5 Realisierung einer selbstorganisierenden Software-Infrastruktur

Das dezentrale Middleware-Modell SODAPOP für selbstorganisierende Systems unterscheidet zwischen Transducern, also den Komponenten eines Ensembles und den Kanälen, die der Abbildung der Nachrichten zwischen den Komponenten dienen. Zur Konfliktlösung im Falle konkurrierender Komponenten und zur Dekomposition von Nachrichten wenden Kanäle Konfliktlösungsstrategien an. Dieses zweistufige Modell der Selbstorganisation mit der Möglichkeit der Aufteilung des Datenflusses durch Kanäle und die Anwendung von Strategien innerhalb der Kanäle zur Zuteilung von Nachrichten im Falle funktionsgleicher (und damit möglicherweise konkurrierender) Komponenten ermöglicht die Dynamisierung des Komponentenensembles. Hinzukommende Komponenten können sich in ein bereits bestehendes Komponentenensemble hineinfügen, am Nachrichtenverkehr innerhalb der unterschiedlichen Kanäle partizipieren, Nachrichten generieren und auf die unterschiedlichen Kanäle geben oder sich unter Bereitstellung ihrer UtilityValue-Funktionen an den Ausschreibungen zur Konsumierung von Events und/oder der Bearbeitung von Aufträgen in Form von RPCs (Remote Procedure Calls) beteiligen. Die Spezifikation soll in zwei aufeinanderfolgenden Teilschritten erfolgen. Zuerst wird die Selbstorganisation von Software-Komponenten auf Basis des SODAPOP-Modells spezifiziert (Kapitel 5.1). Hierzu werden die Prinzipien des SODAPOP-Modells (siehe Kapitel 4) zur Realisierung im Detail definiert und deren Implementation und Kommunikationsstrukturen spezifiziert. Zum Abschluss wird die Anwendung für eigene Implementationen beispielhaft demonstriert. Diese Spezifikation wird dann im Anschluss auf die Selbstorganisation von physikalisch verteilten Geräten und deren Software-Komponenten erweitert (Kapitel 5.2).

5.1 Selbstorganisation von Software-Komponenten

In einer Detaillierung des SODAPOP-Ansatzes, der in Kapitel 4 beschrieben wurde, werden zur Realisierung die folgenden Definitionen verwendet:

Transducer bzw. Komponenten[39] sind die kleinste logische Einheit, die in einem selbstorganisierten System unterstützt werden müssen. Transducer sind im Allgemeinen autonom, in der Lage komplexe Nachrichten zu verarbeiten und von sich aus Nachrichten und Aufträge für andere Komponenten zu generieren. Transducer entsprechen damit weitestgehend den von Ferber [71, 72] definierten Eigenschaften eines Software-Agenten. Transducer sind mit Kanälen verbunden. Die Anzahl der verbundenen Kanäle ist hierbei nicht begrenzt.

Kanalstrategien: Transducer senden ihre Nachrichten ausschließlich an einen Kanal, an dem sie konnektiert sind. Dieser Kanal ermittelt daraufhin die Verfügbarkeit an-

[39] In dieser Arbeit werden die Begriffe Transducer und Komponente nahezu synonymisch verwendet. Sollten es an einigen Stellen des Textes logische Unterschiede geben, werden diese an Ort und Stelle erläutert.

derer konnektierter Transducer und evaluiert deren UtilityValue-Funktion auf Basis der initialen Nachricht. Nach Einholung aller verfügbaren UtilityValues ermittelt die kanalabhängige Strategie den oder die Empfänger der Nachricht. Im Falle konkurrierender Komponenten ist die Kanalstrategie damit in der Lage einer Nachricht eindeutig einen Empfänger zuzuweisen. Kanalstrategien wirken somit als Konfliktlösungsstrategien.

Dekomposition von Nachrichten: Wird eine Nachricht in mehrere Teilnachrichten zerlegt, so spricht man von einer Dekomposition von Nachrichten. Werden diese Teilnachrichten an unterschiedliche Empfänger übermittelt, so findet eine Kooperation der Empfänger statt. Ein Sonderfall ist die Übermittlung der initialen Nachricht an mehrere Empfänger. Hier findet Kooperation ohne vorherige Dekomposition der Nachricht statt.

Events sind ungerichtete Nachrichten, die ein Transducer in einen Kanal schreibt. Der Kanal ermittelt einen geeigneten Empfänger für das Ereignis. Es geschieht nichts, falls sich kein geeigneter Empfänger für ein Ereignis findet. Die Definition der Events hält sich hier streng an die Definition der Ereignisse von bekannten objektorientierten Programmiersprachen.

Remote Procedure Calls werden von Transducern an einen Kanal abgesetzt, falls sie eine Antwort wünschen. Der Kanal ermittelt den oder die geeigneten Bearbeiter eines RPCs, und übermittelt die Antworten an den Anfrager zurück. Für die Absendung eines Remote Procedure Calls erhält der Auftraggeber exakt *eine* Antwort zurück. Führen mehrere Transducer einen Auftrag in Kooperation aus (eventuell nach erfolgter Dekomposition) ist es die Aufgabe der Kanalstrategie die Antworten zu einer semantischen Antwort zusammenzuführen und dem Auftraggeber zu übermitteln. Findet sich kein geeigneter Transducer, der die Aufgabe ausführen kann, so ist vom Kanal eine entsprechende Benachrichtigung an den Auftraggeber zu übermitteln.

Konnektion an Kanälen: Damit Transducer in der Lage sind, Events oder RPCs auf einen Kanal zu schreiben bzw. Events zu konsumieren oder RPCs zu bearbeiten, müssen sie sich entsprechend an einem Kanal anmelden. Entsprechend Abbildung 41 können sich Transducer an einem Kanal in der Art anmelden, dass es ihnen erlaubt ist, Events auf den Kanal zu schreiben (OUT). Sie agieren somit als Event-Quelle. Im anderen Fall (IN) agieren sie als Listener-Transducer, der dem Kanal eine UtilityValue-Funktion zur Verfügung stellt, um die Möglichkeit zu haben Events zu konsumieren. Diese sind somit eine Event-Senke. Da das Versenden eines Remote Procedure Calls als eine aktive Anfrage nach einem Event angesehen werden kann (ein Event-Pulling statt dem Warten auf ein Event-Push), sind Transducer die als Event-Quellen agieren zugleich potentielle Bearbeiter von RPCs. Konsequenterweise sind Event-Senken somit in der Lage initiale Remote Procudure Calls auf den Kanal zu geben. Entsprechend der SODAPOP-Definition geht somit mit einem Kanal der Events bearbeitet immer auch ein inverser RPC-Kanal einher.

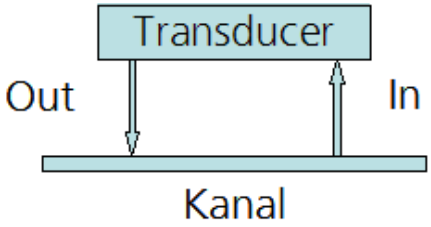


Abbildung 41: Die Verbindungsmöglichkeiten eines Transducers an einem Kanal. Ein Transducer ist in der Lage Events auf den Kanal zu schreiben (OUT) oder Events zu konsumieren (IN). Das Verhalten gegenüber RPCs verläuft dann exakt umgekehrt.

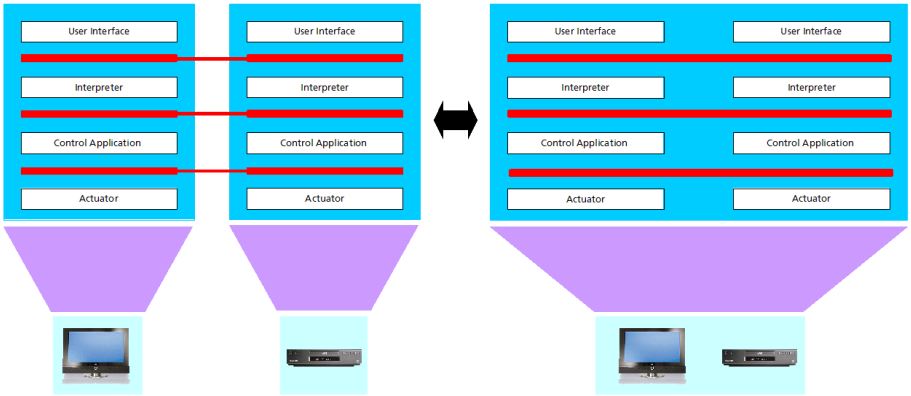


Abbildung 42: Illustration der Verbindung von zwei Einzelgeräten (links) zu einem physikalischen Gerät (rechts).

Die Erläuterung der weiteren Definitionen zur Realisierung einer selbstorganisierenden Software-Infrastruktur auf Basis des SODAPOP-Modells sollen mit Hilfe der Illustration in Abbildung 42 verdeutlicht werden. Betrachtet man zwei Einzelgeräte getrennt voneinander, können die unterschiedlichen Komponenten der Einzelgeräte direkt miteinander kommunizieren. Da davon ausgegangen werden kann, dass die einzelnen Komponenten eines physikalischen Gerätes so aufeinander abgestimmt sind, dass es zu keinen Konflikten zwischen den Komponenten oder anderen Inkompatibilitäten kommen kann, kann hier unter Umständen sogar auf ein spezielles Kommunikationsprotokoll oder gar eine unterlagerte Software-Infrastruktur verzichtet werden[40]. Die Illustration rechts in Abbildung 42 verdeutlicht was passiert, wenn ein Gerätehersteller – in diesem Fall – ein kombiniertes TV / DVD-Gerät produzieren möchte. Hier stellt sich nun die Frage, wie die interne Kommunikationsstruktur dieses Geräteensembles, das physikalisch ein einziges Gerät ist, auf Basis seiner Komponenten und Kanäle bewerkstelligt wird. Bisher hat es keine Definition über die Realisierung von Kanälen gegeben. Kanäle könnten prinzipiell als eigenständige

[40] Auch in diesem Beispiel werden die im Vorkapitel verwendeten Komponentenarten verwendet. Dies soll jedoch nur der Konsistenz der Illustrationen dienen und selbstverständlich an dieser Stelle der Arbeit noch keine Präjudizierung der Anzahl und der Art der Komponenten auf einem einzelnen physikalischen Gerät geben.

Applikationen auf einem physikalischen Gerät laufen. Das würde bedeuten, dass je eine Applikation pro definierten Kanal die Nachrichten der an ihr verbundenen Transducer bearbeitet. Eine weitere Möglichkeit – und damit ein noch höherer Grad an Dezentralisierung – wäre es, die Menge der Transducer als eine Art Peer-to-Peer-System zu definieren, in dem jeder Transducer in der Lage wäre die einzelnen Kanalnachrichten semantisch zu trennen und zu bearbeiten. Hier müsste aber auch jeder Transducer in der Lage sein, die entsprechenden Kommunikationsalgorithmen in Form der Konfliktlösungsstrategien ausführen zu können.

Für den weiteren Fortgang der Spezifikation werden hier einige Annahmen definiert, die die Realisierung erfüllen muss:

1. Es soll die *Selbstorganisation von Software-Komponenten* unterstützt werden: Dies bedeutet, dass Transducer (die hier definierte kleinste Einheit von Software-Komponenten) einem bereits bestehenden Ensemble an Komponenten betreten können und am Nachrichtenverkehr teilhaben können. Software-Komponenten besitzen gemeinhin ein physikalisches Gerät als Host.

2. Die Konzeption und Implementierung von Kanaltopologien (d.h. die Anordnung von Transducern und Kanälen, die Definition der Kanalnamen und der dabei verwendeten Strategien) soll für einen *Software-Ingenieur* möglichst einfach und transparent sein. Ein Software-Ingenieur ist gemeinhin interessiert an der Realisierung komplexer Applikationen bzw. an der Implementierung und Lauffähigkeit seiner Komponenten, weniger an der Realisierung von Nachrichtenverkehr bzw. komplexen Strategien zur Gewährleistung der Selbstorganisation von Komponenten.

3. Der *Endverbraucher* ist an der Lauffähigkeit seines Gerätes interessiert. Er weiß nichts von einzelnen Komponenten, die miteinander in Interaktion stehen. Mit anderen Worten: er bedient das Gerät als eine Einheit, insbesondere werden die darauf laufenden Komponenten im Fehlerfalle einer Komponente im Gesamten ab- und eingeschalten.

4. Eine Software-Infrastruktur für selbstorganisierende Komponenten muss entsprechend schonend mit den vorhandenen *Rechenkapazitäten* und dem vorhandenen Speicherplatz umgehen. Doppelte Datenhaltung bzw. überflüssiger Nachrichtenverkehr zwischen Komponenten auf einem Gerät sind somit möglichst zu vermeiden.

Abbildung 43 illustriert den Realisierungsansatz, der die oben genannten Annahmen erfüllt. Da ein physikalisches Gerät selbst eine Entität darstellt agiert sie als zentraler Host für alle auf ihr laufenden Komponenten. Die Lösung, die in dieser Arbeit vorgeschlagen wird, basiert auf der Realisierung eines singulären Daemons pro physikalischem Gerät, der den auf einem Gerät laufenden Komponenten alle nötigen SODAPOP-Eigenschaften zur Selbstorganisation zur Verfügung stellt. Die am Anfang dieses Abschnittes erfolgten Definitionen können somit vervollständigt werden:

SODAPOPD(AEMON)**:** Der SODAPOPD(AEMON) ist für seinen Host – das physikalische Gerät – eine singuläre Applikation. Alle Transducer auf dem Gerät sind mit

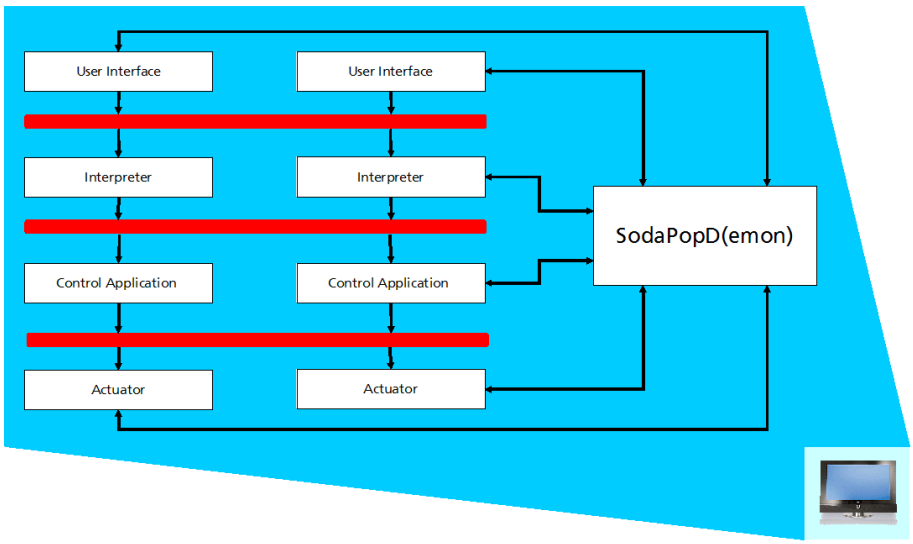


Abbildung 43: Ein physikalisches Gerät besitzt einen SODAPOPD(AEMON). An diesem sind die Transducer direkt konnektiert. Kanäle sind keine physikalischen Einheiten sondern werden als virtuelle logische Einheiten innerhalb der Transducer und des SODAPOPD abgebildet. Zu diesen virtuellen Einheiten sind die Transducer verbunden.

dem auf dem Gerät laufenden SODAPOPD verbunden. Die Komponenten können jederzeit angeben, mit welchen Kanälen sie auf welche Art und Weise verbunden sein möchten. Senden Komponenten eine Nachricht auf einen Kanal, so senden sie diese Nachricht an den SODAPOPD.

Letztlich können die Kanäle definiert werden:

Kanäle sind *virtuelle* Entitäten und werden vom SODAPOPD verwaltet. Der Bezeichner eines Kanals definiert somit die Art der Nachrichten, die auf ihm kommuniziert werden. Jede Kanalentität erlaubt die Definition einer geeigneten Konfliktlösungsstrategie. Diese Konfliktlösungsstrategie wird zentral durch den SODAPOPD ausgeführt. Ein SODAPOPD ist in der Lage eine unlimitierte Anzahl an Kanälen zu verwalten.

5.1.1 Spezifikation des SODAPOPD(AEMON)s

Der in Kapitel 5.1 eingeführte SODAPOPD(AEMON) hat zur Gewährleistung der Selbstorganisation der an ihm konnektierten Transducer die folgenden Aufgaben zu erfüllen:

- Konnektionsmöglichkeit und Verwaltung der mit ihm verbundenen Transducer
- Instantiierung von virtuellen Kanälen und Verwaltung der mit diesen Kanälen verbundenen Transducer
- Kollektion der UtilityValue-Funktionen der mit einem Kanal verbundenen Transducer, Ausführung der entsprechenden Konfliktlösungsstrategie und Zuteilung des

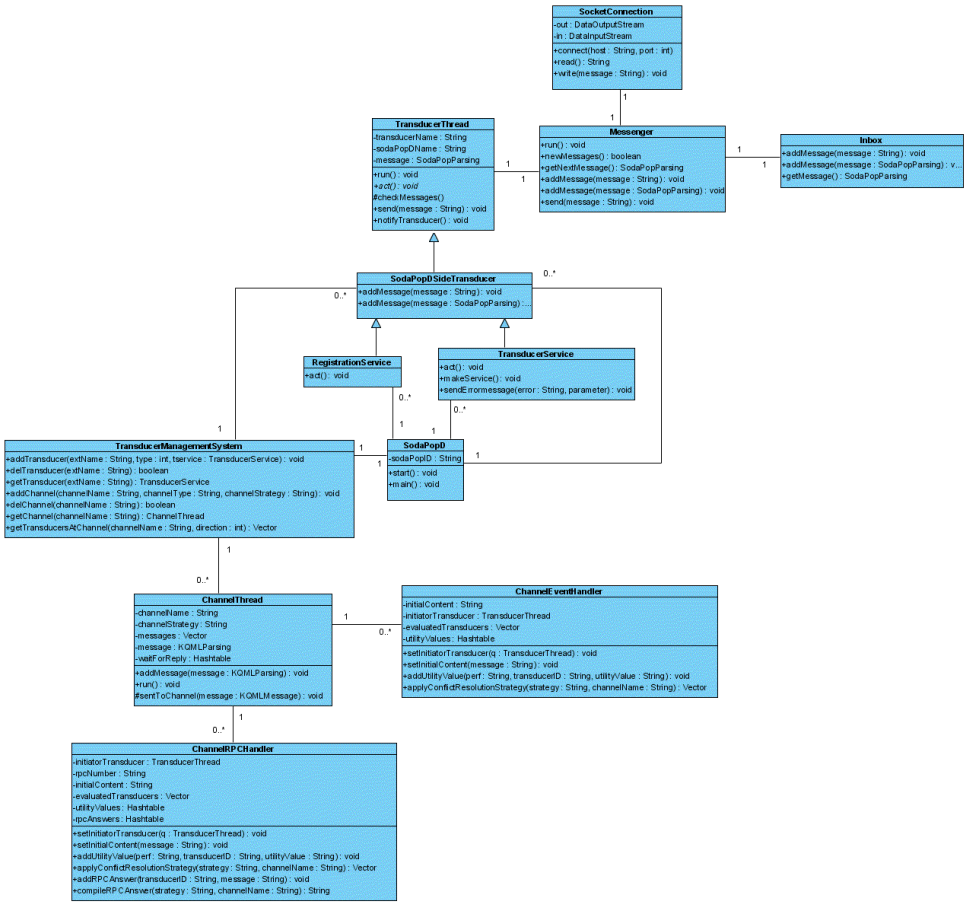


Abbildung 44: Klassenstruktur der Realisierung des SODAPOPD(AEMON)s zur Realisierung der Selbstorganisation von Komponenten.

initialen Events bzw. Remote Procedure Calls in Entsprechung des Strategieergebnisses.

Abbildung 44 veranschaulicht die Klassenstruktur zur Realisierung des SODAPOPD(AEMON). Die hierbei wichtigsten Klassen und die von ihnen zur Verfügung gestellten Funktionen sind:

SODAPOPD**:** Die Hauptklasse des SODAPOPD(AEMON)s. Nach Starten des SODAPOPD wartet dieses Objekt auf die Konnektion von Komponenten. Sobald sich ein Transducer verbunden hat, wird ein Objekt der Klasse *RegistrationService* gestartet.

TransducerManagementSystem: Der SODAPOPD(AEMON) verfügt über genau eine Instanz dieser Klasse. In diesem Objekt werden die Bezeichner aller Transducer, sowie die Bezeichner aller Kanäle und die Verbindungen der einzelnen Transducer zu den Kanälen verwaltet. Dieses Objekt verwaltet zudem alle Objekte der Klassen TransducerService und ChannelThread.

RegistrationService: Dieses von der Instanz des SODAPOPD gestartete Objekt nimmt die erste Nachricht eines konnektierten Transducers in Empfang, bearbeitet sie und sendet dem neuen Transducer eine Nachricht zurück. Zeitgleich wird eine Instanz der Klasse *TransducerService* gestartet.

TransducerService: Ein Objekt dieser Klasse ist während der gesamten Lebenszeit eines Transducers (bzw. der Lebenszeit des SODAPOPD(AEMON)s) der Counterpart eines Transducers. In diesem Objekt gehen zur Weiterverarbeitung alle Nachrichten ein, die von einem bestimmten Transducer gesendet werden. Ein SODAPOPD(AEMON) besitzt somit genauso viele Instanzen der Klasse TransducerService wie Transducer an ihm konnektiert sind.

ChannelThread: Für jeden Kanal, der von den verschiedenen Transducern in ihrer Gesamtheit definiert wird, wird ein Objekt der Klasse ChannelThread instanziiert. Dieses Objekt behandelt alle Nachrichten, die auf diesem Kanal von den angemeldeten Transducern kommuniziert werden. Ein SODAPOPD(AEMON) besitzt exakt so viele Instanzen der Klasse ChannelThread wie Kanäle definiert sind.

ChannelEventHandler: Für jedes einzelne Event, das den Kanal erreicht, wird ein Objekt dieser Klasse erzeugt. In diesem Objekt werden sowohl die initiale Nachricht, als auch die von den an dem Kanal konnektierten Transducern eingesammelten *UtilityValues* verwaltet. Nach Eingang aller Antworten führt dieses Objekt die kanalspezifische Konfliktlösungsstrategie aus und stellt dem Objekt *ChannelThread* die Ergebnisse der Konfliktlösungsstrategie zur Verfügung. Die Ergebnisse der Ausführung einer Konfliktlösungsstrategie sind Informationen über die Bezeichner der Transducer, die die Nachricht empfangen sollen, und die Nachricht (eventuell die veränderte Nachricht) selber.

ChannelRPCHandler: Objekte dieser Klasse sind zuständig für die Verwaltung und Zustellung von Remote Procedure Calls. Sie agieren analog zu *ChannelEventHandlern*. Aufgrund der erhöhten Komplexität von Remote Procedure Calls im Vergleich zu Events verfügt diese Klasse über zusätzliche Methoden, um aufgrund der von den Empfängern eines RPCs versendeten Antworten eine kompilierte Antwort für den Initiator eines RPCs zu genieren und zu verschicken.

Messenger: Ein Objekt dieser Klasse bietet (zusammen mit den Klassen *SocketConnection* und *Inbox*) alle Methoden an, um Nachrichten an den dazugehörigen Transducer zu verschicken und Nachrichten von ihm zu empfangen. Um gleichzeitiges Verschicken und Erhalten von Nachrichten zu gewährleisten, ist der Messenger als Thread implementiert.

5.1.2 Spezifikation der Transducer

Transducer stellen im SODAPOP-Modell die eigentlichen Komponenten dar. Diese sind in der Lage autonom Entscheidungen zu treffen, Nachrichten zu verarbeiten und neue Nachrichten zu definieren. Zum Versenden und Erhalten von Nachrichten sind Transducer mit Kanälen verbunden. Um Nachrichten zu erhalten, stellen sie dem Kanal ihre *UtilityValue*-Funktionen wie in Kapitel 4 beschrieben zur Verfügung. Ein Transducer muss somit die folgenden Eigenschaften erfüllen bzw. die folgenden Möglichkeiten haben:

- Definition von Kanälen und die Definition der eigenen Rolle bezüglich dieses Kanals (Event-Quelle und / oder Event-Senke resp. RPC-Senke und / oder RPC-Quelle)
- Definition der UtilityValue-Funktionen an den unterschiedlichen Kanälen
- Kommunikation mit dem SODAPOPD(AEMON)
- Implementation von autonomen Handlungen
- Bearbeitung von Events bzw. Antworten auf Remote Procedure Calls

Abbildung 45 illustriert die Klassenstruktur eines Transducers. Hierbei ist bedeutsam, dass die Implementation eines Transducer vom Programmierer eines SODAPOP-Systems vollkommen frei gestaltet und definiert werden kann. Ein Transducer erbt von keiner anderen Klasse. In völliger Übereinstimmung zum SODAPOP-Modell *besitzt* ein Transducer die Kanäle an denen er konnektiert ist. Dies wird in Abbildung 45 durch die Multiplizitäten 1 und $1*$ in der Verbindung zwischen den Klassen Transducer und Channel zum Ausdruck gebracht. Ein Transducer kann folglich mit einer Vielzahl an Kanälen verbunden sein. Im Folgenden werden die Klassen im Einzelnen detaillierter besprochen:

Transducer: Der eigentliche Transducer ist nicht gezwungen von einer Oberklasse zu erben. Die Definition des Funktionsumfangs eines Transducers ist somit vollkommen frei. Besonders in der Programmiersprache Java kann damit auch eine graphische Benutzeroberfläche ein Transducer sein[41].

Channel: Ein Kanal wird durch seinen Transducer instanziiert mit der Angabe eines Kanalnamens, der Angabe der auf dem Kanal geltenden Konfliktlösungsstrategie und der eigenen Rolle auf diesem Kanal (*IN* oder *OUT*). Zusätzlich muss diesem Kanal ein *SodaPopHandler*-Objekt mitgegeben werden.

[41] Selbstverständlich würde das Prinzip der Einfachvererbung in Java nicht im Prinzipiellen verhindern, dass ein von einer etwaigen SODAPOP-Klasse erbender Transducer dennoch eine graphische Oberfläche besitzen könnte. Der hier spezifizierte Ansatz der fehlenden Verpflichtung einen Transducer von einer Oberklasse ableiten zu müssen, macht es jedoch möglich, dass ein Transducer eine graphische Benutzeroberfläche *ist* anstatt sie nur zu *besitzen*. Dieser Ansatz fördert das Verständnis bzgl. des SODAPOP-Modells und stellt nochmals heraus, dass das Kommunikationsmodell SODAPOP die Selbstorganisation von Komponenten unterstützt und somit eine Software-Infrastruktur zur Kommunikation von autonomen Komponenten darstellt. Würden Komponenten von Klassen dieser Software-Infrastruktur abgeleitet werden, wären die Schwerpunkte verschoben und unter Umständen falsch gesetzt.

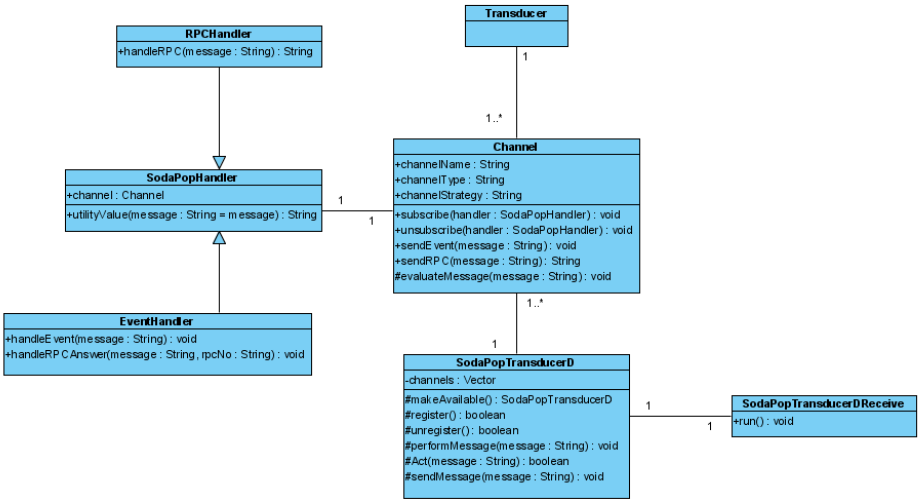


Abbildung 45: Klassenstruktur eines Transducers.

SodaPopHandler: Die abstrakte Klasse *SodaPopHandler* bildet die Oberklasse für die beiden Handler-Klassen *EventHandler* und *RPCHandler*. Einem Kanalobjekt, bei dem der Transducer die Rolle der Event-Quelle (definiert als *OUT*) einnimmt, ist ein *RPCHandler* zuzuweisen. Einem Kanalobjekt, bei dem der Transducer die Rolle der Event-Senke (definiert als *IN*) einnimmt, ist entsprechend ein *EventHandler* zuzuweisen. Beidesmal ist die UtilityValue-Funktion entsprechend den Definitionen der festgelegten Konfliktlösungsstrategie auf dem Kanal zu implementieren. Übernimmt der Transducer die Rolle einer Event-Quelle ist die entsprechende *handleRPC*-Methode, im anderen Fall die korrespondierenden *handleEvent*- und *handleRPCAnswer*-Methoden zu implementieren. Jedes Objekt der Klasse Channel besitzt genau einen SodaPopHandler.

SodaPopTransducerD: Diese Klasse stellt den *Channel*-Instanzen alle nötigen Methoden zur Verfügung, um mit einem SODAPOPD(AEMON) zu kommunizieren. Die Multiplizitäten $1*$ und 1 bei der Verbindung der Kanalobjekte und des Objektes der Klasse *SodaPopTransducerD* geben hierbei an, dass alle Kanäle, an denen der Transducer konnektiert ist, mittels eines einzigen Objektes kommunizieren. Die folgenden Abschnitte über das unterlagerte Protokoll und die Interaktionen zwischen dem SODAPOPD(AEMON) und den Transducern wird hierauf näher eingehen. Der Buchstabe *D* im Bezeichner weist auf die *Daemon*-Eigenschaft dieser Klasse hin.

SodaPopTransducerDReceive: Dieses Objekt ist ein Thread und startet zeitgleich mit dem SodaPopTransducerD. Ein Objekt der Klasse *SodaPopTransducerD* hält die dauerhafte Verbindung zum SODAPOPD(AEMON). Dadurch ist der Nachrichteneingang vom SODAPOPD(AEMON) ohne jede Zeitverzögerung möglich.

5.1.3 SODAPOPD(AEMON) – Transducer – Interaktion

Kanäle liegen in der Implementation des SODAPOP-Modells nicht als physikalische Einheiten vor, sondern sind als virtuelle logische Einheiten definiert. Die Bezeichner der Kanäle sowie die auf dem Kanal gültige Konfliktlösungsstrategie werden durch die Transducer bestimmt, die damit virtuell die einzelnen Kanalobjekte besitzen.

Der SODAPOPD(AEMON), an dem alle Transducer konnektiert sind, verwaltet alle von den unterschiedlichen Transducern definierten Kanäle und stellt die nötigen Algorithmen und Ressourcen für die Ausführung der Konfliktlösungsstrategien bereit. Jeder Transducer besitzt – unabhängig von der Anzahl der Kanäle mit denen er verbunden ist – *eine* Verbindung zum SODAPOPD(AEMON). Diese Konnektion[42] dient der Übermittlung sowohl *logischer Nachrichten* als auch der Übermittlung von *administrativen Nachrichten*. Zu der Klasse der administrativen Nachrichten gehören Informationen, die die Verbindung des Transducers zu seinem SODAPOPD selbst betreffen. Beispiele hierfür sind: Konnektion, Diskonnektion, und Information über Kanalverbindungen und Kanalaufbau. Zu der Klasse der logischen Nachrichten gehören *Events*, *Remote Procedure Calls*, sowie das Abfragen und Übermitteln von *UtilityValue-Funktionen*. Administrative Nachrichten sind generell an den SODAPOPD(AEMON) direkt gerichtet, logische Nachrichten an einen Kanal. Der Adressat einer Nachricht ist im unterlagerten Protokoll, in dem jede Nachricht kodiert ist, angegeben.

Konnektion eines Transducers und Nachrichtenverarbeitung: Abbildung 46 illustriert mittels eines Sequenzdiagrammes die internen Vorgänge im SODAPOPD(AEMON) bei der Konnektion eines Transducers. Wird der SODAPOPD gestartet wartet er dauerhaft auf die Konnektion neuer Transducer. Dies ist ein nebenläufiger Prozess (siehe Abbildung 46 Operation 1). Sobald sich ein Transducer konnektiert (Abbildung 46 Operation 2) startet der SODAPOPD ein neues Objekt der Klasse *RegistrationService*. Dieser nebenläufige Prozess startet ein *Messenger*-Objekt. Jeder konnektierte Transducer verfügt damit auf Seiten des SODAPOPD(AEMON) über ein eigenes *Messenger*-Objekt, in welches seine Nachrichten gesendet werden. Diese Nachrichten werden in einem *Inbox*-Objekt abgelegt (Abbildung 46 Operation 3 und Operation 4). Nachdem der Transducer-eigene *RegistrationService* gestartet ist, wartet er auf die erste eingehende Nachricht des Transducers (siehe Abbildung 46 Operation 5). Sobald diese im *Messenger* eingegangen ist, legt dieser die Nachricht in der *Inbox* ab und informiert das wartende *RegistrationService*-Objekt (Abbildung 46 Operation 6 und Operation 7). Die erste Nachricht, die ein Transducer zu seinem SODAPOPD(AEMON) sendet, ist die Registrationsnachricht (siehe nächsten Abschnitt). Der *RegistrationService* ermittelt auf diese Nachricht hin (Abbildung 46 Operation 8) eine eineindeutige Transducer-Identifikationsnummer (die dieser in allen nachfolgenden Nachrichten verwenden muss), fügt diesen Transducer der SODAPOPD-eigenen Transducerverwaltung hinzu, startet das Transducer-eigene Objekt *TransducerService* und übergibt diesem für alle zukünftigen Nachrichtentransfers das *Messenger*-Objekt (siehe Ab-

[42] In der hier beschriebenen Realisierung ist diese Verbindung über Sockets bzw. Serversockets, wie sie von der Programmierumgebung Java zur Verfügung gestellt werden, realisiert.

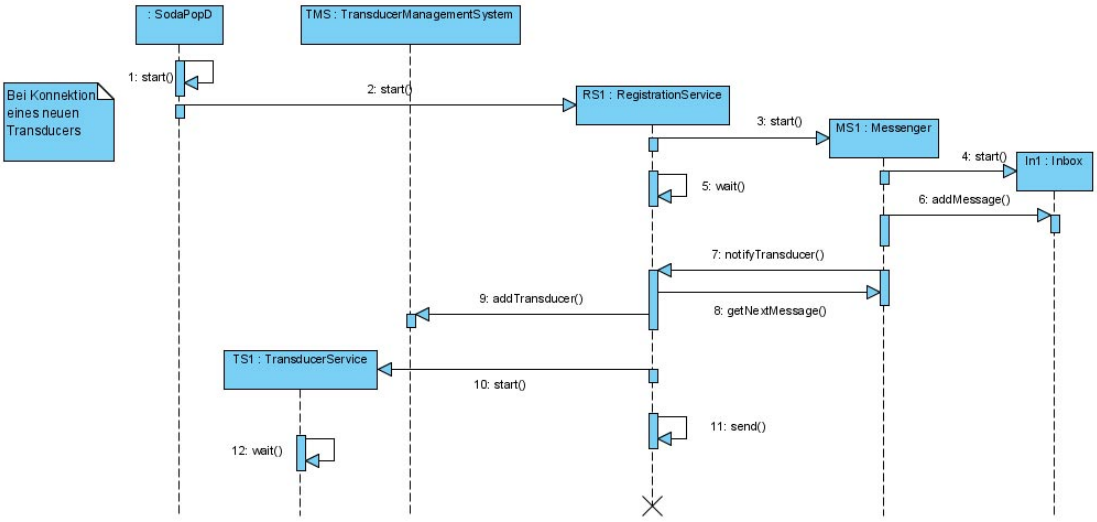


Abbildung 46: Konnektion eines Transducers am SODAPOPD(AEMON).

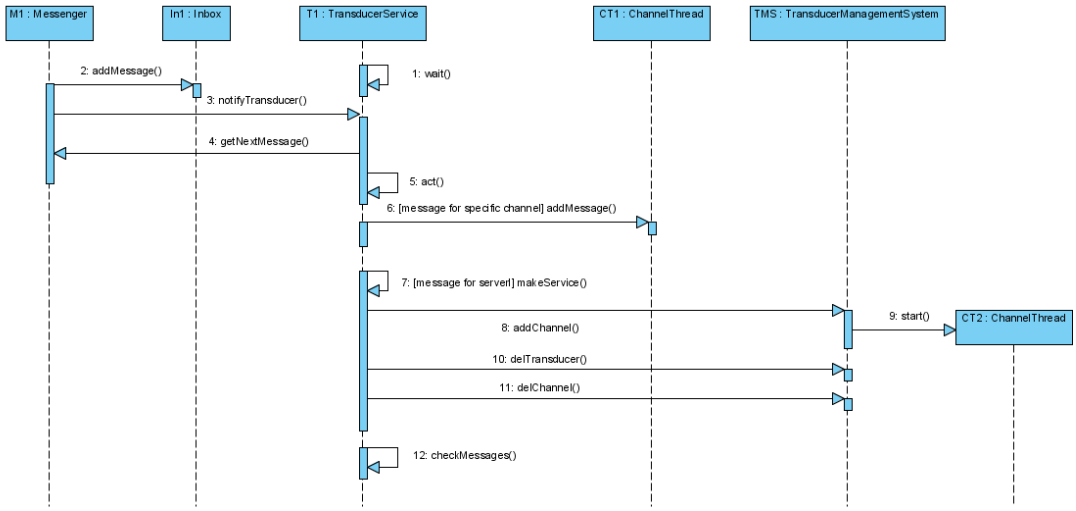


Abbildung 47: Nachrichtenübermittlung eines Transducers an den SODAPOPD(AEMON) zur Verwaltung des Transducers und der Kanäle, an denen dieser verbunden ist.

bildung 46 Operation 9 und 10). Der nebenläufige Prozess *TransducerService* bearbeitet nun alle Nachrichten, die von diesem bestimmten Transducer den SODAPOPD erreichen. Zum Schluss sendet der *RegistrationService* die ermittelte Identifikationsnummer an den gerade konnektierten Transducer und terminiert (Abbildung 46 Operation 11).

Der Nachrichtenverkehr eines Transducers mit seinem SODAPOPD wird im Sequenzdiagramm in Abbildung 47 verdeutlicht. Hierbei entspricht das Warten des *Transducer-*

Service-Objektes (siehe Abbildung 47 Operation 1) der Aktivität 12 desselben Objektes in Abbildung 46. Abbildung 47 bildet somit die zeitliche und logische Fortsetzung der Aktivitäten von Abbildung 46. Nach Eingang einer Nachricht von seinem Transducer wird der *TransducerService* vom seinem *Messenger*-Objekt aktiviert und beginnt die Nachricht zu verarbeiten (Abbildung 47 Operation 2 bis 5). Ist die Nachricht eine logische Nachricht und damit an einen bestimmten Kanal gerichtet, so wird sie dem *ChannelThread*-Objekt mit dem entsprechenden Bezeichner – der in der Nachricht angegeben ist – zugewiesen (Abbildung 47 Operation 6). Diese Nachrichten werden in den Sequenzdiagrammen Abbildung 48 und Abbildung 49 besprochen. Ist die Nachricht eine administrative Nachricht (Abbildung 47 Operation 7), so wird sie vom Objekt *TransducerService* direkt bearbeitet. Hierzu gehören:

- Die Konnektion zu einem Kanal (Abbildung 47 Operation 8) und die damit verbundene Administration der Rolle im *TransducerManagementSystem* des SODAPOPD(AEMON)s. Sollte ein Kanal mit dem angegebenen Bezeichner nicht existieren, wird dieser erzeugt und das damit verbundene Kanalobjekt *ChannelThread* erzeugt und gestartet (Abbildung 47 Operation 9).

- Das Löschen einer Verbindung des Transducers zu einem Kanal (Abbildung 47 Operation 11). Anmerkung: Falls zu dem Kanal mit dem dabei angebenen Bezeichner kein Transducer mehr konnektiert ist, wird das damit verbundene Objekt *ChannelThread* terminiert (in Abbildung 47 nicht dargestellt).

- Das Löschen des Transducers (beim vollständigen Diskonnektieren des Transducern aus dem Ensemble) und dem damit verbundenen Löschen aller Kanalverbindungen dieses bestimmten Transducers im *TransducerManagementSystem* (Abbildung 47 Operation 10). Anmerkung: In Abbildung 47 nicht in allen Details dargestellt.

Nach Bearbeiten der Nachricht wartet des Objekt *TransducerService* auf den Eingang neuer Nachrichten. Der Prozess beginnt von vorne (Abbildung 47 Operation 12 und Operation 1).

Verarbeitung von Events im SODAPOPD(AEMON): Abbildung 47 illustriert (Operation 6 bis Operation 11), dass sowohl logische Nachrichten als auch administrative Nachrichten im Transducer-repräsentierenden Objekt *TransducerService* eingehen und vorinterpretiert werden. Handelt es sich um logische Nachrichten so ist generell der Adressat einer solchen Nachricht der Bezeichner eines Kanals. Abbildung 48 illustriert die Abläufe im SODAPOPD(AEMON), falls es sich bei einer solchen Nachricht um ein Event handelt. Abbildung 48 veranschaulicht hierbei die SODAPOPD-internen Repräsentationen zweier konnektierter Transducer *T1:TransducerService* und *T2:TransducerService*, sowie ein SODAPOPD-internes Kanalobjekt *CT1:ChannelThread*. Erreicht eine Nachricht, die von einem Transducer ausgesendet wurde, ein Kanalobjekt (siehe Abbildung 47 Operation 6 und Abbildung 48 Operation 1) und entspricht diese Nachricht einem

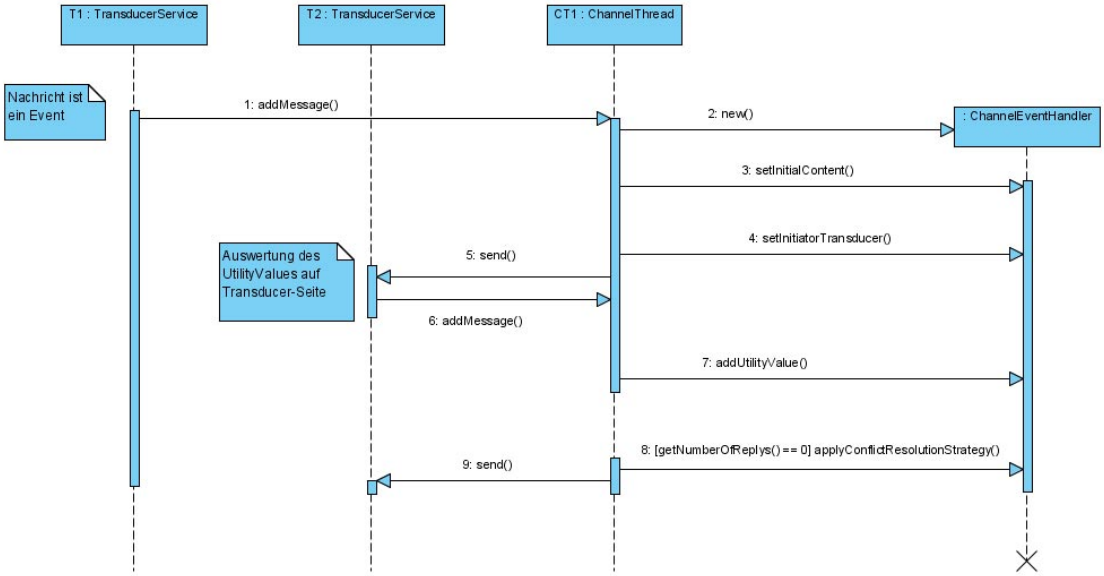


Abbildung 48: Verarbeitung und Zustellung eines Events durch die SODAPOPD-eigenen Kanalobjekte.

kodierten *Event* so initialisiert der nebenläufige Prozess *ChannelThread* hierfür ein Objekt der Klasse *ChannelEventHandler* (Abbildung 48 Operation 2). Diesem werden die Informationen über den Inhalt des Events und über den Initiator (Sender) des Events mitgeteilt (Abbildung 48 Operation 3 und Operation 4). Nach der Ermittlung aller Identifikationsnummern der Transducer, die als Event-Senken auf diesem entsprechenden Kanal registriert sind (in Abbildung 48 nicht dargestellt) wird all diesen über ihren *Transducer-Service* die Nachricht als *Event* mitgeteilt. Diese Nachricht (siehe Abbildung 48 Operation 5) dient der Evaluation der UtilityValues auf der Transducer-Seite (siehe auch Abbildung 50 Operation 5). Nachdem die UtilityValue-Funktionen der Transducer-eigenen *EventHandler* (siehe Abbildung 45) ausgewertet wurden, gelangen diese wiederum als logische Nachrichten in das Kanalobjekt *CT1:ChannelThread* (Abbildung 48 Operation 6). Die in dieser Nachricht kodierte UtilityValue-Funktion wird dem spezialisierten *ChannelEventHandler*-Objekt übergeben (Abbildung 48 Operation 7). Sind alle[43] UtilityValues der an dem entsprechenden Kanal als Event-Senken registrierten Transducer eingegangen, führt das Objekt *ChannelEventHandler* die kanaleigene Konfliktlösungsstrategie aus (Abbildung 48 Operation 8). Als Ergebnis dieser Strategieausführung ergeben sich diejenigen Transducer die das Event (bzw. Teile des Events) erhalten. An diese *Gewinner* wird entsprechend ein *Event* gesendet (Abbildung 48 Operation 9). Nach Bearbeitung aller Aufgaben die mit diesem einen Event verbunden waren, terminiert das *ChannelEventHandler*-Objekt.

[43]Der Übersichtlichkeit geschuldet wurde in Abbildung 48 nur eine potentielle Event-Senke (T2) illustriert. Die Anzahl der möglichen Event-Senken ist jedoch unlimitiert.

Verarbeitung von Remote Procedure Calls im SODAPOPD(AEMON): Ein Vergleich von Abbildung 48 und Abbildung 49 verdeutlicht, dass die internen Abläufe im SODAPOPD zur Behandlung von Events und zur Behandlung von Remote Procedure Calls ähnlich sind. Nach Eingang eines RPCs und Zuweisung zum entsprechenden Kanal (an den der Remote Procedure Call adressiert ist) initialisiert der zuständige *ChannelThread* ein Objekt der Klasse *ChannelRPCHandler* und teilt diesem den initialen RPC sowie die Identifikationsnummer des initiierenden Transducers mit (Abbildung 49 Operation 1 bis Operation 4). Nach der Ermittlung aller Identifikationsnummern der Transducer, die als RPC-Senken (gleich: Event-Quellen) auf diesem entsprechenden Kanal registriert sind (in Abbildung 49 nicht dargestellt) wird all diesen über ihren *TransducerService* die Nachricht als *RPC* mitgeteilt. Diese Nachricht (siehe Abbildung 49 Operation 5) dient der Evaluation der UtilityValues auf der Transducer-Seite (siehe auch Abbildung 51 Operation 4). Nach Eingang der UtilityValues der einzelnen Transducer[44] werden diese dem entsprechenden *ChannelRPCHandler*-Objekt zugewiesen (Abbildung 49 Operation 6 und Operation 7). Sind alle UtilityValues der an diesem Kanal als RPC-Senken konnektierten Transducer eingegangen, so wird die entsprechende auf diesem Kanal definierte Konfliktlösungsstrategie ausgeführt und die *Gewinner* sowie die an diese Gewinner zu sendende Nachricht ermittelt[45] und ihnen zugesendet (Abbildung 49 Operation 8 und Operation 9).

Nach Bearbeiten des Remote Procedure Calls senden die Transducer die Antwort auf den RPC an den SODAPOPD(AEMON) zurück (Abbildung 49 Operation 10) der diese Antwort wiederum dem für diesen RPC zuständigen *ChannelRPCHandler*-Objekt übergibt. Eine erneute Anwendung der kanaleigenen Strategie liefert die sich ergebende Antwort auf den initialen RPC (Abbildung 49 Operation 11 und Operation 12). In einem letzten Schritt wird diese RPC-Antwort an den initialen Anfrager versendet (Abbildung 49 Operation 13) und das *ChannelRPCHandler*-Objekt terminiert. Den Fall, dass die Evaluierung der UtilityValue-Funktionen der beteiligten Transducer keine positiven Werte ergeben hat, ist in Abbildung 49 nicht dargestellt. In diesem Fall wird dem RPC-Initiator eine entsprechende Negativ-Nachricht zugestellt. Diese ersetzt Operation 9 in Abbildung 49.

Verarbeitung von Events auf Seiten der Transducer: Nachdem die Behandlung von Events und Remote Procedure Calls im SODAPOPD(AEMON) spezifiziert und illustriert wurde, soll auch deren Verarbeitung auf Seiten der Transducer berücksichtigt werden. Abbildung 50 und Abbildung 51 verdeutlichen besonders das prinzipielle Vorgehen, auf welche Weise die grundsätzlichen Kommunikationsmechanismen von der Konzeption und Implementierung eines Transducers verborgen werden.

An dieser Stelle sei herausgestellt, dass der Programmierer einzig den Transducer an sich und die Methoden der Objekte *EventHandler* und *RPCHandler* zu implemen-

[44] Auch hier wurde aus Gründen der Übersichtlichkeit nur ein möglicher Transducer als RPC-Senke illustriert. Ähnlich wie bei der Behandlung Events ist die Anzahl der möglichen RPC-Verbraucher unlimitiert.

[45] Selbstverständlich kann es sich hierbei auch nur um einen einzigen Transducer handeln, der den RPC zur Bearbeitung erhält.

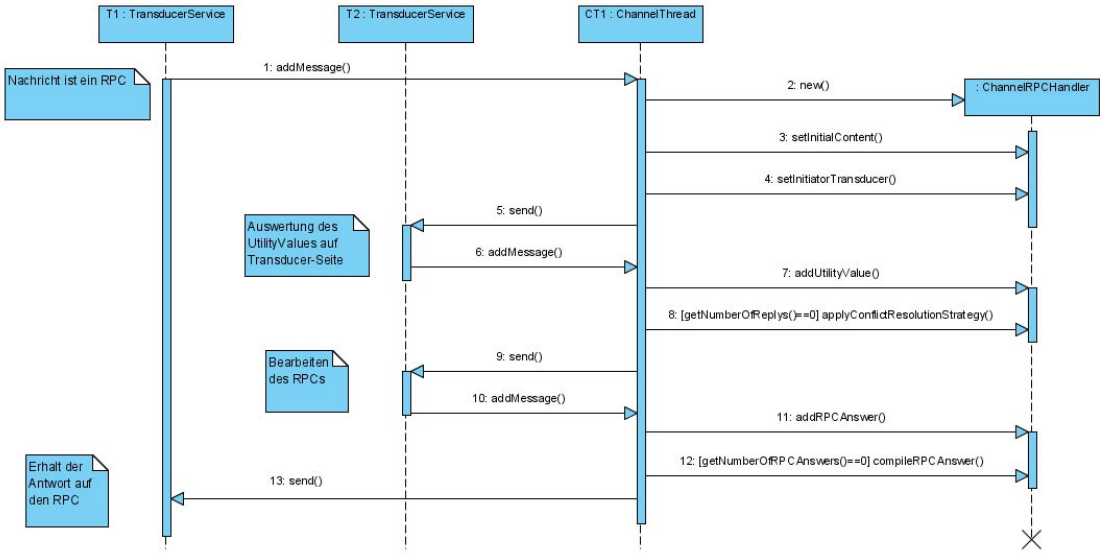


Abbildung 49: Verarbeitung und Zustellung eines Remote Proceduce Calls, sowie die Rückführung der RPC-Antwort an den Initiator durch die SODAPOPD-eigenen Kanalobjekte.

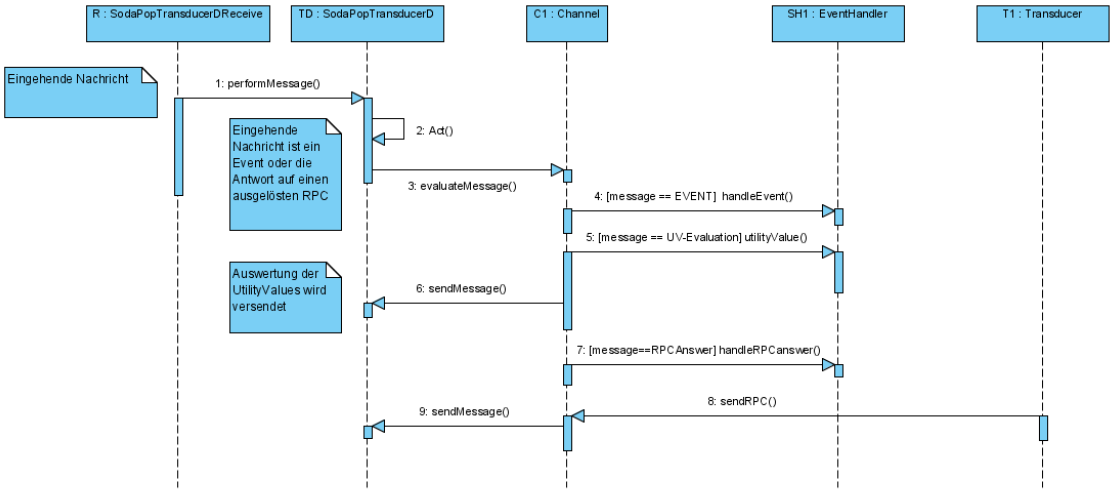


Abbildung 50: Interne Vorgänge eines Transducers, der an einem Kanal als Event-Senke registriert ist. Mittels eines dem Kanalobjekt zugewiesenen EventHandlers ist es möglich, auf Events zu reagieren und diese zu verarbeiten. Gleichzeitig ist es diesem Transducer möglich Remote Procedure Calls abzusetzen und Antworten auf diese RPCs zu verarbeiten.

tieren hat. Aus diesem Grund befinden sich diese Klassen in den Sequenzdiagrammen in Abbildung 50 und Abbildung 51 jeweils rechts illustriert. Nachrichten, die vom SO-

DAPOPD(AEMON) an den Transducer gerichtet sind, erreichen den Transducer immer mittels des Objektes *SodaPopTransducerDReceive*, das nach Erhalt einer Nachricht die Methode *performMessage()* des Objektes *SodaPopTransducerD* aufruft. Nachrichten die somit bei Operation 11 in Abbildung 46, bei den Operationen 5 und 9 in Abbildung 48 und bei den Operationen 5, 9 und 13 in Abbildung 49 an einen Transducer versendet werden, erreichen den Transducer immer im Objekt *SodaPopTransducerDReceive* und werden dann zur weiteren Bearbeitung an das Objekt *SodaPopTransducerD* weitergeleitet. Nach Auswertung der Nachricht durch den nebenläufigen Prozess *SodaPopTransducerD* kann die eingehende Nachricht entweder ein *Event*, ein *RemoteProcedureCall*, die *Antwort* auf einen Remote Procedure Call oder die Aufforderung zur *Evaluierung* der UtilityValue-Funktion sein. Diese vier unterschiedlichen Nachrichtenarten entsprechen den unterschiedlichen möglichen logischen Nachrichten die an einen Kanal gerichtet sein können. Abbildung 50 und Abbildung 51 zeigen die Behandlung aller vier logischen Nachrichten auf. Die Behandlung der administrativen Nachrichten, die neben den logischen Nachrichten existieren können, sind in den Sequenzen von Abbildung 50 und Abbildung 51 nicht illustriert. Diese betreffen die Konnektion des Transducers am SODAPOPD(AEMON) und werden vom Objekt *SodaPopTransducerD* behandelt. Nachdem das Objekt *SodaPopTransducerD* das angesprochene Kanalobjekt aus der Nachricht erkannt hat, übergibt er dem entsprechenden Kanalobjekt die Nachricht, das die weitere Verarbeitung übernimmt (siehe Abbildung 50 Operation 3).

Handelt es sich bei der Nachricht um ein zugeteiltes Event, kann dieses sofort vom entsprechenden *EventHandler* des Transducer-eigenen Kanalobjektes verarbeitet werden (Abbildung 50 Operation 4). Handelt es sich um die Aufforderung die UtilityValue-Funktion auszuwerten, ruft das Kanalobjekt die entsprechende Evaluierungsmethode des zuständigen *EventHandler*-Objektes auf und sendet die ausgewertete UtilityValue-Funktion an den SODAPOPD(AEMON) zurück (Abbildung 50 Operation 5 und Operation 6). Ist die Nachricht eine Antwort auf einen von diesem Transducer auf dem betroffenen Kanal initiierten Remote Procedure Call so ruft das Kanalobjekt die Methode *handleRPCAnswer* im *EventHandler*-Objekt auf. Die Operationen 8 und 9 in Abbildung 51 verdeutlichen, dass es einem Transducer, der als Event-Senke auf dem entsprechenden Kanal konnektiert ist, erlaubt ist Remote Procedure Calls abzusenden.

Verarbeitung von Remote Procedure Calls auf Seiten der Transducer: Falls ein Transducer als Event-Quelle anstatt als Event-Senke an einem Kanal registriert ist, verdeutlicht Abbildung 51 die logischen Unterschiede im Vergleich zu Abbildung 50. Erreicht den Transducer ein Remote Procedure Call (Operation 6 in Abbildung 51) so wird das Ergebnis dieser Ausführung durch das Kanalobjekt an den SODAPOPD(AEMON) zurückgesendet (Operation 7 in Abbildung 51). Die Operationen 8 und 9 in Abbildung 51 verdeutlichen, dass es einem Transducer, der als Event-Quelle auf dem entsprechenden Kanal konnektiert ist, selbstverständlich erlaubt ist, zu jeder Zeit autonom Events in den Kanal abzusetzen. Die Auswertung der UtilityValue-Funktion und deren Versendung (Operation 4 und 5 in Abbildung 51) entspricht dem in vorherigen Abschnitt spezifizierten Vorgehen.

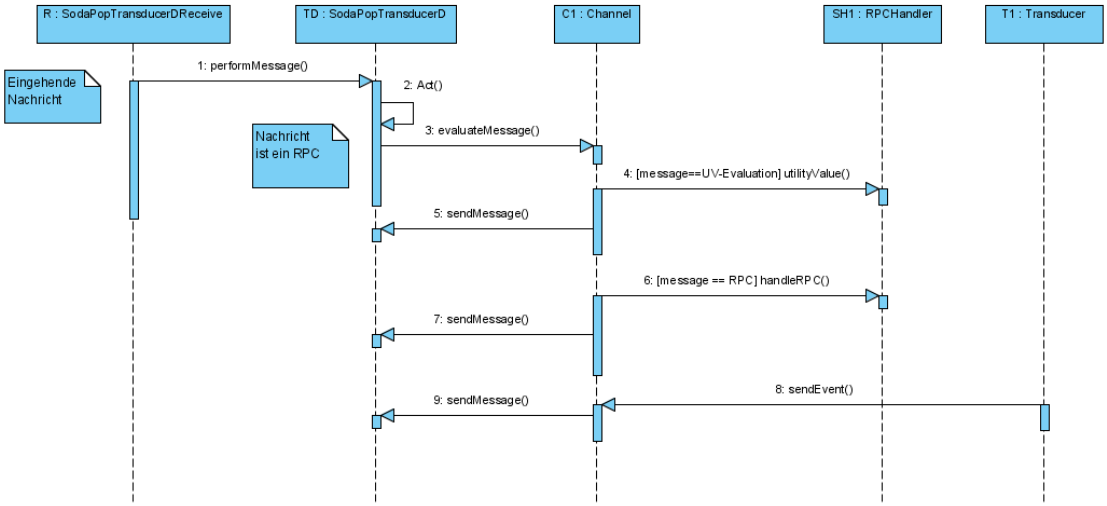


Abbildung 51: Interne Vorgänge eines Transducers, der an einem Kanal als Event-Quelle registriert ist. Mit einem RPCHandler, der dem Kanalobjekt entsprechend zugewiesen ist, ist es möglich auf RPCs zu antworten. Gleichzeitig ist es diesem Transducer möglich, Events auf den Kanal zu verschicken.

5.1.4 Unterlagertes Kommunikationsprotokoll

Für die Übertragung von Nachrichten zwischen einem Transducer und einem SODAPOPD(AEMON) sind im Wesentlichen die folgenden Objekte zuständig:

- *SodaPopTransducerD* und *SodaPopTransducerDReceive* auf Seiten eines Transducers (siehe Abbildung 45)
- und *TransducerService* auf Seiten des SODAPOPD(AEMON)s (siehe Abbildung 44)

Auf Seiten der Transducer werden die Nachrichten gemäß ihres semantischen Inhaltes auf die dem Transducer zugehörigen Kanalobjekte (siehe Abbildung 50 und Abbildung 51) verteilt, auf Seiten des SODAPOPD(AEMON)s auf die dort zugehörigen Kanalobjekte (siehe Abbildung 48 und Abbildung 49). Diese Objekte sind dem Programmierer, der die hier definierte und spezifizierte Software-Infrastruktur zur Erstellung eigener Komponentenensembles verwendet, komplett verborgen. Der Entwickler muss sich nicht auf die Implementation des hier definierten Protokolls konzentrieren, sondern kann sein volles Augenmerk auf die Funktionalitäten legen, die die Eigenschaften der zu entwickelnden Transducer bestimmen. Das Format und die Syntax der unterlagert von den Transducern und den SODAPOPD(AEMON)s ausgetauschten Nachrichten ist angelehnt an das in der Literatur bekannte KQML-Format [145]. Nachrichten besitzen einen Befehl, der gefolgt wird von einer Reihe an Parametern, die mit einem Wert belegt werden können. Der Befehl gibt die grundsätzliche Bestimmung einer Nachricht wider, während die Parameter den Inhalt der Nachricht näher spezifizieren und gewisse Ordnungsmechanismen innerhalb der Partnerkomponenten, die die Nachrichten austauschen, zulassen. Grundsätzlich

lassen sich zwei unterschiedliche Arten an Nachrichten unterscheiden: die administrativen Nachrichten und die logischen Nachrichten.

Administrative Nachrichten zur Konnektion am SODAPOPD(AEMON) **und zur Definition von Kanälen:** Zur Konnektion sendet ein Transducer die folgende Nachricht an den entsprechenden SODAPOPD(AEMON) (siehe Operationen 6 bis 9 in Abbildung 46).

```
(register
     :receiver SODAPOPD
     :ref-no RW)
```

Die Referenznummer *RW* kann hierbei durch den Transducer frei bestimmt werden. Der Begriff *SODAPOPD* als Empfänger der Nachricht ist fest vorgeschrieben. Mit diesem Begriff identifiziert der SODAPOPD(AEMON), dass administrative Dienstleistungen von ihm angefordert werden. Nach Vergabe einer eindeutigen Identifikationsnummer und Eintragung in das *TransducerManagementSystem* sendet der SODAPOPD(AEMON) die folgende Nachricht an den Transducer zurück (siehe Operation 11 in Abbildung 46).

```
(accept
     :receiver Transducer-ID
     :sender SODAPOPD
     :ref-no RW)
```

Die *Transducer-ID* in dieser Nachricht ist die eindeutige Identifikationsnummer für den Transducer, die dieser in allen folgenden Nachrichten verwenden muss. Das generelle Abmelden eines Transducers von seinem SODAPOPD(AEMON) geschieht in gleichartiger Weise. Hierbei wird die folgende Nachricht an den SODAPOPD(AEMON) gesendet:

```
(unregister
     :sender Transducer-ID
     :receiver SODAPOPD)
```

Die Parameter *sender* und *receiver* verdeutlichen hier noch einmal, dass es sich um eine direkte Verbindung von Transducer und dem SODAPOPD(AEMON) handelt. Kanäle werden mittels den folgenden Nachrichten von den Transducern gegenüber dem SODAPOPD(AEMON) definiert:

```
(subscribe
     :receiver SODAPOPD
     :sender Transducer-ID
     :channel-in Kanalname,IN,Kanalstrategie)
```

bzw.

```
(subscribe
     :receiver SODAPOPD
     :sender Transducer-ID
     :channel-out Kanalname,OUT,Kanalstrategie)
```

Hiermit definiert der Transducer einen Kanal mit dem angegebenen Namen als Bezeichner und der angegebenen Strategie als kanaleigene Konfliktlösungsstrategie. Im ersten Fall wird der Transducer als Event-Senke registriert (IN), im zweiten Fall als Event-Quelle (OUT).

Sollte im SODAPOPD(AEMON) der Kanal noch nicht definiert und damit angelegt sein, so startet der SODAPOPD ein Objekt der Klasse *ChannelThread* (siehe Operationen 7 und 8 in Abbildung 47) dessen *ChannelRPCHandler*- bzw. *ChannelEventHandler*-Objekte die mit dem Bezeichner *Kanalstrategie* versehene Konfliktlösungsstrategie anwenden. Der SODAPOPD(AEMON) hält für diesen Zweck unterschiedliche Strategiealgorithmen bereit, die hier verwendet werden können. Die Abmeldung von Kanälen geschieht auf ähnliche Art und Weise mittels

```
(unsubscribe
     :receiver SODAPOPD
     :sender Transducer-ID
     :channel-in Kanalname)
```

bzw.

```
(unsubscribe
     :receiver SODAPOPD
     :sender Transducer-ID
     :channel-out Kanalname)
```

Logische Nachrichten zur Versendung von Events und Remote Procedure Calls, zur Beantwortung von RPCs und zur Evaluierung der UtilityValues: Analog zu den Aktivitäten der Transducer und der SODAPOPD(AEMON)s, die in Abbildung 48, Abbildung 49, Abbildung 50 und Abbildung 51 illustriert wurden, ist ein unterlagertes semantisches Protokoll definiert, welches die Abwicklung von Events und Remote Procedure Calls ermöglicht. Sendet ein Transducer ein Event (siehe Abbildung 51 Operation 8 und Operation 9) so wird die folgende Nachricht an den SODAPOPD(AEMON) verschickt:

```
(event
     :channel Kanalname
     :sender Transducer-ID
     :content ( event ))
```

Hierbei wird im Parameter *channel* der Bezeichner des Kanals kodiert, in welchem das Event versendet wird. *Transducer-ID* bezeichnet den Absender des Events. Im Parameter *content* befindet sich das eigentliche Event. Diese Nachricht erreicht den SODAPOPD(AEMON) im Transducer-eigenen *TransducerService*, wie es in Abbildung 48 in Operation 1 illustriert ist. Das zuständige Kanalobjekt (siehe Abbildung 48 Operation 5) sendet daraufhin die folgende Nachricht an die konnektierten Transducer:

```
(proposeEvent
     :channel Kanalname
     :ref-no RW
     :content ( event ))
```

Ist ein Transducer als Event-Senke an diesem Kanal konnektiert, erreicht diesen die Nachricht, so wie in Abbildung 50 Operation 1 dargestellt. Der Befehl *proposeEvent* zeigt an, dass das entsprechende Kanalobjekt die UtilityValue-Funktion des zugehörigen *Event-Handler*-Objekt zu evaluieren hat. Das Ergebnis dieser Evaluation wird zurückgegeben und an den SODAPOPD(AEMON) gesendet (siehe Abbildung 50 Operationen 5 und 6). Im Falle, dass die Auswertung der UtilityValue-Funktion positiv verlaufen ist, wird die folgende Nachricht versendet:

```
(acceptEvent
     :sender Transducer-ID
     :channel Kanalname
     :ref-no RW
     :content (utilityValue))
```

Im negativen Fall lautet die Nachricht wie folgt:

```
(rejectEvent
     :sender Transducer-ID
     :channel Kanalname
     :ref-no RW
     :content (false))
```

Um den SODAPOPD(AEMON)-eigenen Kanalobjekten die Möglichkeit der Zuordnung von Initialevent zu zugehörigen UtilityValues zu geben, beziehen sich die Antworten auf die Referenznummer *RW*. Nachdem alle UtilityValue-Funktionen der konnektierten Transducer eingegangen sind und die kanaleigene Konfliktlösungsstrategie ausgeführt wurde (siehe Abbildung 48 Operationen 6 bis 8) wird das Event denjenigen Transducer gesendet (bzw. an diejenigen Transducer gesendet), der bei der Ausführung der Konfliktlösungsstrategie als Sieger hervorging (Abbildung 48 Operation 9). Die Nachricht, die dabei vom SODAPOPD(AEMON) an den Transducer gesendet wird, lautet:

```
(event
     :channel Kanalname
     :content ( event ))
```

Der Befehl *event* gibt hierbei an, dass (gemäß Abbildung 50 Operation 4) das Event dem *EventHandler* des entsprechenden Kanalobjektes zur weiteren Bearbeitung zugeführt wird (*handleEvent()-Methode*). Die Zuteilung des Events von der Event-Quelle zu möglichen Event-Verbrauchern ist somit abgeschlossen. Findet sich kein passender Event-Verbraucher, so geschieht nach der Ausführung der kanalabhängigen Konfliktlösungsstrategie nichts weiter. Das letztlich zugeteilte Event muss syntaktisch nicht gleich dem initialen Event sein. Dies kann abhängig sein von der angewendeten Konfliktlösungsstrategie.

Transducer versenden Remote Procedure Calls wie Aktion 8 in Abbildung 50 illustriert. Hierbei versendet der Transducer an den SODAPOPD(AEMON) die folgende Nachricht:

```
(rpc
     :channel Kanalname
     :sender Transducer-ID
     :ref-no RW-RPC
     :content ( rpc ))
```

Im Gegensatz zum Verschicken von Events wird hier der Befehl *rpc* verwendet. Zusätzlich vergibt der Transducer eine RPC-Referenznummer, die beim Aufruf der Methode *sendRPC()* zurückgegeben wird. Dies macht es dem Transducer möglich, eine interne Verwaltung für die nach außen gegebenen RPCs und deren Antworten aufzubauen. Nach Eingang der Nachricht im SODAPOPD(AEMON)-eigenen *TransducerService* und der Initialisierung eines Kanal-eigenen *ChannelRPCHandler*s wird dieser RPC an die konnektieren Transducer, die als Event-Quellen an dem betreffenden Kanal registriert sind, verschickt. Die Nachricht, die in Abbildung 49 den SODAPOPD(AEMON) verlässt, lautet wie folgt:

```
(proposeRPC
     :channel Kanalname
     :ref-no RW1
     :content ( rpc ))
```

Diese Nachricht erreicht die Transducer, wie in Abbildung 51 in den Operationen 3 bis 5 illustriert. Nach Auswertung der UtilityValue-Funktion des entsprechenden *RPCHandler*-Objektes des zuständigen Transducer-eigenen Kanalobjektes, wird im positiven Falle die folgende Antwort an den SODAPOPD(AEMON) geschickt:

```
(acceptRPC
     :sender Transducer-ID
     :channel Kanalname
     :ref-no RW1
     :content (utilityValue))
```

Wird die UtilityValue-Funktion negativ evaluiert, lautet die Antwort wie folgt:

```
(rejectRPC
     :sender Transducer-ID
     :channel Kanalname
     :ref-no RW1
     :content (false))
```

Die Referenznummer *RW1* entspricht im Allgemeinen syntaktisch nicht der Referenznummer *RW-RPC*. Die Letztere wird vom SODAPOPD(AEMON) für die Antwort an den RPC-Initiator weiterverwendet. Für den Erhalt der Zwischenergebnisse wie UtilityValues oder RPC-Antworten werden neue interne Referenznummern zur Vermeidung von Ambiguitäten verwendet. Sind im Kanalobjekt (siehe Abbildung 49 Operation 7) die UtilityValues aller konnektierten Transducer eingegangen, so wird die Konfliktlösungsstrategie ausgeführt, und der Remote Procedure Call den entsprechenden Transducern, die sich als Ergebnis der Ausführung der Kanalstrategie ergeben haben, zugesendet. Hierbei wird im unterlagerten Protokoll die folgende Syntax verwendet:

```
(rpc
     :channel Kanalname
     :ref-no RW2
     :content ( rpc ))
```

Dieser Remote Procedure Call erreicht den Transducer, so wie in Abbildung 51 Operation 6 illustriert. Das Resultat des Remote Procedure Calls wird dann mit der Nachricht

```
(rpcResponse
     :channel Kanalname
     :sender Transducer-ID
     :ref-no RW2
     :content ( rpcResponse ))
```

vom Transducer zurück an den SODAPOPD(AEMON) gesendet. Nach Eingang dieser Antwort wie in Abbildung 49 in den Operationen 10 bis 12 illustriert, geht die endgültige Antwort auf den RPC an den initialen Anfrager zurück (siehe Operation 13 in Abbildung 49 und kann von diesem weiter verarbeit werden (Operation 7 in Abbildung 50):

```
(rpcResponse
     :channel Kanalname
     :ref-no RW-RPC
     :content ( rpcResponse ))
```

Das Protokoll für die RPC-Antworten (Nachricht mit dem Befehl *rpcResponse*) enthält den Parameter *sender*, an dem der SODAPOPD(AEMON) erkennen kann, von welchem Transducer die Antwort gekommen ist. Dies dient der internen Verwaltung des RPCs und der sich daraufhin ergebenden Antworten. Sollte die Ausführung der Konfliktlösungsstrategie ergeben haben, dass ein Remote Procedure Call von mehreren Transducern in Kooperation auszuführen ist, so wird für jeden dieser Teilaufträge dieselbe Referenznummer *RW2* verwendet werden. Diese Referenznummer dient zu internen Verwaltung der *RPCHandler*-Objekte.

5.1.5 Beispiele zur Programmierschnittstelle

An einem einfachen Beispiel sollen die Prinzipien verdeutlicht werden, die der Spezifikation zugrunde liegen. Die Beispielanwendung soll aus drei unterschiedlichen Komponenten bestehen:

1. Order-Transducer: ein Transducer der einen Remote Procedure Call zum Berechnen der Addition $x + 1$ auf den Kanal *addition* gibt

2. SE-Transducer: Ein Transducer, der in der Lage ist die Addition $x + 1$ auszuführen, wenn x eine *gerade* Zahl ist

3. SUE-Transducer: Ein Transducer, der in der Lage ist die Addition $x+1$ auszuführen, wenn x eine *ungerade* Zahl ist.

Abbildung 52 illustriert diese Beispielanwendung. Der schwarze vertikale Balken symbolisiert den Kanal mit dem Bezeichner *addition*, an dem die drei Komponenten konnektiert sind. Die Richtung der Pfeile verdeutlicht hierbei die Art der Konnektion. Der Order-Transducer (rechts in Abbildung 52) wird hierbei Remote Procedure Calls in den Kanal geben, die von den beiden anderen Transducern bearbeitet werden können. Gemäß

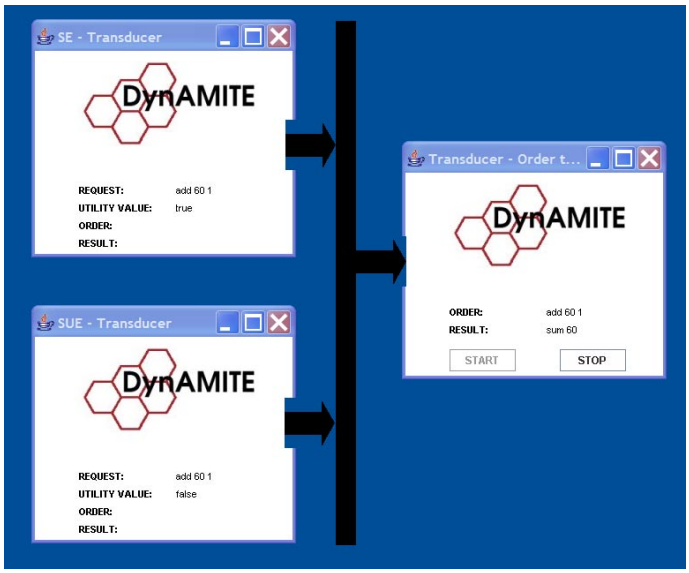


Abbildung 52: Beispielanwendung mit drei unterschiedlichen Transducern, die an demselben Kanal verbunden sind. Die Pfeile stellen die Art der Verbindung – Event-Quelle oder Event-Senke bzw. RPC-Senke oder RPC-Quelle – dar. Rechts ist dargestellt der Order-Transducer, der RPCs in den Kanal sendet, links die beiden Additions-Transducer, die die RPCs bearbeiten.

der Spezifikation ist der Order-Transducer somit eine Event-Senke, die beiden anderen Event-Quellen. Die Funktionsweise erschließt sich beim Blick auf den Programmiercode der unterschiedlichen Komponenten. Zuerst die Komponente Order-Transducer, die Remote Procedure Calls auf den Kanal gibt:

```
1: public class OrderTransducer{

2: static Channel addition = null;

3: public OrderTransducer(){
4:       addition = new Channel("addition", "IN", "random");
5:       addition.subscribe(new OrderEventHandler(this));
6: }

7: public void sendNextRPC(String sum){
8:       addition.sendRPC("add " + sum + " 1");
9: }

10: public static void main (String[] args){
11:       new OrderTransducer();
12: }}
```

Der nötige Code des Order-Transducer ist sehr kompakt. Er besitzt Anteile an einem Kanal (Zeile 2), der initialisiert wird mit einem Namen (hier: *addition*), der eigenen Rolle im Bezug auf diesen Kanal (hier: *IN*) und dem Namen der Kanalstrategie, die im

Konfliktfalle angewendet werden soll (siehe Kapitel 6.2.1 für die detaillierte Diskussion der Konfliktlösungsstrategie *random*). In dem Beispiel hier ist der Bezeichner der Strategie der Begriff *random*, da der hier verwendete SODAPOPD(AEMON) eine Strategie ausführen kann, die im Konfliktfalle per Zufall aus mehreren geeigneten Transducern einen auswählt. Im Anschluss an die Definition des Kanalobjektes (Zeile 4) wird diesem ein *EventHandler* zugewiesen (Zeile 5). Dieser *EventHandler* sieht wie folgt aus:

```
1: public class OrderEventHandler extends EventHandler{

2: OrderTransducer orderTransducer = null;

3: public OrderEventHandler(OrderTransducer ot){
4:       this.orderTransducer = ot;}

5: public void handleRPCAnswer(String rpcAnswer, String rpcNo){
6:       // rpcAnswer looks like: "sum x"
7:       // extract result x from rpcAnswer
8:       // put result x on the graphical user interface
9:       // use x as the new starting point for the next RPC
7:       orderTransducer.sendNextRPC(x);
8: }

9: public String utilityValue(String message){return "true";}

10: public handleEvent(String event){}
11: }
```

Der *EventHandler* muss laut Abbildung 45 drei Methoden implementieren. Die Methode zur Evaluierung des UtilityValues (Zeile 9), die Methode zur Verarbeitung von Events (Zeile 10) und die Methode zur Verarbeitung von Antworten auf versendete Remote Procedure Calls. Da die Aufgabenstellung eingegrenzt ist und im Beispielszenario keine Events auftreten, können die beiden erstgenannten Methoden leer bleiben. Die Antworten auf versendete RPCs werden in der entsprechenden Methode ab Zeile 5 behandelt. Der eigentliche Code ist hier für das Verständnis nicht interessant (wie Abbildung 52 zeigt, wird das Ergebnis auf der GUI ausgegeben). Im Anschluss daran wird sofort der nächste RPC abgeschickt (Zeile 7 und Transducercode ebenfalls Zeile 7). Die Programmierung der beiden anderen Transducer gestaltet sich ähnlich kompakt:

```
1: public class SETransducer {

2: static Channel addition = null;

3: public SETransducer(){
4:       addition = new Channel("addition", "OUT", "random");
5:       addition.subscribe(new SERPCHandler(this));
6: }

7: public static void main (String[] args){
8:       new SETransducer();
9: }}
```

Der SE-Transducer definiert (analog wie der Order-Transducer) einen Kanal mit dem Bezeichner *addition*, der die Kanalstrategie *random* ausführen wird (Zeile 4). Die Rolle dieses Transducers ist hier jedoch die der Event-Quelle. Mit dem Bezeichner *OUT* gibt dieser Transducer an, dass er in der Lage ist Remote Procedure Calls zu bearbeiten. Entsprechend wird diesem Kanalobjekt ein RPCHandler zugeordnet (Zeile 5). Dieser RPCHandler programmiert sich wie folgt:

```
1: public class SERPCHandler extends RPCHandler{

2: SETransducer sETransducer = null;

3: public SERPCHandler(SERPCHandler set){
4:       this.sETransducer = set;}

5: public String handleRPC(String rpc){
6:       // RPC looks like: "add x 1"
7:       // make the addition: y = x + 1
8:       // put task and result on the gui
9:       // return the result
10:       return "sum " + String.valueOf(y);
11: }

12: public String utilityValue(String rpc){
13:       // RPC looks like: "add x 1"
14:       // extract x
15:       if ((x % 2) == 0) return "true";
16:       else return "false";
17:}
18:}
```

Als *RPCHandler* muss diese Klasse die Methoden zur Evaluierung des UtilityValues (Zeile 12) und zur Verarbeitung von Remote Procedure Calls (Zeile 5) implementieren (siehe auch Abbildung 45). Die Evaluierung des UtilityValues ergibt *true* falls die Basiszahl x gerade ist. Im anderen Fall wird *false* zurückgegeben. Bei der Bearbeitung des RPCs wird das Ergebnis berechnet und der String *sum y* mit $y = x + 1$ als RPC-Antwort zurückgegeben. Der Programmiercode des SUE-Transducers gestaltet sich genau analog mit dem Unterschied, dass hier in der Methode zur Evaluierung des UtilityValues bei ungerader Basiszahl x ein *true* und bei gerader Basiszahl x ein *false* zurückgegeben wird. Werden die drei Transducer gestartet so ergibt sich die Anwendung wie in Abbildung 52 illustriert und die beiden Additions-Transducer übernehmen abwechselnd die Aufgabe $x + 1$ zu berechnen. Jedesmal, wenn das Ergebnis beim Order-Transducer ankommt, wird gleich der nächste Remote Procedure auf den Kanal gegeben.

Die Notwendigkeit einer Konfliktlösungsstrategie macht die Illustration in Abbildung 53 deutlich. Hier sind mehrere Instanzen der hier beschriebenen Additions-Transducer gestartet worden. Auf jeden Additionsauftrag melden mehrere Evaluierungsfunktionen einen positiven Wert. Die Konfliktlösungsstrategie *random*, die für diesen Anwendungsfall auf dem beteiligten Kanal eingesetzt wurde (detailliert besprochen in Kapitel 6.2.1), löst diesen Konflikt durch einfaches „Würfeln“. Somit ist diese Anwendung auch im Kon-

Abbildung 53: Die Beispielanwendung mit einer Vielzahl laufender Transducer, die den vom Order-Transducer gesendeten Remote Procedure Call bearbeiten können.

fliktfalle immer lauffähig. Die Anzahl der beteiligten Transducer ist nicht limitiert.

5.1.6 Diskussion der Transducer-Architektur

In den vorangegangenen Abschnitten wurde die Software-Infrastruktur für selbstorganisierende Komponentenensembles basierend auf dem SODAPOPModell spezifiziert. Die Hauptkomponente hierbei ist der SODAPOPD(AEMON), der für die Transducer die Verwaltung der virtuellen Kanäle, sowie die Ausführung unterschiedlicher Konfliktlösungsstrategien zur Verfügung stellt. Er läuft auf einem physikalischen Gerät als Hintergrunddienst. Mittels einer Programmierschnittstelle (siehe Beispiel in Kapitel 5.1.5) ist es dem Entwickler von Komponenten beziehungsweise Ensembles von Komponenten möglich, Kanäle zu definieren, eigene Bezeichner zu vergeben und den Komponenten die Rolle ihrer Verbindung zu diesen Kanälen zuzuweisen. Die Anzahl der Kanäle, sowie die Anzahl der Verbindungen von und zu diesen Kanälen sind nicht limitiert. Der Entwickler ist somit in der Definition der Komponententopologie vollkommen frei. Jedem Kanal kann eine Konfliktlösungsstrategie zugewiesen werden. Es gilt hier: Diejenige Komponente die einen Kanal mit einem bestimmten Bezeichner als Erste definiert, bestimmt auch den Bezeichner der Kanalstrategie. Komponenten, die sich an einen bereits definierten Kanal anmelden, können den Bezeichner der Konfliktlösungsstrategie nicht mehr bestimmen oder gar ändern. Die Algorithmen für diese Strategien sind im SODAPOPD(AEMON) vorzuhalten.

Abbildung 54 illustriert nochmals anschaulich wie die Programmierschnittstelle die internen Kommunikationsabläufe vor dem Komponentenentwickler verbirgt. Anschaulich ist dargestellt, dass ein Transducer über eine Menge an Kanalobjekten verfügen kann. Alle Kanalobjekte wiederum greifen auf dasselbe Objekt der Klasse *SodaPopTransducerD*

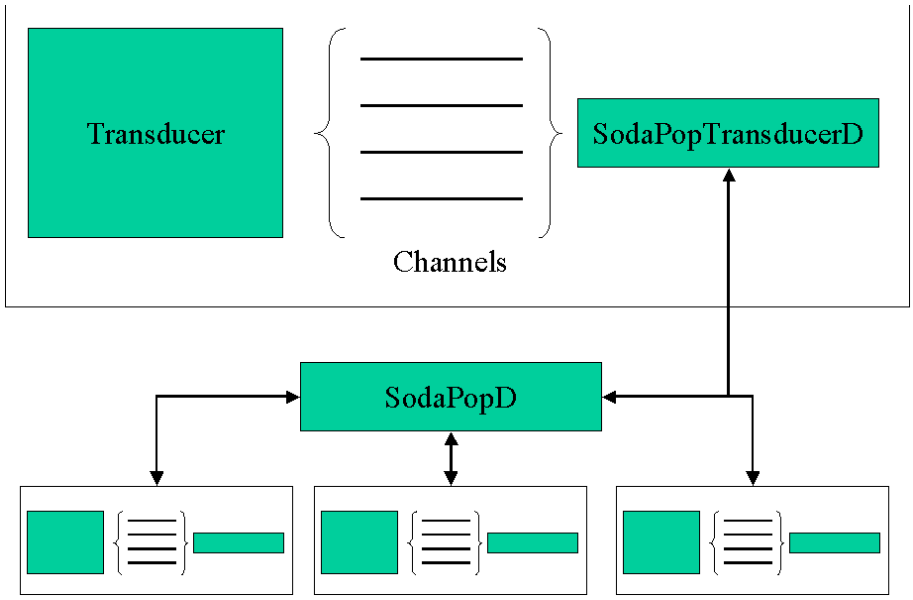


Abbildung 54: Abstrakte Illustration der Programmierschnittstelle zur Entwicklung von SODAPOP-Komponenten.

zu, um die Verbindung und Kommunikation mit dem SODAPOPD(AEMON) zu ermöglichen. Dies ist im Klassendiagramm in Abbildung 45 durch die unterschiedlichen Multiplizitäten der Verbindungen der Klassen *Transducer*, *Channel* und *SodaPopTransducerD* angezeigt. Diese Realisierung entspricht exakt dem ursprünglichen Ansatz im SODAPOP-Modell, dass Transducer Anteile an Kanälen besitzen (ein Kanal somit zwischen Transducern geteilt ist) und diesen Kanälen Funktionen zur Auswertung der UtilityValues bzw. Funktionen zur Bearbeitung von Events und Remote Procedure Calls zur Verfügung stellen. Ein Transducer muss nicht von einer übergeordneten Klasse erben.

Somit können bereits existierende Komponenten durch die Verwendung von Kanalobjekten zu Transducern in einem selbstorganisierenden Komponentenensemble werden. Sie müssen dabei nicht ihre Natur (was der Wechsel der Vererbungshierarchie in einer Komponente bedeuten würde) ändern. Damit der Transducer (durch das Objekt *SodaPopTransducerD*) logische und semantische Nachrichten mit dem SODAPOPD(AEMON) austauschen können, wird das in Kapitel 5.1.4 definierte unterlagerte Protokoll verwendet. Dieses Protokoll wird komplett verborgen vom Benutzer der Programmierschnittstellen benutzt. Wie das Beispiel in Kapitel 5.1.5 zeigt, kommunizieren die Komponenten nur die Inhalte von Events, Remote Procedure Calls, Antworten auf RPCs und die Werte der Evaluierung von UtilityValue-Funktionen.

Die hier in dieser Arbeit vorgestellte Software-Infrastruktur realisiert damit zugleich eine besondere Art des Programmiermodells, in dem die Funktionalitäten von Transducern von deren Kommunikationstätigkeiten gekapselt sind. Die Implementierung der Transducer-API erlaubt es den Anwendungsentwicklern sich auf die Semantik der Nachrichten zu konzentrieren. Die Handlungen der (virtuellen) Kanäle sowie der gesamte unterlagerte Nachrichtenverkehr bleibt während der Implementations- und Ausführungszeit

verborgen.

Die hier vorgestellte Spezifikation ist nicht auf ein physikalisches Gerät beschränkt. Solange eine funktionierende TCP/IP-Verbindung zwischen physikalischen Geräten existiert, können die Komponenten, die ein Ensemble definieren, auf unterschiedlichen Geräten laufen. Eines von diesen Geräten muss den SODAPOPD(AEMON) als Hintergrunddienst vorhalten. Diese Eigenschaft beschränkt *nicht* die Selbstorganisationsmöglichkeiten innerhalb des Komponentenensembles, aber die Möglichkeiten der Dynamik und Selbstorganisation des Geräteensembles. Die Komponenten sind alle gleichberechtigt, jedoch gibt es bei dieser Anordnung mit demjenigen Gerät, das den SODAPOPD(AEMON) als Hintergrunddienst vorhalten muss, eine ausgezeichnete Instanz. Im den nächsten Abschnitten wird die verteilte Implementierung von SODAPOP spezifiziert, die diesen Nachteil aufhebt und die völlige Selbstorganisationsfähigkeiten der beteiligten unterschiedlichen physikalischen Geräte garantieren kann.

5.2 Verteilte Implementierung von SODAPOP

Die SODAPOPD(AEMON)s stellen einen Hintergrunddienst auf jedem Gerät dar. Ein SODAPOPD(AEMON) stellt den an ihm konnektierten Transducern sowohl die virtuellen Kanäle als auch die Ausführung von Konfliktlösungsstrategien und die damit verbundene Zuteilung von Events und Remote Procedure Calls zur Verfügung. Die in Kapitel 5.1 spezifizierte Realisierung für selbstorganisierende Komponentenensembles ermöglicht die Dynamisierung von Komponenten in einem Ensemble bzgl.:

- Komponenten können sich jederzeit in bereits bestehenden Systemen konnektieren und am Datenfluss teilhaben, d.h. Events und Remote Procedure Calls erzeugen, konsumieren und verarbeiten.

- im SODAPOPD(AEMON) können kanalabhängige Konfliktlösungsstrategien spezifiziert und implementiert werden, so dass sie für die Definition von Komponententopologien eingesetzt werden können.

Die in Kapitel 5.1 spezifizierte Realisierung ist somit einsetzbar für Szenarios in denen die Selbstorganisation von Komponenten und deren Dynamik wichtig ist. Dies sind zum Beispiel Szenarios in denen fest installierte Computersysteme oder sonstige feststehende Hardware zum Einsatz kommen. Man kann hier an Raumserver zur Kontrolle eines Vorlesungssaales oder an Heimautomation mit zentralen Recheneinheiten denken. Vollkommene Dezentralisierung wird jedoch mit diesem Ansatz nicht erreicht. Sollten die Transducer von unterschiedlichen Geräten miteinander ein Komponentenensemble bilden, muss immer zuerst ein zentrales Gerät ausgewählt werden, welches für die anderen den Hintergrunddienst des SODAPOPD(AEMON)s bereitstellt. Diese Notwendigkeit verhindert wirklich dynamische Szenarios, in denen physikalische Geräte ein Ensemble betreten und es auch jederzeit wieder verlassen können. Ein vorstellbares Szenario hierbei ist, dass ein Gruppe von Mitarbeitern einen *leeren* Raum betreten. Jeder bringt eine gewisse Anzahl Geräte (z. B. Laptop, PDA) mit sich, vielleicht einer auch einen

Beamer zur Projektion von Dokumenten an eine Leinwand. Dieser ad-hoc zusammengestellte Meetingraum verfügt zu keinem Zeitpunkt über ein zentrales Gerät, welches den Hintergrunddienst SODAPOPD(AEMON) bereithalten könnte. Ein anderes Beispiel ist der Kauf mehrerer Unterhaltungsgeräte zur Ausrüstung eines Wohnzimmers. Es kann einem Benutzer/Käufer schlecht zugemutet werden[46] die Geräte miteinander zu verkabeln, zu definieren welches Gerät bestimmte Hintergrunddienste bereithalten soll und zudem die Geräte dann in der korrekten Reihenfolge (Gerät mit zentralem Hintergrunddienst zuerst) zu starten. Ein Benutzer ist zwar an dem reibungslosen Zusammenspiel seiner Geräte interessiert, jedoch nicht an den implementatorischen Details[47].

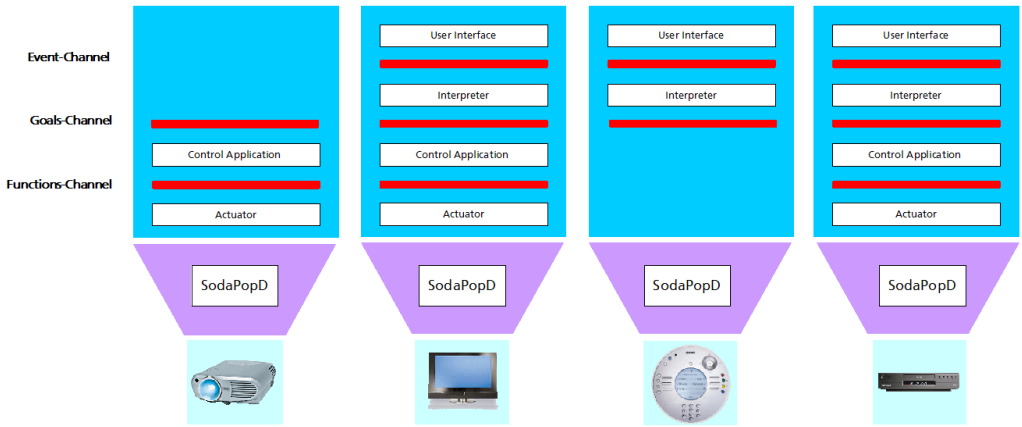
5.2.1 Stellvertreterprinzip der SODAPOPD(AEMON)s

Die SODAPOPD(AEMON)s verwalten sowohl die auf einem physikalischen Gerät vorhandenen Komponenten als auch die durch diese Komponenten definierten Kanäle, die der Kommunikation von Events, Remote Procedure Calls und UtilityValues dienen. Ein SODAPOPD ist somit geeignet der Repräsentant seiner Komponenten nach außen hin zu sein. Speziell kann er die Nachrichten seiner Tranducer nach außen geben, aber auch Nachrichten für seine Transducer empfangen und die Verhandlungen um Events und Remote Procedure Calls mittels der Evaluierung der an ihm konnektierten Transducer stellvertretend übernehmen.

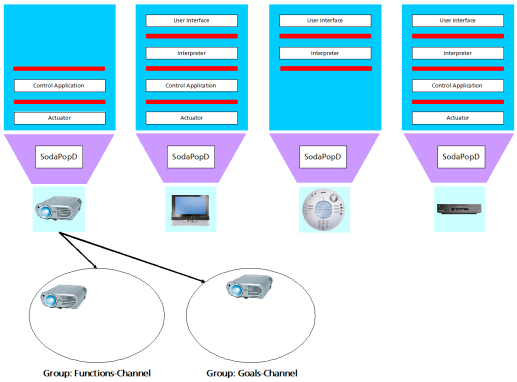
Abbildung 55 illustriert das Prinzip, welches die SODAPOPD(AEMON)s als Stellvertreter ihre Komponenten definiert. Ein Kanal, in den Transducer Ereignisse und Remote Procedure Calls senden und von dem sie Ereignisse und RPCs lesen möchten, repräsentiert gleichfalls eine Kommunikationsgruppe. Ein SODAPOPD(AEMON), der eine solche Kanalgruppe innerhalb seines eigenen physikalischen Gerätes verwaltet ist somit nach außen hin interessiert, ob und welche anderen physikalischen Geräte eine äquivalente Kanalgruppe verwalten. Diese teilen folglich denselben Kanal (siehe Abbildung 39 und Abbildung 55 a)). Deren daran konnektierte Komponenten sind somit Kommunikationspartner. In Abbildung 55 werden beispielhaft vier unterschiedliche Geräte ad-hoc miteinander verbunden. Auf allen vier Geräten haben die geräte-eigenen Transducer Kanäle definiert. Entsprechende Kanaldefinitionen sind geräteübergreifend gleichwertig. Im Beispiel hat der Projektor zwei Transducer, die zusammen zwei Kanäle definiert haben. Diese Kanäle haben eindeutige Bezeichner, im Beispiel *Goals* und *Functions*. Entsprechend definiert der SODAPOPD(AEMON) des Projektors zwei Kanalgruppen mit diesen Bezeichnern (siehe Abbildung 55 b)) und tritt diesen bei. Die Komponente des Fernsehgerätes haben insgesamt drei Kanäle definiert, wovon zwei denselben Bezeichner haben (*Goals* und *Functions*) wie die vom SODAPOPD(AEMON) des Projektors bereits definierten. Der SODAPOPD(AEMON) des Fernsehgerätes tritt somit diesen beiden Gruppen bei und eröffnet im Beispiel zusätzlich eine dritte Gruppe *Event* (Abbildung 55 c)). Die SOD-

[46] Die Realität sieht hier – im Jahr 2007 – immer noch anders aus, wenn man sich die komplizierte Verkabelung von Satellitenreceiver, Videorekorder, DVD-Rekorder und Fernsehapparaten der aktuellen Unterhaltungselektronik vergegenwärtigt.

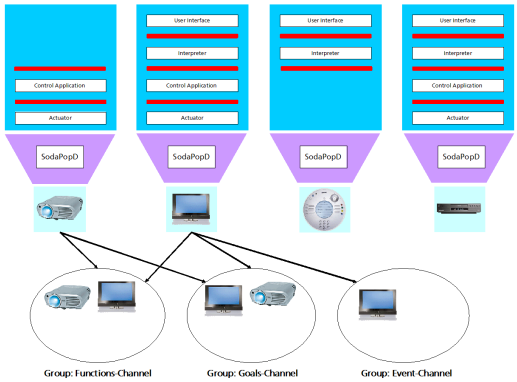
[47] Diese Beobachtung entspricht auch der empirischen Erfahrung, dass ein physikalisches Gerät immer als Ganzes (also als Summe seiner Einzelkomponenten) angesehen wird.



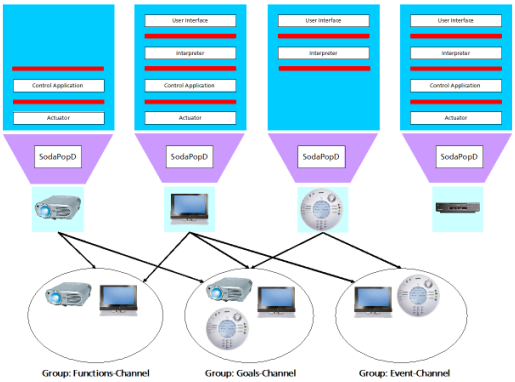
a) Vier Geräte werden ad-hoc miteinander verbunden.



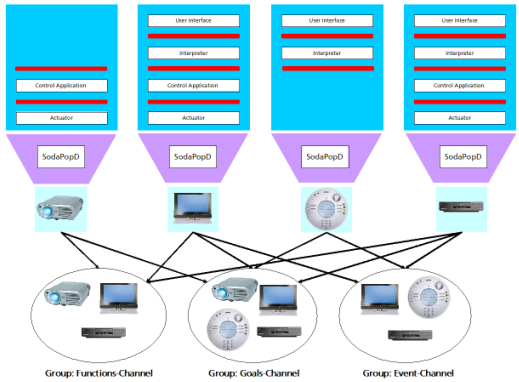
b) Definition von zwei Kanalgruppen . . .



c) Definition einer dritten Kanalgruppe . . .



d) Eintritt in vorhandene Gruppen . . .



e) Erneuter Gruppeneintritt.

Abbildung 55: Die SODAPOPD(AEMON)s der unterschiedlichen Geräte bilden drei unterschiedliche Kanalgruppen aus. Die SODAPOPD(AEMON)s sind in der Abbildung durch ihre Gerätesymbole illustriert.

APOPD(AEMON)s der Fernbedienung und des DVD-Rekorders verhalten sich analog und treten den Gruppen bei, die den auf ihnen definierten Kanälen entsprechen (Abbildung 55

d) und e)).

Die SODAPOPD(AEMON)s müssen somit über eine zusätzliche Kommunikationsebene verfügen, die es ihnen möglich macht:

1. Peer-to-Peer Gruppen zu definieren, diesen Gruppen beitreten zu können und sie auch jederzeit wieder verlassen zu können

2. Events und Remote Procedure Calls innerhalb dieser Gruppen zu kommunizieren

3. Konfliktlösungsstrategien innerhalb einer Gruppe auszuführen und die Ergebnisse dieser Ausführung innerhalb der Gruppe zu kommunizieren.

Die Forderung 1) entspricht dem Ansatz, dass ein SODAPOPD(AEMON) eine Kanalgruppe mit einem bestimmten Bezeichner betritt, sobald der erste seiner Transducer einen Kanal mit diesem Bezeichner definiert hat. Der SODAPOPD(AEMON) verlässt eine solche Gruppe wieder, sobald der letzte seiner Transducer einen Kanal mit einem solchen Bezeichner verlassen hat. Ist eine Kanalgruppe mit dem definierten Bezeichner in der unterlagerten Kommunikationsebene nicht vorhanden, so wird sie neu angelegt. Ist eine Kanalgruppe mit einem solchen Bezeichner vorhanden, so wird keine neue Kanalgruppe angelegt, sondern der bereits vorhandenen beigetreten. Forderung 2) entspricht dem Ansatz, dass Events und Remote Procedure Calls, die von Transducern erzeugt werden und auf einem Kanal kommuniziert werden, über die Kanalgruppe an die anderen Geräte (und damit deren SODAPOPD(AEMON)s und deren Transducer) abgebildet werden. Events sollen in der unterlagerten Kommunikation ebenfalls mit Events abgebildet werden, Remote Procedure Calls entsprechend mit RPCs. Sollte in einem Komponentenensemble nur ein SODAPOPD(AEMON) vorhanden sein, so ist der Ort der Ausführung von Konfliktlösungsstrategien im Falle konkurrierender Komponenten einfach zu bestimmen. Wie in Kapitel 5.1 definiert, werden diese Strategien im SODAPOPD(AEMON) ausgeführt. Sind nun mehrere SODAPOPD(AEMON)s in einem Ensemble beteiligt, so muss vor der Ausführung der Konfliktlösungsstrategie zuerst der ausführende SODAPOPD(AEMON) bestimmt werden. Forderung 3) hebt hervor, dass das Ergebnis der Ausführung der Konfliktlösungsstrategie auch den beteiligten anderen SODAPOPD(AEMON)s mitzuteilen ist. Dies ist nötig, damit diese die Ergebnisse an ihre beteiligten Transducer wiederum weitergeben können.

5.2.2 Der Abstract Connection Layer – ACL

Der *Abstract Connection Layer* als unterlagerte Kommunikationsschicht wird definiert, um den SODAPOPD(AEMON)s die Möglichkeit der Definition von Gruppen zu geben und die Kommunikation untereinander zu unterstützen. Die ACL stellt hierbei eine abstrakte Schicht der Konnektivität dar und bildet die Dienste für die Kommunikation der SODAPOPD(AEMON)s, deren Gruppenbildung und deren dynamisches Auffinden mit entsprechenden Methoden ab. Unterhalb der Ebene des Abstract Connection Layers wer-

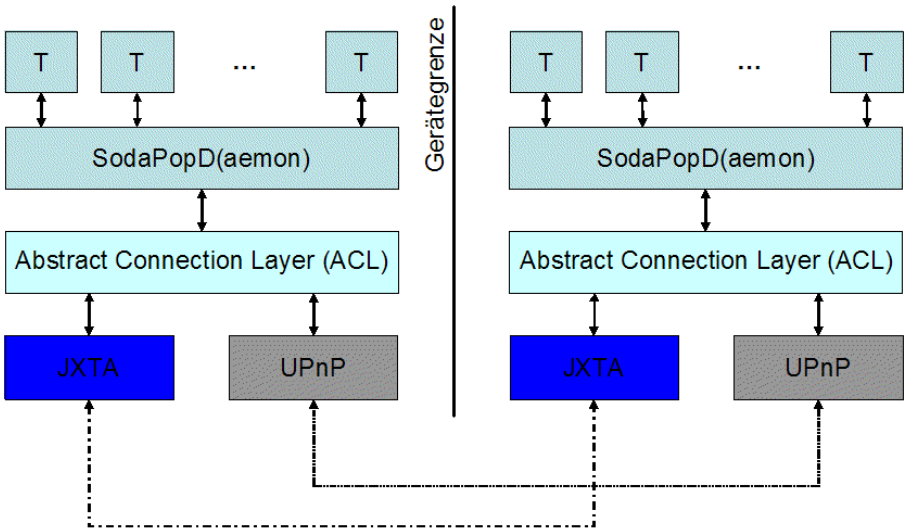


Abbildung 56: Der *Abstract Connection Layer* als abstrakte Kommunikationsschicht zur Ermöglichung der Peer-to-Peer-Kommunikation der SODAPOPD(AEMON)s mittels unterschiedlicher unterlagerte Kommunikationstechnologien.

den zwei unterschiedliche Technologien berücksichtigt [109][48]. Abbildung 56 illustriert, dass sowohl die Technologie JXTA [132] als auch Universal Plug and Play (UPnP) [206] als unterlagerte Kommunikationsinfrastruktur verwendet wird. Sowohl JXTA als auch UPnP benötigen pro physikalischem Gerät einen zusätzlichen Hintergrunddienst. Dieser stellt gewissermaßen die physikalische Infrastruktur zur Kommunikation zur Verfügung, während ein SODAPOPD(AEMON) die logische Infrastruktur auf einem Gerät darstellt. JXTA und/oder UPnP als ständigen Hintergrunddienst auf einem Gerät stellt keine Einschränkung der Dynamik eines Geräteensembles dar. Gemäß Abbildung 56 sind ein SODAPOPD(AEMON), die ACL und JXTA/UPnP als eine einzige Softwarelösung pro Gerät aufzufassen.

Die Klasse *Abstract Connection Layer* stellt einem SODAPOPD(AEMON) die folgenden Methoden zur Verfügung (siehe Abbildung 57):

ACL getInstance() Bei Aufruf dieser Methode erhält der SODAPOPD(AEMON) eine Instanz der lokalen ACL. Mit dieser können die weiteren Kommunikationsmethoden aufgerufen werden.

String registerDaemon(SodaPopD spd): eine SODAPOPD-Instanz meldet sich mit dieser Methode bei der lokalen ACL-Instanz an. Die unterlagerte Kommunikations-

[48]Tatsächlich wurde im Projekt DynAMITE die in diesem Kapitel definierte ACL sowohl auf Basis von JXTA als auch auf Basis von UPnP realisiert. Die JXTA-Version ist hierbei allen Projektpartnern zugänglich, während die UPnP-Version beim Projektpartner Loewe verblieb. Für die Implementation der JXTA-ACL trug maßgeblich in DynAMITE der Projektpartner European Media Laboratory die Verantwortung.

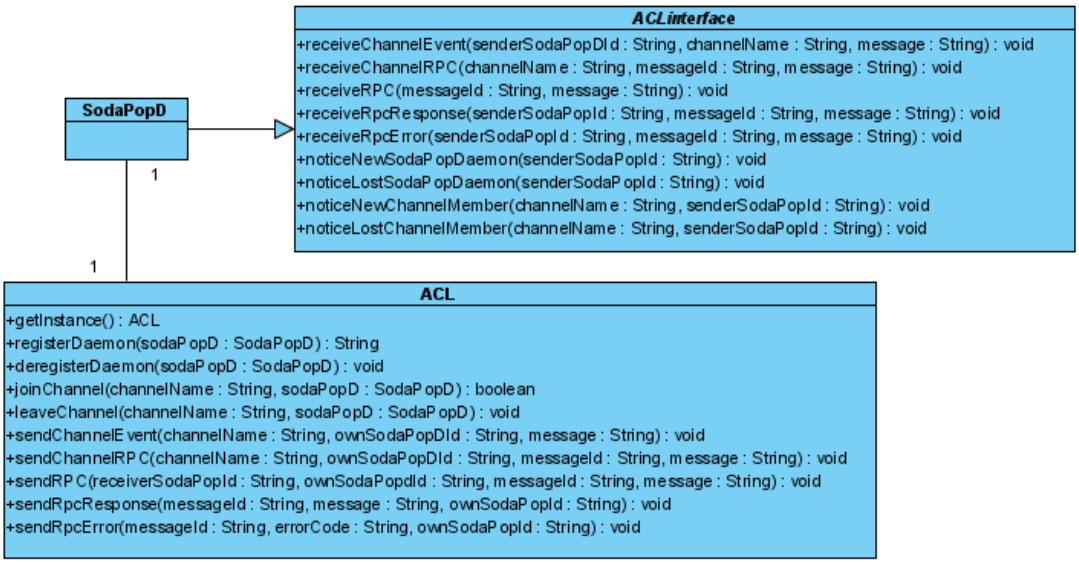


Abbildung 57: Die Erweiterung der SODAPOPD(AEMON)s um die Klassen ACL und ein ACL-Interface, die Methoden zur Definition von Kanalgruppen und zur Kommunikation innerhalb von Kanalgruppen zur Verfügung stellen.

struktur wird daraufhin den anderen vorhandenen SODAPOPD-Instanzen mittels der Methode *noticeNewSodaPopDaemon()* den Eintritt dieser SODAPOPD-Instanz mitteilen (siehe Abbildung 58 Operationen 1 bis 4). Der Rückgabewert entspricht einem lokal eindeutigen Geräteidentifier, der für die weitere eigene Identifizierung verwendet wird.

void deregisterDaemon (SodaPopD spd): nach dem Aufruf dieser Methode ist der SODAPOPD(AEMON) von der ACL abgemeldet. Sein bisheriger Bezeichner zur Identifizierung ist damit ungültig. Bei allen anderen SODAPOPD(AEMON)s wird gleichzeitig die Methode *noticeLostSodaPopDaemon()* aufgerufen. Wird diese Methode aufgerufen, obwohl der SODAPOPD(AEMON) noch bei Kanalgruppen angemeldet ist, wird deren Mitgliedschaft zuerst beendet und dies den beteiligten anderen Gruppenmitgliedern bekannt gemacht (siehe Abbildung 58 Operationen 15 - 17). Erst dann wird über das generelle Verlassen informiert (siehe Abbildung 58 Operation 17).

boolean joinChannel (String channelName, SodaPopD spd): mit dieser Methode meldet sich der SODAPOPD(AEMON) an einem Kanal an. Existiert eine Gruppe mit dem Bezeichner *channelName* noch nicht, wird sie automatisch angelegt. Existiert diese Gruppe bereits, wird der SODAPOPD(AEMON) über die Bezeichner der Gruppenmitglieder informiert. Für jedes vorhandene Gruppenmitglied wird einmal die Methode *noticeNewChannelMember()* aufgerufen (siehe Abbildung 58 Operatio-

nen 5 bis 12).

void leaveChannel (String channelName, SodaPopD spd): mit dieser Methode meldet sich der SODAPOPD(AEMON) von einer Kanalgruppe ab. Jedes zurückbleibende Gruppenmitglied wird mittels der Methode *noticeLostChannelMember()* darüber informiert (siehe Abbildung 58 Operationen 13 bis 14).

void sendChannelEvent (String channelName, String ownSodaPopID, String message): mit dieser Methode können SODAPOPD(AEMON)s Events an einen Kanal versenden. Jeder andere SODAPOPD(AEMON), der Mitglied dieser Kanalgruppe ist, erhält diesen Event. Bei ihm wird automatisch durch das ACL-Interface die Methode *receiveChannelEvent* angesprochen (siehe Abbildung 59). Um Verwechslungen bzw. Irritationen von diesen unterlagerten Events mit den semantischen Events, wie sie in SODAPOP verwendet werden, zu vermeiden, werden die Events auf ACL-Ebene im Folgenden immer als ACL-Events bezeichnet.

void sendChannelRPC (String channelName, String sodaPopID, String messageID, String message): mit dieser Methode können SODAPOPD-Instanzen einen Remote Procedure Call an eine Kanalgruppe versenden. Jeder andere SODAPOPD in dieser Kanalgruppe erhält über die Methode *receiveChannelRPC* diesen RPC. Von jedem der Empfänger des RPCs erhält der Sender eine Antwort über die Methode *receiveRpcResponse* (siehe Abbildung 60). Die hier verwendete *messageID* ist vom SODAPOPD(AEMON) frei wählbar. Sie findet sich in der Antwort wieder. Um eine Verwechslung mit den in SODAPOP definierten RPCs auszuschliessen, werden sie im Folgenden immer ACL-RPCs genannt.

void sendRPC (String receiverSodaPopID, String ownSodaPopID, String messageID, String message): mit dieser Methode können SODAPOPD(AEMON)s einen ACL-RPC an einen bestimmen SODAPOPD(AEMON) versenden. Er erhält vom adressierten SODAPOPD(AEMON) eine Antwort wie in Abbildung 61 dargestellt.

void sendRpcResponse (String messageID, String message, String ownSodaPopID): mit dieser Methode beantwortet ein SODAPOPD(AEMON) einen ACL-RPC.

void sendRpcError (String messageID, String errorCode, String ownSodaPopID): mit dieser Methode versendet ein SODAPOPD(AEMON) eine Fehlernachricht als Antwort auf einen empfangenen ACL-RPC. Die Fehlernachricht enthält einen Error-Code.

Zum Nachrichtenempfang aus der unterlagerten Schicht des *Abstract Connection Layers* muss eine SODAPOPD-Instanz die folgenden Methoden implementieren (siehe Abbildung 57):

void receiveChannelEvent (String senderSodaPopID, String channelName, String message): mittels dieser Methode empfangen SODAPOPD-Instanzen ACL-Events (siehe Abbildung 59).

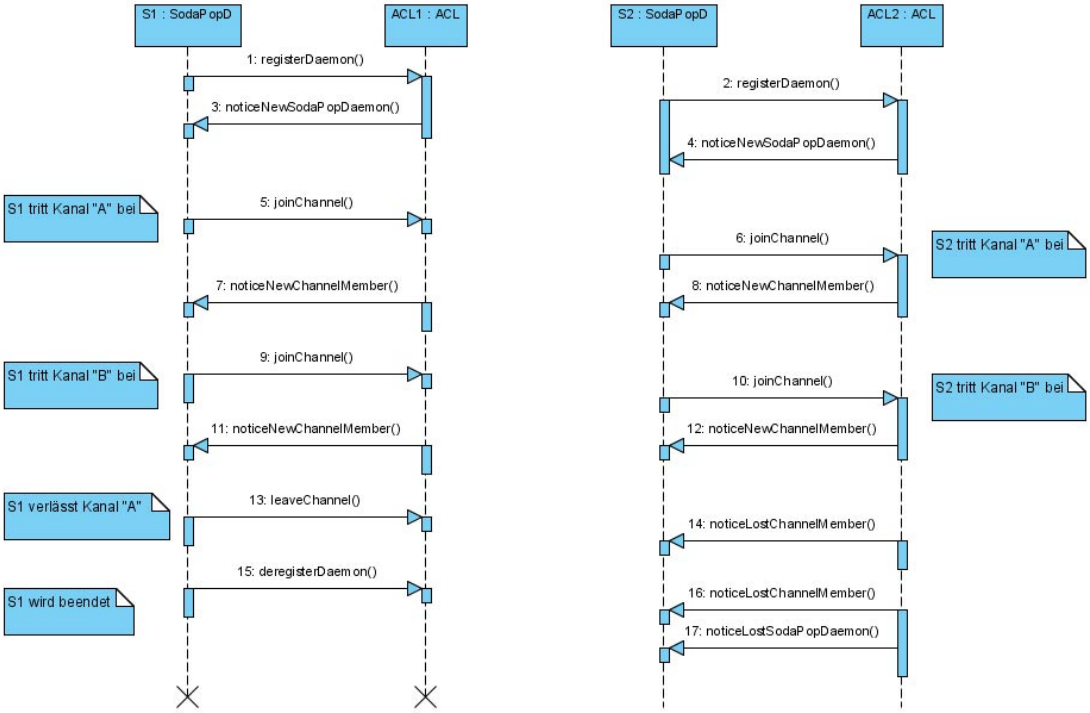


Abbildung 58: Definition von Kanalgruppen, Beitritt und Verlassen von Kanalgruppen am Beispiel zweier SODAPOPD-Instanzen.

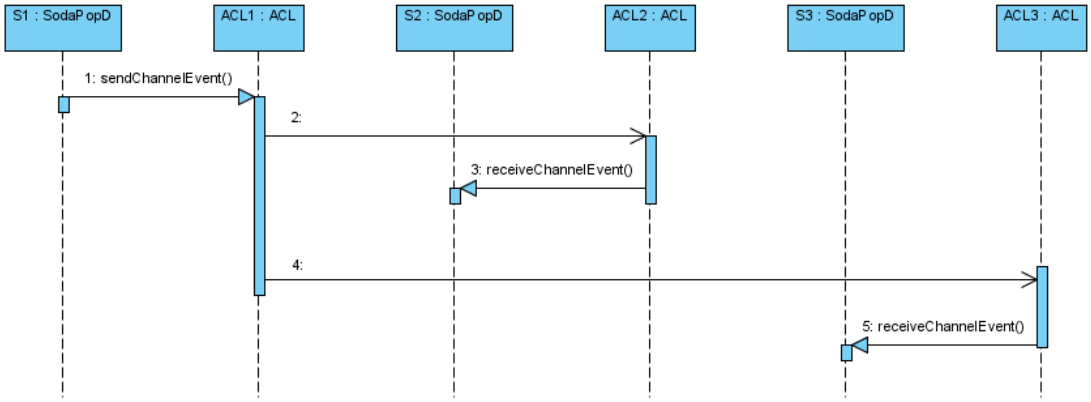


Abbildung 59: Verschickung von ACL-Events durch einen SODAPOPD(AEMON) in eine Kanalgruppe. Die beiden anderen in der Gruppe beteiligten SODAPOPD-Instanzen erhalten *zeitgleich* dieses ACL-Event mittels des unterlagerten Abstract Connection Layers.

void receiveChannelRPC (String channelName, String messageID, String message): über diese Methode empfangen SODAPOPD-Instanzen ACL-RPCs, die an eine Kanalgruppe gerichtet sind (siehe Abbildung 60). Ein solcher Kanal-ACL-RPC enthält

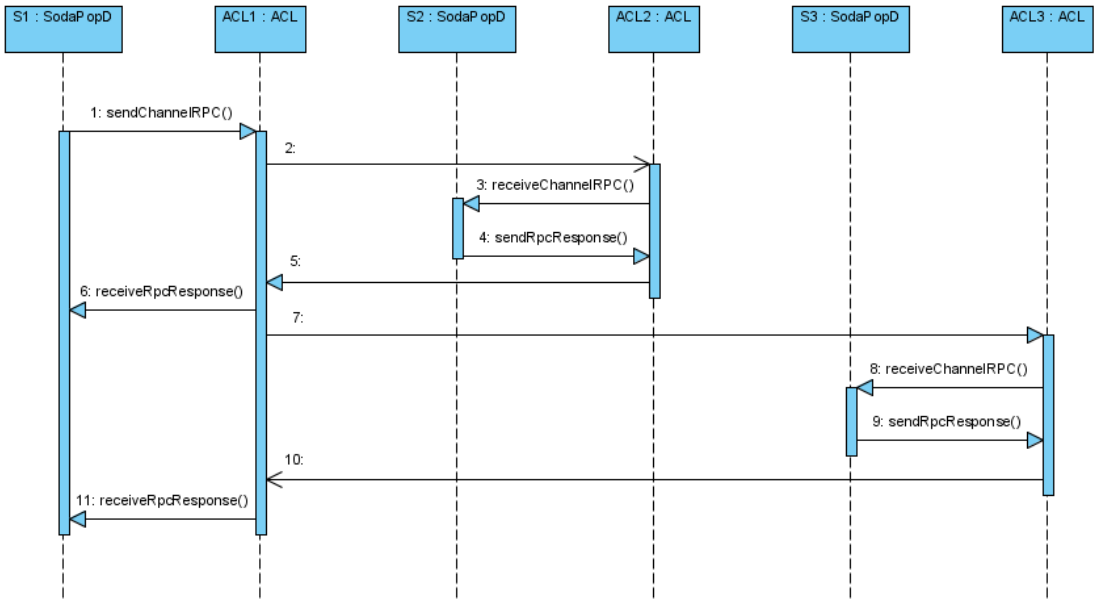


Abbildung 60: Versenden eines ACL-RPCs von einem SODAPOPD(AEMON) in eine Kanalgruppe. Von jedem Kanalmitglied gibt es eine eigenständige Antwort.

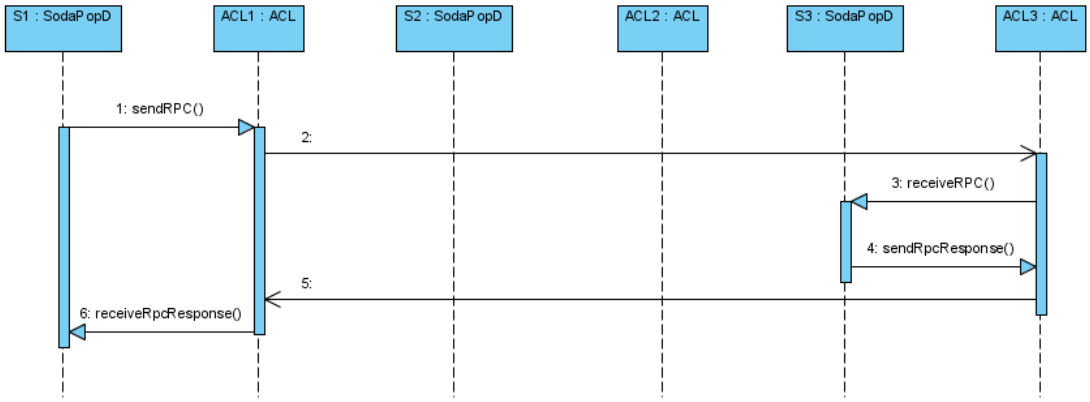


Abbildung 61: Versenden eines ACL-RPCs von einem SODAPOPD(AEMON) an eine bestimmte SODAPOPD-Instanz.

keine Angaben über den Absender. Stattdessen enthält er eine Nachrichten-ID, mittels der geantwortet werden muss. Jede SODAPOPD-Instanz antwortet auf einen ACL-RPC mit exakt einer Antwort. Jeder Absender eines ACL-RPCs erhält von jedem anderen Kanalgruppen-Mitglied exakt eine Antwort.

void receiveRPC (String messageID, String message): über diese Methode empfangen SODAPOP-Instanzen einen ACL-RPC, der direkt an sie gerichtet ist. Er wird

ebenso wie Kanal-ACL-RPCs direkt über die Nachrichten-ID beantwortet (siehe Abbildung 61).

void receiveRpcResponse (String senderSodaPopID, String messageID, String message): über diese Methode empfängt eine SODAPOPD-Instanz eine Nachricht auf einen von ihm versendeten ACL-RPC. Die messageID dieser Antwort entspricht der beim ACL-RPC vergebenen messageID.

void receiveRpcError (String sender SodaPopID, String messageID, String errorCode): über diese Methode empfängt die SODAPOPD-Instanz eine Fehlernachricht auf einen von dieser initiierten ACL-RPC.

noticeNewSodaPopDaemon (String senderSodaPopID): über diese Methode wird eine SODAPOPD-Instanz über das Vorhandensein anderer SODAPOPD-Instanzen informiert. Die *senderSodaPopID* ist eindeutig und kann im Folgenden zur direkten Adressierung von ACL-RPCs verwendet werden.

noticeLostSodaPopDaemon (String senderSodaPopID): über diese Methode wird eine SODAPOPD-Instanz über das globale Abmelden einer anderen SODAPOPD-Instanz unterrichtet (siehe Abbildung 58).

noticeNewChannelMember (String channelName, String senderSodaPopID): über diese Methode wird eine SODAPOPD-Instanz über ein neues Kanalgruppen-Mitglied informiert.

noticeLostChannelMember (String channelName, String senderSodaPopID): über diese Methode wird eine SODAPOPD-Instanz darüber informiert, dass eine andere SODAPOPD-Instanz eine Kanalgruppe verlassen hat.

Zu den Sequenzdiagrammen: Abbildung 58 erläutert die prinzipiellen Vorgänge beim Konnektieren von SODAPOPD(AEMON)s an der *Abstract-Connection-Layer*-Schicht und dem Betreten bzw. Verlassen von Kanalgruppen. Nachdem die SODAPOPD-Instanzen S1 und S2 die ACL instanziiert haben, werden sie gegenseitig über ihre Existenz informiert (Operation 3 und 4). Dann tritt S1 einer Kanalgruppe mit dem Bezeichner *A* bei. Sobald auch S2 dieser Gruppe beigetreten ist (Operation 6) werden sowohl S1 als auch S2 über die gegenseitige Mitgliedschaft in dieser Gruppe informiert (Operationen 7 und 8). Das Gleiche geschieht bei der Definition und Beitritt einer Kanalgruppe mit dem Bezeichner *B* (siehe Operationen 9 bis 12). Nun verlässt die SODAPOPD-Instanz S1 die Kanalgruppe *A* wieder (Operation 13) und S2 wird umgehend darüber informiert (Operation 14). Ohne vorher die Mitgliedschaft in Kanalgruppe *B* zu beenden, verlässt S1 das gesamte Geräteensemble (Operation 15). Die ACL beendet die Mitgliedschaften von S1 korrekt: zuerst wird S2 über den Weggang von S1 aus der Kanalgruppe *B* informiert (Operation 16), dann über den Wegfall aus dem gesamten Geräteensemble (Operation 17). Abbildung 59 illustriert das Versenden eines ACL-Events von einer SODAPOPD-Instanz an die Mitglieder einer Kanalgruppe. Nachdem die SODAPOPD-Instanz S1 das Event mittels

dem Methodenaufruf *sendChannelEvent()* versendet hat, gelangt dies über die beteiligten ACL-Instanzen an die anderen SODAPOPD-Instanzen der Kanalgruppe. Hier wird die Methode *receiveChannelEvent()* der beteiligten SODAPOPD-Instanzen aufgerufen. Die Operationen 2 und 4 in Abbildung 59 laufen zeitgleich ab und sind nur aus Gründen der Übersichtlichkeit zeitlich nacheinander dargestellt. In Abbildung 60 ist gezeigt, wie eine SODAPOPD-Instanz einen ACL-RPC an die anderen Mitglieder einer Kanalgruppe versendet. Mittels Aufruf der Methode *sendChannelRPC()* versendet die ACL den Remote Procedure Call an alle Kanalgruppen-Mitglieder, deren *receiveChannelRPC()*-Methode aufgerufen wird. Analog zu dem vorher Definierten sind hier die Operationen 2 und 7 als zeitgleich anzusehen und sind nur aus Gründen besserer Übersichtlichkeit zeitlich nacheinander dargestellt. Nach Bearbeitung des ACL-RPCs verschicken die beteiligten SODAPOPD-Instanzen S2 und S3 mittels Aufruf der Methode *sendRpcResponse()* die Antworten (Operation 4 und 9) die jeweils für den Aufruf der Methode *receiveRpcResponse()* des initiierenden SODAPOPD(AEMON) verantwortlich sind. Für jeden ACL-RPC gehen exakt so viele Antworten ein, wie andere Kanalgruppen-Mitglieder vorhanden sind. Anders verhält es sich bei Aufruf der Methode *sendRPC()*. Abbildung 61 illustriert das Verhalten der ACL hier. Nur die adressierte SODAPOPD-Instanz erhält den ACL-RPC (Operation 3) und antwortet darauf (Operation 4 und 6).

5.2.3 Behandlung von Events

Events sind von Transducern versendete Ereignisse. Ein Transducer kann ein Event in einen Kanal versenden, an dem er entsprechend als Event-Quelle konnektiert ist. Speziell erwartet ein Transducer auf das Versenden eines Events keine Antwort. Um ein Event zuzuteilen, werden die UtilityValue-Funktionen aller Transducer evaluiert, die an dem entsprechenden Kanal als Event-Senken registriert sind. Mittels der für den entsprechenden Kanal definierten Konfliktlösungsstrategie werden diejenigen Transducer ermittelt, die das Event letztlich verbrauchen dürfen. Werden keine passenden Transducer ermittelt, verliert das Event seine Gültigkeit (siehe Kapitel 5.1). Sind in einem dynamischen System mehrere SODAPOPD-Instanzen beteiligt, müssen die folgenden Einzelschritte ausgeführt werden:

1. Ein Transducer sendet ein Event in einen Kanal
2. Die zugehörige SODAPOPD-Instanz sendet dieses Event in die entsprechende Kanalgruppe, um die beteiligten anderen SODAPOPD-Instanzen aufzufordern die eigenen Transducer, die als Event-Senken (an dem entsprechenden Kanal) registriert sind, zu evaluieren
3. Die Ergebnisse der Evaluierung werden den Gruppenmitgliedern mitgeteilt. Dann beginnt ein SODAPOPD(AEMON) mit der Ausführung der kanal-eigenen Konfliktlösungsstrategie
4. Nach Beendigung der Konfliktlösungsstrategie werden diejenigen SODAPOPD-Instanzen informiert, die der Host derjenigen Transducer sind, die das Event verbrauchen dürfen.

Abbildung 62, Abbildung 63 und Abbildung 64 illustrieren die genaue Vorgehensweise. Hierbei sind in Abbildung 62 vier SODAPOPD(AEMON)s abgebildet, in Abbildung 63 und Abbildung 64 nur deren drei. Dies dient der erhöhten Übersichtlichkeit und stellt keine Einschränkung der Szenarios dar. Abbildung 63 und Abbildung 64 bilden hinzufügend die auf den unterschiedlichen Geräten befindlichen Transducer ab, da die Kommunikation mit diesen für die Ablauflogik von besonderer Wichtigkeit ist. Zur besseren Verständlichkeit sind in Abbildung 63 und Abbildung 64 nur Transducer abgebildet, die am selben Kanal konnektiert sind. Eventuell andere Transducer (an Kanälen mit entsprechend anderen Bezeichnern konnektiert) haben keinen Einfluss auf das hier geschilderte Gesamtgeschehen. Ein Transducer (siehe Abbildung 63 a)) sendet ein Event an seine SODAPOPD-Instanz:

```
(event
     :channel Kanalname
     :sender Transducer-ID
     :content ( event ))
```

Nach Erhalt dieses Events sendet der empfangende SODAPOPD(AEMON) einen ACL-Event in die entsprechende Kanalgruppe (siehe Abbildung 62 Operationen 1 bis 7 und Abbildung 63 b)) mittels der Methode *sendChannelEvent()*:

```
sendChannelEvent(Kanalname, eigeneSodaPopID, message)
message = (proposeEvent
       :ref-no RW
       :content ( event ))
```

Den beteiligten SODAPOPD-Instanzen (im Beispiel Abbildung 63 b) somit die Instanzen S1, S2 und S3) sind somit der Kanal und eine Referenznummer bekannt. Über die Empfangsmethode

```
receiveChannelEvent(senderSodaPopID, Kanalname, message)
message = (proposeEvent
       :ref-no RW
       :content ( event ))
```

ist den empfangenden SODAPOPD-Instanzen (siehe Abbildung 62 Operation 3, 5 und 7) zusätzlich die ID der absendenden SODAPOPD-Instanz bekannt. Die Nachricht selbst kodiert in ihrer Semantik *proposeEvent*, dass ein Event geschehen ist und zugeteilt werden muss.

Die folgenden Schritte betreffen wieder die einzelnen SODAPOPD-Instanzen (siehe Abbildung 63 c) und d)). Es werden die eigenen Transducer evaluiert, die am entsprechenden Kanal als Event-Senke registriert sind (dies geschieht exakt wie in Kapitel 5.1.4 beschrieben). Jede SODAPOPD-Instanz sendet somit an ihre Transducer die Nachricht:

```
(proposeEvent
     :channel Kanalname
     :ref-no RW1
     :content ( event ))
```

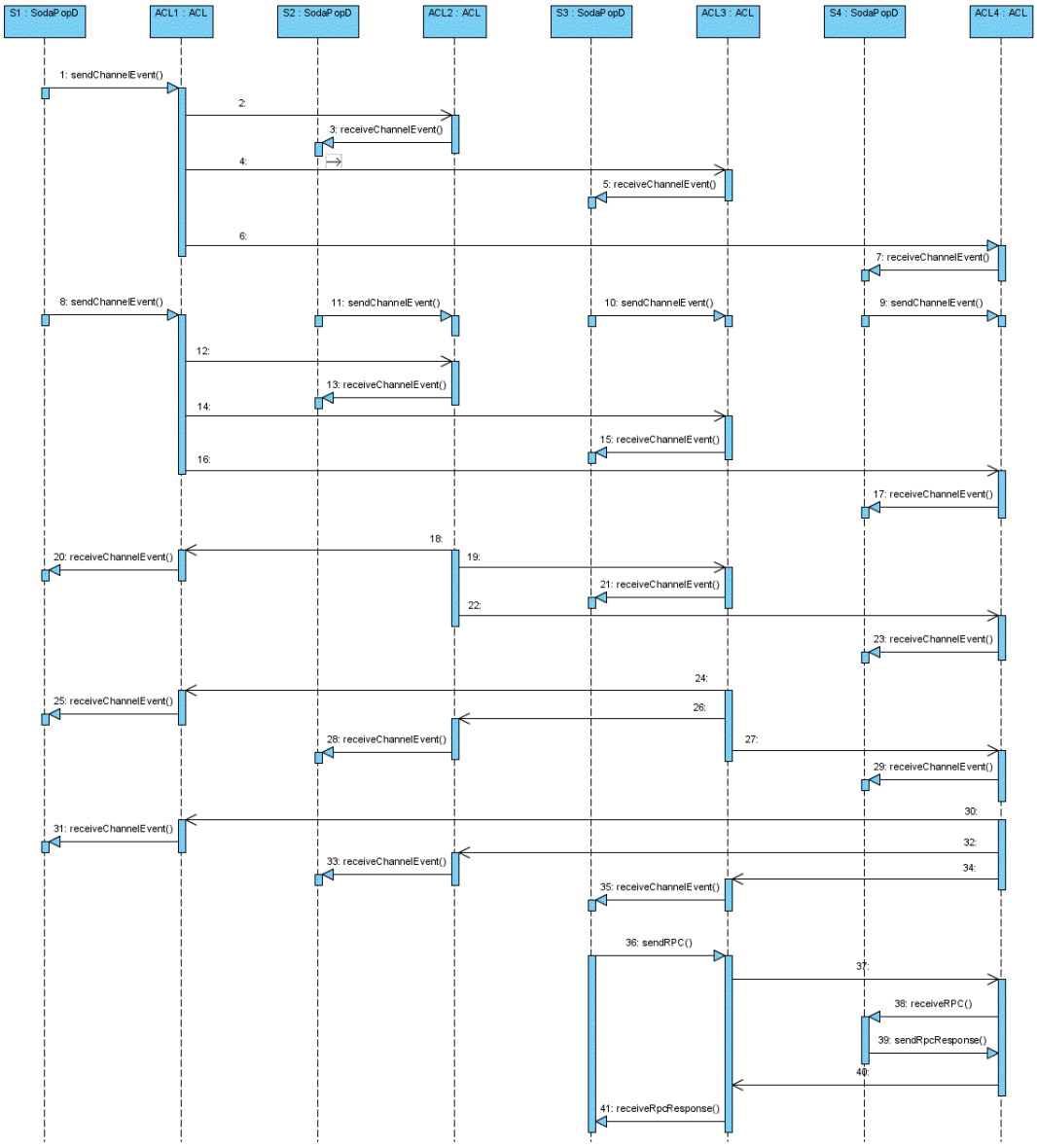
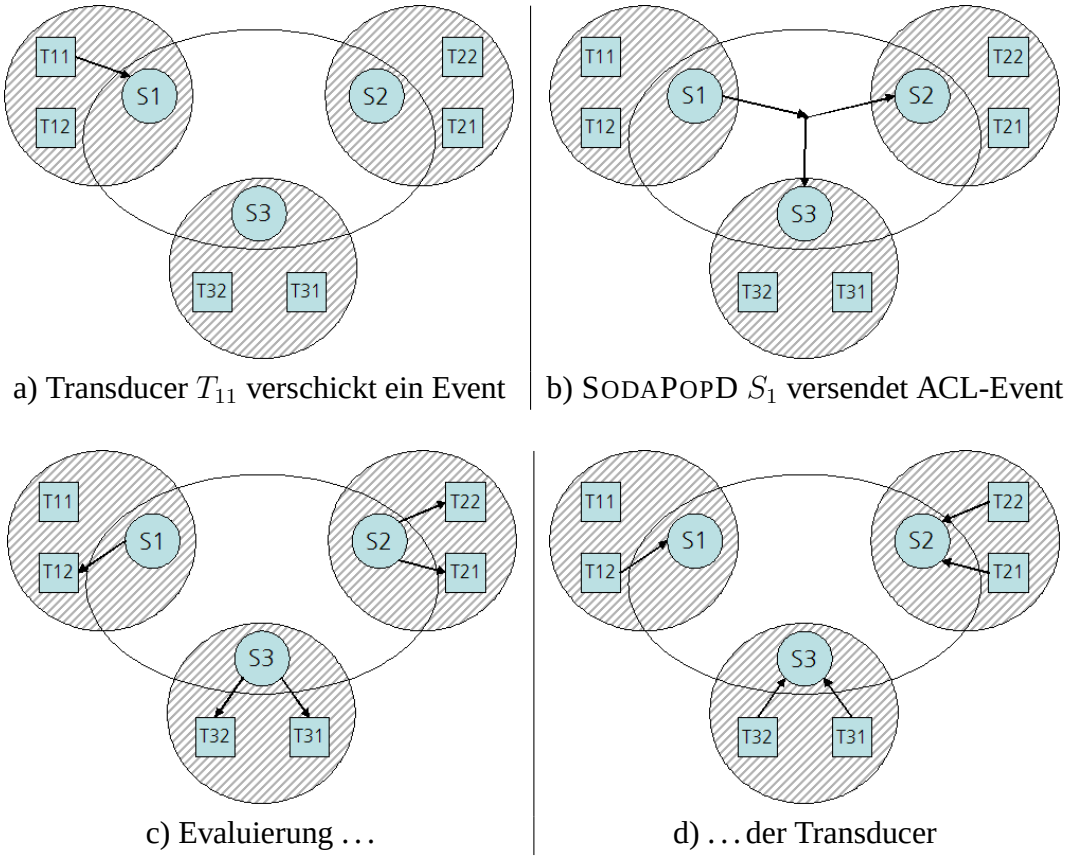


Abbildung 62: Sequenzdiagramm: Verarbeitung eines Events, welches von einem Transducer der SODAPOPD-Instanz S1 initiiert wurde. Die Verarbeitung geschieht innerhalb der Kanalgruppe mittels ACL-Events (Operation 1 - 35) und ACL-Remote Procedure Calls (Operation 36 - 41). Im Beispiel bilden vier SODAPOPD-Instanzen – in Vertretung ihrer Transducer – eine Kanalgruppe.

Anmerkung: Die Nachrichten-Referenznummer *RW1* kann innerhalb der verschiedenen SODAPOPD-Instanzen unterschiedlich sein. Es handelt sich um eine interne Nachrichtennummer, die nicht nach außen, d.h. in die Kanalgruppe gelangt. Im Falle, dass die Auswertung der UtilityValue-Funktion positiv verlaufen ist, versendet der Transducer ent-



a) Transducer T_{11} verschickt ein Event

b) SODAPOPD S_1 versendet ACL-Event

c) Evaluierung ...

d) ... der Transducer

Abbildung 63: Illustration der Behandlung eines Events innerhalb einer Kanalgruppe (Teil 1).

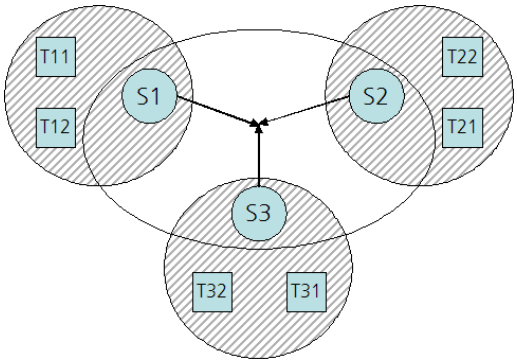
sprechend die folgende Nachricht an seine SODAPOPD-Instanz:

```
(acceptEvent
     :sender Transducer-ID
     :channel Kanalname
     :ref-no RW1
     :content (utilityValue))
```

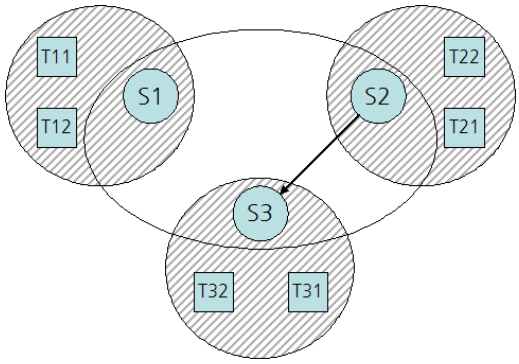
Im negativen Fall lautet die Nachricht wie folgt:

```
(rejectEvent
     :sender Transducer-ID
     :channel Kanalname
     :ref-no RW1
     :content (false))
```

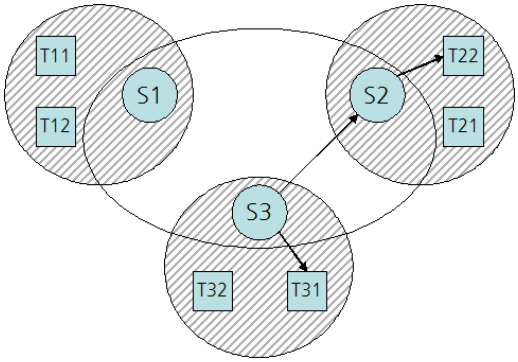
Im folgenden Schritt (siehe Abbildung 64 e) und Abbildung 62 Operation 8 - 35) teilen die verschiedenen SODAPOPD-Instanzen die gesammelten UtilityValues ihrer Transducer innerhalb der Kanalgruppe mit. Jeder SODAPOPD(AEMON) versendet somit ein ACL-Event in die entsprechende Kanalgruppe. Das Sequenzdiagramm Abbildung 62 macht hier die Menge der Nachrichten in der unterlagerten ACL-Schicht deutlich.



e) Mitteilung aller UtilityValues mittels ACL-Events



f) Übermittlung der Ergebnisse ...



g) ... und Zuteilung der Events

Abbildung 64: Illustration der Behandlung eines Events innerhalb einer Kanalgruppe (Teil 2). Für die Kommunikation innerhalb der Kanalgruppe werden ACL-Events und ACL-RPCs verwendet.

Jede SODAPOPD-Instanz erhält von den anderen Instanzen jeweils ein ACL-Event. Für die SODAPOPD-Instanz S1 würde hier gelten:

```
sendChannelEvent(Kanalname, S1, message)
message = (utilityValues
        :ref-no RW
        :content ((performance S1P),
        (appendix S1Z),
        (T12 (UV-T12))))
```

Entsprechend für die SODAPOPD-Instanz S2:

```
sendChannelEvent(Kanalname, S2, message)
message = (utilityValues
        :ref-no RW
        :content ((performance S2P),
        (appendix S2Z),
        (T21 (UV-T21)), (T22 (UV-T22))))
```

Und für die SODAPOPD-Instanz S3:

```
sendChannelEvent(Kanalname, S3, message)
message = (utilityValues
       :ref-no RW
       :content ((performance S3P),
       (appendix S3Z),
       (T31 (UV-T31)), (T32 (UV-T32))))
```

Jeder an der entsprechenden Kanalgruppe beteiligte SODAPOPD(AEMON) ist zu diesem Zeitpunkt in Kenntnis (bei Verwendung der Referenznummer *RW*):

- über das ursprünglichen Event
- über die UtilityValues aller potentiellen Empfangs-Transducer, deren IDs und zu welchem SODAPOPD sie zugehörig sind.

Um diejenige SODAPOPD-Instanz zu ermitteln, welche mit diesen Informationen die Konfliktlösungsstrategie ausführen muss, hat jede SODAPOPD-Instanz Information über die eigene momentane Leistungsfähigkeit ($0 \leq performance \leq 1$) und eine zusätzliche Zahl ($0 \leq appendix \leq 1$) der Nachricht mitgegeben. Diejenige SODAPOPD-Instanz, die die höchste Zahl *performance* abgegeben hat, beginnt mit der Ausführung der Konfliktlösungsstrategie. Sollten zwei SODAPOPD-Instanzen dieselben Zahlen für die *performance* gegeben haben, entscheidet die höhere Zusatzzahl. Da alle Zahlen transparent den beteiligten SODAPOPD-Instanzen bekannt sind, ist dieses Verfahren eindeutig. Alle beteiligten SODAPOPD-Instanzen wissen um ihre Position in diesem Leader-Election-Verfahren. Schon 1982 schlug Garcia-Molina [89] einen Algorithmus vor, der aus einem Satz gleichberechtigter Knoten, die eine Gruppe darstellen, einen zeitlich ausgezeichneten Knoten auswählt. Später wurden diese Algorithmen ausgeweitet für den Einsatz in Breitband-Netzwerken [36] oder auch anonymen Ringen [221]. Besonders in der Arbeit von Feige [70] findet sich eine Analyse von Leader-Election-Protokollen, die in der Lage sind aus mehreren geeigneten Kandidaten ein ausgezeichnetes Gruppenmitglied zu finden. Speziell in der Anwendung von weit verstreuten Sensornetzwerken kommen Leader-Election-Verfahren zum Einsatz. Hier werden Gemeinschaftsaufgaben innerhalb einer Sensorengruppe von einzelnen Sensoren übernommen, um die Gesamtlebensdauer (vor allem in Hinsicht des gemeinschaftlichen Energieverbrauches) zu maximieren. Das Protokoll LEACH (für Low Energy Adaptive Clustering Hierarchy) [100] sieht hierbei vor, dass alle Sensoren in regelmäßigen Abständen mittels eines Zufallsalgorithmus bestimmen, ob sie selbst ein sog. Cluster-Head sind oder nicht. Hat sich ein Sensor somit selbst als Cluster-Head bestimmt, so sendet dieser ein Advertisement zu allen anderen Knoten, um sich bei diesen für den nächsten Zeitraum als ausgezeichneten Knoten zu bewerben. Knoten die kein Cluster-Head sind, können sich selbst aus der Menge der angebotenen Cluster-Heads einen aussuchen, der dann für die folgende Periode für sie bestimmte Aufgaben übernimmt. Dies wird mittels einer Bestätigungsnachricht vorgenommen. Für die Auswahl derjenigen SODAPOPD-Instanz ist die hier definierte Vorgehensweise – die in ihren Prinzipien dem LEACH-Ansatz ähnelt – ausreichend. Es wird mit geringen Mitteln (vor allem ohne zusätzlichen Nachrichtenaufwand oder gar Verhandlungen) ein zeitlich ausgezeichnetes Gruppenmitglied gefunden.

Nach Ausführung der Konfliktlösungsstrategie werden die Ergebnisse mittels gezielten ACL-RPCs mitgeteilt. In Abbildung 62 wurde die Konfliktlösungsstrategie von der SODAPOPD-Instanz S3 ausgeführt, die als Ergebnis hatte, dass ein Transducer mit dem Host S4 das Event verarbeiten soll. Dafür verschickt S3 an S4 einen gerichteten ACL-RPC (Operation 37), deren Erhalt S4 bestätigt (Operation 39 und 41). Abbildung 64 führt das begonnene Beispiel fort. Nachdem die SODAPOPD-Instanz S2 die entsprechende kanal-eigene Konfliktlösungsstrategie ausgeführt hat, seien die Transducer T_{22} und T_{31} als Empfänger des initialen Events hervorgegangen. Der strategie-ausführende SODAPOPD(AEMON) hat somit einen eigenen Transducer und einen fremden Transducer als Empfänger ermittelt. Ebenso sei mit diesem Beispiel hervorgehoben, dass nicht derjenige SODAPOPD(AEMON) die Konfliktlösungsstrategie ausführen *muss*, der der Host des Event-emittierenden Transducers ist. Vielmehr wird die Regel gelten, dass ein Gerät auf welchem ein Transducer läuft, der eine Nachricht verarbeiten möchte eher in der Lage sein sollte auch eine komplexe Konfliktlösungsstrategie auszuführen. Würde diese Regel nicht gelten, dann müssten simple Event-Emitter (wie z. B. Schalter, Mikrophone, RFID-Tags) Ressourcen bereithalten, um Strategien auszuführen. Dies würde eine zu große Einschränkung bedeuten. Im Beispiel wird die SODAPOPD-Instanz S2 der SODAPOPD-Instanz S3 den folgenden ACL-RPC schicken (siehe Abbildung 64 f)):

```
sendRPC(S3, S2, RW2, message)
message = (event
       :channel A
       :receiver T31
       :ref-no RW
       :content ( event ))
```

S3 empfängt die Nachricht in der Methode *receiveRPC()*:

```
receiveRPC(RW2, message)
message = (event
       :channel A
       :receiver T31
       :ref-no RW
       :content ( event ))
```

Die Referenznummer innerhalb der Nachricht entspricht der anfänglichen Referenznummer. Hierdurch ist es dem empfangenden SODAPOPD(AEMON) bis zuletzt möglich das Geschehen nachzuvollziehen. Mit den Angaben über die ID des Transducers (*receiver*) und den Kanal (*channel*) kann er seinem Transducer das Event zuteilen (siehe Abbildung 64 g)). Dies geschieht mittels der (internen) Nachricht:

```
(event
     :channel Kanalname
     :content ( event ))
```

Gleichzeitig antwortet die SODAPOPD-Instanz S3 auf den ACL-Remote Procedure Call:

```
sendRpcResponse(RW2, message)
message = (eventConsumed
       :ref-no RW)
```

Zusätzlich sendet die SODAPOPD-Instanz S2 an seinen Transducer T_{22} die folgende Nachricht:

```
(event
     :channel Kanalname
     :content ( event ))
```

Die Versendung eines von einem Transducer ausgesendeten Events ist somit unter Beteiligung aller in einer Kanalgruppe beteiligten SODAPOPD(AEMON)s abgeschlossen.

5.2.4 Behandlung von Remote Procedure Calls

Remote Procedure Calls werden von Transducern in der Absicht ausgesendet, eine Antwort auf die darin formulierte Anfrage oder den darin befindlichen Auftrag zu erhalten. Es ist Aufgabe der Konfliktlösungsstrategie des Kanals, an den dieser RPC gesendet wurde, geeignete Bearbeiter für den Remote Procedure Call zu finden, die Antworten einzuholen und dem initiierendem Transducer wieder zukommen zu lassen. Um einen RPC zuzuteilen, werden die UtilityValue-Funktionen aller Transducer evaluiert, die als RPC-Senken an dem entsprechenden Kanal konnektiert sind. Werden keine geeigneten Transducer ermittelt, sendet der Kanal eine entsprechende Mitteilung an den Transducer, der den RPC initiiert hat. Auf einen Remote Procedure Call erhält ein Transducer somit immer eine Antwort. Kapitel 5.1 beschreibt den Vorgang Remote Procedure Calls zu verarbeiten in Falle eines singulären SODAPOPD(AEMON)s. Sind in einem dynamischen System mehrere SODAPOPD-Instanzen beteiligt, so sind die folgenden Einzelschritte auszuführen:

1. Ein Transducer sendet ein Remote Procedure Call in einen Kanal

2. Die zugehörige SODAPOPD-Instanz sendet diesen RPC in die entsprechende Kanalgruppe, um die beteiligten anderen SODAPOPD-Instanzen aufzufordern die eigenen Transducer, die als RPC-Senken (an dem entsprechenden Kanal) registriert sind, zu evaluieren

3. Die Ergebnisse der Evaluierung werden an den initiierenden SODAPOPD(AEMON) gesendet. Dieser beginnt mit der Ausführung der kanal-eigenen Konfliktlösungsstrategie

4. Nach Ermittlung der geeigneten Transducer sendet der SODAPOPD(AEMON) einen gerichteten Remote Procedure Call an die entsprechenden SODAPOPD-Instanzen, die die ermittelten Transducer hosten

5. Die SODAPOPD-Instanzen reichen den RPC an ihre Transducer weiter, empfangen deren Antwort und senden sie über die Kanalgruppe an den initiierenden SODAPOPD(AEMON) zurück

6. Dieser sammelt alle Antworten und gibt dem initiierenden Transducer gemäß der Kanalstrategie eine Antwort auf seinen Remote Procedure Call.

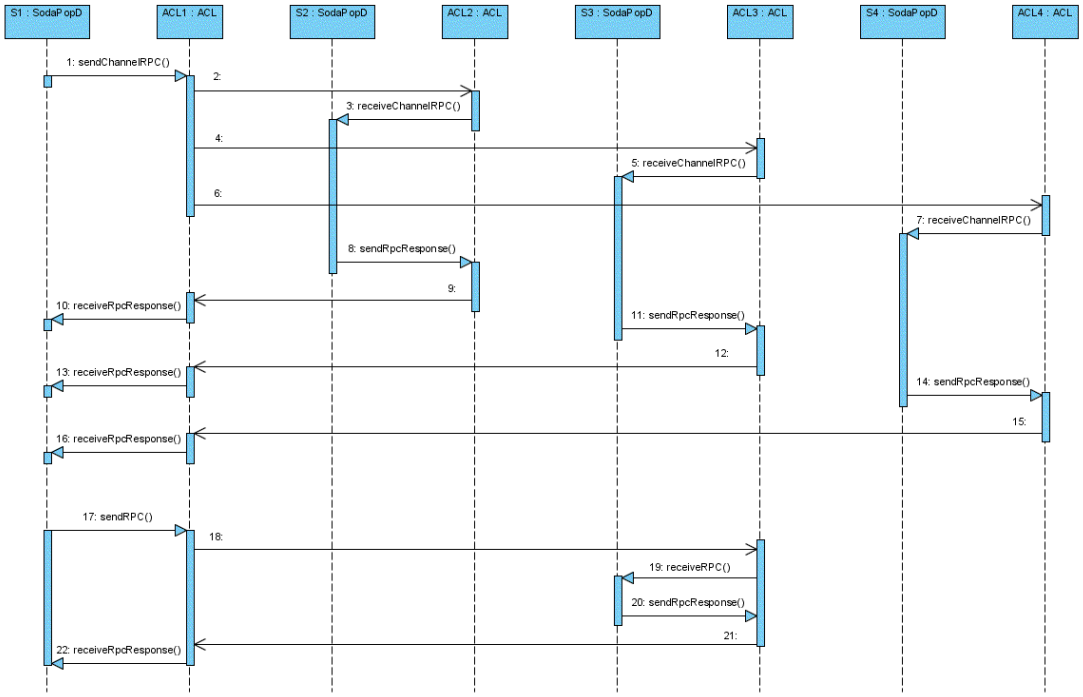
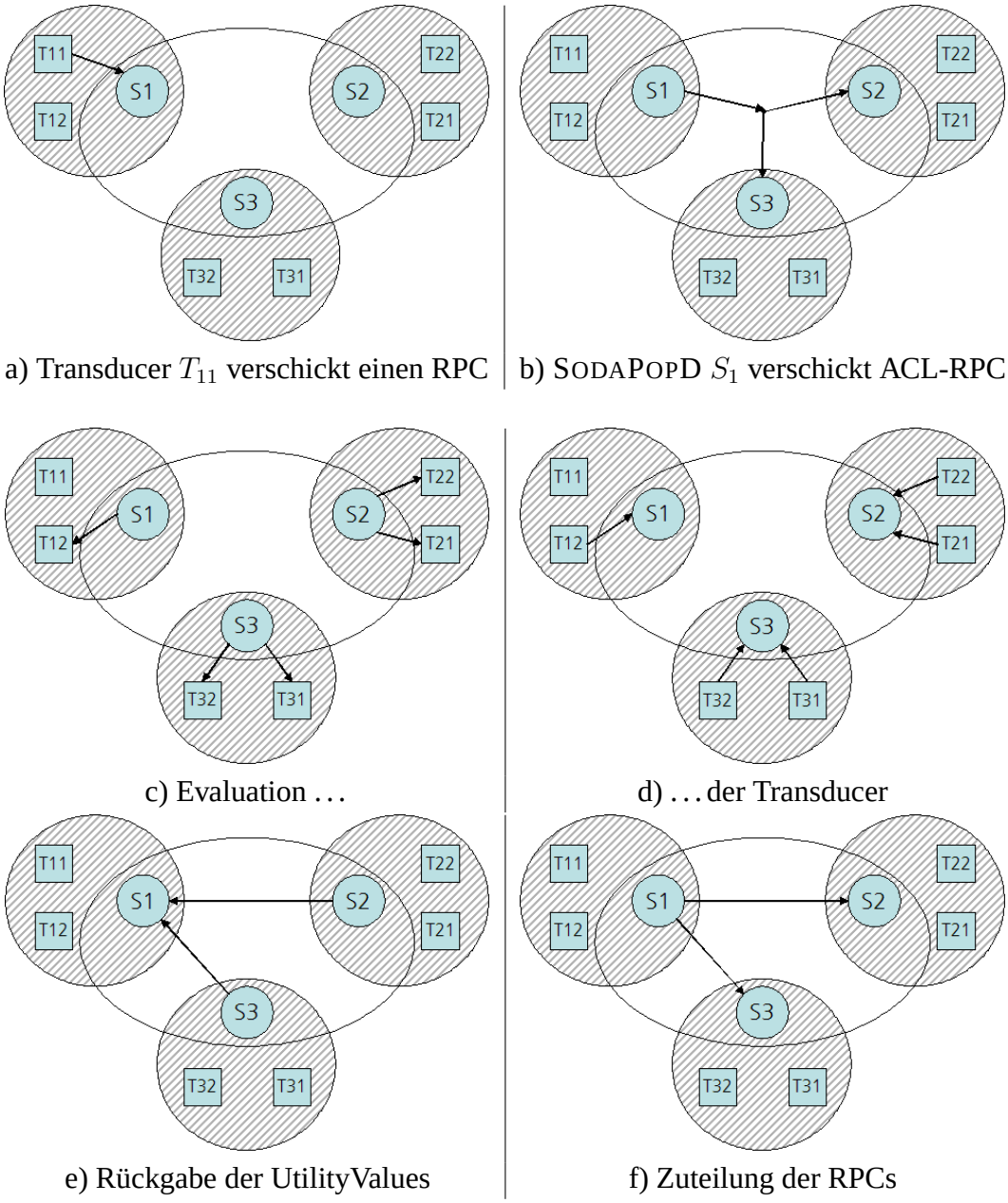


Abbildung 65: Sequenzdiagramm: Verarbeitung eines RPCs, welches von einem Transducer der SODAPOPD-Instanz S1 initiiert wurde. Die Verarbeitung geschieht innerhalb der Kanalgruppe mittels ACL-Remote Procedure Calls. Im Beispiel bilden vier SODAPOPD-Instanzen – in Vertretung ihrer Transducer – eine Kanalgruppe.

Diese Vorgehensweise unterscheidet sich ab Punkt 3 von der in Kapitel 5.2.3 geschilderten Vorgehensweise zur Behandlung von Events innerhalb einer Kanalgruppe. Der grundsätzliche Unterschied ist, dass prinzipiell keine SODAPOPD-Instanz gesucht werden muss, die die Konfliktlösungsstrategie ausführt. Hier setzt die Regel ein, dass bei einem Gerät welches einen Transducer besitzt, der Informationen und Ereignisse verarbeiten kann, genügend Ressourcen vorausgesetzt werden können, um algorithmische Strategien auszuführen. Daher ist diejenige SODAPOPD-Instanz für die Ausführung der Konfliktlösungsstrategie verantwortlich, deren Transducer einen solchen initialen Remote Procedure Call verantwortet.

Abbildung 65, Abbildung 66 und Abbildung 67 illustrieren die genaue Vorgehensweise. Hierbei sind in Abbildung 65 vier SODAPOPD-Instanzen mitsamt deren *ACL*-Instanz abgebildet, in Abbildung 66 und Abbildung 67 sind drei SODAPOPD-Instanzen und deren am selben Kanal beteiligten Transducer illustriert. Dies dient der erhöhten Übersichtlichkeit und stellt keinerlei Einschränkung der zugrunde liegenden Szenarios dar. Das Sequenzdiagramm in Abbildung 65 spezifiziert die Verwendung der von der ACL zur Verfügung gestellten Methoden, die Illustrationen in Abbildung 66 und Abbildung 67 spezifizieren die detaillierte Vorgehensweise und das dabei verwendete semantische Pro-



a) Transducer T_{11} verschickt einen RPC

b) SODAPOPD S_1 verschickt ACL-RPC

c) Evaluation ...

d) ... der Transducer

e) Rückgabe der UtilityValues

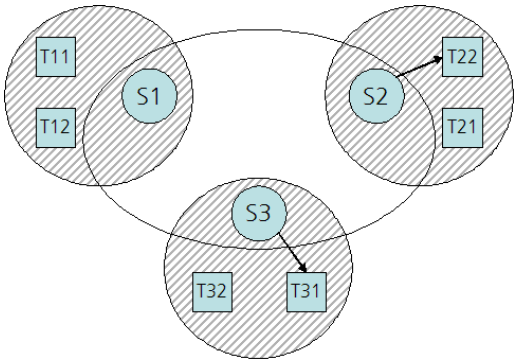
f) Zuteilung der RPCs

Abbildung 66: Illustration der Behandlung eines Remote Procedure Calls innerhalb einer Kanalgruppe (Teil 1).

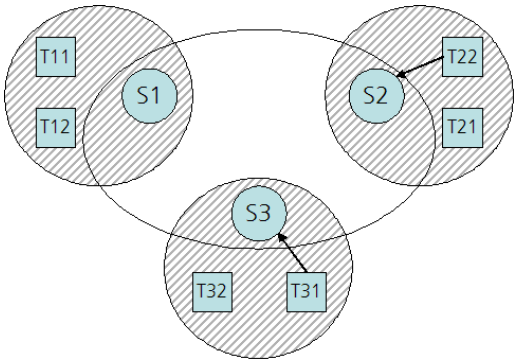
tokoll.

Ein Transducer sendet mittels der folgenden Nachricht einen Remote Procedure Call an einen Kanal (siehe Definition des unterlagerten Protokolls in Kapitel 5.1.4 und Abbildung 66 a)):

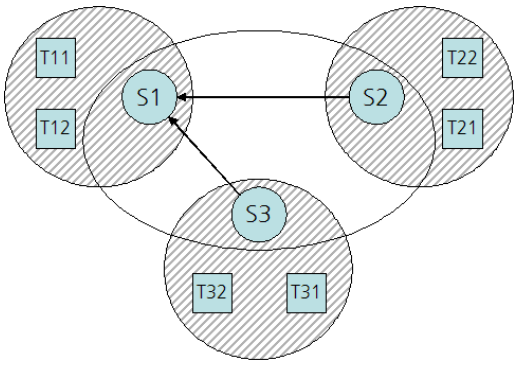
```
(rpc
     :channel Kanalname
     :sender Transducer-ID
```



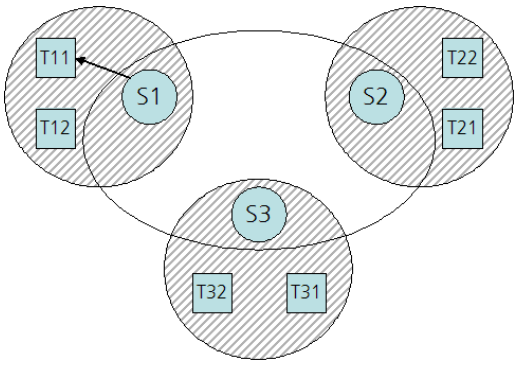
g) Weitergabe der RPCs an die Transcuder



h) Rückgabe der RPC-Antworten



i) Zusammenführung der RPC-Antworten



j) Ende des RPC-Vorganges.

Abbildung 67: Illustration der Behandlung eines Remote Procedure Calls innerhalb einer Kanalgruppe (Teil 2). Für die Kommunikation innerhalb der Kanalgruppe werden ACL-RPCs verwendet.

```
:ref-no RW-RPC
:content ( rpc ))
```

Nach Erhalt dieses RPCs sendet der empfangende SODAPOPD(AEMON) (im Beispiel in Abbildung 66 die SODAPOPD-Instanz S1) einen ACL-Remote Procedure Call an die entsprechende Kanalgruppe (siehe Abbildung 65 Operationen 1 bis 16 und Abbildung 66 b)) mittels der Methode *sendChannelRPC()*:

```
sendChannelRPC(Kanalname, eigeneSodaPopID, messageID, message)
message = (proposeRPC
       :ref-no RW1
       :content ( rpc ))
```

Dieser ACL-RPC wird von den beteiligten SODAPOPD-Instanzen der Kanalgruppe mittels des Methodenaufrufes *receiveChannelRPC()* der ACL empfangen:

```
receiveChannelRPC(Kanalname, messageID, message)
message = (proposeRPC
       :ref-no RW1
       :content ( rpc ))
```

Im Sequenzdiagramm in Abbildung 65 ist dies in den Operationen 3, 5 und 7 dargestellt. Hierbei werden die Operationen 2, 4 und 6 nahezu zeitgleich ausgeführt und sind in Abbildung 65 nur aus Gründen der Übersichtlichkeit und des Verständnisses zeitlich versetzt dargestellt. Die initiierende SODAPOPD-Instanz kann für diese Nachricht eine eigene Nachrichten-Identifikationsnummer (*messageID*) vergeben, ebenso wie eine eindeutige Referenznummer innerhalb der Nachricht zur Identifikation des gesamten RPC-Vorganges. In Abbildung 66 c) und d) werden die sich nun anschließenden internen Vorgänge innerhalb jeder SODAPOPD-Instanz illustriert. Jede SODAPOPD-Instanz evaluiert die UtilityValue-Funktionen ihrer Transducer mittels:

```
(proposeRPC
     :channel Kanalname
     :ref-no RW2
     :content ( rpc ))
```

Die Referenznummer *RW2* kann dabei von jeder SODAPOPD-Instanz frei vergeben werden, da sie nur intern für die Kommunikation mit den eigenen Transducern verwendet wird. Im positiven Falle senden die Transducer die folgende Antwort an ihren SODAPOPD(AEMON) (siehe Abbildung 66 d)):

```
(acceptRPC
     :sender Transducer-ID
     :channel Kanalname
     :ref-no RW2
     :content (utilityValue))
```

Wird die UtilityValue-Funktion negativ evaluiert, lautet die Antwort wie folgt:

```
(rejectRPC
     :sender Transducer-ID
     :channel Kanalname
     :ref-no RW2
     :content (false))
```

Jede SODAPOPD-Instanz kann nun den ACL-Remote Procedure Call beantworten (siehe Abbildung 66 e) und Abbildung 65 Operationen 8 bis 16). In dieser Antwort sind die UtilityValues aller eigenen Transducer enthalten. Gemäß Abbildung 66 e) würde die SODAPOPD-Instanz S2 mittels der ACL-Methode *sendRpcResponse* folgende Nachricht absenden:

```
sendRpcResponse(messageID, message)
message = (utilityValues
       :ref-no RW1
       :content ((T22 (UV-T22)), (T21 (UV-T21))))
```

Der Aufruf derselben Methode sieht für die SODAPOPD-Instanz S3 ähnlich aus:

```
sendRpcResponse(messageID, message)
message = (utilityValues
       :ref-no RW1
       :content ((T31 (UV-T31)), (T32 (UV-T32))))
```

Bei der initiierenden SODAPOPD-Instanz S1 ruft die ACL somit die *receiveRpcResponse()*-Methode auf. Hierin ist die Absender-SODAPOPD-Instanz mitkodiert. Die Aufrufe lauten im Einzelnen (gemäß Abbildung 66 e)):

```
receiveRpcResponse(S2, messageID, message)
message = (utilityValues
       :ref-no RW1
       :content ((T22 (UV-T22)), (T21 (UV-T21))))
```

und

```
receiveRpcResponse(S3, messageID, message)
message = (utilityValues
       :ref-no RW1
       :content ((T31 (UV-T31)), (T32 (UV-T32))))
```

Der initiierende SODAPOPD S1 verfügt nun über die Informationen über den initialen Remote Procedure Call eines seiner eigenen Transducer, den Bezeichner des Kanals, und über die einheitlichen Referenznummern über die UtilityValues aller an dem Kanal als RPC-Senken konnektierten Transducer und deren zuständigen SODAPOPD-Instanzen. Mit diesen Informationen ist er in der Lage, die auf dem Kanal definierte Konfliktlösungsstrategie anzuwenden und die Empfänger des Remote Procedure Calls zu bestimmen. Nach der Ausführung der Strategie kann er an die entsprechenden SODAPOPD-Instanzen einen ACL-Remote Procedure Call versenden (siehe Abbildung 65 Operation 17 - 21). Im Beispiel in Abbildung 66 hat die strategie-ausführende SODAPOPD-Instanz sowohl den Transducer T_{22} (gehostet von der SODAPOPD-Instanz S2) als auch den Transducer T_{31} auf S3 als Bearbeiter des initialen RPCs ermittelt. Dieses Beispiel vermittelt anschaulich die Vorgehensweise bei einer kooperativen Bearbeitung eines Remote Procedure Calls. Die SODAPOPD-Instanz versendet somit sowohl an S2 als auch an S3 jeweils einen ACL-RPC mittels der ACL-Methode *sendRPC()*:

```
sendRPC(S2, S1, messageID, message)
message = (rpc
       :channel Kanalname
       :receiver T22
       :ref-no RW1
       :content ( rpc ))
```

bzw:

```
sendRPC(S3, S1, messageID, message)
message = (rpc
       :channel Kanalname
       :receiver T31
       :ref-no RW1
       :content ( rpc ))
```

Bei der SODAPOPD-Instanz S2 wird durch die ACL die Methode *receiveRPC()* aufgerufen:

```
receiveRPC(messageID, message)
message = (rpc
       :channel Kanalname
       :receiver T22
       :ref-no RW1
       :content ( rpc ))
```

bzw. bei der SODAPOPD-Instanz S3:

```
receiveRPC(messageID, message)
message = (rpc
       :channel Kanalname
       :receiver T31
       :ref-no RW1
       :content ( rpc ))
```

Die Angabe der Referenznummer *RW1* innerhalb der Nachricht macht es dem initiierenden SODAPOPD(AEMON) S1 möglich, den gesamten Verlauf des RPC-Vorganges nachzuvollziehen. Auch die SODAPOPD-Instanzen S2 und S3 können sich auf diese Nummer beziehen. Für die Nummer *messageID* eines ACL-RPCs ist für jeden Vorgang eine neue Nummer zu vergeben. Sie verliert innerhalb der ACL-Schicht nach einmaligem Gebrauch ihre Gültigkeit. Wie in den Parametern der Methoden *sendRPC()* und *receiveRPC()* zu ersehen wird mit dieser Nachrichtennummer eine eindeutige Zuweisung von ACL-RPC zu Sender und Empfänger hergestellt. Eine Nachrichtennummer *messageID* in einem ACL-RPC kann somit nur einmal verwendet werden. Die *messageID* ist somit auch eindeutig als Identifizierung des adressierten Transducers zu verwenden. Jede SODAPOPD-Instanz, die eine solche Nachricht mit dem Befehl *rpc* erhält, kann diesen Remote Procedure Call dem Transducer an dem angegebenen Kanal zuordnen. Dies geschieht – analog zu Kapitel 5.1.4 – mittels der Nachricht (siehe Abbildung 67 g):

```
(rpc
     :sender Kanalname
     :ref-no RW3
     :content ( rpc ))
```

Die Antworten der Transducer entsprechen der folgenden Syntax (siehe Abbildung 67 h)):

```
(rpcResponse
     :channel Kanalname
     :sender Transducer-ID
     :ref-no RW3
     :content ( rpcResponse ))
```

Wiederum ist hierbei die Referenznummer *RW3* von jeder SODAPOPD-Instanz frei bestimmbar, da sie nur für die interne Kommunikation mit den eigenen Transducern eingesetzt wird. Nach Erhalt der Transducer-Antworten sendet jeder SODAPOPD(AEMON) die Antworten als Response auf den vorherigen erhaltenen ACL-Remote Procedure Call (siehe Abbildung 67 i) und Abbildung 65 Operation 20 - 22). Die SODAPOPD-Instanz S2 antwortet folgendermaßen:

```
sendRpcResponse(messageID, message)
(rpcResponse
       :channel Kanalname
       :ref-no RW1
       :content ( rpcResponse ))
```

Die ACL ruft daraufhin bei der SODAPOPD-Instanz S1 die Methode *receiveRpcResponse()* auf (siehe Abbildung 65 Operation 22):

```
receiveRpcResponse(S2, messageID, message)
(rpcResponse
       :channel Kanalname
       :ref-no RW1
       :content ( rpcResponse ))
```

In der Antwort sind sowohl Informationen über den antwortenden SODAPOPD(AEMON) S2, als auch Informationen über die – nur einmalig zu verwendete – Nachrichtennummer *messageID*, die eindeutig mit dem bearbeiten Transducer verknüpft ist, als auch die für diesen Remote Procedure Call global verwendete Referenznummer *RW1* verfügbar. Die Nachricht ist somit eindeutig dem antwortenden Transducer (welcher sich aus der Ausführung der Konfliktlösungsstrategie ergeben hat) als auch dem initialen RPC zuzuordnen. Analog gilt für die SODAPOPD-Instanz S3 (siehe Abbildung 67 i)) die ebenfalls die Methode *sendRpcResponse()* verwendet:

```
sendRpcResponse(messageID, message)
(rpcResponse
       :channel Kanalname
       :ref-no RW1
       :content ( rpcResponse ))
```

Bei der SODAPOPD-Instanz S1 wird wiederum die Methode *receiveRpcResponse()* aufgerufen:

```
receiveRpcResponse(S3, messageID, message)
(rpcResponse
       :channel Kanalname
       :ref-no RW1
       :content ( rpcResponse ))
```

Der SODAPOPD(AEMON) ist nun in der Lage die RPC-Antworten sowohl dem initialen RPC als auch den antwortenden Transducer zuzuordnen und gegebenenfalls nach erneuerter Ausführung der Kanalstrategie diese Antworten zu einer Gesamtantwort zusammenzuführen und dem initiierenden Transducer als Antwort auf seinen Remote Procedure Call zu übermitteln. Dies geschieht ganz analog zu dem unterlagerten Protokoll wie es in Kapitel 5.1.4 beschrieben ist. Gemäß Abbildung 67 sendet der SODAPOPD(AEMON) S1 die folgende Nachricht an seinen Transducer:

```
(rpcResponse
     :channel Kanalname
     :ref-no RW-RPC
     :content ( rpcResponse ))
```

Der Transducer kann anhand der vom ihm selbst vergebenen Referenznummer *RW-RPC* die Antwort auf seinen Remote Procedure Call identifizieren und die Antwort gemäß seiner internen Logik weiterverarbeiten.

5.3 Zusammenfassung

In diesem Kapitel wurde eine Software-Infrastruktur für selbstorganisierende Komponentenensembles auf Basis des SODAPOP-Modells spezifiziert. Die in Kapitel 5.1 spezifizierte Infrastruktur basiert auf einem singulären SODAPOPD(AEMON) der die Verwaltung aller durch die konnektierten Transducer definierten virtuellen Kanäle sowie die Ausführung der kanal-abhängigen Konfliktlösungsstrategien ausführt. Diese Vorgehensweise garantiert die Dynamik und Selbstorganisation der Komponenten, die mit diesem SODAPOPD(AEMON) verbundenen sind.

Um die völlige Dynamik von physikalischen Geräten zu garantieren, wird in Kapitel 5.2 das Stellvertreterprinzip durch die SODAPOPD-Instanzen eingeführt. Anstatt die Kommunikation der unterschiedlichen Komponenten aller in einem Geräteensemble vorhandenen Geräte durch einen singulären SODAPOPD(AEMON) zu gewährleisten, agiert eine SODAPOPD-Instanz als Stellvertreter eines physikalischen Gerätes. In einem solchen Ensemble vertritt ein SODAPOPD(AEMON) seine Transducer und die durch ihn definierten Kanäle innerhalb von Kanalgruppen. In solchen Kanalgruppen gibt es keinerlei ausgezeichnete Komponenten. Somit sind die Voraussetzungen für Dynamik (Hinzutreten und Verlassen von Komponenten und Geräten zu einem Ensemble zu jeder Zeit) und für Selbstorganisation (Aufrechterhaltung des Datenflusses zu jeder Zeit) gegeben. Kapitel 5.2 spezifiziert, wie mit dem Stellvertreterprinzip die Bildung von Kanälen, das Propagieren von Events, die Bearbeitung von Remote Procedure Calls und die hierfür notwendige Ausführung von kanal-abhängigen Konfliktlösungsstrategien dezentral und selbstorganisiert bewerkstelligt wird.

Bei der Ausarbeitung der hier spezifizierten Software-Infrastruktur wurde ein großes Augenmerk darauf gelegt, dass der Anwender – d.h. der Entwickler von Komponenten für dynamische Komponentenensembles – so wenig wie möglich mit den internen Vorgängen der unterlagerten Kommunikation belastet ist. Wie in Kapitel 5.1.5 und Kapitel 5.1.6 in einem Beispiel und einer Diskussion gezeigt, ist ein Transducer leicht aus bereits vorhandenen Softwarekomponenten implementierbar. Ein Transducer definiert die Kanäle, mit denen dieser verbunden sein möchte und weist diesen Kanälen entsprechende *Handler*-Objekte zur Bearbeitung von Events bzw. Remote Procedure Calls zu. Die unterlagerte Kommunikation findet zwischen den Kanalobjekten und der geräte-eigenen SODAPOPD-Instanz statt. Besonders die Definition von Algorithmen für *Service Discovery* oder *Service Composition* sind nicht im Verantwortungsbereich eines Komponentenentwicklers. Diese Konfliktlösungsstrategien werden von den SODAPOPD(AEMON)s zur Verfügung gestellt und ausgeführt. Die wichtigsten Eigenschaften des unterlagerten Protokolls seien hier zusammengefasst:

- *Semantische Kodierung* der Kanalbezeichner (auf Seiten der Transducer für die eindeutige Zuordnung der Kanalobjekte, auf Seiten der SODAPOPD(AEMON)s für die

Befehl	Sender-Empfänger	Parameter	Details
register	Transducer → SODAPOPD	*receiver* *ref-no*	Anmeldevorgang Transducer an seiner geräte-eigenen SODAPOPD-Instanz
accept	SODAPOPD → Transducer	*receiver* *sender* *ref-no*	Anmeldevorgang am SODAPOPD abgeschlossen.
unregister	Transducer → SODAPOPD	*sender* *receiver*	Abmeldevorgang vom SODAPOPD.
subscribe	Transducer → SODAPOPD	*receiver* *sender* *channel-in /-out*	Konnektieren an einem Kanal.
unsubscribe	Transducer → SODAPOPD	*receiver* *sender* *channel-in /-out*	Abmelden von einem Kanal.

Tabelle 1: Administrative Nachrichten zur Konnektion, Definition, Anmeldung und Abmeldung von Kanälen von Transducern und deren geräte-eigenen SODAPOPD-Instanz.

eindeutige Zuordnung aller Kanalnachrichten und der korrekten Ausführung von kanal-eigenen Konfliktlösungsstrategien)

- Garantie der *Selbstorganisation* von kanalgruppenbildenden SODAPOPD-Instanzen
- *Kapselung* unterlagerter Informationen vor den Softwarekomponenten

In jeder der drei Kommunikationsschichten Transducer – SODAPOPD – ACL (siehe Abbildung 56) wird genau diejenige Informationsmenge verarbeitet, die für die Garantie der Selbstorganisation an den jeweiligen Kommunikationsstellen nötig sind. Besonders die Transducer sind somit von unnötigen Verwaltungstätigkeiten entlastet. Tabelle 1 und Tabelle 2 fassen das in Kapitel 5.1 und Kapitel 5.2 in der Anwendung beschriebene unterlagerte Protokoll zwischen den Transducern und ihren geräte-eigenen SODAPOPD-Instanzen in einer Übersicht zusammen.

Es sei hier noch einmal betont, dass das definierte Protokoll *unterlagert* zwischen den Kanalobjekten und den SODAPOPD-Instanzen verwendet wird. Es ist ein semantisches Protokoll, dessen Informationen von den *Channel*- und *SodaPopTransducerD*-Objekten gemäß Abbildung 54 zum Aufruf der entsprechenden Methoden der vom Komponentenentwickler implementierten *EventHandler*- und *RPCHandler*-Objekten (siehe Abbildung 45) genutzt werden. Somit ist hier ein Programmiermodell realisiert worden, dass die funktionale Logik der Anwendung – die im Fokus der Anwendungsentwickler steht – komplett von den unterlagerten Kommunikationsmechanismen abkapselt. Die Kommunikation zwischen den SODAPOPD-Instanzen findet mittels den Technologien JXTA und/oder Universal Plug and Play statt. Die Details über die JXTA- und

Befehl	Sender-Empfänger	Parameter	Details
event	Transducer → SODAPOPD	*channel* *sender* *content*	Transducer sendet Event an einen Kanal.
rpc	Transducer → SODAPOPD	*channel* *sender* *ref-no* *content*	Transducer sendet RPC an einen Kanal.
event	SODAPOPD → Transducer	*channel* *content*	Transducer erhält Event von einen Kanal.
rpc	SODAPOPD → Transducer	*channel* *ref-no* *content*	Transducer erhält RPC von einen Kanal.
proposeEvent	SODAPOPD → Transducer	*channel* *ref-no* *content*	Abfrage der UtilityValues für einen Event.
acceptEvent	Transducer → SODAPOPD	*sender* *channel* *ref-no* *content*	Positive Evaluierung der UtilityValues.
rejectEvent	Transducer → SODAPOPD	*sender* *channel* *ref-no* *content*	Negative Evaluierung der UtilityValues.
proposeRPC	SODAPOPD → Transducer	*channel* *ref-no* *content*	Abfrage der UtilityValues für einen RPC.
acceptRPC	Transducer → SODAPOPD	*sender* *channel* *ref-no* *content*	Positive Evaluierung der UtilityValues.
rejectRPC	Transducer → SODAPOPD	*sender* *channel* *ref-no* *content*	Negative Evaluierung der UtilityValues.

Tabelle 2: Logische Nachrichten zwischen Transducern und ihren SODAPOPD-Instanzen zur Bearbeitung von Events und Remote Procedure Calls.

UPnP-Kommunikationsschicht sind nicht Gegenstand dieser Arbeit. Sie finden sich in der Spezifikation [109] des Projektes DYNAMITE. Beide Technologien können die für diese Spezifikation der Kommunikation innerhalb von Kanalgruppen nötige Peer-to-Peer-Kommunikation zur Verfügung stellen. JXTA zeichnet sich hier durch eine detailreiche Programmierschnittstelle für die Programmiersprachen *Java* und *C* aus. Im Gegensatz dazu zeichnet sich UPnP durch eine schnell wachsende (industrielle) Anwenderschaft aus. Es ist davon auszugehen, dass Kommunikationstechniken mittels UPnP in der Zukunft vermehrt eingesetzt werden. Der in dieser Arbeit definierte Abstract Connection Layer (siehe Abbildung 56) bietet sowohl Event- als auch Remote-Procedure-Call-Mechanismen an. Diese sind nicht äquivalent zu den im SODAPOP-Modell verwendeten Events und RPCs. Die Spezifikation der verteilten Implementierung des SODAPOP-Modells (siehe Kapitel 5.2) wendet diese ACL-Events und ACL-RPCs zur Verarbeitung von SODAPOP-Events und SODAPOP-RPCs an. Durch diese Lösung (siehe Sequenzdiagramme zur Behandlung von Events und Remote Procedure Calls in Abbildung 62 und Abbildung 65) gelingt es in allen Ebenen der Kommunikation konsistent zu bleiben. Generell müssen unter Verwendung der hier spezifizierten Software-Infrastruktur für die Realisierung von dynamischen Komponenten- bzw. Geräteensembles die folgenden Schritte ausgeführt werden:

1. **Identifikation der Komponententypen:** Im ersten Schritt werden die unterschiedlichen beteiligten Komponententypen definiert. Eine Komponente lässt sich definieren als eine Applikation, die Informationen konsumiert und diese Informationen abändert bzw. sie mit neuen Informationen anreichert. Eine Komponente kann somit den Übergang von einer Ontologie zu einer anderen Ontologie bedeuten[49].

2. **Identifikation der beteiligten Kanäle:** Nach der Identifikation der Komponententypen wird die Datenflusstopologie festgelegt. Dies bedeutet die Definition der Kommunikationspartner und der Kommunikationsrichtung. Nachdem dieser Vorgang abgeschlossen ist, ist eine Datenflusstopologie definiert. Außerdem ist hier zu unterscheiden, an welchen Stellen der Datenflusstopologie Events und an welchen Stellen der Datenflusstopologie Remote Procedure Calls eingesetzt werden.

3. **Definition der Kanalstrategien:** Nach Definition der Datenflusstopologie werden die Strategien ausformuliert, die von den jeweiligen identifizierten Kanälen zur Zuteilung von Events und Remote Procedure Calls verwendet werden. Diese werden in den SODAPOPD(AEMON)s implementiert.

4. **Implementierung von Komponenten:** Nachdem die Bezeichner der Kanäle definiert sind (siehe Punkt 2) und in den SODAPOPD(AEMON)s entsprechende Konfliktlösungsstrategien implementiert sind (siehe Punkt 3) können Komponenten spezifiziert und implementiert werden. Dies geschieht analog zu dem Vorgehen, wie es im Beispiel in Kapitel 5.1.5 illustriert ist.

[49] Der Begriff Ontologie wird hier in seiner Bedeutung als *Sprachraum* verwendet. In Abbildung 42 würde dies den Übergang vom Sprachraum der Benutzereingaben zum Sprachraum der Benutzerziele bedeuten.

6 Minimales Set für Ambient Intelligence

Die in Kapitel 2 beschriebenen und analysierten Szenarios zeichnen grundsätzliche Eigenschaften aus. Eine intelligente Umgebung bietet ihren Benutzern die Möglichkeit zur expliziten oder impliziten Interaktion und reagiert darauf mit einer geeigneten Anwendungsstrategie. Somit kann z. B. im EMBASSI-Projekt (vgl. Kapitel 2.1.1) die Wiedergabe von Musik per Spracheingabe durch den Benutzer initiiert werden (explizit) oder im DYNAMITE-Projekt (vgl. Kapitel 2.1.13) die Umgebung die Geräte des Hörsaals steuern, nachdem der Vortragende von seinem Sitz zum Vortragspult gegangen ist (implizit). Auch kann eine intelligente Umgebung im Sinne des Benutzers agieren, ohne dass dieser Aktion eine direkte oder indirekte Interaktion vorangegangen sein muss (z. B. das Wecken mittels Lieblingsmusik in EMBASSI). Offensichtlich muss eine intelligente Umgebung auf Kontextänderungen – seien diese implizit oder explizit durch den Benutzer erfolgt oder durch Änderungen anderer Umgebungsvariablen, die nicht im Zugriff des Benutzers stehen – mit geeigneten Aktionen reagieren. Diese Aktionen bestehen – gemäß den in Kapitel 2 analysierten Anwendungsszenarios – aus dem Aktivieren von Gerätefunktionen. Dies lässt sich auch allgemeiner formulieren (vgl. Kirste [135]): Änderung in Umgebungsvariablen veranlassen eine intelligente Umgebung zu einer geeigneten Adaption, d.h. zur Ausführung von Funktionen die wiederum eine Änderung der allgemeinen Umgebungsvariablen bewirken. Kulkarni, einer der Entwickler in den späteren Phasen des *Intelligent-Room*-Projektes beschreibt diesen Ansatz des prinzipiell zweistufigen Prozesses der *reactive behavioral systems* mit den Worten (vgl. [147]): „first, to understand the context of the current user; next, to adapt the room's response to that context". Etwas weniger allgemein wurde dies später im Sinne der Ideen des Ambient Intelligence von Markopoulos (vgl. [153]) beschrieben: „ambient intelligence vision is the use of system intelligence applications to sense, adapt to, and serve human needs". Eine intelligente Umgebung erfasst somit drei unterschiedliche Arbeitsschritte: Das Beobachten der Umgebung und das damit erfolgende Erfassen von Umgebungs- und Kontextvariablen, das Interpretieren dieser Daten im Sinne des Benutzers zum Erfassen möglicher Benutzerziele und zur Definition angepassten Systemreaktionen und zuletzt die Manipulation der Umgebung. Schon 1988 stellte Norman sein Modell der sieben Aktionsphasen in der Interaktion des Menschen mit technischen Umgebungen vor [166]. Im ersten Schritt, der Zielphase, definiert hier der Benutzer wie seine Umgebung nach Ausführung der zu initiierenden Funktionen aussehen soll. Der zweite Schritt, die Ausführungsphase, wird bei Norman mit drei unterschiedlichen möglichen Aktionsphasen beschrieben. Bei gegebenem Ziel schlägt hier das System entweder nacheinander einzelne Funktionsschritte vor, die der Benutzer dann auswählen und bestätigen kann, oder der Benutzer teilt dem System mit, welche Einzelschritte aus seiner Sicht heraus nötig sind (und das System führt diese dann aus) oder der Benutzer führt die notwendigen Einzelschritte selber aus. Der letzten Schritt, die Evaluationsphase, wird von Norman wiederum durch drei unterschiedliche Phasen definiert. In der ersten Phase nimmt der Benutzer nach Ausführung aller nötigen Funktionsschritte die veränderte Umwelt passiv wahr, während er in der zweiten Phase eine Interpretation und Definition der Unterschiede der jetzt realen Umge-

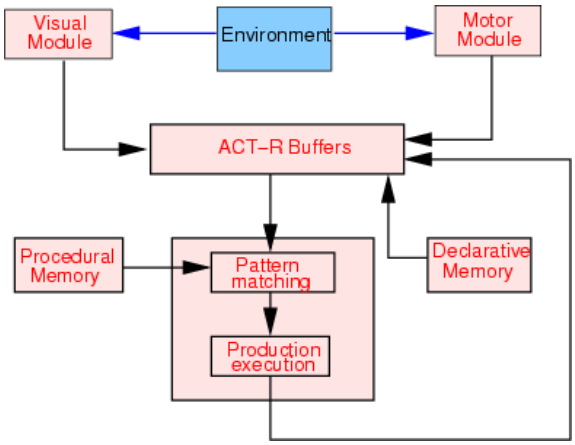


Abbildung 68: Die Architektur der Hauptkomponenten in ACT-R.

bung zu der ursprünglich geplanten Umgebung identifiziert. In der letztmöglichen Phase dieses dritten Schrittes unternimmt der Benutzer selbstständig Schritte zur Minimierung der erkannten Unterschiede. Ein erweitertes Modell – bestehend aus Informationsaufnahme, Informationsverarbeitung, Entscheidungsfindung und Reaktionsanalyse – wurde 2000 von Parasuraman, Sheridan und Wickens als Vier-Phasen-Modell der menschlichen Informationsverarbeitung bezeichnet [172]. Intelligente Systeme auf solcher Basis des Erkennens, Planens und Ausführens werden in der Literatur oftmals mit dem Begriff der *Kognitiven Architektur* (engl: Cognitive Architecture) bezeichnet (vgl. Langley [150], der oftmals auch den Begriff des „Intelligenten Agenten“ verwendet). Dieser der Forschungsrichtung der Künstlichen Intelligenz entnommene Begriff beschreibt Systeme, die in der Lage sind in verschiedenen Anwendungsgebieten auf Basis von Fakten und Ereignissen zusätzliches Wissen und Operationen abzuleiten. Laut Langley stellen diese Formen der Künstlichen Intelligenz aufgrund ihrer Fähigkeit zum Lernen – Erweiterung der eigenen Datenbasis aufgrund neuer Ereignisse, Daten und daraus abgeleiteten Fakten – den Gegensatz von reinen Expertensystem dar. Die ersten eingesetzten Technologien hier waren die *Produktionssysteme* (vgl. Newell [162]). Abbildung 68 illustriert die Architektur des *ACT-R* (Atomic Components of Though) der Carnegie Mellon University[50], welches ein Produktionssystem darstellt (vgl. [17, 18]), das zwei unterschiedliche Gedächtnismodule – für deklaratives und prozedurales Wissen – unterscheidet. Zugleich existieren Verbindungen zur realen Welt, um Umgebungsvariablen aufzunehmen und zu speichern. Die wichtigste Komponente ist die sog. „pattern-matching“-Komponente, die nach Produktionsregeln sucht, die auf den gegenwärtigen Zustand der Umgebungsvariablen passen. In ACT-R kann zur selben Zeit nur eine Produktionsregel „feuern“, d.h. ausgeführt werden. Daraufhin ändert sie die Umgebungswerte und somit den aktuellen Zustand des Gesamtsystems. Das Projekt *Icarus* der Stanford University ist ebenso wie ACT-R eine relativ neue Entwicklung auf dem Gebiet der Kognitiven Architekturen [40]. Hier wird das

[50] Informationen über das Projekt ACT-R sowie die Abbildung 68 sind unter http://act-r.psy.cmu.edu/ zu finden.

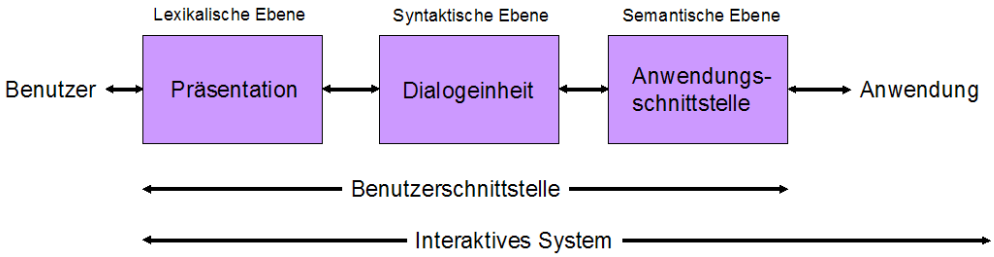


Abbildung 69: Das Seeheim-Modell für Interaktive Systeme.

Wissen des Systems modelliert als reaktive Fähigkeiten, die zielrelevante Reaktionen in Abhängigkeit von Situationsklassen definieren. Dabei werden Fähigkeiten als ein Satz an Zielen, ein Satz an Anforderungen und Vorbedingungen und ein Satz an Mitteln definiert. Jede dieser Eigenschaften kann dabei in Beziehung zu primitiven Aktionen oder Sensorereignissen gesetzt werden. Den unterschiedlichen Architekturen und Herangehensweisen der Kognitiven Architekturen liegt der sog. *Recognize-Act-Cycle* zugrunde. Dieser besteht aus vier Phasen: Der Abgleich der Regelköpfe gegen die vorhandene Wissensbasis („match“), die Erfassung („conflict resolution“) der feuernden Regel (derjenigen Regeln, deren Regelköpfe als wahr evaluiert wurden), die Anwendung der gefeuerten Regeln auf die vorhandene Datenbasis („apply“) und die Überprüfung eines Stopkriteriums („check“) und gegebenenfalls der Neubeginn des Zyklus. Laut Langley (vgl. [150]) muss ein Intelligenter Agent (bzw. eine Kognitive Architektur) über mehrere Fähigkeiten verfügen: Dies betrifft die Umgebungsbeobachtung und die Einordnung dieser Beobachtungen in eine Wissensbasis. Dann die Fähigkeit Entscheidungen zu treffen und die Anwendung von Strategien (Langley verwendet hier selbst den Begriff Konfliktlösungsstrategien), um unter verschiedenen Alternativen geeignet auswählen zu können. Die Fähigkeit zur Umgebungsbeobachtung sollte sich hierbei nicht auf einige wenige Sensoren beschränken. In Analogie zu Argumentationen von Abowd (vgl. [7]) und Oviatt (siehe [170]) ist an alle möglichen Arten von Umgebungssensoren zu denken, angefangen von multimodalen Eingabegeräten bis hin zu simplen Sensoren wie Temperaturerfassung. Abowd selbst erwartet von der Ausweitung der Erfassung von den rein expliziten Interaktionen hin zu vermehrter Berücksichtigung expliziter Interaktionen eine genauere Erfassung der Benutzersituation, während Oviatt die größtmöglichste Freiheit eines Benutzers in der Wahl seiner Interaktionsgeräte fordert. Ähnlich wie in einigen der analysierten Szenarios (z. B. EMBASSI in Kapitel 2.1.1) fordert auch Langley, dass der von ihm definierte intelligente Agent in der Lage sein muss, nicht nur auf sich ändernde Umgebungsvariablen zu reagieren, sondern auch von sich aus selbstständig und autonom zu agieren. Er selbst nennt dies die Fähigkeit eines Systems Vorhersagen aufgrund von Gelerntem zu treffen. Dies korrespondiert zu den Szenarios in EMBASSI oder DYNAMITE, in denen präferierte Sendungen des Benutzers automatisch aufgenommen werden bzw. die Lieblingsmusik zum Wecken verwendet wird[51].

[51] Offensichtlich kann auch dies nicht ohne äußere Änderungen von Umgebungsvariablen geschehen. In den zitierten Beispielen ändert sich zumindest die Uhrzeit. Dies soll jedoch keinen Widerspruch identifizie-

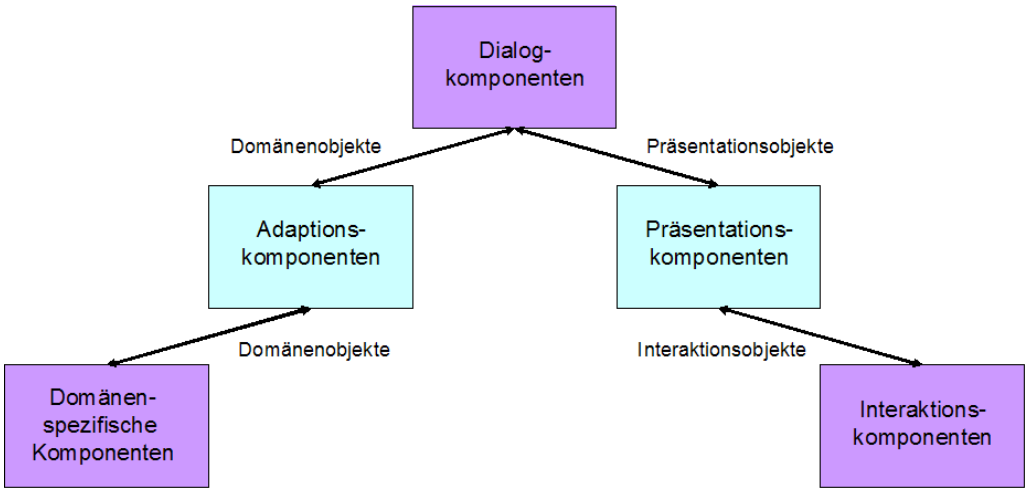


Abbildung 70: Die Komponenten des ARCH-Modells und ihre Schnittstellen nach [205]. Heller unterlegt sind die im Vergleich zum Seeheim-Modell eingeführten Komponenten zur Domänenadaption.

Die Arbeiten zu Kognitiven Architekturen konzentrieren sich sehr stark auf den Ausbau der Technologien der Künstlichen Intelligenz und stellen in sich stark zentralisierte und integrierte Applikationen dar. Auch beruht der Entwicklungsansatz weniger in der realen Manipulation der Umgebung als vielmehr in der Weiterentwicklung und Selbstadaption der internen Wissensbasis der eingesetzten Regelkomponenten (vgl. den Zyklus der Funktionsausführung in Abbildung 68). Ein Meilenstein in der Definition von Software-Architekturen auf der Basis von individuellen Komponenten für den Aufbau von interaktiven Systemen findet sich im Seeheim-Modell [173][52]. Das Seeheim-Modell (vgl. Abbildung 69) untergliedert die Benutzerschnittstelle in drei unterschiedliche Komponenten: Die Schnittstelle zur Anwendung, die Dialogeinheit und die Präsentationseinheit. Sie haben die Aufgabe die domänen-spezifischen Konzepte der Anwendung mittels syntaktischer und lexikalischer Aufbereitung für die Interaktion mit dem Benutzer zu modellieren. Das Seeheim-Modell weist insofern Ähnlichkeiten mit den oben besprochenen Ansätzen des Erfassens von Umgebungsvariablen und Benutzeräußerungen, des Interpretierens und Ausführens auf. In Analogie zu den Kognitiven Systemen, die die geplanten Umgebungsänderungen wiederum in den Regelkreislauf einspeisen, ist auch im Seeheim-Modell eine Rückführung von Information vorgesehen. Hier jedoch nicht in den Regelkreislauf eines Systems, sondern für aktive Dialogführung mit dem Benutzer. Das ARCH-Modell [26, 27] kann als eine Verbesserung des Seeheim-Modells angesehen werden. Es ist entstanden aus den Erfahrungen aus Entwicklungen auf Basis des Seeheim-Modells und beschreibt explizit die Art der Information die zwischen den einzelnen Architek-

ren. Es ist vielmehr gemeint, dass Systeme ohne explizite und implizite Benutzeraktionen in der Lage sein sollten „automatisch“ das Richtige zu tun.

[52] Das Model ist nach dem Teilort Seeheim von Seeheim-Jugenheim, das zwischen Heidelberg und Darmstadt liegt, benannt. Hier fand vom 1.-3. November 1983 ein Workshop für User Interface Management Systeme statt.

turkomponenten ausgetauscht wird (siehe Abbildung 70). Im Vergleich zum Seeheim-Modell befindet sich zwischen den Komponenten *Anwendung* und *Dialog* eine spezielle Adaptionskomponente. Die Einführung dieser Komponente beseitigte den Definitionsmangel im Seeheim-Modell, der hier jeweils Abstimmungen zwischen den Implementierungen der Anwendungskomponenten und Dialogkomponenten erzwang. Die zweite prinzipielle Änderung des ARCH-Modells gegenüber dem Seeheim-Modell besteht in der Abstufung der Präsentation dem Benutzer gegenüber in eine Präsentationskomponente und eine Interaktionskomponente. Die Interaktionskomponenten übernehmen die physische Präsentation und Interaktion mit dem Benutzer während die Präsentationskomponenten einen Satz an (abstrakten) Präsentationsobjekten bereitstellen. Diese werden dann von den Interaktionskomponenten in geeigneter Weise umgesetzt. Konsequenterweise stellt die Präsentationskomponente somit eine zweite Adaptionskomponente dar.

6.1 Komponententopologien für Ambient Intelligence

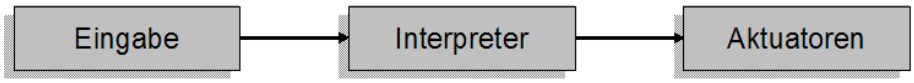
Gemäß den Anforderungen von intelligenten Umgebungen, ihre wesentlichen Kontextvariablen und Benutzeräußerungen beobachten und interpretieren zu können und auf Basis dieser Interpretationen Funktionsaufrufe an Geräten vornehmen zu können, muss ein Basis-Set für die Realisierung von Ambient-Intelligence-Szenarios aus den folgenden Komponenten bestehen:

Eingabekomponenten beobachten die direkten und indirekten Äußerungen des Benutzers, reichern sie gegebenenfalls mit zusätzlichen Annotationen an und senden sie an die nächste Verarbeitungsstufe. Neben expliziten und impliziten Benutzeräußerungen gibt es auch Eingabekomponenten, die eher sensorische Tätigkeiten ohne die Notwendigkeit eines Benutzereingriffes wahrnehmen. Beispiele für direkte Benutzerinteraktion sind Spracheingaben oder Eingaben per graphischer Benutzeroberfläche, Beispiele für indirekte Tätigkeiten sind Beobachtungen eines Bewegungsmelders. Sensorische Tätigkeiten können die Beobachtungen von Umgebungsvariablen wie Helligkeit oder Temperatur umfassen.

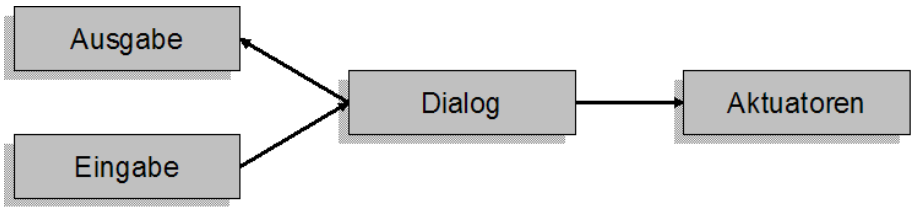
Interpreterkomponenten interpretieren die von den Eingabekomponenten übermittelten atomaren Ereignisse aus Benutzeräußerungen und Umgebungsvariablen und stellen diese in einen integrierten Kontext. Interpreterkomponenten sind verantwortlich ein Systemziel zu erkennen und dieses in passende Funktionsaufrufe umzusetzen.

Aktuatoren besitzen eine direkte Verbindung zu ihren jeweiligen assoziierten Geräten. Sie sind in der Lage die übermittelten Funktionsaufrufe in Gerätefunktionen umzuwandeln.

Abbildung 71 a) illustriert die sich daraus ergebende Basistopologie für Komponenten. Die Eingabekomponenten senden die von ihnen aufgenommenen atomaren Ereignisse zu den Interpreterkomponenten, die daraufhin mögliche Systemziele erarbeiten und diese



a) Basistopologie für Komponenten zur Realisierung von intelligenten Umgebungen

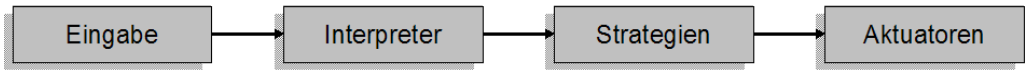


b) Basistopologie ergänzt um Komponenten für Systemausgaben

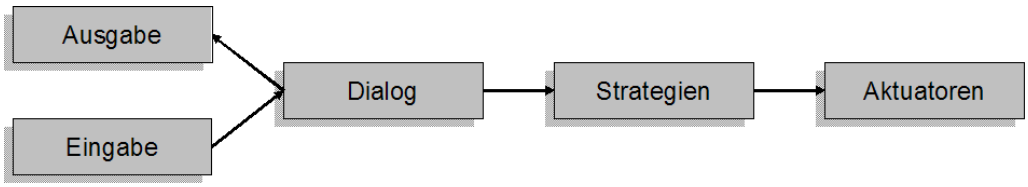
Abbildung 71: Die beiden Varianten der Basistopologie für Ambient-Intelligence-Szenarios auf Basis eines Datenflusses, der dem Prinzip des Beobachtens, Interpretierens und Agierens folgt.

dann in Form von Funktionsaufrufen an die abschließenden Aktuatorkomponenten weitergeben. Diese Topologie entspricht dem sich aus den analysierten Szenarios (vgl. Kapitel 2) ergebenden Datenfluss, der aus Beobachtung, Interpretation und Aktionshandlungen besteht. Die Pfeilrichtungen in Abbildung 71 geben die Hauptrichtung der verarbeiteten Information an. Rein reaktive Szenarios, wie das Verhalten eines intelligenten Raumes (vgl. Beschreibung des Intelligent Rooms in Kapitel 2.1.7) oder das gewünschte Verhalten eines Hörsaales oder der Besprechungsräume in DYNAMITE (vgl. Kapitel 2.1.13) sind mit diesem Modell offensichtlich realisierbar. In Kapitel 6.3 wird dies im Detail anhand von Plausibilitätsbeispielen diskutiert. Andere Szenarios wie die assistierte Auswahl von Musikstücken in EMBASSI (vgl. Kapitel 2.1.1) oder die Auswahldialoge im Projekt Smartkom (vgl. Kapitel 2.1.2) sind mit diesem einfachen Modell nicht realisierbar. Ein intelligentes System muss immer dann Dialoge führen, wenn es ihm nicht gelingt, nur auf Basis der mittels Sensoren und Interaktionskomponenten aufgezeichneten atomaren Ereignisse, geeignete System- und Benutzerziele zu interpretieren. Zur Auflösung von Ambiguitäten muss ein solches System somit Dialoge initiieren und führen können. Abbildung 71 b) zeigt die Ergänzung der Basistopologie, um diesen Herausforderungen gerecht zu werden. Mittels Ausgabekomponenten kann das System einen geeigneten Dialog mit der Außenwelt (i.A. der Benutzer) führen, um ein Ziel vollständig aufzulösen. Interpreterkomponenten mit solchen Fähigkeiten werden konsequenterweise Dialogkomponenten genannt.

Interpreterkomponenten bzw. Dialogkomponenten haben prinzipiell unterschiedliche Aufgaben. Zuerst müssen sie Benutzereingaben und Umgebungsvariablen zu einem geeigneten Benutzerziel interpretieren und dieses Ziel wiederum mittels einer Reihe von Funktionsaufrufen realisieren. Sie müssen also auch in der Lage sein, auf Basis eines gegebenen Ziels eine geeignete Ausführungsstrategie zu entwickeln. In einer Topologie gemäß der Anordnung in Abbildung 71 erfüllen sie somit zugleich Interpreter- und Dialogfunktion als auch Strategiefunktionen. Dies ist für viele Szenarios realisierbar. In Kapi-



a) Erweiterte Basistopologie für Komponenten zur Realisierung von intelligenten Umgebungen



b) Erweiterte Basistopologie ergänzt um Komponenten für Systemausgaben

Abbildung 72: Das erweiterte Topologiemodell nimmt eine physikalische Trennung in Dialogkomponente und Strategiekomponente vor.

tel 7 werden auf Basis dieser Topologien Anwendungen und Realisierungen im Detail beschrieben und analysiert. Für umfangreiche Anwendungen und auch für die Unterstützung verteilter Implementation ist es jedoch geboten diese logische funktionale Trennung der Dialogkomponenten auch physikalisch vorzunehmen. Dies wird in Kapitel 8 anhand des Projektes DYNAMITE beschrieben. Abbildung 72 illustriert diesen Ansatz der Erweiterung des Basis-Sets durch die Teilung der Dialogkomponenten in eine Dialogkomponente und eine Strategiekomponente. Die Dialogkomponente erfüllt nun die Aufgabe auf Basis der ihr übermittelten atomaren Ereignisse ein geeignetes Benutzerziel zu interpretieren und dies den Strategiekomponenten mitzuteilen. Die Strategiekomponenten verfügen über die Fähigkeit Strategien zur Erarbeitung oder Auswahl von Funktionen anzuwenden und diese dann mittels der Aktuatoren umzusetzen. Das Topologiemodell in Abbildung 72 oben wurde als Ausgangspunkt der Definition des SODAPOP-Modells in Kapitel 4 und zur Illustration der Verteilten Implementierung in Kapitel 5 verwendet (siehe auch Abbildung 38 und Abbildung 42).

6.2 Kanalgruppen für Komponententopologien

Abbildung 71 und Abbildung 72 illustrieren geeignete Komponententopologien für die Realisierung von Ambient-Intelligence-Szenarios auf Basis eines Datenflusses des Prinzips des Beobachtens, Interpretierens und Agierens. Die Topologieillustrationen zeigen jedoch nicht, dass jede Komponentenebene aus einer unbestimmten Anzahl an Komponenten bestehen kann. Genau diese Annahme war die Voraussetzung für die Definition des SODAPOP-Modells und für die Realisierung dieses Modells (vgl. Kapitel 4) mit der Anforderung der verteilten Implementation (vgl. Kapitel 5). Eingabekomponenten publizieren die von ihnen erfassten atomaren Ereignisse nicht mehr direkt zu den Interpreter- bzw. Dialogkomponenten, sondern senden diese an den Ereigniskanal (siehe Abbildung 73), an dem sie als Event-Quelle angemeldet sind (vgl. Kapitel 5.1.4). In genau derselben Art publizieren die Interpreter- bzw. Dialogkomponenten in der Basistopologie die von

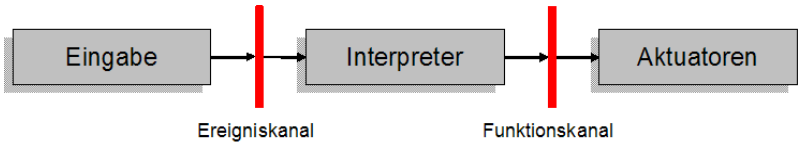
ihnen initiierten Funktionsaufrufe an den Funktionskanal. In der erweiterten Basistopologie mit der Berücksichtigung von Ausgabekomponenten publiziert die Dialogkomponente die Informationen über den zu führenden Dialog an den Ausgabekanal. In genau derselben Art kann in der erweiterten Topologie nach Abbildung 72 ein zusätzlicher Kanal für die System- und Benutzerziele definiert werden. Abbildung 73 unten illustriert, dass die Interpreter- bzw. Dialogkomponente hier die von ihr auf Basis der atomaren Ereignisse interpretierten Ziele auf den Zielkanal zur weiteren Bearbeitung sendet. Von hier aus sind Strategiekomponenten dafür zuständig unter Berücksichtigung des gegebenen Zieles Funktionen zu definieren, die dann dem Funktionskanal übergeben werden (vgl. Hellenschmidt und Kirste [106]). Die Einführung der Kanäle in den Komponententopologien als „Vermittler“ im Konfliktfall haben gewisse Ähnlichkeiten mit den Adaptorkomponenten im ARCH-Modell (vgl. Abbildung 70). Jedoch verfügen die Kanäle über bedeutende zusätzliche Eigenschaften. Neben der Fähigkeit Nachrichten abbilden zu können sind sie in der Lage Strategien im Konfliktfalle von konkurrierenden Komponenten anwenden zu können. Diese Mechanismen sind im statischen ARCH-Modell nicht vorgesehen. Für die möglichen Strategien auf den in den verschiedenen Komponententopologien definierten Kanälen (siehe Abbildung 74) gelten die folgenden Anforderungen:

Ereigniskanal: Eine Strategie auf dem Ereigniskanal zur Zuteilung atomarer Ereignisse wie expliziten und impliziten Benutzeräußerungen muss konkurrierende Interpretationen verhindern können. Diese Strategie muss realisieren, dass Ereignisse zu den Interpreter- und Dialogkomponenten gelangen, die in der Lage sind keine widersprüchlichen Interpretationen auszuführen. Ein intuitives Beispiel wären zwei Interpreterkomponenten die auf dieselbe Benutzeräußerung sowohl mit dem An- als auch mit dem Ausschalten desselben Gerätes reagieren würden.

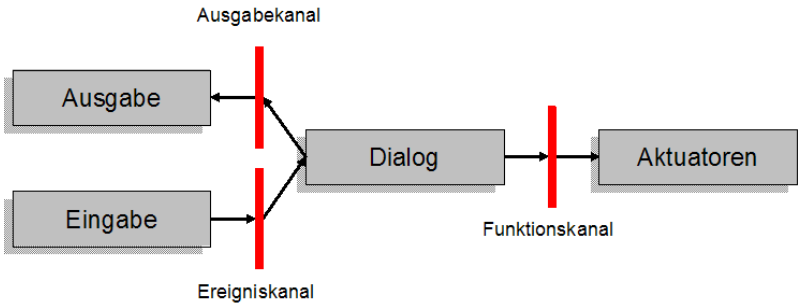
Zielkanal: Auf dem Zielkanal werden semantisch beschriebene Ziele publiziert. Dieses Ziel sollte diejenige Strategiekomponente weiter verarbeiten dürfen, die hierfür am Besten geeignet ist. Eine Konfliktlösungsstrategie auf dem Zielkanal sollte somit die am besten geeignete Strategiekomponente auswählen können.

Funktionskanal: Auf dem Funktionskanal gilt ebenso, dass diejenige Komponente (aus mehreren konkurrierenden Komponenten) eine Funktion ausführen soll, die hierfür am Besten geeignet ist.

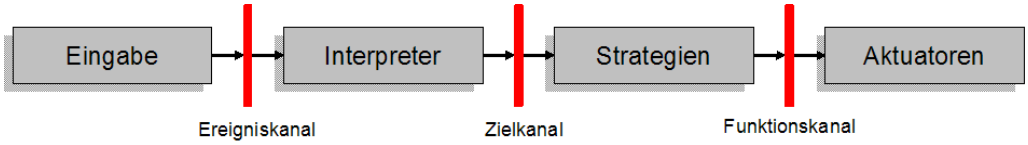
Eine Strategie auf dem Ausgabekanal in der erweiterten Basistopologie muss in der Lage sein, die Informationen von Dialogkomponenten in geeigneter Art und Weise auf die vorhandenen u.U. multi-modalen Ausgabekomponenten zu verteilen. Die Realisierung einer möglichen Strategie hierfür und deren Integration in der in dieser Arbeit spezifizierten Software-Infrastruktur wird in Kapitel 7.2 beschrieben. Die unterschiedlichen Teilszenarios in EMBASSI und DYNAMITE, aber auch die Szenarios der anderen beschriebenen Projekte (siehe Kapitel 2), weisen – obwohl sie sich durch ähnliche Komponententopologien abbilden lassen – ganz unterschiedliche prinzipielle Eigenschaften auf. Dies betrifft sowohl die Art und Anzahl der Interaktionsgeräte und Sensoren auf der Eingabeseite, als



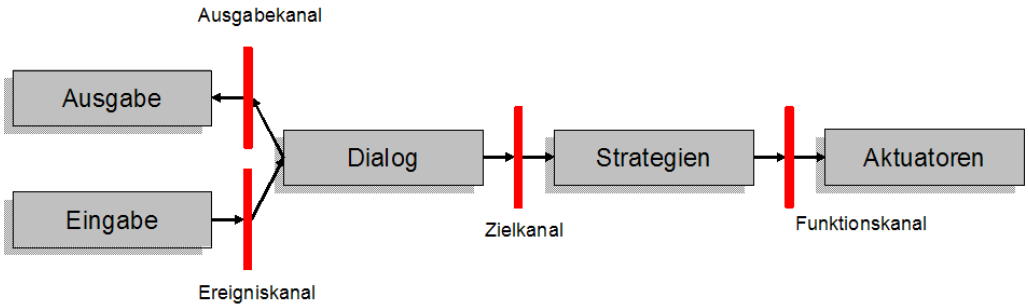
a) Die Basistopologie aus Abbildung 71 a)
unter Verwendung des SODAPOP-Modells.



b) Die Basistopologie ergänzt um Ausgabekomponenten aus Abbildung 71 b)
unter Verwendung des SODAPOP-Modells.



c) Die erweiterte Basistopologie aus Abbildung 72 a)
unter Verwendung des SODAPOP-Modells



d) Die erweiterte Basistopologie ergänzt um Ausgabekomponenten
aus Abbildung 72 b) unter Verwendung des SODAPOP-Modells

Abbildung 73: Die in Abbildung 71 und Abbildung 72 definierten Komponententopologien unter Verwendung des SODAPOP-Modells. Die einzelnen Komponenten sind hier mittels Kanälen verbunden.

auch die Verschiedenartigkeit der Geräte und Applikationen auf der Ebene der Aktuatoren. In unterschiedlichen Szenarios spielen oftmals ganz unterschiedliche Geräte und Applikationen die jeweils wichtigste Rolle. Die Bandbreite der Applikationen reicht von

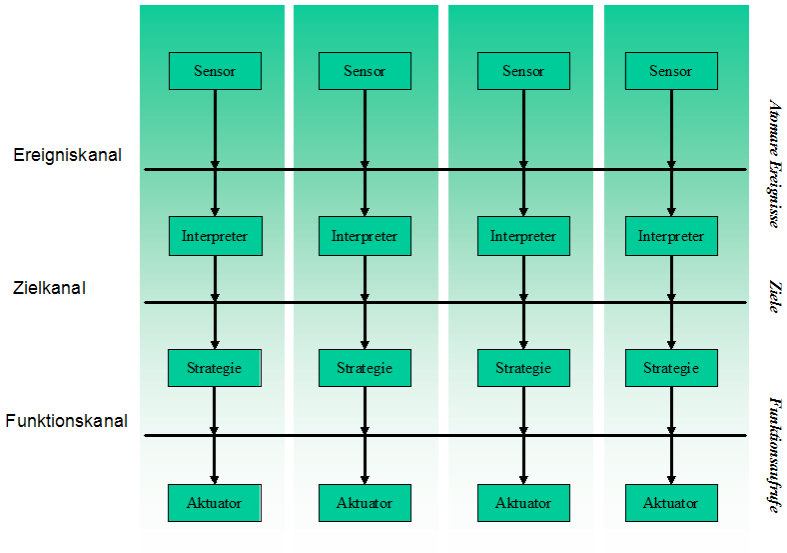


Abbildung 74: Illustration des Zusammenschlusses von vier Geräten auf Basis der erweiterten Basistopologie. Klar zu erkennen ist die Konkurrenzsituation auf den unterschiedlichen Kanälen.

Dokumentenpräsentation bis hin zu Abspielgeräten von Musik, vom Generieren eines Warntons bis hin zum intelligenten Steuern der Raumbeleuchtung. Eine ähnliche Bandbreite findet sich bei den Eingabegeräten, die von Sprache über graphische Benutzeroberflächen, bis hin zu Drucksensoren und Bewegungsmeldern reicht. Für die Bereitstellung einer Infrastruktur mit der Entwickler bzw. Entwicklerteams in der Lage sein sollen, auf einfache und schnelle Weise prototypische Anwendungen zu implementieren, sind somit Konfliktlösungsstrategien nötig, die keine domänenspezifischen Anforderungen stellen. Nur dann sind sie in einem breiten Spektrum von Anwendungsszenarios für die ersten prototypischen Realisierungen anwendbar. Auf den folgenden Seiten werden eine Strategie für die Zuteilung von Ereignissen im Falle konkurrierender Komponenten (siehe Kapitel 6.2.1) vorgestellt. Diese kann durch drei ebenfalls diskutierte Alternativstrategien ohne domänenspezifische Anforderungen ersetzt werden. Im Anschluss wird eine Auswahlstrategie für konkurrierende Komponenten vorgestellt und spezifiziert (siehe Kapitel 6.2.2). Diese stellt keinerlei Voraussetzungen an die jeweilige Domäne und ist somit universell einsetzbar.

6.2.1 Kanalstrategie zur Zuteilung von Ereignissen

Wie die Analyse möglicher Ambient-Intelligence-Szenarios in Kapitel 2 zeigt, sind eine Vielzahl unterschiedlicher Ereignisquellen möglich, die Events publizieren können. Hierbei können solche Ereignisse durch den Menschen initiiert sein (z. B. aktiv durch graphische Benutzeroberflächen, oder durch Sprache und Gestik, aber auch passiv durch das Auslösen von Sensoren), aber auch andere Ereignisse die z. B. durch die Messung von Umgebungsvariablen (z. B. Helligkeit und Temperatur) oder zeitlich gesteuerte Ereignisse sind hier vorstellbar. Laut Oviatt [169] ist gerade unter dem Begriff *Multimodalität* viel mehr zu verstehen als die Kombination von Sprache und Gestik. Besonders die Kombination von passiven Ereignissen und von Ereignissen, deren Quellen keine menschliche Interaktion unterliegen, sei in dem einfachen Sprache-Gestik-Modell nicht berücksichtigt. Auch die Definition von Gleichzeitigkeit (Sprache folgt Gestik, Gestik folgt Sprache) sei oftmals nur unzureichend. Um diesen Einwänden zu begegnen haben Fitzgerald und Firby [79] unterschiedliche Eingabemuster identifiziert und hierfür verschiedene Erkennungssysteme spezifiziert. Das dabei verwendete Eingabenmodell folgt einem hierarchischen rekursiven Ansatz. Sobald bekannte Muster innerhalb einer Kombination von multimodalen Eingaben erkannt werden, werden zusätzliche Ereignisse publiziert, die wiederum als Teil von höherwertigen Mustern erkannt werden können. Als Beispiel wird hier die Kombination von multimodalen Interfaces genannt, die jede Modalität für sich analysieren und strukturiert abbilden. Durch Integration können dann kombinierte Repräsentationen erschlossen werden. Fitzgerald und Firby stellen unterschiedliche Arten an Erkennungssystemen vor: Der *One-Recognizer* analysiert exakt ein Ereignis und wandelt dieses in ein neues Ereignis um (typischerweise sind Event-Handler objektorientierter Programmiersprachen Recognizer dieses Typs). Ähnlich verhält sich der Typ *Binding-Recognizer*, der Ereignissen einen Zustand zuweist. Eine wichtige Typenklasse ist der *In-order Recognizer*, der Muster an Ereignissen, die in einer bestimmten zeitlichen Reihenfolge geschehen, erkennt. Ein Sonderfall der In-order Recognizer stellt der Typus des *All-Recognizers* dar. Hier spielt die zeitliche Reihenfolge der publizierten Ereignisse keine Rolle. Ein weiterer Sonderfall ist der *One-of Recognizer*. Sobald ein Ereignis einer definierten Ereigniskette eingetroffen ist, gilt die betroffene Ereigniskette als erfüllt. Die letzten beiden definierten Erkennertypen sind die *Within-Recognizer* und die *Without-Recognizer*. Ein Within-Recognizer analysiert eine Ereigniskette als geschehen, wenn deren Ereignisse innerhalb eines definierten Zeitraumes publiziert wurden. Konsequenterweise analysiert der Without-Recognizer eine Ereigniskette erfolgreich, sobald deren Ereignisse nicht innerhalb eines bestimmten Zeitraumes erfolgt sind[53].

Für die in Kapitel 2 diskutierten Szenarios soll im Folgenden angenommen werden, dass die Reaktionen der Ambient-Intelligence-Umgebung auf der Analyse von sequentiellen Ereignisketten basiert. Dies bedeutet, dass das Aufeinanderfolgen bestimmter Ereignisse die Geräte eines Raumes zu Interpretation eines bestimmten Zieles veranlasst welches dann (möglicherweise in Kooperation) durch die im Raum befindlichen Geräte

[53] Schon 1983 definierte James Allen [10] das wechselseitige Verhältnis von zeitlichen Intervallen. Die hierauf basierenden Erkenner (*Allen-Recognizer*) können als eine Art von Superklasse der von Fitzgerald und Firby definierten *Within*- und *Without-Recognizer* angesehen werden.

auszuführen ist. Eine mögliche Ereigniskette kann analog zu Kapitel 2 sein: *Anwesender erhebt sich von seinem Stuhl – geht nach vorne – stellt sich ans Stehpult – und beginnt ins Mikrophon zu sprechen*. Zusammengehörende Ereignisse erfolgen somit zeitlich nacheinander und können folglich als Ereigniskette modelliert werden. Analog zu Fitzgerald und Firby [79] wird hierbei angenommen, dass Ereignisse diskret und zu einer bestimmten Zeit stattfinden[54]. Ereignisse werden im Folgenden immer als atomare (im Sinne von unteilbare) Ereignisse verstanden. Im eben geschilderten Beispiel könnte ein mögliches erkanntes Ziel sein, die Beleuchtung um Raum zu dimmen, den Projektor für Präsentationen einzuschalten, die zum Vortragenden gehörende Präsentation zu starten und den Mikrophonton auf die Lautsprecher zu geben. Die in dieser Arbeit diskutierten Systeme werden folglich Parser- und Erkennerkomponenten der beiden Typen *In-order Recognizer* und *One-Recognizer* zum Einsatz bringen[55]. Wie Abbildung 74 illustriert, ist es in einem dynamischen System möglich, dass verschiedene Interpreterkomponenten um dieselben Ereignisse von Sensoren oder direkten Benutzereingaben zur Interpretation von Benutzerzielen oder zur Interpretation von Systemreaktionen konkurrieren. Die Schattierungen in Abbildung 74 verdeutlichen den Zusammenschluss von vier Geräten, die jeweils unterschiedliche Interpreterkomponenten mit sich bringen. Die Definitionen für eine geeignete Konfliktlösungsstrategie, die geeignet ist diese Konkurrenzsituationen aufzulösen, sollen mit einem Beispiel motiviert werden.

Es seien zwei Ereignisse zeitlich dicht hintereinander erfolgt:

Ereignis a: Der Projektor wurde eingeschaltet
Ereignis b: Eine Person steht am Vortragspult

Gegeben seien nun zwei unterschiedliche Interpreterkomponenten (*P* und *Q*), wobei die Komponente *P* das singuläre Ereignis *b* dahingehend interpretiert die Intensität des Lichtes zu erhöhen (um die Person am Stehpult für das Auditorium herauszuheben und zu betonen). Die zweite Interpreterkomponente (Interpreter *Q*) interpretiert aber nun die Ereignissequenz $\{$a,b$\}$, d.h. die Folge von Ereignissen, dass der Projektor eingeschaltet wurde und sich danach eine Person zum Stehpult begeben hat, als das Ziel dieser Person eine Präsentation mit Hilfe des Projektors zu halten und formuliert das Ziel das Licht ein wenig zu verdunkeln. Etwas formaler sei diese Interpretationen von *P* und *Q* ausgedrückt durch:

$$Q : \{a, b\} \Rightarrow \text{„}\textit{Lichtintensität verringern}\text{“}$$

[54]Ohne Beschränkung der Allgemeinheit wird im Folgenden davon ausgegangen, dass Ereignisse einen festen Zeitpunkt haben, jedoch kein Zeitintervall. Ein gesprochener Satz hat somit keinen Anfangs- und Endezeitpunkt, sondern einen Zeitpunkt. Dieser ist der Zeitstempel, zu dem z. B. ein Spracherkenner das zu diesem Satz gehörende Ereignis publiziert hat. Das bekannte *„Put-that-there“*-Beispiel ist durch die Angabe verschiedener Ereignisketten modellierbar. Statt den Satz „Put that there“ durch Anfangs- und Endzeit plus die entsprechende Gestik zu definieren, kann je nach Modellierung des Spracherkenners dies in erster Näherung durch die Ereigniskette „put that there“ + Zeigegestik beschrieben werden. Weitere Näherungen sind: „put that“ + Zeigegestik + „there“ und „put“ + Zeigestik + „that there“ bzw. „put“ + Zeigestik + „that“ + „there“.

[55]*One-Recognizer* können analog wie *In-order Recognizer* behandelt werden, mit dem Unterschied, dass die interpretierbare Ereigniskette der *One-Recognizer* die Länge 1 hat.

$$P : \{b\} \Rightarrow \text{„}\textit{Lichtintensität erhöhen}\text{“}$$

Die Schlussfolgerung der Interpreterkomponente *P* basiert auf der Idee, dass das Publikum einen Vortragenden (der z. B. vom Blatt abliest) gerne genauer sieht, die Schlussfolgerung des Interpreters *Q* auf der Idee, dass ein Publikum einem Vortrag, der mittels eines Projektors gegeben wird, besser folgen kann, wenn dieser in einem etwas verdunkelten Raum gehalten wird (da hier der Kontrast der präsentierten Folien erhöht wird). Es ist leicht vorstellbar, dass die Interpreterkomponente *P* immer das Ziel formulieren würde, die Lichtintensität zu erhöhen, sobald ein Ereignis vom Stehpult publiziert wird. Interpreterkomponente *Q* scheint jedoch auch in der Lage, weitere Kontextinformationen (hier: das zeitnahe Aktivieren des Projektors) mit in seine Interpretationsleistung einzubeziehen. Offensichtlich ist, dass sich beide Interpretationen gegenseitig widersprechen und somit nicht zeitgleich ausgeführt werden sollten. Solche Interpretationen sind nicht zueinander *kompatibel*. Andererseits wirkt es intuitiv richtig, derjenigen Interpreterkomponente die Ausführung seiner Interpretationsleistung zu erlauben, die den breiter gefassten Kontext, d.h. die längere Ereigniskette zu ihrer Interpretation heranzieht.

Definition Kompatibilität: Ziele, die aus Ereignissen bzw. Ereignisketten interpretiert wurden, sind zueinander kompatibel, falls deren Ausführung ohne gegenseitige Widersprüche erfolgen kann.

Insbesondere sind zwei Interpretationen nicht kompatibel zueinander, wenn eine Interpretation in der Form A und die andere in der Form $negA(nichtA)$ beschrieben werden kann. Das eben geschilderte Beispiel entspricht genau diesem Fall. Aber auch der Fall, dass zwei Interpreterkomponenten *dasselbe Ziel* interpretieren kann zueinander nicht kompatibel sein. Sei der Fall gegeben, dass zwei Interpreterkomponenten die Aussage eines Benutzers einen bestimmten Auftrag auszuführen verstehen (z. B. eine Präsentation zu halten oder einen bestimmten Film aufzunehmen). Während z. B. das „doppelte“ Einschalten eines Lichtes (das nachfolgende Einschalten bewirkt im Allgemeinen keine negativen Veränderungen des Umgebungszustandes) hintereinander ausführbar ist, ist der doppelte Auftrag einen bestimmten Film aufzunehmen widersprüchlich (oder zumindest unnötig). Dieses Beispiel zeigt, dass auch äquivalente Ziele widersprüchlich sein können. Eine Konfliktlösungsstrategie, die geeignet ist, mögliche Konflikte in der Zuteilung von Ereignissen zu lösen, muss somit zusammengefasst die folgenden Anforderungen erfüllen:

- Kenntnis besitzen über die möglichen Interpretationen, die durch die unterlagerten Interpreterkomponenten aus publizierten Ereignissen bzw. Ereignisketten interpretiert werden
- Kenntnis haben über Paare nicht kompatibler Ziele
- die Möglichkeit haben, Interpreterkomponenten das Ausführen ihrer Interpretation zu erlauben und zu blockieren. Konkurrenzen werden nach dem Prinzip des

longest-parse gelöst. Im Falle von Nicht-Kompatibilität von Zielen erhält diejenige Komponente die Erlaubnis zur Ausführung der Interpretation, die die längere Ereigniskette berücksichtigt[56].

Die Vorgehensweise der Konfliktlösungsstrategie zur Zuteilung von atomaren Ereignissen sei im Folgenden beschrieben. Seien die Mengen

$$Ziele := \{A, B, C, D, ...\} \tag{1}$$

und

$$N - Ziele := \{(X, Y) | X \in Ziele, Y \in Ziele\} \tag{2}$$

die von den Interpreterkomponenten interpretierbaren Ziele und der miteinander nicht-kompatiblen Zielpaare. Seien nun als Beispiel drei Transducer gegeben, die auf die folgenden Ereignisketten warten:

- P1 wartet auf die Ereignissequenz $\{a, b, c\}$ um Ziel A zu interpretieren
- P2 wartet auf Ereignissequenz $\{a, b, c, d, e\}$ um Ziel B zu interpretieren
- P3 wartet auf Ereignissequenz $\{b, c, d, e\}$ um entsprechend Ziel C zu interpretieren.

Dabei sei $(A, B) \in N - Ziele$ und somit die gleichzeitige Interpretation der Ziele A und B nicht kompabitel. Gelangt nun das Ereignis *a* in den Kanal, so werden die UtilityValue-Methoden der Transducer P1 bis P3 wie folgt evaluiert:

```
P1-UV : (A, wait{a,b,c}, lock a, t1)
P2-UV : (B, wait{a,b,c,d,e}, lock a, t2)
P3-UV : false
```

Interpreterkomponente P1 gibt an, dass es die Ereigniskette $\{a, b, c\}$ zum Ziel A interpretieren möchte. Zu diesem Zweck möchte sie das Ereignis *a* für die Zeitdauer t_1 für sich reservieren lassen. Analog die Komponente P2. Die Komponente P3 ist an Ereignis *a* nicht interessiert. Deren UtilityValue-Methode wird erwartungsgemäß zu *false* evaluiert. Die Zeitangaben entsprechen den Zeiten, die die Interpreterkomponenten auf das jeweils nächste Ereignis in der Sequenzkette warten. Nach Ausführung der Konfliktlösungsstrategie sendet der Kanal die folgenden Events an die Komponenten P1 und P2:

```
P1 erhält: (a, locked t2)
P2 erhält: (a, locked t1)
```

Die Transducer bekommen jeweils das Ereignis, das sie für sich reservieren wollten, mitgeteilt. Zugleich bekommen sie eine Zeit mitgeteilt, bis zu der das Ereignis blockiert ist und nicht verwendet werden darf. Diese Zeit ist die Maximalzeit der Zeitangaben der

[56] Das Prinzip des *longest-parse* oder *longest-match* ist hinreichend bekannt in verschiedenen Anwendungsfeldern. In [6] gibt Steven Abney einen Überblick über verschiedene Methoden des Parsens. Auch in Compilern für Programmiersprachen (z. B: in Haskell [196]) wird die Methode des longest-parse angewendet.

UtilityValue-Funktionen der jeweils anderen Transducer, deren Ziele in Nicht-Kompatibilität zueinander stehen. Zugleich ist davon auszugehen, dass die Transducer die interne Verwaltung der Sequenzketten nach den eigenen Zeitangaben ausrichten. Somit wird Transducer P1 im internen Speicher das Ereignis *a* löschen, sobald die Zeitdauer t_1 verstrichen ist. Kommen nun nach dem Ereignis *b* (hier passiert noch nichts, da keine Komponente ihr Sequenzkette vervollständigen konnte) das Ereignis *c* in den Kanal, so werden die UtilityValue-Funktionen der Interpreterkomponenten P1, P2 und P3 wie folgt evaluiert:

```
P1-UV : (A, execute{a,b,c}, lock c)
P2-UV : (B, wait{a,b,c,d,e}, lock c, t2)
P3-UV : (C, wait{b,c,d,e}, look c, t3)
```

Die Komponente P1 gibt also an, dass sie die Ereignissequenz {a,b,c} abgeschlossen hat, und bereit ist das Ziel A zu interpretieren. Die Konfliktlösungsstrategie erkennt jedoch, dass Ziel A nicht kompatibel zu Ziel B ist. Die Sequenzkette auf die Komponente P2 zur Interpretation des Zieles B wartet, ist jedoch länger als diejenige von Komponente P1. Gemäß der angewendeten *longest-parse*-Regel ist diese Interpretation genauer. Daher wird P1 bis auf weiteres blockiert. Nach Ausführen der Konfliktlösungsstrategie sendet der Kanal daher die folgenden Ereignisse an die beteiligten Komponenten:

```
P1 erhält: (c, locked t2)
P2 erhält: (c)
P3 erhält: (c)
```

Da Komponente P3 ein Ziel interpretieren möchte, dass kompatibel mit den Zielen von P1 und P2 ist ($(A, C) \notin N - Ziele$ und $(B, C) \notin N - Ziele$) kann hier auf die Mitgabe eines zeitbeschränkenden Wertes verzichtet werden. Ebenso bei Komponente P2. Da P2 gemäß der *longest-parse*-Regel die Konkurrenz gewinnen würde, Komponente P1 aber nicht mehr auf ein zusätzliches Ereignis wartet, gibt es auch hier keinen zeitlimitierenden Faktor mehr. An dieser Stelle sind drei Fälle zu unterscheiden:

1. Ereignis *d* tritt innerhalb der Zeit t_2 ein
2. Ein anderes Ereignis *m* tritt innerhalb der Zeit t_2 ein
3. Ereignis *d* tritt nicht innerhalb der Zeit t_2 ein bzw. nach dieser Zeit

Fall 3 wird intern in den Transducern bearbeitet. Komponente P1 weiß, dass das Ereignis *c* für die Dauer t_2 blockiert ist. Geschieht bis dahin nichts, führt P1 somit seine Interpretation der Sequenz aus. Intern wird die Komponente P2 ihre Sequenzkette löschen, da auch hier die Zeitdauer t_2 überschritten ist. Tritt ein anderes Ereignis als *c* ein (Fall 2) so werden die UtilityValue-Funktionen aller Interpreterkomponenten wiederum evaluiert (Anmerkung: Komponente P1 *weiss* nicht, auf welches Ereignis die Konkurrenzkomponente P2 wartet. Somit kann sie hier intern keine Entscheidungen treffen). Sei *m* das nachfolgende Ereignis, so dass:

```
P1-UV : (A, execute{a,b,c}, false)
P2-UV : false
P3-UV : false
```

Die UtilityValue-Funktion der Komponente P1 ergibt *false* für das betreffende Ereignis *m*. Jedoch wirft P1 auf, dass es vor kurzem eine Ereignissequenz {a,b,c} gegeben hat, die zum Ziel A interpretiert werden soll. Die Komponenten P2 und P3 haben ebenso kein Interesse an Ereignis *m*. Nebenbei weiss die kanal-eigene Konfliktlösungsstrategie durch die Vorgehensweise, dass es eine erfolgreich abgeschlossene Ereigniskette a,b,c gegeben hat. Sie weiss ebenso, dass es einen „blockierenden“ Transducer gegeben haben muss, der für eine gewisse Zeit t auf ein zusätzliches Ereignis warten wollte. Dadurch dass alle *false* abgegeben haben, weiss die Konfliktlösungsstrategie, dass eine solche längere Ereigniskette nicht zustande gekommen ist. Die Ausführung von *A* kann der Komponente P1 somit freigegeben werden. Der Kanal sendet somit das folgende Ereignis an P1:

```
P1 erhält: (execute{a,b,c})
```

Im Falle 1 gelangt das Ereignis *d*, auf das Komponente P2 (und auch P3) warten, rechtzeitig (d.h. innerhalb des Zeitraumes t_2) in den entsprechenden Kanal. Die Evaluierung der UtilityValue-Funktionen ergibt somit die folgenden Ergebnisse:

```
P1-UV : (A, execute{a,b,c}, false)
P2-UV : (B, wait{a,b,c,d,e}, lock d, t2)
P3-UV : (C, wait{b,c,d,e}, look d, t3)
```

Die Konfliktlösungsstrategie erkennt, dass Komponente P1 über eine vollständige Ereignissequenz verfügt, deren Interpretation jedoch zu der Interpretation von P2 nicht kompatibel ist. Aufgrund der potentiell besseren Interpretation von P2 muss P1 noch weiter blockiert werden. Die Kanal sendet aufgrund dessen die folgenden Ereignisse zu den Transducern:

```
P1 erhält: (block{a,b,c}, locked t2)
P2 erhält: (d)
P3 erhält: (d)
```

Gelangt nun innerhalb des Zeitraumes t_2 das Ereignis *e* in den Kanal (andernfalls Fall 2 oder 3) entspricht der Auswertung der UtilityValue-Methoden das Folgende:

```
P1-UV : (A, execute{a,b,c}, false)
P2-UV : (B, execute{a,b,c,d,e}, lock e)
P3-UV : (C, execute{b,c,d,e}, lock e)
```

Aufgrund von $(A, B) \in N - Ziele$ und ($(A, C) \notin N - Ziele$ und $(B, C) \notin N - Ziele$) und $size\,\{a, b, c, d, e\} > size\,\{a, b, c\}$ entscheidet die kanal-eigene Konfliktlösungsstrategie, dass die Komponenten P2 und P3 ihre jeweiligen Sequenzen interpretieren dürfen. Komponente P1 wird das Ausführen jedoch untersagt. Folgende Anweisungen schickt der Kanal an die beteiligten Komponenten:

```
P1 erhält: (abort{a,b,c})
P2 erhält: (execute{a,b,c,d,e})
P3 erhält: (execute{b,c,d,e})
```

Die Eigenschaften der Konfliktlösungsstrategie für die Zuteilung von Ereignissen im Falle konkurrierender Ereignisverbraucher in einer Zusammenfassung:

- jede Komponente, die sich für ein Ereignis interessiert, kann dieses Ereignis für eine gewisse Zeit für sich *blockieren*
- mit dem *Blocken eines Ereignisses* wird eine Funktion mitgeliefert, die Auskunft über das interpretierte Ziel gibt
- die Kanalstrategie bewertet die Ziele auf *Kompatibilität* und gibt
  - im Falle *kompatibler Ziele* die Interpretation frei
  - im Falle *nicht kompatibler Ziele* die Interpretation an diejenige Komponente, die die *längste Sequenz an Ereignissen* interpretiert
- eine Komponente, die ihre Sequenz nicht fertig stellen kann, gibt das Blocking auf.

Diskussion und Sonderfälle: Für ein detaillieres Verständnis der oben besprochenen Strategie seien die folgenden Fälle diskutiert:

1. Ein Transducer interpretiert eine sehr kurze Ereigniskette zu einem nicht-kompatiblen Ziel: In einem Beispiel seien zwei Transducer gegeben, für die gilt:

- P1 wartet auf die Ereignissequenz {c} um Ziel A zu interpretieren
- P2 wartet auf Ereignissequenz {a,b,c,d,e} um Ziel B zu interpretieren

Sobald – nach dem Geschehen der Ereignisse *a* und *b* – das Ereignis *c* von dem Kanal zu verarbeiten ist, evaluieren die UtilityValue-Funktionen der Interpreterkomponenten P1 und P2 wie folgt:

```
P1-UV : (A, execute{c}, lock c)
P2-UV : (B, wait{a,b,c,d,e}, lock c)
```

Wenn in diesem Fall gelten sollte, dass $(A, B) \in N - Ziele$, so blockiert die Konfliktlösungsstrategie die Komponente P1 wie oben beschrieben. Im Fall, dass $(A, B) \notin N - Ziele$ kann Komponente P1 die Interpretation ausführen.

2a. Zwei nicht-kompatible Ereignisketten sind in der Länge sehr verschieden: Seien in einem Beispiel zwei Transducer gegeben, für die gelte:

- P1 wartet auf die Ereignissequenz {a,b,c} um Ziel A zu interpretieren
- P2 wartet auf Ereignissequenz {c,d,e,f,g,h,i,j,k,l,m} um Ziel B zu interpretieren

Falls auch hier die Ziele A und B nicht kompatibel zueinander sind (also $(A, B) \in N - Ziele$), so muss P1 von der Konfliktlösungsstrategie in seiner Ausführung blockiert werden, bis die Ereignissequenz von P2 entweder abgeschlossen oder abgebrochen wurde. Im Falle des Abbruchs kann P1 seine Interpretation ausführen. Dies geschieht in exakt der Weise wie oben beschrieben. So ein Fall zeigt – da die Ereigniskette auf die P2 wartet sehr lang und die Überschneidung mit der Ereigniskette von P1 sehr gering ist –, dass die Definition der Kompatibilität von Zielen sehr sensitiv zu behandeln ist. Wäre dies ein realer Anwendungsfall würde – im Falle des Sequenzabbruches der Komponente P2 – die Ausführung von P1 zu einem sehr viel späteren Zeitpunkt als das letzte Ereignis (hier Ereignis *c*) erfolgen.

2b. Zwei nicht-kompatible Ereignisketten sind unterschiedlich lang: Sei das geschilderte Beispiel oben ein wenig verändert, so dass:

- P1 wartet auf die Ereignissequenz{a,b,c,d,e,f,g} um Ziel A zu interpretieren
- P2 wartet auf Ereignissequenz{g,h,i} um Ziel B zu interpretieren

In diesem Fall hat Komponente P1 seine (lange) Sequenz gerade beendet, als P2 beginnt das Ereignis *g* für sich zu blocken. Falls auch hier die Ziele A und B nicht kompatibel zueinander sind (also $(A, B) \in N - Ziele$), kann von der Konfliktlösungsstrategie sofort entschieden werden, dass gemäß der Regel des *longest-parse* der Komponente P1 die Interpretation erlaubt werden kann. Auch im Falle, dass P2 seine Ereigniskette vervollständigen könnte, wird sie diese nicht ausführen dürfen. Die Kanalstrategie sendet somit die folgenden Ereignisse an die Komponenten P1 und P2:

```
P1 erhält: (execute{a,b,c,d,e,f,g})
P2 erhält: (abort{a})
```

3. Ereignisketten sind exakt gleich lang aber nicht kompatibel: In einem weiteren Beispiel seien zwei Transducer gegeben, für die folgendes gilt:

- P1 wartet auf die Ereignissequenz {a,b,c} um Ziel A zu interpretieren
- P2 wartet auf Ereignissequenz {a,b,c} um Ziel B zu interpretieren

Im Falle dass die Ziele A und B nicht kompatibel zueinander sind ($(A, B) \in N - Ziele$) kann die *longest-parse*-Regel hier von der Kanalstrategie nicht angewendet werden, da im Falle gleich langer Ereignisketten keine Entscheidung getroffen werden kann. In diesem Falle ist die Komponente, der die Interpretation der betroffenen Ereignissequenz erlaubt wird, per Losverfahren zu ermitteln. Die Kanalstrategie wendet hier somit das Zufallsverfahren an. Dieser Fall ist besonders in den Fällen bedeutsam, in dem zwei unterschiedliche Interpreterkomponenten dieselbe Ereigniskette zum gleichen Ziel interpretieren möchten. In vielen Fällen kann es jedoch geboten sein, die doppelte bzw. mehrfache Ausführung desselben Zieles zu verhindern (z. B. bei Aufnahmen von Filmen oder dem Präsentieren von Vorträgen). In dem Fall ist $(A, A) \in N - Ziele$ zu definieren.

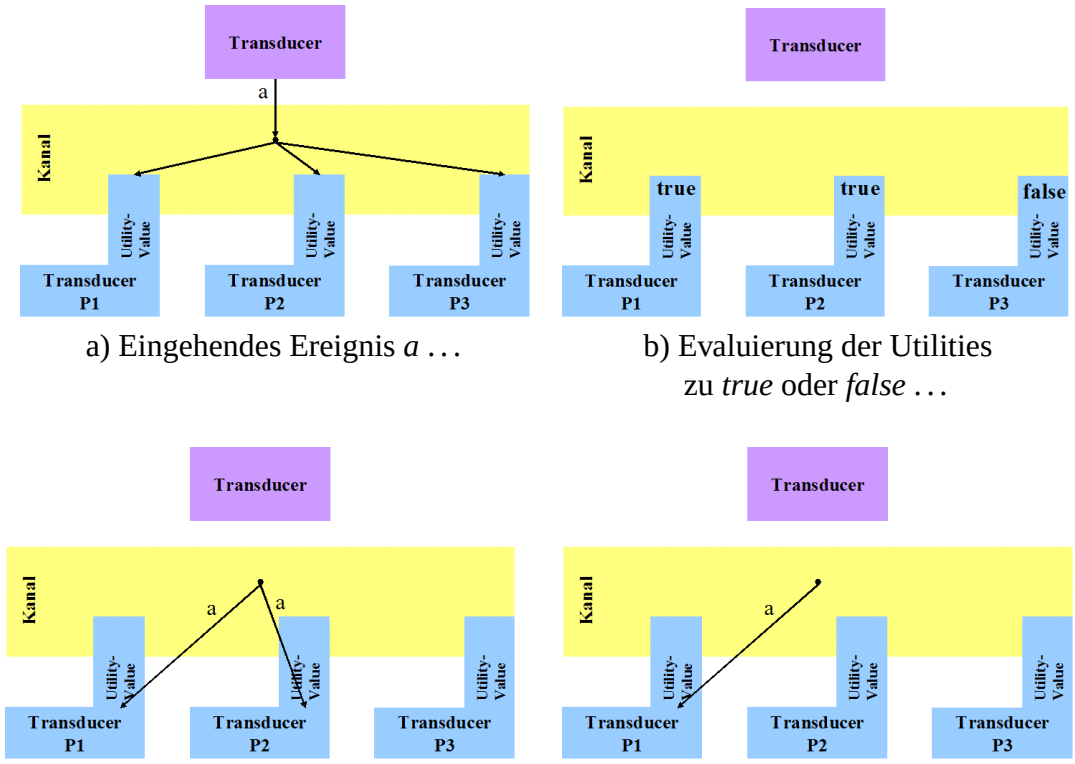
4. Nicht-kompatible Interpretationen bei gleich langen Ereignisketten mit ungleichen Elementen: Seien in einem letzten Fall zwei Komponenten gegeben, für die gelte:

- P1 wartet auf die Ereignissequenz {a,b,c} um Ziel A zu interpretieren
- P2 wartet auf Ereignissequenz {c,d,e} um Ziel B zu interpretieren

Dieser Fall ist eine Kombination der Sonderfälle 2 und 3. Hier wird genauso verfahren, wie es in der Besprechung dieser Fälle beschrieben ist. Komponente P1 wird so lange in der Interpretation seiner Ereigniskette blockiert, bis P2 seine Ereigniskette ebenfalls erfüllt bzw. abgebrochen hat. Seien beide Ereignisketten erfüllt und es gelte die Nicht-Kompatibilität der Interpretationen ($(A, B) \in N - Ziele$), so muss hier aufgrund der gleichen Länge der Ereignisketten wiederum das kanal-eigene Losverfahren die Entscheidung treffen. Die Gewinnerkomponente des Losverfahren erhält die *execute*-Nachricht, die jeweils unterlegenen Komponenten die *abort*-Nachricht.

Diskussion: Bei Analyse der verteilten Implementation und der verteilten Ausführung der Konfliktlösungsstrategie (siehe Kapitel 5.2), bei der bei jedem Ereignis von Neuem in einer Kanalgruppe entschieden wird, welcher SODAPOPD(AEMON) verantwortlich für die Ausführung der kanal-eigenen Konfliktlösungsstrategie ist, ist implizit definiert, dass Kanäle kein Langzeitgedächtnis haben müssen. Es ist offensichtlich, dass die verschiedenen SODAPOPD-Instanzen über die Ausführungen vergangener Konfliktlösungsstrategien über keinerlei Informationen verfügen. Kanalstrategien haben somit nur ein Kurzzeitgedächtnis über das gerade aktuelle Ereignis und die momentan evaluierten Werte der UtilityValue-Funktionen der am Kanal angemeldeten Komponenten. Der hier spezifizierte Ablauf der Konfliktlösungsstrategie zur Zuteilung von Ereignissen im Falle konkurrierender Komponenten basiert auf der Annahme des fehlenden Langzeitgedächtnisses. Es ist der Kanalstrategie jederzeit möglich auf kürzlich vergangene Sequenzketten zu schließen, dadurch dass die Transducer in ihren UtilityValue-Funktionen die von ihnen gewünschte zu interpretierende Sequenzkette angeben. Generell ist eine korrekte Implementation der beteiligten Transducer vorauszusetzen. Es muss davon ausgegangen werden, dass kein Transducer Ereignisse angibt, die gar nicht geschehen sind. Auch muss vorausgesetzt werden, dass die Zeiten t, die die Blockierzeit eines Ereignisses angeben, nicht zu hoch gewählt werden (im Ausnahmefall ist bei zu hohen Zeitangaben das Blockieren jedes Ensembles vorstellbar). Zuletzt liegt es in der Verantwortung jedes Transducers eine Sequenzkette zu einem vernünftigen Zeitpunkt abzubrechen, wenn diese nicht fortgesetzt oder beendet werden kann und in eigene Sequenzketten nicht einbaubare Ereignisse dahingehend zu interpretieren, dass begonnene Ereignisketten abzubrechen sind.

Die hier spezifizierte Konfliktlösungsstrategie weist für den Einsatz im prototypischen Umfeld in dem das dynamische Zusammenspiel von Komponenten experimentell evaluiert werden soll zwei Nachteile auf. Der erste Nachteil ist der Zwang zur Definition der auf diesem Kanal diskutierten Ziele und Interpretationen und die Definition der hierbei auftretenden nicht-kompatiblen Zielpaare. Der zweite Nachteil ist der erhöhte Aufwand



a) Eingehendes Ereignis *a* ...

b) Evaluierung der Utilities zu *true* oder *false* ...

c) Zuteilung mittels *all*-Strategie.

d) Zuteilung mittels der *random*-Strategie.

Abbildung 75: Illustration der Alternativstrategien der Konfliktlösungsstrategie für die Zuteilung von Ereignissen im Falle konkurrierender Interpreter-Komponenten. Abbildung c) illustriert die Zuteilungsweise der *all*-Strategie, Abbildung d) die Zuteilungsweise der *random*-Strategie.

in der Implementation der Interpreterkomponenten. Im Vergleich mit dem in Kapitel 5.1.5 beschriebenen Beispiel muss ein Transducer nun auch eine interne Logik zur Verwaltung der Ereignisketten und zur Interpretation der *abort*-, *block*- und *execute*-Befehle realisieren. In vielen Szenarios wird wohl ein Vergleich dieses Aufwandes mit dem Nutzen für die realisierte Anwendung gezogen werden müssen. Wenn die Nachteile aufgrund der Interpretation nicht-kompatibler Ziele als vernachlässigbar betrachtet werden können (bzw. falls die Menge der nicht-kompatiblen Ziele ohnehin Null ist, $N - Ziele = \varnothing$) können die Alternativstrategien des folgenden Abschnittes, die die hier spezifizierte Konfliktlösungsstrategie in einer ersten Näherung ersetzen können, eingesetzt werden.

Alternativstrategie all: Sollten bei der Definition eines Ensembles ausschließlich zueinander kompatible Ziele zu erwarten sein, bzw. die Folgen der Ausführung von nicht-kompatiblen Zielen gering sein, so kann die Konfliktlösungsstrategie zur Zuteilung von Ereignissen im Falle konkurrierender Komponenten in einer ersten Näherung durch die *all*-Strategie ersetzt werden (siehe Abbildung 75). Hier werden die UtilityValue-Funk-

tionen der konnektierten Transducer nur dahingehend evaluiert, ob der Transducer das Event verarbeiten will oder nicht (*true* oder *false*, siehe Abbildung 75 b)). Alle Transducer, deren UtilityValue-Funktion zu *true* evaluiert wurden, erhalten daraufhin das Ereignis (Abbildung 75 c)). Diese Strategie ist eine hinreichende Näherung der oben definierten Konfliktlösungsstrategie. Es wird lediglich auf die Auswertung der Ziele und damit auf die Anwendung einer Strategie zum Auflösen konkurrierender Ziele verzichtet. Die *all*-Strategie weist weitestgehende Ähnlichkeit mit bekannten *publish-subscribe*-Mechanismen auf, wie sie von Webservices oder Agenten-Kommunikationsplattformen bekannt sind. Ein wesentlicher Unterschied zu den dort eingesetzten Verfahren ist offensichtlich. Transducer tragen sich nicht für eine unbestimmte Dauer an einer zentralen Stelle für den Erhalt bestimmter Ereignisse ein, sondern sind mittels der Abfrage ihrer UtilityValue-Funktionen dynamisch eingebunden. Ein Ereignis wird immer mittels der transducer-eigenen UtilityValue-Funktion evaluiert.

Alternativstrategie random: Sollten konkurrierende Interpreterkomponenten in einem Ensemble zu erwarten sein, die zu interpretierenden Ereignisketten jedoch sehr kurz (Interpreterkomponenten als *One-Recognizer*) so kann zur Vermeidung von Konflikten die *random*-Strategie eingesetzt werden. Das Implementierungsbeispiel in Kapitel 5.1.5 basiert auf dem Einsatz dieser Strategiealternative. Analog der Alternativstrategie *all* werden in dieser ersten Näherung die UtilityValue-Funktionen der konnektierten Transducer ebenso ausschließlich dahingehend evaluiert, ob der Transducer das Event verarbeiten will oder nicht. Die Strategie entscheidet dann per Losverfahren, welcher der Transducer, deren UtilityValue-Funktion zu *true* evaluiert wurde, das Ereignis zur weiteren Verarbeitung erhält (Abbildung 75 d)).

Alternativstrategie fastest: Diese Alternativstrategie stellt ebenso eine Alternative zur *random*-Strategie dar. Es erhält diejenige Komponente ein Ereignis zur weiteren Verarbeitung, deren UtilityValue-Funktion am schnellsten zu *true* evaluiert wurde.

6.2.2 Kanalstrategie zur Auswahl unter konkurrierenden Komponenten

Im Falle, dass sich mehrere Komponenten um die Ausführung einer Aufgabe bewerben, muss der Kanal diese Aufgabe den am besten geeigneten Transducer zuweisen können. Ein einfaches Beispiel sei die Ausgabe von Musikdateien, worauf sich zwei verschiedene Wiedergabegeräte bewerben. Unter der Voraussetzung, dass eine solche Art der Auswahlstrategie in vielen unterschiedlichen Szenarios unter prototypischen Bedingungen funktionieren muss, so ist hier eine Konfliktlösungsstrategie zum Auflösen eines Konfliktes nötig, die sowohl ontologie-unabhängig ist als auch keine Vorkenntnisse über die Art der angeschlossenen möglichen Komponenten benötigt. Eine solche Auswahlstrategie kann somit keine spezifischen Kenntnisse über die Semantik der möglichen Konflikte besitzen, da sonst die Anwendung dieser Strategie in verschiedenen dynamischen Szenarios nicht mehr möglich sein wird. Die Aufgabe die sich hier stellt, ähnelt der Suche eines Laien nach einem Experten und somit auch etwas dem menschlichen Verhalten während eines

Beratungs- oder Verkaufsgespräches (siehe Abbildung 76). Man stelle sich eine Situation vor, in der ein Mensch eine Aufgabe zu vergeben hat, jedoch keinerlei Detailkenntnisse über die Art der Aufgabe und über die Art der Bewältigung hat. Vernünftig ist der Ansatz die Meinung verschiedener Experten einzuholen und sich auf Basis der Einzelmeinungen ein Gesamtbild der Lage zu machen[57]. Typische Meinungen von Experten (z. B. bei der Reparatur eines Autos) könnten lauten: *„Da muss Teil A ersetzt werden. Da bin ich mir ganz sicher, das habe ich schon oft gemacht“* oder *„Ich bin mir nicht ganz sicher, aber ich denke da muss Teil B ersetzt werden und Teil C mit Teil D vertauscht werden. Aber das habe ich noch nicht so oft gemacht“*.

Offensichtlich antworten Menschen, die an einer Ausschreibung für eine Auftragsvergabe teilnehmen[58] mit Angaben über die Art der Tätigkeit, die sie ausführen möchten, mit Angaben über ihre eigenen Erfahrungswerte und mit Angaben wie sicher sie sich sind, dass sie mit ihren Einschätzungen der aufkommenden Tätigkeiten richtig liegen. Etwas abstrakter formuliert geben Anbieter eine Diagnose darüber ab, welche Variablen (bzw. welche Aspekte) zur Erfüllung der gestellten Aufgabe berücksichtigt werden müssen. Zusätzlich geben sie noch eine Einschätzung der Wichtigkeit dieser Aspekte zusammen mit einer Angabe, wie sicher sie sich sind, dass diese Aspekte tatsächlich wichtig sind. Letztlich wird noch eine Einschätzung gegeben, wie sicher man sich ist, diese Aufgabe in Berücksichtigung der erwähnten Aspekte ausführen zu können.

Dieses (menschliche) Verhaltensmodell kann auf konkurrierende Komponenten formalisiert werden (siehe [137]) in dem jede Komponente einen Satz an Aspekten erhebt, die sie für die Bewältigung der Aufgabe für wichtig erachtet[59]. Gleichzeitig liefert jede Komponente für jeden Aspekt die folgenden Angaben:

- relative Wichtigkeit jedes Aspektes (im Folgenden *importance-value* genannt)
- einen Vertrauenswert, der das eigene Vertrauen definiert, dass der erwähnte Aspekt tatsächlich die behauptete Wichtigkeit besitzt (im Folgenden *confidence-value* genannt)
- und ein Vertrauenswert, der angibt, wie gut die Komponente glaubt, den erwähnten Aspekt bei der Ausführung der Aufgabe auch berücksichtigen zu können (im Folgenden *fidelity-value* genannt)

Die beiden letztgenannten Punkte sind offensichtlich, wenn man sich einen *ehrlichen* Anbieter vorstellt, der zwar weiß, dass ein bestimmter Aspekt von ungemeiner Wichtigkeit

[57] Als typische Fälle für Gespräche dieser Art können Besuche in einer Autowerkstatt herangezogen werden. Aber auch die Information eines Kunden zur Einholung einer Dienstleistung unter mehreren miteinander konkurrierenden Anbietern läuft nahezu exakt nach dem geschilderten Vorgang ab.

[58] Hierbei wird von dem Fall ausgegangen, dass derjenige Anbieter der sich an einer Ausschreibung beteiligt, diesen Auftrag auch bekommen will. Weiterhin muss vorausgesetzt werden, dass ein Anbieter nicht lügt oder absichtlich die Unwahrheit sagt. Absichtlich falsch geäußerte Meinungen erlauben es einem Beobachter natürlich nicht, sich ein objektives Gesamtbild zu verschaffen.

[59] Das im Folgenden beschriebene Verfahren ist unter dem Amtsaktenzeichen 10 2004 055 811.6-32 zur Patentierung eingereicht worden [138]. Es ist im Frühjahr 2007 als Patent genehmigt worden.

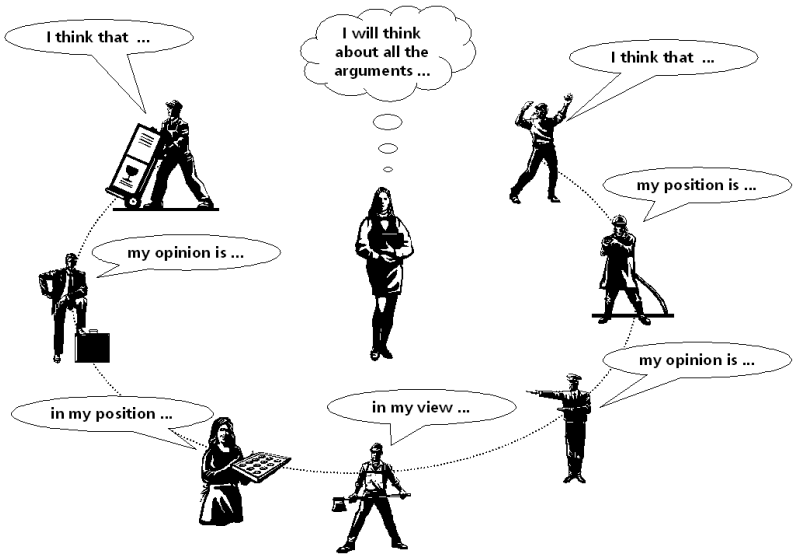


Abbildung 76: Menschliches Verhalten bei der Befragung von Experten. Bei hinreichend großer Unkenntnis über die Details einer Diskussionsdomäne werden alle Meinungen angehört, miteinander verglichen und daraufhin eine Entscheidung getroffen.

für die Ausführung einer Aufgabe ist, er selber aber auch weiß, dass er diesen nur unzureichend erfüllen könnte, wenn er die Aufgabe auszuführen hätte[60]. Es ist sinnvoll einige Einschränkungen zu definieren, um Komponenten davon abzuhalten, unrealistisch hohe Wichtigkeits- und Vertrauenswerte anzugeben. Dies ist unmittelbar einsichtig, da auch Menschen die von ihnen in einer Diskussion erwähnten Aspekte gewichten müssen und nicht alle unrealistisch hoch eingeschätzt werden können:

- die *importance-values* aufaddiert über alle Aspekte die eine Komponente aufbringt, müssen die Summe von 1 ergeben
- die *confidence-values* aufaddiert über alle Aspekte die eine Komponente aufbringt, müssen die Summe von 1 ergeben
- der *fidelity-value* von jedem Aspekt muss im Bereich von $[0, 1]$ liegen

Folgende mathematischen Schritte müssen dann im Folgenden vollzogen werden, um die Leistungsfähigkeit jeder einzelnen Komponente zu ermitteln:

1. zuerst werden alle erwähnten Aspekte in einem gemeinsamen Satz von Aspekten aufgezeichnet. Werden bestimmte Aspekte nur von einzelnen Komponenten erhoben, so werden die Werte für den *importance-value*, den *confidence-value* und den *fidelity-value* für den entsprechenden Aspekt bei den jeweils anderen Komponenten

[60] Hier könnte als Beispiel das Hausarzt-Facharzt-Modell dienen: Ein Hausarzt überweist einen Patienten an einen Facharzt, nachdem er die Wichtigkeit einer Untersuchung zwar diagnostiziert hat, sie jedoch nicht selber durchführen kann.

mit dem Wert 0 angenommen. Durch dieses Verfahren ist eine Einheitlichkeit und Vergleichbarkeit der Aspekte gegeben.

2. für jeden Aspekt wird eine *objektive Wichtigkeit* berechnet (die Summe aller subjektiven Wichtigkeiten multipliziert mit dem jeweiligen Vertrauenswert, und dividiert durch die Summe aller Vertrauenswerte) um die Vergleichbarkeit zu ermöglichen.

3. für jeden Aspekt wird danach das objektive Vertrauen definiert als die Summe aller subjektiven Vertrauenswerte. Dieser *effektive objektive Vertrauenswert* wird berechnet, indem für jeden Aspekt ein Skalierungsfaktor berechnet wird, der umgekehrt proportional zur objektiven Wichtigkeit ist.

Dieser Ansatz ist vergleichbar mit der Dempster-Shafer *Theory of Evidence* (siehe [48, 193]), der die Meinungen von verschiedenen Domänenexperten kombiniert und daraus eine gemeinsame Wahrscheinlichkeitsvermutung extrahiert. Leider eliminiert die Dempster-Shafer-Theorie alle Aspekte, die nicht von allen Experten erwähnt werden. Insofern wird hier ein anderer Ansatz gewählt, in dem alle erwähnten Aspekte berücksichtigt werden unter der Annahme, dass diejenige Komponente (derjenige menschliche Experte) am meisten geeignet sein kann, die die meisten Details berücksichtigen kann. Somit werden stark spezialisierte Komponenten automatisch eine höhere Anzahl an Aspekten benennen als weniger stark spezialisierte Komponenten. Nun lässt sich das Folgende definieren:

- G sei der Satz an Komponenten, die auf eine Ausschreibung zur Bewältigung einer Aufgabe geantwortet haben

- A sei der Satz an Aspekten, die von den Komponenten $g \in G$ erwähnt wurden

- für jedes $a \in A$ hat eine Komponente $g \in G$ einen *confidence-value* $c_{a_0}(g)$ und einen *importance-value* $w_{a_0}(g)$ definiert. (Anmerkung: $c_{a_0}(g)$ und $w_{a_0}(g)$ werden automatisch auf 0 gesetzt, wenn dieser Aspekt bei der jeweiligen Komponente nicht erwähnt wird.)

- für jedes $a \in A$ hat eine Komponente zusätzlich einen *fidelity-value* $f_a(g)$ anzugeben.

Das *objektive Vertrauen* c_a für einen Aspekt $a \in A$ wird nun definiert als:

$$c_a = \sum_{g \in G} c_a(g) \tag{3}$$

Und die *objektive Wichtigkeit* w_a eines Aspektes a als:

$$w_a = \sum_{g \in G} \frac{w_a(g) c_a(g)}{c_a} \tag{4}$$

Es kann nicht immer angenommen werden, dass die Summe über alle objektiven Wichtigkeiten $(\sum_{a \in A} w_a)$ gleich 1 ist. Üblicherweise wird diese Summe größer oder kleiner

als 1 sein. Daher kann keine relative Aussage über die Vergleichbarkeit der objektiven Wichtigkeiten getroffen werden und es ist eine Normierung erforderlich. Zur Normierung müssen die einzelnen Werte w_a aus diesem Grunde mit einem geeigneten Faktor gewichtet werden. Mit einfachen Umformungen und der Einführung eines Wichtungsfaktors x_a erhält man:

$$\sum_{a \in A} w_a = \sum_{a \in A} \frac{w_a c_a}{x_a + c_a} = 1 \tag{5}$$

Und mit der Annahme das für $\forall a, a^{'} \in A : x_a = x_{a'} = \hat{x}$ gilt:

$$\sum_{a \in A} w_a = \sum_{a \in A} \frac{w_a c_a}{\hat{x} + c_a} = 1 \tag{6}$$

Es ist somit die Nullstelle der Funktion

$$f(x) = (\sum_{a \in A} \frac{w_a c_a}{\hat{x} + c_a}) - 1 \tag{7}$$

gesucht. Laut dem Newtonschen Verfahren[61] zur Nullstellensuche einer Funktion [34] muss die Iterationsfolge:

$$x^{(m+1)} = x^{(m)} - \frac{f(x^{(m)})}{f'(x^{(m)})} \tag{8}$$

gebildet werden bis, die Differenz $x^{(m+1)} - x^{(m)} \leq \epsilon$ erfüllt ist. Mit:

- $\epsilon = 0,01$
- $f^{'} = -(\sum_{a \in A} \frac{w_a c_a}{(\hat{x}+c_a)^2})$
- und einem Startwert $\hat{x}_0$ mit $\hat{x}_0 + c_a \neq 0$ für $\forall a \in A$

wird die Berechnung bis zum Abbruch durchgeführt. Das Resultat dieser Berechnung ist der Wert $\hat{x}$ in (6) und die Möglichkeit die *normierte objektive Wichtigkeit* $\hat{w}_a$ für $\forall a \in A$ mittels

$$\hat{w}_a = \frac{w_a c_a}{\hat{x} + c_a} \tag{9}$$

zu berechnen.

Die *objektive Leistungsfähigkeit P* jeder Komponente, die sich für eine Aufgabe beworben hat, ist nun die Summe über das Produkt der *normierten objektiven Wichtigkeit* jedes Aspektes mit dem von der jeweiligen Komponente genannten *fidelity-value* $f_a(g)$ für diesen Aspekt:

$$P_g = \sum_{a \in A} \hat{w}_a * f_a(g) \tag{10}$$

Die am besten geeignete Komponente für eine Aufgabe wird nun folgendermaßen bestimmt:

[61] In [137] wird die Lösung dieser Aufgabenstellung mittels des *Regula-falsi*-Verfahrens vorgeschlagen. Die Anwendung dieses Verfahrens benötigt jedoch zwei Startwerte a und b mit den Bedingungen $f(a) > 0$ und $f(b) < 0$, während das hier verwendete Newton-Verfahren nur einen Startwert benötigt. Die Wahl für das Newton-Verfahren bedeutet keine Wertung der beiden zur Auswahl stehenden Näherungsverfahren *Regula falsi* und *Newton*.

1. diejenige Komponente mit dem höchsten Wert der *objektiven Leistungsfähigkeit P* bekommt die Aufgabe durch den Kanal zugestellt

2. haben mehrere Komponenten den *gleichen* Wert der objektiven Leistungsfähigkeit P wird die Komponente aufgrund fehlender andere objektiver Mechanismen aus diesen per Zufall bestimmt.

Syntax der UtilityValue-Funktionen: Um an einer Ausschreibung um Aufträge und Nachrichten an einem Kanal mitzuwirken, an dem die hier beschriebene Konfliktlösungsstrategie eingesetzt wird, müssen die UtilityValue-Funktionen bei der Evaluation eine bestimmte Syntax einhalten. Wird die UtilityValue-Funktion zu *false* evaluiert oder möchte die betroffene Komponente nicht an der Ausschreibung teilnehmen, so ist das Ergebnis der Evaluation der UtilityValue-Funktion ein einfaches „false“. Im anderen Fall gibt die Funktion eine Aspektliste zurück. Diese kann in zwei verschiedenen Syntaxarten formuliert werden. Die XML-Syntax schreibt sich wie folgt:

```
<aspect_list>
      <aspect name="aspect_name_1" importance="i1"
             confidence="c1" fidelity="f1"/>
      .....
</aspect_list>
```

In einem Beispiel aus Kapitel 7 würde dies z. B. für eine Medienwiedergabe-Applikation die folgende Aspektliste ergeben:

```
<aspect_list>
      <aspect name="screensize" importance="0.2"
             confidence="0.33" fidelity="1.0"/>
      <aspect name="interactivity" importance="0.5"
             confidence="0.34" fidelity="1.0"/>
      <aspect name="solution" importance="0.3"
             confidence="0.33" fidelity="0.8"/>
</aspect_list>
```

Hierbei ist zu beachten, dass sowohl die *importance*-Werte als auch die *confidence*-Werte in der Summe über alle aufgeworfenen Aspekte genau 1 ergeben. Für die *fidelity*-Werte gilt, dass der Wert 1 pro Aspekt nicht überschritten werden darf. Es ist auch möglich dies in einer Lisp-ähnlichen Syntax anzugeben. Die Beispielliste würde hierbei wie folgt aussehen:

```
(aspectlist (screensize i1 c1 f1)
      (interactivity i2 c2 f2)
      (solution i3 c3 f3))
```

Diskussion: Der hier vorgestellte Algorithmus wird durch einen zusätzlichen dynamischen Faktor ergänzt. Ähnlich dem menschlichen Verhalten bekommen Komponenten einen *Bonus*, je öfter sie sich an solchen Ausschreibungen unter konkurrierenden Komponenten beteiligen. Die Begründung hierfür ist Folgende: Angenommen eine Komponente

erachtet innerhalb einer solchen Ausschreibung Aspekte für wichtig, die jedoch nicht der „Mehrheitsmeinung“ entsprechen. Es ist aber vorstellbar, dass eine solche Komponente so intelligent implementiert ist, dass sie in der Lage auf viele verschiedene Ausschreibungen zu reagieren. Würde sie keinen Bonus für ihre oftmaligen Beteiligungen an Ausschreibungen bekommen, würde sie vom Kanal niemals einen Auftrag erhalten (außer im Sonderfalle, diese Komponente wäre die einzige Komponente die zum Ausschreibungszeitpunkt am betroffenen Kanal konnektiert wäre)[62]. Die Handhabung dieses Bonus in einer Ausschreibung erfolgt intuitiv. Die Meinungen der jeweiligen Komponente (die Liste der Aspekte) geht so oft in die eben spezifizierte Berechnung der Leistungsfähigkeit ein, wie die jeweilige Komponente an Ausschreibungen in der Vergangenheit beteiligt war. Im Falle des Einsatzes eines singulären SODAPOPD(AEMON)s (wie in Kapitel 5.1 übernimmt die aktive SODAPOPD-Instanz die Verwaltung der dynamischen Bonusregelung und übernimmt die Aspektlisten komponenten-abhängig so oft in die Berechnungen mit hinein, wie die betroffene Komponente sich in der Vergangenheit an Ausschreibungen beteiligt hat. Sind mehrere SODAPOPD-Instanzen in einem dynamischen Ensemble beteiligt (wie in Kapitel 5.2) so übermittelt die SODAPOPD(AEMON)s die UtilityValues ihrer Transducer mit der Angabe einer zusätzlichen Zahl, die angibt wie oft die jeweilige Aspektliste in die Berechnung miteinbezogen werden soll. Die hier spezifizierte Konfliktlösungsstrategie zur Auswahl der am besten geeigneten Komponente hatte eine wichtige Vorbedingung zum Einsatz in einer Entwicklungsumgebung für schnelles prototypische Entwicklung zu erfüllen: Um für möglichst viele Einsatzgebiete verwendbar zu sein, musste die Strategie über keinerlei Vorbedingungen bezüglich ontologischer Kenntnisse verfügen. Die definierte Strategie ist in der Lage ohne Vorwissen, nur auf Basis der von den Komponenten aufgeworfenen Variablen eine Entscheidung zu treffen. Somit sind im prototypischen Entwicklungsfall keine Re-Implementierungen der Konfliktlösungsstrategie vorzunehmen. Es sind lediglich Aspekte und die damit verbundenen *values* innerhalb dier UtilityValue-Methoden der Komponenten gemäß der vorgeschriebenen Syntax zu definieren. Der hier spezifizierte Algorithmus scheint gewisse Ähnlichkeiten mit anderen Matching-Verfahren, wie sie z. B. in WebServices bekannt sind, zu haben. Andere Verfahren vergleichen die gewünschte Funktion mit den angebotenen Funktionen (der potentiell zu beauftragenden Komponenten) und kalkulieren mittels Ähnlichkeitsverfahren einen Wert (gewöhnlich im Interval $[0, 1]$) der Aufschluss über die wahrscheinliche Leistungsfähigkeit der jeweiligen Komponenten geben kann (vgl. oder [151] oder [201] und Kapitel 3.10). Klein und König-Ries argumentieren für diese Art der Algorithmen in Hand von Anwendungsbeispielen (vgl. [141]), dass solche nur im Falle der Eindeutigkeit (Ähnlichkeit gleich Null oder 0) zu interpretierbaren Ergebnissen führen. Im Falle eines Zwischenwertes sei kein Dienstanbieter eindeutig zu identifizieren. Klein und König-Ries schlagen hierzu die Einbeziehung von Nutzerpräferenzen vor. Der in dieser Arbeit vorgeschlagene Ansatz basiert einzig auf den subjektiven Meinungen der beteiligten Komponenten und ist daher universeller zu verwenden. Eine wichtige Ergänzung zu der in diesem Abschnitt beschriebenen Herausforderung zu einer gegebenen Aufga-

[62] Dieses Vorgehen entspricht der menschlichen Erfahrungen, dass je öfters jemand seine Meinung äußert, die Wahrscheinlichkeit steigt, dass diese Meinung auch endlich einmal angehört wird.

be ein passendes Gerät aus einer Liste konkurrierender Geräte zu finden ist in Otto et al. [168] beschrieben. Hier wird die kongruente Fragestellung aufgeworfen, wie Konflikte gelöst werden können, wenn mehrere Benutzer zur selben Zeit Zugriff auf dasselbe Gerät nehmen. Otto et al. schlagen hier vier unterschiedliche Konfliktlösungsmöglichkeiten vor: (1) ausschließlich einem bestimmten Benutzer wird das Recht zur Interaktion gewährt oder (2) abwechselnd innerhalb bestimmter Zeiträume geht das Benutzungsrecht von einem Benutzer auf einen anderen über oder (3) die Anwendung der first-comes-first-serves-Methode oder (4) nur diejenige Interaktion ist möglich, der alle konkurrierenden Benutzer explizit zugestimmt haben. Besonders der vierte Ansatz hier scheint Ähnlichkeiten mit dem in diesem Abschnitt besprochenen Algorithmus aufzuweisen. Auch hier werden identische Angaben identifiziert und auf Basis einer solchen Mehrheitsmeinung die letztendliche Interaktionsentscheidung getroffen.

6.3 Anwendung auf die DYNAMITE-Szenarios

Die in den vorherigen Abschnitten vorgestellte Komponententopologie für Ambient-Intelligence-Szenarios soll anhand der in DYNAMITE definierten Anwendungsszenarios auf ihre Plausibilität überprüft werden. Hierbei wird Ebene für Ebene der Datenfluß anhand der definierten Konfliktlösungsstrategien nachvollzogen. Im Projekt DYNAMITE wurden Anwendungsszenarios im Bereich Hörsaal bzw. Besprechungsraum und im Wohnzimmer definiert (siehe Kapitel 2.1.13). Wenn die Geräte des *Hörsaals* miteinander vernetzt sind, bildet sich eine Komponententopologie aus, wie sie in Abbildung 77 illustriert ist. Ein solcher beispielhafter Hörsaal besteht aus einem Projektor für die Präsentation von Medien, mehreren Beleuchtungseinrichtungen, ein Mikrophon nebst zugehörigem Lautsprecher und einem Stehpult. Die Schattierungen in Kapitel 77 demonstrieren die physikalischen Grenzen der Komponenten, die jeweils bestimmten Geräten angehören. Das Mikrophon, die Beleuchtungseinrichtungen und der Projektor können dementsprechend auch als autonome Geräte verwendet werden. Das Stehpult verfügt mittels der eingebauten Drucksensoren nur über reine Sensorfunktionalitäten. Die geräteeigenen Interpreterkomponenten sind in der Lage die atomaren Ereignisse der Interaktionskomponenten ihres eigenen physikalischen Gerätes zu interpretieren und geeignet umzusetzen. Ein Tastendruck auf den Schalter für die Raumbeleuchtung steuert die Beleuchtungsinstallation des Raumes, ein Sprechen in das Mikrophon aktiviert die zum Mikrophon gehörigen eingebauten Lautsprecher. Jedes Gerät kann somit innerhalb des Ensembles seine Autonomie wahren. Im Szenario *Hörsaal* in DYNAMITE hat der Benutzer einen Vortrag in einem ihm unbekannten Hörsaal zu halten. Schließt dieser Benutzer nun seinen Laptop an, so integrieren sich die unterschiedlichen Komponenten des Laptops in die Kommunikationsstruktur des bereits vorhandenen Ensembles (siehe Abbildung 78). Sperrt nun der Benutzer seinen Laptop und geht nach vorne zum Stehpult, so werden eine Reihe von Ereignissen (mit (1) bezeichnete Punkte in Abbildung 78) ausgelöst, die die vom Laptop mitgebrachte Interpreterkomponente zum semantischen Ziel „Ablauf einer Präsentation im Raum von Rech-

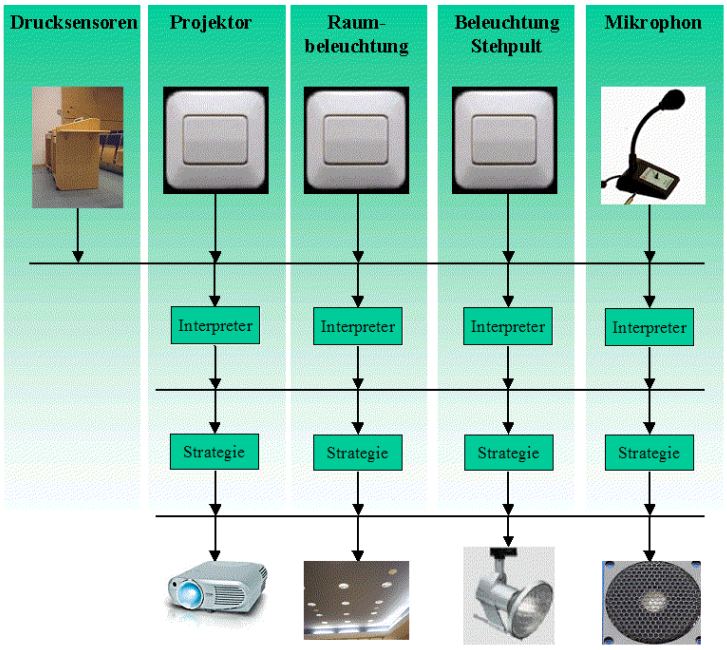


Abbildung 77: Der Komponentenaufbau auf Basis der erweiterten Topologie mit typischen Geräten und Installationen für einen Hörsaal.

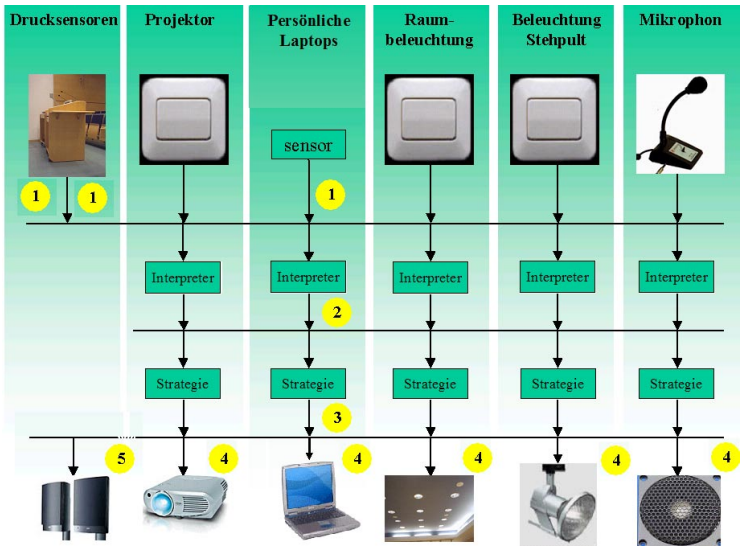


Abbildung 78: Das ursprüngliche Komponentenensemble im Hörsaal erweitert mit den Komponenten des hinzugefügten Laptops.

ner mit der IP-Nummer xzy“[63] interpretiert. Dieses Ziel wird auf den Zielkanal gegeben

[63]Die IP-Nummer xyz ist hierbei die dem Laptop vom jeweiligen Intranet vergebenen IP-Adresse. Diese kann eine Interpreterkomponente intern im Laptop erfragen. Sie kann jedoch auch als Ereignis in Punkt (1) in Abbildung 78 beim Aufstehen vom Laptop von einer eingebauten Sensorkomponente übermittelt werden.

und von der ebenfalls vom Laptop mitgebrachten Strategiekomponenten weiterverarbeitet (Punkt (3) in Abbildung 78). Diese Strategiekomponente verarbeitet dieses Ziel zu einer Reihe von Funktionsaufrufen:

- Anschalten des Projektors und Übergabe der IP-Adresse
- Anschalten der Lautsprecher zur Wiedergabe des Tons aus dem Mikrophon
- Steuerung der Raumbeleuchtung auf einen mittleren Wert
- Herunterdimmen der Beleuchtung an der Präsentationsfläche zur Erhöhung des Kontrastes
- Aktivieren des Präsentationsprogrammes auf dem Laptop und Starten des Präsentationsvorganges.

Jedes einzelne dieser Funktionsaufrufe wird auf den Funktionskanal publiziert und von den einzelnen vorhandenen Aktuatoren (Punkte (4) in Abbildung 78) ausgeführt. In Kapitel 8 wird dieses Anwendungsszenario detailliert spezifiziert und anhand weiterer Beispiele die Anwendung für prototypische Realisierungen demonstriert.

In einem nicht-statischen Anwendungsfall in einem Hörsaal lässt sich vorstellen, dass ein Hörsaal von Zeit zu Zeit sukzessive mit neuen Geräten ausgestattet wird. Abbildung 78 zeigt dies in Punkt (5) wenn die Raumausstattung mit neuen modernen Lautsprechern erweitert wird. Es sind jedoch immer noch die zum Mikrophon gehörenden Lautsprecher vorhanden – vielleicht sind diese ja auch im Stehpult fest eingebaut. Es wird nun aber von dem erweiterten Geräteensemble erwartet, dass es die neuen Lautsprecher in die folgenden Anwendungen miteinbezieht. Die alten Lautsprecher und die neuen Lautsprecher stehen somit in einem unmittelbaren Konflikt zueinander und konkurrieren miteinander um den Auftrag Töne und Sprache im Raum wiederzugeben. Gemäß der in Kapitel 6.2.2 beschrieben Strategie zur Auswahl unter konkurrierenden Komponenten geben die beiden Aktuatoren die von ihnen berücksichtigten Aspekte ab. Für die alten integrierten Lautsprecher mag dies die folgende Liste sein:

```
<aspect_list>
      <aspect name="loudness" importance="0.5"
             confidence="0.5" fidelity="0.7"/>
      <aspect name="highfidelity" importance="0.5"
             confidence="0.5" fidelity="0.8"/>
</aspect_list>
```

Die neuen Lautsprecher geben die folgende Aspektliste bei der Evaluierung ihrer UtilityValue-Funktionen ab:

```
<aspect_list>
      <aspect name="loudness" importance="0.5"
             confidence="0.5" fidelity="1.0"/>
      <aspect name="highfidelity" importance="0.3"
             confidence="0.4" fidelity="0.9"/>
      <aspect name="roomfilling" importance="0.2"
             confidence="0.1" fidelity="0.8"/>
</aspect_list>
```

Die alten Lautsprecher nennen die Aspekte *loudness* und *highfidelity*, die sie als gleich wichtig erachten. Das neue Paar Lautsprecher erwähnt zusätzlich den Aspekt der Raumfüllung (*roomfilling*). Entsprechend den Vorgaben der Strategie summieren sich jeweils die Werte über die *importance* und die Werte über die *confidence* in einer Aspektliste zu 1. Der *fidelity*-Wert eines Aspektes, der angibt inwieweit die jeweilige Komponenten in der Lage ist, diesen Aspekt auch bei der Ausführung des Auftrages erfüllen zu können, nimmt jeweils einen Wert zwischen 0 und 1 ein. Für die folgenden Berechnungen sei der Aspektname *loudness* dem Indexwert 0 zugeordnet, der Aspektname *highfidelity* dem Indexwert 1 und entsprechend der Indexwert 2 dem Aspektnamen *roomfilling*. Nach Gleichung 3 ergeben sich für die Werte des objektiven Vertrauens für die Aspekte die folgenden Werte:

```
C[0]: 1.0
C[1]: 0.9
C[2]: 0.1
```

Mit diesen Werten ergeben sich die folgenden Werte der objektiven Wichtigkeit aus Gleichung 4 (gerundet):

```
W[0]: 0.5000
W[1]: 0.4111
W[2]: 0.2000
```

Nach Ablauf des Newtonschen Verfahrens zur Nullstellensuche in Gleichung 7 ergibt sich $\hat{x} = 0.0494278$ und im Folgenden für die Werte der normierten objektiven Wichtigkeit in Gleichung 9 für jeden Aspekt:

```
V[0]: 0.476450
V[1]: 0.389708
V[2]: 0.133844
```

Dies wird nun verwendet um die potentielle Leistungsfähigkeit nach Gleichung 10 für die beiden Aktuatoren zu berechnen. Für das alte Lautsprecherpaar ergibt sich hier ein Wert von $P = 0.645282$ und für das neue Lautsprecherpaar ein Wert von $P = 0.934263$. Für das neue Lautsprecherpaar wird somit eine signifikant höhere potentielle Leistungsfähigkeit ermittelt und der Kanal wird somit den Auftrag an diesen Aktuator vergeben. In Kapitel 7 wird das DYNAMITE-Anwendungsbeispiel Wohnzimmer anhand eines komplexeren Szenarios beschrieben und diskutiert.

7 Anwendungen

Im Rahmen dieser Arbeit wurden drei unterschiedliche Projekte realisiert, die sich mit der Implementierung intelligenter Umgebungen beschäftigen. Dabei wurden in den Projekten unterschiedliche Schwerpunkte gesetzt. Im EMBASSI-Projekt wurde ein adaptiver Auswahlassistent spezifiziert und implementiert und in einer darauffolgenden Evaluation auf die Güte seiner Assistenz untersucht (siehe Kapitel 7.1). Bei dieser Anwendung handelt es sich um eine Interpreter- bzw. Strategiekomponente, wie sie in Kapitel 6 definiert ist. Im DYNAMITE-Projekt wurde ein Software-Demonstrator realisiert, der komplett auf der in dieser Arbeit spezifizierten verteilten Software-Infrastruktur aufbaut. Hier wurde die gesamte Topologie der Komponenten, beginnend von Eingabekomponenten, über Interpreterkomponenten, über Strategiekomponenten, bis hin zu den Aktuatoren realisiert und in einem dynamischen Umfeld erprobt (siehe Kapitel 7.2). Die in DYNAMITE realisierte Anwendung dient damit als proof-of-concept der in dieser Arbeit erarbeiteten Spezifikation. Zuletzt wurde in Zusammenarbeit mit der Volkswagen AG im *Reiseradar*-Projekt eine regelgestützte Interpreterkomponente implementiert, die basierend auf dynamischen Kontextdaten Hilfestellungen für den Benutzer im Sinne präferierter Orte in der Umgebung inferiert (siehe Kapitel 7.3). Die hier verwendete Regelmaschine basiert auf der im folgenden Kapitel (vgl. Kapitel 8.2) spezifizierten Interpreterkomponente, die sich für die Realisierung unterschiedlicher Szenarios einsetzen lässt.

7.1 Adaptiver Auswahlassistent in EMBASSI

EMBASSI [114] ist eines der sechs Leitprojekte zur Mensch-Technik-Interaktion des Bundesministerium für Bildung und Forschung (BMB+F) [54]. Vor allem in den Szenarios *Privathaushalt* und *Kraftfahrzeug* (vgl. Kapitel 2.1.1) spielt der Umgang mit Medien – Filmen und Musikstücken – eine wesentliche Rolle. Die meisten Interaktionen des Benutzers im Szenario Privathaushalt von EMBASSI handeln vom Umgang mit Medien. Per Spracheingabe und graphischen Benutzeroberflächen fordert der Benutzer das gerade aktuelle Fernsehprogramm an, und ebenso selbstverständlich nimmt er Zugriff auf Medienressourcen. Ebenso wird per Spracheingabe das Aufnehmen und die Wiedergabe von Filmen und Sendungen gesteuert. Ein wichtiges Ziel des EMBASSI-Projektes war es daher auch, dem Benutzer eine bestmögliche Unterstützung in der Auswahl von Filmen und Medien zu ermöglichen. Hierbei sollte aber nicht nur das sog. „Browsen“ durch eine Film- und Sendungsangebotsliste möglich sein, oder das Aufnehmen von bestimmten Sendungen ermöglicht werden (Bsp.: „Ich möchte den Tatort aufnehmen“, „Ich will die Tagesschau sehen“), sondern auch Szenarios mit persönlicher Assistenz. Hierbei soll dem Benutzer auf Fragen wie „Gibt es einen interessanten Film für mich?“, „Läuft etwas Schönes?“, oder „Meine Lieblingssendung heute Abend bitte aufnehmen!“ persönliche Assistenz im Sinne einer präferierten Liste oder einer selbstständigen Programmierung anhand eines Benutzerprofils geleistet werden. Der in dieser Arbeit entwickelte User-Assistant soll hierbei nicht nur dem Benutzer bekannte Fernsehsendungen empfehlen, sondern auch Vorhersagen treffen, ob andere Filme in das persönliche Präferenzprofil

des Benutzers passen würden. Der User-Assistant ist damit eine direkte Realisierung wichtiger EMBASSI-Szenarios, die auf Selbstlerneffekten („...hat sich sehr gut auf mich eingestellt...“ vgl. [62]) und aktiver Assistenz beruht. Zur Realisierung solcher Anwendungsfälle wurde mit der Konzeption und Implementierung des User-Assistant in EMBASSI hierbei das Konzept der adaptiven Assistenz verfolgt, d.h. das System ermittelt selbstständig aufgrund der Handlungen – ohne direkten Eingriff des Benutzers – ein Benutzerprofil, wertet dies aus und bietet auf Basis dieser Auswertungen aktive und passive Assistenz an. Passive Assistenz meint hier z. B. die Unterstützung in vom Benutzer initiierten Auswahlprozessen, aktive Assistenz das automatische Aufnehmen bzw. Erinnern an präferierte Sendungen („Lieblingssendungen“) oder die Bereitstellung von persönlichen Informationen für andere EMBASSI-Komponenten.

Mit dem Anspruch auf reine Adaptivität geht der User-Assistant in EMBASSI über bekannte technische Ansätze für die Empfehlung von Filmen hinaus. Im DIVA-System [163] muss der Benutzer aus vordefinierten Listen eigene Listen über präferierte und nicht-präferierte Filme bilden. Diese Daten gehen in das langfristige Profil ein und werden dann im Folgenden für die Empfehlung (und Nicht-Empfehlung) anderer Filme verwendet. Fordert der Benutzer hier eine Liste mit Empfehlungen an, so kann er hier zusätzlich sein sogenanntes kurzfristiges Profil definieren. In ihm gehen momentane Wünsche und Stimmungen als initiale Filterparameter ein. Der Decision-Theoretic Interactive Video Advisor reichert diese direkt vom Benutzer bezogenen Daten mit zusätzlichen Präferenzinformationen aus einer fallbasierten Datenbank an. Hierbei werden Vergleiche des aktiven Benutzerprofils mit bereits vorhandenen Fällen gezogen. Das DIVA-System verwendet hierbei zur Ermittlung der Distanz einen Markov-Algorithmus. Nachdem somit für den aktiven Benutzer und jeden in der Datenbank vorhandenen Fall die Distanzen ermittelt wurden, wird derjenige Fall mit der kürzesten Distanz zur aktuellen Ermittlung der zu empfehlenden Filme herangezogen. Das DIVA-System ist somit aufgrund der Einbeziehung von vielen Benutzerprofilen nicht für den Gebrauch im Privathaushalt zu benutzen, zumal es für jede einzelne Empfehlung den direkten Benutzereingriff zur Editierung des kurzfristigen Profils benötigt. Einen ziemlich ähnlichen Ansatz verfolgt Masthoff [155] mit der Untersuchung unterschiedlicher Strategien und deren Evaluation. Hier wird versucht auf Basis von Einzelprofilen eine Filmempfehlung für Gruppen zu geben, die z. B. zusammen vor einem Fernsehgerät sitzen. Die Vorbedingung hierbei ist, dass niemand aus der Gruppe mit der angebotenen Filmauswahl unzufrieden sein soll. Das Empfehlungssystem von Kurapati et al. [148] basiert auf der impliziten Sehhistorie des Benutzers, seinen explizit definierten Präferenzen und auf dem Feedback das der Benutzer den von ihm konsumierten Sendungen und Filmen gegeben hat. Andere Arbeiten beschäftigen sich mehr mit technischen Fragestellungen bzgl. Ausgabegeräten von Filmen und Fernsehsendungen bzw. mit den Fragestellungen von Netzwerkeffizienz und Netzwerkausnutzung auf Basis von Nutzerpräferenzen (vgl. Bougand et al. [32]). Kommerzielle Systeme wie TiVo[64] und ReplayTV[65] sind auch in der Lage proaktiv für den Benutzer die Programmierung von Sendungen vorzunehmen. Dabei verwaltet TiVo das persönliche Benutzer-

[64] TiVo Inc., http://www.tivo.com
[65] ReplayTV Inc., http://www.replaytv.com

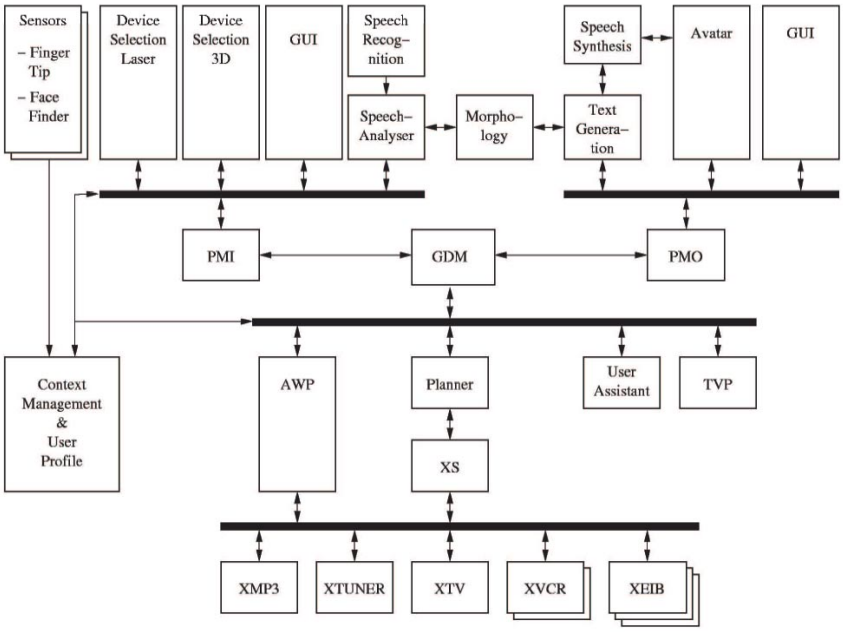


Abbildung 79: Die Komponentenarchitektur aus dem EMBASSI-Szenario Privathaushalt (aus [116]).

profil, das aus präferierten Sendungen besteht. Diese Sendungen wurden vom Benutzer entweder explizit oder per Feedback nach dem Konsumieren einer Sendung bestimmt. Bei ReplayTV gelangen nur die Sendungen in das Profil, die explizit ausgewählt wurden. Sowohl ReplayTV als auch TiVo geben keine Vorhersagen über „unbekannte“ Sendungen und Filme ab. Sie stellen somit eher Erinnerungssysteme als Empfehlungssysteme dar.

7.1.1 Architekturansatz und Realisierung

Abbildung 79 illustriert die Komponentenarchitektur im EMBASSI-Szenario Privathaushalt als Instanz der generischen EMBASSI-Komponententopologie (vgl. Abbildung 37 und [115]). Die horizontal eingezogenen Ebenen stellen Kanäle im Sinne der SODAPOP-Definition dar. Die Kommunikation der Komponenten untereinander wurde in EMBASSI jedoch auf Basis der Agentenkommunikationssprache KQML (vgl. Kapitel 3.10) mittels eines speziell zum Anfang des Projektes realisierten Kommunikationsrouters gesteuert[66] Der User-Assistant gliedert sich in der EMBASSI-Architektur auf der Assistenzebene ein und kommuniziert hier mit dem Context-Manager, den zu den Fernsehgeräten und Videorekordern gehörigen X-Komponenten und dem Electronic Program Guide. Die Fernsehgeräte und Videorekorder tragen hierbei ihren aktuellen Zustand, der vor allem aus der gegenwärtig eingestellten Kanalnummer besteht, in die Datenstruktur des Context-Managers ein. Der User-Assistant liest daraufhin diese Daten aus dem Context-Manager

[66] Der hier eingesetzte Router – EMBASSI-Server – ist bei Forkl und Hellenschmidt [83] detailliert beschrieben.

(für Details siehe Schirmer und Bach [188]) in seinen internen Speicher und befragt die Electronic-Program-Guide-Komponente nach den weiteren Programmdetails. Auf diese Art und Weise ist der User-Assistant in der Lage nach und nach eine detaillierte Sehhistorie von jedem Benutzer des Privathaushaltes anzulegen[67]. Das Sehprofil sowie die Konzepte zu den präferierten Sendungen, den präferierten Sendern und der präferierten Sendungsgenres sind Teil des szenarienübergreifenden Benutzerprofils in multimodalen Umgebungen, das während der Projektlaufzeit von EMBASSI definiert wurde (siehe Hellenschmidt et al. für Details [103]).

7.1.2 Adaptive Assistenz und Evaluation

Um den Benutzer bei der Auswahl aus einer (sehr) großen Menge an Optionen zu unterstützen, können unterschiedliche Arten an Assistenz in ihren Grundprinzipien unterschieden werden. Wandke [212] unterscheidet hier die *Angebotsassistenz*, die *Filterassistenz* und die *Beraterassistenz*. Bei der *Angebotsassistenz* wird der Benutzer über alle verfügbaren Optionen informiert und er kann selbstständig darin suchen und auswählen. Laut Wandke führt dies bei einer sehr großen Menge an Optionen – z. B. sehr großen Filmdatenbanken, Fernsehprogramm eines Abends von hunderten Fernsehsendern – zu einer immensen kognitiven Leistungsanforderung. Daher reduziert die *Filterassistenz* die Anzahl der Optionen. Dies kann laut Wandke auf der Basis von Benutzerprofilen (sog. adaptive Assistenz) geschehen oder auf Basis von Kriterien, die der Benutzer eigenständig definieren kann (adaptierbare Assistenz). Adaptive Assistenz nimmt hierbei im Generellen Rückgriff auf die Vergangenheit eines Benutzers, während die adaptierbare Assistenz dem Benutzer die volle Kontrolle über den aktuellen Vorgang lässt. Die *Beraterassistenz* reduziert die Anzahl der verfügbaren Optionen auf eins. Hier entscheidet der Benutzer, ob er den Beratervorschlag annehmen möchte oder nicht. Das Vorgehen basiert ebenso wie bei der Filterassistenz auf der Anwendung von adaptiven oder adaptierbaren Auswahlkriterien.

Zur Ermittlung der am besten geeigneten adaptiven Auswahlregel, die intern im User-Assistant zur Bewertung der vorhandenen Menge an Spielfilmen und Sendungen angewendet wird, wurde eine Benutzerevaluation in zwei aufeinanderfolgenden Schritten ausgeführt. Im ersten Schritt wurden Versuchspersonen auf Basis ihrer individuellen Sehhistorie mittels unterschiedlichen Auswahlregeln Listen an empfohlenen Filmen präsentiert. Diese Listen wurden von den Versuchspersonen bewertet. Im zweiten Schritt wurde die am höchsten bewertete adaptive Auswahlstrategie gegen die adaptierbare Assistenz evaluiert. Diese zweite Evaluationsstufe soll Aufschluss über die Akzeptanz von adaptiver Assistenz im Vergleich zur adaptierbarer Assistenz liefern.

Abbildung 80 illustriert die bei beiden Evaluationen verwendeten Komponentenarchitektur[68], die mit drei unterschiedlichen Komponentenebenen auf Basis der in Kapitel 5.2

[67] Dieses Vorgehen wurde im Detail auf dem projektübergreifenden MTI-Workshop bzgl. Nutzer- und Kontextadaptivität vorgestellt und diskutiert (siehe Hellenschmidt und Nitschke [102]).

[68] Bei den Evaluationen beim EMBASSI-Projektpartner Humboldt-Universität zu Berlin am Institut für Psychologie konnte selbstverständlich nicht der EMBASSI-Gesamtdemonstrator Privathaushalt verwendet werden.

beschriebenen SODAPOP-Implementation realisiert wurde. Sie besteht hierbei aus zwei Kanälen, die einerseits die graphischen Benutzeroberflächen (links in Abbildung 80) und die User-Assistant-Komponente und andererseits die User-Assistant-Komponente mit einer EPG-Komponente, die Zugriff auf eine Filmdatenbank besitzt, verbinden. Die beiden Eingabe-Komponenten senden hierbei Remote Procedure Calls ab, um den User-Assistant nach Filmempfehlungen zu fragen (Komponente links unten in Abbildung 80) und die globalen Parameter der Evaluation zu steuern (Komponente links oben in Abbildung 80). Der User-Assistant wiederum sendet Remote Procedure Calls ab, um das EPG nach Filmlisten mittels bestimmter Suchkritieren zu fragen. Abbildung 81 illustriert die graphischen Benutzeroberflächen.

Für die erste Evaluation wurden 10 Regeln festgelegt, mittels deren die Filme aus der EPG-Datenbank (ungefähr 700 verschiedene Filme) für den Benutzer bewertet werden sollten. Ein Film wird hierbei – neben seinem Titel – durch vier unterschiedliche Eigenschaften beschrieben. Es handelt sich hierbei um das Genre (z. B. Abenteuerfilm, Drama, oder Heimatfilm), das Entstehungsland, dem Entstehungsjahr und der Stimmung die ein Film im Wesentlichen vermittelt. Als wichtige Stimmungen für die Beschreibung eines Filmes haben sich in früheren Untersuchungen, die an der Humboldt-Universität zu Berlin stattgefunden haben, die Stimmungen lustig, spannend, tragisch und unterhaltend ergeben. Vor den Evaluationen wurde den jeweiligen Versuchspersonen jeweils eine Fernsehzeitschrift der vergangenen Wochen vorgelegt, in der sie diejenigen Filme ankreuzen sollten, die sie in diesen Wochen konsumiert haben. Mit den Daten zu den Filmen, inklusive Genre, Entstehungsjahr, Entstehungsland und der Stimmung wurde dann eine Datei erstellt, die von dem User-Assistant eingelesen werden konnte. Der User-Assistant war somit in der Lage, aus den Daten die meist präferierten Genres, Entstehungsjahre von Filmen, Entstehungsländer von Filmen und Stimmungen in jeweils unabhängigen Listen zu erstellen, sowie auch die meist präferierten Verknüpfungen (z. B. Heimatfilme aus Deutschland) zu inferieren. Die 10 Regeln zur adaptiven Auswahl von Filmen waren die Folgenden:

1. Variante 1: Anhand des Kriteriums Genre werden die Präferenzen aufgrund der am häufigsten besetzten Ausprägung ausgebildet. Das bedeutet, dass in dieser Einstellung der Benutzer diejenigen Filme angeboten bekommt, die seinem meist präferierten Genre entspricht.

2. Variante 2: Von den drei Kriterien (Genre, Entstehungsland und Entstehungsjahr) wird jeweils die am häufigsten besetzte Ausprägung berücksichtigt. Es werden somit Filme empfohlen, die dem am höchsten präferierten Genre, dem am höchsten präferierten Entstehungsland und dem am höchsten präferierten Entstehungsjahr entsprechen.

3. Variante 3: In drei Kriterien (Genre, Entstehungsland und Entstehungsjahr) wird nach tatsächlich vorhandenen Verknüpfungen gesucht. Der Unterschied zu Variante 2 besteht darin, dass hier Verknüpfungen berücksichtigt werden, die im Sehprofil auch tatsächlich bestanden haben.

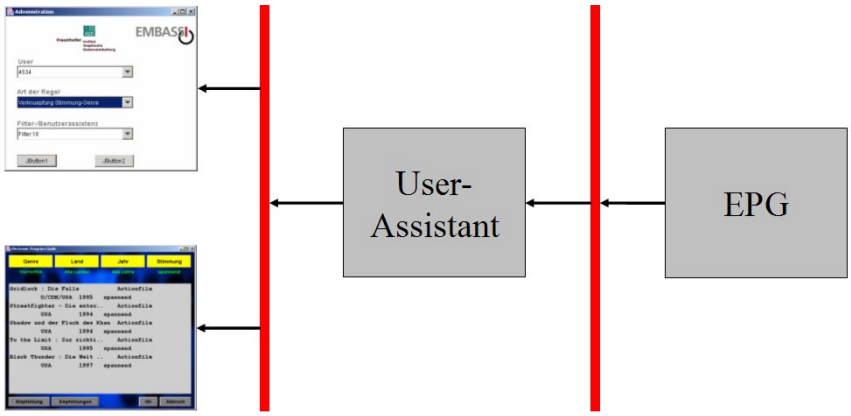


Abbildung 80: Komponententopologie zur Evaluierung der adaptiven Regeln des User-Assistant.

4. Variante 4: Mittels eines Algorithmus auf Basis einer Wahrscheinlichkeitsrechnung soll bestimmt werden, wie hoch die Wahrscheinlichkeit ist, dass ein Film dem Benutzer gefallen könnte. Der Algorithmus hierzu sei kurz beschrieben: Jeder Instanz von Genre, Entstehungsjahr, Entstehungsland und Stimmung lässt sich aufgrund der Rangfolge in den individuellen Präferenzlisten eine Wahrscheinlichkeit zuordnen. Diese berechnet sich mittels der Formel:

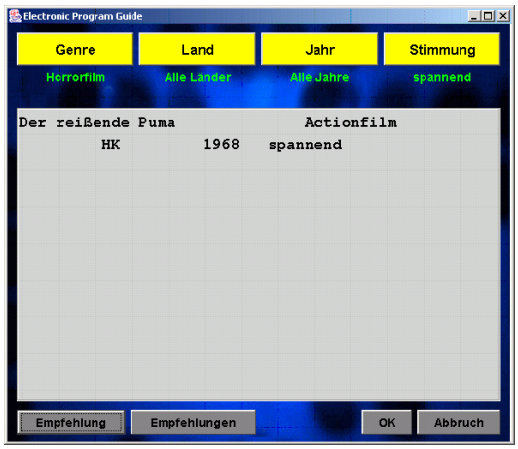
$$P_{Attribut} = 1 - [1/(\sum Attributliste)] * Attributstelle$$

Besteht die persönliche Liste der präferierten Genres eines Benutzers aus 5 unterschiedlichen Genres, so ergeben sich Wahrscheinlichkeiten von 1.0, 0.8, 0.6, 0.4 und 0.2. Für jeden Film der von der EPG-Komponente durch die Datenbank bereitgestellt wird, wird so mittels

$$P_{gesamt} = P_{Genre} * P_{Entstehungsland} * P_{Entstehungsjahr} * P_{Stimmung}$$

die Gesamtwahrscheinlichkeit berechnet. Besitzt ein Film eine Eigenschaft, die nicht in der präferierten Attributsliste vorkommt, so wird für diese Eigenschaft eine Wahrscheinlichkeit von 0.5 angenommen (z. B. wenn ein Film ein Genre besitzt, welches in der präferierten Genrelist nicht vorkommt). Die Filme werden dann durch den User-Assistant nach absteigender Wahrscheinlichkeit sortiert.

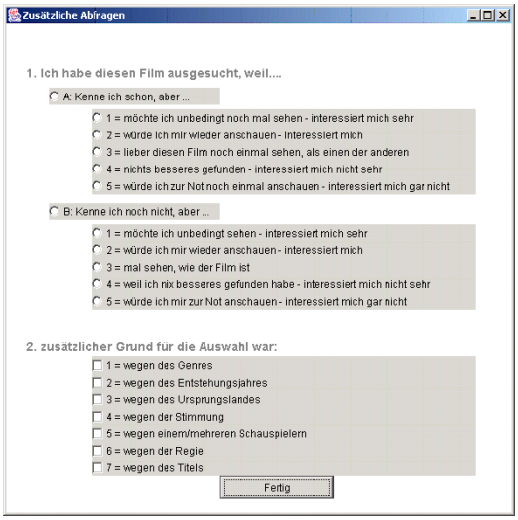
5. Variante 5: Anhand des Kriteriums Stimmung werden die Präferenzen aufgrund der am häufigsten besetzten Ausprägung ausgebildet. In dieser Einstellung bekommt der Benutzer diejenigen Filme angeboten, die seiner meist präferierten Stimmung entspricht.



a) Beraterassistenz liefert einen Filmvorschlag

b) Filterassistenz liefert 5 Filmvorschläge

c) Graphische Benutzeroberfläche zum Einstellen der BenutzerID und der aktuellen Auswahlregel



d) elektronischer Fragebogen

Abbildung 81: Graphische Oberflächen für die Versuchspersonen und den Administrator der Evaluation (links unten).

6. Variante 6: In den zwei Kriterien Genre und Stimmung wird nach tatsächlich vorhandenen Verknüpfungen gesucht. Diese Regel ähnelt Variante 3, nur dass hier Entstehungsland und Entstehungsjahr unberücksichtigt bleiben.

7. Variante 7: Von allen vier Kriterien (Genre, Entstehungsland, Entstehungsjahr und Stimmung) wird jeweils die am häufigsten besetzte Ausprägung berücksichtigt. Es werden somit Filme empfohlen, die dem am höchsten präferierten Genre, dem am höchsten präferierten Entstehungsland, dem am höchsten präferierten Entstehungs-

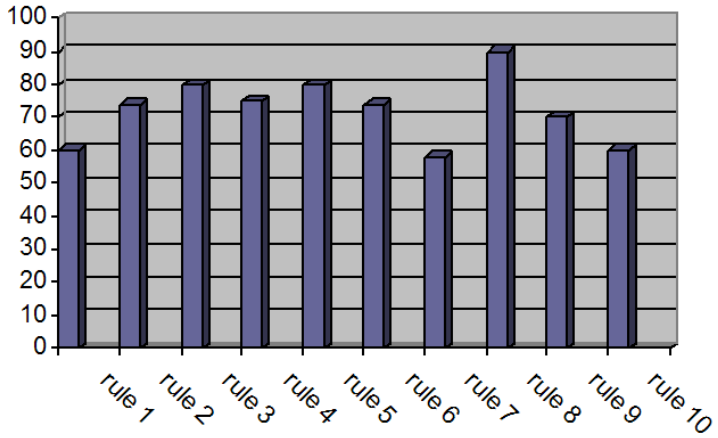


Abbildung 82: Akzeptanz der vorgeschlagenen Filme in Prozent (aus [164]).

jahr und der am höchsten präferierten Stimmung entsprechen (vgl. auch Variante 2).

8. Variante 8: In allen vier Kriterien (Genre, Entstehungsjahr, Entstehungsland und Stimmung) wird nach tatsächlich vorhandenen Verknüpfungen gesucht. Diese Regel ähnelt den Varianten 3 und 6, nur dass hier alle vorhandenen Kriterien berücksichtigt werden.

9. Variante 9: Zufallsauswahl: Das Benutzerprofil spielt hier keinerlei Rolle. Dem Benutzer wird von allen vorhandenen Filmen eine zufällige Liste angeboten.

10. Variante 10: Dissonanz: Mittels dieser Regel werden nur Filme angeboten, die nicht den jeweils ersten Plätzen der präferierten Filmattribute entsprechen. Ein Benutzer, der z. B. am liebsten Komödien mag, bekommt somit auf keinen Fall eine Komödie angeboten.

Regel 4 unterscheidet sich in der Vorgehensweise stark von den anderen Regeln. Während die Regeln 1-3 und 5-10 mittels eines konkreten Datenbankbefehls modelliert werden können, wird bei Regel 4 für jeden möglichen vorhandenen Film eine Wahrscheinlichkeit berechnet. Insofern gleicht diese Herangehensweise dem Ansatz des DIVA-Algorithmus (vgl. [163]). Bei den Regeln 1-3, 5-8 und 10 sendete die User-Assistant-Komponente konsequenterweise einen Remote Procedure Call mit konkreten Filterangaben an den Kanal, den diese Komponente mit der EPG-Komponente teilte. Bei Regel 4 wurde jedoch die gesamte vorhandene Filmliste angefordert und bewertet. Regel 9 stellt insofern eine Ausnahme dar, dass hier ebenso wie bei Ausführung der Regel 4 die gesamte Liste angefordert wurde, jedoch kein spezieller Algorithmus für jeden Film angewendet wurde, sondern zufällig entweder ein oder fünf Filme aus der gesamten Liste ausgewählt wurden.

Bei der *ersten Evaluation* bekamen 20 Versuchspersonen (Alter zwischen 20 und 40, durchschnittliches Alter der Versuchspersonen 27, 10 weiblich, 10 männlich) Filmvorschläge auf Basis der 10 adaptiven Regeln. Herbei wurde jeweils sowohl die Filterassistenz (1 Vorschlag) als auch die Beraterassistenz (5 Vorschläge) getestet. Die Entscheidungen der Benutzer wurden mitprotokolliert. Zum Einstellen der Testbedingungen hatte die Überwachungsperson der Evaluation (Administrator) die graphische Oberfläche wie in Abbildung 81 c) abgebildet zur Verfügung. Hier kann die Identifikationsnummer der Testperson, der Name der Regel und die Art der Assistenz (Filter oder Berater) eingestellt und abgesendet werden[69]. Betätigte die Testperson den Knopf „Empfehlung“ (wie in Abbildung 81 a)) so erhielt sie einen Filmvorschlag mittels der eingestellten Regel auf Basis seiner Präferenzlisten. Betätigen des Knopfes „Empfehlungen“ resultierte analog in fünf Empfehlungen (siehe Abbildung 81 b)). War eine Testperson mit einem Filmvorschlag einverstanden, so konnte sie diesen markieren und den „OK“-Knopf aktivieren. Bei Nicht-Einverständnis betätigte die Testperson den „Abbruch“-Knopf. Bei „OK“ erzeugte die graphische Benutzeroberfläche eine zusätzliche GUI (siehe Abbildung 81 d)) die einen kurzen Fragebogen darstellte, in dem die Versuchspersonen die Gründe für die Filmwahl näher spezifizieren konnten.

Das Balkendiagramm in Abbildung 82 illustriert die zusammenfassende Auswertung der ersten Evaluation. Hier ist für jede Regel die prozentuale Akzeptanz unter den Versuchspersonen aufgetragen. Auffällig sind die 70% Akzeptanz bei der Anwendung der Regel 9, die die zu empfehlenden Filme per Zufall ohne jede Berücksichtigung von individuellen Persönlichkeitsprofilen ermittelt. Ein möglicher Grund für diese hohe Akzeptanz kann sein, dass bei Versuchspersonen die viele Ausprägungen der einzelnen möglichen Filmattribute in ihren persönlichen Präferenzlisten haben per Zufall bei einer hinreichend großen Auswahlmenge (über 700 Filmtitel) immer passende Filme ausgewählt werden können. Eine Person der viele Filmgenres gefallen kann per Zufallsauswahl besser beraten werden, als eine Person die nur wenige oder gar nur ein Filmgenre mag. Aber auch ohne genaue Analyse der Gründe sollte die Anwendung von adaptiver Assistenz signifikant höhere Akzeptanzwerte als die Anwendung von Wahrscheinlichkeiten haben[70]. Die Akzeptanzwerte der Regel 9 können somit als „Nulllinie“ angesehen werden, die eine Regel für adaptive Assistenz signifikant überschreiten muss. Abbildung 82 zeigt, dass Regel 8 die deutlich höheren Akzeptanzwerte unter den Versuchspersonen ergeben hat, als die anderen Regeln (vgl. Nitschke und Hellenschmidt [164]).

In einer *zweiten Evaluation* wurde die Akzeptanz von adaptiver Assistenz im Vergleich von adaptiver und adaptierbarer Assistenz ermittelt. Hierzu waren 40 Versuchspersonen (Alter zwischen 20 und 44 Jahren, Mittelwert 27 Jahre, 19 Frauen, 19 Männer (2x

[69] Von den in Abbildung 80 abgebildeten Komponenten liefen die Administrator-GUI, der User-Assistant und die EPG-Komponenten auf einem nur der Überwachungsperson der Evaluation zugänglichen Rechner. Die Komponente mit der graphischen Oberfläche für die Versuchspersonen lief auf einem zusätzlichen Rechner. Somit war den Versuchspersonen die Einstellungen der Tests nicht einsehbar.

[70] Man denke sich hier einen intelligenten Raum, der mittels Wahrscheinlichkeiten die vorhandenen Geräte ansteuert. Eine hier zu verwendende intelligente Komponente muss signifikant höhere Akzeptanzwerte aufweisen können, um als intelligent zu gelten.

a) Adaptierbare Suche nach Genres ... b) ... nach dem Entstehungsland ...

c) ... nach dem Entstehungsjahr ... ... und der Stimmung eines Films.

Abbildung 83: Graphische Oberfläche zur Auswahl von Filmen mittels adaptierbarer Assistenz.

k.A.)[71]) aufgefordert, sich mittels der in der ersten Evaluation ermittelten Regel 8 adaptiv Empfehlungen (jeweils als Beraterassistenz und Filterassistenz) geben zu lassen als auch per graphischer Benutzeroberfläche adaptierbar nach Filmen zu suchen. Abbildung 83 und Abbildung 84 zeigen die graphische Benutzeroberfläche bei der Unterstützung der Filmauswahl mittels adaptierbarer Assistenz. Die Versuchspersonen hatten hierbei die Möglichkeit das Genre, das Entstehungsland, das Entstehungsjahr und die Stimmung der auszusuchenden Filme festzulegen und somit das Suchen nach Filmen selbstständig einzuschränken. Die graphische Benutzeroberfläche war hierbei dem „Original“ des Electronic Program Guide des Projektpartners Grundig AG nachempfunden. Dies gewährlei-

[71]Da die Tests anonym erfolgten und auf dem zugehörigen Fragebogen das Feld bezüglich des Geschlechts in zwei Fällen unausgefüllt blieb, lässt sich nachträglich zu zwei Versuchspersonen hierzu keine Angabe machen.

a) Ergebnisliste nach adaptierbarer Auswahl

b) Original-GUI des Projektpartners Grundig AG

Abbildung 84: Anzeige von Spielfilmen unter adaptierbarer Assistenz.

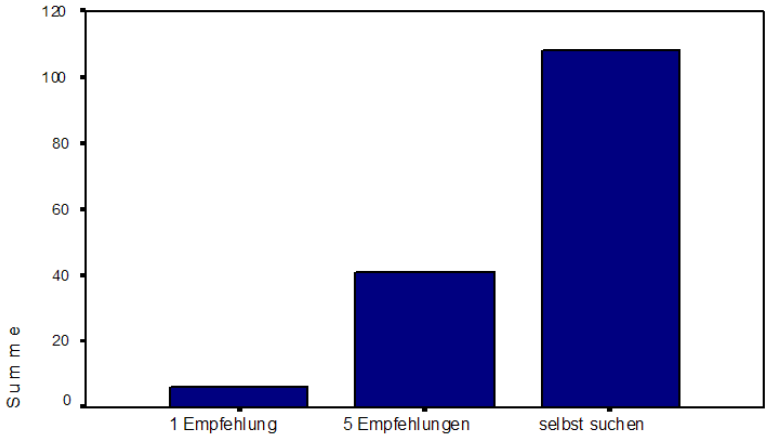


Abbildung 85: Die Akzeptanz von adaptiver (links und mitte) und adaptierbarer Assistenz in absoluten Häufigkeiten.

stete in Hinsicht von Bedienung und Aussehen ein gewisses Echtheitsempfinden unter den Versuchspersonen. Nach Akzeptanz eines Verschlages bzw. der Auswahl eines bestimmten Filmes wurde den Versuchspersonen wiederum der in Abbildung 81 d) dargestellte Fragebogen zum Ausfüllen eingeblendet.

7.1.3 Evaluationsergebnisse und Diskussion

Der im EMBASSI-Projekt realisierte User-Assistant zur Bereitstellung adaptiver Assistenz basiert auf der rein impliziten Aufnahme von Benutzerprofilen. Diese Herangehensweise zur Realisierung von Assistenz unterscheidet sich von denen, die einen direkten Benut-

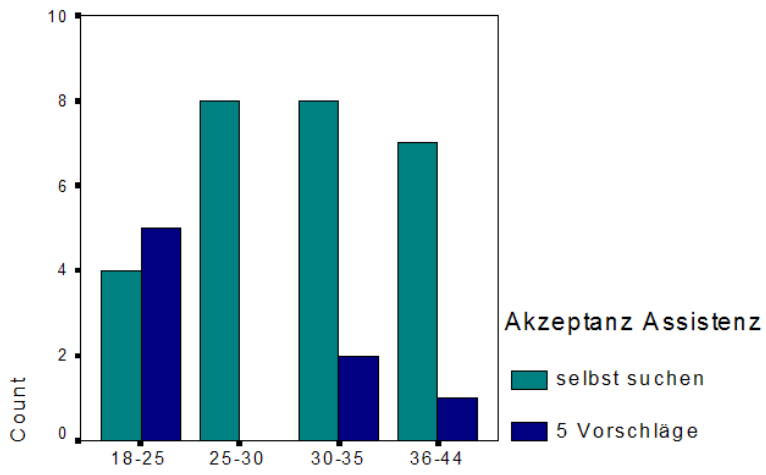


Abbildung 86: Vergleich der Akzeptanzwerte von adaptiver (dunkel) und adaptierbarer (hell) Assistenz. Nach rechts sind verschiedene Altersgruppen abgetragen.

zereingriff, sei es direkt beim Konsumieren eines Films oder bei einer Vorkonfiguration, erfordern. Zur Ermittlung geeigneter Regeln zur Auswahl von Filmen zeichnete sich in einer ersten Evaluation eine Regel aus, die alle Parameter eines Films hinsichtlich der Benutzerpräferenzen berücksichtigt (Regel 8). Aber auch die Regel auf der Basis der Berechnung von Wahrscheinlichkeiten (Regel 4) schnitt in der Akzeptanzrate (vgl. Abbildung 82) signifikant höher ab, als Regeln die nur einzelne Attribute von Filmen oder Kombinationen bzw. Verknüpfungen davon berücksichtigen. Eine mögliche Ursache könnte darin liegen, dass sowohl bei der Filterassistenz (5 Empfehlungen) als auch bei der Beraterassistenz (1 Empfehlung) aus einer aufgrund der Regelvorgaben passenden Liste (die bei über 700 Filmen in einer Datenbank o.B.d.A. immer mehr als 5 Einträge umfasst) per Zufall einige wenige zur Empfehlung bestimmen musste. Die vierte Regel verfügte nach der Wahrscheinlichkeitsberechnung immer über eine absteigende Liste von Filmen und konnte die am höchsten priorisierten Filme den Versuchspersonen anbieten. Aber obwohl andere Regeln eine kleine Zufallskomponente beinhalten, hätten sie dennoch aufgrund der individuellen Auswahlkriterien auf Basis ihres Präferenzprofils signifikant besser abschneiden müssen als die Anwendung des reinen Zufallsprinzips (Balken 9 in Abbildung 82). Die zusätzliche Auswertung der nach dem Akzeptieren eines Filmes auszufüllenden Fragebögen ergab keine signifikanten Abweichungen der Akzeptanzraten. Filme die von den Testpersonen ausgewählt wurden, wurden auch jeweils in den Fragebögen als „interessant“ oder „sehr interessant“ bewertet. Die zweite durchgeführte Evaluation gibt Aufschluss über die generelle Akzeptanz von adaptiver Assistenz im Vergleich zur adaptierbaren Assistenz bei der die Versuchspersonen jeden Auswahlschritt selbst durchführen mussten. Abbildung 85 zeigt eine signifikant höhere Akzeptanz des sog. „Selbstsuchens“ gegenüber der Akzeptanz der beiden adaptiven Assistenzarten Filterassistenz (mittlerer Balken in Abbildung 85) und Beraterassistenz (rechter Balken). Abbildung 86 illustriert diese Akzeptanzwerte aufgeschlüsselt nach Altergruppen. Hier ergibt sich – wohl aufgrund der geringen Stichprobe pro Altersgruppe – ein indifferentes Bild. Die minimal

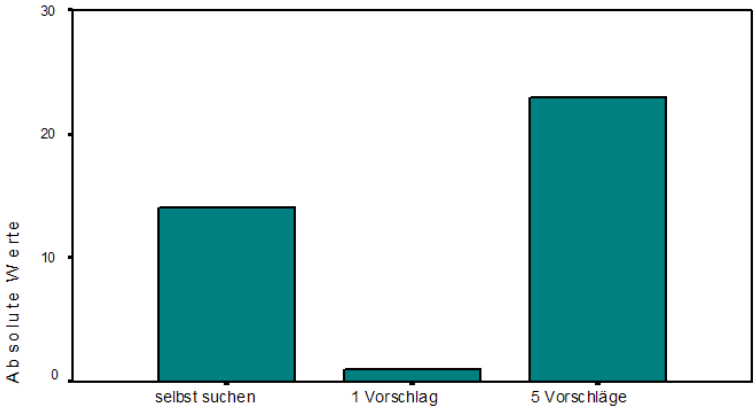
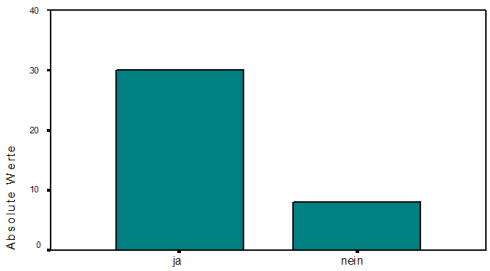
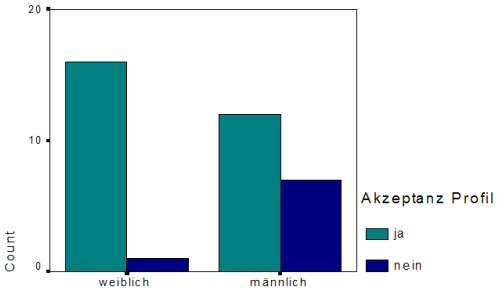


Abbildung 87: Akzeptanz von adaptiver Assistenz bei angenommener idealer Adaptivität.



a) Beide Geschlechter



b) nach Geschlechtern getrennt.

Abbildung 88: Einverständnis mit Profilbildung und Auswertung der Versuchspersonen.

höhere Akzeptanz der adaptiven Assistenz bei der jüngsten Altersgruppe – und damit die Vermutung dass jüngere Versuchspersonen der Assistenztechnologie zugeneigter sind – scheint durch die nicht vorhandenen Akzeptanz der nächsthöheren Altergruppe (25-30) aufgehoben zu sein. Die zweite Evaluation wurde mit allen Versuchspersonen noch zusätzlich unter verschiedenen Bedingungen ausgeführt. Aber weder Zeitdruck (bis zu einer gewissen Zeit müssen Filme ausgesucht sein), noch Aufwand (mit wenigen Schritten zum Ziel kommen) noch eine Variation der Optionenmenge (Erhöhung der Anzahl der Filme in der Filmdatenbank) scheinen einen nennenswerten Einfluss zu haben (vgl. Nitschke [165]). Bemerkenswert sind die Ergebnisse zweier zusätzlicher Fragen an die Testpersonen im Anschluss an die jeweiligen Tests. Abbildung 87 zeigt in absoluten Zahlen die Ergebnisse bei der Frage, ob die Versuchspersonen die adaptive Assistenz dem adaptierbaren Selbstsuchen vorziehen würden, wenn garantiert werden könnte, dass die Ergebnisse in allen Fällen exakt dem persönlichen Profil entsprechen würden und jeweils das Optimum aus der gegebenen Optionenmenge darstellen würden. Hier gibt es eine signifikant höhere Akzeptanz bezüglich der adaptiven Assistenz (mittlerer und rechter Balken in Abbildung 87) gegenüber der adaptierbaren Assistenz. Offenbar begründet vor allem das Misstrauen gegebenüber den empfohlenen Filmen, dass die adaptive Assistenz

signifikant schlechtere Werte in der zweiten Evaluation erfahren hat (vgl. Abbildung 85). Offenbar ist die Furcht vor Kontrollverlust im Falle der adaptiven Assistenz (und die damit verbundene Furcht nicht das Optimum zu bekommen bzw. zu erfahren) sehr hoch. Einen ähnlichen Wunsch nach „optimalen“ Filterfunktionen im Falle einer zu hohen Optionsmenge und damit verbunden der Wunsch nach optimaler adaptiver Assistenz findet sich auch in einer Untersuchung von Roeker et al. [182] im Zuge des AMIGO-Projektes. Andere Untersuchungen von Westerbrink et al. [218], die Konzepte zur Realisierung personalisierter Programmzeitschriften (EPGs) erforschen, haben aber auch ergeben, dass Benutzer nicht glauben, dass es elektronische Systeme geben könnte, die in der Lage sind, fehlerfreie Vorschläge zu geben, die eigenen persönlichen Präferenzen entsprechen. Ähnlich wie Misker et al. [160] argumentieren sie, dass Benutzer in jeder Phase in der vollständigen Kontrolle sein möchten. Offensichtlich besteht zur Behandlung der Frage der optimalen adaptiven Assistenz und ihrer Akzeptanz noch vermehrter Forschungs- und Klärungsbedarf. Ein interessantes und aufschlussreiches Ergebnis der Evaluation des User-Assistants soll noch detailliert genannt werden. Adaptive Assistenz ist nicht realisierbar ohne die Ermittlung und Aufzeichnung persönlicher Daten und dem intelligenten Inferieren von Schlussfolgerungen daraus. Befragt nach der Akzeptanz und dem prinzipiellen Einverständnis, dass ein solches Benutzerprofil gebildet werden darf, äußerten fast ein Viertel der Testpersonen (vgl. Abbildung 88 a)) ihr Nichteinverständnis. Hier ist besonders der Unterschied der Meinungen der verschiedenen Geschlechter auffallend. Während die überwiegende Mehrheit der Frauen keine Bedenken gegen das Aufzeichnen und Verwerten von persönlichen Profilen hat (vgl. Abbildung 88 b) links), äußerten bei den Männern mehr als ein Drittel Bedenken (vgl. Abbildung 88 b) rechts).

7.2 Gerätekoordination in DYNAMITE

DYNAMITE (Dynamisch Adaptive Multimodale IT-Ensembles), in dessen Kontext diese Arbeit und speziell die Arbeiten an der verteilten Realisierung der in Kapitel 4 und Kapitel 5 beschriebenen und spezifizierten Software-Infrastruktur entstanden, definiert dynamische Anwendungsszenarios unter anderem im Unterhaltungsbereich (vgl. Kapitel 2.1.13). Diese – oftmals unter dem Begriff Wohnzimmer – beschriebenen Szenarios decken den Bereich des Umgangs mit Medien im Allgemeinen, aber vor allen den Bereich des Abspielens und Ansehens von Filmen und von Musik ab. Unterschiedliche Unterhaltungsgeräte wie Fernsehapparate und Hifi-Anlagen sollen beim Abspielen von Musik und Filmen kooperieren und den Benutzer bei der Bedienung bestmöglichst unterstützen. Die Bedienung hierbei soll integriert und multimodal erfolgen, das heißt in einer Kombination von Spracheingabe und graphischer Benutzerschnittstelle möglich sein. Äußert der Benutzer seinen Wunsch einen Film zu sehen, z. B. mit den Worten „Ich möchte einen Film sehen“ oder „Film bitte“, so wird das Ziel einen Film abzuspielen erkannt und gegebenenfalls zur genauen Zieldefinition durch Rückfragen an den Benutzer näher spezifiziert. Wurde das Ziel in einem Dialog mit dem Benutzer definiert, so sollte es unter Einbeziehung aller vorhandenen Geräte ausgeführt werden. Im Rahmen der Arbeiten in DYNAMITE ist zur Demonstration der Möglichkeiten der dynamischen Integration von neuen Geräten und zur Illustration der Ideen der Selbstorganisation eine Anwendung im Bereich der Unterhaltung entstanden, die in diesem Abschnitt beschrieben werden soll. Hierbei werden vor allem Experimente im Bereich der Dynamik sowie die Anwendung der Konfliktlösungsalgorithmen (vgl. Kapitel 6.2.1 und Kapitel 6.2.2) eine bedeutende Rolle spielen. Zuletzt soll die Nutzbarkeit der in dieser Arbeit entwickelten Infrastruktur durch die Integration einer neuen Konfliktlösungsstrategie nachgewiesen werden.

7.2.1 Topologie und Komponenten

Abbildung 89 illustriert die Anwendung der in Kapitel 6.2 definierten erweiterten Basistopologie mit der Ergänzung um Ausgabekomponenten in einer Teil-Instantiierung mit Ausgabekomponenten und Eingabekomponenten. Als Eingabekomponenten wirken eine Komponenten zur Spracheingabe und eine Komponente mit einer graphischen Benutzeroberfläche. Mit dem Ausgabekanal ist eine Sound-Komponente verbunden, die in der Lage ist eine Sprachausgabe wiederzugeben. Die in der Applikation wirkenden Komponenten auf Ein- und Ausgabeebene, wie in Abbildung 89 dargestellt, erfüllen damit die Grundanforderungen an multimodaler Interaktion. Als Konfliktlösungsstrategie definieren die Komponenten auf dem Eingabekanal die Kanalstrategie zur Zuteilung von Ereignissen, wie in Kapitel 6.2.1 beschrieben. Auf dem Ausgabekanal wird die dazugehörige Alternativstrategie „all“ verwendet. Die Verwendung dieser Strategien wird in den Anwendungsbeispielen im Folgenden deutlich. Die Richtung der Pfeile illustriert, dass die Eingabekomponenten Ereignisse auf den Ereigniskanal publizieren und dementsprechend von diesem Kanal Remote Procedure Calls empfangen können. Die Ausgabekomponenten empfangen konsequenterweise Ereignisse aus dem Ausgabekanal. Ebenso verdeutlicht die Richtung der Pfeile der Komponenten zum Zielkanal und zum Funktionska-

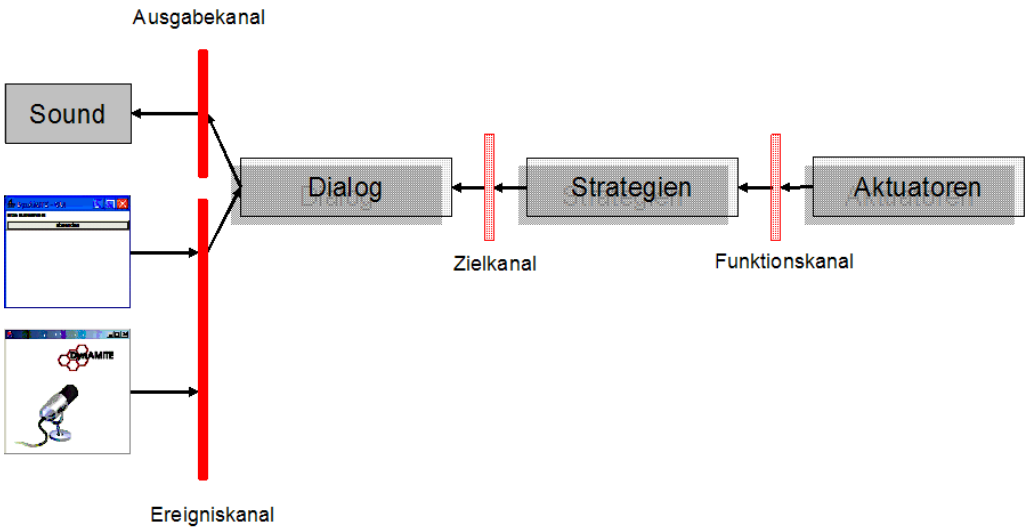


Abbildung 89: Anwendung der erweiterten Basistopologie zur Realisierung eines DYNAMITE-Szenarios im Anwendungsbereich Unterhaltung.

nal, dass Dialogkomponenten, die von ihnen erkannten Ziele per Remote Procedure Call auf den Zielkanal publizieren. Davon werden sie von den Strategiekomponenten bearbeitet. Die Strategiekomponenten wiederum veröffentlichen ihre Funktionsaufrufe per RPC auf den Funktionskanal[72]. Transparent dargestellt sind in Abbildung 89 die möglichen Dialog-, Strategie- und Aktuatorkomponenten. Diese sollen in der hier beschriebenen Anwendung ad-hoc von den unterschiedlichen Geräten mitgebracht und mit den bereits vorhandenen Eingabe- und Ausgabekomponenten vernetzt werden. An diesen Beispielen werden im Folgenden die spontane Kooperation von Geräten und die Möglichkeit zur Konfliktlösung eines solchen Geräteensembles untersucht. Die Komponente *Spracherkenner* ist in Java implementiert und basiert auf der ViaVoice Technologie von IBM[73]. Diese Komponente ist als Event-Quelle am Ereigniskanal registriert und ist damit auch in der Lage auf Remote Procedure Calls zu antworten. Der folgende Programmiercode gibt die Implementierung der Komponente Spracherkenner als Transducer innerhalb einer SODAPOP-Umgebung wieder. Die Komponente definiert den Ereigniskanal und weist dieser die Konfliktlösungsstrategie *event_strategy* (vgl. Kapitel 6.2.1) zu. Zugleich registriert sie sich hier als Event-Quelle und definiert den entsprechenden RPC-Handler (Zeilen 4 und 5).

[72]Diese Vorgehensweise ist vollkommen äquivalent zu den Definition aus Kapitel 6.2. Dort ist in den zur Definition der Basistopologie definierten Abbildungen der Hauptnachrichtenfluss illustriert. Dieser weist weiterhin von den Eingabekomponenten hin zu den Aktuatoren. In Abbildung 89 wird die in Kapitel 5 verabredete Pfeilrichtung der Publikation von Ereignissen verwendet.

[73]Informationen zu ViaVoice sind unter http://www-306.ibm.com/software/pervasive/embedded_viavoice/ (Stand Frühjahr 2007) zu erhalten. Die Anbindung an die Programmiersprache Java findet unter Verwendung einer API mittels des Java Native Interface statt. Diese API ist als Java Speech 1.0 API von IBM unter dem AlphaWorks-Label unter http://www-128.ibm.com/developerworks/ibm/library/i-voice/ [200] zu erhalten.

```
1: public class SpeechRecognizer {

2: static Channel event_channel = null;

3: public SpeechRecognizer(){
4:       event_channel =
              new Channel("event_channel", "OUT",
                   "event_strategy");
5:       event_channel.subscribe(new SR_RPCHandler(this));
6: }

7: public static void main (String[] args){
8:       new SpeechRecognizer();
9: }}
```

Bei der Verwendung von ViaVoice als Spracherkenner wird gewöhnlich eine *grammar*-Datei definiert. Eine solche Datei definiert den Sprachschatz eines Spracherkenners und wird von diesem zum Startzeitpunkt eingelesen. Die Zeilen

```
grammar javax.speech.demo;
public <sentence> = Spielfilm bitte | Spielfilm stoppen
              | Musik bitte | Musik beenden;
```

definieren, dass der Spracherkenner genau die vier definierten (Halb-)sätze versteht und bei Erkennung das Schlüsselwort *sentence* zur weiteren Datenverarbeitung auswirft. Eine solche Vorgehensweise zeichnet sich durch verschiedene Nachteile aus, die den Einsatz in dynamischen Umgebungen erschweren. Ein statischer Wortschatz macht einen Spracherkenner in unbekannten Umgebungen unbrauchbar und erschwert zudem den Einsatz in dynamischen Anwendungsszenarios, in denen neue Geräte hinzukommen sollen. Daher wurde die Komponente Spracherkenner in der Art implementiert, dass der erkennbare Wortschatz dynamisiert werden kann. Die Komponente ist in der Lage entsprechende Nachrichten zu verarbeiten, die zur dynamischen Erstellung einer passenden *grammar*-Datei führen. Zugleich ist die Komponente Spracherkenner mit einer graphischen Oberfläche ausgestattet (siehe Abbildung 90) die mittels roten und grünen Signalen den Benutzer informiert, ob seine Spracheingabe auch verstanden wurde. Die gleichen Aussagen – wie für den Spracherkenner gelten – lassen sich auch für eine graphische Benutzeroberfläche treffen, die dem Benutzer wählbare Optionen für die Äußerung seines Benutzerwunsches bieten soll. Auch hier kann in dynamischen Umgebungen keine vorgefertigte Oberfläche angeboten werden, da sonst entweder die Oberfläche nicht alle möglichen Optionen enthalten kann oder die Oberfläche mit so vielen Optionen überfrachtet sein kann, dass es dem Benutzer keine Hilfe ist. Die Komponente *Graphische Benutzeroberfläche* ist somit auch in der Lage mit der dynamischen Generierung von Auswahlmöglichkeiten zu reagieren. Dies entspricht analog der Generierung des dynamischen Wortschatzes in der Komponente Spracherkenner. Die *Sound-Komponente*, die mit dem Ausgabekanal verbunden ist, ist in der Lage Sätze als gesprochene Wörter wiederzugeben. Sie ist als Event-Senke am Ausgabekanal verbunden und mittels der folgenden Programmierzeilen implementiert:

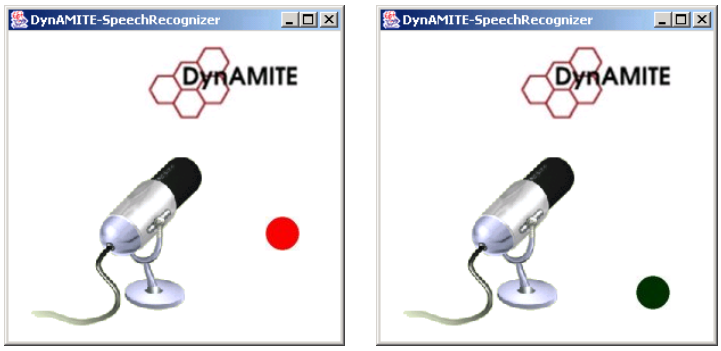


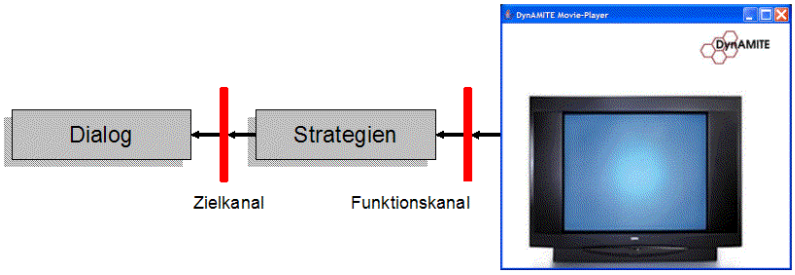
Abbildung 90: Die graphische Oberfläche des Spracherkenners. Ein grünes Signal bedeutet, dass die Spracheingabe des Benutzers erkannt wurde und weiterverarbeitet wird.

```
1: public class SoundEngine {

2: static Channel output_channel = null;

3: public SoundEngine(){
4:       output_channel =
              new Channel("output_channel", "IN",
                  "all");
5:       output_channel.subscribe(new SE_EventHandler(this));
6: }

7: public static void main (String[] args){
8:       new SoundEngine();
9: }}
```

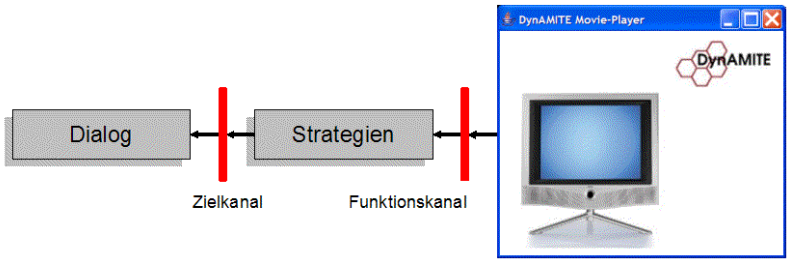
Sie ist am Ausgabekanal (*output_channel*) als Event-Senke registriert und hat an diesem Kanal die Strategie *all* – eine Alternative zur komplexeren Konfliktlösungsstrategie zur Behandlung von Ereignissen – definiert.

7.2.2 Komponenten zur Medienwiedergabe

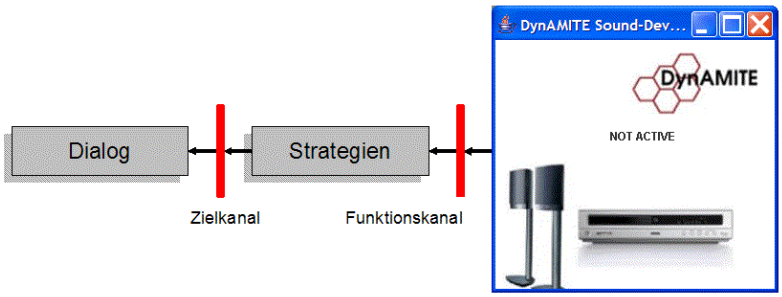
Für die Wiedergabe von Medien (Filmdateien und Musikdateien) stehen im Rahmen dieser Anwendung drei unterschiedliche virtuelle Geräte zur Verfügung. Hierbei wurden Geräteapplikationen realisiert, die einen Fernsehapparat mit großen Bildschirm, einen mit einem etwas kleineren Bildschirm und eine Anlage für die Wiedergabe von hochwertigem Ton darstellen. Wie Abbildung 91 illustriert, bringt jedes Gerät eine eigene Strategiekomponente und eine eigene Dialogkomponente mit sich. Somit ist für jedes Gerät die Autonomie gewährleistet. Die Aufgabe einer *Dialogkomponente* ist es nach den Definitionen in Kapitel 6 das Ziel des Benutzers zu bestimmen und gegebenenfalls genauer zu spezifizieren. Zur genauen Spezifizierung kann sich eine Dialogkomponente den Ausgabekomponenten bedienen und Rückfragen an den Benutzer stellen. Mittels der Ereignisse der Eingabekomponenten kann es der Dialogkomponente so gelingen, ein Ziel so



a) Applikation zur Filmwiedergabe mit großem Bildschirm.



b) Applikation zur Filmwiedergabe mit kleinerem Bildschirm.



c) Applikation zur Musikwiedergabe.

Abbildung 91: Drei Geräteapplikationen – großer Fernsehapparat, kleiner Fernsehapparat und Hifi-Gerät zur Musikwiedergabe – mit ihren individuellen Dialog- und Strategiekomponenten.

zu definieren, dass es auf den Zielkanal gegeben werden kann. Ein solcher Dialog kann so ablaufen, dass ein Benutzer seinen generellen Wunsch, einen Film zu sehen, äußert und mittels Nachfragen zu den Details zu dem gewünschten Film befragt wird. Eine solche Dialogkomponente würde sich in diesem Falle wie eine State-Machine verhalten, die mit Hilfe von Nachfragen feste *if-then*-Pfade abläuft. Eine solche Dialogkomponente ist

nicht in der Lage einen eingeschlagenen Weg zu verlassen und entsprechend auf den Benutzer zu reagieren. Vielmehr muss der Benutzer die im gestellten Fragen beantworten (siehe Ludwig [152] und Bücher et al. [37] für eine andere Herangehensweise zur Implementierung eines Dialogmanagers, wie er im Projekt EMBASSI realisiert wurde). Für die Integration von neuen Geräten und die Ermöglichung der Bedienung dieser Geräte ist dieser Ansatz der State-Machine aber ausreichend. Für die hier beschriebene Anwendung wurde eine konfigurierbare Dialogkomponente realisiert, die mittels beim Start einzulesenden XML-Dateien konfiguriert werden kann. Somit wird in allen drei (virtuellen) Geräten in Abbildung 91 prinzipiell dieselbe Dialogkomponente – aber in verschiedenen Instanzen und Konfigurationen – verwendet. Der folgenden Programmzeilen definieren die DTD (Document Type Definition) concept.dtd einer solchen XML-Datei, wie sie von einer Dialogkomponente bei ihrem Start eingelesen wird.

```
<?xml version="1.0"?>
<!ELEMENT Concept (SelfID, Keywords, Aspects, Commands)>
<!ATTLIST Concept Name CDATA #REQUIRED>
<!ELEMENT SelfID (#PCDATA)>
<!ELEMENT Keywords (Keyword+)>
<!ELEMENT Keyword (#PCDATA)>
<!ELEMENT Aspects (Aspect)>
<!ELEMENT Aspect (Question, Answers+)>
<!ATTLIST Aspect Name CDATA #REQUIRED>
<!ATTLIST Aspect Mapto CDATA #REQUIRED>
<!ELEMENT Question (#PCDATA)>
<!ATTLIST Question Type CDATA #FIXED "MultiChoice">
<!ELEMENT Answer (#PCDATA)>
<!ATTLIST Answer MapValue CDATA #REQUIRED>
<!ELEMENT Commands (Command+)>
<!ELEMENT Command (#PCDATA)>
```

Für Filmapplikationen wie in Abbildung 91 a) und b) dargestellt sieht eine mögliche Konfigurationsdatei wie folgt aus:

```
1: <?xml version="1.0" encoding="ISO-8859-1"?>
2: <!DOCTYPE Concept SYSTEM "concept.dtd">
3: <Concept Name="PlayMovie" command="add">
4:       <SelfID><!-- Self identification text -->
5:              Ich kann Spielfilme abspielen.
6:       </SelfID>
7:       <Keywords><!-- Keywords for Mapping of Concept -->
8:              <Keyword> Spielfilm zeigen   </Keyword>
9:              <Keyword> Spielfilm gucken   </Keyword>
10:             <Keyword> Spielfilm spielen </Keyword>
11:             <Keyword> Spielfilm bitte </Keyword>
12:       </Keywords>
13:       <Aspects>
14:              <Aspect Name="Genre" Mapto="Asp1">
15:                     <Question Type="MultiChoice">
16:                            Was fuer einen Film moechten Sie.
17:                            Ich habe Nachrichten und Berichte.
```

```
18:                     </Question>
19:                     <Answers><!-- Possible answers -->
20:                            <Answer MapValue="news">
21:                                   Nachrichten </Answer>
22:                            <Answer MapValue="report">
23:                                   Berichte </Answer>
24:                     </Answers>
25:              </Aspect>
26:              <Aspect Name="Loudness" Mapto="Asp2">
27:                     <Question Type="MultiChoice">
28:                            In welcher Lautstaerke.
29:                            Aus, mittel oder laut.
30:                     </Question>
31:                     <Answers>
32:                            <Answer MapValue="0">
33:                                   aus</Answer>
34:                            <Answer MapValue="3">
35:                                   mittel</Answer>
36:                            <Answer MapValue="5">
37:                                   laut</Answer>
38:                     </Answers>
39:              </Aspect>
40:       </Aspects>
41:       <Commands>
42:              <Command>
43:                     <chooseMovie>
44:                            <genre>Asp1</genre>
45:                            <volume>Asp2</volume>
46:                     </chooseMovie>
47:              </Command>
48:       </Commands>
49: </Concept>
```

Das Konzept *PlayMovie* besteht aus einem sogenannten Identifikationstext (Zeilen 4-6), den Schlüsselwörtern, mit denen der Benutzer das entsprechende Konzept aktivieren kann (Zeilen 7-12) und den Aspekten. Die Aspekte definieren nacheinander die Fragen, die die Dialogkomponente an den Benutzer stellen wird. Im Beispiel oben sind zwei unterschiedliche Fragen definiert: nach der Art des Films (Zeilen 14-24) und der einzustellenden Lautstärke (Zeilen 26-39). Sind diese beiden Fragen beantwortet, so kann das damit verbundene Benutzerziel weitergeben werden. Dies wird in den Zeilen 41-48 definiert. Dabei werden die Variablen *Asp1* und *Asp2* durch die vom Benutzer gegebenen Antworten auf die beiden vorangegangenen Fragen verwendet. Die Zuordnung hierzu fand in den Zeilen 14 und 26 statt. Die Dialogkomponente sendet im Anschluss das erkannte und fertig spezifizierte Benutzerziel mittels eines Remote Procedure Calls auf den Zielkanal von dem aus das Benutzerziel von der mitgebrachten Strategiekomponente in passende Funktionsaufrufe umgewandelt wird. Abbildung 92 illustriert die Komponententopologie im Falle des Startens der oben beschriebenen Komponenten Spracherkenner, graphische Benutzeroberfläche, Sound-Engine und der Komponente der Applikation zur Filmwiedergabe mit dem etwas größerem Bildschirm. Die Dialogkomponente registriert sich mittels der

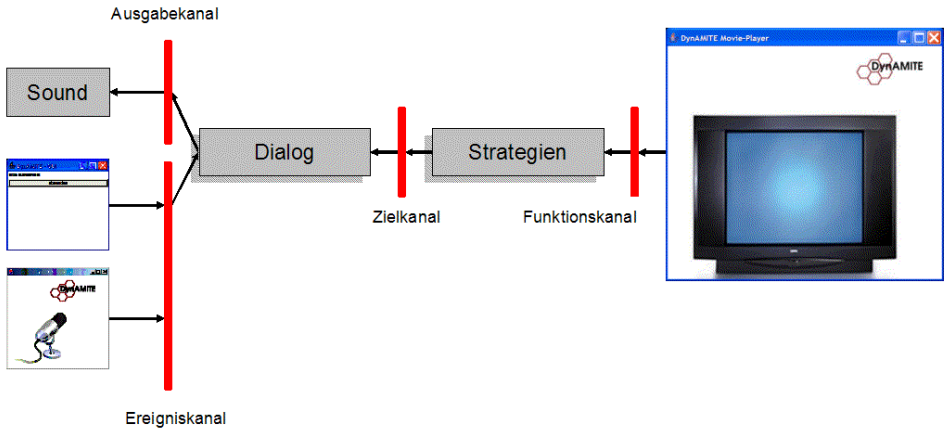


Abbildung 92: Darstellung der Komponententopologie mit einem Gerät zur Wiedergabe von Filmen.

Programmierzeilen

```
1: public class DialogManager {

2: static Channel output_channel = null;
3: static Channel input_channel = null;
4: static Channel goal_channel = null;

5: public DialogManager(){
6:        output_channel =
               new Channel("output_channel", "OUT",
                   "all");
7:        output_channel.subscribe(new DM_OutputRPCHandler(this));

8:        input_channel =
               new Channel("input_channel", "IN",
                   "event_strategy");
9:        input_channel.subscribe(new DM_InputEventHandler(this));

10:        goal_channel =
               new Channel("goal_channel", "IN",
                   "obs_strategy");
11:        goal_channel.subscribe(new DM_GoalEventHandler(this));
12: }

13: public static void main (String[] args){
14:        new DialogManager();
15: }}
```

als Event-Senke auf dem Ereigniskanal (und ist damit auch in der Lage Remote Procedure Calls auf diesen Kanal zu geben) und als Event-Quelle auf dem Ausgabekanal. Mittels

den Definitionen der Zeilen 10 und 11 registriert sich die Dialogkomponente als Absender von Remote Procedure Calls auf dem Zielkanal und definiert hier als zu verwendende Konfliktlösungsstrategie die in Kapitel 6.2.2 definierte Kanalstrategie zur Auswahl unter konkurrierenden Komponenten. Für die Implementation der Strategiekomponente als Empfänger von Remote Procedure Calls vom Zielkanal und als Versender von Remote Procedure Calls auf den Funktionskanal gilt entsprechend:

```
1: public class Strategy {

2: static Channel goal_channel = null;
3: static Channel functions_channel = null

4: public Strategy(){
5:       goal_channel =
              new Channel("goal_channel", "OUT",
                  "obs_strategy");
6:       goal_channel.subscribe(new S_GoalRPCHandler(this));

5:       functions_channel =
              new Channel("functions_channel", "IN",
                     "obs_strategy");
6:       functions_channel.subscribe(new
              S_FunctionsEventHandler(this));
7: }

8: public static void main (String[] args){
9:       new Strategy();
10: }}
```

Zuletzt noch die Realisierung der eigentlichen Komponente zum Abspielen von Filmen. Diese empfängt Remote Procedure Calls vom Funktionskanal und wird daher wie folgt implementiert:

```
1: public class Actuator {

2: static Channel functions_channel = null

3: public Actuator(){
4:       functions_channel =
              new Channel("functions_channel", "OUT",
                     "obs_strategy");
5:       functions_channel.subscribe(new A_RPCHandler(this));
6: }

7: public static void main (String[] args){
8:       new Actuator();
9: }}
```

Dabei definieren die Strategiekomponenten und die Aktuatoren auf dem Funktionskanal analog zur Definition auf dem Zielkanal die Verwendung der Konfliktlösungsstrategie zur Auswahl unter konkurrierenden Agenten (vgl. Kapitel 6.2.2). Im folgenden Abschnitt wird deren Wirkungsweise genauer erläutert.