiserslautern, den 28. Juni 2004

Christian Webel

Inhaltsverzeichnis

1 Einleitung 1
1.1 Einleitung und Motivation 1
1.2 Überblick 1

2 Grundlagen 3
2.1 Die verwendete Spezifikationssprache SDL-96 3
2.2 Basistechnologie 3
2.2.1 Die verwendete Hardware 3
2.2.2 Der QoS-WLAN-Treiber 4
2.3 Dienstgüte (Quality of Service) 4
2.3.1 Dienstgüte-Spezifikation 5
2.3.2 Dienstgüte-Bereitstellung 6
2.3.3 Dienstgüte-Kontrolle 7
2.3.4 Dienstgüte-Management 8
2.4 SDL Design-Patterns 9
2.5 Mikroprotokolle 9
2.5.1 Beschreibung von Mikroprotokollen 9
2.5.2 Die Mikroprotokollbibliothek 16
2.5.3 Komposition bzw. Integration von Mikroprotokollen 17
2.5.4 SDL-Mikroprotokoll Entwicklungsprozess 17
2.6 Zusammenfassung 19

3 Applikation. 21
3.1 Das Luftschiff 21
3.2 Beschreibung der Anwendung 22
3.3 Architektur der Anwendung 22
3.4 Funktionalität der Anwendung 23
3.4.1 Die Videoanwendung 23
3.4.2 Die Steuerungsanwendung 25
3.5 Dienstgüte-Spezifikation der Anwendung 27
3.5.1 Level of Service 29
3.5.2 Flow Performance Specification 29
3.5.3 Qos-Management Policies 32
3.5.4 Cost of Service 34
3.5.5 Flow synchronisation specification 34
3.6 Zusammenfassung 34

4 Middleware 35
4.1 Funktionalitäten einer Middleware 35
4.1.1 Kommunikation 35
4.1.2 Sicherheit 36
4.1.3 Dienstgüte 36
4.2 Anforderungen der Anwendung an die Middleware 36
4.3 Dienstgüte-Bereitstellung 37
4.3.1 Dienstgüte-Mapping 37
4.3.2 Zugangstest und Reservierungsprotokoll 41
4.4 Dienstgüte-Kontrolle 41
4.5 Dienstgüte-Management 41

4.6 Architektur 42
4.7 Applikationsspezifische Middleware 43
4.8 Dienstgütespezifische Middleware 46
4.9 Zusammenfassung 48

5 Mikroprotokolle & Design-Patterns 49
5.1 Mikroprotokolle und Design Patterns der applikationsspezifischen Middleware 49
5.1.1 Die Codierer und Decodierer VideoCoDec und PilotCoDec 49
5.1.1.1. Designproblem 49
5.1.1.2. Design-Pattern SimpleSignalCoDecArchitecture 50
5.1.1.3. Design Pattern SDLSignal2ASN.1 52
5.1.1.4. Design Pattern SimpleSignalCoDecBehaviour 56
5.1.2 PSU und PSUTrigger 59
5.1.2.1. Designproblem 59
5.1.2.2. Design-Pattern Watchdog 59
5.1.3 Das Mikroprotokoll VideoMapping 62
5.1.3.1. Designproblem 62
5.1.3.2. Design Pattern QosMapping 62
5.1.3.3. Die Beschreibung von VideoMapping 64
5.1.4 Das Mikroprotokoll VideoScaling 66
5.1.4.1. Designproblem 66
5.1.4.2. Die Beschreibung von VideoScaling 67
5.1.5 Das semi-generische Mikroprotokoll DelayMapping 69
5.1.5.1. Beschreibung 69
5.1.5.2. Die Prozesstypen PilotServerMapping und PilotClientMapping 71
5.1.6 Das semi-generische Mikroprotokoll DelayScaling 72
5.1.6.1. Beschreibung 72
5.1.6.2. Die Prozesstypen PilotServerScaling und PilotClientScaling 74
5.2 Mikroprotokolle und Design-Patterns der dienstgütespezifischen Middleware 77
5.2.1 QosReservationLinkMgr 77
5.2.2 QosMonitoring 82
5.2.2.1. QosTransferMgr 83
5.3 Anmerkungen 83
5.4 Zusammenfassung 84

6 Ergebnis und Ausblick 85

7 Literatur. 87
Anhang A: Design-Patterns 89
SIMPLESIGNALCODECARCHITECTURE 91
SDLSIGNALS2ASN.1 95
SIMPLESIGNALCODECBEHAVIOR 103
WATCHDOG 109
QOSMAPPING 115
Anhang B: ASN.1 Datentypen und -strukturen 119
BasicTypesASN1 121
VideoASN1 123
PilotASN1 125
Anhang C: Hinweise zur Benutzung von ASN.1 und Telelogic Tau. 127
Anhang D: Die Mikroprotokollbibliothek und der Systementwurf (CD) 131

Danksagung

Diese Arbeit entstand in der Arbeitsgruppe Vernetzte Systeme der Technischen Universität Kaiserslautern in der Zeit von Dezember 2003 bis Juni 2004.

Mein besonderer Dank gilt Herrn Prof. Reinhard Gotzhein für das in mich gesetzte Vertrauen und die Möglichkeit, diese Diplomarbeit anzufertigen. Insbesondere möchte ich mich bei ihm für seine Tätigkeit als mein Studienberater und für die vielen anregenden Diskussionen bedanken.

Herrn Dipl. Technoinf. Ingmar Fliege und Herrn Dipl. Inf. Alexander Geraldy danke ich für die vielen kritischen Diskussionen und Anregungen, die zum Gelingen dieser Arbeit beigetragen haben.

Großer Dank gilt meinen Eltern, die mir mein Studium ermöglicht haben und mir stets zur Seite standen.

Mein ganz besonderer Dank gilt meiner Freundin Ingrid Maywurm, die mich mit sehr viel Verständnis während meines Studiums begleitet hat.

Nicht zuletzt gilt mein Dank den verschiedenen Leuten, die mich während meines Studiums und im Rahmen dieser Arbeit unterstützt haben.

1 Einleitung

1.1 Einleitung und Motivation

In der Kommunikationsdomäne spielen kürzere Entwicklungszeiten, sinkende Ressourcenvorgaben und Qualitätssicherung in großen Systemen eine immer größere Rolle. Noch immer neu sind nicht-funktionalen Anforderungen wie z.B. Dienstgüte, die speziell in Ad-Hoc Netzen neuen Einzug erhalten. Neue Entwicklungen auf hohem Niveau lassen sich ohne Wiederverwendung und Maßschneiderung nicht mehr in dem vorgegebenen Zeit- und Ressourcenrahmen bewältigen. Fertige Standardkomponenten, wie sie schon seit langem im Softwareentwicklungsbereich eingesetzt werden, halten nun auch Einzug in den Kommunikationssektor, in Form von Mikroprotokollen (Kapitel 2.5). Diese gekapselten Funktionalitäten ermöglichen im Gegensatz zur monolithischen Entwicklung eines Protokolls einen hohen Grad an Modularisierung und Wiederverwendung.

Die Maßschneiderung einer Middleware, also die genaue Anpassung von Protokollen oder Protokollkomponenten an die Anforderungen, besteht zum größten Teil aus der Komposition und Integration von kleinen Basisfunktionalitäten, die jedes Kommunikationsprotokoll gemeinsam hat, und der Erweiterung um die anforderungsspezifische Komponenten. Basisfunktionalitäten sind z.B. der Verbindungsauf- oder -abbau oder die Fehlererkennung.

Im Rahmen dieser Diplomarbeit werden neue Mikroprotokolle mit Dienstgüteunterstützung entwickelt und in einer Middleware integriert werden.

Durch die Verwendung von Mikroprotokollen ergeben sich neben der verkürzten Entwicklungszeit weitere Vorteile. Durch die Verwendung fertiger Bausteine werden die potentiellen Fehlerquellen einer Neuentwicklung minimiert und zusätzlich ermöglicht der komponentenbasierte Ansatz eine rasche Prototyperstellung[1].

1.2 Überblick

Das nächste Kapitel beschreibt die nötigen Grundlagen. Es werden Begriffe aus der Welt der Dienstgüte eingeführt und erläutert, die zum Verständnis dieser Arbeit wichtig sind. Desweiteren gibt es einen kurzen Einblick in Design Patterns. Der Hauptteil des Kapitels beschäftigt sich mit Mikroprotokollen. Es werden ein im Rahmen dieser Arbeit entwickeltes Beschreibungstemplate und eine Mikroprotokollbibliothek vorgestellt.

Kapitel 3 beschäftigt sich mit der Applikation „Zeppelin“ (Steuerung eines Fluggerätes und Videoübertragung). Es wird eine Beschreibung der Anwendung gegeben und die einzelnen Funktionalitäten genau erläutert. Anschließend gibt es eine Dienstgütespezifikation nach den in Kapitel 2 vorgestellten Gesichtspunkten.

1. Prototyp: ablauffähiges Modell zur Überprüfung von Ideen oder zum Experimentieren (aus [11], S.103)

Kapitel 4 befasst sich mit der zu entwickelnden Middleware. Es wird die fertige Middleware vorgestellt, ohne auf die genaue Funktionalität der einzelnen Komponenten einzugehen. Ferner wird näher auf die Dienstgütemechanismen in der Middleware eingegangen.

In Kapitel 5 werden die einzelnen Funktionalitäten der Middleware genauer betrachtet. Dazu werden neu erstellten Mikroprotokolle und Design Patterns vorgestellt und zu einer Middleware komponiert.

2 Grundlagen

In diesem Kapitel werden die notwendigen Begriffe und Grundlagen dieser Diplomarbeit kurz erläutert. Der Begriff Dienstgüte wird genauer erläutert und es werden die benötigten Techniken und Methoden vorgestellt, um Systeme mit Dienstgüte zu entwerfen.

Desweiteren werden Mikroprotokolle und eine Mikroprotokollbibliothek mit dem dazugehörigen Beschreibungstemplate, sowie ein darauf basierender Entwicklungsprozess vorgestellt. Dadurch wird es ermöglicht, erstellte Mikroprotokolle zu sammeln und wiederzuverwenden.

2.1 Die verwendete Spezifikationssprache SDL-96

SDL-96 (Specification and Description Language) ist eine objektorientierte, formale Beschreibungssprache zur Spezifikation von verteilten Systemen und Protokollen. Näheres ist unter [7] und [12] zu finden. Das in dieser Arbeit verwendete Tool ist Tau 4.4 von der Firma Telelogic.

2.2 Basistechnologie

2.2.1 Die verwendete Hardware

In dem Zeppelin findet neben einem Embedded-PC im 3.5" Format eine Webcam von Phillips (ToUCam Pro 740K) und ein USB-WLAN-Stick von Netgear (MA111) mit einem Prism2,5-

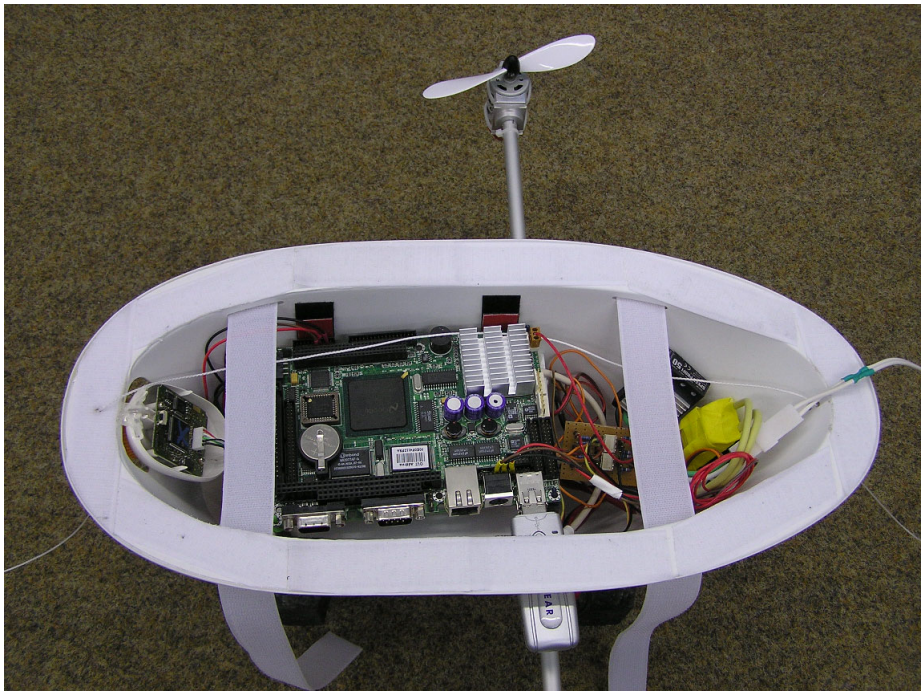


Abb. 2-1: Die Gondel des Zeppelins

Chipsatz. Abbildung 2-1 zeigt die Hardware der Gondel des Demonstrators mit den erwähnten Komponenten.

Der Embedded-PC GX1LCD/3,5´´ Plus der Firma INSIDE Technology besitzt einen Geode GX1 Prozessor mit 300MHz, durch seine kleinen Abmessungen und dem geringen Stromverbrauch eignet er sich für den Einsatz in einem Fluggerät. Über seine beiden USB-Schnittstellen kann der verwendete WLAN-Stick und die Kamera problemlos angeschlossen werden.

2.2.2 Der QoS-WLAN-Treiber

In [3] wurde ein WLAN-Treiber für den Prism2-Chipsatz mit Dienstgüteunterstützung entwickelt. Dieser Treiber unterteilt das Medium in eine feste Anzahl von Zeitschlitzen. Mit 10 dieser Zeitschlitze ist er jedoch für eine Steuerungsanwendung mit gleichzeitiger Videoübertragung ungeeignet. Daher beschränkt sich diese Diplomarbeit auf die Spezifikation und Simulation einer Middleware, bis ein geeigneter Treiber verfügbar bzw. unser Treiber dahingehend optimiert ist.

2.3 Dienstgüte (Quality of Service)

Der Begriff Dienstgüte gliedert sich in zwei Teile, Dienst (Service) und Güte (Quality). Unter einem Dienst versteht man im Allgemeinen eine Leistung, die von Benutzern oder anderen Diensten dazu verwendet werden kann, eine Aufgabe zu erfüllen. Den Bereitsteller dieser Leistung nennt man Diensterbringer, den Nutzer Dienstnutzer. Ein Dienst kann eine Anwendung oder ein Protokoll sein, wie z.B. Samba oder HTTP oder eine Funktionalität z.B. eine Leitwegbestimmung in einer Netzwerkschicht.

Güte beschreibt die Eigenschaft eines Diensterbringers bzw. eines Dienstes. Mit ihr werden Kriterien wie Verlust, Verzögerung und Durchsatz beschrieben. Die Güte eines Dienstes ist stark von dem verwendeten Netzwerkprotokoll und Basisdienst abhängig.

Dienstgüte beschreibt somit die Eigenschaft, einen Dienst mit einer gegebenen Güte bereitzustellen. Dazu gehören neben der Bereitstellung der benötigten Ressourcen auch Maßnahmen, um die gewünschte Dienstgüte zu verwalten und aufrecht zu erhalten.

In mobilen Ad-Hoc Netzen stellt vor allem die Verwaltung der benötigten Ressourcen ein großes Problem dar, da solche Netze einen hohen Grad an Dynamik besitzen und im Gegensatz zu festverdrahteten Netzen nicht von einer gleichbleibenden Verbindung zwischen zwei Knoten ausgegangen werden kann. Durch Störungen oder Ändern der Topologie (Mobilität) kann sich die Ressourcensituation innerhalb des Netzes schnell ändern, wovon die Dienstgüte natürlich auch betroffen ist.

Im Rahmen dieser Diplomarbeit wird eine Middleware mit Dienstgüteunterstützung für mobile Ad-Hoc Netze entwickelt, welche genau diese Probleme behandelt. Dazu werden im Folgenden grundlegende Methoden und Techniken [5] vorgestellt. Für eine voll funktionsfähige Middleware mit Dienstgüteunterstützung ist es nicht notwendig, dass alle folgenden Punkte verwendet bzw. beachtet werden. Optionale Komponenten wie Flusssynchronisation oder Dienstgütesignalisierung können entfallen, da diese nicht von jeder Anwendung benötigt werden (eine reine Datenübertragung z.B. muss mit keinem anderen Datenstrom synchronisiert werden). Einige Kernaspekte, wie die Leitwegbestimmung oder die Ressourcenreservierung sind jedoch unerlässlich. Optionale Komponenten sind in der folgenden Gliederung mit einem **[opt]** gekennzeichnet.

2.3.1 Dienstgüte-Spezifikation

Die Spezifikation bildet die Grundlage einer jeden Dienstgütebetrachtung. In diesem Zusammenhang besteht sie im Allgemeinen aus den Dienstgüteanforderungen und den Management Policies. Auf jeder Systemebene ist die Spezifikation recht unterschiedlich, jedoch wird immer beschrieben, was benötigt wird und nicht wie es erreicht wird.

Im Wesentlichen benutzt man die Spezifikation dazu, die verschiedenen Dienstgütemechanismen auf den verschiedenen Ebenen festzulegen und zu verwalten. Im Einzelnen besteht eine Spezifikation aus den folgenden Punkten:

flow synchronisation specification [opt]: Die *flow synchronisation spec.* legt fest, wie die Synchronisation zwischen verschiedenen oder gleichartigen Datenflüssen auszusehen hat. Es wird also die maximale Verzögerungsdifferenz der Ströme festgelegt. Diese Differenz ist stark von der Anwendung abhängig. Bei einem Audio- und einem Videostrom z.B. ist eine Synchronisation auf einige Millisekunden genau akzeptabel, wohingegen es z.B. bei einer exakten Positionsbestimmung mit drei Abstandssensoren wichtig ist, dass diese drei Werte bzw. Ströme exakt gleichzeitig bei einem Empfänger ankommen müssen.

flow performance specification: Hier wird die benötigte Performanz der Anwendung beschrieben. Kriterien, die hierbei wichtig sind, sind z.B. Durchsatz, Delay, Jitter oder auch die Verlustrate. Die *flow performance spec.* ist, genau wie die *flow synchronisation spec.*, sehr von der Art der Anwendung abhängig. So spielt z.B. bei der Steuerung eines Fluggerätes die maximale Verzögerung der Steuersignale eine große Rolle, wohingegen bei der Videoübertragung der Durchsatz und der Jitter im Vordergrund stehen.

level of service: Der *level of service* beschreibt die qualitative Festlegung der Dienstgüte. Man unterscheidet zwischen *deterministisch*, *statistisch* und *best effort*.

Deterministisch bedeutet, dass selbst beim Eintreten des worst case (Überlastung des Netz, Fehlerszenario) die benötigte Dienstgüte garantiert werden kann. In mobilen Ad-Hoc Netzwerken kann eine solche deterministische Garantie nicht gegeben werden, da in diesem Fall der worst case das Wegfallen des einzigen Links zwischen zwei Stationen und somit der Totalausfall der Kommunikation ist. Auch verhindert die starke Schwankung der Verbindungsqualität zweier Stationen – durch Störungen (Umwelteinflüsse, störende Sender) oder gegenseitiges Stören – selbst bei Nichtausfall eines Links eine deterministische Vorhersage der verfügbaren Bandbreite.

Statistische Garantien können nur gegeben werden, falls es möglich ist, Reservierungen auf dem Medium zu tätigen. Dann kann mit einer bestimmten Wahrscheinlichkeit die benötigte Dienstgüte geliefert werden. Diese Wahrscheinlichkeit variiert wiederum stark mit den Umgebungseigenschaften. In Bereichen, in denen vermehrt Störung auftritt, können diese Garantien unter Umständen nicht eingehalten werden, in „normalen“ Umgebungen schon. Je größer die benötigte Wahrscheinlichkeit ist, desto mehr Ressourcen müssen reserviert werden, jedoch gilt hier der Umkehrschluss nicht, dass mit der vollständigen Reservierung eine Wahrscheinlichkeit von 100% erreicht wird. Kurz gesagt liefern statistische Garantien mit einer gegebenen (hohen) Wahrscheinlichkeit die benötigte Dienstgüte.

Sind keine Garantien möglich, sei es, weil der Basisdienst keine geeignete Funktionalitäten zur Überwachung oder Reservierung bietet oder die Anwendung keine Garantien braucht, so spricht man von *best effort*. In diesem Fall ist eine Untersuchung zur Feststellung der Netzhöchstlast oder die Analyse von Fehlerszenarien zur Verwaltung des Basisdienstes nicht nötig (da es keine Veranlassung dazu gibt). Die Ansprüche der Anwendung werden so gut es geht befriedigt, Zusagen werden nicht gegeben.

Ein Echtzeitsystem muss innerhalb einer gegebenen Zeitspanne, der *deadline*, auf Stimuli von außen reagieren. Ist das Ergebnis eines Stimulus nach Ablauf der *deadline* noch brauchbar, so spricht man von *weichen*, andernfalls von *festen* Echtzeitsystem. Ist das Ergebnis einer Überschreitung der *deadline* eine Katastrophe, so liegen *harte* Anforderungen an das System vor. In mobilen Ad-Hoc Netzen können mit den vorherigen Einschränkungen nur feste und weiche Echtzeitsysteme realisiert werden, keine harten [4].

QoS management policy: Mit den *Qos management policies* wird die Verwaltung der Dienstgüte geregelt. Darunter fällt die Behandlung von Dienstgüteverletzungen, die Anpassung oder Skalierung von Daten(strömen) an eine neue Ressourcensituation und die Beschreibung der Anzeige des aktuellen Dienstgütezustandes und eventueller Verletzungen (z.B. durch unterschiedliche Signale (Anwendung) oder verschiedene Farben (Benutzer)). Im Falle einer anpassungsfähigen Anwendung fällt auch die Festlegung spezieller Dienstgütesignale darunter, welche die aktuelle Parametern wie Jitter, Delay, Bandbreite, etc. übertragen.

cost of service: Festlegung des „Preises“, den der Benutzer bereit ist zu zahlen, um eine bestimmte Dienstgüte zu erlangen. Falls in der Spezifikation der Dienstgüte hierzu keine Angaben gemacht werden, gibt es für den Benutzer/die Dienstgüteanwendung keinen Grund, etwas anderes als das Maximum zu wählen. Bei einem Audio/Video-System ist es denkbar, dass der Benutzer bereit ist, auf den Audiodatenstrom zu verzichten, um den Videostrom mit der gewünschten Qualität bei einer Überlastung des Netzes zu empfangen. Vor allem in mobilen Ad-Hoc Netzwerken ist dieser Punkt besonders wichtig, da man es hier oft mit einer ständig wechselnden Topologie und somit auch mit einer ständig wechselnden Verbindungsqualität zu tun hat.

2.3.2 Dienstgüte-Bereitstellung

In diesem Kapitel werden die Maßnahmen beschrieben, die zur Umsetzung der Spezifikation in eine Ressourcenbelegung/-verteilung führen. In der Ressourcenbelegung/-verteilung ist Kommunikation mit der vorher vereinbarten Dienstgüte möglich. Diese Maßnahmen fasst man unter dem Begriff *Dienstgütebereitstellung* zusammen. Folgende Punkte spielen hierbei eine wichtige Rolle:

mapping: Unter *mapping* versteht man die (automatische) Umsetzung von Dienstgütespezifikationen zwischen benachbarten Schichten des Systems. Spezifikationen auf einem hohen Level sind meistens in der Sprache des Benutzers verfasst und dadurch auf niedrigere Schichten nicht anwendbar. Je weiter man sich der Basistechnologie nähert, desto konkreter werden die Angaben. Bei der Videoübertragung wird meist eine *flüssige* Wiedergabe gefordert. Tiefere Schichten können jedoch mit dem Begriff *flüssig* nichts anfangen. Also findet eine Übersetzung von *flüssig* in *Bilder pro Sekunde* statt. Diese Einheit kann weiter in *kBit pro Sekunde* und in *Rahmen pro Sekunde* verfeinert und somit für die Basistechnologie „verständlich“ gemacht werden.

admission testing: Der *Zugangstest (admission test)* vergleicht die benötigten (minimalen) Ressourcen einer Anforderung mit den aktuell verfügbaren, reserviert sie mit Hilfe eines Reservierungsprotokolls, gibt sie wieder frei und passt bestehende Reservierungen dynamisch an neue Ressourcensituationen an. Der Zugangstest ist von der Art und der Bedienstrategie der Ressource abhängig. Die Bedienstrategie und die Verfügbarkeit bestimmen, ob eine gegebene Gesamtanforderung bedienbar ist. Ein Schedulability-Test prüft dies und kann somit Bestandteil des Zugangstestes sein.

ressource reservation: Das *Ressourcenreservierungsprotokoll (ressource reservation protocol)* regelt die Ende-zu-Ende Reservierung von Ressourcen anhand der bekannten Verbindun-

gen zwischen den Knoten. Die Fixierung und Belegung des Leitweges erfolgt gemäß der Dienstgütespezifikation. Die Schwierigkeit dabei besteht aus der exakten Koordination der räumlich verteilten Einzelreservierungen und Freigaben von Übertragungsressourcen (Kapazität, Puffer) zwischen benachbarten Knoten.

In mobilen Ad-Hoc Netzen ist die globale Ressourcensituation im allgemeinen nicht „vor Ort", also in den jeweilig beteiligten Knoten, bekannt, da sie sich ständig ändert und ein Aktualisieren aller Knoten zu aufwändig wäre. Daher kann eine Reservierung (vorläufig) nur auf die bekannten lokalen Verbindungen optimiert werden. Als Resultat davon kann es vorkommen, dass das Protokoll zu viele Ressourcen reserviert. Aufgrund der bekannten Ende-zu-Ende Reservierung der gewählten Route können überschüssige Ressourcen jedoch wieder freigegeben werden. Dazu ist eine Unterstützung eines Ressource-Routing-Protokolls notwendig. Unser bestehendes Routing-Protokoll wird momentan dahingehend optimiert, ist aber zum jetzigen Zeitpunkt noch nicht einsatzbereit. Momentan verfügbare Protokolle liefern keine Rückmeldung über die Qualität der vorhandenen Ende-zu-Ende Verbindung.

Betrachtet man die Verbindungsdynamik mobiler Netze, so kommen noch weitere Aufgabe eines Ressourcenreservierungsprotokolls dazu. Die Verwaltung und Neureservierung von Verbindungen, die sich unter Umständen ständig ändern können, ist nicht trivial und erfordert einen hohen Aufwand. Auch die Unterstützung von Multicast-Verbindungen und die Kopplung mehrerer (heterogener) Netze sind nicht einfach zu lösende Probleme, da z.B. bei Multicast mehrere Einzelverbindungen gleichzeitig hinsichtlich Dienstgüte untersucht und geeignet reserviert werden müssen.

2.3.3 Dienstgüte-Kontrolle

Unter Dienstgütekontrolle fasst man alle kurzfristigen Maßnahmen zusammen, die Kommunikation in vereinbarter Dienstgüte sicherstellen. Im Einzelnen besteht die Kontrolle von Dienstgüte aus:

traffic (flow) shapping: Die Regulierung/Anpassung von Datenströmen gemäß eines vom Benutzer festgelegten Verkehrskontraktes (Datenrate, Verzögerung, etc.), also die Anpassung des tatsächlichen Verkehrsmusters an das vereinbarte, nennt man *Verkehrsformung (traffic/flow shapping)*.

traffic (flow) scheduling: Unter *Verkehrsplanung (traffic/flow scheduling)* versteht man die Überwachung und Festlegung der Weiterleitung einzelner Datenpakete und Datenrahmen. Das ist vor allem dann sinnvoll, wenn die Sendereihenfolge von Paketen geändert werden muss, um Timing Constraints einzuhalten.

traffic (flow) control: Die *Verkehrsüberwachung (traffic/flow control)* steuert oder regelt den Datenfluss. Die Steuerung erfolgt dadurch, dass anhand der allokierten Ressourcen Daten gesendet werden, die Regelung erfordert eine Anpassung des Datenstroms an eventuelle Änderungen der benötigten Ressource. Es findest also ein Abgleich des tatsächlichen Datenstroms mit dem Vereinbarten statt (im Gegensatz zur Verkehrsformung, bei der eine Anpassung stattfindet). Der Dienstanbieter wird überwacht, ob dieser seine reservierte (versprochene) Dienstgüte auch wirklich liefern und der Dienstnutzer, ob dieser seine zugeteilten Ressourcen nicht überschreitet.

flow synchronisation [opt]: Ein wichtiger Punkt der Dienstgütekontrolle ist die *Flusssynchronisation (flow synchronisation)*. Sie koordiniert zeitlich exakt die verschiedenen (parallelen) Datenströme. Sind keine solchen Datenströme vorhanden, so ist dieser Vorgang obsolet.

2.3.4 Dienstgüte-Management

Unter Dienstgütemanagement fasst man alle längerfristigen Maßnahmen zusammen, um die Kommunikation in vereinbarter Dienstgüte zu ermöglichen und zu überwachen. Das Management umfasst folgende Punkte:

Dienstgüteerfassung: Die Aufgabe des *Dienstgüteerfassung* (*monitoring)* ist die Überwachung und Erfassung der aktuellen Verfügbarkeit von Ressourcen und Dienstgüteparametern. Der momentane Zustand (Durchsatz, Delay, Jitter, Pufferauslastung) der jeweiligen Ende-zu-Ende Verbindung und des Netzes bildet die Grundlage für alle verkehrskontrollierenden Funktionen. So können durch ständiges Überwachen Verbindungsausfälle oder andere Fehler rechtzeitig erkannt und darauf angemessen reagiert werden.

Wartung: Die *Dienstgütewartung (maintenance)* vergleicht die verfügbare Dienstgüte mit dem erwarteten Ergebnis und führt gegebenenfalls eine neue Abstimmung (Tuning) der vorhandenen Ressourcen durch, um die spezifizierte Dienstgüte aufrechtzuerhalten oder wieder herzustellen. Die Grundlage hierzu bildet der durch Dienstgüteerfassung aktuelle Netzzustand. So können z.B. Verzögerungen durch einen Scheduler oder Durchsatz durch eine Flusskontrolle verändert werden.

Signalisierung [opt]: Die *Dienstgütesignalisierung (signalling)* legt die Art und Weise fest, wie die betreffenden Dienstgüteparameter überwacht und der Benutzer über die erwartete Leistung informiert wird. Es kann festgelegt werden, in welchem Intervall eine Erfassung stattfindet und welche Parameter genau zu überwachen sind. Es müssen hier ferner geeignete Signale definiert werden, um den Benutzer über die aktuelle Lage zu informieren.

Herabsetzung: Falls es nicht möglich ist, die geforderte Dienstgüte zu befriedigen und auch die Dienstgütewartung keine Alternative mehr hat, sie wieder herzustellen, wird dieses dem Benutzer bzw. der Anwendung mitgeteilt und durch die *Dienstgüteherabsetzung (degradation)* die Möglichkeit gegeben, sich den vorhandenen Ressourcen anzupassen oder zu einer anderen Dienstgütestufe zu wechseln. Die Anpassung kann z.B. durch eine kontinuierlichen Reduzierung des Durchsatzes erfolgen (soweit möglich). Ein Wechsel zu einer anderen Stufe kann (oder sollte) nur erfolgen, falls für diese Stufe ausreichend Ressourcen zur Verfügung stehen. Eine Neuverhandlung ist auch denkbar, falls eine Herabsetzung aus nicht in Frage kommt. Jedoch besteht in diesem Fall besteht die Gefahr, dass keine neue Reservierung getätigt werden kann, falls keine weiteren Ressourcen auf anderen Leitwegen frei sind oder werden.

Skalierung: Die *Dienstgüteskalierung (scaling)* gliedert sich in zwei Teile, der D*ienstgütefilterung (filtering)* und der *Dienstgüteanpassung (adaption)*. Unter *Filterung* versteht man die Manipulation von Datenströmen oder Paketfolgen während der Übertragung. So können irrelevante Daten ausgeblendet (nicht gesendet) werden, falls es zeitweise keine Geräte im Netz gibt, die diese Daten brauchen oder man zu bestimmten Zeiten nur eine Teilmenge davon braucht. Hat man z.B. eine Videoübertragung, die Farb- und S/W-Bilder liefert, so kann man bei Multicast die verschiedenen Endgeräte separat versorgen und so die Last minimieren. Eine senderseitige Herausfilterung von weniger wichtigen Daten bei „Netzhochlast“ ist auch denkbar.

Im Gegensatz zu den Dienstgütekontrollmaßnahmen bezieht sich die obige *Dienstgüteanpassung* nicht auf eine Regulierung des vorhandenen Datenstromes, sondern auf eine senderseitige Anpassung des Datenstromes. Darunter fällt z.B. die Anpassung der Bildgröße und der Framerate einer Kamera, und auf eine veränderte Ressourcensituation zu reagieren oder die Vergrößerung von Abtastintervallen von Sensoren, um die Datenrate zu senken. Die Anpassung findet auf Anwendungsseite statt und nicht auf der Netzwerkseite.

2.4 SDL Design-Patterns

Ein *SDL-Design Pattern* beschreibt ein wiederkehrendes Designproblem und die dazugehörige generische Lösung. Mehr ist hierzu unter [2] oder [14] zu finden. Im Rahmen dieser Arbeit wurden einige neue Muster erstellt.

2.5 Mikroprotokolle

Ein *Mikroprotokoll* ist eine abgeschlossene Komponente, die eine Protokollfunktionalität kapselt. Beispiele hierfür sind z.B. der Verbindungsauf- und Abbau oder eine Flusskontrolle. Durch ihre Abgeschlossenheit kann man Mikroprotokolle sehr gut wiederverwenden. Dadurch lassen sich schnell maßgeschneiderte Kommunikationssysteme entwerfen, indem man auf bekannte und bewährte Komponenten zurückgreift und diese komponiert und integriert. Arbeiten mehrere Mikroprotokolle sinnvoll miteinander oder nicht separat, so kann man diese Mikroprotokolle zu einem *Makroprotokoll* zusammenfügen. Ein Makroprotokoll ist z.B. eine Dienstgüte-Mapping-Scaling-Komponente, die sich aus den komponierten Mikroprotokollen *Mapping* und *Scaling* zusammensetzt.

Da Mikroprotokolle als Blackbox gesehen werden, ist die Definition der Schnittstelle wichtig. Über diese Schnittstellen läuft die ganze Kommunikation mit der Umgebung und mit anderen Protokollen ab. Eine eindeutige Beschreibung dieser Schnittstelle und der dazugehörigen Signale ist essentiell, da nur so eine sinnvolle Einbettung in den umgebenden Kontext möglich ist. Durch die externe Sicht eines Mikroprotokolls als Blackbox (Information-Hiding, Geheimnisprinzip) lassen sich in einem fertig komponierten und integrierten System leicht Funktionalitäten austauschen. So kann z.B. durch den Austausch eines einzigen Mikroprotokoll ein anderes Routingverfahren verwendet werden, ohne dass das Kommunikationsprotokoll neu entwickelt oder angepasst werden muss.

Als Richtlinie für die Entwicklung von Mikroprotokollen gilt, möglichst wenig Funktionalität in eine Komponente zu integrieren. Diese kleinen Protokolle lassen sich so flexibler zu mächtigeren Makroprotokollen zusammenfügen, die dann wiederum leicht in eine Middleware integriert werden können.

2.5.1 Beschreibung von Mikroprotokollen

Ähnlich wie Design-Patterns fasst man auch Mikroprotokolle in einer Bibliothek (Kapitel 2.5.2) zusammen. Der Unterschied besteht jedoch darin, dass keine generische Lösung für wiederkehrende Designprobleme, sondern eine spezifische Implementierung einer bestimmten, oft benötigten Funktionalität als Blackbox mit wohldefinierten Schnittstellen beschrieben wird. Damit der Entwickler leicht ein geeignetes Mikroprotokoll auswählen kann, ist es notwendig, eine einheitliche Beschreibung zu finden und die Protokolle in einer Bibliothek zu ordnen und zu verwahren. Je nach verwendeter Spezifizierungssprache hat die Mikroprotokollbibliothek und die Beschreibung ein anderes Aussehen. In unserem Fall wird SDL verwendet und somit finden sich auch SDL-spezifische Aspekte in der Beschreibung. Im Folgenden wird ein Beschreibungsframework(-template) für Mikroprotokolle vorgestellt, welches im Rahmen dieser Diplomarbeit entwickelt wurde.

Die Elemente dieser Beschreibung unterstützen die Selektion (*Name, Intent, Interfacing Behaviour*) und die Komposition (*Interface Definition, Structure, Environment Assumptions, Composition*) von Mikroprotokollen. Mikroprotokolle unterstützen auch die Durchführung von (Design-) Reviews durch speziell auf das Protokoll angepasste *Checklisten*. Im folgenden Template werden die einzelnen Elemente der Mikroprotokollbeschreibung erläutert:

Name:

Jedes Mikroprotokoll wird durch einen eindeutigen Namen identifiziert. Er sollte so gewählt werden, dass Rückschlüsse auf die Verwendung bzw. auf die gekapselte Funktionalität möglich sind.

Version:

Die Versionsnummer des Protokolls. Nach jeder Iteration sollte diese Nummer entsprechend erhöht werden.

Intent (Kurzform):

Eine informelle, textuelle Beschreibung des Mikroprotokolls mit einem ersten Hinweis auf die Funktionalität/Verwendung bzw. den geeigneten Kontext. Dieses Beschreibungselement bildet die Grundlage des Auswahlprozesses.

Structure (Struktur):

Die *Struktur* beschreibt die Abhängigkeiten zwischen den Teilkomponenten eines Mikroprotokolls (z.B. Schnittstellendefinitionen, Datentyp-Packages). Sie abstrahiert komplett von der konkreten Implementierung und gibt lediglich einen Überblick über den Umfang des Protokolls (benötigte Packages, Datentypen, etc.).

Abbildung 2-2 zeigt die Struktur des Mikroprotokolls *MicroProto*. Es benötigt drei Packages *UsedPackage1..3* und die ASN-Module *Modules1ASN1* und *Modules2ASN1*.

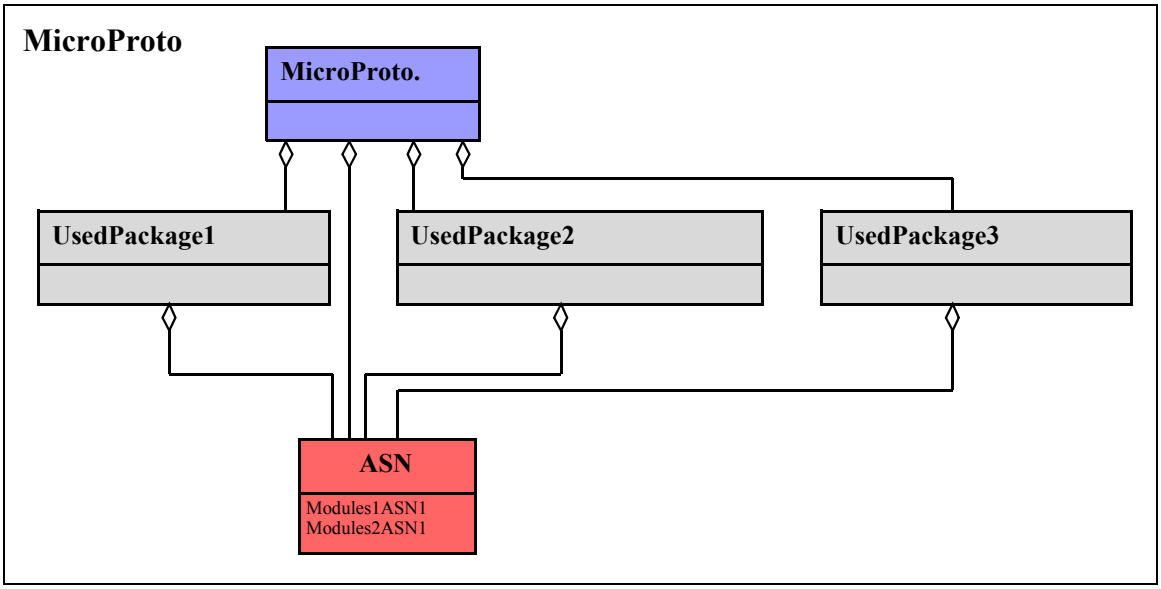


Abb. 2-2: Struktur des Mikroprotokolls *„DelayMapping“*

Interface Definition (Schnittstellenbeschreibung):

Die *Interface Definition* ist eine der wichtigsten Beschreibungselemente. Sie beschreibt die Schnittstellen in einer realisierungsunabhängigen Form. Teilfunktionalitäten des Mikroprotokolls (wie z.B. eine Initialisierung oder eine Parameterabfrage) werden durch abstrakte Operationen beschrieben, die in einzelne Aktionen verfeinert werden können. Zur graphi-

schen Darstellung werden verschiedene Symbole eingeführt, um unterschiedliche Arten von Operationen beschreiben zu können.

Tabelle 2-1: Interfacing Symbols

Symbol	Name	Beschreibung
#>	value	Operation liefert eine neue Information für das Mikroprotokoll.
?>	challengeResponse	Operation liefert nach einem bestimmten Input (Trigger) eine Information oder fordert eine Information an.
@>	event	Operation beschreibt das Auftreten eines (unerwarteten) Ereignisses

Folgende Abbildung (Abb. 2-3) zeigt den schematischen Aufbau der graphischen Darstellung der Schnittstellenbeschreibung. Operationen sind an Schnittstellen gebunden. An Schnittstellen wiederum können anderen Komponenten angekoppelt werden, die dann genau diese Operationen nutzen oder benötigen. Jede Operation wird durch ein *Interfacing Symbol* beschrieben, welches die grundlegende Natur der Operation wiedergibt. Neben den normalen Operationen gibt es auch die zu verfeinernden Operationen. Diese werden bei den *semi-generischen* Mikroprotokollen (siehe Kap. 2.5.2) dazu genutzt, die Operationen zu beschreiben, die für eine Einbettung an den Kontext angepasst werden müssen. Eine detaillierte Beschreibung der Operationen erfolgt informell (z.B. tabellarisch).

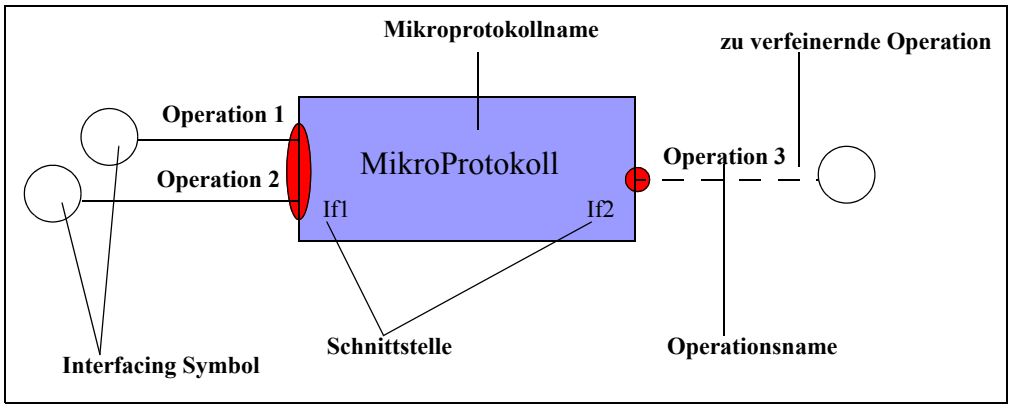


Abb. 2-3: Graphische Beschreibung der Schnittstellen

Interface Behaviour (Schnittstellenverhalten):

Das *Interface Behaviour* beschreibt das Verhalten des Mikroprotokolls, liefert also eine Blackbox-Sicht, ohne auf die genaue Realisierung einzugehen. In Form von sogenannten *Interface Action Charts (IAC)* werden die Abhängigkeiten zwischen den verschiedenen Operationen bzw. Aktionen der Schnittstellen aufgezeigt. Die IACs orientieren sich stark an den Message Sequence Charts, jedoch haben die einzelnen Symbole eine etwas andere Bedeutung (Abbildung 2-4).

Jedes IAC wird durch einen *Namen* beschrieben, der von einer *Operation* abgeleitet ist. *Operationen* werden in *Aktionen* unterteilt. Die einzelnen *Aktionen* einer *Operation* werden durch einen eindeutigen Bezeichner und eine Liste von Parametern, den *Informationen*

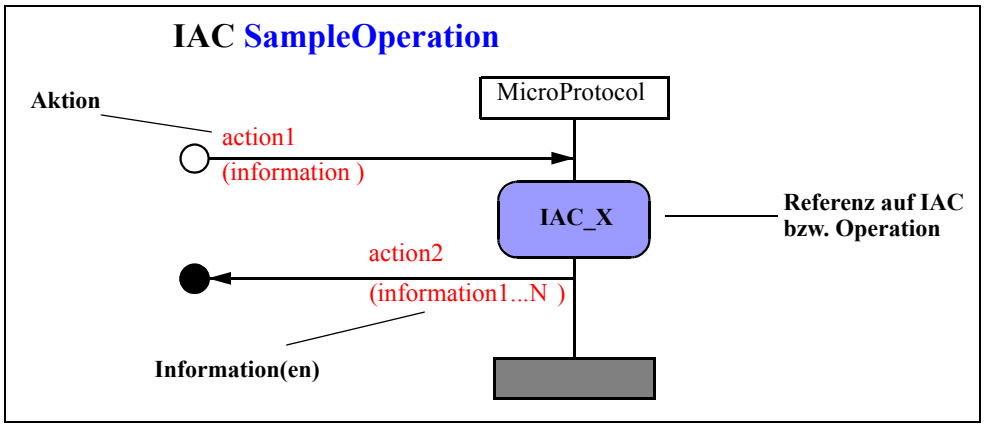


Abb. 2-4: Interface Action Chart

beschrieben. Sind bei einer Aktion mehrere Reaktionen möglich, so kann ein IAC um eine unbestimmte Anzahl von Diagrammen erweitert werden. Mit Hilfe der Abfolge dieser Aktionen und dem Zusammenspiel mit anderen IACs, auf die mit Hilfe von Referenzen verwiesen wird, lassen sich leicht Interaktionsmuster bestimmen, die das Verhalten des Mikroprotokolls im umgebenden Kontext widerspiegeln. Die Einführung von Assertions oder Bedingungen ist geplant.

In Abbildung 2-5 ist eine Beschreibung des Schnittstellenverhaltens von *DelayMapping* zu sehen. Es gibt drei Operationen auf dem Mikroprotokoll: *Init, SetQosClass* und *Mapping*. *Init* legt eine neue Dienstgüteklasse mit der dazugehörigen maximalen und optimalen Verzögerung an, *SetQosClass* bildet eine Dienstgüteklasse in einem *Mapping* auf Delays ab (falls die Klasse vorhanden ist) und die Aktion *Mapping* liefert als Information genau diese Delays.

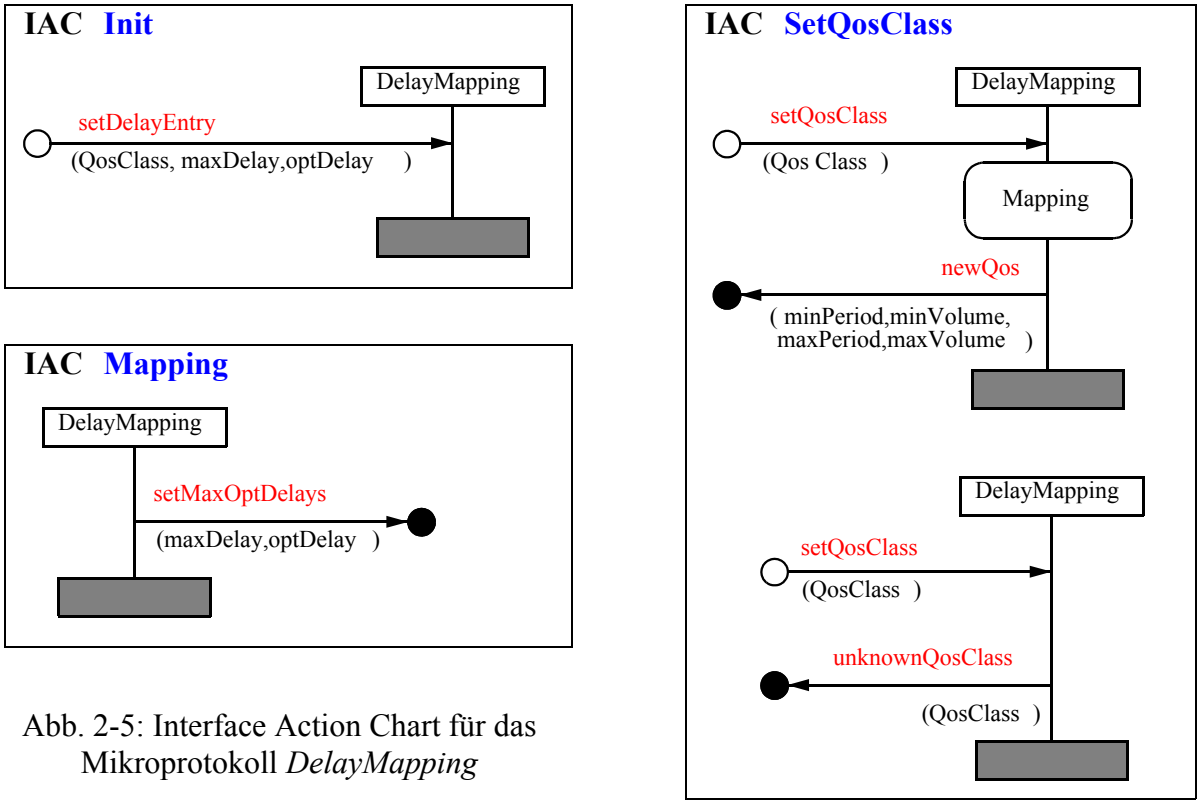


Abb. 2-5: Interface Action Chart für das Mikroprotokoll *DelayMapping*

(SDL-)Representation:

Die Repräsentation eines Mikroprotokolls ist stark von der verwendeten Spezifizierungs-/ Implementierungssprache abhängig und gliedert sich in mehrere Teile. Im Rahmen dieser Diplomarbeit werden die Mikroprotokolle ausschließlich in SDL spezifiziert und somit sind die folgenden Untergruppierungen stark SDL-abhängig.

- **Structure (konkrete Struktur):** Struktur des Mikroprotokolls. Überblick über die Klassen- bzw. Packagehierarchie und die Abhängigkeiten untereinander. Dieser Abschnitt ist die konkrete Realisierung der abstrakten Struktur auf oberster Beschreibungsebene.

- **Data Types and Structures (Datentypen und Datenstrukturen):** Auflistung der in dem Mikroprotokoll neu definierten Datentypen und Strukturen. Die Definition findet in ASN.1 statt.

- **Signal Definition (Signalbeschreibung):** Beschreibung der (SDL-)Signale des Protokolls und deren Parameter. Das kann z.B. in Form einer Tabelle erfolgen.

- **(SDL-)Specification/excerpt (Auszug aus der Spezifikation):** Auszug aus der Spezifikation des Mikroprotokolls, der Aufschluss über die gewählte Realisierung gibt und somit zur Integration oder Komposition benötigt wird (z.B. Realisierung als Prozess oder Service). Falls es sich um ein semi-generisches Mikroprotokoll handelt, müssen hier alle virtuellen Transitionen bzw. Fragmente aufgelistet werden, also diejenigen die verfeinert werden können oder müssen. Erweiterungen wie zusätzlich benötigte Signale sollten informell beschrieben werden.

- **Refinement (Verfeinerung):** Dieses Element der Beschreibung dient dazu, semi-generische Mikroprotokolle korrekt zu spezialisieren. Dazu enthalten die semi-generischen Mikroprotokolle bestimmte Spezialisierungselemente bzw. -beschreibungen (Abb. 2-6).

  Für jedes dieser Elemente muss in dem spezialisierten Mikroprotokoll ein entsprechendes Element angelegt werden. Die nächste Abbildung (Abb. 2-7) zeigt eine konkrete Verfeinerung am Beispiel von *DelayMapping*. Das generalisierte Mikroprotokoll liefert nach dem Setzen einer Dienstgüteklasse eine minimale und eine optimale Dienstgüteanforderung (jeweils in Form eines Periode-Datenvolumen-Paares). Man verwendet dieses Mikroprotokoll dazu, (wichtige) Daten periodisch zu übertragen (Steuerdaten eines mobilen Gerätes, Sensorwerte einer Produktionslinie,...). Je nach aktueller Situation kann die Verzögerung zwischen Vorhandensein der Daten beim Sender und dem Empfang der Daten variieren, sollte aber immer innerhalb eines bestimmten Intervalls liegen (dieses Intervall wiederum kann je nach Situation variieren). Die Größe der Daten ist im Allgemeinen jedoch immer gleich. Daher wird in dem Beispiel die Starttransition neu definiert, um die Größe eines Datenpackets anzugeben.

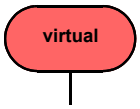
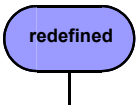
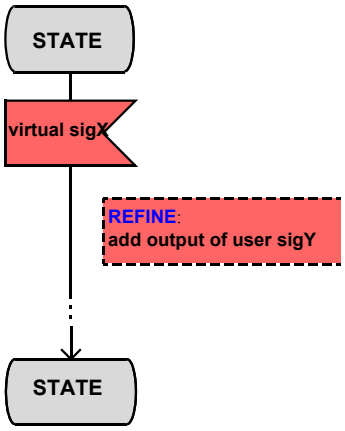
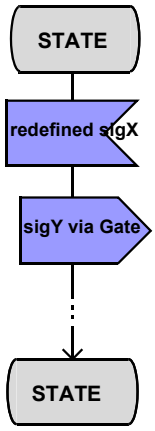
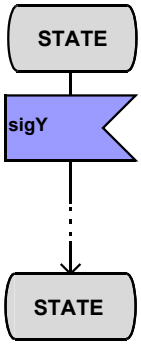
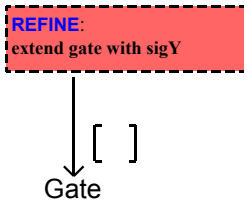
vor der Spezialisierung	nach der Spezialisierung	Anmerkung
		Spezialisierung des Startzustandes. Wird dazu genutzt, um kontextbezogene Variablen/Parameter zu initialisieren
		Verfeinerung einer vorhandenen Transition. Da das Neudefinieren von Signalen nicht möglich ist (mangels Toolunterstützung), kann eine evtl. Anpassung eines Mikroprotokolls nur durch Vererbung und Neudefinierung von Transitionen erfolgen. In diesem Fall muss eine Ausgabe des kontextspezifischen Signals *sigY* erfolgen.
 		Hinzufügen einer neuen Transition im Zustand *STATE*. Es ist darauf zu achten, dass jedes Auftreten von *sigY* in der Beschreibung durch das gleiche Benutzersignal ersetzt wird.
 		Erweiterung eines Gates um ein neues Signal.

Abb. 2-6: Elemente der Spezialisierung

Eine Alternative zum Überladen der Starttransition ist die Parametrisierung der Prozessinstanz. Sie wird aber von den uns verfügbaren Werkzeugen nicht oder nur eingeschränkt unterstützt. So ist es z.B. mit Telelogic Tau nicht möglich, statisch erzeugte Prozessinstanzen zu parametrisieren. Hat man jedoch eine dynamische Struktur, so kann man sich die Spezialisierung der Starttransition sparen und direkt das *Supermikroprotokoll* (analog zu einer Superklasse) verwenden. Es ist denkbar, für beide Fälle semi-generische Protokolle mit in die Bibliothek aufzunehmen, um sowohl statische als auch dynamische Prozesserzeugung zu unterstützen und für den Entwickler möglichst komfortabel anzubieten. Mehr zu semi-generischen Mikroprotokollen ist in Kapitel 2.5.2 zu finden.

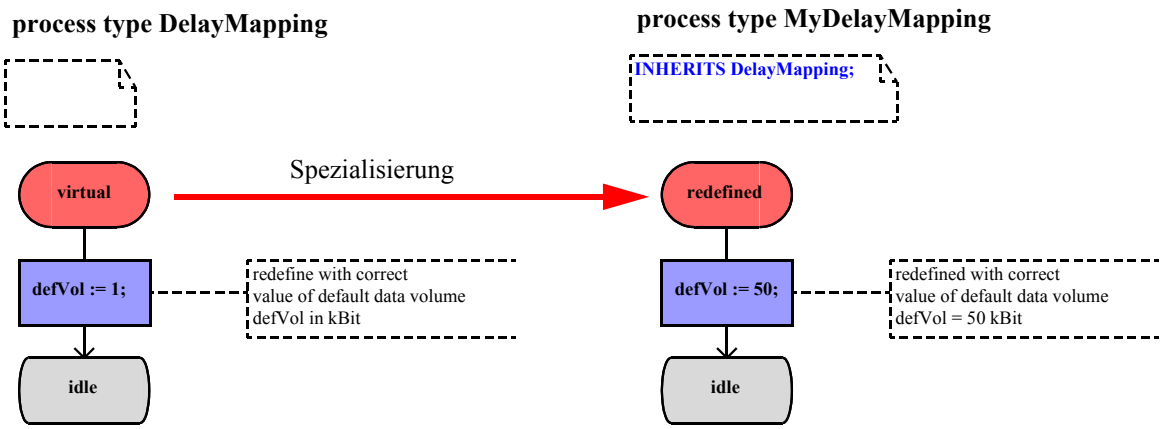


Abb. 2-7: Spezialisierung von *DelayMapping*

- **Environment Assumptions (Forderungen am den Kontext):** Wichtige Annahmen und Forderungen an den umgebenden Kontext (Umgebung oder andere Mikroprotokolle) damit das Mikroprotokoll korrekt funktioniert. Diese Annahmen/Forderungen sollten unabhängig von der gewählten Spezifizierung/Implementierung (hier SDL) sein. Hier können z.B. andere Mikroprotokolle oder Gattungen von Mikroprotokollen angegeben werden, mit denen ein bestimmtes Mikroprotokoll interagieren muss, damit es korrekt in den Kontext eingebettet ist. Dieser Punkt ist vor allem dann wichtig, falls eine *challenge-response* Operation auf eine Aktion der Umgebung wartet. Verhält sich die Umgebung nicht korrekt, so ist mit einer Verklemmung zu rechnen.

- **Composition (Komposition):** Beschreibt die Vorgehensweise bei der Integration und Komposition von Mikroprotokollen. Das Vorgehen ist stark von der Struktur des Mikroprotokolls abhängig. Mehr dazu ist in Kap. 2.5.3 zu finden.

Cooperative Usage (Kooperative Verwendung):

Liste von weiteren Mikroprotokollen, die sich sinnvoll mit dem beschriebenen zusammen anwenden lassen.

Known Uses (Anwendungen):

Namen von Systemen, in die dieses Mikroprotokoll erfolgreich integriert wurde. Dadurch kann eine Evaluierung des Mikroprotokolls nachgewiesen werden.

Checklist (Prüfliste):

Liste von Aussagen, die gelten sollten, damit dieses Mikroprotokoll korrekt funktioniert. Die Checkliste sollte die *Environment Assumptions* beinhalten und kann somit bei der Komposition und Integration als Vorgehensweise angesehen werden, um Fehler zu vermeiden. Häufige Fehler, die bei der Anwendung des Protokolls gemacht wurden, sollten auch aufgelistet werden, damit diese vermieden werden können. Eine Checkliste unterstützt auch einen späteren Design-Review.

2.5.2 Die Mikroprotokollbibliothek

Eine Mikroprotokollbibliothek ist eine Sammlung von Mikroprotokollen und deren Beschreibungen. In diesem Kapitel wird die im Rahmen dieser Arbeit neu entwickelte Bibliothek vorgestellt.

Die Mikroprotokollbibliothek gliedert sich in zwei Teile. Einer vollständigen Spezifikation der Protokolle in SDL und einer Dokumentation der Bibliothek inklusive der einzelnen Protokolle. Um die Navigation innerhalb der Bibliothek zu erleichtern, wurde eine baumartige Verzeichnisstruktur gewählt. Folgende Abbildung (Abb. 2-8) zeigt die Struktur der Bibliothek.

Im Wurzelverzeichnis *Bibliothek* liegt eine vollständige Beschreibung aller Patterns in der Mikroprotokollbliothek und das Telelogic Tau Systemfile, da die Bibliothek mit diesem Werkzeug spezifiziert wurde. Das Verzeichnis *DataTypes* enthält die benutzten Datentypen und Strukturen in Form von ASN.1 Modulen. Für jedes spezifizierte Modul ist ein eigenes Dokument anzulegen. Die verschiedenen ASN.1 Module sind inhaltlich voneinander verschieden, d.h. alle Datentypen, die z.B. für die Spezifizierung von Dienstgüte-Mikroprotokollen genutzt werden, liegen im gleichen ASN.1 Modul.

Das Verzeichnis *GlobalSignalPackages* enthält alle Signale, die von „globaler“ Bedeutung sind, in sinnvoll gruppierten SDL-Packages. Von globaler Bedeutung heißt, dass z.B. alle Mikroprotokolle, die eine Dienstgütefunktionalität bereit stellen, bestimmte (SDL-)Dienstgütesignale kennen müssen.

Mikroprotokolle werden im Rahmen dieser Diplomarbeit in drei Kategorien unterteilt. Die *spezifischen*, die *semi-generischen* und die *generischen* Mikroprotokolle. Diese Klassifizierung bezieht sich auf den Grad ihrer Wiederverwendbarkeit. *Spezifische* Mikroprotokolle sind auf eine bestimmte Klasse von Anwendungen maßgeschneidert und lassen sich für andere Klassen nur schwer wiederverwenden. *Semi-generische* beschreiben Funktionalitäten, die in jeder Dienstgüte-Middleware vorhanden sein können, aber dennoch angepasst werden müssen. Das geschieht durch Spezialisierung oder Parametrisierung. Als letzte Klasse sind die *generischen* Mikroprotokolle ohne Einschränkung oder Anpassung für jede Dienstgüte-Middleware wiederverwendbar. Sie beschreiben feste, anwendungsunabhängige Funktionalitäten. Von der Beschreibung sind die *generischen* und *spezifischen* identisch, da beide keine Spezialisierung vorsehen.

Für jede Klasse von Mikroprotokollen wird ein eigener Verzeichnisbaum angelegt. Dieser besteht aus dem Unterverzeichnis *Signals*, welches die Signalpackages enthält, die von den spezifizierten Protokollen benötigt werden, und aus den jeweiligen Mikroprotokoll Unterverzeichnissen (*MicroProtocol1..N*). In diesen sind neben der SDL-Spezifikation auch Unterverzeichnisse für Dokumente (*Docs*) und Diagramme (*Diagrams*) angelegt. Darin befinden sich die jeweiligen Mikroprotokollbeschreibungen und die verschiedenen Diagrammtypen zur Dokumentation (UML, Struktur, IAC).

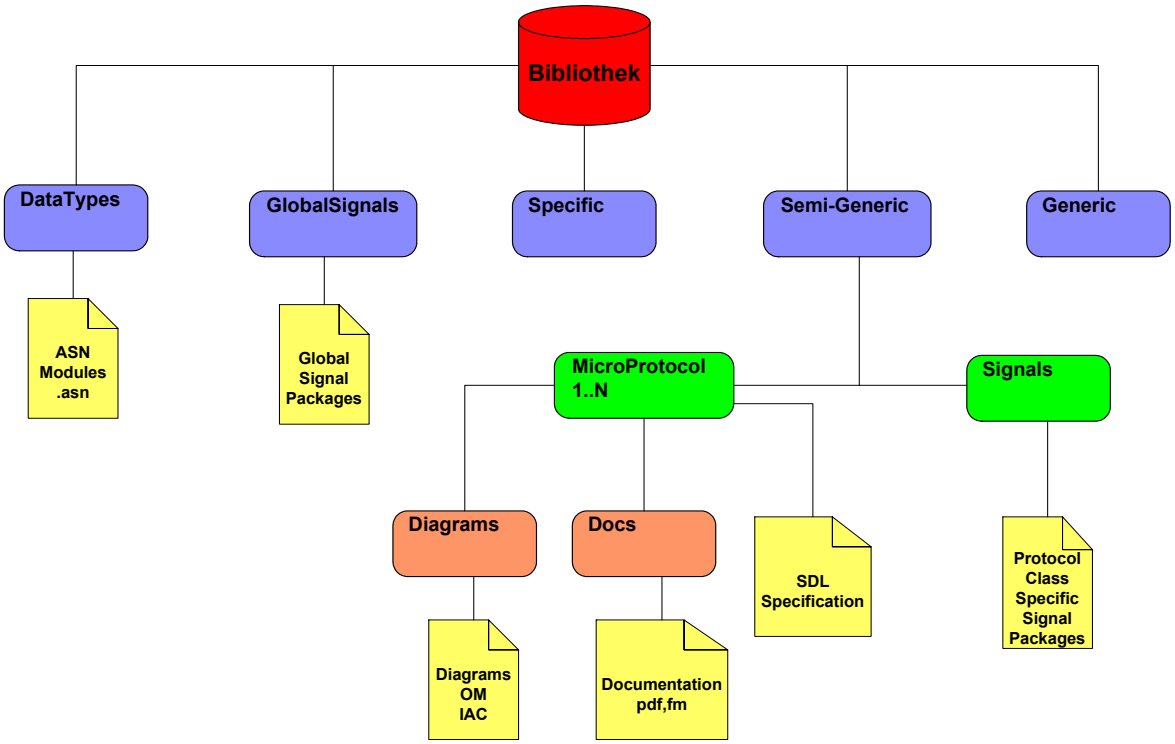


Abb. 2-8: Die Mikroprotokollbibliothek

Das SDL-System auf oberster Ebene gibt die Struktur der Bibliothek wieder.

2.5.3 Komposition bzw. Integration von Mikroprotokollen

Durch die Unterteilung der Mikroprotokollbibliothek in einen funktionellen Teil (Mikroprotokolle), einen Datenstruktur- (ASN.1 Module) und einen Signalteil (Packages mit den entsprechenden Signalen) ergeben sich bei der Integration und Komposition einige Besonderheiten. Nach der Selektion eines Protokolls der Bibliothek finden sich in dessen Beschreibung klare Angaben zu den verwendeten Packages. Diese sind nun einfach in das zu entwickelnde System einzubinden. Am Mikroprotokoll selbst muss nichts geändert werden, es muss lediglich sichergestellt sein, dass die benötigten Packages verfügbar sind. Die benötigten ASN.1 Module mit den Datentypen und Strukturen können einfach zu den bestehenden Modulen hinzugefügt werden, falls schon welche vorhanden sind. Ist dies nicht der Fall, kann man auch einfach das komplette ASN.1 Package der Bibliothek integrieren und die nicht benötigten Module löschen.

Wird ein weiteres Mikroprotokoll in das System integriert, muss an dieser Stelle untersucht werden, welche Packages der Mikroprotokollbibliothek bereits im System vorhanden sind. Diese sind um die noch Fehlenden zu erweitern.

2.5.4 SDL-Mikroprotokoll Entwicklungsprozess

Analog zum Entwicklungsprozess für SDL-Patterns wird hier ein Entwicklungsprozess für Mikroprotokolle vorgestellt, der in einen vorhandenen Entwicklungsprozess (z.B. aus [1]) eingebettet werden kann (Abbildung 2-9). Er ist mit dem in [8] vorgestellten Entwicklungsprozess verwandt, jedoch liegt der Fokus bei dem hier vorgestellten Prozess auf einer iterativen Entwicklung und einem abschließenden Unit-Test nach jedem Inkrement. Im Folgenden wird der Entwicklungsprozess kurz erläutert.

Ausgehend von den Kommunikations- und in diesem Fall auch Dienstgüteanforderungen wird

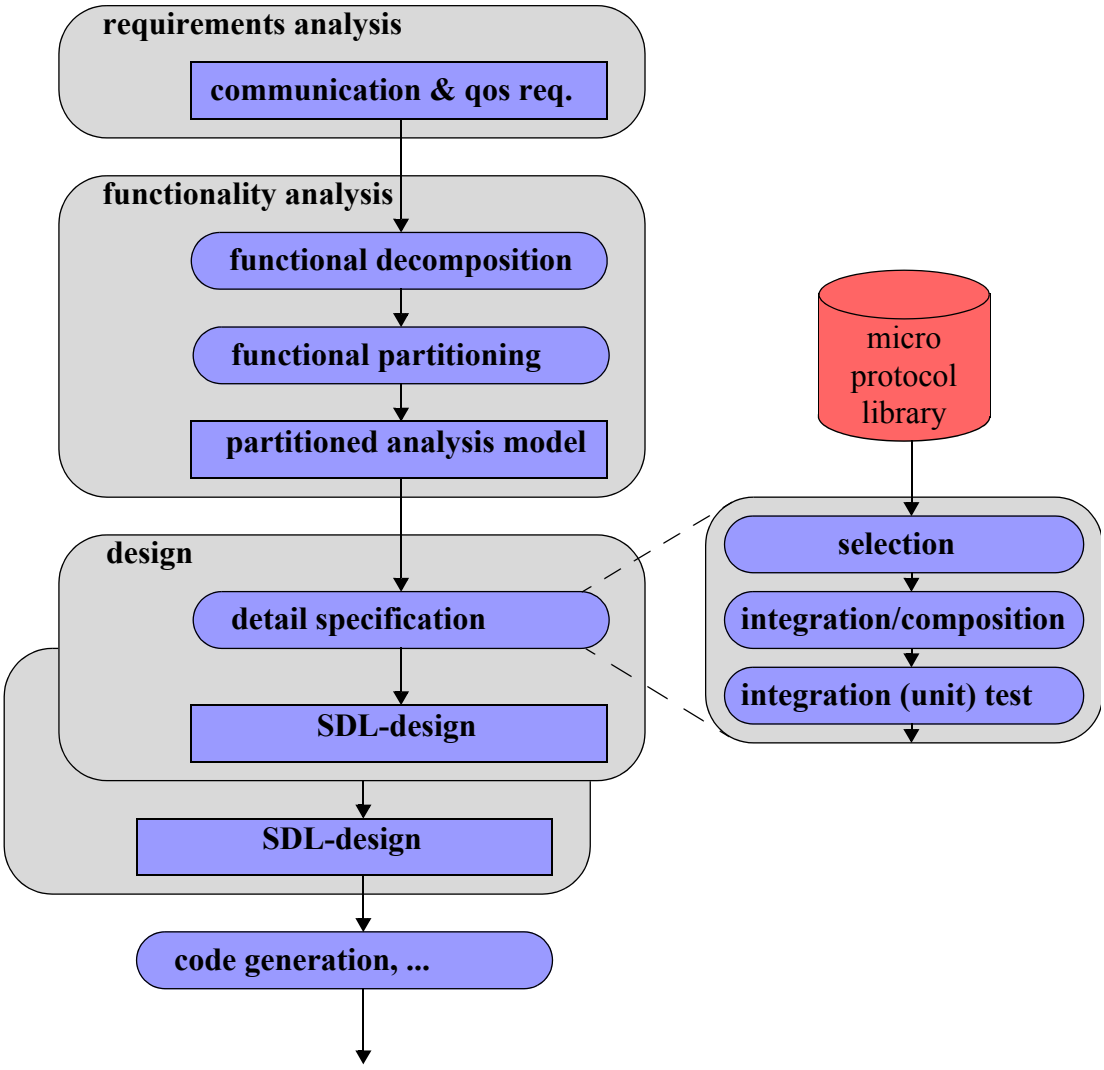


Abb. 2-9: Iterativer Mikropotokoll-basierter Entwichklungsprozess

ein partitioniertes Analysemodell erstellt. Dieses beinhaltet die weiter verfeinerten Benutzeranforderungen, partitioniert nach verwandten Funktionalitäten. So könnte eine Partition die allgemeinen Kommunikationsanforderungen enthalten, eine andere die anwendungsspezifischen Dienstgütefunktionalitäten (Mapping und Skalierung) oder die anwendungsunabhängigen (Reservierung, etc.).

Jede dieser Partitionen wird nun iterativ entwickelt. Anhand der erkannten Teilfunktionalitäten kann ein geeignetes Mikroprotokoll aus der Bibliothek ausgewählt und in das System integriert oder ein Pattern angewandt werden. Da es noch keine Technik gibt, die es uns erlaubt, ein konzeptionelles Design z.B. in Form eines Mikroprotokoll-Frameworks (z.B. [10]) zu testen, ist der letzte Schritt in dem Integrations- bzw. Kompositionsprozess ein Integrations/Unit-Test. Durch geeignete Stubs und Driver, die dem System bis zur nächsten Iteration hinzugefügt werden, können die nicht vorhandenen Komponenten simuliert werden.

Das Ergebnis dieses Entwicklungsprozesses ist eine fertige und getestete SDL-Spezifikation.

2.6 Zusammenfassung

In diesem Kapitel wurden die notwendigen Grundlagen zum Verständnis dieser Arbeit vorgestellt. Der Begriff der Dienstgüte wurde ausführlich erläutert und es wurden Techniken und Methoden vorgestellt, Dienstgüteanforderungen zu beschreiben und die zur Realisierung nötigen Funktionalitäten zu selektieren. Der zweite Teil dieses Kapitels beschäftigte sich mit Mikroprotokollen. Es wurde eine neue Klassifizierung für Mikroprotokolle eingeführt, die eine Unterteilung in spezifische, semi-generische und generische Protokolle vorsieht. Weiterhin wurde ein Beschreibungstemplate für Mikroprotokolle spezifiziert, welches Unterstützung für die Komposition und Selektion von Protokollen bereitstellt. Abschließend wurde eine Struktur einer Mikroprotokollbibliothek vorgestellt und eine Erweiterung zu einem bestehenden Entwicklungsprozess für Mikroprotokolle.

3 Applikation

In diesem Kapitel wird die verwendete Hardware und die Anwendung vorgestellt. Die Anforderungen, die die Anwendung an die Middleware stellt, wird auf Dienstgüte hin untersucht und es wird eine Dienstgütespezifikation nach den in Kapitel 2.3 vorgestellten Gesichtspunkten gegeben.

3.1 Das Luftschiff

Das verwendete Luftschiff ist ein FS40 von der Firma Evolution (www.evo.aero). Es ist 4 Meter lang und hat ein Volumen von 2,8 m³. Die Gondel wurde komplett umgestaltet, damit der Embedded-PC inkl. WLAN-Stick und Webcam (siehe Kapitel 2.2.1) darin Platz hat. Desweiteren wurden schwere Teile entfernt und durch leichtere ersetzt. Dadurch kann das Luftschiff trotz der zusätzlichen Komponenten ohne Einschränkung der Flugeigenschaften betrieben werden. Mit dem verwendeten Hochleistungsakku (144g, 11.1V, 2000mAh), der alle Komponenten ge-

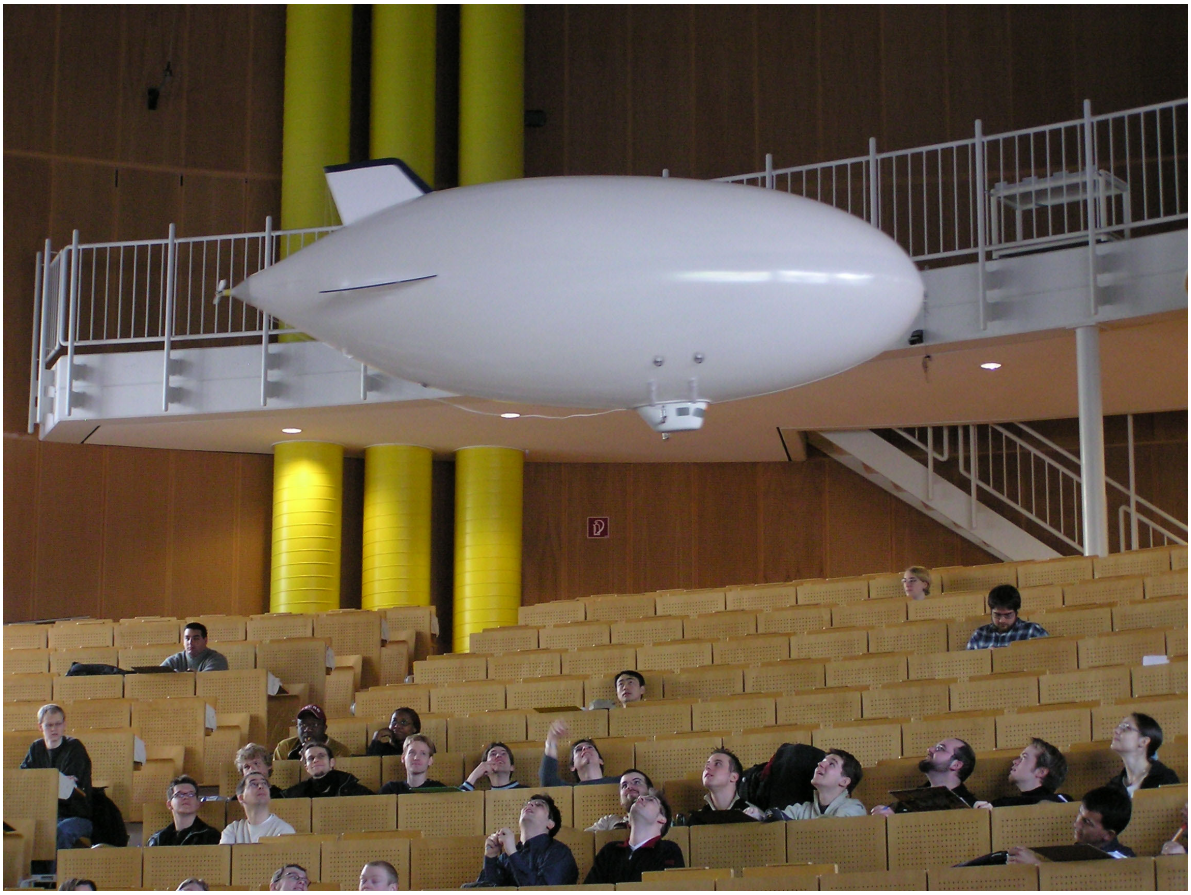


Abb. 3-1: Der Zeppelin bei einer Vorführung

meinsam versorgt, kann eine Flugzeit von 30-60 Minuten erreicht werden. Abbildung 3-1 zeigt den Zeppelin bei einer Demonstration.

3.2 Beschreibung der Anwendung

Die Anwendung besteht aus zwei Teilsystemen, einer Videoapplikation (linker Teil von Abbildung 3-2) und einer Steuerapplikation (rechter Teil von Abbildung 3-2). Die Steuerapplikation ermöglicht eine Steuerung des Zeppelins mit Hilfe eines Gamepads, die Videoapplikation überträgt die Bilder der Webcam und stellt diese auf Benutzerseite geeignet dar.

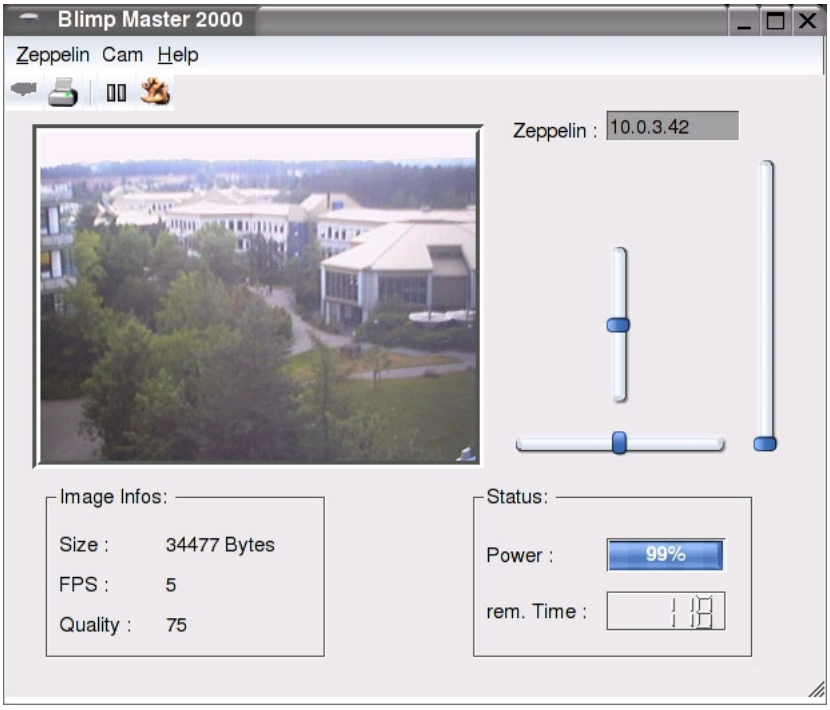


Abb. 3-2: Screenshot der GUI

3.3 Architektur der Anwendung

Aufgrund der Eigenschaften der beiden Teilanwendungen liegt eine klassische Client-Server-Architektur vor. Ein *Videoserver* im Luftschiff liefert in periodischen Abständen Bilder, die von einem oder mehreren *Videoclients* entgegengenommen werden. Der *Pilotserver*, die Steuerungskomponente des Luftschiffes, nimmt von einem *Pilotclient* Steuerungsignale entgegen und regelt mit den übermittelten Werten die Motoren (Heck und Auftrieb) und den Servo (horizontale Ausrichtung der Auftriebsmotoren) ein. Abbildung 3-3 gibt einen Überblick über die vorliegende Architektur und Struktur des *Servers* (Luftschiff) und der *Clients* (mobile Stationen).

Die Anwendungen sind verteilt und kommunizieren über WLAN (IEEE 802.11b [15]) miteinander. Der Server (Luftschiff) ist in zwei Teile unterteilt, einem *Videoserver* und einem *Pilot(Steuerungs)server*. Sie kommunizieren über eine *Middleware* mit den entsprechenden Hardwarekomponenten (*Cam*, *Servo/Motoren*) und mit ihren entsprechenden Clients (über die Basistechnologie *WLAN*). Die Motoren sorgen für den benötigten Auftrieb und drehen den Zeppelin um die vertikale Achse, der Servo steuert die horizontale Achse und ist somit für den Vortrieb verantwortlich.

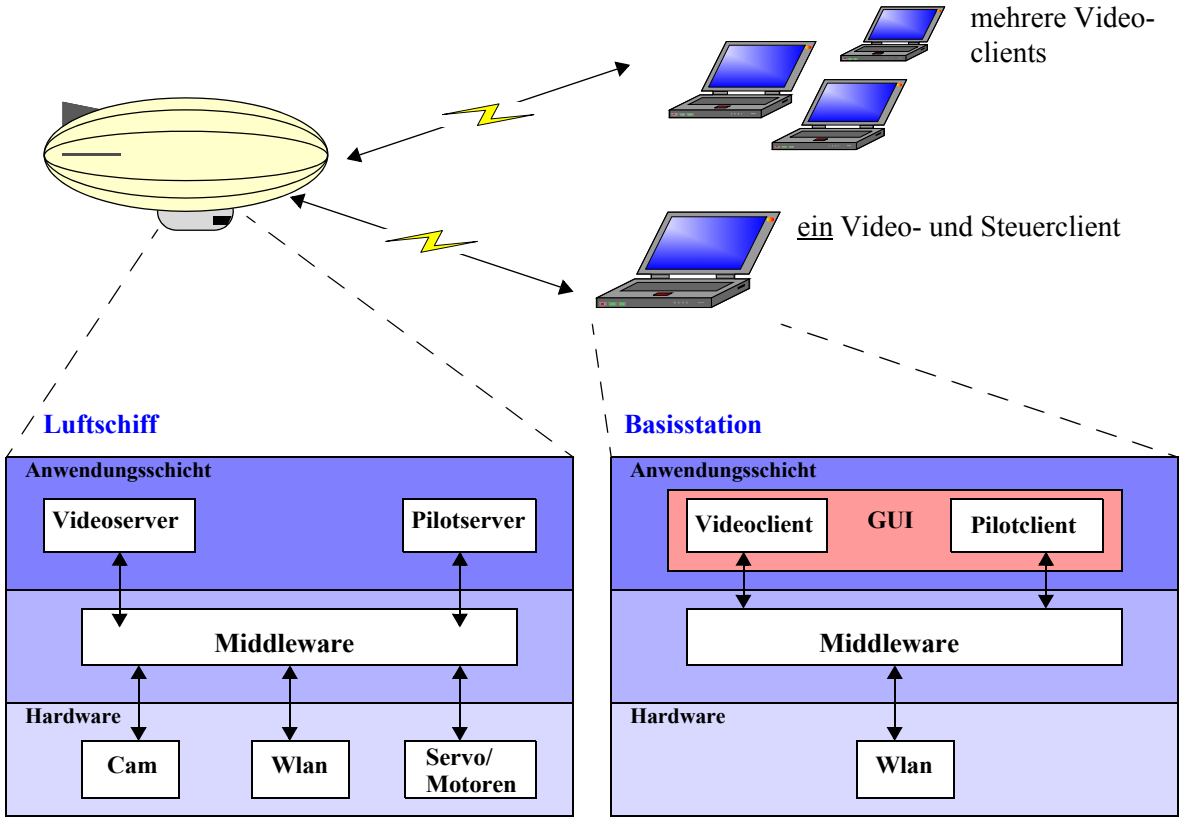


Abb. 3-3: Architektur und Struktur der

Analog dazu ergibt sich die Struktur auf der Clientseite. Die beiden Clients interagieren über eine *Middleware* mit ihren jeweiligen Servern. Hier ist die benötigte Hardware, das Gamepad und das Ausgabegerät nicht explizit dargestellt, da diese von einer entsprechenden Benutzerschnittstelle (*GUI*) zur Verfügung gestellt wird und somit nicht direkt mit der Middleware kommuniziert.

3.4 Funktionalität der Anwendung

In diesem Kapitel werden die einzelnen Funktionalitäten der Anwendung vorgestellt. Für eine Maßschneiderung der Middleware ist es wichtig, alle benötigten Funktionalitäten genau zu kennen, da diese einen sehr großen Einfluss auf die benötigten Komponenten des Kommunikationssystems haben.

3.4.1 Die Videoanwendung

Die Videoanwendung besteht aus einem Videoserver und mehreren Videoclients (Basisstationen). Der Client behandelt die Benutzereingaben und stellt die Videodaten und -parameter geeignet dar. Benutzereingaben können nur von der mobilen Station entgegengenommen werden, die das Fluggerät auch steuert. Die Videoanwendung besteht aus folgenden Komponenten:

- Videofenster zur Darstellung der Videodaten
- Dialog zur Eingabe bzw. Auswahl des Videoservers
- Dialog zur Eingabe von neuen Dienstgüteklassen

- Textfeld zur Darstellung der Verbindungsparameter
- einem Videoserver zum Übertragen der Bilder

Die Funktionalitäten der Videoanwendung lassen sich in zwei Gruppen unterteilen, zum einen die Sende-/Empfangs-, zum anderen die Dienstgütefunktionalitäten. Die Sende- und Empfangsfunktionalitäten umfassen folgende Punkte:

(F-1) Server: Senden der Bilder Unicast/Broadcast

(F-2) Client: Empfangen der Bilder

(F-3) Client: Anzeigen der Bilder inkl. Framerate und Qualität, und

(F-4) Client: Speichern und Drucken der Bilder

Die Dienstgütefunktionalitäten sind:

(F-5) Client/Server: Hinzufügen und Ändern von Dienstgüteklassen und

(F-6) Client/Server: Setzen der aktuellen Dienstgüteklasse

Die Dienstgüteklassen beschreiben den aktuellen Anwendungsfall der Videoübertragung. Es wurden folgende Klassen identifiziert:

Klasse 1 Suchflug (search): Ein Suchflug hat die primäre Aufgabe, vermisste Personen aufzuspüren. Dazu ist eine hohe Qualität der Bilder notwendig. Eine mittlere bis hohe Bildrate hilft zudem, auch langsame Bewegungen zu erkennen.

Klasse 2 Überwachungsflug (watch): Beim Überwachungsflug will man vor allem kleine und langsame Bewegungen erkennen können. Dazu ist eine mittlere bis hohe Qualität der Bilder und eine hohe bis sehr hohe Bildrate erforderlich.

Klasse 3 Panoramaflug (panorama): Eine sehr hohe Qualität der Bilder ist erwünscht, um Panoramabilder zu erstellen. Die Bildrate ist nebensächlich.

Klasse 4 Soloflug (solo): Der Soloflug dient alleine dazu, das Luftschiff zu fliegen. Es findet keine Videoübertragung statt. Dieser Anwendungsfall dient nicht dazu, das Luftschiff mit Hilfe der kamer zu steuern.

Die obigen Klassen können nach Systemstart durch neue ergänzt werden, falls weitere Anwendungsfälle im laufenden Betrieb identifizieren werden. Dabei muss jedoch auch ein Dienstgüte-Mapping mit angegeben werden (siehe Kapitel 4.3.1).

Zur Kommunikation mit dem Server werden folgende Benutzerkommandos verwendet:

- *identifyVideoQosClass(QosClass):* legt den aktuellen Anwendungsfall *QosClass* fest
- *defineNewVideoQosClass(QosClass, minFPS, optFPS, minQual, optQual):* Festlegung eines neuen Anwendungsfalls
- *enableCam(QosClass):* startet die Videoübertragung mit der gewünschten Dienstgüteklasse
- *disableCam:* beendet die Videoübertragung

Der Server kommuniziert mit den entsprechenden Clients über folgenden Nachrichten, die dem Benutzer entsprechend dargestellt werden:

- *qosClassSet(QosClass):* Anwendungsfall *QosClass* erfolgreich gesetzt
- *qosClassNotSet(QosClass):* Anwendungsfall *QosClass* konnte momentan nicht gesetzt werden.
- *image(Image, FPS, Quality):* aktuelles Bild *Image* mit den Parametern Bildrate *FPS* und Qualität des Bildes *Quality*
- *curFeedback(Feedback)*: dieses Signal wird später benötigt, um den Videoclient über die

aktuelle Dienstgütesituation zu informieren

Die dienstgütespezifischen Signale (z.B. Reservierung oder aktuelle Netzlast) werden nur mit der Middleware ausgetauscht und in Kapitel 5 beschrieben. Die folgende Abbildung zeigt zwei typische Szenarien der Videoanwendung. Im linken Bild konnte nach dem *enableCam* Befehl die aktuelle Dienstgüteklasse erfogreich gesetzt werden. Der VideoServer quittiert dies mit einem *qosClassSet* und beginnt sofort mit der Übertragung der Bilder, bis der Client diese Übertragung mit einem *disableCam* abbricht. Im rechten Bild war ein Setzen der Dienstgüteklasse auf Serverseite nicht möglich. Der Server benachrichtigt den Client darüber mit einem *qosClassNotSet* Signal.

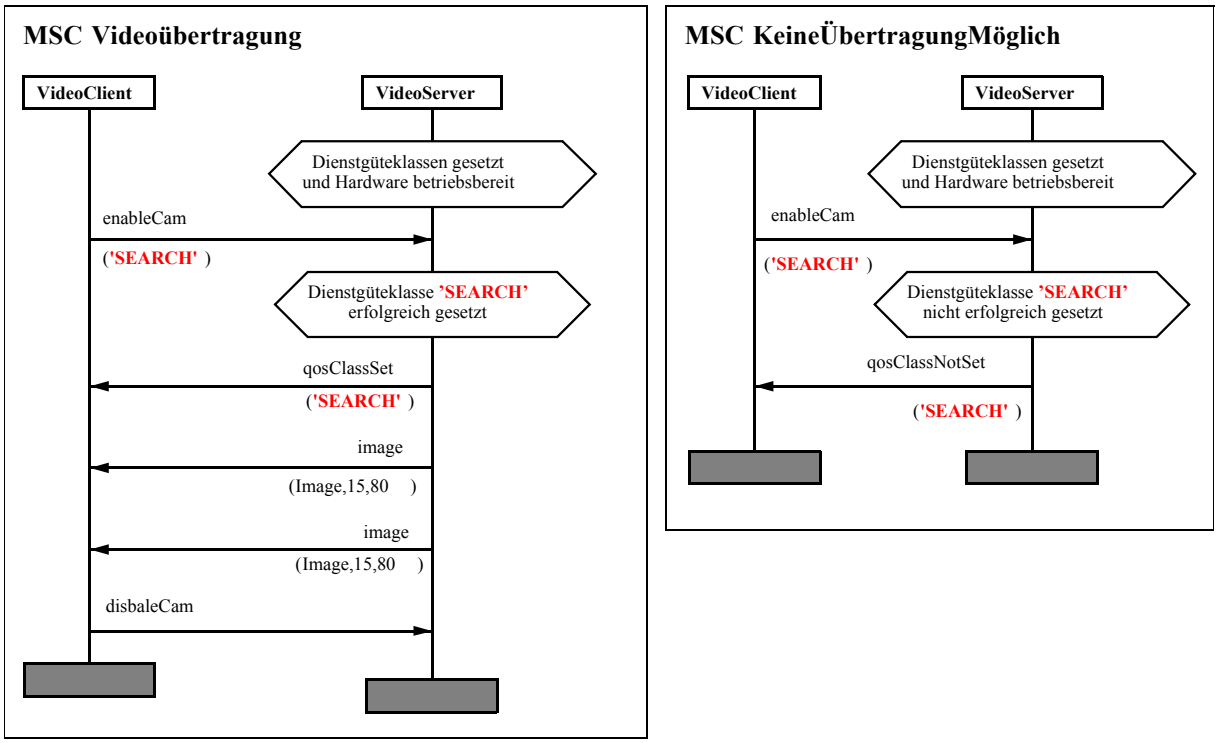


Abb. 3-4: MSCs der Videoanwendung

3.4.2 Die Steuerungsanwendung

Die Steuerung besteht wie die Videoanwendung aus einem Server (Fluggerät) und mehreren Clients, von denen jedoch immer nur einer aktiv sein kann. Der Server nimmt die Steuerungssignale entgegen und kontrolliert so das Fluggerät. Der Steuerungsclient, die aktive mobile Station, nimmt Benutzereingaben in Form von Steuerkommandos und Dienstgüteparametern entgegen, sendet diese zum entsprechenden Server und stellt die aktuellen Motor-, Servo- und Spannungswerte des internen Akkus graphisch dar. Die Anwendung besteht aus folgenden Komponenten:

- Server zur Steuerung des Fluggerätes und Senden der Akkurestkapazität
- Dialog zur Eingabe bzw. Auswahl des Fluggerätes
- Fenster zur Darstellung der Steuerungsparameter
- Dialog zur Eingabe bzw. Auswahl der Dienstgüteklasse

Die Steuerung bietet folgende Funktionalitäten:

(F-7) Client/Server: Steuerung des Luftschiffes (inklusive Senden und Empfangen der Steu-

erungsdaten)

(F-8) Server: Übertragen der aktuellen Luftschiffparameter. Im Einzelnen sind dies die Steuerungswerte des Servos und der Motoren und die momentane Akkuspannung zur Berechnung der verbleibenden Flugdauer

(F-9) Client: Anzeigen der aktuellen Steuerparameter und des Akkustandes

(F-10) Client/Server: Flugunterstützung durch Schwebefunktion

(F-11) Client: Speichern der aktuellen Parameter (zum Wiederherstellen der Schwebefunktion nach einem Neustart)

(F-12) Client/Server: Hinzufügen und Ändern von Dienstgüteklassen und

(F-13) Client/Server: Setzen der aktuellen Dienstgüteklasse

Analog zur Videoanwendung gibt es auch hier verschiedene Anforderungen an die Übertragung der Steuerungssignal, die je nach Situation variieren können. Für die Übertragung der Luftschiffparameter werden keine verschiedenen Anwendungsfälle identifiziert.

Klasse 1 Ungünstige Umgebung (hard): Die Flugumgebung des Luftschiffes erfordert rasche und präzise Richtungswechsel, da wenig Platz zum Manövrieren ist (z.B. kleiner Hörsaal). Die Verzögerung der Steuersignale muss sehr klein sein.

Klasse 2 Günstige Umgebung (easy): Das Einsatzgebiet des Zeppelins ist bietet ausreichend Platz für sämtliche Flugmanöver (z.B. große überdachte Arena). Diese müssen nicht besonders schnell und präzise ausgeführt werden. Die Steuersignale können auch mit leichter Verzögerung ankommen.

Zur Kommunikation mit dem Server verwendet der Client folgende Kommandos:

- *identifyPilotQosClass(QosClass):* Legt den aktuellen Anwendungsfall *QosClass* für die Übertragung der Luftschiffparameter fest. Da nur eine Dienstgüteklasse dafür verfügbar ist, wird der Befehl erst nach Anlegen von weiteren Klassen benötigt.
- *defineNewPilotQosClass(QosClass, maxDelay, optDelay):* Festlegung einer neuen Dienstgüteklasse für die Übertragung der Luftschiffparameter
- *newCtrlValues(CtrlVal1,CtrlVal2, CtrlVal3):* Signal mit den neuen Steuerkommandos für das Luftschiff (Servo, Motor1, Motor2)
- *startZeppelin*: Startet den Zeppelin, schaltet die Motoren an. Es wird keine Angabe einer Dienstgüteklasse benötigt, da in diesem Fall die Festlegung lokal auf Clientseite erfolgt.
- *stopZeppelin*: schaltet die Motoren aus

Der Server kommuniziert mit der steuernden Basisstationen über folgenden Nachrichten:

- *curCtrlValues(CtrlVal1,CtrlVal2, CtrlVal3, Voltage):* die aktuellen Steuerparameter und die Akkuspannung des Luftschiffes
- *qosClassSet(QosClass):* Anwendungsfall *QosClass* erfolgreich gesetzt
- *qosClassNotSet(QosClass):* Anwendungsfall *QosClass* konnte momentan nicht gesetzt werden.

Auch hier werden die dienstgütespezifischen Signale nur mit der Middleware ausgetauscht und in Kapitel 5 beschrieben. Abbildung 3-5 zeigt eine typische Client-Server-Kommunikation. nachdem die Dienstgüteklasse 'HARD' lokal erfolgreich gesetzt wurde, signalisiert der Client dem Server seine Bereitschaft zur Steuerung indem er mit dem Signal *enableZeppelin* die Motoren und den Servo des Zeppelins aktiviert. Die Steuerungswerte für den Zeppelin werden ab jetzt ununterbrochen periodisch übertragen, bis das Luftschiff mit dem Befehl *disableZeppelin* wieder deaktiviert wird. Während des Steuerungsvorganges sendet der Server ununterbrochen die aktuellen Luftschiffparameter *curCtrlValues* an den Client.

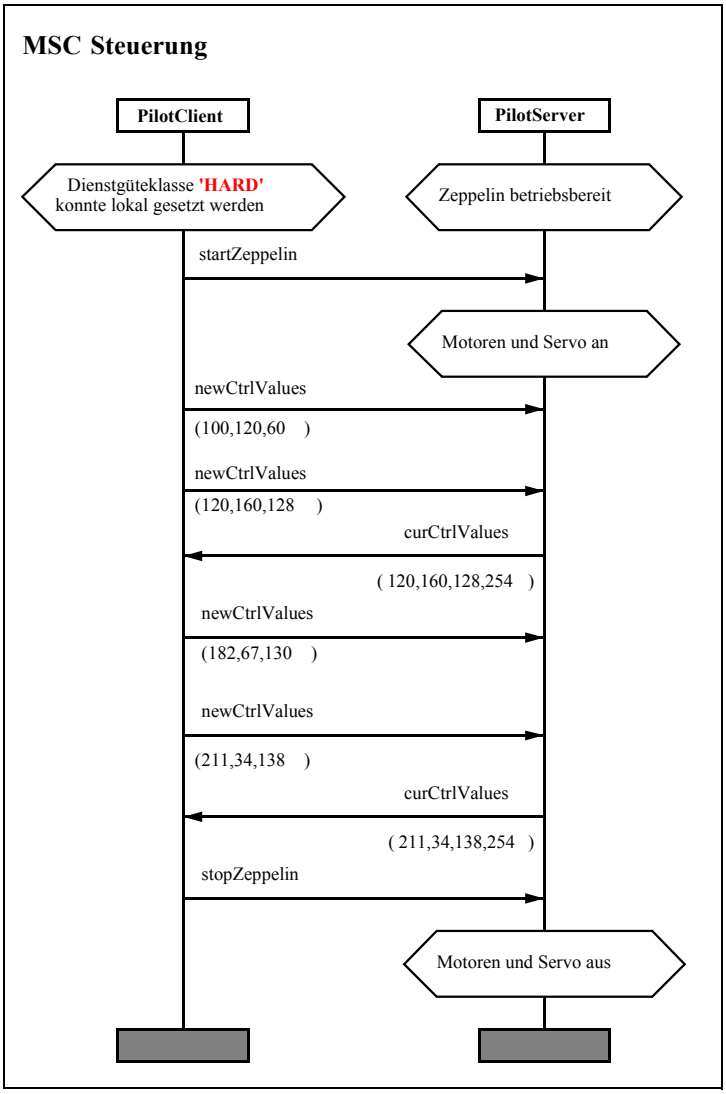


Abb. 3-5: MSC der Steuerungsanwendung

3.5 Dienstgüte-Spezifikation der Anwendung

Die QoS-Spezifikation erfolgt nach den in Kapitel 2.3.1 vorgestellten Kriterien. Es müssen zwei Gesichtspunkte berücksichtigt werden:

Erstens die Dienstgüte, die ein Benutzer oder eine Anwendung benötigt. Es wird für jede Richtung, in der Dienstgüte reserviert werden muss, eine Spezifikation angegeben. Dienstgüte muss genau dann reserviert werden, falls regelmäßig (wichtige) Daten übertragen werden und man sichergehen will, dass diese mit einer hohen Wahrscheinlichkeit gemäß gegebener Vorgaben ankommen. Man darf nicht vergessen, dass in mobilen Ad-Hoc Netzen nur statistische Garantien gegeben werden können. Regelmäßige Datenaufkommen sind Steuersignale, Videodaten und Luftschiffparameter, da diese permanent von dem Client oder Server gesendet werden. Für sporadische Signale wie das Setzen der Dienstgüteklasse wird keine Reservierung vorgenommen. Das Datenvolumen dieser Signale ist sehr gering im Vergleich zu den regelmäßig gesendeten Signalen und deshalb sollten diese Signale immer gescheduled werden können (z.B. als

Fülldaten für einen nicht vollständig belegten Zeitslot des Basisdienstes).

Zweitens ist eine geeignete Anzeige der gewährten Dienstgüte unerlässlich. Abhängig von der gewählten Dienstgüteklasse wird dem Benutzer ein entsprechendes Feedback gegeben. Das Dienstgütefeedback *Grün* bedeutet, dass die geforderte Dienstgüte momentan garantiert werden kann, wohingegen *Rot* keine Garantie anzeigt. *Gelb* stellt eine befriedigende Dienstgüte dar und ist für die meisten Anwendungen ausreichend.

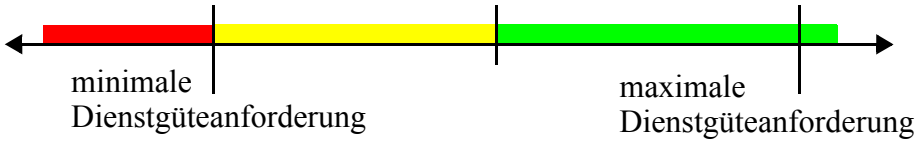


Abb. 3-6: Feedback der gewährten Dienstgüte

Die unterschiedlichen abstrakten Sichten (Spezifikationen) der verschiedene Schichten der Middleware auf die Dienstgüteressourcen (z.B. Bildrate auf Anwendungsebene, Rahmen/Sekunde auf Verbindungsebene) und die Umsetzung zwischen diesen Sichten bzw. Spezifikationen werden in Kapitel 4.3.1 (Dienstgüte-Mapping) beschrieben. Eine Übersicht über die Spezifikationen auf den verschiedenen Systemebenen geben die folgenden Tabellen (Tabelle 3-2 und Tabelle 3-3).

Tabelle 3-2: Dienstgütespez. auf den verschiedenen Systemebenen: Video

Ebene	geforderte Dienstgüte	gewährte Dienstgüte
Benutzer	Anwendungsfall[a]	rot, gelb grün
Anwendung	Bilder/Sekunde, Qualität der Bilder[b] (min. und opt.)	Datenvolumen/Delay[c]
Middleware	Datenvolumen/Delay[c] (min. und opt.)	Zeitschlitze/Sekunde
Basisdienst	-	Zeitschlitze/Sekunde

a. Dienstgüteklasse nach Kap. 3.4.1
b. Einer Qualität von 100% entspricht eine JPEG-Kompression von 0%
c. in [kBit],[ms]

Tabelle 3-3: Dienstgütespez. auf den verschiedenen Systemebenen: Steuerung

Ebene	geforderte Dienstgüte	gewährte Dienstgüte
Benutzer	Anwendungsfall[a]	rot, gelb, grün
Anwendung	Delay (min. und opt.)	Datenvolumen/Delay[b]
Middleware	Datenvolumen/Delay (min. und opt.)	Zeitschlitze/Sekunde
Basisdienst	-	Zeitschlitze/Sekunde

a. Dienstgüteklasse nach Kap. 3.4.2
b. in [kBit],[ms]

3.5.1 Level of Service

Der *level of service* beschreibt die qualitative Festlegung der Dienstgüte. Die Übertragung der Videodaten stellt eine wichtige Funktionalität dar, auf die bei den meisten der vorliegenden Anwendungsfälle nicht verzichtet werden kann. Über die meiste Zeit gesehen muss die von der jeweiligen Dienstgüteklasse vorgegebene minimale Bildrate und Qualität (Bandbreite) verfügbar sein (statistische Garantien).

Noch wichtiger ist die Verfügbarkeit von Ressourcen bei der Steuerung des Fluggerätes. Die maximale Verzögerung und die minimale Bandbreite darf nicht über- bzw. unterschritten werden. Diese deterministische Anforderung kann in mobilen Ad-Hoc Netzen nicht umgesetzt werden. Um dennoch zwischen verschiedenen dringenden Dienstgüteanforderungen der gleichen Dienstgütestufe unterscheiden zu können, werden Prioritäten eingeführt. Somit erhält die dringendste Anwendung im Bedarfsfall die gesamte verfügbare Bandbreite.

Bei der Steuerungsanwendung muss nicht nur Bandbreite von der jeweiligen mobilen Station zum Fluggerät reserviert werden, sondern auch in umgekehrter Richtung für die Übertragung der Luftschiffparameter (aktuelle Stellung des Servos und der Motoren, Akkuspannung). Diese Reservierung ist notwendig, da diese Daten regelmäßig übertragen werden und wichtige Informationen beinhalten. Dennoch ist hier die Priorität am geringsten.

Folgende Tabelle zeigt die genaue Spezifizierung des Level of Service.

Tabelle 3-4:

Anwendung	Level of Service	Priorität
Steuerung	statistisch	1
Videoübertragung	statistisch	2
Luftschiffparameter	statistisch	3

3.5.2 Flow Performance Specification

Die *Flow Performance specification* beschreibt die benötigte Performanz der Teilanwendungen. Die Videoübertragung benötigt einen hohen Durchsatz, um die benötigten Bilder mit der gewünschten Bildrate übertragen zu können. Abbildung 3-7 zeigt die benötigte Bandbreite in Abhängigkeit zur Framerate und Bildqualität.

Für eine Videoübertragung reicht es nicht aus, nur Bandbreite statistisch zu reservieren. Für eine gleichmäßige („flüssige“) Darstellen der Bilder muss auch sichergestellt werden, dass die Sendeverzögerungen und die maximalen Abweichungen von diesen Verzögerungen (Jitter) nicht zu groß werden. Da kein Empfangspuffer verwendet wird, erfolgt die Ausgabe der Bilder sobald diese beim Empfänger angekommen sind. Es wird in diesem Fall davon ausgegangen, dass der Basisdienst reihenfolgeerhaltend ist. Bei der vorliegenden Anwendung erfolgt nur eine statistische Reservierung des Jitters, nicht der Sendeverzögerung. Das hat den Grund, dass in diesem Fall mehr Wert auf gleichmäßige als auf schnelle Darstellung gelegt wird. Mit der statistischen Reservierung von Jitter wird somit der Unterschied zwischem dem maximalen und minimalen Delay mit einer bestimmten Wahrscheinlichkeit festgelegt. Wählt man die Sendeverzögerung als Dienstgüteparameter, so erhält man einen maximalen Jitter als Parameter (auch hier gibt es nur statistische Garantien für den maximalen Wert).

Abbildung 3-8a zeigt den optimalen Fall. Alle Bilder kommen mit der gleichen Verzögerung beim Empfänger an. In der Realität ist dieser Fall nicht zu erreichen, da es in mobilen Ad-Hoc

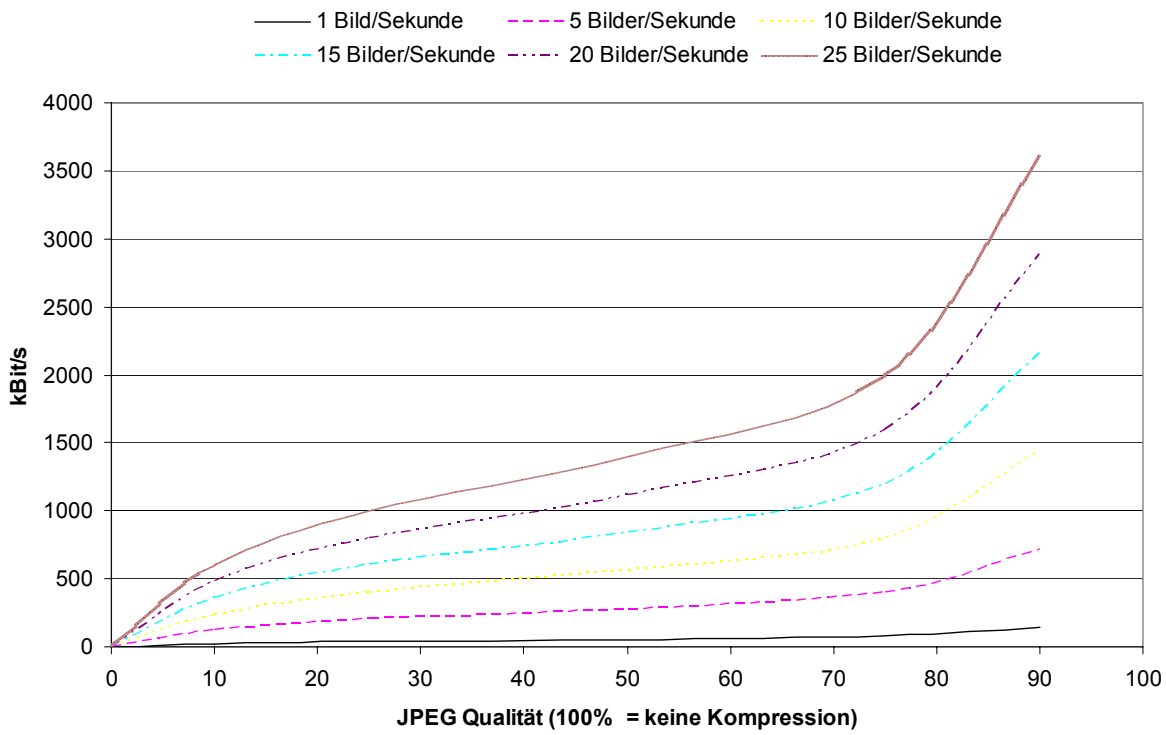


Abb. 3-7: Im Durchschnitt benötigte Bandbreite der Videoübertragung (gemessen, 320x240)

Netzen immer Schwankungen in der Sendeverzögerungen beispielsweise durch Störungen gibt. Um dem optimalen Ergebnis möglichst nahe zu kommen, legt man wie erwähnt einen maximalen Jitter fest. Dieser muss bei der Dienstgütereservierung berücksichtigt werden. Mit der Festlegung und der Reservierung des Jitters sorgt man dafür, dass die Bilder statistisch gesehen immer in einem bestimmten Intervall um die mittlere Sendeverzögerung ankommen und so dem Benutzer ein gleichmäßiges Betrachten ermöglichen (Abbildung 3-8b). Abbildung 3-8c zeigt ein unbrauchbares Übertragungsverhalten. Obwohl bei allen drei Szenarien (Abb. 3-8a,b,c) die gleiche Bandbreite benötigt wird, ist in diesem Fall durch die starke Sendeverzögerung der Bilder 2, 3 und 4 keine zufriedenstellende Übertragung zu erwarten. Die genaue Größe der Reservierung des Jitters *j* ist bei dieser Anwendung das Reziproke der Bildrate *fps*:

$$j = \frac{1}{fps}\ ,\ j = [s]$$

Bei der Übertragung der Steuersignale sieht es ähnlich aus. Im Gegensatz zur Videoübertragung wird hier jedoch ein Delay zusätzlich zur Bandbreite reserviert, da Daten schnellstmöglich beim Luftschiff ankommen müssen. Die Festlegung von einem maximalen Jitter als Dienstgüteparameter ist nicht erforderlich. Eine geeignete Komponente in der anwendungspezifischen Middleware wird für eine regelmäßige Ansteuerung der Servo/Motoren sorgen (hier liegt eine Art Puffer vor). Die Motoren und der Servo haben je 256 verschiedene Einstellung. Die zu reservierende Bandbreite ist in jedem Anwendungsfall gleich und beträgt somit $3 \times 8\text{Bit} = 24\text{Bit}$ pro Signal. Sendet man die neuen Werte alle 20 ms, so ergibt sich eine gesamte Bandbreite von 1200 Bit/s.

Abbildung 3-9a zeigt eine optimale Übertragung. Alle Steuersignale kommen innerhalb des vorgegebenen Zeitrahmens beim Empfänger und somit bei der zu steuernden Einheit an und ein problemloses Verhalten kann somit garantiert werden. Im Gegensatz dazu zeigt Abbildung 3-9b den inakzeptablen Fall. Einige Steuersignal kommen nicht innerhalb des von der Dienstgü-

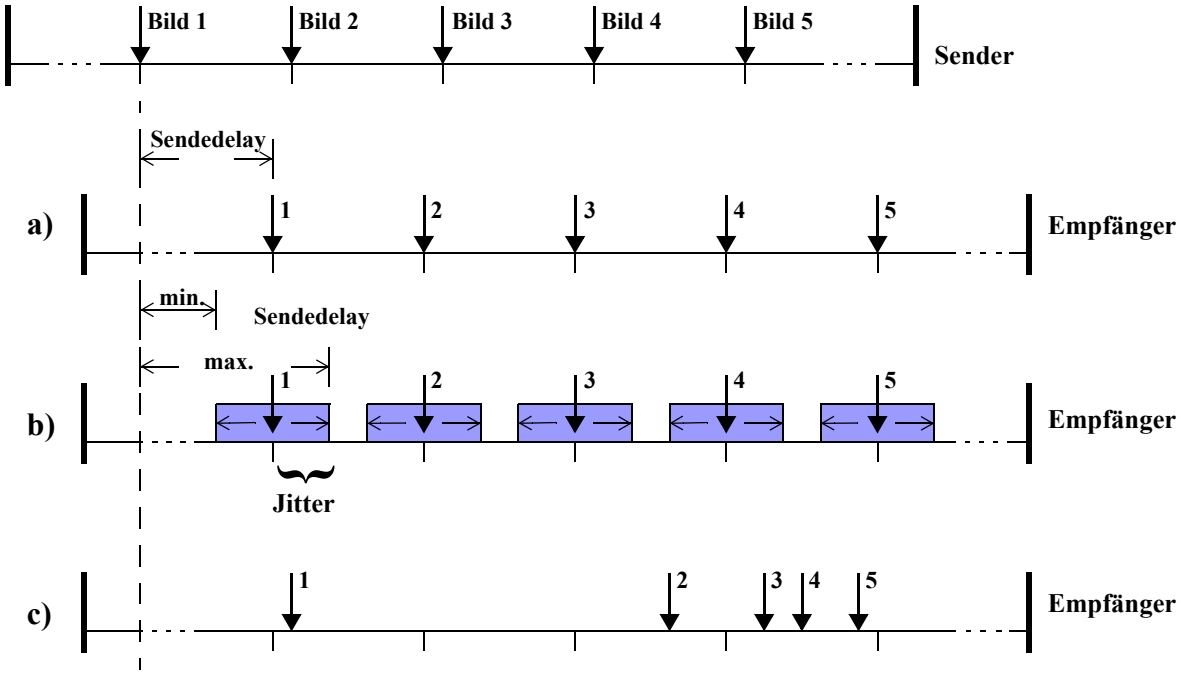


Abb. 3-8: a) Optimale Übertragung
b) Akzeptable Übertragung
c) Inakzeptable Übertragung

teklasse festgelegten Zeitraums beim Empfänger an. Das Luftschiff kann so unter Umständen nicht schnell genug auf die Steuerbefehle reagieren, was zu einer kritischen Situation führt. Die Übertragung der Luftschiffparameter ist prinzipiell identisch mit der Übertragung der Steuerungswerte. Deshalb wird an dieser Stelle nicht genauer auf diese Teilanwendung eingegangen.

Die endgültige Performance-Spezifikation kann bei dieser Anwendung nur zusammen mit dem Dienstgüte-Mapping (Kapitel 4.3.1) erfolgen, da erst dort die Umsetzung von den Dienstgüteklassen auf die konkreten Werte erfolgt.

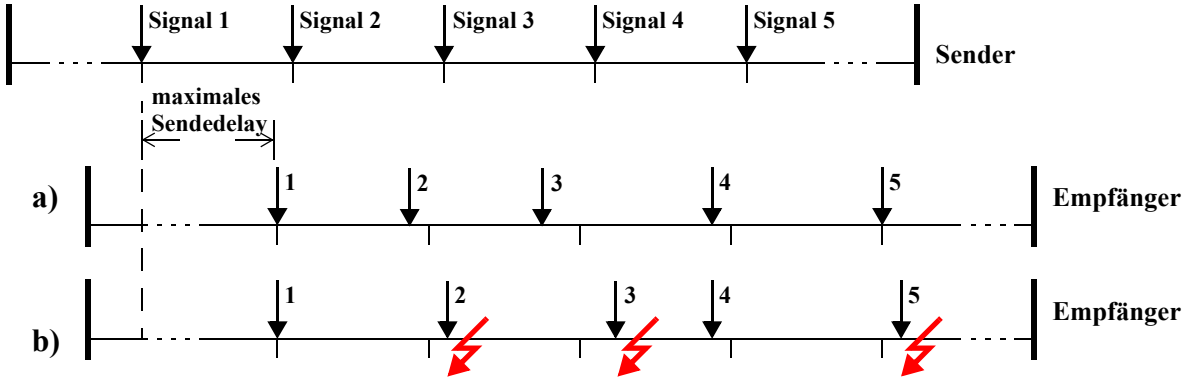


Abb. 3-9: a)Optimale Übertragung der Steuerbefehle
b) Inakzeptable Übertragung der Steuerbefehle

3.5.3 Qos-Management Policies

Im Falle einer Dienstgüteverletzung, also einer Nichterfüllung oder Nichteinhaltung der reservierten Dienstgüte, müssen geeignete Maßnahmen ergriffen werden. Diese werden durch die *Qos-Management Policies* beschrieben.

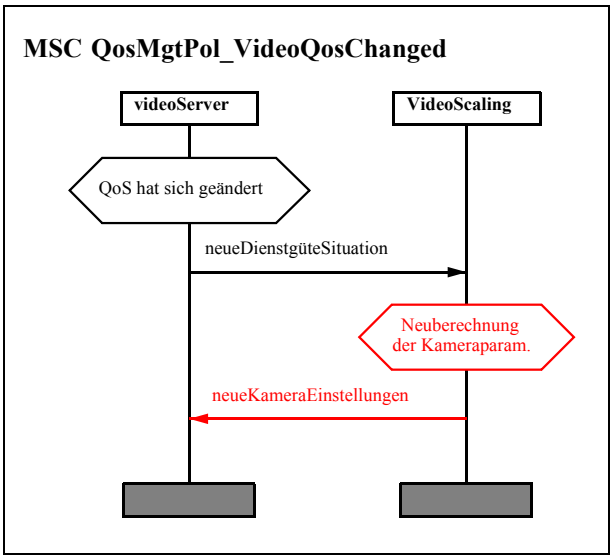


Abb. 3-10: Anpassung an veränderte Dienstgütesituation

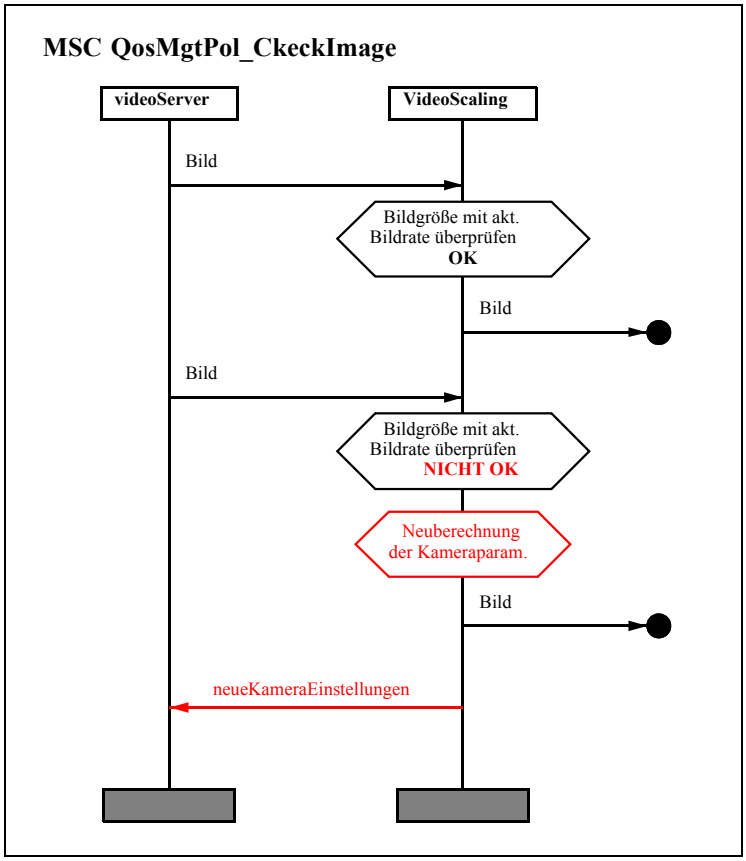


Abb. 3-11: Anpassung an neue Bildgröße

Die Übertragung der Videodaten stellt in diesem Fall die schwierigere Anwendung dar. Kann die reservierte Bandbreite vom Basisdienst nicht garantiert bzw. aufrecht gehalten werden, so müssen die Parameter der Videoübertragung (Bildrate, Qualität) an die neue Situation angepasst werden. Das geschieht weitgehend automatisch in einer Skalierungskomponente (siehe Kapitel 5.1.4). Die Größe der aktuellen Bilder wird mit der verfügbaren Bandbreite verglichen. Dadurch können, durch Neueinstellung der Kameraparameter innerhalb des Intervalls, welches durch die vorliegende Dienstgüteklasse gegeben ist, zukünftige Bilder so angepasst werden, so dass die vorliegende Bandbreite nicht überschritten wird (Abbildung 3-10). Es ist darauf zu achten, dass die minimalen Parameter der Dienstgüteklasse nicht unterschritten werden. Falls die Bandbreite zu gering für eine Videoübertragung ist, also die Reservierung aufgrund der Eigenschaften von Ad-Hoc Netzen nicht mehr getätigt werden kann, muss dem Benutzer ein geeignetes Feedback gegeben werden. Ein weiteres Problem bei der Übertragung der Bilder ist die

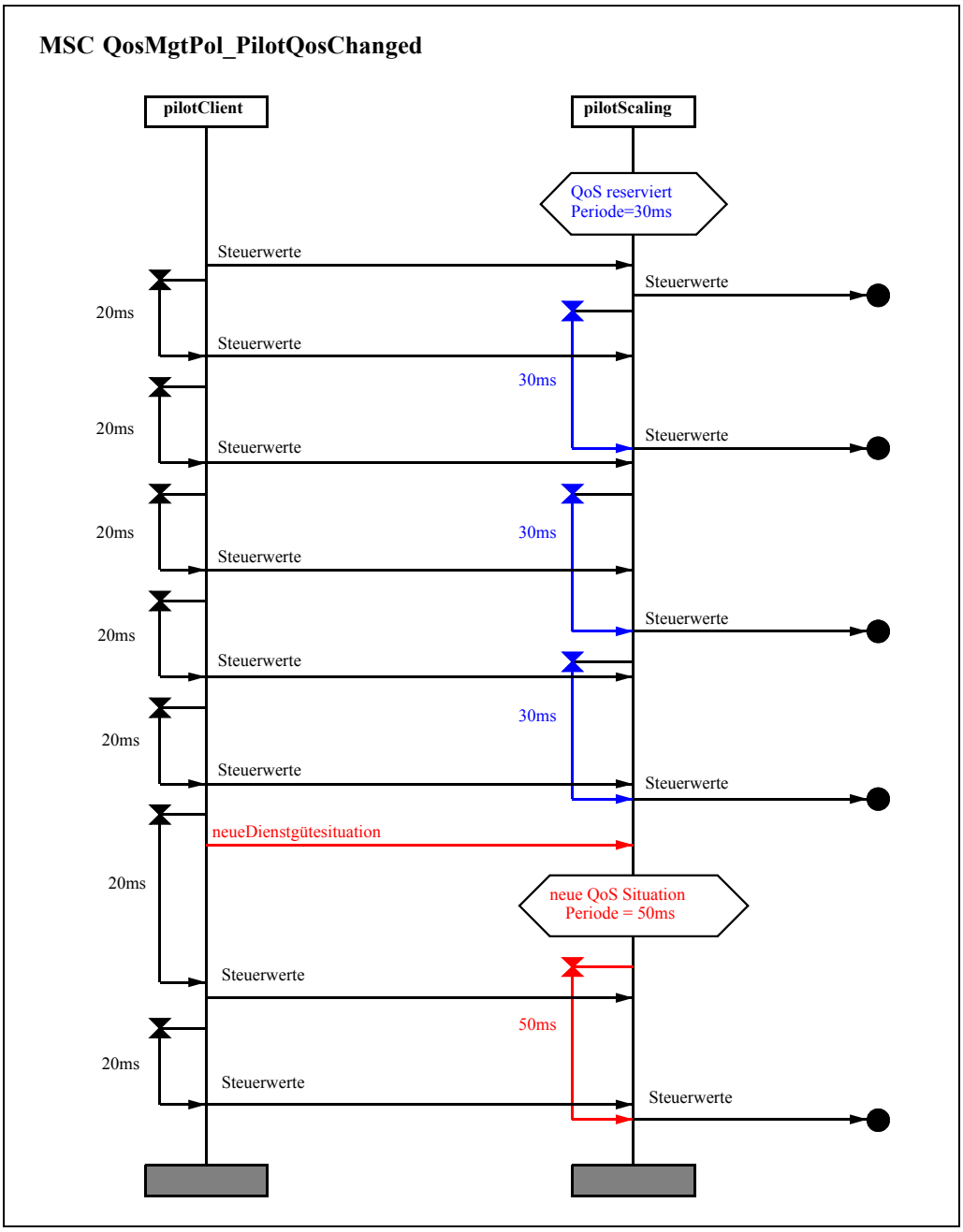


Abb. 3-12: Anpassung an neue Dienstgütesituation

schlechte Vorhersagbarkeit der tatsächlichen Größe eines Bildes. Das verwendete Jpeg-Format liefert abhängig von dem aktuellen „Motiv“ bei gleichbleibendem Kompressionsfaktor eine unterschiedliche Bildgröße. Deswegen ist es unerlässlich, permanent die verwendete Bandbreite zu überwachen und ggf. die Videoparameter neu einzustellen (Abb. 3-11).

Die Übertragung der Steuerdaten und der Luftschiffparameter erfordert auch eine Neuanpassung im Falle einer verschlechterten Dienstgütesituation. Bei diesen beiden Teilanwendungen kann eine Anpassung nur durch ein Vergrößern der Sendeintervalle erreicht werden, was eine Verringerung der benötigten Bandbreite bedeutet (Abbildung 3-12). Es ist auch hier darauf zu achten, dass die minimalen Vorgaben der jeweiligen Klasse nicht unterschritten werden.

Die Anwendung kennt nur ihre jeweilige Dienstgüteklasse und abstrahiert vollständig von den Parametern und Konfigurationen der Middleware. Sie übermittelt ständig mit einer festen Periode ihre Daten an die Middleware, die diese dann geeignet verarbeitet und mit einer auf die Netzsituation angepassten Periode an den Server sendet. Findet eine Verletzung des vorliegenden Dienstgütevertrags aufgrund einer veränderten Netzsituation statt oder ist eine Verletzung durch sinkende Bandbreiten wahrscheinlich, so muss dies dem Anwender bzw. der Anwendung durch ein entsprechendes Feedback in jedem Fall mitgeteilt werden (vgl. Kap. 3.5). Für die Übertragung der Luftschiffparameter wird dem Benutzer kein Feedback gegeben.

3.5.4 Cost of Service

Der Preis, den ein Benutzer oder eine Anwendung bereit ist zu zahlen, um eine von ihm bestimmte Dienstgüte zu erlangen, ist implizit schon durch die Festlegung der Dienstgüteklassen, also durch das Mapping (Kapitel 4.3.1), gegeben. Der Benutzer ist bereit, seine Anforderungen an die Übertragung auf ein Minimum zu reduzieren, um eine bestimmte Dienstgüteklasse aufrecht zu erhalten. Beispielsweise kann eine Videoübertragung die Bildrate herabsetzen, bei gegebener Bandbreite dennoch die Bilder in höchster Qualität zu senden. Diese Herabsetzung der Bildrate auf einen unteren Wert ist in diesem Fall der Preis, der gezahlt werden muss.

Die Prioritäten, die durch die Spezifizierung des *level of service* (Kap. 3.5.1) festgelegt werden, beschreiben eine weitere „Kosteneinteilung“ der Dienstgüte. Je nach Situation nimmt der Benutzer in Kauf, dass niederpriore Funktionalitäten oder Dienste nicht mehr erfüllt, damit hochpriore weiterhin funktionieren können.

3.5.5 Flow synchronisation specification

Da es keine Datenströme gibt, die synchronisiert werden müssen, entfällt die *flow synchronisation specification*.

3.6 Zusammenfassung

In diesem Kapitel wurde die Architektur und die Funktionalität der Anwendung vorgestellt. Anwendungsfälle bzw. Dienstgüteklassen wurden identifiziert und beschrieben. Für die beiden Teilapplikationen, die Videoübertragung und die Steuerungsanwendung, wurde eine detailierte Dienstgütespezifikation gemäß den in Kapitel 2.3.1 Gesichtspunkten erstellt. Diese Spezifikation und die Funktionalitäten bilden die Grundlage für die Maßschneiderung der Middleware.

4 Middleware

Eine Middleware ist eine Protokollschicht, die eine Menge von Diensten kapselt. Sie liegt zwischen Anwendung und Systemhardware und regelt alle Zugriffe auf diese. Die typische Aufgabe der Middleware ist es, Dienste bereit zustellen und Dienstnutzer (Anwendungen) mit diesen Diensten zusammenzufügen. Allgemein hat eine Middleware folgende Aufgaben:

- Herstellung von Homogenität (semantische und technische Unterschiede zwischen Anwendung und Hardware verbergen)
- Dienstbereitstellung
- Dienstaktivierung/-beendigung
- Persistenz (der Dienste)

4.1 Funktionalitäten einer Middleware

Eine maßgeschneiderte Middleware besteht aus genau der Anzahl von Diensten und Funktionalitäten, die nötig ist, um die Anforderungen der Anwendung zu erfüllen. In unserem Fallbeispiel „Zeppelin“ kann man die Funktionalitäten in drei Kategorien einteilen: Kommunikation, Sicherheit und Dienstgüte.

4.1.1 Kommunikation

Da wir Kommunikation in Ad-Hoc Netzen betrachten, uns also mit einer sehr dynamischen Domäne befassen, brauchen wir sehr viele Funktionalitäten. Die meisten werden schon von der Basistechnologie bereitgestellt. Der Vollständigkeit halber werden sie jedoch an dieser Stelle noch einmal aufgeführt.

Fehlererkennung (und -behebung): Die Fehlererkennung findet in der MAC-Schicht des WLAN-Treibers statt. Pakete, die unsere maßgeschneiderte Middleware empfängt, sind somit immer fehlerfrei. Verlorengegangene Pakete können jedoch noch nicht von der Basistechnologie erkannt werden.

Flusskontrolle: Unter Flusskontrolle fasst man die Maßnahmen zusammen, die einen Datenstrom regulieren bzw. formen. Darunter fallen z.B. auch Methoden, die verhindern, dass ein Sender schneller Daten sendet, als ein Empfänger empfangen oder das Medium aufnehmen kann. Die Flusskontrolle ist in Funknetzen ein sehr wichtiger Teil der Middleware, da sich die Übertragungsrate sehr schnell ändern kann.

Verbindungsmanagement: Unter Verbindungsmanagement versteht man den Verbindungsauf- und abbau.

Leitwegbestimmung/Routing: Die Leitwegbestimmung ermittelt einen oder mehrere mögliche Routen von einem Quell- zu einem Zielknoten.

Adressierung: Die Adressierung legt den Empfänger anhand eines definierten Adressierungsschemas fest. Mit Hilfe der Adressinformation findet die Leitwegbestimmung statt.

Im Rahmen dieser Diplomarbeit wird nur Single-Hop Kommunikation betrachtet. Desweiteren liegt der Schwerpunkt auf der Entwicklung und Integration von QoS-Mikroprotokollen. Aus diesen Gründen und da es für alle genannten Funktionalitäten schon fertige Implementierungen gibt oder parallele Arbeiten zu diesen Themen stattfinden, wird in dieser Arbeit nicht näher auf Kommunikation im obigen Sinne eingegangen.

4.1.2 Sicherheit

Sicherheitsfunktionalitäten in dieser Arbeit beziehen sich weniger auf verschlüsselte Verbindungen als auf ein sicheres, fehlertolerantes Verhalten der Middleware und der Anwendung bei Auftreten von Fehlverhalten. Der Übergang in einen vordefinierten Systemzustand im Falle eines Fehlers führt zu sogenannten *fail-operational* oder *fail-safe* Zuständen [4], je nachdem ob ein sicherer Zustand definiert werden kann (*fail-safe*) oder ob noch eine minimale Funktionalität benötigt wird, um eine Katastrophe zu verhindern (*fail-operational*).

Bei einer Ampelsteuerung lässt sich z.B. leicht ein *fail-safe* Zustand finden. Schaltet man alle Ampeln beim Auftreten eines kritischen Fehlers auf rot, so kann man sicher sein, dass eine Katastrophe verhindert werden kann. Tritt jedoch z.B. ein schwerer Fehler in einer Steuerung eines Fluggerätes auf, so kann ein wirklich sicherer Zustand nicht identifiziert werden, da noch ein minimaler Satz von Funktionalitäten benötigt wird, um das Flugzeug sicher unter Kontrolle zu halten. Der Zustand, der diesen minimalen Satz von Funktionalitäten bereitstellt, nennt man folglich *fail-operational*.

In unserem Fall bedeutet das, dass es uns nicht möglich ist, einen *fail-state* Zustand zu definieren. Wir können nur das Fluggerät im Falle eines kritischen Fehlers (z.B. kein Empfang von Steuersignalen mehr möglich) kontrolliert landen lassen und somit einen *fail-operational* Zustand definieren. Ein möglicher Zustand wäre z.B. kein Vortrieb und ein minimales Ansteuern der Auftriebsmotoren zur langsamen Landung (da ein Zeppelin schwerer als Luft ist, sorgt ein minimales Ansteuern der Motoren für einen langsamen Sinkflug).

4.1.3 Dienstgüte

Der zweite Haupteil dieser Arbeit beschäftigt sich mit Dienstgüte. Man kann die Dienstgütefunktionalitäten einer Middleware allgemein in zwei Gruppen einteilen, die applikationsspezifischen und die applikationsunabhängigen Dienstgütefunktionalitäten. Eine umfassende Beschreibung findet sich in Kapitel 2.3.

Die applikationsspezifischen Funktionalitäten sind speziell auf eine Anwendung zugeschnitten. Darunter fällt z.B. das *Dienstgütemapping* oder das *Dienstgüteskaling*. Sie befinden sich in der geschichteten Middleware deshalb direkt unter der Applikation.

Die anwendungsunabhängigen Dienstgütefunktionalitäten abstrahieren komplett von den darüberliegenden Applikationen. Sie ermöglichen der Applikation eine Dienstgüteunterstützung, die unabhängig von der verwendeten Basistechnologie ist. Ein Beispiel ist die *Dienstgüteresservierung* oder allgemein die *Dienstgütekontrolle*. Jedoch müssen einige Forderungen an die Basistechnologie gestellt werden. So ist hier *Zugangstest* unerlässlich, um eine Anforderung auf ihre Realisierbarkeit zu überprüfen und ein *Ressource Reservation Protokoll* wird benötigt, eine zuvor geprüfte Anforderung dann letztendlich zu reservieren.

4.2 Anforderungen der Anwendung an die Middleware

Die Anforderungen der Anwendungen an die Middleware lassen sich entsprechend Kapitel 4.1 auch in drei Kategorien einteilen. Die kommunikationsspezifischen Anforderungen betreffen

alles, was mit reinen Kommunikation ohne Dienstgüte zwischen den Stationen zu tun hat. Man kann sie zu einer Anforderung zusammenfassen:

- fehlerfreies Versenden und Empfangen von Daten (Verbindungsauf- und abbau, Fehlersicherung, etc.).

Diese Anforderung bildet die Grundlage für die zu entwickelnden Mikroprotokolle, da diese einen solchen Basisdienst benötigen. Im weiteren Verlauf der Arbeit wird diese Anforderung als gegeben angesehen.

Eine weitere Anforderung an die Middleware ist die Unterstützung eines *fail-operational* Zustands, d.h. bei Ausfall von einer oder mehrerer Anwendungen soll der Zeppelin dennoch kontrolliert zur Landung gebracht werden können. In unserem Fall ist das nur durch ein langsames Verringern der Flughöhe möglich. Da die Steuerungs- und Videoanwendungen auf Zeppelinseite voneinander getrennt sind, bezieht sich diese Anforderung nur auf den Server der Steuerungsanwendung (*PilotServer*).

Die dienstgütespezifischen Anforderungen beschreiben Forderungen, die von einer Middleware erfüllt sein müssen, damit Dienstgüte ermöglicht werden kann. Wie in den vorangegangen Kapiteln beschrieben, besteht z.B. die Notwendigkeit, Dienstgüte zu reservieren oder die momentane Ressourcensituation zu kennen. Diese Funktionalitäten müssen einer Middleware hinzugefügt werden. Im Einzelnen sind das:

- Reservierung von Dienstgüte auf einem Leitweg
- Freigabe von Dienstgüte auf einem Leitweg
- Leitwegbestimmung anhand bestimmter Dienstgütemerkmale bei Multi-Hop Anwendungen (*Ressource Routing*)
- Verwalten von unterschiedlichen Dienstgüte-Prioritäten
- Feststellung der momentanen Ressourcensituation auf einem gewählten Leitweg (*Dienstgüteüberwachung*)
- Kontrolle von Dienstgütedatenströmen
- Anpassung der Nutzdaten an eine neue Ressourcensituation

Einige dieser Anforderungen müssen oder können nur vom Basisdienst erfüllt werden, andere von den entsprechenden Schichten darüber. In dieser Arbeit werden nur Dienstgüteanforderungen auf höheren Schichten betrachtet. Da wir nur Single-Hop Kommunikation betrachten, entfallen in unserem Fall bestimmte Funktionalitäten wie Routing oder eine komplexe Ressourcenreservierung auf einem Leitweg.

4.3 Dienstgüte-Bereitstellung

In diesem Kapitel wird die Umsetzung der Dienstgütespezifikation von Kapitel 3.5 in eine konkrete Ressourcenbelegung/-verteilung beschrieben. Der wichtigste Punkt ist das *Dienstgüte-Mapping*.

4.3.1 Dienstgüte-Mapping

Das *Dienstgüte-Mapping* beschreibt die Umsetzung der Dienstgüteklassen aus Kapitel 3.5 in eine Ressourcenbelegung. Anhand dieser Umsetzung werden je nach Klasse entsprechende Ressourcen reserviert. Tabelle 4-1 zeigt das Dienstgütemapping für die Videoanwendung. Den vier identifizierten Klassen wird je eine minimale und eine maximale Dienstgüte zugewiesen, die speziell auf die Anforderungen des Benutzers an die Anwendung angepasst sind. Die erste Umsetzung erfolgt auf Bilder/Sekunde und auf Bildqualität, da wir uns noch auf der Ebene des Benutzers befinden. Durch das Mapping ist auch implizit schon ein Feedback an den Benutzer

gegeben, da wir nun genau das Dienstgüteintervall für eine Dienstgüteklasse kennen. Sinkt die Dienstgüte unter die minimale Schranke, so wird ihm die gewährte Dienstgüte durch die Feedback-Klasse *Rot* mitgeteilt, sonst entsprechend Abbildung 3-6.

Tabelle 4-1: QoS-Mapping Videoübertragung

Dienstgüteklasse	minmale Dienstgüte (Bilder/Sekunde, JpegQualität)[a]	optimale Dienstgüte (Bilder/Sekunde, JpegQualität)[a]
search	15/50	20/75
watch	15/25	25/50
panorama	5/75	10/90
solo	0/0	0/0

a. ([1/s],[%]), 100% = keine Kompression (beste Qualität)

Abbildung 4-2 zeigt die aus dem Mapping resultierende Reservierung der Dienstgüte. Für eine aktive Klassen wird eine Reservierungsanfrage mit den entsprechenden minimalen und optimalen Anforderungen getätigt. Im Falle der Videoanwendung setzt sich die Anforderung aus einer benötigten Bandbreite, einem Delay und aus dem Jitter zusammen. Einzige Ausnahme bildet die Dienstgüteklasse *solo*, bei der keine Ressourcen reserviert werden müssen, da keine Videoübertragung stattfindet. Die genaue Ermittlung der Werte erfolgt mit Hilfe von Tabelle 3-7.

Tabelle 4-2: Reservierungen der Videoübertragung[a]

Dienstgüteklasse	minimale Reservierung (Datenvolumen, Delay, Jitter)[b]	optimale Reservierung (Datenvolumen, Delay, Jitter)[b]
search	840, 1000, 66	1600,1000, 50
watch	600, 1000, 66	1400,1000, 40
panorama	400,1000, 200	1440,1000, 100
solo	0,0,0	0,0,0

a. Erfahrungswerte, genaue Reservierungsparameter können auch exakt berechnet werden.
b. in ([kBit],[ms],[ms])

Analog zum Mapping der Video-Dienstgüteklassen gibt es auch ein Mapping für die Steuerungsanwendung. Für die beiden Klassen *easy* und *hard* werden gemäß den Benutzervorgaben Sendeverzögerungen festgelegt (Tabelle 4-3).

Tabelle 4-3: QoS-Mapping Steuerung

Dienstgüteklasse	maximale Dienstgüte (Delay)[a]	optimale Dienstgüte (Delay)[a]
easy	300	150
hard	150	20

a. in [ms]

Tabelle 4-4: Reservierungen der Steuerung

Dienstgüteklasse	minimale Reservierung (Datenvolumen, Delay)[a]	optimale Reservierung (Datenvolumen, Delay)[a]
easy	2^b, 300	2^b , 150
hard	2^b , 150	2^b , 50

a. in ([kBit/s],[ms])
b. es werden nur 1200 Bit benötigt. Reservierungen erfolgen in 1kBit-Schritten

Die Umsetzung in eine konkrete Reservierung zeigt Tabelle 4-4. Im Gegensatz zur Videoanwendung findet keine Festlegung des Jitter statt. Da das Datenvolumen pro Steuerungssignal mit $3 \cdot 8\text{Bit} = 24\text{Bit}$ sehr gering ist, wird immer die gleiche Bandbreite reserviert, unabhängig von der tatsächlichen Anzahl der gesendeten Signale. Im optimalen Fall werden der Servo und die Motoren alle 20 ms angesteuert, was einer Bandbreite von 1200 Bit/s entspricht. Die tatsächliche Reservierung beträgt allerdings 2kBit/s, da Reservierungen nur in 1kBit Schritten getätigt werden können.

Die Ansteuerung der Servo/Motoren-Anordnung bzw. das Senden der aktuellen Steuerungswerte könnte eigentlich immer alle 20 ms stattfinden, da keine Reservierung einer Periode getätigt wird und die Sendeverzögerung offensichtlich nichts mit der Periode zu tun hat. Das Problem bei der Steuerungsanwendung besteht aus dem geringen Datenvolumen. Um den Zeppelin möglichst fein steuern zu können, würde es durchaus Sinn machen, in möglichst kurzen Abständen die nötigen Informationen zu senden. Hat man dann wie in unserem Fall eine statistische Reservierung der Sendeverzögerung getätigt, so sieht es auf den Blick nach einer optimalen Lösung aus. Nach 100 ms (zum Beispiel) käme so beim Empfänger, dem Zeppelin, alle 20 ms ein neues Steuerungssignal an. Und nun kommt das geringe Datenvolumen ins Spiel. Nehmen wir an, das Medium bietet uns genug Zeitslots, um alle 10 ms ein Packet senden zu können. Das wären doppelt so viele, wie wir benötigen würden, nämlich 100. Vernachlässigen wir sämtliche Verwaltungsnachrichten oder Interframespacings. Unsere Reservierung des Delays würde mindestens alle 100 ms einen Zeitschlitz reservieren, da ja sichergestellt werden muss, dass zu

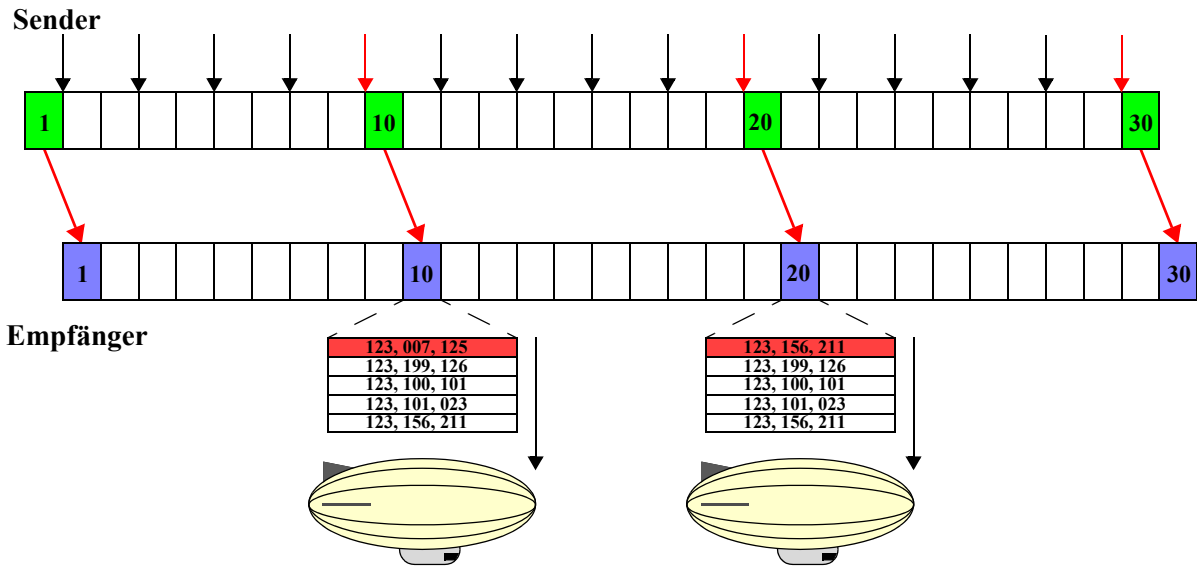


Abb. 4-13: Sendeperiode 20ms

jedem Zeitpunkt, zu dem die Anwendung sendet, die Reservierung statistisch gesehen eingehalten wird. Die reine Sendezeit bei einer Single-Hop Topologie ist zu vernachlässigen. Bei einer Übertragungsrate von 2Mbit/s ergibt sich pro Zeitslot ein Übertragungsvolumen von 20kBit. D.h. ohne es zu wollen hat unsere Reservierung eine Bandbreite von $10 \cdot 20\text{kBit} = 200\text{kBit}$ beschlagnahmt, obwohl nur 2kBit benötigt werden (ob die Bandbreite verschwendet ist oder nicht, d.h ob es möglich ist, mehrere Pakete von verschiedenen Reservierungen/Anwendungen in einen Zeitschlitz zu multiplexen hängt davon ab, ob die Zeitschlitze für eine Verbindung oder eine Anwendung reserviert worden sind). Im Umkehrschluss heißt das, dass auf gar keinen Fall noch zusätzliche Zeitslots reserviert werden. Abbildung 4-13 zeigt das eben beschriebene Szenario.

Unabhängig ob Zeitschlitze für Links oder Anwendungen reserviert werden, macht es nur Sinn, alle verfügbaren Daten einer Anwendung in einen Slot zu packen, falls dieser noch nicht vollständig belegt ist. Als Konsequenz dazu werden die kleinen periodischen Pakete alle in einen Slot gepackt. Das Widerspricht in keinster Weise unserer Reservierung der Sendeverzögerung. Nach maximal 100 ms empfängt der Server das Paket und arbeitet die empfangenen Steuerwerte der Reihe nach ab. Im Endeffekt werden die Servos bis das nächste Paket empfangen wird, auf den aktuellsten Wert eingestellt, da die älteren sofort von den neueren Paketen des Eingangspuffers überschrieben werden. Es findet also eine Verschwendung von Ressourcen statt.

Sorgt man jedoch dafür, dass wie in Abbildung 4-14 zu sehen ist die Daten nur alle 100 ms versendet werden, so ist das Resultat das gleiche. Aus diesem Grund wird als Sendeperiode der Steuerungsdaten das aktuell gewährte Delay gewählt. Im Normalfall sind die Größen Delay und Periode nicht miteinander korreliert. Das Gleichsetzen von Delay und Periode ist an dieser Stelle eine willkürliche Festlegung, die meiner Meinung nach sinnvoll ist.

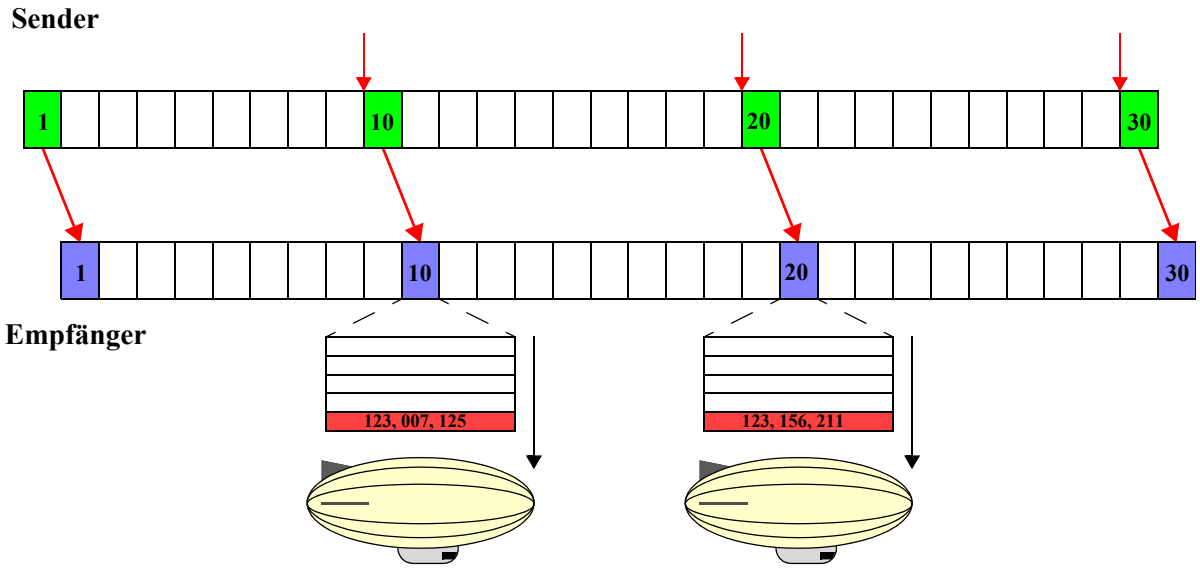


Abb. 4-14: Sendeperiode 100ms

Für die Übertragung der Luftschiffparameter werden keine unterschiedlichen Anwendungsfälle identifiziert. Die benötigte Bandbreite pro Signal beträgt im optimalen Fall $4 \cdot 8\text{Bit} = 32\text{Bit}$, da neben den drei Werten der Servo/Motoren zusätzlich die aktuelle Akkuspannung übertragen wird und je 256 Werte möglich sind. Auch hier gilt das gleiche wie bei der Videoübertragung.

Die Daten sollen in einem vorgegebenen Zeitrahmen beim Empfänger ankommen. Setzt man auch hier die Periode, mit der gesendet wird, und das reservierte Delay gleich, so ergibt sich für die gesamte Bandbreite $5 \cdot 32\text{Bit} = 160\text{Bit}$ und somit 1kBit für die Reservierung.

Tabelle 4-5: QoS-Mapping Übertragung der Luftschiffparameter

Dienstgüteklasse	maximale Dienstgüte (Delay)[a]	optimale Dienstgüte (Delay)[a]
default	1000	200

a. in [ms]

Tabelle 4-6: Reservierungen der Übertragung der Luftschiffparameter

Dienstgüteklasse	minimale Reservierung (Datenvolumen, Delay)[a]	optimale Reservierung (Datenvolumen, Delay)[a]
default	1[b], 1000	1[b], 200

a. in ([kBit/s],[ms])
b. es werden nur 160 Bit benötigt. Reservierungen erfolgen in 1kBit-Schritten

4.3.2 Zugangstest und Reservierungsprotokoll

Der Zugangtest und das Reservierungsprotokoll stellen zwei wichtige Dienstgütefunktionalitäten da. Diese müssen beide von dem darunterliegenden WLAN-Treiber zur Verfügung gestellt werden (Single-Hop). Im Folgenden wird davon ausgegangen, dass eine solche Unterstützung vorliegt.

4.4 Dienstgüte-Kontrolle

Die Dienstgütekontrolle befasst sich mit allen kurzfristigen Maßnahmen, die nötig sind, um Kommunikation in der geforderten Dienstgüte bereitzustellen. Diese Aufgabe muss zum einen die maßgeschneiderte Middleware übernehmen, zum anderen auch die Basistechnologie. Auch hier ist die Entwicklung der Basistechnologie noch nicht so weit fortgeschritten, dass man geeignete Mikroprotokolle auf ihr aufsetzen kann, die z.B. eine adäquate *Verkehrsplanung* (Scheduling der Pakete) oder *Vehrkehrsformung* implementieren. Je nachdem ob die Basistechnologie Reservierungen für einen Link oder für eine Anwendung tätigt, muss ein anders Mikroprotokoll erstellt werden.

4.5 Dienstgüte-Management

Das Dienstgütemanagement ist eine weitere wichtige Funktionalität jeder Dienstgüte-Middleware. Mikroprotokolle für die Überwachung (*monitoring*) und für die Skalierung (*scaling*) wurden spezifiziert und werden in Kapitel 5 näher beschrieben. Die Herabsetzung der Dienstgüte (*degradation*) erfolgt bei dieser Middleware manuell, das heißt der Benutzer muss von Hand eine neue Dienstgüteklasse setzen, falls die aktuelle nicht mehr befriedigt werden kann.

4.6 Architektur

Da wir ein maßgeschneidertes Kommunikationssystem betrachten, gibt es client- und serverseitig verschiedene Middlewares, die auf die jeweiligen Bedürfnisse der darauf aufsetzenden Applikation zugeschnitten ist.

Abbildung 4-1 zeigt die maßgeschneiderte Middleware des Servers. Die Middleware ist zweigeteilt. Zum einen in eine anwendungsspezifische (*VideoServerAppMw* und *PilotServerAppMw*) und zum anderen in eine dienstgütespezifische Middleware (*QosMw*). Die jeweiligen spezifischen Middlewareblöcke kommunizieren über feste Schnittstellen miteinander bzw. mit ihrer Applikation oder der Basistechnologie.

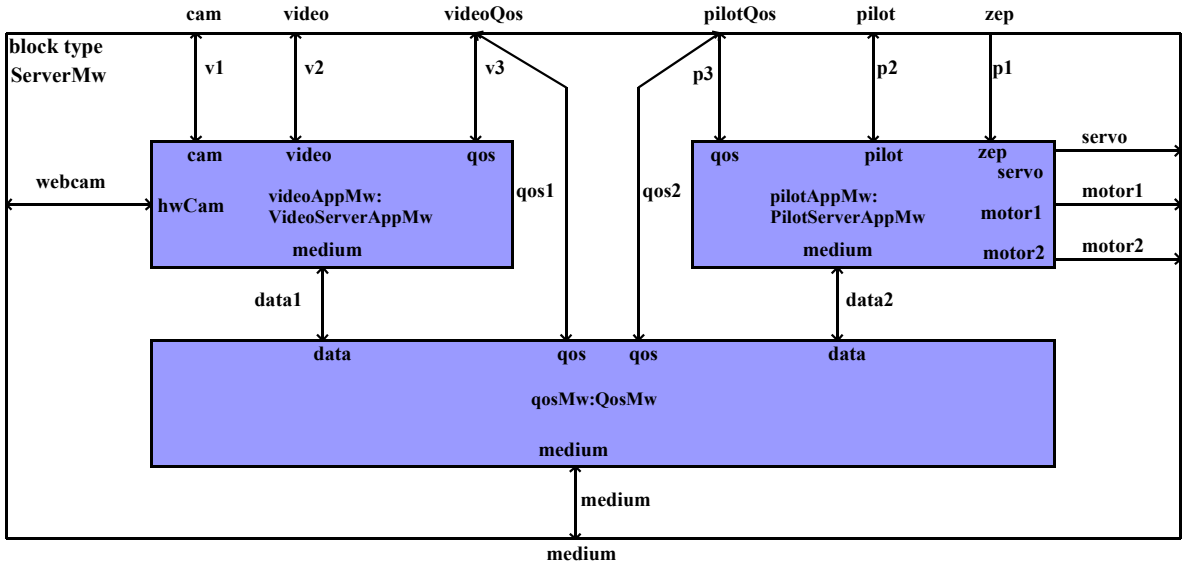


Abb. 4-1: Die Middleware des Servers

Analog zur Struktur der Middleware auf der Serverseite (Abb. 4-2) ergibt sich die Struktur auf Clientseite. Auch hier sitzt die anwendungsspezifische Middleware (*VideoClientAppMw* und *PilotClientAppMw*) direkt auf der dienstgütespezifischen (*QosMw*). Ein Unterschied besteht jedoch in den Schnittstellen zur Umgebung. Im Gegensatz zur Ansteuerung der Kamera und des Servos bzw. der Motoren des Zeppelins gibt es auf Clientseite keine zusätzliche Hardware, die kontrolliert werden muss. Die nötigen Steuerdaten liefert ausschließlich die Benutzeroberfläche. Da wir in Kapitel 3.5 drei Dienstgüteanwendungen identifiziert haben (Videoübertragung und Übertragung der Steuerdaten sowie der aktuellen Luftschiffparameter) besitzt die Videoapplikation auf Clientseite keine Schnittstelle zur Dienstgüte-Middleware. Reservierungen finden werden immer in Richtung der Übertragung getätigt, unabhängig davon, wer genau die Dienstgüteklasse setzt. Eine genaue Beschreibung der obersten, anwendungsspezifischen Schicht erfolgt im nächsten Kapitel.

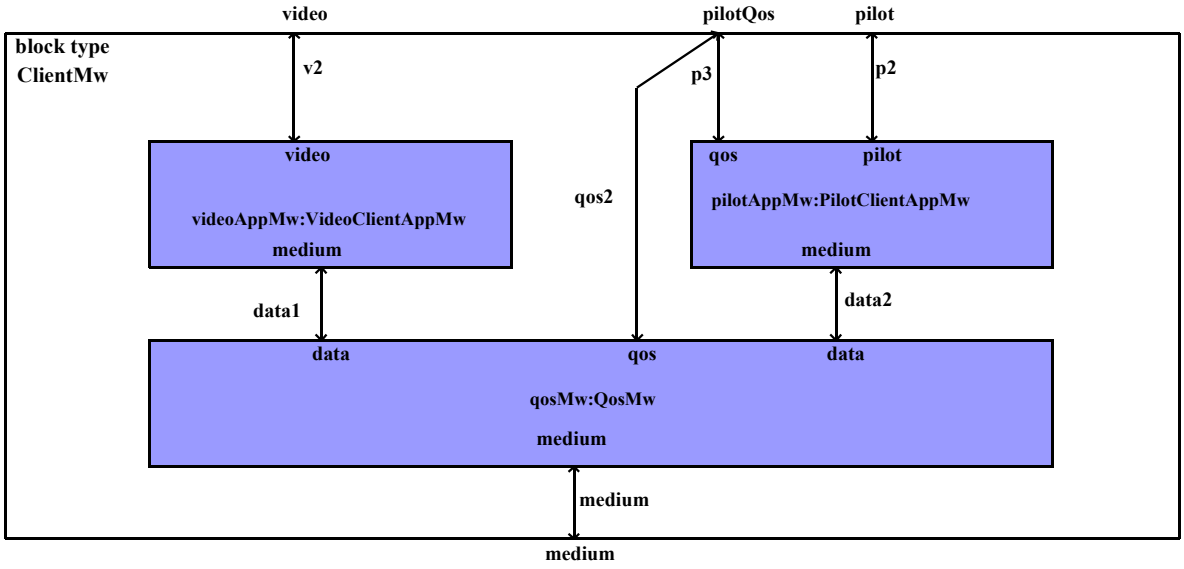


Abb. 4-2: Die Middleware des Clients

4.7 Applikationsspezifische Middleware

In diesem Kapitel werden die vier applikationsspezifischen Middlewares genauer betrachtet und erläutert. Sie sind alle speziell auf die jeweilige Anwendung zugeschnitten und somit im Gesamten nicht wiederverwendbar. Der Vorteil einer solchen Middleware ist eine geringerer Ressourcenbedarf, da sie ausschließlich mit genau den Funktionalitäten ausgestattet ist, die benötigt werden.

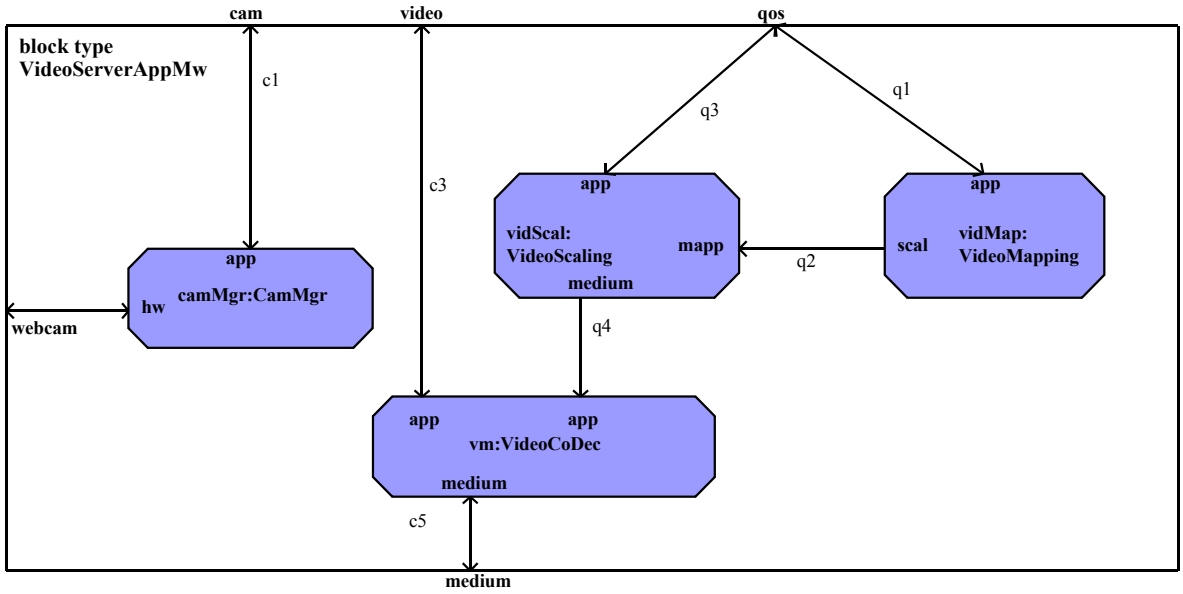


Abb. 4-3: Die anwendungsspezifische Middleware des Video-Servers

Die Middleware *VideoServerAppMw* sitzt auf der Serverseite unter der VideoServer-Applikation (Abb. 4-3). Sie kommuniziert über ihre fünf Schnittstellen (*cam, video, qos, webcam, medium*) mit der Umgebung. Der *CamMgr* dient zur Abstraktion von der eigentlichen Hardware der Webcam. Über die Schnittstelle *cam* nimmt dieser Steuerungssignale entgegen und liefert Bilder mit der geforderten Bildrate und Qualität an die Anwendung. Die Anwendung kann so komplett von der Hardware der Webcam abstrahieren und diese über eine einheitliche Schnittstelle ansprechen.

Über die Schnittstelle *video* kommuniziert die VideoServer-Applikation aus ihrer Sicht direkt mit ihrem Pendant auf Clientseite, der VideoClient-Applikation. Die dazwischenliegende Middleware ist vollkommen transparent für die Anwendungen. Dazu werden die verschiedenen Signale in einer Prozessinstanz von *VideoCoDec* zu einem einzigen Octet String encodiert und somit serialisiert und zum Versenden an die darunterliegende Schicht über die Schnittstelle *medium* weitergereicht. Die Instanz übernimmt auch das Decodieren. Dadurch wird eine virtuelle Kommunikation zwischen zwei Anwendungen auf eine Kommunikation zwischen zwei Middlewares abgebildet und somit eine weitere Abstraktionsebene eingeführt.

Die Prozessinstanzen von *VideoScaling* und *VideoMapping* sorgen für ein korrektes (Um-)Setzen der Dienstgüteklasse und für ein Skalieren der Daten auf eine gewährte Dienstgüte. Sie werden über die Schnittstelle *qos* der applikationsspezifischen Middleware angesprochen. Alle Signale, die Nutzdaten (z.B. Bilder oder Steuerungswerte) für den Client enthalten und somit skaliert werden müssen, werden über die Schnittstelle *qos* durch *VideoScaling* an *VideoCoDec* durchgereicht.

Die applikationsspezifische Middleware der Pilotensteuerung sieht ähnlich aus (Abb. 4-4). Analog zur VideoServer-Middleware gibt es auch hier eine Instanz eines Signalmultiplexers und -codierers (*PilotCoDec*), die die Signale der Anwendung verarbeitet. Auch sie verarbeitet nur die anwendungsspezifischen Signale und ist somit gleichermaßen maßgeschneidert. *PilotServerMapping* und *PilotServerScaling* sorgen für ein korrektes Setzen der Dienstgüteklasse wie in Tabelle 4-3 angegeben und für ein angebrachtes Skalieren der Daten, in diesem Fall ein Anpassen der Sendeverzögerung der Luftschiffparameter.

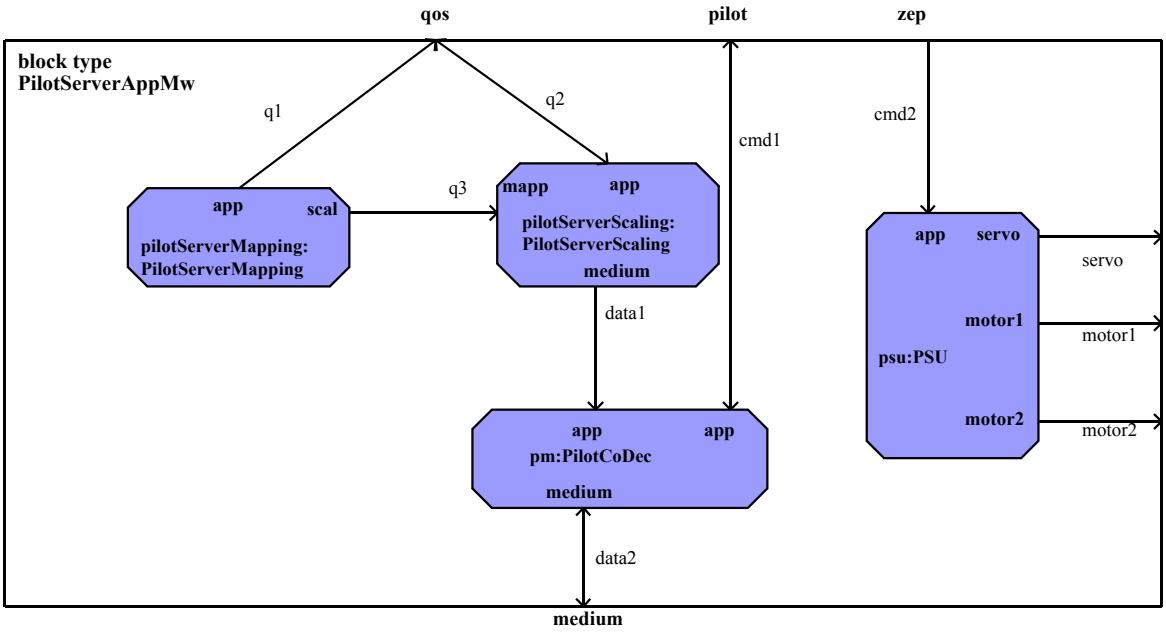


Abb. 4-4: Die anwendungsspezifische Middleware des Steuerungs-Servers

Neu in der applikationsspezifischen Middleware des Servers ist die Komponente *PSU (Periodical System Update)*. Sie stellt den angesprochenen *fail-operational* Zustand im Falle eines Versagens einer der Komponenten her, die nötig sind, um den Zeppelin zu steuern. Sie ist direkt an den Servo und die Motoren des Luftschiffes angeschlossen, da sie sie ohne Umwege über die Anwendung kontrollieren muss. Diese Komponente muss in regelmäßigen Abständen angesprochen werden, damit der interne Watchdog zurückgesetzt wird. Desweiteren spricht diese den Servo und die Motoren mit einer Periode von 20 ms an, damit z.B. der Servo nicht in seinen sleep-Modus fällt. Daher auch der Name dieser Komponente.

Im Gegensatz zur Middleware der VideoServer-Applikation ist die anwendungsspezifische Middleware des VideoClients bei weitem nicht so umfangreich (Abb. 4-5). Sie besteht lediglich aus einer Komponente zur De- und Enkodierung der Kommunikation zwischen Server und Client. Weitere Komponenten werden nicht benötigt, da diese Anwendung nur Daten entgegennimmt. Für die sporadisch gesendeten Benutzerkommandos an den Server muss keine Dienstgüte reserviert werden, da die Datenmenge zu gering ist.

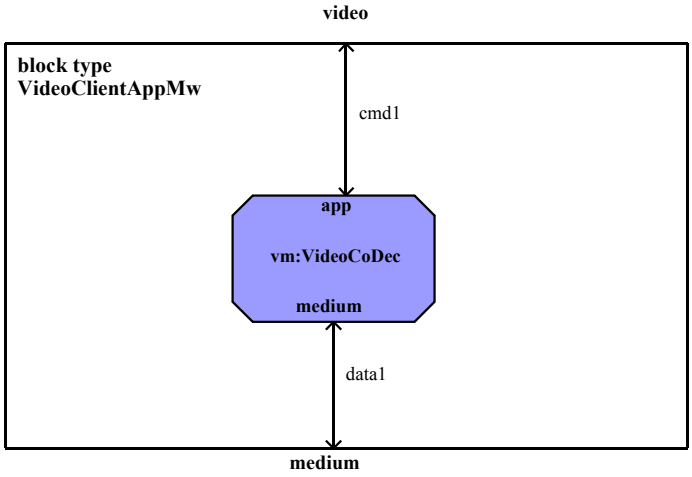


Abb. 4-5: Die anwendungsspezifische Middleware des Video-Clients

Abbildung 4-6 zeigt die applikationsspezifischen Komponenten, mit denen der Steuerungs-Client direkt kommuniziert. Diese Komponenten sind in einem Blocktyp *PilotClientAppMw* zusammengefasst. Die Middleware auf Clientseite unterscheidet sich nur geringfügig von der auf Serverseite. Der einzige Unterschied besteht aus einem leicht veränderten *Mapping* und *Scaling*, da die Steuerparameter ein Byte weniger als die Luftschiffparameter betragen und aus dem Ersatz der Komponente *PSU* durch einen entsprechenden *PSUTrigger*, der in regelmäßigen Abständen ein Heartbeat-Signal generiert, welches der Instanz von *PSU* entsprechend mitgeteilt wird. In unserem Fall ist das Heartbeat-Signal ein gewöhnliches Steuersignal, da für ein solches Signal bereits Dienstgüte reserviert ist und wir daher sicher sein können, dass es ankommt (gemäß den statistischen Garantien). Desweiteren sendet *PSUTrigger* sonst nur das Steuerungssignal, falls sich die Parameter geändert haben. Dies dient dazu, Netzlast gering zu halten, auch wenn das Datenvolumen nicht besonders hoch ist.

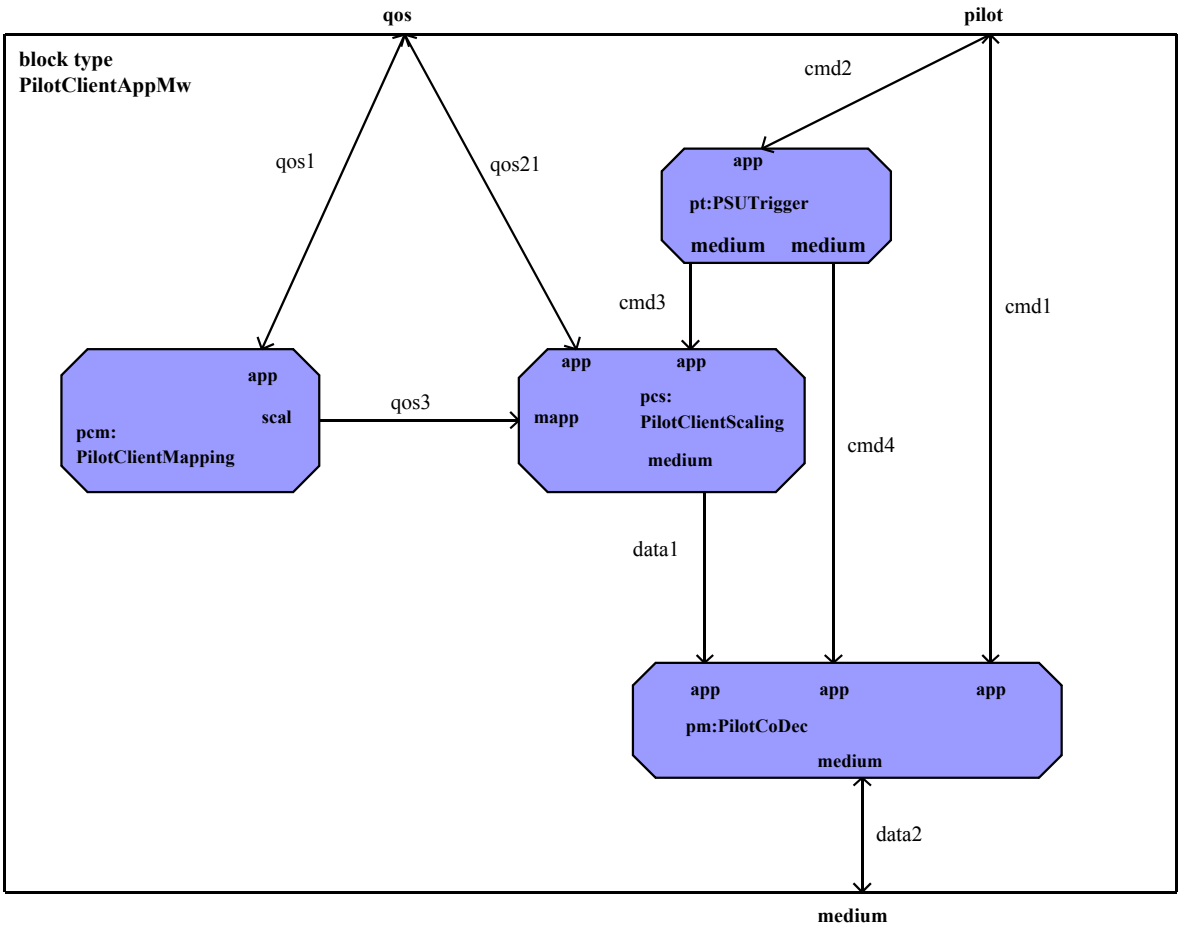


Abb. 4-6: Die anwendungsspezifische Middleware des Steuerungs-Clients

4.8 Dienstgütespezifische Middleware

Die dienstgütespezifische (oder applikationsunabhängige) Middleware (Abb. 4-7) realisiert die applikationsunabhängigen Funktionalitäten unserer Middleware. Sie baut auf einer vorhandenen Basistechnologie auf, die geeignete Schnittstellen bereitstellt, um auf deren Dienstgütefunktionalität zuzugreifen. Die Middleware gliedert sich in zwei Hauptkomponenten. Eine Komponente namens *QosReservationLinkMgr*, die die Reservierung tätigt und die vorhandene Dienstgüte auf einem Link an die einzelnen Applikation gemäß ihrer Priorität aufteilt und einen Block *QosMonitoringTransfer*, welcher die Statusnachrichten der Basistechnologie auswertet und daraus eine Tabelle mit den aktuellen Verbindungen und deren Qualität aufbaut. Desweiteren werden über die Schnittstelle *data* Nutzdaten von der anwendungsspezifischen Middleware und der Basistechnologie entgegengenommen und entsprechend der Adressinformation versendet.

Abbildung 4-8 zeigt die genaue Struktur des Blockes *QosReservationLinkMgr*. Er besteht aus den Prozesstypen *QosReservationMgr* und *QosLinkMgr*. Über die Schnittstelle *qos* werden vom *QosReservationMgr* Reservierungen entgegengenommen und falls diese erfolgreich vom Basisdienst getätigt werden können wird ein entsprechender *QosLinkMgr* dynamisch neu erstellt oder über die neue Reservierung informiert. Dieser berechnet mit Hilfe der Prioritäten der Reservierungen und der aktuellen Dienstgütesituation die Zuordnung der Ressourcen zu den einzelnen Applikationen pro Link.

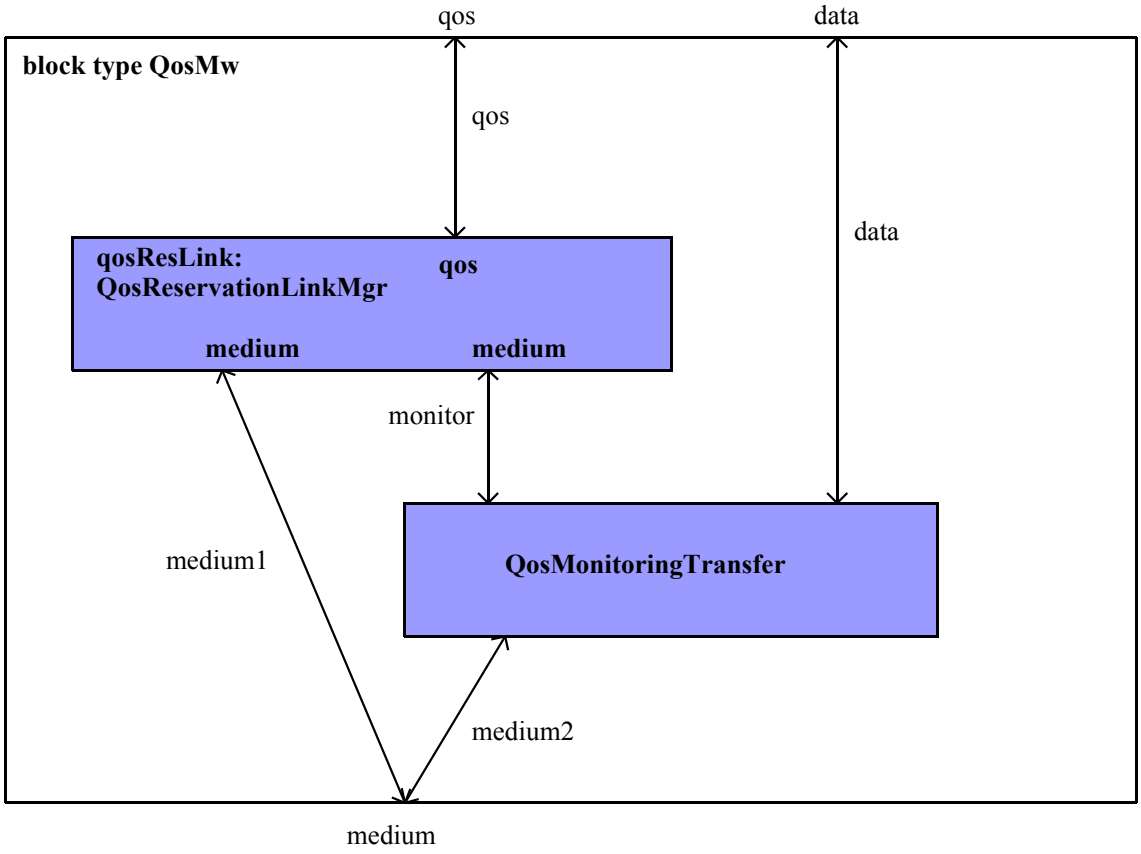


Abb. 4-7: Die dienstgütespezifische Middleware

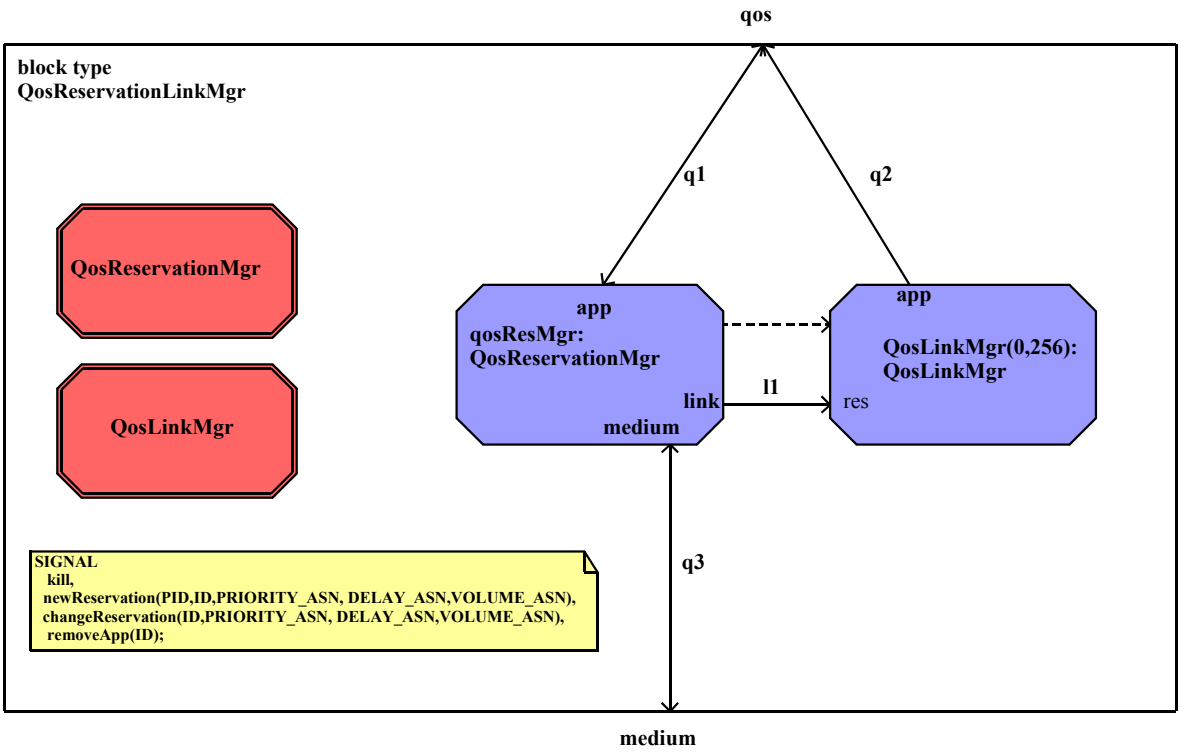


Abb. 4-8: Der Blocktyp „QosReservationMgr“ der dienstgütespezifischen Middleware

Folgende Abbildung zeigt den zweiten Hauptteil der dienstgütespezifischen Middleware. Die Komponente *QosMonitoring* überwacht sämtliche Verbindungen des Hosts. Falls auf einer dieser Verbindungen Dienstgüte reserviert worden ist, erfolgt bei jeder Änderung der Verbindungsqualität ein Feedback an die darüberliegende Komponente über die Schnittstelle *monitor*. Der *TransferMgr* leitet die adressierten Pakete an den Basisdienst weiter. Je nachdem ob dieser die verfügbaren Zeitschlitze des Medium direkt für eine bestimmte Anwendung oder eine Verbindung reserviert, müssen die Pakete geeignet gescheduled bzw. gemultiplext werden. Ferner nimmt der *QosTransferMgr* auch die Pakete vom Basisdienst entgegen und verteilt diese anhand der Adressierung (jede Applikation bekommt einen eindeutigen Port zugeordnet) an die entsprechenden Applikationen.

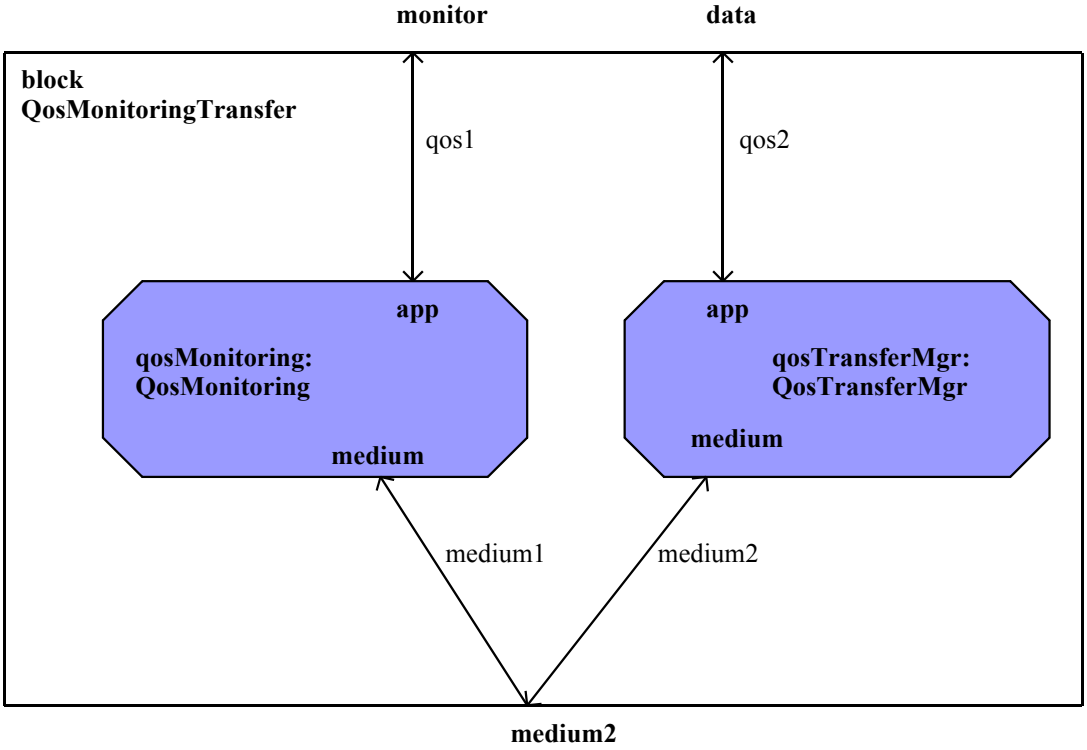


Abb. 4-9: Der Block „QosMonitoringTransfer“ der dienstgütespezifischen Middleware

4.9 Zusammenfassung

Dieses Kapitel beschäftigte sich ausschließlich mit der Middleware. Es wurden allgemeine und auch anwendungsspezifische Funktionalitäten vorgestellt. Die Anforderungen der beiden Applikationen wurden beschrieben und es wurde ausführlich auf das Dienstgüte-Mapping eingegangen. Für jede Reservierung gibt es ein eigenes Mapping, das genau spezifiziert ist. Abschließend wurde die Struktur der Middleware vorgestellt. Sie ist geschichtet und nach anwendungsspezifischen und anwendungsunabhängigen (dienstgütespezifischen) Gesichtspunkten unterteilt. Ferner wurden den einzelnen Komponenten der Middleware kurz beschrieben. Im nächsten Kapitel wird auf die Funktionsweise bzw. Funktionalität der einzelnen Komponenten genauer eingegangen.

5 Mikroprotokolle & Design-Patterns

In diesem Kapitel werden die in Kapitel 4 eingeführten Komponenten genauer erläutert. Es werden die neu entwickelten Mikroprotokolle und Design Patterns vorgestellt und ihre Anwendung demonstriert. Wie das letzte Kapitel, gliedert sich dieses in zwei Hauptteile. Der erste beschäftigt sich mit den applikationsspezifischen Komponenten. Dieser Teil wird von den Design Patterns dominiert, da sie eine generische Lösung für ein spezifisches Problem bieten, ohne eine genaue Spezifikation zu geben.

Der zweite Teil beschäftigt sich ausschließlich mit Mikroprotokollen. Es werden erste Dienstgüte-Mikroprotokolle vorgestellt die applikationsunabhängig in jeder Dienstgüte-Middleware zum Einsatz kommen können.

In diesem Kapitel werden nur Auszüge aus den Prozessen und Prozesstypen der Middleware wiedergegeben. Einen umfangreichen Einblick in das gesamte funktionale Verhalten gewährt der Anhang D mit dem kompletten Entwurf des Systems und der Mikroprotokolle.

5.1 Mikroprotokolle und Design Patterns der applikationsspezifischen Middleware

Die applikationsspezifische Middleware wurde in Kapitel 4.7 vorgestellt. Hier wird nun im Detail auf die einzelnen Komponenten eingegangen und ihr Verhalten genauer beschrieben. Im Laufe der Entwicklung dieser Middleware sind ferner einige wiederkehrende Designprobleme in Form von Design Patterns beschrieben worden.

5.1.1 Die Codierer und Decodierer *VideoCoDec* und *PilotCoDec*

5.1.1.1 Designproblem

Damit zwei Applikationen über ein Medium, welches nur mit Byteströmen umgehen kann, kommunizieren können, muss die gesamte Kommunikation zwischen ihnen serialisiert, also in eine Bytefolge konvertiert werden. Dazu ist es nötig, alle Signale inklusive Parametern zuvor in eine Datenstruktur zu packen, die dann z.B. zu einem BIT STRING oder OCTET STRING encodiert werden kann.

Die beiden Prozesstypen *VideoCoDec* und *PilotCoDec* bilden Signale und Parameter auf eine ASN.1 Datenstruktur ab und en- bzw. decodieren sie. Man kann die DeCodierer (kurz für Decodierer und Codierer) als neue Abstraktionsebene, also als einen *Service Access Point* sehen. Es wird vollständig von den Signalen der oberen Schichten abstrahiert. Diese können in einer Datenstruktur z.B. als Signalparameter einer tieferen Schicht zur Verfügung gestellt werden.

5.1.1.2 Design-Pattern *SimpleSignalCoDecArchitecture*

Das neu erstellte Design-Pattern *SimpleSignalCoDecArchitecture* gehört zur Gruppe der *Architekturmuster*. Es befasst sich mit der Struktur der Realisierung einer einfachen CoDec-Multiplexer Anordnung. Da es sich um ein Architekturmuster handelt, wird kein Verhalten spezifiziert. Dieses wird an späterer Stelle geeignet hinzugefügt.

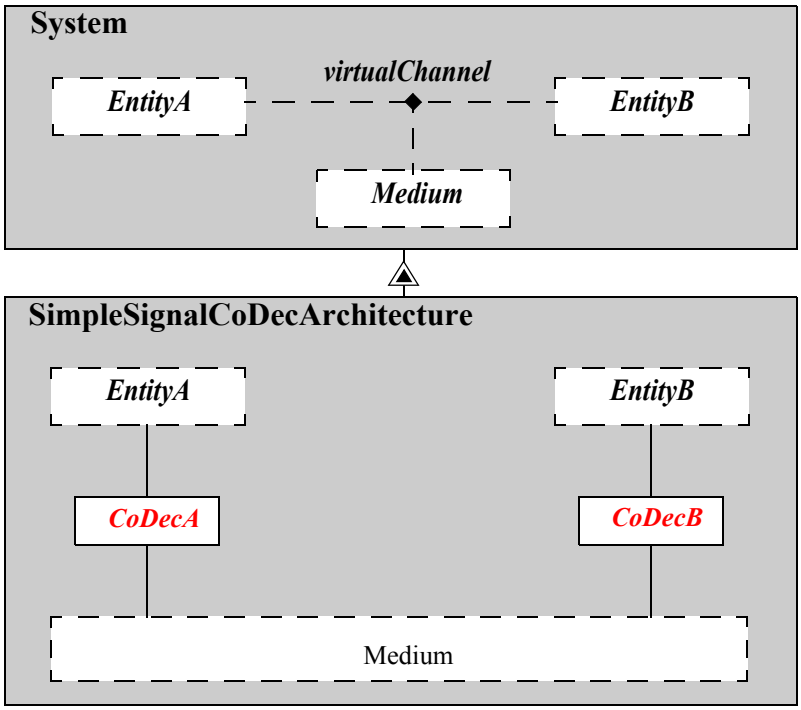


Abb. 5-1: Struktur des Patterns

Vor der Anwendung des Patterns besteht unser System aus je zwei Kommunikationsentitäten, die über einen virtuellen Kanal miteinander kommunizieren, der von einem *Medium* beschrieben wird. Nach der Anwendung wird unser System dahingehend verfeinert, dass nun für jede Entität eine neue Komponente *CoDec* eingeführt wird, die über ein abstraktes Medium die beiden Entitäten verbindet. Abbildung 5-1 zeigt die Struktur des Patterns.

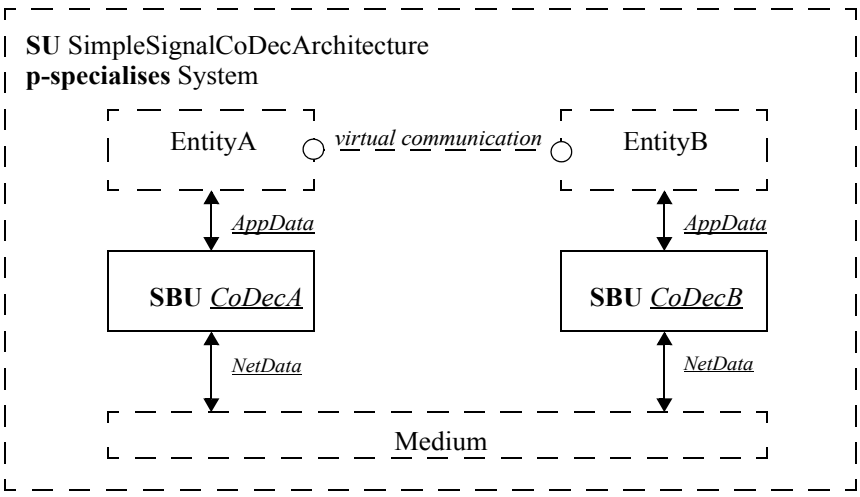


Abb. 5-2: SDL-Fragment von *SimpleSignalCoDecArchitecture*

Wie in [2], [14] beschrieben handelt es sich bei *SignalCoDecArchitecture* (Abb. 5-2) um eine **S**tructural **U**nit. Sie kann ein Block(typ) oder ein System(typ) sein. Die *CoDec* Komponenten werden in einem eigenen Block dargestellt, der weiter verfeinert werden muss. Sie kommunizieren mit den jeweiligen Entitäten über die Signale *AppData* und mit dem Medium über *NetData*.

Folgende Abbildung zeigt die Verfeinerung der *CoDecs*. Die **S**tructural **B**lock **U**nit *CoDec* wird zu einem Prozess verfeinert, der mit einer initialen Starttransition spezifiziert wird. Dadurch wird zwar kein Verhalten beschrieben, das Design-Pattern ist aber aus SDL-Sicht vollständig.

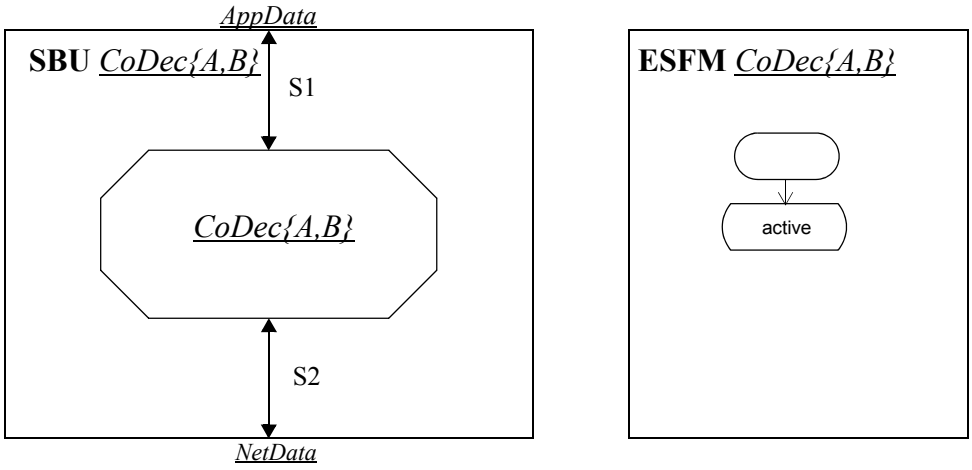


Abb. 5-3: Verfeinerung von *CoDec*

Die Anwendung dieses Architekturmusters auf unser System ergibt die vorliegende Struktur der anwendungsspezifischen Middlewares. Wie in Abbildung 5-4 gezeigt wird unterhalb der Videoanwendung je eine Instanz eines speziell auf sie zugeschnittenen *CoDecs* installiert. Auf Client und Serverseite wurde der gleiche *CoDec* verwendet. Das widerspricht ein wenig der Beschreibung des Musters, aber diese Entscheidung wurde getroffen, damit man die erstellten Komponenten wiederverwenden kann. Analog dazu ergibt sich die Struktur der Steuerungsanwendung, die jedoch an dieser Stelle nicht beschrieben wird (vgl. Kap. 4.7). Das genaue Verhalten der Prozesstypen muss noch spezifiziert werden.

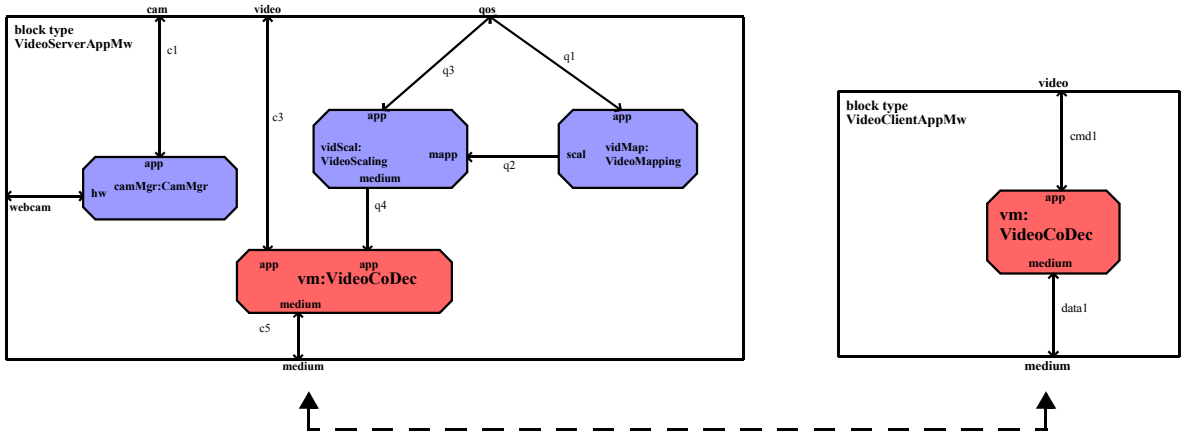


Abb. 5-4: Anwendung des SimpleCoDecArchitecture Pattern

5.1.1.3 Design Pattern *SDLSignal2ASN.1*

Wie erwähnt besteht die Notwendigkeit, die verschiedenen SDL-Signale auf eine einzige Datenstruktur abzubilden. In unserem Fall werden die benötigten Strukturen (je eine pro Anwendung) in ASN.1 [6] spezifiziert. Das *SDLSignal2ASN.1* Pattern hilft bei der Erstellung solcher Strukturen. Abbildung 5-5 zeigt die ASN-Modulstruktur, die das Pattern vorsieht.

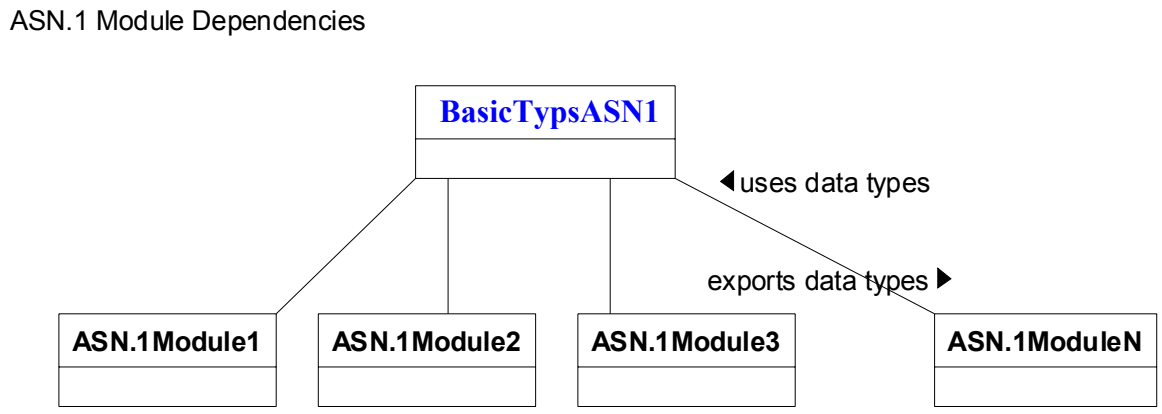


Abb. 5-5: ASN.1 Modulabhängigkeiten

In einem Modul namens *BasicTypesASN1* werden die benötigten Basisdatentypen erstellt. Der Vorteil hierbei ist, dass durch die Verwendung von Synonymen z.B. eine spätere Änderung eines Stringtyps von *IA5String* auf *BIT STRING* nur an einer Stelle zu erfolgen hat. Das Modul besteht aus drei Teilen und einer *export*-Anweisung. Der erste Teil, die *ASN1SimpleDataTypes*, müssen genau einmal im Modul vorhanden sein. Sie beschreiben einfache Datentypen wie Integer oder Real. Der zweite Teil des Moduls behandelt eventuell vorhandene *Syntypes* in SDL, da wir SDL-Signale kapseln wollen. Hier werden zur besseren Verfolgbarkeit die gleichen Syntypes angelegt, wie in SDL, mit dem Unterschied, dass sie einen ASN.1 Typ oder eine ASN.1 Struktur zugewiesen bekommen. Der dritte und letzte Teil beschreibt die Umsetzung von SDL-Strukturen in ASN.1 Strukturen. Genau wie die Erstellung von Syntypes ist dieser Teil nicht zwingend erforderlich (*: Anzahl 0..∞). Alle spezifizierten Datentypen und -strukturen müssen exportiert werden, damit andere Module diese Typen kennen und benutzen können.

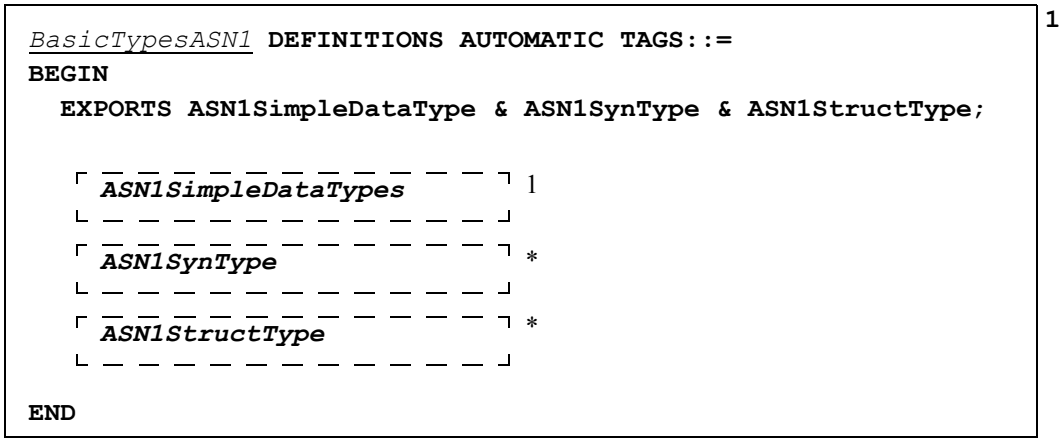


Abb. 5-6: Definiton des ASN Modules *BasicTypesASN1*

Die nächste Abbildung zeigt den detailierten Aufbau der drei Teile. In dem Block *ASN1SimpleDataTypes* wird für jeden einfachen SDL-Haupttyp (z.B. Integer, Real oder Charstring) eine entsprechende Zuweisung auf einen ASN.1 Typ gemacht. Änderungen an den Basistypen sind so nur an einer Stelle vorzunehmen. Zusätzlich dazu muss ein neuer Typ *Noparam_Type* eingeführt werden, der ein SDL-Signal ohne Parameter beschreibt (genaueres folgt). Der Block *ASN.1SynTypes* kann entweder eine Definition eines SDL-Syntypes auf einen einfachen Datentyp oder auf eine Struktur enthalten. Der *ASN1StructType* spezifiziert eine SDL-Struktur. Dazu wird diese als *SET* in ASN.1 nachgebildet, mit dem einzigen Unterschied, dass die Variablentypen auch hier ASN.1 Typen sein müssen.

```
ASN1SimpleDataType

Real_Type      ::= REAL
Integer_Type   ::= INTEGER
String_Type    ::= IA5String
Noparam_Type   ::= NULL
[MoreSimpleTypes_Type ::= SIMPLETYPE]
```

```
ASN1SynType

SdlSynType ::= ASN1SimpleDataType|ASN1StructType
```

```
ASN1StructType

SdlStructName ::= SET{
 member1  ASN1SimpleDataType|ASN1SynType|ASN1StructType
 ...
 memberN  ASN1SimpleDataType|ASN1SynType|ASN1StructType
}
```

Abb. 5-7: Detailierte Beschreibung des Moduls *BasicTypesASN1*

Abbildung 5-9 zeigt einen Auszug aus dem nach den Vorgaben des Patterns erstellten ASN.1 Modul *BasicTypesASN1*. Die einfachen Datentypen wurden exakt aus der Patternbeschreibung übernommen und für die Syntypes *QosClass* und *Fps* sind entsprechende ASN.1 Typen erstellt worden. Damit man diese in der SDL-Spezifikation des Systems von den SDL-Typen unterscheiden kann, wurden die ASN.1 Typen um das Suffix „_ASN“ erweitert. Für die Struktur *Image* wurde ein ASN.1 Typ *IMAGE_STRUCT_ASN* erstellt, der aus den Datentypen aufgebaut, die die dazugehörige SDL-Struktur beschreiben (vgl. Abb. 5-8 und Abb. 5-9). Für weitere SDL-Strukturen ist entsprechend zu verfahren.

Falls man schon in einem frühen Stadium der Systementwicklung die Entscheidung trifft, ASN.1 für die zu spezifizierenden Datentypen zu nutzen, kann das Pattern auch dazu verwendet werden, alle benötigten Datentypen und Strukturen von Beginn an in ASN.1 zu beschreiben. Noch vorhandene SDL-Typen können in der Spezifikation durch Syntypes auf die ASN.1 Repräsentation ersetzt werden. Diese Syntypes können dann genauso wie reine SDL-Strukturen verwendet werden. In dieser Arbeit wurde das so gemacht.

SDL

```
NEWTYPE Image
  Struct
    data Bit_String;
    length Integer;
ENDNEWTYPE;
```

Abb. 5-8: SDL-Struktur *Image*

```
BasicTypesASN1
   DEFINITIONS
      AUTOMATIC TAGS ::=

BEGIN

EXPORTS REAL_TYPE,INTEGER_TYPE,STRING_TYPE,NOPARAM_TYPE,...,FPS_ASN;

  REAL_TYPE     ::= REAL
  INTEGER_TYPE  ::= INTEGER
  STRING_TYPE   ::= IA5String
  NOPARAM_TYPE  ::= NULL

  QOSCLASS_ASN ::= STRING_TYPE
  FPS_ASN ::= INTEGER_TYPE
  ...

  IMAGE_STRUCT_ASN ::= SET{
    data  BIT STRING,
    length INTEGER_TYPE
  }
...
END
```

Abb. 5-9: Auszug aus dem ASN.1 Modul *BasicTypesASN1* der Middleware

Nachdem nun die notwenigen Datentypen spezifiziert wurden, kann mit der Umsetzung der SDL-Signale begonnen werden. Dazu muss für jede Anwendung ein eigenes Modul erstellt werden, in dem dann ein abstrakter Datentyp *UniqueASNDataType* definiert wird, welchem nach der Definition eines CHOICE-Types dann ein konkreter Wert zugewiesen wird. Dieser Datentyp kann dann encodiert und versendet werden. Für die CHOICE gibt es zwei Gruppen von Signalen: Signale mit und Signale ohne Parameter. Die einzelnen Signaltypen (SIGNAL in SDL) werden im Abschnitt *SignalTypes* spezifiziert.

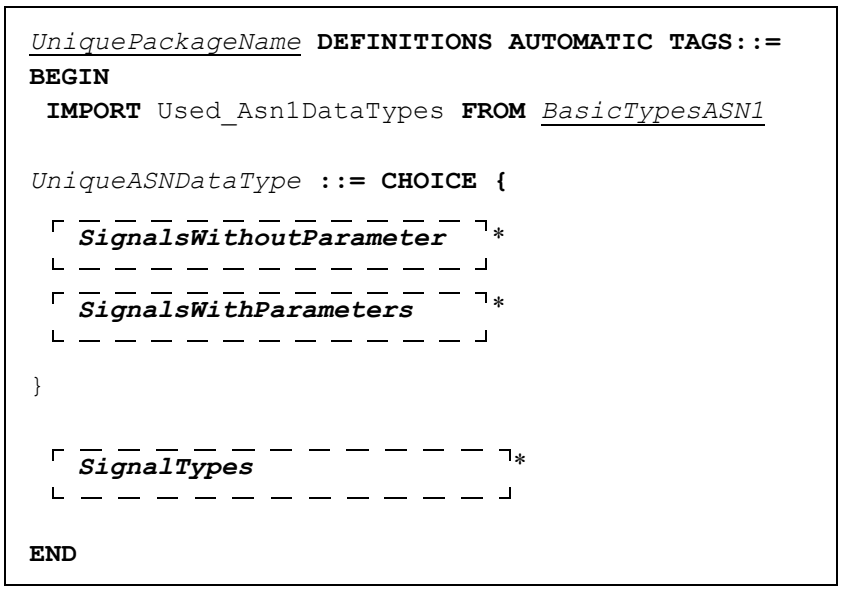


Abb. 5-10: Umsetzung der Signale einer Anwendung

Für Signale ohne Parameter findet eine Zuweisung von *sdlSignalName* zu dem vorher spezifizierten *NoParamType* statt (Abb. 5-11). Da keine weiteren Informationen mehr vorhanden sind, ist an dieser Stelle die Übersetzung von SDL zu ASN.1 abgeschlossen. Für Signale mit Parametern muss ähnlich der Umsetzung einer SDL-Struktur ein neuer Typ angelegt werden. Für jeden Parameter muss ein neues Element im *SET* erstellt werden.

```
SignalsWithoutParameter

sdlSignalName  ::= NoParamType

SignalsWithParameters

sdlSignalName  ::= SdlSignalNameType

SignalTypes

SdlSignalNameType  ::= SET {
  param1 AsnSimpleDataType|AsnSynType|AsnStructType
  ...
  paramN AsnSimpleDataType|AsnSynType|AsnStructType
}
```

Abb. 5-11: Detailierte Umsetzung von SDL-Signalen nach ASN.1

In Abbildung 5-12 gibt es einen Auszug aus dem ASN.1 Module *VideoASN1*. Es zeigt die exakte Übersetzung der SDL-Signale der Video-Anwendung nach ASN.1.

```
VideoASN1
  DEFINITIONS
    AUTOMATIC TAGS::=

BEGIN
 IMPORTS INTEGER_TYPE, ..., IMAGE_STRUCT_ASN FROM BasicTypesASN1;

 Video_Message_Type ::= CHOICE {
  image                   IMAGE_TYPE,
  identifyVideoQosClass   IDENTIFYVIDEOQOSCLASS_TYPE,
  enableCam               ENABLECAM_TYPE,
  disableCam              NOPARAM_TYPE,
  qosClassNotSet          QOSCLASSNOTSET_TYPE,
  qosClassSet             QOSCLASSSET_TYPE,
  defineNewVideoQosClass  DEFINENEWVIDEOQOSCLASS_TYPE
 }

 IMAGE_TYPE ::= SET{
  param1 IMAGE_STRUCT_ASN,
  param2 FPS_ASN,
  param3 JPEGQUALITY_ASN
 }
...
END
```

Abb. 5-12: Auszug aus dem ASN.1Modul VideoASN.1

5.1.1.4 Design Pattern *SimpleSignalCoDecBehaviour*

Nachdem nun die Struktur der Anwendung angepasst und die Umsetzung der SDL-Signale nach ASN.1 erfolgt ist, muss nun das Verhalten spezifiziert werden. Da es sich um ein wiederkehrendes Designproblem in der Domäne der Kommunikationssysteme handelt, konnte auch hier ein Pattern identifiziert werden.

Das *SimpleSignalCoDecBehaviour* Pattern gehört zur Klasen der *Interfacing Patterns.* Es stellt eine Übersetzung von verschiedenen SDL-Signalen zu einem einzigen Signal mit allen notwendigen Informationen und umgekehrt vor. Im Kontext der Protokollentwicklung findet eine Umsetzung von virtueller zu realer Kommunikation statt. Die Struktur des Patterns wurde bereits durch das *SimpleSignalCoDecArchitecture* Pattern eingeführt. Der Vollständigkeit halber wird sie an dieser Stelle noch einmal aufgeführt.

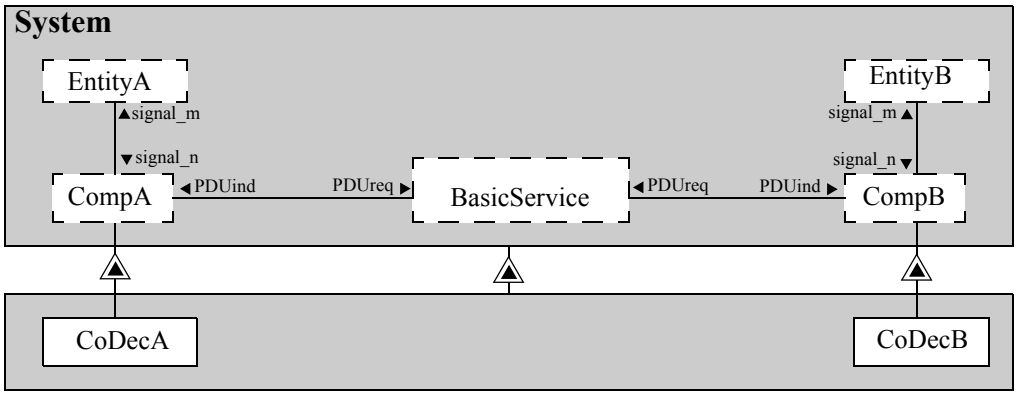


Abb. 5-13: Die Struktur des *SimpleSignalCoDecArchitecture* Patterns

Abbildung 5-14 zeigt ein typisches Szenario nach Anwendung des Patterns. Die beiden Kommunikationsendpunkte *sender(EntityA)* und *receiver(EntityB)* kommunizieren über einen Basisdienst miteinander. Die beiden *CoDecs* übersetzen und kapseln die verschiedenen Anwendungssignale *signal_n* und reichen diese in serialisierter Form an den Basisdienst weiter.

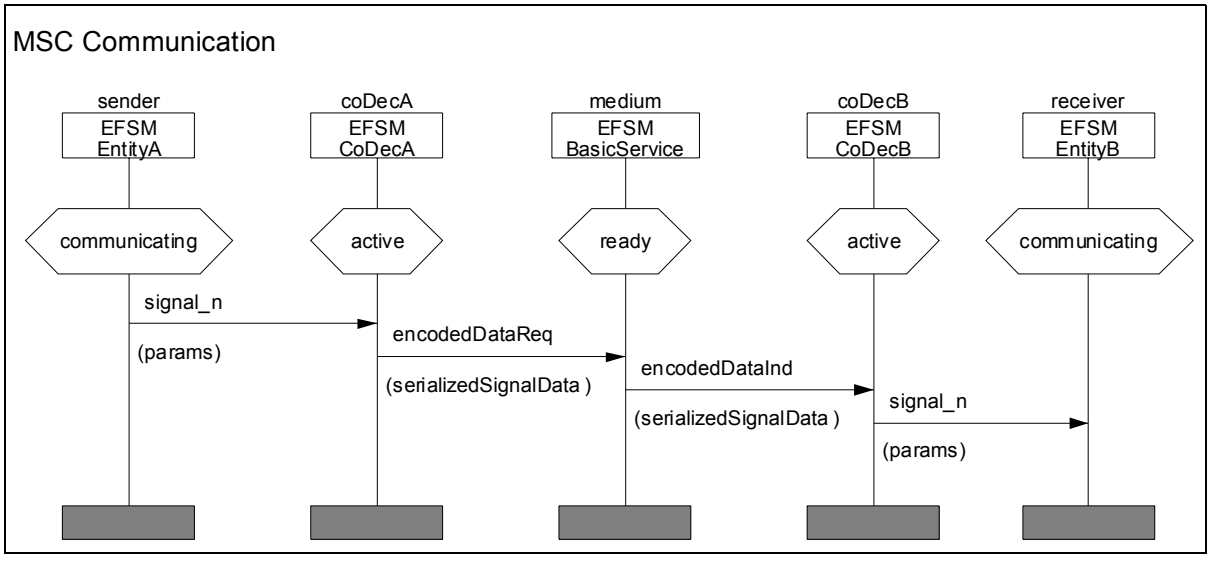


Abb. 5-14: MSC *SimpleSignalCoDecBehaviour*

Das dazugehörige SDL-Fragment von *CoDecA* ist in Abbildung 5-15 zu sehen. Die Spezifikation des *PDUType* (Protocol Data Unit) ist in ASN.1 erfolgt. Diese Struktur beinhaltet die möglichen Signale der Anwendung. Als weitere Variable *sdu* (Service Data Unit) wird z.B. ein OCTET STRING benötigt, der die gekapselten und serialisierten Signale aufnehmen kann. Tau besitzt eingebaute Methoden zur Encodierung (encode) und Decodierung (decode). Diese beiden Methoden transformieren eine ASN.1 Struktur z.B. in einen OCTET STRING.

Im Zustand *communicating* werden die Signale der Anwendung verarbeitet, gekapselt und encodiert. Mit dem Signal *PDUreq(data)* wird an den Basisdienst übertragen. Im Zustand *receiving* nimmt *CoDecA* Signale vom Basisdienst entgegen. Diese werden decodiert und als Signal mit Parameter an die Anwendung weitergereicht. Gewöhnlich sind die beiden Zustände gleich, da Signale immer verarbeitet werden sollen. Analog dazu ist *CoDecB* aufgebaut. Die vollständige Beschreibung des Patterns findet sich in Anhang A.

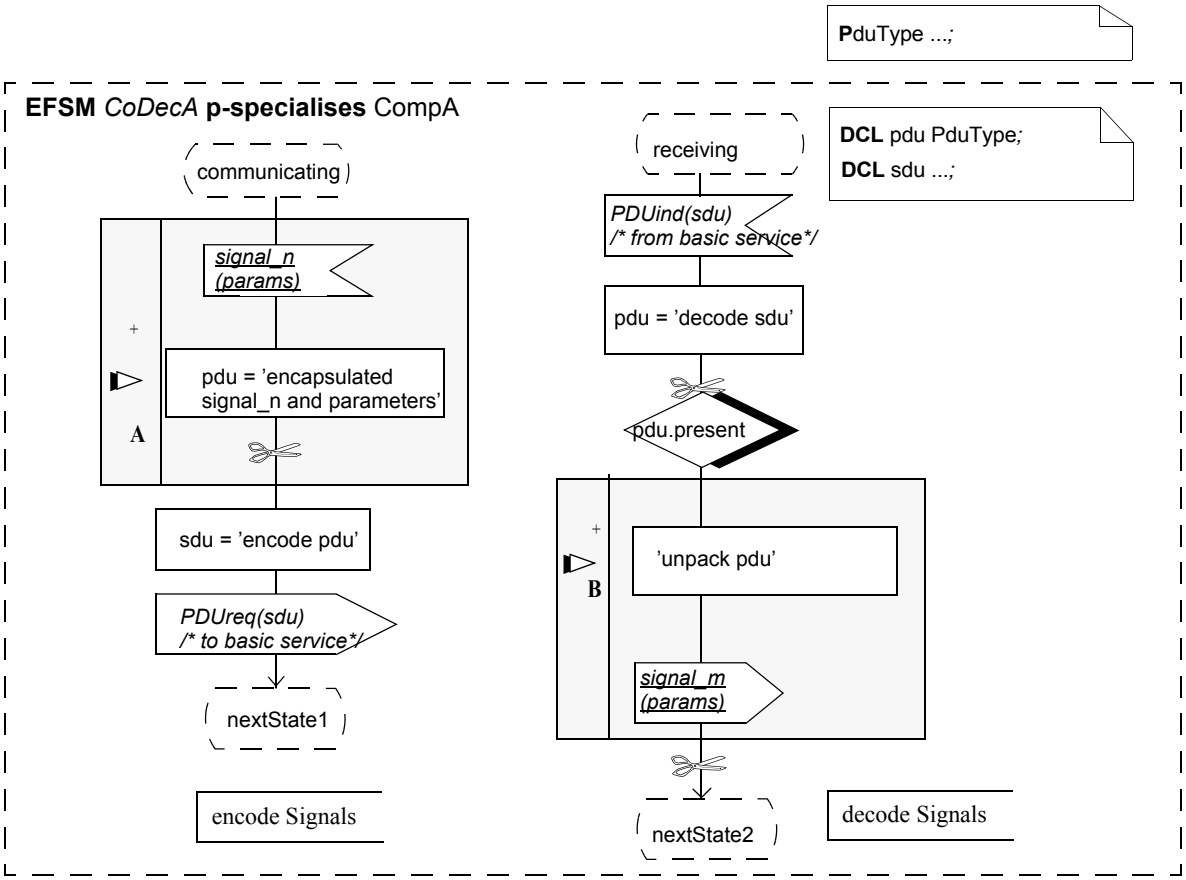


Abb. 5-15: SDL-Fragment von *CoDecA*

Wie erwähnt wird in unserem System keine Unterscheidung zwischen dem *Codec* auf Client- und dem *CoDec* auf Serverseite gemacht. Als Konsequenz muss der jeweilige anwendungsspezifische *CoDec* sowohl die Signale des Servers, als auch des Clients en- und decodieren können. Dadurch ist die Komponente wiederverwendbar. Die Abbildung 5-16 zeigt einen Ausschnitt der Spezifikation des *VideoCoDec*. Die Signale der Anwendung werden gekapselt und in einen OCTET STRING *sdu* encodiert. Dieser wird an die Schicht darunter weitergereicht. Der Ablauf beim Empfang von Signalen der darunterliegenden Schicht ist genau invertiert (Abb. 5-17). Zuerst muss der entsprechende ASN.1 Datentyp aus dem OCTET STRING decodiert werden. Mit

dem in Tau eingebauten Operator !present kann nun überprüft werden, welche Variable der ASN.1 CHOICE gesetzt wurde bzw. welches Signal encodiert worden ist. Abschließend müssen aus der ASN.1 Struktur die Parameter des entsprechenden Signals extrahiert und an die Applikation weitergereicht werden (Abb. 5-18).

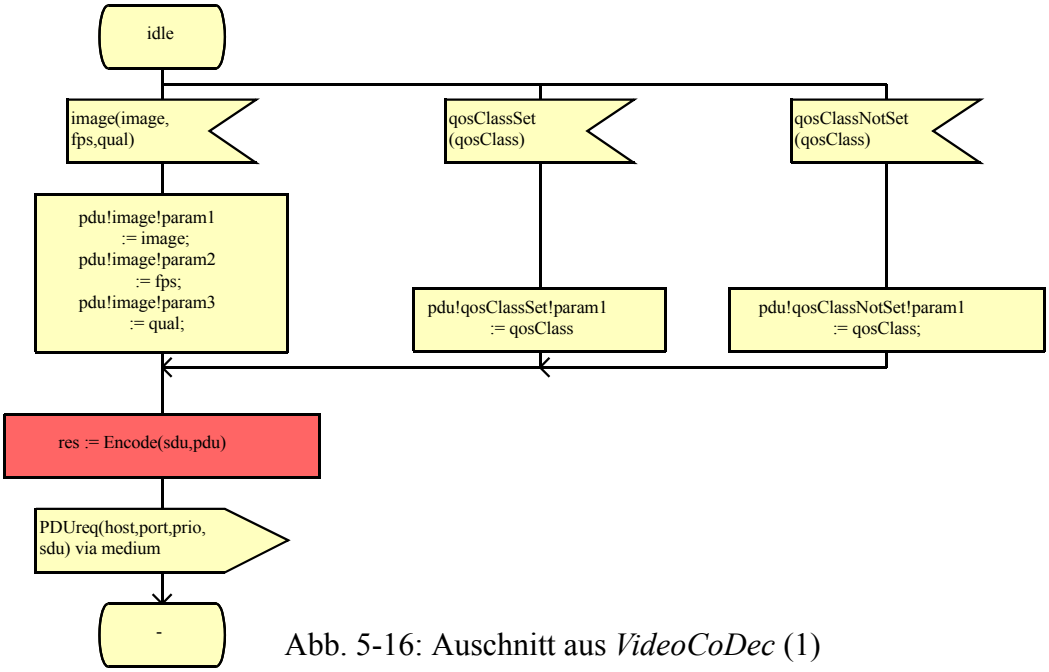


Abb. 5-16: Auschnitt aus *VideoCoDec* (1)

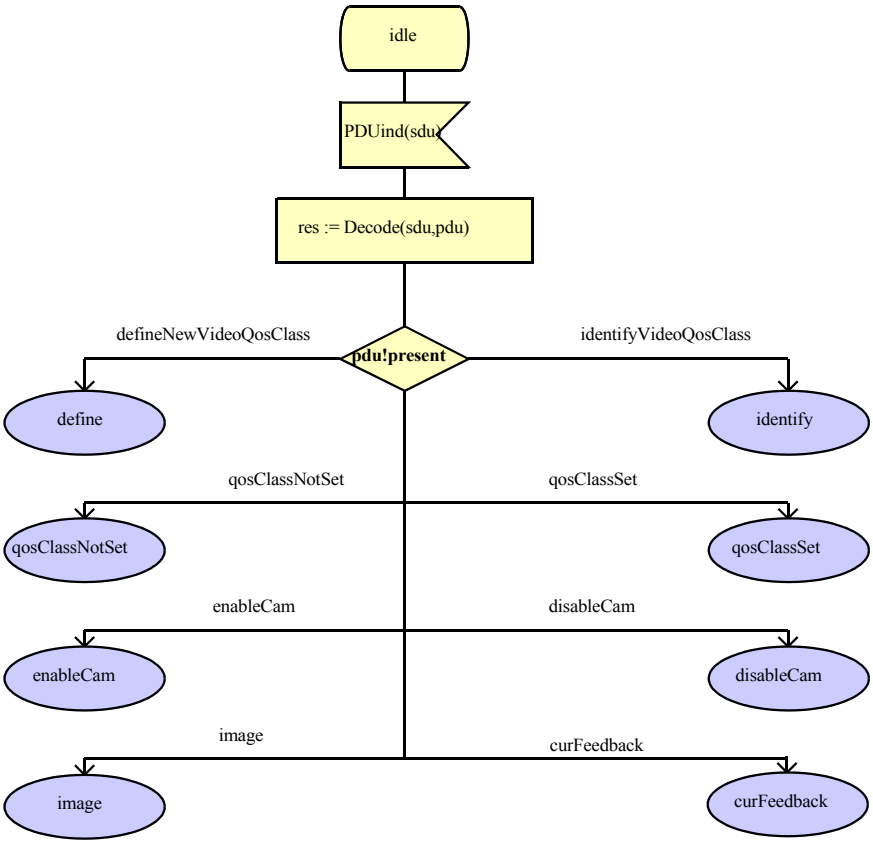


Abb. 5-17: Ausschnitt aus *VideoCoDec* (2)

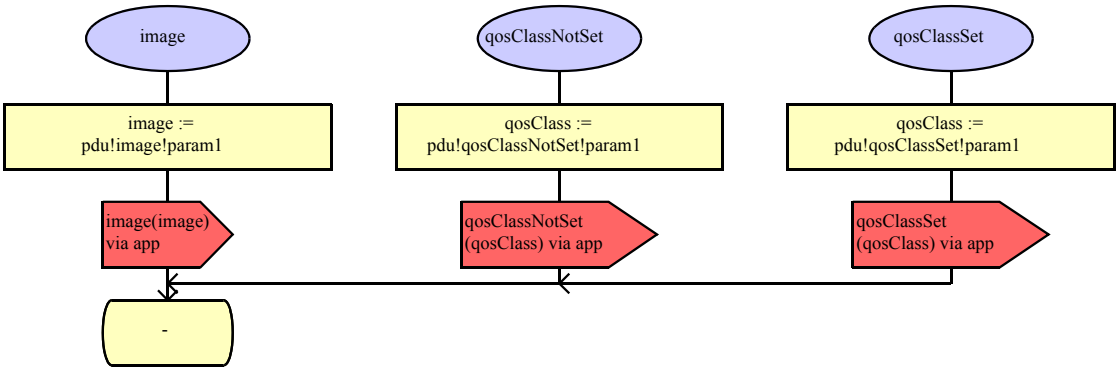


Abb. 5-18: Ausschnitt aus *VideoCoDec* (3)

5.1.2 *PSU* und *PSUTrigger*

5.1.2.1 Designproblem

Damit eine Middleware einen *fail-safe* oder *fail-operational* Zustand erlangen kann, muss sie die Fähigkeit besitzen, Fehlverhalten zu erkennen. Im weitesten Sinne fasst man unter Fehlverhalten all das zusammen, was zu einer Katastrophe im Sinne der Anwendung führen könnte. In unserem Fall müssen deshalb sowohl der Steuerungsserver als auch der Steuerungsclient überwacht werden, da ein Ausfall von einem der beiden zu einem Ausfall der gesamten Steuerung und somit zu einem unkontrollierten Zeppelin führt. Es wird dabei davon ausgegangen, dass der Zeppelin nicht über die Kamera gesteuert wird. Sonst müsste diese Anwendung auch überwacht werden.

Typischerweise wird bei einer solchen Konstellation ein *Watchdog* eingesetzt. Ein *Watchdog* beschreibt eine Komponente, die in regelmäßigen Abständen angesprochen werden muss, um nicht in einen *fail-safe* oder *fail-operational* Zustand überzuwechseln. In solchen Zuständen übernimmt im Allgemeinen der Watchdog die Kontrolle über das zu steuernde System oder er schaltet es ab (z.B. der Ausfall einer Kraftwerkssteuerung resultiert normalerweise zu einem Shutdown des kompletten Systems).

Die Prozesstypen *PSU (Periodical System Update)* und *PSUTrigger* übernehmen in unserer Middleware diese Funktionalität. Die serverseitige Komponente *PSU* übernimmt die Watchdog-Funktionalität, wohingegen PSUTrigger auf Clientseite regelmäßig Signale an PSU vom Client schickt.

5.1.2.2 Design-Pattern *Watchdog*

Das Design-Pattern *Watchdog* gehört zur Klasse der *Interaktionsmuster*. Es befasst sich mit der Realisierung einer Sicherheitsfunktionalität (safety) wie in Kapitel 4.1.2 vorgestellt. Es wird keine Architektur vorgegeben, nur ein Verhalten das eine vorhandene (oder auch neu erstellte) Komponente um die beschriebene Funktionalität erweitert.

Folgende Abbildung zeigt die strukturellen Aspekte der vorgeschlagenen Lösung des Designproblems. Wie erwähnt muss *WatchdogController* nicht notwendigerweise eine Komponente des Systems verfeinern, es kann auch eine neue hinzugefügt werden. *Trigger* ist eine Komponente des Umgebung. Sie stellt das (periodische) Signal bereit, welches den Watchdog immer wieder anstößt. *Controller* ist das zu überwachende System. Im Falle eines Fehlverhaltens gilt es, dieses System in einen sicheren Zustand zu bringen.

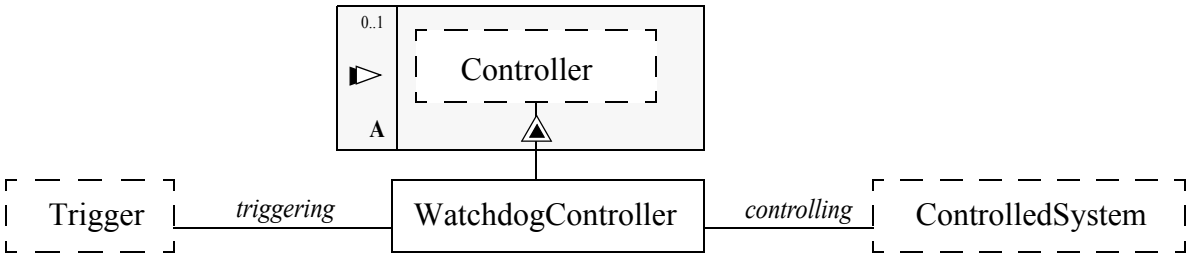


Abb. 5-19: Struktur des *Watchdog* Patterns

In der Komponente *WatchdogController* muss das Verhalten spezifiziert werden, welches die Watchdog Funktionalität bereitstellt. Das SDL-Fragment in Abbildung 5-20 beschreibt die Umsetzung des Patterns. Nach dem Trigger durch den Kontext muss *WatchdogController* den internen Watchdog-Timer neu starten. Wird ein Timeout des Watchdog-Timers empfangen, so muss sichergestellt werden, dass das kontrollierte System in einen sicheren Zustand wechselt (z.B. durch Senden bestimmter zuvor festgelegter Steuerungssignale). Falls der *Trigger* nach einem Ausfall wieder reaktiviert werden konnte, so muss der Watchdog-Timer erneut gesetzt werden und das System kann wie gewohnt kontrolliert werden.

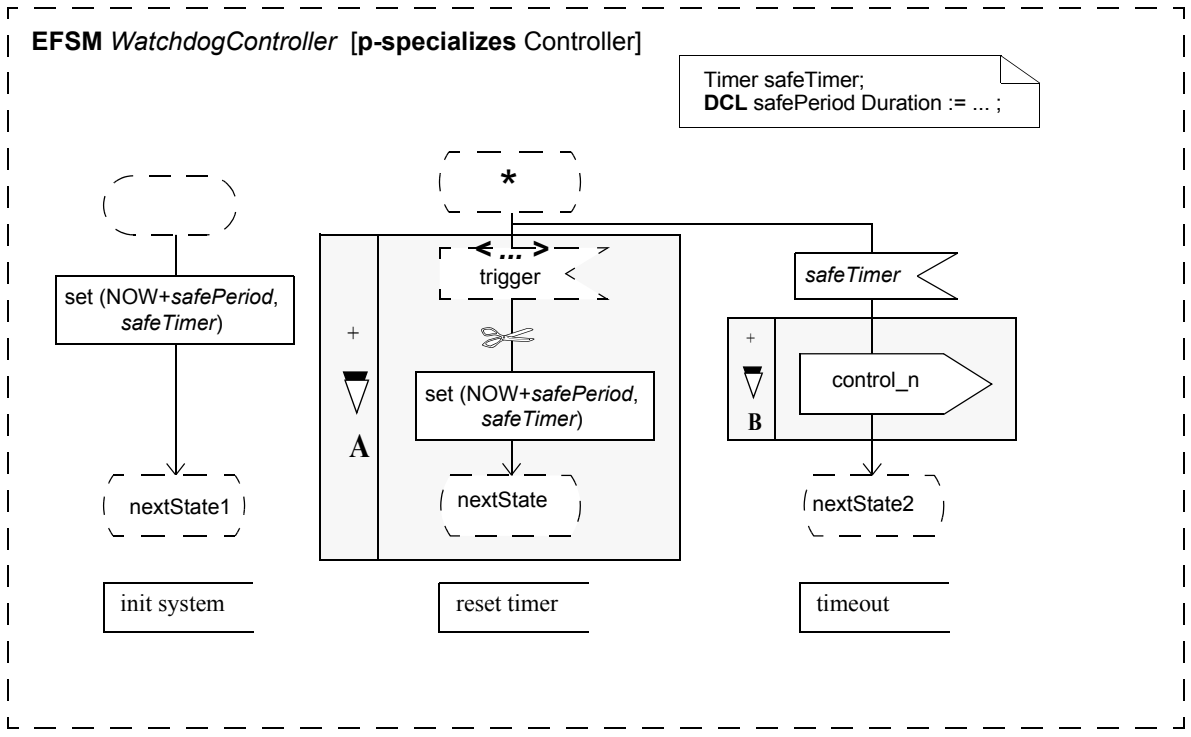


Abb. 5-20: SDL-Fragment des *Watchdog* Patterns

Die Abbildung 5-21 zeigt die Anwendung des Patterns auf die Komponente *PSU*. Die eine Funktionalität dieser Komponente besteht aus der regelmäßigen Ansteuerung des Servos und der Motoren, damit z.B. der Servo nicht in seinen sleep-Modus fällt. Dies geschieht alle 20 ms (Timer *update*). Die andere Funktionalität ist der beschriebene Watchdog. In dieser Middleware übernimmt das *newCtrlValues*-Signal die Rolle des *trigger* (vgl. Abb. 5-20). Wird es empfangen, so setzt es den Watchdog-Timer *watchdog* neu. Dieser wurde initial auf 3 Sekunden ge-

setzt. Erhalten wir einen Timeout von *watchdog*, so setzten wir die aktuellen Steuerungswerte des Zeppelins auf zuvor definierte *fail-operational*-Werte, die es dem Luftschiff ermöglichen, selbstständig zu landen. Es wird davon ausgegangen, dass ein solcher Landevorgang 10 Sekunden dauert. Hierfür ist der Timer *off* zuständig, der im zusätzlichen Zustand *fail* den *update*-Timer deaktiviert und somit die Motoren ausschalten. Der Zustand *fail* ist fast identisch mit dem Zustand *enabled* und nicht zwingend notwendig. Er dient lediglich dazu, dass System dahingehend zu optimieren, dass nicht bei jedem Eintreffen eine Steuerungswertes zusätzlich zum Zurücksetzen des Watchdog-Timer auch der *off*-Timer deaktiviert wird. Hier gilt Performanz vor „Zustandsminimalismus“. Das Signal *stopZeppelin* deaktiviert den Zeppelin und somit auch den Watchdog.

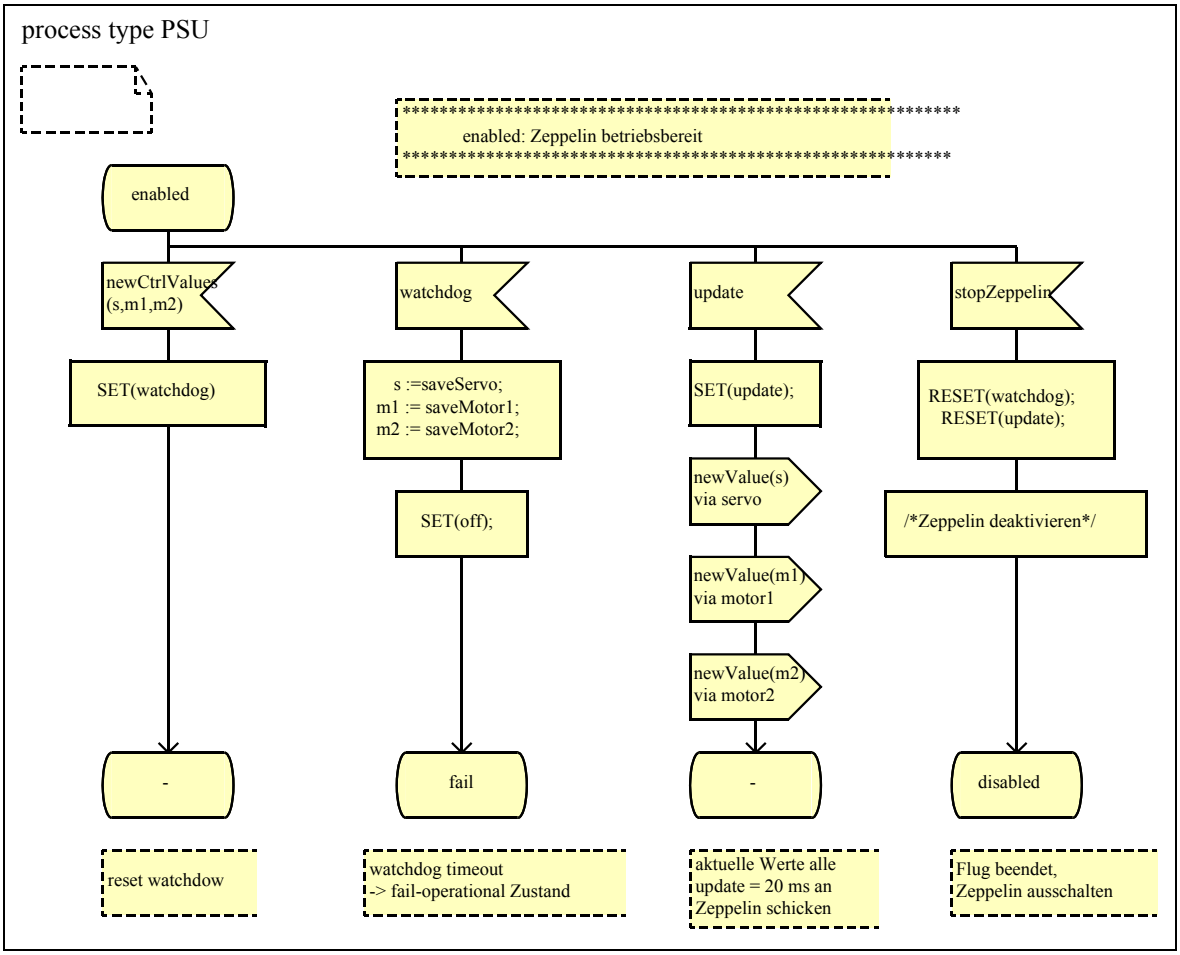


Abb. 5-21: Ausschnitt aus dem Prozesstyp *PSU*

Ein Ausschnitt des Prozesstyps *PSUTrigger* ist in Abbildung 5-22 dargestellt. Nach dem Starten des Zeppelin wird auch ein *update*- , ein *resend*- und ein *alive*-Timer gestartet. Der *update*-Timer reicht alle 50 ms die aktuellen Werte an die darunterliegende Schicht weiter, sofern diese sich geändert haben, der *resend*-Timer sendet die aktuellen Wert alle 2 Sekunden, falls diese sich nicht geändert haben und der *alive*-Timer überwacht die Applikation. Sollte diese innerhalb von 3 Sekunden kein Signal senden, so geht diese Komponente davon aus, dass ein Fehlverhalten vorliegt und beendet die Kommunikation mit dem Server (es werden keine Steuerungssignale mehr weitergereicht). Dabei wird davon ausgegangen, dass die Anwendung permanent Signale an die Middleware sendet (alle 20 ms, da das die optimale Zeit ist, die Komponenten auf Serverseite anzusteuern). Somit kann der Server oder genauer die *PSU* ein Versagen des Client erkennen, da das benötigte *Heartbeat*-Signal nicht gesendet wird.

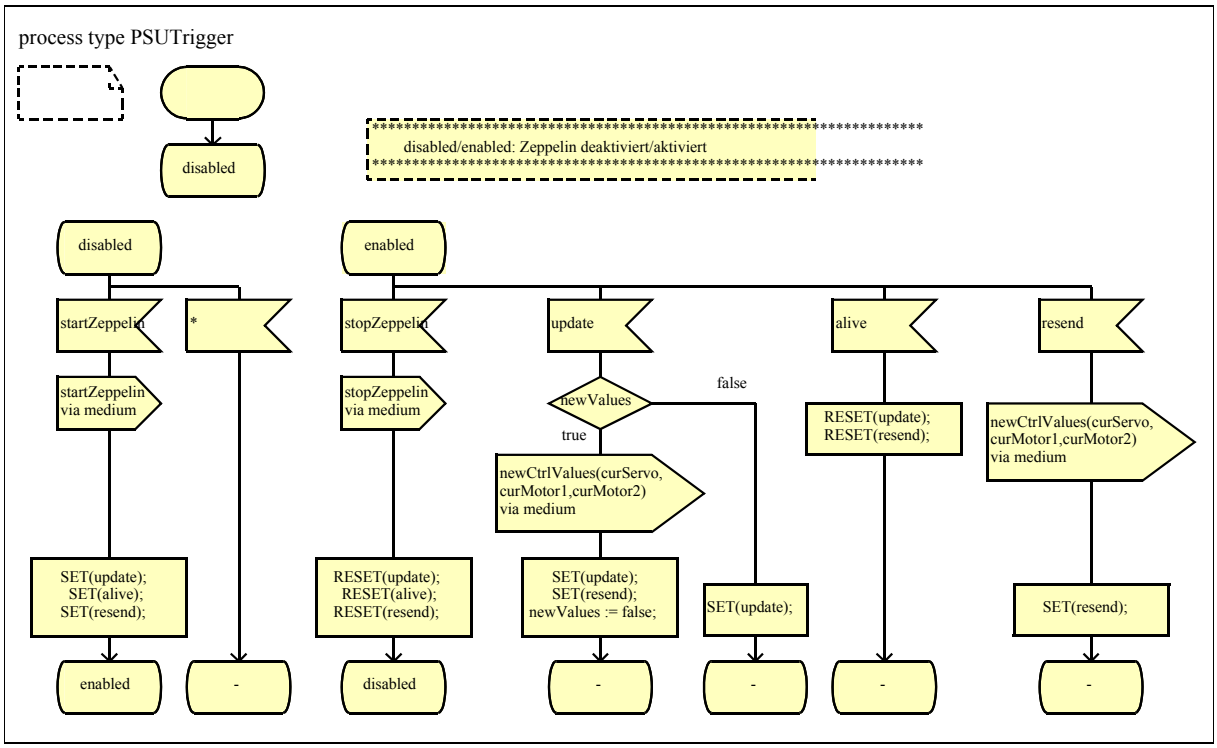


Abb. 5-22: Ausschnitt aus dem Prozesstyp *PSUTrigger*

5.1.3 Das Mikroprotokoll *VideoMapping*

5.1.3.1 Designproblem

Das in Kapitel 4.3.1 beschrieben Dienstgüte-Mapping der Videoanwendung muss durch die Middleware umgesetzt werden. Den einzelnen Dienstgüteklassen müssen die korrekten Werte zugewiesen werden und die Middleware muss diese in geeigneter Form speichern.

5.1.3.2 Design Pattern *QosMapping*

Das Design-Pattern *QosMapping* gehört zur Klassen der *Interaktionsmuster*. Es beschreibt generisch das Problem des Mappings einer Dienstgüteklasse auf die dazugehörigen Werte und findet vor allem in Dienstgütearchitekturen Anwendung. Abbildung 5-23 zeigt die Struktur des Patterns. Das ursprüngliche System besteht aus zwei Komponenten, die nicht zwingend miteinander kommunizieren müssen. Die vorgeschlagene Lösung des gegebenen Designproblems sieht vor, beide Komponenten dahingehend zu verfeinern, dass eine Mapping-Funktionalität gegeben ist.

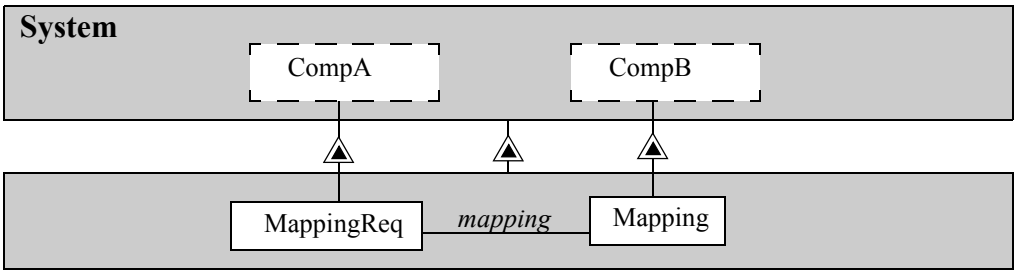


Abb. 5-23: Struktur des Design-Patterns *QosMapping*

Das folgende MSC zeigt ein typisches Szenario nach Anwendung des Patterns. Auf Anfrage von *MappingReq* setzt *Mapping* lokal die geforderte Dienstgüteklasse und liefert die benötigte Dienstgüte zurück. Vorher muss sichergestellt werden, dass die geforderte Dienstgüteklasse spezifiziert wurde.

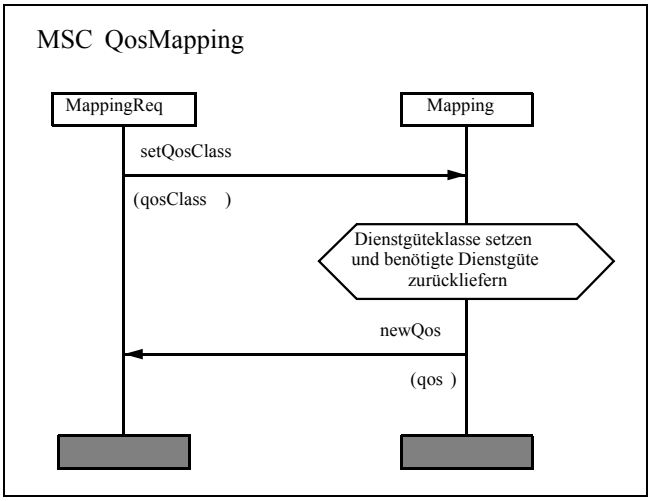


Abb. 5-24: MSC Dienstgüte-Mapping

Das dazugehörige SDL-Fragment ist in Abbildung 5-25 beschrieben. *CompA* wird dahingehend verfeinert, dass diese nach einem Stimulus *trigger*, eine neue Dienstgüteklasse setzt. Danach muss sie auf eine entsprechende Antwort warten. Der Typ *QosClass* und die Variable(n) *qos* sind geeignet zu spezifizieren (z.B. *qos* als Bandbreite-Delay-Paar). Die Verfeinerung von *CompB* sieht vor, dass diese Komponente nach dem Empfang einer Dienstgüteklasseanforderung eine Überprüfung durchführt, um festzustellen, ob die neue Dienstgüteklasse *qosClass* existiert. Dazu ist es wichtig, geeignete Container zu spezifizieren (*QosTable*) und mit den möglichen Dienstgüteklassen und deren Parametern zu initialisieren. Falls die Überprüfung erfolgreich ist, kann die benötigte Dienstgüte z.B. als Bandbreite/Delay Tupel berechnet und an den Sender der Anforderung gesendet werden. Falls im System eine Scaling-Komponente vorhanden ist, so muss diese über die aktuelle Dienstgüteklasse in Form von einem Parameterintervall, wie in Tabelle 4-1 gezeigt, informiert werden. Die genaue Art der Benachrichtigung ist von der Spezifikation der betreffenden Scaling-Komponente abhängig (gemeinsame Variablen, RPC oder ein Signal).

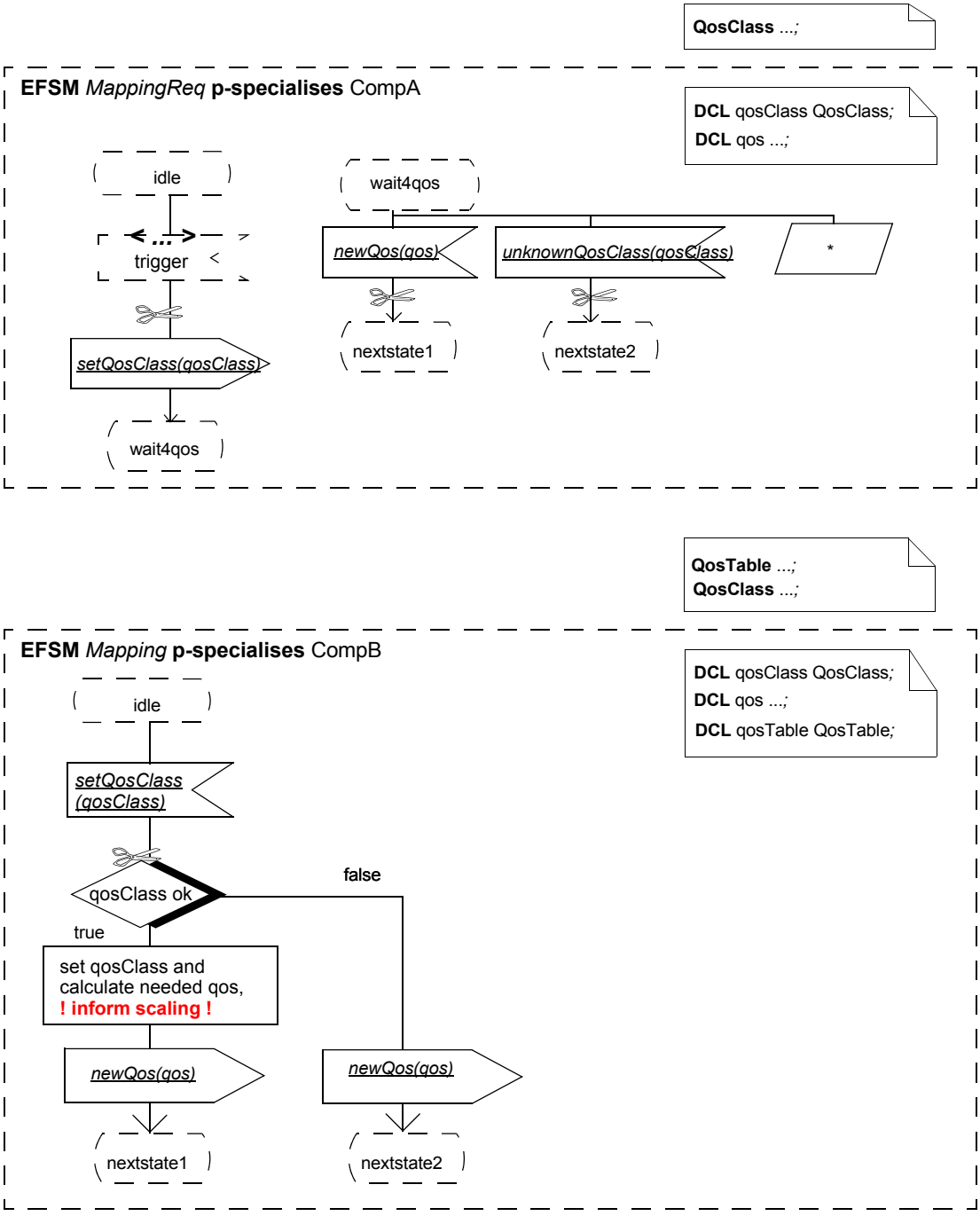


Abb. 5-25: SDL-Fragment *QosMapping*

5.1.3.3 Die Beschreibung von *VideoMapping*

Das Mikroprotokoll *VideoMapping* wurde nach den Vorgaben des Patterns erstellt. Es spezifiziert eine applikationsspezifische Funktionalität, nämlich die Umsetzug der Tabelle 4-1. Wie jedes der neu entwickelten Mikroprotokolle benutzt es in den Schnittstellen nach außen nur ASN.1 Datenstrukturen, um in der Kommunikation mit anderen Komponenten nicht auf eine bestimmte Implementierungs- bzw. Spezifizierungssprache eingeschränkt zu sein. Betrachten wir zuerst die Schnittstellen und Datenstrukturen des Mikroprotokolls (Abb. 5-26). Die Kommunikation mit der Anwendung erfolgt über das Gate *app*. Die entsprechenden Parameter einer

Dienstgüteklasse, z.B. die benötigte Bildrate, wird über das Gate *scal* propagiert. Um die vorgegebene Dienstgütetabelle umsetzen zu können, wurden neue Strukturen erstellt, die die Daten aufnehmen können. Die Struktur *QosTable* besteht aus einem Index, der eine Dienstgüteklasse (*QOSCLASS_ASN*) ist, und einem Eintrag, der wiederum aus den minimalen und optimalen Anforderungen der Klasse aufgebaut ist. In diesem Fall besteht ein Eintrag aus den jeweiligen Bildraten und Qualitätsangaben für die Kamera. Die Menge *QosSet* beinhaltet die momentan spezifizierten Anwendungsfälle. Dadurch wird eine umständliche Suche in *QosTable* vermieden.

Abbildung 5-27 zeigt das Verhalten des Mikroprotokolls. Nach dem Signal *setQosClass* findet eine Überprüfung statt, ob die gewünschte Dienstgüteklasse existiert. Falls sie erfolgreich ist, werden aus der Tabelle die entsprechenden Parameter ausgelesen und eine Instanz von *VideoScaling* (siehe Kapitel 5.1.4) wird über die neue Situation informiert (*setMinOptValues*). Was jetzt noch zu tun bleibt, ist die Berechnung der aktuell benötigten Dienstgüte in Form von zwei Bandbreite/Delay Tupeln. Eines für die minimal benötigte Dienstgüte und eines für die optimale. Die Berechnung und das Versenden der Daten geschieht in der Prozedur *calculateQos*. Dadurch kann z.B. durch Austausch der Prozedur leicht die Berechnung der benötigten Bandbreite geändert werden, ohne das Mikroprotokoll ändern zu müssen.

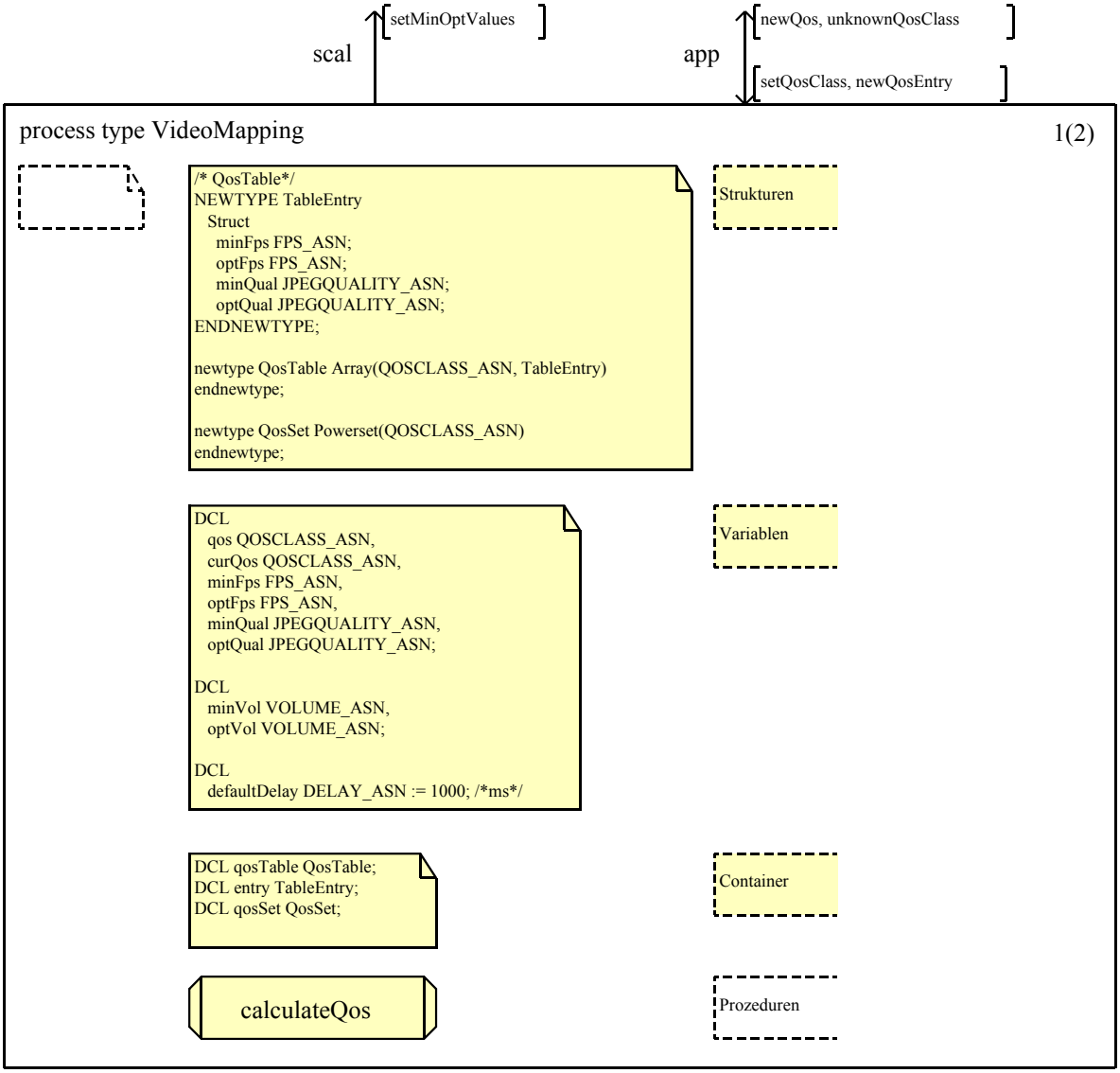


Abb. 5-26: Mikroprotokoll *VideoMapping*: Datentypen, Strukturen und Gates

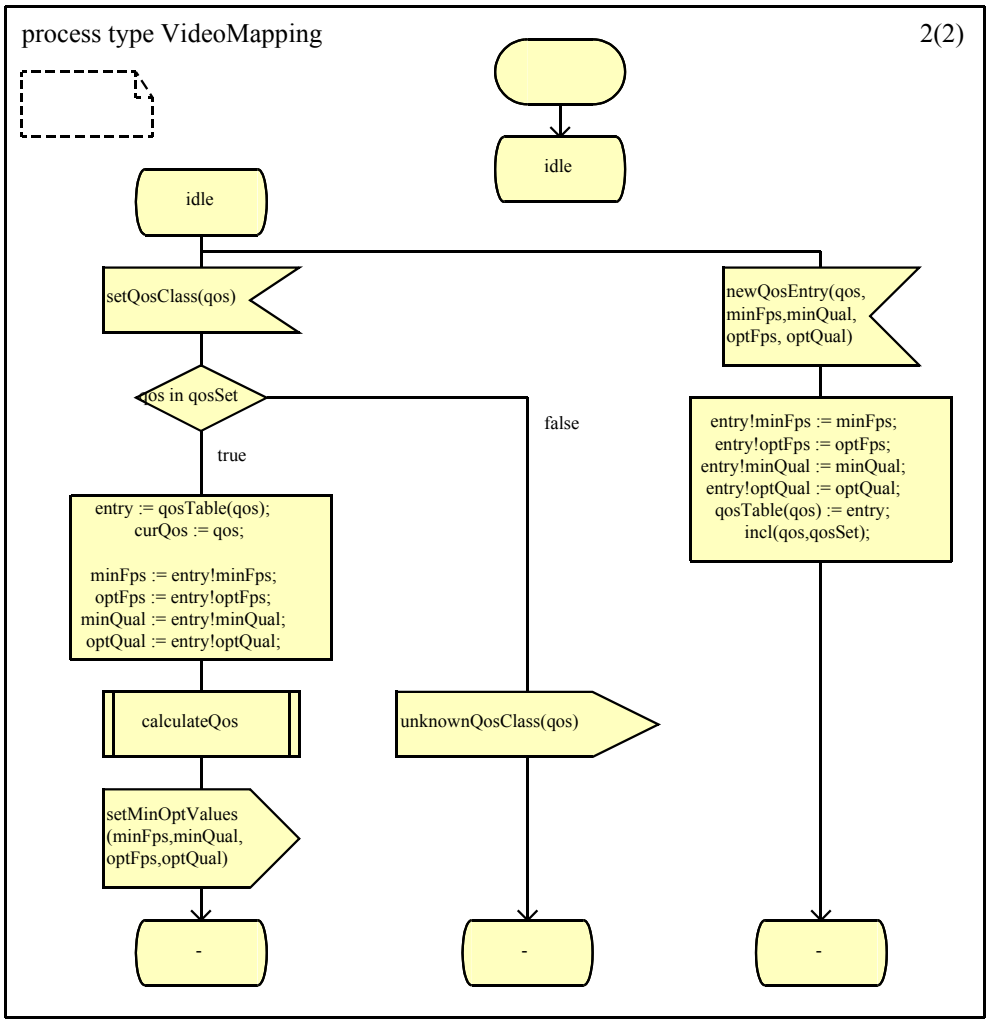


Abb. 5-27: Mikroprotokoll *VideoMapping*: Verhalten

Auch wenn es sich bei dem vorgestellten Mikroprotokoll um ein sehr anwendungsspezifisches handelt, so ist eine Wiederverwendbarkeit dadurch gegeben, dass es möglich ist, neue Einträge über das Signal *newQosEntry* der Dienstgütetabelle hinzuzufügen. Da diese initial nicht gefüllt ist, kann *DelayMapping* für eine ganz spezielle Klasse von Anwendungen genutzt werden.

5.1.4 Das Mikroprotokoll *VideoScaling*

5.1.4.1 Designproblem

Die zweite wichtige Funktionalität hinsichtlich Dienstgüte in der applikationsspezifischen Middleware ist die Dienstgüte-Skalierung. Hier werden die Nutzdaten der aktuellen Situation angepasst bzw. eine Datenquelle wird so gesteuert, dass diese die Daten in der gewünschten Form liefert (vgl Kapitel 3.5.3).

Weit mehr als das Mapping ist das Skalieren der Daten applikationsabhängig. Aus diesem Grund wird darauf verzichtet, ein Design-Pattern anzugeben. Prinzipiell ist es zwar möglich, die generische Struktur oder die „Quintessenz“ des Verhaltens abstrakt zu beschreiben, aber die Beschreibung wäre so allgemein gehalten, dass ein Pattern wahrscheinlich keine große Hilfe ist.

5.1.4.2 Die Beschreibung von *VideoScaling*

Das Mikroprotokoll *VideoScaling* verarbeitet die Bilder der Webcam und Informationen über die aktuelle Dienstgüte, um daraus neue Vorgaben für die Kamera zu erstellen. Betrachten wir die Spezifikation in Abbildung 5-28. Über das Gate *mapp* nimmt es die Vorgaben für die aktuelle Klasse entgegen. Für das Mikroprotokoll spielt es keine Rolle, wer diese Vorgaben liefert, in diesem Fall ist es das Mikroprotokoll *VideoMapping*. Die Schnittstelle *app* dient zur Kommunikation mit der Anwendung. Über sie empfängt *VideoScaling* die aktuelle Dienstgütesituation und vor allem die Bilder von der Kamera. Die Anwendung bekommt über dieses Gate die neuen Kameraparameter und ein Feedback, ob die aktuelle Dienstgüteklasse befriedigt werden kann. Das Gate *medium* ist unidirektional und dient zur Kommunikation mit der darunterliegenden Schicht. Da dieses Mikroprotokoll zwischen Anwendung und *VideoCoDec* platziert ist, muss es sich für den *VideoCoDec* wie die Applikation verhalten, d.h. es darf nur das Signal *image* weitergereicht werden. Die anderen Signale der Anwendung werden um das Mikroprotokoll herumgereicht (siehe dazu auch Abb. 4-3: Das Signal *image* läuft über q3 und q4, die

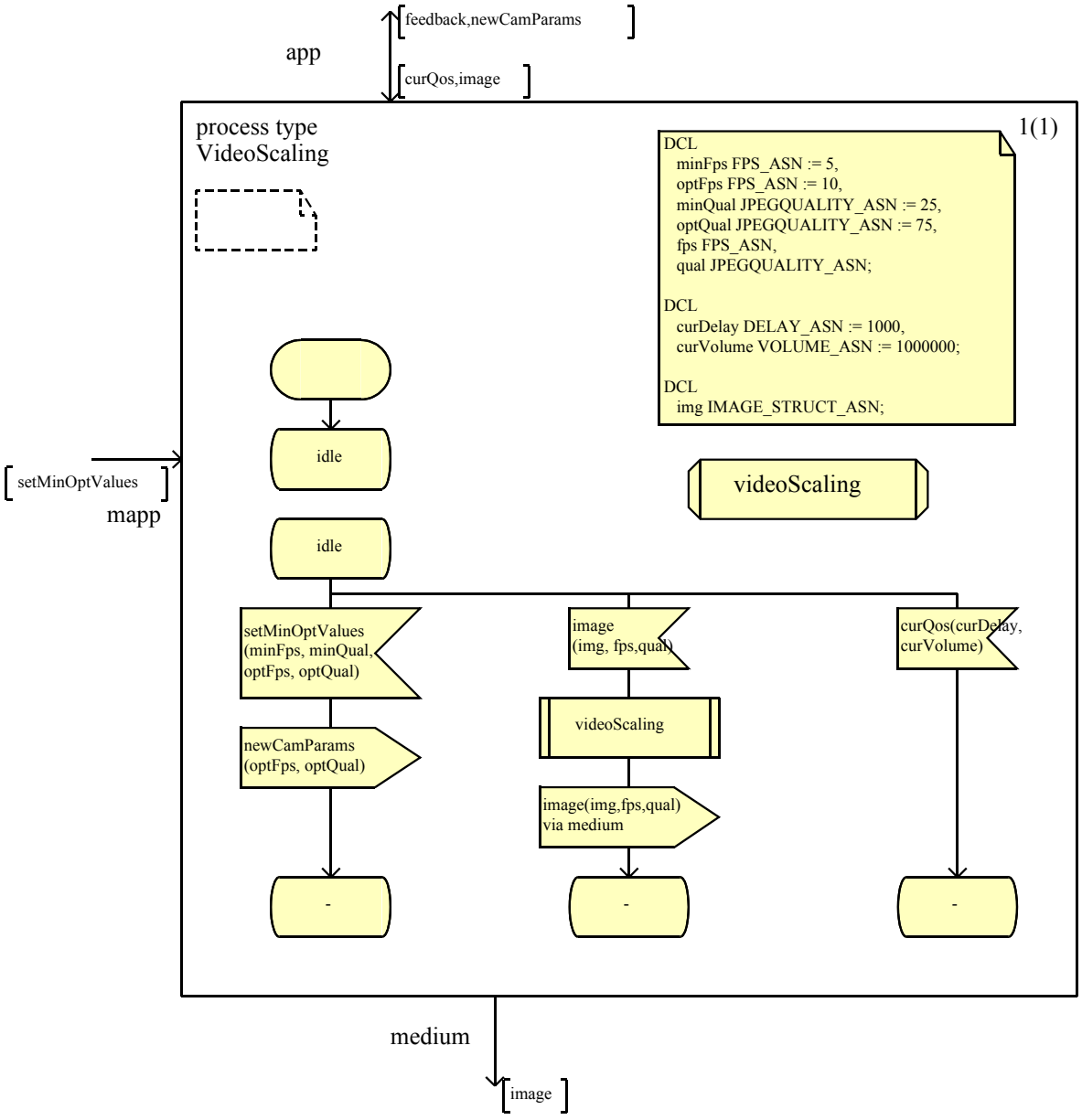


Abb. 5-28: Mikroprotokoll *VideoScaling*

restlichen Signale über c3. Die Signallisten sind nicht eingezeichnet).

Nach jedem Empfang eines Bildes findet die Skalierung der Daten in der Prozedur *videoScaling* statt. In dieser Prozedur wird auch anhand der verfügbaren Daten das entsprechende Feedback für den Benutzer festgelegt und an ihn versendet. Es gibt zwei mögliche Arten der Skalierung. Erstens eine Anpassung der Datenquelle: die Nutzdaten werden nicht verändert, es wird aber sichergestellt, dass die Datenquelle auf die veränderte Dienstgütesituation eingestellt wird. das wird in unserem Fall durch eine Justierung der Kameraparameter erreicht. Und zweitens eine zusätzliche oder ausschließliche Anpassung der Nutzdaten. In unserem Fall wäre es denkbar, ein Bild zusätzlich zu komprimieren bzw. in der Qualität herabzusetzen, um die benötigte Bandbreite zu verringern. Dieses Verfahren wird hier jedoch nicht verwendet. Natürlich findet auch eine Skalierung statt, falls sich die Ressourcensituation verbessert, nur kann dies nicht durch die zweite Methode erfolgen.

Über das Signal *curQos* wird das Mikroprotokoll über die aktuelle Situation informiert. Die neuen Daten werden bei der nächsten Skalierung berücksichtigt. Das MSC in Abb. 5-29 zeigt einen typischen Ablauf.

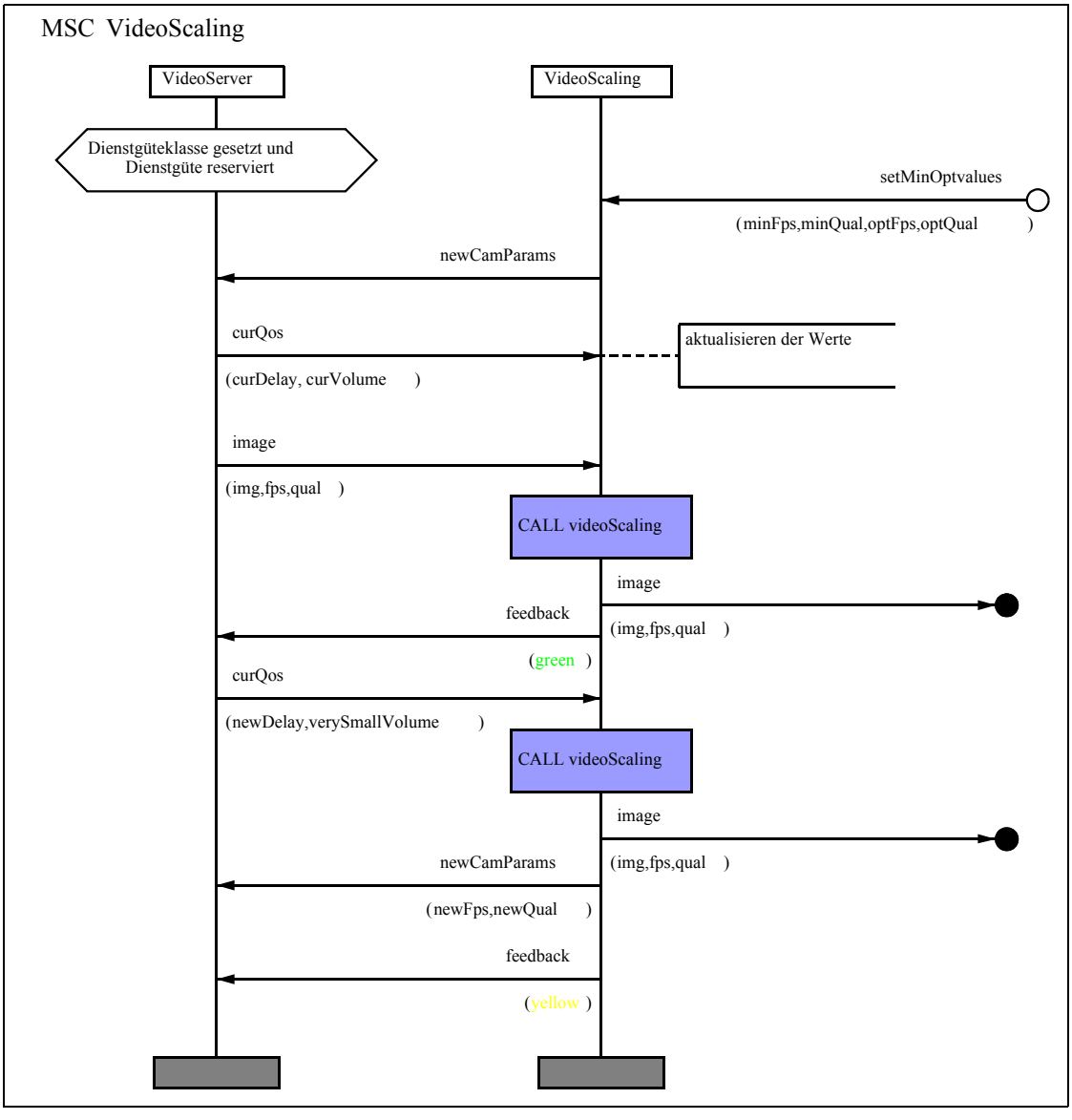


Abb. 5-29: MSC des Bildskalierung

5.1.5 Das semi-generische Mikroprotokoll *DelayMapping*

5.1.5.1 Beschreibung

Neben den komplexeren Dienstgüteklassen wie z.B. für die Videoanwendung, gibt es auch nicht so umfangreiche. Typische „einfache“ Dienstgüteklassen legen z.B. ein maximales und ein optimales Delay oder eine minimale und eine optimale Bandbreitenforderung fest (aber nicht beides zusammen). Diese einfachen Mappings unterscheiden sich nur in zwei Punkten von Anwendung zu Anwendung. Zum einen in den genauen Werten der Dienstgüteklassen, zum anderen in der benötigten Bandbreite pro Signal bzw. im benötigten Delay bei einer Bandbreitenreservierung.

Für eine einfache Abbildung auf ein Delay Mapping wird ein semi-generischen Mikroprotokoll vorgestellt, das *DelayMapping*. Wie in Abbildung 5-30 zu sehen ist, nimmt es über das Gate *app* die entsprechenden Signale von der Anwendung entgegen und das Gate *scal* wird zur Kommunikation mit einer Skalierungskomponente benutzt. Die entsprechenden Dienstgüteklasse mit ihren Parametern werden in einer Tabelle abgelegt (*QosTable*). Auch hier sind alle Datentypen in ASN.1 spezifiziert, wobei aber auch andere Repräsentationen (wie z.B. das *Powerset*) für interne, also private Daten oder Typen erlaubt ist.

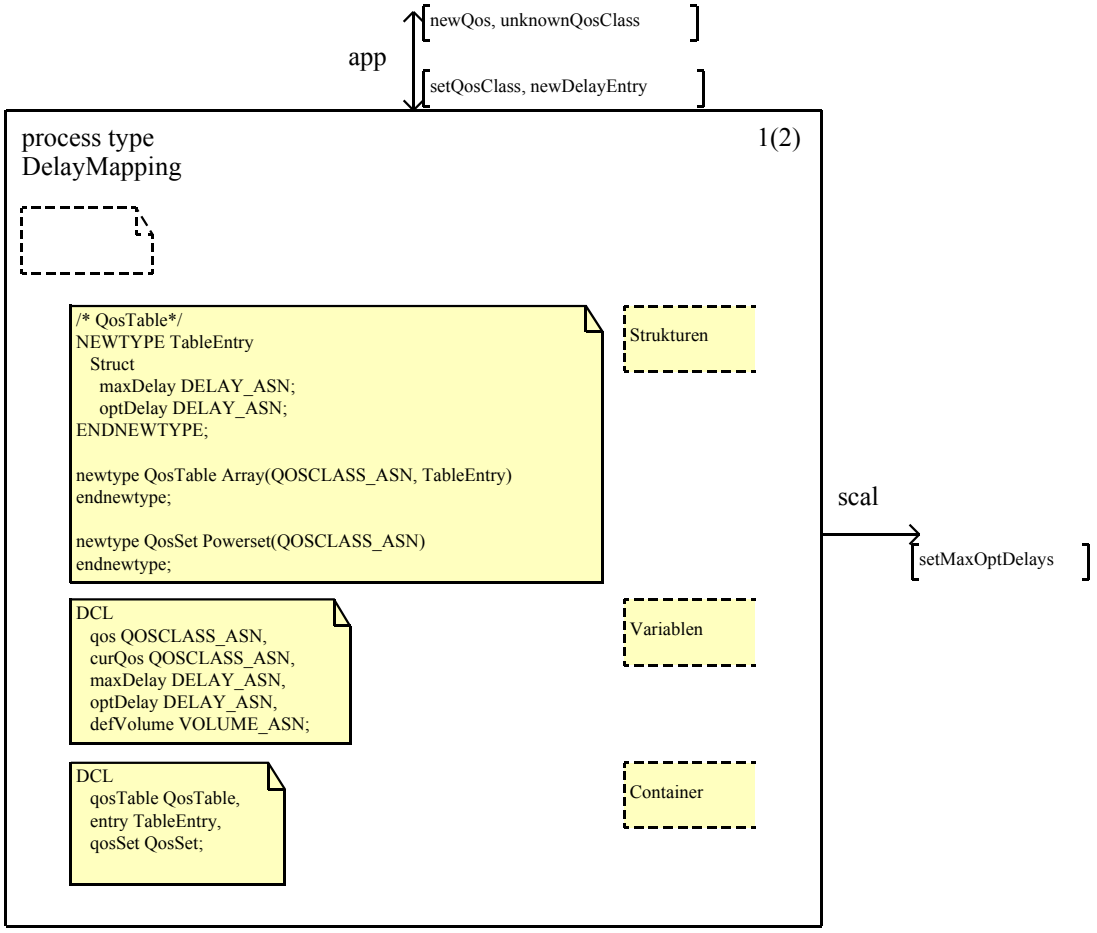


Abb. 5-30: Mikroprotokoll *DelayMapping*: Datentypen, Strukturen und Gates

Folgende Abbildung 5-31 zeigt das durch das Protokoll spezifizierte Verhalten. Nach Empfang des Signals *setQosClass* wird geprüft, ob die neue Dienstgüteklasse vorhanden ist. Ist das der Fall, so werden aus der Tabelle mit den Dienstgüteklassen die entsprechenden Werte ausgelesen und über das Gate *scal* propagiert. Eine aufwendige Berechnung der benötigten Bandbreite entfällt, da eine Forderung an der umgebenden Kontext ist, dass die zu sendende Datenmenge immer gleich ist. Zu diesem Zweck ist vorgesehen, die Starttransition zu überschreiben (deshalb semi-generisch) und das Protokoll dadurch individuell an eine Anwendung anzupassen, indem man die Variable *defVolume* auf die benötigte Bandbreite setzt.

Eine weitere Besonderheit ist das Erstellen von neuen Dienstgüteklassen mit Hilfe des Signals *newDelayEntry*. Durch Angabe einer neuen Dienstgüteklasse, z.B. 'Klasse1', kann in der Dienstgütetabelle *qosTable* ein neuer Eintrag erstellt werden. Initial ist die Tabelle leer. Bevor das Mikroprotokoll wie gewohnt genutzt werden kann, muss durch die Umgebung dafür gesorgt werden, dass die Tabelle gefüllt wird. Es ist auch denkbar, die Tabelle mit statischen Daten zu füllen, doch dann wäre das Mikroprotokoll wieder applikationsabhängig und in keinster Weise wiederverwendbar.

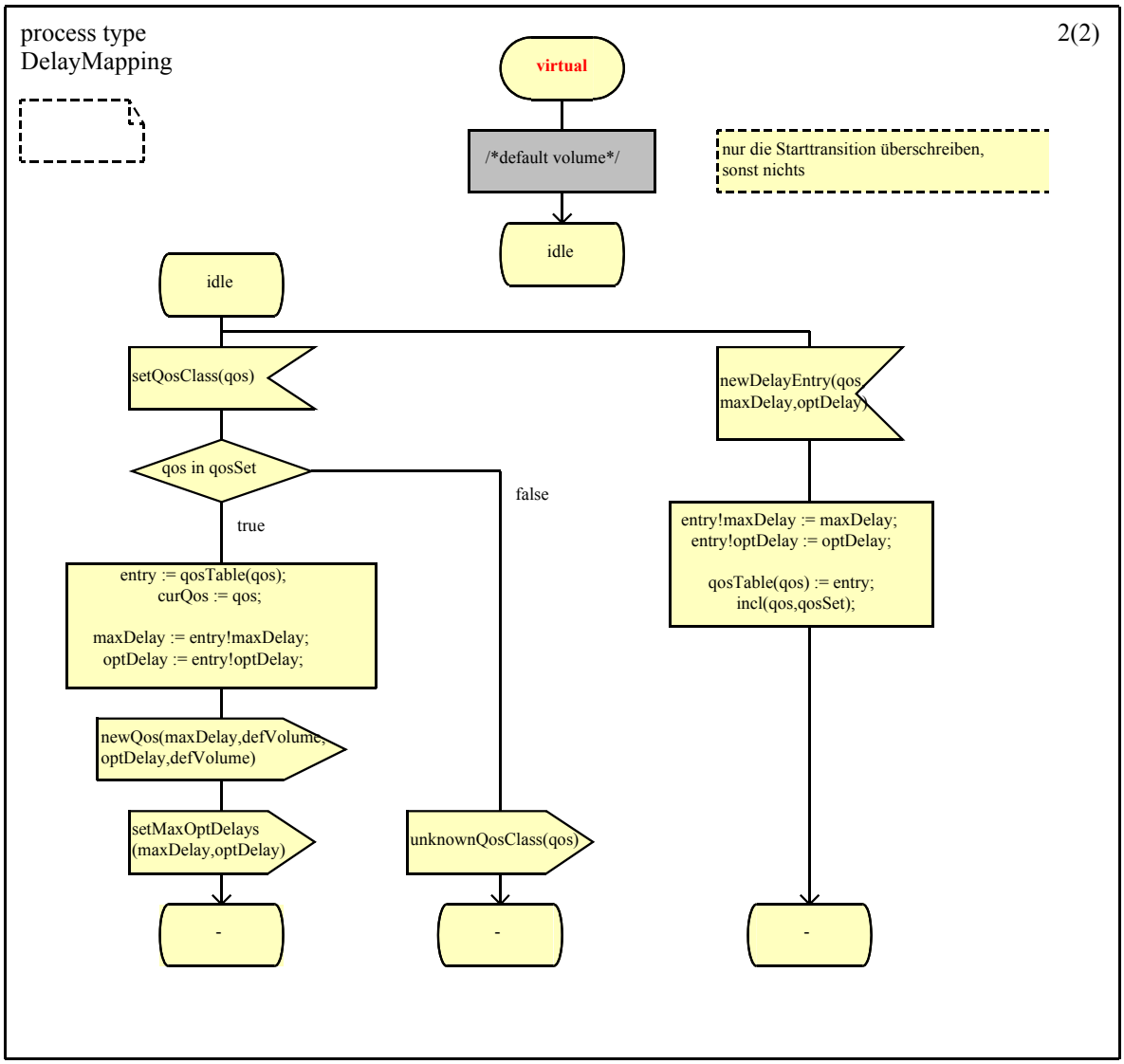


Abb. 5-31: Mikroprotokoll *DelayMapping*: Verhalten

5.1.5.2 Die Prozesstypen *PilotServerMapping* und *PilotClientMapping*

Für die Steuerungsanwendung und für die Übertragung der Luftschiffparameter wird eine Sendeverzögerung reserviert. Aus diesem Grund wurde für beide Anwendungen das vorherige Mikroprotokoll verfeinert.

Der Prozesstyp *PilotClientMapping* sorgt für die korrekte Umsetzung der Dienstgüteklassen der Steuerungsanwendung. Die benötigte Bandbreite wurde gemäß Tabelle 4-4 auf 2kBit/s gesetzt. Die Anwendung muss nur die betreffenden Dienstgüteklassen über das Signal *newDelayEntry* setzten und schon kann das verfeinerte Mikroprotokoll eingesetzt werden.

Der Prozesstyp *PilotServerMapping*, der zur Anwendung zur Übertragung der Luftschiffparameter gehört, ist identisch aufgebaut. Nur benötigt dieser eine Bandbreite von 2kBit/s (siehe Tabelle 4-6). Wichtige Schlüsselwörter des Mechanismus der Verfeinerung in SDL bzw. in Telelogic Tau sind in der folgenden Abbildung **rot** gedruckt.

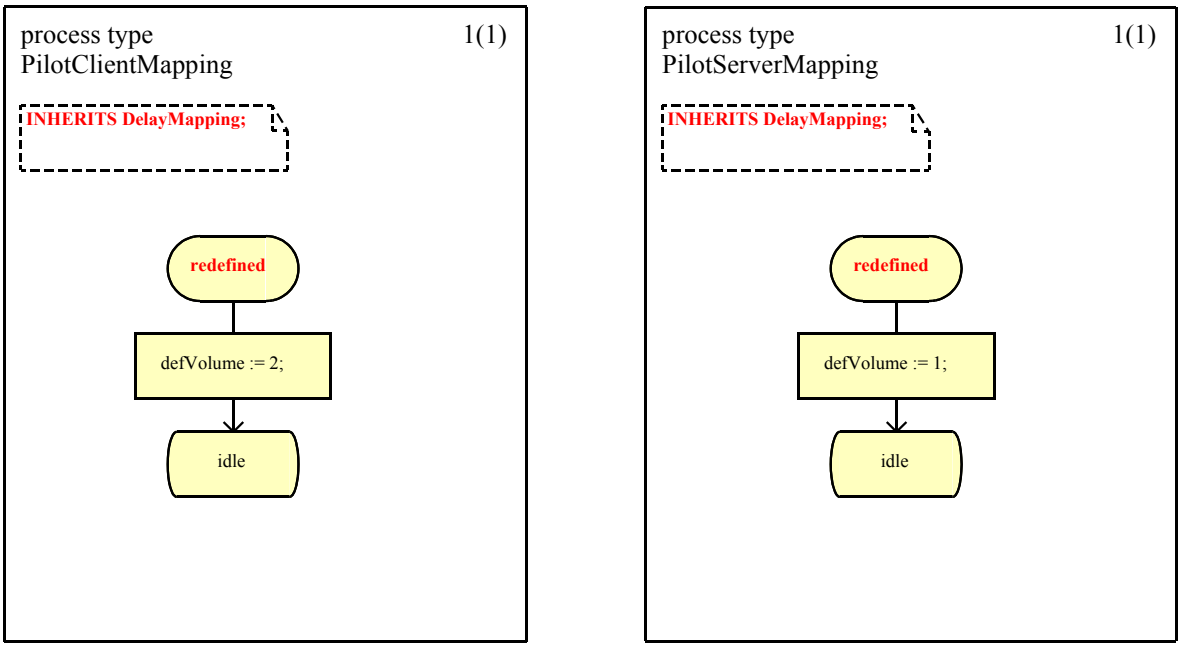


Abb. 5-32: Verfeinerungen des Mikroprotokolls *DelayMapping*

Es ist auch denkbar, für die Festlegung der Bandbreite ein eigenes Signal einzuführen. Nur dann hätten wir keine einheitliche Schnittstelle für alle mappingspezischen Mikroprotokolle. Sowohl des sehr spezifische *VideoMapping* als auch das generischere *DelayMapping* werden von der Anwendung über die selben Signale angesprochen. Eine andere Lösung ist die Verwendung von Kontextparametern. Leider unterstützt das verwendete Tool *Telelogic Tau* diese nur bei dynamischer Prozesserzeugung.

5.1.6 Das semi-generische Mikroprotokoll *DelayScaling*

5.1.6.1 Beschreibung

Eine Dienstgüteskalierung ist in den meisten Fällen eine sehr applikationsspezifische Aufgabe. Da jede Anwendung unterschiedliche Daten bereitstellt, kann für dieses Problem keine generische Lösung erstellt werden. So ist es nicht möglich, eine Komponente, die einen vorliegenden Audiodatenstrom an eine gegebene Bandbreite anpasst, für die Übertragung von Videobildern wiederzuverwenden. Eine solche Skalierung kann nur mit der genauen Kenntnis der vorliegenden Daten und deren Aufbau erfolgen. Unterschiedliche Daten erforden unterschiedliche Kompressionsverfahren, unterschiedliche Datenquellen eine unterschiedliche Justierung (Ansteuerung). Aus diesem Grund ist es nur möglich, für bestimmte Klassen von Anwendungen Mikroprotokolle zu erstellen, die wiederverwendbar sind. In diesem Kapitel wird ein Mikroprotokoll zur Skalierung von Delay-Reservierungen vorgestellt. Da nur Reservierungen auf die Sendeverzögerung getätigt werden, müssen in der Skalierungskomponente die Nutzdaten nicht untersucht oder angepasst werden. In diesem Fall wird davon ausgegangen, dass das Datum nicht verändert werden darf.

DelayScaling ist ein semi-generisches Mikroprotokoll. Es wurde für Anwendungen entwickelt, die eine Reservierung auf die Sendeverzögerung getätigt haben. Dieses Protokoll ist für kleinere Datenmengen gedacht, da für größere Datenaufkommen (> 200kBit) die in Kapitel 4.3.1 beschriebenen Annahmen nicht mehr gelten. Das Mikroprotokoll vermeidet ein unnötiges Datenaufkommen, indem es nur die aktuellen Werte versendet (siehe dazu auch hier Kapitel 4.3.1). Müssen alle Daten, die ein Sender versenden will, auch beim Empfänger ankommen (z.B. ein

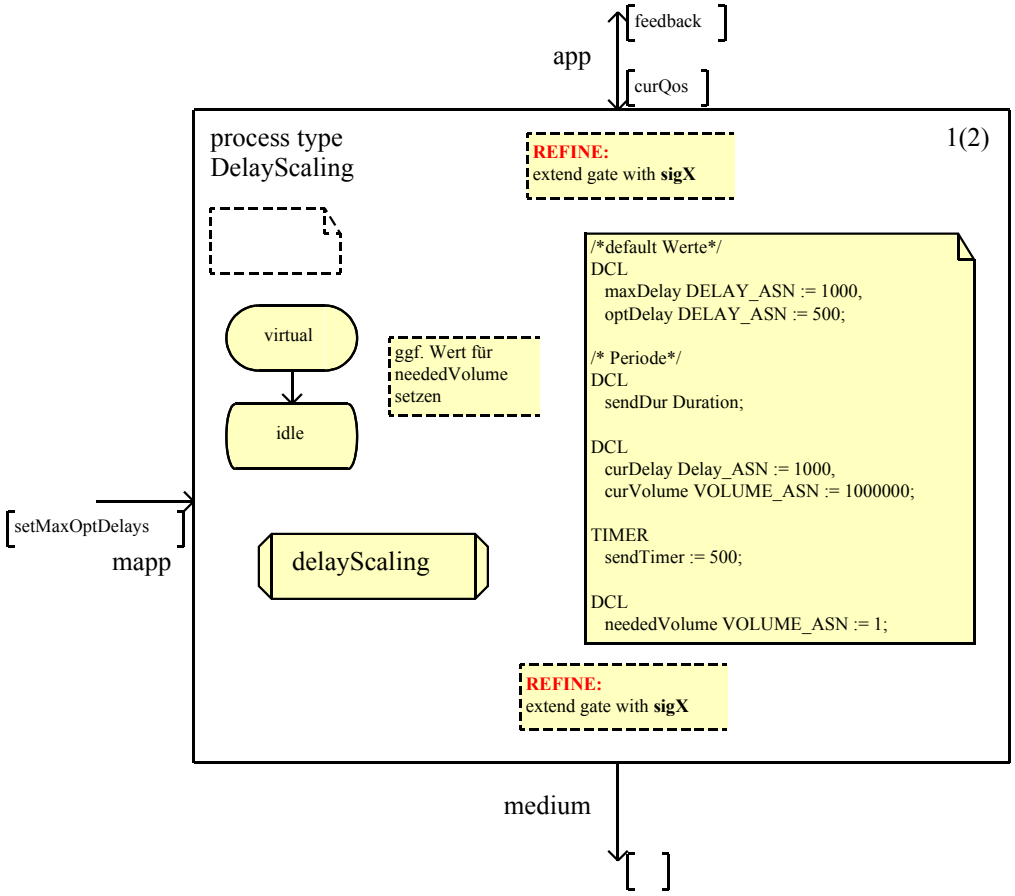


Abb. 5-33: Mikroprotokoll *DelayScaling*: Datentypen und Gates

exaktes Mitloggen von Prozessparametern), so kann dieses Mikroprotokoll nicht verwendet werden. Für ein Senden von Steuerungsparametern ist es ideal, da z.B. wie in unserem Fall der Zeppelin immer auf die aktuellsten Werte eingestellt werden muss. Ein Burst von 5 oder mehr Werten hintereinander haben das selbe Resultat wie ein einmaliges Senden des aktuellsten Wertes, da die Zuweisung der Werte zu den Motoren, also die Änderung der Pulsweitenmodulation, augenblicklich erfolgt.

Die verwendeten Datentypen und die Gates von *DelayScaling* sind Abbildung 5-33 zu sehen. Über das Gate *app* nimmt es die aktuelle Dienstgüte (*curQos*) entgegen und informiert die Anwendung über das Einhalten der aktuellen Dienstgüteklasse mit dem Signal *feedback*. Diese wird zuvor durch *setMaxOptDelays* (Gate *mapp*) festgelegt. Da dieses Mikroprotokoll zwischen Anwendung und Medium sitzt, müssen die beiden Gates *app* und *medium* um das zu skalierende Signal erweitert werden. Das wird durch die Kommentarblöcke mit dem Schlüsselwort REFINE dargestellt. Das Überladen der Starttransition dient genau wie bei *DelayMapping* dazu, das Datenvolumen der Signale zu bestimmen. Für den eigentlichen Skalierungsvorgang wird es zwar nicht gebraucht, jedoch zur Berechnung des Feedbacks. So kann es vorkommen, dass das gewährte Delay für die jeweilige Dienstgüteklasse in Ordnung ist, die gewährte Bandbreite aber zu gering. Dieser Fall ist zwar äußerst selten, aber nicht unmöglich und muss somit auch berücksichtigt werden. Die Prozedur *delayScaling* dient zur Berechnung der aktuellen Sendeperiode und des Feedbacks an den Benutzer über die Einhaltung der aktuellen Dienstgüteklasse. Die reine Sendedauer in einer Single-Hop Umgebung ist zwar sehr gering, darf aber dennoch nicht vernachlässigt werden. Aus diesem Grund ist die aktuelle Sendeperiode etwas kleiner als das gewährte Delay. Die genaue Berechnung des Feedbacks erfolgt nach Abbildung 3-6.

Das Verhalten von *DelayScaling* ist in Abbildung 5-34 zu sehen. Eine Skalierung findet immer

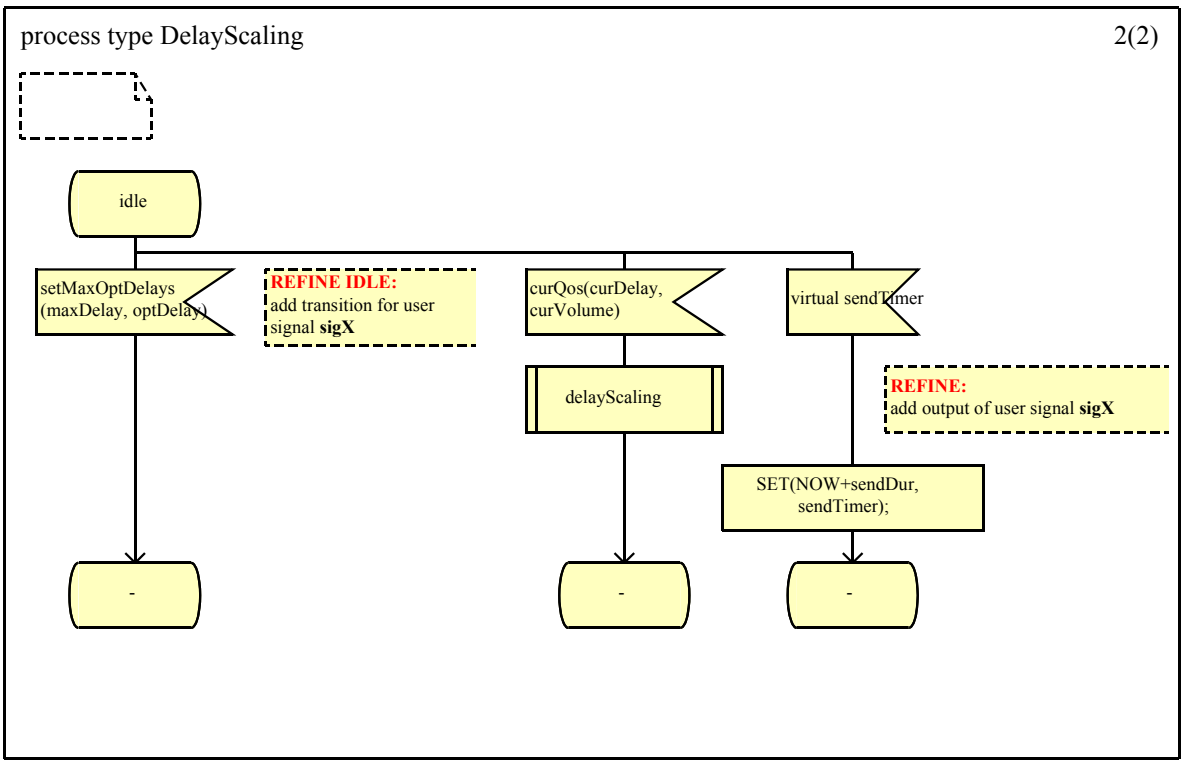


Abb. 5-34: Mikroprotokoll *DelayScaling*: Verhalten

dann statt, wenn sich die die Dienstgütesituation geändert hat. Die beiden wichtigsten Aspekte des Verhaltens sind die *REFINE*-Anweisungen. Damit eine Verfeinerung des Mikroprotokolls korrekt funktioniert, muss dem Zustand *idle* eine Transition hinzugefügt werden, die das zu skalierende Signal inklusive Parameter empfängt. Zusätzlich werden geeignete Variablen benötigt, um die Signalparameter aufzunehmen. Zudem muss die Transition, die den Timeout des Timers *sendTimer* behandelt, auch um des obige Benutzersignal erweitert werden. Nach jedem Empfang eines *curQos* Signals wird der Timer neu gestartet. Aus diesem Grund kann es passieren, dass der Timer abläuft, noch bevor ein Signal *sigX* empfangen wird, sollte entweder die Applikation dafür sorgen zuerst das betreffende Signal *sigX* zu schicken oder die Variablen, die die Signalparameter aufnehmen müssen entsprechend initialisiert werden. Das nächste MSC zeigt einen typischen Ablauf.

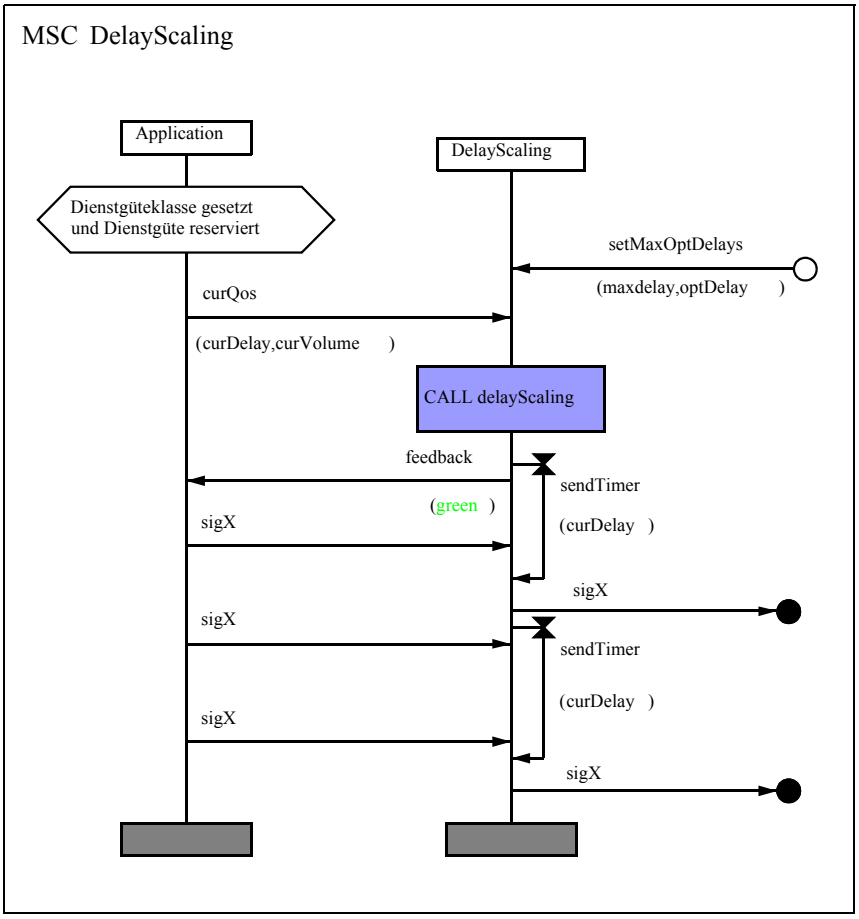


Abb. 5-35: MSC *DelayScaling*

5.1.6.2 Die Prozesstypen *PilotServerScaling* und *PilotClientScaling*

Die Prozesstypen *PilotServerScaling* und *PilotClientScaling* sind eine Verfeinerung von *DelayScaling*. In beiden Fällen müssen kleine periodische Datenaufkommen schnell (bzw. gemäß der aktuellen Dienstgüteklasse) bei einem Empfänger ankommen. Somit ist *DelayScaling* die ideale Generalisierung der zu spezifizierenden Funktionalität.

Der Prozesstyp *PilotServerScaling* (Abb. 5-36) sorgt für die korrekte Skalierung zur Übertragung der Luftschiffparameter. Zu diesem Zweck wurden die beiden Gates *app* und *medium* um das Signal *curCtrlValues* erweitert, welches die aktuellen Parameter beinhaltet. Diese wurden aus genannten Gründen sinnvoll initialisiert. Die Starttransition wurde überschrieben, um die

benötigte Bandbreite festzulegen. In diesem speziellen Fall hätte man auf eine Verfeinerung des Startzustandes verzichten können, da standardmäßig das Datenvolumen von *DelayScaling* auf 1kBit gesetzt ist. Der Vollständigkeit halber und zur besseren Dokumentation wurde jedoch explizit nicht darauf verzichtet. Das Verhalten von *DelayScaling* wurde verfeinert, indem dem Zustand *idle* gemäß den Vorgaben eine neue Transition hinzugefügt wurde. Die Timeout-Transition von *sendTimer* wurde um ein Output-Symbol mit den aktuellen Luftschiffparametern ergänzt.

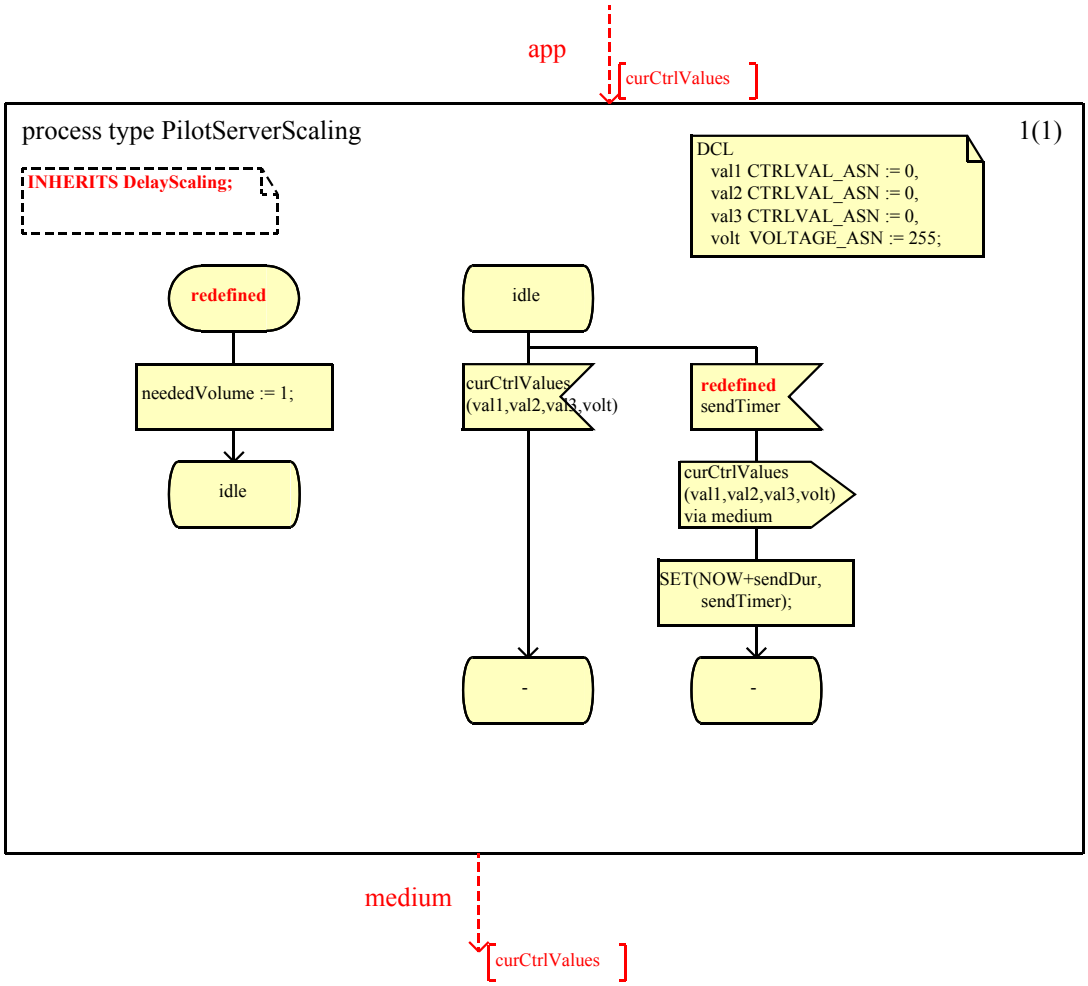


Abb. 5-36: Verfeinerung des Mikroprotokolls *DelayScaling* (1)

Die Skalierung der Steuerungsdaten übernimmt der Prozesstyp *PilotClientScaling* (Abb. 5-37). Genau wie *PilotServerScaling* ist dieser von *DelayScaling* abgeleitet. Jedoch wurde bei *PilotClientScaling* wert auf Performanz gelegt. Da dieser von der Komponente *PSUTrigger* (vgl. Abb. 4-6 und Kap. 5.1.2) nur bei Änderung der aktuellen Steuerungswerte oder nach Ablauf eines 2 Sekunden Timers Steuerungssignal empfängt, ist es sehr ineffizient, nach Ablauf des *sendTimers* das letzte Signal zu wiederholen. Aus diesem Grund wird ein neuer Zustand *newSig* im verfeinerten Mikroprotokoll eingeführt, um feststellen zu können, dass ein zu versendender Wert vorliegt. Der Zustand *newSig* kann nur verlassen werden, falls eine Bedingung auf dem lokalen Zustand erfüllt ist. Diese Bedingung lautet „kein Signal zu versenden und Eingangswarteschlange leer“. Die Variable *s* drückt den ersten Teil der Bedingung aus. Sobald ein neues Signal empfangen wird, wird *s* auf *true* gesetzt. Der zweite Teil der Bedingung wird durch die

Verwendung einer *enabling condition* implizit ausgewertet. Diese kann nur schalten, falls die Eingangswarteschlange leer ist. Alle anderen Erweiterungen von *PilotClientScaling* wurden gemäß den Vorgaben von *DelayScaling* gemacht. Die Gates wurden um das entsprechende Signal *newCtrlvalues* erweitert, die Parameter des Signals sinnvoll initialisiert (Werte für Motoren aus und Servo in Nullstellung) und die benötigte Bandbreite wurde durch das Überschreiben der Starttransition gemäß Tabelle 4-4 gesetzt.

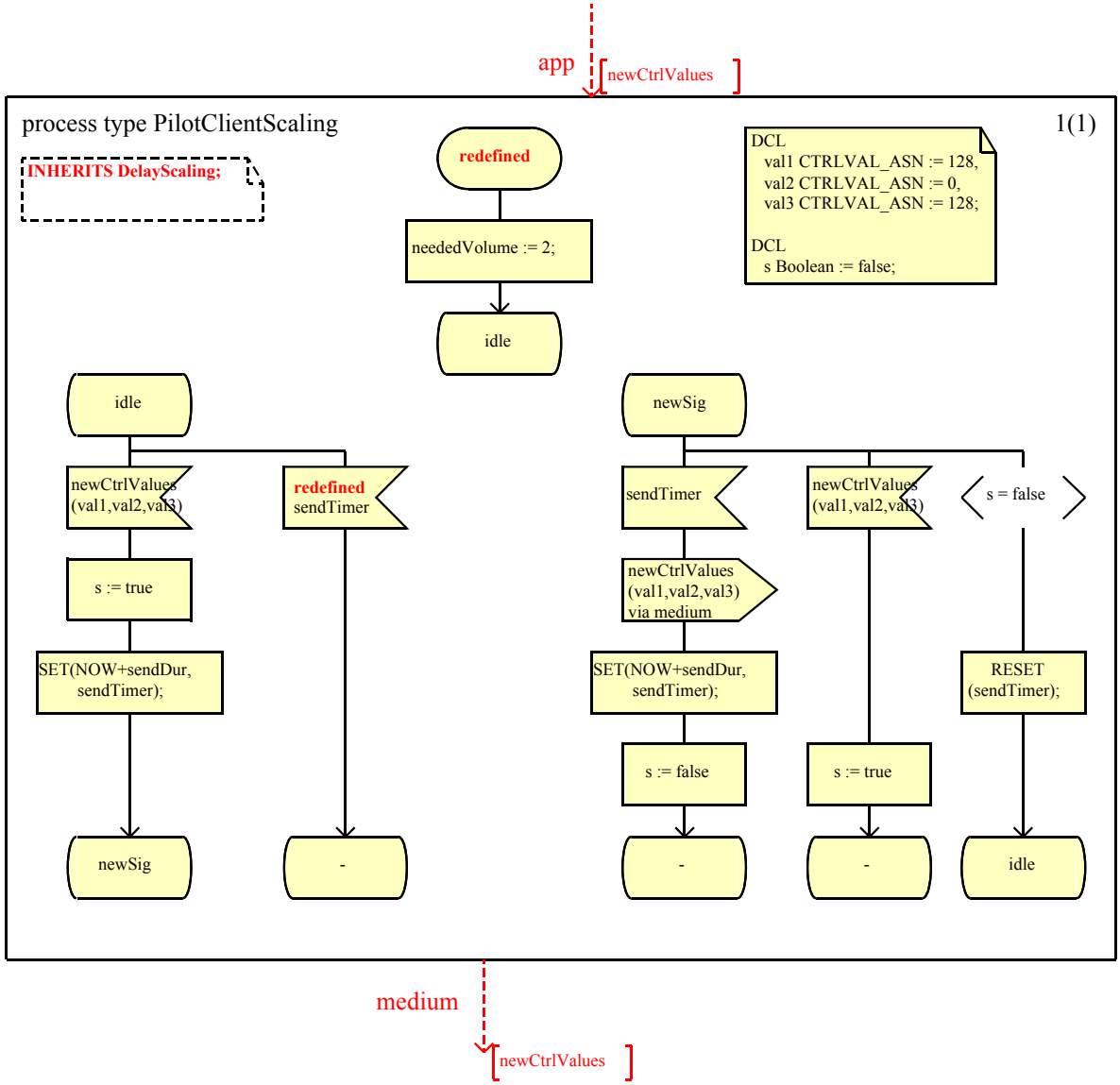


Abb. 5-37: Verfeinerung des Mikroprotokolls *DelayScaling* (2)

5.2 Mikroprotokolle und Design-Patterns der dienstgütespezifischen Middleware

Die dienstgütespezifische oder applikationsunabhängige Middleware wurde in Kapitel 4.8 vorgestellt. Dieser Teil der gesamten Middleware wurde vollständig durch Mikroprotokolle beschrieben. In diesem Kapitel wird auf die verwendeten Mikroprotokolle genauer eingegangen und ihr Verhalten und ihre Schnittstelle beschrieben.

5.2.1 QosReservationLinkMgr

Eine wichtige Aufgabe bzw. Funktionalität einer Middleware mit Dienstgüteunterstützung ist die Reservierung von Ressourcen und die Überwachung der selbigen. Das hier vorgestellte Mikroprotokoll *QosReservationLinkMgr* verwaltet die Reservierungen und Dienstgüteverbindungen einzelner Anwendungen. Es nimmt Reservierungen entgegen und verarbeitet diese. Für die Anwendung ist dieses Mikroprotokoll transparent, d.h. sie tätigt aus ihrer Sicht die Reservierung direkt auf der Basistechnologie und erhält auch von ihr das entsprechende Feedback über die aktuelle Dienstgütesituation. Somit ergibt sich eine weitere Abstraktionsebene. Durch den Einsatz des *QosReservationLinkMgr* wird es möglich, mehrere Dienstgüteapplikation auf einer Middleware aufzusetzen.

Der *QosReservationLinkMgr* besteht wie in Abbildung 5-38 gezeigt aus zwei Teilen, einem *QosReservationMgr* und einem *QosLinkMgr*. Im Gegensatz zu den bisher beschriebenen Mikroprotokollen ist es als Blocktyp realisiert, da dynamische Prozesserzeugung verwendet wird.

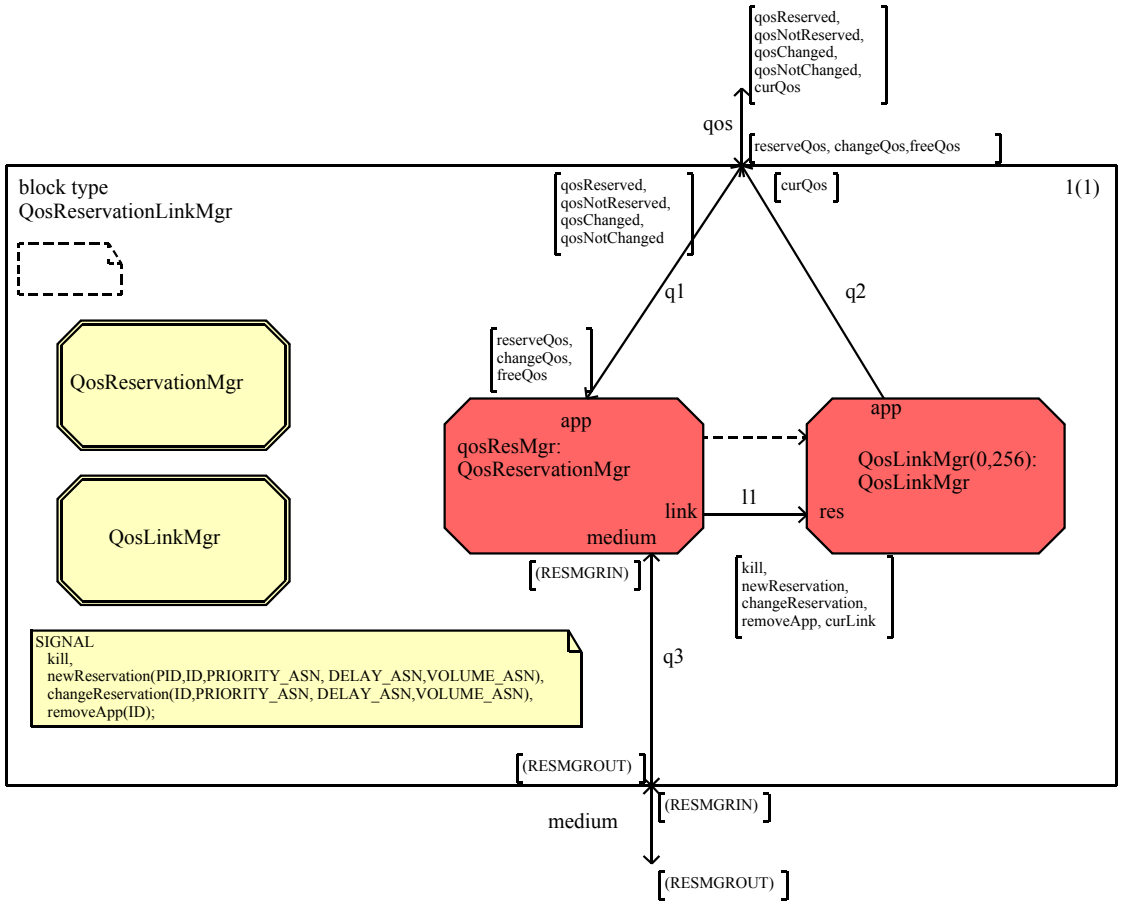


Abb. 5-38: QosReservationLinkMgr

Die Aufgabe des *QosReservationMgr* ist die Reservierung und die Verwaltung von Dienstgüte. Über das Gate *app* werden die Reservierungen der Anwendung entgegengenommen und das entsprechende Feedback gegeben. Die Anwendung kann eine neue Anforderung tätigen (*reserveQos*), eine vorhandene Reservierung ändern (*changeQos*) oder reservierte Dienstgüte freigeben (*freeQos*). Für jede aktive Dienstgüteverbindung wird eine Instanz des *QosLinkMgr* erzeugt. Dieser überwacht die aktuelle Dienstgütesituation auf dem entsprechenden Link und verteilt die verfügbaren Ressourcen gemäß den Prioritäten der einzelnen Anwendungen. Reserviert eine Anwendung z.B. Dienstgüte zu einem Host *zeppelin1* und einem Host *zeppelin2*, so werden zwei *QosLinkMgr lmgr1* und *lmgr2* erzeugt, da zwei aktive Dienstgüteverbindungen vorhanden sind. Tätigt eine zweite Anwendung nun auch eine Reservierung zu Host *zeppelin2*, so wird kein neuer *QosLinkMgr* erzeugt, da auf dieser Verbindung bereits Dienstgüte reserviert ist. *lmgr2* wird über die neue Anwendung und ihre Priorität informiert und kann nun die verfügbare Dienstgüte auf dem Link entsprechend zwischen den beiden Anwendungen aufteilen. In der hier vorgestellten Middleware gibt es genau einen *QosLinkMgr* auf Client- und Serverseite, da wir nur Single-Hop Kommunikation betreiben. Mit dem *QosLinkMgr* wird über das Gate *link* kommuniziert. Die dritte Schnittstelle *medium* dient zur Kommunikation mit dem Basisdienst.

Vorerst bietet das Mikroprotokoll drei verschiedene Arten von Dienstgütereservierung an. Eine Reservierung von Bandbreite, eine Reservierung für Bandbreite und Jitter und eine Reservierung für Delay (Abb. 5-39). Die Reservierung von Bandbreite eignet sich für die Übertragung

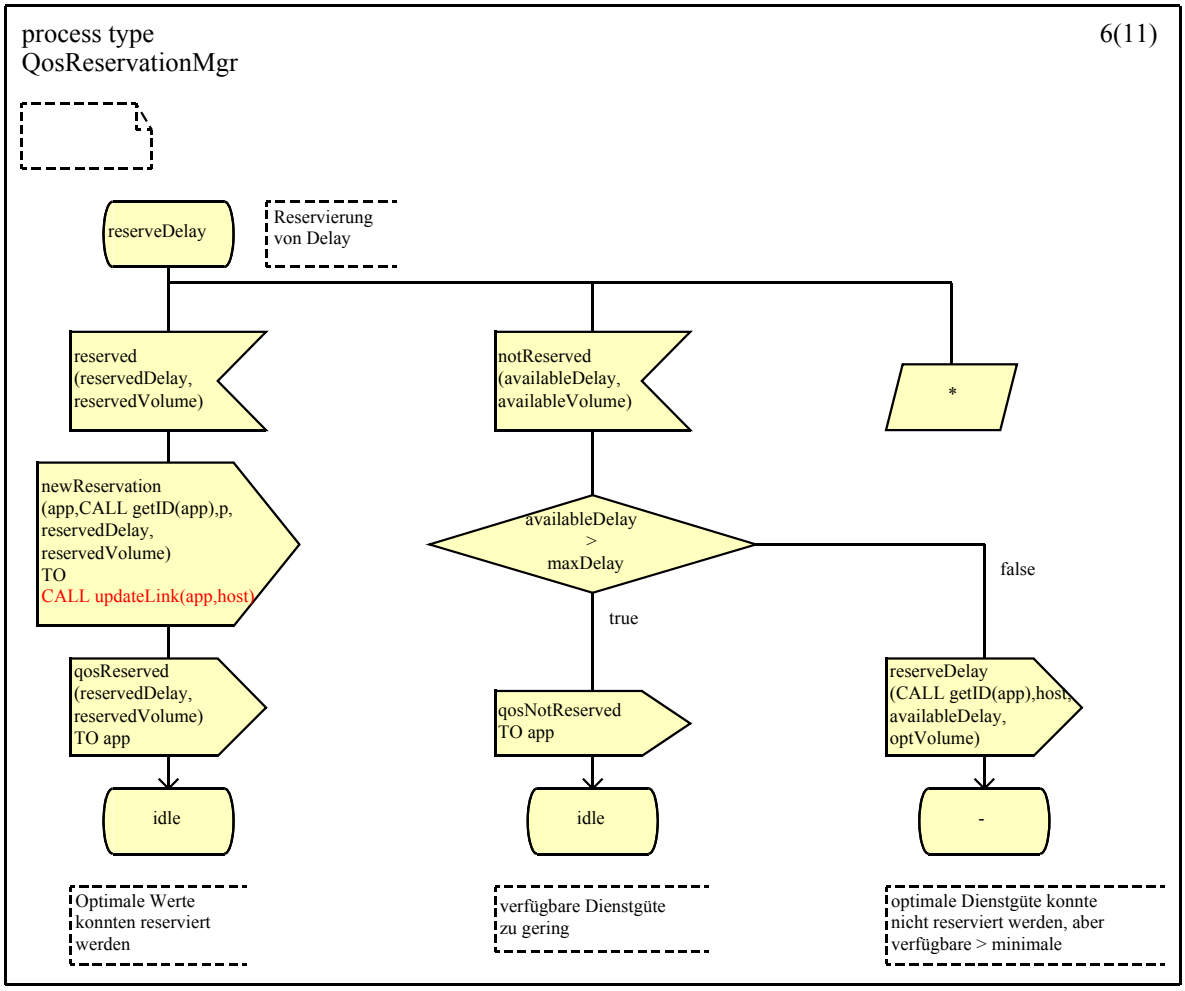


Abb. 5-39: QosReservationMgr

von reinen Nutzdaten im Sinne von Datenbursts, die Reservierung von Bandbreite und Jitter ist gut für isochrone Datenströme geeignet. Hierbei muss jedoch sichergestellt werden, dass die darunterliegende Basistechnologie nicht nur eine reine Reservierung des Jitters tätigt, sondern die zu übertragenden Daten auch gleichmäßig über die Zeitslots verteilt (z.B. jeden 5. Slot und den entsprechenden Jitter reservieren). Die Reservierung von Delay wird für wichtige Daten benötigt, die entsprechend schnell beim Empfänger ankommen sollen. Die Unterscheidung, welche Reservierung genau von der Anwendung getätigt wird, findet auf dieser Schicht mit Hilfe der Signalparameter des Reservierungssignals *reserveQos* statt. Das Signal ist folgendermaßen aufgebaut:

reserveQos(dest,prio,maxDelay,minVol,optDelay,optVol,jitter)

dest gibt den entsprechenden Zielhost an, *prio* die Priorität der Reservierung, *maxDelay,...optVol* die minimale und optimale Dienstgüte der Reservierung, und *jitter* den entsprechenden Jitter. Hier wird keine Unterscheidung zwischen einem minimalen und einem maximalen Wert gemacht. Für eine Reservierung der Bandbreite gilt es, den Jitter auf -1 zu setzen und die Delays gleich zu wählen. Sind die Delays unterschiedlich, so wird eine Sendeverzögerung reserviert und ein Jitter von größer als -1 bedeutet eine Reservierung von Bandbreite und Jitter. Dieser Ansatz zur Reservierung von Dienstgüte dient dazu, den Anwendungen eine möglichst kompakte Schnittstelle anzubieten. In weiteren Studien muss geklärt werden, ob dieses Vorgehen sinnvoll ist.

Abbildung 5-39 zeigt eine Reservierung der Sendeverzögerung. Nach dem Empfangen des entsprechenden *reserveQos*-Signals (nicht in der Abbildung) geht eine Anfrage mit der optimalen Reservierung an die darunterliegende Schicht. Entweder resultiert die Anfrage in einer positiven Quittung (*reserved*) oder in einer negativen (*notReserved*). Bei einer positiven Quittung wird die Anwendung (*qosReserved*) und der betreffende *QosLinkMgr* (*newReservation*) informiert. Die Adresse des *QosLinkMgr* wird zuvor durch die Prozedur *updateLink* ermittelt. In dieser Prozedur werden auch neue LinkMgr angelegt, falls für eine Verbindung zum ersten Mal Dienstgüte reserviert wird.

Eine negative Quittung liefert automatisch die verfügbare Dienstgüte zurück. Ist diese größer als die minimal benötigte, so findet ein erneuter Reservierungsversuch statt, im anderen Fall wird die Anwendung über das Fehlschlagen der Reservierung informiert (*qosNotReserved*). Die Reservierungen der anderen Dienstgüteparameter oder das Ändern einer vorhandenen Reservierung laufen analog dazu ab.

Die angesprochene Prozedur *updateLink* ist in Abbildung 5-40 zu sehen. Die Input-Parameter sind die PID der Applikation und der Zielhost. Die Prozedur liefert die PID des zum Host *addr* gehörenden *QosLinkMgr* zurück. In der Menge *addrSet* befinden sich die Adressen aller aktiven Dienstgüteverbindungen. Ist *addr* nicht in dieser Menge, so muss ein neuer *QosLinkMgr* erzeugt und die lokalen Tabellen mit allen Informationen aktualisiert werden. So enthält die Tabelle *lTable* z.B. die kompletten Daten eines Links (Host, PID, aktive Anwendungen). Ein neuer Link muss zusätzlich beim *QosMonitoring* (Kap. 5.2.2) aktiviert werden.

Befindet sich die Adresse des Links in der Menge *addrSet*, so muss kein neuer *QosLinkMgr* erzeugt werden. Es genügt die PID des dazugehörigen zurückzuliefern und die lokalen Daten zu aktualisieren.

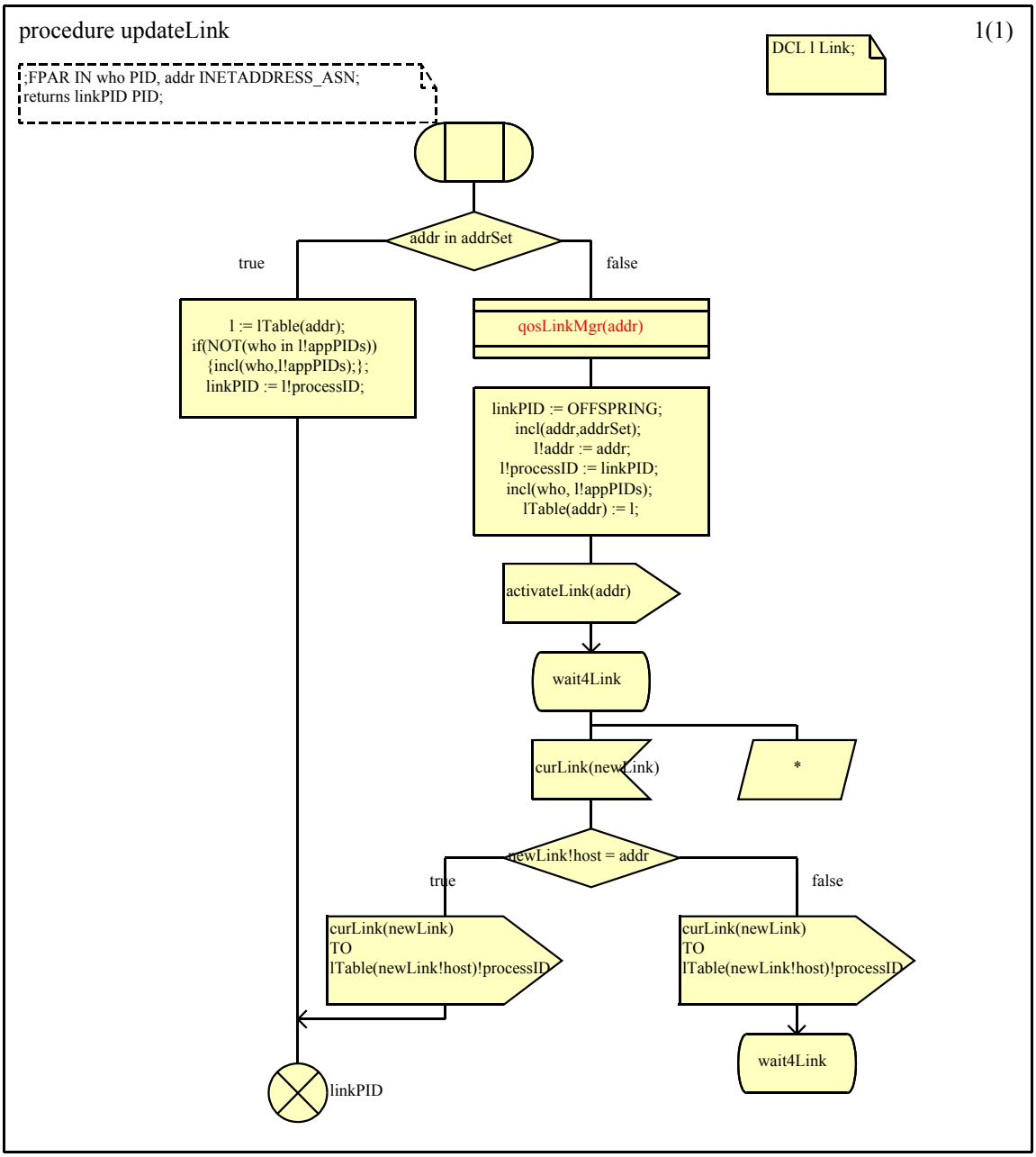


Abb. 5-40: Prozedur *updateLink*

Das folgende MSC zeigt den beschriebenen Ablauf. Nach einer Dienstgüteanforderung durch die Applikation beginnt der *QosReservationMgr* mit dem Reservierungsprotokoll. Das Medium erhält zusätzlich zu den dienstgüterelevanten Daten eine global eindeutige Anwendungs-ID. Dies ist nötig, falls des Basisdienst Zeitschlitze für eine Applikation und nicht für eine Verbindung reserviert. In diesem Fall muss eine eindeutige Zuordnung von Slot zu Applikation gegeben sein. Der *QosLinkMgr* wird danach mit der Zieladresse erzeugt und durch das Signal *newReservation* über die neue Reservierung informiert. Mit dem Signal *curQos* benachrichtigt dieser dann die Anwendung über die aktuelle Situation.

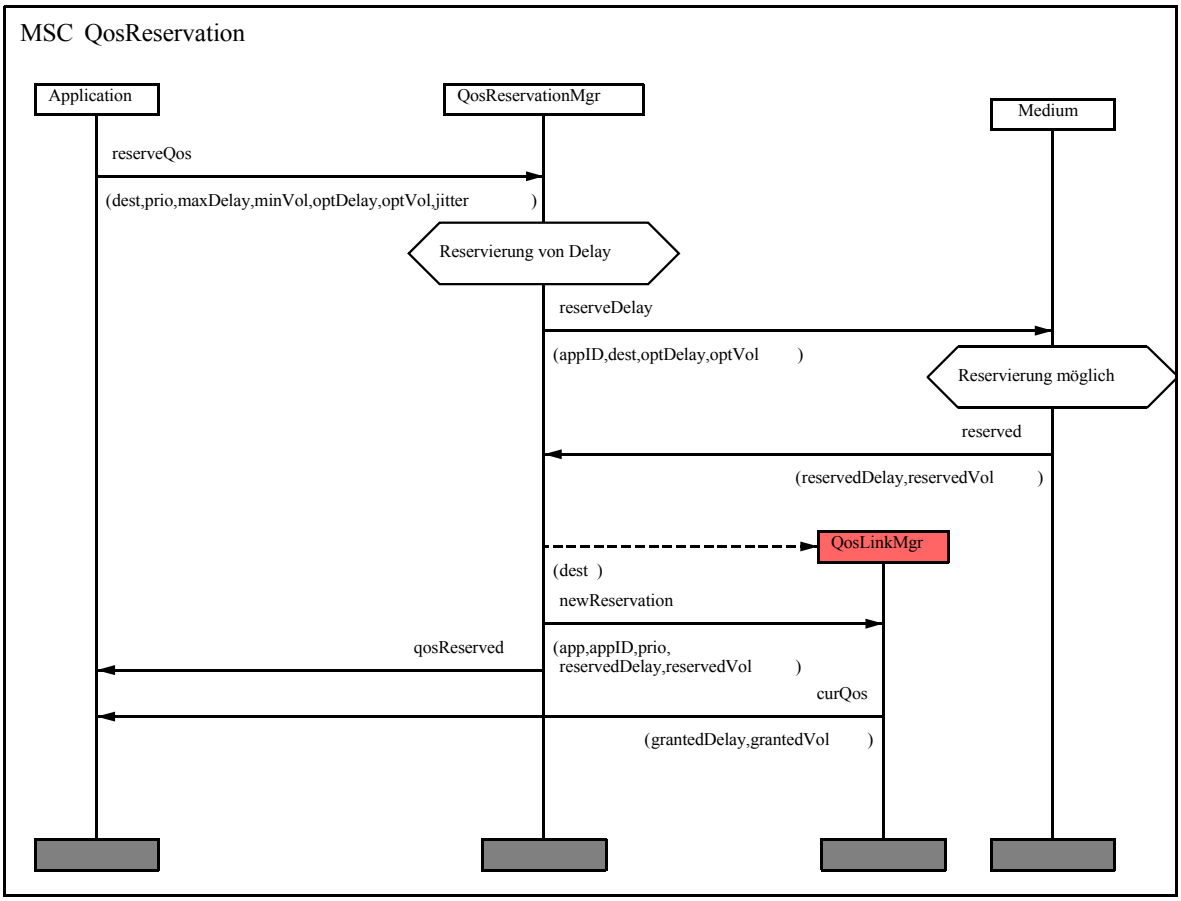


Abb. 5-41: MSC Dienstgütereservierung

Die Hauptaufgabe des *QosLinkMgr* besteht aus der Aufteilung der aktuellen Dienstgüte eines Links auf die dazugehörigen Applikationen. Dazu muss er bei jeder Änderung der aktuellen Dienstgütesituation informiert werden. Abbildung 5-42 beschreibt das Verhalten der Komponente. Die Signale *newReservation, changeReservation* und *removeApp* dienen zur Verwaltung der zugehörigen Applikation. Es können neue Anwendungen hinzugefügt, geändert oder entfernt werden. Das Signal *curLink* informiert den *QosLinkMgr* über die aktuelle Situation. Der genaue Aufbau der ASN.1 Struktur *link* ist in Anhang B zu finden. Der Kern dieser Komponente ist die Prozedur *calcQosForApps*. In ihr wird die verfügbare Dienstgüte gemäß vorgegebener Prioritäten auf die einzelnen Anwendungen verteilt. Der Vorteil der Spezifizierung eines solchen Verteilungsalgorithmus als Prozedur ist die Austauschbarkeit. Man kann den Algorithmus ändern, ohne die Komponente genau zu kennen oder zu verändern. Es ist eine einfache Berechnung implementiert worden, die Ressourcen nur nach Prioritäten verteilt und nicht auf die einzelnen Anforderungen schaut.

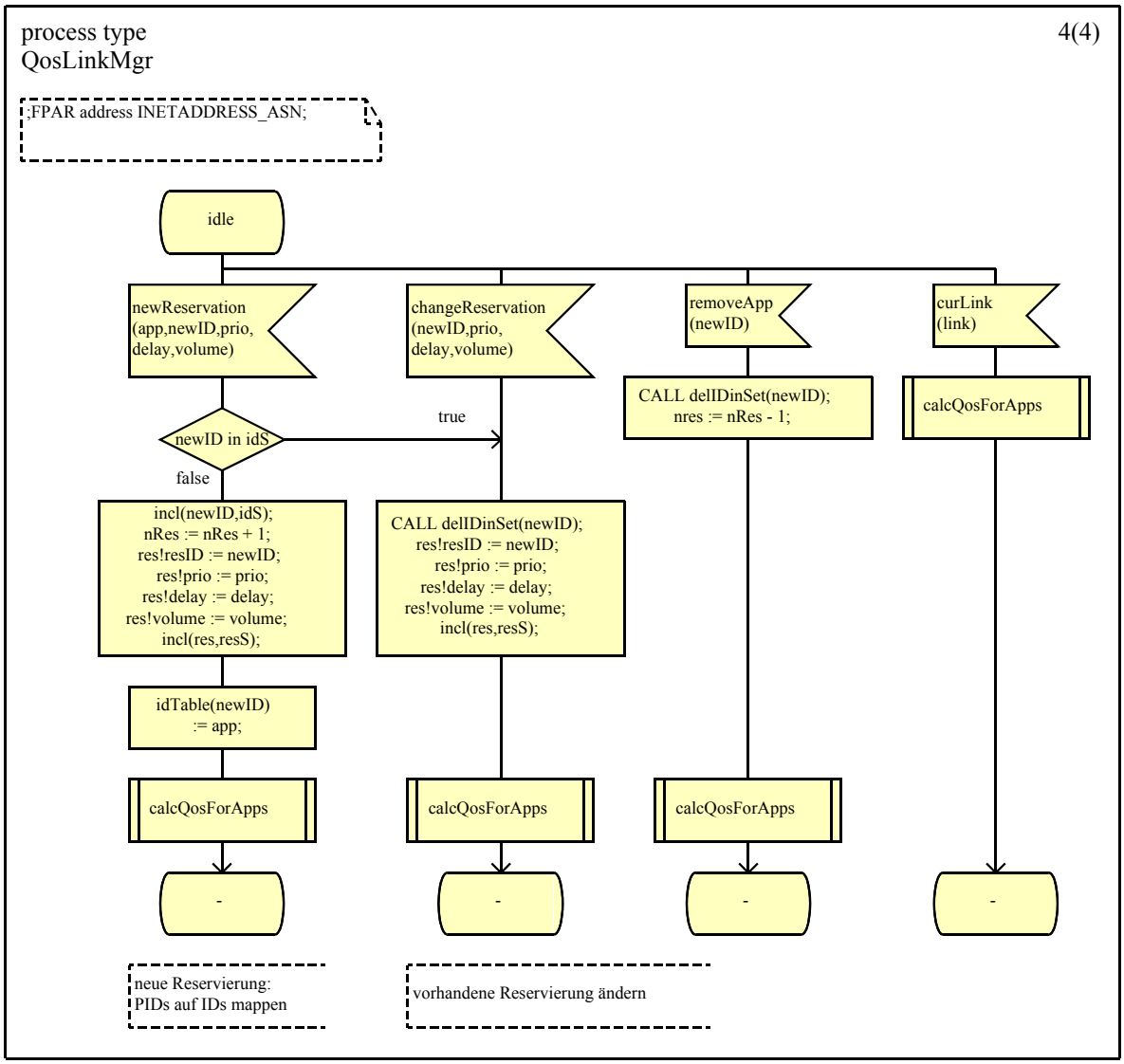


Abb. 5-42: QosLinkMgr

5.2.2 QosMonitoring

Die Dienstgüteerfassung ist ein Hauptmerkmal der Dienstgüteüberwachung. Das Mikroprotokoll *QosMonitoring* überwacht alle Verbindungen eines Host, egal ob Dienstgüte auf ihnen reserviert wurde oder nicht.

Abbildung 5-43 zeigt die verwendeten Datentypen und die Schnittstellen. Über das Gate *app* nimmt es Aktivierungen oder Deaktivierungen für bestehende Verbindungen entgegen. *curLink* liefert die dazugehörigen Daten. Über das Gate *medium* erfolgt die Benachrichtigung über neue Verbindungen. Wird ein Link aktiviert, z.B. von *QosReservationLinkMgr* so bedeutet das für die Dienstgüteerfassung, dass eine Reservierung für Verbindung getätigt wurde, eine Änderung also nach „oben“ mitgeteilt werden muss. Zu diesem Zweck wird bei jeder Änderung des Links umgehend ein *curLink* Signal über *app* versendet. *QosMonitoring* wählt also aus der Menge aller Verbindungen des Host diejenigen aus, auf denen Dienstgüte reserviert wurde und propagiert deren Status, sobald eine Änderung eingetreten ist.

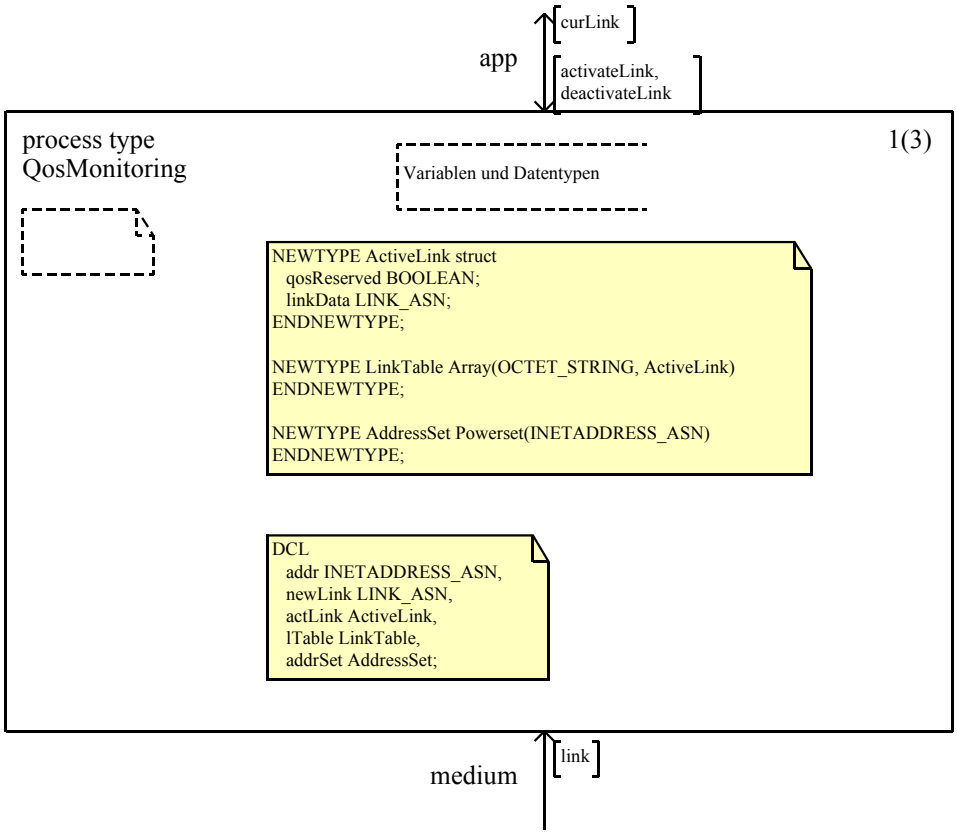


Abb. 5-43: QosMonitoring (1)

5.2.2.1 QosTransferMgr

Der *QosTransferMgr* dient zum Versenden der Daten. Da der verfügbare WLAN-Treiber mit Dienstgüteunterstützung zum einen zu wenige Zeitschlitze bereitstellt und zum anderen es noch nicht festgelegt ist, ob die Belegung der Zeitschlitze des Mediums pro Link oder pro Anwendung erfolgt, kann der *QosTransferMgr* noch nicht spezifiziert werden. In der vorliegenden Version fügt er lediglich den zu sendenden Daten Adressinformationen hinzu und reicht diese ans Medium weiter.

Je nach Entscheidung über die Belegung der Slots kommen noch mehr oder weniger komplexe Funktionalitäten hinzu, wie z.B. das Multiplexen verschiedener Anwendungen auf einen bestimmten Slot.

5.3 Anmerkungen

In diesem Kapitel sind nicht alle Aspekte oder Funktionalitäten der zu entwickelnden Middleware vorgestellt worden. Es wurden lediglich die thematisch zur vorliegenden Arbeit passenden betrachtet, mit Schwerpunkt auf Dienstgüte. Das komplette System mit der vollständig spezifizierten Middleware findet sich im Anhang.

Auch im Bezug auf Mikroprotokolle wurden oft nur die besonders interessanten Aspekte vorgestellt. Auch hier gilt, dass die vollständige Beschreibung jedes Mikroprotokolls in Anhang D zu finden ist. Sie wurden alle vollständig spezifiziert und der Mikroprotokollbibliothek hinzu-

gefügt.

5.4 Zusammenfassung

In diesem Kapitel wurden die entwickelten Mikroprotokolle und Design-Patterns vorgestellt. Es wurden zwei spezifische (*VideoMapping* und *VideoScaling),* zwei semi-generische (*DelayMapping* und *DelayScaling)* und zwei generische Mikroprotokolle (*QosReservationLinkMgr* und *QosMonitoring)* beschrieben und zu einer Kommunikationsmiddleware mit Dienstgüteunterstützung integriert. Erkannte generische Lösungen zu wiederkehrenden Designproblemen wurden in Form von Design-Patterns erfasst.

Desweiteren wurden sämtliche Prozesstypen der Middleware vorgestellt und funktionell beschrieben.

6 Ergebnis und Ausblick

Im Rahmen dieser Arbeit wurde zunächst der Begriff der Dienstgüte und der Begriff der Mikroprotokolle eingeführt. Dabei wurde die Notwendigkeit von Mikroprotokollen und ihr Einsatz in einem modernen Kommunikationssystementwicklungsprozess erläutert.

Im Verlauf dieser Arbeit wurde eine maßgeschneiderte Kommunikationsmiddleware mit Dienstgüteunterstützung entwickelt. Die Hauptbestandteile der Middleware sind Mikroprotokolle, die ebenfalls neu entwickelt wurden. Um die Mikroprotokolle wiederverwenden zu können, wurde ein Template entworfen, welches eine spezifizierungs- bzw. implementierungsunabhängige Sicht auf ein Protokoll liefert. Desweiteren unterstützen die Elemente des Templates die Selektion (*Name, Intent, Interfacing Behaviour*) und die Komposition (*Interface Definition, Structure, Environment Assumptions, Composition*). Zusätzlich dazu gibt es auch Unterstützung für einen Design Review durch speziell auf das Mikroprotokoll angepasste Checklisten (*Checklist*). Um Mikroprotokolle sammeln und aufbewahren zu können, ist eine Bibliothek erstellt worden. Dieser sind die neu erstellten Protokolle hinzugefügt worden. Ferner wurde ein Ansatz zur Komposition von Mikroprotokollen (der Bibliothek) vorgestellt.

Im Einzelnen wurden folgende Protokolle erstellt:

- 2 **spezifische** Mikroprotokolle : *VideoMapping* und *VideoScaling*
- 2 **semi-generische** Mikroprotokolle: *DelayMapping* und *DelayScaling*
- 2 **generische** Mikroprotokolle: *QosReservationLinkMgr* und *QosMonitoring*

Für die Mikroprotokolle wurde eine erste Klassifizierung eingeführt, nach dem Grad ihrer Wiederverwendbarkeit. Die spezifischen Mikroprotokolle sind speziell auf eine Anwendung zugeschnitten, wohingegen die generischen Mikroprotokolle applikationsunabhängige Funktionalitäten kapseln. Die semi-generischen Mikroprotokolle sind auf eine bestimmte Klasse von Anwendungen zugeschnitten und können durch Spezialisierung individuell an die jeweilige Problemstellung angepasst werden.

Desweiteren wurden fünf neue Design-Patterns beschrieben.

Die bisherigen Erfahrungen mit Mikroprotokollen und deren Beschreibung haben gezeigt, dass es sinnvoll wäre, die in Kapitel 2.5.1 neu eingeführte graphische Schnittstellenbeschreibungssprache *IAC* zu erweitern. Mögliche Erweiterungen sind Assertions. Auch empfiehlt sich die Spezifizierung von weiteren Mikroprotokollen um einen vollständigen Systementwurf mit Hilfe von Mikroprotokollen zu ermöglichen, ohne zum Beispiel auf vorhandene Kommunikationsprotokolle des Betriebssystems zurückgreifen zu müssen.

Vor allem im Bereich der Basistechnologie ist noch erhebliche Forschungs- und Entwicklungsarbeit zu leisten. Insbesondere der verwendete WLAN-Treiber mit Dienstgüteunterstützung sollte soweit optimiert werden, dass eine höhere Anzahl von Zeitfenstern unterstützt wird. Erst dadurch wird es ermöglicht, Dienstgüte in einer entsprechend feinen Granularität (Millisekunden und kBit) bereitzustellen.

Der nächste Schritt ist die Erweiterung der Kommunikationsmiddleware von Single-Hop auf Multi-Hop Routing. Dazu müssen geeignete Dienstgüte-Routing und Ressourcenreservierungsprotokolle entwickelt werden.

Ein weiteres mögliches Einsatzgebiet von Mikroprotokollen ist die schnelle Prototyp[1]-Entwicklung (*Rapid Protocol Prototyping*). Ist eine umfangreiche Mikroprotokollbibliothek vorhanden, so können komplexere Protokolle schnell komponiert werden, ohne großen Wert auf Maßschneiderung zu legen. Dieses Vorgehen kann z.B. in der Erstellung von Produktdemonstratoren im Automobil- oder im Mobilkommunikationsbereich Anwendung finden. Diesen Punkt halte ich für besonders interessant, da z.B. Prototypen im automobilen Bereich schon lange nicht mehr nur Motor und Karosserie sind, sondern auch komplexe Kommunikationssysteme beherbergen, die entweder aufgrund des frühen Entwicklungsstadiums oder aufgrund mangelnder Ressourcen (personell oder finanziell) nur von prototypischer Natur sind.

1. ablauffähiges Modell zur Überprüfung von Ideen oder zum Experimentieren (aus [11], S.103)

7 Literatur

[1] P. Jalote: *„An Integrated Approach to Software Engineering“*. Springer Verlag, New York, 1997.

[2] R. Grammes: *„Evaluierung und Anwendung des SDL-Pattern Ansatzes“*. Diploma Thesis, University of Kaiserslautern, 2003

[3] D. Schmidt: „Entwicklung eines WLAN-Treibers mit Dienstgüteunterstützung“. Diploma Thesis, University of Kaiserslautern, 2003

[4] H. Kopetz: *„Real-Time Systems - Design Principles for Distributed Embedded Applications“*. Kluwer Academic Publishers, 1997

[5] Andrew Campbell, Cristina Aurrecoechea, and Linda Hauw: „A review of qos architectures“. In Proceedings of 4th IFIP International Workshop on Quality of Service, Paris, France, March 1996.

[6] J. Larmouth: *„ASN.1 Complete“*. Morgan Kaufmann Publishers, 1999

[7] J. Ellsberger, D. Hogrefe, A. Sarma: *„SDL – Formal Object-oriented Language for Communicating Systems“*. Prentice Hall, 1997.

[8] P. Schaible: „*Wiederverwendungsbasierte Entwicklung von Kommunikationssystemen“*. Ph.D. Thesis, University of Kaiserslautern, 2004.

[9] I. Fliege: *„Definition und Einsatz einer SDL-Mikroprotokollbibliothek und eines Mikroprotokollframeworks am Beispiel des InRes-Systems“*. Diploma Thesis, University of Kaiserslautern, 2003

[10] I. Fliege, A. Geraldy, R. Gotzhein, P. Schaible: *„A Flexible Micro Protocol Framework“*. In Proccedings of 4th SDL and MSC Workshop SAM'04, Ottawa, Canada, June 2004.

[11] H. Balzert: *„Lehrbuch der Software-Technik I, 2.Auflage“*. Spektrum Akademischer Verlag, Heidelberg, Germany, 2000

[12] ITU-T Recommendation Z.100 (11/99): *Specification and Description Language (SDL)*, International Telecommunication Union (ITU), 1999.

[13] ITU-T Recommendation Z.120 (11/99) - *Message Sequence Chart (MSC)*, International Telecommunication Union (ITU), 1999.

[14] B. Geppert: *„The SDL-Pattern Approach - A Reuse-Driven SDL Methodology for Designing Communication Software Systems“*, Ph.D. Thesis, University of Kaiserslautern, Germany, 2000.

[15] IEEE 802.11b, *http://www.ieee.org*, *http://standards.ieee.org/getieee802/*

Anhang A

DESIGN-PATTERNS

SIMPLESIGNALCODECARCHITECTURE

Version 1.2

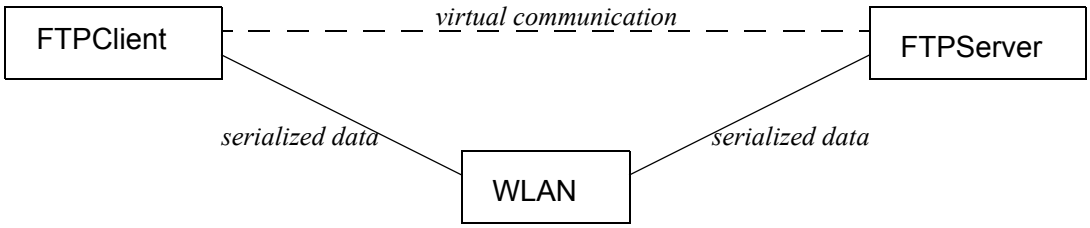
Intent:

The SIMPLESIGNALCODECARCHITECTURE pattern captures the structural aspects of a simple communcation via a (virtual) medium.

Two (or more) applications, which communicate over SDL channels, are to communicate via a given medium (basic service). Therefore the SDL signals have to be encapsulated and transformed to something the medium can handle with.

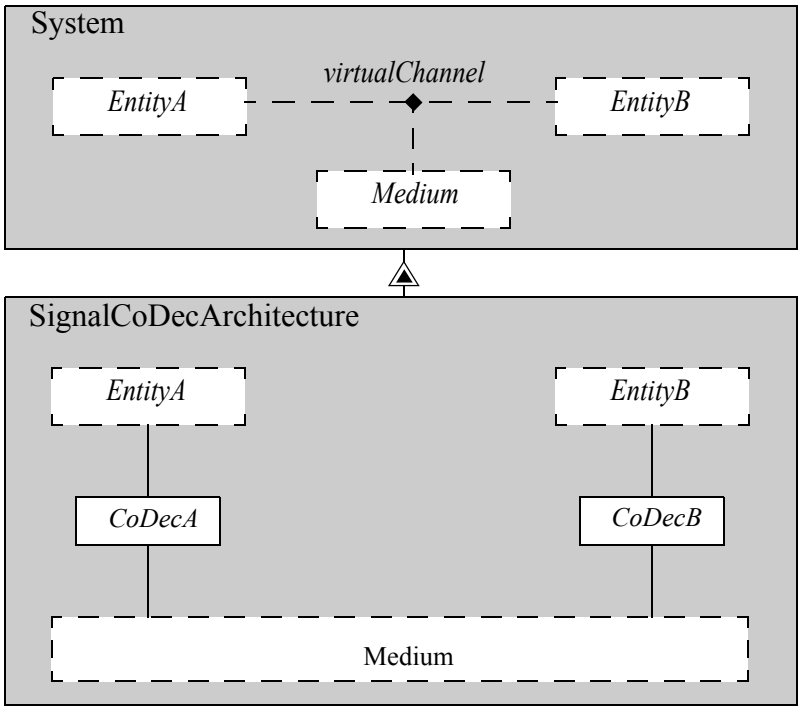
Motivation:

The following shows a typical design problem where the SIMPLESIGNALCODECARCHITECTURE pattern is applicable. In order to communcate via a (virtual) medium it is not possible to hand over the application data to the medium. The different ftp commands (or SDL signals) and the data have to be encapsulated and serialized.



Structure:

The following shows the graphical representation of the structural aspects of the pattern's solution. Simple communication/translation architecture consists of two communication entities which communicate via a virtual channel. The properties of this channel are decribed by a given medium (or basic service). In order to do the encapsulation and transformation to something a medium can handle with, a *CoDec*-component is added.

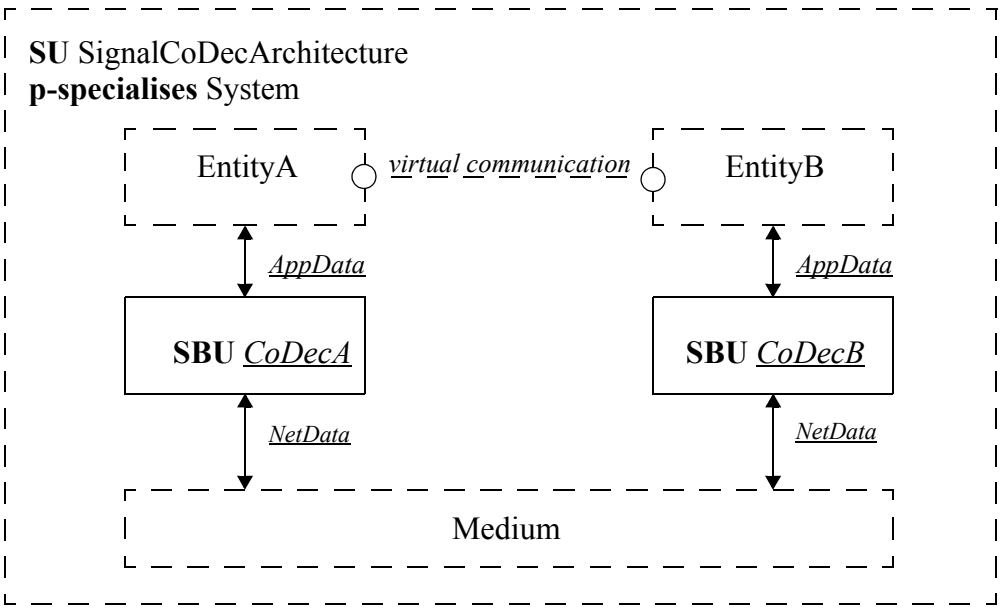


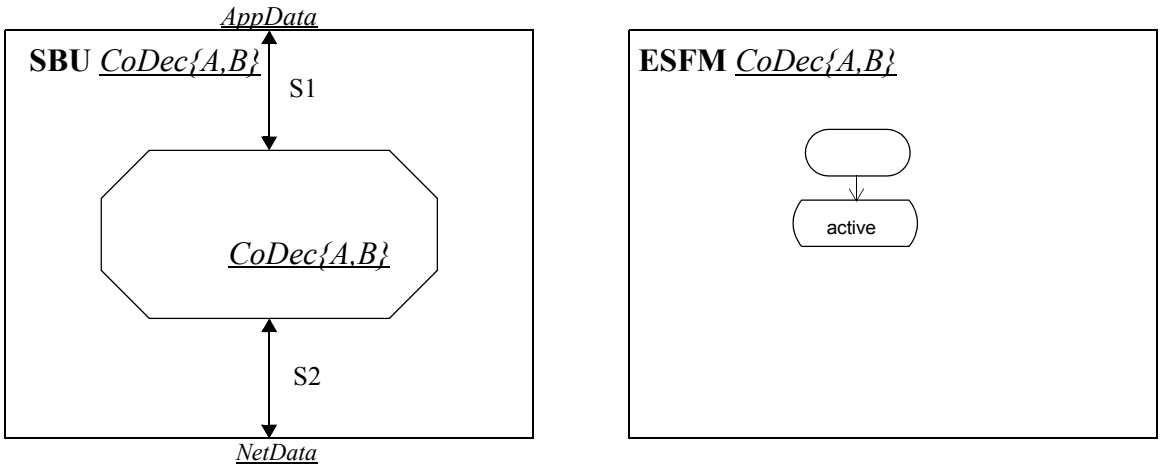
Message Scenario:

The SIMPLESIGNALCODECARCHITECTURE pattern captures only structural aspects of the design problem, therefore no message scenario is supplied.

SDL Fragment:

Applying the pattern introduces one new structural unit per communication component to the context, *CoDecA* and *CoDecB*. A channel connects the units for bi-directional communication. Since the number of components interacting is two, the structural units *CoDecA* and *CoDecB* are refined by exactly one process of the same name.





Syntactical Embedding Rules:

none

Example Application:

- *PilotCoDec* and *VideoCoDec* in the system 'zeppelin', Diploma Thesis C.Webel

Semantic Properties:

Under the assumption

(A-1) EntityA and EntityB produce and consume exactly the same (SDL-)signals (incoming signals = outgoing signals)

The following commitments hold

(C-1) Only one *CoDec*-component is needed

Refinement:

The pattern instance has to be refined in order to add a behaviour to the structure introduced by applying the pattern. The following properties determine refinements of the *SignalCoDecArchitecture* pattern that are considered to be safe:

(R-1) Other structural units which are not involved in the communication between the two communication partners are not affected by the pattern.

(R-2) It is possible to have more than two *CoDec*-components in a system.

Cooperative Usage:

- SIMPLESIGNALCODECBEHAVIOUR: Adds behaviour to the components introduced above.

Known Uses:

- C. Webel: „Entwicklung und Integration von QoS-Mikroprotokollen zur Steuerung eines Fluggerätes über WLAN“, Diploma Thesis, University of Kaiserlsautern, 2004

Checklist

no checklist needed

SDLSIGNALS2ASN.1

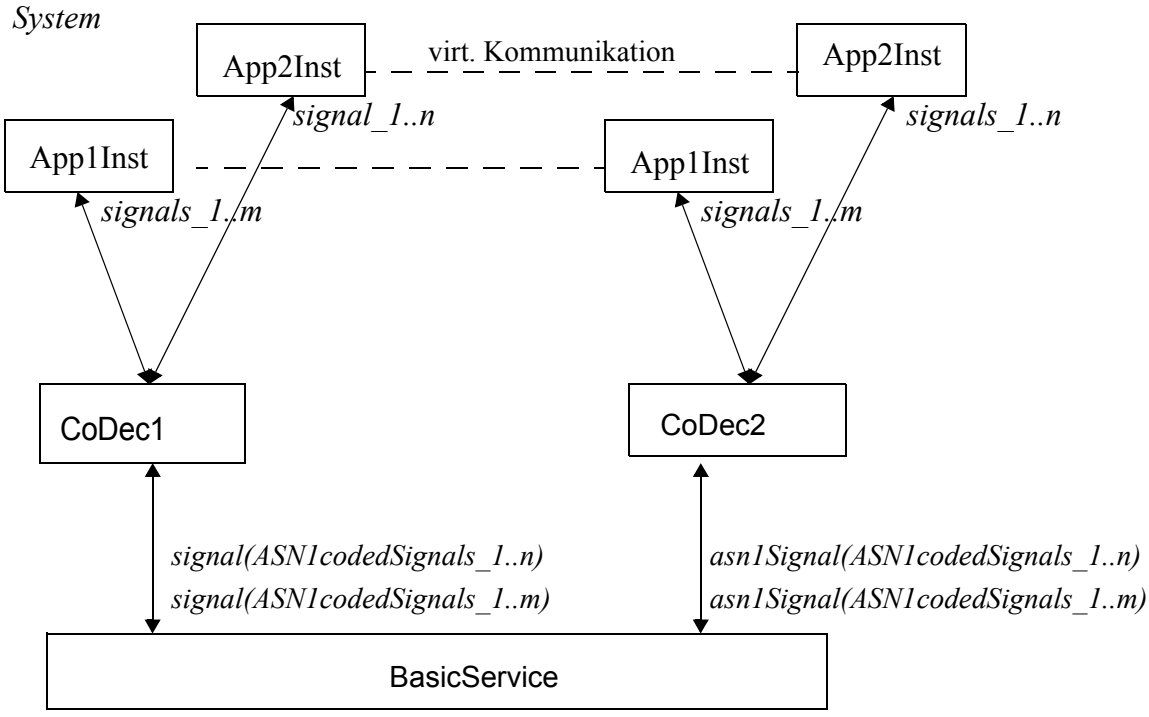
Version 1.3

Intent:

Das SDLSIGNALS2ASN.1 Pattern beschreibt die Umsetzung von SDL-Signalen in eine ASN.1 Datenstruktur. Auf diese Weise können in einer einzigen Struktur sämtliche (Kommunikations-) Informationen gespeichert werden, was zu einer zusätzlichen Abstraktionsebene führt. Es werden mehrere virtuelle Verbindungen über ein Medium unterstützt.

Motivation:

Ein Problem bei der Maßschneiderung von Kommunikationssystemen besteht darin, virtuelle Kommunikation (horizontal) in abstrakte Kommunikation (vertikal) zu überführen. Eine Möglichkeit besteht darin, Daten und Nachrichten geeignet zu kapseln, zu codieren und an die darunterliegende Schicht oder an einen Basisdienst weiterzureichen. Die formale Sprache ASN.1 bietet sich an, solche Nachrichten abstrakt zu beschreiben.

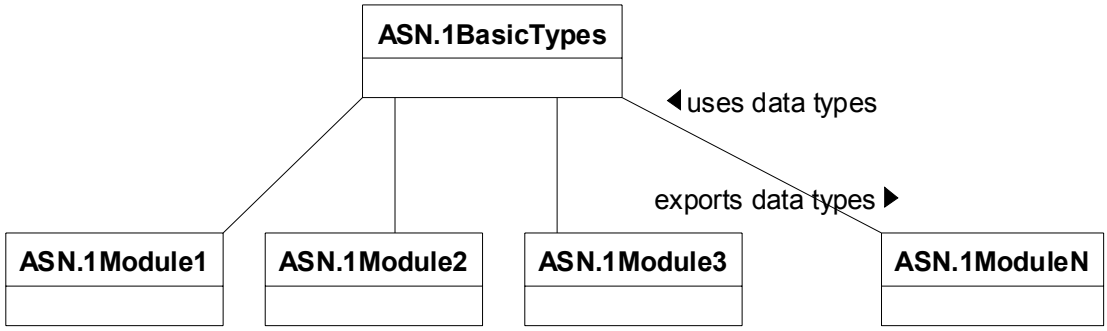


Structure:

Folgende Abbildung zeigt die Struktur bzw. die Abhängigkeiten zwischen den ASN.1 Modulen, die durch die Anwendung des Patterns neu einfegührt wurden. Ein Modul *ASN.1BasicTypes* beinhaltet die Basisdatentypen (Integer, Real, String, ...) und die SDL-spezifischen Syntypes und Strukturen und stellt diese allen ASN.1 Modulen zur Verfügung. Diese

Struktur ist notwendig, um Namenskonflikte im SDL2C-Compiler/Codegenerator zu umgehen.

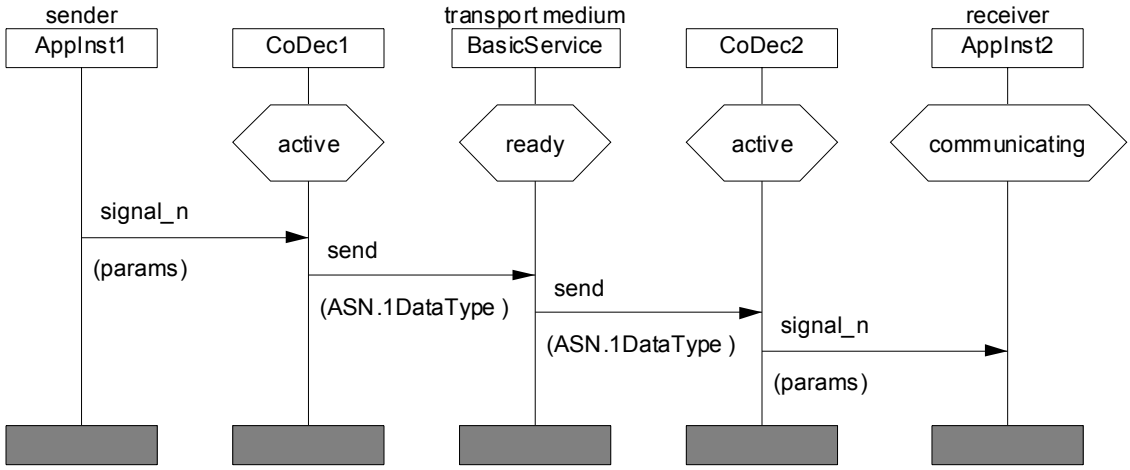
ASN.1 Module Dependencies



Message Scenario:

Folgende Abbildung zeigt ein typisches Szenario, wo die Transformation von SDL-Signalen in eine ASN.1 Datenstruktur notwendig ist, das Pattern daher angewandt werden kann. Ein Sender *AppInst1* kommuniziert über einen virtuellen Kanal mit dem Empfänger *AppInst2*. Die an der Kommunikation beteiligten Signale müssen daher mit den Parametern in eine ASN.1 Struktur gepackt werden. Diese ASN.1 Struktur muss alle benötigten Informationen aufnehmen können. Die ASN.1 Struktur kann nun serialisiert werden, d.h. sie wird in eine Form gebracht, mit der das Medium umgehen kann (z.B. BIT STRING oder OCTET STRING). Die formale Sprache ASN.1 bietet Möglichkeiten zur De- und Encodierung. Die Kapsellung der Signale in die ASN.1 Struktur muss jedoch manuell erfolgen.

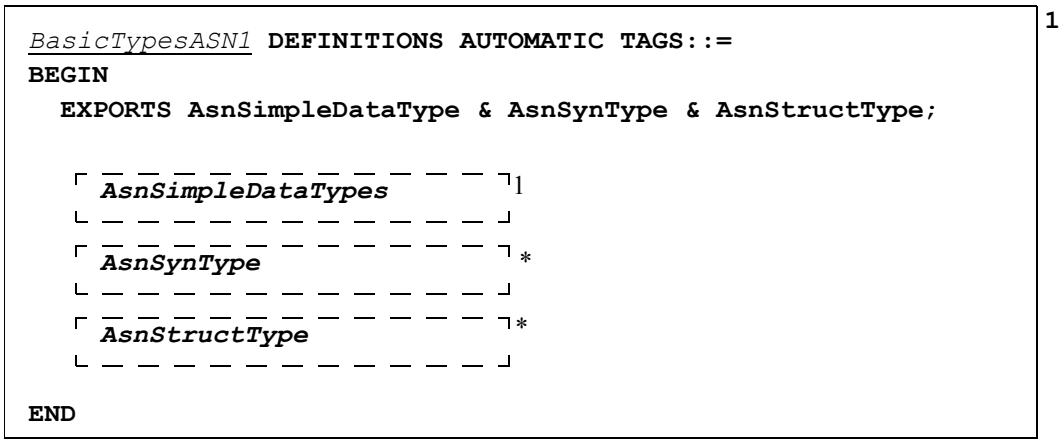
MSC Communication with ASN.1

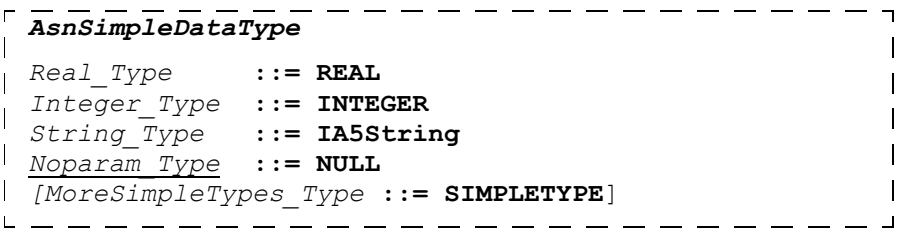


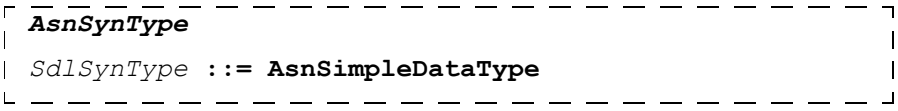
ASN.1 Fragment:

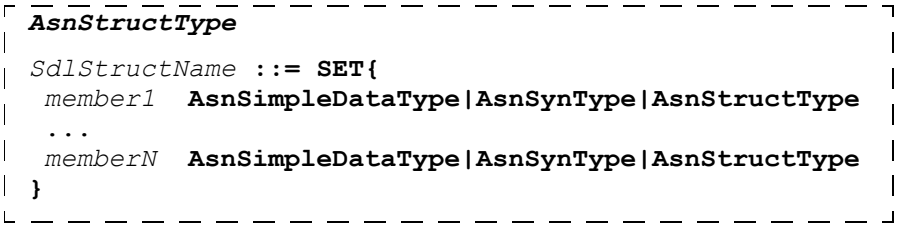
Ein ASN.1 Modul *BasicTypesASN1* stellt die einfachen Datentypen, Synonyme und SDL-Strukturen bereit. Dieses Modul muss alle Datentypen exportieren, um sie anderen Modulen zugänglich zu machen.

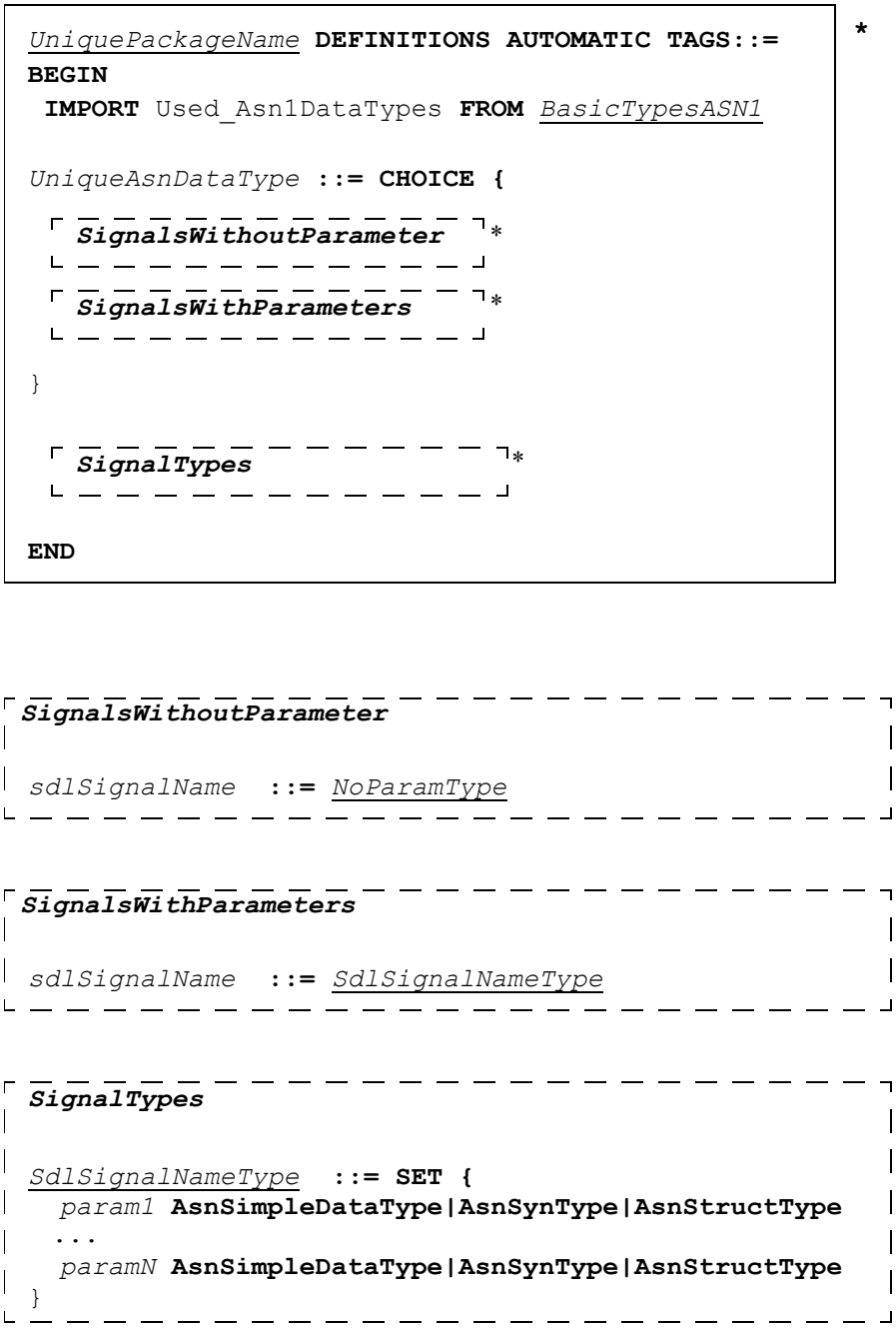
Alle weiteren ASN.1 Module *UniquePackageName* beinhalten eine anwendungsspezifische, komplexere Datenstruktur um die SDL-Signale codieren zu können. Die Module müssen alle benötigten Datentypen importieren.









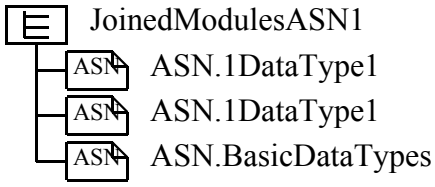


Syntactical Embedding Rules:

Gibt es mehrere SDL-Signale mit gleichem Namen, die sich nur in ihrer Signatur unterscheiden, so sind für diese Signale verschiedene *sdlSignalName*- und somit auch verschiedene *SdlSignalNameType*-Bezeichner zu wählen.

Das ASN.1 Modul BasicTypesASN1 muss sichtbar für alle sein.

Bei der Umsetzung in ein SDL-Package und vor allem bei der Codegenerierung ist auf die Auflösung von Namenskonflikten zu achten. Dazu muss ein neues Organizer-Modul erstellt werden, in diesem dann die ASN.1 Module geeignet gemergt werden



Example Application:

Diplomarbeit C. Webel: „Entwicklung und Integration von QoS-Mikroprotokollen zur Steuerung eines Fluggerätes über WLAN“, Technische Universität Kaiserslautern, 2004

```
/* TYPES und so ...    */

SYNTYPE
  QosClass = Charstring
ENDSYNTYPE;

NEWTYPE Image
  Struct
    data Charstring;
    length Integer; /* in Byte*/
ENDNEWTYPE;
```

```
BasicTypesASN1 DEFINITIONS AUTOMATIC TAGS ::=

BEGIN
EXPORTS   Real_Type,Integer_Type,String_Type,
          NoParam_Type,Image, QosClass;

Real_Type     ::= REAL
Integer_Type  ::= INTEGER
String_Type   ::= IA5String
NoParam_Type  ::= NULL

Image ::= SET{
 data  String_Type,
 length Integer_Type
}
QosClass ::= String_Type
END
```

```
/* Signal Pilot*/

signal newServoValues(Integer, Integer, Integer)

Signallist PCMD = dummy;
```

```
PilotASN1 DEFINITIONS AUTOMATIC TAGS ::=

BEGIN
IMPORTS Integer_Type FROM BasicTypesASN1;

Pilot_Message_Type ::= CHOICE {
   newServoValues NewServoValues_Type
}
NewServoValues_Type ::= SET{
   param1 Integer_Type,
   param2 Integer_Type,
   param3 Integer_Type
}
END
```

```
/* Signal Video*/

signal image(Image);
signal enableCam;
signal disableCam;
signal identifyQosClass(QosClass);
signal qosClassNotSet(QosClass);
signal qosClassSet(QosClass);
```

```
VideoASN1 DEFINITIONS AUTOMATIC TAGS::=

BEGIN
IMPORTS   Integer_Type,Noparam_Type,String_Type,
          Image, QosClass from BasicTypesASN1;

Video_Message_Type ::= CHOICE {
  image             Image_Type,
  identifyQosClass  IdentifyQosClass_Type,
  enableCam         NoParam_Type,
  disableCam        NoParam_Type,
  qosClassNotSet    QosClassNotSet_Type,
  qosClassSet       QosClassSet_Type
}

Image_Type ::= SET{
  param1 Image
}

IdentifyQoSClass_Type ::= SET{
  param1 QosClass
}

QosClassNotSet_Type ::= SET{
  param1 QosClass
}

QosClassSet_Type ::= SET{
  param1 QosClass
}

END
```

Refinement:

keine

Cooperative Usage:

- *SimpleSignalCoDecBehaviour*: Die Kapselung und Codierung in diesem Pattern kann mit ASN.1 realisiert werden.

Known Uses:

C. Webel: „Entwicklung und Integration von QoS-Mikroprotokollen zur Steuerung eines Fluggerätes über WLAN“, Diplomarbeit, Technische Universität Kaiserslautern , 2004

Checklist:

- Sind alle SDL-Signale vorhanden?
- Sind alle SDL-Datentypen korrekt umgesetzt?
- Wurden alle notwendigen ASN.1 Daten exportiert oder importiert?
- Wurden keine doppelten Namen vergeben?

- Sind alle Felder der ASN.1 Struktur vor Benutzung initialisiert?
- Sind uninitialiserte Felder mit dem Schlüsselwort OPTIONAL gekennzeichnet
- Ist das implizite Tagging aktiviert (DEFINITIONS AUTOMATIC TAG)
- Wurde die Groß-/Kleinschreibung beachtet?

SIMPLESIGNALCODECBEHAVIOR

Version 1.2

Intent:

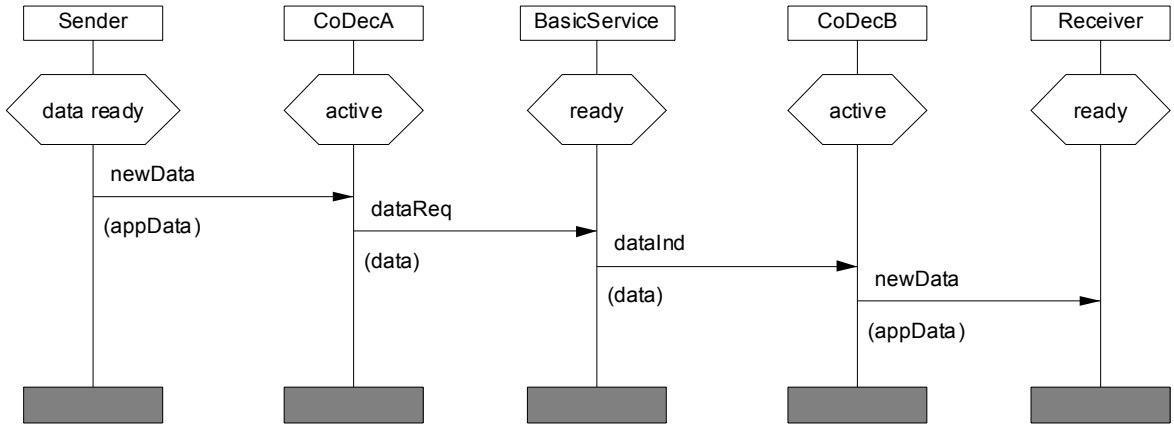
The SIMPLESIGNALCODECBEHAVIOR pattern introduces an translation between different SDL-signals and one signal containing all the necessary data of the SDL-signals and vice versa. The pattern thus captures the translation from virtual to real communication in the context of protocol engineering.

Motivation:

Here is one example from the network system domain where the described design problem arises, which can be solved by the suggested design solution.

- A network component has to send data to another component. Therefore the data has to be serialized before being sent via the medium.

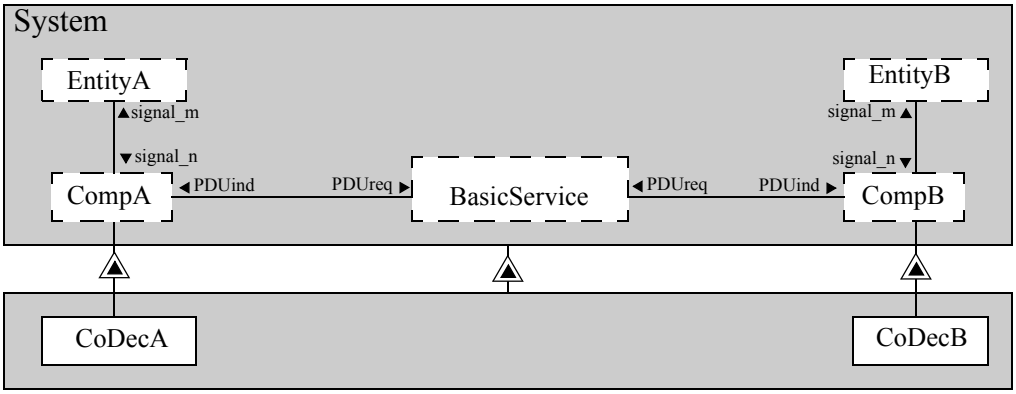
ISC Network Image Transfer



Structure:

The following shows the graphical representation of the structural aspects of the pattern's solution. Two protocol entities *EntityA* and *EntityB* are combined each together with *CompA* and *CompB* entities. These entities are refined by two *CoDecA* and *CoDecB* components. Note that there can be several machines (automatas) that can act as *CoDecA* or *CoDecB* in the SDL desining specification, e.g. several instantiations of the same SDL process type. However, during runtime, only one pair of *CoDecA* and *CoDecB* per time communicate which each other.

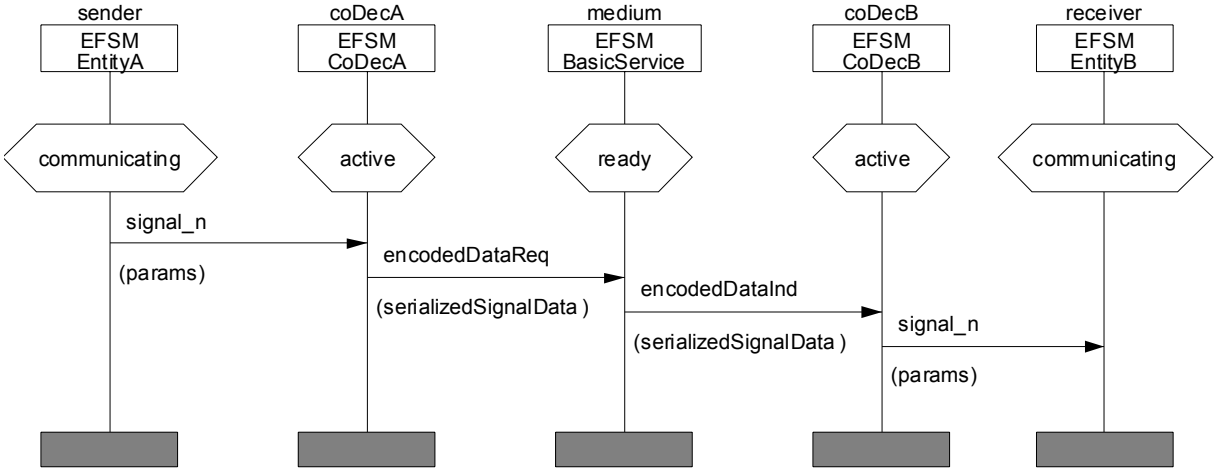
There may exist different implementations of *CoDec{A,B}* which handle different signals of different entities in the system and don't affect each other.



Message Scenario:

The following shows a typical scenario following the *SignalCoDecBehavior* pattern. The peer *sender* and *receiver* is of type *EntityA* and *EntityB*, *medium* is a *basic service* (e.g. Token Ring, WLAN,...). Between these peers the components *CoDecA* and *CoDecB* do the translation and encapsulation.

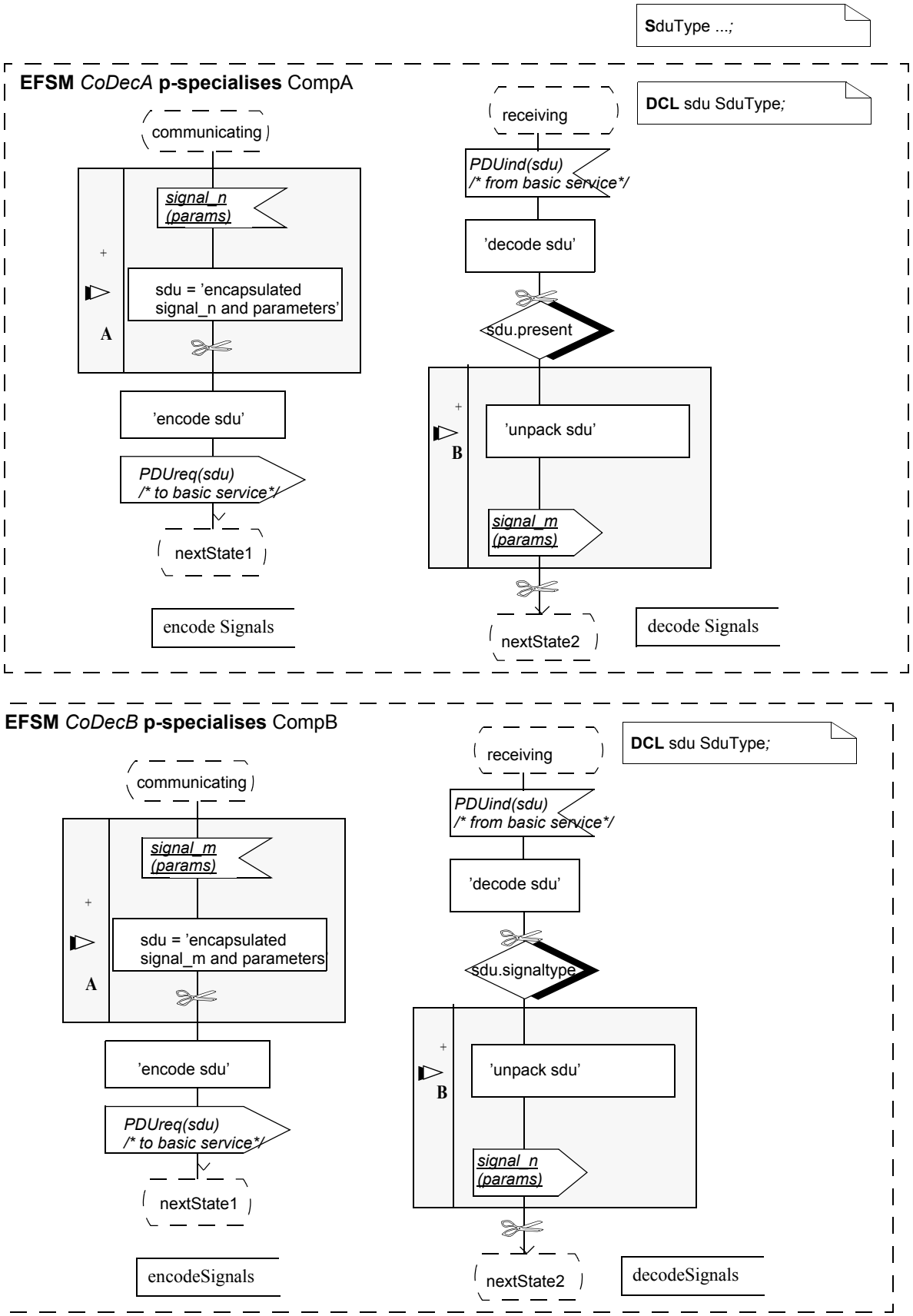
MSC Communication



SDL Fragment:

After being triggered e.g. by a signal *signal_n* from *EntityA* to *EntityB* , *CoDecA* encapsulates the signal in a data structure (e.g. ASN.1), encodes the data structure (serialization) and sends the simple data via the *basic service* to a corresponding *CoDecB* instantiation.

Therefore the original transition is refined by adding the task and the send signal. For the corresponding *CoDec* instantiation there must be at least one state where the message containig the encoded data is received and decoded.


SduType ...;
EFSM CoDecA p-specialises CompA
communicating
signal_n (params)
sdu = 'encapsulated signal_n and parameters'
A
'encode sdu'
PDUreq(sdu) /* to basic service*/
nextState1
encode Signals
receiving
DCL sdu SduType;
PDUind(sdu) /* from basic service*/
'decode sdu'
sdu.present
'unpack sdu'
B
signal_m (params)
nextState2
decode Signals
EFSM CoDecB p-specialises CompB
communicating
signal_m (params)
sdu = 'encapsulated signal_m and parameters'
A
'encode sdu'
PDUreq(sdu) /* to basic service*/
nextState1
encodeSignals
receiving
DCL sdu SduType;
PDUind(sdu) /* from basic service*/
'decode sdu'
sdu.signaltype
'unpack sdu'
B
signal_n (params)
nextState2
decodeSignals

Syntactical Embedding Rules:

In the following, we describe how to instantiate the considered SDL fragments.
-> save symbol in state receiving or receiving = communicating

- **CoDec:**
  - Renaming:
    - The signal *PDUreq* in transition *encodeSignals* may be renamed.
    - The signal *PDUind* in transition *decodeSignals* may be renamed different to signal *PDUreq* in transition *encodeSignals* if it is ensured that the embedding context does a correct re-renaming (e.g. a component which simulates a network medium).
    - The state (set) *communicating, receiving, nextState1, nextState2* are set to the states of the embedding context.
  - Composition:
    - Overlap transitions *encodeSignals* and *decodeSignal* of different SDL **EFSM** *CoDecA* or *CoDecB* and thereby extend block A and B.

Example Application:

- *PilotCoDec* and *VideoCoDec* in the system 'zeppelin', diploma thesis C.Webel

Semantic Properties:

Under the **Assumption** that ...

(A-1) The *PDUind* signal is not implicitly consumed by the respective superclasses.
Sufficient condition: the *PDUind* signal is saved in all states where it is not explicitly consumed.

(A-2) The state (set) *communicating* of *CoDecA/B* will always eventually be reached.
Sufficient condition: *communicating* is the only state of *CoDec.*

... the following **Commitment** holds:

(C-1) Every time *CoDecA/B* sends a *PDUreq* signal a corresponding *CoDecB/A* will eventually receive *PDUind* from *BasicService*.

Refinement:

Normally the pattern instance has to be refined in order to provide appropriate data structures and to encode and decode them (e.g. ASN.1 encode and decode functions). Therefore e.g. the SDLSIGNALS2ASN1 pattern needs to be applied.

Cooperative Usage:

- *SignalCodecArchitecture*: This pattern provides an architecture for the behaviour described above.

Known Uses:

C. Webel: „Entwicklung und Integration von QoS-Mikroprotokollen zur Steuerung eines Fluggerätes über WLAN“, Diploma Thesis, University of Kaiserslautern , 2004

Checklist:

- **CoDecA/B:**
  - *PDUind* and *PDUreq* are properly addressed so that they arrive at the correct process instance.
  - *PDUreq* is always sent exactly once before the next state is entered.
  - *receicving* has a transition for every possible incoming SDL-signaltype *sdu.signaltype* after the decision
  - *communicating* has a transition for every possible incoming SDL-signal *signal_n.*
  - All signals not saved in *receiving* are not needed anymore after the *receiving* state is left
  - *SduType* is proper declared.
  - the state (set) *receiving* is always eventually received and all necessary signals are saved.
  - the state (set) *communicating* is the only state (set) in *CoDecA/B*
  - there is at least one state handlich the *PDUind* signal
- **BasicService:**
  - *PDUreq* has to be renamed to *PDUind* for proper behaviour, if not, *PDUreq* and *PDUind* have to be renamed to the same signal in *CoDecA/B*

WATCHDOG

Version 1.1

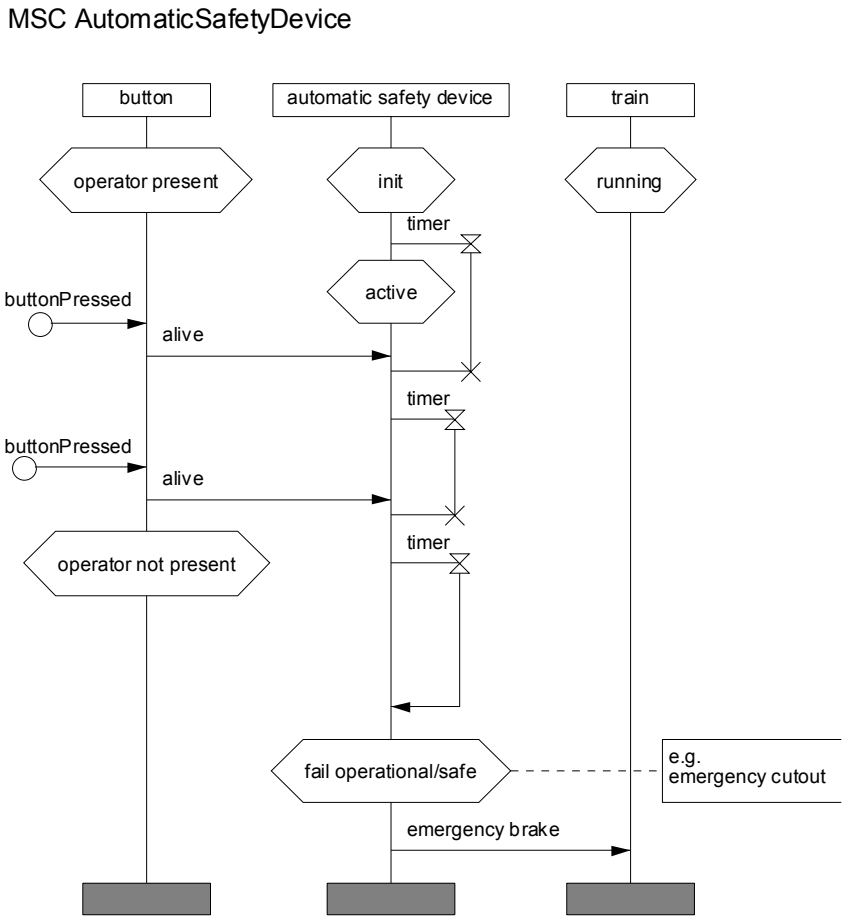
Intent:

The WATCHDOG pattern realizes a safety functionality generally known as a watchdog. A component (system) has to be triggered periodically in order to keep on running (active) otherwise a fail-operational or fail-safe state is reached.

Motivation:

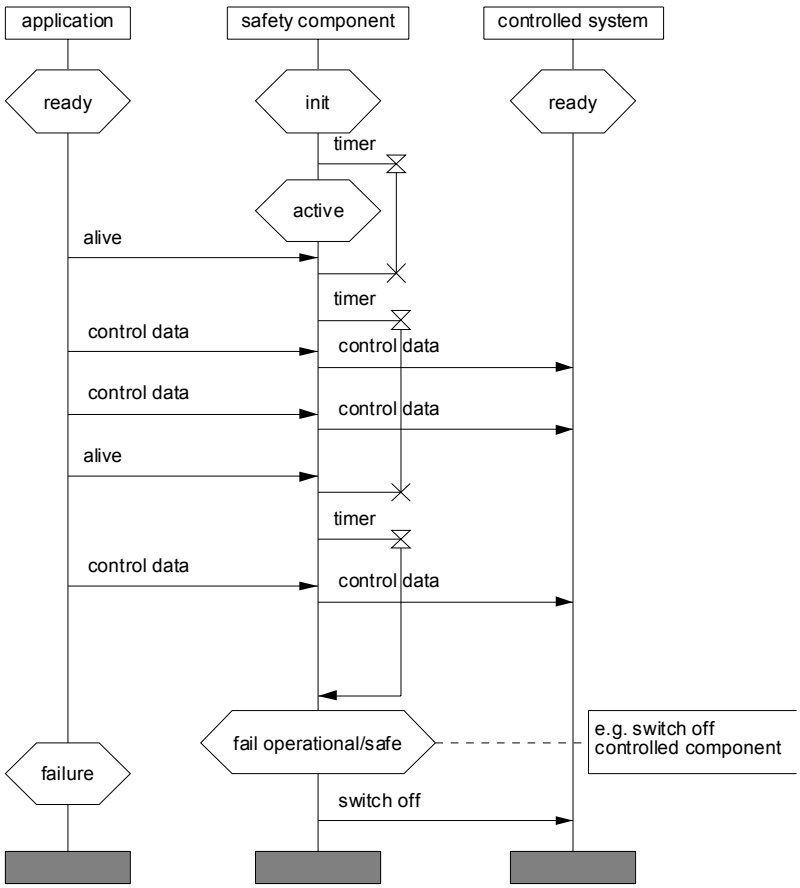
Here are some examples where the described design problem arises, which can be solved by the suggested solution.

- Automatic safety device (dead man's control):
  An operator (e.g. a train conducter) has to activate periodically a button or switch in accordance with a prior well- defined time interval.



- Application with safety aspects:
  An application periodically sends an alive signal to a safety component to propagate its 'running' state. If this signal fails to appear, the safety component has to switch to a fail-operational or fail-safe state (e.g. switch off controlled system).

MSC Application with safety aspects

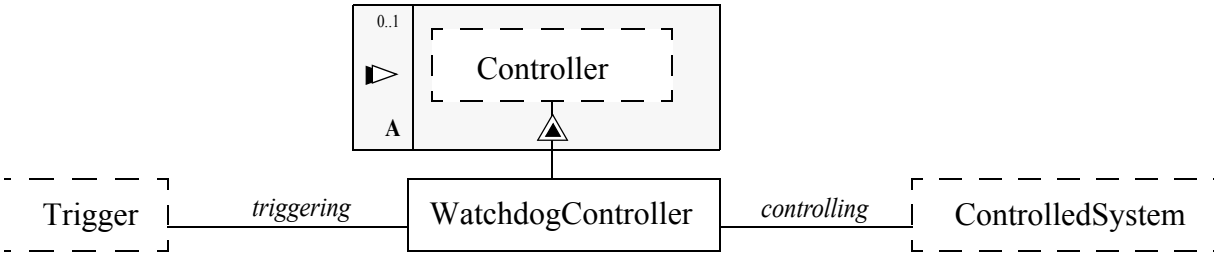


Structure:

The following shows the graphical representation of the strutural aspects of the pattern's solution. Note that *WatchdogController* not necessarily has to refine a component from the context. It is also possible to newly add this component to the structure.

Trigger is a component from the context, which provides a peridocal alive signal. It can be a button or a switch (environment) or any kind of system.

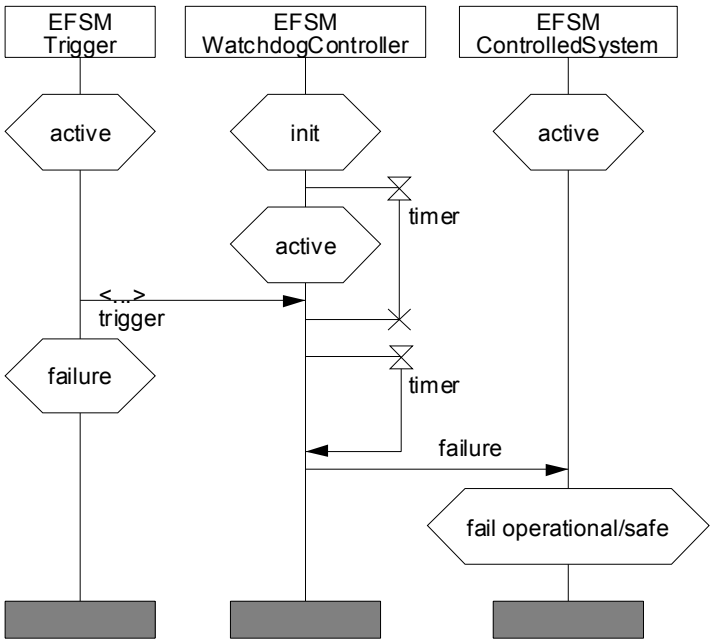
Controller is, where necessary, refined by *WatchdogController* and can be for example some controll system where watchdog functionality should be added.



Message Scenario:

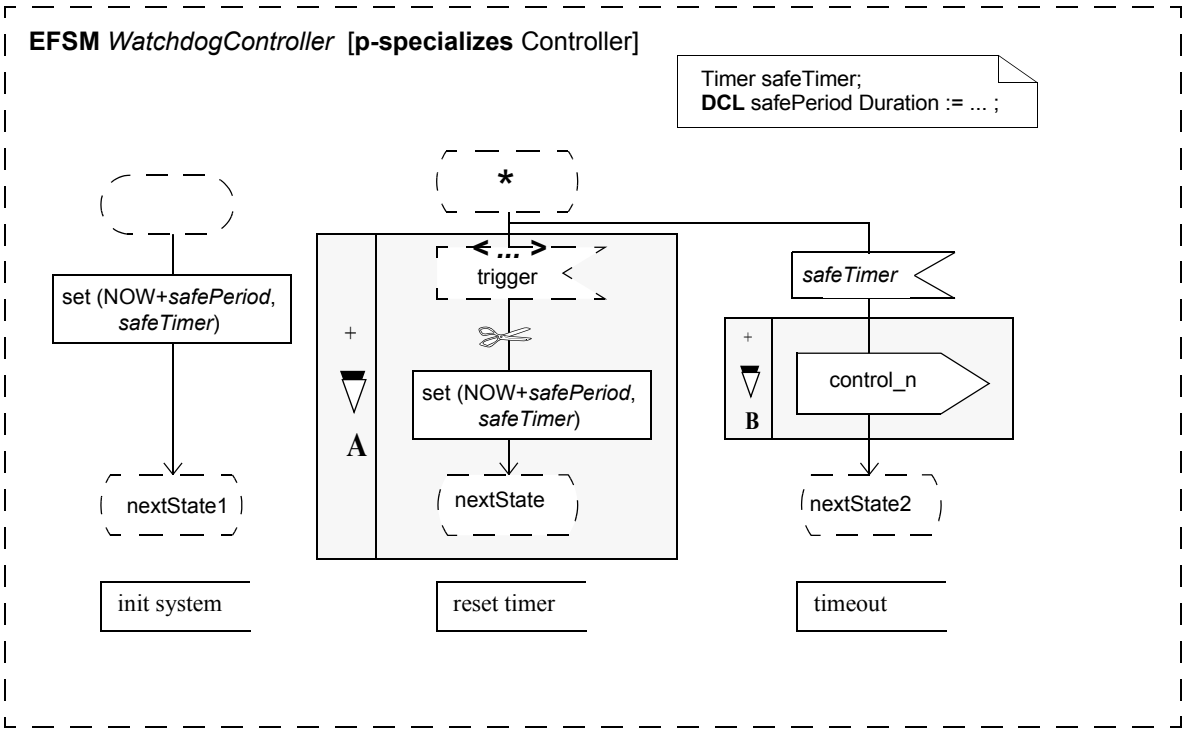
The following shows a typical generic usage scenario between the components described above.

MSC Watchdog



SDL Fragment:

After being triggered, *WatchdogController* resets its timer to a given period and continues working. After receiving a timeout, *WatchdogController* has to ensure, that *ControlledSystem* reaches a safe state (e.g. by sending one ore more control signals). If *Trigger* becomes active again (after a failure), WatchdogController has to set its timer to a given period in order to resume controlling.



Syntactical Embedding Rules:

In the following, we describe how to instantiate the considered SDL fragments.

- **WatchdogController:**
  - Variants:
    - Resolve the trigger symbol - this could result in an input symbol, conditioned input, continuous signal, or an input symbol followed by a condition symbol.
    - If no component *Controller* is refined (thus *WatchdogController* is added to the structure) all symbols and states from the context have to be solved and added.
  - Renaming:
    - If refinement is provided, the state (set) *nextState, nextState1, nextState2* and the signal/variable *trigger* are set to the states/signals/variables of the embedding context.
    - If no refinement is provided, the state (set) *nextState, nextState1, nextState2* and * are to set to a new state and the signal/variable *trigger* is set to an 'alive' signal send by the component *Trigger.*
    - The signals *control_n* are to set to suitable names of the embedding context (*ControlledSystem*) to reach a safe state.

Example Application:

- *PSU* in the system 'zeppelin', diploma thesis C.Webel

Semantic Properties:

Under the **Assumption** that ...

(A-1) The state (set) *active* of *WatchdogController* will always eventually be reached.
Sufficient condition: *active* is the only state of *WatchdogController.*

... the following **Commitment** holds:

(C-1) Every time *trigger* arises, the timer *safeTimer* is reset.

(C-2) Every time a timeout arises, the timer *safeTimer* is not reset and suitable signals are sent to *ControlledSystem*.

Refinement:

No Refinement needed.

Cooperative Usage:

None.

Known Uses:

C. Webel: „Entwicklung und Integration von QoS-Mikroprotokollen zur Steuerung eines Fluggerätes über WLAN“, Diploma Thesis, University of Kaiserslautern , 2004

Checklist

- **WatchdogController:**
  - After receiving a trigger, *safeTimer* is set.
  - After receiving a timeout, *safeTimer* is not set.
  - *reset timer* has a transition for every possible trigger signal
  - *safePeriod* is set to a suitable value.
  - *control_n* is always sent after a timeout.
  - the signals *control_n* lead to a fail-safe/operational state of the controlled component.

QOSMAPPING

Version 1.0

Intent:

Das Design-Pattern QOSMAPPING beschreibt die Umsetzung von einer Dienstgüteklasse auf dazugehörigen Werte. Dieses Pattern findet vor allem in Dienstgütearchitekturen Anwendung.

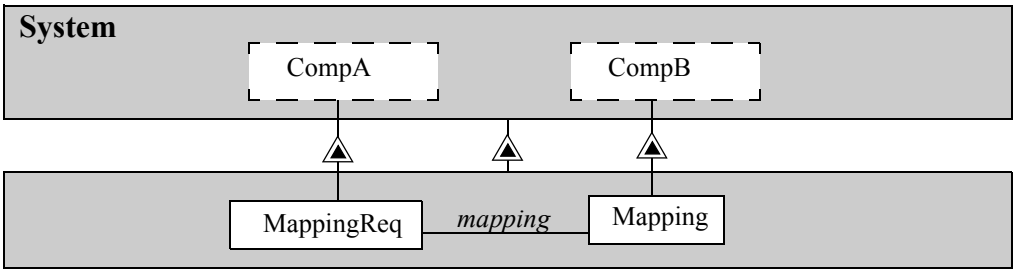
Motivation:

Unter *mapping* versteht man die (automatische) Umsetzung von Dienstgütespezifikationen zwischen benachbarten Schichten des Systems. Spezifikationen auf einem hohen Level sind meistens in der Sprache des Benutzers verfasst und dadurch auf niedrigere Schichten nicht anwendbar. Je weiter man sich der Basistechnologie nähert, desto konkreter werden die Angaben. Bei der Videoübertragung wird meist eine *flüssige* Wiedergabe gefordert. Tiefere Schichten können jedoch mit dem Begriff *flüssig* nichts anfangen. Also findet eine Übersetzung von *flüssig* in *Bilder pro Sekunde* statt. Diese Einheit kann weiter in *kBit pro Sekunde* und in *Rahmen pro Sekunde* verfeinert und somit für die Basistechnologie „verständlich“ gemacht werden.

Dieses Pattern hilft, die obige Funktionalität als Bestandteil einer Middleware umzusetzen.

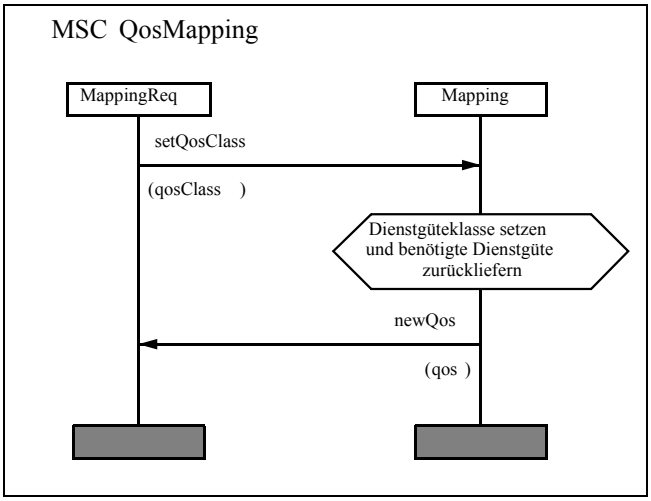
Structure:

Das ursprüngliche System besteht aus zwei Komponenten A und B, die nicht notwendigerweise miteinander kommunizieren müssen. Die durch das Pattern gegebene Lösung sieht vor, die Komponenten dahingehend zu verfeinern, dass eine Mapping-Funktionalität gegeben ist. Eine Komponente übernimmt dabei die Rolle der Dienstgüteanwendung (*MappingReq*), während die andere das eigentliche Mapping übernimmt (*Mapping*).



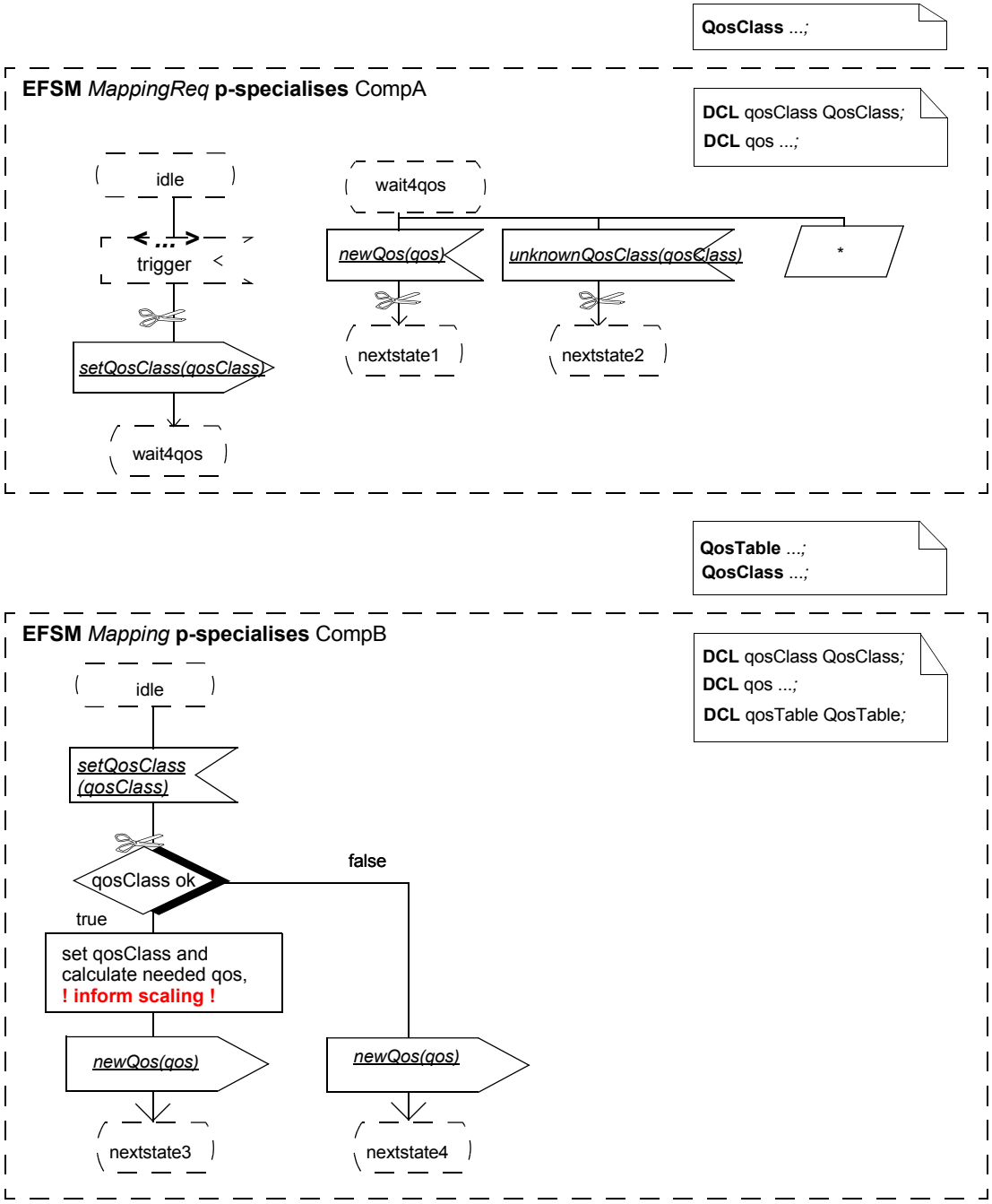
Message Scenario:

Das folgende MSC zeigt ein typisches Szenario nach Anwendung des Patterns. Auf Anfrage von *MappingReq* setzt *Mapping* lokal die geforderte Dienstgüteklasse und liefert die benötigte Dienstgüte zurück. Vorher muss sichergestellt werden, dass die geforderte Dienstgüteklasse spezifiziert wurde.



SDL Fragment:

CompA wird dahingehend verfeinert, dass diese nach einem Stimulus *trigger*, eine neue Dienstgüteklasse setzt. Danach muss sie auf eine entsprechende Antwort warten. Der Typ *QosClass* und die Variable(n) *qos* sind geeignet zu spezifizieren (z.B. *qos* als Bandbreite-Delay-Paar). Die Verfeinerung von *CompB* sieht vor, dass diese Komponente nach dem Empfang einer Dienstgüteklasseanforderung eine Überprüfung durchführt, um festzustellen, ob die neue Dienstgüteklasse *qosClass* existiert. Dazu ist es wichtig, geeignete Container zu spezifizieren (*QosTable*) und mit den möglichen Dienstgüteklassen und deren Parametern zu initialisieren. Falls die Überprüfung erfolgreich ist, kann die benötigte Dienstgüte z.B. als Bandbreite/Delay Tupel berechnet und an den Sender der Anforderung gesendet werden. Falls im System eine Scaling-Komponente vorhanden ist, so muss diese über die aktuelle Dienstgüteklasse in Form von einem Parameterintervall, wie in Tabelle 4-1 gezeigt, informiert werden. Die genaue Art der Benachrichtigung ist von der Spezifikation der betreffenden Scaling-Komponente abhängig (gemeinsame Variablen, RPC oder ein Signal).



Syntactical Embedding Rules:

Im folgenden wird beschrieben, wie das Pattern instanziert werden muss

- Varianten:
  - Auflösung des *trigger*-Symbols: input, conditioned input, continuous signal oder ein input gefolgt von einem condition-symbol.
  - Falls keine neue Komponente *Mapping* eingeführt wird, müssen alle Symbole und Zustände dem Kontext entsprechend angepasst werden.

- Renaming:
  - Falls ein vohandener Prozess verfeinert wird, müssen die Zustände (Zustandsmengen) *idle, ait4qos, nextState1, nextState2,nextstate3, nextstate4* und das Signal/die Variable *trigger* an den Kontext angepasst werden.
  - Falls keine Verfeinerung angewandt wird, so müssen die Zustände neu gesetzt werden.

Example Application:

- *PilotMapping* und *VideoMapping* Diplomarbeit C.Webel

Semantic Properties:

Unter der **Annahme** dass ...

(A-1) Die Dienstgüteklassen alle spezifiziert sind

... gilt die folgende **Aussage**:

(C-1) Jede Anfrage einer Dienstgüteklasse liefert die benötigte Dienstgüte zurück

Refinement:

Keine

Cooperative Usage:

Keine

Known Uses:

C. Webel: „Entwicklung und Integration von QoS-Mikroprotokollen zur Steuerung eines Fluggerätes über WLAN“, Diplomarbeit, Technische Universität Kaiserslautern , 2004

Checklist:

- Dienstgüteklassen gesetzt
- Berechnung der Bandbreite korrekt
- Skalierungskomponente vorhanden

Anhang B

ASN.1 Datentypen und -strukturen

```
BasicTypesASN1
   DEFINITIONS
     AUTOMATIC TAGS ::=

BEGIN
EXPORTS REAL_TYPE,INTEGER_TYPE,STRING_TYPE,NOPARAM_TYPE, IMAGE_STRUCT_ASN,
     QOSCLASS_ASN, CTRLVAL_ASN, FPS_ASN, JPEGQUALITY_ASN, DELAY_ASN,
     VOLTAGE_ASN, FEEDBACK_ASN;

REAL_TYPE     ::= REAL
INTEGER_TYPE  ::= INTEGER
STRING_TYPE   ::= IA5String
NOPARAM_TYPE  ::= NULL

IMAGE_STRUCT_ASN ::= SET
{
   data  BIT STRING,
   length INTEGER_TYPE
}

CTRLVAL_ASN  ::= INTEGER_TYPE
FPS_ASN ::= INTEGER_TYPE
JPEGQUALITY_ASN ::= INTEGER_TYPE
VOLTAGE_ASN ::= INTEGER_TYPE

-- QOS
QOSCLASS_ASN ::= STRING_TYPE
VOLUME_ASN ::= INTEGER_TYPE -- kBit
DELAY_ASN ::= INTEGER_TYPE -- ms
PRIORITY_ASN ::= INTEGER_TYPE -- 1 : highest, oo : lowest
JITTER_ASN ::= INTEGER_TYPE -- in ms
FEEDBACK_ASN ::= ENUMERATED { red(0), yellow(1), green(2) }

-- IP and Mac-Address
INETADDRESS_ASN ::= OCTET STRING (SIZE (4|6)) -- IPV4 or IPV6
MACADDRESS_ASN ::= OCTET STRING ( SIZE (6) )  -- without :

-- Data Packet
PACKET_ASN ::= SET
{
  host INETADDRESS_ASN, --target
  port INTEGER_TYPE,
  length INTEGER_TYPE,
  data OCTET STRING
}

-- LINK
LINK_ASN ::= SET
{
  host INETADDRESS_ASN,
  bandwidth INTEGER_TYPE, -- kBit/s
  delay INTEGER_TYPE -- localhost to host in ms
}

-- MAC + Application ID, limited to 256 Apps per host
ID ::= OCTET STRING (SIZE (7) )

END
```

```
VideoASN1
  DEFINITIONS
    AUTOMATIC TAGS::=

BEGIN
IMPORTS INTEGER_TYPE,NOPARAM_TYPE,STRING_TYPE, IMAGE_STRUCT_ASN,
     QOSCLASS_ASN, FPS_ASN, JPEGQUALITY_ASN, FEEDBACK_ASN FROM
     BasicTypesASN1;

Video_Message_Type ::= CHOICE {
  image                    IMAGE_TYPE,
  identifyVideoQosClass    IDENTIFYVIDEOQOSCLASS_TYPE,
  enableCam                ENABLECAM_TYPE,
  disableCam               NOPARAM_TYPE,
  qosClassNotSet           QOSCLASSNOTSET_TYPE,
  qosClassSet              QOSCLASSSET_TYPE,
  defineNewVideoQosClass   DEFINENEWVIDEOQOSCLASS_TYPE,
  curFeedback              CURFEEDBACK_TYPE
}

IMAGE_TYPE ::= SET{
  param1 IMAGE_STRUCT_ASN,
  param2 FPS_ASN,
  param3 JPEGQUALITY_ASN
}

IDENTIFYVIDEOQOSCLASS_TYPE ::= SET{
  param1 QOSCLASS_ASN
}
ENABLECAM_TYPE ::= SET{
  param1 QOSCLASS_ASN
}
DEFINENEWVIDEOQOSCLASS_TYPE ::= SET{
  param1 QOSCLASS_ASN,
  param2 FPS_ASN,
  param3 FPS_ASN,
  param4 JPEGQUALITY_ASN,
  param5 JPEGQUALITY_ASN

}

QOSCLASSNOTSET_TYPE ::= SET{
  param1 QOSCLASS_ASN
}

QOSCLASSSET_TYPE ::= SET{
  param1 QOSCLASS_ASN
}

CURFEEDBACK_TYPE ::= SET {
  param1  FEEDBACK_ASN
}

END
```

```
PilotASN1
  DEFINITIONS
    AUTOMATIC TAGS ::=

BEGIN
IMPORTS CTRLVAL_ASN, QOSCLASS_ASN, DELAY_ASN,VOLTAGE_ASN, NOPARAM_TYPE FROM
     BasicTypesASN1;

Pilot_Message_Type ::= CHOICE
{
  startZeppelin NOPARAM_TYPE,
  stopZeppelin NOPARAM_TYPE,
  newCtrlValues  NEWCTRLVALUES_TYPE,
  curCtrlValues  CURCTRLVALUES_TYPE,
  qosClassNotSet QOSCLASSNOTSET_TYPE,
  qosClassSet    QOSCLASSSET_TYPE,
  identifyPilotQosClass   IDENTIFYPILOTQOSCLASS_TYPE,
  defineNewPilotQosClass   DEFINENEWPILOTQOSCLASS_TYPE

}

QOSCLASSNOTSET_TYPE ::= SET{
  param1 QOSCLASS_ASN
}

QOSCLASSSET_TYPE ::= SET{
  param1 QOSCLASS_ASN
}

IDENTIFYPILOTQOSCLASS_TYPE ::= SET{
  param1 QOSCLASS_ASN
}

DEFINENEWPILOTQOSCLASS_TYPE ::= SET{
  param1 QOSCLASS_ASN,
  param2 DELAY_ASN,
  param3 DELAY_ASN
}


NEWCTRLVALUES_TYPE ::= SET
{
   param1 CTRLVAL_ASN,
   param2 CTRLVAL_ASN,
   param3 CTRLVAL_ASN
}

CURCTRLVALUES_TYPE ::= SET
{
   param1 CTRLVAL_ASN,
   param2 CTRLVAL_ASN,
   param3 CTRLVAL_ASN,
   param4 VOLTAGE_ASN
}

END
```

Anhang C

HINWEISE ZUR BENUTZUNG VON ASN.1 UND TELELOGIC TAU

Kompilieren und Linken eines SDL-Systems mit ASN.1 Datenstrukturen

Schritt 1: Generate -> Make

Häckchen an „*Generate environment header files*“

Häckchen an *„Generate ASN.1 coder“*

Häckchen an *„Generate makefile“*

Häckchen an *„Makefile“*

Häckchen an *„Compile und Link“*

übrige Optionen wählen wie benötigt

FullMake drücken -> Beendet mit Fehler (ok)

Schritt 2: Generate -> Targeting Expert

Einstellungen wählen wie benötigt

Registerkarte *Communication* anwählen

Häckchen *„generate ASN.1 coder functions“*

Full Make drücken -> Bei Fehlern ggf. Einstellungen anpassen

Targeting Expert schliessen

Schritt 3: Generate -> Make

Einstellungen sollten so sein wie vorher

Häckchen von *„Generate makefile“* auf *„Generate makefile and use template“* ändern

vom Targeting Expert generiertes Template auswählen: im Zielverzeichnis der Codeerstellung unter *Systemname*_**env.tpm**

FullMake drücken -> sollte ohne Fehler durchlaufen, sonst ggf Einstellungen geeignet ändern

Schritt 4:

Falls nur noch SDL-Teile des Systems geändert werden, entfallen nach dem ersten Kompilieren und Linken die Schritte 1 und 2.

Anhang D

Die Mikroprotokollbibliothek und der Systementwurf (CD)