RWTH Aachen

Department of Computer Science
Technical Report

Softwareunterstützung für adaptive eHome-Systeme

Daniel Retkowitz

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2010-05

RWTH Aachen · Department of Computer Science · July 2010

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Softwareunterstützung für adaptive eHome-Systeme

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Daniel Retkowitz

aus Münster in Westfalen

Berichter: Universitätsprofessor Dr.-Ing. Manfred Nagl
Universitätsprofessor Dr. rer. nat. Kurt Geihs
Universitätsprofessor Dr. rer. nat. Bernhard Rumpe

Tag der mündlichen Prüfung: 13. Januar 2010

Die Druckfassung dieser Dissertation ist unter der ISBN 978-3-8322-9246-1 erschienen.

Zusammenfassung

eHomes sind Umgebungen, die sich durch eine Vernetzung der vorhandenen Geräte auszeichnen, und in denen komplexe geräteübergreifende Funktionalitäten in Form von Diensten angeboten werden. Übergreifende Funktionalitäten gehören im Automobil bereits heute zum Standard, in Wohnumgebungen sind sie jedoch bislang nicht verbreitet. Aber auch hier gibt es verschiedenste Anwendungsfelder für solche Funktionalitäten, z. B. in den Bereichen Sicherheit, Komfort, Multimedia, Kommunikation oder auch bei der medizinischen Überwachung und der Unterstützung von Senioren. Daher werden eHomes in Zukunft an Bedeutung gewinnen.

Für die Entwicklung von eHomes müssen eine Reihe von Herausforderungen überwunden werden. Ein wesentlicher Aspekt ist die Berücksichtigung von *Mobilität* und *Dynamik* in eHomes, die sich sowohl auf die Bewohner als auch auf die Geräteumgebung beziehen. Da in eHomes eine Vielzahl von Standards Anwendung finden, muss außerdem der daraus resultierenden *Heterogenität* Rechnung getragen werden, um die Interoperabilität der Dienste zu gewährleisten. Damit eHomes kostengünstig realisiert werden können, muss insbesondere eine geeignete *Infrastruktur* bereitgestellt werden, die die Entwicklung von Diensten und die Interaktion mit dem eHome-System vereinfacht. Sowohl für die Dienstentwicklung als auch für den Betrieb von eHomes sind daher geeignete *Softwarewerkzeuge* erforderlich.

Bisher wurde am Lehrstuhl für Informatik 3 (Softwaretechnik) der RWTH Aachen bereits ein Ansatz erarbeitet, der die Entwicklung von eHome-Diensten durch den Einsatz wiederverwendbarer Standardkomponenten für die Dienstimplementierung vereinfacht. Im sogenannten SCD-Prozess, der sich aus den Phasen *Spezifizierung*, *Konfigurierung* und *Deployment* zusammensetzt, werden die Benutzerwünsche erfasst, und es wird darauf basierend eine eHome-spezifische Dienstkomposition erstellt und dann zur Ausführung gebracht. Ein entsprechendes Werkzeug unterstützt diesen Prozess.

Die vorliegende Arbeit erweitert die bisherigen Konzepte durch Mechanismen zur dynamischen Adaption. Dabei wird der bisherige SCD-Prozess, der die Installationsphase eines eHomes abdeckt, zu einem *kontinuierlichen SCD-Prozess* erweitert, der eine Unterstützung im laufenden Betrieb des eHomes ermöglicht.

Die *strukturelle Adaption* dient der dynamischen Anpassung der Dienstkomposition. Änderungen, die sich aus Mobilität und Dynamik ergeben, können so zur Laufzeit berücksichtigt werden. Damit dies soweit möglich automatisch durchgeführt werden kann, werden Erweiterungen der Dienstspezifikation eingeführt, die es dem Entwickler erlauben, das spätere Bindungsverhalten sowie mögliche Einschränkungen der Bindungen festzulegen. Durch die neu eingeführten Mechanismen kann die Mehrfachentwicklung von Querschnittsfunktionalitäten zur personen- und kontextbezogenen Konfigurierung vermieden werden, was den Aufwand der Dienstentwicklung reduziert.

Die *semantische Adaption* ermöglicht die Überwindung syntaktischer Inkompatibilitäten, die sich aus der Heterogenität in eHomes ergeben. Dazu wird die Dienstspezifikation um eine semantische Beschreibung ergänzt, die die syntaktischen Elemente der Dienstschnittstellen auf semantische Konzepte einer domänenspezifischen Ontologie abbildet. Dies schafft die Grundlage für eine Dienstkomposition auf Basis semantisch übereinstimmender Funktionalitäten. Ein im Rahmen der vorliegenden Arbeit entwickelter Mechanismus zur automatischen Adaptergenerierung ermöglicht die praktische Umsetzung des Ansatzes.

Die Ergebnisse dieser Arbeit liefern einen wissenschaftlichen Beitrag in verschiedenen Bereichen. Der verfolgte Ansatz ist spezifisch auf die *Anwendungsdomäne eHomes* ausgerichtet. Darin unterscheidet er sich von anderen Ansätzen, die sich häufig mit allgemeinen Fragen der dynamischen Dienstkomposition befassen. So ist eine gezieltere Unterstützung der Besonderheiten im eHome möglich, z. B. bei der personen- und kontextbezogenen Konfigurierung. Der kontinuierliche SCD-Prozess wird auf Basis eines globalen Graphmodells in einem modellgetriebenen Entwicklungsprozess realisiert. Die *graphbasierten Techniken* ermöglichen dabei den einfachen Zugriff auf alle relevanten Daten. Das Graphmodell wird außerdem für die Werkzeugentwicklung genutzt. Ein weiterer wichtiger Beitrag dieser Arbeit ist der Laufzeitmechanismus zur *dynamischen Adaptergenerierung* auf Basis der semantischen Dienstbeschreibung. Dieser zeigt, wie mittels struktureller Reflection in Java eine Adaption zwischen syntaktisch inkompatiblen Schnittstellen erreicht werden kann. Weitere Ergebnisse sind ein Ansatz zur *Laufzeitüberwachung der Dienstkommunikation*, für dessen Realisierung auf die aspektorientierte Programmierung zurückgegriffen wird, und die Entwicklung von *Werkzeugen*, die sowohl die Dienstentwicklung als auch die Laufzeit des eHome-Systems abdecken.

Danksagung

Zum Entstehen der vorliegenden Arbeit haben zahlreiche Menschen beigetragen, bei denen ich mich an dieser Stelle für ihre Unterstützung bedanken möchte.

Zunächst gilt mein besonderer Dank Professor Manfred Nagl, der meine Arbeit als Doktorvater betreut und mir die Promotion an seinem Lehrstuhl ermöglicht hat. Seine Unterstützung meines Promotionsvorhabens, die anregenden Diskussionen aber auch der mir gewährte Freiraum haben meine Promotion zu einer lehrreichen Erfahrung gemacht. Ebenso danke ich Professor Kurt Geihs für sein Interesse an meiner Arbeit und die Übernahme des Zweitgutachtens. Professor Bernhard Rumpe gilt mein Dank dafür, dass er sich als weiterer Gutachter zur Verfügung gestellt hat.

Außerdem danke ich allen Mitgliedern des eHome-Projekts, die im Laufe der Zeit an diesem spannenden Thema mitgearbeitet und so zum Gelingen der Arbeit beigetragen haben. Besonderer Dank gilt meinem Kollegen Ibrahim Armaç für die vielen fruchtbaren Diskussionen, die richtungsweisenden Anregungen beim Korrekturlesen dieser Arbeit und die sehr gute Zusammenarbeit in all den Jahren. Meinen ehemaligen Kollegen im eHome-Projekt Michael Kirchhof und Ulrich Norbisrath danke ich für die Unterstützung in der Anfangsphase und ihre Vorarbeiten, die die Basis der vorliegenden Arbeit geliefert haben. Allen Studentinnen und Studenten, die im eHome-Projekt mitgearbeitet haben, danke ich für ihren Beitrag zum Erfolg des Projekts und damit auch dieser Arbeit. Mein besonderer Dank gilt dabei Ralf Frotscher, Marvin Hoffmann, Sven Kulle, Monika Pienkos, Mark Stegelmann und Chengzhi Xue. Die vielen intensiven Gespräche mit ihnen haben bedeutenden Einfluss auf die verschiedenen Themenbereiche dieser Arbeit genommen.

Während meiner Zeit am Lehrstuhl haben mich zahlreiche Kollegen begleitet. Ihnen allen möchte ich an dieser Stelle für das hervorragende Arbeitsklima und die vielen gemeinsamen Erlebnisse danken. Es war eine spannende und schöne Zeit, die mir dauerhaft in Erinnerung bleiben wird. Besonderer Dank gilt meinem ehemaligen Kollegen, Kommilitonen und Nachbarn Christof Mosler für die unterhaltsamen gemeinsamen Unternehmungen, seine tatkräftige Unterstützung sowie die kritische Durchsicht dieser Arbeit. Bodo Kraft danke ich für die vielen lehrreichen Ratschläge, die mir die Anfangszeit am Lehrstuhl erleichtert haben, und die gemeinsamen Konferenzreisen, die immer eine bereichernde Erfahrung während meiner Promotion waren. Auch den weiteren „alten“ Kollegen Simon Becker, Boris Böhlen, Christian Fuß, Thomas Haase, Markus Heller und Ulrike Ranger möchte ich für die vielen Tipps zur Promotion danken, die mir eine wertvolle Hilfe waren. Thomas Heer, Cem Mengi, Theresa Körtgen, Erhard Weinell und René Wörzberger danke ich für die gemeinsame Zeit, die fachlichen Gespräche und nicht zuletzt auch die vielen Erlebnisse außerhalb des akademischen Alltags. Allen „neuen“ Mitarbeitern, die zusammen mit Professor Bernhard Rumpe an den Lehrstuhl für Informatik 3 nach Aachen gekommen sind, wünsche ich viel Erfolg bei ihren Promotionsvorhaben und eine gute und lehrreiche Zeit.

Nicht zuletzt danke ich meiner Familie für die stete Unterstützung. Auf meine Eltern Klaus und Maria Retkowitz konnte ich immer zählen. Ihr Rückhalt hat mir geholfen, meinen Weg zu gehen. Darüber hinaus gilt meiner Mutter, Maria Retkowitz, besonderer Dank für die Korrekturhinweise zur vorliegenden Arbeit.

Meiner Freundin Nadine danke ich ganz herzlich für ihre Unterstützung und Geduld auch in den schwierigeren Phasen der Promotion. Sie hat jederzeit an mich geglaubt und mir den Ausgleich und das Glück während der vergangenen Jahre gegeben. Ohne ihre Unterstützung wäre diese Arbeit nicht möglich gewesen.

Aachen, Januar 2010 *Daniel Retkowitz*

Inhaltsverzeichnis

Kapitel 1

Einführung

In heutigen Wohnumgebungen werden zahlreiche elektronische Geräte für die verschiedensten Aufgaben des Alltags eingesetzt. Von der Steuerung einer Heizungsanlage über Küchengeräte bis hin zu mobiler Unterhaltungselektronik werden ganz unterschiedliche Bereiche abgedeckt. Die heute verwendeten Geräte bieten jeweils spezifische Funktionalitäten an, die entweder vollständig in Hardware implementiert sind, oder durch eine kombinierte Lösung aus Hardware und Software umgesetzt werden. Im letzteren Fall ist die Software an die zugehörige Hardware angepasst und lässt sich nur im Zusammenhang mit dieser spezifischen Hardware nutzen. Durch die enge Kopplung von Geräten und Funktionalitäten ist es kaum möglich, vorhandene Geräte für andere Funktionalitäten zu nutzen, als die vom Hersteller vorgesehenen. Umgekehrt gibt es auch keine Möglichkeit, Funktionalitäten losgelöst von einer spezifischen Hardware zu nutzen, etwa auf Basis verschiedener, zuvor nicht festgelegter Geräte, die je nach Umgebung ganz unterschiedlich sein können.

Ein weiterer Aspekt ist die fehlende Vernetzung unter den Geräten in heutigen Wohnumgebungen. Dies verhindert die Nutzung übergreifender Funktionalitäten, die auf mehreren verschiedenen Geräten basieren. Nur in Einzelfällen ist ein Zusammenspiel der Geräte problemlos möglich, z. B. wenn diese aus einer gemeinsamen Produktlinie eines bestimmten Herstellers stammen und auf Interoperabilität ausgelegt sind. Oder aber, es handelt sich um Geräte, die über eine sehr schmale und strikt standardisierte Schnittstelle miteinander kommunizieren. Ein typisches Beispiel für diesen Fall ist ein Fernsehgerät, das mit einem DVD-Player, einem Satellitenempfänger, einer Stereoanlage und anderen ähnlichen Geräten verbunden ist. Für diese existieren standardisierte und weit verbreitete Schnittstellen, die sich durch Stecker- und Buchsenformate physisch manifestieren. Die einzelnen Geräte werden durch Kabel miteinander verbunden, wobei im Allgemeinen nur Stecker und Buchsen des gleichen Standards zusammenpassen. Dadurch wird eine fehlerhafte Konfigurierung in den meisten Fällen verhindert. Der Datenaustausch über diese Verbindungen findet häufig auf einer sehr niedrigen Abstraktionsstufe statt. Daher gibt es auch nur eine geringe Flexibilität in Bezug auf die Kombinierbarkeit und den Zugriff auf Funktionalitäten durch andere Geräte.

Um die verschiedenen Geräte, die im Haus genutzt werden, zu einem Gesamtsystem zu integrieren, sind zahlreiche Hürden zu überwinden. Sollen Geräte vernetzt und eine Fernsteuerung ermöglicht werden, so wird meist von *Heimautomatisierung* (engl. *Home Automation*) gesprochen. Bereits vor rund 20 Jahren gab es Bestrebungen, durch Automatisierung und übergreifende Funktionalitäten mehr Komfort zu erreichen. In einem Artikel der Zeitschrift *Computer Live* [Gra90] wurde z. B. schon 1990 über ein Projekt in Tokio berichtet, bei dem es um die Automatisierung in einem Wohnhaus geht. Im beschriebenen Haus können Fenster und Türen elektronisch geöffnet und geschlossen sowie per Flüssigkristalltechnik abgedunkelt werden. In der Küche können die Kochplatten über einen Computer, abhängig vom ausgewählten Rezept, automatisch gesteuert werden. Die Dunstabzugshaube wird bei Bedarf automatisch ausgefahren und eingeschaltet. Im Bad werden über Sensoren in der WC-Brille Blutdruck und Herzfrequenz gemessen, bei ungewöhnlichen Werten werden diese an den Hausarzt weitergeleitet. Alle Funktionen können über die hausinterne Telefonanlage ferngesteuert werden. Dies sind typische Szenarien, die auch heute die Projekte zur Heimautomatisierung bestimmen. Allerdings gibt es inzwischen bereits sehr viel mehr Produkte zur Automatisierung, die am Markt allgemein verfügbar sind, z. B. in Elektronik- oder Baumärkten. Außerdem werden inzwischen neue Gebäude – sowohl im gewerblichen Bereich wie auch bei privaten Gebäuden – des öfteren bereits von vornherein mit Sensorik und Bussystemen ausgestattet. Aufgrund der höheren Kosten ist dies aber nach wie vor nicht der Standard und eine übergreifende Integration auch in heutigen Haushalten nicht üblich. Die Steuerung von Beleuchtung, Temperatur und Belüftung beispielsweise erfolgt im Allgemeinen nach wie vor manuell. Nur isolierte Geräte bieten eine gewisse

Automatisierung, wie etwa eine programmierbare elektronische Rollladensteuerungen oder über eine Zeitschaltuhr gesteuerte Lampen.

In einem anderen alltäglichen Bereich sieht die Situation diesbezüglich anders aus – nämlich im Automobil. In heutigen Fahrzeugen sind alle Teilbereiche, mit denen der Benutzer interagiert, automatisiert und integriert. Auch bei Kleinwagen gehört eine Funkfernbedienung, die das komfortable Öffnen und Schließen von Türen, Kofferraum und Fenstern ermöglicht, zum Standard. Radio, CD-Player und Navigationsgeräte arbeiten zusammen. Sie werden z. B. automatisch über das Starten und Stoppen des Motors ein- bzw. ausgeschaltet. Die Sprachausgabe des Navigationsgeräts führt zur Stummschaltung oder zu einer geringeren Lautstärke des Radios. Die Lautstärke kann insgesamt automatisch in Abhängigkeit von Geschwindigkeit und Fahrgeräusch geregelt werden. Es gibt zahlreiche Sensoren im gesamten Fahrzeug, z. B. für Fahrerassistenzsysteme oder andere Sicherheitssysteme, deren Daten für den Fahrer aufbereitet werden und über die er mittels Displays oder akustischer Signale informiert wird. Des weiteren sind in vielen Fahrzeugen bereits Schnittstellen integriert, die es ermöglichen, unterschiedliche Mobiltelefone oder MP3-Player an das Multimediasystem des Fahrzeugs anzuschließen und es so z. B. über Bedienelemente am Lenkrad zu steuern.

Obwohl Menschen im Durchschnitt wesentlich mehr Zeit zu Hause und in Gebäuden im Allgemeinen verbringen als in ihrem Auto, gibt es dort bisher kaum Vernetzung, Integration und Automatisierung. Ein Grund ist natürlich, dass ein Automobilhersteller das komplette Fahrzeug zusammenstellt und von seinen Zulieferern Teile nach genauen Anforderungen ordern kann. So kann der Hersteller genau kontrollieren, welche Funktionalitäten im Fahrzeug verfügbar sein sollen, welche Hardware dazu verwendet wird und wie die entsprechenden Schnittstellen aussehen müssen. Diese Situation ist bei der Ausstattung eines Hauses mit Gebäudetechnik und Haushaltselektronik nicht gegeben. Dennoch ist absehbar, dass es in Zukunft auch in Wohnhäusern eine zunehmende Integration bisher isolierter Systeme und damit die Möglichkeit übergreifender Funktionalitäten geben wird. Dabei werden zunächst einfache aber nützliche Funktionalitäten im Vordergrund stehen, z. B. das Aktivieren einer Abwesenheitsfunktion, durch die in einem einzigen Schritt zahlreiche kombinierte Aktionen automatisch ausgeführt werden, beispielsweise das Ausschalten von Geräten wie einer Stereoanlage oder von Küchengeräten, das Ausschalten der Beleuchtung, das Schließen von Fenstern und der Haustür, das Aktivieren einer Alarmanlage etc. Darauf aufbauend sind komplexere Funktionalitäten denkbar, die z. B. verschiedene Beleuchtungs- oder Klimatisierungsszenarien realisieren, die Fernsteuerung der Systeme im Haus ermöglichen, zur Steigerung der Energieeffizienz dienen oder die medizinische Überwachung älterer Menschen unterstützen.

1.1 Motivation

In dieser Arbeit werden sogenannte *eHomes*, auch *Smart Homes* genannt, betrachtet. Damit werden Wohnumgebungen bezeichnet, in denen oben geschilderte Szenarien unterstützt werden und den Bewohnern geräteübergreifende, in Software realisierte *Funktionalitäten* durch *Dienste* zur Verfügung gestellt werden. Die Ziele gehen dabei über die bisher verfügbaren Ansätze zur Heimautomatisierung hinaus. Durch die Entkopplung von Funktionalität und Hardware wird die Realisierung komplexer integrierter Dienste in einer *vernetzten Umgebung* ermöglicht. Zum einen können Funktionalitäten auf Basis ganz unterschiedlicher Hardware genutzt werden und zum anderen werden Funktionalitäten damit auch mobil und können innerhalb der Umgebung und in andere Umgebungen mitgenommen werden. Während die Heimautomatisierung häufig auf Hardware zentriert ist und aus einer Reihe voneinander unabhängiger Insellösungen für Teilprobleme basiert, wird in eHome-Systemen eine die gesamte Umgebung umfassende Infrastruktur vorausgesetzt. Diese hat die Aufgabe, übergreifende Funktionalitäten durch integrierende Dienste zu ermöglichen. Viele der Konzepte, die sich auf eHomes beziehen, sind nicht nur in Wohnumgebungen anwendbar, sondern auch in anderen Umgebungen, wie etwa Büros, Hotels, Krankenhäusern, Einkaufszentren, Bahnhöfen und anderen öffentlichen Bereichen. Dann wird auch von *Smart Buildings* gesprochen.

Im Bereich der Softwaretechnik gibt es auf der einen Seite die klassischen Ansätze der Softwareentwicklung, die sich besonders für *statische Systeme* eignen, also für Systeme, deren Aufbau und Zusammensetzung sich zur Laufzeit nicht verändert. Für diesen Typ von Systemen ist es möglich, bereits im Entwicklungsprozess eine Architektur mit detaillierten Festlegungen über das System zu erstellen. Die Teilsysteme und Module aus denen das System besteht sind bekannt und auch deren Zusammenspiel kann bereits festgelegt werden. Im Fall von eHome-Systemen ist dieser Ansatz jedoch nicht anwendbar, wenn nicht für jedes individuelle eHome ein eigener Entwicklungsprozess durchgeführt werden soll. Dies ist aus Zeit- und Kostengründen in der Praxis nicht durchführbar.

Auf der anderen Seite gibt es serviceorientierte Ansätze, die in *dynamischen Systemen* Anwendung finden. Der Aufbau solcher Systeme ergibt sich im Wesentlichen erst zur Laufzeit. Es können oder sollen im Vorhinein keine Festlegungen darüber gemacht werden, aus welchen Teilen sich das System zusammensetzt und wie diese Teile zueinander in Verbindung stehen. Ein System dieses Typs besteht aus *Diensten* (engl. *Services*). Benötigt ein Dienst zum Ausüben seiner Funktionalität andere Dienste, so muss er solche Dienste in einem Verzeichnis suchen, um dann auf die benötigten Funktionalitäten zurückgreifen zu können. Der Aufbau des späteren Systems ergibt sich somit implizit und ist außerdem fortlaufenden Änderungen unterworfen. Ein wesentliches Problem dieses Ansatzes ist, dass im Vorhinein kaum verlässliche Aussagen über die Funktionsfähigkeit des Systems gemacht werden können. Es ist nicht davon auszugehen, dass immer geeignete Dienste zur Verfügung stehen, die auch noch über eine passende Schnittstelle und ein kompatibles Kommunikationsprotokoll verfügen. Darüber hinaus gibt es zur Laufzeit des Systems üblicherweise keine Einflussmöglichkeiten auf das Zusammenwirken der Dienste. In eHomes ist es aber erforderlich, dass Bewohner oder ein Administrator ggfs. Anpassungen vornehmen können. Ein weiterer Aspekt ist der hohe Entwicklungsaufwand, der sich ergibt, wenn jeder Dienstentwickler aufs Neue die Suche nach benötigten Diensten und die Verwaltung von Dienstabhängigkeiten implementieren muss. Diese Problemstellungen müssen adressiert werden, bevor eHome-Systeme in der Praxis zum Einsatz kommen können.

Ein spezifischer Entwicklungs- und Installationsprozess für jedes individuelle Gebäude, wie bei der Heimautomatisierung bislang üblich, ist zu kostenintensiv, als dass eHomes auf diesem Wege Verbreitung finden könnten. Wie oben bereits erläutert wird flexible und adaptive Software benötigt, um geräteübergreifende Dienste zu realisieren. Daher bietet es sich an, Dienste als *Standardkomponenten* zu entwickeln, die dann angepasst an die spezifischen Benutzeranforderungen und die jeweilige eHome-Umgebung verwendet werden können. Damit dies ohne manuelle Eingriffe möglich ist, muss eine entsprechende Infrastruktur vorhanden sein, die die Konfiguration und die Adaption der Dienste an den jeweiligen

Kontext unterstützt. Um den Entwicklungsprozess zu vereinfachen und unnötige Mehrfachentwicklung zu verhindern, wird eine *Middleware*-Infrastruktur benötigt, die die Basis für Ausführung, Interaktion und Kommunikation der eHome-Dienste bildet. Eine solche Infrastruktur kann darüber hinaus auch Mechanismen für *kontextbezogene und personalisierte Dienste* zur Verfügung stellen. Durch die Verschiebung von Standard- und Querschnittsfunktionalitäten auf eine Middleware-Ebene, wird der Entwicklungsaufwand für eHome-Dienste reduziert, was zu sinkenden Kosten und vor allem geringerer Komplexität und Fehleranfälligkeit bei der Entwicklung führt und damit eine größere Verbreitung von eHomes ermöglicht.

Eine der größten Herausforderungen ist die Unterstützung einer dynamischen, kontextbezogenen *Dienstkomposition* in einem *heterogenen Dienstumfeld*. eHomes müssen sich den Bewohnern anpassen und auf Veränderungen in ihrem Umfeld reagieren. Ein wesentlicher Einflussfaktor auf *dynamische Veränderungen* ist die *Mobilität* von Benutzern und Geräten. Eine geeignete Laufzeitumgebung für eHome-Dienste muss also entsprechende Mechanismen für Anpassungen zur Laufzeit bereitstellen, damit diese nicht für jeden Dienst erneut entwickelt werden müssen. Neben unnötig hohem Entwicklungsaufwand und Fehleranfälligkeit würde eine Mehrfachentwicklung auch zu inkonsistentem Verhalten der Dienste führen, da nicht jeder Dienst in gleicher Weise auf Veränderungen reagieren würde. Dies gilt es zu vermeiden. Auch in Zukunft werden die Dienste, die in einem spezifischen eHome zur Anwendung kommen, nicht von einem einzigen Anbieter stammen. Da kein globaler Standard existiert, sind die Dienste verschiedener Hersteller nicht notwendigerweise zueinander kompatibel. Dennoch müssen auch Dienste mit unterschiedlichen Schnittstellen und Implementierungen komponierbar sein, um ein funktionierendes eHome-System zu ermöglichen. Eine Laufzeitumgebung für eHome-Dienste muss daher die *Adaption* von Diensten unterstützen, damit eine Komposition überhaupt möglich ist.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es, neue Konzepte für die *Laufzeitunterstützung* von eHome-Diensten zu entwickeln, die insbesondere die Aspekte *Mobilität* und *Dynamik* und die in eHomes auftretende *Heterogenität* berücksichtigen. Die dynamische Konfigurierung des eHome-Systems ist ein zentraler Punkt der Überlegungen. Kernaufgaben sind die Komposition und Adaption der eHome-Dienste und die Erfassung und Berücksichtigung der zur Laufzeit erforderlichen Anpassungen im System, die sich unter anderem aus der Mobilität von Benutzern und tragbaren Geräten ergeben sowie aus sich ändernden Benutzeranforderungen. Die geringe Standardisierung und die damit einhergehende Heterogenität des Anwendungsgebiets stellen eine zusätzliche Herausforderung dar.

Die in der vorliegenden Arbeit verfolgten Ziele bestehen zusammengefasst aus den folgenden Hauptaspekten:

- Entwicklung einer **Laufzeitumgebung** für eHome-Dienste.
- Unterstützung von **Mobilität** und **Dynamik**.
- Berücksichtigung der **Heterogenität** der Anwendungsdomäne.
- Entwicklung von **Werkzeugen** zur Realisierung des Gesamtprozesses.

Die zu entwickelnde Infrastruktur muss Mechanismen zur Erleichterung der Dienstentwicklung bereitstellen, sodass bei der Entwicklung einzelner Dienste nicht die gesamte Komplexität des eHome-Systems betrachtet werden muss. Querschnittsfunktionalitäten, wie die Verwaltung der Dienstabhängigkeiten oder die Berücksichtigung von Kontextänderungen, sollten daher auf die Ebene der Laufzeitumgebung verlagert werden. Dienste können dann auf diese Funktionalitäten zurückgreifen, sodass keine wiederholte Neuentwicklung erforderlich ist. Die zur Lösung dieser Aufgaben erarbeiteten Konzepte werden in Form eines Prototypen implementiert, um die Anwendbarkeit der Konzepte zu evaluieren.

Im Rahmen dieser Arbeit wird insbesondere auch der Entwicklungsprozess von eHome-Systemen betrachtet, sowohl in Bezug auf die Dienstentwicklung als auch das eHome selbst. Dabei wird untersucht, welche Schritte Teil der Entwicklung und Installation eines eHomes sind und welche erst zur Laufzeit Anwendung finden. Hierbei sind insbesondere die Unterschiede zwischen der klassischen Entwicklung eines Softwaresystems und einem dynamischen serviceorientierten System von Bedeutung. Die im Rahmen der Prototypentwicklung zu erstellenden Werkzeuge müssen sich ebenfalls an den verschiedenen Phasen des Gesamtprozesses orientieren, damit sie für die Aufgaben der jeweiligen Phase

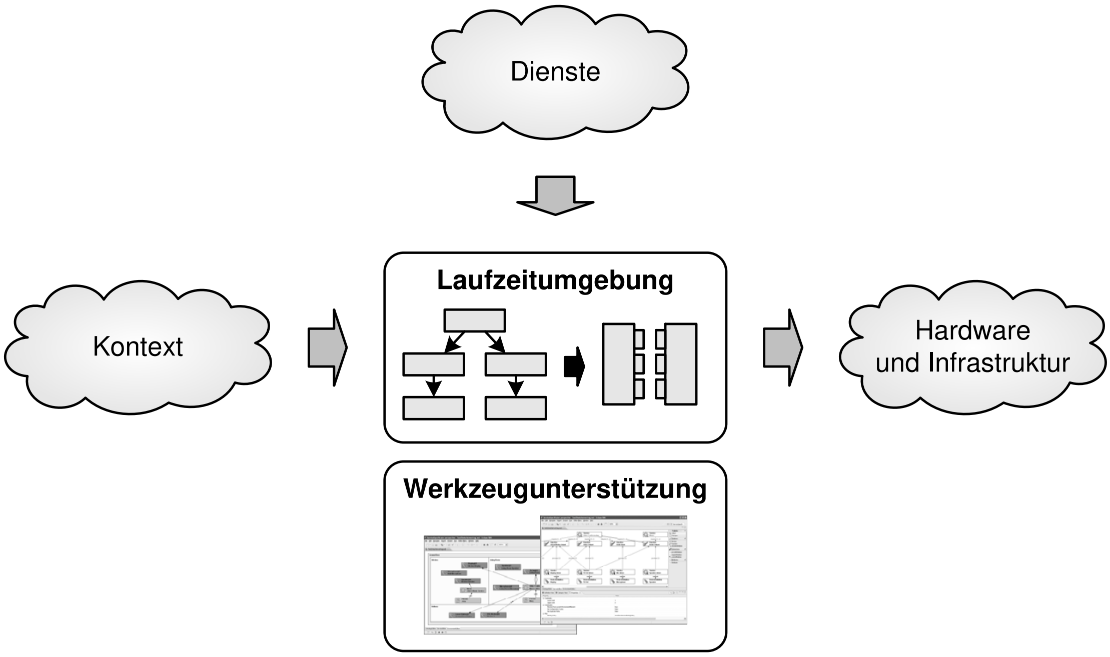


Abbildung 1.1: Softwareunterstützung für eHome-Systeme.

eine geeignete Unterstützung bieten können. Die Phasen, die zur Laufzeit ausgeführt werden, müssen in großen Teilen automatisch ablaufen, da es in eHome-Systemen keinen Administrator gibt, der dauerhaft verfügbar ist und jederzeit in das System eingreifen kann. Stattdessen müssen diejenigen Aufgaben, die vom System nicht automatisch verarbeitet werden können oder sollen, von den Benutzern des eHomes interaktiv gelöst werden können. Die Werkzeuge müssen daher auch zur Laufzeit entsprechende manuelle Eingriffe in das System ermöglichen. Abbildung 1.1 zeigt schematisch die Bereiche, die in dieser Arbeit betrachtet bzw. ausgeklammert werden.

In dieser Arbeit betrachtete Bereiche

Im Zentrum der Arbeit steht die Entwicklung von Konzepten für eine *Laufzeitumgebung*, die die Ausführung von eHome-Diensten unterstützt und dabei den Entwicklungsprozess der Dienste und des eHome-Systems als Ganzes berücksichtigt. Dabei werden insbesondere die dynamische Anpassung an Kontextänderungen und die Adaption heterogener Dienste betrachtet.

Zur Anwendung der Laufzeitumgebung werden *Werkzeuge* benötigt, die eine Interaktion durch den Benutzer und eine Administration des eHome-Systems zulassen. Die Laufzeitumgebung wird in Form eines Prototypen realisiert, der verschiedene Arten der Adaption unterstützt, um die dynamischen Veränderungen im eHome-System zu erfassen und Kompatibilität zwischen heterogenen Diensten herzustellen. Um die Verwaltung von Diensten durch die Laufzeitumgebung zu ermöglichen, werden Dienstbeschreibungen benötigt, die alle relevanten Daten eines Dienstes enthalten. Auch für die Erstellung dieser Dienstbeschreibungen werden geeignete Werkzeuge benötigt, deren Entwicklung ebenfalls Gegenstand dieser Arbeit ist. Zur Überprüfung der Anwendbarkeit werden die entwickelten Konzepte anhand einiger Beispielszenarien untersucht.

In dieser Arbeit ausgeklammerte Bereiche

Dienste, die auf Basis der Laufzeitumgebung ausgeführt werden sollen, müssen gemäß einiger Vorgaben entwickelt werden. Insbesondere ist die Dienstspezifikation von Bedeutung, es können also nicht beliebige Dienste verwendet werden. Im Rahmen dieser Arbeit werden verschiedene Dienste betrachtet und die Entwicklung von Diensten spielt auch eine wichtige Rolle für die Konzepte der Laufzeitumgebung und des Entwicklungsprozesses, dennoch dienen die Dienste hier hauptsächlich der Evaluation. Das Entwickeln bis ins Detail ausgearbeiteter Dienste für den praktischen Einsatz steht nicht im Zentrum dieser Arbeit.

Die Adaption der Konfiguration eines eHome-Systems dient dazu, Änderungen des *Kontextes* zur Laufzeit abzubilden. Dazu müssen die relevanten Kontextinformationen zunächst erfasst werden. Besonders wichtig sind dabei räumliche Informationen wie der Aufenthaltsort von Bewohnern und den zu nutzenden Geräten in der eHome-Umgebung. Die Erfassung dieser Informationen wird in dieser Arbeit ebenfalls nicht explizit betrachtet. Es gibt verschiedene technische Möglichkeiten, räumliche Informationen zu erfassen [WHFG92, Tek05, NLLP04, Ait03]. Diese werden hier nicht näher analysiert, stattdessen wird lediglich die Verarbeitung der daraus gewonnenen Informationen betrachtet.

Damit Dienste in einer eHome-Umgebung ausgeführt werden können, muss die verfügbare *Hardware* in das System eingebunden und für die Laufzeitumgebung verfügbar sein. Es existieren viele verschiedene *Infrastrukturen*, über die Hardware in einem Gebäude vernetzt werden kann. Eine Betrachtung, wie die verschiedenen Infrastrukturtechniken eingebunden werden können, um die vorhandenen Geräte mit der Laufzeitumgebung zu vernetzten, wird in dieser Arbeit ebenfalls nicht vorgenommen. Solche Ansätze finden sich beispielsweise in [GVR+07]. Für die untersuchten Konzepte genügt es anzunehmen, dass die vorhandene Hardware über passende Treiberdienste im eHome-System verfügbar gemacht wird, sodass eine Nutzung mittels dieser Dienste erfolgen kann.

1.3 Projektkontext

Die vorliegende Arbeit entstand im Rahmen des eHome-Projekts am Lehrstuhl für Informatik 3 (Softwaretechnik) der RWTH Aachen, das sich mit der softwaretechnischen Unterstützung von eHome-Systemen und den zugehörigen Entwicklungsprozessen befasst. In bisherigen Arbeiten des Projekts wurden dazu bereits verschiedene Ansätze entwickelt.

In der Arbeit von NORBISRATH wurden Konzepte zur Verwendung von Standardkomponenten für die Implementierung von eHome-Diensten entwickelt [Nor07]. Das Ziel der Arbeit war es, eine eHome-spezifische Konfigurierung auf Basis gegebener Standardkomponenten und eines Umgebungsmodells durchzuführen. Dadurch konnte der Grad an Wiederverwendung erhöht werden, was sich positiv auf die Kosten eines einzelnen eHomes auswirkt. Dazu wurde ein graphbasiertes Werkzeug entwickelt, der sogenannte *eHomeConfigurator*. Dieser unterstützt die Konfigurierung und wurde mittels der modellbasierten Entwicklungsumgebung *Fujaba* [FNTZ98] realisiert. Der eHomeConfigurator wird jedoch im Rahmen dieser Arbeit durch neue Werkzeuge ersetzt, die insbesondere auch die Laufzeitphase des eHome-Systems unterstützen, was mit dem bisherigen Werkzeug nicht möglich ist. Die Ergebnisse der Vorarbeit konnten jedoch teilweise auch wiederverwendet werden. Wie schon im Ansatz von NORBISRATH wird auch in dieser Arbeit eine Graphstruktur zur Repräsentation des eHome-Systems verwendet. Das zugrunde liegende Datenmodell konnte im Kern beibehalten werden, es wurden jedoch an vielen Stellen Erweiterungen und Anpassungen vorgenommen.

Eines der Ergebnisse der Arbeit von NORBISRATH ist der sogenannte *SCD-Prozess*. Der SCD-Prozess besteht aus den Phasen *Spezifizierung*, *Konfigurierung* und *Deployment*, die als eHome-spezifischer Teil des Entwicklungsprozesses einmalig im Rahmen der Installation des Systems durchgeführt werden. Die unabhängig von einem individuellen eHome entwickelten Dienstkomponenten werden im SCD-Prozess zu einem Gesamtsystem zusammengesetzt und dann im eHome zur Ausführung gebracht. Der SCD-Prozess wird durch den eHomeConfigurator realisiert [NMA06]. Die vorliegende Arbeit setzt auf dem SCD-Prozess auf, dessen Konzepte angepasst und erweitert werden. Der eHomeConfigurator wird dabei durch neue Werkzeuge abgelöst.

In einer weiteren Vorarbeit von KIRCHHOF wurde insbesondere der Geschäftsprozess untersucht, der zwischen Anbietern von eHome-Diensten und den Verbrauchern, d. h. den Bewohnern von eHomes, abläuft [Kir05]. Dabei wurde der gesamte Lebenszyklus der eHome-Dienste betrachtet, um eine bessere Unterstützung der Dienstentwicklung zu erreichen. Ziel war es, die Aufgaben der Dienstentwickler auf die Implementierung der Kernfunktionalitäten zu beschränken. Dazu wurde unter anderem ein Ansatz zur regelbasierten Dienstentwicklung und Konflikterkennung entwickelt. Als Basisinfrastruktur wurde das dreischichtige Architekturmodell *PowerArchitecture* entwickelt, welches der Abstraktion von konkreten Hardwarefunktionalitäten dient und somit die Dienstentwicklung vereinfachen soll.

Parallel zu der vorliegenden Arbeit wird im Rahmen des eHome-Projekts auch an Konzepten zur mobilen Dienstmitnahme und den damit zusammenhängenden Fragestellungen der Sicherheit und Privatsphäre gearbeitet. Dazu wurden verschiedene Ansätze von ARMAÇ entwickelt [APPR09, AE08, AR08, Arm08]. Die Konzepte der vorliegenden Arbeit sind als grundlegende Basistechnologien für die Arbeit von ARMAÇ zu betrachten. Die dynamische Konfigurierung und der Adaptionsansatz zur Komposition heterogener Dienste stellen eine Grundlage für mobile Dienste dar. Darauf aufbauend können Dienste, die von Benutzern mitgeführt werden, zur Laufzeit in das laufende eHome-System integriert werden. Die Wahrung der Privatsphäre benutzerbezogener Profildaten sowie die Sicherheit der Dienste und des eHome-Systems selbst sind daraus resultierende Problemfelder, die in der Arbeit von ARMAÇ adressiert werden. Diese Fragestellungen werden daher in dieser Arbeit nicht näher betrachtet.

1.4 Lösungsansatz

In dieser Arbeit werden zwei Arten von Adaption betrachtet, um die in Abschnitt 1.2 skizzierten Ziele zu erreichen, die *strukturelle Adaption* und die *semantische Adaption*. Beide Adaptionsarten werden zu einem Gesamtansatz kombiniert.

Strukturelle Adaption: Die strukturelle Adaption bezeichnet die Anpassung der Dienstkomposition im eHome zur Laufzeit. So können Änderungen des Kontextes, die sich aus der *Mobilität* und *Dynamik* im eHome ergeben, aber auch Änderungen der individuellen Benutzeranforderungen zur Laufzeit berücksichtigt werden. Die Softwarearchitektur des eHome-Systems, die sich aus der Dienstkomposition ergibt, wird somit zur Laufzeit anpassbar. Der von NORBISRATH [Nor07] beschriebene Entwicklungsprozess für eHome-Systeme wurde angepasst, sodass eine strukturelle Adaption ermöglicht wird. Dazu wurde der SCD-Prozess von der Installationsphase in die Laufzeitphase des eHome-Systems verlagert.

Semantische Adaption: Die semantische Adaption dient dazu, eine Interoperabilität zwischen Diensten herzustellen, die aufgrund fehlender Standardisierung nicht syntaktisch zueinander kompatibel sind. Aufgrund der *Heterogenität* in eHome-Systemen, die sich aus verschiedenster Hardware und Software zusammensetzen, können Inkompatibilitäten auftreten, beispielsweise bei der Verwendung von Diensten unterschiedlicher Hersteller. In der vorliegenden Arbeit wird ein Ansatz entwickelt, in dem der Komposition von Diensten anstatt der syntaktischen Dienstschnittstellen eine semantische Beschreibung der Dienstfunktionalitäten zugrunde liegt. Für eine Komposition ist daher keine gemeinsame syntaktische Schnittstelle Voraussetzung, sondern eine semantische Übereinstimmung der Dienstfunktionalitäten.

Sowohl strukturelle Adaption als auch semantische Adaption werden durch eine *Laufzeitumgebung für eHome-Dienste* unterstützt, die im Rahmen dieser Arbeit entwickelt wurde. Wie schon im Ansatz von NORBISRATH wird eine Graphstruktur, das sogenannte *eHome-Modell*, zur Repräsentation des Laufzeitzustands des eHome-Systems verwendet. Die nötigen Transformationen auf der Graphstruktur werden ebenfalls graphbasiert spezifiziert. Auf diese Weise wird die zugrunde liegende Anwendungslogik der Laufzeitumgebung realisiert, die die Basis für die in dieser Arbeit entwickelten Werkzeuge bildet. Die Werkzeuge dienen der Unterstützung des Entwicklungsprozesses und der Administration des eHome-Systems zur Laufzeit.

Dienstentwicklung

Zunächst wird die *Dienstentwicklung* betrachtet. Neben der *Implementierung* eines Dienstes, für die in dieser Arbeit die *OSGi Service Plattform* [WHKL08] als Komponentenplattform eingesetzt wird, muss jeder Dienst zusätzliche eine *Spezifikation* mitbringen, damit er im eHome-System verwendet werden kann. Diese Spezifikation beinhaltet unter anderem Informationen darüber, welche Funktionalitäten ein Dienst zur Verfügung stellt und welche anderen Funktionalitäten er benötigt. Eine solche Spezifikation kann vom Hersteller im Zusammenhang mit der Dienstimplementierung erstellt werden, es ist aber auch möglich, die Spezifikation erst nachträglich zu erstellen.

In dieser Arbeit wird die bisherige Dienstspezifikation um verschiedene Aspekte erweitert. Die Laufzeitumgebung berücksichtigt die entsprechenden Vorgaben dann bei der dynamischen Konfigurierung. Zum einen kann ein bestimmtes Kompositionsverhalten durch sogenannte *Bindungsstrategien* spezifiziert werden. Zum anderen können durch *Bindungsbeschränkungen* kontextbezogene Einschränkungen festgelegt werden, die bestimmte Konfigurationsmuster ausschließen. Dieses Konzept bietet die Möglichkeit, die spätere dynamische Konfigurierung zu beeinflussen, was für einen automatischen Ablauf notwendig ist.

Zur Unterstützung der dynamischen *Komponentenadaption* wurde die Dienstspezifikation um eine *semantische Beschreibung* erweitert. Darin werden die Elemente der Dienstschnittstellen durch eine semantische Abbildung mit den Konzepten einer zuvor entwickelten *domänenspezifischen Ontologie* in Relation gebracht. Auf diese Weise kann die Dienstkomposition durch Vergleich der semantischen Eigenschaften von Diensten erfolgen, im Gegensatz zur syntaktischen Komposition, die auf den Dienstschnittstellen basiert. Die Komposition von Diensten wird

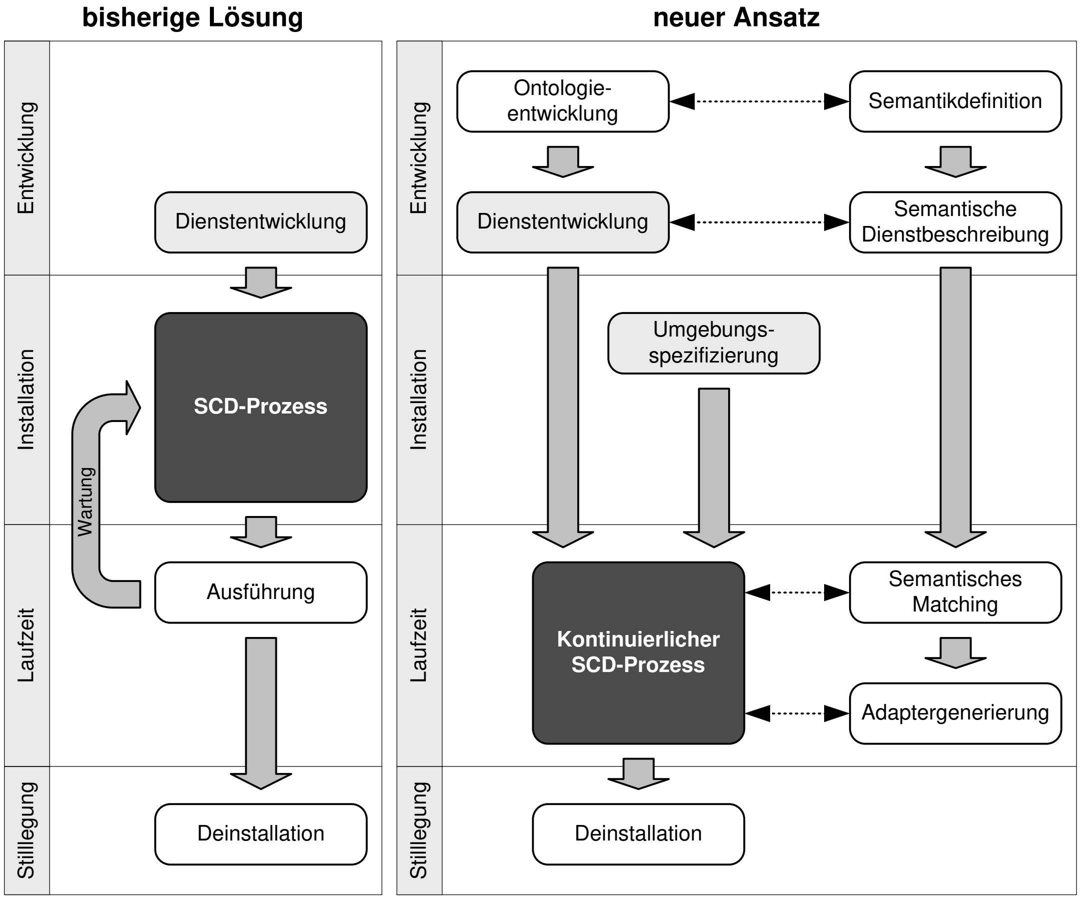


Abbildung 1.2: Übersicht des neuen Ansatzes.

dadurch flexibler, da auch bei inkompatiblen Schnittstellen eine Interaktion zwischen den Diensten möglich ist.

Laufzeitunterstützung

Eine wesentliche Veränderung, die im Rahmen dieser Arbeit durchgeführt wurde, ist die Transformation des von NORBISRATH eingeführten *SCD-Prozesses* zu einem *kontinuierlichen SCD-Prozess*. Damit geht eine Verschiebung der Konfigurierung des eHome-Systems von der *Installationsphase* hin zur *Laufzeitphase* einher. Diese Verschiebung ist erforderlich, um dynamische Anpassungen zu ermöglichen.

In Abbildung 1.2 wird dies verdeutlicht. Auf der linken Seite ist die bisherige Lösung dargestellt. Nach der Dienstentwicklung findet eine Installation im Rahmen des SCD-Prozesses statt, der die ausgewählten Dienste an das spezifische eHome anpasst, um sie dort zur Ausführung zu bringen. Die Phasen des SCD-Prozesses wurden bisher einmalig vor dem Start des eHome-Systems ausgeführt. Nach dem Deployment war das System in Betrieb, eine Anpassung der Konfiguration war dann nicht mehr möglich. Kontextbezogene Dienste waren daher nur begrenzt realisierbar. Dazu musste ein Dienst alle relevanten Kontextinformationen durch Bindung geeigneter Sensoren selbst erfassen und alle Ressourcen, die zur Nutzung verwendet werden sollen, zuvor gebunden haben.

Durch den in dieser Arbeit entwickelten *kontinuierlichen SCD-Prozess* wird eine Anpassung der Konfiguration zur Laufzeit ermöglicht. Die *Komposition* der Dienste wird nun fortwährend im laufenden Betrieb angepasst, wenn sich der Umgebungskontext aufgrund von Mobilität und Dynamik im eHome ändert oder Benutzeranforderungen modifiziert werden. Neben der Auswahl von Diensten und der Verfügbarkeit von Geräten sind auch manuelle Anpassungen bezüglich der Komposition möglich. Der neue Prozess ist in Abbildung 1.2 im rechten Teil dargestellt. Nach der Entwicklungsphase und einer Spezifizierung der Umgebung wird der kontinuierliche SCD-Prozess gestartet und unterstützt die Laufzeitphase des eHome-Systems. Somit können Anpassungen der Spezifikation berücksichtigt werden, aber auch eine automatische Anpassung aufgrund von Kontextänderungen ist möglich. Dies vereinfacht die Entwicklung kontextbezogener Dienste wesentlich und ist ein entscheidender Vorteil für die Entwicklung kostengünstiger Software für eHome-Systeme. Die *strukturelle Adaption* bezeichnet genau diese Anpassungsschritte des Systems zur Laufzeit.

Darüber hinaus wurde die *semantische Adaption* in den kontinuierlichen SCD-Prozess integriert. Dies ist am rechten Rand der Abbildung 1.2 zu sehen. Bei der Suche nach passenden Diensten wird während der Konfigurierungsphase auf eine semantische Dienstbeschreibung zurückgegriffen. So besteht eine Grundlage für die Komposition, die nicht implizit eine Kompatibilität voraussetzt aber auch keinen expliziten syntaktischen Standard bei der Dienstentwicklung vorschreibt, der in der Praxis nicht gegeben ist. Durch die semantische Dienstbeschreibung in Bezug auf eine vorgegebene Ontologie existiert eine klar definierte Grundlage für die Komposition. So können auch Dienste komponiert werden, die keine gemeinsame syntaktische Schnittstelle haben. Bevor eine Konfiguration deployt werden kann, müssen ggfs. Adapter für Dienstbindungen erzeugt werden, wenn syntaktische Inkompatibilitäten vorliegen. Aus den semantischen Dienstbeschreibungen kann die Laufzeitumgebung diese Adapterkomponenten automatisch erzeugen und an den nötigen Stellen in die Dienstkomposition einfügen.

Werkzeuge

Zur Unterstützung des Lösungsansatzes wurden in dieser Arbeit verschiedene Werkzeuge entwickelt. Diese decken die bereits in Abbildung 1.2 dargestellten Phasen ab.

- **Ontologie-Editor.** Die eHome-Ontologie, die zur semantischen Dienstbeschreibung notwendig ist, wird mit Hilfe des Werkzeugs *Protégé* [GMF+02] entwickelt. Es war daher keine Neuentwicklung eines eigenen Werkzeugs für diesen Zweck erforderlich.

- **Service-Editor.** Um die erweiterte Dienstbeschreibung einschließlich der semantischen Abbildung zu erstellen, wurde ein *Service-Editor* für die Spezifizierung von Diensten entwickelt. Dieser erlaubt es, alle relevanten Informationen zu einem Dienst festzulegen, sodass dieser mit der neuen Laufzeitumgebung verwendet werden kann. Für die semantische Beschreibung wird die mit dem Ontologie-Editor entwickelte Ontologie verwendet.

- **Laufzeit-Manager.** Zur Administration des eHomes und zur Interaktion des Benutzers mit der Laufzeitumgebung wurde ein weiteres Werkzeug entwickelt, das den bisher eingesetzten *eHomeConfigurator* ablöst. Dieses neue Werkzeug ist der sogenannte *Laufzeit-Manager*, der eine zentrale Rolle bei der Realisierung der Konzepte dieser Arbeit spielt. Der Laufzeit-Manager ermöglicht eine Visualisierung der momentanen Konfiguration sowie eine manuelle Anpassung dieser Konfiguration, wenn dies erforderlich oder gewünscht ist.

- **Umgebungseditor.** Für die Umgebungsspezifikation zur Kontextverarbeitung wurde ein *Umgebungseditor* entwickelt. Dieser ermöglicht eine Festlegung der räumlichen Kontextinformationen der eHome-Umgebung.

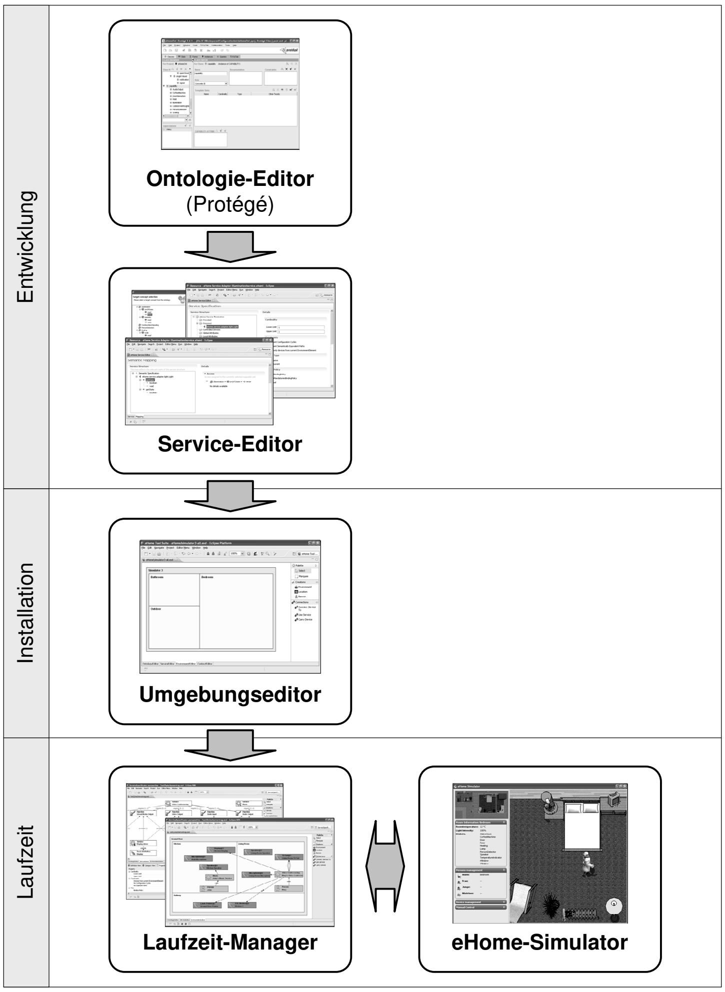


Abbildung 1.3: Übersicht in dieser Arbeit entwickelter Werkzeuge.

- **eHome-Simulator.** Schließlich wurde zu Evaluationszwecken im Rahmen eines Studentenprojekts eine Simulationssoftware für eHome-Umgebungen entwickelt, wodurch die neuen Konzepte dieser Arbeit in verschiedenen Szenarien untersucht werden konnten. Der Vorteil dieses *eHome-Simulators* liegt dabei insbesondere in der gewonnenen Flexibilität, die es ermöglicht, ohne zusätzliche Anschaffungskosten auch kurzfristig neue Testszenarien zu realisieren.

Eine Übersicht der oben beschriebenen Werkzeuge, die im Rahmen der vorliegenden Arbeit entwickelt wurden, ist in Abbildung 1.3 dargestellt. Die Werkzeuge sind den verschiedenen Phasen des Gesamtablaufs zugeordnet. Die Reihenfolge ihrer Verwendung wird durch die Pfeile in der Abbildung verdeutlicht.

1.5 Gliederung der Arbeit

In diesem Abschnitt wird die Struktur der weiteren Kapitel dieser Arbeit kurz erläutert.

- **Kapitel 2** führt in das Thema *eHome-Systeme* ein und motiviert diese Arbeit. Dabei wird auf die Hintergründe der Thematik und die damit zusammenhängenden Bereiche sowie die verwendete Terminologie eingegangen. Es wird außerdem auf die Anforderungen an eHome-Systeme aus Sicht der Benutzer eingegangen, gefolgt von einer Betrachtung der daraus resultierenden technischen Anforderungen an solche Systeme. Zur Illustration der Betrachtungen werden drei charakteristische Anwendungsszenarien für eHomes untersucht, die verschiedene Bereiche der Benutzeranforderungen abdecken. Diese Szenarien werden in Ausschnitten auch in späteren Teilen der Arbeit zur Veranschaulichung der entwickelten Konzepte wieder aufgegriffen. Schließlich werden einige Beispielprojekte vorgestellt, in denen prototypische eHomes in der Praxis umgesetzt wurden und es wird untersucht, welche der in diesen Projekten gewonnenen Erfahrungen für diese Arbeit von Bedeutung sind.

- **Kapitel 3** erläutert die Grundlagen dieser Arbeit, wobei insbesondere auch auf die Vorarbeiten im eHome-Projekt eingegangen wird. Zunächst wird die komponentenbasierte Softwareentwicklung diskutiert und es werden einige Komponentenmodelle und Beispielarchitekturen erläutert. Es folgt der Übergang zu serviceorientierten Ansätzen. Dabei wird auf die Unterschiede zur rein komponentenbasierten Entwicklung eingegangen, die sich auch in den Erweiterungen dieser Arbeit ausgehend von den Ergebnissen von NORBISRATH widerspiegeln. Auf die Zusammenhänge mit Middleware-Infrastrukturen wird ebenfalls eingegangen. Anschließend werden Grundlagen der modellgetriebenen Softwareentwicklung diskutiert, die bei der Umsetzung der in dieser Arbeit entwickelten Konzepte angewandt wurde.

Dies ist insbesondere für die graphbasierte Modellierung des Laufzeitzustands im eHome von Bedeutung, sowie die darauf aufbauende Entwicklung der Werkzeuge.

- **Kapitel 4** beschreibt den Lösungsansatz zur Unterstützung der *strukturellen Adaption* im Detail. Zunächst wird genauer auf die Problemstellung eingegangen, insbesondere die Dynamik, die in eHomes auftritt, und ihre Ursachen. Die zur Unterstützung der strukturellen Adaption vorgenommenen Erweiterungen des Ansatzes von NORBISRATH werden hier beschrieben. Dabei werden sowohl die Dienstentwicklung als auch die einzelnen Phasen des SCD-Prozesses ausführlich erläutert. Abschließend wird auf das Zusammenspiel mit Lösungen zur Kontexterfassung in der eHome-Umgebung eingegangen. Eine Diskussion verwandter Arbeiten, die sich mit vergleichbaren Themen befassen, schließt das Kapitel ab.

- **Kapitel 5** dient der Erörterung des Lösungsansatzes zur *semantischen Adaption*, der eine Ergänzung des kontinuierlichen SCD-Prozesses darstellt, aber auch unabhängig von der Anwendung in eHome-Systemen betrachtet werden kann. Zunächst wird die Problemlage – das heterogene Dienstumfeld – und dessen Ursachen im Kontext von eHomes näher erläutert. Es folgt eine Beschreibung von Grundlagen der semantischen Modellierung, die für das Verständnis der Lösung nötig sind. Darauf basierend wird ein Teilprozess zur semantischen Adaptierung eingeführt, der in die Gesamtlösung eingebettet wird. Die verschiedenen Phasen dieses Teilprozesses werden im Detail beschrieben, beginnend mit der Ontologiemodellierung über die semantische Abbildung und das Matching bis hin zur Generierung der benötigten Adapterkomponenten zur Laufzeit des eHome-Systems. Anschließend werden weitere verwandte Arbeiten untersucht, die sich mit der semantischen Komposition von Diensten befassen.

- **Kapitel 6** erläutert die Realisierung der beschriebenen Lösung und der entsprechenden Werkzeuge. Dazu werden eingangs technische Grundlagen eingeführt, die zur Implementierung angewandt wurden. Im Hauptteil werden die verschiedenen Werkzeuge vorgestellt, die die einzelnen Bestandteile der Lösung umsetzen. Abschließend wird die für Tests verwendete Anwendungsumgebung – ein softwarebasierter Simulator – beschrieben, und es folgt eine Betrachtung des Umfangs der Implementierung.

➪ **Kapitel 7** schließt die Arbeit ab. Die Ergebnisse der Arbeit werden zusammengefasst und bewertet. In den abschließenden Betrachtungen wird ein Fazit gezogen und die gewonnenen wissenschaftlichen Erkenntnisse der Arbeit werden diskutiert. Darüber hinaus werden verschiedene, auf den Ergebnissen dieser Arbeit basierende, weiterführende Forschungsrichtungen angerissen, die Potential für weitere Arbeiten bieten.

Kapitel 2

eHome-Systeme

Dieses Kapitel führt in des Thema *eHome-Systeme* ein. Zunächst werden Hintergründe und verwandte Entwicklungen erläutert, wobei auch auf die verwendeten Terminologien eingegangen wird. Ein wichtiger Aspekt sind die verschiedenen Anforderungen, die Lösungen für eHome-Systeme erfüllen müssen. Dabei wird zwischen den Anforderungen aus Benutzersicht und technischen Anforderungen unterschieden, letztere ergeben sich teils aus den Benutzeranforderungen aber auch aus der Anwendungsdomäne selbst. Zur Motivation und um ein besseres Bild von typischen Einsatzmöglichkeiten von eHome-Systemen zu geben, werden drei Anwendungsszenarien beschrieben. Diese decken die für eHomes charakteristischen Bereiche *Komfort*, *Sicherheit* und *Unterstützung von Senioren* ab. Abschließend wird eine Übersicht über einige Projekte gegeben, in denen eHomes mit den derzeit verfügbaren Mitteln praktisch umgesetzt wurden.

2.1 Hintergrund und Terminologie

eHomes, wie sie in dieser Arbeit betrachtet werden, gründen auf unterschiedlichen aber verwandten neueren Entwicklungen der Informatik und Kommunikationstechnik. Im Laufe der technischen Entwicklung wurden Prozessoren immer kleiner und leistungsfähiger und sind inzwischen auch in kleinsten Alltagsgeräten präsent. Durch die Entwicklung von Kommunikationstechnik wurde zunehmend die Vernetzung von unterschiedlichen Geräten ermöglicht, z. B. durch den

Einsatz mobiler Ad-hoc-Netze (engl. *Mobile Ad Hoc Network, MANET*). Diese Entwicklung und insbesondere dessen zu erwartender weiterer Verlauf in der Zukunft wird als Ubiquitous Computing bezeichnet. Der Begriff *Ubiquitous Computing* kam Ende der 1980er Jahre in der amerikanischen Forschung auf, insbesondere durch die Arbeiten von WEISER et al. am Forschungszentrum von Xerox in Palo Alto (PARC) in Kalifornien [WGB99]. Zahlreiche weitere Begriffe entstanden, die im Wesentlichen sehr ähnliche Themen beschreiben. Auf die dem Ubiquitous Computing verwandten Begriffe *Pervasive Computing*, *Internet der Dinge* und *Ambient Intelligence* wird daher in diesem Abschnitt ebenfalls eingegangen.

Ubiquitous Computing

MARK WEISER ist einer der Vordenker des Ubiquitous Computing. Er hat in verschiedenen Arbeiten seine Vision der Zukunft des Computers in unserem Alltag beschrieben. Eine seiner am häufigsten zitierten Arbeiten ist *The Computer for the 21st Century* [Wei91]. Darin beschreibt er bereits im Jahr 1991 die zu erwartende zunehmende Verflechtung von Computern mit unserer alltäglichen Umgebung.

Nach WEISER wird es in Zukunft nicht mehr den *Personal Computer* geben, wie wir ihn heute kennen. Vielmehr wird der PC als Arbeitsplatzrechner in den Hintergrund treten und stattdessen werden wir von zahlreichen Mikrocomputern umgeben sein, welche in die Gegenstände, die wir benutzen, integriert sind. Die Benutzerschnittstelle wird nicht mehr der Bildschirm, die Tastatur oder die Maus sein. Die Interaktion mit dem Computer wird auch nicht mehr unsere unmittelbare Aufmerksamkeit in Anspruch nehmen, sondern in den Hintergrund treten. Wir werden auf natürliche Weise mit Anwendungen interagieren, z. B. mittels Sprach- und Gestenerkennung.

Die Integration zahlreicher Funktionen in kleine, mobile Geräte ist in den unterschiedlichsten Bereichen zu beobachten. Mit dem Verlauf der 1990er Jahre und dem Beginn des neuen Jahrtausends hat sich die Mobiltelefonie zum allgegenwärtigen Standard entwickelt und ist aus dem Leben der Menschen kaum noch wegzudenken. Stand anfangs noch die mobile Kommunikation als Funktionalität im Mittelpunkt, ist der Umfang an Funktionen bis heute bedeutend erweitert worden. Obwohl die Geräte viel kleiner sind als in der Anfangszeit, erfüllen sie heute neben der Sprachkommunikation auch Funktionen wie das Fotografieren, das Abspielen von Musikstücken und Videos, das Speichern von Kontaktdaten, die GPS-basierte Navigation und das Ausführen nahezu beliebiger Softwareanwendungen. Heutige Mobiltelefone können über unterschiedlichste Netze mit der Außenwelt verbunden werden und ermöglichen so z. B. auch den Zugriff auf das Internet. Die Geräte sind damit prinzipiell allen Aufgaben gewachsen, für die zuvor noch Arbeitsplatzrechner erforderlich waren.

Die von WEISER beschriebene Entwicklung umfasst noch mehr, als lediglich die Miniaturisierung von Computern in den verschiedensten Formen, wie wir sie heute bereits beobachten können. Die Miniaturisierung von Geräten und die Überladung dieser Geräte mit verschiedensten Funktionen erfordert ein hohes Maß an Aufmerksamkeit durch den Benutzer. Ubiquitous Computing bedeutet, dass Funktionalitäten in unserer natürlichen Umgebung verteilt auftreten werden und somit allgegenwärtig sind. Demnach wird es nicht mehr erforderlich sein, alle Funktionalitäten in Form mobiler Geräte mit sich herumzutragen. Die Funktionalitäten werden stattdessen in Mikrocomputer ausgelagert, die in Alltagsgegenstände integriert sind. Diese müssen nicht mitgenommen werden, da sie überall präsent sind. Die Interaktion mit diesen allgegenwärtigen Funktionalitäten wird nach WEISER intuitiver geschehen als bisher. Das Problem des Überflusses an Informationen wird dadurch aufgelöst werden, da nicht mehr permanent und für alles die Aufmerksamkeit der Benutzer erforderlich sein wird. Computer werden in den Hintergrund treten und unauffällig bleiben. Letztendlich werden sie aus Benutzersicht gar nicht mehr in Erscheinung treten, sondern unsichtbar ihre Aufgaben verrichten [Mat03].

WEISER und BROWN nennen diese Art unauffälligen Auftretens von Computern *Calm Technology*. In [WB98] beschreiben sie dazu drei Phasen der Computerentwicklung. Die erste Phase wird die *Mainframe-Ära* genannt. In dieser frühen Phase waren Computer sehr groß und haben ganze Räume eingenommen. Sie wurden nur von Experten genutzt, die sich die teuren Rechenanlagen teilen mussten. Die zweite Phase wird die *PC-Ära* genannt. Der Personal Computer zeichnet sich dadurch aus, dass er von Einzelpersonen genutzt wird und nicht mehr von einer Vielzahl von Personen. Diese Entwicklung wurde möglich, da sowohl die Größe der Geräte als auch die Anschaffungskosten stark gesunken waren. In einer Übergangsphase, die durch die Einführung des Internet und damit verteilten

Anwendungen charakterisiert ist, sehen WEISER und BROWN eine Verknüpfung aus der Mainframe-Ära und der PC-Ära. Die dritte Phase schließlich ist die von WEISER und BROWN erwartete *Ubiquitous-Computing-Ära*. In dieser Phase werden Mikrocomputer in verschiedenste Alltagsgegenstände integriert sein, wie etwa Lichtschalter, Stühle, Kleidung oder sogar die Wände eines Gebäudes. Damit wird jede Person zahlreiche Computer nutzen, was eine Umkehrung der Verhältnisse der Mainframe-Ära bedeutet.

Pervasive Computing

Pervasive Computing ist ein Begriff, der mit dem Ubiquitous Computing eng verwandt ist, und ebenfalls die allgegenwärtige Informationsverarbeitung beschriebt [HMNS03]. Er entstand jedoch aus dem industriellen Umfeld und setzt daher den Fokus vornehmlich auf sogenannte *eBusiness*- oder *eCommerce*-Szenarien. Durch die dezentralisierte Datenverarbeitung können Unternehmen ihre Geschäftsprozesse verbessern und effizienter arbeiten. Durch die zunehmende Vernetzung können alle relevanten Informationen digital erfasst werden und stehen für die weitere Verarbeitung zur Verfügung. Die Szenarien des Pervasive Computing sind dergestalt, dass sie bereits heute oder zumindest in naher Zukunft umsetzbar sind. Demgegenüber beschreibt die akademisch geprägte Vision des Ubiquitous Computing eine längerfristige Entwicklung, die möglicherweise erst in einer fernen Zukunft Realität wird. In den eCommerce-Szenarien des Pervasive Computing spielt z. B. die Unauffälligkeit der Technik eine geringere Rolle als im Ubiquitous Computing, das insbesondere auch die Nutzung im privaten Umfeld vorsieht. Dies wird auch deutlich, wenn man ein eng mit dem Pervasive Computing zusammenhängendes Thema betrachtet, das *Internet der Dinge*.

Internet der Dinge

Das *Internet der Dinge* (engl. *Internet of Things*) ist eine Forschungsrichtung, die hauptsächlich die Entwicklung der Informationsverarbeitung in Unternehmen betrachtet [FM05] und daher ebenfalls die Bereiche *eBusiness* und *eCommerce* adressiert. Dabei sind insbesondere bereits verfügbare Technologien Gegenstand der Forschung, z. B. *Radio Frequency Identification (RFID)* als Lokalisierungstechnik [Int05]. RFID ist auch heute schon einsatzbereit und besonders in der Transportlogistik von hohem Nutzen. Dabei werden die zu lokalisierenden Gegenstände mit Funketiketten, sogenannten *RFID-Tags*, ausgestattet, die jeweils eine individuelle Kennung haben. Über ein Lesegerät kann diese Kennung dann ausgelesen werden. So können Wertschöpfungsketten vom Rohstoff bis zum Endprodukt erfasst werden, was für die betriebliche Datenverarbeitung nützlich ist und eine Rückverfolgung der Abläufe bei der Herstellung und im Vertrieb ermöglicht. Diese Anwendungsdomäne ist jedoch für die vorliegende Arbeit weniger von Interesse, wenngleich die Lokalisierung von Personen und Objekten auch für eHomes eine große Bedeutung hat.

Ambient Intelligence

Ein weiterer Begriff, der mit dem Thema *eHomes* zusammenhängt, ist die sogenannte *Ambient Intelligence* (dt. *Umgebungsintelligenz*). Der Begriff *Ambient Intelligence* wurde insbesondere in der europäischen Forschung geprägt und betrachtet Szenarien, die mit dem eingangs beschriebenen Ubiquitous Computing eng verbunden sind [Sha03]. Während aber bei den Begriffen *Ubiquitous Computing* und *Pervasive Computing* die grundlegenden Technologien der Informationsverarbeitung und Kommunikation im Vordergrund stehen, setzt der Begriff *Ambient Intelligence* die Schwerpunkte stärker auf die Mensch-Maschine-Interaktion (engl. *Human Computer Interaction, HCI*) und die Künstliche Intelligenz. Ambient Intelligence setzt den Fokus also auf die intuitive Nutzung allgegenwärtiger Technologien, was besonders den Aspekt des *Calm Computing* aus [WB98] hervorhebt.

Im Bereich der sogenannten *Information Society Technologies (IST)* [Eur03] des Sechsten Rahmenprogramms zur Forschungsförderung der Europäischen Kommission [Eur02] wurden viele Projekte aus dem Themenbereich *Ambient Intelligence* gefördert. Einige Szenarien, die die Zielrichtung der Forschungsbemühungen veranschaulichen sollen, wurden in [IST01] vorgestellt. In verschiedenen Forschungsprojekten, die meist von Kooperationen aus akademischen Einrichtungen und Unternehmen durchgeführt wurden, entstanden Konzepte zur Realisierung von Ambient Intelligence [GVR+07, AEHS06, RBD+09, MSS08]. Auch das Bundesministerium für Bildung und Forschung hat das Thema *Ambient Intelligence* aufgegriffen und fördert eine Initiative *Ambient Assisted Living* [Bun08].

Diese hat das Ziel, die Lebensqualität insbesondere älterer Menschen durch „intelligente“ Assistenzsysteme zu verbessern. In [BSG+06], einer Studie, die im Auftrag des Bundesministeriums für Bildung und Forschung erstellt wurde, werden die zu erwartenden Folgen des Ubiquitous Computing untersucht. Dabei wird besonders auf Risiken bezüglich des Datenschutzes und die informationelle Selbstbestimmung eingegangen. Diese Risiken werden im Rahmen der vorliegenden Arbeit nicht behandelt, finden aber in Arbeiten von ARMAÇ Berücksichtigung [APPR09, Arm08] (vgl. auch Abschnitt 1.3).

Im Rahmen der Initiativen zur Förderung von Ambient Intelligence spielt auch die *Telemedizin* eine prominente Rolle. In diesem Zusammenhang sollen Lösungen zu den Herausforderungen des demographischen Wandels und der damit zunehmenden Zahl von Senioren entwickelt werden. Dies zeigt die Bedeutung der gesellschaftlichen Aspekte der technischen Entwicklung, die bei den Initiativen zur *Ambient Intelligence* im Fokus stehen. In einer Studie des Zentrums für Technologiefolgen-Abschätzung beim Schweizerischen Wissenschafts- und Technologierat werden gerade diese gesellschaftlichen Folgen untersucht [EKS+04]. Die *Age Stiftung*, im Jahr 2000 in der Schweiz gegründet, widmet sich der Förderung des Wohnens im Alter [Höp09] und führt Projekte zur Heimautomatisierung durch. Dabei geht es meist darum, älteren Menschen ein selbstbestimmtes Leben zu Hause zu ermöglichen [US08]. Ambient Intelligence hat also gerade in Bezug auf die Unterstützung von Senioren und die Telemedizin mit Heimautomatisierung und eHomes zu tun. Auch die Unterstützung von Menschen mit Behinderungen ist ein breites Themenfeld. Menschen mit Querschnittslähmung, die auf einen Rollstuhl angewiesen sind, können z. B. durch Dienste zur Steuerung von Türen und Fenstern auf Basis von Spracherkennung unterstützt werden [Bru07]. Die Fernsteuerung von Heizung, Beleuchtung, Rollläden und anderen Geräten ist ebenfalls möglich. Auf diese Weise ist auch unter erschwerten Bedingungen ein selbstständiges Leben möglich und die Abhängigkeit von externer Hilfe wird verringert.

eHomes

eHomes, wie sie in dieser Arbeit verstanden werden, basieren auf den oben beschriebenen Konzepten des Ubiquitous Computing und der Ambient Intelligence angewandt auf Wohnumgebungen. Häufig wird auch der Begriff *Smart Home*

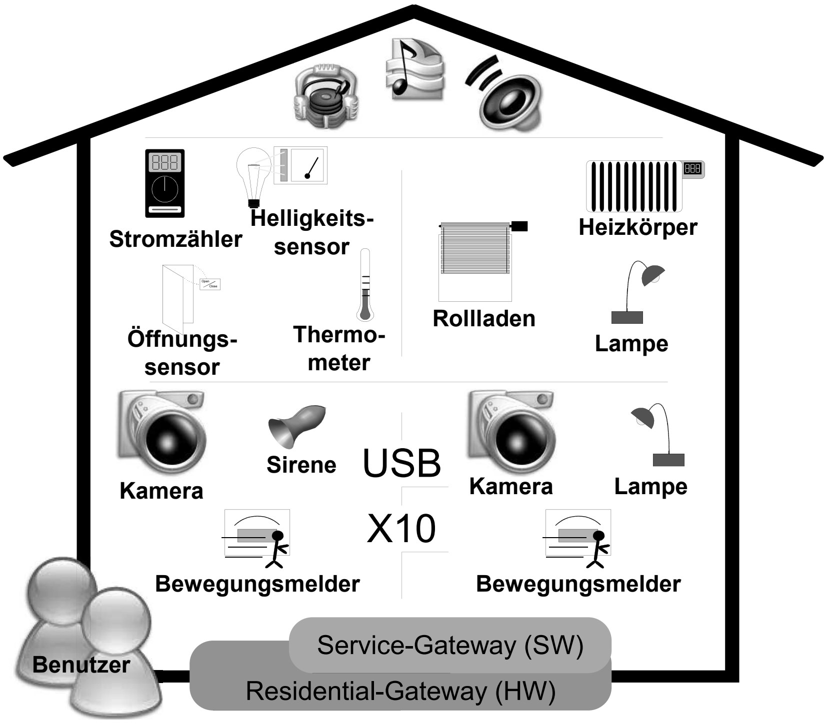


Abbildung 2.1: Übersicht eines eHome-Systems. (Quelle: [Kir05])

verwendet, insbesondere in der englischsprachigen Literatur. Im Deutschen sind auch die Begriffe *intelligente Umgebung* oder *intelligentes Haus* verbreitet.

Bei der Umsetzung von eHomes geht es darum, die im Haus vorhandenen Geräte zu vernetzen und den Bewohnern durch *Dienste* Funktionalitäten anzubieten, die diese Geräte übergreifend nutzen. In Abbildung 2.1 ist ein solches eHome skizziert. Darin sind unterschiedliche Geräte in den verschiedenen Räumen dargestellt. Zur übergreifenden Nutzung der Geräte durch eHome-Dienste wird eine Infrastruktur vorausgesetzt, die die Vernetzung der Hardware und die Ausführung der Dienste ermöglicht.

Häufig ist auch eine Außenanbindung des eHomes, z. B. über einen Internetanschluss, zu einem Provider vorgesehen, der verschiedene Dienste bereitstellt. Diese können dann abonniert und genutzt werden. Eine solche Anbindung geschieht über ein sogenanntes *Residential Gateway*, das in der Abbildung unten dargestellt ist. Das Residential Gateway ist der zentrale Knotenpunkt, an dem die verschiedenen im Haus genutzten Infrastrukturen zusammenlaufen und integriert werden. Auch unabhängig von der Vernetzung mit der Außenwelt spielt das Residential Gateway daher eine wichtige Rolle im eHome.

Auf Basis des Residential Gateways wird eine Softwareinfrastruktur zur Ausführung von eHome-Diensten betrieben, ein sogenanntes *Service Gateway*. Das Service Gateway ermöglicht die Komposition, Adaption und Ausführung der eHome-Dienste. Ein *eHome-System* umfasst die Geräteumgebung, die zugehörige Infrastruktur sowie das Residential Gateway, das Service Gateway und die darauf ausgeführten eHome-Dienste. Eine genauere Beschreibung des Systems, das in dieser Arbeit entwickelt wird, folgt in Abschnitt 4.1.2.

In der Arbeit von KIRCHHOF werden eHomes in verschiedene Entwicklungsstufen eingeteilt, sogenannte Generationen [Kir05]. Diese sind abhängig von der Konnektivität der im eHome eingesetzten Geräte. Die *nullte Generation* (0G) bezeichnet Systeme ohne Vernetzung der Geräte, wie es auch heute vielfach noch der Fall ist.

Die Systeme der *ersten Generation* (1G) bestehen demgegenüber bereits zum Teil aus miteinander verbundenen Geräten. Die Konnektivität ist dabei meist auf bestimmte Gerätegruppen beschränkt und der Zugriff auf Gerätefunktionalitäten ist ebenfalls sehr begrenzt. Die Funktionalität ist lokal in den Geräten implementiert.

Die *zweite Generation* von Systemen (2G) beschreibt eine weitere Entwicklungsstufe, die sich durch stärkere Vernetzung und insbesondere das Vorhandensein nicht-lokaler Funktionalitäten auszeichnet. In diesen Systemen findet bereits eine Integration statt, die geräteübergreifende Funktionalitäten ermöglicht. Diese Integration ist mit heute vorhandener Technik realisierbar, allerdings nur in begrenzter Form und mit einem hohen Aufwand bei der Umsetzung, da noch keine geeigneten Infrastrukturen existieren.

Die in [Kir05] beschriebene Vision sind Systeme der *dritten Generation* (3G), bei denen die vollständige Integration der Geräteumgebung angestrebt ist, sodass übergreifende nicht-lokal realisierte Funktionalitäten zum Standard werden. Die dritte Generation stellt die Zielvorgabe für eHome-Systeme dar, die Anwendung von Ubiquitous Computing und Ambient Intelligence in der alltäglichen Wohnumgebung. Welche wesentlichen Anforderungen aus Benutzersicht dabei zu beachten sind, wird im folgenden Abschnitt betrachtet.

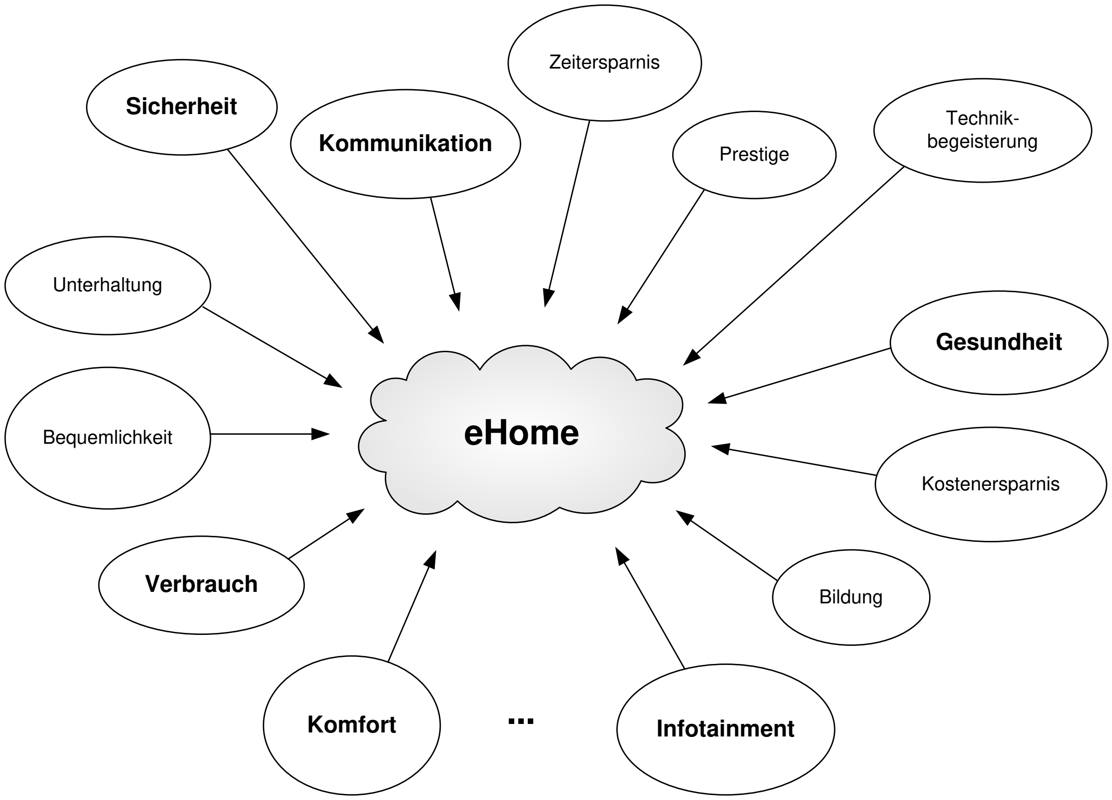


Abbildung 2.2: Motivierende Faktoren für eHomes.

2.2 Anforderungen aus Benutzersicht

eHomes ermöglichen verschiedenste Anwendungen, deren Nutzen für die Bewohner von den individuellen Bedürfnissen und Vorlieben abhängt. Funktionalitäten auf Basis von Diensten, die in einer integrierten Hardwareumgebung ausgeführt werden, sind bisher nicht in der Praxis verfügbar. Es existieren lediglich entsprechende Prototypen. Daher können bisher keine klaren Aussagen darüber gemacht werden, welche Anwendungen von Bewohnern angenommen werden und welche eher unerwünscht sind. Dennoch können bestimmte Faktoren identifiziert werden, die eine Nachfrage nach eHomes und den damit verbundenen Funktionalitäten erwarten lassen.

In Abbildung 2.2 sind einige motivierende Faktoren für eHome-Funktionalitäten aufgeführt. Als wichtigste dieser Faktoren wurden von KIRCHHOF *Komfort*, *Sicherheit*, *Kommunikation*, *Infotainment*, *Gesundheit* und *Verbrauch* identifiziert. Die meisten Dienste, die einen unmittelbar erkennbaren Nutzen erwarten lassen, können diesen sechs Anwendungsfeldern zugeordnet werden.

Als *Komfortdienste* sind z. B. die Steuerung von Umgebungseigenschaften, wie etwa Beleuchtung, Temperatur oder Belüftung, naheliegende Anwendungen. Auch die zentrale Steuerung von Geräten ist ein wichtiger Aspekt. So muss der Nutzer z. B. nicht in jeden Raum gehen, nur um die Beleuchtung im gesamten Haus auszuschalten.

Auch für *Sicherheitsdienste* gibt es verschiedene Einsatzmöglichkeiten. Die im Haus verfügbaren Sensoren könnten beispielsweise genutzt werden, um einen Einbruch zu erkennen. Ein Sicherheitsdienst könnte aktiviert werden, wenn alle Bewohner das Haus verlassen haben. Die Informationen von Bewegungsmeldern, Infrarotsensoren, Videokameras sowie Sensoren an Türen und Fenstern könnten dann ausgewertet werden, um unberechtigten Zutritt zu erkennen und eine entsprechende Reaktion auszulösen.

Im Bereich der *Kommunikation* gibt es ebenfalls viele Anwendungen. Ein Kommunikationsdienst im eHome könnte Anrufe entgegen nehmen und je nach aktuellem Aufenthaltsort des Adressaten Lautsprecher, Mikrofone sowie Displays und Kameras nutzen, um eine Kommunikation zu ermöglichen, ohne dass der Nutzer erst einen bestimmten Raum aufsuchen oder ein mobiles Kommunikationsgerät mit sich tragen muss.

Als *Infotainment* werden im eHome Dienste zur Unterhaltung oder zur Information des Nutzers betrachtet. Ein Musikdienst kann z. B. in der jeweiligen unmittelbaren Umgebung eines Nutzers Lautsprecher verwenden, um die Lieblingsmusik oder favorisierte Radiosender abzuspielen. Wenn sich der Nutzer im eHome bewegt, wird die Musik mitgenommen. Ebenso könnten bestimmte Beleuchtungsszenarien oder -profile zu verschiedenen Zeiten und Anlässen aktiviert werden. Beim Anschauen eines Films kann dann eine andere Beleuchtung aktiviert werden als beim Frühstück.

Der Faktor *Gesundheit* wurde bereits in Abschnitt 2.1 unter dem Thema *Telemedizin* adressiert. Hier ist ein hohes Potential zu sehen, da die medizinische Überwachung bei gleichzeitig sinkenden Kosten verbessert werden kann. Auch das selbstständige Leben im Alter kann durch Dienste, die z. B. Meldungen zur Erinnerung geben, unterstützt werden.

Auf Kostensenkung zielt auch der Bereich *Verbrauch* ab. Zunächst kann durch eine detailliertere Erfassung und Darstellung von Energie- und Wasserverbrauch, ein Bewusstsein für unnötigen Verbrauch geweckt werden. Dadurch allein ist eine spezifischere und bewusstere Nutzung von Ressourcen möglich. Ein zusätzlicher Effekt kann erreicht werden, indem Geräte durch Dienste intelligent gesteuert werden. Wenn sich z. B. niemand im eHome aufhält, kann die Beleuchtung ausgeschaltet und die Heizung heruntergeregelt werden. Das Bad kann morgens vorgeheizt werden, ohne dass die Heizung die gesamte Nacht über laufen muss. Ein Lüftungsdienst könnte bei Bedarf Fenster öffnen und parallel die Heizkörper im Raum herunterregeln und das Lüften ebenso auch wieder beenden. Dies könnte sogar automatisch ausgeführt werden, wenn niemand im Raum anwesend ist. Durch Messen der Windgeschwindigkeit und mittels Regensensoren könnte der Dienst zuvor prüfen, ob ein Lüften momentan möglich ist oder nicht.

Die oben beschriebenen typischen Anwendungsfelder skizzieren bereits verschiedene Möglichkeiten, eHome-Dienste sinnvoll in eine Wohnumgebung zu integrieren. Die Umsetzung scheint auf den ersten Blick recht unkompliziert. Die größten Schwierigkeiten liegen jedoch insbesondere bei der Anpassung einer Dienstkomposition an Veränderungen der Umgebung und im sinnvollen Zusammenwirken aller in einem eHome ausgeführten Dienste. Eine ganze Reihe von Diensten bietet personalisierte oder personalisierbare Funktionalitäten an. Diese Dienste benötigen Kontextinformationen, z. B. über den Aufenthaltsort des Nutzers. Ebenso muss bekannt sein, welche Geräte wo im eHome verfügbar sind. Um diese Informationen nutzbar zu machen und eine Anpassung bei Kontextänderungen zu ermöglichen, ist eine geeignete Infrastruktur erforderlich. Aus Sicht der Benutzer ist in den meisten Fällen keine *direkte* Interaktion mit dem eHome gewünscht. Dies ist gerade der in [WB98] beschriebene Aspekt der unauffälligen Technologie und des Verschwindens von Computern in den Hintergrund. Daher muss das eHome selbstständig auf Veränderungen reagieren und die Dienstkomposition anpassen.

Das Zusammenspiel der Dienste ist eine besondere Herausforderung, da Konflikte bei der Ressourcennutzung und den Auswirkungen der Dienstfunktionalitäten vermieden werden müssen. Nur dann kann insgesamt ein Nutzen für die Bewohner erzielt werden. Ein Dienst, der ein Telefongespräch ermöglicht, benötigt Lautsprecher. Wenn diese bereits durch einen Musikdienst – möglicherweise eines anderen Nutzers – belegt sind, so liegt ein Konflikt vor, der verhindert, dass beide Dienste ihre Funktionalität wie vorgesehen erfüllen können. Weitere Konflikte können entstehen, wenn z. B. auch noch ein Klingeldienst, ein Weckdienst und ein Alarmdienst Lautsprecher benötigen. Eine manuelle Auflösung durch den Benutzer ist umständlich und ein wiederkehrendes Auftreten derselben Konfliktfälle würde den Nutzen schnell soweit reduzieren, dass niemand solche Dienste verwenden würde. Es ist also auch hier eine weitestgehend automatische Lösung gefragt.

Allerdings können viele Fälle nicht durch allgemeine Vorgaben abgehandelt werden. Ein wichtiges Telefongespräch sollte nicht verhindert werden, nur weil gerade Musik über die Lautsprecher abgespielt wird. Die Musik sollte dann nur mit verminderter Lautstärke weiterlaufen oder ganz ausgeschaltet werden. Es können natürlich noch weitere Geräuschquellen aktiv sein, die das Telefonat stören, wie z. B. ein Ventilator oder eine Waschmaschine, ein offenes Fenster könnte ebenfalls dazu führen, dass Lärm von draußen zu hören ist. Die Frage ist, ob diese Geräuschquellen ebenfalls beachtet werden sollen, ob also auch der Ventilator und die Waschmaschine ausgeschaltet und das Fenster geschlossen werden sollen.

Wie weit ein eHome-System automatisch reagieren soll ist kaum allgemein festzulegen, sondern hängt vielmehr von den jeweiligen Nutzern und den verwendeten Diensten ab. Bestimmte Situationen sind nicht automatisch zu bewerten, in solchen Fällen wird immer der Benutzer eine Wahl treffen müssen. Wenn ein Telefonat geführt wird und gleichzeitig an der Tür geklingelt wird, kann das System nicht entscheiden, ob eine der beiden Funktionalitäten wichtiger ist. Auch der Benutzer kann dazu keine pauschale Vorgabe machen. In einem Fall handelt es sich um ein wichtiges Telefonat zur Bewerbung auf eine Stelle, das nicht unterbrochen werden soll, nur weil ein unerwünschter Händler vor der Tür steht. In einem anderen Fall geht es um ein dringend erwartetes Einschreiben und das Klingeln des Postboten hat oberste Priorität. Dann soll das Klingelsignal nicht unterbunden werden, nur weil gerade ein Anrufer am Telefon ist, der sich verwählt hat. Auch hier wird eine individuelle Entscheidung durch den Benutzer gefordert sein.

Es muss also untersucht werden, welche Fälle überhaupt durch die Dienste und die eHome-Laufzeitumgebung abgehandelt werden können, mit welchen Fällen der Benutzer konfrontiert werden muss und wie dann eine geeignete Benutzerinteraktion aussehen kann. Dabei ist zu berücksichtigen, dass die Motivation eines eHomes nicht ist, den Benutzern Verantwortung und Entscheidungen abzunehmen. Viele der Konflikte, die bei der Umsetzung von eHomes offenkundig werden, existieren auch außerhalb eines eHomes und können nicht ohne Weiteres durch das Implementieren einer technischen Lösung eliminiert werden.

Vor dem Hintergrund der oben beschriebenen Umsetzungsprobleme und Konfliktfälle, die bei der Realisierung von eHomes in Erscheinung treten, ist es umso wichtiger, dass die Nutzer das Verhalten des eHome-Systems nachvollziehen können. Die Effekte, die sich durch die Ausführung von Diensten ergeben, und die Auswirkungen der Dienstfunktionalitäten auf die Umgebung müssen also plausibel sein. Dadurch dass Informationstechnologie allgegenwärtig wird, ist auch eine vielfältige Interaktion mit dem Benutzer erforderlich. Diese Interaktion muss aber intuitiv sein und darf nicht die permanente Aufmerksamkeit des Benutzers verlangen, denn nur dann kann von Computing im Hintergrund gesprochen werden. Ziel ist es, den Alltag der Nutzer zu erleichtern, nicht ihn mit technischen Problemen zu überhäufen. Dazu sind natürliche Interaktionsschnittstellen und Kontextbezogenheit erforderliche [AM00]. Die Interaktion mittels Maus, Tastatur und Monitor, wie es bei Arbeitsplatzrechnern bislang üblich ist, kann hier nicht angewandt werden. Stattdessen können z. B. Mechanismen der Gesten- und Spracherkennung genutzt werden.

Der Nutzer muss jederzeit die Möglichkeit haben, das Verhalten des Systems einfach und intuitiv zu beeinflussen, sodass eine natürliche Interaktion mit der Umgebung und den dort verfügbaren Diensten erfolgt. Besonders wichtig ist, dass der Nutzer unerwünschtes Verhalten abstellen oder korrigieren kann, ohne dabei durch technische Hürden behindert zu werden. Das Zusammenspiel der Dienste muss ggfs. sofort geändert werden können, z. B. muss es möglich sein, eine Auswahl zu treffen, welche Geräte in der Umgebung zur Umsetzung einer bestimmten Dienstfunktionalität herangezogen werden sollen. Die *Nachvollziehbarkeit* des eHome-Verhaltens und die *Interaktionsmöglichkeiten* sind daher wesentliche Benutzeranforderungen, die von zukünftigen eHome-Systemen erfüllt werden müssen, damit sich Benutzer nicht hilflos einem System ausgesetzt sehen, dessen Verhalten sie weder verstehen noch beeinflussen können.

2.3 Technische Herausforderungen

Neben den Anforderungen aus Benutzersicht gibt es verschiedene technische Herausforderungen, die sich teils aus den Benutzeranforderungen ergeben und teils aus der Anwendungsdomäne eHomes ableiten lassen. Diese technischen Aspekte müssen bei der Entwicklung einer Lösung berücksichtigt werden, damit diese in der Praxis umgesetzt werden kann.

In [BB02] werden drei Hauptanforderungen an die Umsetzung von Ubiquitous Computing genannt: die *Dynamik* der Umgebung und der Benutzerwünsche, die *Heterogenität* von Geräten und die Nutzung von Informationstechnik in einer *sozialen Umgebung*. Letzteres hat insbesondere Konsequenzen auf die Privatsphäre und Fragen nach dem Zugriff auf Daten, die z. B. durch die verschiedenen Sensoren in einer Umgebung erfasst wurden. In der vorliegenden Arbeit werden speziell die Themen *Dynamik* und *Heterogenität* im Hinblick auf die Softwareunterstützung von eHomes untersucht. Die Auswirkungen von eHomes auf Sicherheit und Privatsphäre werden unter anderem in Arbeiten von ARMAÇ betrachtet [APPR09, Arm08]. Die in [BB02] aus den Hauptanforderungen abgeleiteten Forschungsrichtungen sind die *semantische Modellierung*, das Schaffen einer geeigneten *Softwareinfrastruktur*, das *Entwickeln* und *Konfigurieren* von Anwendungen durch Komposition von Diensten und schließlich die Auswertung von *Erfahrungen durch die Anwender* bei der Nutzung von Ubiquitous Computing. Von diesen Aspekten werden viele auch in der vorliegenden Arbeit adressiert. Die Entwicklung einer Softwareinfrastruktur zur Komposition und Ausführung von Diensten wird in Kapitel 4 behandelt. Dabei geht es insbesondere auch um kontextbezogene Anpassungen der Dienstkomposition zur Laufzeit durch eine Rekonfigurierung. Die semantische Modellierung wird in Kapitel 5 behandelt. Hier wird ein Ansatz zur semantischen Beschreibung von Diensten auf Basis einer Ontologie vorgestellt, sowie zu dem darauf basierenden semantischen Matching und der Adaptierung heterogener Dienste.

EDWARDS und GRINTER identifizieren in [EG01] sieben Herausforderungen, die auf dem Weg zu intelligenten *Wohnumgebungen* überwunden werden müssen. Dabei wurden sowohl technische und pragmatische als auch soziale Einflüsse betrachtet. Auf einige dieser Herausforderungen wird im Verlauf der Arbeit Bezug genommen.

1. **Das „zufällige“ Smart Home:** Es ist zu erwarten, dass bereits vorhandene Wohnumgebungen schrittweise in intelligente Umgebungen umgewandelt werden. Insbesondere in der Einführungsphase werden neue Technologien nur in Teilen zur Anwendung kommen und dies in Umgebungen, die nicht von vornherein auf diese Technologien ausgelegt wurden. Solche Umgebungen werden irgendwann eine Komplexität erreicht haben, die nur noch schwer zu beherrschen ist. Im Zusammenspiel der neuen Technologien und ihrer Benutzer werden auch neue Probleme auftreten und die Frage ist, ob die Benutzer in der Lage sein werden, diese zu lösen.

2. **Interoperabilität:** Die in der Umgebung eingesetzten verschiedenen Technologien müssen zusammenspielen. Dabei ist es entscheidend, dass Geräte nicht nur miteinander verbunden werden können, sondern auch auf sinnvolle Art und Weise zusammenarbeiten, ohne dass zuvor eine spezifische Anpassung erfolgt oder der Benutzer eingreifen muss. Die Frage ist, in wie weit Interoperabilität erreicht werden kann, ohne dass eine vorherige Standardisierung auf syntaktischer oder semantischer Ebene erfolgt. Einfache Festlegungen auf der Ebene von Protokollen und Schnittstellen reichen hier nicht aus.

3. **Kein Systemadministrator:** In intelligenten Wohnumgebungen wird es keinen Systemadministrator geben. Am Beispiel der Einführung von PCs konnte beobachtet werden, dass die Nutzer Aufgaben zu bewältigen hatten, die zuvor in Großrechnerzeiten von Spezialisten ausgeführt wurden. Solche Vorgänge, wie die Installation und das Aktualisieren von Software und Hardware, mussten deutlich vereinfacht werden. Auch bei intelligenten Wohnumgebungen handelt es sich um komplexe Systeme, deren Nutzer nicht sämtliche technischen Details erfassen und beherrschen können. Daher müssen diese Systeme darauf ausgelegt werden, im normalen Betrieb ohne manuelle Administration auszukommen.

4. **Entwurf für den Hausgebrauch:** Durch die Einführung neuer Technologien werden die Abläufe der Nutzer in Wohnumgebungen beeinflusst. Für die Entwickler solcher Technologien ist es entscheidend, die bereits vorhandenen Abläufe zu erkennen und zu berücksichtigen, damit Nutzer nicht gezwungen werden, sich von gewohnten Abläufen zu verabschieden und sich an die technischen Vorgaben anzupassen.

5. **Soziale Auswirkungen:** Die sozialen Auswirkungen der neuen Technologien können nicht mit Sicherheit vorhergesagt werden. Sie sind nicht nur abhängig von den Erwartungen und Gewohnheiten der Nutzer, sondern sie verändern diese auch.

6. **Zuverlässigkeit:** Je mehr die neuen Technologien den Alltag der Nutzer bestimmen, desto mehr Zuverlässigkeit wird auch eingefordert. Waschmaschinen, Telefone, Fernsehgeräte oder Autos haben eine hohe Komplexität, sie arbeiten aber dennoch sehr zuverlässig im Gegensatz zu vielen Softwareanwendungen die heute am PC genutzt werden. Diese Zuverlässigkeit ist für intelligente Wohnumgebungen von essentieller Bedeutung. Darüber hinaus gibt es zahlreiche gesetzliche Regelungen, die die Anforderungen von Gebrauchsgegenständen und Haushaltsgeräten in Bezug auf ihre Funktionalität und Qualität regeln. Diese müssen ebenfalls berücksichtigt werden.

7. **Automatisches Schlussfolgern:** Mit einer intelligenten Umgebung wird assoziiert, dass sie auf den Benutzer reagiert und aus dessen Verhalten lernt, um in Zukunft die Benutzerwünsche besser und genauer zu erfüllen. Hier stellt sich jedoch die Frage, in wie weit das System automatisch Schlussfolgerungen ziehen soll. In welchem Umfang ist eine Lernfähigkeit der intelligenten Umgebung sinnvoll, erreichbar und gewünscht?

Ein besonderes technisches Problem bei der Realisierung von eHomes ist die Anbindung der unterschiedlichen Geräte. Die Problematik ist mehrschichtig, da zum einen verschiedene Infrastrukturen verwendet werden und zum anderen die Geräte selbst keine einheitlichen Schnittstellen anbieten und keinem einheitlichen Protokoll entsprechen. Geräte können drahtlos angebunden sein, wobei es ja nach Anwendungsgebiet verschiedene Standards gibt, z. B. Bluetooth, WLAN (IEEE 802.11), GSM oder UMTS. Andere Geräte sind kabelgebunden mit dem eHome vernetzt. Auch dort gibt es unterschiedliche Standards, z. B. X10 [Sma09a], HomePlug [Hom05], die auf dem Stromnetz im Gebäude basieren, oder EIB/KNX (EN 50090 und ISO/IEC 14543) [ZVE06], das ein separates Steuerungsnetz verwendet.

Standards, die wie X10 auf dem vorhandenen Stromnetz im Haus basieren, bieten sich an, wenn eHome-Funktionalität nachträglich in ein Gebäude integriert werden soll. In Abbildung 2.3 sind zwei X10-Geräte zu sehen. Abbildung 2.3(a)

(a) X10-Lampe. (b) X10-Fernbedienung.

Abbildung 2.3: Verschiedene X10-Geräte.

zeigt eine X10-Lampenfassung mit Glühbirne, die in die Fassung einer gewöhnlichen Lampe geschraubt werden kann. Eine Steuerung ist dann über das Stromnetz und einen X10-Controller möglich. Zur Steuerung der Lampe kann beispielsweise eine X10-Fernbedienung, wie in Abbildung 2.3(b) dargestellt, verwendet werden. Die Fernbedienung sendet ein Funksignal, das von einem entsprechenden Empfangsmodul in X10-Steuersignale umgewandelt wird. Je nachdem welche Geräte verwendet werden, müssen verschiedene Standards unterstützt und integriert werden. Es ergibt sich also keine einheitliche Infrastruktur, sondern es ist im Allgemeinen von einer heterogenen Umgebung auszugehen.

Gerade bei drahtlos angebundenen mobilen Geräten ergeben sich häufig Veränderungen, wenn z. B. ein neues Gerät den Empfangsbereich des eHomes erreicht oder diesen verlässt. Aber auch bei kabelgebundenen Geräten ergeben sich Änderungen, da neue Geräte angeschlossen werden können oder vorhandene Geräte vom System getrennt werden. Diese Dynamik muss bei der Hardwareanbindung berücksichtigt werden. Die Situation ist ähnlich zu der Anbindung von Peripheriegeräten an einen PC. Anfang der 1990er Jahre waren Betriebssysteme auf eine relativ unveränderliche Hardwareumgebung ausgelegt [Wei91]. Die Einbindung von Peripheriegeräten ist heute wesentlich flexibler geworden. Für alle verbreiteten Betriebssysteme ist eine automatische Einbindung der meisten Geräte möglich, sodass der Benutzer in vielen Fällen keine Gerätetreiber mehr von Hand installieren muss. Eine derartige Plug-&-Play-Funktionalität ist möglich, da es entsprechende Standards gibt, sodass Geräte identifiziert werden können. Dennoch wird ein für das Betriebssystem passender Treiber benötigt, der entweder mit dem Betriebssystem bereits mitgeliefert ist oder über einen Datenträger oder das Internet nachgeladen werden kann. Die Zuordnung des passenden Treibers wird anhand einer Geräteidentifikation vorgenommen. In einem eHome-System muss ein ähnlicher Mechanismus verfügbar sein, der es ermöglicht die verschiedenen Geräte zu erkennen und auf Änderungen der Geräteumgebung zu reagieren.

2.4 Anwendungsbeispiele

In diesem Abschnitt werden Anwendungsbeispiele für eHome-Dienste vorgestellt, die zum einen der Motivation von eHomes und möglichen Dienstfunktionalitäten im Allgemeinen dienen und zum anderen Beispielszenarien zur Veranschaulichung der Problemstellungen und Lösungsansätze dieser Arbeit darstellen. Auf diese Beispieldienste wird in späteren Kapiteln dieser Arbeit zurückgegriffen. Die hier vorgestellten Szenarien decken die Bereiche *Komfort*, *Sicherheit* und *Unterstützung von Senioren* ab. Die entsprechenden Anwendungsfelder wurden bereits in Abschnitt 2.2 eingeführt.

2.4.1 Komfortszenario

In diesem Szenario werden verschiedene Komfortdienste in einer privaten Wohnumgebung eingesetzt, um alltägliche Abläufe zu vereinfachen und den Bewohnern damit Arbeit abzunehmen. Die vorgestellten Dienste können in einer Wohnumgebung eingesetzt werden, in der verschiedene Geräte in den jeweiligen

Räumen verfügbar sind. Im Folgenden werden einige Top-Level-Dienste aus dem Bereich *Komfort* beschrieben. *Top-Level-Dienste* sind Anwendungen, die vom Benutzer ausgewählt werden, um die gewünschten Funktionalitäten auf Basis der vernetzten Geräteumgebung im eHome zu realisieren. Eine genauere Diskussion verschiedener Diensttypen erfolgt in Abschnitt 3.4.3 und Abschnitt 4.3.1.

- **Musikdienst.** Dieser Dienst spielt eine personalisierte Zusammenstellung von Musikstücken ab. Für jede Person gibt es eine Liste mit Musikstücken, die von dem Dienst abgespielt werden, dabei können sowohl lokale Quellen als auch Internetressourcen – beispielsweise ein Web-Radiosender – verwendet werden. Es lassen sich zwei Varianten unterscheiden. Die *raumbasierte* Variante des Musikdienstes ist einem spezifischen Raum zugeordnet und spielt Musik für momentan anwesende Personen ab. Dabei muss festgestellt werden, ob und welche Personen im Raum anwesend sind. Bei mehreren Personen muss eine Strategie festgelegt werden, welche Musik abgespielt wird, z. B. kann die Musik der am längsten anwesenden Person abgespielt werden. Dies ist natürlich eine vereinfachende Strategie. Im Rahmen dieser Arbeit wird die Möglichkeit geschaffen, eine andere Variante des Dienstes zu implementieren, die verschiedene Vorteile aufweist, die im weiteren Verlauf der Arbeit noch diskutiert werden. Dabei handelt es sich um die *personenbezogene* Variante des Musikdienstes. In dieser Variante ist der Musikdienst einer einzelnen Person zugeordnet und spielt nur deren Musikwünsche ab. Der Dienst bewegt sich dann mit der zugehörigen Person durch die Umgebung, d. h. der Auswirkungsort der Musikfunktionalität ändert sich. Abhängig davon wo sich die Person im eHome befindet, wird die Musik über die jeweils verfügbaren Lautsprecher abgespielt.

- **Weckdienst.** Der Weckdienst hat die Aufgabe einen Bewohner zu einer vorgegebenen Zeit oder unter einer anderen zuvor definierten Bedingung durch ein entsprechendes Signal zu wecken. Dabei kann eine einfache Zeitbedingung vorliegen, es sind aber auch komplexere Varianten möglich. Verschiedene zusätzliche Informationen können einbezogen werden, von denen die Weckzeit abhängig ist. Basierend auf persönlichen Terminen kann z. B. eine variable Weckzeit berechnet werden. Dazu kann auch der Ort eines Termins berücksichtigt werden, sodass ggfs. die Zeit für die Anreise mit einkalkuliert werden kann.

- **Heizungsdienst.** Der Heizungsdienst steuert die Umgebungstemperatur eines bestimmten Raums oder der Umgebung einer zugeordneten Person. Dies hängt davon ab, ob es sich um eine raumbezogene oder personenbezogene Variante des Dienstes handelt. Die gewünschte Temperatur kann zuvor spezifiziert werden. Abhängig von diesem Sollwert steuert der Dienst die verfügbaren Geräte zur Temperaturregulierung, wie z. B. Heizkörper, Klimaanlage, Belüftungssystem oder auch die Fenster. Um den momentanen Istwert der Temperatur zu bestimmen, muss ein entsprechender Sensor verwendet werden. Dann kann die Temperatur so geregelt werden, dass der Istwert in einem bestimmten Intervall um den Sollwert herum gehalten wird.

- **Beleuchtungsdienst.** Dieser Dienst steuert die Beleuchtung eines Raums oder der Umgebung einer bestimmten Person. Dabei sind unterschiedliche Varianten möglich. Eine einfache Variante ist ein Beleuchtungsdienst, der das Licht einschaltet, wenn sich Personen im Raum aufhalten. Eine personenbezogene Variante schaltet das Licht immer dort ein, wo sich die Person derzeit befindet. Eine erweiterte Variante des Dienstes kann unterschiedliche Beleuchtungsszenarien berücksichtigen und so eine individuell angepasste Beleuchtungssituation herstellen. So ist es z. B. möglich, eine bestimmte Helligkeit oder ggfs. auch farbiges Licht vorzugeben. Solche Beleuchtungsprofile können auch von anderen Faktoren abhängig sein, wie etwa der Tageszeit oder einem bestimmten Zweck. Wird z. B. ein Film geschaut, so wird eine spezielle gedimmte Beleuchtung aktiviert. Während des Essens wird wiederum eine andere Beleuchtung hergestellt. Eine weitere Variante des Dienstes schaltet die gesamte Beleuchtung im eHome aus, was beim Verlassen des Hauses oder der Wohnung eine sinnvolle Funktion darstellt. Umgekehrt kann der Dienst auch eine Funktionalität anbieten, die Beleuchtung im gesamten eHome zu aktivieren. Dies ist z. B. nachts als Panikfunktion einsetzbar.

- **TV-Dienst.** Ein TV-Dienst gehört ebenfalls zu den Komfortdiensten, da im eHome die Funktionalitäten nicht mehr fest in die Geräte integriert sind. Der TV-Dienst benötigt daher zum einen die Videodatenquelle eines TV-Senders, die z. B. von einem Empfangsgerät oder auch über das Internet angeboten wird, und zum anderen ein Display, das die Darstellung der Videodaten ermöglicht. So kann die TV-Funktionalität durch passende Basisdienste realisiert werden. Das dabei verwendete Display kann aber auch für verschiedenste weitere Funktionalitäten genutzt werden, was durch die Entkopplung von der TV-Funktionalität ermöglicht wird.

Top-Level-Dienste benötigen verschiedene *Basis-* oder *Treiberdienste*, um ihre Funktionalitäten zu realisieren. Im Folgenden sind einige Beispieldienste aufgeführt, die von den oben beschriebenen Top-Level-Diensten verwendet werden können.

- **Lautsprechersteuerung.** Der Treiberdienst zur Lautsprechersteuerung bietet Funktionalität zur Audioausgabe an. Diese Funktionalität kann von Top-Level-Diensten verwendet werden, um verschiedene Audioquellen abzuspielen. Hierbei ist zu beachten, dass Lautsprecher hier als aktive Geräte betrachtet werden, die z. B. auch über ein Netzwerk mit Audiodaten versorgt werden können und diese wiedergeben. Es handelt sich also nicht um passive Einheiten, die wie in einem klassischen Szenario über einen Verstärker angesteuert werden und außerhalb dieser Anordnung nicht nutzbar sind. Solche Geräte sind auch heute schon am Markt erhältlich, teilweise ist sogar die drahtlose Anbindung über WLAN möglich.

- **Personenerkennung.** Ein Dienst zur Personenerkennung kann auf ganz unterschiedliche Weisen realisiert werden. Der mögliche Einsatz von RFID-Technik zur Erkennung und Lokalisierung von Personen wurde bereits in Abschnitt 2.1 angesprochen. Dies setzt voraus, dass die Bewohner im eHome mit entsprechenden RFID-Tags ausgestattet sind. Andere Verfahren basieren auf einer Videoüberwachung und der Erkennung von Personen im Videobild. Eine weitere Möglichkeit ist die Ortung von mobilen Geräten, wie Handys, die einzelnen Personen zugeordnet werden können. Dabei muss aber vorausgesetzt werden, dass das mobile Gerät immer von der Bezugsperson mitgeführt wird, weshalb diese Methode im eHome eher ungeeignet ist. Die Personenerkennung ist nicht nur für bestimmte Top-Level-Dienste relevant, sondern ebenso für die Laufzeitumgebung selbst. Diese hat in ihrer Funktion als Middleware die Aufgabe, eHome-Dienste von Infrastrukturaufgaben zu entlasten und damit deren Implementierung zu vereinfachen. Dazu gehört auch die Personenerkennung zur Bereitstellung von Kontextinformationen. Wird die Erkennung und Lokalisierung der Personen von der Middleware unterstützt, so können personenbezogene Dienste auf diese Basisfunktionalität zurückgreifen und müssen keine eigenen Mechanismen zur Ermittlung und Verwaltung der relevanten Personendaten mitbringen.

- **Heizkörpersteuerung.** Dieser Treiberdienst ermöglicht die Ansteuerung einzelner Heizkörper im eHome. Dabei wird eine stufenlose Regulierung ermöglicht, so wie es auch bei manueller Steuerung über einen Heizkörperregler üblich ist. Dieser Dienst kann von Top-Level-Diensten zur Regulierung der Umgebungstemperatur verwendet werden.

- **Temperatursensor.** Dieser Dienst ermöglicht den Zugriff auf einen Temperatursensor durch andere eHome-Dienste. Er wird von Regulierungsdiensten verwendet, die von der Temperatur abhängig sind.

- **Lampensteuerung.** Der Lampensteuerungsdienst wird zur Steuerung der Beleuchtung verwendet. Er erlaubt das Ein- und Ausschalten von Lampen, ggfs. auch das Dimmen, wenn dies von der Lampe unterstützt wird. Je nach Art der Lampe können weitere Funktionen unterstützt werden. Besteht die Lampe aus mehreren Leuchtkörpern, die nicht über separate Dienste angesteuert werden, so kann der Dienst dennoch ein getrenntes Ein- und Ausschalten erlauben oder das Ein- und Ausschalten bestimmter Beleuchtungsmuster. Bei farbigen Leuchtkörpern kann der Treiber darüber hinaus auch eine Farbwahl ermöglichen.

- **Bewegungsmelder.** Dieser Treiberdienst ermöglicht die Auswertung der Signale eines Bewegungsmelders. Hier sind wieder verschiedene Varianten möglich. Die erste Variante ermöglicht die aktive Abfrage des aktuellen Status, d. h. ob momentan Bewegungen erkannt werden oder nicht. Eine zweite Variante erlaubt das Anmelden zum Empfang von Benachrichtigungen, die gesendet werden, sobald sich der Status ändert. Wenn Bewegungen erkannt werden, so wird eine entsprechende Meldung an registrierte Dienste gesendet. Werden keine weiteren Bewegungen mehr erkannt, so kann dies ebenfalls den registrierten Diensten mitgeteilt werden.

- **Kaffeemaschinensteuerung.** Auch übliche Haushaltsgeräte, wie eine Kaffeemaschine, können über Treiberdienste angesteuert werden. Dabei können die Funktionen des Geräts durch andere Dienste aktiviert werden. Auf diese Weise ist z. B. eine Zeitsteuerung möglich. Entsprechende Top-Level-Dienste können aber auch verschiedene Benutzerschnittstellen zur Fernsteuerung anbieten. So wäre etwa eine Fernsteuerung über ein Handy oder PDA eine geeignete Anwendung.

2.4.2 Sicherheitsszenario

Dienste zur Steigerung der Sicherheit im eHome sind ein weiteres typisches Anwendungsfeld. Dabei kann es um den Schutz vor Einbrüchen gehen, es sind aber weitere Sicherheitsdienste möglich, z. B. zur Erkennung von Rauch oder Bränden und zur entsprechenden Reaktion auf solche Fälle. Im Folgenden wird ein Alarmdienst als Beispiel für einen Top-Level-Dienst aus dem Bereich *Sicherheit* näher erläutert.

- **Alarmdienst.** Der Alarmdienst dient dem Schutz vor Einbrüchen im eHome. Aufgrund der Daten verschiedener Sensoren im eHome kann durch den Dienst eine Einbruchsituation erkannt werden. Als Reaktion auf eine solche Situation löst der Dienst ein Alarmsignal aus und setzt einen Notruf ab. Der Alarmdienst muss zunächst aktiviert werden, was manuell durch die Bewohner erfolgen kann oder aber automatisch, wenn keine Personen mehr im Haus erkannt werden. Eine solche Abwesenheitsfunktion kann neben dem Aktivieren des Alarmdienstes auch weitere Aktionen ausführen, wie etwa das Ausschalten der Beleuchtung und das Herunterregeln der Heizung. Das Erkennen einer Einbruchsituation kann z. B. durch Verwendung von Glasbruchsensoren, Bewegungsmeldern und Kameras erfolgen. Je nach der Ausstattung des eHomes können auch dynamisch weitere Sensoren mit einbezogen werden. Liegt nach der Analyse der Sensordaten eine Einbruchsituation vor, so werden Reaktionen durch den Dienst eingeleitet. Es werden z. B. die Türen und Fenster verschlossen, die Beleuchtung wird in einen Blinkzustand versetzt und eine Sirene an der Außenseite des Gebäudes wird aktiviert. Zusätzlich wird der Einbruch automatisch gemeldet. Dazu kann eine Meldung an einen Sicherheitsdienst, die Polizei oder den Bewohner gesendet werden. Schließlich kann der Alarmzustand wieder zurückgesetzt werden.

Die für den Alarmdienst benötigten Treiberdienste wurden teils schon oben angesprochen. Hier werden noch einige weitere Details erläutert.

- **Glasbruchsensor.** Der Treiber für Glasbruchsensoren ermöglicht die Weiterleitung einer Benachrichtigung, wenn ein Glasbruch erkannt wird. Da jeder Sensor über einen eigenen Treiberdienst im eHome repräsentiert wird, kann der Glasbruch auch geortet werden, da die Dienste in das Kontextmodell eingebunden sind.

- **Kamerasteuerung.** Die Kamerasteuerung erfolgt ebenfalls über einen entsprechenden Treiberdienst. Die Hauptaufgabe des Dienstes liegt darin, den Zugriff auf die Videodaten, die von der Kamera aufgezeichnet werden, zu ermöglichen. Diese Daten stehen dann zur weiteren Verarbeitung, z. B. zur Erkennung von Bewegung oder auch zur Identifizierung von Personen zur Verfügung. Je nach Art der Kamera kann auch die Ausrichtung des Blickwinkels der Kamera über den Treiberdienst realisiert werden.

- **Türsteuerung.** Der Dienst zur Türsteuerung ermöglicht das Abfragen des Zustands der Tür, d. h. ob sie geöffnet oder geschlossen ist. Außerdem kann die Tür über die Schnittstelle des Dienstes auch ver- und wieder entriegelt werden. Abhängig von der Tür kann auch das aktive Schließen oder Öffnen der Tür realisiert werden, z. B. bei elektrisch betriebenen Schiebetüren.

- **Fenstersteuerung.** Die Fenstersteuerung ist der Türsteuerung ähnlich. Neben der Abfrage des Öffnungszustands kann auch hier die Verriegelung ermöglicht werden. Ebenso kann hier die aktive Steuerung des Fensters unterstützt werden. So kann das Fenster geöffnet, geschlossen oder auf Kipp gestellt werden. Dies ist neben der Nutzung durch den Alarmdienst auch eine Möglichkeit durch automatisches und effektives Lüften, Energie einzusparen.

- **Alarmsignalgeber.** Der Alarmsignalgeber ist außen am eHome angebracht und dient der Abschreckung von Einbrechern. Es wird Aufmerksamkeit bei Passanten und Nachbarn erzeugt, was den Eindringling zum Aufgeben bewegen soll. Neben einem akustischen Signal kann auch durch ein visuelles Blinksignal Aufmerksamkeit erzeugt werden.

2.4.3 Unterstützung von Senioren

Die Unterstützung von Senioren ist ein bedeutendes Szenario bei der Entwicklung von eHome-Systemen, besonders vor dem Hintergrund der demographischen Entwicklung. Dies zeigen auch die Forschungsaufwendungen im Themenbereich *Ambient Intelligence* in Europa, insbesondere auch in Deutschland (vgl. Abschnitte 2.1 und 2.2). In einer altengerechten Wohnumgebung können sehr viele Dienste aus den unterschiedlichsten Kategorien eingesetzt werden, allein die Telemedizin bietet hier vielfältige Möglichkeiten, Dienste zu entwerfen. Diese Dienste sind jedoch häufig auch auf eine spezielle Hardwareausstattung angewiesen und weisen eine hohe Komplexität auf. Zur Betrachtung in dieser Arbeit wird ein sehr viel einfacherer aber dennoch nützlicher Dienst betrachtet, der sich an die Bedürfnisse älterer Menschen individuell anpassen lässt.

➯ **Klingeldienst.** Ein Klingeldienst hat die Aufgabe die Bewohner zu benachrichtigen, wenn an der Haustür des eHomes ein Besucher ein entsprechendes Signal auslöst. Die Benachrichtigung kann dabei auf unterschiedliche Arten erfolgen. Die offensichtlichsten Möglichkeiten sind ein akustischer Signalton oder ein visuelles Lichtsignal. Visuelle Signale werden z. B. auch in Lichtrufanlagen eingesetzt, die in Krankenhäusern zum Rufen einer Schwester verwendet werden. In Brandmeldeanlagen werden sie ebenfalls eingesetzt. Im Fall eines schwerhörigen Bewohners kann es sinnvoll sein, anstatt des sonst üblichen akustischen Klingelsignals ein solches visuelles Signal über die im eHome vorhanden Lampen auszulösen. So kann z. B. ein Blinken auf das Klingeln aufmerksam machen. Eine weitere Alternative zur akustischen Benachrichtigung wäre das Einblenden einer Nachricht auf einem Display, was insbesondere sinnvoll ist, wenn der zu benachrichtigende Bewohner gerade fernsieht. Dieses Beispiel zeigt, wie Dienste, die auch von allgemeiner Verwendung im eHome sind, spezifisch an Bedürfnisse älterer Bewohner angepasst werden können.

2.4.4 Allgemeine Dienste

Einige allgemeine Treiberdienste können von den verschiedensten Top-Level-Diensten verwendet werden. Als Beispiel seien hier Treiberdienste für Geräte zur Interaktion der Benutzer erwähnt. Dazu können übliche Schaltelemente verwendet werden, die auch heute in einer Wohnumgebung eingesetzt werden.

- **Schalter.** Dieser Treiberdienst ermöglicht die Einbindung von Schaltern in das eHome-System. Ein Schalter hat die zwei Zustände *ein* und *aus*. Der Treiber ermöglicht die Abfrage des aktuellen Zustands, es ist auch möglich automatische Benachrichtigungen über alle Schaltvorgänge zu empfangen. Diese Variante ist die einfachste Form des Schalters, spezielle Schalter können auch mehr als zwei Zustände unterscheiden oder Kombinationen verschiedener Schaltelemente ermöglichen.
- **Taster.** Der Taster hat im Gegensatz zum Schalter nur einen Zustand. Wird der Taster gedrückt, so wird ein Signal ausgelöst, das über den Treiberdienst an andere Dienste weitergeleitet werden kann.
- **Regler.** Ein Regler ermöglicht die stufenlose Einstellung einer bestimmten Größe. Dadurch können Parameter von Top-Level-Diensten manuell durch den Benutzer eingestellt werden. Typische Anwendungen sind z. B. das Dimmen einer Lampe, das Einstellen der Heizung oder das Regulieren einer Lautstärke. Regler gibt es in verschiedenen Ausführungen, häufig werden Schieberegler oder Drehregler verwendet.

2.5 eHome-Systeme in der Praxis

In der Praxis sind eHomes noch nicht verbreitet. Bei den bisher realisierten Gebäuden handelt es sich üblicherweise um Prototypen, die von Forschungseinrichtungen und Unternehmen betrieben werden, um neue Anwendungen zu entwickeln und in einem praktischen Umfeld zu erproben. Vereinzelt existieren auch private Installationen, die in Hobbyprojekten entstanden sind. Während die oben genannten Prototypen meist nur zeitweise tatsächlich bewohnt sind, werden die privaten Umsetzungen tatsächlich und dauerhaft genutzt, sodass hier der real zu erreichende Anwendernutzen im Vordergrund steht. In Forschungs- und Industrieprojekten werden andererseits auch weiterführende technische Entwicklungen vorangetrieben, die sich in der Zukunft in praktische Anwendungen umsetzen lassen. Solche Konzepte können im Rahmen einer rein privaten Initiative kaum umgesetzt werden.

Abbildung 2.4: inHaus1 am Fraunhofer IMS in Duisburg. (Quelle: [Fra09])

Das *inHaus-Zentrum* [Fra09] wurde vom Institut für mikroelektronische Schaltungen und Systeme (IMS) der Fraunhofer-Gesellschaft in Duisburg errichtet. Das Ziel des Zentrums ist die Entwicklung neuer Lösungen für „intelligente Gebäude". Dabei werden vollwertige Gebäude als Prototypen eingesetzt. Mit dem *inHaus1* wurde im Jahr 2001 zunächst ein Wohngebäude errichtet, das als Testumgebung zur Entwicklung von *Smart Home*-Anwendungen für private Wohnhäuser dient. Abbildung 2.4 zeigt eine Außenansicht des *inHaus1*, das als Doppelhaus konstruiert wurde. In der einen Hälfte befinden sich die Wohnräume, die andere Hälfte dient als Arbeitsbereich und Werkstatt. Dort ist auch der größte Teil der Gebäudetechnik untergebracht. An dem Projekt sind zahlreiche Industriepartner beteiligt, die unter anderem die Geräte liefern, die im *inHaus1* installiert wurden. Um neue Anwendungen zu ermöglichen, ist ein vernetzter Haushalt erforderlich. Dazu müssen viele verschiedene Standards integriert werden, weshalb eine Middleware-Infrastruktur entwickelt wurde, die zwischen den verwendeten Geräten vermittelt. Auch hier werden für eHomes typische Anwendungsfelder, wie das Senken des Energieverbrauchs, das Steigern von Komfort und Sicherheit, sowie die Unterstützung von Senioren, untersucht. In vielen der betrachteten Szenarien geht es um die Fernsteuerung von Geräten und die Automatisierung von Abläufen. So dient z. B. der Fernseher als zentrale Benutzerschnittstelle zur Steuerung des Gebäudes. Andere Installationen ermöglichen die Überwachung des Gebäudes, so z. B. eine Anzeigetafel an der Haustür, die beim Verlassen des Hauses an noch eingeschaltete Geräte erinnert.

Abbildung 2.5: inHaus2 am Fraunhofer IMS in Duisburg. (Quelle: Fotograf Fotostudio Arnolds e. K., Aachen)

Im November 2008 wurde das *inHaus2* eröffnet, das für weiterführende Projekte des Fraunhofer IMS genutzt wird. Im Gegensatz zum *inHaus1* handelt es sich dabei um eine Nutzimmobilie zur Entwicklung von Techniken für intelligente Gebäude im Allgemeinen, sogenannte *Smart Buildings*. Dabei werden verschiedene Ansätze aus dem *inHaus1* weiterentwickelt und neue Anwendungen erschlossen. Ein wesentlicher Bereich ist dabei die medizinische Versorgung. Neben der bereits im *inHaus1* angestrebten besseren Versorgung von Senioren und behinderten Menschen werden hier auch Assistenzsysteme für Krankenhäuser entwickelt. Dabei geht es z. B. um die Optimierung von Abläufen in einem Operationssaal, wie etwa die Zeiten für Umbau und Einrichtung der Gerätschaften und die Anbindung an Krankenhausinformationssysteme (KIS). Die gewonnenen Erkenntnisse sollen in zukünftigen medizinischen Versorgungszentren (MVZ) und Pflegeheimen Anwendung finden. Darüber hinaus wird im *inHaus2* auch an integrierten Systemen für Hotels, Büros und Veranstaltungen geforscht. Abbildung 2.5 zeigt das *inHaus2* in einer Außenansicht.

Anders als in der vorliegenden Arbeit wird in den Projekten des inHaus-Zentrums von der Neuinstallation eines Gebäudes ausgegangen und es werden derzeit verfügbare Bustechniken und entsprechende Geräte installiert. Der Schwerpunkt der Forschung liegt in der Vernetzung und Integration der verfügbaren Techniken. In dieser Arbeit wird hingegen untersucht, wie kostengünstige eHome-Systeme zu realisieren sind, die auf einer Verschiebung von Funktionalitäten auf die Softwareebene basieren, und welche Probleme dabei zu überwinden sind. Dazu wird nicht in erster Linie die Vermittlung zwischen derzeit verfügbaren Produkten untersucht, sondern die Realisierung von Funktionalitäten auf Basis dynamisch komponierter eHome-Dienste, die die vorhandene Hardware nutzen. Im inHaus-Zentrum werden teils ganz neue Produkte entwickelt, die entsprechend spezifischer Anforderungen, Hard- und Software zu einer Gesamtlösung integrieren, z. B. für medizinische Anwendungen. Die Vernetzung steht dabei im Vordergrund, nicht die Nutzung bereits verfügbarer Geräte durch verschiedene eHome-Dienste.

Ähnlich dem inHaus-Zentrum gibt es verschiedene Forschungsprojekte von Unternehmen, die ebenfalls eHome-Umgebungen realisieren. So z. B. das *Philips HomeLab* [Phi03] im niederländischen Eindhoven. Dabei wird besonders auf die Benutzerinteraktion und die Vernetzung von Unterhaltungselektronik fokussiert. Das *SmartHome Paderborn* [Sma09b] wird von einem Verein betrieben, der verschiedene heute am Markt verfügbare Produkte für vernetzte und automatisierte Häuser demonstriert. Das *T-Com-Haus* [Deu09] ist ein Gemeinschaftsprojekt der Firmen T-Com, WeberHaus, Siemens und Neckermann. Es hat eine ähnliche Zielrichtung wie die obigen Projekte, wobei hier insbesondere durch die Partnerschaft mit der Deutschen Telekom auf Multimedia und Kommunikationstechniken eingegangen wird.

Auch an Universitäten, insbesondere in den USA, wurden bereits seit längerer Zeit Prototypen von eHomes entwickelt. Ein Beispiel für ein solches Projekt ist das *MIT House_n* [Hou09, Int06] des Massachusetts Institute of Technology (MIT). Das Projekt wird von einer interdisziplinären Forschungsgruppe durchgeführt. Das Zusammenspiel von Menschen und zukünftigen Technologien steht bei diesem Projekt im Mittelpunkt. Den Bewohnern soll aber nicht durch umfassende Automatisierung jegliche Kontrolle und Entscheidungskompetenz entzogen werden.

Das *AwareHome* [Awa09] des Georgia Institute of Technology ist eine weitere Versuchsumgebung für eHomes. Dabei soll die Umgebung die Fähigkeit besitzen, sich und seine Bewohner „wahrzunehmen“. Um Kontextdaten zu ermitteln, werden verschiedene Sensoren im Haus eingesetzt. Auf Grundlage der Kontextinformationen kann sich das Haus dann automatisch den Benutzern und dem aktuellen Umgebungszustand anpassen.

Das *Adaptive House* [Uni09, Moz05] der University of Colorado in Boulder hat zum Ziel, Wohnumgebungen mit künstlicher Intelligenz zu versehen. Dazu wird das Benutzerverhalten auf Basis von Sensordaten analysiert, um daraus zukünftiges Verhalten abzuleiten. Den Vorhersagen entsprechend können dann Parameter der Umgebung, wie Temperatur oder Beleuchtung automatisch gesteuert werden. Das Ziel ist ein Haus, das von sich aus agiert und reagiert, anstatt Funktionalitäten nur auf Basis expliziter Benutzerinteraktion auszuführen.

Die hier erläuterten Projekte stellen nur einen kleinen Ausschnitt bisheriger Forschungsaktivitäten auf Basis realer Gebäude mit eHome-Charakter dar. Sie geben einen Überblick über den derzeitigen Stand der Technik, d. h. die Möglichkeiten eHomes auf Basis der verfügbaren Technik umzusetzen. Dabei sind jedoch nach wie vor noch viele weitere Schritte erforderlich bis man tatsächlich von Ubiquitous Computing und Ambient Intelligence sprechen kann. Die heutigen Implementierungen stellen erste Versuche dar. Die darin gewonnenen Erfahrungen bieten jedoch wertvolle Informationen für die zukünftige Entwicklung.

2.6 Zusammenfassung

In diesem Kapitel wurden Hintergründe und begriffliche Unterscheidungen im Zusammenhang mit eHomes beschrieben. Außerdem wurde auf Anforderungen aus technischer und aus Benutzersicht eingegangen. Die betrachteten Beispiele und bisherigen Implementierungen verdeutlichen die zukünftigen Anwendungsmöglichkeiten. Nicht alle der besprochenen Themenbereiche finden in den Lösungsansätzen dieser Arbeit gleichermaßen Anwendung, sie stellen jedoch eine Übersicht dar, die für das weitere Verständnis der Arbeit hilfreich ist.

Bei den in der vorliegenden Arbeit entwickelten Konzepten geht es in erster Linie um die softwaretechnische Sicht auf eHomes. In vielen der hier diskutierten Projekte werden hingegen andere Aspekte in den Vordergrund gestellt, wie die Vernetzung am Markt verfügbarer Geräte oder die Integration unterschiedlicher technischer Infrastrukturen. Auch diese Aspekte sind für ein Gesamtbild der Thematik wichtig, da sich daraus die Abgrenzung der in dieser Arbeit untersuchten Konzepte und ein Verständnis der Zusammenhänge ergibt. Auch die in der vorliegenden Arbeit betrachteten Konzepte werden in verschiedenen anderen Forschungsprojekten untersucht. Diese weisen aber nicht immer einen spezifischen Anwendungsbezug auf und sind nicht explizit auf eHome-Systeme ausgerichtet, wenngleich die Lösungsansätze auch in diesem Bereich anwendbar sind. Verwandte Forschungsansätze werden in den Abschnitten 4.5 und 5.8 der folgenden Kapitel genauer betrachtet.

Kapitel 3

Grundlagen

Für das Verständnis der in dieser Arbeit entwickelten Lösungskonzepte sind einige Grundlagen erforderlich, die in diesem Kapitel beschrieben werden. Zunächst wird auf komponentenbasierte Software eingegangen und auf den Übergang und die Unterschiede zu serviceorientierten Ansätzen. Ein weiteres Thema ist die modellgetriebene Softwareentwicklung, die insbesondere für die Umsetzung des Ansatzes zur strukturellen Adaption in Kapitel 4 eine Rolle spielt. Abschließend werden Vorarbeiten im eHome-Projekt vorgestellt, die eine Grundlage für die Lösungsansätze in dieser Arbeit darstellen.

3.1 Komponentenbasierte Software

Anfangs befasste sich die Softwaretechnik im Wesentlichen mit monolithischen Systemen, die für individuelle Aufgaben, angepasst an die spezifischen Anforderungen des Anwenders, entwickelt wurden. Die Entwicklung solcher *Individualsoftware* führt zu einem Produkt, das idealerweise die Anforderungen genau erfüllt und die Abläufe der Anwender optimal unterstützt. Allerdings ist die Entwicklung solch individueller Komplettlösungen sehr kostenintensiv, da ein vollständiger Entwicklungsprozess durchgeführt werden muss. Um solche Systeme an geänderte Anforderungen oder Systemumgebungen anzupassen, sind darüber hinaus aufwendige Wartungsarbeiten erforderlich.

Eine Alternative zur Entwicklung von Individualsoftware ist die Verwendung einer *Standardsoftware*. Sofern eine Standardlösung verfügbar ist, kann diese sofort eingesetzt werden, ohne dass ein spezifischer Entwicklungsprozess erforderlich ist. Die Wartung und Weiterentwicklung wird vom Hersteller gewährleistet und bei Bedarf können neue Versionen auf dem System des Anwenders installiert werden. Eine Anpassung an die individuellen Bedürfnisse des Anwenders ist jedoch nur schwer oder gar nicht möglich, der Anwender muss sich vielmehr der Standardlösung anpassen. Diese Variante ist daher auch nicht optimal, besonders wenn die individuellen Abläufe in einem Unternehmen für dessen Marktvorteil essentiell sind.

Komponentenbasierte Software stellt einen Mittelweg zwischen den beiden Extremformen der Individualsoftware und der Standardsoftware dar. Dabei werden vorgefertigte *Standardkomponenten*, d. h. wiederverwendbare Softwarebausteine, zu einer *individuellen* Softwarelösung komponiert. Durch die Wiederverwendung der standardisierten Komponenten können Kosten und Entwicklungsaufwand eingespart werden. Die Wartung und Weiterentwicklung wird üblicherweise von den entsprechenden Herstellern durchgeführt. Neue Versionen einzelner Komponenten können so in kleinen Schritten in die komponentenbasierte Softwarelösung eingepflegt werden. Trotz der Verwendung vorgefertigter Software kann aber auch eine individuelle Anpassung erfolgen, da die Gesamtlösung aus vielen verschiedenen Komponenten besteht, die je nach Anwendungsfall ausgewählt, angepasst und komponiert werden. Im Idealfall führt ein solcher Kompromiss zu einer individuellen Software, die aber trotzdem die Vorteile einer Standardlösung bietet.

3.1.1 Begriffsklärung

Für die Entwicklung komponentenbasierter Software werden in erster Linie zunächst die passenden *Komponenten* benötigt. Davon, wie Komponenten auszusehen haben, die die oben beschriebenen Eigenschaften besitzen, gibt es unterschiedliche Auffassungen. Verschiedene Definitionen und Modelle existieren parallel zueinander [LW05]. Von den in der Literatur verbreiteten Definitionen seien hier zwei genannt:

> „A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. [SGM02]“

> „We define software components simply as building blocks from which different software systems can be composed. [. . .] We want them to be plug-compatible by design and to be combinable in as many ways as possible. We want to minimize code duplication and maximize reuse. [CE00]“

Die wesentliche Eigenschaft von Komponenten ist die *Komponierbarkeit* zu größeren Softwareeinheiten. Dazu benötigen Komponenten definierte Schnittstellen, die jedoch eine größtmögliche Flexibilität bezüglich der Komponierbarkeit mit anderen Komponenten aufweisen. Nur dann ist eine sinnvolle Wiederverwendung möglich. Sämtliche Abhängigkeiten einer Komponente sind explizit in Form ihrer Schnittstellen definiert. Die Schnittstellen bilden die Grundlage der Komposition. Hersteller und Verwender einer Komponente sind meist verschieden. Es sollen schließlich gerade die bereits existierenden Komponenten wiederverwendet werden, um ein neues Softwaresystem zu realisieren. Die Austauschbarkeit von Komponenten ist ein wichtiger Aspekt, der die Flexibilität eines Systems erhöht. Ist eine effizientere, günstigere oder anderweitig geeignetere Komponente mit derselben Funktionalität verfügbar, so sollte die bisher eingesetzte Komponente durch diese austauschbar sein. Bei der Entwicklung einer komponentenbasierten Software kann daher eine höhere Unabhängigkeit von Drittanbietern erreicht werden.

Je nach Anwendungsfall werden weitere Eigenschaften mit Softwarekomponenten assoziiert. Häufig wird z. B. gefordert, dass eine programmiersprachen- und plattformunabhängige Komposition möglich ist [BDH+98]. So kann ein System aus Softwarebausteinen entwickelt werden, die zum einen in unterschiedlichen Programmiersprachen implementiert wurden und zum anderen auf verschiedenen Plattformen verwendbar sind. Auf diese Weise kann die Flexibilität bei der Systementwicklung weiter erhöht werden. Anderenfalls müssen Komponenten für jede Programmiersprache und Zielplattform neu entwickelt werden, was zu einer Einschränkung der Wiederverwendung führt. Teilweise wird auch die Verschachtelung von Komponenten gefordert, sodass aus einzelnen Komponenten größere zusammengesetzt werden können. Auch Anforderungen an die Komponenteninfrastruktur werden verschiedentlich gefordert, wie etwa Persistenz, Event-Handling oder Introspektion [BDH+98]. Letztendlich hängt die anzuwendende Definition vom Anwendungsfall ab.

Das Konzept der komponentenbasierten Softwareentwicklung sieht vor, dass vorgefertigte Komponenten *ohne Änderung* zu einem Gesamtsystem komponiert werden können. Die Komposition sollte daher im Idealfall ohne manuelle Eingriffe und Modifikationen am Quellcode der Komponenten möglich sein. Da Komponenten häufig von Drittanbietern entwickelt werden, ist bei der Komposition der Quellcode meist auch gar nicht verfügbar. Dieser wird vom Hersteller nicht offengelegt, da dieser die Komponenten auch an andere Nutzer vermarkten will. Dann sind Eingriffe in den Quellcode nicht möglich und die Komponente muss ohne Modifikationen verwendet werden. In der Realität ist jedoch die direkte Komposition aller zu verwendenden Komponenten ohne Eingriffe häufig nicht möglich. Abweichend von der Idealvorstellung müssen dann die Verbindungen zwischen den Komponenten durch sogenannten *Glue Code* überbrückt werden. Dieser Glue Code ist Programmcode, der zur Adaption und Vermittlung zwischen Komponenten dient, sodass eine Verbindung von Komponenten möglich ist, auch wenn diese nicht unmittelbar zusammenpassen. Die entsprechenden Bausteine, die diese Vermittlung übernehmen, werden auch *Konnektoren* genannt. Bei der Entwicklung eines komponentenbasierten Systems muss der Aufwand für die Entwicklung solcher Konnektoren berücksichtigt werden. Dieser Aufwand ist nicht zu vernachlässigen und muss daher mit dem Aufwand der Neuentwicklung eines genau abgestimmten passenden Bausteins abgewogen werden.

3.1.2 Anwendung von Komponenten

Damit Softwarekomponenten zur Entwicklung eines Systems verwendet werden können, müssen sie einem gemeinsamen Komponentenmodell entsprechen. Ein solches Komponentenmodell legt die Semantik und die Syntax der Komponenten fest, sowie den Kompositionsmechanismus, nach dem die Komponenten miteinander verbunden werden [LW07]. Mit *Semantik* ist hier die Festlegung gemeint, was im gegebenen Komponentenmodell unter einer Komponente verstanden wird. Die Beschreibung einer Komponente umfasst sowohl technische als auch nicht-technische Aspekte [SGM02]. Wie eine solche Beschreibung aussieht, wird durch das Komponentenmodell festgelegt. Die *Syntax* einer Komponente wird im Allgemeinen durch die zugrunde liegende Programmiersprache des Komponentenmodells festgelegt. Die *Komposition* kann in unterschiedlichen Phasen des Softwareentwicklungsprozesses geschehen. Je nach Komponentenmodell kann sie in der Entwurfsphase, bei der Installation oder auch erst zur Laufzeit stattfinden.

Ein Komponentenmodell verlangt die Festlegung von Schnittstellen und Abhängigkeiten sowie die Festlegung, wie Komponenten installiert, gestartet, beendet und deinstalliert werden. Der Zugriff auf eine Komponente ist ausschließlich über ihre Schnittstellen möglich, da die Implementierung im Sinne des *Information Hiding* verborgen ist. Wie oben erwähnt liegen Komponenten häufig in Form von Binärcode vor, sodass auch aus diesem Grund kein Einblick in die Implementierungsdetails möglich ist. Damit eine Komposition erfolgen kann, müssen daher die Schnittstellen alle erforderlichen Informationen beinhalten. *Angebotene* Schnittstellen werden durch eine Komponente implementiert und anderen Komponenten zur Verwendung bereitgestellt. *Benötigte* Schnittstellen werden importiert und stellen Abhängigkeiten einer Komponente dar. Diese Abhängigkeiten müssen aufgelöst werden, bevor die Komponente ausgeführt werden kann.

Damit Komponenten verwendet werden können, wird eine dem Komponentenmodell entsprechende Plattform benötigt. Diese verwaltet den Lebenszyklus der Komponenten und erlaubt es, Instanzen zu erzeugen, diese zu konfigurieren und miteinander zu komponieren. Das geschieht auf Basis der Komponentenbeschreibung. Darüber hinaus kann eine Komponentenplattform auch Unterstützung für nicht-funktionale Aufgaben bieten, wie etwa Verteilung, Sicherheit oder Persistenz [CH04b].

Beispiele für Komponentenmodelle und ihre zugehörigen Plattformen sind JavaBeans[1] und Enterprise JavaBeans (EJB)[2] von Sun Microsystems oder das Component Object Model (COM)[3] und .NET[4] von Microsoft oder das CORBA Com-

[1] http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp
[2] http://java.sun.com/products/ejb/
[3] http://www.microsoft.com/com/default.mspx
[4] http://msdn.microsoft.com/de-de/netframework/default.aspx

ponent Model (CCM)[5] der Object Management Group (OMG). Auch das in Abschnitt 3.2.4 vorgestellte OSGi[6] der OSGi Alliance beschreibt ein Komponentenmodell. OSGi bietet darüber hinaus aber auch serviceorientierte Mechanismen, die in Abschnitt 3.2.4 erläutert werden.

Die Verwendung von Komponenten ist besonders dann von Bedeutung, wenn ein Softwaresystem nicht in einem klassischen Entwicklungsprozess realisiert werden kann. Dies ist bei eHome-Systemen der Fall, denn hier muss das System an die spezifischen Gegebenheiten des eHomes und die Bedürfnisse der Bewohner angepasst werden. Die Auswahl und Komposition der Komponenten kann nicht im Voraus durchgeführt werden, sondern erst zum Zeitpunkt der Installation oder sogar erst zur Ausführungszeit des Systems. Ein monolithisches System wäre dazu nicht geeignet, da für jede Anpassung an ein spezifisches eHome umfangreiche und aufwendige manuelle Eingriffe erforderlich wären. Werden hingegen Komponenten verwendet, so können diese zunächst unabhängig von der konkreten Anwendungsumgebung erstellt werden. Später müssen sie dann nur noch entsprechend parametrisiert und komponiert werden, was durch eine geeignete Softwareplattform unterstützt oder sogar vollständig automatisiert werden kann. Ein solcher Ansatz, der die Vorteile von Softwarekomponenten nutzt, wurde von NORBISRATH umgesetzt [Nor07] (siehe Abschnitt 3.4). Im Zusammenhang mit eHome-Systemen bietet jedoch die Verwendung von Komponenten allein keine ausreichende Unterstützung. Um eine bessere Unterstützung von Dynamik in eHome-Umgebungen zu erreichen, sind Konzepte der serviceorientierten Architektur nötig, die im folgenden Abschnitt erläutert werden.

3.2 Serviceorientierte Architektur

Die *serviceorientierte Architektur (SOA)* hat sich zu einem populären Ansatz bei der Entwicklung von Softwaresystemen entwickelt. Besonders zur informationstechnischen Unterstützung von Geschäftsprozessen in Unternehmen wird häufig serviceorientierte Architektur in Betracht gezogen. Die Serviceorientierung kann als eine weitere Entwicklungsstufe des komponentenbasierten Paradigmas

[5] http://www.omg.org/technology/documents/formal/components.htm
[6] http://www.osgi.org/Specifications/HomePage

betrachtet werden, die sich durch eine höhere Flexibilität als ein rein komponentenbasierter Ansatz auszeichnet. Daher können serviceorientierte Systeme auch in sehr dynamischen Kontexten eingesetzt werden. Was unter serviceorientierter Architektur verstanden wird, ist jedoch, wie auch bei der komponentenbasierten Softwareentwicklung, vom Anwendungsfall abhängig. Auch hier existieren verschiedene Auffassungen und Definitionen.

3.2.1 Von Komponenten zu Services

Serviceorientierte Systeme setzen sich ebenso wie komponentenbasierte Systeme aus verschiedenen Softwarebausteinen zusammen. In diesem Fall heißen diese Bausteine *Services* oder *Dienste*. Ein serviceorientiertes System zeichnet sich im Unterschied zu einem komponentenbasierten System durch die lose Kopplung und das späte Binden der Softwarebausteine aus. Die Bindung von Services findet erst zur Laufzeit des fertigen Systems statt. Die *Organization for the Advancement of Structured Information Standards (OASIS)*, ein Konsortium zur Entwicklung von IT-Standards, definiert in ihrem Referenzmodell serviceorientierte Architektur folgendermaßen:

> „Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. [MLM+06]“

Als besonderes Merkmal wird hier also die Verteilung von Funktionalitäten (engl. *Capabilities*), die von unterschiedlichen Anbietern zur Verfügung gestellt werden, hervorgehoben. Die Definition ist recht allgemein gehalten, es werden keine konkreten Vorgaben gemacht, wie die Verteilung umzusetzen ist und welche Infrastruktur dazu vorausgesetzt wird. Somit kann serviceorientierte Architektur in unterschiedlichsten Ausprägungen realisiert werden. Um serviceorientierte Architektur enger zu fassen, liegt es nahe, zunächst den Begriff *Service* genauer zu definieren. Auch der Servicebegriff ist unterschiedlich belegt. Eine mögliche Definition ist die folgende:

> „A service is functionality that is contractually defined in a service description, which contains some combination of syntactic, semantic, and behavioral information. [CH04b]“

Demnach ist ein Dienst eine angebotene Funktionalität, die durch eine Dienstbeschreibung definiert wird. Die Dienstbeschreibung macht dabei Festlegungen über Syntax, Semantik und das Verhalten des Dienstes. Eine andere Definition beschreibt Dienste als verteilte plattformunabhängige Anwendungen, die komponierbare Funktionalitäten anbieten:

> „Services are selfdescribing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications. Services perform functions, which can be anything from simple requests to complicated business processes. [Pap03]"

Darüber hinaus sollten Dienste *technologieneutral*, *lose gekoppelt*, *ortsunabhängig* sowie *flexibel konfigurierbar* sein [Pap03, HS05]. In einem serviceorientierten System sollten also Dienste unabhängig von der verwendeten Implementierungstechnologie und ihrem Ausführungsort interoperabel sein. Außerdem wird eine lose Kopplung gefordert. Zur Entwicklungszeit kann hier, anders als bei der komponentenbasierten Entwicklung, nur auf Dienstbeschreibungen zurückgegriffen werden. Zur Laufzeit können dann Dienstbindungen angelegt oder auch wieder entfernt werden, wobei über die Verfügbarkeit der gesuchten Dienste zuvor keine Aussagen gemacht werden können. Ein wichtiges Merkmal der Serviceorientierung ist, dass der Nutzer eines Dienstes zuvor nicht wissen muss, ob und wo dieser ausgeführt wird. Das Auffinden passender Dienste erfolgt erst bei Bedarf zur Laufzeit.

3.2.2 Aufbau serviceorientierter Systeme

In serviceorientierten Systemen werden drei wesentliche Rollen unterschieden: *Dienstanbieter*, *Dienstnutzer* und *Dienstvermittler* [HS05]. In Abbildung 3.1 ist dieser Aufbau und die Interaktion zwischen den drei Parteien dargestellt. Ein Dienstanbieter veröffentlicht seine Dienste bei einem Dienstvermittler. Dabei handelt es sich um einen Verzeichnisdienst, bei dem die veröffentlichten Dienste registriert werden. Dienstnutzer können Anfragen an den Dienstvermittler stellen, um benötigte Dienste zu finden. Ist ein passender Dienst verfügbar, so wird dem Dienstnutzer eine entsprechende Referenz übergeben, sodass dieser eine Verbindung zum Dienstanbieter herstellen kann. Dann ist eine direkte Interaktion zwischen dem Dienstnutzer und dem Dienstanbieter möglich. Aufgrund

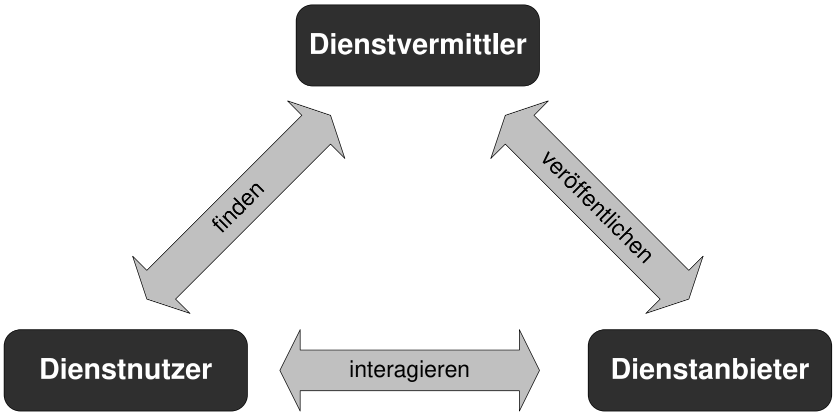


Abbildung 3.1: Rollen und Interaktion in einem serviceorientierten System.

der losen Kopplung besteht im Allgemeinen keine Garantie, dass ein gewünschter Dienst jederzeit zur Verfügung steht. Anfragende Dienste müssen daher mit dem Fall umgehen können, dass ein benötigter Dienst zum gewünschten Zeitpunkt nicht verfügbar ist. Auch während der Nutzung eines Dienstes kann es vorkommen, dass dieser nicht länger erreichbar ist, z. B. weil der Anbieter des Dienstes nicht länger zur Verfügung steht oder aufgrund eines Fehlers im Dienst oder im Netzwerk. Auch solche Fälle müssen von Dienstnutzern abgefangen und behandelt werden.

Eine Softwareplattform, die die Komplexität einer Infrastruktur vor der eigentlichen Anwendungssoftware, die auf ihr ausgeführt wird, verkapselt, wird häufig als *Middleware* bezeichnet. Der Einsatz einer Middleware vereinfacht die Anwendungsentwicklung wesentlich. Viele Middlewares verbergen die Komplexität einer Netzwerkinfrastruktur und ermöglichen so, auf einfache Weise verteilte Anwendungen zu entwickeln. Auch Komponentenplattformen nehmen häufig die Rolle einer Middleware ein und unterstützen die Entwicklung verteilter Anwendungen. Der Anwendungsentwickler braucht sich dann nicht um die Details der Netzwerkkommunikation zu kümmern. Eine Middleware kann aber noch weitere Querschnittsfunktionalitäten verkapseln, deren Komplexität dann ebenfalls vor dem Anwendungsentwickler verborgen wird. Die Entwicklung serviceorientierter Anwendungen kann dadurch unterstützt werden, dass Mechanismen zum Veröffentlichen, Suchen und Binden von Diensten durch eine Middleware angeboten werden. Dann muss der Entwickler diese Funktionalitäten nicht selbst implementieren, was den Entwicklungsaufwand und die Fehleranfälligkeit deutlich senkt. Serviceorientierte Architektur wird daher im Allgemeinen auf Basis einer serviceorientierten Middleware umgesetzt, die eine Infrastruktur für die Entwicklung von Diensten und Anwendungen gemäß dem Paradigma der Serviceorientierung ermöglicht.

3.2.3 Webservices

Webservices sind eine technische Umsetzung des serviceorientierten Paradigmas, die auf dem Internet als Netzwerkinfrastruktur und verschiedenen offenen Standards basiert [PTDL07]. Das *World Wide Web Consortium (W3C)* definiert Webservices wie folgt:

> „A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. [Wor04b]“

Wichtige Eigenschaften von Webservices sind also die Interaktion mit anderen Webservices oder Anwendungen, die Verteilung über mehrere Knoten in einem Netzwerk und der Einsatz spezifischer gemeinsamer Standards. Die wichtigsten Standards, die im Umfeld von Webservices eingesetzt werden, sind:

- **Web Services Description Language (WSDL)[7].** WSDL ist eine Beschreibungssprache zur Definition von Webservices. Dabei wird insbesondere die Schnittstelle definiert, die alle durch den Webservice zur Verfügung gestellten Operationen beschreibt. Auf diese Weise ist es anderen Webservices möglich, die angebotenen Operationen zu nutzen.

- **Simple Object Access Protocol (SOAP)[8].** SOAP ist ein Protokoll zum Austausch von Nachrichten zwischen Webservices. Die Nachrichten werden dazu in ein XML-Format übertragen, für die Übertragung wird meist auf

[7] http://www.w3.org/TR/wsdl20/
[8] http://www.w3.org/TR/soap/

das verbreitete Protokoll HTTP zurückgegriffen. SOAP ermöglicht es, auf die Schnittstellenoperationen eines Webservice zuzugreifen.

- **Universal Description, Discovery and Integration (UDDI)[9].** UDDI ist ein Standard zum Beschreiben und Auffinden von Webservice-Anbietern. Ein Verzeichnisdienst nach dem UDDI-Standard ermöglicht es, Webservices zu suchen und zu finden. Die Kommunikation erfolgt dabei über das oben genannte SOAP.

- **Web Services Business Process Execution Language (WS-BPEL)[10].** WS-BPEL ist eine Sprache zur Koordination von Webservices. Mit WS-BPEL können Geschäftsprozesse formalisiert werden, was eine genau Festlegung der Interaktion zwischen Webservices ermöglicht. Dies ist mit WSDL und SOAP allein nicht möglich. BPEL-Prozesse können von einer entsprechenden BPEL-Engine ausgeführt werden und kommunizieren dann entsprechend der Prozessdefinition mit den Webservices.

Webservices sind aus dem industriellen Umfeld heraus entstanden und verfolgen im Wesentlichen zwei Zielsetzungen [EF03]:

1. Eine bessere Unterstützung von B2B-Anwendungen (engl. *Business to Business, B2B*).

2. Die Integration von heterogenen Unternehmensanwendungen (engl. *Enterprise Application Integration, EAI*).

Klassische Anwendungen im Web sind B2C-Anwendungen (engl. *Business to Consumer, B2C*), die direkt von Nutzern verwendet werden und daher eine ansprechende grafische und für die manuelle Interaktion geeignete Benutzerschnittstelle benötigen. Sollen jedoch Anwendungen untereinander kommunizieren, so ist eine maschinell interpretierbare Schnittstelle für die Interaktion und den Datenaustausch erforderlich. Webservices bieten dazu eine geeignete Infrastruktur. Sollen Anwendungen über Unternehmensgrenzen hinweg miteinander interagieren, so handelt es sich um B2B-Anwendungen, die auf Basis von Webservices realisiert werden können. Sollen heterogene Anwendungen, die

[9] http://uddi.org/pubs/uddi_v3.htm
[10] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

nicht unmittelbar miteinander interagieren können, innerhalb eines Unternehmens integriert werden, so handelt es sich um das Anwendungsfeld der Enterprise Application Integratien. Auch dies ist auf Basis einer Webservice-Infrastruktur realisierbar.

Auch für die Realisierung von eHome-Diensten können Webservices eingesetzt werden. Die im Umfeld von Webservices verfügbaren Standards könnten dabei die Interoperabilität der eHome-Dienste fördern. Insbesondere die Entwicklung eHome-fähiger Geräte könnte auf Basis von Webservices vorangetrieben werden. Würden Geräte über eine eingebettete Webservice-Engine verfügen und selbst Webservices anbieten, so könnten ihre Funktionalitäten direkt über eine Intranet-Infrastruktur im eHome verfügbar gemacht werden. Solche Geräte sind bisher jedoch so gut wie nicht verfügbar. Sollen dennoch Webservices zur Realisierung von eHome-Diensten eingesetzt werden, so muss zunächst die Gerätefunktionalität durch einen Webservice verkapselt werden, der auf einem Knoten im Netz ausgeführt wird und das eigentliche Gerät dann über eine andere, proprietäre Infrastrukturtechnologie ansteuert. Wenn Webservices aber ohnehin nicht auf den Geräten selbst ausgeführt werden, ist die Verteilung auf mehrere Knoten im Netz nicht erforderlich. Die Dienste können in diesem Fall auch direkt auf dem Residential Gateway, dem zentralen Knotenpunkt im eHome, ausgeführt werden. Dann ist die Webservice-Infrastruktur jedoch nicht länger erforderlich und es kann eine nicht-verteilte serviceorientierte Middleware eingesetzt werden. Dazu eignet sich die OSGi Service Plattform, die im folgenden Abschnitt vorgestellt wird.

3.2.4 Die OSGi Service Plattform

Die *OSGi Service Plattform* bietet ein dynamisches Modulsystem auf Basis der Programmiersprache Java [WHKL08]. Darüber hinaus ist die OSGi Service Plattform eine serviceorientierte, komponentenbasierte Middleware, die eine standardisierte Verwaltung des Lebenszyklus von Komponenten bietet [OSG09]. OSGi unterstützt sowohl die komponentenbasierte Softwareentwicklung als auch serviceorientierte Architektur. Komponenten werden in OSGi *Bundles* genannt und können dynamisch installiert, gestartet, gestoppt und wieder deinstalliert werden. Sie können ihrerseits Dienste anbieten und diese bei einem Verzeichnisdienst, der OSGi Service Registry, anmelden. Dort können sie dann von anderen Diensten gefunden und genutzt werden.

Die OSGi Service Plattform ist in einer Spezifikation der OSGi Alliance definiert [OSG07b]. Darin ist unter anderem festgelegt, wie das OSGi Rahmenwerk, das die Infrastruktur der OSGi Service Plattform bereitstellt, implementiert werden muss. Darüber hinaus sind in [OSG07c] verschiedene Standarddienste definiert, die bei der Entwicklung eigener Dienste genutzt werden können. Die OSGi Alliance bietet selbst keine Implementierung der OSGi Service Plattform an, es existieren aber verschiedene andere Implementierungen. Darunter befinden sich sowohl kommerzielle als auch quelloffene und kostenlose Varianten. Beispiele sind Eclipse Equinox[11], Apache Felix[12], ProSyst mBedded Server[13] oder Knopflerfish OSGi[14].

Als die OSGi Service Plattform 1999 unter dem Namen *Open Services Gateway initiative* eingeführt wurde, war die primäre Zielrichtung die Unterstützung von Residential Gateways zur Heimautomatisierung. Heute hat sich das Anwendungsfeld von OSGi deutlich ausgeweitet und umfasst neben der Verwendung in Residential Gateways und eingebetteten Systemen, wie z. B. in Mobiltelefonen oder im Bereich Automotive, auch Anwendungen für Arbeitsplatzrechner und sogar Serveranwendungen [OSG07a]. In allen Fällen, in denen Modularisierung, Verwaltung von Abhängigkeiten, dynamisches Deployment und Serviceorientierung benötigt werden, bietet sich der Einsatz von OSGi an.

Die Architektur der OSGi Service Plattform ist eine Schichtenarchitektur. In Abbildung 3.2 ist der Aufbau schematisch dargestellt. Nicht alle von OSGi zur Verfügung gestellten Schichten müssen gleichermaßen in einer Anwendung genutzt werden. *Hardware und Betriebssystem* stellen die unterste Schicht dar und werden benötigt, um die darüber dargestellte *Ausführungsumgebung* zu betreiben. Da OSGi auf der Java-Plattform basiert, wird die Ausführungsumgebung immer durch eine konkrete Java-Laufzeitumgebung realisiert. Die folgenden vier Schichten stellen das eigentliche OSGi Rahmenwerk dar und implementieren die von OSGi zur Verfügung gestellten Mechanismen:

[11] http://www.eclipse.org/equinox/
[12] http://felix.apache.org/
[13] http://www.prosyst.com/products/osgi_framework.html
[14] http://www.knopflerfish.org/

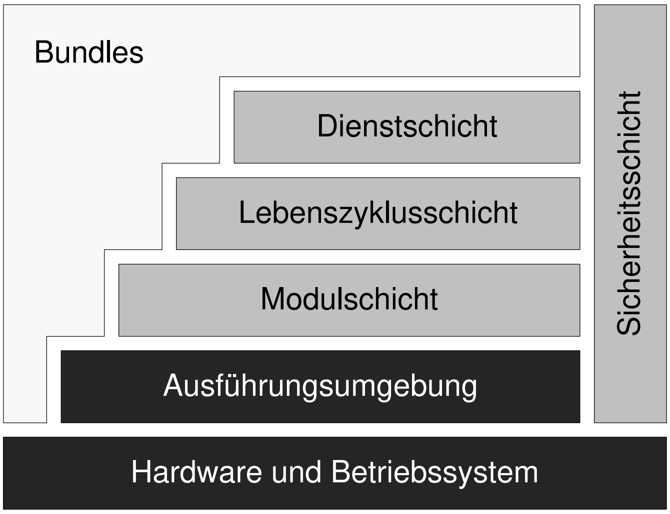


Abbildung 3.2: Aufbau der OSGi Service Plattform. (in Anlehnung an [OSG07b])

- Die **Modulschicht** legt fest, wie *Bundles*, die grundlegenden Softwarebausteine in OSGi, aufgebaut sind. Bundles stellen Komponenten dar und ergänzen die Java-Plattform um ein Modularisierungskonzept. Bundles bestehen aus Java-Klassen und ggfs. weiteren Ressourcen, die zusammen mit einem Manifest, das das Bundle beschreibt, in einem Java-Archiv zusammengefügt werden. Dieses kann dann auf der OSGi Service Plattform deployt werden.

- Die **Lebenszyklusschicht** definiert die möglichen Laufzeitzustände eines Bundles und ist für die Verwaltung dieser Zustände verantwortlich. Die Zustände im Lebenszyklus eines Bundles sind in Abbildung 3.3 dargestellt. Ein Bundle geht nach seiner Installation zunächst in den Zustand installed über. Dann müssen seine Abhängigkeiten zu anderen Bundles aufgelöst werden. Durch update/refresh kann eine erneute Prüfung der Abhängigkeiten stattfinden. Sind diese aufgelöst, so geht das Bundle in den Zustand resolved über. Aufgrund der dynamischen Eigenschaften von OSGi können Bundles auch wieder entfernt werden, wodurch die davon abhängigen Bundles in den Zustand installed zurückfallen, falls Abhängigkeiten dadurch nicht mehr erfüllt sind. Aus dem Zustand resolved heraus kann ein Bundle gestartet werden und befindet sich nach einem Übergangszustand starting im Zustand active. Das Stoppen erfolgt über einen weiteren

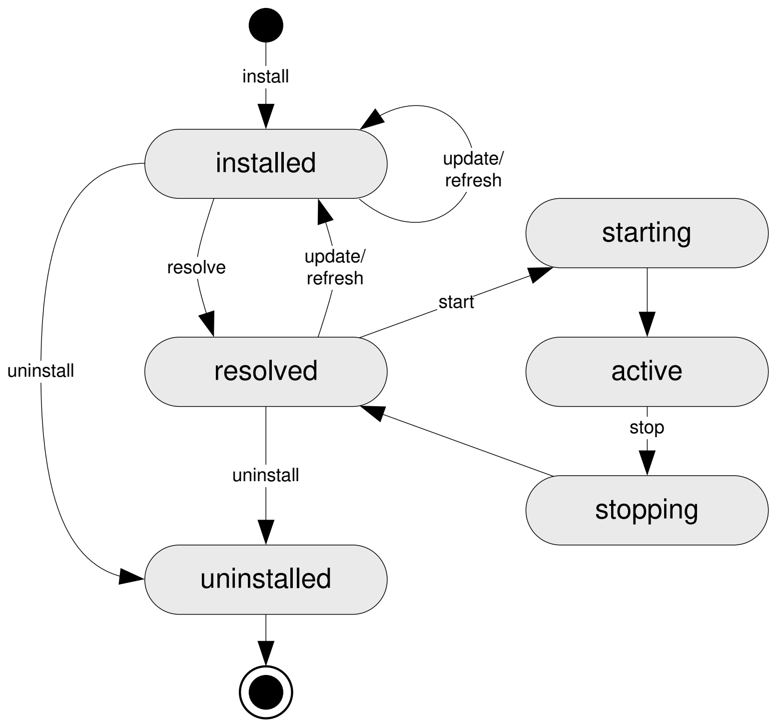


Abbildung 3.3: Lebenszyklus eines OSGi Bundles.

Übergangszustand stopping zurück in den Zustand resolved. Aus dem Zustand resolved und installed heraus kann ein Bundle deinstalliert werden. Mit dem Zustand uninstalled endet der Lebenszyklus des Bundles.

- Die **Dienstschicht** legt das Dienstmodell von OSGi fest. Ein Bundle kann Dienste erstellen und diese bei der OSGi Service Registry anmelden, um sie für andere Dienste verfügbar zu machen. In OSGi kann jedes Java-Objekt als Dienst registriert werden, es handelt sich also um einen sehr allgemein gehaltenen Mechanismus. Um passende Dienste finden zu können, werden diese unter einem oder mehreren Schnittstellennamen bei der OSGi Service Registry angemeldet. Da Dienste jederzeit an- und abgemeldet werden können, ist ihre Verfügbarkeit zu einem bestimmten Zeitpunkt nicht garantiert. Dies muss bei der Dienstimplementierung berücksichtigt werden. Die Verwaltung der Abhängigkeiten zwischen Diensten wird von OSGi nicht übernommen, es gibt jedoch verschiedene Mechanismen, die den Umgang mit dynamischen Diensten erleichtern.

- Die **Sicherheitsschicht** setzt verschiedene Sicherheitsaspekte um, die im Wesentlichen auf den Sicherheitsmechanismen der Java-Plattform basieren. OSGi-spezifische Erweiterungen ermöglichen es z. B. die Ausführungsrechte von Bundles festzulegen.

Ein wichtiger Aspekt ist die Verwaltung der Dienstabhängigkeiten in OSGi. Die Dienstabhängigkeiten sind nicht gleichzusetzen mit den Abhängigkeiten zwischen Bundles. Sie müssen daher getrennt von diesen in der Dienstschicht betrachtet werden. In OSGi bestehen verschiedene Möglichkeiten diese Abhängigkeiten zu verwalten.

- **Startlevels** ermöglichen die Festlegung einer relativen Startreihenfolge von Bundles. Auf diese Weise kann festgelegt werden, dass zuerst Bundles installiert werden, die Dienste registrieren, die dann von den Diensten anderer Bundles, die danach installiert werden, verwendet werden. Da sich Startlevels auf die Reihenfolge der *Bundle*-Installation beziehen, können die Abhängigkeiten der *Dienste* nur indirekt beeinflusst werden. Dynamische Situationen, in denen Dienste erst später zur Laufzeit bei der Service Registry an- und wieder abgemeldet werden, sind über diesen Mechanismus nicht abzudecken.
- **Service-Listener** können im Gegensatz zu Startlevels auch zur Behandlung von Abhängigkeiten dynamischer Dienste eingesetzt werden. Service-Listener werden bei der OSGi Service Registry angemeldet und empfangen dann sogenannte Service Events, wenn Dienste an- und abgemeldet werden oder ihren Zustand anderweitig ändern. Durch den Einsatz von Filtern können die zu empfangenden Service Events eingeschränkt werden. Dies schafft die Möglichkeit, auf dynamische Änderungen benötigter Dienste zu reagieren. Service-Listener sind zwar Bestandteil der OSGi Spezifikation, es wird jedoch empfohlen, sie nicht direkt einzusetzen, da sich die Verwendung recht komplex gestaltet und fehleranfällig ist. Alternativ sollten Service-Tracker eingesetzt werden, die zwar auf Service-Listenern basieren aber einfacher zu handhaben sind.
- **Service-Tracker** dienen der besseren Unterstützung im Umgang mit dynamischen Diensten. Ein Service-Tracker verkapselt dazu den Zugriff auf die OSGi Service Registry und kann automatisch auf das an- und abmelden relevanter Dienste reagieren. Beim Instanziieren eines konkreten Service-

Trackers kann durch Angabe von Klassennamen und Filtern festgelegt werden, auf welche Dienste der Service-Tracker reagieren soll. Grundlage der Service-Tracker sind die oben beschriebenen Service-Listener. Diese sind jedoch wegen der hohen Komplexität nicht zur direkten Verwendung empfohlen, der Service-Tracker ist daher die vorzuziehende Variante.

- **Declarative Services** sind eine Erweiterung von OSGi, die eine deklarative Beschreibung von Dienstabhängigkeiten erlaubt. Dazu werden sogenannte *Service Components* eingeführt, die von einer speziellen Laufzeitumgebung, der *Service Component Runtime (SCR)*, ausgeführt werden. Die Auflösung der Abhängigkeiten wird dann von dieser Laufzeitumgebung durchgeführt, der Dienst muss keine eigenen Mechanismen zu diesem Zweck implementieren. Eine genauere Beschreibung der Declarative Services erfolgt in Abschnitt 4.5. Dort werden Declarative Services auch mit dem in Kapitel 4 vorgestellten Mechanismus zur strukturellen Adaption verglichen.

Die Kommunikation zwischen Diensten lässt sich in OSGi klassisch über Methodenaufrufe realisieren. Ein Dienst kann einen benötigten Dienst über die Service Registry finden und die Operationen der dort eingetragenen Dienstschnittstelle aufrufen. Gerade in eingebetteten Systemen wird aber häufig auch eine ereignisbasierte Kommunikation benötigt. Dies bedeutet, dass Dienste durch andere Dienste über bestimmte Ereignisse mittels sogenannter *Events* informiert werden. Die übliche Implementierung eines solchen Mechanismus geschieht durch Einsatz des *Listener*- oder *Observer*-Entwurfsmusters [GHJV95]. Bei einer großen Anzahl von Dienstabhängigkeiten wird die ereignisbasierte Kommunikation über diese Entwurfsmuster jedoch ineffizient. Dies liegt daran, dass jeder Dienst, der Events sendet, die Zustände aller seiner Listener mittels eines Service-Trackers verfolgen muss. Nur dann kann sichergestellt werden, dass keine Dienste kontaktiert werden, die gar nicht mehr bei der Service Registry angemeldet und somit auch nicht mehr verfügbar sind. Um den Overhead an Verwaltungsaufwand und die damit verbundene Komplexität zu vermeiden, wird als Alternative das Entwurfsmuster *Whiteboard* vorgeschlagen [OSG04]. Das Whiteboard-Entwurfsmuster verwendet die OSGi Service Registry zur zentralen Verwaltung von Listenern, sodass diese Verwaltung nicht in den Event-Quellen selbst geschehen muss. Die Listener müssen sich dann nicht bei den Event-Quellen direkt anmelden, sondern lediglich bei der Service Registry. Vor dem Senden der Events muss die Event-Quelle bei der Service Registry die momentan angemeldeten Listener für das jeweilige Event abfragen und kann dann diese Listener benachrichtigen. Die Service Registry nimmt also die Rolle eines Whiteboards ein, was zu Vereinfachungen in den Dienstimplementierungen führt und die Effizienz steigert.

OSGi wird in dieser Arbeit für die Umsetzung einer Laufzeitumgebung für eHome-Dienste verwendet, es wird jedoch nicht der volle Umfang von OSGi eingesetzt. Insbesondere die Verwaltung der Dienstabhängigkeiten wird nicht mittels der OSGi Service Registry umgesetzt, da diese keine hinreichenden Mechanismen für die Verwaltung von Kontextinformationen bietet. Daher wird im Rahmen dieser Arbeit ein graphbasierter Ansatz zur Verwaltung von Diensten und Kontextinformationen eingeführt (siehe Kapitel 4). Die eHome-Dienste werden deklarativ beschrieben, ähnlich den OSGi Declarative Services. Diese Beschreibungen werden jedoch um verschiedene Informationen ergänzt, z. B. kontextbezogene Abhängigkeiten, die für die automatische Konfigurierung im eHome benötigt werden. Die graphbasierte Modellierung umfasst sowohl die Kontextinformationen, als auch die Dienstbeschreibungen, die Abhängigkeiten der Dienste und ihre momentanen Zustände. Alle relevanten Informationen sind in einem gemeinsamen Modell vereint. Darüber hinaus wird für die semantische Adaption eine Semantic Registry eingeführt, die benötigt wird, um semantische Dienstbeschreibungen zu verwalten und semantisch passende Dienste für die Komposition zu ermitteln. Darauf basierend können im später vorgestellten Ansatz zur semantischen Adaption (siehe Kapitel 5) zur Laufzeit automatisch Adapter generiert werden. OSGi bietet für diese Zwecke keine Mechanismen an, sodass hierfür ein neuer und von OSGi unabhängiger Lösungsansatz entwickelt werden musste.

3.3 Modellgetriebene Softwareentwicklung

Die in dieser Arbeit vorgestellte Laufzeitumgebung für eHome-Dienste ist größtenteils in einem modellgetriebenen Entwicklungsprozess entstanden. In der modellgetriebenen Softwareentwicklung (engl. *Model-Driven Software Development, MDSD*) stellen Modelle die zentralen Dokumente im Entwicklungsprozess dar. Die Verwendung graphischer Modelle eines höheren Abstraktionsgrades soll zu mehr Übersichtlichkeit und leichterem Verständnis führen als üblicher Quelltext. Durch die höhere Abstraktionsebene soll außerdem die plattformunabhängige Entwicklung unterstützt werden. So ist es meist möglich, aus den Modellen mit Hilfe geeigneter Werkzeuge Quelltext in unterschiedlichen Sprachen und für unterschiedliche Zielplattformen zu generieren. Dies bietet Vorteile im Entwicklungsprozess, da die Festlegung auf eine Plattform erst später geschehen kann und auch nachträglich die Übertragung auf andere Plattformen leichter möglich ist. Eine Plattform kann dabei ein bestimmtes Betriebssystem, eine bestimmte Programmiersprache oder auch eine Middleware sein [GPR06]. Modelle können nicht nur statische Informationen über die Struktur einer Anwendung festlegen, sondern auch dynamische Informationen, die das Verhalten der Anwendung bestimmen. Dies ist insbesondere vorteilhaft, weil damit die Konsistenz zwischen der Struktur und dem Verhalten der Anwendung besser gewährleistet werden kann. Die phasenübergreifende Verwendung von Modellen im Softwareentwicklungsprozess soll zudem die Divergenz zwischen formalisierten Anforderungen, dem entsprechenden Entwurf und dem letzten Endes implementierten und installierten Softwareprodukt verringern.

Die *Object Management Group (OMG)* hat unter dem Begriff *Model Driven Architecture (MDA)* [MM01, MM03] eine konkrete Realisierung des Paradigmas der modellgetriebenen Softwareentwicklung entworfen, die auf der getrennten Betrachtung von fachlichen und technischen Aspekten beruht. Zunächst werden abstraktere Modelle definiert. Die *Computation Independent Models (CIMs)* dienen der umgangssprachlichen Beschreibung und die *Platform Independet Models (PIMs)* zur fachlichen Modellierung von Geschäftsprozessen. Diese sind unabhängig von der Zielplattform und den einzusetzenden Technologien. Später werden konkretere Modelle erstellt. Die *Platform Specific Models (PSMs)* dienen der technischen Modellierung auf Basis einer spezifischen Zielplattform. Der Übergang von einem PIM zu einem PSM sollte ebenso wie die Quellcodegenerierung durch entsprechende Werkzeuge unterstützt werden.

3.3.1 Graphersetzungssysteme

Graphersetzungssysteme basieren auf Graphen als Datenstrukturen und Graphgrammatiken als mathematischem Fundament [Roz97, EEKR99]. Da Graphen sehr allgemeine Datenstrukturen sind, können sie in den verschiedensten Anwendungsfeldern eingesetzt werden [Hec06]. Ein Graphersetzungssystem wird durch eine Menge an Graphersetzungsregeln definiert, die lokale Transformationen auf der zugrunde liegenden Graphstruktur beschreiben [AEH+99]. Jede Regel hat eine linke und eine rechte Regelseite. Zur Laufzeit wird in einem Laufzeitgraphen (auch Wirtsgraph genannt) nach Vorkommnissen der linken Regelseite gesucht. An den entsprechenden Stellen wird dann eine Ersetzung durch die rechte Regelseite durchgeführt. Auf diese Weise werden Veränderungen am Laufzeitgraphen vorgenommen.

Graphersetzungssysteme können zur modellgetriebenen Softwareentwicklung eingesetzt werden, wenn sie das Generieren von Quellcode unterstützen. Ein Graphschema legt die Klasse von Graphen der zu erstellenden Anwendung fest und macht damit die strukturellen Vorgaben. Die Graphersetzungsregeln hingegen legen das Verhalten der Anwendung fest. Am Lehrstuhl für Informatik 3 der RWTH Aachen wurde bereits seit längerer Zeit an der Entwicklung von Graphersetzungssystemen und entsprechenden Softwarewerkzeugen gearbeitet. Dabei ist das Graphersetzungssystem *PROGRES* [Sch91, SWZ99] entstanden, sowie das *UPGRADE*-Rahmenwerk [BJSW02, Böh99], das die Entwicklung interaktiver graphischer Editoren zu einer PROGRES-Spezifikation ermöglicht.

PROGRES steht für ***Pro**grammed **G**raph **Re**writing **S**ystem* und ist eine streng typisierte, deklarative, operationale, visuelle Spezifikationssprache sowie eine zugehörige Entwicklungsumgebung. PROGRES basiert auf einer Verallgemeinerung attributierter Bäume, den gerichteten, attributierten, knoten- und kantenmarkierten Graphen. Diese Graphen werden auch *gakk*-Graphen genannt [Roz97]. Sie bestehen aus markierten (d. h. getypten) Knoten, markierten (d. h. getypten) und gerichteten Kanten zur Abbildung zweistelliger Beziehungen zwischen Objekten und Knotenattributen zur Abbildung von Objekteigenschaften.

Die Sprachdefinition legt fest, wie eine PROGRES-Spezifikation aufgebaut sein muss. Die PROGRES-Entwicklungsumgebung analysiert Spezifikationen bereits während der Erstellung in einem syntaxgesteuerten Editor und erlaubt die Ausführung durch einen Interpreter. Eine weitere wesentliche Funktionalität ist die Generierung von Quellcode, der für die Entwicklung von auf PROGRES-Spezifikationen basierenden Werkzeugen verwendet werden kann. PROGRES wurde bereits in größeren Projekten angewandt, z. B. für die integrierte Softwareentwicklungsumgebung IPSEN [Nag96] oder AHEAD [JSW00], ein System zur Verwaltung technischer Entwicklungsprozesse.

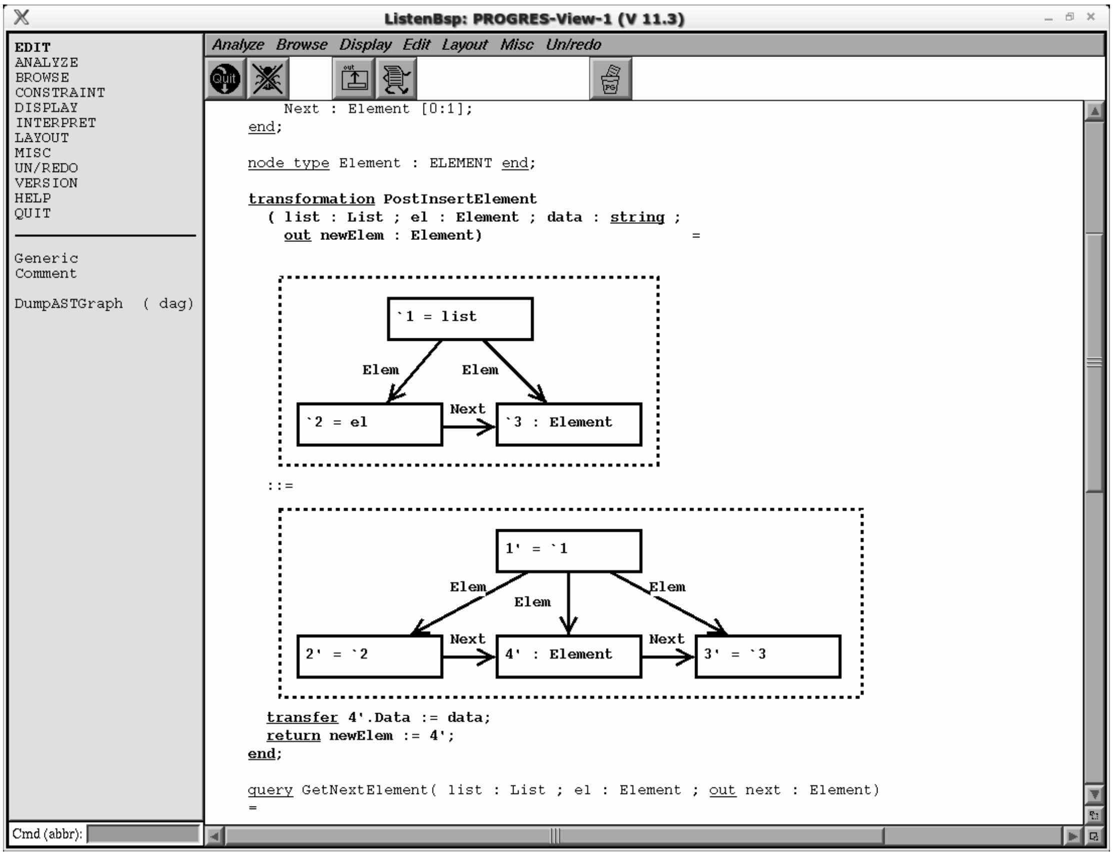


Abbildung 3.4: Editor der PROGRES-Entwicklungsumgebung.

Die Grundlagen von PROGRES und der zugehörigen Entwicklungsumgebung wurden in [Sch91] gelegt. In [Zün96] wurde die Arbeit an der PROGRES-Entwicklungsumgebung fortgeführt. Der syntaxgesteuerte Editor des PROGRES-Systems ist in Abbildung 3.4 zu sehen. Darin werden PROGRES-Spezifikationen erstellt und bearbeitet. Wie in der Abbildung zu erkennen ist, bestehen die Spezifikationen nicht nur aus textuellen Elementen, sondern umfassen auch größere graphische Anteile. So werden z. B. die beiden Regelseiten einer Graphersetzungsregel graphisch spezifiziert.

Zur Entwicklung interaktiver graphischer Werkzeuge auf Basis von PROGRES dient das UPGRADE-Rahmenwerk. UPGRADE steht für ***U**niversal **Pl**atform for **Gra**ph-Based **De**velopment* und ermöglicht das Erstellen einer spezifischen Anwendung zu einer gegebenen PROGRES-Spezifikation. Diese verkapselt die in der Spezifikation definierten Graphtransformationen durch komplexere Operationen, die über Menüs, Werkzeugleisten und Dialogfenster vom Anwender aufgerufen werden können. Die Darstellung des Laufzeitgraphen und seiner Elemente ist an die spezifischen Anforderungen anpassbar. Auch UPGRADE wurde in verschiedenen Projekten zur Entwicklung graphbasierter Editoren eingesetzt, unter anderem in einem Projekt zur softwareseitigen Unterstützung des konzeptuellen Gebäudeentwurfs [Kra07, HRK08a, HRK09], das am Lehrstuhl für Informatik 3 der RWTH Aachen durchgeführt wurde.

3.3.2 UML-basierte Ansätze

Viele Ansätze zur modellgetriebenen Softwareentwicklung basieren auf der *Unified Modeling Language (UML)* [Obj09a, Obj09b, Bal05, Rum04a, Rum04b] als Modellierungssprache. Im Folgenden wird Fujaba, ein Werkzeug, das auch im Rahmen dieser Arbeit eingesetzt wird, als Beispiel eines UML-basierten Ansatzes betrachtet. *Fujaba* steht für ***F**rom **U**ML to **Ja**va **a**nd **b**ack **a**gain* und ist ein Werkzeug zur modellgetriebenen Softwareentwicklung [FNTZ98, NNZ00].

Die Grundlagen von Fujaba sind zum einen die UML, die als Modellierungssprache eingesetzt wird, und zum anderen die Konzepte der Graphersetzung, die im vorherigen Abschnitt erläutert wurden. Wie PROGRES ermöglicht auch Fujaba das Generieren von Quellcode, in diesem Fall Java-Quellcode. Fujaba bietet eine ähnliche Ausdrucksstärke wie PROGRES, es gibt jedoch auch verschiedene Unterschiede. So wird in Fujaba beispielsweise kein Backtracking bei der Ausführung von Graphtransformationen unterstützt.

Die strukturellen Festlegungen über die zu entwickelnde Anwendung, die in PROGRES als Graphschema definiert werden, sind in Fujaba in Form von UML-Klassendiagrammen spezifiziert. Klassendiagramme bieten eine statische Sicht auf die Anwendung und stellen den Aufbau des Systems dar. In Abbildung 3.5 ist die Fujaba-Entwicklungsumgebung mit einem zur Bearbeitung geöffneten Klassendiagramm dargestellt. Zur besseren Übersicht ist es möglich, mehrere Sichten auf größere Klassendiagramme zu definieren, die jeweils nur Ausschnitte des Gesamtmodells zeigen. Außerdem können die Details der Klassen, d. h. Attribute und Operationen, je nach Bedarf im Diagramm ein- und ausgeblendet werden.

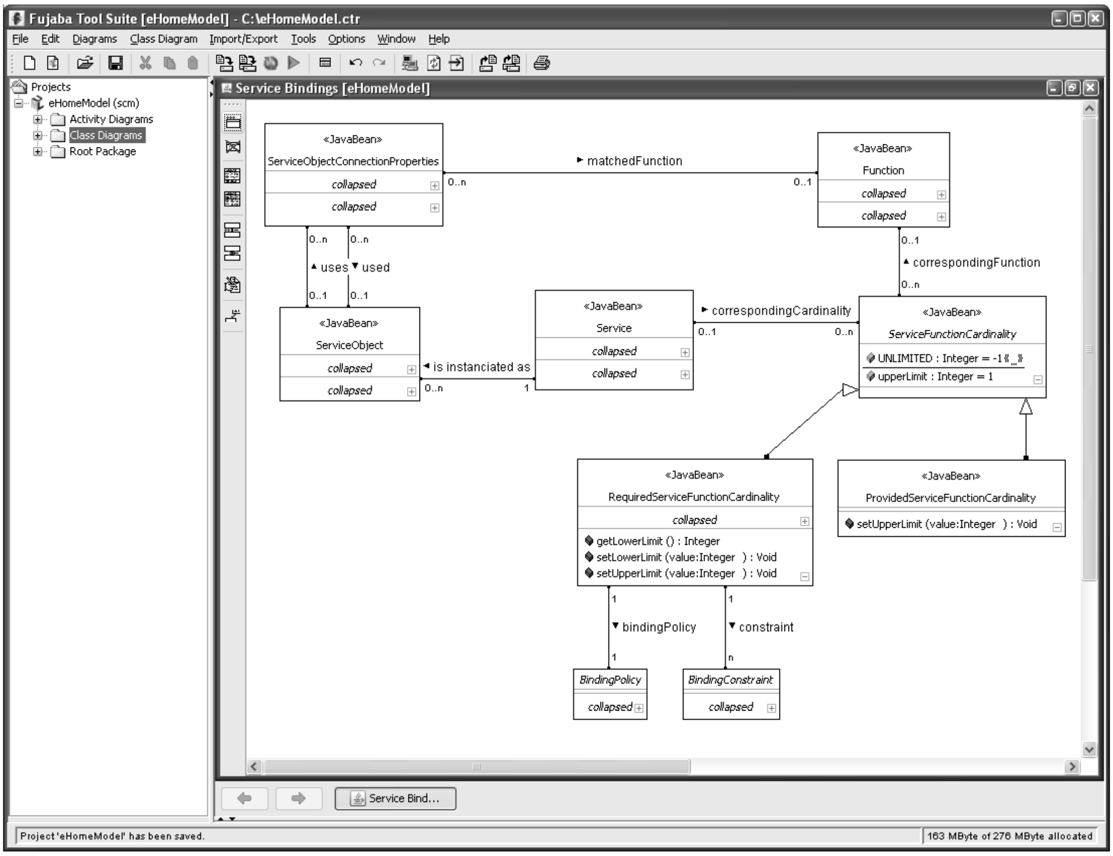


Abbildung 3.5: Die Fujaba-Entwicklungsumgebung.

Zur Spezifikation des Verhaltens der Anwendung, das in PROGRES durch Graphtransformationen modelliert wird, dienen in Fujaba sogenannte Story-Diagramme. Dabei handelt es sich um eine Kombination aus zwei Diagrammtypen der UML, nämlich Aktivitätsdiagrammen und Kollaborationsdiagrammen. Aktivitätsdiagramme stellen den Kontrollfluss eines Systems als Verkettung von Aktionen graphisch dar. Diese werden durch gerichtete Kanten miteinander verbunden. Kollaborationsdiagramme, die seit UML 2.0 Kommunikationsdiagramme heißen, beschreiben ebenfalls das Verhalten einer Anwendung. Dabei werden die ausgetauschten Nachrichten zwischen Kommunikationspartnern spezifiziert.

Die Story-Diagramme in Fujaba stellen Graphersetzungsregeln dar, auch wenn sich die Darstellung in Fujaba von derjenigen in PROGRES unterscheidet. Story-Diagramme modellieren die Methoden der im Klassendiagramm spezifizierten

Klassen. Wird eine Methode aufgerufen, so wird die entsprechende Graphtransformation durchgeführt. Dabei entspricht der Laufzeitgraph der Objektstruktur des Programms während der Ausführung. In Story-Diagrammen werden die linken und rechten Regelseiten kombiniert dargestellt. Durch farbliche Markierungen werden die Veränderungen der Graphtransformation kenntlich gemacht.

Fujaba lässt sich in den unterschiedlichsten Anwendungsgebieten der Softwareentwicklung verwenden. In [KNNZ00] wird Fujaba beispielsweise zur Entwicklung eines Systems zur Produktionssteuerung eingesetzt. Auch im eHome-Projekt am Lehrstuhl für Informatik 3 der RWTH Aachen wurde Fujaba bereits eingesetzt [NSSK05]. Neben PROGRES und Fujaba gibt es noch weitere Sprachen für Graphersetzungssysteme wie z. B. AGG. In [FMRS07] wird ein Vergleich dieser drei verbreiteten Systeme durchgeführt.

In der vorliegenden Arbeit wird Fujaba für die Realisierung der eHome-Laufzeitumgebung verwendet. Fujaba eignet sich besonders für die Zusammenarbeit mit OSGi, aufgrund der Möglichkeit Java-Quellcode aus den Modellen zu generieren. Auch für die Entwicklung graphischer Werkzeuge mit Eclipse – ebenfalls in Java implementiert – bietet sich Fujaba an. Ein Vorteil von Fujaba bei der Verwendung zur modellgetriebenen Softwareentwicklung ist die Möglichkeit, auch das Verhalten der Anwendung bereits auf Modellebene spezifizieren zu können. Die Codegenerierung erzeugt dabei nicht nur ein strukturelles Gerüst, sondern direkt ausführbaren Code. Damit geht der Funktionsumfang über den einfacher UML-Modellierungswerkzeuge weit hinaus. Bereits in den Vorarbeiten zu dieser Arbeit wurde Fujaba eingesetzt, sodass keine nachträgliche Migration erforderlich war. Diese Vorarbeiten werden im folgenden Abschnitt 3.4 näher erläutert.

3.4 Vorarbeiten

Dieser Abschnitt beschreibt Vorarbeiten, die am Lehrstuhl für Informatik 3 im Rahmen des eHome-Projekts durchgeführt wurden und als Grundlage dieser Arbeit dienen. Insbesondere wird hier die Arbeit von NORBISRATH beschrieben, die zu einem ersten Ansatz zur Unterstützung kostengünstiger eHome-Systeme geführt hat [Nor07]. Der bisherige Konfigurierungsansatz stellt eine Grundlage der Konzepte der vorliegenden Arbeit dar. Der Ausgangspunkt des Ansatzes

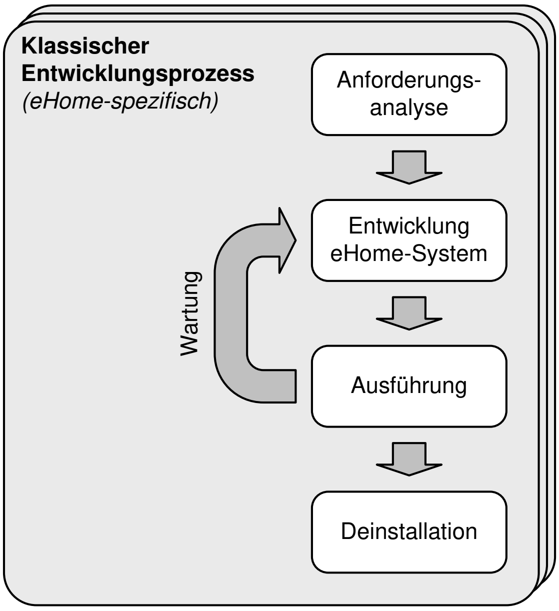


Abbildung 3.6: Klassischer Entwicklungsprozess für eHome-Systeme. (in Anlehnung an [Nor07])

war die bis dahin unbefriedigende Vorgehensweise zur Realisierung von eHome-Systemen, die durch geringe Wiederverwendung und Automatisierbarkeit gekennzeichnet war.

3.4.1 Klassischer Entwicklungsprozess

Bislang müssen eHome-Projekte im wesentlichen für jeden Kunden individuell realisiert werden. Das bedeutet, dass entsprechend den Wünschen der Bewohner ein technisches System in ein spezifisches, meist schon bestehendes Haus integriert wird. Dieser *klassische Entwicklungsprozess* für eHome-Systeme ist in Abbildung 3.6 dargestellt.

In der ersten Phase, der *Anforderungsanalyse*, werden die Kundenwünsche und Rahmenbedingungen erfasst. Darauf folgt die Phase der *Systementwicklung*. Dabei muss zunächst für die gewünschte Funktionalität spezifische Hardware installiert und vernetzt werden. Bestimmte Teile der Funktionalität werden als nächstes in Form von Software realisiert, welche dann speziell auf die gewählte

Hardware abgestimmt wird. Nach Abschluss dieser Phase ist schließlich eine spezifische Lösung im Haus eingerichtet, die sowohl auf Hardware-Ebene als auch auf Software-Ebene mehr oder weniger fest-verdrahtet ist. Das so entwickelte System kommt dann zur *Ausführung*, die gewünschte Funktionalität wird dadurch realisiert.

Eine solche Lösung ist jedoch weder anpassbar noch auf andere eHomes übertragbar. Falls später also Änderungen gewünscht sind oder die Funktionalität auch in einer anderen Umgebung zur Verfügung gestellt werden soll, muss der gesamte Prozess wiederholt werden. Dies impliziert einen hohen Aufwand an Zeit und Kosten, insbesondere auch bei der *Wartung* des Systems, was dazu führt, dass eHome-Systeme bisher keine größere Verbreitung gefunden haben und nur in wenigen Einzelfällen in tatsächlichen Wohnumgebungen zu finden sind. Viele dieser existierenden Realisierungen sind zudem Forschungsprototypen, sodass häufig keine längerfristigen Erfahrungen in der realen Anwendung gewonnen werden können.

3.4.2 Konfigurierungsansatz

Der von NORBISRATH eingeführte Ansatz hat zum Ziel, durch Einsatz und *Wiederverwendung* von *Standardkomponenten* den Aufwand für Entwicklung und Wartung von eHome-Systemen zu reduzieren. Einmal entwickelte Dienste sollen in vielen verschiedenen eHomes einsetzbar sein, ohne dass eine individuelle Anpassung der Implementierung nötig wäre. Der Ansatz basiert auf einer Konfigurierung der bereits vorgefertigten Komponenten, die die eHome-Dienste implementieren.

Durch Einführung der Komponenten wird zunächst einmal die Wiederverwendung besser unterstützt. Dienste werden nicht mehr für ein spezielles eHome entwickelt, sondern als Standardkomponenten. Diese haben einen höheren Grad an Allgemeinheit und lassen sich in verschiedenen eHomes einsetzen. Der Einsatz von Komponenten bietet zudem eine wesentliche Erleichterung bei der Wartung. Standardkomponenten können stetig verbessert und in allen eHomes, die sie nutzen, bei Bedarf aktualisiert werden.

Damit eine Wiederverwendung möglich ist, muss jedoch eine gemeinsame Plattform in allen eHomes verfügbar sein. Um die verschiedenen Standards bezüglich

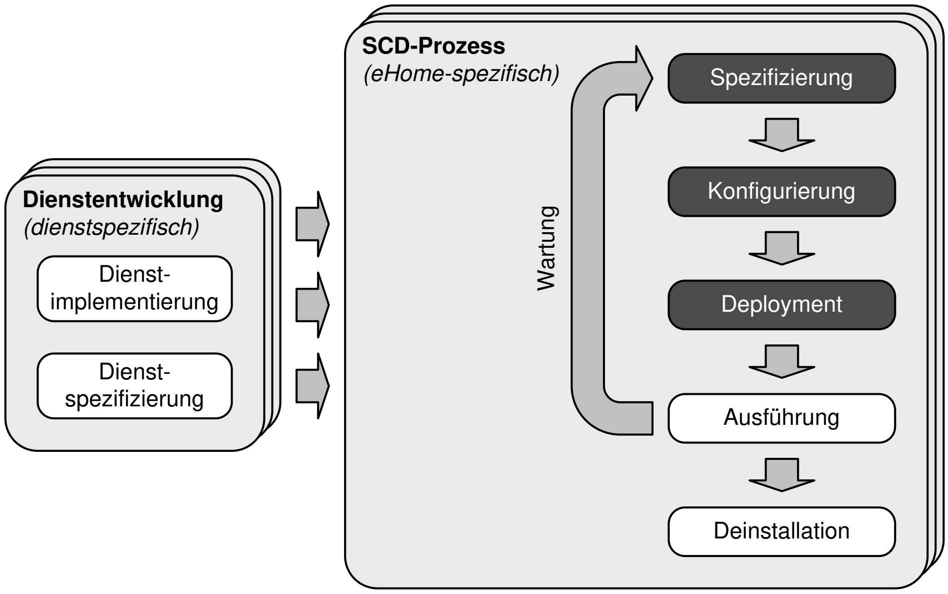


Abbildung 3.7: Entwicklungsprozess für eHome-Systeme nach NORBISRATH mit eingebettetem SCD-Prozess. (in Anlehnung an [Nor07])

der Infrastruktur und der Hardware anbinden zu können, müssen entsprechende Treiberdienste zur Verfügung stehen. Außerdem müssen die verwendeten Komponenten entsprechend ihren Abhängigkeiten miteinander verbunden werden, damit ein funktionsfähiges System entsteht. Dieser Vorgang wird *Konfigurierung* genannt. Da die Auswahl und Zusammenstellung der Dienste eHome-spezifisch ist, muss die Konfigurierung für jedes eHome individuell vorgenommen werden. Auch wenn der Implementierungsaufwand durch Verwendung von vorgefertigten Komponenten nur einmalig anfällt, so bleibt immer noch die ebenfalls aufwendige *manuelle Konfigurierung*.

Der Ansatz von NORBISRATH zielt darauf ab, auch den Konfigurierungsaufwand durch geeignete Werkzeugunterstützung zu reduzieren. Die manuelle Konfigurierung soll durch eine sogenannte *generative Konfigurierung* ersetzt werden. Bei der generativen Konfigurierung wird die Konfiguration für ein individuelles eHome automatisch aus einer Spezifikation erzeugt. Dazu wurde der sogenannte SCD-Prozess eingeführt, der die generative Konfigurierung unterstützt. Die nötige Spezifikation wird mit einem dafür vorgesehenen Softwarewerkzeug, dem *eHomeConfigurator*, erstellt. Dieser implementiert den SCD-Prozess und ermöglicht damit die generative Konfigurierung.

Der *Entwicklungsprozess nach* NORBISRATH basiert auf der Nutzung vorgefertigter Komponenten, welche die eHome-Dienste implementieren aber unabhängig vom spezifischen eHome entwickelt werden können. Auf Basis einer Dienstspezifikation wird im sogenannten *SCD-Prozess* eine gültige Komposition von Diensten erstellt, die dann zur Ausführung gebracht wird. Der gesamte Ablauf wird durch ein Werkzeug, den eHomeConfigurator, unterstützt und kann teilweise automatisch ausgeführt werden. In Abbildung 3.7 ist der Entwicklungsprozess dargestellt, der den SCD-Prozess beinhaltet. Im Unterschied zum klassischen Entwicklungsprozess ist hier eine strikte Trennung zwischen der Dienstentwicklung und dem eHome-spezifischen Teil des Entwicklungsprozesses gegeben, woraus sich eine Reduzierung des Entwicklungsaufwands ergibt.

Dienstentwicklung

Auf der linken Seite der Abbildung 3.7 ist die Phase der Dienstentwicklung dargestellt, welche zum einen die Dienstimplementierung und zum anderen die Dienstspezifizierung umfasst. Dieser Teil des Entwicklungsprozesses ist dienstspezifisch, d. h. er wird für jeden zu entwickelnden Dienst individuell ausgeführt. Durch die Unabhängigkeit vom eHome-spezifischen Teil des Prozesses müssen einmal entwickelte Dienste nicht weiter modifiziert werden, wenn sie in einem spezifischen eHome zum Einsatz kommen sollen. Die Entwicklung eines Dienstes muss daher nur einmal für alle eHomes stattfinden.

Die *Dienstimplementierung* umfasst alle Aktivitäten der Implementierung einer Komponente, die einen eHome-Dienst zur Verfügung stellt, insbesondere das Erstellen des eigentlichen Quellcodes. Die Komponenten müssen eine vorgegebene Schnittstelle implementieren, um später durch den eHomeConfigurator konfiguriert werden zu können. Darüber hinaus müssen sie im Hinblick auf die spätere Dienstkomposition entwickelt werden, es ist also wichtig, dass sie möglichst kompatibel zu anderen Diensten sind und mit diesen interagieren können.

Die *Dienstspezifizierung* ist erforderlich, damit später eine automatische Komposition der Dienste durchgeführt werden kann. Die Spezifikation gibt an, welche Funktionalitäten ein Dienst anbietet und welche er benötigt. Die Funktionalitäten, die zur Spezifikation zur Verfügung stehen, müssen dazu im Voraus definiert werden. Aus der Dienstspezifikation ergeben sich Abhängigkeiten zwischen den

Diensten. Diese werden in der Konfigurierungsphase des im Folgenden beschriebenen SCD-Prozesses aufgelöst und es wird eine ausführbare Konfiguration des eHome-Systems erstellt.

SCD-Prozess

Im rechten Teil der Abbildung 3.7 ist der eHome-spezifische Teil des Entwicklungsprozesses dargestellt, der für jedes eHome individuell durchgeführt wird. Wie oben beschrieben, ist dieser Teil unabhängig von der Dienstentwicklung, sodass hier kein weiterer Implementierungsaufwand bezüglich der Dienste anfällt. Der eHome-spezifische Teil wird durch den *SCD-Prozess* realisiert. Das Akronym *SCD* steht dabei für die englischen Begriffe ***S**pecification*, ***C**onfiguration* und ***D**eployment* und beschreibt die drei Phasen des Prozesses:

1. *Spezifizierung* der eHome-Umgebung und der Dienstauswahl
2. *Konfigurierung* auf Basis der Spezifikation
3. *Deployment* der Konfiguration auf dem Service Gateway des eHomes

Spezifizierung: In der ersten Phase, der *Spezifizierung*, werden die eHome-spezifischen Anforderungen erfasst. Dies sind zum einen physikalische Rahmenbedingungen wie der Grundriss der eHome-Umgebung. Damit werden die unveränderlichen Gegebenheiten der Gebäudearchitektur erfasst, z. B. Räume, Türen oder Fenster in der Umgebung. Auf diese Weise ergibt sich eine abstrakte Sicht auf den räumlichen Kontext des eHomes, der in den späteren Phasen von Bedeutung ist. Zum anderen werden in der Spezifizierungsphase auch die Wünsche und Bedürfnisse der Benutzer modelliert, was durch eine Auswahl geeigneter eHome-Dienste und deren Parametrisierung erfolgt. Hierzu trifft der Benutzer eine Auswahl aus den zur Verfügung stehenden Diensten und wählt aus, in welchen Räumen deren Funktionalität gewünscht wird. Anschließend können falls erforderlich die Parameter der gewählten Dienste angepasst werden. Schließlich werden auch die bereits vorhandenen Geräte im eHome erfasst und den jeweiligen Räumen, in denen sie sich befinden, zugeordnet. Diese Informationen ergeben zusammengenommen eine Anforderungsspezifikation für das konkrete eHome-System.

Konfigurierung: In der zweiten Phase findet die *Konfigurierung* des eHome-Systems statt. Die Konfigurierung erfolgt größtenteils automatisch und basiert auf der Spezifikation, die in der ersten Phase erstellt wurde. Die Aufgabe der Konfigurierungsphase ist es, eine Systemkonfiguration zu erstellen, die die Ausführung der Dienste ermöglicht. Dazu müssen vor allem die Abhängigkeiten der Dienste berücksichtigt werden. Die vom Benutzer ausgewählten Dienste setzen bestimmte Funktionalitäten voraus, die von anderen Diensten zur Verfügung gestellt werden müssen. Um diese Abhängigkeiten zu erfüllen, müssen also für alle benötigten Funktionalitäten passende Dienste gefunden werden, die diese Funktionalitäten anbieten. Es wird eine *Dienstkomposition* gesucht, die alle Abhängigkeiten der Dienste erfüllt und auf Treiberdiensten basiert, zu denen passende Geräte in der Umgebung vorhanden sind.

Die Informationen zu den verfügbaren Geräten werden der oben beschrieben Spezifikation der Umgebung entnommen. Falls es nicht möglich ist, geeignete Geräte für eine Dienstkomposition zu finden, werden sogenannte *virtuelle Geräte* erstellt. Diese dienen als Platzhalter, die später, vor der Inbetriebnahme des eHomes, durch reale Geräte ausgefüllt werden müssen. Dazu kann dem Benutzer eine Liste der zusätzlich benötigten Geräte ausgegeben werden. Sind alle Dienste entsprechend ihrer spezifizierten Abhängigkeiten miteinander verbunden, so ergibt sich eine gültige Konfiguration. Diese beschreibt, wie die Dienste miteinander verbunden werden, welchen Räumlichkeiten sie zugeordnet sind und welche Geräte der Umgebung durch welche Treiberdienste gesteuert werden. Auf Basis der Konfiguration können die Dienste dann in der nächsten Phase auf dem Service Gateway des eHomes zur Ausführung gebracht werden.

Deployment: In der letzten Phase des SCD-Prozesses, dem *Deployment*, werden die Dienste zur Ausführung gebracht. Entsprechend der Konfiguration werden die Dienste auf dem Service Gateway des eHomes installiert und gestartet. Aufgrund der Abhängigkeiten zwischen den Diensten muss dies in einer bestimmten Reihenfolge geschehen. Diese kann automatisch aus den Abhängigkeiten abgeleitet werden. Damit ist die Anpassung der vorgefertigten Komponenten an das spezifische eHome abgeschlossen und das eHome-System befindet sich in der Ausführung.

3.4.3 Dienstkonzept

Ein eHome-Dienst wird durch eine Komponente realisiert, die diesen Dienst zur Verfügung stellt. Um die gewünschten Funktionalitäten zu liefern, greift der Dienst in der Regel auf Funktionalitäten anderer Dienste zurück. Es werden also weitere Komponenten benötigt, die diese anderen Dienste realisieren. Auf diese Weise bestehen *Abhängigkeiten* zwischen den Komponenten, die beim Erstellen der Komposition, die das Gesamtsystem bildet, berücksichtigt werden müssen. Da in dem hier diskutierten Ansatz jede Komponente genau einen Dienst realisiert, werden beide Begriffe häufig synonym gebraucht. Daher wird analog auch von Abhängigkeiten zwischen Diensten gesprochen. Abhängigkeiten werden durch *Bindungen* aufgelöst, die beim Deployment zwischen den Diensten angelegt werden. So erhält ein Dienst die Möglichkeit auf einen anderen Dienst zuzugreifen. Sind alle Abhängigkeiten eines Dienstes erfüllt, so kann dieser gestartet und ausgeführt werden.

Diensttypen

Das Abstraktionsniveau eines Dienstes hat Einfluss auf seine Abhängigkeiten. Dadurch ergeben sich drei verschiedene Typen von Diensten.

Basisdienste: Basisdienste haben das niedrigste Abstraktionsniveau. Im Allgemeinen handelt es sich bei Basisdiensten um Treiber, die den Zugriff auf einen bestimmten Typ von Geräten erlauben. Ein Basisdienst fungiert als Treiber indem er die Funktionalität des Geräts für andere Dienste zur Verfügung stellt und somit im eHome-System nutzbar macht. Ein Basisdienst benötigt keine weiteren Dienste, um arbeiten zu können, da er auf die in Hardware realisierte Gerätefunktionalität zurückgreift. Basisdienste bieten somit nur Funktionalitäten an, benötigen selbst jedoch keine. Basisdienste sind z. B. ein Lampensteuerungsdienst, ein Rollladensteuerungsdienst oder ein Bewegungsmelderdienst.

Integrierende Dienste: Integrierende Dienste haben ein höheres Abstraktionsniveau als Basisdienste. Sie nutzen die Funktionalitäten, die von Basisdiensten angeboten werden, und integrieren diese zu neuen Funktionalitäten auf einem höheren Abstraktionsniveau. Diese abstrakteren Funktionalitäten können dann wiederum von weiteren Diensten genutzt werden. Integrierende Dienste zeichnen sich dadurch aus, dass sie sowohl Funktionalitäten von anderen Diensten benötigen als auch selbst Funktionalitäten anbieten. Ein Beispiel für einen integrierenden Dienst ist ein Dienst, der die Beleuchtung eines Raums in Abhängigkeit vom gegebenen Tageslicht entweder durch künstliche Beleuchtung mittels Lampen oder durch Steuerung der Rollladen reguliert.

Top-Level-Dienste: Top-Level-Dienste stellen das höchste Abstraktionsniveau im eHome-System dar. Sie stellen die von ihnen realisierten Funktionalitäten direkt den Nutzern des eHomes zur Verfügung. Der Nutzer wählt Top-Level-Dienste entsprechend den von ihm gewünschten Funktionalitäten aus. Top-Level-Dienste bieten daher keine Funktionalitäten an andere Dienste an, sondern ausschließlich den eHome-Nutzern. Sie selbst benötigen jedoch im Allgemeinen die Funktionalitäten von integrierenden Diensten oder Basisdiensten. Ein personenbezogener Beleuchtungsdienst, wie in Abschnitt 2.4.1 beschrieben, ist z. B. ein Top-Level-Dienst. Dieser steuert die Beleuchtung der Umgebung eines Nutzers in Abhängigkeit von seinen Vorlieben, seinem derzeitigen Aufenthaltsort und den dort verfügbaren Ressourcen. Der Dienst kann z. B. auf den oben beschriebenen integrierenden Dienst zurückgreifen und damit verschiedene Möglichkeiten der Beleuchtung ausnutzen.

Dienstkomposition

Abbildung 3.8 zeigt ein Beispiel zur Dienstkomposition und den sich aus den Diensttypen ergebenden Schichten. Auf der linken Seite der Abbildung sind die verfügbaren *Dienste* dargestellt, die jeweils durch eine entsprechende Dienstspezifikation repräsentiert werden. Die graphische Notation in der Abbildung stellt die Dienste und ihre Abhängigkeiten dar.

Ganz oben in der Abbildung ist ein Top-Level-Dienst Visual Alarm zu sehen, der im Fall eines Einbruchs in das eHome visuelle Signale zur Abschreckung und Warnung erzeugen soll. Dazu benötigt er die Funktionalität illuminate, da er auf

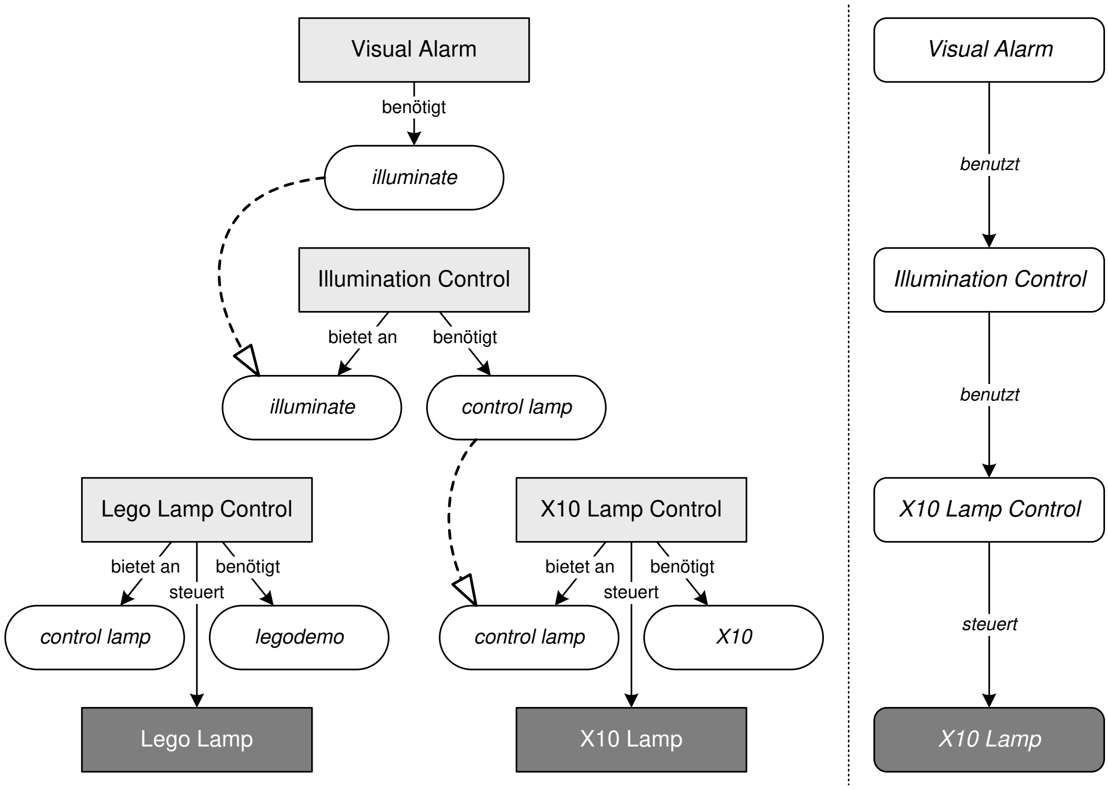


Abbildung 3.8: Schichtenarchitektur der eHome-Dienste.

die Beleuchtung in der Umgebung zugreifen muss. Ein integrierender Dienst Illumination Control stellt diese Funktionalität zur Verfügung und benötigt dazu seinerseits die Funktionalität control lamp zum Steuern der Lampen in der Umgebung. Für diese Funktionalität stehen zwei Basisdienste zur Verfügung. Für Lampen auf Grundlage des X10-Standards wird der Treiber X10 Lamp Control verwendet. Für die Lampen eines Lego-Demonstrators (siehe Abschnitt 6.4) wird der Treiber Lego Lamp Control verwendet. Auf Basis der verfügbaren Dienste kann nun eine Dienstkomposition erstellt werden.

Auf der rechten Seite der Abbildung sind die *Dienstobjekte* und ihre Bindungen in einer spezifischen Konfiguration zu sehen. Dienstobjekte repräsentieren Instanzen von Diensten, die einem bestimmten Raum in der Umgebung zugeordnet werden. Jeder Dienst stellt einen Typ dar und wird durch eine eindeutig zuzuordnende Spezifikation definiert. Zu jedem Dienst können beliebig viele Dienstobjekte erstellt werden.

Das Dienstobjekt Visual Alarm auf der rechten Seite ist somit eine Instanz des zugehörigen Dienstes Visual Alarm auf der linken Seite. Es benutzt entsprechend der Spezifikation weitere Dienstobjekte, in diesem Fall Illumination Control, welches wiederum X10 Lamp Control benutzt. Über diese Dienstkomposition wird das Gerät X10 Lamp in der eHome-Umgebung angesteuert.

Die Kommunikation zwischen den Diensten erfolgt zur Laufzeit über sogenannte *States*, die die Laufzeitzustände der Dienstobjekte repräsentieren. Die Implementierungen der Dienste, die über eine bestimmte Funktionalität miteinander verbunden sind, können die States lesen und schreiben. Auf diese Weise ist eine Kommunikation möglich. Damit dies funktioniert, müssen die ausgetauschten Daten möglichst einfach gehalten werden. Außerdem müssen die beteiligten Dienste entsprechend den verwendeten States entwickelt werden. Dazu muss der Entwickler die Datentypen der States kennen und wissen, wann der State initialisiert wird und wann ein Lese- oder Schreibzugriff möglich ist.

Damit Dienste automatisch komponiert werden können, wird wie oben beschrieben zunächst eine Dienstspezifikation benötigt. Diese beschreibt welche Funktionalitäten der Dienst anbietet und welche er benötigt. Bei benötigten Funktionalitäten wird darüber hinaus noch zwischen zwingend benötigten und optional benötigten Funktionalitäten unterschieden. Außerdem beinhaltet die Spezifikation der Dienste Kardinalitätsinformationen, welche festlegen, wie oft eine Funktionalität angeboten bzw. benötigt wird.

Jede Funktionalität hat eine bestimmte Semantik. Zwei Dienste passen zusammen, wenn angebotene und benötigte Funktionalitäten übereinstimmen. Wenn damit auch eine syntaktische Kompatibilität einhergeht, können die Dienste miteinander kommunizieren. Im Ansatz von NORBISRATH wird die Semantik einer Funktionalität durch ein sogenanntes *Semantic Label* repräsentiert. Ein solches Semantic Label wird durch einen *Namen*, der die Funktionalität beschreibt, identifiziert. Gleiche Semantic Labels bedeuten somit eine kompatible Funktionalität. Dabei wird implizit vorausgesetzt, dass Dienste, die sich auf ein bestimmtes Semantic Label beziehen, auch interoperabel sind. Sie müssen daher auch syntaktisch kompatibel sein und über gemeinsame States (siehe oben) kommunizieren. Es ist den Dienstentwicklern überlassen, dafür Sorge zu tragen, dass die Kommunikation über States in den beteiligten Dienstimplementierungen tatsächlich korrekt umgesetzt wird.

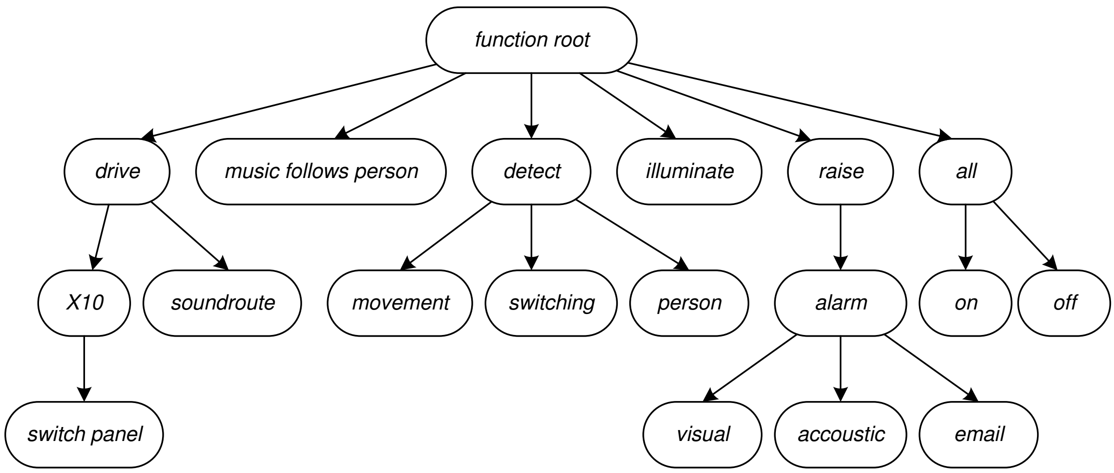


Abbildung 3.9: Hierarchie von Dienstfunktionalitäten. (in Anlehnung an [Nor07])

Funktionalitäten

Um die Dienstspezifikation auswerten zu können, ist ein übergreifender Standard erforderlich, in dem alle zur Verfügung stehenden Funktionalitäten als Semantic Labels definiert sind. Beim Spezifizieren von Diensten kann dann auf diesen Standard zurückgegriffen werden. NORBISRATH schlägt eine hierarchische Strukturierung der Semantic Labels vor. Ein Ausschnitt aus der Funktionalitätenhierarchie in [Nor07] ist in Abbildung 3.9 dargestellt. Ausgehend vom Wurzelknoten function root sind auf der obersten Ebene Funktionalitäten wie music follows person, detect oder illuminate definiert. Die Funktionalität detect steht für das Erkennen einer Entität oder eines Zustands. Sie ist weiter verfeinert in die Funktionalitäten detect.switching, detect.movement und detect.person. Diese schränken die Funktionalität detect auf eine spezifischere Semantik ein. So steht detect.movement für das Erkennen einer Bewegung während detect.person für das Erkennen einer Person steht.

Die Semantic Labels werden entsprechend ihrem Pfad in der Hierarchie ausgehend von der Wurzel benannt. In der Abbildung sind die Bezeichner aus Platzgründen abgekürzt dargestellt. Die hierarchische Strukturierung dient hier im Wesentlichen der Übersichtlichkeit, für das Matching der Dienste wird die Hierarchiebeziehung jedoch nicht genutzt. Ein Match ergibt sich daher nur, wenn Semantic Labels exakt übereinstimmen. Nur dann wird davon ausgegangen, dass

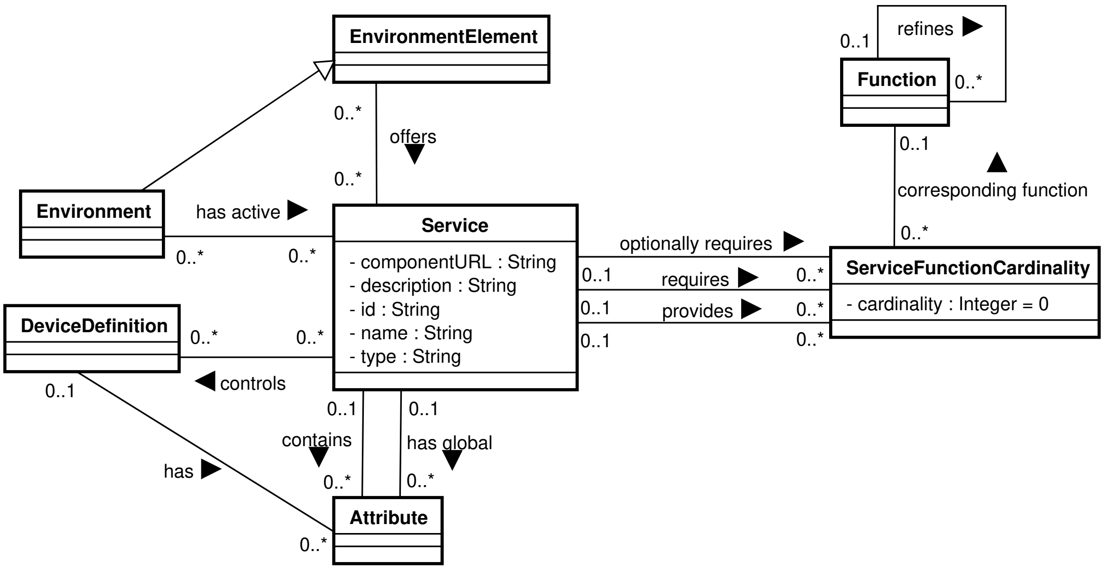


Abbildung 3.10: Ausschnitt des eHome-Modells nach NORBISRATH.

die Dienste kompatibel sind und verbunden werden können. Das Matching unter Verwendung der Semantic Labels ist auf Basis eines eHome-spezifischen Modells realisiert. Die Modellierung der Anwendungsdomäne wird im folgenden Abschnitt näher beschrieben.

3.4.4 Modellierungsansatz

Zur Unterstützung des oben beschriebenen SCD-Prozesses wurde das *eHome-Modell* [NARS06, Nor07] entwickelt. Der Modellierungsansatz basiert auf einer globalen Sicht, die alle für den SCD-Prozess relevanten Daten in einem einheitlichen Modell zusammenfasst. Das eHome-Modell dient insbesondere als Basis für den eHomeConfigurator, der in einem modellgetriebenen Softwareentwicklungsprozess erstellt wurde. Dazu wurde das Werkzeug *Fujaba* eingesetzt (vgl. Abschnitt 3.3.2). Durch das eHome-Modell werden sowohl statische als auch dynamische Festlegungen getroffen.

Die statische Modellierung legt fest, aus welchen Entitäten die Repräsentation des Laufzeitzustands im eHome-System besteht und in welchen Beziehungen diese zueinander stehen können. Die Umsetzung erfolgt durch UML-Klassendiagramme in Fujaba. Die dort definierten Klassen repräsentieren sowohl Objekte der physikalischen Umgebung als auch immaterielle Entitäten, wie beispielsweise die Funktionalitäten eines Dienstes.

In Abbildung 3.10 wird ein Ausschnitt des eHome-Modells dargestellt. In der Mitte ist als zentrales Element die Klasse Service zu sehen, die einen Dienst repräsentiert. Jeder Dienst hat bestimmte Eigenschaften, eine ID, einen Namen, einen Typ, eine Beschreibung und eine URL, die auf eine Datei mit der Implementierung des Dienstes verweist. Außerdem kann ein Dienst mit globalen und lokalen Attributen versehen werden, die individuell definiert werden können. Zur Dienstspezifikation gehört aber auch die Festlegung angebotener und benötigter Funktionalitäten des Dienstes. Dies wird im Modell durch die Verbindung von Service zur Klasse Function repräsentiert. Die Verbindung erfolgt indirekt über die Klasse ServiceFunctionCardinality, die weitere Details festgelegt, wie etwa die Kardinalität der Funktionalität. Durch die unterschiedlichen Kantentypen provides, requires und optionally requires wird zwischen angebotenen, benötigten und optional benötigten Funktionalitäten unterschieden. Die Funktionalitäten selbst können hierarchisch strukturiert werden, was durch die refines-Kante modelliert wird. Die räumliche Zuordnung von Diensten zu Umgebungselementen erfolgt durch die Verbindung zu Environment und EnvironmentElement. Schließlich gibt es im dargestellten Teil des Modells noch eine Klasse DeviceDefinition, mit der Gerätetypen modelliert werden. Eine Verbindung zwischen Service und Device-Defintion bedeutet, dass ein Dienst einen bestimmten Typ von Geräten steuern kann. Der statische Modellanteil besteht aus weiteren Elementen, die der Einfachheit halber an dieser Stelle nicht näher beschrieben werden. Weitere Details können [NARS06] und [Nor07] entnommen werden.

Neben dem statischen Modellanteil wird in Fujaba auch ein dynamischer Modellanteil spezifiziert. Das bedeutet, dass auch das Verhalten des eHomeConfigurators mit Hilfe von Fujaba modelliert werden kann. Auch dieser Teil kann durch Codegenerierung mit Fujaba automatisch in lauffähigen Java-Quellcode übersetzt werden. Zur dynamischen Modellierung werden die Methoden der Klassen aus dem statischen Modellanteil in Form von Story-Diagrammen spezifiziert (siehe auch Abschnitt 3.3.2). Auf die Details der dynamischen Modellierung wird hier nicht näher eingegangen, es sei erneut auf [Nor07] verwiesen.

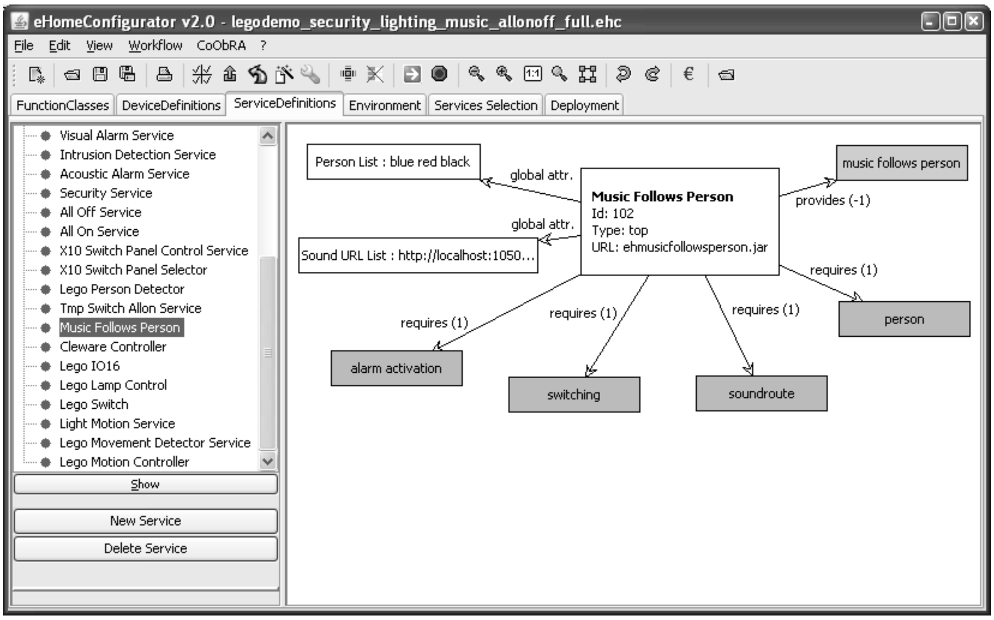


Abbildung 3.11: Dienstspezifikation im eHomeConfigurator.

3.4.5 Werkzeugunterstützung

Auf Basis des eHome-Modells wurde der *eHomeConfigurator* entwickelt [NMA06, Nor07], ein Werkzeug zur Unterstützung des SCD-Prozesses. Der eHomeConfigurator deckt neben den Phasen des eigentlichen SCD-Prozesses auch die Dienstspezifikation ab. Die Erfassung und Verarbeitung aller Informationen erfolgt somit über ein einziges integriertes Werkzeug.

Abbildung 3.11 zeigt wie mit Hilfe des eHomeConfigurators eine Dienstspezifikation erstellt wird. Am oberen Rand des Fensters befinden sich unterhalb einer Menüleiste und einer Werkzeugleiste verschiedene Reiter, für die Editoren. In der dargestellten Ansicht ist der Editor zum Erstellen von Dienstspezifikationen geöffnet. Auf der linken Seite können verschiedene bereits erstellte Dienstspezifikationen aus einer Liste ausgewählt werden, des Weiteren können Spezifikationen auch neu erstellt bzw. wieder gelöscht werden. Auf der rechten Seite ist die Spezifikation des Dienstes *Music Follows Person* zu sehen, der dazu dient, den Bewohnern des eHomes in der jeweils aktuellen Umgebung ihre präferierte Musik abzuspielen. Der Dienst bietet bestimmte Funktionalitäten an (hier music follows person) und benötigt andere Funktionalitäten, um ausgeführt werden zu

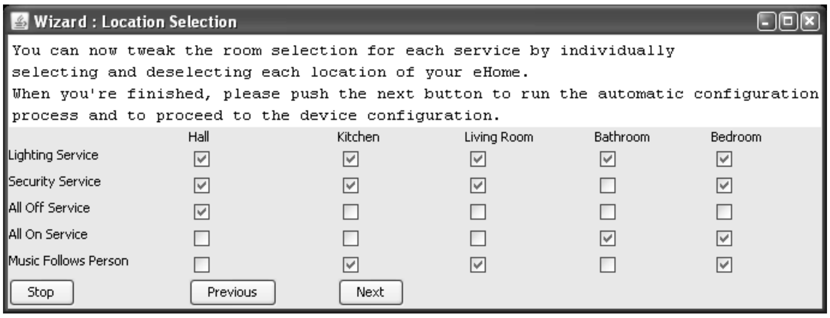


Abbildung 3.12: Zuordnung von Diensten und Räumen im eHomeConfigurator.

können (hier z. B. soundroute). Diese Abhängigkeiten werden ebenfalls mit Hilfe des Editors spezifiziert und im Hauptfenster des Editors graphisch dargestellt. Zu jeder Dienstspezifikation muss außerdem eine Implementierung in Form eines OSGi-Bundles vorliegen. Ein Verweis auf die jeweilige Implementierung wird in der Dienstspezifikation festgelegt, um den Dienst später starten zu können.

Im nächsten Schritt kann die Auswahl und Anpassung von Diensten an ein konkretes eHome erfolgen. Dazu wird zunächst die Umgebung in Form eines Grundrisses mit den unterschiedlichen Räumen des eHomes spezifiziert. Dann kann eine Auswahl von Diensten getroffen werden, die im eHome zum Einsatz kommen sollen. Ist dies geschehen, so muss eine Zuordnung von Diensten zu den Räumen erfolgen. Dieser Vorgang wird durch den in Abbildung 3.12 dargestellten Dialog unterstützt. Hier wird festgelegt, in welchen Räumen der eHome-Umgebung welche Dienste gestartet werden sollen.

3.4.6 Bewertung

Der bisherige Konfigurierungsansatz zielt darauf ab, die Entwicklung von eHome-Systemen zu vereinfachen und Entwicklungskosten zu reduzieren. Dadurch soll die Nutzung von eHomes für eine größere Anzahl von Nutzern attraktiv werden. Der Ansatz erhöht die Wiederverwendbarkeit und die Automatisierbarkeit durch den Einsatz von vorgefertigten Softwarekomponenten und der Spezifikation von Dienstfunktionalitäten. Durch die Entkopplung der Dienstentwicklung von den individuellen Anforderungen eines spezifischen eHomes wird bereits eine Verbesserung der Situation erreicht. Dennoch hat der bisherige Ansatz an verschiedenen Punkten Potential für Verbesserungen, die für die Entwicklung zukünftiger eHome-Systeme von großer Bedeutung sind. Die vorliegende Arbeit setzt an diesen Punkten an und beschreibt neue Konzepte, die den ursprünglichen Ansatz verbessern und in wesentlichen Punkten erweitern. Im Folgenden werden die verschiedenen Bereiche, an denen die Verbesserungen ansetzen, näher erläutert.

Statischer Prozess

Der SCD-Prozess, der die Konfigurierungsphase beinhaltet, wurde im bisherigen Ansatz als Bestandteil des Entwicklungsprozesses des eHome-Systems betrachtet. Da die Entwicklung zur Ausführungszeit als abgeschlossen gilt, sind danach keine weiteren Anpassungen bezüglich der Konfiguration möglich. Gerade ein eHome kann jedoch nicht als statische Systemumgebung betrachtet werden. Eine Anpassung ist im bisherigen Ansatz jedoch nur auf dem Weg von Wartungsmaßnahmen möglich (vgl. Abbildung 3.7). Dazu muss aber das System zunächst angehalten werden. Erst wenn von Hand alle gewünschten Änderungen durchgeführt wurden und eine angepasste Spezifikation erstellt wurde, kann der SCD-Prozess erneut ausgeführt und das eHome-System wieder gestartet werden.

Die Bewohner eines eHomes werden auch im laufenden Betrieb Dienste hinzufügen oder bereits laufende Dienste deaktivieren wollen. Außerdem können sich die Wünsche der Benutzer im Laufe der Zeit ändern. Insbesondere wenn mehrere Nutzer eine Umgebung gemeinsam bewohnen, werden des öfteren Fälle auftreten, in denen ein Dienst nachinstalliert oder ein nicht mehr benötigter Dienst vom Service Gateway des eHomes entfernt werden soll. In solchen Fällen sollte ein Neustart des Systems und eine Wiederholung der manuellen Schritte im SCD-Prozess vermieden werden.

Bezüglich der von den eHome-Diensten direkt oder indirekt genutzten Hardware-Ressourcen können sich ebenfalls im laufenden Betrieb Änderungen ergeben. Ein neues Gerät kann in das eHome integriert werden, ebenso kann ein bereits vorhandenes Gerät aus dem eHome entfernt werden, z. B. weil ein aktuelleres Gerät mit besseren Funktionen verfügbar ist. Des Weiteren kann ein Gerät auch aufgrund eines Hardwarefehlers ausfallen und würde somit nicht länger für das eHome-System genutzt werden können. Auch in diesen Fällen ist eine Anpassung der Konfiguration nötig, die eine weitergehende automatische Unterstützung durch das eHome-System erforderlich macht.

In Bezug auf die Geräte im eHome gibt es weitere Fälle, die eine Anpassung erfordern. Die Nutzung mobiler Geräte, wie z. B. von Mobiltelefonen, PDAs oder MP3-Playern, hat in der Vergangenheit mehr und mehr zugenommen. Mobile Geräte gehören heute zum Alltag, daher werden auch in eHomes mobile Geräte genutzt werden. Da diese naturgemäß mitgeführt werden und damit ihren Ort regelmäßig wechseln, muss das eHome-System in der Lage sein, diese Veränderungen zu berücksichtigen. In Bezug auf den SCD-Prozess bedeutet dies, dass sich die Zuordnung von Geräten zu Räumen laufend ändern kann. Es sind daher neue Mechanismen erforderlich, damit Dienste auch von mobilen Geräten Gebrauch machen können. Die Anpassung zur Laufzeit muss unterstützt werden, um der *Mobilität* und *Dynamik* in eHomes Rechnung zu tragen.

Unterstützung von Querschnittsfunktionalitäten

Neben mobilen Geräten bewegen sich auch die Benutzer innerhalb des eHomes. Diese *Mobilität* spielt eine wichtige Rolle bei der Betrachtung personalisierter Dienste, die von den Bewohnern genutzt werden. In den meisten Fällen wird von solchen Diensten erwartet, dass sie ihre Funktionalität in der unmittelbaren Umgebung des Nutzers zur Verfügung stellen. Im bisherigen Ansatz wird eine statische Zuordnung zwischen Diensten und Räumen hergestellt. Diese Zuordnung kann zur Laufzeit des Systems nicht mehr geändert werden.

Personenbezogene Dienste werden im bisherigen Ansatz dadurch umgesetzt, dass ein Dienst die Verwaltung von Personen und der Personalisierung individuell realisiert. Dazu muss der Dienst mittels geeigneter Basisdienste Personen erkennen, die gewonnenen Informationen verarbeiten und dazu nutzen, eine personalisierte Funktionalität in der Umgebung bereitzustellen. Dafür benötigt der Dienst Informationen über die persönlichen Wünsche aller Personen, für die eine personalisierte Funktionalität angeboten werden soll.

Darüber hinaus muss der Dienstentwickler einen Mechanismus implementieren, der Konflikte auflöst, die entstehen, wenn mehrere Personen gleichzeitig in der Umgebung des Dienstes anwesend sind. In dem Fall können üblicherweise nicht die Wünsche aller Personen berücksichtigt werden. Dienste können z. B. so implementiert werden, dass immer die Person, die zuletzt in die Umgebung eingetreten ist, berücksichtigt wird. Ebenso könnte auch eine Strategie implementiert werden, die immer der zuerst in die Umgebung eingetretenen Person Vorrang gewährt. Da für jeden Dienst eine eigene Strategie entwickelt wird, kann es leicht zu inkonsistentem Verhalten bei mehreren Diensten kommen. Der Benutzer hat daher keinen Einfluss auf die Umsetzung der Personalisierung in seiner räumlichen Umgebung.

Die wiederholte Implementierung von Mechanismen zur Personalisierung führt zu einem hohen Entwicklungsaufwand solcher Dienste. Da der verfolgte Ansatz darauf abzielt, den Entwicklungsprozess zu vereinfachen und ein hohes Maß an Wiederverwendung zu erreichen, sollte in diesem Punkt eine bessere Unterstützung angestrebt werden. Daher sollten Querschnittsfunktionalitäten, wie die Berücksichtigung von Kontextinformationen und die personenbezogene Verwaltung von Diensten, durch die Laufzeitumgebung des eHome-Systems zur Verfügung gestellt werden. Dann kann eine wiederholte Implementierung diese Querschnittsfunktionalitäten vermieden werden.

Semantische Konzepte

Zur semantischen Beschreibung der eHome-Dienste und der darauf basierenden Dienstkomposition wurden im bisherigen Ansatz bereits erste Konzepte eingeführt. Wie oben erläutert, werden Dienste mit einer Spezifikation versehen, in der die angebotenen und benötigten Funktionalitäten durch Semantic Labels beschrieben sind. Die semantische Beschreibung besteht dabei aus einem Namen, der zur Identifikation der Funktionalität dient.

Das Auffinden passender Dienste zur Komposition geschieht durch Vergleich der Semantic Labels. Es wird davon ausgegangen, dass Dienste nach erfolgter Komposition auch unmittelbar miteinander kommunizieren können. Dazu muss einem Dienst zunächst bekannt sein, mit welchen anderen Diensten er entsprechend der Konfiguration kommunizieren kann. In der Dienstimplementierung muss dafür auf das oben beschriebene eHome-Modell zugegriffen werden, welches der Repräsentation des globalen Zustands des eHome-Systems dient. Dazu muss jedoch jeder Dienstentwickler den genauen Aufbau des eHome-Modells kennen und mit dessen Nutzung vertraut sein. Bei einer unsachgemäßen Verwendung des eHome-Modells können leicht Fehler entstehen. Es ist also eine entsprechende Einarbeitungszeit nötig und der Aufwand der Dienstentwicklung erhöht sich. Das sollte im Sinne der kostengünstigen Entwicklung von eHome-Systemen vermieden werden.

Nachdem die Kommunikationspartner eines Dienstes über das eHome-Modell festgestellt wurden, muss die eigentliche Kommunikation ermöglicht werden. Der bisherige Ansatz sieht vor, dass die Dienstkommunikation ebenfalls über das eHome-Modell mittels der oben erwähnten States abläuft. Bei den States handelt es sich um Objekte, die Zustände von Diensten repräsentieren und die sowohl gelesen als auch geschrieben werden können. Bisher ist nicht genau festgelegt, wann ein Dienst lesen oder schreiben darf und wie der jeweils andere Dienst darauf reagieren soll. Der Typ und die Semantik der Informationen, die in einem State gespeichert werden, müssen daher informell vereinbart werden. Da die Kommunikation nicht über Schnittstellen der zugrunde liegenden Programmiersprache realisiert ist, können leicht Fehler entstehen, wenn die beteiligten Dienste nicht exakt aufeinander abgestimmt sind. Dann ist keine Komposition der Dienste möglich. In diesem Punkt sollte der Dienstentwickler weitergehend unterstützt werden. Um der *Heterogenität* in eHomes Rechnung zu tragen, sollten Schnittstellen zur klaren Festlegung der syntaktischen Details eingesetzt werden, die, um semantische Beschreibungen ergänzt, dennoch eine Dienstkomposition auf der semantischen Ebene ermöglichen.

Dienste müssen so entwickelt werden, dass sie mit möglichst vielen anderen Diensten interagieren können. Die gute Komponierbarkeit ist eine wesentliche Anforderung an komponentenbasierte Ansätze, wie in Abschnitt 3.1.1 bereits diskutiert wurde. Sie ist Voraussetzung für die Wiederverwendbarkeit und damit die Reduzierung des Entwicklungsaufwands und der Entwicklungskosten.

Fazit

Der Ansatz von NORBISRATH liefert einen grundlegenden Schritt in Richtung einer kostengünstigen Softwareentwicklung für eHomes auf Basis vorgefertigter wiederverwendbarer Komponenten. Der SCD-Prozess und der eHomeConfigurator bieten die nötige Unterstützung, um Komponenten, die nach den entsprechenden Vorgaben entwickelt wurden, zu konfigurieren und auf dem Gateway eines eHome-Systems zu deployen. Der Ansatz lässt sich aber, wie oben beschrieben, in wichtigen Punkten verbessern, wenn es um die Unterstützung von Mobilität und Dynamik oder die semantische Adaption geht. Bisher können diese Aspekte nur durch Mechanismen unterstützt werden, die von den Entwicklern für den jeweiligen Dienst individuell implementiert werden. Es ist eine Laufzeitumgebung erforderlich, die in diesen Punkten Unterstützung bietet, damit der Entwicklungsaufwand reduziert wird und inkonsistente oder fehlerhafte Implementierungen vermieden werden. Änderungen der Benutzeranforderungen können dann abgebildet werden und einer weiterführende semantische Dienstkomposition wird ermöglicht. Die beschriebenen Weiterentwicklungen des bisherigen Ansatzes werden in dieser Arbeit adressiert. Die folgenden Kapitel beschreiben die dazu entwickelten Lösungsansätze.

3.5 Zusammenfassung

In diesem Kapitel wurden verschiedene Themen behandelt, die dieser Arbeit als Grundlage dienen. Die Ansätze der komponentenbasierten Softwareentwicklung stellen die Basis der Dienstentwicklung dar und fanden bereits in den Vorarbeiten Anwendung. Die Konzepte serviceorientierter Architekturen sind für die Erweiterung des bisherigen Ansatzes und die Modifikation des SCD-Prozesses zur Laufzeitunterstützung von Bedeutung. Die modellgetriebene Softwareentwicklung findet bei der Umsetzung des Lösungsansatzes dieser Arbeit Anwendung. Auch in den Vorarbeiten wurde dieser Weg bereits verfolgt. Schließlich wurden die wichtigsten Aspekte der Vorarbeiten im eHome-Projekt am Lehrstuhl für Informatik 3 vorgestellt, gefolgt von einer Diskussion der Defizite des bisherigen Ansatzes. Die Lösungskonzepte, die in den folgenden Kapiteln vorgestellt werden, haben das Ziel, diese Defizite zu überwinden.

Kapitel 4

Strukturelle Adaption

In diesem Kapitel werden zunächst die zu berücksichtigenden Einflussfaktoren auf dynamische Veränderungen in eHomes erläutert. Daraufhin wird die *strukturelle Adaption* eingeführt und die zugrunde liegende Idee näher erläutert. Anschließend werden die im Rahmen der vorliegenden Arbeit entwickelten Konzept zur Umsetzung der strukturellen Adaption vorgestellt. Diese stellen ein wesentliches Ergebnis dieser Arbeit dar. Dabei werden auch die Erweiterungen und Unterschiede zum bisherigen Konfigurierungsansatz deutlich gemacht. Diese betreffen insbesondere den SCD-Prozess, die Dienstspezifikation und die Laufzeitumgebung des eHome-Systems. Abschließend werden verwandte Arbeiten untersucht und im Zusammenhang mit dem hier entwickelten Ansatz bewertet.

4.1 Mobilität und Dynamik in eHome-Systemen

In Kapitel 2 und in Abschnitt 3.4.6 wurde bereits erwähnt, dass eHome-Systeme zur Laufzeit zahlreichen Veränderungen unterliegen. Eine wichtige Anforderung an eHome-Systeme ist daher, Unterstützung für diese Veränderungen zu bieten. Eine Anpassung an Veränderungen der Umgebung ist für die Funktionalität vieler Dienste unerlässlich. Zwar kann jeder Dienst eine individuelle Lösung zur Berücksichtigung von *Mobilität* und *Dynamik* implementieren, es wurde aber bereits darauf hingewiesen, dass dies verschiedene Nachteile mit sich bringt. Zum einen bedingt dies einen deutlichen Mehraufwand bei der Implementierung der Dienste. Zum anderen verhält sich dadurch jeder Dienst anders in Bezug auf Veränderungen, da jeder Dienst eine eigene Implementierung mit sich bringt, die im Allgemeinen nicht konsistent zu anderen Implementierungen ist. Um diese Situation zu verbessern, ist es erforderlich, dass dynamische Veränderungen bereits auf der Ebene der Laufzeitumgebung Beachtung finden und den Dienstentwicklern Mechanismen zur Verfügung gestellt werden, auf die sie bei der Dienstimplementierung zurückgreifen können.

4.1.1 Einflussfaktoren

Veränderungen zur Laufzeit eines eHome-Systems können verschiedene Ursachen haben. Es gibt fünf wichtige Einflussfaktoren, die in dieser Arbeit betrachtet werden:

1. Variabilität der Benutzeranforderungen
2. In-Home-Mobilität von Benutzern
3. Inter-Home-Mobilität von Benutzern
4. Variabilität der Geräteumgebung
5. Mobile Geräte

Diese Einflussfaktoren werden im Folgenden näher beleuchtet.

Variabilität der Benutzeranforderungen

Die *Benutzeranforderungen* spiegeln die Wünsche des Benutzers wider, d. h. welche Dienste gewünscht werden und wo bzw. für wen deren Funktionalitäten erbracht werden sollen. Diese Wünsche können sich natürlich im Laufe der Zeit ändern. Gibt es z. B. einen neuen Dienst von dem der Benutzer Gebrauch machen möchte, so muss das System entsprechen angepasst werden. Es muss möglich sein, weitere Dienste im laufenden Betrieb hinzuzufügen. Ein neuerer Dienst ersetzt möglicherweise einen älteren, sodass dieser nicht länger genutzt werden kann. Daher muss auch das Entfernen bereits aktiver Dienste unterstützt werden.

Die Häufigkeit, in der solche Veränderungen auftreten, kann je nach Dienst und Benutzer ganz unterschiedlich ausfallen. Manche Dienste können über mehrere Jahre hinweg unverändert genutzt werden, während andere Dienste innerhalb eines Tages mehrfach verändert werden müssen. Ein Sicherheitsdienst, der das eHome gegen Einbrüche schützt, wird vielleicht über einen sehr langen Zeitraum verwendet ohne irgendwelchen Veränderungen zu unterliegen. Ein medizinischer Betreuungsdienst hingegen wird möglicherweise nur für einen begrenzten Zeitraum einer Erkrankung des Benutzers benötigt. Ein Musikdienst wiederum kann mehrmals am selben Tag in Anspruch genommen werden.

Das eHome-System muss mit diesen unterschiedlichen Situation umgehen können. Stehen verschiedene Geräte für das Erbringen einer Funktionalität zur Verfügung, so möchte der Benutzer in die Auswahl eingreifen können. Für den Musikdienst könnten z. B. verschiedene Lautsprecher verfügbar sein, die eine unterschiedliche Klangqualität ermöglichen. Dann sollte es dem Benutzer möglich sein, die Zuordnung zwischen Diensten und Geräten zu beeinflussen. So könnte der Benutzer ggfs. eine automatisch vom eHome-System hergestellte Bindung manuell anpassen.

In-Home-Mobilität von Benutzern

Nicht nur die Auswahl der Dienste, die genutzt werden sollen, unterliegt Veränderungen, sondern auch die räumliche Zuordnung, also wo innerhalb der eHome-Umgebung die Dienste genutzt werden sollen. Betrachtet man den oben erwähnten Musikdienst, so wird klar, dass der Dienst seine Funktionalität in räumlicher Nähe zum aktuellen Aufenthaltsort des Benutzers erbringen sollte. Wechselt der Nutzer seinen Aufenthaltsort, weil er sich innerhalb es eHomes bewegt, so wird erwartet, dass die Funktionalität des Musikdienstes sich „mitbewegt“ und die Musik am neuen Aufenthaltsort des Benutzers zu hören ist. Hier ist die Nutzung eines Dienstes also vom räumliche Kontext abhängig, der unter anderem die aktuelle Position des Benutzers innerhalb des eHomes umfasst. Ändert der Benutzer seine Position, so können davon auch seine persönlichen Dienste betroffen sein.

Die Mobilität der Benutzer eines eHomes innerhalb der eHome-Umgebung wird auch *In-Home-Mobilität* genannt. Gerade für die In-Home-Mobilität ist eine Unterstützung durch die Laufzeitumgebung des eHome-Systems von hoher Bedeutung. Es muss den Benutzern möglich sein, Dienstfunktionalitäten innerhalb des eHomes mitzunehmen. Für viele Dienste wird gerade dadurch ein Mehrwert gegenüber einer klassischen Realisierung ohne eHome-System erreicht. Die bisherige Lösung, die darauf basiert, raumbezogene Dienste an allen denkbaren Aufenthaltsorten des Benutzer vorzuinstallieren, bindet unnötig Ressourcen und ist daher für größere Umgebungen und eine größere Anzahl von Diensten nicht anwendbar. Darüber hinaus muss dann jeder Dienst eine eigene Personenerkennung samt einer Strategie für die Auflösung von Interessenkonflikten bei mehreren Personen implementieren. Das erhöht den Entwicklungsaufwand pro Dienst und führt außerdem, wie bereits in Abschnitt 3.4.6 erläutert, zu inkonsistentem Verhalten unter den Diensten. Um die Entwicklungskosten zu senken und die Akzeptanz der Benutzer zu erhöhen, ist daher eine entsprechende Unterstützung dynamischer Veränderungen durch das eHome-System erforderlich.

Inter-Home-Mobilität von Benutzern

Die Benutzer eines eHomes bewegen sich nicht nur innerhalb der eHome-Umgebung, sondern verlassen das eHome und betreten das eHome später erneut. Sie können sich somit auch von einer eHome-Umgebung in eine andere eHome-Umgebung bewegen. Im Gegensatz zur In-Home-Mobilität, die sich innerhalb eines eHomes abspielt, wird hier von *Inter-Home-Mobilität* gesprochen. Auch die Inter-Home-Mobilität erfordert Anpassungen des eHome-Systems. Verlässt etwa ein Bewohner das eHome, so können Funktionalitäten, die nur dieser Benutzer in Anspruch genommen hat, wegfallen. Die entsprechenden Dienste können dann gestoppt und deinstalliert werden. Wenn umgekehrt ein neuer Benutzer das eHome betritt, müssen entsprechend die Funktionalitäten, die von diesem Benutzer gewünscht werden, hinzugefügt werden. Dazu sind die entsprechenden Dienste zu installieren und zu starten.

Auch die Inter-Home-Mobilität verlangt daher nach einer Unterstützung für dynamische Veränderungen durch die Laufzeitumgebung des eHomes. Wie die Mitnahme von Diensten und deren Funktionalitäten von einem eHome zum anderen realisiert werden kann steht nicht im Fokus dieser Arbeit. Daher werden hier auch keine Details zu einer Lösung dieser Problemstellung diskutiert. Das Thema der Migration von Diensten, Dienstfunktionalitäten und Benutzerprofilen im

Rahmen der Inter-Home-Mobilität und die damit zusammenhängenden Fragestellungen der Sicherheit und des Schutzes der Privatsphäre werden insbesondere in den Arbeiten von ARMAÇ ausführlicher betrachtet [AE08, Arm08, AR08].

Variabilität der Geräteumgebung

Bisher wurden verschiedene Einflussfaktoren untersucht, die Veränderungen des eHome-Systems bedingen und die auf Benutzeraktivitäten innerhalb des eHomes zurückzuführen sind. Es sind jedoch nicht nur die Benutzer für Veränderungen zur Laufzeit verantwortlich. Es können sich auch Veränderungen der Geräteumgebung ergeben. Zum einen können neue Geräte angeschlossen werden, die dann auch vom eHome-System genutzt werden sollen. Diese Geräte müssen also für die laufenden Dienste zur Verfügung gestellt werden. Daher muss die Dienstkomposition ggfs. angepasst werden, sodass die neu verfügbaren Ressourcen auch tatsächlich für die Bereitstellung der Funktionalitäten verwendet werden. Zum anderen kann es vorkommen, dass vorhandene, von den Diensten verwendete Geräte in der Umgebung nicht länger genutzt werden können. Dies kann auftreten, wenn ein vorhandenes Gerät vom Benutzer aus dem eHome-System entfernt wird. Ebenso kann es aber vorkommen, dass ein Gerät aufgrund eines Defekts nicht länger die gewünschte Funktionalität liefern kann. In diesen Fällen sind dynamische Anpassungen des eHome-Systems erforderlich, die auf einer Veränderung der Geräteumgebung basieren.

Mobile Geräte

Ein weiterer Einflussfaktor auf Veränderungen im eHome-System sind mobile Geräte. Ähnlich wie bei der In-Home-Mobilität der Benutzer, erfordern auch mobile Geräte eine Anpassung des eHome-Systems. Naturgemäß werden mobile Geräte von den Bewohnern mitgeführt und bewegen sich entsprechend in der eHome-Umgebung. Daher können Dienste, die ihre Funktionalität in einem bestimmten Raum zur Verfügung stellen, auch nur zeitweise auf solche mobilen Ressourcen zurückgreifen. Hier ist also erneut eine laufende Anpassung des eHome-Systems erforderlich. Falls ein personenbezogener Dienst von einem Benutzer mitgeführt wird und auf die Funktionalitäten eines mobilen Geräts zurückgreift, das von demselben Benutzer mitgeführt wird, so kann diese Konstellation ausgenutzt werden, um die erforderlichen Anpassungen zu reduzieren. Da sich in diesem Fall der Dienst und die genutzte Ressource gemeinsam in der Umgebung bewegen, braucht keine Anpassung stattzufinden. Erst wenn das mobile Gerät abgelegt wird, müssen andere Ressourcen in der jeweiligen Umgebung des Benutzers gesucht werden.

Die in dieser Arbeit betrachtete Art von Dynamik in eHome-Systemen wird durch die fünf oben erläuterten Faktoren beeinflusst. Ein wichtiges Ziel ist es, eine geeignete Unterstützung von eHome-Systemen für diese Dynamik zu ermöglichen. Die dynamischen Veränderungen müssen durch das eHome-System erfasst und verarbeitet werden können. Dies ist durch den bisherigen Ansatz nicht gegeben (vgl. Abschnitt 3.4.6), da nur die einmalige Ausführung des SCD-Prozesses zur Konfigurierung des eHome-Systems möglich ist.

4.1.2 Adaptionsansatz

Der Ansatz von NORBISRATH (vgl. Abschnitt 3.4) basiert auf der Konfigurierung des eHome-Systems durch die Komposition von Dienstkomponenten im Rahmen des SCD-Prozesses. Das in der vorliegenden Arbeit neu entwickelte Konzept zur Unterstützung dynamischer Veränderungen basiert auf der Adaption durch die im SCD-Prozess erstellten Konfigurationen [Ret07]. In diesem Kapitel wird dazu im Folgenden die *strukturelle Adaption* eingeführt, die ein wesentliches Ergebnis dieser Arbeit darstellt.

Der in dieser Arbeit vorgestellte Lösungsansatz zur Adaption von eHome-Systemen lässt sich in mehrere Schichten einteilen. In Abbildung 4.1 ist dies dargestellt. Im unteren Teil ist die *Systemumgebung* zu sehen. Diese besteht zum einen aus der *Hardwareplattform*, darauf läuft ein *Betriebssystem*. Beide Schichten bilden zusammen das sogenannte *Residential Gateway*. Auf dem Betriebssystem wird im Fall des hier vorgestellten Ansatzes eine virtuelle Maschine für Java, die *Java VM*, ausführt.

Oberhalb der Systemumgebung, die sich aus den oben beschriebenen Schichten ergibt, liegt eine *Middleware*-Schicht. Diese stellt den Kern dieser Arbeit dar. Hier befindet sich die *eHome-Laufzeitumgebung*. Durch diese wird das eHome-System konfiguriert und eine *Dienstkomposition* erstellt. Die Java VM und die eHome-

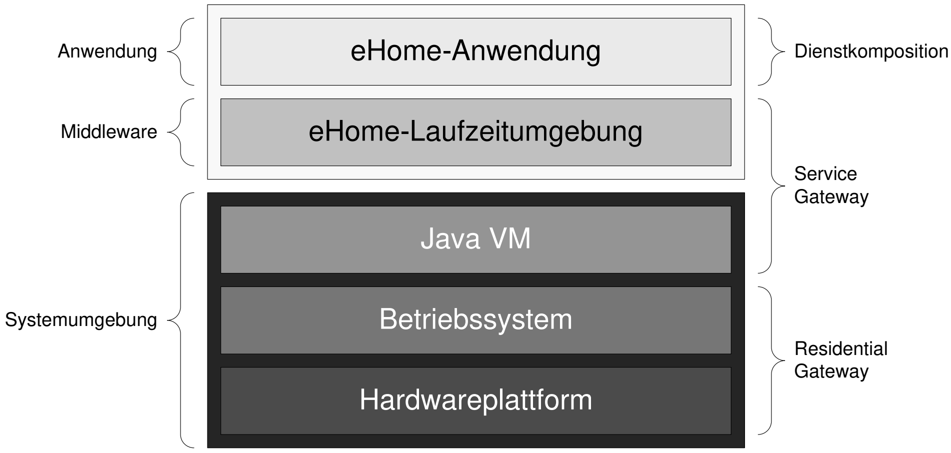


Abbildung 4.1: Schichten des Lösungsansatzes.

Laufzeitumgebung, die auf dem Residential Gateway ausgeführt werden, heißen auch *Service Gateway*. Sie bilden zusammen die Grundlage zur Ausführung der Dienstkomposition, die hier *eHome-Anwendung* genannt wird.

Die Dienste werden im Ansatz dieser Arbeit also allgemein auf dem Residential Gateway des eHomes ausgeführt, also einem zentralen Server, auf dem die Softwareplattform, das Service Gateway, ausgeführt wird. Das eHome-System ist nur in so fern verteilt, als dass die genutzte Hardware, also die im eHome vorhanden Geräte, auf die verschiedenen Räume verteilt sind. Auch auf diesen Geräten kann natürlich Software ausgeführt werden, die über ein Netzwerk Funktionalitäten zur Verfügung stellt. Diese Funktionalitäten können dann durch Treiberdienste im eHome verfügbar gemacht werden. Die Betrachtung der Dienstkomposition endet im Rahmen dieser Arbeit aber auf der Ebene der Treiberdienste, die eine Abstraktionsschicht in Bezug auf die Gerätefunktionalitäten bildet. Daher wird hier nicht von einem verteilten System ausgegangen.

Die eHome-Anwendung stellt die den Anforderungen des Benutzers und der Umgebung entsprechende Anwendungssoftware dar. Es sind hier also zwei unterschiedliche Ebenen zu betrachten. Zum einen die Ebene der Laufzeitumgebung, welche als Ausführungsplattform für die eHome-Dienste dient und verschiedene unterstützende Funktionen anbietet, auf die vom Dienstentwickler zurückgegriffen werden kann. Zum anderen die Ebene der eHome-Anwendung, welche die den Anforderungen entsprechenden Funktionalitäten im eHome realisiert. Die

Softwarearchitektur wird in einem klassischen Softwareentwicklungsprozess in einer der Laufzeit und Implementierung vorausgehenden Entwurfsphase festgelegt [Nag90]. Eine solche statische, apriorische Festlegung ist im Fall der eHome-Anwendung jedoch nicht möglich, da diese laufend den aus der Dynamik resultierenden Änderungen angepasst werden muss. Die eHome-Anwendung ist daher flexibel und muss schrittweise durch die Laufzeitumgebung restrukturiert werden. Dazu wird in dieser Arbeit die strukturelle Adaption eingeführt.

Die *strukturelle Adaption* bezeichnet die Anpassung der im Konfigurierungsansatz erzeugten Struktur der eHome-Anwendung, d. h. seiner Softwarearchitektur. Die Struktur der eHome-Anwendung ergibt sich aus der Komposition der eHome-Dienste. Diese wird in der vorliegenden Arbeit wie auch bereits im bisherigen Ansatz als Graphstruktur modelliert und verarbeitet. Bisher war die Komposition jedoch nach der Erzeugung nicht weiter veränderbar. Durch die strukturelle Adaption wird die Anpassung der Dienstkomposition ermöglicht. So kann die eHome-Anwendung entsprechend den oben erläuterten dynamischen Veränderungen adaptiert werden.

Im Rahmen dieser Arbeit wird die strukturelle Adaption nicht durch einen zusätzlichen ergänzenden Prozess zur Laufzeit des eHome-Systems realisiert, sondern es wird auf den bereits vorhandenen SCD-Prozess zurückgegriffen. Der SCD-Prozess selbst wird dahingehend erweitert, dass er einen repetitiven Ablauf unterstützt. Er kann also wiederholt ausgeführt werden während das eHome-System bereits in Betrieb genommen wurde und bietet somit eine Laufzeitunterstützung die zuvor nicht gegeben war. Jeder Schritt des SCD-Prozesses kann so auch als wiederkehrender Anpassungsschritt aufgefasst werden. Dieser neue, adaptive Prozess wird auch *kontinuierlicher SCD-Prozess* genannt und ist in einen erweiterten Entwicklungsprozess eingebettet, der im folgenden Abschnitt 4.2 vorgestellt wird.

4.2 Erweiterter Entwicklungsprozess

Zur Berücksichtigung der in Abschnitt 4.1 beschriebenen *Mobilität* und *Dynamik* in eHome-Systemen wird in diesem Abschnitt ein *erweiterter Entwicklungsprozess* vorgestellt. Dieser erweiterte Entwicklungsprozess wurde im Rahmen der vorliegenden Arbeit zur Umsetzung der *strukturellen Adaption* entwickelt. Dazu wird ein angepasster SCD-Prozess eingeführt, der in den neuen Entwicklungsprozess eingebettet ist. Der neu entwickelte Ansatz basiert außerdem auf einer erweiterten Dienstspezifikation, die in Abschnitt 4.3 vorgestellt wird. Zunächst wird der gesamte Lebenszyklus eines eHome-Systems betrachtet, der die Entwicklungsphasen der eHome-Dienste, die Nutzung der Dienste in einem konkreten eHome und letztendlich die Stilllegung des eHome-Systems umfasst.

Der erweiterte Entwicklungsprozess ist eine weitere Ausbaustufe des bereits in Abschnitt 3.4.1 erläuterten klassischen Entwicklungsprozesses und des in Abschnitt 3.4.2 von NORBISRATH eingeführten Entwicklungsprozesses, der bereits eine Verbesserung durch wiederverwendbare Komponenten und Werkzeugunterstützung ermöglicht, jedoch keine Adaption zur Laufzeit unterstützt.

Der hier vorgestellte Ansatz zur *strukturellen Adaption* basiert auf Grundlagen, die in den Arbeiten [Ste08], [Xue08] und [Kul09] gelegt wurden und die auf dem von NORBISRATH eingeführten Modellierungsansatz (vgl. Abschnitt 3.4.4) basieren. Dadurch können auch die im Rahmen dieser Arbeit neu entwickelten Werkzeuge in einem modellgetriebenen Softwareentwicklungsprozess realisiert werden. Zur Umsetzung des erweiterten Entwicklungsprozesses wurden jedoch auch an der Modellierung zahlreiche Änderungen vorgenommen, die in den folgenden Abschnitten diskutiert werden.

Abbildung 4.2 zeigt den erweiterten Entwicklungsprozess für eHome-Systeme. Wie bereits im bisherigen Ansatz ist oben der entkoppelte Bereich der Dienstentwicklung zu sehen. Dienste werden unabhängig vom individuellen eHome entwickelt. Hier ändert sich auf der Prozessebene zunächst nichts. Im Detail finden sich aber viele Unterschiede was die Dienstentwicklung und die Dienstspezifikation betrifft. Darauf wird in Abschnitt 4.3 näher eingegangen.

Die Neuerungen des Entwicklungsprozesses werden im eHome-spezifischen Teil deutlich. Der ursprüngliche SCD-Prozess findet sich hier in einer neuen Form wieder. Wie in Abschnitt 3.4.2 beschrieben, wurde in der Arbeit von NORBISRATH von der einmaligen Ausführung des SCD-Prozesses ausgegangen. Um die strukturelle Adaption zur Laufzeit zu ermöglichen, müssen nun auch im laufenden Betrieb des eHomes Änderungen berücksichtigt werden. Der wesentliche Entwicklungsschritt besteht darin, Änderungen, die im bisherigen Prozess

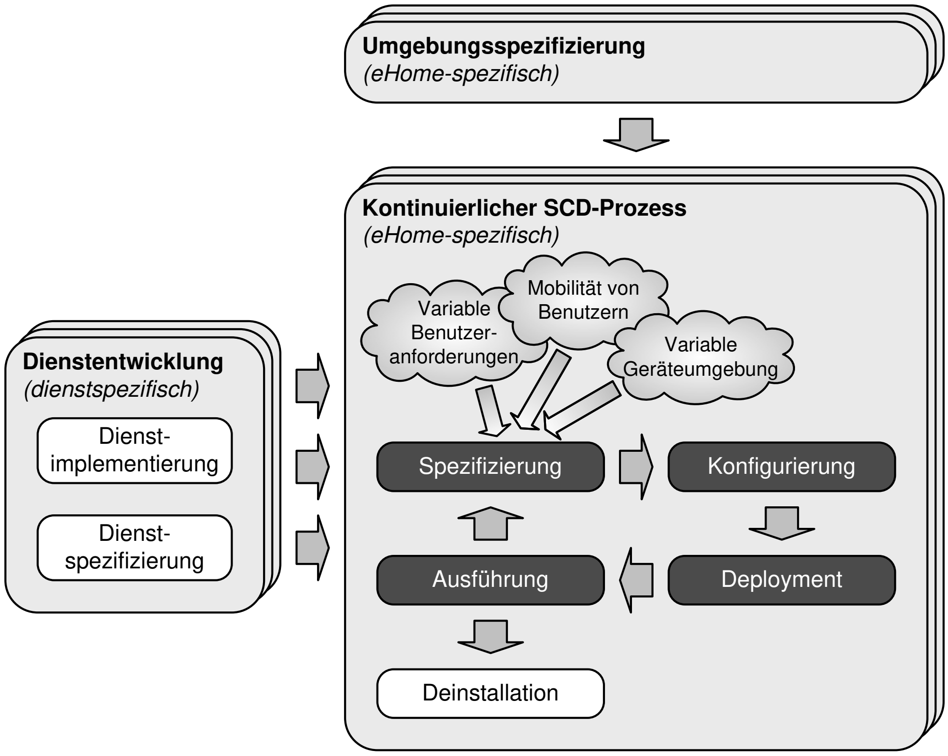


Abbildung 4.2: Erweiterter Entwicklungsprozess für dynamische eHome-Systeme mit eingebettetem kontinuierlichen SCD-Prozess.

nur durch einen Wartungsschritt möglich waren, in den regulären Ablauf des SCD-Prozesses einzubeziehen. Eine strukturelle Adaption war zuvor nur durch manuelle Anpassungen im Sinne von Wartung möglich. Diese bedingt, wie in Abschnitt 3.4.6 beschrieben, das Herunterfahren und Neustarten des eHome-Systems und stellt somit einen weitreichenden Eingriff dar. Es ist nicht möglich, die in Abschnitt 4.1 beschriebene Dynamik auf diese Weise abzubilden. Werden jedoch diese Änderungen nicht mehr durch Wartung, sondern durch den regulären Ablauf des SCD-Prozesses im System erfasst, so ist eine automatisierte Verarbeitung möglich. Jede Veränderung zur Laufzeit führt zu Anpassungsoperationen in allen Phasen des SCD-Prozesses. Der SCD-Prozess wird also fortwährend durchgeführt, um die Änderungen im System zu erfassen. Aus diesem Grund wird der neue SCD-Prozess auch *kontinuierlicher SCD-Prozess* genannt.

In Abbildung 4.2 ist der kontinuierliche SCD-Prozess abgebildet. Zu erkennen ist die neue, zyklische Anordnung der Phasen des SCD-Prozesses, die den repetitiven Charakter zum Ausdruck bringt. Auch der kontinuierliche SCD-Prozess umfasst die Phasen der *Spezifizierung*, der *Konfigurierung* und des *Deployments*, welche die Nutzung der zuvor entwickelten eHome-Dienste in einem konkreten eHome-System ermöglichen. Die Phasen können jedoch nun wiederholt durchlaufen werden, um eine schrittweise Anpassung durchzuführen. Befindet sich das eHome-System in der Ausführungsphase, so kann ein Rückschritt zur Spezifizierung, der ersten Phase des SCD-Prozesses, erfolgen. Hierdurch können Änderungen der Anforderungen erfasst werden. Auf Basis der angepassten Spezifikation kann eine bestehende Konfiguration des eHome-Systems strukturell adaptiert und dann deployt werden. Auf diese Weise wird der SCD-Prozess fortlaufend durchgeführt und alle Änderungen, die sich zur Laufzeit des eHome-Systems ergeben, werden erfasst und in das System einbezogen. Um die Erweiterungen der Aktivitäten in den einzelnen Phasen im Vergleich zum ursprünglichen SCD-Prozess zum Ausdruck zu bringen, werden die Phasen des kontinuierlichen SCD-Prozesses auch *adaptive Spezifizierung*, *(Re-)Konfigurierung* und *Delta-Deployment* genannt.

Die Phasen des kontinuierlichen SCD-Prozesses bezeichnen jeweils Veränderungen an einer bestehenden Struktur. Da es sich um Vorgänge zur Laufzeit des eHome-Systems handelt, sind die jeweiligen Daten bereits vorhanden. Durch die erneute Ausführung des SCD-Prozesses werden Änderungen an diesen vorhandenen Daten vorgenommen. Die Besonderheiten der einzelnen Phasen werden in den folgenden Abschnitten dieses Kapitels genauer beschrieben. Zunächst soll jedoch im Abschnitt 4.3 auf die Änderungen der Dienstentwicklung eingegangen werden. Da der dienstspezifische Teil des Entwicklungsprozesses den eHome-spezifischen Phasen vorgelagert ist (vgl. Abbildung 4.2), müssen zunächst die nötigen Änderungen dieses Teils untersucht und verstanden werden.

4.3 Dienstentwicklung

Die Dienstentwicklung war bereits im bisherigen Ansatz vom eHome-spezifischen Teil des Entwicklungsprozesses entkoppelt. Durch die Entwicklung wiederverwendbarer Standardkomponenten zur Realisierung von eHome-Diensten ergeben sich Vorteile, die zu einer Reduzierung des Entwicklungsaufwands und der Kosten führen.

Der in der vorliegenden Arbeit in Abschnitt 4.2 neu eingeführte *erweiterte Entwicklungsprozess* ermöglicht eine kontinuierliche Adaption in Bezug auf dynamische Veränderungen in der eHome-Umgebung. Davon sind auch die eHome-Dienste betroffen, die laufend an Veränderungen der Umgebung angepasst werden müssen. Anders als bei der generativen Konfigurierung im bisherigen Ansatz kann bei der strukturellen Adaption im Allgemeinen keine Benutzerinteraktion stattfinden, da zahlreiche feingranulare Änderungen berücksichtigt werden müssen und zu häufige Anfragen beim Benutzer als sehr störend wahrgenommen werden. Daher müssen die verschiedenen Vorgänge bei der Adaption automatisiert durchgeführt werden können. Die bisherige Dienstspezifikation ist dazu nicht ausreichend, da sie nicht auf dynamische Anpassungen durch die Laufzeitumgebung ausgelegt war. Um die strukturelle Adaption zu unterstützen muss die vorhandene Dienstspezifikation daher an verschiedenen Stellen erweitert werden. Die im Rahmen der vorliegenden Arbeit entwickelten Erweiterungen werden im Folgenden näher beschrieben.

4.3.1 Diensttypen

In Abschnitt 3.4.3 wurde bereits eine Unterscheidung zwischen Diensttypen eingeführt und zwar im Zusammenhang mit der dreischichtigen Architektur, die sich bei der Dienstkomposition ergibt. Diese Einteilung in *Basis-* bzw. *Treiberdienste*, *integrierende Dienste* sowie *Top-Level-Dienste* kam bereits in den Vorarbeiten zur Anwendung.

Die Schichtung der Dienste ergibt sich aus den Diensttypen und der Dienstkomposition auf Basis der spezifizierten Funktionalitäten. Komponiert werden können Dienste, die sich auf gleiche Funktionalitäten beziehen. Zusätzlich müssen jedoch auch die bereits in Abschnitt 3.4.4 erwähnten Kardinalitäten berücksichtigt werden. Zu jeder von einem Dienst angebotenen oder benötigten Funktionalität werden Kardinalitätsgrenzen spezifiziert. Für angebotene Funktionalitäten wird durch eine Obergrenze angegeben, wie oft der Dienst die Funktionalität parallel anbieten kann. Dies kann z. B. bei Treiberdiensten durch Beschränkungen der Hardware erforderlich sein. Für benötigte Funktionalitäten wird im erweiterten Ansatz dieser Arbeit zusätzlich eine Untergrenze angegeben, welche angibt, wie oft der Dienst diese Funktionalität mindestens benötigt, um ausgeführt werden zu können. Im Gegensatz zur in Abschnitt 3.4.4 beschriebenen Modellierung wird nun also nicht mehr zwischen benötigten und optional benötigten Funktionalitäten unterschieden, da sich dies unmittelbar aus den Kardinalitätsangaben ergibt. Für benötigte Funktionalitäten wird wie oben beschrieben ein Intervall angegeben, sodass eine genauere Angabe möglich ist als zuvor. Durch die Kardinalitätsangaben wird festgelegt, wie viele Bindungen zu anderen Diensten hergestellt werden können bzw. müssen. Für die strukturelle Adaption sind die Kardinalitätsangaben ein wichtiger Teil der Dienstspezifikation, was in den weiteren Abschnitten deutlich wird.

Raumbezogene und personenbezogene Dienste

Die oben beschriebene Einteilung auf Basis der dreischichtigen Dienstarchitektur dient dazu, die funktionale Dienstkomposition zu veranschaulichen. Es gibt aber weitere Kriterien in denen sich Dienste unterscheiden. Von besonderer Bedeutung für die strukturelle Adaption ist die Unterscheidung zwischen *raumbezogenen* und *personenbezogenen* Diensten, die bei Top-Level-Diensten Anwendung findet. Im bisherigen Ansatz wurde diese Unterscheidung nicht getroffen, es gab nur raumbezogene Dienste. Wie in Abschnitt 3.4.6 erläutert, konnten personalisierte Dienste nur dadurch realisiert werden, dass raumbasierte Dienste eine eigene Personenerkennung und eine Verwaltung der Personendaten implementieren. Von der Laufzeitumgebung wurde dazu keinerlei Unterstützung geboten. Eine solche Unterstützung ist jedoch von großer Bedeutung für die Entwicklung personenbezogener Dienste. In der Praxis kann nicht jeder Dienst eine eigene Kontextverwaltung implementieren, es ist daher eine übergreifende, globale Behandlung von Kontextdaten erforderlich, besonders wenn auch Inter-Home-Mobilität unterstützt werden soll. Weitere Überlegungen zu verschiedenen Typen von eHome-Diensten und ihre Realisierung, insbesondere bezüglich der Inter-Home-Mobilität von Benutzern und Geräte, werden in [RAN09] betrachtet.

Auswirkungsort und Ausführungsort

Die Unterscheidung zwischen raumbezogenen und personenbezogenen Diensten basiert auf dem Kriterium des *Auswirkungsortes* der Dienstfunktionalität. Dieser ist zu unterscheiden vom *Ausführungsort* eines Dienstes. Wie bereits oben in Abschnitt 4.1.2 erläutert, werden Dienste im hier besprochenen Ansatz auf dem Residential Gateway des eHomes ausgeführt. Das Residential Gateway ist somit der Ausführungsort.

Der Auswirkungsort einer Dienstfunktionalität ist im Gegensatz zum Ausführungsort eines Dienstes sehr flexibel und stellt ein zentrales Kriterium bei der Konfigurierung des eHome-Systems dar. Bereits während der Dienstentwicklung ist der Auswirkungsort einer Dienstfunktionalität von Bedeutung. Bei raumbezogenen Diensten wird der Auswirkungsort der angebotenen Dienstfunktionalitäten durch die vom Benutzer festgelegte Zuordnung zwischen dem Dienst und einem bestimmten Raum vorgegeben. Bei personenbezogenen Diensten wird eine Zuordnung zwischen dem Dienst und einer Person hergestellt, für die die personalisierte Funktionalität angeboten werden soll. Dadurch wird implizit festgelegt, dass der Auswirkungsort der Dienstfunktionalität mit dem momentanen Aufenthaltsort der Bezugsperson übereinstimmen soll.

Im kontinuierlichen SCD-Prozess muss daher die Mobilität der Benutzer berücksichtigt werden, der Auswirkungsort des Dienstes muss also entsprechend den Bewegungen der Bezugsperson angepasst werden. Da die Funktionalitäten eines Top-Level-Dienstes auf Basis gebundener Dienste niedrigerer Abstraktionsschichten realisiert werden, ergeben sich zusätzliche Anforderungen an die Dienstkomposition. Die direkt oder indirekt verwendeten konkreten Hardwareressourcen müssen am gewünschten Auswirkungsort des personenbezogenen Top-Level-Dienstes lokalisiert sein.

Dienste benötigen unter Umständen viele unterschiedliche Funktionalitäten. Diese können letztendlich durch verschiedene Ressourcen erfüllt werden. Um Festlegungen zur Bindung dieser Ressourcen in der Dienstspezifikation zu machen, muss die Betrachtung auf der Ebene der einzelnen Funktionalitäten ansetzen. Es reicht nicht aus, den Dienst als atomare Einheit der Spezifikation zu betrachten. Vielmehr ist eine feingranulare Betrachtung des Dienstes unter Berücksichtigung seiner benötigten Funktionalitäten erforderlich. So kann der Entwickler in der erweiterten Dienstspezifikation für jede von einem Dienst benötigte Funktionalität individuelle Anforderungen zu den zu bindenden Ressourcen und zur automatischen Komposition machen.

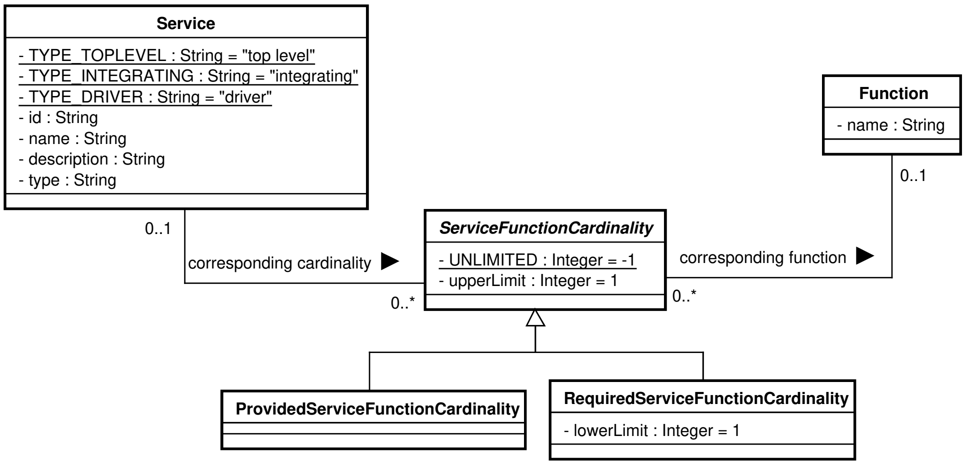


Abbildung 4.3: Modellierung der Dienstspezifikation.

Modellierung

Abbildung 4.3 zeigt die Modellierung der Dienste, wie sie in dieser Arbeit vorgenommen wurde. Das dargestellte UML-Klassendiagramm zeigt einen Ausschnitt des Gesamtmodells, das für die modellgetriebene Umsetzung des Ansatzes eingesetzt wird. Die Klasse Service repräsentiert einen eHome-Dienst. Der Dienst wird durch eine id identifiziert und besitzt darüber hinaus einen Namen (name) und einen Beschreibungstext (description). Der Typ des Dienstes wird mit dem Attribut type festgelegt, für das die Konstanten TYPE_TOPLEVEL, TYPE_INTEGRATING und TYPE_DRIVER definiert sind. Ein wichtiger Bestandteil der Dienstspezifikation ist die Beschreibung der Funktionalitäten, die ein Dienst anbietet und die er benötigt. Zu diesem Zweck besteht eine Verbindung zwischen den Klassen Service und Function. Die Verbindung ist indirekt über die Klasse ServiceFunctionCardinality realisiert, was die Möglichkeit eröffnet, zu jeder Verbindung verschiedene spezifische Attribute zu speichern. So wird beispielsweise die Unterscheidung zwischen einer angebotenen und einer benötigten Funktionalität durch die zwei Unterklassen ProvidedServiceFunctionCardinality und RequiredServiceFunctionCardinality modelliert. Da eine zugeordnete Funktionalität entweder angeboten oder benötigt wird, ist die Oberklasse ServiceFunctionCardinality abstrakt. Die Kardinalitätsangaben werden als Attribute gespeichert. In jedem Fall wird eine Obergrenze festgelegt, daher ist das Attribut upperLimit der Oberklasse ServiceFunctionCardinality zugeordnet. Die Konstante UNLIMITED bedeutet, dass die Kardinalität nach oben unbegrenzt ist. Eine Kardinalitätsuntergrenze kann nur für benötigte Funktionalitäten angegeben werden, da die Angabe einer Mindestanzahl von Nutzern einer Funktionalität keinen Sinn ergibt. Das Attribut lowerLimit ist daher der Unterklasse RequiredServiceFunctionCardinality zugeordnet.

4.3.2 Bindungsbeschränkungen

Um personenbezogene Dienste durch die Laufzeitumgebung zu unterstützen, wurden im Rahmen der vorliegenden Arbeit sogenannte *Bindungsbeschränkungen* (engl. *Binding Constraints*) eingeführt. Diese stellen eine der in dieser Arbeit entwickelten Erweiterungen der bisherigen Dienstspezifikation dar. Bindungsbeschränkungen dienen dazu, bei der Dienstkomposition die Menge der potentiell nutzbaren Ressourcen auf Grund bestimmter Kriterien, wie etwa der räumlichen Zuordnung im eHome, einzuschränken. Auf diese Weise kann das Erstellen einer Bindung bei der Konfigurierung des eHome-Systems an zuvor spezifizierte Bedingungen geknüpft werden.

Bindungsbeschränkungen können in der Dienstspezifikation den vom Dienst benötigten Funktionalitäten zugewiesen werden. Die Zuordnung von Bindungsbeschränkungen kann dabei individuell für jede benötigte Funktionalität erfolgen. So können für einen Dienst mit mehreren benötigten Funktionalitäten auch verschiedene Bindungsbeschränkungen gelten. Das Anlegen von Bindungen für diese Funktionalitäten ist dann an die jeweils zugeordneten Bedingungen gekoppelt. Nur wenn die Bedingungen erfüllt sind, kann eine Bindung hergestellt werden. Einer einzelnen benötigten Funktionalität können auch mehrere Bindungsbeschränkungen zugewiesen werden, die dann alle erfüllt sein müssen, bevor eine Bindung hergestellt werden kann. Der Konfigurierungsmechanismus des kontinuierlichen SCD-Prozesses berücksichtigt die Bindungsbeschränkungen bei der Dienstkomposition (siehe Abschnitt 4.4.2).

Es gibt verschiedene Möglichkeiten, Bindungsbeschränkungen zu formulieren. Im Rahmen dieser Arbeit wurde ein Ansatz gewählt, der auf vordefinierten Bindungsbeschränkungen basiert. Die Bindungsbeschränkungen werden also nicht vom Dienstentwickler selbst formuliert, sondern aus einer Liste vorgegebener

Bindungsbeschränkungen ausgewählt. Dies hat den Vorteil, dass die Dienstentwicklung nicht unnötig verkompliziert wird. Die Bindungsbeschränkungen liegen in Form einer deklarativen Beschreibung vor. Da der Ansatz auf einer graphischen Modellierung basiert, sind die Bindungsbeschränkungen in Form von Graphmustern spezifiziert. Diese Graphmuster werden bei der Konfigurierung des eHome-Systems geprüft und mit dem Konfigurationsgraphen, der den aktuellen Laufzeitzustand des Systems darstellt, abgeglichen. Auf diese Weise wird eine Konfiguration erstellt, die die Bindungsbeschränkungen erfüllt.

Durch den graphbasierten Ansatz können in den Graphmustern der Bindungsbeschränkungen alle Informationen des eHome-Modells verwendet werden, wodurch sich ein breites Spektrum an möglichen Bedingungen ergibt. Zwar kann der Dienstentwickler bei der Spezifikation dieses Spektrum nicht beliebig ausschöpfen, da er auf vorgegebene Bindungsbeschränkungen zurückgreift, jedoch ist es auf diese Weise einfach möglich, neue Bindungsbeschränkungen in das Spezifikationswerkzeug zu integrieren. Im Folgenden werden einige der im Rahmen der vorliegenden Arbeit eingeführten Bindungsbeschränkungen erläutert.

Beschränkung auf lokale Ressourcen

Personenbezogene Dienste bieten ihre Funktionalität am aktuellen Aufenthaltsort ihrer Bezugsperson an. Daher müssen sie bei der Realisierung ihrer Funktionalität auf Ressourcen an genau diesem Aufenthaltsort zurückgreifen. Es ist für die Komposition der Dienste daher wichtig, festzulegen, ob gebundene Ressourcen im gesamten eHome genutzt werden können oder ob nur Ressourcen aus demjenigen Umgebungselement eingesetzt werden sollen, das dem Ausführungsort des Dienstes und damit dem Aufenthaltsort der Bezugsperson entspricht. Zu diesem Zweck wurde die *Bindungsbeschränkung auf lokale Ressourcen* eingeführt. Diese Beschränkung sorgt dafür, dass für die jeweilige Funktionalität nur auf Ressourcen im selben Umgebungselement zurückgegriffen werden darf. Dazu muss ausgehend vom betrachteten Dienst die Dienstkomposition bis hin zu den Basisdiensten überprüft werden. Diese Verbindung kann sich über mehrere Schichten erstrecken, da möglicherweise integrierende Dienste verwendet werden. Wichtig ist, dass sich die letztendlich angesteuerten Geräte dort befinden müssen, wo der Dienst seinen Auswirkungsort hat.

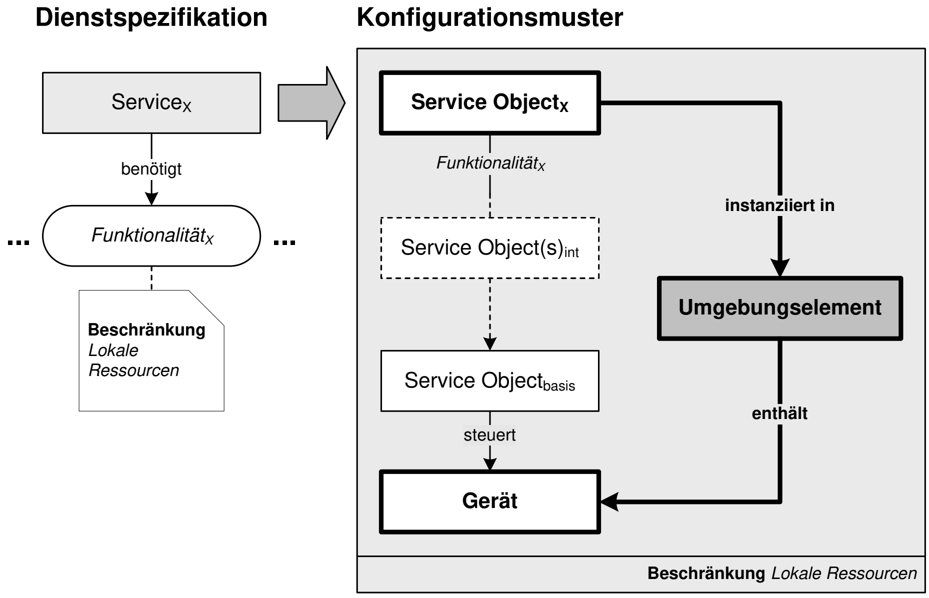


Abbildung 4.4: Bindungsbeschränkung auf lokale Ressourcen.

In Abbildung 4.4 ist die deklarative Beschreibung der Bindungsbeschränkung auf lokale Ressourcen graphisch dargestellt. Auf der linken Seite ist zunächst die Spezifikation eines Dienstes Service$_X$ dargestellt, der die Funktionalität Funktionalität$_X$ benötigt. Diese Anforderung ist zusätzlich mit der Bindungsbeschränkung Lokale Ressourcen versehen. Der Dienst benötigt möglicherweise noch weitere Funktionalitäten, diese sind jedoch für die Betrachtung der Bindungsbeschränkung nicht von Bedeutung.

Auf der rechten Seite der Abbildung ist das Konfigurationsmuster dargestellt, das durch die Bindungsbeschränkung auf lokale Ressourcen impliziert wird. In diesem Graphmuster ist oben zunächst ein Service Object$_X$ dargestellt, welches ein Dienstobjekt des Dienstes Service$_X$ auf der linken Seite ist. Wie jedes Dienstobjekt ist Service Object$_X$ über eine Kante instanziiert in einem Umgebungselement zugeordnet. Dadurch wird ausgedrückt, dass sich die von dem Dienstobjekt angebotene Funktionalität in diesem Umgebungselement auswirken soll. In einer gültigen Konfiguration muss die für den Dienst Service$_X$ benötigte Funktionalität Funktionalität$_X$ durch ein gebundenes Dienstobjekt zur Verfügung gestellt werden. Dies kann z. B. durch einen integrierenden Dienst Service Object$_{int}$ geschehen. Ein solcher integrierender Dienst kann wiederum auf weiteren integrierenden Diensten basieren. In jedem Fall wird zur tatsächlichen Realisierung der Funktionalität mittelbar (über integrierende Dienste) ein Basisdienst Service $\text{Object}_{\text{basis}}$ benötigt, der direkt ein entsprechendes Gerät steuert. Um sicherzustellen, dass sich die Funktionalität $\text{Funktionalität}_{\text{X}}$ von Service Object_{X} tatsächlich im zugehörigen Umgebungselement auswirkt, muss geprüft werden, ob genau dieses Umgebungselement das auf der Basisschicht angesteuerte Gerät enthält. Letztendlich werden alle Funktionalitäten eines Dienstes durch die gebundenen Ressourcen auf der Basisschicht realisiert und wirken sich somit am Standort der angesteuerten Ressourcen aus.

Da integrierende Dienste auf mehreren anderen Diensten basieren können, ist es möglich, dass ausgehend von Service Object_{X} mehrere Basisdienste mittelbar angebunden sind. Die $\text{Funktionalität}_{\text{X}}$ wird dann durch mehrere verschiedene Ressourcen umgesetzt. In diesem Fall müssen alle diese Ressourcen im spezifizierten Umgebungselement liegen.

In Abschnitt 4.3.1 wurde zwischen raumbezogenen und personenbezogenen Diensten unterschieden. Die Bindungsbeschränkung auf lokale Ressourcen kann zur Realisierung solcher personenbezogener Dienste eingesetzt werden. Im Graphmuster aus Abbildung 4.4 ist keine Bezugsperson für den spezifizierten Dienst $\text{Service}_{\text{X}}$ dargestellt. Dies ist auch nicht erforderlich, da es keine Bedeutung hat für *welche* Person der Dienst seine Funktionalität anbietet. Wird das Dienstobjekt Service Object_{X} zur Laufzeit einer Person zugeordnet, so wird dieses bei einer Änderung des Aufenthaltsortes dieser Person automatisch mitbewegt. Auf diese Weise „folgt“ das Dienstobjekt der Person. Die Bindungsbeschränkung ist daher nicht nur für personenbezogene Dienste einsetzbar. Auch für raumbasierte Dienste kann zwischen Funktionalitäten, die sich im selben Umgebungselement wie der Dienst auswirken müssen, und Funktionalitäten, die sich im gesamten eHome auswirken können, unterschieden werden. Diese Dienste folgen keiner Person, können aber im erweiterten Ansatz dieser Arbeit dennoch vom Benutzer im laufenden Betrieb neu positioniert werden. Auch dann muss ggfs. eine Rekonfigurierung erfolgen, wenn es Beschränkungen auf lokale Ressourcen gibt.

Im Folgenden soll das Konzept der Bindungsbeschränkungen an einem Beispiel veranschaulicht werden. In Abbildung 4.5 ist ein Beispiel der Bindungsbeschränkung auf lokale Ressourcen für einen personenbezogenen Dienst dargestellt. Hier geht es um einen Dienst, der das Raumklima in der Umgebung des Benut-

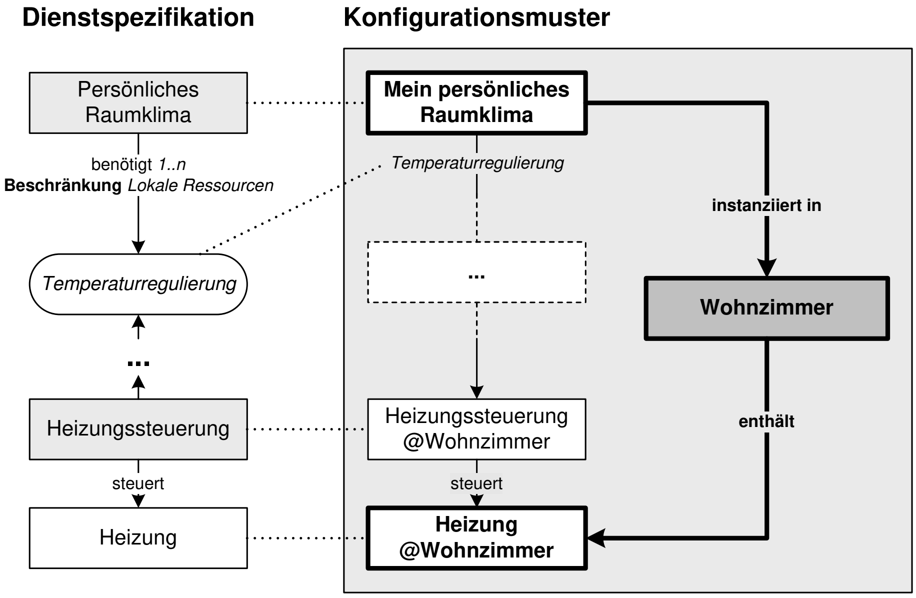


Abbildung 4.5: Beispiel zur Bindungsbeschränkung auf lokale Ressourcen.

zers an dessen persönliche Vorlieben anpasst. Auf der linken Seite sind Dienstspezifikationen dargestellt. Oben links ist der Dienst Persönliches Raumklima zu sehen. Dies ist ein Top-Level-Dienst, der die Funktionalität Temperaturregulierung benötigt. Als Kardinalität ist für diese Funktionalität das Intervall 1..n angegeben. Die erweiterte Spezifikation umfasst außerdem die Bindungsbeschränkung auf lokale Ressourcen. Unten links ist die Spezifikation eines Treiberdienstes Heizungssteuerung zu sehen, der Geräte vom Typ Heizung steuern kann. Dieser Dienst bietet direkt oder indirekt, d. h. über hier nicht einzeln dargestellte integrierende Dienste, die Funktionalität Temperaturregulierung an. Auf der rechten Seite ist ein Konfigurationsmuster dargestellt, das eine mögliche Konfiguration zur Laufzeit des eHomes darstellt. Hier ist ein Dienstobjekt Mein persönliches Raumklima zu sehen, das einer bestimmten Person im eHome zugeordnet sei. Befindet sich diese Person etwa im Wohnzimmer der eHome-Umgebung, so muss das Dienstobjekt ebenfalls diesem Raum zugeordnet sein, damit die entsprechende Funktionalität dort verfügbar wird. Dies wird durch die Kante instanziiert in repräsentiert. Im Konfigurationsgraphen ist nun eine Bindung für die benötigte Funktionalität Temperaturregulierung erforderlich. Hier im Beispiel wird diese, möglicherweise über eine Kette mehrerer integrierender Dienste, letztlich durch das Dienstobjekt Heizungssteuerung@Wohnzimmer bereitgestellt. Die Angabe @Wohnzimmer in der Bezeichnung des Dienstobjekts zeigt an, dass dieses im Wohnzimmer lokalisiert ist und damit dort die Funktionalität Temperaturregulierung anbietet. Der Treiberdienst steuert also die Heizung im Wohnzimmer, welche in der Konfiguration mit Heizung@Wohnzimmer bezeichnet wird. Die räumliche Zuordnung des Geräts wird durch eine Kante enthält vom Wohnzimmer zum Gerät Heizung@Wohnzimmer repräsentiert. Durch die Bindungsbeschränkung auf lokale Ressourcen wird somit sichergestellt, dass für die Erfüllung der benötigten Funktionalität nur Ressourcen aus demselben Umgebungselement des eHomes verwendet werden. Dies ist im Konfigurationsmuster durch den mit der dicken Linie markierten Pfad dargestellt.

Beschränkung gegen semantisch äquivalente Bindungen

Bindungsbeschränkungen können nicht nur eingesetzt werden, um die kontextbezogene Konfigurierung zu unterstützen. Sie dienen auch dazu, bestimmte unerwünschte Konfigurationen auszuschließen. Die Dienstspezifikationen umfassen für alle angebotenen und benötigten Funktionalitäten auch Kardinalitätsangaben. Mit diesen kann genau festgelegt werden, wie oft eine angebotene Funktionalität durch andere Dienste parallel genutzt werden kann und wie oft eine benötigte Funktionalität zur Verfügung stehen muss, um die Anforderungen des Dienstes zu erfüllen. Es ist also möglich, dass ein Dienst seine Funktionalität mehreren Diensten gleichzeitig zur Verfügung stellt. Genauso ist es auch möglich, dass eine bestimmte Funktionalität mehrfach benötigt wird.

Für den effizienten Umgang mit Kardinalitäten bei der Konfigurierung des Systems sind in bestimmten Fällen Bindungsbeschränkungen erforderlich. Diese können eingesetzt werden, um zu verhindern, dass äquivalente Pfade entstehen. Hierbei wird die Äquivalenz auf der semantischen Ebene betrachtet. Semantisch äquivalente Pfade bezeichnen Ketten von Dienstbindungen in einer Konfiguration, die ausgehend von einem Dienstobjekt über die gleichen Funktionalitäten und Diensttypen zu gleichen Ressourcen führen. Eine solche Situation ist in vielen Fällen nicht erwünscht. Die Mehrfachverbindungen würden keinen weiteren Nutzen bringen, jedoch ggfs. einen erheblich höheren Verbrauch an Ressourcen bedeuten. Generell sind Mehrfachverbindungen jedoch durchaus sinnvoll einsetzbar, wenn mehrere verschiedene benötigte Funktionalitäten von ein und derselben Ressource zur Verfügung gestellt werden können. Dabei handelt es sich

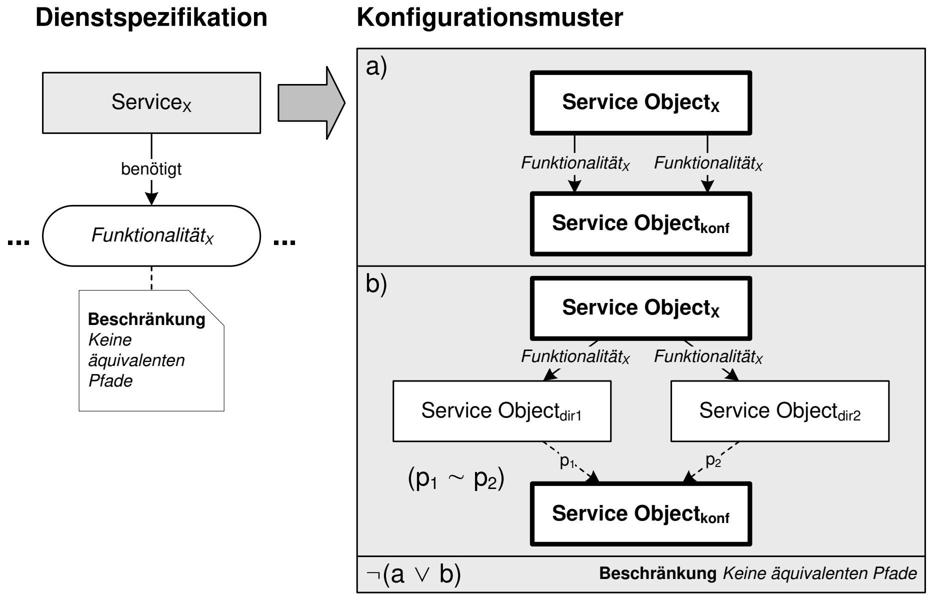


Abbildung 4.6: Bindungsbeschränkung zum Verhindern semantisch äquivalenter Bindungspfade.

dann jedoch um semantisch unterschiedliche Verbindungen. Semantisch äquivalente Verbindungen hingegen sind häufig nicht gewünscht. Die Bindungsbeschränkung gegen semantisch äquivalente Bindungen ermöglicht es daher, diesen Fall auszuschließen.

Abbildung 4.6 zeigt die graphische Darstellung der Bindungsbeschränkung gegen semantisch äquivalente Bindungen. Auf der linken Seite ist wieder die Spezifikation eines Dienstes Service$_X$ dargestellt, der die Funktionalität Funktionalität$_X$ benötigt. Hier ist für die Funktionalität die Bindungsbeschränkung Keine äquivalenten Pfade angegeben. Weitere Funktionalitäten werden hier wie auch im oben besprochenen Fall von der Betrachtung ausgeschlossen.

Die rechte Seite der Abbildung ist in verschiedene Bereiche unterteilt. Der obere Teil a) zeigt ein Konfigurationsmuster zur Verhinderung semantisch äquivalenter Bindungen. Hier ist das Service Object$_X$ dargestellt, ein Dienstobjekt des Dienstes Service$_X$ auf der linken Seite. Ein weiteres Dienstobjekt Service Object$_{konf}$ eines anderen Dienstes bietet die Funktionalität Funktionalität$_X$ an, die von Service$_X$ benötigt wird. Durch das Konfigurationsmuster im Teil a) der Abbildung wird ausgedrückt, dass gleich zwei Bindungen über die Funktionalität$_X$ zu einem einzelnen Dienstobjekt Service $\text{Object}_{\text{konf}}$ hergestellt werden. Eine solche Konfiguration soll verhindert werden. Falls es noch mehr als zwei semantisch äquivalente Bindungen gibt, so trifft das angegebene Muster auf zwei beliebige dieser Bindungen zu, wodurch die unerwünschte Situation identifiziert werden kann. Im Teil a) wurde zunächst die direkte Verbindung zwischen zwei Dienstobjekten betrachtet. Im darunter liegenden Teil b) wird der Fall betrachtet, dass zwei Dienstobjekte über einen Pfad verbunden sind, also über eine Kette von Dienstbindungen. Es werden zunächst einige Begriffe formal definiert, um das dargestellte Muster näher zu erläutern.

Definition 4.1 (Konfigurationsgraph)
Ein Konfigurationsgraph G ist gegeben durch $G = (V, E)$ mit einer endlichen Knotenmenge V von Dienstobjekten und Geräten und einer Bindungskantenmenge in Form einer Relation $E \subseteq V \times V \times F$, wobei F die Menge der definierten Funktionalitäten ist.

Es gilt zusätzlich:

- *Für ein Dienstobjekt $v \in V$ bildet die Funktion $s : V \to S$ auf den Dienst ab, von dem v eine Instanz ist, wobei die Menge S alle definierten Dienste enthält.*
- *Ein Knoten $v \in V$ ohne direkte Nachfolger, d. h. $\nexists v' \in V$ mit $(v, v', f) \in E$ und $f \in F$, wird Blatt genannt; alle anderen Knoten heißen innere Knoten des Konfigurationsgraphen.*

Der Konfigurationsgraph ist ein Teilgraph der Instanz des eHome-Modells, die zur Laufzeit des eHome-Systems angelegt wird. In diesem Teilgraphen sind als Knoten die Dienstobjekte und die Geräte enthalten. Die Knoten sind durch gerichtete Kanten verbunden, dies entspricht der Dienstkomposition. Die Geräte liegen an den Blättern dieser Graphstruktur. Die Kanten sind mit der jeweiligen zugrunde liegenden Funktionalität $f \in F$ gekennzeichnet. Zu jedem Dienstobjekt kann mittels der oben definierten Abbildung s der zugehörige Dienst ermittelt werden. Für die Ermittlung semantisch äquivalenter Bindungen müssen Pfade im Konfigurationsgraphen betrachtet werden. In Abbildung 4.6 sind Pfade durch gestrichelte Linien dargestellt.

Definition 4.2 (Konfigurationspfad)
Ein Konfigurationspfad von $u_0 \in V$ nach $v_{n-1} \in V$ in einem Konfigurationsgraphen $G = (V, E)$ ist eine endliche, nichtleere Folge von Kanten $p = ((u_0, v_0, f_0), \ldots, (u_{n-1}, v_{n-1}, f_{n-1}))$, wobei $n \in \mathbb{N}^+$, $(u_i, v_i, f_i) \in E$ für alle $i \in \{0, \ldots, n-1\}$ und $u_j = v_{j-1}$ für alle $j \in \{1, \ldots, n-1\}$. Die Länge des Pfades p ist n.

Nun müssen Konfigurationspfade noch auf semantische Äquivalenz geprüft werden. In Abbildung 4.6 ist die entsprechende Äquivalenzrelation zwischen den Konfigurationspfaden p_1 und p_2 durch das Symbol $\sim$ dargestellt. Die Definition der semantischen Äquivalenz bezogen auf Konfigurationspfade ist wie folgt.

Definition 4.3 (Semantische Äquivalenz von Konfigurationspfaden)
Sei $G = (V, E)$ ein Konfigurationsgraph und P die Menge der Konfigurationspfade von G. Die Äquivalenzrelation der semantisch äquivalenten Konfigurationspfade $R_\sim \subseteq P \times P$ auf P ist dann wie folgt definiert. Ein Element $(p_1, p_2) \in P \times P$ mit $p_1 = ((u_0, v_0, f_0), \ldots, (u_{n-1}, v_{n-1}, f_{n-1}))$ und $p_2 = ((u'_0, v'_0, f'_0), \ldots, (u'_{n-1}, v'_{n-1}, f'_{n-1}))$ liegt in der Äquivalenzrelation $R_\sim$ g. d. w.

$$s(u_i) = s(u'_i) \wedge s(v_i) = s(v'_i) \wedge f_i = f'_i \text{ für } i = \{0, \ldots, n-1\}$$

Zwei Pfade p_1 und p_2 eines Konfigurationsgraphen G heißen semantisch äquivalent g. d. w. $(p_1, p_2) \in R_\sim$, d. h.

$$p_1 \sim p_2 \Leftrightarrow (p_1, p_2) \in R_\sim$$

Gilt hingegen $p_1 \nsim p_2$ bzw. $(p_1, p_2) \notin R_\sim$ so heißen p_1 und p_2 semantisch verschieden.

In Abbildung 4.6 wird im Teil a) zunächst der triviale Fall semantisch äquivalenter Konfigurationspfade geprüft in dem der Pfad die Länge 1 hat. Wie oben bereits beschrieben, ist hier das Dienstobjekt Service Object_X mit dem Dienstobjekt Service $\text{Object}_\text{konf}$ eines anderen Dienstes gleich doppelt über die Funktionalität $\text{Funktionalität}_\text{X}$ angebunden. Dieses Muster gilt es zu verhindern.

Im Teil b) werden längere Pfade betrachtet, die eine Kette von Dienstbindungen repräsentieren. In diesem Fall sind zunächst zwei verschiedene Dienstobjekte Service $\text{Object}_\text{dir1}$ und Service $\text{Object}_\text{dir2}$ direkt an Service Object_X gebunden. Die

Bindung dieser integrierenden Dienste erfolgt in beiden Fällen über die Funktionalität Funktionalität_X. Von $\text{Service Object}_{dir1}$ aus gibt es nun einen Konfigurationspfad p_1 zu einem Dienstobjekt $\text{Service Object}_{konf}$. Von $\text{Service Object}_{dir2}$ aus gibt es ebenfalls einen Konfigurationspfad p_2 zum Dienstobjekt $\text{Service Object}_{konf}$. Sind nun p_1 und p_2 semantisch äquivalent, d. h. $p_1 \sim p_2$, so sind die beiden Pfade von Service Object_X zu $\text{Service Object}_{konf}$ insgesamt ebenfalls semantisch äquivalent. Dieser Fall muss daher ebenfalls ausgeschlossen werden.

Insgesamt wird daher gefordert, dass weder das Muster aus Teil a) noch das Muster aus Teil b) auftreten darf. Wie in Abbildung 4.6 ganz unten dargestellt, muss die Bedingung $\neg(a \vee b)$ erfüllt werden, damit keine semantisch äquivalenten Pfade vorliegen. Die Unterscheidung in zwei Teile ist hier für die deklarative Definition der Graphmuster erforderlich, da mehrere Indirektionsstufen durch integrierende Dienste vorliegen können. Die Überprüfung soll darüber hinaus nur stattfinden, wenn sie explizit in der jeweiligen Dienstspezifikation für eine benötigte Funktionalität gefordert wird. Zwar wäre es auch möglich, semantisch äquivalente Pfade in der Konfiguration pauschal auszuschließen, durch die Implementierung als Bindungsbeschränkung ergibt sich aber mehr Flexibilität. So muss eine Überprüfung z. B. nur stattfinden, wenn mehrere äquivalente Bindungen überhaupt möglich sind. Darüber hinaus gibt es auch Konstellationen, in denen semantisch äquivalente Pfade für eine sinnvolle Komposition benötigt werden. Durch die Realisierung als Bindungsbeschränkung kann diese Entscheidung vom Entwickler für jede Dienstspezifikation individuell getroffen werden.

In Abbildung 4.7 ist ein Beispiel für die Bindungsbeschränkung gegen semantisch äquivalente Pfade gezeigt. Hier wird ein Top-Level-Dienst Videoüberwachung betrachtet, ein Sicherheitsdienst, der Videoaufnahmen auswertet und aufzeichnet. Der Dienst benötigt dazu die Funktionalität Videosignal und zwar mindestens einmal, es können aber auch mehrere Videosignale gleichzeitig angebunden werden. Das spezifizierte Kardinalitätsintervall ist daher 1..n. Zusätzlich ist für die Funktionalität Videoüberwachung die Bindungsbeschränkung Keine äquivalenten Pfade angegeben. Im dargestellten Beispielszenario gibt es einen weiteren Dienst Kamerasteuerung. Dieser dient als Treiber und stellt die Funktionalitäten einer Überwachungskamera zur Verfügung. Die Kamerasteuerung bietet die Funktionalität Videosignal beliebig oft an, die angegebene Kardinalität ist daher n. Das bedeutet, dass das angebotene Videosignal von beliebig vielen anderen Diensten genutzt werden kann.

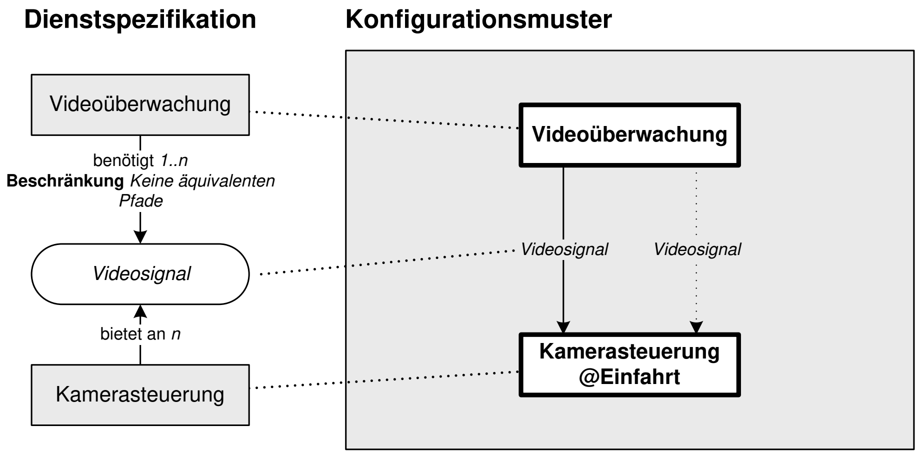


Abbildung 4.7: Beispiel einer Bindungsbeschränkung zum Verhindern semantisch äquivalenter Bindungspfade.

Auf der rechten Seite der Abbildung ist wieder ein zugehöriges Konfigurationsmuster dargestellt. Das Dienstobjekt Videoüberwachung wird über die Funktionalität Videosignal mit einem Dienstobjekt des Treibers Kamerasteuerung@Einfahrt verbunden, der eine Kamera an der Einfahrt des eHomes steuert. Da der Dienst Videoüberwachung mehrere Videosignale verarbeiten kann, z. B. von mehreren Kameras, und der Treiberdienst Kamerasteuerung sein Videosignal beliebig oft anbietet, wäre es nun denkbar, dass mehrere parallele Bindungen über die Funktionalität Videosignal zwischen beiden Dienstobjekten angelegt werden. Es könnten in diesem Fall sogar unbegrenzt viele Bindungen erzeugt werden, eine Begrenzung wäre nur durch Limitationen der Ausführungsplattform gegeben. Dieser Fall wird hier durch die Bindungsbeschränkung Keine äquivalenten Pfade auf einfache Weise verhindert. In der Abbildung wird dies durch die gestrichelt dargestellte zweite Bindung symbolisiert. Diese semantisch äquivalente Bindung kann in diesem Szenario nicht angelegt werden.

In Abbildung 4.8 wird ein weiteres Beispiel zur Bindungsbeschränkung gegen semantisch äquivalente Pfade gezeigt. Hier ist ein Top-Level-Dienst Komfortlicht spezifiziert, der eine individuelle Beleuchtungssituation entsprechend den Wünschen einer zugeordneten Person herstellt. Dazu wird die Funktionalität Beleuchtung mindestens einmal benötigt. Für diese Funktionalität sind hier keine Bindungsbeschränkungen spezifiziert. Die Funktionalität Beleuchtung wird von einem Treiberdienst Lampensteuerung sechs mal angeboten, da dieser Treiber-

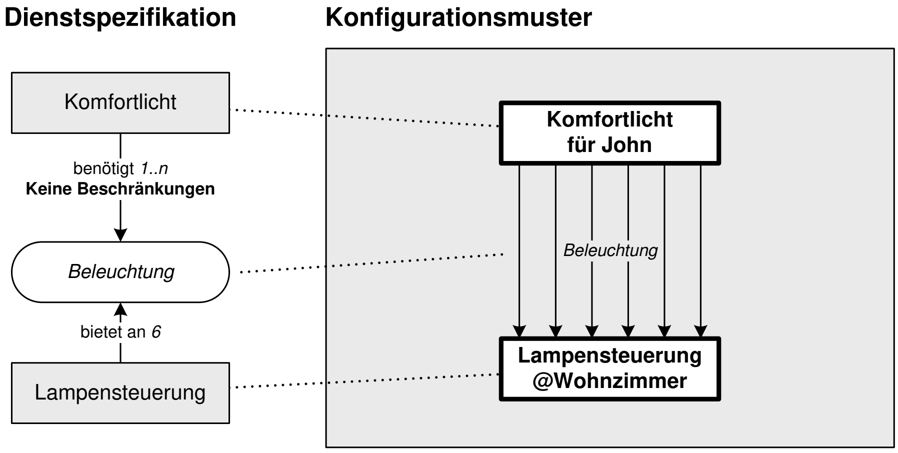


Abbildung 4.8: Beispiel ohne Bindungsbeschränkungen mit erwünschten semantisch äquivalenten Bindungspfaden.

dienst einen speziellen Typ von Lampen ansteuert, der aus sechs einzelnen aber gleichartigen Leuchtkörpern besteht, die über den Treiberdienst auch individuell angesteuert werden können. Im Konfigurationsmuster auf der rechten Seite ist nun eine Situation dargestellt, in der ein Dienstobjekt Komfortlicht für John erzeugt wurde und ein Dienstobjekt Lampensteuerung@Wohnzimmer zur Steuerung einer Lampe im Wohnzimmer, dem aktuellen Aufenthaltsort von John. Da in der Spezifikation des Top-Level-Dienstes keine Bindungsbeschränkungen angegeben sind, können in diesem Fall sechs semantisch äquivalente Bindungen über die Funktionalität Beleuchtung zwischen den beiden Dienstobjekten hergestellt werden. Dies ist gewünscht, da der Dienst für das Komfortlicht auf alle Leuchtkörper der Lampe Zugriff haben soll, um so die Beleuchtungssituation im Raum genau steuern zu können. Hier liegt also eine Situation vor, in der aufgrund der Art und der Funktionalität der Dienste semantisch äquivalente Bindungen erforderlich sind und daher nicht ausgeschlossen werden dürfen.

Beschränkung gegen Konfigurationszyklen

In diesem Abschnitt wird eine weitere Bindungsbeschränkung betrachtet, die dazu dient, unerwünschte Konfigurationen auszuschließen. Hier wird jedoch nicht Bezug auf die Kardinalitäten der angebotenen und benötigten Funktionalitäten genommen, sondern es geht um bestimmte Konfigurationsmuster, nämlich zyklische Bindungen. Konfigurationszyklen können durch die Verwendung integrierender Dienste auftreten. Mittels integrierender Dienste können mehrere Abstraktionsschichten gebildet werden, um zwischen Top-Level-Diensten und Treiberdiensten zu vermitteln. Durch integrierende Dienste können jedoch auch Probleme bei der Konfigurierung auftreten, wenn eine angebotene Funktionalität eines integrierenden Dienstes als benötigte Funktionalität in der Spezifikation eines anderen integrierenden Dienstes auftritt. Man könnte vermuten, dass dieser Fall nicht auftreten kann, da ja zwischen verschiedenen Abstraktionsstufen vermittelt wird und die Funktionalitäten daher immer verschieden sein müssten. Dies ist jedoch nicht immer der Fall, da integrierende Dienste z. B. Signale verarbeiten können und sich dabei die Art des Signals nicht zwangsläufig ändert. Entsprechend kann dabei auch die Funktionalität die gleiche bleiben. Dies wäre etwa der Fall, wenn ein Bildverarbeitungsdienst ein Videosignal benötigt (als Eingabe) und ebenfalls ein Videosignal anbietet (als Ausgabe). In diesem Fall könnte eine beliebig lange Kette konfiguriert werden, ohne dass ein Gerät angebunden wird. Es würde so keine gültige Konfiguration entstehen. Eine feste Beschränkung der Länge von Konfigurationspfaden wäre willkürlich und würde viele benötigte und sinnvolle Konfigurationen von vornherein ausschließen. Daher wurde die Bindungsbeschränkung gegen Konfigurationszyklen eingeführt. Dabei ist ein Konfigurationszyklus wie folgt definiert.

Definition 4.4 (Konfigurationszyklus)
Ein Konfigurationspfad p von u_0 nach v_{n-1} heißt Zyklus g. d. w.

$$s(v_{n-1}) = s(u_0)$$

Abbildung 4.9 zeigt die deklarative graphische Beschreibung der Bindungsbeschränkung gegen Konfigurationszyklen. Der Dienst Service_X auf der linken Seite benötigt die Funktionalität Funktionalität_X. Für diese Funktionalität ist die Bindungsbeschränkung Keine Bindungszyklen angegeben. Weitere Funktionalitäten werden wieder von der Betrachtung ausgeschlossen.

Auf der rechten Seite werden erneut verschiedene Bereiche unterschieden. Der Teil a) zeigt den Fall, dass ein Dienstobjekt Service Object_X die Funktionalität_X benötigt, die von einem anderen Dienstobjekt $\text{Service Object}_{int}$ angeboten wird. Gleichzeitig ist $\text{Service Object}_{int}$ jedoch auch an Service Object_X gebunden und

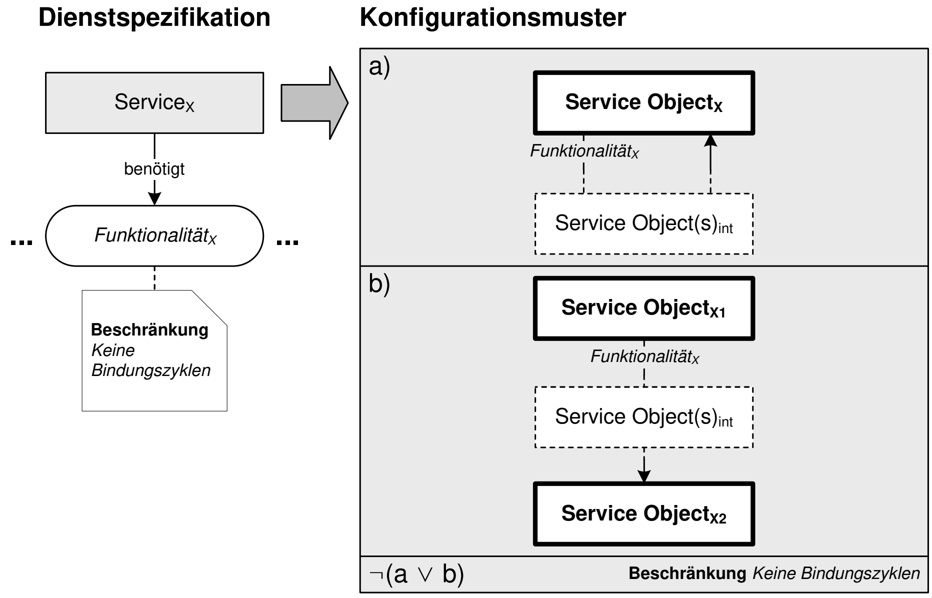


Abbildung 4.9: Bindungsbeschränkung zum Verhindern zyklischer Bindungen.

nutzt eine Funktionalität dieses Dienstobjekts. Dabei kann es sich um Funktionalität$_X$ oder auch eine beliebige andere Funktionalität handeln. Die gestrichelte Darstellung von Service Object$_{int}$ deutet an, dass es sich hierbei auch um eine Kette von mehreren Dienstobjekten handeln kann. Es ergibt sich also über einen oder mehrere Dienste ein Zyklus im Konfigurationsgraphen. Die Bindungsbeschränkung Keine Bindungszyklen dient dazu, dieses Muster auszuschließen.

Teil b) zeigt einen abgewandelten Fall, in dem sich ein Zyklus nicht auf ein einzelnes Dienstobjekt bezieht. Hier wird stattdessen der zugehörige Dienst betrachtet. Es besteht ein Pfad zwischen Service Object$_{X1}$ und Service Object$_{X2}$ über die Funktionalität$_X$. Da Service Object$_{X1}$ und Service Object$_{X2}$ Instanzen desselben Dienstes sind, liegt hier gemäß Definition 4.4 ein Konfigurationszyklus vor. Der in Teil a) dargestellte Fall wird durch das Muster in Teil b) ebenfalls abgedeckt. Der Sonderfall, dass nicht nur der Dienst, sondern auch die Dienstobjekte identisch sind, wird jedoch des besseren Verständnisses wegen in Teil a) explizit modelliert. Da die Bindungsbeschränkung die dargestellten Konfigurationsmuster ausschließen soll, muss also insgesamt die Bedingung $\neg(a \vee b)$ erfüllt werden.

In Abbildung 4.10 ist ein Beispiel für die Bindungsbeschränkung gegen Konfigurationszyklen dargestellt. Hier werden zwei Dienste betrachtet. Der eine dient

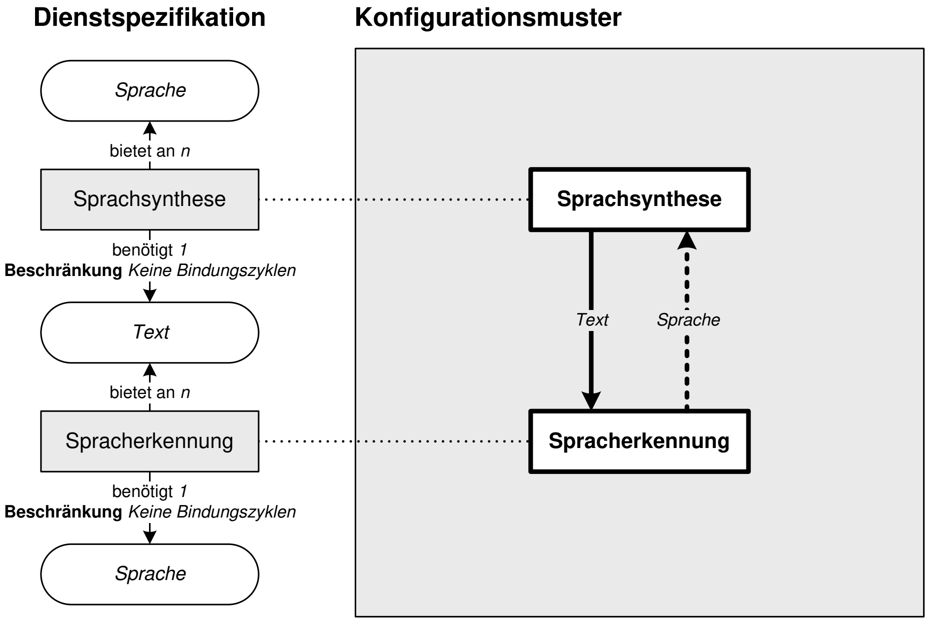


Abbildung 4.10: Beispiel einer Bindungsbeschränkung gegen Konfigurationszyklen.

zur Sprachsynthese und benötigt Text als Eingabe und bietet Sprache, die aus dem eingelesenen Text synthetisiert wird, als Ausgabe an. Ein weiterer Dienst ermöglicht eine Transformation in genau der entgegengesetzten Richtung. Der Dienst Spracherkennung benötigt Sprache als Eingabe und bietet Text als Ausgabe an, es wird also der gesprochene Text erkannt und ausgegeben. Für beide Dienste wird spezifiziert, dass Bindungszyklen ausgeschlossen werden sollen.

Auf der rechten Seite der Abbildung ist ein Konfigurationsmuster für dieses Szenario angegeben. Ein Dienstobjekt Sprachsynthese verarbeitet hier den Text, der von einem Dienstobjekt Spracherkennung erzeugt wird. Dieses verarbeitet wiederum Sprache, die vom eingangs betrachteten Dienstobjekt Sprachsynthese erzeugt wurde. Dieser Kreislauf ließe sich durch weitere Dienstobjekte beliebig vergrößern. In jedem Fall wird der entstehende Konfigurationszyklus mit Hilfe der Bindungsbeschränkung ausgeschlossen. Eine gültige Konfiguration könnte so nicht entstehen, da keine Geräte oder Top-Level-Dienste involviert sind. Eine Verkettung der angegebenen Art würde, selbst wenn sich an den Enden des Pfades ein Top-Level-Dienst bzw. ein Gerät befindet, beliebig ineffizient werden. Die Bindungsbeschränkung gegen Konfigurationszyklen bietet daher die Möglichkeit, Ketten mit zyklischen Wiederholungen auszuschließen.

Weitere Bindungsbeschränkungen

Über die bisher beschriebenen Szenarien hinaus bietet das Konzept der Bindungsbeschränkungen viele Möglichkeiten der Erweiterung, um die Dienstspezifikation speziellen Anforderungen anzupassen und die spätere Konfigurierung im eHome zu beeinflussen. Für kontextbezogene Anwendungen kann es durchaus sinnvoll sein, die Ressourcenbindung nicht auf Ressourcen zu beschränken, die sich in genau demjenigen Umgebungselement wie der Dienst befinden. Eine schwächere Anforderung, wie etwa die Einschränkung auf Ressourcen in benachbarten Räumen, könnte hier sinnvoll eingesetzt werden. Auch eine Beschränkung auf die Bindung einer am nächsten gelegenen passenden Ressource ist in verschiedenen Fällen nützlich.

Eine andere mögliche Erweiterung wären zeitlich parametrisierte Bindungsbeschränkungen. In-Home-Mobilität bedeutet, wie im Teil über die Dynamik in eHomes beschrieben, dass sich die Benutzer des eHome-Systems in ihrer Umgebung umher bewegen. Daher muss bei personenbezogenen Diensten unter Umständen häufig rekonfiguriert werden. Eine zeitlich parametrisierte Bindungsbeschränkung auf Basis der Bindungsbeschränkung auf lokale Ressourcen könnte dafür nützlich sein. Sie könnte etwa dazu dienen, erst nach Ablauf einer durch einen Schwellwert angegebenen Zeitspanne die Rekonfigurierung auszulösen. Wenn z. B. ein Benutzer, der einen Musikdienst verwendet, einen Raum verlässt und bereits nach wenigen Sekunden in diesen Raum zurückkehrt, könnte eine Rekonfigurierung entfallen. Dadurch würde sich der Aufwand für die Rekonfigurierung reduzieren und das Verhalten des eHome-Systems wäre auch aus Sicht des Benutzers angenehmer, da sich keine sprunghafte Änderung der von ihm genutzten Funktionalitäten durch die sehr kurzfristige Rekonfigurierung der Musikwiedergabe ergeben würde.

Auch für nicht kontextbezogene Bindungsbeschränkungen gibt es Möglichkeiten der Erweiterung. Für bestimmte Fälle könnte es etwa nützlich sein, dass für eine benötigte Funktionalität mit einem Kardinalitätsintervall größer als eins jede Bindung ein anderes Dienstobjekt binden muss. Die angebotenen Funktionalitäten müssen dabei natürlich der Anforderung entsprechen und die Dienstobjekte könnten auch Instanzen desselben Dienstes sein, es dürfte sich nur nicht um die identische Instanz handeln. Dies könnte durch eine Erweiterung der Bindungsbeschränkung gegen semantisch äquivalente Pfade erreicht werden.

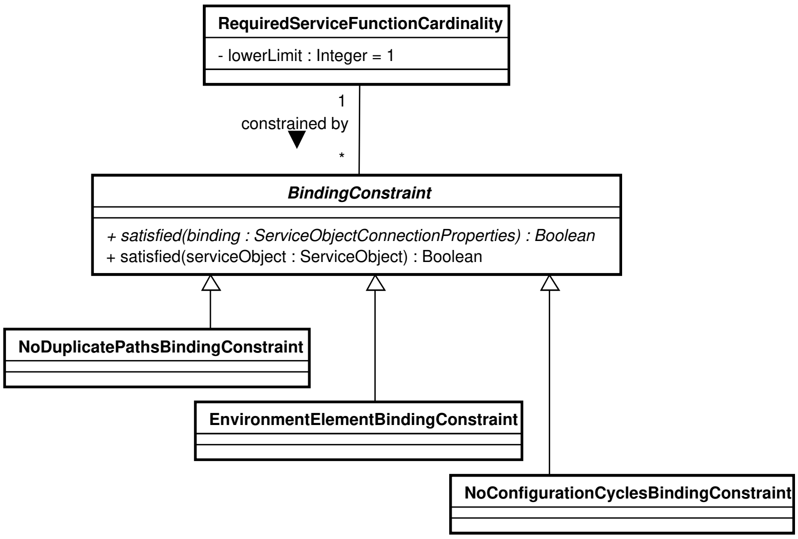


Abbildung 4.11: Modellierung der Bindungsbeschränkungen.

Durch die deklarative Beschreibung der Bindungsbeschränkungen in Form von Graphmustern, ist festgelegt, wie die Bedingungen bei der Dienstkomposition überprüft werden können. Da die Konzepte im Rahmen dieser Arbeit graphbasiert umgesetzt werden, kann die graphische Beschreibung der Muster unmittelbar Anwendung finden und in die ebenfalls graphbasierte Implementierung der Laufzeitumgebung übertragen werden. Die globale Sicht des eHome-Modells erweist sich für die Entwicklung der Bindungsbeschränkungen als vorteilhaft, da auf alle Elemente des Modells in einfacher Weise zugegriffen werden kann, z. B. auf Kontextelemente, die bei der Konfigurierung berücksichtigt werden sollen. Die graphbasierte Modellierung der Bindungsbeschränkungen ist übersichtlicher und einfacher zu handhaben als eine direkte Implementierung in Java.

Modellierung

Abbildung 4.11 zeigt, wie die Bindungsbeschränkungen modelliert werden. Bindungsbeschränkungen können für jede benötigte Funktionalität eines Dienstes individuell festgelegt werden. Daher ist die für Bindungsbeschränkungen zuständige Klasse BindingConstraint mit der Klasse RequiredServiceFunctionCardinality assoziiert, die die Verbindungen zwischen Diensten und ihren benötigten Funktionalitäten modelliert (vgl. auch Abbildung 4.3). Jeder benötigten Funktionalität können beliebig viele Bindungsbeschränkungen zugeordnet werden. Die drei oben vorgestellten Bindungsbeschränkungen sind durch die Unterklassen NoDuplicatePathsBindingConstraint, EnvironmentElementBindingConstraint und NoConfigurationCyclesBindingConstraint realisiert.

Weitere Beschränkungen können einfach ergänzt werden, sie müssen dann ebenfalls von der Klasse BindingConstraint abgeleitet sein. Die abstrakte satisfied-Methode der Oberklasse BindingConstraint wird in den Unterklassen der jeweiligen Bindungsbeschränkungen implementiert. Sie überprüft, ob die zugehörige Bindungsbeschränkung für eine bestimmte Bindung erfüllt ist. Die konkrete, namensgleiche satisfied-Methode dient dazu, alle Bindungsbeschränkungen eines Dienstobjekts gemeinsam zu überprüfen. Nur wenn alle diese Bindungsbeschränkungen erfüllt sind, wird true zurückgeliefert.

4.3.3 Bindungsstrategien

Der erweiterte Entwicklungsprozess sieht eine Laufzeitunterstützung zur strukturellen Adaption vor. Diese muss, wie oben bereits erwähnt wurde, im Allgemeinen automatisch durchführbar sein, da der Benutzer nicht für die vielen feingranularen Adaptionsschritte während der Laufzeit zur Interaktion gezwungen werden kann. Damit ein automatisierter Ablauf möglich ist, muss der Dienstentwickler bereits zur Entwicklungszeit Vorgaben in der Dienstspezifikation machen, die festlegen, wie die spätere Dienstkomposition ablaufen soll. Die Dienstspezifikation wurde daher in dieser Arbeit um die Festlegung von *Bindungsstrategien* (engl. *Binding Policies*) erweitert.

Mit der Festlegung einer Bindungsstrategie in der Dienstspezifikation ist es dem Entwickler möglich, Einfluss auf das Verhalten der Laufzeitumgebung zu nehmen. Diese führt die Dienstkomposition später entsprechend der spezifizierten Strategie durch. Bindungen werden im Rahmen dieser Arbeit grundsätzlich als dynamisch und damit zur Laufzeit änderbar betrachtet. Wann eine Bindung angelegt oder geändert wird und wie viele Bindungen hergestellt werden, hängt neben den bereits beschriebenen Bindungsbeschränkungen auch von der Bindungsstrategie ab. Die Konfigurierung des eHome-Systems kann daher auch in diesem Punkt feingranular, auf der Ebene der Dienste und deren Funktionalitäten, beeinflusst werden.

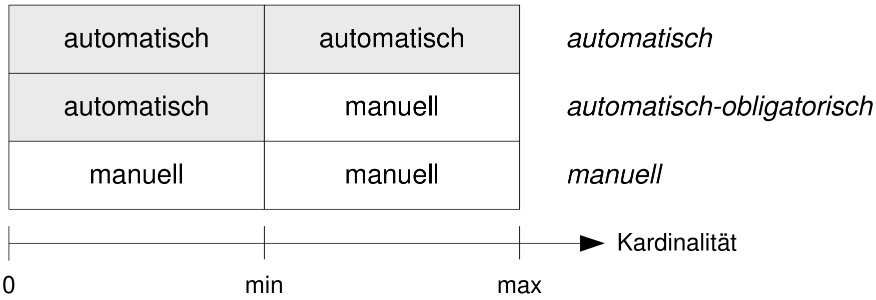


Abbildung 4.12: Festlegung des Bindungsverhaltens.

In dieser Arbeit wird zwischen drei Bindungsstrategien unterschieden, die sich in verschiedenen Kriterien unterscheiden. Zunächst wird zwischen einer automatischen und einer manuellen Vorgehensweise unterschieden. Darüber hinaus sind die Strategien von den Kardinalitätsgrenzen der Funktionalitäten in der Dienstspezifikation abhängig. Dies ist in Abbildung 4.12 veranschaulicht. Die Achse am unteren Rand der Abbildung zeigt das Kardinalitätsintervall min..max einer benötigten Dienstfunktionalität. Darüber wird für jede Strategie angegeben, in welchem Intervall automatisch und in welchem manuell gebunden wird. Nachfolgend wird auf die Bindungsstrategien im Einzelnen eingegangen.

Automatische Bindungsstrategie

Die *automatische Bindungsstrategie* legt fest, dass alle Bindungen für eine benötigte Funktionalität von der Laufzeitumgebung automatisch verwaltet werden sollen. Damit wird das Anlegen, Ändern und Entfernen von Bindungen im gesamten Kardinalitätsintervall 0..max abgedeckt. Diese Bindungsstrategie führt dazu, dass automatisch so viele Bindungen wie möglich, d. h. bis max, über die jeweilige Funktionalität gebunden werden. Es wird also das Maximum an verfügbaren Ressourcen für diese Funktionalität zur Verfügung gestellt. In Abbildung 4.12 ist dies in der obersten Zeile dargestellt. Falls eine automatisch verwaltete Bindung vom Benutzer manuell gelöscht wird, so wird sie bei dieser Bindungsstrategie falls möglich sofort wieder hergestellt. Es können manuell keine zusätzlichen Bindungen erstellt werden, da diese, sobald sie möglich sind, ohnehin automatisch angelegt werden. Nichtsdestotrotz kann der Benutzer Anpassungen vornehmen, z. B. kann die Bindung einer anderen Ressource als Alternative zu einer bereits automatisch erstellten Bindung ausgewählt werden. Dies führt zu einer Rekonfigurierung in der die entsprechende Ressource dem Benutzerwunsch gemäß ausgetauscht wird.

Automatisch-obligatorische Bindungsstrategie

Die *automatisch-obligatorische Bindungsstrategie* sorgt dafür, dass nur die mindestens erforderlichen Ressourcen an eine Dienstfunktionalität gebunden werden. Es werden also nur die unbedingt benötigten Ressourcen belegt, damit der Dienst ausgeführt werden kann. Bezogen auf das spezifizierte Kardinalitätsintervall bedeutet dies, dass bis zur Untergrenze min automatisch Bindungen angelegt werden. Weitere Bindungen werden nicht angelegt, der Benutzer hat jedoch die Möglichkeit, manuell weitere Bindungen bis zur Obergrenze max hinzuzufügen, zu entfernen oder zu ändern. Wie auch bei der automatischen Bindungsstrategie ist es hier möglich, automatisch angelegte Bindungen manuell anzupassen und so Alternativen für gebundene Ressourcen zu wählen.

Manuelle Bindungsstrategie

Bei der *manuellen Bindungsstrategie* werden keine automatischen Bindungen hergestellt. Die Bindungen werden ausschließlich vom Benutzer manuell angelegt. Dazu steht ein interaktives Werkzeug zur Verfügung, das die manuelle Verwaltung von Bindungen ermöglicht. Selbst wenn alle benötigten Ressourcen eines Dienstes zur Verfügung stehen, ist der Dienst zunächst nicht gültig konfiguriert, es sei denn, alle benötigten Funktionalitäten sind optional. Damit der Dienst ausgeführt werden kann, muss der Benutzer zunächst manuell Bindungen anlegen bis die Mindestanforderungen erfüllt werden.

Modellierung

Abbildung 4.13 zeigt die Modellierung der Bindungsstrategien, die ähnlich zur Modellierung der Bindungsbeschränkungen ist. Die abstrakte Klasse BindingPolicy ist ebenso wie BindingConstraint mit RequiredServiceFunctionCardinality assoziiert. Daher kann auch die Bindungsstrategie für jede benötigte Funktionali-

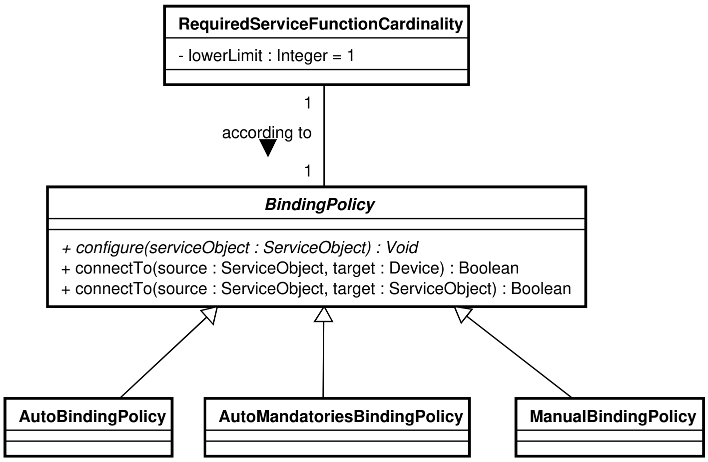


Abbildung 4.13: Modellierung der Bindungsstrategien.

tät eines Dienstes individuell festgelegt werden. Anders als bei den Bindungsbeschränkungen muss jedoch immer genau eine Strategie für jede benötigte Funktionalität festgelegt werden, nur dann ist eine Konfigurierung möglich. Die abstrakte Methode configure der Klasse BindingPolicy wird von den konkreten Klassen der drei Bindungsstrategien AutoBindingPolicy, AutoMandatoriesBindingPolicy und ManualBindingPolicy implementiert. Sie ist für die eigentliche Konfigurierung eines Dienstobjekts gemäß der jeweiligen Bindungsstrategie zuständig. Die connectTo-Methoden ermöglichen das manuelle Modifizieren einer Konfiguration. So kann eine gewünschte Verbindung zwischen einem Dienstobjekt und einem bestimmten anderen Dienstobjekt oder einem bestimmten Gerät spezifiziert werden. Dann wird versucht, die entsprechende Bindung herzustellen, was voraussetzt, dass passende Funktionalitäten und ggfs. passende integrierende Dienste verfügbar sind und dass die Bindungsbeschränkungen erfüllt werden.

4.3.4 Bindungstypen

Die Entwicklung von eHomes hat das Ziel einer Entkopplung von Hardware und Software. Dadurch werden die bereits beschriebenen Vorteile einer Kostenreduzierung durch Wiederverwendung und einer Flexibilisierung zur Unterstützung von Mobilität und Adaptivität erreicht. Ein Problem, das daraus resultiert, ist jedoch, dass nicht mehr automatisch jeder Funktionalität die benötigten Ressourcen gegenüberstehen, da die Funktionalität in Software realisiert und nicht mehr mit der benötigten Hardware fest verdrahtet ist [RK09]. So kann es leicht vorkommen, dass nicht alle benötigten Ressourcen jederzeit verfügbar sind. Insbesondere können, wie oben beschrieben, Veränderungen zur Laufzeit eintreten, sodass eine benötigte Ressource nicht länger verfügbar ist. Es ist jedoch nicht wünschenswert, dass eine vom Benutzer ausgewählte Funktionalität immer nur zeitweise angeboten wird und sich die Verfügbarkeit aus externen, vom Benutzer nicht überschaubaren Faktoren ergibt. Es muss also möglich sein, festzulegen, wann eine bestimmte Funktionalität nutzbar ist, wann sie nicht nutzbar ist und welche Funktionalitäten im Konfliktfall vorrangig zu behandeln sind.

In eHomes können sich unter Umständen viele verschiedene Benutzer gleichzeitig aufhalten, die jeweils eine Vielzahl personenbezogener Dienste nutzen. Durch Bindungsbeschränkungen kann spezifiziert werden, dass nur lokale Ressourcen für bestimmte Funktionalitäten in Betracht kommen. Somit ist die Zahl der nutzbaren Ressourcen noch weiter eingeschränkt. Durch die Entkopplung von Hardware und Software bei der Umsetzung von Diensten kann also eine Diskrepanz zwischen benötigten und verfügbaren Ressourcen entstehen. Das eHome-System muss diesem Umstand gerecht werden und Möglichkeiten bieten, das Problem zu entschärfen.

Im Rahmen der vorliegenden Arbeit wurde daher ein neues Konzept eingeführt, das die geteilte Nutzung von Ressourcen unterstützt. Auf diese Weise wird die Möglichkeit geschaffen, dass sich mehrere Dienste Ressourcen teilen, die sie nicht durchgehend benötigen.

Es gibt viele Beispiele für solche Ressourcen, die nur zeitweise genutzt werden. Ein Klingeldienst benötigt z. B. eine Ressource zur Ausgabe eines akustischen Signals. Diese wird jedoch nur in dem Moment benötigt, in dem die Klingel betätigt wird. Ein Telefondienst benötigt Ressourcen zur Tonaufnahme und -wiedergabe, ggfs. auch zur Videoaufnahme und -wiedergabe. Auch diese Ressourcen werden nur benötigt, solange ein Telefongespräch stattfindet. Ein Alarmdienst muss ein Warnsignal ausgeben können, jedoch nur in dem Fall, dass eine entsprechende Notsituation eingetreten ist. Eine solche Situation ist jedoch der Ausnahmefall. Auch in den anderen Fällen müssen die Dienste lauffähig sein und müssen daher Ressourcen binden, die ihre benötigten Funktionalitäten zur Verfügung stellen. Die tatsächliche Dauer, in der diese Ressourcen genutzt werden, stellt jedoch nur einen Bruchteil der Laufzeit des Dienstes dar. Es ist daher für diese Dienste sinnvoll, die Ressourcen geteilt zu nutzen und im Fall eines Nutzungskonflikts eine Priorisierung vorzunehmen.

Damit eine geteilte Nutzung von Ressourcen durch das eHome-System unterstützt werden kann, wurde in dieser Arbeit eine weitere Ergänzung der Dienstspezifikation eingeführt, die Unterscheidung zwischen verschiedenen *Bindungstypen*. Dadurch kann bereits bei der Dienstentwicklung festgelegt werden, ob ein Dienst eine bestimmte Funktionalität dauerhaft verwendet oder ob eine geteilte Nutzung der Funktionalität möglich ist. Entsprechend dieser Unterscheidung wird zischen *exklusiven Bindungen* und *nebenläufigen Bindungen* unterschieden. In der Spezifikation eines Dienstes kann für jede benötigte Funktionalität individuell der Bindungstyp festgelegt werden. Davon hängt die spätere Verwaltung der Bindungen zur Laufzeit ab.

Exklusive Bindungen

Für benötigte Funktionalitäten kann eine *exklusive Bindung* gefordert werden. Exklusive Bindungen bestehen während der gesamten Laufzeit des zugehörigen Dienstobjekts. Im Vergleich zum Ansatz von NORBISRATH werden auch diese Bindungen zur Laufzeit angelegt und wieder gelöscht, dies ergibt sich aus der Dynamik, die im erweiterten Ansatz unterstützt wird. Allerdings belegt eine exklusive Bindung die gebundenen Ressourcen dauerhaft. Wird die Funktionalität mehrfach angeboten, so kann diese zwar von mehreren Diensten genutzt werde, es ist jedoch nicht möglich eine geteilte Nutzung einer einzeln angebotenen Funktionalität in disjunkten Zeitintervallen zu ermöglichen.

Dieser Typ von Bindungen kann eingesetzt werden, wenn der nutzende Dienst die Ressource in häufigen und kurzen Intervallen benötigt, sodass es nicht sinnvoll wäre, die Ressource in der verbleibenden Zeit anderweitig zu nutzen. Auch bei Diensten von besonderer Bedeutung kann es wichtig sein, dass bestimmte Ressourcen dauerhaft gebunden werden und nicht temporär durch andere Dienste belegt werden können. Für Dienste, die nicht personenbezogen sind und keiner anderweitigen Dynamik unterliegen, kann so eine dauerhafte Zuordnung zwischen dem Softwareanteil und dem Hardwareanteil des Systems realisiert werden.

Nebenläufige Bindungen

In vielen Fällen ist ein höheres Maß an Dynamik erwünscht, als dies durch exklusive Bindungen abgebildet werden kann. Wird eine Dienstfunktionalität etwa nur für einen Nutzer gleichzeitig angeboten, d. h. die Kardinalität ist gleich eins, so kann diese Funktionalität während der gesamten Laufzeit des nutzenden Dienstobjekts nicht anderweitig verwendet werden. Wenn die Ressource aber gar nicht dauerhaft benötigt wird, so kann eine deutlich effizientere Nutzung erreicht werden. Dazu wurde im Rahmen dieser Arbeit als neuer Bindungstyp die *nebenläufige Bindung* eingeführt. Bei diesem Bindungstyp wird zwischen der potentiellen Nutzung und der konkreten Nutzung unterschieden. Die gebundene Ressource wird nutzungsabhängig blockiert, nicht für die gesamte Lebensdauer der Bindung. Wird die Ressource konkret genutzt, so ist sie während dieser Zeit für andere Dienste nicht verfügbar. Wird die Ressource momentan nicht konkret genutzt, so bleibt die Bindung erhalten, die Ressource kann aber trotzdem durch andere Dienste genutzt werden. Auch wenn eine Funktionalität nur einmalig zur Verfügung steht, können mehrere nebenläufige Bindungen gleichzeitig diese Funktionalität binden und damit potentiell nutzen. Es kann aber immer nur ein Dienst zur gleichen Zeit die Ressource konkret nutzen. Auf diese Weise ist die geteilte Nutzung von Ressourcen durch mehrere Dienste möglich.

Da die Nutzung geteilter Ressourcen nur in zeitlich disjunkten Intervallen möglich ist, müssen diese zunächst identifiziert werden. Dazu wurde ein Session-Konzept eingeführt. Eine *Session* definiert ein Nutzungsintervall einer Bindung durch ein Dienstobjekt. Eine Bindung ist daher *aktiv*, wenn gerade eine Session über diese Bindung vorhanden ist, anderenfalls ist sie *inaktiv*. Ist die Funktionalität einer Ressource durch mehrere nebenläufige Bindungen gebunden, so kann immer nur eine dieser Bindungen gleichzeitig aktiv sein.

Abbildung 4.14 zeigt ein Beispiel für eine Konfiguration mit nebenläufigen Bindungen. Es sind drei Dienstobjekte, Musikdienst, Alarmdienst und Weckdienst dargestellt. Es sind drei Lautsprecher als Ressourcen verfügbar, die von allen drei Dienstobjekten benötigt werden. Der Musikdienst und der Alarmdienst sind über nebenläufige Bindungen mit allen drei Lautsprechern verbunden. Der Weckdienst ist nur mit dem dritten Lautsprecher verbunden, da er nur einen Lautsprecher benötigt. Die benötigten Funktionalitäten aller drei Dienstobjekte sind gebunden, sodass sie gestartet werden können. Von den genannten Bindungen

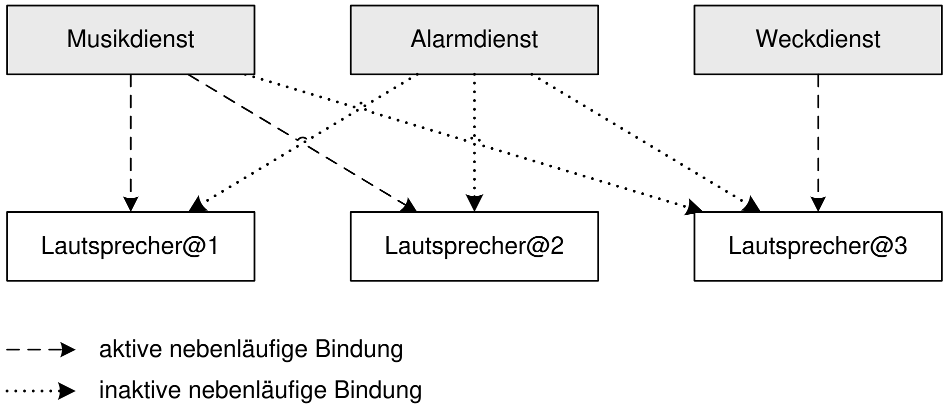


Abbildung 4.14: Beispielkonfiguration mit nebenläufigen Bindungen.

sind aber nur einige aktiv, abhängig vom momentanen Laufzeitzustand des Systems. Der Musikdienst ist im Beispiel eingeschaltet und spielt Musik ab. Dazu benötigt er die Funktionalität der Lautsprecher. Entsprechend sind Sessions zu den ersten beiden Lautsprechern aktiv. Der dritte Lautsprecher wird durch den Weckdienst genutzt, da dieser gerade ein Wecksignal ausgibt. Daher kann der dritte Lautsprecher momentan nicht durch den Musikdienst verwendet werden. Der Alarmdienst ist in Betrieb, benutzt aber derzeit keine Lautsprecher, da kein Alarmsignal ausgegeben wird. Nur wenn der Alarmdienst ein Alarmsignal ausgibt, werden die Lautsprecher benötigt.

Die Verwendung nebenläufiger Bindungen ermöglicht die geteilte Nutzung der Lautsprecher durch die drei Dienstobjekte. Ohne nebenläufige Bindungen wäre es beispielsweise nicht möglich, den Alarmdienst zu verwenden, wenn bereits der Musikdienst alle Lautsprecher belegt. Wenn umgekehrt der Alarmdienst verwendet wird und dieser alle Lautsprecher bindet, um im Notfall überall im eHome ein Alarmsignal zu geben, so könnte der Musikdienst oder der Weckdienst nicht zusätzlich im eHome verwendet werden, auch wenn der Alarmfall nur äußerst selten oder nie eintritt.

Priorisierung

Die Einführung nebenläufiger Bindungen und des Session-Konzepts allein reicht jedoch nicht aus, um die geteilte Nutzung von Ressourcen zu ermöglichen. In der

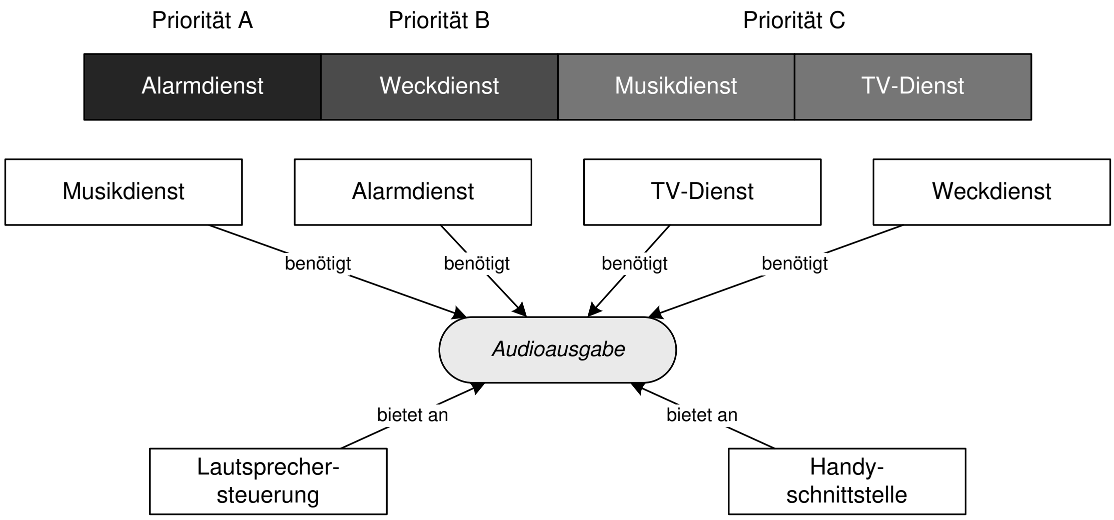


Abbildung 4.15: Beispiel der Priorisierung auf Basis von Funktionalitäten.

dargestellten Situation könnte der Alarmdienst keinen Lautsprecher benutzen, da alle drei bereits durch andere Dienstobjekte genutzt werden. Der Alarmdienst hat aber eine besondere Bedeutung und ist möglicherweise für die Sicherheit im eHome unerlässlich. Aus diesem Grund muss diesem Dienst auch eine höhere Priorität eingeräumt werden können. Es ist also zusätzlich ein Konzept zur Priorisierung erforderlich. Im Beispiel aus Abbildung 4.14 wäre eine denkbare Priorisierung etwa, dem Alarmdienst die höchste, dem Weckdienst eine mittlere und dem Musikdienst die niedrigste Priorität zuzuweisen.

Allgemein kann die Priorisierung jedoch nicht bereits zur Entwicklungszeit festgelegt werden. Es gibt viele Fälle, in denen die individuellen Vorlieben des Benutzers die Priorisierung der Dienste bedingen. Diese Festlegung kann daher nicht zur Entwicklungszeit pauschal vorweggenommen werden. Darüber hinaus ist die Zuweisung von Prioritätswerten auch von der Zusammenstellung von Diensten im spezifischen eHome abhängig. Daher ist eine sinnvolle Priorisierung erst zur Laufzeit des eHome-Systems möglich, wenn die Zusammenstellung der Dienste und die Benutzerwünsche bekannt sind und in die Festlegung der Prioritäten einfließen können.

In Abbildung 4.15 ist ein Beispiel für die Priorisierung dargestellt. Neben den Diensten im Beispiel aus Abbildung 4.14, dem Musikdienst, dem Alarmdienst und dem Weckdienst, ist hier noch ein TV-Dienst enthalten. Alle vier Dienste benötigen die Funktionalität Audioausgabe, die von den Diensten Lautsprechersteuerung und Handyschnittstelle angeboten wird. Im oberen Teil der Abbildung ist die Priorisierung für die Funktionalität Audioausgabe dargestellt. Es wird eine Einteilung aller Dienste, die die Funktionalität Audioausgabe benötigen, in Prioritätsgruppen vorgenommen. In diesem Beispiel sind drei Prioritätsgruppen gegeben, *Priorität A*, *Priorität B* und *Priorität C*, in absteigender Reihenfolge. Der Alarmdienst ist der höchsten Prioritätsgruppe zugeordnet, da er von besonderer Bedeutung für die Sicherheit im eHome ist. Da er als einziger Dienst in der höchsten Prioritätsgruppe *Priorität A* ist, kann er allen anderen Diensten Ressourcen zur Audioausgabe entziehen. In der mittleren Gruppe *Priorität B* ist der Weckdienst. Dieser hat eine niedrigere Priorität als der Alarmdienst, hat aber immer noch Vorrang vor dem Musikdienst und dem TV-Dienst. Die letzten beiden Dienste sind der niedrigsten Gruppe *Priorität C* zugewiesen. Sie können daher keinem Dienst Ressourcen zur Audioausgabe entziehen, sondern müssen solche Ressourcen abgeben, wenn ein Dienst aus einer höheren Prioritätsgruppe diese Ressourcen benötigt und keine anderen, freien Ressourcen verfügbar sind. Der Musikdienst und der TV-Dienst können sich gegenseitig auch keine Ressourcen zur Audioausgabe entziehen, da sie in derselben Prioritätsgruppe liegen. Es ist möglich, wie im dargestellten Beispiel, mehreren Diensten dieselbe Prioritätsgruppe zuzuweisen. Diese werden dann gleichrangig behandelt.

Da die Priorisierung erst zur Laufzeit des eHome-Systems durchgeführt wird, kann die Einteilung in die Prioritätsgruppen eHome-spezifisch erfolgen. Es werden daher auch nur Dienste berücksichtigt, die im betreffenden eHome vorhanden sind. Die Prioritätsgruppen können nach Bedarf über ein Administrationswerkzeug erstellt werden. Für jede Funktionalität kann eine individuelle Priorisierung der benutzenden Dienste spezifiziert werden. So ist eine Anpassung an die Besonderheiten der jeweiligen eHome-Umgebung und an die individuellen Benutzerwünsche möglich.

Modellierung

In Abbildung 4.16 ist die Modellierung der Bindungstypen dargestellt. Die Klassen Service, Function und deren Verbindung über die Klasse ServiceFunctionCardinality wurde bereits in Abbildung 4.3 diskutiert. Zur Festlegung der Bindungstypen wurde der Klasse RequiredServiceFunctionCardinaltiy ein Attribut binding-

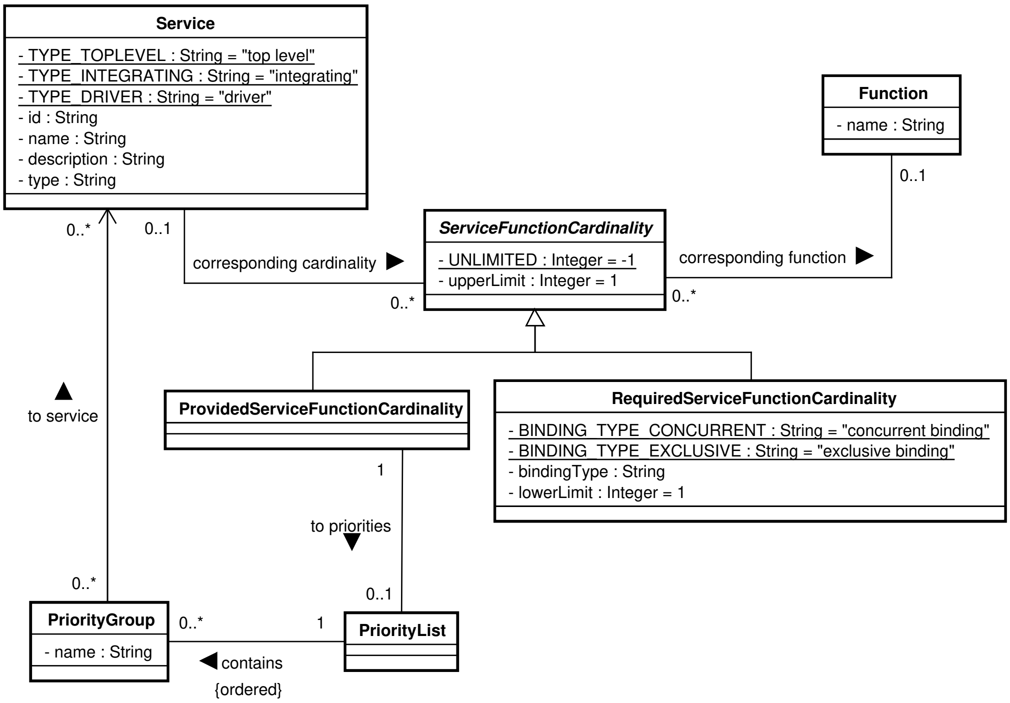


Abbildung 4.16: Modellierung der Bindungstypen.

Type hinzugefügt, für das die Konstanten BINDING_TYPE_CONCURRENT und BINDING_TYPE_EXCLUSIVE für nebenläufige bzw. exklusive Bindungen definiert sind. Der Bindungstyp wird somit für jede benötigte Funktionalität individuell festgelegt. Zur Unterstützung der Priorisierung wird, wie oben beschrieben, für jede Funktionalität eine Prioritätsliste verwaltet. Dazu dient die Klasse PriorityList, die mit der Klasse ProvidedServiceFunctionCardinaltiy verbunden ist. Somit kann die Prioritätsliste für jede angebotene Funktionalität eines Dienstes spezifisch definiert werden. Die Elemente der Prioritätsliste sind Prioritätsgruppen, modelliert durch die Klasse PriorityGroup. Jede PriorityList enthält beliebig viele solcher Gruppen, die in einer bestimmten Reihenfolge geordnet sind. Dies wird durch das Constraint {ordered} ausgedrückt. Jede Gruppe hat einen Namen (name) und verweist auf eine Menge beliebig vieler Dienste. Alle Dienste innerhalb einer Gruppe sind gleichrangig, die Rangfolge der Prioritäten wird wie oben beschrieben durch die Reihenfolge der Gruppen innerhalb der Prioritätsliste vorgegeben.

4.4 Kontinuierlicher SCD-Prozess

In diesem Abschnitt wird der eHome-spezifische Teil des erweiterten Entwicklungsprozesses beschrieben. Dieser Teil wird durch den neu eingeführten, *kontinuierlichen SCD-Prozess* [RS08] umgesetzt, der aus den Phasen *Spezifizierung*, *Konfigurierung* und *Deployment* besteht. Die anschließende *Ausführung* der konfigurierten eHome-Anwendung wird ebenfalls im Rahmen dieses Abschnitts behandelt. Insbesondere wird auf die Umstrukturierung zu einem kontinuierlichen Prozess und dessen Umsetzung eingegangen.

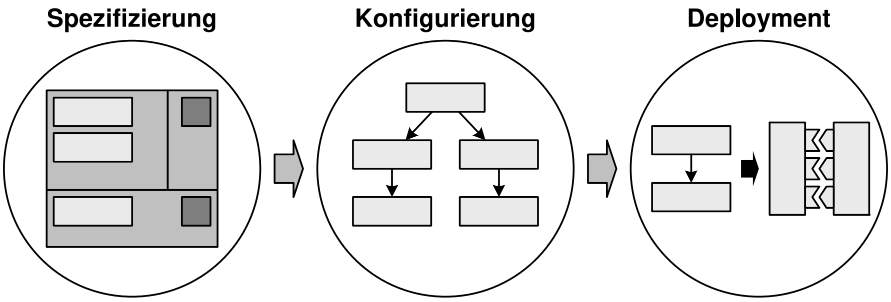


Abbildung 4.17: Phasen des SCD-Prozesses.

In Abbildung 4.17 sind die drei Phasen des SCD-Prozesses veranschaulicht. In der *Spezifizierungsphase* werden die eHome-spezifischen Anforderungen erfasst. Darauf folgt die *Konfigurierung* des eHome-Systems. Dabei wird eine Dienstkomposition erstellt, welche die Anforderungen aus der Spezifizierungsphase erfüllt und außerdem alle Abhängigkeiten der Dienste auflöst. Eine so erstellte Konfiguration wird schließlich in der *Deploymentphase* durch die Laufzeitumgebung auf dem Residential Gateway zur Ausführung gebracht. Der SCD-Prozess erzeugt so eine spezifische, den Benutzerwünschen und den aktuellen Gegebenheiten der Umgebung entsprechende eHome-Anwendung. Die Details der einzelnen Phasen des SCD-Prozesses werden im Folgenden genauer beschrieben.

4.4.1 Spezifizierung

In der ersten Phase des kontinuierlichen SCD-Prozesses, der *Spezifizierung*, werden die Anforderungen an das eHome-System erfasst. Diese können explizit vom Benutzer festgelegt werden, z. B. welche Dienste gewünscht sind oder welches konkrete Gerät für eine bestimmte Dienstfunktionalität genutzt werden soll. Anforderungen können sich aber auch implizit ergeben, z. B. aus Kontextinformationen der eHome-Umgebung. Die Position von Benutzern und Geräten im eHome etwa stellt bestimmte Anforderungen an die Konfiguration des eHome-Systems. Die Spezifikation beschreibt die Rahmenbedingungen des eHome-Systems, also alle Informationen, die für die Konfigurierung des eHomes von Bedeutung sind. Im Gegensatz zur Spezifizierungsphase im ursprünglichen SCD-Prozess wird die Umgebungsspezifikation nicht mehr als Bestandteil der Spezifikation im kontinuierlichen SCD-Prozess aufgefasst. Dies liegt daran, dass die Beschreibung der physikalischen Umgebung im laufenden Betrieb nicht geändert wird, der kontinuierliche SCD-Prozess aber fortlaufend ausgeführt wird. Der Grundriss des Gebäudes wird auch über einen längeren Zeitraum kaum geändert und daher im Rahmen des kontinuierlichen SCD-Prozesses als fix angesehen. Die Beschreibung der Umgebung liegt im neuen Prozess daher außerhalb des SCD-Prozesses und ist in Abbildung 4.2 am oberen Rand dargestellt.

Durch die adaptive Spezifizierung wird eine bestehende Spezifikation bei dynamischen Veränderungen angepasst. Im ursprünglichen SCD-Prozess war hierzu ein aufwendiger Wartungsschritt erforderlich mit den oben bereits beschriebenen nachteiligen Implikationen. Für die Berücksichtigung neuer Anforderungen musste im bisherigen Ansatz eine vollständige Neuspezifizierung erfolgen. Der damit verbundene Aufwand ist nur gerechtfertigt, wenn es sich um grundlegende Korrekturen an der Spezifikation handelt, wie etwa einer größeren Umstellung der Dienstauswahl und Raumzuordnung.

Die adaptive Spezifizierung im kontinuierlichen SCD-Prozess ermöglicht hingegen auch feingranulare Modifikationen, wie etwa die Änderung einer einzelnen Bindung zwischen zwei Diensten. Solch eine Modifikation kann nun direkt umgesetzt werden. Im ursprünglichen SCD-Prozess waren keine individuellen Bindungen modifizierbar. Die Integration neuer Geräte oder das Entfernen vorhandener Geräte kann im neuen Ansatz ebenfalls als einzelner Änderungsschritt in der Spezifikation erfasst werden.

Gleichermaßen werden Änderungen des Aufenthaltsorts einer Person berücksichtigt. Jeden Raumwechsel eines Benutzers durch einen Wartungsschritt abzubilden wäre nicht realisierbar. Im erweiterten Ansatz kann außerdem die aktuelle Position eines mobilen Geräts berücksichtigt werden. Dies ist eine wesentliche Weiterentwicklung, da im bisherigen Ansatz diese Art von Dynamik nur innerhalb der Dienstimplementierung berücksichtigt werden konnte. Aufgrund der statischen Struktur war auf der Ebene der Dienstkomposition, also der Struktur der eHome-Anwendung, keine Berücksichtigung möglich. Die adaptive Spezifizierung ermöglicht es, den aktuellen Zustand der eHome-Umgebung zu modellieren. Dies ist eine entscheidende Voraussetzung zur Unterstützung kontextbezogener eHome-Anwendungen.

Die adaptive Spezifizierung wird durch verschiedene Faktoren beeinflusst. Es kann zwischen expliziten und impliziten Änderungen unterschieden werden. Explizite Änderungen sind geänderte Benutzeranforderungen, die vom Benutzer direkt eingegeben werden, z. B. das Hinzufügen, Entfernen oder Verschieben von Dienstobjekten. Implizite Änderungen sind beispielsweise Änderungen des räumlichen Kontextes von Diensten. Ein Dienst, der an eine Person gebunden ist, bewegt sich mit dieser Person durch die Umgebung. So können sich implizit auch die Anforderungen dieses Dienste in Bezug auf die verwendbaren Basisdienste bzw. Ressourcen ändern. Gleiches gilt auch für mobile Geräte. Zu den impliziten Änderungen gehören auch Änderungen von bestimmten Dienstzuständen, die für die Struktur des Systems von Bedeutung sind. So benötigt ein Dienst nicht zu jedem Zeitpunkt alle seiner gebundenen Ressourcen. Die Nutzung der Bindungen eines Dienstes zu anderen Diensten ist wichtig für die effektive Ausnutzung der in der Umgebung verfügbaren Ressourcen. Darauf wird in Abschnitt 4.4.4 genauer eingegangen.

Die oben beschriebenen Anpassungen im Rahmen der adaptiven Spezifizierung können durch die in der vorliegenden Arbeit entwickelten Werkzeuge interaktiv durchgeführt werden. Der Laufzeit-Manager (siehe Abschnitt 6.3.3) ermöglicht die Visualisierung des momentanen Systemzustands und die Interaktion mit dem Benutzer. Die Visualisierung dieses Zustands ist von Bedeutung, da der Benutzer ggfs. die Komposition der Dienste und damit das Zusammenspiel der Komponenten des eHome-Systems untersuchen will. Es ist wichtig, dass der Benutzer die von ihm wahrgenommene Funktionalität der eHome-Anwendung nachvollziehen kann, insbesondere wenn ein unerwünschtes Verhalten des Systems auftritt. Damit der Benutzer in diesem Fall Änderungen direkt vornehmen kann, muss auch die Interaktion mit dem eHome-System unterstützt werden. So ermöglicht die adaptive Spezifizierung auch die gezielte Änderung einzelner

Dienstbindungen. Dies wurde bereits in Abschnitt 4.3.3 erläutert. Auf diese Weise kann z. B. die Verwendung einer bestimmten Ressource durch einen Dienst vom Benutzer unmittelbar festgelegt werden. Die Visualisierung und Interaktion mit dem System adressiert die Punkte 1 (das „zufällige Smart Home“) und 3 (kein Systemadministrator) der von EDWARDS und GRINTER beschriebenen sieben Herausforderungen bei der Entwicklung von eHome-Systemen (vgl. Abschnitt 2.3 und [EG01]).

Kontexterfassung

Die interaktive Anpassung der Spezifikation durch den Laufzeit-Manager ermöglicht die Abbildung von dynamischen Veränderungen zur Laufzeit im eHome. Diese Anpassungen werden im eHome-Modell erfasst und werden dann durch die darauf folgende Konfigurierung verarbeitet. Für die Realisierung von eHome-Systemen ist aber auch eine automatische Erfassung von Kontextinformationen wichtig.

Der Begriff *Kontext* ist recht allgemein und wird je nach Anwendung unterschiedlich ausgelegt. Eine häufig zugrunde gelegte Definition ist die von ABOWD et al.:

> „Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves. [ADB+99]“

Demnach werden unter Kontext alle Informationen zur Charakterisierung z. B. einer Person oder eines Ortes aufgefasst, die für die Interaktion relevant sind. Für die strukturelle Adaption wird vorausgesetzt, dass Kontextinformationen über Geräte und Personen verfügbar sind, die insbesondere auch deren Position innerhalb des eHomes umfassen. In der vorliegenden Arbeit wird der Aufbau des eHomes selbst durch die Umgebungsspezifikation einmalig erfasst. Zwar können die Positionen von Personen und Geräten der Laufzeitumgebung auch interaktiv über die Werkzeuge übermittelt werden, für die praktische Anwendung ist aber eine automatische Behandlung sinnvoll.

Wie bereits in Abschnitt 1.2 erläutert wurde, werden in dieser Arbeit keine Mechanismen für die automatische Kontexterfassung und die Lokalisierung von Personen untersucht. Dies würde den Rahmen der Arbeit überschreiten. Im eHome-Projekt am Lehrstuhl für Informatik 3 wurde jedoch bereits in [Fro08] eine Kontextkomponente erarbeitet, die als Schnittstelle zwischen externen Sensordaten und dem eHome-Modell, das die Grundlage der Komposition darstellt, verwendet werden kann. Auf dieser Basis kann eine automatische Erfassung von Kontextinformationen und deren Verarbeitung im eHome-Modell stattfinden.

Das eHome-Modell umfasst die Kontextinformationen, die im kontinuierlichen SCD-Prozess verwendet werden. Darin werden auch die in der eHome-Umgebung anwesenden Personen erfasst. In der Arbeit von ARMAÇ werden weiterführende Konzepte für die Erfassung und Nutzung personenbezogener Kontextinformationen entwickelt. Dazu wird der Personenkontext in einem Benutzermodell gespeichert [APPR09, AR08]. Durch eine selektive Zugriffskontrolle wird dabei sichergestellt, dass Benutzerdaten nur an vertrauenswürdige Dienste weitergegeben werden.

In der Literatur sind verschiedene Ansätze beschrieben, die die automatische Ermittlung von Ortsinformationen auf Basis unterschiedlichster Sensoren betreffen. Bereits 1992 wurde von WANT et al. ein System zur Lokalisierung von Personen auf Basis von Infrarotsendern entwickelt [WHFG92]. Damals wurde noch nicht die Anwendung für „intelligente“ Umgebungen betrachtet, sondern es sollten Mitarbeiter z. B. in einem Krankenhaus schnell gefunden und koordiniert werden können. Inzwischen wurden auch zahlreiche weitere Techniken auf ihre Einsatzmöglichkeiten zur Lokalisierung untersucht.

In [Tek05] wird beispielsweise die Nutzung von WLAN für die Lokalisierung von Personen innerhalb von Gebäuden untersucht. Die Nutzung von RFID in der Logistik wurde bereits in Abschnitt 2.1 erwähnt. In [NLLP04] wird ein Prototyp vorgestellt, der auf Basis von RFID die Lokalisierung in Gebäuden ermöglicht. Eine Übersicht weiterer Techniken wird in [Ait03] gegeben. Auch wenn noch nicht erkennbar ist, welche Technik sich letztendlich für die Nutzung in eHomes am besten eigenen wird und welche auch von den Nutzern am ehesten angenommen wird, sind verschiedene Verfahren möglich.

Diese Verfahren können an den hier vorgestellten Ansatz angebunden werden, sodass eine automatische Aktualisierung der Kontextinformationen möglich ist.

Die Entwicklung von Diensten ist davon nicht betroffen, da die Kontextinformationen von der Laufzeitumgebung im eHome-Modell verwaltet werden und dort ggfs. zu einer Rekonfigurierung führen. Die Auswirkungen auf die Dienste ergeben sich dann implizit durch Ausführung des kontinuierlichen SCD-Prozesses.

Realisierung

Abbildung 4.18 zeigt eine Übersicht der wesentlichen Elemente des eHome-Modells. Das Modell basiert in den Grundzügen auf den Vorarbeiten, die in Abschnitt 3.4 beschrieben sind. Durch verschiedene Erweiterungen und Modifikation wurde es an den neuen, kontinuierlichen SCD-Prozess angepasst. Einige Teile des Modells wurden bereits in Abschnitt 4.3 im Detail erläutert.

Ein wesentlicher Teil ist die Spezifikation der Dienste. Die Aspekte der Dienstspezifikation sind im rechten Teil der Abbildung dargestellt und hängen mit den beiden Klassen Service und Function zusammen, die schattiert dargestellt sind. Die dazwischenliegende Klasse ServiceFunctionCardinality wird für das Zusammenspiel von Diensten und ihren Funktionalitäten benötigt, was bereits in Abbildung 4.3 beschrieben wurde. Die zwei Unterklassen von ServiceFunctionCardinality dienen der Unterscheidung zwischen angebotenen und benötigten Funktionalitäten und speichern unter anderem auch die Bindungsbeschränkungen und die Bindungsstrategien. Die Details hierzu sind in den Abbildungen 4.11, 4.13 und 4.16 zu finden.

Die Klasse ServiceObject im zentralen Teil der Abbildung modelliert die Dienstobjekte und ist mit der Klasse Service verbunden, die die Dienstbeschreibungen speichert. Jeder Dienst kann beliebig viele Dienstobjekte als Instanzen haben. Diese werden im Rahmen der Konfigurierung miteinander verbunden. Die Dienstkomposition wird im Modell durch Kanten zwischen den Dienstobjekten modelliert, wobei auch hier wieder ein Zwischenknoten benötigt wird, der bestimmte Attribute zu den Dienstbindungen speichert. Die von einem Dienstobjekt gebundenen anderen Dienstobjekte können über die Kante uses erreicht werden. Diese verweist auf Objekte der Klasse ServiceObjectConnectionProperties, die dann auf die gebundenen Dienstobjekte über eine Kante is used by verweisen. Da Bindungen auf Funktionalitäten basieren, besteht eine Verbindung zwischen den Klassen ServiceObjectConnectionProperties und Function. So wird

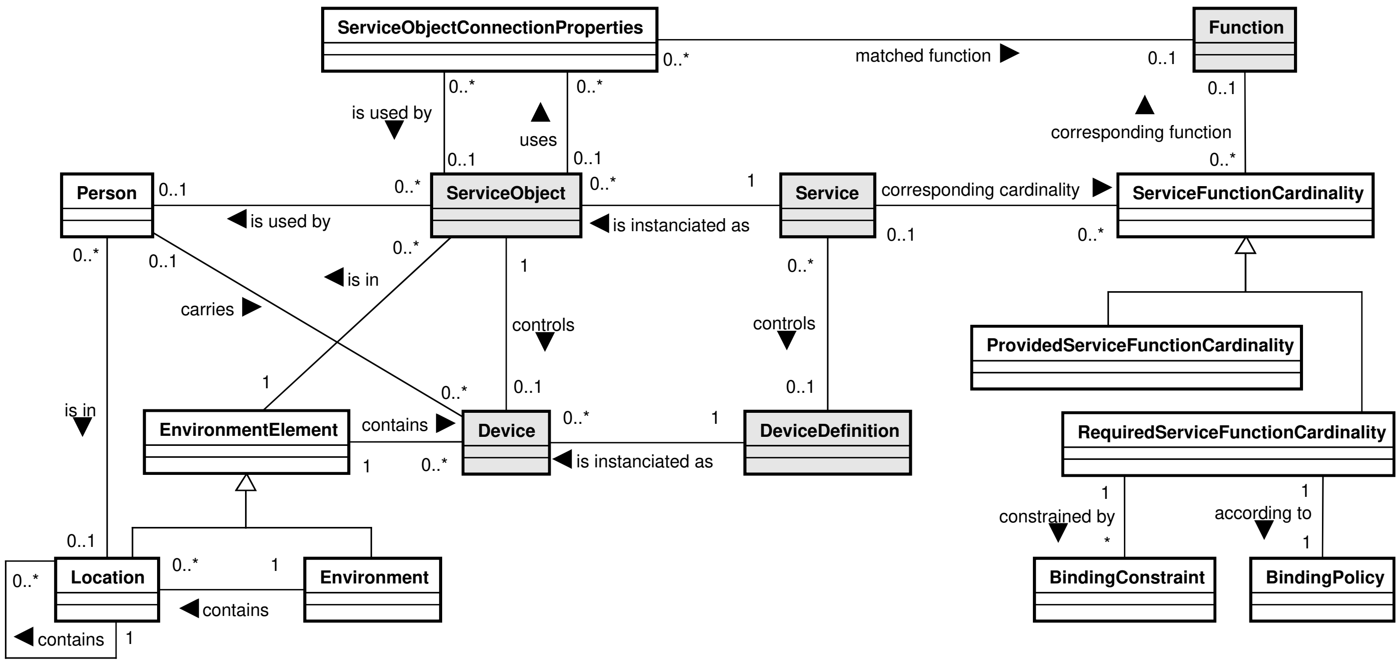


Abbildung 4.18: Übersicht der Hauptelemente des neuen eHome-Modells.

festgehalten, auf welcher Funktionalität eine bestimmte Bindung basiert. Die Modellierung der Dienstkomposition wird detailliert in Abbildung 4.24 in Abschnitt 4.4.2 beschrieben.

Treiberdienste steuern Geräte, was ebenfalls im eHome-Modell erfasst wird. Die Klasse DeviceDefinition beschreibt Gerätetypen, die Klasse Device hingegen entspricht den konkreten Geräten, die im eHome vorhanden sind. Analog zur Modellierung der Dienste gibt es also auch hier eine Unterscheidung zwischen einer Typ- und einer Instanzebene. Die Dienste (Service) beziehen sich daher auch auf Gerätetypen (DeviceDefinition), hingegen beziehen sich Dienstobjekte (ServiceObject) auf konkrete Geräte (Device).

Im linken Teil der Abbildung 4.18 sind Elemente zur Kontextmodellierung dargestellt. Dies sind zum einen Personen, die durch die Klasse Person modelliert werden, und zum anderen verschiedene Umgebungselemente im eHome, repräsentiert durch die Klasse EnvironmentElement. Bei den Umgebungselementen wird zwischen der gesamten eHome-Umgebung (Environment) und einzelnen Räumen oder Raumbereichen (Location) unterschieden. Die Modellierung ist wenig restriktiv, so kann die Klasse Location grundsätzlich beliebige Teile des eHomes beschreiben, z. B. auch verschiedene Etagen, Wohneinheiten oder auch Raumteilbereiche innerhalb eines Raums. Hierarchische Strukturen können über die contains-Kante abgebildet werden. Zur Lokalisierung von Personen innerhalb des eHomes wird im Modell die Kante is in zwischen Person und Location verwendet. Die Kante is used by zwischen ServiceObject und Person ermöglicht es, Dienstobjekte einer spezifischen Person zuzuordnen. Dies ist für die Behandlung personenbezogener Dienste von großer Bedeutung. Außerdem können Personen auch mobile Geräte mitführen, was durch die Kante carries zwischen Person und Device im Modell erfasst werden kann. Eine wichtige Aufgabe des eHome-Modells ist außerdem die Modellierung des räumlichen Kontextes von Dienstobjekten und Geräten. Dies wird durch die Kanten is in bzw. contains unterstützt. Diese Informationen werden für die Konfigurierung benötigt.

Die Übersicht des eHome-Modells zeigt die verschiedenen Aspekte, die im Ansatz der vorliegenden Arbeit zu einem globalen Modell integriert sind. Zum einen werden Anforderungen erfasst, die bei der Dienstentwicklung oder der Administration des eHome-System festgelegt werden. Zum anderen werden Informationen über den Ist-Zustand des eHomes im Modell abgebildet, insbeson-

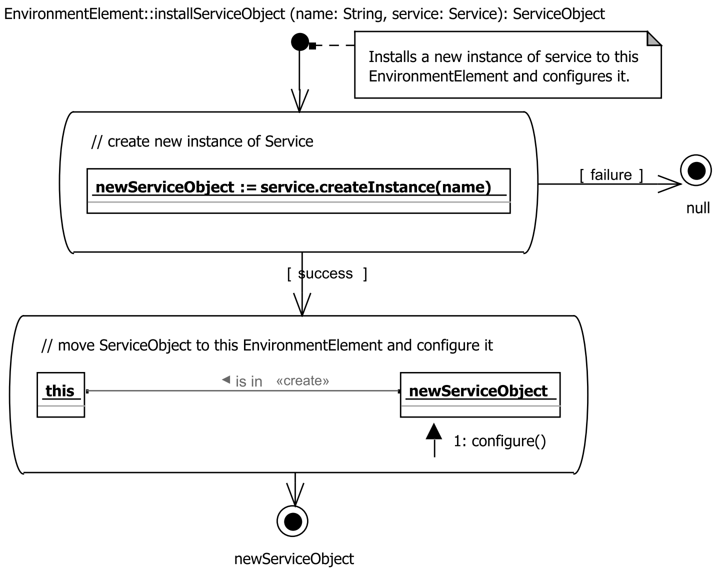


Abbildung 4.19: Erstellen eines ServiceObject.

dere die Kontextinformationen, die im kontinuierlichen SCD-Prozess Berücksichtigung finden. Schließlich wird auch die Konfigurierung selbst auf Basis des eHome-Modells durchgeführt, die Ergebnisse der Dienstkomposition werden also ebenfalls im Modell erfasst. Die Konfigurierungsphase wird im folgenden Abschnitt 4.4.2 genauer vorgestellt.

In Abbildung 4.19 ist ein Beispiel für die Realisierung einer Methode in Fujaba dargestellt. Dazu werden Story-Diagramme eingesetzt (vgl. Abschnitt 3.3.2). Da Fujaba in dieser Arbeit zur Umsetzung der strukturellen Adaption verwendet wird, ist die entsprechende Anwendungslogik in Form solcher Story-Diagramme modelliert. Abbildung 4.19 zeigt den einfachen Fall des Erstellens eines neuen Dienstobjekts in einem bestimmten Umgebungselement durch die Methode installServiceObject der Klasse EnvironmentElement. Beim Aufruf wird ein Name für das Dienstobjekt übergeben, sowie der Dienst, der instanziiert werden soll. Der Ablauf beginnt in der Abbildung oben und als erster Schritt wird das Dienstobjekt erzeugt. Dazu wird die Methode createInstance des angegebenen Dienstes aufgerufen. Im Fall eines Fehlers wird null zurückgeliefert, anderenfalls wird mit der Zuordnung des neu erstellten Dienstobjekts newServiceObject zum Umgebungselement this, auf das sich die Methode bezieht, fortgefahren. Dazu wird eine entsprechende Kante is in zwischen den Objekten angelegt, was in Story-

Diagrammen durch den Stereotyp «create» dargestellt wird. Abschließend wird mit der Konfigurierung des neu erstellten Dienstobjekts fortgefahren. Dazu wird die Methode configure von newServiceObject aufgerufen. Welche Schritte bei der Konfigurierung durchgeführt werden, wird im folgenden Abschnitt erläutert.

4.4.2 Konfigurierung

Die Phase der *Konfigurierung* folgt auf die Spezifizierungsphase. Aufgabe dieser Phase ist es, eine gültige Konfiguration des eHome-Systems zu erzeugen, sodass alle Dienste auf dem Service Gateway deployt und ausgeführt werden können. Durch die Einführung der strukturellen Adaption ergeben sich neue Anforderungen an die Konfigurierungsphase. Auch hier müssen nun Anpassungsschritte ausgeführt werden, die die Änderungen an der Spezifikation berücksichtigen und ggfs. eine bestehende Konfiguration anpassen, sodass wieder ein ausführbarer Zustand des Systems erreicht wird.

Anwendung von Bindungsbeschränkungen und Bindungsstrategien

Für personen- und kontextbezogene Dienste werden die in Abschnitt 4.3.2 beschriebenen Bindungsbeschränkungen eingesetzt. Diese müssen bei der Konfigurierung berücksichtigt werden. Beim Auffinden passender Dienstobjekte für zu erstellende Bindungen müssen die Bindungsbeschränkungen daher geprüft werden. Eine Bindung ist nur möglich, wenn alle spezifizierten Bindungsbeschränkungen auch eingehalten werden. Der erweiterte Entwicklungsprozess ermöglicht im Rahmen der strukturellen Adaption die Anpassung einer bereits erstellten Konfiguration. Mittels der Bindungsbeschränkungen kann dabei auch eine deutlich einfachere Entwicklung personenbezogener Dienste erfolgen. Die Verlagerung der kontextbezogenen Adaption von der Dienstimplementierung in die Laufzeitumgebung des eHome-Systems ist daher einer der wesentlichen Aspekte des erweiterten Entwicklungsprozesses und betrifft insbesondere auch die Konfigurierungsphase des kontinuierlichen SCD-Prozesses.

Neben den Bindungsbeschränkungen müssen auch die Bindungsstrategien im kontinuierlichen SCD-Prozess berücksichtigt werden. Durch das iterative Vorgehen ist eine automatische Behandlung in vielen Fällen zwingend erforderlich, da der Benutzer nicht bei allen feingranularen Änderungen im Betrieb des eHome-Systems manuell eingreifen kann. In Abschnitt 4.3.3 wurde zwischen der automatischen, der automatisch-obligatorischen und der manuellen Bindungsstrategie unterschieden. Je nach Art des Dienstes und der betroffenen Funktionalität kann zwischen diesen Strategien gewählt werden, wodurch der Dienstentwickler den Ablauf der Konfigurierung beeinflussen kann.

Verwaltung der Dienstobjekte

Im kontinuierlichen SCD-Prozess wird die Konfigurierung iterativ zur Laufzeit des eHome-Systems durchgeführt. Dies bedeutet, dass keine virtuellen Geräte mehr bei der Konfigurierung erstellt werden, wie dies im bisherigen Ansatz der Fall war (vgl. Abschnitt 3.4.2). Virtuelle Geräte, die als Platzhalter für noch anzuschließende oder anzuschaffende Geräte dienen, haben im kontinuierlichen SCD-Prozess keine sinnvolle Funktion. Da es nicht möglich ist, während des Betriebs auf Funktionalitäten nicht verfügbarer Hardwareressourcen zurückzugreifen, werden nur tatsächlich vorhandene Ressourcen bei der Konfigurierung berücksichtigt.

Durch das Fehlen virtueller Geräte kann es jedoch vorkommen, dass ein Dienstobjekt Funktionalitäten benötigt, die von den derzeit vorhandenen Ressourcen nicht angeboten werden. Wenn mittels integrierender und Treiberdienste keine vollständige Konfigurierung möglich ist, so ist das betroffene Dienstobjekt *ungültig*. Ein ungültiges Dienstobjekt kann in einer Konfiguration enthalten sein, in diesem Fall kann es jedoch nicht deployt und gestartet werden. Daher kann ein Dienstobjekt in diesem Zustand seine angebotenen Funktionalitäten nicht für den Benutzer bzw. für andere Dienstobjekte, die an das ungültige Dienstobjekt gebunden sind, zur Verfügung stellen. Ein bereits vorhandenes ungültiges Dienstobjekt kann während des Betriebs jederzeit *gültig* werden, wenn z. B. ein fehlendes Gerät angeschlossen wird oder ein entsprechendes mobiles Gerät in der Umgebung verfügbar wird. Auch wenn ein neuer integrierender Dienst zur Verfügung steht, der die Nutzung einer bereits vorhandenen Ressource ermöglicht, kann eine gültige Konfigurierung möglich werden. Ein Dienstobjekt wird in einen gültigen Zustand überführt, sobald eine Möglichkeit besteht, die Anforderungen unter Beachtung der spezifizierten Bindungsbeschränkungen und Bindungsstrategien zu erfüllen. Im Unterschied zum Ansatz von NORBISRATH kann also ein Dienstobjekt zunächst ungültig sein und erst später, nach dem Hinzufügen einer benötigten Ressource, gültig werden.

Der Fall, dass ein bereits gültig konfiguriertes Dienstobjekt während der Laufzeit ungültig wird, kann ebenfalls vorkommen. Falls umgekehrt zum oben beschriebenen Fall eine benötigte Ressource, die an ein Dienstobjekt gebunden ist, nicht länger verfügbar ist, so kann das Dienstobjekt ungültig werden. Zunächst ist eine Rekonfigurierung erforderlich, um die geänderte Situation im eHome-System zu erfassen und zu berücksichtigen. Dabei wird nach Ersatz für weggefallene Ressourcen gesucht, um alle Dienstobjekte soweit möglich in einem gültigen Zustand zu halten damit so viele von ihnen wie möglich nutzbar sind.

Die Rekonfigurierung ist insbesondere erforderlich, da Dienstobjekte im erweiterten Entwicklungsprozess jederzeit im laufenden Betrieb hinzugefügt werden können. Damit auch diese Dienstobjekte ausführbar werden, muss bei der Rekonfigurierung versucht werden, einen gültigen Zustand des Dienstobjekts herzustellen. Für die Erfüllung der benötigten Funktionalitäten eines neuen Dienstobjekts werden soweit möglich bereits bestehende Dienstobjekte verwendet. Wenn also ein bereits vorhandenes Dienstobjekt eine Funktionalität mehrfach anbietet und bisher durch bestehende Bindungen nicht vollständig ausgelastet ist, so kann dieses Dienstobjekt weiter genutzt werden. Nur wenn keine weiteren Bindungen zu existierenden Dienstobjekten möglich sind, wird ggfs. eine neues Dienstobjekt erstellt.

Falls keine der angebotenen Funktionalitäten eines Dienstobjekts von irgendeinem anderen Dienstobjekt in der Konfiguration genutzt wird, so wird das Dienstobjekt nicht länger benötigt. In diesem Fall sorgt die Konfigurierung dafür, dass dieses Dienstobjekt aus der Konfiguration entfernt wird. Ein solcher Aufräummechanismus ist notwendig, da sonst nach längerer Betriebszeit des Systems zahlreiche ungenutzte Dienstobjekte in der Konfiguration enthalten wären, die für die vom Benutzer gewünschten Funktionalitäten nicht benötigt werden. Dies würde zu einer unnötigen Verschwendung von Systemressourcen führen und das System im schlimmsten Fall bei der Ausführung der tatsächlich benötigten Dienste behindern. Der Aufräummechanismus betrifft ausschließlich integrierende Dienste und Basisdienste, da nur diese Typen von Diensten automatisch vom Konfigurierungsmechanismus instanziiert werden. Top-Level-Dienste werden hingegen vom Benutzer manuell ausgewählt, wenn er die entsprechenden Funktionalitäten nutzen möchte. Basisdienste sind beispielsweise betroffen, wenn die von ihnen gesteuerten Geräte entfernt werden und nicht mehr verfügbar sind. Dann wird der steuernde Treiberdienst ebenfalls nicht mehr benötigt und wird entsprechend aus der Konfiguration entfernt.

Manuelle Anpassungen der Konfiguration

In Abschnitt 4.3.3 wurde unter anderem auch eine manuelle Bindungsstrategie eingeführt, bei der sämtliche Bindungen zu anderen Dienstobjekten vom Benutzer manuell angelegt werden müssen. Dies wird für Dienste eingesetzt, bei denen der Benutzer über die zu nutzenden Ressourcen selbst entscheiden muss. Dies könnte z. B. für einen Dienst zur Videoüberwachung eingesetzt werden, wenn nur vom Benutzer explizit angebundene Kameras verwendet werden sollen. Auch bei der automatisch-obligatorischen Bindungsstrategie gibt es ein Intervall, in dem der Benutzer manuelle Bindungen anlegen kann. Selbst bei der automatischen Bindungsstrategie, die alle Bindungen automatisch erstellt, ist es möglich, eine vorhandene Bindung manuell zu ändern, um z. B. ein anderes Gerät für die Realisierung einer Funktionalität zu verwenden.

In solchen Fällen wird die Konfiguration durch die Interaktion des Benutzers beeinflusst. Dies ist ein neuer Ansatz, der eine Erweiterung der Konfigurierungsphase darstellt. Die manuelle Anpassbarkeit der Konfiguration ist ein wichtiger Aspekt des erweiterten Entwicklungsprozesses. Auch wenn eine automatische Adaption für die vielen feingranularen Schritte zwingend erforderlich ist, um den Benutzer nicht von seiner eigentlichen Tätigkeit abzuhalten, so ist es dennoch entscheidend, dass der Benutzer auch die Kontrolle über seine Umgebung behält. Dies ist eine Abwägung, von der abhängen wird, ob eHomes in der Praxis eingesetzt und Verbreitung finden werden oder nicht.

Wird dem Benutzer im laufenden Betrieb des eHomes zu viel unnötige Interaktion abverlangt, so wird das eHome-System schnell zur Last, die durch den Mehrwert an Funktionalität und Komfort nicht zu rechtfertigen ist. Der Benutzer des eHome-Systems kann nicht als Administrator betrachtet werden, was auch als Punkt 3 (kein Systemadministrator) der sieben Herausforderungen bei der Entwicklung von eHome-Systemen von EDWARDS und GRINTER beschrieben wird (vgl. Abschnitt 2.3 und [EG01]). Ist aber umgekehrt gar keine Interaktion

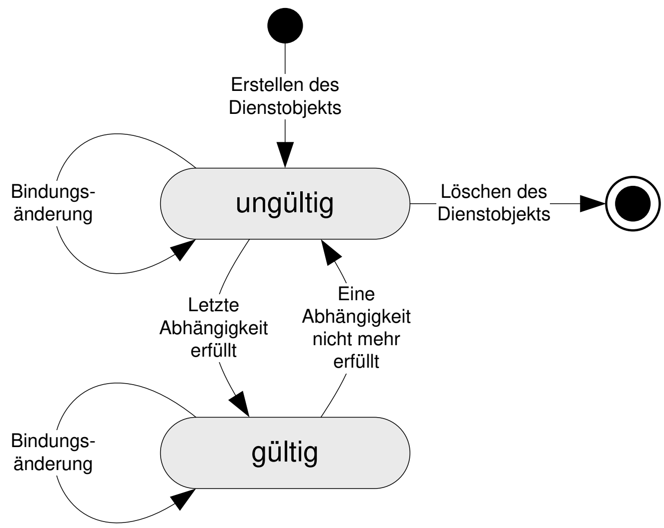


Abbildung 4.20: Zustandsdiagramm für die Konfigurierung von eHome-Diensten.

durch den Benutzer möglich, so wird sich dieser dem eHome-System früher oder später ausgeliefert fühlen, da er die Abläufe, die zu einem Effekt in seiner Umgebung führen, möglicherweise gar nicht mehr nachvollziehen kann und somit auch keine Kontrolle über seine Umgebung hat. Solch ein System wird von den Benutzern nicht angenommen werden. Für die Konfigurierung bedeutet dies, dass eine passende Abwägung der beiden Extreme angestrebt werden muss. Im Rahmen dieser Arbeit wird dies durch mehrere Maßnahmen umgesetzt. Auf der einen Seite wird mittels der Bindungsstrategien und der Bindungsbeschränkungen eine Automatisierung der vielen feingranularen Adaptionsschritte ermöglicht. Auf der anderen Seite gibt es durch die manuelle Anpassbarkeit der Konfiguration auch an zahlreichen Stellen die Möglichkeit der Interaktion durch den Benutzer.

Konfigurierungszustände eines eHome-Dienstes

Aufgrund der Adaption zur Laufzeit des eHome-Systems muss wie oben diskutiert bei der Rekonfigurierung zwischen verschiedenen Zuständen der Dienstobjekte unterschieden werden. Dabei wird zunächst der *Konfigurierungszustand* betrachtet, der für die Konfigurierungsphase relevant ist. Später wird noch ein

Deploymentzustand eingeführt, der vom Konfigurierungszustand abhängig ist, und für das Deployment entscheidend ist. Bei Dienstobjekten wird zwischen den beiden Konfigurierungszuständen gültig und ungültig unterschieden. In Abbildung 4.20 ist dazu ein Zustandsdiagramm dargestellt, aus dem die möglichen Zustandsübergänge ersichtlich werden. Zunächst ist das Dienstobjekt nach dem Erstellen im Zustand ungültig. Der Konfigurierungsmechanismus versucht nun das Dienstobjekt in einen ausführbaren Zustand zu überführen, d. h. eine gültige Konfiguration herzustellen. Dazu wird versucht, Bindungen zu anderen Dienstobjekten herzustellen, die die benötigten Funktionalitäten anbieten. Es erfolgen also Bindungsänderungen bis die letzte Abhängigkeit erfüllt ist. Dann erfolgt der Übergang zum Zustand gültig. Hier können abhängig von der Bindungsstrategie und von der manuellen Einflussnahme des Benutzers weitere Bindungsänderungen stattfinden. Wird durch Bindungsänderungen des Dienstobjekts eine Abhängigkeit nicht mehr erfüllt, so findet ein Wechsel zurück zum Zustand ungültig statt. Nachdem alle Bindungen entfernt wurden, kann das Dienstobjekts schließlich gelöscht werden. Solange sich das Dienstobjekt im Zustand gültig befindet, kann es ausgeführt werden und damit die angebotenen Funktionalitäten zur Verfügung stellen. Befindet sich das Dienstobjekt im Zustand ungültig, so kann es nicht länger ausgeführt werden. Der Konfigurierungszustand ist damit wichtig für die spätere Ausführung. Dies wird durch das in Abschnitt 4.4.3 beschriebene Deployment realisiert.

Realisierung

Abbildung 4.21 zeigt, wie die Bindungsstrategien aus der Dienstspezifikation in die Konfigurierung einbezogen werden. Die Methode configure der Klasse ServiceObject, die in der Abbildung dargestellt ist, wird immer dann aufgerufen, wenn ein Dienstobjekt konfiguriert werden muss. Dies ist erforderlich, wenn das Dienstobjekt neu erstellt wurde, aber auch, wenn die Konfigurierung aktualisiert werden muss. Der wesentliche Teil der configure-Methode ist eine sogenannte *for-each*-Aktivität, die in Fujaba durch eine doppelte Umrandung gekennzeichnet wird. Diese Aktivität wird für alle passenden Anwendungsstellen im Laufzeitgraphen ausgeführt. Das gesuchte Graphmuster geht vom Dienstobjekt this, auf das die Methode angewendet wird, aus. Über den zugehörigen Dienst service wird jeweils eine benötigte Funktionalität function gesucht. Dann wird die

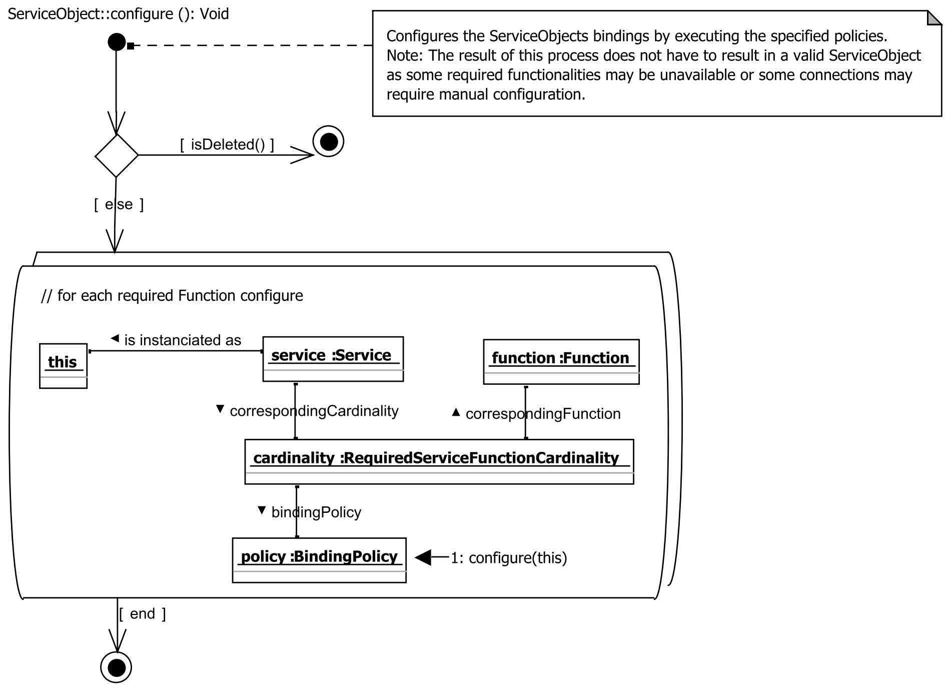


Abbildung 4.21: Konfigurierung eines ServiceObject.

zu der benötigten Funktionalität spezifizierte Bindungsstrategie policy bestimmt, und es wird die configure-Methode des policy-Objekts auf das ServiceObjekt this angewendet. Wie oben erwähnt, wird dieser Vorgang für alle benötigten Funktionalitäten des Dienstes wiederholt.

In Abbildung 4.22 ist ein Beispiel für die configure-Methode einer Bindungsstrategie dargestellt. Dabei handelt es sich um die automatisch-obligatorische Bindungsstrategie (vgl. Abschnitt 4.3.3). Als Eingabeparameter wird das zu konfigurierende Dienstobjekt serviceObject übergeben. Im ersten Schritt wird die zur Bindungsstrategie gehörige Funktionalität function bestimmt. Dann erfolgt der eigentliche Konfigurierungsschritt, der in der Abbildung durch die annotierte Kante am unteren Rand dargestellt wird. Dabei wird durch die Methode mandatoryRequiredInstancesLeft bestimmt, wie viele Bindungen der Funktionalität noch benötigt werden, um die Mindestanforderung zu erfüllen. Falls noch weitere Bindungen erforderlich sind, müssen neue Bindungen angelegt werden, bis die Anforderungen erfüllt sind. Dazu wird die Methode addBinding der Bindungsstrategie, in diesem Fall AutoMandatoriesBindingPolicy, aufgerufen. Die Im-

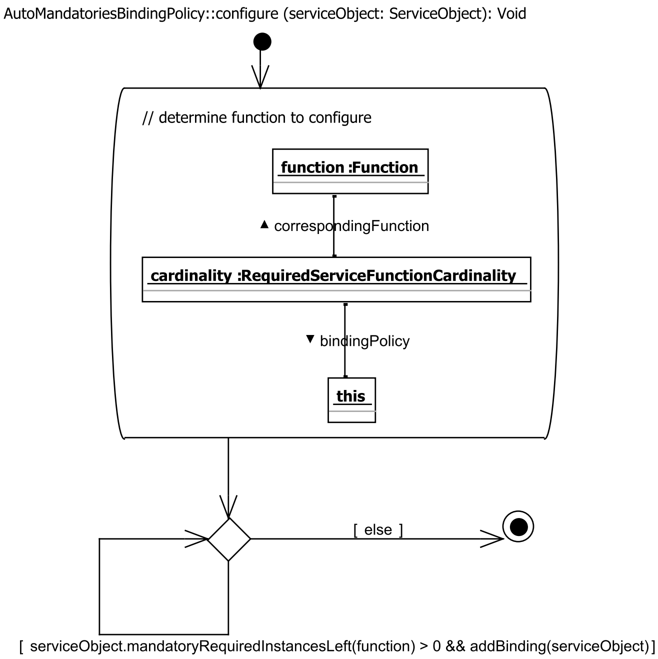


Abbildung 4.22: Konfigurierung mit automatisch-obligatorischer Bindungsstrategie.

plementierung dieser Methode wird aufgrund ihres Umfangs an dieser Stelle nicht im Detail beschrieben. Die addBinding-Methode erstellt probeweise neue Bindungen zu bereits als gültig konfigurierten Dienstobjekten, die die benötigte Funktionalität anbieten und noch weitere Bindungen aufnehmen können. Dann wird jeweils geprüft, ob durch die neu erstellte Bindung das betrachtete Dienstobjekt gültig wird, d. h. alle benötigten Funktionalitäten gebunden sind. Dabei müssen insbesondere auch die Bindungsbeschränkungen berücksichtigt werden, denn nur wenn diese erfüllt werden, ist eine Bindung zulässig.

In Abbildung 4.23 ist die Überprüfung einer Bindungsbeschränkung am Beispiel der *Beschränkung auf lokale Ressourcen* dargestellt. Die Überprüfung wird durch die Methode satisfied der entsprechenden Klasse EnvironmentElementBindingConstraint realisiert. Als Eingabeparameter wird das ServiceObjectConnectionProperties-Objekt, das zur prüfenden Bindung gehört, übermittelt. Im ersten Schritt werden die mit diesem Objekt assoziierten Objekte ermittelt, die für die Überprüfung von Belang sind. Dann wird geprüft, ob es sich bei dem gebundenen Dienstobjekt um einen Treiberdienst handelt. Wenn dies der Fall ist, so wird geprüft, ob das von diesem Treiberdienst gesteuerte Gerät im selben Umge-

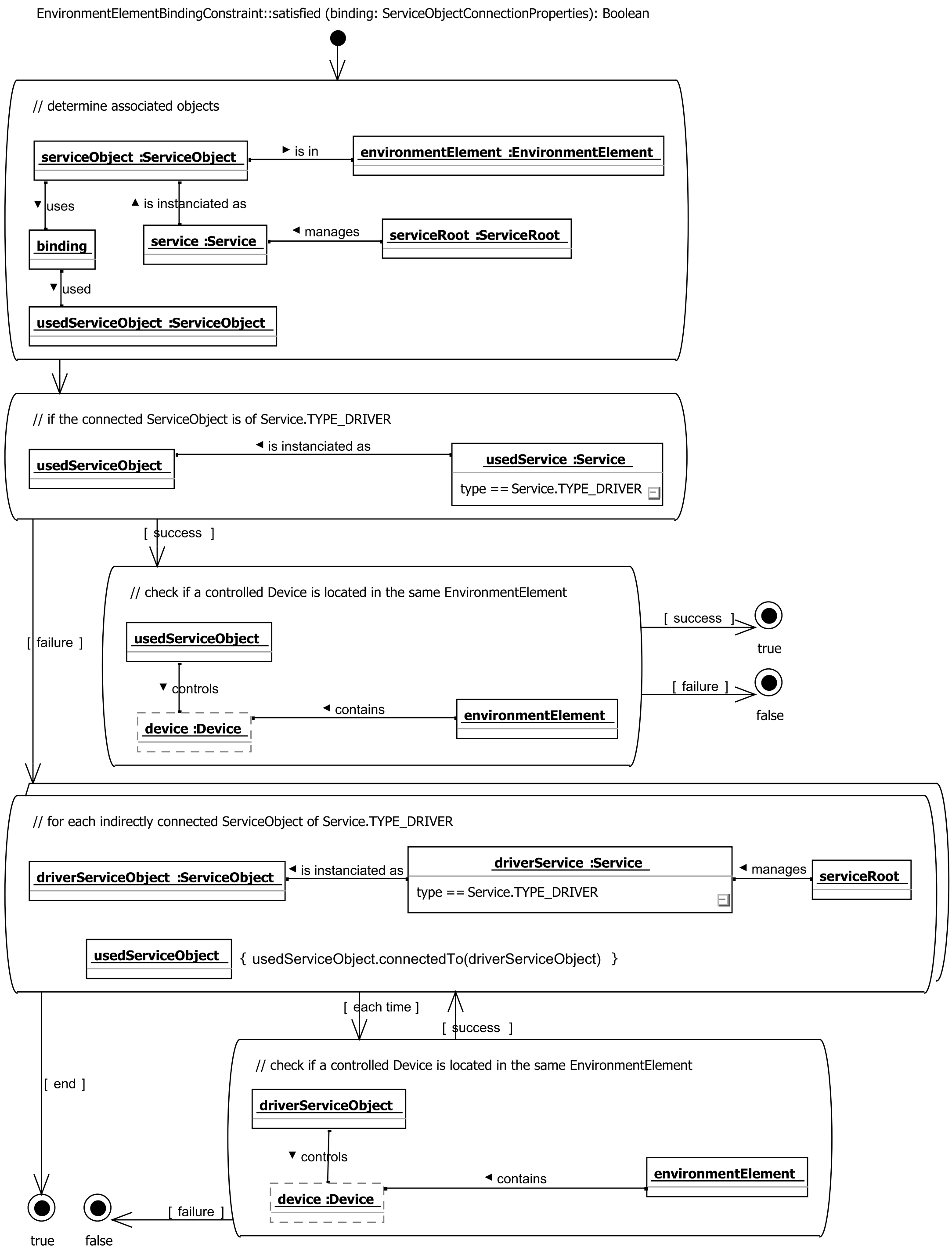


Abbildung 4.23: Überprüfung der Bindungsbeschränkung auf lokale Ressourcen.

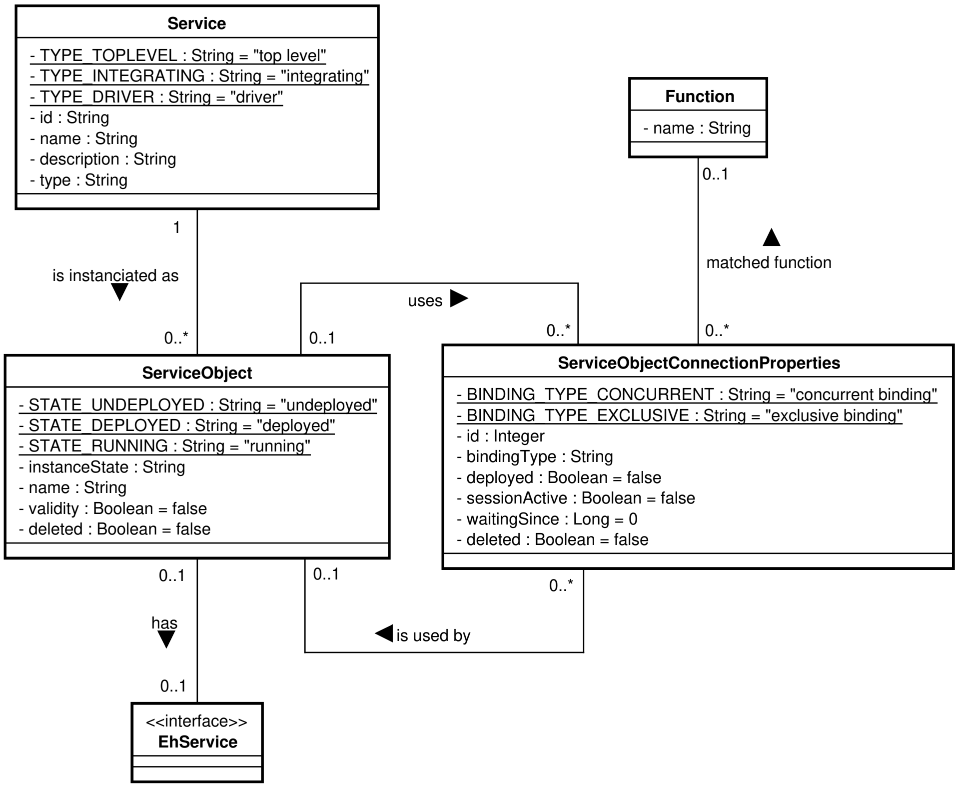


Abbildung 4.24: Modellierung von Dienstobjekten und -bindungen.

bungselement environmentElement lokalisiert ist, wie das serviceObject, von dem die zu untersuchende Bindung ausgeht. Falls die Umgebungselemente übereinstimmen, ist die Bedingung erfüllt und es wird true zurückgeliefert, anderenfalls wird false zurückgeliefert. Wenn das gebundene Dienstobjekt kein Treiberdienst ist, so werden in einer *for-each*-Aktivität alle indirekt gebundenen Treiberdienste driverServiceObject untersucht, d. h. die über eine Kette von Dienstbindungen erreichbare Treiberdienste. Die indirekt gebundenen Treiberdienste werden durch das Constraint {usedServiceObject.connectedTo(driverServiceObject)} ermittelt. Die Methode connectedTo bestimmt dabei, ob driverServiceObject über einen Pfad von Bindungen erreichbar ist. Für jeden auf diese Weise indirekt gebundenen Treiberdienst wird dann wie auch im ersten Fall geprüft, ob das angesteuerte Gerät im selben Umgebungselement lokalisiert ist, wie das serviceObject, von dem die Bindung ausgeht. Entsprechend wird dann true oder false zurückgeliefert.

Die Umsetzung der Dienstbindungen im eHome-Modell wurde bereits im Zusammenhang mit Abbildung 4.18 skizziert. In Abbildung 4.24 ist eine Detailansicht der relevanten Bereiche des eHome-Modells dargestellt. Oben in der Abbildung sind die bereits bekannten Klassen Service und Function dargestellt. Die Dienstobjekte werden durch die Klasse ServiceObject modelliert und sind durch die Kante is instanciated as mit der Service-Klasse verbunden. Jedes ServiceObject hat einen instanceState, der für das Deployment, das im Abschnitt 4.4.3 erläutert wird, von Bedeutung ist. Daneben hat es einen Namen (name), ein Attribut, das den Konfigurierungszustand des Dienstobjekts speichert (validity), sowie ein Attribut, mit dem zu löschende Dienstobjekte markiert werden (deleted), bevor die zugehörigen Objekte endgültig entfernt werden. Die zu einem Dienstobjekt gehörige Implementierung ist durch die has-Kante zwischen der Klasse ServiceObject und der Schnittstelle EhService realisiert. Diese Schnittstelle wird durch die Realisierungen der eHome-Dienste implementiert. Die Bindungen zwischen den Dienstobjekten werden durch Assoziationen der Klasse ServiceObject modelliert. Dabei wird von dem Dienstobjekt, von dem die Bindung ausgeht, eine Assoziation über die uses-Kante zu einem Objekt der Klasse ServiceObjectConnectionProperties hergestellt, welches dann weiter über die is used by-Kante auf das gebundene Dienstobjekt verweist. Der Zwischenknoten ServiceObjectConnectionProperties speichert die zu der Dienstbindung gehörenden Informationen. Die Funktionalität, auf dessen Basis die Bindung erfolgt ist, wird durch die matched function-Kante referenziert. Darüber hinaus gehört zu jeder Bindung eine id, die eine eindeutige Identifizierung ermöglicht, sowie der Bindungstyp bindingType (vgl. auch Abbildung 4.16). Das Attribut deployed speichert den Deploymentzustand der Bindung, das Attribut sessionActive den Zustand der Bindungssession, was für nebenläufige Bindungen von Bedeutung ist. Das Attribut waitingSince ist ebenfalls für die Behandlung nebenläufiger Bindungen von Belang. Das deleted-Attribut wird schließlich, wie auch in der Klasse ServiceObject, für zu löschende Bindungen benötigt, bevor diese endgültig entfernt werden.

4.4.3 Deployment

Als letzte Phase des SCD-Prozesses ist es die Aufgabe des Deployments, die zuvor erstellte Konfiguration zur Ausführung zu bringen. Dabei wird zu jedem Dienstobjekt in der Konfiguration eine *Dienstinstanz* erzeugt. Dienstinstanzen bezeich-

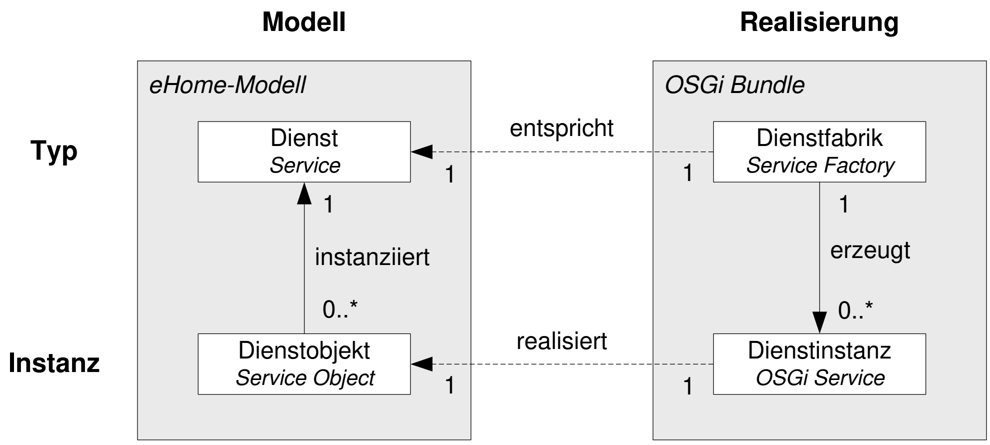


Abbildung 4.25: Zusammenhang von Modell- und Realisierungsebene.

nen hier die Instanzen der Dienstimplementierungen. Zur Umsetzung der Deploymentphase des SCD-Prozesses wurde eine *Deployer*-Komponente entwickelt. Genügte im bisherigen Ansatz noch das einmalige Erstellen der Dienstinstanzen in einer den Abhängigkeiten entsprechenden Reihenfolge, so gestaltet sich das Deployment im kontinuierlichen SCD-Prozess komplexer. Auch hier müssen nun Anpassungsschritte zur Laufzeit des eHome-Systems unterstützt werden. Wenn in der Konfiguration neue Dienstobjekte hinzugefügt oder entfernt werden, so müssen auch entsprechende Dienstinstanzen erzeugt oder wieder gelöscht werden. Die Dienstinstanzen müssen also mit der Konfiguration synchronisiert werden. Findet eine Änderung der Dienstbindungen statt, so müssen ebenfalls die laufenden Dienstinstanzen angepasst werden, indem neu erstellte Bindungen deployt oder entfernte Bindungen undeployt werden.

Realisierung von Diensten

Die Abläufe in der Deploymentphase sind abhängig von der Realisierung der Dienste und der Kommunikation und Interaktion zwischen ihnen. Daher wird an dieser Stelle zunächst das Zusammenspiel zwischen dem eHome-Modell und dem dadurch repräsentierten Zustand des eHome-Systems betrachtet. In Abbildung 4.25 sind die beiden Betrachtungsebenen *Modell* und *Realisierung* in Bezug auf eHome-Dienste dargestellt. Die Modellierung ist auf der linken Seite zu sehen, die Realisierung auf der rechten. Zusätzlich wird zwischen *Typen* und *Instanzen* unterschieden. Die Typebene liegt in der oberen Hälfte der Abbildung, während die Instanzebene unten dargestellt ist.

Der linke Teil der Abbildung betrifft das eHome-Modell, das die Spezifikation von Diensten und eHome-spezifischen Anforderungen sowie die Konfiguration des eHome-Systems repräsentiert. Im hier betrachteten Ausschnitt sind Dienste und Dienstobjekte die relevanten Entitäten. Ein Dienst wird durch Knoten vom Typ Service im eHome-Modell repräsentiert. Ein Dienstobjekt wird durch Knoten vom Typ Service Object repräsentiert. Jedes Dienstobjekt ist eine Instanz von genau einem Dienst, was durch die instanziiert-Kante modelliert wird. Ein Dienst kann durch beliebig viele Dienstobjekte instanziiert werden. Dies entspricht der bereits von NORBISRATH verwendeten Modellierung.

Der rechte Teil der Abbildung betrifft die Realisierung der Dienste und Dienstinstanzen auf Basis der Laufzeitumgebung. Um den Instanziierungsmechanismus im eHome-Modell auf der Realisierungsebene abzubilden, wird das Entwurfsmuster *Factory Method* (dt. *Fabrikmethode*) eingesetzt (vgl. [GHJV95]). Dieses Entwurfsmuster ist in Abbildung 4.26 dargestellt. Die abstrakte Klasse Product definiert die Klasse der von der Fabrikmethode zu erzeugenden Objekte. Die abstrakte Klasse Creator enthält die Fabrikmethode factoryMethod und ist für das Erzeugen von Objekten der Klasse Product verantwortlich. Die Methode factoryMethod der Klasse Creator ist ebenfalls abstrakt. Sie wird erst in der konkreten Erzeugerklasse ConcreteCreator implementiert und erzeugt dann Objekte der konkreten Klasse ConcreteProduct, die von Product abgeleitet ist. Mit Hilfe des Entwurfsmusters *Factory Method* wird eine Schnittstelle zum Erzeugen von Objekten zur Verfügung gestellt, die von der Laufzeitumgebung des eHome-Systems verwendet wird, um zu den Dienstobjekten aus dem eHome-Modell Dienstinstanzen auf der Realisierungsebene zu erzeugen.

Das eHome-Modell umfasst die Spezifikation der Dienste, der eHome-spezifischen Anforderungen sowie der Konfiguration des eHome-Systems. Dem gegenüber steht die Realisierung des im Modell repräsentierten Systemzustands. Die Realisierungsebene ist auf der rechten Seite in Abbildung 4.25 dargestellt. Im eHome-Modell wird durch die Konfigurierung ein Soll-Zustand modelliert. In der Deploymentphase wird der Ist-Zustand auf der Realisierungsebene an diesen Soll-Zustand angeglichen. Im Rahmen dieser Arbeit wird OSGi zur Implemen-

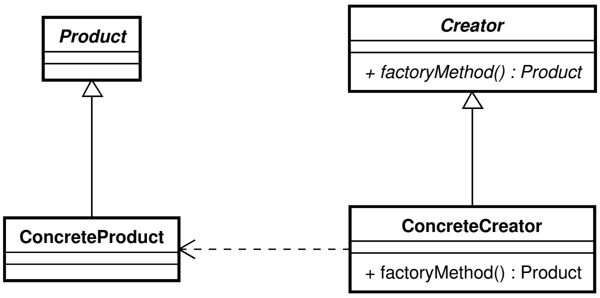
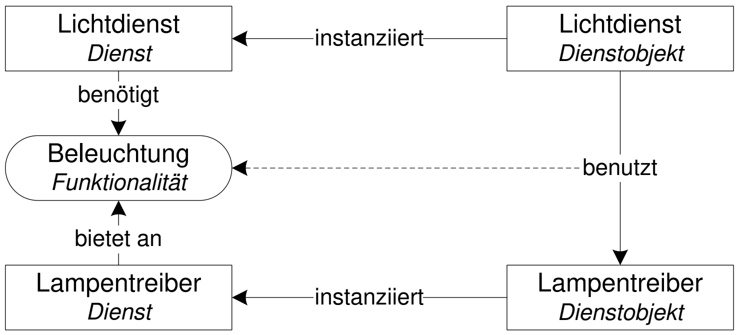


Abbildung 4.26: Entwurfsmuster *Factory Method*.

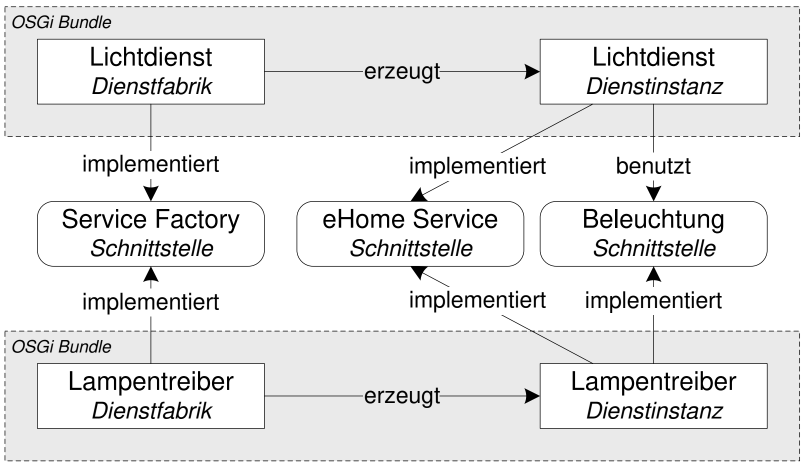
tierung der Dienste in Form von Softwarekomponenten eingesetzt. Dem dargestellten Modellausschnitt entspricht daher auf der Realisierungsebene ein OSGi Bundle, welches die Implementierung der Anwendungslogik enthält. Das Bundle umfasst eine Dienstfabrik, die dem Dienst im Modell entspricht. Die Dienstfabrik wiederum erzeugt Instanzen, die auf der Realisierungsebene Dienstinstanzen genannt werden. Diese Instanzen werden als *OSGi Services* bei der OSGi Service Registry registriert und entsprechen den Dienstobjekten auf der Modellebene. Jedem Dienstobjekt im Modell entspricht somit ein individueller OSGi Service in der Realisierung. Für die Erzeugung wird das oben beschriebene *Factory Method* Entwurfsmuster verwendet. Jedes OSGi Bundle, das einen eHome-Dienst implementiert, muss eine entsprechende Dienstfabrik mitbringen, die von der Laufzeitumgebung verwendet wird, um entsprechend der aktuellen Konfiguration Dienstinstanzen zu erzeugen oder zu löschen. Die spezifische Dienstfabrik, entspricht dem ConcreteCreator im *Factory Method* Entwurfsmuster. Die abstrakte Klasse Creator wird durch eine Schnittstelle umgesetzt. Die von der Dienstfabrik erzeugten Dienstinstanzen entsprechen dem ConcreteProduct im Entwurfsmuster. Auch hierzu gibt es eine Schnittstelle, die von allen eHome-Diensten implementiert wird, diese ist jedoch in der Abbildung der Einfachheit halber nicht dargestellt.

Realisierung von Dienstbindungen

In Abbildung 4.27 wird verdeutlicht, wie die Bindungen und die Kommunikation zwischen Dienstinstanzen umgesetzt werden und wie im Deployment die Komposition der Dienste erfolgt. In Abbildung 4.27(a) wird die Bindung zwischen



(a) Dienstbindungen im eHome-Modell.



(b) Realisierung der Dienstbindungen mit OSGi.

Abbildung 4.27: Modellierung und Realisierung von Dienstbindungen.

zwei Dienstobjekten auf der Modellebene dargestellt. Auf der linken Seite wird zunächst die Typebene betrachtet, die in Abbildung 4.25 in der oberen Hälfte dargestellt war. Im hier verwendeten Beispiel wird ein Lichtdienst betrachtet, der die Beleuchtung eines Raumes steuert. Dieser Dienst benötigt daher die Funktionalität Beleuchtung, die von anderen Diensten zur Verfügung gestellt werden muss. Im dargestellten Fall bietet der Treiberdienst Lampentreiber diese Funktionalität an. Auf der rechten Seite ist die Instanzebene dargestellt, die in Abbildung 4.25 in der unteren Hälfte zu sehen war. Von den beiden Diensten Lichtdienst und Lampentreiber ist jeweils eine Dienstobjekt erzeugt worden. Diese beiden Dienstobjekte sind in der dargestellten Konfiguration durch eine

Bindung, die auf der Funktionalität Beleuchtung basiert, miteinander verbunden. Diese Bindung wird durch die Kante benutzt dargestellt.

In Abbildung 4.27(b) wird die Realisierung der oben beschriebenen Modellierung dargestellt. Für die beiden Dienste Lichtdienst und Lampentreiber gibt es jeweils ein OSGi Bundle, welches die Implementierung des Dienstes enthält. Zu jedem der Dienste gibt es eine zugehörige *Dienstfabrik* innerhalb des OSGi Bundles, die wie oben beschrieben für die Umsetzung des *Factory Method* Entwurfsmusters benötigt wird. Jede *Dienstfabrik* implementiert eine entsprechende Schnittstelle, damit sie vom Deployer des eHome-Systems angesteuert und verwaltet werden kann. Die Dienstfabrik erstellt dann eine Dienstinstanz für jedes Dienstobjekt aus dem Modell.

Die Dienstinstanzen implementieren die Schnittstelle eHome Service, damit sie vom Deployer verwaltet werden können. Dies ist insbesondere zur Steuerung der Dienstbindungen wichtig. Dienstbindungen werden ebenfalls über eine Schnittstelle realisiert. Diese entspricht der Funktionalität auf der die jeweilige Bindung basiert. In diesem Fall wird die Schnittstelle Beleuchtung verwendet, da die Bindung zwischen Lichtdienst und Lampentreiber über diese Funktionalität realisiert wird. Der Lichtdienst benutzt diese Schnittstelle, da er die Funktionalität Beleuchtung nutzt. Der Lampentreiber hingegen bietet diese Funktionalität an. Daher implementiert er die Schnittstelle Beleuchtung. Eine gemeinsame Schnittstelle bedeutet, dass eine gemeinsame Funktionalität vorliegt. Dann kann eine Bindung hergestellt werden. Die gebundenen Dienstinstanzen können über die Schnittstelle miteinander kommunizieren.

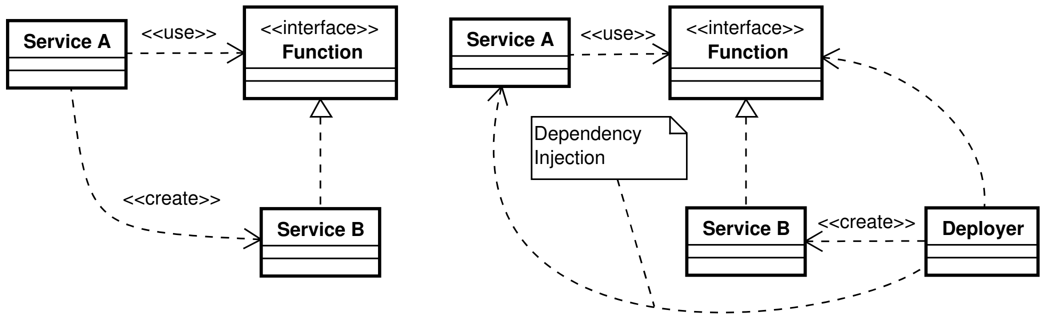
Die Umsetzung der funktionalen Dienstkomposition auf Grundlage von Schnittstellen steht im Gegensatz zum Deployment im Ansatz von NORBISRATH, in dem Bindungen nicht durch die Laufzeitumgebung verwaltet wurden. Stattdessen wurden die States verwendet (vgl. Abschnitt 3.4.3), die zur Modellierung der Laufzeitzustände dienten. Dabei trat das Problem auf, dass Dienstentwickler bei der Entwicklung das eHome-Modell und insbesondere die Zustände möglicher zu bindender Dienste kennen mussten. Innerhalb der Dienstimplementierung musste auf die im eHome-Modell repräsentierten Dienstbindungen und Zustände Bezug genommen werden. Der Typ und das Format der ausgetauschten Daten musste auf informeller Basis vereinbart werden, damit in allen an einer Bindung beteiligten Dienstimplementierungen darauf zurückgegriffen werden konnte.

Da eine solche Vereinbarung in der Realität schwer zu erreichen ist, wird im Rahmen dieser Arbeit die Komposition auf Basis von Java-Schnittstellen durchgeführt. Dies bietet sich an, da der Umgang mit Schnittstellen für den Dienstentwickler keine Besonderheit darstellt, für die eine spezielle Einarbeitung erforderlich wäre. Zudem ist die Komposition auf Basis von Java-Schnittstellen auch im Kontext von OSGi die übliche Vorgehensweise.

Schnittstellen ermöglichen die formale Festlegung der auszutauschenden Daten und der verfügbaren Operationen. Sie werden bei der Kompilation des Quellcodes überprüft und bieten dadurch eine Typsicherheit, die bei der Verwendung der States nicht gegeben war. Die internen Laufzeitzustände sind im eHome-Modell nicht mehr sichtbar, sondern in den Dienstimplementierungen verkapselt. Sie können daher nicht mehr durch fehlerhafte oder böswillige Dienste missbräuchlich manipuliert werden.

Für das Deployment des erweiterten Entwicklungsprozesses ist es nicht mehr erforderlich, dass in der Dienstimplementierung auf das Modell zugegriffen wird und die Informationen über Bindungen zu anderen Dienstinstanzen selbst ermittelt werden. Ein solcher Zugriff wird sogar explizit ausgeschlossen, das Modell soll vor den Dienstimplementierungen verborgen bleiben. Dies stellt nicht nur eine wesentliche Vereinfachung dar, sondern dient auch der Sicherheit des eHome-Systems vor fehlerhafter oder missbräuchlicher Verwendung des eHome-Modells. Da die Dienstinstanzen Bindungen zu anderen Dienstinstanzen nicht mehr selbständig herstellen, muss dies auf anderem Wege geschehen. Im erweiterten Ansatz werden den Dienstinstanzen daher die Bindungen „injiziert". Dazu erhalten sie über eine in der Schnittstelle der Dienstimplementierung festgelegte Methode eine Referenz auf eine Dienstinstanz mitgeteilt, die für die Nutzung durch den Dienst zur Verfügung steht. Zur Realisierung dieses Mechanismus wird das Entwurfsmuster *Dependency Injection* eingesetzt.

Das Entwurfsmuster *Dependency Injection* [Fow04] wird verwendet, um eine Anwendung aus verschiedenen Komponenten zusammenzusetzen. Die Konfigurierung der Anwendung wird dabei von der Benutzungsbeziehung zwischen den Komponenten getrennt betrachtet. Da die Abhängigkeiten der Komponenten extern verwaltet werden, wird hierbei auch von *Inversion of Control* gesprochen. In Abbildung 4.28 ist das Entwurfsmuster *Dependency Injection*, so wie es im Kontext des Deployments im SCD-Prozess eingesetzt wird, dargestellt.



(a) Klassische Instanziierung. (b) Bindung mit *Dependency Injection*.

Abbildung 4.28: Entwurfsmuster *Dependency Injection*.

Zunächst ist in Abbildung 4.28(a) der Fall der klassischen Instanziierung einer verwendeten Klasse ohne *Dependency Injection* dargestellt. Hier benötigt eine Dienstinstanz Service A die Funktionalität Function, dargestellt durch die entsprechende Schnittstelle. Da Service B diese Funktionalität anbietet, d. h. die Schnittstelle implementiert, würde Service A üblicherweise die Klasse Service B instanziieren, um die Funktionalitäten der Klasse nutzen zu können.

Diese klassische Vorgehensweise ist jedoch für eHome-Dienste nicht anwendbar, da zur Entwicklungszeit nicht bekannt ist, welche Dienste zur Laufzeit verfügbar sein werden. Es sind lediglich Schnittstellen bekannt, die eine Funktionalität beschreiben. In der Implementierung von Service A kann daher nicht auf die Klasse Service B Bezug genommen werden, es kann nur die Schnittstelle Function adressiert werden. Außerdem kann es sein, dass bereits eine Dienstinstanz zu Service B existiert, die von anderen Dienstinstanzen verwendet wird. Es könnte sich z. B. um eine Treiberdienstinstanz handeln, die ein ganz bestimmtes Gerät im eHome ansteuert. Service A kann daher nicht ohne Weiteres eine neue Instanz von Service B erzeugen und für seine benötigten Funktionalitäten verwenden. Stattdessen übernimmt der Deployer der eHome-Laufzeitumgebung die Aufgabe der Verwaltung der Dienstinstanzen. Im Ansatz von NORBISRATH wurden alle Zustandsinformationen im Rahmen der States im eHome-Modell erfasst, daher waren keine Dienstinstanzen, wie sie hier verwendet werden, zu erstellen.

In Abbildung 4.28(b) ist die Verwaltung der Dienstbindungen durch den Deployer bei Verwendung von *Dependency Injection* dargestellt. Hier werden Dienstin-

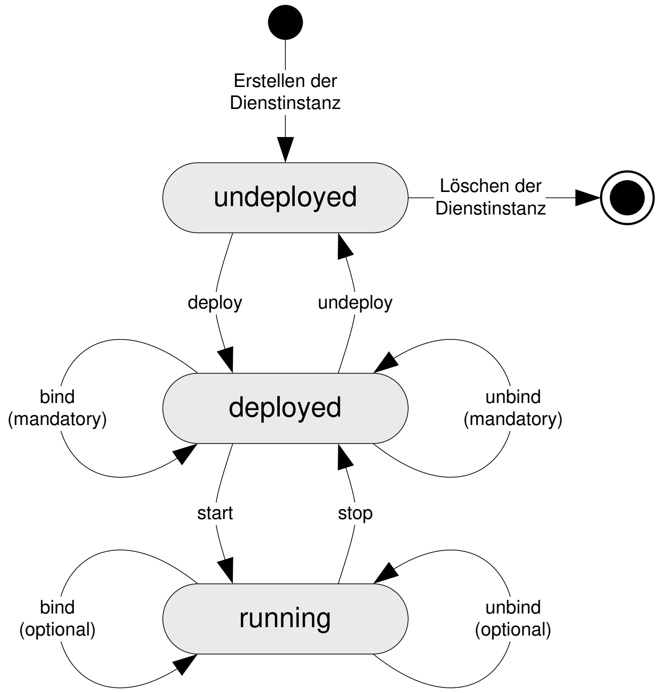


Abbildung 4.29: Zustandsdiagramm für das Deployment von eHome-Diensten.

stanzen und Bindungen extern verwaltet. Service A erzeugt nicht selbst eine Instanz von Service B, sondern überlässt diesen Schritt der Klasse Deployer. Damit Service A auf Service B zugreifen kann, übermittelt der Deployer eine Referenz auf Service B an Service A, indem eine spezielle Methode in Service A aufgerufen wird. Dieser Schritt stellt die eigentliche „Injektion" der Abhängigkeit dar. Service A muss Service B nicht kennen und braucht sich auch nicht mit der Suche nach einer passenden Dienstinstanz befassen, diese Aufgaben übernimmt der Deployer. Falls bereits eine geeignete Dienstinstanz von Service B vorhanden ist, kann diese auch direkt verwendet werden, es muss also nicht zwangsläufig eine neue Instanz durch den Deployer erzeugt werden.

Deploymentzustände eines eHome-Dienstes

Zur Unterstützung der Deploymentphase wird jedem Dienstobjekt und jeder Bindung in der Konfiguration ein *Deploymentzustand* zugewiesen. Dieser Deploymentzustand ist abhängig vom Konfigurierungszustand des Dienstobjekts, der in Abschnitt 4.4.2 bereits erläutert wurde. Es wird für Dienstobjekte zwischen den drei Zuständen undeployed, deployed und running unterschieden.

In Abbildung 4.29 ist ein entsprechendes Zustandsdiagramm dargestellt. Bei Dienstbindungen wird lediglich zwischen den Zuständen undeployed und deployed unterschieden. Wird ein Dienstobjekt neu erzeugt, so befindet es sich zunächst im Zustand undeployed. Das bedeutet, dass das bereits konfigurierte Dienstobjekt noch nicht deployt worden ist. Obwohl es in der Konfiguration angelegt wurde, existiert noch keine Dienstinstanz. Wenn der Deployer eine Dienstinstanz für das Dienstobjekt erzeugt (Vorgang deploy), so erfolgt ein Übergang zum Zustand deployed. Nun können Bindungen zu anderen Dienstinstanzen durch den oben beschriebenen Mechanismus der *Dependency Injection* erstellt werden. Dies ist durch den Vorgang bind in der Abbildung dargestellt. Wenn die Dienstinstanz gestartet wird, erfolgt der Übergang in den Zustand running, in dem der Dienst ausgeführt wird und seine Funktionalität zur Verfügung stellt. Der Vorgang start bezeichnet diesen Vorgang in der Abbildung. Auch im Zustand running können weitere Bindungen durch bind-Vorgänge hinzugefügt werden, sofern die Dienstspezifikation dies vorsieht. Bindungen können außerdem durch unbind-Vorgänge wieder entfernt werden. Der Zustand running kann dabei beibehalten werden, wenn es sich um optionale Bindungen handelt. Wird die Dienstinstanz gestoppt (Vorgang stop), so wechselt der Zustand erneut zu deployed. Durch weitere unbind-Vorgänge können hier die verbleibenden Bindungen entfernt werden. Schließlich kann durch den undeploy-Vorgang die Dienstinstanz wieder entfernt werden. Danach kann ggfs. auch das zugehörige Dienstobjekt aus der Konfiguration entfernt werden.

Der Konfigurierungszustand und der Deploymentzustand eines Dienstobjekts hängen miteinander zusammen. Die aus beiden Zuständen resultierenden Aktionen des Deployers sind in Tabelle 4.1 zusammengefasst. Ist ein Dienstobjekt *gültig* konfiguriert aber *undeployed*, so wird eine Dienstinstanz erzeugt und kann anschließend, nach dem Erstellen der erforderlichen Bindungen, auch gestartet werden. Falls das Dienstobjekt *gültig* und bereits *deployed* ist, so kann, wieder nach dem Erstellen der Bindungen, die Dienstinstanz gestartet werden. Für ein *gültig* konfiguriertes Dienstobjekt, das sich bereits im Zustand *running* befindet, braucht nichts unternommen zu werden.

Konfigurierung	Deployment	Adaptionsschritt
gültig	undeployed	Dienstinstanz erzeugen und starten
gültig	deployed	Dienstinstanz starten
gültig	running	keine
ungültig	undeployed	Dienstinstanz erzeugen
ungültig	deployed	keine
ungültig	running	Dienstinstanz stoppen

Tabelle 4.1: Adaptionsschritte im Deployment.

Auch für ein *ungültig* konfiguriertes Dienstobjekt kann bereits eine Dienstinstanz erzeugt werden. Aufgrund der fehlenden benötigten Ressourcen kann dieses allerdings nicht gestartet werden. Ist das Dienstobjekt *ungültig* und bereits *deployed*, so braucht erneut nichts unternommen werden, da die Dienstinstanz in dieser Situation nicht gestartet werden kann. Befindet sich ein *ungültig* konfiguriertes Dienstobjekt im Deploymentzustand *running*, so muss die Dienstinstanz gestoppt werden, da entsprechend der Konfiguration bestimmte erforderliche Ressourcen nicht länger genutzt werden können. Erst wenn das Dienstobjekt wieder *gültig* konfiguriert werden kann, ist ein Starten der Dienstinstanz erneut möglich.

Die oben beschriebenen Zustandsübergänge werden in der Deploymentphase durch die Deployer-Komponente der eHome-Laufzeitumgebung umgesetzt. Auf diese Weise werden die Dienstinstanzen und damit der Laufzeitzustand des eHome-Systems mit der aktuellen Konfiguration im eHome-Modell abgeglichen. Im Folgenden wird genauer beschrieben, wie dieser Abgleich für eine gegebene Konfiguration durchgeführt wird.

Ablauf der Deploymentphase

Abbildung 4.30 stellt den Algorithmus zur Adaption in der Deploymentphase in Form von Flussdiagrammen dar. Für die Adaptionsschritte, die in Tabelle 4.1 zusammengefasst sind, ist es entscheidend, dass die richtige Reihenfolge eingehalten wird. Nur so kann sichergestellt werden, dass alle Abhängigkeiten der gestarteten Dienstinstanzen zur Laufzeit auch erfüllt werden. Dazu gibt es zwei wichtige Regeln, die entsprechend der Tabelle 4.1 beim Ablauf eingehalten werden müssen.

- Eine Dienstinstanz kann erst gestartet werden, wenn alle Dienstinstanzen, die von ihr genutzt werden, zuvor deployt und gestartet wurden.

- Eine Dienstinstanz kann erst gestoppt und undeployt werden, wenn alle Dienstinstanzen, die sie nutzen, zuvor gestoppt wurden.

Um diese Regeln zu erfüllen, werden Dienstobjekte in der Deploymentphase gemäß der in Abbildung 4.30 dargestellten Vorgehensweisen bearbeitet. In Abbildung 4.30(a) wird zunächst die sogenannte *Aufwärtsprüfung* beschrieben. Diese wird ausgeführt, wenn ein Dienstobjekt gestoppt oder undeployt wird. Analog dazu gibt es die *Abwärtsprüfung*, die verwendet wird, wenn ein Dienstobjekt deployt oder gestartet wird. Die Abwärtsprüfung ist in Abbildung 4.30(b) dargestellt.

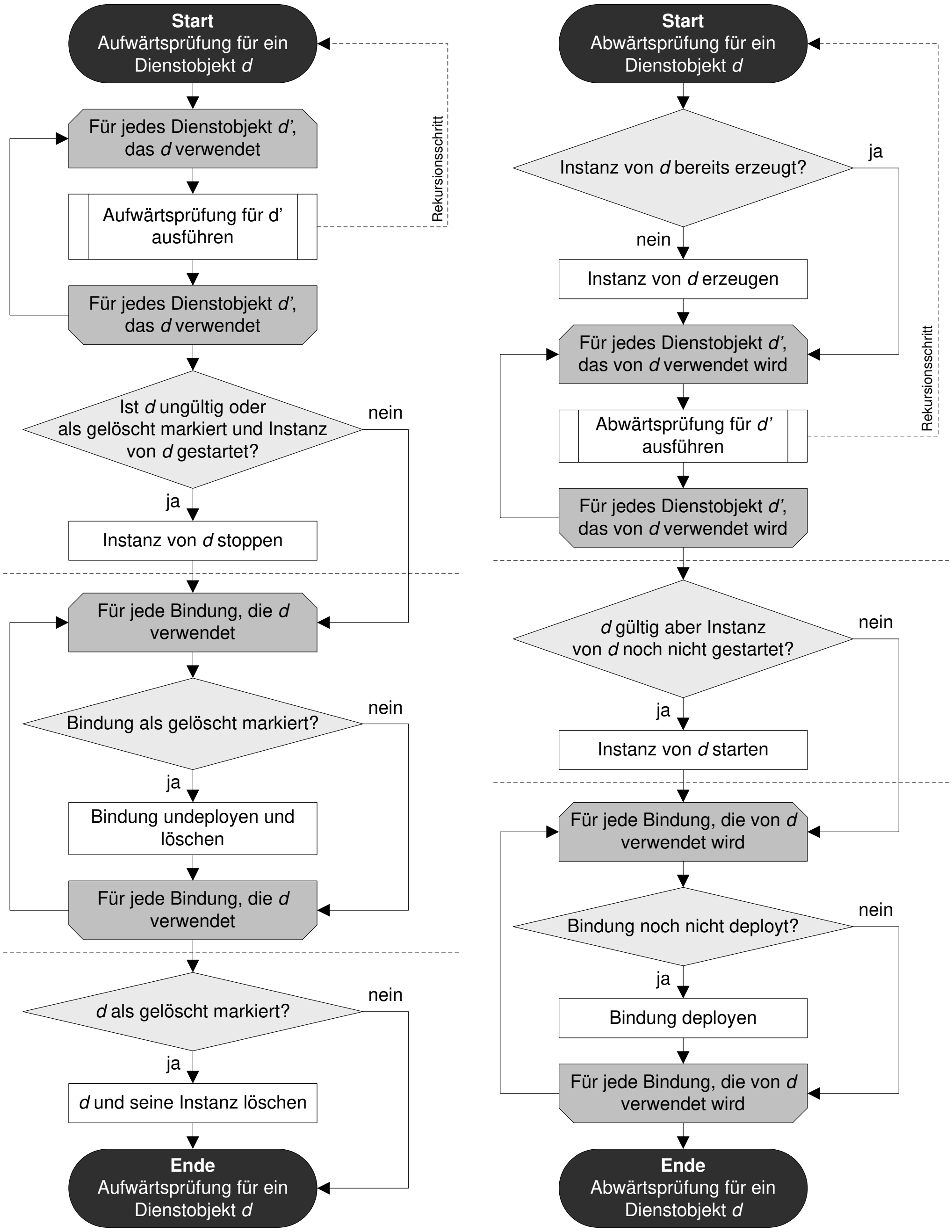
Die Deploymentphase muss auf Adaptionsschritte bei der Konfigurierung reagieren und diese umsetzen. Durch Rekonfigurierung können etwa Dienstobjekte erzeugt oder gelöscht werden, ihr Konfigurierungszustand kann sich verändern, es können neue Bindungen erzeugt oder bestehende gelöscht werden. Beim Löschen von Dienstobjekten ist eine Besonderheit zu beachten. Damit auch für gelöschte Dienstobjekte eine Deploymentanpassung möglich ist – es muss z. B. die Dienstinstanz entsprechend entfernt werden – erhält das zu löschende Dienstobjekt zunächst eine entsprechende Markierung. Dadurch ist eine Weiterverarbeitung möglich und das Dienstobjekt kann gelöscht werden, nachdem alle noch auszuführenden Schritte im Deployment abgearbeitet wurden. Um das Deployment einer geänderten Konfiguration entsprechend den oben genannten Regeln anzupassen, werden die folgenden Schritte ausgeführt, welche die Aufwärts- und die Abwärtsprüfung beinhalten:

1. Für alle als gelöscht markierten Dienstobjekte wird die Aufwärtsprüfung durchgeführt.

2. Alle noch nicht im ersten Schritt geprüften Bindungen, die als gelöscht markiert sind, werden undeployt und gelöscht.

3. Für alle Dienstobjekte wird die Abwärtsprüfung durchgeführt.

4. Alle Bindungen, die noch undeployt sind, werden deployt.



(a) Aufwärtsprüfung.

(b) Abwärtsprüfung.

Abbildung 4.30: Adaptionsalgorithmus für die Deploymentphase.

Die im ersten Schritt durchzuführende Aufwärtsprüfung ist in Abbildung 4.30(a) im Detail beschrieben. Der Ablauf ist in drei Phasen unterteilt, die durch gestrichelte Linien getrennt dargestellt sind. In der ersten Phase wird für das zu prüfende Dienstobjekt d über alle Dienstobjekte d' iteriert, die d verwenden, um ihre Funktionalität zu realisieren. Für diese Dienstobjekte d' wird zunächst selbst eine Aufwärtsprüfung ausgeführt. Auf diese Weise wird der Vorgang rekursiv bis hin zu den Top-Level-Diensten in der Konfiguration durchgeführt, bevor mit d fortgefahren wird. Ist d ungültig konfiguriert oder als gelöscht markiert, aber eine Dienstinstanz zu d ist vorhanden und befindet sich in der Ausführung, so wird diese gestoppt. In der zweiten Phase werden alle Bindungen überprüft, die zu d führen, d. h. zwischen Dienstobjekten, die d verwenden, und d. Falls eine solche Bindung als gelöscht markiert ist, so wird sie undeployt und dann aus der Konfiguration entfernt. Schließlich werden in der dritten Phase, falls d als gelöscht markiert ist, die zu d gehörige Dienstinstanz so wie auch das Dienstobjekt d selbst gelöscht.

Die drei Phasen der in Schritt drei des obigen Ablaufs durchzuführenden Abwärtsprüfung sind ähnlich aufgebaut und werden in Abbildung 4.30(b) dargestellt. Während es bei der Aufwärtsprüfung um Löschoperationen geht, so geht es bei der Abwärtsprüfung um die entsprechenden Erzeugungsoperationen. Zunächst wird in der ersten Phase überprüft, ob für d bereits eine Dienstinstanz vorhanden ist. Wenn dies nicht der Fall ist, so wird eine solche erzeugt. Dann wird über alle Dienstobjekte d' iteriert, die von d verwendet werden. Für alle d' wird nun ebenfalls rekursiv die Abwärtsprüfung gestartet, sodass alle Dienstobjekte bis zu den von d aus verwendeten Basisdiensten abgearbeitet werden. Danach wird in der zweiten Phase geprüft, ob d gültig konfiguriert ist aber die zugehörige Dienstinstanz noch nicht gestartet wurde. Diese wird dann ggfs. gestartet. In der dritten Phase der Abwärtsprüfung werden die Bindungen untersucht, die von d verwendet werden, d. h. zwischen d und Dienstobjekten, die d verwendet. Falls eine solche Bindung noch nicht deployt wurde, so wird dies an dieser Stelle getan. Sind alle von d verwendeten Bindungen deployt, so ist die Abwärtsprüfung abgeschlossen.

Die Schritte zwei und vier des oben beschriebenen Ablaufs beziehen sich auf die verbleibenden Bindungen, die noch nicht im Rahmen der Aufwärts- und Abwärtsprüfung abgedeckt wurden. In Schritt zwei werden solche Bindungen, wenn sie als gelöscht markiert sind, undeployt und aus der Konfiguration gelöscht. In Schritt vier werden in der Konfiguration enthaltene Bindungen deployt, sofern dies noch nicht geschehen ist.

Im Deployment wird neben den oben behandelten Abläufen auch eine Parametrisierung der Dienstinstanzen durchgeführt. Dienste können mit verschiedenen konfigurierbaren Attributen versehen sein. Durch Attribute kann das Verhalten eines Dienstes beeinflusst werden. Die Attributwerte der Dienstobjekte können zur Laufzeit mittels eines Werkzeugs (siehe Abschnitt 6.3) durch den Benutzer oder einen Administrator jederzeit angepasst werden.

In der Dienstimplementierung werden die Attribute verwendet, die Abläufe der Anwendungslogik des Dienstes sind somit von den aktuellen Attributwerten abhängig. Beim Deployment müssen die aktuellen Werte der Attribute eines Dienstobjekts berücksichtigt werden. Die zugehörigen Dienstinstanzen müssen daher in Bezug auf die momentanen Attributwerte mit der Konfiguration abgeglichen werden. Die Implementation eines Dienstes muss für die spezifizierten Attribute Methoden zum Schreiben und Lesen beinhalten, um einen Zugriff auf die Attributwerte der Dienstinstanz zu ermöglichen. Diese Methoden werden vom Deployer verwendet, um die Instanz entsprechend der aktuellen Konfiguration zu parametrisieren. Attributwerte werden dabei in die Dienstinstanz „injiziert“, ähnlich der Vorgehensweise bei der *Dependency Injection* für das Deployment der Dienstbindungen.

Realisierung

In Abbildung 4.31 ist das Story-Diagramm zur Methode adjustDeployment der Deployer-Klasse dargestellt. Diese Methode ist der Einstiegspunkt für die Anpassung des Deployments nach Änderungen in den vorangehenden Phasen des SCD-Prozesses. Die Aufwärts- und Abwärtsprüfung, die im Zusammenhang mit Abbildung 4.30 beschrieben wurden, werden dabei aufgerufen. Im ersten Schritt wird für alle als gelöscht markierten Dienstobjekte die Aufwärtsprüfung durchgeführt. Dazu wird ausgehend von serviceRoot für alle Dienstobjekte serviceObject mit deleted == true die Methode checkUpward für die Aufwärtsprüfung aufgerufen. Im nächsten Schritt wird für alle Dienstobjekte, die durch als gelöscht markierte Bindungen gebunden sind, selbst aber nicht als gelöscht markiert sind, die Aufwärtsprüfung durchgeführt. Im gesuchten Muster ist daher das Attribut deleted

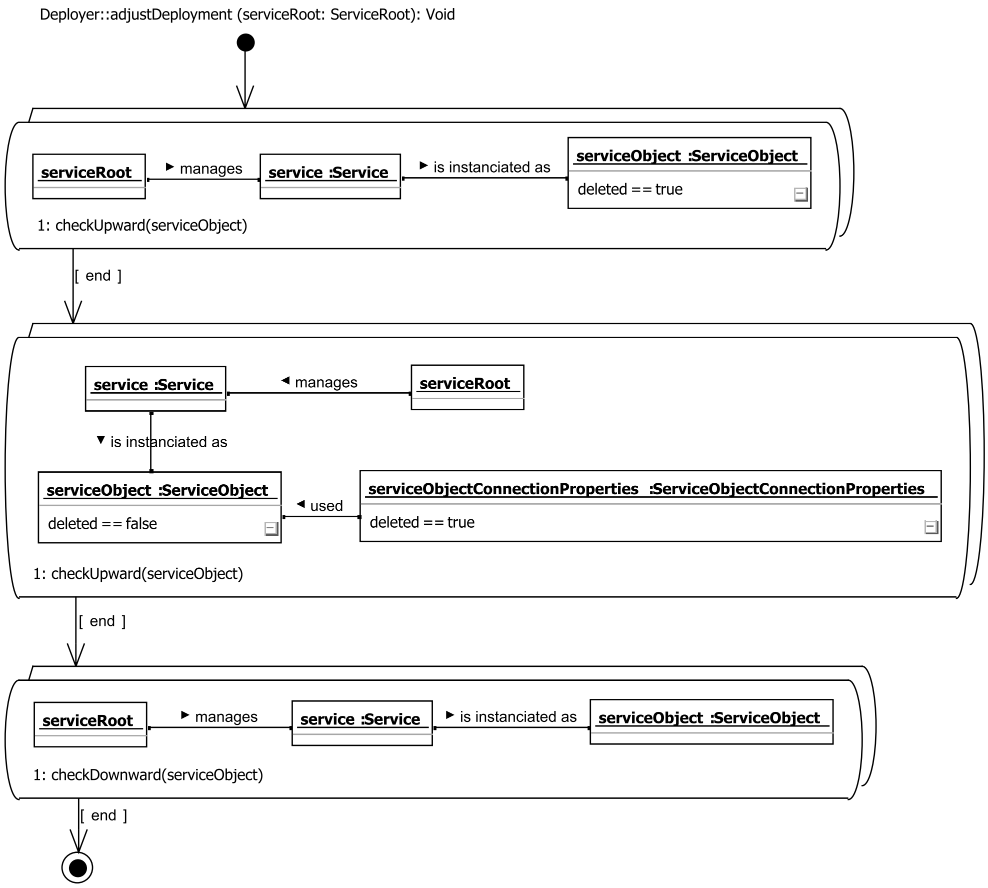


Abbildung 4.31: Anpassung des Deployments.

von serviceObject gleich false. Die Bindung wird durch das Objekt serviceObject-ConnectionProperties repräsentiert, dabei muss das Attribut deleted im gesuchten Muster gleich true sein. Auf diese Weise können die Bindungen undeployt werden. In der letzten Aktivität des Diagramms wird für alle Dienstobjekte serviceObject die Methode checkDownward für die Abwärtsprüfung aufgerufen. So wird sichergestellt, dass alle noch nicht deployten Dienstobjekte und Bindungen deployt werden.

In Abbildung 4.32 ist exemplarisch das Story-Diagramm der Methode checkUpward zur Aufwärtsprüfung dargestellt. Die Implementierung von checkDownward erfolgt analog und wird daher hier nicht im Detail besprochen. Der grundsätzliche Ablauf der Aufwärtsprüfung wurde bereits in Abbildung 4.30(a) be-

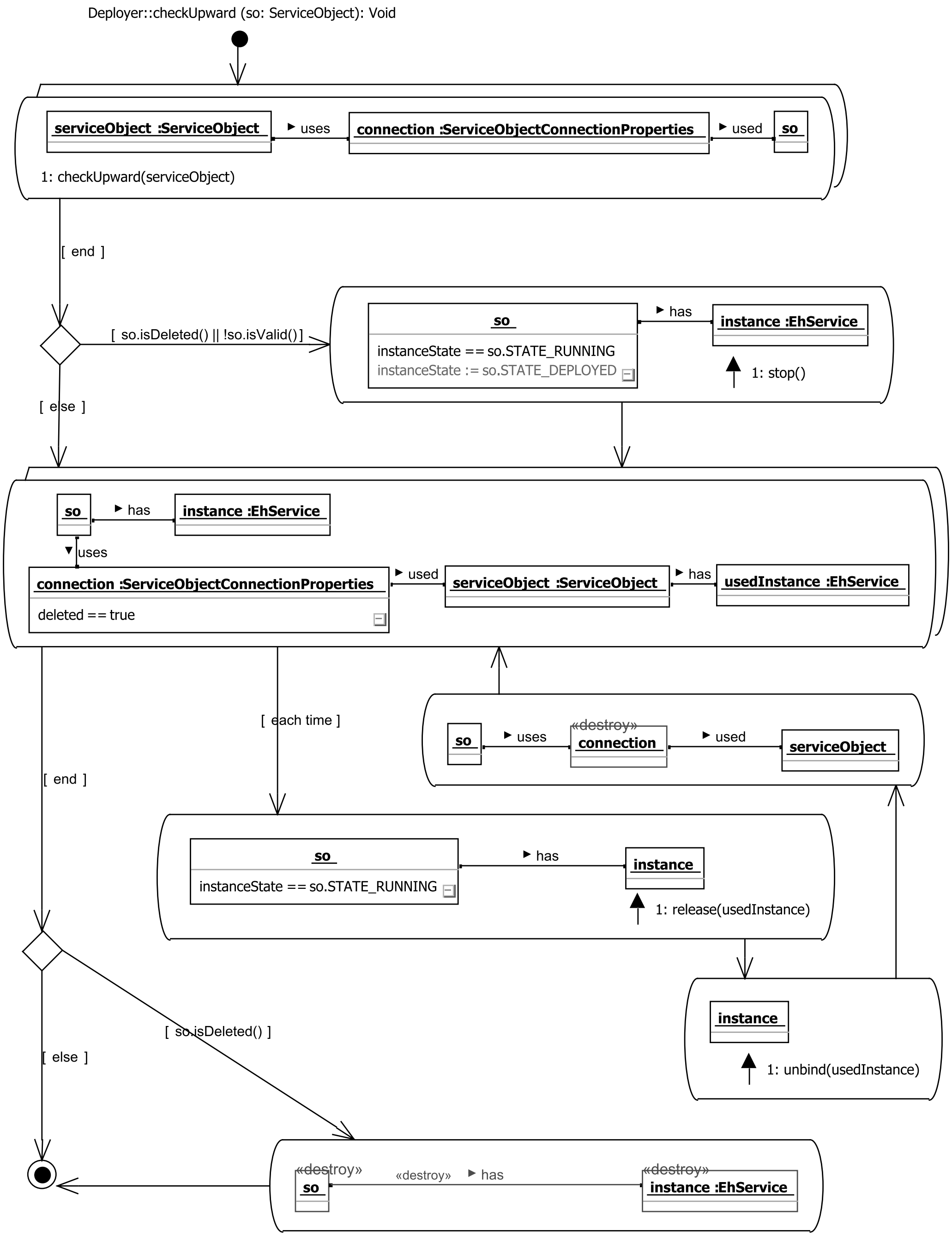


Abbildung 4.32: Realisierung der Aufwärtsprüfung.

schrieben. Die Methode wird bezogen auf ein Dienstobjekt so aufgerufen. Im ersten Schritt wird für alle Dienstobjekte serviceObject, die so benutzen, rekursiv ebenfalls die Methode checkUpward aufgerufen. Dann wird, falls so als gelöscht markiert (so.isDeleted()) oder ungültig ist (!so.isValid()), der Deploymentzustand von *running* auf *deployed* zurückgesetzt und es wird die Methode stop der zu so gehörigen Dienstimplementierung instance aufgerufen. Im nächsten Schritt werden die von so benutzen Dienstobjekte untersucht, die über als gelöscht markierte Bindungen gebunden sind. Für diese Bindungen wird im Graphen jeweils nach einem Objekt connection der Klasse ServiceObjectConnectionProperties gesucht, für das deleted gleich true ist. Falls so im Deploymentzustand *running* ist, wird für die zugehörige Dienstimplementierung instance die Methode release für die gebundene Dienstimplementierung usedInstance aufgerufen. Dadurch kann instance die Nutzung von usedInstance kontrolliert beenden. Danach wird mittels der unbind-Methode die Bindung undeployt und es wird das connection-Objekt gelöscht, was durch den Stereotyp «destroy» gekennzeichnet ist. Nachdem dieser Ablauf für alle als gelöscht markierten Bindungen durchgeführt wurde, beginnt der letzte Teil der checkUpward-Methode. Dabei werden, falls das Dienstobjekt so als gelöscht markiert ist, sowohl so als auch die zugehörige Dienstimplementierung instance gelöscht.

4.4.4 Ausführung

Nach Abschluss der Deploymentphase befindet sich das eHome-System in der Ausführung. In dieser Phase befindet sich das eHome für den größten Teil der Laufzeit, nämlich wenn alle Änderungen erfasst und verarbeitet wurden und die laufenden Dienstinstanzen ihre Funktionalitäten umsetzen. Die Gesamtfunktionalität des eHome-Systems ergibt sich aus der deployten Dienstkomposition gemäß den spezifizierten Wünschen und Anforderungen der Benutzer. Auch in der Ausführungsphase laufen Kontrollprozesse des Laufzeitsystems ab, da sich auch durch die Abläufe der Anwendungslogik der Dienstinstanzen die Notwendigkeit zur Adaption ergeben kann. Dies ist beispielsweise im Zusammenhang mit den in Abschnitt 4.3.4 eingeführten nebenläufigen Bindungen erforderlich.

Für die Umsetzung nebenläufiger Dienstbindungen muss eine Überwachung der Dienstkommunikation stattfinden, damit ermittelt werden kann, welche dieser

Bindungen momentan genutzt werden und welche nicht. Dienstinstanzen kommunizieren mittels Methodenaufrufen. Ein Dienst, der eine Funktionalität anbietet, implementiert eine entsprechende Schnittstelle. Dienste, die diese Funktionalität nutzen, greifen auf die entsprechenden Schnittstellenmethoden einer gebundenen Instanz zu. Auf Basis der Methodenaufrufe kann die Laufzeitumgebung des eHomes feststellen, wann eine Bindung genutzt wird.

Sessions

Da die Abarbeitung eines Methodenaufrufs nicht mit einem Nutzungsintervall einer bestimmten Funktionalität gleichgesetzt werden kann, wurde bereits in Abschnitt 4.3.4 ein *Session*-Konzept eingeführt. Die Kommunikation zwischen Diensten beschränkt sich im Wesentlichen auf den Austausch von Steuersignalen. Diese werden sehr schnell verarbeitet, sodass eine Nutzung auf Grundlage dieses Signalaustauschs immer nur punktuell stattfinden würden. Natürlich ist die Dauer der Methodenausführung auch davon abhängig, in welcher Weise ein bestimmter Dienst implementiert ist und welche Aufgaben er erfüllt. Um den unterschiedlichen Fällen gerecht zu werden, muss die Nutzung einer Ressource auf andere Weise bestimmt werden. Im Rahmen dieser Arbeit werden Sessions durch Annotationen der Schnittstellenmethoden ermittelt. Diese Annotationen werden bei der Dienstspezifikation vorgenommen und legen fest, ob der Aufruf einer bestimmten Methode eine Session startet oder eine Session beendet. Eine Session kann also von einem Methodenaufruf bis zu einem anderen Methodenaufruf dauern. Auf diese Weise lässt sich ein Großteil der Fälle handhaben, in denen sich eine Nutzungsphase über mehrere Methodenaufrufe erstreckt.

Am Beispiel eines Musikdienstes lässt sich das Konzept der Sessions veranschaulichen. Ein personenbezogener Musikdienst wählt Musikstücke aus, die der Benutzer in der gegeben Umgebung und zur gegebenen Zeit gerne hört. Um die Musik auszugeben, muss eine Ressource in der aktuellen Umgebung gebunden werden, die die Funktionalität Audioausgabe anbietet, z. B. ein Dienst zur Lautsprechersteuerung, wie in Abbildung 4.15. Durch Methodenaufrufe kann der Musikdienst die gebundenen Lautsprecher steuern, er kann z. B. durch entsprechende Signale das Abspielen einer Audioquelle starten. In diesem Fall beginnt eine Session, da das Abspielen einer neuen Audioquelle als Start eines Nutzungsintervalls interpretiert wird. Durch eine entsprechende Annotation der zugehörigen Methode wird dies gekennzeichnet, sodass die Laufzeitumgebung des eHomes eine neue Session angelegen kann. Durch einen anderen Methodenaufruf wird das Abspielen der Audioquelle gestoppt. Durch diesen Methodenaufruf wird die Session wieder geschlossen, da die Nutzung beendet ist. Auf diese Weise kann die Laufzeitumgebung ermitteln, wann eine nebenläufige Bindung von einer Dienstinstanz genutzt wird. Die benötigten Annotationen der Methoden der Dienstimplementierung werden im Rahmen der Dienstspezifikation vorgenommen. Dazu wird ein entsprechendes in dieser Arbeit entwickeltes Werkzeug (siehe Abschnitt 6.3) verwendet.

Prioritäten

Für die nebenläufige Nutzung von Ressourcen wurde in Abschnitt 4.3.4 ein Mechanismus zur Priorisierung von Diensten eingeführt. Dieser wird benötigt, um bestimmten Diensten Vorrang vor anderen Diensten einzuräumen. Oben wurde beschrieben, wie die Nutzung von nebenläufigen Bindungen zur Laufzeit ermittelt werden kann. Die Sessions geben an, ob eine Bindung momentan genutzt wird oder nicht. Durch die Priorisierung von Diensten kann es vorkommen, dass eine Dienstinstanz auf eine Ressource zugreift, die bereits durch eine andere Dienstinstanz genutzt wird. Wenn die Dienstinstanz, die Zugriff verlangt, eine höhere Priorität hat als die Dienstinstanz, die die Ressource momentan nutzt, so muss die gebundene Ressource an den priorisierten Dienst freigeben werden. Die Laufzeitumgebung sorgt dafür, dass eine Dienstinstanz benachrichtigt wird, eine bestimmte Ressource freizugeben. Dann kann die Ressource der priorisierten Dienstinstanz zugewiesen werden.

Wie die Überwachung der Bindungen zur Laufzeit im Detail abläuft, ist in Abbildung 4.33 dargestellt. Geprüft werden die Methodenaufrufe, die der Dienstkommunikation dienen. Für einen Methodenaufruf m wird zunächst die zugehörige Bindung b ermittelt, d. h. die Bindung zwischen den Dienstinstanzen, die der Funktionalität entspricht, zu der der Methodenaufruf gehört. Dann wird geprüft, ob es sich bei der Bindung b überhaupt um eine nebenläufige Bindung handelt. Falls dies nicht der Fall ist, kann der Methodenaufruf m ohne weitere Prüfung durchgeführt werden. Falls es sich aber um eine nebenläufige Bindung handelt, so wird als nächstes überprüft, ob bereits eine Session für b aktiv ist. Ist dies der Fall, so wird noch geprüft, ob der Methodenaufruf m eine Annotation zum

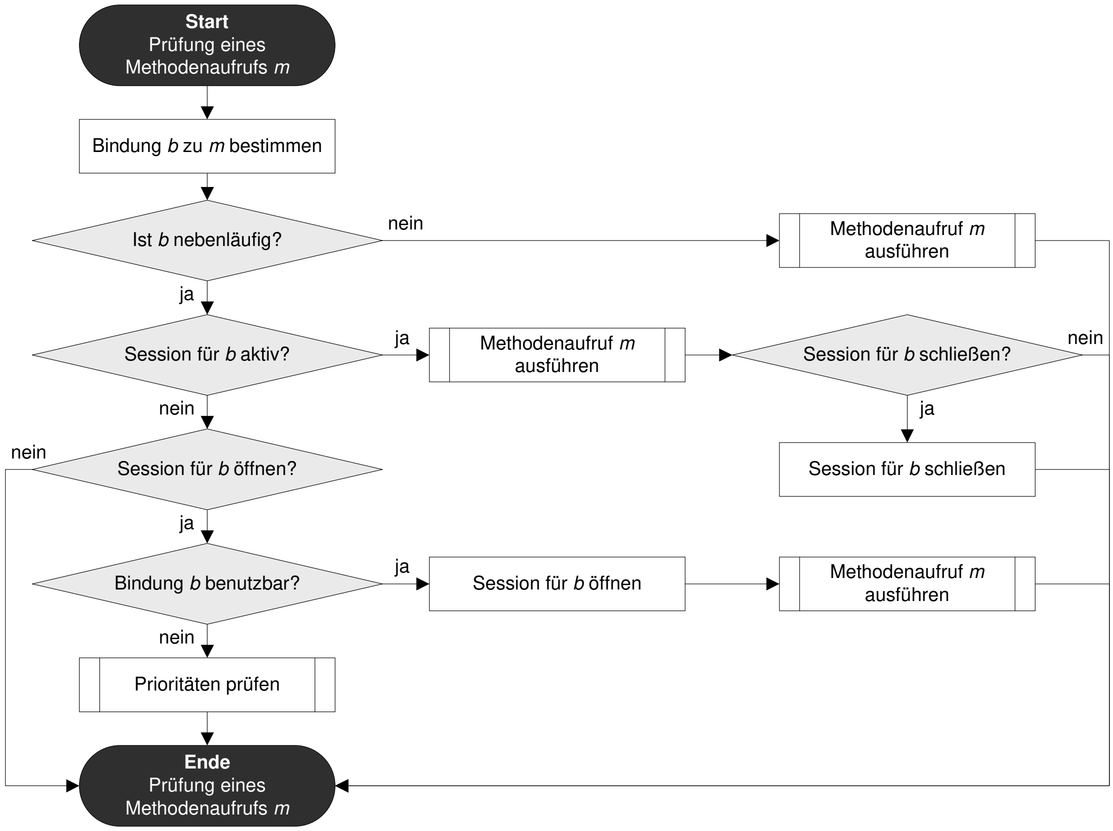


Abbildung 4.33: Algorithmus zur Prüfung eines Methodenaufrufs bei der Dienstkommunikation.

Beenden der Session hat. Dann wird zunächst die Session beendet und dann der Aufruf m ausgeführt. Falls m nicht zum Beenden der Session markiert ist, kann direkt mit dem Aufruf von m fortgefahren werden. Wenn für die Bindung b noch gar keine Session aktiviert wurde, so wird überprüft, ob m eine Annotation zum Öffnen einer neuen Session hat. Falls keine solche Annotation vorhanden ist, so wird der Vorgang ohne einen Aufruf von m beendet, da Methodenaufrufe nur innerhalb einer aktiven Session erlaubt sind. Wenn eine Annotation zum Öffnen einer Session vorhanden ist, dann muss noch geprüft werden, ob die Bindung b aktuell benutzbar ist. Dazu muss untersucht werden, ob bereits andere Sessions aktiv sind, über die die Ressource genutzt wird. Dann können die Kapazitäten der Ressource möglicherweise bereits ausgeschöpft sein. Ist b direkt nutzbar, weil noch ausreichend Kapazitäten der gebundenen Ressource frei sind, so kann eine neue Session für b angelegt werden und der Aufruf von m erfolgen. Anderenfalls muss zuvor eine Prüfung der Priorisierung ausgeführt werden. Wenn der Dienst, dessen Instanz den Methodenaufruf m gestartet hat, eine höhere Priorität hat als der Dienst, dessen Instanz die zugehörige Ressource aktuell benutzt, so wird die Ressource der priorisierten Dienstinstanz zugewiesen und der Methodenaufruf m kann dann erfolgen. Anderenfalls kann kein Aufruf von m durchgeführt werden.

Aspektorientierte Programmierung

Zur Realisierung der Überwachung der Dienstkommunikation wird in dieser Arbeit *AspectJ* verwendet, eine aspektorientierte Erweiterung der Programmiersprache Java [KHH$^+$01, CCHW05]. Mittels *aspektorientierter Programmierung (AOP)* [KLM$^+$97] ist es möglich, zusätzlichen Programmcode an definierten Stellen des Zielprogramms einzufügen.

Die Stellen, an denen Programmcode eingefügt werden soll, werden durch sogenannte *Pointcuts* definiert. Pointcuts beschreiben, an welchen Stellen unter welchen Bedingungen im Zielprogramm der zusätzliche Programmcode ausgeführt werden soll. Pointcuts können sich z. B. auf Aufrufe bestimmter Methoden beziehen.

Der zusätzliche auszuführende Programmcode wird in Form sogenannter *Advices* definiert. Ein Advice ähnelt einer Methode und wird an den durch Pointcuts vorgegebenen Stellen in das Zielprogramm *eingewoben*. Es werden verschiedene Arten von Advices unterschieden. Ein *Before-Advice* definiert Programmcode, der *vor* einer durch einen Pointcut definierten Stelle ausgeführt werden soll. Ein *After-Advice* definiert entsprechend Programmcode für die Ausführung *nach* einem Pointcut. Schließlich gibt es noch den *Around-Device*, der sowohl Programmcode zur Ausführung *vor* dem Pointcut als auch Programmcode zur Ausführung *nach* dem Pointcut enthält.

Die konkreten Stellen im Zielprogramm, die sich aus den Pointcuts ableiten, werden auch *Join Points* genannt. In AspectJ beziehen sich die Join Points auf Methodenaufrufe. Jeder Aufruf einer Methode zur Laufzeit stellt einen individuellen Join Point dar, sodass sich bei mehreren Aufrufen derselben Methode mehrere Join Points ergeben. Ein sogenannter *Weaver* sorgt dafür, dass bei der

```
1 pointcut functionMethods(EhService usedServiceInstance):
2   call(public * ehome.interfaces.*.*(..)) &&
3   target(usedServiceInstance);
4
5 pointcut catchBindingUnusableException():
6   handler(ehome.base.model.generated.exception.BindingUnusableException);
```

Listing 4.1: Pointcuts zum Abfangen der Dienstkommunikation

Kompilation des Quellcodes der Advice-Code an den jeweiligen Join Points in das Zielprogramm eingefügt wird.

Ziel der aspektorientierten Programmierung ist es, Programmcode von *Querschnittsfunktionalitäten*, die nicht einzelnen definierten Komponenten des Systems zugeordnet werden können, zu bündeln. Ein häufig genanntes Beispiel solcher Querschnittsfunktionalitäten ist das Logging, also das Protokollieren bestimmter Aktivitäten des Systems. In dieser Arbeit wird die aspektorientierte Programmierung jedoch für einen anderen Zweck eingesetzt, nämlich das Überwachen und Verwalten der Interaktion von eHome-Diensten.

Realisierung

In Listing 4.1 ist die Deklaration zweier Pointcuts dargestellt, die für die Überwachung der Dienstkommunikation verwendet werden. Der Pointcut `functionMethods` in Zeile 1 wählt alle Aufrufe von Methoden aus, die in den `EhService`-Schnittstellen der eHome-Dienste enthalten sind. Dabei handelt es sich auch um die Methoden, die für die Funktionalitäten der Dienste zuständig sind, daher müssen sie überwacht werden. In Zeile 2 wird zunächst festgelegt, dass es sich bei dem Pointcut um Aufrufe von Methoden einer bestimmten Signatur handeln muss, was durch das `call`-Konstrukt ermöglicht wird. Die gewünschte Signatur ist `public * ehome.interfaces.*.*(..)`, was festlegt, dass die Methode öffentlich sein und in einem Paket, beginnend mit `ehome.interfaces`, enthalten sein muss. Der Rückgabewert kann beliebig sein (spezifiziert durch das Symbol `*`), die Anzahl und der Typ der Parameter ist ebenfalls nicht festgelegt (spezifiziert durch den Ausdruck `(..)`). Mit dem logischen Und-Operator wird in Zeile 3 die Bedingung hinzugefügt, dass das Ziel des Aufrufs das Objekt `usedServiceInstance` sein

```
Object around(EhService usedServiceInstance) throws
        BindingUnusableException: functionMethods(usedServiceInstance) {
    String methodSignature = thisJoinPoint.getSignature().getName();
    String interfaceName = thisJoinPoint.getSignature()
        .getDeclaringType().getSimpleName();
    EhService usingServiceInstance = (EhService) thisJoinPoint.getThis();

    String returnString = interceptor.intercept(usingServiceInstance,
        usedServiceInstance, methodSignature, interfaceName);

    if (returnString.equals(Interceptor.RETURN_SESSION_END)) {
        Object obj = proceed(usedServiceInstance);
        interceptor.checkWaitingServices(interfaceName,
            usedServiceInstance, usingServiceInstance);
        return obj;
    } else if (returnString.equals(Interceptor.RETURN_PROCEED)) {
        return proceed(usedServiceInstance);
    } else {
        int hashCode = Integer.valueOf((returnString.split(";")[1]));
        throw new BindingUnusableException(usingServiceInstance.toString()
            + ", " + usedServiceInstance.toString(), hashCode);
    }
}
```

Listing 4.2: Advice zum Prüfen von Dienstbindungen

muss, das zuvor als Parameter vom Typ EhService für diesen Pointcut deklariert wurde. Dazu wird das target-Konstrukt verwendet.

In Zeile 5 ist ein weiterer Pointcut namens catchBindingUnusableException deklariert. Dieser filtert die Behandlung einer Ausnahme. Hier wird die Behandlung der Ausnahme BindingUnusableException, die im Paket ehome.base.model.generated.-exception deklariert ist, abgefangen. Dieser Pointcut wird für die Verwaltung wartender Dienstinstanzen benötigt.

Ein Advice zur Überwachung der Dienstbindungen, ist in Listing 4.2 dargestellt. Dabei handelt es sich um einen sogenannten Around-Advice, was durch das Schlüsselwort around zum Ausdruck gebracht wird. Ein Around-Advice beinhaltet sowohl Code, der vor dem jeweiligen Join Point ausgeführt wird, als auch Code, der danach ausgeführt wird. Der hier gezeigte Advice hat einen Parameter `usedServiceInstance` vom Typ `EhService` und ist mit dem Pointcut `functionMethods` aus Listing 4.1 verknüpft. Er wird also immer dann ausgeführt, wenn ein Aufruf einer Methode eines eHome-Dienstes erfolgt. Statt des eigentlichen Methodenaufrufs wird dann zunächst der Advice aufgerufen. Im ersten Schritt werden der Name der aufgerufenen Methode (`methodSignature`), der Name der zugehörigen Schnittstelle (`interfaceName`) und die aufrufende Dienstinstanz (`usingServiceInstance`) bestimmt. Dann wird die `intercept`-Methode der `Interceptor`-Klasse aufgerufen, die den aktuellen Status der Sessions im eHome-Modell überprüft. Dabei wird der Algorithmus zur Bindungsprüfung gemäß Abbildung 4.33 angewandt. Je nachdem welches Ergebnis sich daraus ergibt, wird unterschiedlich fortgefahren. In Zeile 11 wird der Fall betrachtet, dass eine Methode aufgerufen wird, die eine bestehende Session beendet. Dann wird mit `proceed` der eigentliche Methodenaufruf durchgeführt, danach werden eventuell wartende Dienste benachrichtigt. In dem Fall in Zeile 16 braucht außer der Durchführung des eigentlichen Methodenaufrufs nichts weiter unternommen werden. Im letzten Fall in Zeile 18 schließlich kann die Bindung nicht genutzt werden. Dann wird die Ausnahme `BindingUnusableException` ausgelöst, die neben einem Hashcode, der zur Identifikation dient, noch Informationen über die aufrufende und die aufgerufene Dienstinstanz enthält.

4.5 Verwandte Arbeiten

In diesem Abschnitt werden einige Projekte vorgestellt und diskutiert, die sich mit ähnlichen Konzepten beschäftigen, wie denen, die in diesem Kapitel zur strukturellen Adaption vorgestellt wurden. Nach einer Beschreibung der Kernideen wird jeweils ein Vergleich zur vorliegenden Arbeit hergestellt und es werden einige Unterschiede sowie Vor- und Nachteile der Ansätze diskutiert.

Gravity-Projekt

Das Gravity-Projekt verfolgt das Ziel, eine Infrastruktur zu schaffen, die es erlaubt, Anwendungen aus wiederverwendbaren Bausteinen zu entwickeln. Dies soll durch ein hierarchisches dienstorientiertes Komponentenmodell ermöglicht werden, das insbesondere die dynamische Verfügbarkeit von Komponenten und Kontextinformationen berücksichtigt [CH03]. Die grundlegende Annahme ist, dass die verwendeten Komponenten zur Laufzeit jederzeit erscheinen und wieder verschwinden können. Gravity verbindet Ansätze der komponentenbasierten Softwareentwicklung mit dem Konzept der Serviceorientierung (vgl. auch Abschnitt 3.1 und 3.2). Beide Ansätze werden als komplementär betrachtet [HC03, CH04b]. Im Gravity-Komponentenmodell stellen Softwarekomponenten Dienste zur Verfügung, die über ein zentrales Dienstverzeichnis auffindbar sind. Dabei wird unter einem Dienst eine von einer Komponente implementierte Schnittstelle verstanden. Gravity unterstützt die Entwicklung kontextbezogener Anwendungen durch die dynamische Verwaltung von Bindungen auf Basis von Kontextinformationen. Bei Änderungen des Kontextes müssen ggfs. Bindungen geändert werden.

Dienstbindungen werden in Gravity durch den sogenannten *Service Binder* verwaltet. Der Service Binder ermöglicht eine automatische Dienstkomposition auf Basis zuvor spezifizierter Abhängigkeiten. Anwendungsentwickler werden auf diese Weise von der Implementierung einer entsprechenden Funktionalität entlastet. Dazu übernimmt der Service Binder die Überwachung der Dienstabhängigkeiten zur Laufzeit, das Erstellen und Löschen von Dienstinstanzen und Dienstbindungen, sowie das Registrieren und Deregistrieren von Diensten der von Komponenten angebotenen Dienste beim Dienstverzeichnis. Zur Realisierung des Service Binder wird die OSGi Service Plattform als Grundlage verwendet (vgl. auch Abschnitt 3.2.4). Die Komponenten für den Service Binder werden daher als OSGi Bundles implementiert. Zusätzlich wird vom Service Binder eine Komponentenbeschreibung in einem vorgegebenen XML-Format benötigt, damit dieser die Komponente instanziieren und verwalten kann.

Das Listing 4.3 zeigt ein Beispiel einer Komponentenbeschreibung für den Service Binder. Eine solche Spezifikation der Dienstabhängigkeiten wird benötigt, damit der Service Binder die Bindungsverwaltung übernehmen kann. Darin wird spezifiziert, welche Dienste die Komponente anbietet und welche sie benötigt. Entsprechend gibt es `provides`- bzw. `requires`-Tags, letztere umfassen weitere Parameter, wie `cardinality`, `policy`, `filter` und `bind-method` sowie `unbind-method`, die genauer festlegen, welche Abhängigkeiten bestehen [CH04a]. Die Spezifikation im Beispiel beschreibt eine Komponente `ehome.services.personal.light.EhLightFollowsPerson`, die die Aufgabe hat, die Beleuchtung in der Nähe des Benutzers zu

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <bundle>
3     <component class="ehome.services.personal.light.EhLightFollowsPerson">
4         <provides service="ehome.services.EhLightFollowsPerson"/>
5         <requires
6             service="ehome.services.Lamp"
7             cardinality ="1..n"
8             policy="dynamic"
9             bind-method="addLampReference"
10            unbind-method="removeLampReference"
11        />
12        <requires
13            service="ehome.services.PersonDetector"
14            cardinality ="1..1"
15            policy="static"
16        />
17    </component>
18 </bundle>
```

Listing 4.3: Beispiel einer Komponentenbeschreibung für den Service Binder

steuern. Dazu implementiert sie die Schnittstelle `ehome.services.EhLightFollowsPerson` (Zeile 4), dies ist also der angebotene Dienst der Komponente. Um den Dienst anbieten zu können, benötigt die Komponente andere Dienste. Zunächst benötigt sie den Dienst `ehome.services.Lamp` (Zeile 5), der die Steuerung einer Lampe realisiert. Das `cardinality`-Attribut (Zeile 7) legt fest, dass mindestens eine Lampe benötigt wird, es können aber auch beliebig viele gebunden werden. Die Policy `dynamic` (Zeile 8) legt fest, dass Bindungen auch nachträglich noch hinzugefügt, entfernt oder geändert werden können. Die letzten beiden Attribute für den benötigten Dienst `ehome.services.Lamp` sind `bind-method` (Zeile 9) und `unbind-method` (Zeile 10). Diese legen fest, welche Methoden der Komponente aufgerufen werden müssen, um Bindungen des Typs `ehome.services.Lamp` herzustellen oder wieder zu entfernen. Die Komponente muss die Methoden entsprechend implementieren und Referenzen auf gebundene Dienste speichern bzw. wieder löschen. Im Beispiel wird darüber hinaus ein Dienst zur Personenerkennung benötigt (Zeile 12), der die nötigen Kontextinformationen über den Aufenthaltsort der Person liefert. Hier ist die Policy `static` angegeben, daher wird diese Bindung einmalig erstellt und kann danach nicht mehr verändert werden.

In [CH04a] beschreiben CERVANTES und HALL Erfahrungen, die mit dem Service Binder in der praktischen Anwendung in zwei Projekten gemacht wurden. Dabei stellte sich heraus, dass besonders die nichtdeterministische Auswahl geeigneter Dienste für die Bindungen ein Problem darstellt. Sind mehrere Kandidaten für einen zu bindenden Dienst verfügbar, so kann nicht vorhergesagt oder bestimmt werden, welcher der Kandidaten für die Bindung ausgewählt wird. In [CH04b] werden des Weiteren einige zukünftige Erweiterungen von Gravity diskutiert. Darunter findet sich auch die Überlegung, Komponenten semantisch zu beschreiben, anstatt die Komposition auf Basis der Schnittstellennamen durchzuführen. Ein solcher Ansatz wird in dieser Arbeit in Kapitel 5 vorgestellt.

Der Service Binder zielt auf eine dynamische Verwaltung von Diensten ab, die das Hinzufügen, Entfernen und Modifizieren zur Laufzeit ermöglicht. Für die Realisierung wird in erster Linie auf die Funktionalitäten der OSGi Service Plattform zurückgegriffen, wie etwa die OSGi Service Registry. In dieser Arbeit wurde dazu ein anderer Ansatz gewählt. Die Informationen über die Dienste und ihre Abhängigkeiten sind im zentralen eHome-Modell zusammen mit den relevanten Kontextinformationen erfasst. Der Vorteil liegt darin, dass eine globale Sicht auf die Daten vorliegt und sowohl die Verfügbarkeit der Dienste als auch die jeweilige Kontextsituation für die Komposition berücksichtigt werden kann.

Die Entwicklung personenbezogener Dienste wird vom Service Binder nicht direkt unterstützt, da nur statische Attribute verwaltet werden können. Daher muss der Dienstentwickler die entsprechende Funktionalität selbst realisieren, im Beispiel aus Listing 4.3 wird daher ein Dienst zur Personenerkennung benötigt, auf dessen Basis der Dienst die Personalisierung umsetzen kann. Im Ansatz dieser Arbeit werden personenbezogene Dienste durch die Laufzeitumgebung unterstützt. Der personenbezogene Kontext wird ebenfalls im eHome-Modell erfasst und eine entsprechende Rekomposition von Diensten wird ermöglicht.

Der Service Binder hat den Vorteil, dass er nicht auf eine spezifische Anwendungsdomäne ausgerichtet ist und daher in verschiedenen Gebieten eingesetzt werden kann. Allerdings bietet er aus diesem Grund im Gegensatz zur vorliegenden Arbeit auch keine spezifische Werkzeugunterstützung zur Visualisierung der Komposition und zur Interaktion mit dem Benutzer an. Ein manueller Eingriff in die Dienstkomposition ist daher nicht möglich.

ANSO-Projekt

Im Forschungsprojekt ANSO, das zum Teil von der französischen Regierung finanziert wird, wurde von Bottaro und Gérodolle eine Erweiterung des Service Binder entwickelt, der *Extended Service Binder* [BG06]. Unter Beteiligung mehrerer Industriepartner wird im ANSO-Projekt an einer verteilten Middleware zur Unterstützung von Diensten in Klein- und Heimbüros gearbeitet. Der Extended Service Binder unterstützt eine transparente Nutzung verteilter Dienste, im Service Binder wurde hingegen keine Verteilung unterstützt, da auch das zugrunde liegende OSGi keine Verteilung unterstützt. Darüber hinaus wurden mit dem Extended Service Binder dynamische Diensteigenschaften eingeführt, und es ist möglich, aufgrund dieser Eigenschaften eine Sortierung der potentiellen Dienstanbieter vorzunehmen, um damit die Auswahl der zu bindenden Dienste zu beeinflussen [BH07]. Dies war eines der Defizite, die bei der Anwendung des Service Binder erkannt wurden, dort konnten nur statische Eigenschaften in der Spezifikation Verwendung finden. Im Extended Service Binder kann hingegen für jedes Attribut unter dem Eintrag `property-method` in der Spezifikation eine Methode angegeben werden, die die dynamische Berechnung dieses Attributs ermöglicht. Für die benötigten Dienste kann dann eine Sortiermethode unter dem Eintrag `sort-method` angegeben werden. Diese Methode legt dann dynamisch eine Dienstreihenfolge unter allen passenden Diensten fest. Daraus ergibt sich, welche Dienste anderen Diensten vorzuziehen sind. Das ist eine Möglichkeit, um beispielsweise Kontextinformationen bei der Erstellung von Dienstbindungen mit einzubeziehen. Im Beispiel aus Listing 4.3 könnte so festgelegt werden, dass nur Lampen in der Nähe des Benutzers angesteuert werden sollen.

Der Extended Service Binder adressiert verschiedene Defizite des Service Binder, die teils bereits oben diskutiert wurden. Da der Extended Service Binder auf dem Service Binder basiert, wird auch hier die OSGi Service Plattform zugrunde gelegt. Durch die Einführung dynamisch berechenbarer Attribute kann nun aber eine einfachere Realisierung kontextbezogener Dienste umgesetzt werden. Allerdings können diese Attribute lediglich zur Berechnung einer Rangfolge verwendet werden. Die in dieser Arbeit eingeführten Bindungsbeschränkungen bieten weiterführende Möglichkeiten, da sie auf der Basis des eHome-Modells realisiert werden und sämtliche darin enthaltenen Informationen berücksichtigen können. So kann z. B. in den Graphmustern einer Bindungsbeschränkung

Bezug auf bestimmte Räume der Umgebung genommen werden. Im Fall eines personenbezogenen Lichtdienstes würde es nicht ausreichen, die Lampe auf dem obersten Platz der Rangliste zu binden, wenn diese sich dennoch nicht in unmittelbarer Nähe des Benutzers befindet. Die Bindungsbeschränkungen in der vorliegenden Arbeit sind bisher allerdings nicht frei definierbar. Eine Werkzeugunterstützung wie in dieser Arbeit wird wie schon im Fall des Service Binder auch vom Extended Service Binder nicht angeboten.

OSGi Declarative Services

Die OSGi Service Plattform [WHKL08] (vgl. auch Abschnitt 3.2.4) unterscheidet Abhängigkeiten zwischen Bundles von Abhängigkeiten zwischen Diensten. Die Abhängigkeiten zwischen Bundles ergeben sich aus den entsprechenden Bundle-Manifesten und werden von OSGi beim Installieren der Bundles aufgelöst. Bundleabhängigkeiten sind weder dynamisch noch kontextbezogen, sie bestimmen lediglich die Voraussetzungen, die zur Verwendung eines Bundles erfüllt sein müssen. Erst mit der Dienstschicht von OSGi erhalten dynamische Konzepte Einzug. Dienste können in OSGi zur Laufzeit bei der Service Registry an- und auch wieder abgemeldet werden. Die Abhängigkeiten zwischen Diensten werden jedoch nicht von OSGi verwaltet, dies muss stattdessen der Dienstentwickler selbst realisieren. Jeder Dienst muss die Suche nach geeigneten Diensten bei der OSGi Service Registry und die Bindung dieser Dienste selbst implementieren. Insbesondere muss dabei auch die dynamische Verfügbarkeit beachtet werden. Dienste sind nicht jederzeit nutzbar, was bei der Implementierung berücksichtigt werden muss.

Um eine einfachere Verwaltung von Dienstabhängigkeiten in OSGi zu ermöglichen, wurde das Konzept der sogenannten *Declarative Services* in die OSGi Spezifikation [OSG07b] aufgenommen. Dieses Konzept basiert auf dem oben beschriebenen *Service Binder* aus dem Gravity-Projekt. Declarative Services erweitern OSGi, sodass Dienstabhängigkeiten *deklarativ* beschrieben werden können. Die Abhängigkeiten werden dann von der OSGi Laufzeitumgebung aufgelöst, der Dienstentwickler braucht daher keine entsprechenden Mechanismen im eigentlichen Anwendungscode implementieren. Auf diese Weise wird die Entwicklung von Diensten vereinfacht und die Komplexität des Codes verringert.

Um Declarative Services zu nutzen, muss neben dem Bundle-Manifest eine XML-basierte Beschreibung der Dienste angegeben werden. Diese ist bis auf wenige syntaktische Unterschiede identisch mit der Komponentenbeschreibung im Service Binder. Auch in dieser Arbeit erfolgt, wie in Abschnitt 4.3 beschrieben, eine deklarative Dienstbeschreibung, um Dienstabhängigkeiten, Einschränkungen bei der Komposition und das gewünschte Bindungsverhalten festzulegen. Kontextbezogene Abhängigkeiten wie in den Bindungsbeschränkungen können mit Declarative Services jedoch nicht umgesetzt werden. Für jeden spezifizierten Declarative Service wird in OSGi ein sogenannter Instanzmanager erzeugt, der die deklarativ beschriebenen Abhängigkeiten verwaltet und dafür sorgt, dass diese erfüllt werden. Wie auch im Ansatz der vorliegenden Arbeit wird dazu zwischen gültigen und ungültigen Instanzen unterschieden. Allerdings erfolgt in der vorliegenden Arbeit eine globale Verwaltung der Dienstinstanzen und ihrer Abhängigkeiten unter Verwendung des eHome-Modells. Die Möglichkeit übergreifende Kontextinformationen zu berücksichtigen, wie es in den Bindungsbeschränkungen der Fall ist, besteht mit Declarative Services nicht. Wie schon beim Service Binder ist auch bei OSGi Declarative Services keine manuelle Konfigurierung der Dienste möglich, es existiert auch kein Werkzeug zur Visualisierung und manuellen Interaktion. Dies ist ein wesentlicher Aspekt dieser Arbeit, da Nutzer die Kontrolle über ihre Wohnumgebung behalten wollen. Wenngleich eine möglichst weitreichende Automatisierung erwünscht ist, muss dennoch die manuelle Einflussnahme erlaubt werden.

iPOJO

Neben den Declarative Services, die in die OSGi Spezifikation aufgenommen wurden, gibt es eine weitere Laufzeitumgebung, die aus dem Gravity-Projekt und dem Service Binder hervorgegangen ist. Bei dieser Laufzeitumgebung handelt es sich um *iPOJO*[1] [EH07, EHL07], das als Teilprojekt von Apache Felix, einer Implementierung der OSGi Service Plattform, vorangetrieben wird. iPOJO ist kein Bestandteil von OSGi, bietet aber weiterführende Mechanismen, die über die Möglichkeiten der Declarative Services hinausgehen. Ziel ist es, den Anwendungscode von Komponenten möglichst von nicht-funktionalem Code freizuhalten. Bei der Anwendungsentwicklung sollen reguläre Java-Objekte, sogenannte

[1] `http://ipojo.org/`

POJOs (engl. *Plain Old Java Objects*), verwendet werden können. Die nichtfunktionalen Aspekte, wie die Verwaltung von Abhängigkeiten, werden deklarativ beschrieben. Bindungen zwischen Diensten werden in iPOJO beispielsweise über normale Variablen, die Referenzen auf gebundene Objekte speichern, abgebildet. Die Verwaltung der Bindungen wird dann von iPOJO übernommen. In der Komponentenbeschreibung wird festgelegt, welche Variablen welche Bindungen speichern. Diese Variablen werden dann von iPOJO unter Verwendung von *Dependency Injection* [Fow04] gesetzt, wodurch die Bindungen festgelegt werden. Den Typ der Variablen ermittelt iPOJO dabei selbstständig mittels Java-Reflection. In der Komponentenbeschreibung können neben Variablen auch Methoden für die „Injektion" von Bindungen angegeben werden.

Listing 4.4 zeigt eine vereinfachte Beispielimplementierung eines Lichtdienstes, der über iPOJO zur Verfügung gestellt werden soll. Die Variablen `detector` und `lamps` speichern Referenzen auf andere Komponenten, die Funktionalitäten zur Lokalisierung des Benutzers bzw. zur Beleuchtung anbieten. Die Verwaltung dieser Referenzen wird nicht in der Komponente selbst vorgenommen, sondern von iPOJO. In der Implementierung der Komponente können die Referenzen direkt benutzt werden, indem Methoden auf ihnen aufgerufen werden. Dies ist in der `start`-Methode zu sehen. Damit iPOJO die Komponente verwalten kann, muss eine entsprechende Spezifikation erstellt werden. In Listing 4.5 ist eine iPOJO-Spezifikation für das erwähnte Beispiel gegeben. Das `component`-Tag bezeichnet die Klasse, auf die sich die Spezifikation bezieht. Die `dependency`-Tags legen die Abhängigkeiten der Komponente fest. Dabei braucht nur der Variablenname angegeben werden, da iPOJO den Typ mittels Reflection feststellt. Das `provides`-Tag legt lediglich fest, dass ein Dienst durch die Komponente angeboten wird. Die angebotenen Schnittstellen werden von iPOJO berechnet und müssen daher nicht explizit aufgeführt werden. Das `instance`-Tag sorgt schließlich dafür, dass beim Starten der Komponente eine Dienstinstanz erzeugt wird.

iPOJO erlaubt es, sogenannte Handler zu entwickeln, die eine spezifische Behandlung der nicht-funktionalen Aspekte einer Komponente erlauben. Das POJO, das lediglich die Anwendungslogik implementiert, wird durch einen Container verkapselt, der durch Handler erweitert werden kann. Dies ist in Abbildung 4.34 schematisch dargestellt. Handler können wiederum mit anderen Handlern interagieren und so Informationen sammeln, die für die Verwaltung des zugehörigen POJOs relevant sind. In [BBEL07] wird die Anwendung von

```java
package ehome.services.personal.light;

public class EhLightFollowsPerson
    implements ehome.services.EhLightFollowsPerson {

    private PersonDetector detector;
    private Lamp[] lamps;

    private User user;

    public void start() {
        for (int i = 0 ; i < lamps.length; i++) {
            if (getLocation(lamps[i]) == detector.getLocation(user)) {
                lamps[i].switchOn();
            }
        }
    }

    // Other methods of the EhLightFollowsPerson service ...

}
```

Listing 4.4: Vereinfachte Beispielimplementierung eines Lichtdienstes

```xml
<ipojo>
    <component
    className="ehome.services.personal.light.EhLightFollowsPerson">
        <dependency field="detector"/>
        <dependency field="lamps"/>
        <provides/>
    </component>
    <instance
    component="ehome.services.personal.light.EhLightFollowsPerson"/>
</ipojo>
```

Listing 4.5: Beispiel einer Komponentenbeschreibung in iPOJO für den Lichtdienst

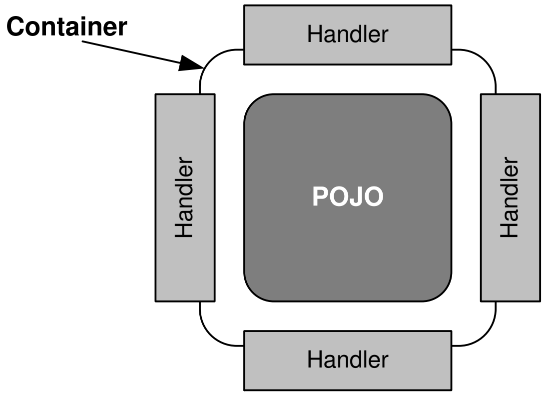


Abbildung 4.34: Nutzung von Handlern in iPOJO. (in Anlehnung an [EHL07])

iPOJO in einem Home Control Gateway vorgestellt. Dazu wurde ein spezieller Kontext-Handler implementiert, der über die Komponentenbeschreibung parametrisiert werden kann. Zusätzlich wurde ein Kontextdienst eingeführt, der diesen Handler mit Kontextinformationen versorgt. Der Kontext-Handler kann dann die Dienstbindungen auf Basis der Kontextinformationen verwalten. Handler können aber auch für andere Zwecke genutzt werden, beispielsweise für die Persistierung von Komponenten.

iPOJO ergänzt OSGi um eine flexible und dynamische Verwaltung von Dienstbindungen. Der Ansatz, nicht-funktionale Aspekte von der eigentlichen Anwendungslogik getrennt zu betrachten, wird auch in der vorliegenden Arbeit verfolgt. Auch hier wird die *Dependency Injection* im Deployment angewandt, um die Bindungen der Dienste anzulegen oder zu löschen. Dazu müssen die Dienste ein spezifisches Interface implementieren, während in iPOJO beliebige Variablen oder Methoden für das Binden in der Spezifikation angegeben werden können. Die Komposition selbst kann in iPOJO zwar durch Angabe von Filtern beeinflusst werden, es steht aber kein Mechanismus für die kontextbezogene Komposition zur Verfügung. In dieser Arbeit stellt das eHome-Modell die Grundlage der Komposition dar, sodass durch Verwendung von Bindungsbeschränkungen eine kontextbezogene Komposition möglich ist. In iPOJO muss dazu, wie in [BBEL07] vorgestellt, zunächst ein eigener Kontext-Handler entwickelt werden, sowie ein Kontextdienst zur Verteilung der Informationen. Eine direkte Kontextunterstützung durch iPOJO ist jedoch nicht vorgesehen. Im Unterschied zum Service Binder werden in iPOJO keine angebotenen und benötigten Schnittstellen spezifiziert, sondern es werden Variablen angegeben, die für die Realisierung von Bindungen verwendet werden. Die Schnittstellen ergeben sich implizit aus den Typen der spezifizierten Variablen. In dieser Arbeit wird neben der Komposition auf Basis von Schnittstellen auch die semantische Dienstkomposition unterstützt, was im nächsten Kapitel 5 vorgestellt wird. Semantische Informationen werden in iPOJO nicht in Betracht gezogen. Ebenso wie in den übrigen vorgestellten Ansätzen wird keine manuelle Modifikation der von iPOJO vorgenommenen Bindungen ermöglicht. Die Vorgänge laufen sämtlich automatisch und im Hintergrund ab. Dementsprechend gibt es auch keine Werkzeuge zur Interaktion mit dem Benutzer, wie sie in dieser Arbeit entwickelt wurden.

AWARENESS-Projekt

Das AWARENESS-Projekt ist ein Projekt des Forschungsprogramms *Freeband Communication* der niederländischen Regierung. Ziel von Freeband Communication ist es, Forschungsprojekte aus dem Bereich der Ambient Intelligence und der intelligenten Kommunikation zu fördern. Im Rahmen des AWARENESS-Projekts wurde von Broens et al. die Infrastruktur *CACI* entwickelt, die der dynamischen Verwaltung kontextbasierter Bindungen dient [BQvS07]. *CACI* steht für ***C**ontext-**A**ware **C**omponent **I**nfrastructure*. Es handelt sich dabei um eine kontextbezogene Middleware, die eine automatische Verwaltung der Bindungen personalisierter, mobiler, kontextbezogener Anwendungen ermöglicht. Unter kontextbezogenen Anwendungen werden dabei Anwendungen verstanden, die die momentane Situation des Benutzers berücksichtigen und ihr Verhalten dieser Situation anpassen. Die Middleware sorgt dafür, dass Anwendungsentwickler von der komplexen Aufgabe, Kontextinformationen zu ermitteln und zu verwalten, entlastet werden und sich stattdessen auf die eigentliche Anwendungslogik konzentrieren können. Dazu wird die Verwaltung der Kontextinformationen von der Middleware übernommen, die diese Informationen dann den Anwendungen zur Verfügung stellt.

In CACI werden kontextproduzierende und kontextkonsumierende Komponenten unterschieden. Kontextproduzierende Komponenten sind Sensoren, die bestimmte Informationen über die momentane Situation eines Objekts oder einer Person liefern. Das können z. B. GPS-Empfänger oder RFID-Tags zur Lokalisierung von Entitäten sein. Kontextkonsumierende Komponenten sind kontextbe-

```
1 eHome_env_context_binding ::
2     requires Location
3     from Person.John
4     expressed_in LatLong
5     with accuracy > 75%
6     scope global
7     policy dynamic
8     optional false
```

Listing 4.6: Beispiel einer CCDL-Spezifikation

zogene Anwendungen, die Kontextinformationen benötigen, um ihr Verhalten an die jeweilige Umgebung anzupassen. In manchen Fällen ist die Zuordnung nicht eindeutig, manche Applikationen können sowohl als Kontextkonsumenten als auch als Kontextproduzenten auftreten. Kontextproduzenten und Kontextkonsumenten werden in CACI durch sogenannte *Kontextbindungen* miteinander verknüpft. Diese Kontextbindungen sind nicht fest verdrahtet, sondern werden von der CACI-Middleware dynamisch erstellt und verwaltet. Um passende Kontextproduzenten für die Kontextkonsumenten zu finden und die Kontextbindungen anzulegen, wird in [BvHvS06] die ***C****ACI* ***C****omponent* ***D****escription* ***L****anguage (CCDL)* eingeführt, eine Beschreibungssprache für CACI-Komponenten. Darin können Anforderungen an die benötigten Kontextinformationen abstrakt spezifiziert werden. Diese werden dann von der Middleware bei der Verwaltung der Kontextbindungen berücksichtigt. Für den Anwendungsentwickler wird die Nutzung von Kontextinformationen somit transparent, da er nur Anforderungen spezifizieren muss, nicht jedoch die Suche und Bindung der kontextproduzierenden Komponenten selbst.

In Listing 4.6 ist ein Beispiel einer CCDL-Spezifikation aufgeführt. Darin können verschiedene Parameter angegeben werden, die die spätere Erstellung der Kontextbindungen beeinflussen. Im Beispiel wird der momentane Aufenthaltsort der Person John gesucht. Ein passender Kontextproduzent muss gemäß der Spezifikation den gesuchten Ort in den Koordinaten Länge und Breite (Zeile 4) mit einer vorgegebenen Genauigkeit (Zeile 5) anbieten. Weitere Parameter der Spezifikation bestimmen die Verwaltung der Bindungen zur Laufzeit. Im Beispiel ist für die Bindung eine Policy `dynamic` spezifiziert (Zeile 7), d. h. die Bindung soll austauschbar sein. Die Bindung ist aber nicht optional (Zeile 8), d. h. sie wird in jedem Fall benötigt, damit die Anwendung ausgeführt werden kann. Für die Policy kann neben `dynamic`, was für austauschbare Bindungen steht, auch `semi-static` oder `static` ausgewählt werden. Bindungen mit der Policy `static` werden nach dem ersten Anlegen nicht mehr verändert. Die Policy `semi-static` bewirkt, dass die Bindung geändert wird, falls der bisherige Kontextproduzent nicht mehr verfügbar ist. Im Fall von `dynamic`, wie im obigen Beispiel, wird die Bindung auch dann geändert, wenn ein Kontextproduzent verfügbar wird, der die Anforderungen besser, d. h. mit einer höheren Qualität, erfüllen kann. Im Beispiel aus Listing 4.6 könnte etwa ein Kontextproduzent verfügbar werden, der eine höhere Genauigkeit der Personenlokalisierung bietet.

CACI und CCDL sind für die Verwaltung kontextproduzierender und kontextkonsumierender Komponenten ausgelegt. Die Middleware verwaltet Kontextbindungen zwischen diesen Komponenten. Der in dieser Arbeit vorgestellte Ansatz betrachtet eHome-Dienste und deren angebotene und benötigte Funktionalitäten, wobei nicht zwischen Kontextproduzenten und Kontextkonsumenten unterschieden wird. Anders als in CACI werden in dieser Arbeit alle für die Bindungen zwischen Diensten relevanten Funktionalitäten betrachtet, nicht nur solche, die Kontextinformationen übermitteln. Somit wird in dieser Arbeit die Komposition von Diensten und die Verwaltung der entsprechenden Dienstbindungen im Allgemeinen betrachtet.

Kontextinformationen werden in der vorliegenden Arbeit differenziert betrachtet. Einerseits werden solche Kontextinformationen, die Einfluss auf die Dienstkomposition haben, nicht an die Dienste weitergereicht, sondern von der Laufzeitumgebung direkt verwendet, um die Dienstbindungen auf Basis des Kontextmodells zu verwalten. Das gewünschte kontextbezogene Verhalten eines Dienstes wird dabei durch die in diesem Kapitel eingeführten Bindungsbeschränkungen und Bindungsstrategien deklarativ spezifiziert. Andererseits werden Kontextinformationen, die keinen Einfluss auf die Dienstkomposition haben, von den Diensten selbst verwaltet. Diese Informationen werden über entsprechende Funktionalitäten ausgetauscht, z. B. das Messen der Umgebungstemperatur. Der Temperaturwert kann direkt von einem Treiberdienst über eine entsprechende Funktionalität an einen Top-Level-Dienst weitergereicht werden. Der Ansatz, der CACI zugrunde liegt, sieht nur den direkten Austausch von Kontextinformationen zwischen Diensten vor.

Dennoch gibt es auch verschiedene Übereinstimmungen in Bezug auf die Verwaltung der Bindungen. CACI bietet ebenso wie der hier vorgestellte Ansatz die Möglichkeit, Bindungen dynamisch zu verwalten und Änderungen zur Laufzeit vorzunehmen. Die CCDL-Spezifikation beinhaltet ähnliche Parameter wie die Bindungsstrategien in dieser Arbeit. Eine manuelle Beeinflussung der Kontextbindungen ist dabei jedoch nicht vorgesehen, es kann lediglich zwischen verschiedenen automatischen Varianten gewählt werden. Daraus resultierend ist auch keine Werkzeugunterstützung für CACI vorgesehen, es ist also keine Interaktion mit dem Benutzer möglich.

Qualitätsattribute, wie z. B. die Genauigkeitsangabe in der Spezifikation in Listing 4.6, werden in dieser Arbeit nicht betrachtet. Dies wäre eine Ergänzung, die in zukünftigen Arbeiten in den bestehenden Ansatz integriert werden könnte. Allerdings ist auch in CCDL die Spezifikation von Qualitätsattributen noch nicht geklärt, sondern ebenfalls ein offener Punkt für zukünftige Arbeiten [BvHvS06]. So ist z. B. bislang nicht klar, was die Genauigkeit von mindestens 75% im Beispiel bedeuten soll. Ebenso ist unklar, wie der Bezug zur Person John hergestellt werden kann. Die konkrete Angabe der Person in der CCDL-Spezifikation wird nicht möglich sein, da diese Spezifikation vom Dienstentwickler vorgenommen werden soll. Dieser kann kein Wissen über die konkrete Anwendungsumgebung und die dort anwesenden Personen haben. In CCDL werden außerdem nur die Anforderungen von Kontextkonsumenten spezifiziert. Eine entsprechende Spezifikation der von Kontextproduzenten angebotenen Informationen wird bisher nicht unterstützt [BQvS07]. Dies wird jedoch für die Umsetzung des Ansatzes unerlässlich sein. Nur wenn auch Informationen über die Anbieterseite gegeben sind, kann eine sinnvolle Verwaltung der Kontextbindungen realisiert werden.

4.6 Zusammenfassung

In diesem Kapitel wurde erläutert, warum eHomes dynamische Umgebungen sind und welche unterschiedliche Faktoren die Dynamik beeinflussen. Neben den sich ändernden Anforderungen sind die Benutzer und die in der Umgebung verfügbaren Ressourcen die wichtigsten Einflussfaktoren. Es wurde gezeigt, dass es vor dem Hintergrund dieser Dynamik wichtig ist, die Entwicklung von eHome-Diensten durch eine geeignete Infrastruktur zu unterstützen. Aufbauend auf den Vorarbeiten wurde dazu ein Ansatz zur strukturellen Adaption entwickelt, der eine dynamische Anpassung der Dienstkomposition ermöglicht. Dazu wurde der bestehende SCD-Prozess zu einem kontinuierlichen SCD-Prozess weiterentwickelt. Bei dem kontinuierlichen SCD-Prozess handelt es sich nun nicht mehr um einen Installationsprozess, sondern um einen Laufzeitprozess, der zur Ausführungszeit des eHome-Systems durchgeführt wird.

Um eHome-Dienste im kontinuierlichen SCD-Prozess auszuführen, wurden verschiedene Veränderungen bei der Dienstentwicklung eingeführt. Die Laufzeitumgebung ermöglicht nun die Verwaltung von Kontextdaten. Dadurch können neben raumbezogenen Diensten auch personenbezogene Dienste realisiert werden. Damit die im eHome-Modell erfassten Kontextinformationen bei der Dienstkomposition berücksichtigt werden können, wurde das Konzept der Bindungsbeschränkungen eingeführt. Diese erlauben es, die Komposition von zuvor festgelegten Bedingungen abhängig zu machen. Um das Bindungsverhalten zur Laufzeit festlegen zu können, wurden Bindungsstrategien eingeführt. Davon abhängig können Bindungen automatisch oder manuell verwaltet werden. Zur besseren Ausnutzung der verfügbaren Ressourcen im eHome wurden nebenläufige Bindungen und die Möglichkeit der Priorisierung von Diensten eingeführt.

Die Erweiterungen der Dienstspezifikation werden im kontinuierlichen SCD-Prozess durch entsprechende Mechanismen ausgewertet. Bei der Dienstentwicklung kann auf Funktionen der Laufzeitumgebung zurückgegriffen werden, wodurch die Entwicklung vereinfacht wird. Querschnittsfunktionalitäten, die viele Dienste betreffen, wurden auf die Ebene der Laufzeitumgebung verlagert. Diese Funktionalitäten müssen daher nicht immer wieder neu implementiert werden. Damit werden Zeitaufwand und Kosten der Dienstentwicklung reduziert, was dem Ziel der kostengünstigen Realisierung von eHomes entspricht.

Abschließend wurden einige verwandte Arbeiten diskutiert. Die wesentlichen Unterschiede bestehen darin, dass in den vorgestellten Projekten Kontextinformationen meist keine explizite Berücksichtigung finden. Dies ist für die Komposition von Diensten in eHomes ein wichtiger Aspekt. Außerdem wird in dieser Arbeit die Laufzeitumgebung durch Werkzeuge ergänzt, die eine Visualisierung der Dienstkomposition und eine Interaktion durch den Benutzer ermöglicht. Eine solche Unterstützung ist in den vorgestellten Projekten nicht vorgesehen.

Kapitel 5

Semantische Adaption

In diesem Kapitel wird die *semantische Adaption* eingeführt. Diese Art der Adaption ist komplementär zur strukturellen Adaption, die in Kapitel 4 beschrieben wurde. Zunächst wird die Zielsetzung der semantischen Adaption erläutert und es werden einige grundlegende Überlegungen zu semantikbasierten Ansätzen diskutiert. Darauf folgt die Beschreibung der im Rahmen dieser Arbeit entwickelten semantischen Konzepte, die sich auf die Anwendungsdomäne eHome-Systeme beziehen und die Besonderheiten dieser Domäne berücksichtigen. Abschließend werden auch hier verschiedene verwandte Arbeiten analysiert und bewertet.

5.1 Heterogenität in eHome-Systemen

Eine wesentliche Herausforderung bei der Entwicklung von eHome-Systemen ist es, die Komponierbarkeit und Interoperabilität der eHome-Dienste sicherzustellen [RP08]. In eHome-Systemen müssen die verschiedensten Dienste zusammenarbeiten, um die vom Nutzer gewünschten Funktionalitäten zu realisieren. Um Dienste miteinander komponieren zu können, sind passende Schnittstellen erforderlich. Damit die Interoperabilität gewährleistet werden kann, ist ein gemeinsames Verständnis der Kommunikationsprotokolle unter den beteiligten Entwicklern notwendig. Die Vielzahl der verschiedenen Dienste, die in einem eHome zum Einsatz kommen können, wird jedoch häufig nicht von einem einzelnen Hersteller oder gar einem einzelnen Entwickler realisiert. Viele Hersteller verwenden zudem proprietäre Lösungen, was sowohl Protokolle und Infrastrukturen als auch den angebotenen Funktionsumfang betrifft. In dieser Konstellation ergibt sich ein hohes Maß an *Heterogenität*.

Der Umgang mit dieser Heterogenität ist eine zentrale Aufgabe bei der Realisierung von eHome-Systemen und der Weiterentwicklung der Konzepte des Ubiquitous Computing im Allgemeinen [EG01, dCYG08]. Dieser Aspekt wird durch den Punkt 2 (Interoperabilität) der von EDWARDS und GRINTER beschriebenen sieben Herausforderungen bei der Entwicklung von eHome-Systemen adressiert (vgl. Abschnitt 2.3 und [EG01]). Da es derzeit im Bereich von eHome-Systemen weder seitens der Hardware noch seitens der Software übergreifende Standards gibt, verstärkt sich das Problem weiter. Hersteller wollen ihre jeweiligen Lösungen als Standard für den gesamten Markt etablieren, um sich eine bessere wirtschaftliche Position im Markt zu sichern. Darüber hinaus wird jeder Hersteller versuchen, seine Produkte durch spezifische Funktionalitäten von der Konkurrenz abzugrenzen. Dies ist ein wichtiges Kriterium, um einen Vorteil auf dem Markt zu erlangen und die Weiterentwicklung von Produkten zu fördern. Aus diesen Gründen ist nicht zu erwarten, dass sich in absehbarer Zeit ein übergreifender Standard durchsetzen wird.

Hersteller werden daran interessiert sein, sämtliche ihrer Dienste und Geräte miteinander komponierbar zu machen. Dies ist innerhalb eines Unternehmens realisierbar. Auch eine Gruppe von Herstellern kann sich um ein gemeinsames Vorgehen bemühen und Interoperabilität unter ihren Diensten und Geräten erreichen, wenn es möglich ist, sich auf einen Standard zu einigen. Sind bei einem eHome-System nur diese wenigen Hersteller involviert, so ist eine geringe Heterogenität zu erwarten. Im Idealfall könnten die Dienste und Geräte direkt miteinander interagieren. Es muss jedoch davon ausgegangen werden, dass auch andere Hersteller, außerhalb eines möglicherweise gegebenen Standards, involviert sind. Deren Dienste und Geräte müssen ebenfalls genutzt werden können. Außerdem ist bei der Entwicklung von Top-Level-Diensten nicht bekannt, welcher von mehreren eventuellen Gerätestandards in einem eHome eingesetzt wird. Da die Dienste auch in anderen Umgebungen zum Einsatz kommen sollen, z. B. wenn Benutzer ihre Dienste in andere Umgebungen mitnehmen, ist eine Festlegung auf einen bestimmten Standard im Allgemeinen nicht möglich.

5.1.1 Problemfelder

Die vorhandene Hardware und Infrastruktur variiert nicht nur von einer Umgebung zur anderen, sondern kann selbst innerhalb ein und derselben Umgebung sehr heterogen sein. Es ist außerdem davon auszugehen, dass eHomes sich schrittweise aus bereits vorhandenen Wohnumgebungen entwickeln. Dies geschieht dadurch, dass zunehmend netzwerkfähige Geräte und in Software realisierte Funktionalitäten zum Einsatz kommen. Diese zu erwartende Entwicklung wird auch durch den Punkt 1 (das „zufällige Smart Home“) in [EG01] adressiert (vgl. Abschnitt 2.3). Daher muss davon ausgegangen werden, dass die verfügbare Hardware und die genutzten Dienste im eHome zu sehr unterschiedlichen Zeiten entwickelt wurden. Ein eHome-System wird also typischerweise weder aus eine Hand geliefert noch entsteht es zu einem einzigen klar definierten Zeitpunkt. Auch aufgrund der unterschiedlichen Entwicklungszeiten muss damit gerechnet werden, dass unterschiedliche Technologien zum Einsatz kommen, da der Stand der Technik sich schnell ändern kann. Somit kann auch aus diesem Grund nicht von einem standardisierten Umfeld ausgegangen werden. Trotz der gegebenen Heterogenität muss aber die Basisschicht aus Geräten und der zugehörigen Infrastruktur für Top-Level-Dienste nutzbar sein. Nur so können eHome-Systeme und Ubiquitous Computing zur Anwendung kommen.

Ein wesentliches Problem einer heterogenen Umgebung sind Schnittstellen, die nicht zusammenpassen. Selbst wenn es zu einer gegebenen Funktionalität passende Dienste gibt, die diese Funktionalität anbieten, kann so keine Kommunikation zwischen den Diensten stattfinden. Eine Interoperabilität ist also nicht gegeben. Inkompatible Schnittstellen werden in eHomes jedoch der Regelfall sein, da eine Standardisierung schon auf gröberer Ebene schwer erreichbar ist. Ein detaillierter Standard, der genaue Beschreibungen aller im eHome denkbaren Schnittstellen umfasst, ist darum unwahrscheinlich. Eine flexible Anpassbarkeit von Diensten an unterschiedliche Schnittstellen ist aus diesem Grund eine wesentliche Voraussetzung für die Realisierung von eHome-Systemen.

Inkompatibilitäten können aber auch auf der Ebene von Funktionalitäten auftreten. Häufig ist es möglich, eine Funktionalität auf unterschiedliche Art und Weise zu realisieren. Insbesondere bei abstrakteren Funktionalitäten bestehen viele Möglichkeiten, diese umzusetzen. Dies ist ein Vorteil, da so mehr Flexibilität gegeben ist, eine Funktionalität je nach den Spezifika des jeweiligen eHomes umzusetzen. Damit diese Flexibilität im eHome nutzbar ist, muss jedoch ein tiefgehendes Verständnis der Funktionalitäten gegeben sein. Nur so können die verschiedenen Realisierungsoptionen erkannt werden. Hier ergibt sich die Heterogenität aus der mit einer Funktionalität assoziierten Semantik. Unabhängig von der Schnittstelle, die ein Dienst erwartet, wird auch eine bestimmte Semantik der benötigten Funktionalitäten des Dienstes erwartet. Hier muss eine ausreichende Flexibilität gegeben sein, da sonst, selbst wenn die Kommunikation möglich ist, dennoch keine sinnvolle Interoperabilität erreicht wird. Es müssen außerdem verschiedenen Realisierungen einer Funktionalität ermöglicht und zugelassen werden.

5.1.2 Zielsetzung des semantischen Ansatzes

Die vorhandene Heterogenität in einem eHome-System muss sowohl vor den Entwicklern der Dienste als auch vor den eigentlichen Nutzern des eHomes verborgen werden. Die Anwendungslogik von Diensten soll nur einmalig entwickelt werden und in verschiedenen eHomes wiederverwendet werden können. Dies ist ein wesentlicher Aspekt bei der Entwicklung kostengünstiger eHome-Systeme. Dazu ist es jedoch nötig, dass die einmal entwickelte Anwendungslogik auch tatsächlich ohne manuelle Anpassungen in den verschiedenen Umgebungen zum Einsatz kommen kann. Es muss eine möglichst hohe Flexibilität erreicht werden, damit Dienste in vielen Umgebungen verwendet werden können. Im Rahmen dieser Arbeit werden Inkompatibilitäten auf der Ebene von Schnittstellen und Funktionalitäten von eHome-Diensten betrachtet. Die folgenden konkreten Herausforderungen werden adressiert.

- Geräte desselben Typs bieten die gleiche Funktionalität an, jedoch über unterschiedliche Schnittstellen. Dieser Fall tritt auf, wenn die Geräte von verschiedenen Herstellern stammen, die keinem gemeinsamen Standard folgen.
- Dieselbe Funktionalität kann auf unterschiedliche Art und Weise realisiert werden. Verschiedene Geräte und Kombinationen von Geräten können zur Realisierung einer Funktionalität verwendet werden. Welche Möglichkeit vorzuziehen ist, hängt von den Gegebenheiten des spezifischen eHomes und den Anforderungen der Benutzer ab.

Da ein durchgehender Standard, wie oben beschrieben wurde, nicht zu erwarten ist, müssen alternative Wege gefunden werden, die Komponierbarkeit und Interoperabilität von eHome-Diensten sicherzustellen. In dieser Arbeit wird ein Ansatz auf Basis semantischer Beschreibungen von Diensten und Funktionalitäten verfolgt. Auf Grundlage der Dienstbeschreibungen ist dann eine automatische Adaption von Diensten möglich. So kann die Interoperabilität der Dienste trotz fehlender syntaktischer Standards erreicht werden.

Die semantische Modellierung eröffnet zahlreiche verschiedene Erweiterungsmöglichkeiten der bestehenden Konzepte, die im Rahmen früherer Arbeiten umgesetzt wurden. Nicht alles was dabei prinzipiell vorstellbar ist, ist auch sinnvoll umzusetzen und nicht alles was umsetzbar ist, ist im Rahmen dieser Arbeit zu realisieren. Die Ziele des in dieser Arbeit vorgestellten Ansatzes sind:

- Die Erweiterung der Dienstbeschreibung, um eine semantische Ebene auf höherem Abstraktionsniveau einzuführen, damit unabhängig von syntaktischen Details Aussagen über Dienste gemacht werden können.
- Dienste sollen auf Basis ihrer semantischen Beschreibung komponiert werden anstelle syntaktischer Schnittstellen. Dabei sollen auch zuvor spezifizierte Zusammenhänge zwischen den semantischen Konzepten Verwendung finden, um beispielsweise nicht verfügbare Funktionalitäten soweit möglich zu substituieren.
- Auf Basis der semantischen Dienstbeschreibung soll es möglich sein, die für die Kommunikation mit dem Dienst zu verwendende Syntax abzuleiten.
- Syntaktische Inkompatibilitäten zwischen Diensten sollen soweit möglich mittels der semantischen Beschreibung zu überwinden sein.

Diese Anforderungen sind nicht trivial und im Allgemeinen nicht ohne weiteres zu erreichen. Im Rahmen dieser Arbeit ist aber zum einen die Anwendungsdomäne auf eHome-Systeme eingeschränkt und zum anderen soll der Fokus zunächst auf die Basisdienste im eHome gelegt werden. Dadurch ist das Anwendungsfeld hinreichend speziell, sodass Annahmen getroffen werden können, welche die Entwicklung einer Lösung erleichtern. So werden die oben genannten Ziele realisierbar. Im folgenden Abschnitt werden zunächst einige Grundlagen zum Begriff *Semantik* und zur *semantischen Modellierung* eingeführt, bevor der in dieser Arbeit entwickelte Ansatz im Detail vorgestellt wird.

5.2 Semantik und Ontologien

Bisher wurde bereits vielfach der Begriff *Semantik* verwendet. Was unter Semantik insbesondere im Rahmen dieser Arbeit verstanden wird, ist in diesem Abschnitt erläutert. Ein weiterer Begriff, der im Zusammenhang mit der semantischen Adaption in dieser Arbeit eine Rolle spielt, ist der *Ontologie*-Begriff. Unter *Ontologie* werden sehr unterschiedliche Dinge verstanden, daher ist für das Verständnis des semantischen Adaptionsansatzes auch hier eine genauere Festlegung erforderlich. Auf Basis dieser Begriffe wird am Ende des Abschnitts ein Überblick über den Ansatz zur semantischen Adaption gegeben.

5.2.1 Begriffsklärung

Semantik bezeichnet die Bedeutungslehre bezüglich sprachlicher Symbole. In der Informatik werden formale Sprachen betrachtet, sodass entsprechend auch von formaler Semantik gesprochen wird. Während die *Syntax* festlegt, welcher formalen Ordnung die Symbole einer Sprache genügen müssen, beschreibt die Semantik eines Symbols dessen Bedeutung. In formalen Sprachen muss den sprachlichen Symbolen eine Bedeutung zugewiesen werden, damit eine Interpretation möglich ist. Dazu wird eine Repräsentationssprache verwendet, mittels der eine solche Zuweisung festgelegt werden kann. Dies geschieht analog zur Interpretation natürlicher Sprachen, in denen die Wörter eine Bedeutung übermitteln. In [OR23] untersuchen OGDEN und RICHARDS was unter *Bedeutung* zu verstehen ist und welche Mechanismen der Interpretation von sprachlichen Symbolen zugrunde liegen. Das *semiotische Dreieck* nach OGDEN und RICHARDS veranschaulicht diesen Interpretationsvorgang. Es stellt den Zusammenhang zwischen den sprachlichen Symbolen, der ihnen zugeordneten Bedeutung und den mit dieser Bedeutung verbundenen Objekten der realen Welt dar.

In Abbildung 5.1 ist das semiotische Dreieck am Beispiel des Begriffs „Haus“ dargestellt. Der Begriff „Haus“ ist ein Symbol, das für ein bestimmtes Bezugsobjekt steht. Durch Interpretation des Symbols wird ein Begriff aktiviert, d. h. eine bestimmte Bedeutung wird mit dem Symbol assoziiert. Ein solcher Begriff wird vom Interpretierenden dann mit einem oder mehreren Bezugsobjekten der realen Welt verbunden. Falls mehrere Bezugsobjekte existieren, muss versucht

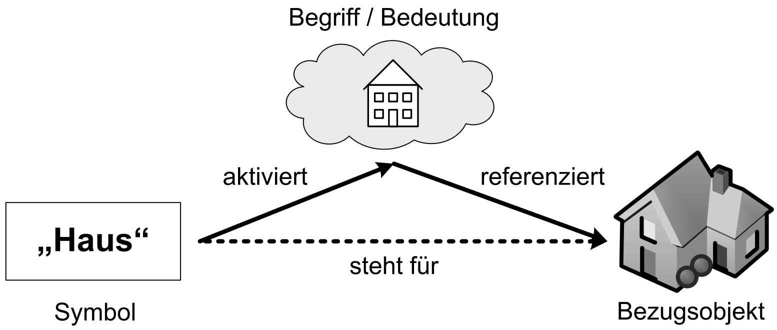


Abbildung 5.1: Semiotisches Dreieck.

werden, aus dem Kontext abzuleiten, welches gemeint ist. Im Beispiel aus Abbildung 5.1 steht der Begriff „Haus“ für ein Haus der realen Welt – in der Abbildung rechts angedeutet. Diese Assoziation ist lediglich *indirekt* über die durch das Symbol „Haus“ aktivierte Bedeutung gegeben.

In [HR04] wird von HAREL und RUMPE der Begriff *Semantik* in Bezug auf die Definition von Modellierungssprachen, insbesondere die UML (siehe auch Abschnitt 3.3.2), analysiert. Die Semantik der UML wurde bereits kurze Zeit nach ihrer Standardisierung im Jahr 1997 intensiv diskutiert [HR00]. Eine präzise Semantikdefinition der zahlreichen Diagrammtypen der UML ist nicht Bestandteil der Spezifikation, es wird im Wesentlichen lediglich die abstrakte Syntax der UML erfasst. In [KR97] werden die zunehmenden Bestrebungen zur Standardisierung von Modellierungssprachen diskutiert. Dabei wird neben einer präzisen Syntax auch eine präzise Semantik gefordert. Eine Übersicht der derzeitigen Forschung in der modellgetriebenen Entwicklung und der dabei auftretenden Herausforderungen wird in [FR07] gegeben. Die bisherigen Erfahrungen haben gezeigt, dass Modellierungssprachen eine formal spezifizierte Semantik erfordern, wenn sie zur modellgetriebenen Softwareentwicklung eingesetzt werden sollen. Im Laufe der Zeit wurden bereits verschiedene Ansätze zur Semantikdefinition der UML unternommen, z. B. mittels formaler Spezifikationstechniken wie in [EFLR98, EFLR99]. Dort werden Schritte für eine formale Semantikspezifikation der UML definiert, dabei wird die Z-Notation [Spi92] eingesetzt. Die Formalisierungsstrategie zur Präzisierung der informell spezifizierten UML-Semantik wird in [ELFR99] vertieft. Dort werden Vorteile einer präzisen Semantikdefinition erläutert und es wird ein semantisches Kernmodell der UML entwickelt. Eine ausführliche Beschreibung eines Systemmodells der UML als mögliche Grundlage der UML-Semantikdefinition ist in [BCGR08, BCGR09a, BCGR09b] beschrieben. Die genannten Arbeiten zeigen die große Bedeutung einer präzisen Semantikbeschreibung im Bereich von Modellierungssprachen, aber auch, dass der Weg zu solchen formalen Beschreibungen aufwendig ist.

Um in einer Modellierungssprache formulierte Daten zu verstehen, muss jedes syntaktische Element seiner jeweiligen Bedeutung zugeordnet werden [HR04]. Wie im oben beschriebenen semiotischen Dreieck, ist dazu eine Abbildung auf die semantische Domäne erforderlich. Die semantische Domäne muss dabei ihrerseits durch eine Sprache oder einen Formalismus mit definierter Semantik beschrieben sein. Auch in der vorliegenden Arbeit wird eine solche Abbildung verwendet, die die syntaktischen Konstrukte den zugehörigen semantischen Konzepten zuordnet. Hinsichtlich der semantischen Abbildung bestehen also Gemeinsamkeiten zwischen dem Semantikbegriff in [HR04] und dieser Arbeit. Hier wird jedoch nicht die vollständige formale Beschreibung der Semantik einer Modellierungssprache betrachtet, sondern die semantische Beschreibung von eHome-Diensten, insbesondere der syntaktischen Schnittstellen dieser Dienste, die nur einen kleinen Teil der zugrunde liegenden Sprache verwenden. Die Elemente der syntaktischen Schnittstellen müssen auf domänenspezifische semantische Konzepte abgebildet werden, damit eine semantische Beschreibung von Dienstfunktionalitäten erreicht wird.

Wie in Abschnitt 5.1.2 erläutert, hat die semantische Dienstbeschreibung das Ziel, eine flexiblere Dienstkomposition zu ermöglichen, die nicht nur auf Basis der syntaktischen Dienstschnittstellen erfolgt, sondern auch in einem heterogenen Dienstumfeld durchgeführt werden kann. Dazu wird eine semantische Beschreibung verwendet, die nicht die allgemeine Semantik von Schnittstellen in der für die Dienstimplementierung verwendeten Programmiersprache Java beschreibt. Es werden stattdessen die von einer Schnittstelle zur Verfügung gestellten Funktionalitäten in Bezug auf die Anwendungsdomäne *eHomes* spezifiziert. Somit liegt eine andere Abstraktionsebene der semantischen Beschreibung vor. Mittels dieser semantischen Beschreibung können syntaktisch inkompatible Dienste so adaptiert werden, dass eine Interaktion zwischen ihnen möglich ist.

Die für eine semantische Abbildung verwendete semantische Domäne kann auf verschiedene Weisen spezifiziert werden, von ausformuliertem Text bis hin zu einer streng mathematischen Formalisierung. In dieser Arbeit werden zur Modellierung der semantischen Domäne sogenannte Ontologien verwendet, die weiter unten eingeführt werden. Dadurch wird ein mittlerer Grad an Formalisierung erreicht, der strukturierter ist als ein natürlichsprachlicher Text, jedoch weniger präzise als eine mathematische Formalisierung. Der Semantikbegriff in dieser Arbeit bezieht sich auf eine anwendungsbezogene Betrachtungsebene. Diese Form der Beschreibung ermöglicht es nicht, beliebige Implementierungen von eHome-Diensten hinsichtlich ihrer Semantik zu analysieren. Die semantische Abbildung muss vielmehr für jeden Dienst manuell spezifiziert werden, z. B. durch den Entwickler des Dienstes. Es ergibt sich also eine andere Vorgehensweise als bei der in [HR04] betrachteten Semantikbeschreibung von Modellierungssprachen. Somit wird hier auch keine allgemeine Abbildungsfunktion definiert, mittels der für beliebige Dienstschnittstellen die Semantik abgeleitet werden könnte. Stattdessen ergibt sich die Abbildungsfunktion aus den manuell erstellten Beschreibungen der einzelnen Dienste. Dadurch wird zwar keine automatische Interpretation der Dienste hinsichtlich ihrer Funktionalitäten ermöglicht, die manuell erstellte Abbildung der Schnittstellenelemente erlaubt jedoch die semantikbasierte Komposition und Adaption von Diensten, was für die Ziele der semantischen Adaption im Rahmen dieser Arbeit hinreichend ist.

Semantische Dienstbeschreibungen werden, wie oben bereits skizziert, zur Komposition von eHome-Diensten verwendet. Diese Beschreibungen dienen dazu, den angebotenen und benötigten Funktionalitäten der Dienste eine bestimmte Bedeutung zuzuweisen. Dann können diese Funktionalitäten durch die Laufzeitumgebung der eHome-Dienste interpretiert werden und es kann bei der Konfigurierung überprüft werden, welche Dienste miteinander komponiert werden können. Um die Semantik von Dienstfunktionalitäten beschreiben zu können, ist zunächst eine Repräsentationssprache für die Beschreibung der semantischen Domäne erforderlich. Damit werden dann die Elemente, die in der Anwendungsdomäne *eHomes* auftreten können, beschrieben. Das Wissen über die Anwendungsdomäne muss also erfasst und in eine explizite Form gebracht, d. h. formalisiert werden. Erst dadurch wird eine automatisierte Verarbeitung ermöglicht.

In der Informatik wird die *Wissensrepräsentation* häufig dem Bereich der künstlichen Intelligenz zugeordnet. Wissen ist in vielen Fällen nur in impliziter Form vorhanden. Aufgabe der Wissensrepräsentation ist es, solches implizite Wissen in eine explizite und maschinenlesbare Form zu bringen, um es so in einer Softwareanwendung verwenden zu können. Als semantische Modelle zur Wissensrepräsentation werden häufig sogenannte *Ontologien* eingesetzt. Der Begriff *Ontologie* stammt ursprünglich aus der Philosophie und bezeichnet die Lehre vom *Sein* bzw. vom *Seienden*, also von dem, was ist und was nicht ist [SN99]. Ausgehend von ARISTOTELES ist die Ontologie ein Teilbereich der Metaphysik und fragt nach der Herkunft alles Seienden [KBW07]. Die Ontologie beschäftigt sich mit Konzepten, die Individuen, Objekte, Eigenschaften, Relationen, Zustände, Ereignisse, Prozesse etc. zum Gegenstand haben, sowie deren wechselseitigen Beziehungen. Sie stellt die Frage danach *was* ist, im Gegensatz zur Frage *wie* etwas ist. In der Informatik wird der Ontologie-Begriff formaler gefasst. Die bekannteste Definition stammt von GRUBER [Gru93]:

> „An ontology is an explicit specification of a conceptualization."

Unter einer Ontologie wird also eine *explizite Spezifikation einer Konzeptualisierung* verstanden. Dabei wird unter *Konzeptualisierung* eine abstrakte, vereinfachte Sicht auf einen Ausschnitt der realen Welt verstanden. Dieser Ausschnitt wird in *Konzepte* eingeteilt, über die in der Ontologie weitere Aussagen getroffen werden. Die formale Spezifikation der Konzepte bildet die Ontologie. Für die Wissensmodellierung legt die Ontologie fest, nach welchen Regeln die Realität angemessen wiedergegeben werden kann und welche Dinge und Zusammenhänge überhaupt gegeben sind. Somit legt die Ontologie eine Art Grammatik der modellierten Welt fest sowie ein Vokabular oder eine Enzyklopädie, der für die Modellierung relevanten Entitäten. Ontologien unterscheiden sich von Wissensdatenbanken (engl. *Knowledge Bases*) dadurch, dass sie keine zustandsabhängigen Informationen enthalten. Ontologien beschreiben die Konzepte einer Anwendungsdomäne und welche Zusammenhänge zwischen diesen bestehen. Eine Wissensdatenbank hingegen speichert insbesondere Zustandsinformationen, d. h. welche Entitäten der Anwendungsdomäne in einem konkreten Fall existieren und in welchen Zuständen sie sich aktuell befinden.

Es lassen sich unterschiedliche Arten von Ontologien unterscheiden. Dabei kann zunächst nach der Ausdrucksstärke der Ontologie unterschieden werden. Eine grundlegende Art von Ontologien ist das *Vokabular*, das üblicherweise die Begriffe einer speziellen Anwendungsdomäne umfasst. Dabei macht nicht das Vokabular als solches die Ontologie aus, sondern vielmehr die durch die Begriffe des Vokabulars erfassten Konzepte der Anwendungsdomäne [CJB99]. Ontologien in der Funktion eines Vokabulars oder einer Taxonomie werden häufig auch

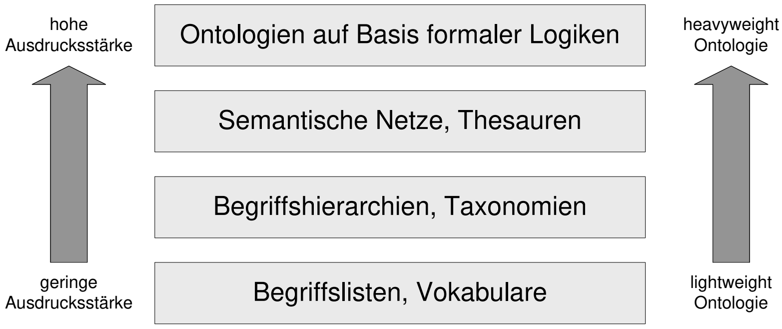


Abbildung 5.2: Arten von Ontologien nach Ausdrucksstärke.

lightweight Ontologien genannt. Demgegenüber heißen Ontologien, die weitere strukturelle Informationen über die Anwendungsdomäne modellieren, auch *heavyweight Ontologien* [GFC04]. Abbildung 5.2 zeigt eine Übersicht verschiedener Arten von Ontologien, die nach ihrer Ausdrucksstärke unterschieden sind. Die einfachste Form von Ontologien sind die bereits erwähnten Vokabulare, also einfache Begriffslisten, die die Konzepte der Anwendungsdomäne benennen. Taxonomien sind ausdrucksstärker. Sie beschreiben Begriffshierarchien, d. h. die Begriffe werden hier bereits strukturiert in Form einer Hierarchie erfasst. Noch ausdrucksstärker sind Thesauren oder semantische Netze, die die Konzepte der Anwendungsdomäne zueinander in Beziehung setzen. Eine besonders hohe Ausdrucksstärke haben schließlich heavyweight Ontologien auf Basis formaler Logiken. Diese erlauben das Schlussfolgern, wodurch umfassendes Wissen über die Anwendungsdomäne ableitbar ist.

Ontologien lassen sich jedoch nicht nur nach Ausdrucksstärke unterscheiden. Auch nach dem Grad ihrer Allgemeinheit ist eine Klassifikation möglich. In Abbildung 5.3 sind dazu verschiedene Ontologiearten dargestellt, die aufeinander aufbauen. Der Grad an Allgemeinheit nimmt nach oben hin zu.

1. *Top-Level-Ontologien* (engl. auch *Upper Ontologies*) beschreiben sehr allgemeine Konzepte, die sich nicht auf eine spezifische Anwendungsdomäne beziehen. Hier werden grundlegende Konzepte wie Raum, Zeit, Objekt, Ereignis etc. modelliert. Da die Konzepte sehr allgemein sind, können sie in den verschiedensten Anwendungsdomänen Verwendung finden.

2. *Domänenontologien* beschreiben die Konzepte, d. h. das Vokabular, einer spezifischen Anwendungsdomäne.

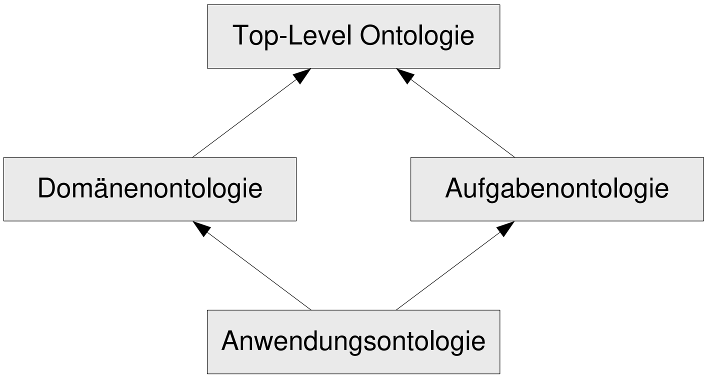


Abbildung 5.3: Arten von Ontologien nach ihrem Grad an Allgemeinheit. (in Anlehnung an [Gua98])

3. *Aufgabenontologien* beschreiben die Aktivitäten, die in einer spezifischen Anwendungsdomäne anfallen.
4. *Anwendungsontologien* beschreiben Konzepte, die innerhalb einer spezifischen Anwendung relevant sind. Diese beziehen sich sowohl auf das Vokabular der entsprechenden Anwendungsdomäne als auch auf die zugehörigen Aktivitäten. Daher sind Anwendungsontologien häufig Spezialisierungen sowohl von Domänenontologien als auch von Aufgabenontologien.

Um eine Ontologie zu modellieren, müssen die dabei verwendeten Sprachelemente und deren Zusammenhänge zuvor definiert werden. Dazu kann ebenfalls eine Ontologie verwendet werden, die z. B. Klassen, Attribute, Relationen etc. festlegt. Eine solche Ontologie ist in Bezug auf die oben beschriebenen Ontologiearten der Meta-Ebene zuzuordnen und wird auch *Darstellungsontologie* (engl. *Representation Ontology*) genannt. In [Gru93] wird von GRUBER eine sogenannte *Frame Ontology* als Darstellungsontologie vorgestellt. Die verschiedenen Modellierungsebenen werden im folgenden Abschnit 5.2.2 genauer diskutiert.

5.2.2 Semantikbasierte Komposition

Bereits im Ansatz von NORBISRATH wurde die Dienstkomposition auf Basis semantischer Auszeichnungen durchgeführt. Dazu wurden die sogenannten *Semantic Labels* (vgl. Abschnitt 3.4) eingesetzt. Die Ontologie der Semantic La-

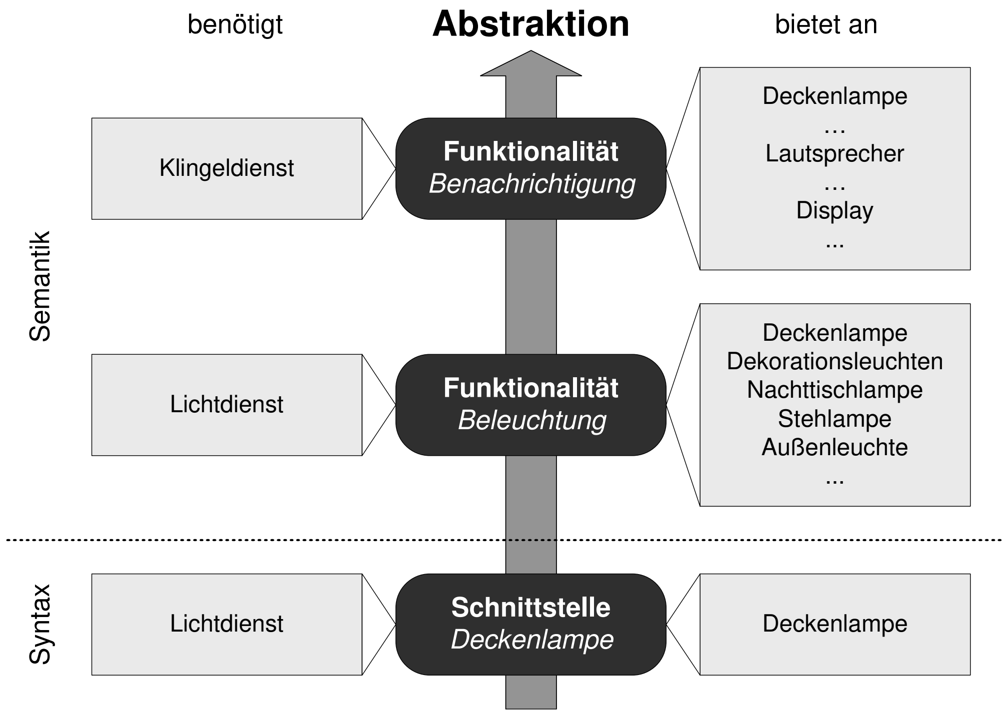


Abbildung 5.4: Abstraktionsebenen der semantischen Modellierung.

bels hatten dabei den Charakter eines Vokabulars, es handelte sich also um eine vergleichsweise ausdrucksschwache Art von Ontologie. Grundsätzlich bedeutet *semantische Komposition*, dass das Matching der Dienste, d. h. das Auffinden passender Dienste zur Komposition, auf Basis einer semantischen Beschreibung erfolgt und nicht auf Basis der Syntax von Schnittstellen. Um die Interoperabilität der komponierten Dienste zu erreichen, muss jedoch in jedem Fall auch eine Kompatibilität auf der semantischen Ebene erreicht werden. Im Ansatz von NORBISRATH wird das Matching zwar auf Basis der Semantic Labels durchgeführt, damit die daraus resultierende Dienstkomposition jedoch ausführbar ist, wird implizit vorausgesetzt, dass bei einem Match auch eine syntaktische Übereinstimmung vorliegt. Konkret müssen im Ansatz von NORBISRATH die States und ihre Verwendung übereinstimmen. Dies ist jedoch ein Fall, der in der Realität nur selten eintreten wird. Es muss also ein weiterführender Ansatz entwickelt werden, wenn die semantische Dienstkomposition in realen Szenarien umgesetzt werden soll.

Abbildung 5.4 dient der genaueren Erläuterung der semantischen Komposition. Dargestellt sind verschiedene Abstraktionsebenen der semantischen Modellierung von eHome-Diensten. Unten in der Abbildung ist die syntaktische Ebene dargestellt. Hier erfolgt die Komposition der Dienste auf der Ebene ihrer Schnittstellen. Ein Lichtdienst, der die Beleuchtungssituation in der Umgebung des Benutzers steuert, muss z. B. mit einer Lampe verbunden werden, damit er seine Funktionalität erfüllen kann. Angenommen dazu steht in der Umgebung eine Deckenlampe zur Verfügung, so muss die Kommunikation über eine spezifische Schnittstelle Deckenlampe erfolgen. Auf syntaktischer Ebene müssen also beide Dienste entsprechend einer speziellen, zuvor festgelegten Schnittstelle implementiert werden, damit eine Komposition möglich ist.

Betrachtet man nun die semantische Ebene, so kann die Komposition der Dienste auf einer höheren Abstraktionsebene erfolgen. Hier werden nicht mehr die Schnittstellen der Dienste betrachtet, sondern die *Bedeutung* der von ihnen angebotenen oder benötigten Funktionalitäten. Auf dieser Ebene kann unabhängig von den Signaturen und syntaktischen Details der Dienstschnittstellen untersucht werden, ob Dienste semantisch zusammenpassen. In der Mitte der Abbildung wird dazu die Funktionalität Beleuchtung betrachtet. Der Lichtdienst benötigt diese Funktionalität, um ausgeführt werden zu können. Durch diese Anforderung ist keine spezielle Schnittstelle vorgegeben. Die Funktionalität Beleuchtung kann von vielen verschiedenen Diensten angeboten werden, die jeweils unterschiedliche Schnittstellen implementieren. In der Abbildung sind als Beispiele verschiedene Lampenarten dargestellt: Deckenlampe, Dekorationsleuchte, Nachttischlampe, Stehlampe, Außenleuchte etc. All diese Lampenarten bieten die Funktionalität Beleuchtung an und sind daher geeignet, die vom Lichtdienst benötigte Funktionalität zur Verfügung zu stellen.

Auf der obersten Ebene der Abbildung ist eine weitere Abstraktionsebene der Semantik dargestellt. Funktionalitäten können einen unterschiedlichen Grad an Abstraktion haben. Im dargestellten Beispiel benötigt ein Klingeldienst die abstrakte Funktionalität Benachrichtigung, d. h. er muss die Möglichkeit haben, den Benutzer zu benachrichtigen, wenn eine Person an der Haustür die Klingel betätigt. Diese Funktionalität ist abstrakt, weil sie durch verschiedene konkretere Funktionalitäten realisiert werden kann. Um den Benutzer zu benachrichtigen, kann z. B. ein visuelles Signal erzeugt werden, was mittels einer Deckenlampe realisierbar wäre. Es kann aber auch ein akustisches Signal zur Benachrichtigung verwendet werden. Dazu könnte ein Lautsprecher in der Umgebung eingesetzt werden. Ebenso wäre es möglich, dem Benutzer auf einem Display einen Hinweis einzublenden. Wenn der Benutzer beispielsweise fern sieht, so kann für eine bestimmte Zeit ein Hinweis auf dem Bildschirm angezeigt werden. Auch dies wäre eine Alternative, um die Funktionalität Benachrichtigung umzusetzen. Dieses Beispiel zeigt, dass innerhalb der semantischen Ebene mehrere Abstraktionsschichten unterschieden werden können.

Die Betrachtung der semantischen Ebene bietet verschiedene Vorteile für die Dienstkomposition. Zum einen ist es wichtig, dass passende Dienste für benötigte Funktionalitäten gefunden werden. Wenn Dienste eine gemeinsame Schnittstelle haben und direkt miteinander komponiert werden können, so kann dies nicht ohne Weiteres mit semantischer Kompatibilität gleichgesetzt werden. Umgekehrt kann bei semantischer Kompatibilität nicht vorausgesetzt werden, dass die Schnittstellen der beteiligten Dienste übereinstimmen. Dies wird nur in den wenigsten Fällen gegeben sein. Zunächst wird eine semantische Beschreibung benötigt, damit semantische Kompatibilität überhaupt festgestellt werden kann. Dann muss ein Mechanismus vorhanden sein, der Inkonsistenzen auf syntaktischer Ebene zu überwinden ermöglicht. Auf diese Weise stehen mehr Dienste für die Komposition zur Verfügung, da die strikte Voraussetzung exakt übereinstimmender Schnittstellen aufgelockert wird. Darüber hinaus ergibt sich die Möglichkeit, auf semantischer Ebene die Flexibilität der Dienstkomposition zu erhöhen und Alternativen für die Realisierung abstrakter Funktionalitäten zu schaffen. Diese können dann auch spezifische Benutzeranforderungen berücksichtigen.

Im oben genannten Beispiel können z. B. bei der Realisierung der Funktionalität Benachrichtigung, die der Klingeldienst benötigt, körperliche Einschränkungen des Benutzers berücksichtigt werden. Eine ältere Person, die unter Schwerhörigkeit leidet, kann z. B. durch ein visuelles Signal benachrichtigt werden. Auf diese Weise wird ein Mehrwert für den Benutzer geschaffen, da Geräte so für Funktionalitäten genutzt werden können, die nicht zu ihren originären Aufgaben gehören. Eine Lampe für die Benachrichtigung des Benutzers einzusetzen, gehört nicht zu den ursprünglichen Aufgaben der Lampe, ist aber eine sinnvolle Realisierungsmöglichkeit. Die Schnittstelle der Lampe ist nicht darauf ausgelegt, die Funktionalität Benachrichtigung direkt durch entsprechende Methoden zu unterstützen. Erst durch die semantische Modellierung ist es möglich, einen weitergehenden Nutzen und damit Mehrwert für den Anwender zu schaffen.

Im Ansatz von NORBISRATH werden wie oben bereits beschrieben die semantischen Ebenen nur unzureichend unterstützt. Die Semantic Labels stellen eine semantische Auszeichnung dar, es muss jedoch zu jedem Semantic Label eine eindeutige Schnittstelle geben, die von den Diensten verwendet bzw. implementiert wird. Der Mechanismus ermöglicht also keinerlei Abstraktion von den syntaktischen Details der Schnittstellen. Ebenso werden keine höheren semantischen Abstraktionsebenen unterstützt, sodass das Potential der semantischen Beschreibung nicht ausgeschöpft werden kann. Durch den in dieser Arbeit entwickelten Ansatz werden die Abstraktion von syntaktischen Details sowie die Modellierung von Zusammenhängen innerhalb der semantischen Ebene unterstützt.

5.2.3 Adaption

Die Komponierbarkeit vorgefertigter Softwarekomponenten ist ein generelles Problem. Das Konzept komponentenbasierter Software basiert auf der Annahme, dass vorgefertigte Softwarebausteine miteinander kompatibel sind und unmittelbar miteinander komponiert werden können, um die gewünschten Funktionalitäten umzusetzen [SGM02]. Diese Annahme ist allerdings in realen Anwendungen häufig nicht gegeben [BBG+06], weil z. B. Schnittstellen nicht genau passen oder die gegebenen Anforderungen nicht exakt von der Komponente abgedeckt werden. Um Komponenten zu einer Anwendung zu komponieren oder Komponenten in eine bestehende Anwendung zu integrieren, ist daher häufig eine Adaption erforderlich.

Wenn Komponenten nicht zusammenpassen, wird auch von einem *Mismatch* gesprochen. Komponenten können aus verschiedenen Gründen nicht zusammenpassen. In Abbildung 5.5 sind verschiedene Schnittstellenmodelle und die zugehörigen Arten von Mismatches aufgeführt. Auf der ersten und untersten Ebene wird die *Syntax* der Schnittstellen betrachtet. Mismatches treten hier in den Signaturen der Schnittstellenoperationen auf. Diese Ebene entspricht der syntaktischen Ebene der Überlegungen in Abschnitt 5.2.2. Auf der zweiten Ebene wird das *Verhalten* einer Komponente betrachtet. Wird das zugesicherte Verhalten einer Komponente nicht erfüllt, so handelt es sich um ein Mismatch auf der Verhaltensebene. Auf der dritten Ebene wird die Kommunikation zwischen

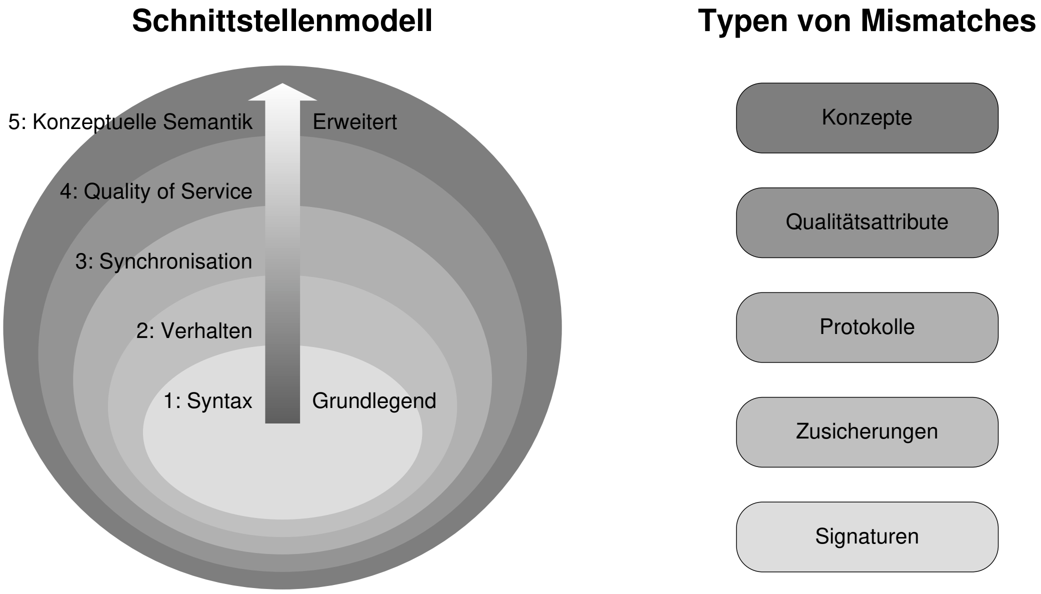


Abbildung 5.5: Verschiedene Typen von Mismatches. (in Anlehnung an [BBG$^+$06])

Komponenten betrachtet. Als Schnittstellenmodell dient hier die *Synchronisation* zwischen den Komponenten. Für jede Komponente existiert ein festgelegtes *Protokoll*, das die Möglichkeiten der Kommunikation definiert. Passen Protokolle nicht zusammen, so liegt ein Mismatch auf dieser Ebene vor. Auf der nächsten Ebene werden die *Qualitätseigenschaften* der von einer Komponente angebotenen Dienste festgelegt. Hier kann es vorkommen, dass Qualitätsattribute nicht zusammenpassen, was ein Mismatch bedeutet. Auf der fünften und obersten Ebene dient die *konzeptuelle Semantik* einer Komponente als Schnittstellenmodell. Hier werden die *Konzepte* erfasst, die einer Komponente und den von ihr realisierten Funktionalitäten zugrunde liegen. Auch hier können Mismatches auftreten, wenn Komponenten konzeptuell inkompatibel sind. Es kann z. B. der Fall auftreten, dass Konzepte gleich benannt, inhaltlich jedoch verschieden sind. Man spricht in dem Fall auch von einem *Homonym*. Auf der semantischen Ebene können so wie auch auf den anderen Ebenen noch verschiedene weitere Inkompatibilitäten auftreten [BBG$^+$06]. Die Ebene der konzeptuellen Semantik entspricht den semantischen Ebenen der Überlegungen in Abschnitt 5.2.2.

Werden bei der Entwicklung eines Softwaresystems vorgefertigte Komponenten eingesetzt, so ist häufig eine Adaption der Komponenten erforderlich. Unter

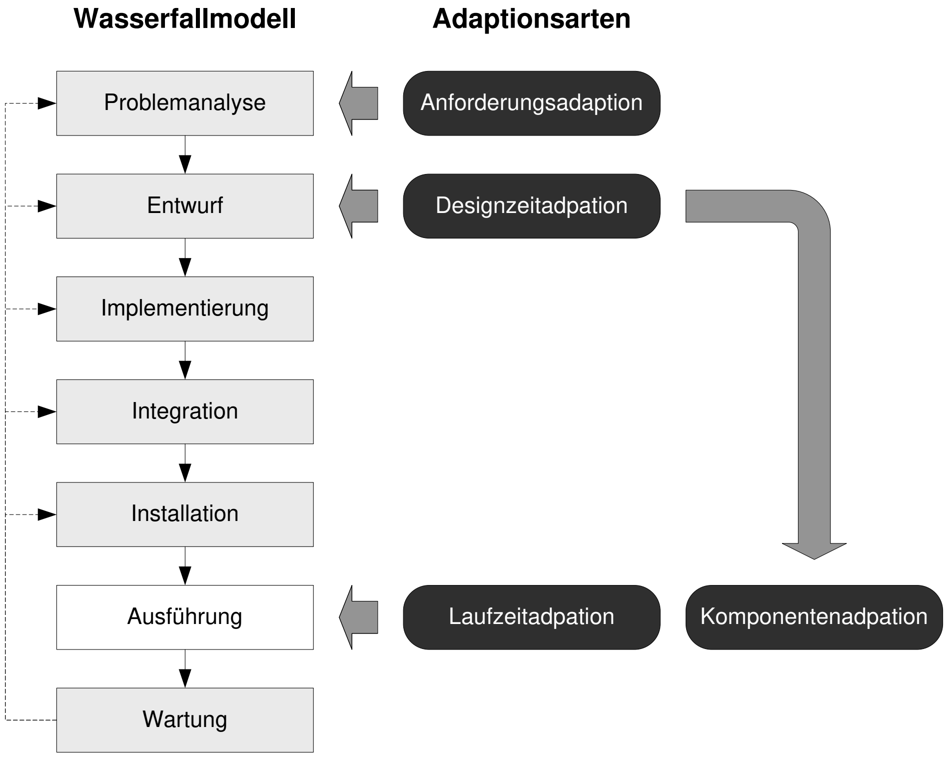


Abbildung 5.6: Zuordnung verschiedener Adaptionsarten zu den Phasen des Wasserfallmodells der Softwareentwicklung.

Adaption werden je nach Anwendungsfall unterschiedliche Dinge verstanden. In [BBG+06] werden drei Arten von Adaption unterschieden.

1. *Anforderungsadaption* bedeutet Adaption auf Grund neuer oder geänderter Anforderungen.
2. *Designzeitadaption* bedeutet Adaption von zueinander inkompatiblen Komponenten auf Architekturebene.
3. *Laufzeitadaption* bedeutet Adaption zur Laufzeit auf Grund unterschiedlichen Verhaltens von Komponenten in Abhängigkeit ihres Kontextes.

Die Arten unterscheiden sich in der jeweils betroffenen Phase des Softwareentwicklungsprozesses, in der die Adaption stattfindet. In Abbildung 5.6 ist der Zusammenhang mit dem Wasserfallmodell der Softwareentwicklung dargestellt. Für die Entwicklung von eHome-Systemen kann diese Zuordnung jedoch nicht ohne Weiteres übernommen werden. Die Architektur der eHome-Anwendung entsteht erst zur Laufzeit im Rahmen der dynamischen Dienstkomposition. Bei der Adaption der eHome-Anwendung handelt es sich daher nicht um eine Laufzeitadaption, sondern um eine Designzeitadaption, da die Architektur der Anwendung betroffen ist. Diese Adaption findet hier jedoch erst zur Laufzeit statt. Aus diesem Grund wird in dieser Arbeit anstatt *Designzeitadaption* der Begriff *Komponentenadaption* vorgezogen.

Da die Komponentenadaption erst zur Laufzeit erfolgt, muss sie automatisch durchgeführt werden. Die eHome-Anwendung wird während der Ausführung an die Anforderungen der Nutzer und die Gegebenheiten in der eHome-Umgebung angepasst. Die Dienstkomposition erfolgt im Wesentlichen automatisch, indem dynamische Änderungen schrittweise durch die strukturelle Adaption erfasst werden. Ein manuelles Eingreifen des Nutzers oder Administrators in die Dienstkomposition ist möglich, um bei Bedarf einzelne Bindungen anzupassen oder manuell zu bindende Ressourcen den gewünschten Diensten zuzuweisen. Dazu wird ein entsprechendes Werkzeug zur Interaktion angeboten. Die Adaption zueinander inkompatibler Dienste zur Laufzeit ist jedoch nicht interaktiv möglich. Eine manuelle Adaption der Dienste zur Laufzeit wäre zu Zeitaufwendig und wäre darüber hinaus vom Nutzer gar nicht durchführbar, da die erforderlichen Kenntnisse der Dienstentwicklung nicht gegeben sind. Ein manuelles Eingreifen bei der Komponentenadaption ist daher nicht möglich.

In [Hei98] wird außerdem zwischen *Softwareevolution*, *Adaption* und *Anpassung an Kundenwünsche* (engl. *Customization*) unterschieden.

1. *Softwareevolution* bedeutet, dass vom Komponentenentwickler eine Weiterentwicklung durchgeführt wird. Der Komponentenentwickler hat naturgemäß ein tiefes Verständnis der Komponentenarchitektur und hat Zugriff auf den Quelltext. Er hat daher viele Möglichkeiten, Anpassungen durch Weiterentwicklung vorzunehmen.

2. Eine *Adaption* hingegen wird nach der Klassifikation in [Hei98] durch den Anwendungsentwickler vorgenommen, der die Komponenten bei der Entwicklung seiner Anwendung einsetzt. Die Kenntnisse der Komponente und auch die Zugriffsmöglichkeiten auf deren Interna sind in diesem Fall deutlich eingeschränkt im Vergleich zur Softwareevolution.

3. Bei der *Anpassung an Kundenwünsche* wird eine Komponente durch eine bei der Entwicklung bereits vorgesehene Parametrisierung an spezifische Anforderungen des Nutzers angepasst. Nur bei der Komponentenentwicklung vorgesehene Anpassungen sind möglich.

Bei der Adaption von Diensten wird in [GR07, MSKC04] zwischen zwei Arten von Adaption unterschieden, der *parametrischen Adaption* und der *kompositionellen Adaption*. Die *parametrische Adaption* bezeichnet die Anpassung von Variablen eines Dienstes, die sein Verhalten bestimmten. Bei der Dienstentwicklung müssen dazu bestimmte Anpassungen antizipiert worden sein, z. B. die Anpassung der Lautstärke bei einem Musikdienst. Diese Art der Adaption spielt insbesondere in kontextsensitiven Anwendungen eine Rolle. Die *kompositionelle Adaption* findet auf der Architekturebene statt. Die Struktur der Dienstkomposition wird verändert, um sie an den aktuellen Zustand der Umgebung anzupassen. Dies entspricht der strukturellen Adaption aus Kapitel 4. Die Anpassungen, die sich bei der kompositionellen Adaption ergeben, sind nicht zuvor antizipiert, sondern ergeben sich aus den aktuellen Bedingungen der Umgebung.

5.3 Teilprozess der semantischen Adaptierung

In Kapitel 4 wurde die strukturelle Adaption beschrieben, deren Aufgabe es ist, die Komposition der eHome-Dienste an dynamische Veränderungen in der eHome-Umgebung im laufenden Betrieb anzupassen. Die semantische Adaption adressiert hingegen das Problem der Heterogenität in eHome-Systemen. Die Zielsetzung dabei ist es, die Komposition von Diensten zu ermöglichen und Inkompatibilitäten zu überwinden.

Der Adaptionsansatz basiert auf einer semantischen Beschreibung der Dienste und ihrer angebotenen und benötigten Funktionalitäten. Die semantische Beschreibung wird zur Laufzeit ausgewertet und für die automatische Komposition und Adaption genutzt. Die Grundlagen dazu wurden in den Arbeiten [Pie09] und [Hof09] gelegt.

Der *semantische Teilprozess*, der die Adaption unterstützt, ist in Abbildung 5.7 dargestellt. Er besteht aus den vier Phasen *Semantikdefinition*, *Abbildung*, *Matching* und *Adaptierung*. In der ersten Phase erfolgt zunächst die Definition einer

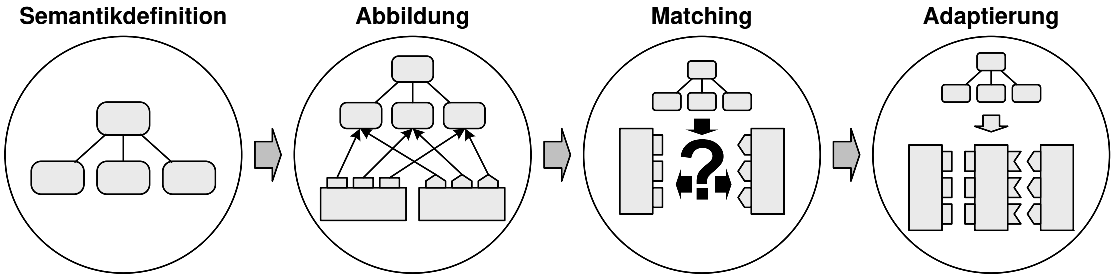


Abbildung 5.7: Semantischer Teilprozess.

Ontologie domänenspezifischer Konzepte, die für die semantische Beschreibung benötigt werden. Im Rahmen dieser Arbeit werden darin Funktionalitäten typischer Basisdienste in eHomes beschrieben. Diese Konzepte werden in der zweiten Phase verwendet, um semantische Dienstbeschreibungen zu erstellen. Dabei werden die Operationen der Dienstschnittstellen auf Funktionalitäten der domänenspezifischen Ontologie abgebildet. In der dritten Phase erfolgt dann das Matching der Dienste, wobei auf die semantische Dienstspezifikation aus der vorherigen Phase zurückgegriffen wird. Abschließend wird in der vierten Phase die eigentliche Adaption durchgeführt, sofern inkompatible Dienste komponiert werden sollen. Dazu können auf Basis der semantischen Beschreibung automatisch Adapterkomponenten durch die Laufzeitumgebung generiert werden.

Der semantische Teilprozess ist in den erweiterten Entwicklungsprozess, der in Abschnitt 4.2 vorgestellt wurde, eingebettet. Abbildung 5.8 macht die Zusammenhänge deutlich.

Die erste Phase beschreibt die *Semantikdefinition* und ist erforderlich, damit eHome-Dienste überhaupt semantisch spezifiziert werden können. Die Semantikdefinition dient als übergreifender Standard und erfolgt somit auch vor dem eigentlichen Entwicklungsprozess, also der Entwicklung von Diensten und dem eHome-spezifischen SCD-Prozess. Im Rahmen dieser Arbeit wird die Semantik in einer *Ontologie* formalisiert.

Die *Abbildungsphase* ist mit der Dienstentwicklung verbunden, im Speziellen mit der Dienstspezifizierung. Dabei wird die semantische Beschreibung der Dienste erstellt. Die semantische Dienstbeschreibung erfolgt durch Abbildung von syntaktischen Elementen der durch den Dienst implementierten und verwendeten

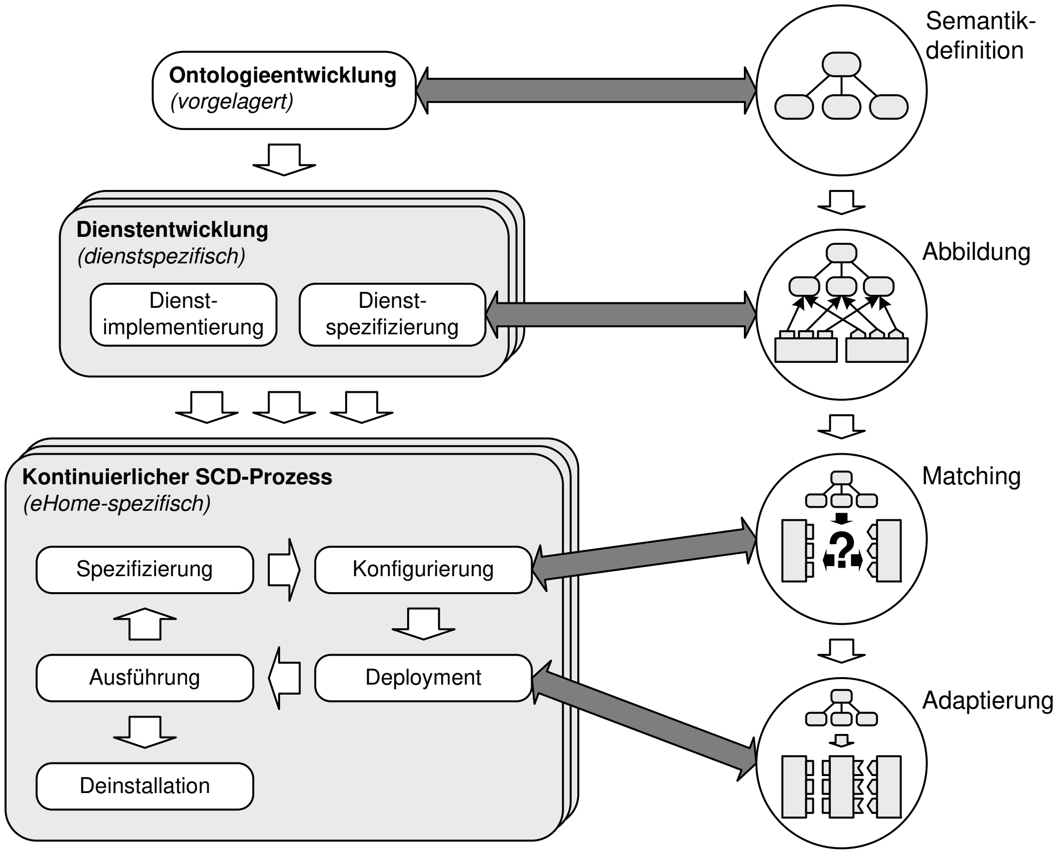


Abbildung 5.8: Einbettung des semantischen Teilprozesses in den Entwicklungsprozess für dynamische eHome-Systeme.

Schnittstellen auf Konzepte der Ontologie. Diese Abbildung wird später für die semantische Dienstkomposition und Adaptierung benötigt.

In der *Matchingphase* wird überprüft, ob zwei Dienste semantisch zueinander passen. Dieser Vorgang erfolgt in der Konfigurierungsphase des SCD-Prozesses. Dabei wird geprüft, ob die benötigte Funktionalität eines Dienstes mit der angebotenen Funktionalität eines anderen Dienstes semantisch korrespondiert. Dazu wird die semantische Dienstbeschreibung, d. h. die Abbildung der Dienstschnittstelle auf die Ontologie, analysiert. Davon abhängig können in der Konfiguration Bindungen zwischen den Diensten erstellt werden.

In der abschließenden Phase der *Adaptierung* können automatisch Adapter für die Komposition von Diensten generiert werden. Dieser Vorgang fällt in die De-

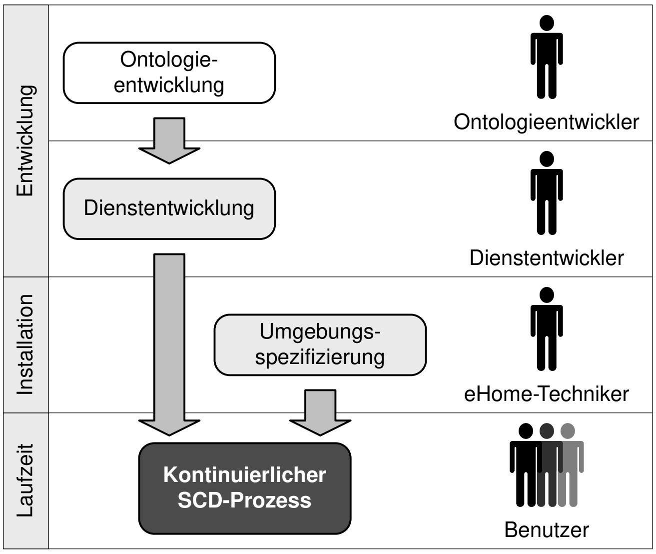


Abbildung 5.9: Akteure im Entwicklungsprozess.

ploymentphase des SCD-Prozesses, in der Dienstinstanzen und Bindungen entsprechend der Konfiguration erzeugt werden. Die Bindungen werden in der Matchingphase auf Basis semantisch korrespondierender Funktionalitäten erstellt. Sind die entsprechenden Schnittstellen der Dienste jedoch inkompatibel, so ist ein direktes Deployment nicht möglich. Daher werden Adapter erzeugt, die die Inkonsistenzen zwischen den Diensten überbrücken. Dieser Vorgang basiert ebenso wie das Matching auf den semantischen Dienstbeschreibungen.

Mit der Einführung des semantischen Teilprozesses kann nun der vollständige erweiterte Entwicklungsprozess dieser Arbeit betrachtet werden. Er umfasst die Konzepte sowohl der strukturellen Adaption als auch der semantischen Adaption. Bei der Umsetzung des Prozesses sind verschiedene Akteure vorgesehen, die in Abbildung 5.9 dargestellt sind.

Am Anfang muss eine domänenspezifische Ontologie entwickelt werden, die einen übergreifenden Standard für die semantische Dienstspezifikation bietet. Diese wird von einem *Ontologieentwickler* erstellt. Dabei kann es sich auch um eine Gruppe von Personen handeln, z. B. im Rahmen eine Kommission, die sich mit der Definition eines gemeinsamen Standards befasst. Eine solche Kommission kann dann auch die Wartung und Weiterentwicklung der Ontologie übernehmen. Die genaue Vorgehensweise dabei ist für die vorliegende Arbeit nicht entscheidend, sodass hier keine endgültigen Festlegungen in Bezug auf die Ontologieentwicklung getroffen werden.

Die Dienstentwicklung wird von einem Dienstentwickler vorgenommen. Dabei kann es sich um einen Anwendungsentwickler handeln, der Top-Level-Dienste entwickelt, oder aber auch um einen Entwickler von Basis- oder Treiberdiensten für bestimmte Geräte. Treiberdienste können z. B. auch von den Herstellern der entsprechenden Geräte selbst entwickelt werden. Die Dienstentwicklung umfasst sowohl die Implementierung als auch die Spezifizierung von Diensten. In bestimmten Fällen ist eine Dienstimplementierung bereits gegeben und es wird erst nachträglich eine passende Spezifikation erstellt, dann sind Implementierer und Spezifizierer unterschiedliche Personen.

Bei der Installation eines individuellen eHomes muss zunächst die Umgebungsspezifikation erstellt werden. Es ist davon auszugehen, dass viele Benutzer die Einrichtung des Systems nicht selbst übernehmen können oder wollen. Daher wird diese Aufgabe, einschließlich der Spezifizierung der eHome-Umgebung, in der Praxis im Allgemeinen an einen Techniker delegiert werden.

Während der Laufzeit können die Benutzer selbst mit dem eHome-System interagieren. Dazu dient ein entsprechendes Werkzeug, das in Abschnitt 6.3.3 genauer beschrieben wird. Wenn möglich und gewünscht werden Adaptionsschritte von der Laufzeitumgebung automatisch durchgeführt. Der Ansatz dieser Arbeit sieht aber vor, dass der Benutzer jederzeit die Möglichkeit der Interaktion hat, damit er die Konfiguration des Systems bei Bedarf manuell anpassen kann und so die Kontrolle über das eHome behält.

In den folgenden Abschnitten werden zunächst die vier Phasen des semantischen Teilprozesses im Einzelnen vorgestellt. Dabei wird auf die konzeptuellen Details des Ansatzes zur semantischen Adaption eingegangen.

5.4 Semantikdefinition

In diesem Abschnitt wird die Definition der semantischen Konzepte für eHome-Systeme beschrieben. Diese stellt die erste Phase des semantischen Teilprozesses dar (vgl. Abbildung 5.7) und wird für alle folgenden Phasen benötigt. Für

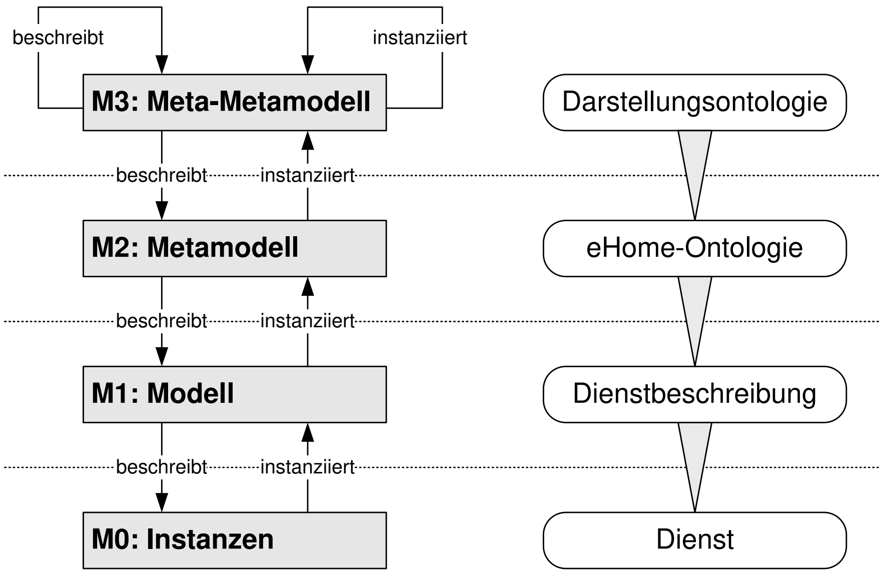


Abbildung 5.10: Vier Ebenen der Metamodellierung (nach [Völ04]) und deren Anwendung auf die Modellierung von eHome-Diensten.

die *Semantikdefinition* wird, wie bereits in Abschnitt 5.2.1 erläutert, ein geeignetes Modell benötigt. Dazu eignen sich Ontologien, welche die Konzepte und Zusammenhänge beschreiben, die in der Domäne *eHome* auftreten. Damit die semantische Beschreibung ausdrucksstark genug ist, um für die Komposition und die Adaption verwendet zu werden, muss sie über ein einfaches Vokabular, wie es bei NORBISRATH verwendet wurde, hinausgehen.

Bei der Erstellung einer geeigneten Ontologie für eHome-Dienste muss zunächst eine zu verwendende Darstellungsontologie (siehe Abschnitt 5.2.1) definiert werden. Diese dient als Schema für die Ontologie, welche die Konzepte definiert, die für die spätere semantische Beschreibung von eHome-Diensten verwendet wird. Nach der Festlegung der Darstellungsontologie wird eine eHome-Ontologie entwickelt, die dann zur semantischen Beschreibung verwendet werden kann. Die verschiedenen Modellierungsebenen sind in Abbildung 5.10 dargestellt. Auf der untersten Ebene befinden sich die *Instanzen*, in diesem Fall die eHome-Dienste. Diese werden auf der darüber liegenden Ebene als *Modell* beschrieben. Teil dieser Dienstbeschreibung ist auch die semantische Spezifikation des Dienstes. Für die Dienstbeschreibung wird die übergreifende eHome-Ontologie verwendet, die hier als *Metamodell* dient. Auf der obersten Ebene befindet sich die Darstellungsontologie, die als *Meta-Metamodell* fungiert. Das Meta-Metamodell kann zur Beschreibung verschiedener eHome-Ontologien verwendet werden. Eine eHome-Ontologie wiederum kann für die Beschreibung vieler verschiedener Dienste eingesetzt werden. Die Dienstbeschreibung kann dann durch unterschiedliche Implementierungen realisiert werden.

5.4.1 Ontologiesprachen

Die Festlegung der Darstellungsontologie entscheidet darüber, wie die eHome-Ontologie aussieht und welche Möglichkeiten zur Formalisierung des Domänenwissens sie bietet. Die eHome-Ontologie ihrerseits muss es erlauben, die syntaktischen Elemente der Dienstschnittstellen abzubilden. Daher muss sie sich an den Konzepten orientieren, die bei der Kommunikation zwischen Diensten Verwendung finden, die also in den Schnittstellen der Dienste verwendet werden.

Zur Spezifikation der Dienstbeschreibung, der eHome-Ontologie und der Darstellungsontologie müssen entsprechende Sprachen verwendet werden, in denen die jeweiligen Konzepte formalisiert werden können. Natürlich ist es möglich, ganz neue Sprachen für die spezifischen Einsatzgebiete zu entwickeln. Diese Vorgehensweise wurde z. B. auch im ConDes-Projekt am Lehrstuhl für Informatik 3 gewählt, in dem unter anderem eine Sprache speziell zur regelbasierten Wissensdefinition im konzeptuellen Gebäudeentwurf entwickelt wurde [Ret05, KR05, KR06a, KR06b]. Die dabei eingesetzte graphbasierte Ontologie wurde ebenfalls speziell für dieses Projekt entworfen.

In der vorliegenden Arbeit soll jedoch auf bestehende und in der Anwendung erprobte Sprachen zurückgegriffen werden, um den Entwicklungsaufwand zu verringern und die bereits verfügbaren Werkzeuge nutzen zu können. Die Dienstbeschreibungen werden zwar in einem eigens für diesen Zweck erstellten XML-Format gespeichert, für die eHome-Ontologie und die Darstellungsontologie kann aber auf bestehende Sprachen zurückgegriffen werden. Unter den verbreiteten Ontologiesprachen lassen sich zwei Paradigmen unterscheiden, zum einen die *Beschreibungslogiken* (engl. *Description Logics, DL*) und zum anderen die sogenannten *framebasierten Sprachen*.

Beschreibungslogiken dienen der formalen Beschreibung von Domänenwissen und verwenden dazu eine entscheidbare Teilmenge der Logik erster Ordnung,

d. h. der Prädikatenlogik. In Beschreibungslogiken wird der Fokus auf Eigenschaften gelegt, d. h. es geht vornehmlich darum, festzulegen, aufgrund welcher Eigenschaften eine Entität als Mitglied einer bestimmten Klasse anzusehen ist [WNR+06]. Unter den Sprachen, die auf Beschreibungslogiken basieren, ist die *Web Ontology Language (OWL)*[1], die vom *World Wide Web Consortium (W3C)* als Ontologiesprache empfohlen wird, am stärksten verbreitet. OWL ist eine XML-basierte Sprache und stellt eine Erweiterung der Ausdrucksstärke des *Resource Description Frameworks (RDF)*[2] dar, welches zur Speicherung von Meta-Informationen über Web-Ressourcen dient. Für OWL sind drei Sprachvarianten definiert, die sich in ihrer Ausdrucksstärke unterscheiden: *OWL Lite*, *OWL DL* und *OWL Full*. Eine wichtige Eigenschaft von OWL ist die *Open World Assumption*. Demgemäß werden alle Aussagen als wahr interpretiert, die nicht explizit durch eine entsprechende Deklaration ausgeschlossen werden. Diese Eigenschaft ist nachteilhaft, wenn das Domänenwissen genauen Vorgaben entsprechen soll. Dann ist es meist sinnvoller eine Sprache zu wählen, die der *Closed World Assumtion* genügt.

Framebasierte Sprachen basieren auf der Frame-Theorie von MINSKY [Min75], die sogenannte *Frames* als Konstrukt zur Erklärung der Wissensrepräsentation im menschlichen Gehirn einführt. Framebasierte Sprachen haben die Eigenschaft der *Closed World Assumption*, d. h. alles, was nicht explizit als wahr bekannt ist, wird als falsch interpretiert. Im Gegensatz zu Sprachen auf Basis von Beschreibungslogiken wird bei der Verwendung von Frames für jede Klasse im Voraus bereits festgelegt, welche Eigenschaften mit dieser Klasse einhergehen [WNR+06]. Dieses Vorgehen ist eng verwandt mit der objektorientierten Analyse und Modellierung in der Softwareentwicklung [BME+07, Mey97]. Am bekanntesten unter den framebasierten Sprachen ist *F-Logic* [KL89]. Als Sprachelemente stehen in F-Logic Klassen, Attribute und Relationen zur Verfügung, sowie Regeln, mit denen diese Elemente beschrieben werden können. Des Weiteren gibt es die Möglichkeit, Anfragen zu stellen, die dann auf Basis des formalisierten Wissens ausgewertet und beantwortet werden.

Für die Modellierung der eHome-Ontologie wird in dieser Arbeit F-Logic verwendet. Vorteilhaft an F-Logic ist zum einen die Closed World Assumption, aufgrund derer nur explizit formalisiertes Wissen als wahr betrachtet wird. Dies ist

[1] http://www.w3.org/TR/owl-semantics/
[2] http://www.w3.org/TR/rdf-concepts/

für den Ansatz dieser Arbeit nützlich, da das zu formalisierende Wissen auf die Domäne *eHomes* und zunächst auch auf die Basisfunktionalitäten von eHome-Diensten beschränkt sein soll. Die Ontologie soll nicht zur Zuordnung von Klassen dienen, sondern zur Modellierung der Funktionalitäten gemäß einem vorgegebenen Schema. Dies ist notwendig, damit eine Adaption möglich wird. Dazu ist ein framebasierter Ansatz besser geeignet. Zum anderen bietet F-Logic aber auch den Vorteil, dass die Modellierung und Verwendung ähnlich zur objektorientierten Programmierung ist. Dies ermöglicht eine leichtere Einarbeitung und die intuitive Verwendung durch Entwickler von eHome-Diensten.

5.4.2 Struktur der Darstellungsontologie

Betrachtet man die Schnittstellen, die üblicherweise den Funktionalitäten von Basisdiensten in eHomes zugrunde liegen, so stellt man fest, dass diese in den meisten Fällen eine einfache Struktur aufweisen. Die Geräte in der eHome-Umgebung lassen sich häufig über Tasten, Schalter oder Regler steuern, wodurch sich der Betriebszustand eines Geräts verändern lässt. Entsprechend gestalten sich auch die Schnittstellen der Basisdienste, die für die Steuerung der Geräte zuständig sind. Die Schnittstellenoperationen haben nur einen oder auch gar keinen Parameter und dienen dazu, ein Attribut zu lesen oder zu schreiben oder ein Signal zu geben. Diese Struktur der betrachteten Schnittstellen dient als Grundlage für die Darstellungsontologie, die in dieser Arbeit verwendet wird.

Abbildung 5.11 veranschaulicht schematisch die Elemente der Darstellungsontologie. Die Darstellungsontologie definiert bestimmte Klassen, die beim Erstellen der eHome-Ontologie zur Modellierung verwendet werden können. Die grundlegenden Klassen sind Capability, Attribute und Access.

Eine *Funktionalität*, die ein Dienst anbietet oder benötigt, wird in der Ontologie als Capability modelliert. Ein Lampentreiberdienst kann z. B. die Funktionalität Beleuchtung anbieten, die dann anderen Diensten zur Verfügung steht. Eine Funktionalität entspricht auf syntaktischer Ebene einer Schnittstelle. Die Zustände, die mit einer Funktionalität assoziiert sind und die von außen sichtbar sind, werden als *Attribute* modelliert. In der Ontologie dient dazu die Klasse Attribute. Attribute haben einen bestimmten Typ und können nicht nur Zustände des Geräts selbst betreffen, sondern auch Zustände der Umgebung des Geräts. Dies

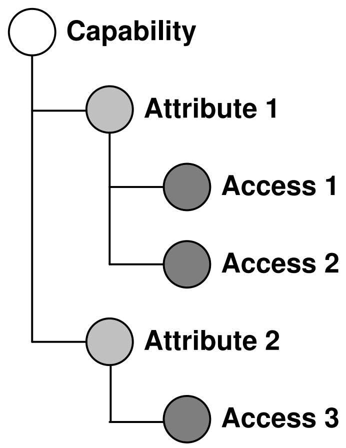


Abbildung 5.11: Elemente der Darstellungsontologie.

ist z. B. bei Sensoren relevant, die eine Eigenschaft der Umgebung messen. Attribute sind wiederum mit sogenannten *Zugriffen* ausgestattet, die einen Zugriff auf das Attribut erlauben. Die üblichen Zugriffe sind das Lesen und Schreiben von Attributen, je nach Art des Attributs sind aber auch andere Zugriffe möglich. In der Ontologie wird ein Zugriff als Access repräsentiert. Die Zugriffe entsprechen im Wesentlichen den Operationen der Dienstschnittstellen. Sie stellen die Interaktionsmöglichkeiten dar, die bei der Kommunikation zwischen Diensten zur Verfügung stehen. Zu einer Funktionalität gehören darüber hinaus noch Ausnahmefälle. Ein Ausnahmefall kann beim Zugriff auf ein Attribut auftreten und wird in der Darstellungsontologie als Fault modelliert.

Im Folgenden wird die Struktur der Darstellungsontologie formal definiert. Diese Festlegungen präzisieren die informellen Beschreibungen und dienen dazu, die Darstellungsontologie mit einem Werkzeug spezifizieren zu können.

Definition 5.1 (Capability)
Eine ***Capability*** *ist ein Tupel* $capability = (identifier, attributes, capabilities)$ *mit*

- $identifier$ *ist der Bezeichner*
- $attributes$ *ist eine Liste von Attributes, die leer sein kann*
- $capabilities$ *ist eine Liste von Capabilities, die leer sein kann.*

Definition 5.1 beschreibt das Konzept der Capability. Jede Capability wird durch einen eindeutigen Bezeichner identifiziert. Darüber hinaus können beliebig viele Attributes zu einer Capability gehören. Außerdem kann eine Capability weitere Capabilities enthalten, was einer Verfeinerung im Sinne einer *Realisierungsbeziehung* entspricht. Damit wird ausgedrückt, dass eine Capability auf einer abstrakteren Ebene durch eine oder mehrere Capabilities auf einer konkreteren Ebene realisiert werden kann. In umgekehrter Richtung betrachtet können Capabilities andere Capabilities realisieren.

Es ist keine Vererbungsbeziehung unter den Capabilities vorgesehen. Der Grund liegt darin, dass es sehr schwierig ist, eine allgemeingültige Klassifikation der Konzepte einer Anwendungsdomäne festzulegen, auch wenn sie, wie in diesem Fall, auf einen speziellen Bereich eingeschränkt ist. Auch für die Funktionalitäten im eHome gibt es, wie in den meisten anderen Domänen auch, sehr viele Möglichkeiten, eine Klassifizierung vorzunehmen. Je nach Anwendungsfeld und Kontext des Erstellers können unterschiedliche Klassifikationen entstehen und auch sinnvoll sein. Für die eHome-Ontologie ist jedoch die Verwendung als Standard wichtig, da ja gerade eine Adaption unter heterogenen Diensten erreicht werden soll. Durch eine Vererbungsbeziehung und die damit erforderliche Klassifikation würde das Erstellen einer eHome-Ontologie weiter erschwert werden, ohne dass wesentliche Vorteile für den Ansatz dieser Arbeit erzielt würden.

Definition 5.2 (Attribute)
*Ein **Attribute** ist ein Tupel* $attribute = (identifier, type, accesses)$ *und*

- $identifier$ *ist ein Bezeichner*
- $type$ *ist ein Typbezeichner, der dem Typ des Attributes entspricht*
- $accesses$ *eine nichtleere Liste von Accesses.*

Attributes sind in Definition 5.2 spezifiziert. Auch das Attribute besitzt einen eindeutigen Bezeichner. Da Attributes typisiert sind, haben sie einen Typbezeichner. Der Typ eines Attributes ist insbesondere für die spätere Adaption relevant. Schließlich hat jedes Attribute noch einen oder mehrere Accesses, die den Zugriff auf das Attribute ermöglichen. Es muss für jedes Attribute mindestens einen Access geben, sonst ist das Attribute als Schnittstellenattribut nicht relevant und braucht dann auch nicht als Element der Ontologie modelliert zu werden.

Definition 5.3 (Access)
*Ein **Access** ist ein Tupel* $access = (type, result, slots, faults)$ *mit*

- $type$ *ist ein Typbezeichner*
- $result$ *ist ein Typbezeichner*
- $slots$ *ist eine Liste von Slots*
- $faults$ *ist eine Menge von Fehlerfallbeschreibungen.*

Ein Access ist wie in Definition 5.3 spezifiziert. Ein Typbezeichner *type* gibt an, um welche Art von Access es sich handelt. Übliche Typen sind z. B. read und write. Ein weiterer Typbezeichner *result* gibt den Typ des Rückgabewertes für diesen Access an. Es folgt eine Liste von Slots. Diese repräsentieren die Eingabeparameter für den Access. Schließlich gibt es noch eine Menge von Faults, die mögliche Ausnahmen definiert, die beim Verwenden des Access auftreten können.

Dieser Definition entsprechend gibt es viele Freiheiten bei der Spezifikation von Accesses. Dies ist sinnvoll, da so auch sehr unterschiedliche Schnittstellenoperationen prinzipiell abgebildet werden können. In der Praxis werden Accesses aber in den allermeisten Fällen bestimmten Mustern entsprechen. Ein Access vom Typ *read* wird häufig einen Rückgabewert haben, jedoch keine Eingabeparameter, d. h. keine Slots. Umgekehrt wird ein Access vom Typ *write* häufig genau einen Eingabeparameter haben, jedoch keinen Rückgabewert.

Definition 5.4 (Slot)
*Ein **Slot** ist ein Tupel* $slot = (identifier, type)$ *mit*

- $identifier$ *ist ein Bezeichner*
- $type$ *ist ein Typbezeichner.*

Definition 5.4 beschreibt die Slots, die den Accesses zugeordnet sind. Jeder Slot hat einen Bezeichner, sodass er von anderen unterschieden werden kann, und einen Typ. Mit diesen Definitionen sind alle Elemente der Darstellungsontologie festgelegt, sodass diese mit Hilfe eines Werkzeugs spezifiziert und zum Erstellen der eHome-Ontologie eingesetzt werden kann.

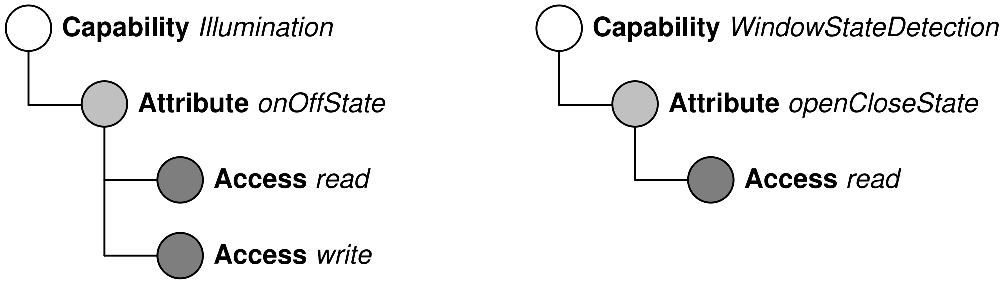


Abbildung 5.12: Beispiele zur Verwendung der Darstellungsontologie.

Neben den Hauptstrukturelementen, die oben beschrieben wurden, gibt es in der Darstellungsontologie noch das Konzept der Fragments. Fragments bieten die Möglichkeit, die Darstellungsontologie um Elemente zu erweitern, die nicht in die oben beschriebenen Strukturen integriert werden können. Sie haben keine innere Struktur und sind daher auf der Ebene eines Vokabulars angesiedelt.

Bisher wurde das Tag als einziges Fragment in die Darstellungsontologie eingeführt, welches die semantische Auszeichnung von Schnittstellenoperationen ermöglicht und bereits in verschiedenen Arbeiten innerhalb des eHome-Projekts eingesetzt wurde [Fro08, Kul09]. Dadurch kann z. B. die Laufzeitumgebung um verschiedene semantische Funktionalitäten erweitert werden, wie etwa durch das in Abschnitt 4.4.4 vorgestellte Session-Konzept.

5.4.3 Anwendung zur Ontologiemodellierung

Zwei Beispiele, wie die Darstellungsontologie verwendet wird, sind in Abbildung 5.12 dargestellt. Auf der linken Seite ist die Spezifikation der Funktionalität Illumination dargestellt. Diese hat ein Attribut onOffState, welches angibt, ob die Beleuchtung momentan ein- oder ausgeschaltet ist. Dieses Attribut kann über Zugriffe von außen abgefragt oder geändert werden. Dazu stehen die Accesses read und write zur Verfügung. Auf der rechten Seite der Abbildung ist eine weitere Funktionalität WindowStateDetection zur Zustandsabfrage eines Fensters zu sehen. Diese Funktionalität besitzt das Attribut openCloseState, das angibt, ob das Fenster momentan geöffnet oder geschlossen ist. Für openCloseState gibt es nur einen Access read, daher kann der Zustand des Fensters über diese Funktionalität nur abgefragt aber nicht geändert werden.

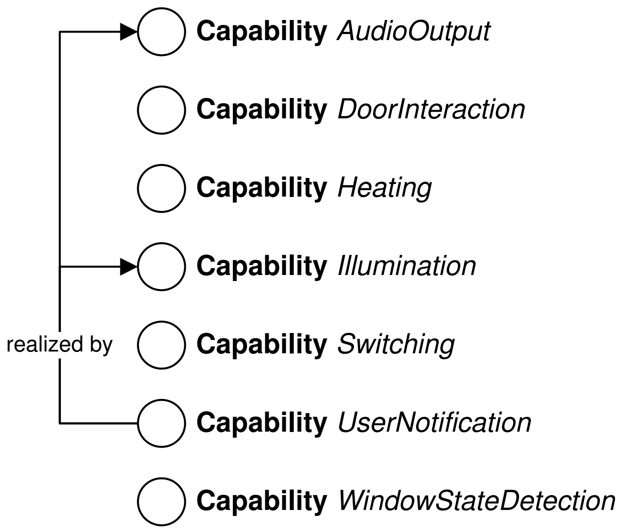


Abbildung 5.13: Beispielontologie mit abstrakter Funktionalität.

Abbildung 5.13 zeigt ein Beispiel mit einer abstrakten Funktionalität. Im dargestellten Ausschnitt der Ontologie sind verschiedene Capabilities zu sehen. Davon beschreibt die Capability UserNotification eine abstrakte Funktionalität mit der Semantik, den Benutzer zu benachrichtigen. Eine solche Funktionalität kann z. B. von einem Klingeldienst benötigt werden, wie im Beispiel in Abbildung 5.4. In der Ontologie aus Abbildung 5.13 ist spezifiziert, dass UserNotification durch die konkreten Capabilities AudioOutput und Illumination realisiert werden kann. Damit wird festgelegt, dass eine Benachrichtigung des Benutzers mittels Audioausgabe oder mittels Beleuchtung umgesetzt werden kann. Der Benutzer kann also durch ein Tonsignal oder auch durch ein visuelles Signal aufmerksam gemacht werden.

Für die Modellierung der eHome-Ontologie wird in dieser Arbeit *Protégé* verwendet. Protégé ist ein Werkzeug für wissensbasierte Systeme und Ontologien, das seit 1987 an der Universität Stanford entwickelt wird [GMF$^+$02]. Es unterstützt sowohl Beschreibungslogiken durch einen OWL-Editor als auch Frames/F-Logic durch einen entsprechenden Frames-Editor. Da in dieser Arbeit, wie in Abschnitt 5.4.1 erläutert, F-Logic zur Modellierung verwendet wird, kommt hier nur der Frames-Editor von Protégé zum Einsatz. Abbildung 5.14 zeigt einen Screenshot von Protégé und der darin modellierten eHome-Ontologie. Auf der linken Seite der Abbildung ist der *Class Browser* dargestellt, der in einer hierar-

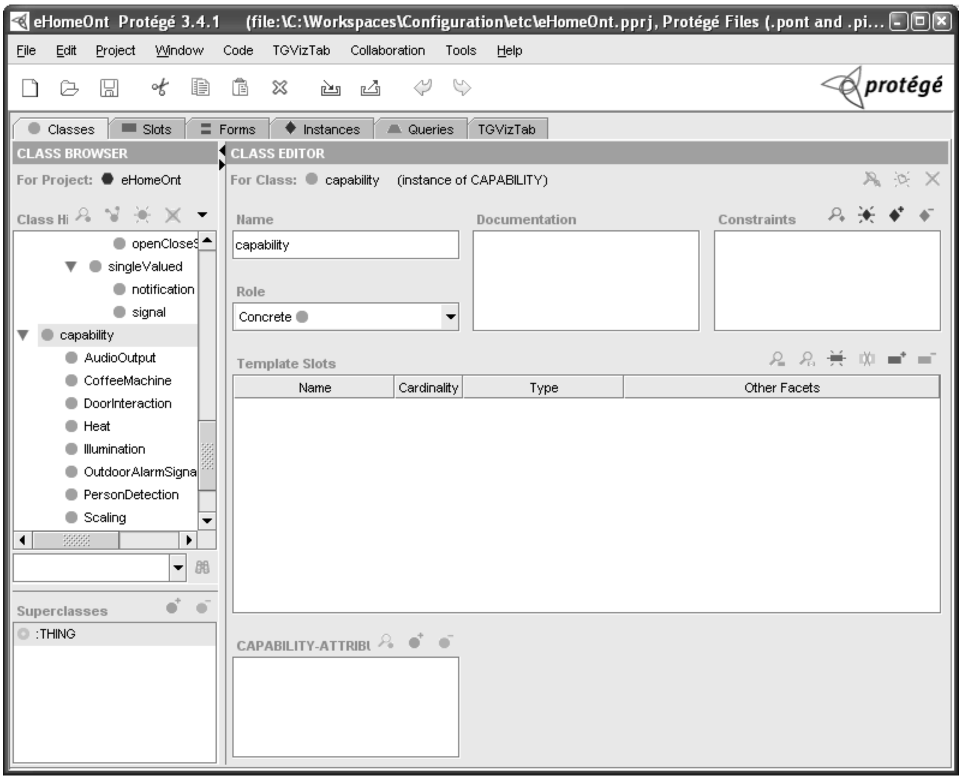


Abbildung 5.14: Protégé-Editor mit eHome-Ontologie.

chischen Darstellung die Klassen der Ontologie anzeigt. Im Ausschnitt, der in der Abbildung zu sehen ist, sind die Capabilities dargestellt. Die eHome-Ontologie kann in Protégé als Projekt-Datei abgespeichert werden und dann in den Werkzeugen zur Dienstspezifikation verwendet werden, um die Schnittstellen der Dienste auf die Ontologie abzubilden.

Zur genaueren Beschreibung des Zusammenhangs zwischen Capabilities wird in dieser Arbeit *Jess* eingesetzt. Jess ist eine Regelmaschine und Skriptumgebung für Java und steht für ***J**ava **E**xpert **S**ystem **S**hell* [Fri03]. Jess verarbeitet Fakten und Regeln, die in einer der Programmiersprache Prolog ähnlichen Notation spezifiziert werden. Die Regeln bestehen aus einer linken Regelseite, die gegen Fakten geprüft wird, und einer rechten Regelseite, die bei einem Match aus der linken Regelseite gefolgert wird. Zur Regelauswertung verwendet Jess eine erweiterte Version des Rete-Algorithmus [For82]. Jess lässt sich über ein Plug-In in Protégé einbinden und bietet ebenso wie Protégé eine Java-Programmierschnitt-

```
1 (defrule MAIN::userNotify
2     (initial-fact)
3     =>
4     (defclass write (is-a ACCESS)
5         (slot Illumination.onOffState (type boolean)))
6     (for (bind ?i 0) (< ?i 3) (++ ?i)
7         (slot-set write Illumination.onOffState TRUE)
8         (call ?*jessconnect* wait 3000)
9         (slot-set write Illumination.onOffState FALSE)
10        (call ?*jessconnect* wait 3000)))
```

Listing 5.1: Jess-Regel zur Spezifikation einer Realisierungsbeziehung

stelle an, die eine einfache Verwendung bei der Entwicklung der Werkzeuge dieser Arbeit ermöglicht. Die Realisierungsbeziehungen zwischen Capabilities der Ontologie können so mit Hilfe von Jess-Regeln genauer spezifiziert werden.

Das Listing 5.1 zeigt ein Beispiel, wie die Spezifikation einer Realisierungsbeziehung in Jess aussieht. Die angegebene Regel bezieht sich auf die Capability UserNotification und die zu dessen Realisierung verwendete Capability Illumination. Zur Benachrichtigung des Benutzers soll die Beleuchtung dreimal ein- und wieder ausgeschaltet werden, wobei immer mit einem Zeitabstand von jeweils drei Sekunden umgeschaltet werden soll. Da die Regel nicht automatisch, sondern explizit durch den später generierten Adapter ausgelöst werden soll, steht auf der linken Regelseite in Zeile 2 der Fakt `(initial-fact)`, der immer gegeben ist und damit als wahr ausgewertet wird. Auf der rechten Regelseite, ab Zeile 4, wird für den Access `write` ein Slot `Illumination.onOffState` vom Typ `boolean` angelegt. Dieser wird verwendet, um einen Instanzwert für das Attribute onOffState der Capability Illumination zu speichern. Ab Zeile 6 folgt dann eine Schleife, die dreimal wiederholt wird und in der der Wert des onOffState abwechselnd auf `TRUE` (Zeile 7) und wieder auf `FALSE` (Zeile 9) gesetzt wird. Dazwischen findet jeweils eine Pause von drei Sekunden statt, was in Zeile 8 und 10 durch den Aufruf einer externen Java-Methode `wait` realisiert wird.

Mit Hilfe der Jess-Regeln kann bereits innerhalb der Ontologie ein konkreter Zusammenhang zwischen abstrakten Funktionalitäten und konkreten Funktionalitäten spezifiziert werden. Dazu werden die in der Ontologie definierten

Konzepte verwendet. Die Definition der Zusammenhänge innerhalb der Ontologie bietet Vorteile gegenüber der Entwicklung von integrierenden Diensten, die ebenfalls zwischen verschiedenen Abstraktionsebenen vermitteln können. Zum einen kann direkt auf der semantischen Ebene gearbeitet werden. Dies ist weniger aufwendig, als einen vollständigen eHome-Dienst mit allen dazu benötigten Schnittstellen und Klassen zu implementieren. Zum anderen kann bereits der Ontologieentwickler die Zusammenhänge zwischen den von ihm definierten Konzepten festlegen. Der Ontologieentwickler hat einen besseren Überblick über die gesamte Ontologie und kann daher diese Zusammenhänge besser erkennen. Ein Dienstentwickler befasst sich hingehen jeweils mit einem spezifischen Ausschnitt der Ontologie, der die von ihm implementierte Funktionalität betrifft.

5.5 Semantische Abbildung

Die semantische Abbildung ist die zweite Phase des semantischen Teilprozesses (vgl. Abbildung 5.7). Die Elemente der Darstellungsontologie wurden im vorherigen Abschnitt bereits formal definiert. Die auf Basis der Darstellungsontologie spezifizierte eHome-Ontologie entspricht dem *Wertebereich* der semantischen Abbildung. Damit eine Komposition heterogener Dienste möglich ist, müssen alle Dienste auf Basis derselben eHome-Ontologie spezifiziert werden. Nur dann ist ein semantischer Vergleich möglich.

Wenn Dienste auf Basis verschiedener Ontologien spezifiziert werden, so ist vor der gemeinsamen Verwendung dieser Dienste eine Integration der verwendeten Ontologien notwendig. Zur Unterstützung von Ontologieintegration gibt es verschiedene Ansätze und Werkzeuge, z. B. [HRK08b]. Ein solcher Ansatz wird im Rahmen dieser Arbeit nicht näher betrachtet, in zukünftigen Arbeiten ist jedoch eine entsprechende Erweiterung in diesem Bereich denkbar.

Der *Definitionsbereich* der semantischen Abbildung setzt sich aus den Schnittstellen der eHome-Dienste zusammen, denn diese sollen semantisch beschrieben werden. Damit dies möglich ist, muss zunächst, so wie auch für die eHome-Ontologie, ein Modell zur Repräsentation gefunden werden. Diese Modell kann dann als Grundlage der Abbildung dienen.

5.5.1 Dienstschnittstellen

Ein Modell für die Erfassung der Schnittstellen von eHome-Diensten wird im Folgenden formal definiert.

Definition 5.5 (Schnittstelle)
*Eine **Schnittstelle** ist ein Tupel* $interface = (identifier, operations)$ *und*

- $identifier$ *ist ein Bezeichner*
- $operations$ *ist eine nichtleere Menge von Operationen.*

Definition 5.5 spezifiziert den Aufbau einer Schnittstelle. Jede Schnittstelle hat einen eindeutigen Bezeichner und eine Menge von Operationen. Es muss mindestens eine Operation geben, sonst wäre die Schnittstelle leer und damit keine Abbildung möglich. Mit dieser Definition werden nicht alle Bestandteile einer Java-Schnittstelle erfasst. So kann in Java eine Schnittstelle z. B. andere Schnittstellen erweitern oder in anderen Schnittstellen enthalten sein. Da diese Informationen für die semantische Abbildung nicht benötigt werden, müssen sie hier auch nicht weiter berücksichtigt werden.

Definition 5.6 (Operation)
*Eine **Operation** ist ein Tupel* $operation = (identifier, type, parameters, exceptions)$ *mit*

- $identifier$ *ist ein Bezeichner, der Name der Operation*
- $type$ *ist ein Typbezeichner und entspricht dem Rückgabetyp der Operation*
- $parameters$ *ist eine Liste von Parametern, die Liste kann leer sein*
- $exceptions$ *ist eine Menge von Typbezeichnern für mögliche Ausnahmen, die Menge kann leer sein.*

Der Aufbau einer Schnittstellenoperation ist in Definition 5.6 festgelegt. Jede Operation hat einen Namen und einen Rückgabetyp, sowie eine Liste von Parametern und eine Menge von möglichen Ausnahmen. Entsprechend der Elemente einer Operation o_1 sei $identifier(o_1)$ der Name, $type(o_1)$ der Rückgabetyp, $parameters(o_1)$ die Parameterliste und $exceptions(o_1)$ die Menge der Ausnahmen von o_1. Diese Notation gilt auch für die folgenden Definitionen.

Definition 5.7 (Parameter)
*Ein **Parameter** ist ein Tupel* $parameter = (identifier, type, position)$ *und*

- $identifier$ *ist ein Bezeichner, der Name des Parameters*
- $type$ *ist ein Typbezeichner, der dem Parametertyp entspricht*
- $position$ *ist ein positiver, ganzzahliger Wert mit* $position \geq 0$*, welcher der Position des Parameters in der Parameterliste entspricht.*

Der Aufbau der Parameterliste ist in Definition 5.7 festgelegt. Neben einem Bezeichner und Typ hat jeder Parameter eine bestimmte Position innerhalb der Signatur einer Operation. Der Ausdruck $parameters(o_1, 2)$ liefert den zweiten Parameter einer Operation o_1.

Die semantische Abbildung stellt eine Relation zwischen einer Schnittstelle und einer passenden Capability der eHome-Ontologie her. Dabei werden die Operationen der Schnittstelle auf Accesses bestimmter Attributes dieser Capability abgebildet. Im einfachsten Fall ist eine direkte Zuordnung möglich, sodass von der Operation der Schnittstelle auf eine zugehörige Capability geschlossen werden kann und umgekehrt. Das ist möglich, wenn die Schnittstellenoperation in Übereinstimmung mit der eHome-Ontologie definiert wurde und somit keine strukturellen Inkompatibilitäten vorliegen. Allerdings wird dieser Fall in der Praxis nur selten auftreten. Damit Schnittstellen in den anderen Fällen dennoch abgebildet werden können, müssen zusätzliche Informationen spezifiziert werden.

In Abbildung 5.15 ist ein Klassendiagramm dargestellt, das einen Ausschnitt zeigt, durch den die Abbildung von Schnittstellen auf die Ontologie modelliert wird. Der Ausschnitt ist auf die wesentlichen Elemente vereinfacht, um eine übersichtlichere Darstellung zu erreichen.

In der unteren Hälfte sind die Elemente der Dienstschnittstellen zu finden. Diese werden durch die Klassen ServiceInterface, Return, Operation, Parameter und Fault repräsentiert, die Spezialisierungen von ServiceInterfaceElement sind. Eine Operation kann beliebig viele Parameters in einer bestimmten Reihenfolge haben und muss immer genau einen Rückgabewert Return haben. Weiterhin können Operationen beliebig viele Ausnahmen Faults auslösen. Diese Elemente sind die Quelle der Abbildung auf die Ontologie.

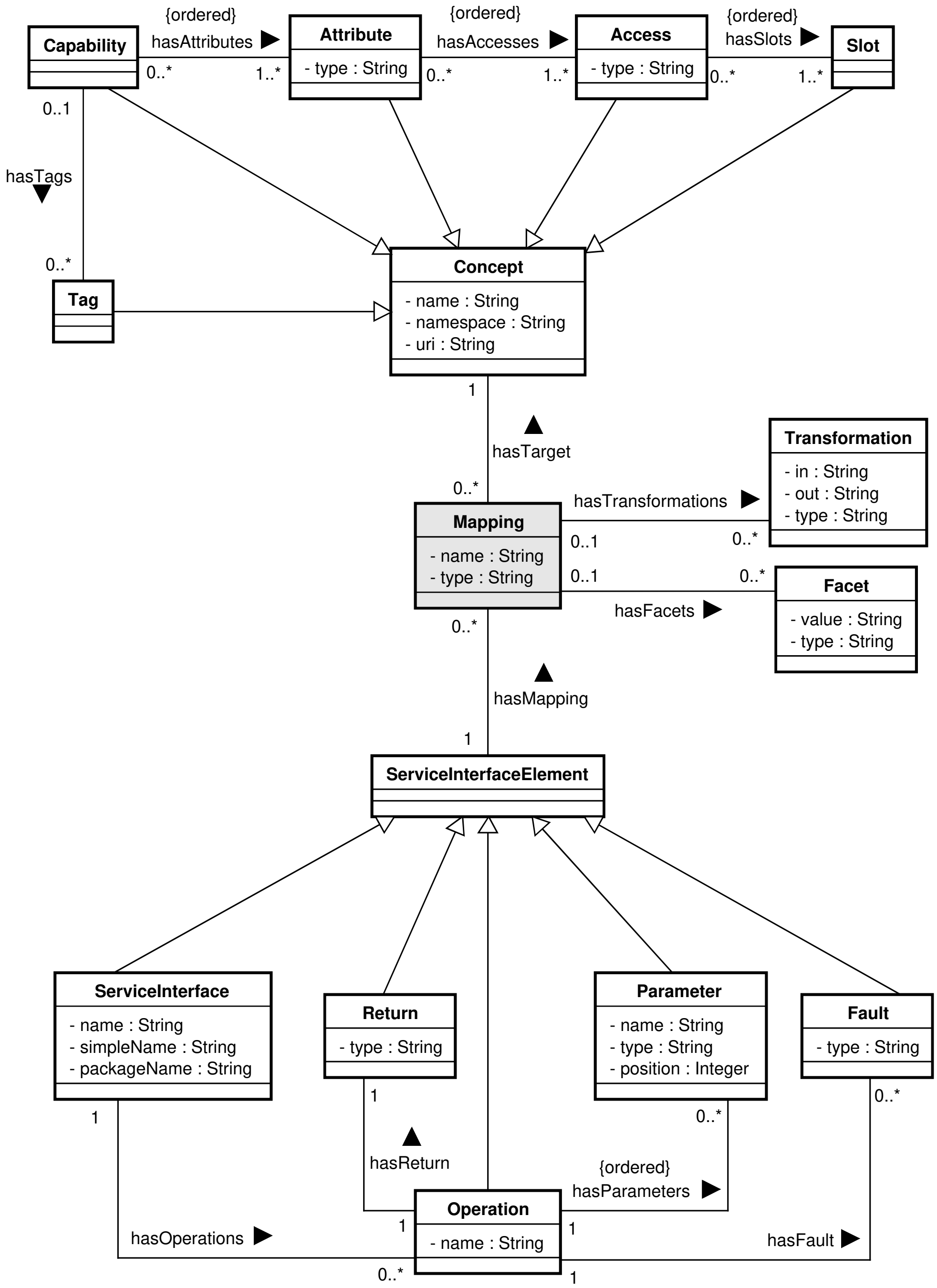


Abbildung 5.15: Modellierung der semantischen Abbildung.

Die Abbildung auf die Ontologiekonzepte wird durch Kanten mit je einem Zwischenknoten Mapping repräsentiert. Die Klasse Mapping speichert zusätzliche Informationen der semantischen Abbildung, die für das semantische Matching und die spätere Adaptergenerierung verwendet werden. Dazu gehören auch die Klasse Transformation, die für Transformationen bei Inkompatibilitäten eingesetzt werden kann, und die Klasse Facet, die benutzt wird, um parameterlose Operationen auf bestimmte Attributwerte abzubilden.

Die Klasse Concept dient als Oberklasse der spezifischen Konzepte der eHome-Ontologie, die durch die Klassen Tag, Capability, Attribute, Access und Slot repräsentiert werden. Wie in Abschnitt 5.4.2 beschrieben, hat jede Capability ein oder mehrere Attributes, diese haben wiederum ein oder mehrere Accesses, welche ein oder mehrere Slots haben. Außerdem können Capabilities mit beliebig vielen Tags ausgezeichnet werden. Welche Schnittstellenelemente welchen Konzepten der Ontologie zuzuordnen sind, ist durch diese Modellierung nicht festgelegt.

Der *Service-Editor* (siehe Abschnitt 6.3.1), der das Anlegen der Abbildung ermöglicht, muss sicherstellen, dass nur sinnvolle Abbildungen spezifiziert werden können. Dennoch bleiben Freiheitsgrade bestehen, die auch gewünscht und notwendig sind, um eine hinreichende Flexibilität bei der semantischen Dienstspezifikation zu haben. Ein eHome-Dienst kann auch mehrere Schnittstellen benutzen oder implementieren. Diese können semantisch beschrieben werden, indem für alle Operationen dieser Schnittstellen Abbildungen spezifiziert werden.

5.5.2 Inkompatibilitäten

Im Rahmen der semantischen Abbildung werden vier Inkompatibilitäten unterschieden, die durch die Abbildung zwischen Schnittstellen und der eHome-Ontologie berücksichtigt werden. Diese betreffen Datentypen, Einheiten sowie die Zuordnung von Parametern zu den Attributes der Ontologie.

1. **Inkompatible Datentypen:** Die Attributes der Capabilities sind typisiert. Daher kann das Problem auftreten, dass ein Parametertyp einer Schnittstellenoperation nicht mit dem Typ des semantisch zugehörigen Attribute der Ontologie übereinstimmt. Solche inkompatiblen Typen können auch in einfachen Fällen vorkommen. Strukturell ist die Zuordnung häufig einfach,

wenn es sich um simple Schreib- und Lesezugriffe handelt. Die Schreiboperation hat dann einen Parameter, der einem Attribute entspricht und die Leseoperation hat einen Rückgabewert, der dem Attribute entspricht. Doch auch in diesem Fall können die Typen inkompatibel sein. Wird z. B. ein Geldbetrag in Euro modelliert, so kann dieser einerseits als Ganzzahl, die dem Wert in Eurocent entspricht, gespeichert werden. Andererseits ist aber auch eine Repräsentation als Gleitkommazahl möglich. Zwischen diesen Varianten muss dann übersetzt werden und es sind entsprechende Angaben zur Abbildung erforderlich. Eine solche Transformation ist ggfs. nur mit inakzeptablem Informationsverlust möglich. Unter Umständen kann eine Transformation auch nur in einer Richtung vorgenommen werden, oder es ist gar keine Transformation möglich.

2. **Inkompatible Einheiten:** Wird durch ein Attribute eine Größe in einer bestimmten Einheit gespeichert, so muss dies durch die Abbildung berücksichtigt werden. Eine physikalische Größe, wie etwa die Temperatur, kann in verschiedenen Einheiten repräsentiert werden, in diesem Fall z. B. Grad Celsius und Grad Fahrenheit. Häufig ist die Einheit einer Größe nicht Bestandteil des Typs, der zur Repräsentation verwendet wird. Daher müssen Einheiten gesondert betrachtet werden. Liegen unterschiedliche Einheiten vor, so ist eine entsprechende Umrechnung erforderlich. Dies ist in den meisten Fällen ohne Informationsverlust möglich, sodass die Transformation keinen Einschränkungen, wie etwa bei den oben beschriebenen Typen unterliegt.

3. **Split:** Operationen, die den Wert eines Attributes setzen, verwenden dazu nicht zwangsläufig einen Parameter. Die Wertzuweisung kann auch implizit geschehen. Wenn das Attribute einen Zustand modelliert, z. B. ob ein Gerät eingeschaltet oder ausgeschaltet ist, so kann der Zustandsübergang durch zwei Operationen `activate()` und `deactivate()` realisiert sein. Somit gibt es mehrere Operationen, die aber demselben Attribute zuzuordnen sind. Für diskrete Typen, wie Wahrheitswerte oder Aufzählungstypen, ist dies einfach realisierbar, in anderen Fällen müssen Intervalle definiert werden, die dann einzeln abgebildet werden können.

4. **Join:** Auch der umgekehrte Fall kann auftreten, dass nämlich eine Schnittstellenoperation mehrere Attributes gleichzeitig betrifft. Für einen Treiber-

dienst, der eine dimmbare Lampe steuert, kann es eine Operation geben, die sowohl das Ein- und Ausschalten als auch das Festlegen des Dimmwertes der Lampe ermöglicht. Wenn diese Eigenschaften durch mehrere Attributes in der Ontologie modelliert werden, so muss dies bei der Abbildung berücksichtigt werden.

Die obige Auflistung der Inkompatibilitäten ist keineswegs vollständig. Die genannten Fälle reichen jedoch für den in dieser Arbeit beschriebenen Adaptionsansatz aus. Die am häufigsten bei Basisdiensten auftretenden Situationen können damit abgedeckt werden, sodass eine Adaption möglich ist.

5.5.3 Transformationen

In diesem Abschnitt werden die bei der Abbildung von Schnittstellen auf die Ontologie durchzuführenden Transformationen formal beschrieben.

Definition 5.8 (Typdomäne)
Für einen Datentyp mit dem Bezeichner typeIdentifier entspricht seine ***Typdomäne*** $dom(typeIdentifier)$ *dem Wertebereich dieses Typs. Einen Spezialfall bildet der Typ* $void$ *mit* $dom(void) = void$.

Definition 5.9 (Transformation)
Eine ***Transformation*** *ist eine Funktion*

$$transform : dom(t_1) \times \cdots \times dom(t_n) \to dom(t'_1) \times \cdots \times dom(t'_m)$$

mit $n, m \in \mathbb{N}^+$. *Diese Funktion transformiert eine Liste von Werten in eine andere Liste von Werten, wobei die Listen eine unterschiedliche Länge aufweisen können. Einen Sonderfall bildet das Element* $dom(void)$. *Es darf nur als einziges Element eines Kreuzprodukts auftreten und entspricht einer leeren Liste. Konkret dient diese Funktion dazu, die Transformation von Eingabeparametern oder Rückgabewerten zu repräsentieren. Im ersten Fall entspricht das einer Transformation einer Liste von Eingabeparametern in eine andere solche Liste, die sich sowohl in der Länge als auch in den vorkommenden Typen von der ursprünglichen Liste unterscheiden kann. In dieser Anwendung wird die Funktion auch* ***Eingabetransformation*** *bzw.* $transform_{in}$ *genannt.*

Bezieht sich die Transformation auf Rückgabewerte, ist eine Einschränkung der Funktion notwendig. Für eine ***Ausgabetransformation****, die mit* $transform_{out}$ *notiert wird, gilt zusätzlich die Bedingung* $n = m = 1$.

Die oben definierte Transformation dient dazu, die Elemente der Dienstschnittstellen auf Elemente der Ontologie abzubilden. Dazu müssen im Allgemeinen auch Unterschiede berücksichtigt werden, da die Modellierung einer Schnittstelle nicht unmittelbar der Modellierung der zugehörigen Accesses in der Ontologie entsprechen muss.

Dienste und deren Schnittstellen können unabhängig von der eHome-Ontologie entwickelt werden. Das ist insbesondere wichtig für bereits bestehende Dienstimplementierungen oder unabhängig von der Nutzung im eHome entwickelte Software. Die semantische Abbildung wird dann erst im Nachhinein vorgenommen. Wie die Abbildung der Dienstfunktionalitäten auf die Ontologie formal modelliert wird, ist in Definition 5.10 beschrieben.

Definition 5.10 (Funktionalitätsabbildung)
Gegeben seien ein Interface i *sowie eine Capability* c. *Eine* ***Funktionalitätsabbildung*** *ist ein bipartiter Graph*

$$capabilityMapping = (O \dot{\cup} A, E, l)$$

mit

- $O = operations(i)$
- $A = \bigcup_{s \in attributes(c)} accesses(s)$, *im Folgenden auch* $allAccesses(c)$
- $E \subseteq (O \times A) \cup (A \times O)$ *den Kanten des Graphen*
- $l : E \to transform_{in} \times transform_{out}$ *einer Auszeichnungsfunktion für die Kanten des Graphen.*

Dabei gilt zusätzlich, dass die Transformationen $l((u, v) \in E)$ *kompatibel mit den Knoten* u *und* v *sein müssen. Ein Tupel* $(transformation_{in}, transformation_{out})$ *aus Eingabe- und Ausgabetransformation ist dann kompatibel zu einer Operation* op *und einem Access* acc, *wenn gilt:*

1. $(transformation_{in}, transformation_{out}) = l(op, acc)$ *und*

- ➯ $transformation_{in} : \times_{p \in parameters(op)} dom(p) \rightarrow \times_{s \in slots(acc)} dom(s)$
- ➯ $transformation_{out} : dom(return(op)) \rightarrow dom(result(acc))$

2. $(transformation_{in}, transformation_{out}) = l(acc, op)$ *und*
   - ➯ $transformation_{in} : \times_{s \in slots(acc)} dom(s) \rightarrow \times_{p \in parameters(op)} dom(p)$
   - ➯ $transformation_{out} : dom(result(acc)) \rightarrow dom(return(op))$

Die so definierte Kompatibilität von Transformationen bezüglich der Knoten, zwischen denen sie zulässig sind, bezieht sich also nur auf die Übereinstimmung der Struktur von Ein- und Ausgabe.

Die Funktionalitätsabbildung assoziiert die Zuordnungen zwischen Operationen und Accesses der Ontologie mit Transformationen. Diese Transformationen ermöglichen es, die in Abschnitt 5.5.2 beschriebenen Inkompatibilitäten zu berücksichtigen und zu spezifizieren, welche Anpassungsschritte notwendig sind. Die Details der verwendeten Transformationen sind in Definition 5.10 zunächst nicht festgelegt. Auch komplexere Transformationen sind möglich, da jedoch, wie bereits oben erläutert, die Ebene der Basisdienste adressiert wird, kann von einfachen Operationen zum Setzen und Auslesen von Attributen ausgegangen werden. Im Folgenden sind Beispiele aufgeführt, die zeigen, wie die beschriebenen Inkompatibilitäten aus Abschnitt 5.5.2 transformiert werden können.

1. **Inkompatible Datentypen:** Wird eine Operation op auf einen Slot abgebildet, dessen zugehöriges Attribute einen anderen Datentyp hat als ein Parameter von op, so ist die Zuordnung möglich, wenn eine Transformation

$$transform_{in} : dom(parameters(op, i)) \rightarrow dom(slots(acc, j))$$

angegeben werden kann, die den Eingabeparameter von op in den Datentyp konvertieren kann, der dem Slot des Accesses entspricht. Ist diese Abbildung nur mit einem Informationsverlust möglich, so muss dieser für die entsprechende Dienstfunktionalität vernachlässigbar sein. Anderenfalls kann es zu unerwünschtem Verhalten zur Laufzeit kommen, insbesondere wenn bei der späteren Dienstkomposition eine mehrstufige Transformation zwischen Diensten nötig ist und in jedem Schritt Informationsverluste auftreten können. Falls auch eine Transformation

$$transform'_{in} : dom(slots(acc, i)) \to dom(parameters(op, j))$$

existiert, die diese Eigenschaften erfüllt, so ist die Abbildung in beiden Richtungen möglich.

2. Inkompatible Einheiten: Falls sich die Einheiten von Parametern einer Operation op und einem zugehörigen Attribute unterscheiden, so kann ebenfalls eine Zuordnung vorgenommen werden, wenn eine Transformation

$$transform_{in} : dom(parameters(op, i)) \to dom(slots(acc, j))$$

angegeben werden kann. Falls eine verlustfreie Rücktransformation

$$transform'_{in} : dom(slots(acc, i)) \to dom(parameters(op, j))$$

definiert werden kann, so ist die Zuordnung in beiden Richtungen verwendbar.

3. Split: Ein *Split* kann ebenfalls mittels einer Transformation angegeben werden. Falls z. B. eine Operation close auf einen Access acc eines Attributes closed vom Typ Boolean abgebildet wird, so kann die folgende konstante Funktion angegeben werden:

$$\begin{aligned} transform_{in} \quad : \quad & dom(void) \to dom(slots(acc, i)) \\ & void \mapsto false. \end{aligned}$$

Für die umgekehrte Richtung kann eine partielle Funktion direkt aus der ersten abgeleitet werden. In diesem Beispiel ergibt sich

$$transform'_{in}(x) = \begin{cases} void & \text{falls } x = false \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Der volle Umfang eines gesplitteten Attributes kann nur dann abgebildet werden, wenn gilt

$$\bigcup_{(acc,op) \in E} urbild(transformation_{in}(l(acc, op))) = dom(type(attr))$$

mit $acc = access(attr)$. Eine Eingabetransformation, die auf den ausgehenden Kanten definiert ist, muss also die gesamte Typdomäne des Attribute abdecken. Es ist hier keine disjunkte Vereinigung erforderlich. So kann es z. B. für das Attribute closed vom Typ Boolean für den Wert false mehrere passende Zuordnungen, etwa die Operationen `close()` und `setClosed(boolean state)`, geben. Beide Zuordnungen sind dann als äquivalent zu betrachten.

4. Join: Der Fall eines *Join* unterscheidet sich strukturell deutlich von den obigen Fällen und bedarf einer gesonderten Behandlung. Beim *Join* wird eine Operation zu mehreren Accesses einer Capability in Beziehung gesetzt. Dazu wird eine zusammengesetzte Abbildung benötigt, die aus mehreren Transformationen der oben beschriebenen Arten besteht. Für diese Transformationen kann eine partielle Ordnung gegeben sein, welche die Reihenfolge der Ausführung bestimmt. Darüber hinaus ist eine sinnvolle Rücktransformation für eine solche Abbildung nicht möglich. Aufgrund der Komplexität wird keine formale Betrachtung dieses Falls vorgenommen.

Bei der Funktionalitätsabbildung sind weitere Aspekte zu beachten. Als einer dieser Aspekte sind exemplarisch *Ausnahmen* zu nennen, die bei der Ausführung einer Operation auftreten können. Falls eine solche Ausnahme inhaltlichen Charakter hat und keine technischen Aspekte betrifft, so kann sie ebenfalls auf die Ontologie abgebildet werden. Dazu ist es möglich, Fehlerfälle für Accesses in der Ontologie zu definieren und dann Ausnahmen darauf abzubilden. Es ist davon auszugehen, dass die Ausnahmefälle, die die Dienstfunktionalität betreffen, für Basisdienste eine sehr einfache Struktur haben und keine angehängten Daten mitführen. Nur unter diesen Voraussetzungen ist eine Berücksichtigung in der semantischen Beschreibung im Rahmen dieser Arbeit möglich.

In der Darstellungsontologie sind zusätzlich zu den oben genannten Konzepte Fragments, wie z. B. die Tags, vorgesehen. Diese wurden in Abschnitt 5.4.2 bereits erwähnt. Da es sich hierbei um einfache Konstrukte handelt, ist für das Verständnis der semantischen Abbildung keine Formalisierung erforderlich. Ebenso sind für die Verwendung dieser Konzepte keine Transformationen nötig.

In Abbildung 5.16 ist ein Beispiel für die Funktionalitätsabbildung dargestellt. Auf der linken Seite ist eine Dienstschnittstelle `LampDriver` zu sehen, die zur Steuerung einer Lampe dient und drei Operationen umfasst. Auf der rechten Seite ist die Capability Illumination aus der eHome-Ontologie dargestellt. Diese besitzt das

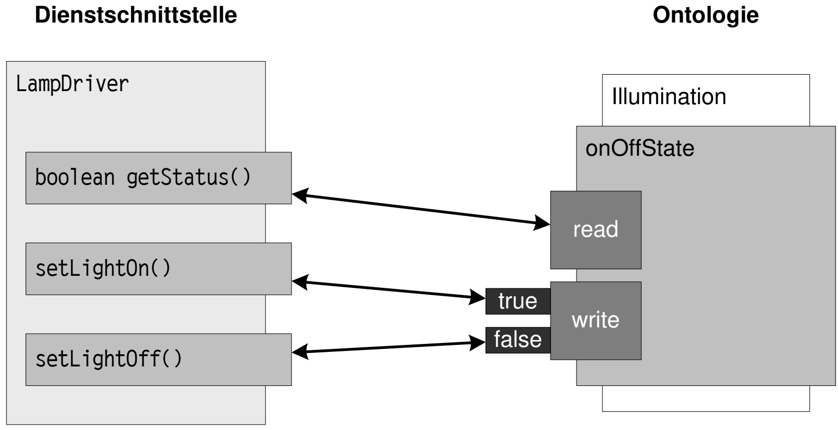


Abbildung 5.16: Beispiel einer Funktionalitätsabbildung.

Attribute onOffState, das beschreibt, ob die Beleuchtung momentan aktiviert oder deaktiviert ist. Das Attribute ist vom Typ Boolean und hat die Accesses read und write zum Lesen und Schreiben von onOffState.

Die erste Operation der Schnittstelle ist `getStatus` und gibt zurück, ob die Lampe momentan ein- oder ausgeschaltet ist. Diese Operation ist daher auf den Access read von onOffState abgebildet. Die beiden verbleibenden Operationen `setLightOn` und `setLightOff` benötigen keine Parameter und haben die Aufgabe, die Lampe ein- bzw. auszuschalten. Beide Operationen sind auf den Access write von onOffState abgebildet, da sie den Beleuchtungszustand verändern. Dabei ist `setLightOn` auf den Slot true abgebildet, da die Lampe eingeschaltet wird, und `setLightOff` auf den Slot false, da die Lampe mit dieser Operation ausgeschaltet wird. Hier liegt also eine Split-Transformation vor, da der Access write von onOffState durch verschiedene Operationen abgedeckt wird. So werden durch die im Beispiel gezeigte Funktionalitätsabbildung alle Operationen der Schnittstelle des Lampentreibers semantisch beschrieben.

5.6 Semantisches Matching

Das semantische Matching stellt die dritte Phase des semantischen Teilprozesses aus Abbildung 5.7 dar. Mit Hilfe der semantischen Abbildung aus der zweiten Phase können die Dienstspezifikationen um semantische Informationen erweitert werden. Diese Informationen können dann zum Auffinden passender Dienste, dem sogenannten *Matching*, verwendet werden. Im Unterschied zum syntaktischen Matching, das auf Basis der Java-Schnittstellen der Dienstimplementierungen arbeitet, wird beim semantischen Matching die Dienstbeschreibung betrachtet. Mittels der semantischen Spezifikation können passende Dienste gefunden werden, die die gesuchte Funktionalität anbieten, auch wenn keine übereinstimmenden Schnittstellen vorliegen. Eine sogenannte *Semantic Registry* verwaltet die semantischen Dienstbeschreibungen und erlaubt es, semantische Matches auf Basis dieser Beschreibungen zu ermitteln.

Das Matching von Diensten kann auf unterschiedlichen Ebenen erfolgen. Es werden drei Ebenen unterschieden, die bereits in Abbildung 5.4 angedeutet wurden. Sie unterscheiden sich im Grad der Übereinstimmung.

1. Die *Schnittstelle* stimmt überein.
2. Die *konkrete Funktionalität* stimmt überein.
3. Die *abstrakte Funktionalität* stimmt überein.

Erste Ebene: Auf der ersten Ebene werden Matches auf der Ebene der *Schnittstelle* betrachtet. Bei dieser Art von Matching werden keinerlei semantische Informationen über den Dienst benötigt, der gesamte Vorgang spielt sich auf der syntaktischen Ebene ab. Ein Match liegt nur vor, wenn von den Diensten eine gemeinsame Schnittstelle verwendet wird. Daher ist in diesem Fall auch keine spätere Adaption erforderlich. Das schnittstellenbasierte Matching ist zu unflexibel und liefert daher in vielen Situationen keine Matches. Deshalb wurde die semantische Betrachtungsebene eingeführt, die ein flexibleres Matching ermöglicht.

Zweite Ebene: Auf der zweiten Ebene werden Matchings auf Basis der Semantik, die den Dienstschnittstellen zugeordnet ist, durchgeführt. Damit ein Match vorliegt, muss auf dieser Ebene die *konkrete Funktionalität*, die den Schnittstellen zugewiesen ist, übereinstimmen. Die konkrete Funktionalität ist dabei die einem Dienst direkt zugewiesene Funktionalität. Ein Match liegt daher nur dann vor, wenn ein Dienst gefunden wird, der genau diese Funktionalität anbietet. Hier wird somit eine höhere Abstraktionsebene betrachtet als auf der ersten Ebene. Nicht mehr die syntaktische Schnittstelle ist von Bedeutung, wenn es darum geht, Dienste zu binden, sondern die diesen Schnittstellen zugeordnete Semantik. Entscheidend ist, welche Funktionalität ein Dienst über seine Schnittstelle anbietet oder benötigt. Diese Betrachtungsebene erlaubt eine höhere Flexibilität bei der Dienstkomposition, da diese nicht mehr auf Dienste mit gemeinsamen Schnittstellen eingeschränkt ist. Damit semantisch kompatible Dienste zur Laufzeit tatsächlich interagieren können, ist im Allgemeinen eine Adaption erforderlich. Diese wird jedoch erst im letzten Schritt des semantischen Teilprozesses durchgeführt, der in Abschnitt 5.7 beschrieben wird.

Dritte Ebene: Auf der dritten Ebene werden ebenfalls semantische Matches betrachtet. Hier können jedoch auch Matches gefunden werden, die auf *abstrakten Funktionalitäten* basieren. Eine abstrakte Funktionalität ist eine Funktionalität, die nicht unmittelbar von anderen Diensten angeboten wird, zu deren Realisierung aber konkrete Funktionalitäten verwendet werden können. Um solche Matches zu erkennen, werden Realisierungsbeziehungen, die zwischen den Konzepten der Ontologie definiert sind, herangezogen. Diese Beziehungen werden bei der Ontologieentwicklung angelegt und definieren Zusammenhänge zwischen Capabilities der Ontologie (vgl. Abschnitt 5.4). Durch Auswertung dieser Beziehungen können Dienste gebunden werden, auch wenn eine benötigte Funktionalität nicht direkt verfügbar ist.

Top-Level-Dienste können zunächst abstrakte Funktionalitäten benötigen, die nicht direkt von anderen Diensten zur Verfügung gestellt werden. Diese werden dann auf konkrete Funktionalitäten abgebildet, sodass entsprechende Dienste gebunden werden können. Ebenso ist es möglich, dass zwar eine konkrete Funktionalität benötigt wird, diese jedoch von keinem der verfügbaren Dienste angeboten wird. Dann ist es ggfs. möglich, eine alternative Funktionalität zu binden, die zur Substitution geeignet ist. Die entsprechenden Beziehungen zwischen den Funktionalitäten müssen zuvor in der Ontologie spezifiziert worden sein, damit solche Matches erkannt werden können.

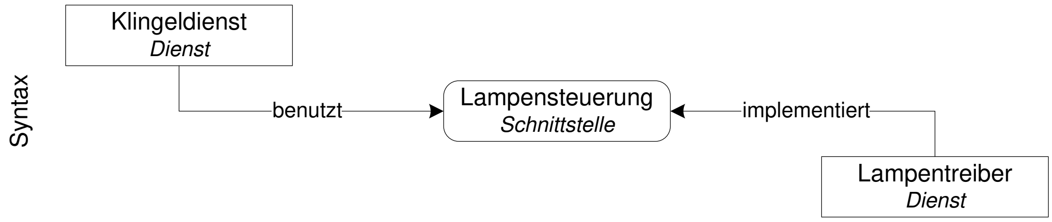
Mit dieser Ebene wird ein weiterer Schritt zur Erhöhung der Flexibilität bei der Dienstkomposition gemacht. Dabei ist zu beachten, dass die erreichbare Flexibilität klare Grenzen aufweist und nicht alle denkbaren Verknüpfungen von Funktionalitäten auch sinnvoll im eHome anzuwenden sind.

> Für die Realisierung der Dienstbindungen ist hier ebenfalls eine Adaption erforderlich. Die Adaption ist in vielen Fällen noch komplexer als auf der zweiten Ebene, da aufgrund der höheren Abstraktionsebene der semantischen Betrachtung die syntaktischen Schnittstellen noch größere Differenzen aufweisen können. Bei der Adaption müssen darüber hinaus auch die zu den Realisierungsbeziehungen innerhalb der eHome-Ontologie spezifizierten Jess-Regeln angewandt werden, damit eine Dienstkomposition erfolgen kann.

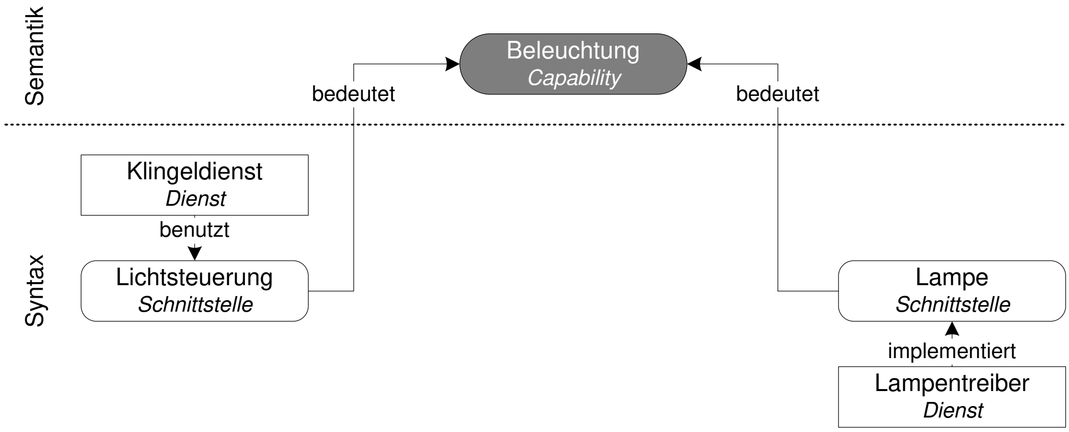
Die drei oben beschriebenen Ebenen korrespondieren mit den Ebenen der semantischen Modellierung in Abbildung 5.4. Wie die verschiedenen Ebenen zum Matching der eHome-Dienste verwendet werden, ist in Abbildung 5.17 an einem Beispiel verdeutlicht. Ganz oben, in Abbildung 5.17(a), ist das syntaktische Matching der ersten Ebene dargestellt. Wie beschrieben werden hier ausschließlich die Schnittstellen der Dienste betrachtet, die semantische Betrachtung bleibt zunächst außen vor. Im dargestellten Fall wird ein Klingeldienst verwendet, der die Aufgabe hat, den Benutzer zu benachrichtigen, wenn die Klingel am Eingang des eHomes gedrückt wird. Dazu benutzt er eine Schnittstelle Lampensteuerung, sodass durch ein Blinken der Beleuchtung im eHome die Bewohner aufmerksam gemacht werden sollen. Die Schnittstelle Lampensteuerung wird durch einen Dienst Lampentreiber implementiert, der eine konkrete Lampe ansteuert. Hier ist eine direkte Bindung der Dienste auf Basis der gemeinsamen Schnittstelle möglich.

Die zweite Ebene des semantischen Matchings ist in Abbildung 5.17(b) dargestellt. Hier wird dasselbe Beispiel betrachtet, jedoch unterscheiden sich hier die verwendeten Schnittstellen. Der Klingeldienst benutzt nun die Schnittstelle Lichtsteuerung, während der Lampentreiber die Schnittstelle Lampe implementiert. Da die beiden Schnittstellen nicht übereinstimmen, was hier durch den unterschiedlichen Namen symbolisiert wird, ist keine direkte Bindung zwischen den Diensten möglich. Die Schnittstellen sind jedoch im Zuge der semantischen Abbildung derselben Capability Beleuchtung zugeordnet worden. Daraus ergibt sich eine semantische Übereinstimmung. Ein solches Match kann mit Hilfe eines Adapters umgesetzt werden.

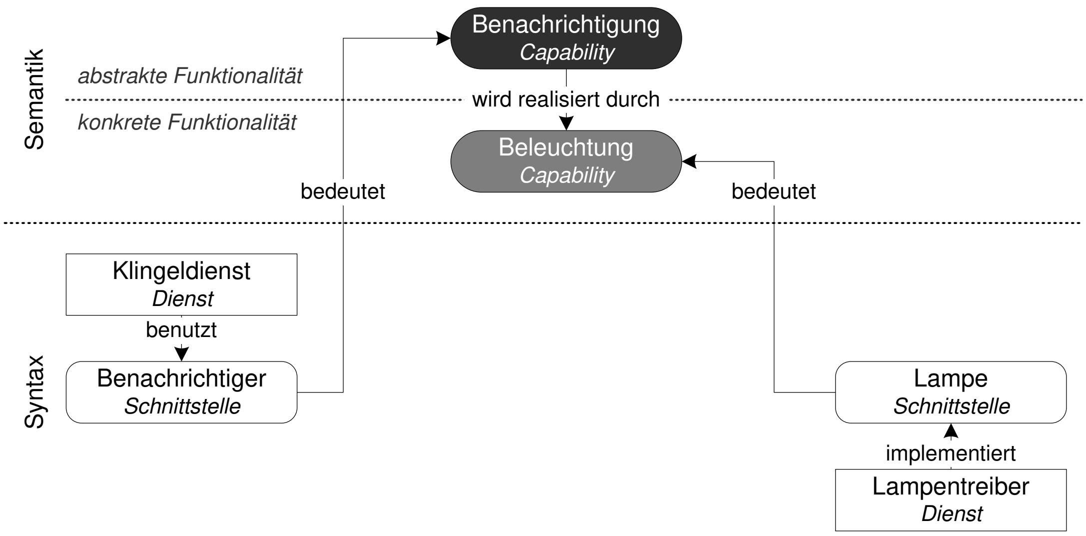
In Abbildung 5.17(c) wird ein semantisches Matching über eine abstrakte Funktionalität dargestellt. Dieses Beispiel betrifft somit die dritte Matchingebene. Auf



(a) Schnittstelle stimmt überein.



(b) Konkrete Funktionalität stimmt überein.



(c) Abstrakte Funktionalität stimmt überein.

Abbildung 5.17: Verschiedene Ebenen des Matchings.

der Seite des Dienstes Lampentreiber hat sich nichts geändert. Der Lampentreiber implementiert nach wie vor die Schnittstelle Lampe, welche wiederum der Capability Beleuchtung zugeordnet ist. Auf der Seite des Klingeldienstes stellt sich die Situation im Beispiel nun anders dar. Der Klingeldienst ist nun so implementiert, dass er eine Schnittstelle Benachrichtiger benutzt. Diese Schnittstelle ist wiederum einer Capability Benachrichtigung zugeordnet. Der Klingeldienst wurde hier so implementiert, dass er eine abstrakte Funktionalität – die Benachrichtigung – verwendet. Aus Sicht des Klingeldienstes ist also lediglich festgelegt, dass eine Benachrichtigung der Bewohner des eHomes erfolgen soll. Wie diese Benachrichtigung konkret umgesetzt wird, ist nicht eindeutig festgelegt. Es können je nach Spezifikation der Ontologie verschiedene Möglichkeiten bestehen, die abstrakte Funktionalität umzusetzen.

In Abbildung 5.17(c) ist in der Ontologie spezifiziert, dass die Capability Benachrichtigung durch die Capability Beleuchtung realisiert werden kann. Dies bedeutet, dass zur Benachrichtigung der Bewohner die Beleuchtung verwendet werden kann, um ein visuelles Signal zu geben. Die Details dieser Realisierung müssen in der Ontologie spezifiziert sein, damit ein solches Match möglich ist und die Dienste verbunden werden können. Für die Capability Benachrichtigung wurde in Abschnitt 5.4.3 in Listing 5.1 bereits eine Jess-Regel für die Realisierung der Benachrichtigung durch die Beleuchtung angegeben. Darin wurde festgelegt, dass der Benutzer durch dreimaliges Blinken der Beleuchtung aufmerksam gemacht wird. In diesem Fall ist ebenso wie auf Ebene zwei ein Adapter erforderlich, der die Interaktion zwischen den Diensten entsprechend der Spezifikation auf Ontologieebene ermöglicht. Die Adaptergenerierung wird im Rahmen der vierten Phase, der Adaptierung, in Abschnitt 5.7 beschrieben.

Die Qualität der erkannten semantischen Matches hängt entscheidend von der Modellierung der Semantik in der Ontologie sowie der semantischen Abbildung in der Dienstbeschreibung ab. Eine exakte semantische Übereinstimmung ist in der Praxis nie gegeben, es sind immer Differenzen in der Bedeutung einer Funktionalität möglich, auch wenn sie äußerst gering sind. Das Problem liegt darin, festzulegen, wann und in welchem Kontext solche Differenzen vernachlässigbar sind und wann nicht. Diese Problematik wird jedoch nicht erst durch die Betrachtung der semantischen Ebene eingeführt. Auch bei ausschließlicher Betrachtung der Dienstschnittstellen sind semantische Unterschiede zwischen Diensten mit passenden Schnittstellen vorhanden, nur dass sie in diesem Fall nicht explizit betrachtet werden. Wenn aus einer Übereinstimmung von Schnittstellen automatisch eine semantische Übereinstimmung abgeleitet wird, so wird das Problem nur ausgeblendet, nicht jedoch gelöst. Die semantische Betrachtung offenbart also ggfs. nur eine Problematik, die bereits zuvor gegeben war. Dennoch ist bei einer unsauberen semantischen Dienstbeschreibung oder einer unpräzise definierten Ontologie mit unerwünschtem Verhalten bei der Dienstkomposition zu rechnen, in dem Sinne, dass Dienste aneinander gebunden werden, die semantisch zu weit differieren. Dies würde dann zu einem unerwünschten Verhalten im eHome führen.

5.7 Adaptierung

Die vierte und letzte Phase des semantischen Teilprozesses ist die Adaptierung, die zwischen semantisch passenden Diensten vorgenommen wird. In der dritten Phase wurden semantische Matches gesucht, um Dienste entsprechend ihrer benötigten und angebotenen Funktionalitäten miteinander zu verbinden. Das Auffinden semantischer Matches wird durch die *Service Registry* ermöglicht und erlaubt eine Dienstkomposition unabhängig von den konkreten Dienstschnittstellen. Die Komposition wird so flexibler und es können genauere Matches bestimmt werden. Damit eine Dienstkomposition ausführbar ist, muss jedoch eine Umsetzung der Matches auf syntaktischer Ebene erfolgen. Dazu werden Adapterkomponenten benötigt, die die syntaktischen Inkompatibilitäten auf Basis der semantischen Dienstbeschreibungen überbrücken. Nicht jedes semantische Match ist adaptierbar. Es sind daher weitere Schritte erforderlich, um die Adaptierbarkeit der Dienste zu einem semantischen Match zu bestimmten. Dann kann ggfs. eine Adapterkomponente durch die Laufzeitumgebung generiert und zur Komposition der Dienste verwendet werden.

5.7.1 Adaptierbarkeit von Komponenten

Wird ein semantisches Match gefunden, so geht damit im Allgemeinen kein Match auf der syntaktischen Ebene einher. Durch die semantische Dienstbeschreibung können die Operationen der betroffenen Schnittstellen ineinander überführt werden. Die Signaturen der Operationen können jedoch ebenfalls Differenzen aufweisen. Häufig ist die Adaption dieser Mismatches auf Signaturebene jedoch unproblematisch, wenn davon ausgegangen werden kann, dass die in der Signatur enthaltenen Elemente korrespondieren [BBG+06, LFJ07]. Betrachtet man eine angebotene Operation $o_{provided}$ eines Dienstes und eine benötigte Operation $o_{required}$ eines semantisch passenden Dienstes, so treten häufig die folgenden typischen Fälle auf:

- Unterschiedliche *Bezeichner* korrespondierender Elemente.
  - Namen der Operationen
  - Namen der Parameter in den Signaturen
- Inkompatible *Typen* korrespondierender Elemente. Damit inkompatible Typen adaptiert werden können, muss eine Generalisierungs- bzw. Spezialisierungsbeziehung zwischen den Typen vorliegen. Die Notation $T_1 \subseteq T_2$ bedeutet, dass der Datentyp T_1 ein Untertyp bzw. eine Spezialisierung des Datentyps T_2 ist. Dann ist T_2 ein Obertyp bzw. eine Generalisierung von T_1.
  - Typen der Rückgabewerte können adaptiert werden, wenn gilt:

    $$type(o_{provided}) \subseteq type(o_{required})$$

  - Typen der Parameter können adaptiert werden, wenn für korrespondierende Parameter an den Positionen $p_{provided}$ bzw. $p_{required}$ gilt:

    $$parameters(o_{provided}, p_{provided}) \supseteq parameters(o_{required}, p_{required})$$

  - Typen der Ausnahmen können adaptiert werden, wenn für korrespondierende Ausnahmen an den Positionen $p_{provided}$ bzw. $p_{required}$ gilt:

    $$exceptions(o_{provided}, p_{provided}) \subseteq exceptions(o_{required}, p_{required})$$

- Bei einer inkompatiblen *Struktur* der Signaturen von Operationen kann adaptiert werden:
  - Eine unterschiedliche Reihenfolge der Eingabeparameter.

- Eine unterschiedliche Anzahl von Parametern, falls für alle fehlenden oder zusätzlichen Parameter konstante Werte angegeben werden können.

Die obigen Inkompatibilitäten auf der Ebene von Signaturen können auch in Kombination auftreten. Dann müssen bei der Adaptierung die entsprechenden Schritte verknüpft und hintereinander ausgeführt werden. Die hier behandelten Inkompatibilitäten hängen mit denen aus Abschnitt 5.5.2 zusammen. Dort wurden Inkompatibilitäten bei der Abbildung von Schnittstellenelementen auf die Ontologie diskutiert. Nachdem ein Match auf Basis der semantischen Abbildung gefunden wurde, treten diese Inkompatibilitäten entsprechend auch bei der Adaption zwischen den Schnittstellen auf. Zunächst wird analysiert, in wie weit die Funktionalitäten der Ontologie durch die an einem Match beteiligten Schnittstellen abgedeckt werden. Daraus ergibt sich die Adaptierbarkeit der Dienste.

5.7.2 Funktionalitätsüberdeckung

Ein semantisches Match ist Voraussetzung zur Analyse der Adaptierbarkeit von Diensten. Bisher wurde die Ebene von Capabilities betrachtet. Damit eine Adaption erfolgen kann, müssen jedoch zusätzlich die Details der Schnittstellen betrachtet werden. Die Inkompatibilitäten aus Abschnitt 5.5.2 können bei der Abbildung von Schnittstellenelementen auf die Ontologie auftreten, sie müssen aber auch bei der Analyse semantischer Matches berücksichtigt werden. Ein wichtiger Schritt ist die Analyse der sogenannte *Funktionalitätsüberdeckung*. Dabei wird überprüft, in welchem Umfang eine Schnittstelle die zugehörige in der Ontologie spezifizierte Funktionalität abdeckt. Eine solche Überprüfung ist erforderlich, um festzustellen, ob ein semantisches Match auch durch eine passende Adapterkomponente umsetzbar ist.

In Abbildung 5.18 ist ein Beispiel dargestellt, um die Problemstellung zu veranschaulichen. Auf der linken Seite der Abbildung ist eine Schnittstelle `DoorDriver` dargestellt, die in diesem Beispiel von einem Dienst benutzt werden soll. Diese Schnittstelle umfasst zwei Operationen `getClosed` und `setClosed`. Erstere Operation liefert einen Wert vom Typ `boolean` zurück, während letztere einen Eingabeparameter vom Typ `boolean` erwartet. Beide Operationen sind auf die Accesses get bzw. set des Attributes Closed der Ontologie abgebildet. Auf der

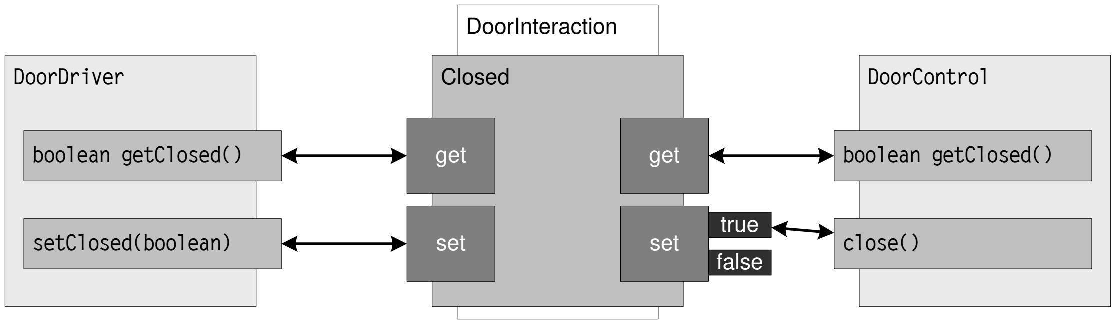


Abbildung 5.18: Funktionalitätsüberdeckung für ein semantisches Match.

rechten Seite der Abbildung ist die Schnittstelle `DoorControl` dargestellt, die von einem Treiberdienst zur Steuerung von Türen angeboten wird. Diese Schnittstelle hat ebenfalls eine Operation `getClosed`, die einen Wert vom Typ `boolean` zurück liefert. Die Abbildung auf die Ontologie ist analog zur entsprechenden Operation der Schnittstelle `DoorDriver`. In `DoorControl` ist noch eine zweite Operation `close` enthalten, die keine Parameter erwartet. Diese Operation unterscheidet sich zur vorher beschriebenen Operation `setClosed`. Während `setClosed` die Möglichkeit bietet, die Tür durch Angabe eines entsprechenden Parameterwertes zu öffnen oder zu schließen, erlaubt die Operation `close` nur das Schließen einer Tür. Der Treiberdienst steuert in diesem Fall eine Tür, die zwar einen elektronischen Schließmechanismus hat, jedoch nur manuell geöffnet werden kann. Im Fall der Schnittstelle `DoorDriver` wird eine Tür erwartet, die elektronisch sowohl geöffnet als auch geschlossen werden kann. Durch die Schnittstelle `DoorControl` ist also nur ein Teil der in der Ontologie beschriebenen Funktionalität verfügbar. Die Operation `close` wird daher auf den Slot true des Access set abgebildet. Der Slot partitioniert den Wertebereich des zugehörigen Attributes, sodass über einen Access nur auf bestimmte Werte des Wertebereichs zugegriffen werden kann.

Wenn davon ausgegangen wird, dass es eine kanonische Ordnung der Operationen gibt, so kann die Funktionalitätsüberdeckung in Form einer Liste angegeben werden. Im obigen Beispiel ergibt sich für die Schnittstelle `DoorDriver` die Liste $[\{0,1\},\{0,1\}]$ und für die Schnittstelle `DoorControl` die Liste $[\{0,1\},\{0\}]$. Im Folgenden wird das Konzept der Funktionalitätsüberdeckung formal eingeführt.

Definition 5.11 (Funktionalitätsüberdeckung)
Gegeben seien eine Schnittstelle i, eine Capability c und eine Funktionalitäts-

abbildung m mit $m = (operations(i) \mathbin{\dot{\cup}} allAccesses(c), E, l)$. Sei $accessList(c)$ eine Liste, in der alle Accesses *von c in kanonischer Weise geordnet sind. Die* ***Funktionalitätsüberdeckung*** *$coverage(i, c)$ von i bezüglich c ist eine Liste von Mengen mit*

$$coverage(i, c)[k] = coverage_{in}(i, c)[k] \cap coverage_{out}(i, c)[k]$$

mit

$$coverage_{in/out}(i, c)[k] = accessCoverage_{in/out}(m, accessList(c)[k])$$

Es ist $|coverage(i, c)| = |accessList(c)|$. Hierbei gilt für $acc \in allAccesses(c)$ mit $acc = access(attr)$

$$accessCoverage_{in}(m, acc) = \bigcup_{(op,acc)\,\in\, E} bild(transformation_{in}(l(op, acc)))$$

und

$$accessCoverage_{out}(m, acc) = \bigcup_{(acc,op)\,\in\, E} urbild(transformation_{in}(l(acc, op)))$$

mit

$$accessCoverage_{in/out} \subseteq dom(type(attr))$$

Für zwei Funktionalitätsüberdeckungen cc_1 und cc_2 mit $cc_1 = coverage(i_1, c)$ und $cc_2 = coverage(i_2, c)$ gilt:

$$cc_1 \subseteq cc_2 \Leftrightarrow cc_1[k] \subseteq cc_2[k] \qquad \forall k \in 1..|allAccesses(c)|$$

Die Funktionalitätsüberdeckung umfasst diejenigen Accesses einer Capability, die durch eine Schnittstelle abgedeckt werden und für die eine Abbildung von der Schnittstelle in die Ontologie und umgekehrt möglich ist. Betrachtet man nun die Schnittstellen zweier Dienste, die auf Basis eines semantischen Matches komponiert werden sollen, so dient die Funktionalitätsüberdeckung der beiden Schnittstellen als Grundlage für die Adaptierung. Abhängig davon, in welchem Grad die in der Ontologie beschriebene Funktionalität abgedeckt wird, kann eine Adaption zwischen Diensten erfolgen. Im Folgenden wird dazu die *semantische*

Adaptierbarkeit formal definiert. Vorbereitend werden die Begriffe der *semantischen Inklusion* und der *semantischen Äquivalenz* eingeführt.

Definition 5.12 (Semantische Inklusion)
Für zwei Schnittstellen i_1 und i_2 gilt bezüglich einer Capability c:

$$i_1 \subseteq_c i_2 \Leftrightarrow coverage_{in}(i_1, c) \subseteq coverage_{in}(i_2, c)$$

*Schnittstelle i_1 ist in Schnittstelle i_2 **semantisch enthalten**.*

Definition 5.13 (Semantische Äquivalenz)
Für zwei Schnittstellen i_1 und i_2 gilt

$$i_1 \equiv_c i_2 \Leftrightarrow i_1 \subseteq_c i_2 \text{ und } i_2 \subseteq_c i_1$$

*Die Schnittstellen sind bezüglich einer Capability c **semantisch äquivalent**.*

Definition 5.14 (Semantische Adaptierbarkeit)
*Seien i_1 und i_2 Schnittstellen, c eine Capability und m eine Funktionalitätsabbildung mit $m = (operations(i_1) \dot{\cup} accesses(c), E, l)$. Die Schnittstelle i_1 ist dann auf die Schnittstelle i_2 **semantisch adaptierbar**, wenn gilt:*

1. $\forall op \in operations(i_1) : \exists (op, v) \in E$
2. $coverage_{in}(i_1, c) \subseteq coverage_{out}(i_2, c)$

Punkt eins entspricht der Bedingung, dass für alle Operationen von i_1 eine Abbildung gegeben sein muss. Die Forderung in Punkt zwei gewährleistet, dass zu jeder Operation in i_1 eine semantisch zugehörige Operation in i_2 vorhanden ist.

Nur wenn eine semantische Adaptierbarkeit zwischen Dienstschnittstellen gegeben ist, kann eine passende Adapterkomponente generiert werden. Der vorgestellte Ansatz garantiert nicht, dass in jedem Fall, in dem theoretisch eine Adaption möglich wäre, auch die semantische Adaptierbarkeit gegeben ist. Damit kann es prinzipiell vorkommen, dass adaptierbare Dienste nicht als solche erkannt werden. Dies liegt daran, dass die Abbildung zwischen den Dienstschnittstellen in indirekter Form über die Ontologie vorliegt. Zwei Schnittstellen könnten z. B. in semantisch korrespondierenden Operationen einen Parameter desselben Typs enthalten, während in der Ontologie das entsprechende Attribut einen anderen Typ aufweist. Wenn also der in den Schnittstellen verwendete Typ nicht mit dem

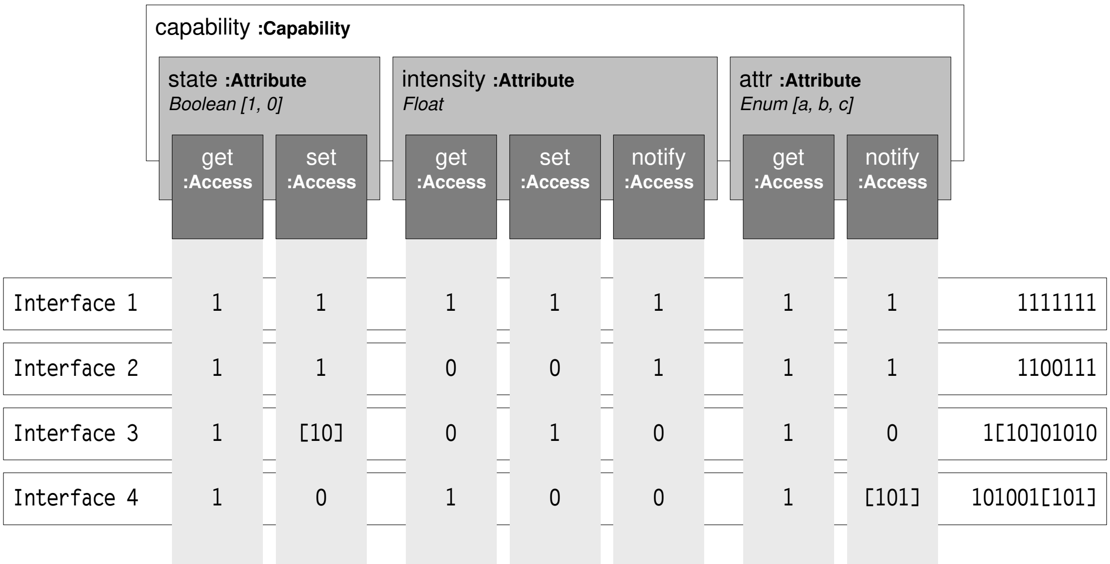


Abbildung 5.19: Beispiel zur Funktionalitätsüberdeckung.

Typ in der Ontologie kompatibel ist, dann ist ggfs. keine Adaption möglich, selbst wenn in den Schnittstellenoperationen genau derselbe Typ verwendet wird. Ein ähnlicher Fall kann auftreten, wenn zwei Schnittstellen eine Operation beinhalten, die in der Ontologie gar nicht erfasst ist. Diese könnten zwar prinzipiell adaptiert werden, dies ist jedoch im Rahmen des vorgestellten Ansatzes nicht möglich, da das Wissen über die semantische Äquivalenz dann nicht vorhanden ist, selbst wenn die Operationen genau dieselbe Signatur aufweisen und daher ohne Adaptionsschritte passen würden. Diese Fälle können im hier vorgestellten Ansatz nicht aufgelöst werden, da die Abbildung auf die Ontologie Voraussetzung für das Auffinden semantischer Matches und die Adaption ist.

Bei der Diskussion des Beispiels in Abbildung 5.18 wurde bereits die Darstellung der Funktionalitätsüberdeckung in Form einer Liste angesprochen. Dort ergab sich für die Schnittstelle `DoorControl` die Liste $[\{0,1\},\{0\}]$. Eine solche Liste wird für alle Schnittstellen verwaltet, um die Überdeckung der in der Ontologie spezifizierten Funktionalitäten zu speichern und für die Dienstkomposition zu verwenden. Die vier Interfaces in Abbildung 5.19 zeigen ein Beispiel dazu. Für jedes Interface wird ein Bitstring erzeugt und gespeichert, der die Funktionalitätsüberdeckung repräsentiert. Die Repräsentation der Überdeckung als Bitstring erlaubt eine effiziente Verarbeitung bei der semantischen Dienstkomposition. Alle Capabilities, Accesses etc. sind in einer bestimmten Reihenfolge in der Ontologie gespeichert. Dieser gegebene Aufbau kann verwendet werden, um eine kanonische Reihenfolge der Elemente zu erreichen. In Abbildung 5.19 wird eine Capability betrachtet, die die Attributes state vom Typ Boolean, intensity vom Typ Float und attr vom Typ Enum hat. Das Enum-Attribute kann hier die Werte a, b und c annehmen. Für state sind die Accesses get und set definiert. Für intensity sind get, set und notify definiert. Über notify können sich Dienste registrieren, die über Veränderungen des Attributes benachrichtigt werden wollen. Schließlich gibt es für attr noch die Accesses get und notify.

Die Schnittstelle `Interface 1` überdeckt die in der Ontologie spezifizierte Funktionalität vollständig. Das wird dadurch ausgedrückt, dass im zugehörigen Bitstring die Bits für alle Accesses gesetzt sind. Der Bitstring ist daher `1111111`. `Interface 2` stellt keine Operationen für den get- und den set-Access des Attributes intensity zur Verfügung. Die entsprechenden Bits sind daher im Bitstring von `Interface 2` nicht gesetzt. Im `Interface 3` gibt es eine Besonderheit. Hier wird der set-Access des Attributes state nicht vollständig abgedeckt. Hier wird nur das Setzen auf den Wert true bzw. 1 unterstützt, ein Setzten auf false bzw. 0 ist über diese Schnittstelle nicht möglich. Dieser Fall ist analog zum Fall der Schnittstelle `DoorControl` aus dem Beispiel in Abbildung 5.18. Hier erfolgt somit eine Partitionierung des Wertebereichs, die im Bitstring durch eckige Klammern an der entsprechenden Stelle eingefügt wird. In diesem Fall wird die Bitfolge `[10]` eingefügt, da der an Position eins in der Ontologie festgelegte Wert true gesetzt werden kann, der an Position zwei festgelegte Wert false jedoch nicht. Ein ähnlicher Fall tritt in `Interface 4` für das Attribute attr auf. Hier tritt eine Partitionierung des notify-Accesses auf. Der Dienst zum `Interface 4` kann nur Benachrichtigungen für die Werte a und c übermitteln, was die Bitfolge `[101]` ergibt. Für den möglichen Wert b erfolgt keine Benachrichtigung, dieser Wert kann über den get-Accesses abgefragt werden, denn dieser wird vollständig abgedeckt. Ein solcher Fall könnte beispielsweise auftreten, wenn der Betriebszustand eines Geräts über eine Schnittstelle abgefragt werden kann. Dann könnten etwa die Zustandswerte on, standby und off möglich sein. Der oben beschriebene Fall würde dann bedeuten, dass eine Benachrichtigung nur für die Zustände on und off möglich ist während standby nicht abgedeckt wird.

Das Listing 5.2 zeigt einen Ausschnitt der Implementierung zur Prüfung von Schnittstellen auf semantische Inklusion. Die Methode `containsAll` erwartet zwei

```
public boolean containsAll(String[] cov_s1, String[] cov_s2) {
    if(cov_s1.length != cov_s2.length) {
        return false;
    }
    for(int i = 0; i < cov_s1.length; i++) {
        if(!contains(cov_s1[i], cov_s2[i])) {
            return false;
        }
    }
    return true;
}

public boolean contains(String cov1, String cov2) {
    if(!cov1.equals(cov2)) {
        String[] eq = equalLength(cov1, cov2);
        if(eq == null) {
            return false;
        }
        String c1 = eq[0];
        String c2 = eq[1];
        if(c1.length() == 1 && c1.compareTo(c2) < 0) {
            return false;
        }
        for(int i = 0; i < c1.length(); i++) {
            boolean contained =
                contains(c1.substring(i, i+1), c2.substring(i, i+1));
            if(!contained) {
                return false;
            }
        }
    }
    return true;
}
```

Listing 5.2: Überprüfung der semantischen Inklusion

String-Arrays `cov_s1` und `cov_s2` als Parameter und überprüft, ob die Schnittstelle zu `cov_s2` in der zu `cov_s1` semantisch enthalten ist. Die String-Arrays speichern die Bitstrings der Funktionalitätsüberdeckung der Schnittstellen eines Matches im String-Format, was Vorteile bei der Verarbeitung mit sich bringt. Für den Bitstring `1[10]01010` ergibt sich z. B. das Array `[1,10,0,1,0,1,0]`. Unterschiedlich lange Bitstrings können nicht verglichen werden und werden daher in Zeile 2 direkt verworfen. In der folgenden Schleife in Zeile 5 werden der Reihe nach alle Einträge beider String-Arrays miteinander verglichen. Dazu wird die Methode `contains` verwendet, die für zwei Strings `cov1` und `cov2` feststellt, ob `cov1` in `cov2` semantisch enthalten ist. Wenn es Einträge gibt, bei denen keine semantische Inklusion vorliegt, so besteht auch auf der Ebene der ganzen Schnittstellen keine semantische Inklusion und die `containsAll` liefert `false` zurück. Die Implementierung der Methode `contains` beginnt im Listing mit Zeile 13. Zunächst werden in Zeile 15 einstellige Strings ggfs. auf gleiche Länge gebracht. Dazu werden die führenden Stellen im String mit `1` bzw. `0` aufgefüllt, aus `1` und `101` wird damit das Array `[111,101]`. In Zeile 21 wird der String-Vergleich für einstellige Strings durchgeführt. Dabei wird überprüft, ob die Funktionalitätsüberdeckung des ersten Strings kleiner als die des zweiten ist. Wenn dies der Fall ist, so ist `cov2` nicht in `cov1` semantisch enthalten. Für längere Arrays wird ab Zeile 24 über alle Einträge iteriert und es erfolgt jeweils ein rekursiver Aufruf der `contains`-Methode. Um zwei Schnittstellen `s1` und `s2` auf semantische Äquivalenz zu prüfen, muss sowohl `containsAll(cov_s1,cov_s2)` als auch `containsAll(cov_s2,cov_s1)` mit `true` ausgewertet werden.

5.7.3 Aktive und passive Komponenten

Ein weiterer Aspekt, der für die Adaptierung von Dienstfunktionalitäten von Belang ist, ist die Unterscheidung zwischen aktiven und passiven Komponenten. Die Kommunikation zwischen Diensten kann unterschiedlichen Paradigmen folgen. Ein Dienst, der eine Information bereitstellt, kann diese aktiv anderen Diensten mitteilen, wenn sie an dieser Information interessiert sind. Andererseits kann ein Dienst auch passiv Informationen zur Verfügung stellen. Interessierte Dienste müssen dann Anfragen stellen und diese Informationen bei Bedarf abrufen. Der erste Fall kann durch das Entwurfsmuster *Observer*, auch *Listener*

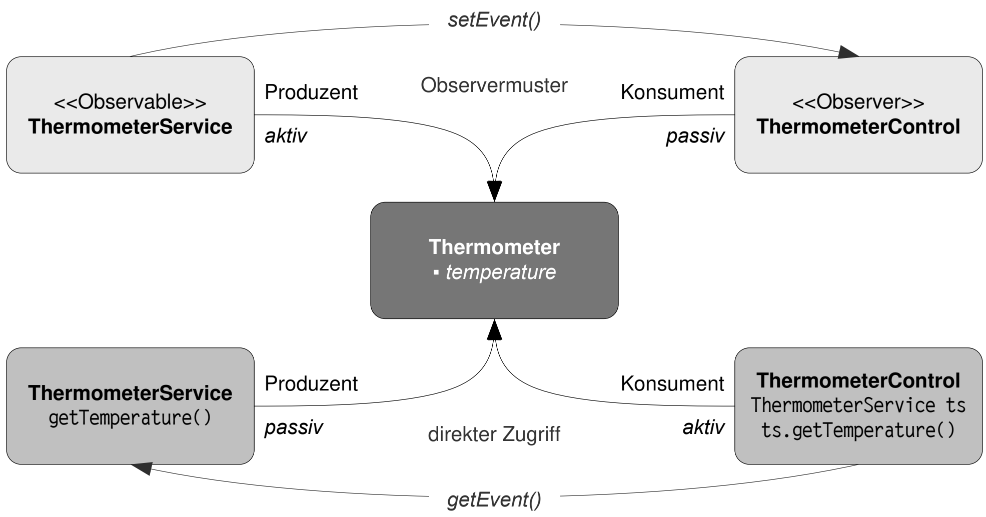


Abbildung 5.20: Adaption aktiver und passiver Komponenten.

genannt, realisiert werden. Der letzte Fall kann durch den normalen Aufruf einer Operation und der Übermittlung eines Rückgabewertes umgesetzt werden.

Das Observermuster wird realisiert, indem sich interessierte Dienste als Observer bei einem als Observable (dt. *observierbar*) gekennzeichneten Dienst anmelden, um mit Informationen versorgt zu werden. Analog können sich Dienste auch wieder abmelden. Observer-Dienste implementieren eine bestimmte Schnittstelle, die dem Observable-Dienst bekannt ist, sodass dieser eine entsprechende Methode der Schnittstelle, z. B. `update` oder `notify`, aufrufen kann. So können alle angemeldeten Observer-Dienste über ein Ereignis benachrichtigt werden.

Zwei aktive Dienste oder zwei passive Dienste können nicht direkt miteinander interagieren, selbst wenn ansonsten keine Inkompatibilitäten vorliegen. Die verschiedenen möglichen Fälle sind in Abbildung 5.20 an einem Beispiel veranschaulicht. Auf der linken Seite der Abbildung ist ein Dienst ThermometerService zu sehen, der den Treiber eines Sensors zur Messung der Umgebungstemperatur darstellt. Auf der rechten Seite der Abbildung ist ein Dienst ThermometerControl dargestellt, der einen Top-Level-Dienst repräsentiert, der auf die Daten des Temperatursensors zurückgreifen soll. In der Mitte der Abbildung ist die verknüpfende Funktionalität Thermometer dargestellt. Diese hat das Attribut temperature, das den aktuellen Temperaturwert repräsentiert.

Die Interaktion zwischen den Diensten kann nun auf verschiedene Arten geschehen. Der Produzent der Temperaturinformation ThermometerService kann als aktive Komponente vorliegen, wie im Fall oben links dargestellt. Dann bietet er als Observable-Dienst die Möglichkeit, andere Dienste bei Veränderungen der Temperatur zu benachrichtigen. Ein passender Konsument ist der Observer-Dienst TemperatureControl, der oben rechts dargestellt ist. Dieser kann als passive Komponente die Benachrichtigungen des aktiven ThermometerService entgegen nehmen. Unten links ist der ThermometerService als passive Komponenten dargestellt. In diesem Fall werden Temperaturänderungen nicht aktiv an andere Dienste übermittelt, die Temperatur kann stattdessen über die Methode `getTemperature` abgefragt werden. Ein passender Konsument für diese Art der Implementierung ist der aktive Dienst TemperatureControl, der unten rechts dargestellt ist. Für eine Instanz `ts` des ThermometerService ruft er aktiv die Methode `getTemperature` auf, um die aktuelle Umgebungstemperatur abzufragen.

Für eine funktionierende Kommunikation ist, wie an diesem Beispiel zu sehen, immer auf der einen Seite eine aktive und auf der anderen Seite eine passive Realisierung der beteiligten Komponenten erforderlich. In der oberen Hälfte der Abbildung 5.20 ist der Fall zu sehen, in dem der Produzent der Information die aktive Rolle übernimmt, der Konsument ist in diesem Fall passiv. Der Produzent ruft den Konsumenten auf, was durch die setEvent()-Kante angedeutet wird. In der unteren Hälfte der Abbildung ist der umgekehrte Fall dargestellt, in dem der Produzent die passive Rolle übernimmt und der Konsument aktiv ist. Hier ruft der Konsument den Produzenten auf, was durch die setEvent()-Kante in der Abbildung dargestellt wird.

Für die Abbildung der Schnittstellen auf die Ontologie ist die Unterscheidung zwischen aktiven und passiven Komponenten nicht von Bedeutung. Bei der Adaption muss diese Unterscheidung jedoch berücksichtigt werden, wie obiges Beispiel zeigt. Wenn durch einen Adapter zwei passive Komponenten verbunden werden sollen, dann muss der Adapter eine aktive Rolle übernehmen und selbstständig Informationen des Produzenten abrufen und an den Konsumenten weiterreichen. Wenn zwei aktive Komponenten verbunden werden sollen, so muss der Adapter eine passive Rolle übernehmen und die Informationen des Produzenten entgegennehmen und zwischenspeichern, sodass sie für den Konsumenten abrufbar sind. Je nachdem welcher Fall vorliegt, muss der zu generierende Adapter anders aufgebaut sein.

5.7.4 Adaptergenerierung

Die Adaptergenerierung findet zur Laufzeit des eHome-Systems statt und basiert auf den zuvor durchgeführten Analysen. Es handelt sich dabei um eine *Komponentenadaption* (vgl. Abschnitt 5.2.3), die wie eine Designzeitadaption die Architektur des Systems betrifft, jedoch wie eine Laufzeitadaption erst zur Laufzeit durchgeführt wird. *Adapter* werden häufig auch *Wrapper* genannt. Die Aufgabe eines Adapters ist es, zwischen zwei inkompatiblen Schnittstellen zu übersetzen [GHJV95]. Dabei wird die Schnittstelle, die von einer Klasse implementiert wird, in eine andere Schnittstelle transformiert, sodass diese von einem Klienten gemäß seinen Anforderungen verwendet werden kann. Genau dies ist erforderlich, damit semantisch passende Dienste trotz syntaktischer Inkompatibilitäten miteinander kommunizieren können. Die semantische Dienstbeschreibung umfasst alle Informationen, die benötigt werden, um passende Adapterkomponenten zu generieren. Voraussetzung für die Adaptergenerierung ist, dass die Adaptierbarkeit gegeben ist.

Um dynamisch Adapterklassen zur Adaption zweier spezifischer Dienstschnittstellen zu generieren, wird ein entsprechender Laufzeitmechanismus benötigt. Im Rahmen dieser Arbeit wird dazu ein Werkzeug zur strukturellen Reflection in Java namens *Javassist* eingesetzt. Javassist bringt einen eingebauten Compiler mit, der es ermöglicht, zur Laufzeit Quellcode zu übersetzen und in den Bytecode einer Java-Klasse zu integrieren [Chi00, Chi09]. Bevor ein semantisches Match deployt werden kann, muss eine passende Adapterklasse generiert werden. Von dieser Klasse wird dann eine Instanz erzeugt, die beim Deployment der Dienstbindungen zwischen den betroffenen Diensten vermittelt.

In Abbildung 5.21 ist das Zusammenspiel zwischen zwei komponierten Diensten und der dazwischen eingefügten Adapterkomponente veranschaulicht. Oben links befindet sich ein Top-Level-Dienst LightingService, der eine Schnittstelle LampControl benutzt. Unten rechts befindet sich ein Treiberdienst LampDriverService, der die Schnittstelle LampDriver implementiert und damit die Steuerung von Lampen ermöglicht. Beide Schnittstellen sind semantisch äquivalent und entsprechen der Capability Illumination in der Ontologie. Damit eine Komposition der Dienste möglich ist, wird eine Adapterkomponente generiert, die beide Dienste miteinander verbindet. Der Adapter ist in der Mitte der Abbildung dargestellt. Er implementiert die Schnittstelle LampControl, die vom Top-Level-

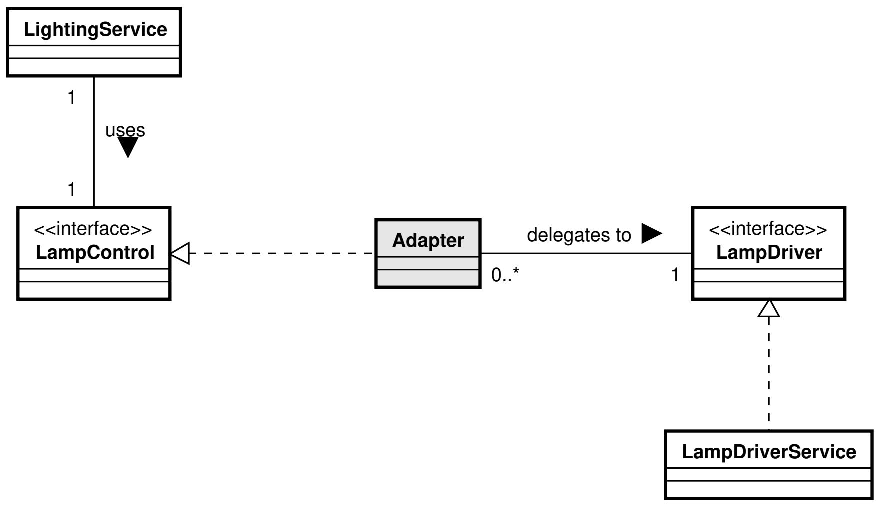


Abbildung 5.21: Struktur des Adaptermusters.

Dienst benutzt wird. Die Implementierung der Operationen der LampControl-Schnittstelle im Adapter stützt sich dabei auf die Operationen der Schnittstelle LampDriver ab, die durch den Treiberdienst zur Verfügung gestellt wird. Auf diese Weise können Aufrufe der Schnittstelle LampControl an Methoden der semantisch äquivalenten Schnittstelle LampDriver weitergereicht werden. Damit die Dienstkommunikation funktioniert, müssen in den Implementierungen der Methoden die spezifischen Anpassungen vorgenommen werden, die sich aus den semantischen Beschreibungen der Schnittstellen ergeben. Das Grundgerüst eines Adapter kann dabei nach einem festen Schema erzeugt werden.

Listing 5.3 zeigt den allgemeinen Teil einer generierten Adapterklasse. Dieser Teil ist bis auf die eingesetzten Schnittstellennamen unabhängig von den beteiligten Diensten. Die Methoden der Zielschnittstelle müssen diesem Gerüst eines Adapters noch hinzugefügt werden. Die Erzeugung der entsprechenden Methoden samt zugehöriger Implementierung ist die eigentliche Aufgabe, die es bei der Adaptergenerierung zu lösen gilt. Für das Beispiel des Lampentreibers, das bereits in Abbildung 5.21 verwendet wurde, muss der Adapter die Schnittstelle `LampControl` implementieren. Dies ist im Listing 5.3 in Zeile 1 zu sehen. Für die zu verwendende Zielschnittstelle `LampDriver` wird in Zeile 3 eine lokale Variable `_adaptee` deklariert. Die Methode `_bind` in Zeile 6 dient dazu, den Zieldienst der Adaption zu binden. Es wird zunächst geprüft, ob das übergebene Objekt vom

```
1  public class Adapter implements LampControl {
2
3      private LampDriver _adaptee;
4
5      // Bind the adaptee
6      public void _bind(Object obj) {
7          if(obj instanceof LampDriver) {
8              this._adaptee = (LampDriver)obj;
9          }
10     }
11
12     // Clear the association
13     public void _unbind(Object obj) {
14         if(obj == _adaptee) {
15             this._adaptee = null;
16         }
17     }
18
19     // Other methods of the LampControl interface ...
20
21 }
```

Listing 5.3: Aufbau einer Adapterklasse

Typ `LampDriver` ist. Falls dies der Fall ist, wird die Referenz als Bindung in der Variablen `_adaptee` gespeichert. An dieses Objekt werden später alle Aufrufe des Adapters delegiert. Um eine Bindung wieder zu entfernen, gibt es in Zeile 13 die Methode `_unbind`, die überprüft, ob das übergebene Objekt zur Zeit gebunden ist, und dann ggfs. die Bindung entfernt. Die Methoden `_bind` und `_unbind` treten in ähnlicher Form auch in den Implementierungen der eHome-Dienste auf. Dort werden sie beim Deployment aufgerufen, um die Bindungen zwischen den Diensten herzustellen. In diesem Fall dienen sie dazu, den Adapter zwischen zwei Dienstinstanzen einzufügen und so das in Abbildung 5.21 dargestellte Adaptermuster umzusetzen. Ab Zeile 19 des Listings folgen dann die restlichen Methoden der vom Adapter implementierten Schnittstelle `LampControl`. Die Implementierung dieser Methoden ist von der semantischen Beschreibung der beiden Schnittstellen abhängig und den sich daraus ergebenen notwendigen Transformationen. Wie die Methoden generiert werden, wird weiter unten erläutert.

```
1 public void setLight(boolean light) {
2     if(light == true) {
3         _adaptee.setLightOn();
4         return;
5     }
6     if(light == false) {
7         _adaptee.setLightOff();
8         return;
9     }
10 }
```

Listing 5.4: Methode `setLight` der Adapterklasse im Beispiel

In Listing 5.4 ist ein Beispiel für eine generierte Adaptermethode zu sehen. Dabei wird die Methode `setLight` mit einem booleschen Parameter auf die Methoden `setLightOn` und `setLightOff` der Zielschnittstelle abgebildet. In Zeile 2 wird geprüft, ob der Eingabeparameter `light` den Wert `true` hat, dann wird in Zeile 3 die Methode `setLightOn` der Zielschnittstelle aufgerufen. In Zeile 6 wird der Fall behandelt, dass `light` den Wert `false` hat. Dann wird in Zeile 7 die Methode `setLightOff` aufgerufen. Dieses einfache Beispiel zeigt, wie der Aufruf einer Methode der zu adaptierenden Schnittstelle auf zwei unterschiedliche Methoden der Zielschnittstelle aufgeteilt wird.

In Listing 5.5 ist die Methode `createAdapterClass` dargestellt, die für die Codegenerierung der Adaptermethoden zuständig ist. Diese Methode liefert als Rückgabewert ein Objekt vom Typ `Class` zurück, das die eigentliche Adapterklasse darstellt und dann mit dem zu Anfang dieses Abschnitts erwähnten Javassist kompiliert und dynamisch dem bereits laufenden System hinzugefügt werden kann. Zum Erzeugen dieser Adapterklasse benötigt die Methode als Eingabe eine Liste der Quelltexte der benötigten Methoden. Diese Quelltexte werden durch den Eingabeparameter `methods`, der eine Liste mit Elementen vom Typ `String` ist, übergeben. Im Rumpf der Methode wird nach einigen Initialisierungsoperationen zunächst in Zeile 6 eine neue Klasse `adapterClass` erzeugt. Dazu dient die Methode `makeClass`, die als Eingabeparameter den Namen der neu erzeugten Klasse übergeben bekommt. Der Name beginnt immer mit `ehome.adapter.` und setzt sich dann aus den Namen der zu adaptierenden Schnittstelle `adapter` und der Zielschnittstelle `adaptee` zusammen. So würde die Adapterklasse für den in Abbildung 5.21 dar-

```
private Class<?> createAdapterClass(List<String> methods) {
    try {
        ClassPool pool = ClassPool.getDefault();
        pool.insertClassPath(new ClassClassPath(this.getClass()));

        CtClass adapterClass = pool.makeClass("ehome.adapter."
                + adaptee.substring(adaptee.lastIndexOf(".") + 1) + "2"
                + adapter.substring(adapter.lastIndexOf(".") + 1));

        CtField adapteeField =
                new CtField(pool.getCtClass("java.lang.Object"),
                "_adaptee", adapterClass);
        adapterClass.addField(adapteeField);
        CtConstructor constr =
                CtNewConstructor.defaultConstructor(adapterClass);
        adapterClass.addConstructor(constr);

        adapterClass.addInterface(pool.getCtClass(adapter));

        for(String methodText : methods) {
            CtMethod meth = CtNewMethod.make(methodText, adapterClass);
            adapterClass.addMethod(meth);
        }
        try {
            return adapterClass.toClass();
        } catch (CannotCompileException e) {
            e.printStackTrace();
        }
    } catch (NotFoundException e) {
        e.printStackTrace();
    } catch (CannotCompileException e) {
        e.printStackTrace();
    }
    return null;
}
```

Listing 5.5: Methode `createAdapterClass` zur Generierung einer Adapterklasse gestellten Fall z. B. ehome.adapter.LampDriver2LampControl heißen. In Zeile 10 wird eine neue Variable vom Typ Object mit dem Namen _adaptee erstellt, die später die Referenz auf das Zielobjekt des Adapters speichern wird. In Zeile 13 wird die Variable der Adapterklasse hinzugefügt. In Zeile 14 bis 16 wird ein Konstruktor erstellt und ebenfalls der Adapterklasse hinzugefügt, damit diese später instanziiert werden kann. In Zeile 18 wird dann der Adapterklasse die zu adaptierende Schnittstelle adapter hinzugefügt, die von der Adapterklasse implementiert werden muss. Ab Zeile 20 wird in einer for-Schleife über alle Einträge methodText der Liste methods mit den Methodenquelltexten iteriert. Die make-Methode der Javassist-Klasse CtNewMethod sorgt für die Kompilation der Quelltexte, die dann jeweils in Zeile 22 als neue Methoden der Adapterklasse hinzugefügt werden. Abschließend wird der neue Adapter in Zeile 25 durch die toClass-Methode in ein Objekt der Klasse java.lang.Class konvertiert. Dieses stellt den neu erzeugten Adapter dar und wird von der Methode zurückgeliefert.

Die Methodenquelltexte der Adapterklasse, die mit dem oben beschriebenen Verfahren kompiliert und dem Adapter hinzugefügt werden, müssen zuvor ebenfalls generiert werden. Dazu wird in dieser Arbeit ein Mechanismus verwendet, der auf JET-Templates basiert. *JET*[3] steht für ***J**ava **E**mitter **T**emplates* und ist eine Technik zum dynamischen Erzeugen von Quellcode auf der Basis zuvor erstellter Vorlagen, den sogenannten *Templates*. Die Notation von JET basiert auf Pfadausdrücken, die eingesetzt werden, um bestimmte Elemente eines XML-Dokuments zu finden. Da die Spezifikationen der Dienste, die an einem zu adaptierenden Match beteiligt sind, ohnehin in einem XML-Format vorliegen, bietet sich dieser Weg für die Generierung des Quellcodes an.

In Listing 5.6 ist ein Beispiel für ein JET-Template zu sehen, das zur Generierung von Adaptermethoden verwendet wird. Variablen beginnen in der verwendeten Notation mit einem $-Symbol, Attribute von XML-Tags beginnen wiederum mit einem @-Symbol. Ineinander verschachtelte XML-Elemente werden in der Pfadnotation durch /-Symbole voneinander getrennt. In den Zeilen 1 und 2 werden zunächst einige Variablen definiert. Ab Zeile 5 beginnt dann der eigentliche Methodenquelltext. Da alle generierten Adaptermethoden öffentliche Methoden sind, beginnt auch der entsprechende Quelltext mit dem Schlüsselwort public. Danach folgt der Typ des Rückgabewerts der Methode, der sich aus der XML-

[3] http://www.eclipse.org/emft/projects/jet/

```
<c:setVariable select="$currentAdapterOp/@id" var="currentAdapterOpId"/>
<c:setVariable select="$adapter/mapping[@source='{$currentAdapterOpId}']
    /@target" var="currentMappingTarget"/>

public <c:get select="$currentAdapterOp/return/@type"/>
    <c:get select="$currentAdapterOp/@name"/>
    <c:include template="templates/parameterList.jet"/>
    <c:include template="templates/exceptionList.jet"/> {

    <c:setVariable select="count($adaptee
        /mapping[@target='{$currentMappingTarget}'])"
        var="currentAdapteeMappingCount"/>
    <c:setVariable select="$adaptee
        /mapping[@target='{$currentMappingTarget}']/@source"
        var="currentAdapteeOpId"/>
    <c:setVariable select="$adaptee
        /operation[@id='{$currentAdapteeOpId}']"
        var="currentAdapteeOp"/>

    <c:if test="$currentAdapterOp/return/@type != 'void'">
        return
    </c:if>
    ((<c:get select="$adaptee/@name"/>)
        this.<c:get select="$adapteeFieldName"/>)
            .<c:get select="$currentAdapteeOp/@name"/>
            <c:include template="templates/argumentListSimple.jet"/>;
    <c:iterate select="$adaptee
        /mapping[@target='{$currentMappingTarget}']
        /@source" var="adapteeMappingSource">
    </c:iterate>

}
```

Listing 5.6: JET-Template zur Generierung einer Adaptermethode basierten Dienstbeschreibung ergibt. Dort ist er im Attribut `type` des Eintrags `return` unter der aktuell zu adaptierenden Operation (Variable `$currentAdapterOp`) gespeichert. Auf die gleiche Weise wird auf den Rückgabetyp folgend der Name der Methode aus der XML-Beschreibung ermittelt und in den Quellcode eingesetzt. In Zeile 7 folgt dann die Liste der Parameter der Methode, die in einem externen Template `parameterList.jet` erzeugt wird, gefolgt von einer Liste der durch die Methode ausgelösten Ausnahmen in Zeile 8, die ebenfalls in einem externen Template `exceptionList.jet` erzeugt wird. Der Methodenkopf endet mit der geöffneten geschweiften Klammer am Ende von Zeile 8.

Nach der Generierung des Mehtodenkopfs folgt der Quelltext des Methodenrumpfs. Ab Zeile 10 werden erneut einige Variablen definiert, die für die Erzeugung des Methodenrumpfs benötigt werden. Insbesondere wird an dieser Stelle die Methode der Zielschnittstelle bestimmt, auf die die aktuell erzeugte Adaptermethode abbilden soll. Das Ergebnis des Aufrufs der Zielmethode wird dann zurückgeliefert, was ab Zeile 20 im Template spezifiziert ist. Wenn der Rückgabetyp der Adaptermethode nicht `void` ist, wird dazu in Zeile 21 das Schlüsselwort `return` verwendet. In Zeile 23 bis 26 erfolgt dann der eigentliche Aufruf der Zielmethode. Für die Parameterliste des Aufrufs wird wieder ein externes Template `argumentListSimple.jet` verwendet. Das Ende der erzeugten Methode wird durch die geschlossene geschweifte Klammer in Zeile 32 markiert.

Der durch die JET-Templates erzeugte Quellcode der Adaptermethoden wird dann durch die Methode `createAdapterClass` aus Listing 5.5 weiterverarbeitet und zu einer vollständigen Adapterklasse kompiliert. Der dynamisch erzeugte Adapter wird beim Deployment instanziiert und zwischen die entsprechenden Dienstinstanzen in die Dienstkomposition eingebunden.

5.8 Verwandte Arbeiten

Im Folgenden werden verwandte Arbeiten betrachtet, die sich mit semantischen Aspekten und der Adaption auf Basis semantischer Beschreibungen befassen. Dabei werden einige Vor- und Nachteile herausgestellt und es werden Gemeinsamkeiten und Unterschiede zu dem in diesem Kapitel vorgestellten Ansatz zur semantischen Adaption diskutiert.

Das Semantische Web

BERNERS-LEE et al. haben 2001 das sogenannte *Semantische Web* als Weiterentwicklung des von BERNERS-LEE bereits 1989 am CERN in Genf entwickelten *World Wide Web* vorgeschlagen [BHL01]. Das World Wide Web hat dem Internet zu seiner rasanten Entwicklung hin zu einem allgegenwärtigen Informations- und Kommunikationsmedium verholfen. Durch die Einführung sogenannter Hypertexte konnten Informationen auf einfache Weise zugänglich gemacht und miteinander verknüpft werden. Bis heute sind Webseiten üblicherweise darauf ausgelegt, ausschließlich von Menschen gelesen und verstanden zu werden.

Die Zielvorstellung des Semantischen Web ist es, durch Erweiterung um semantische Angaben auch eine maschinelle Verarbeitung von Webseiten zu ermöglichen. Dabei soll nicht das vorhandene World Wide Web ersetzt, sondern vielmehr eine Ergänzung geschaffen werden. Es soll erreicht werden, dass aus der Fülle an Informationen die jeweils relevanten Informationen schneller und zielsicherer gefunden werden können. Software Agenten könnten Suchaufträge automatisch bearbeiten und die gesuchten Informationen auffinden. Auch komplexere Aufgaben, die eine Verknüpfung unterschiedlicher Quellen und eine weitergehende Verarbeitung der Daten erfordern, könnten auf Basis der semantischen Beschreibungen von Software Agenten automatisiert durchgeführt werden. Dazu soll auf Techniken der Wissensrepräsentation und der künstlichen Intelligenz aufgesetzt werden.

BERNERS-LEE et al. schlagen die *Extensible Markup Language (XML)* sowie das *Resource Description Framework (RDF)* als technische Grundlagen für das Semantische Web vor. Mit Hilfe von XML können einzelne Teile einer Webseite gekennzeichnet werden, um beispielsweise eine Adressangabe zu identifizieren. Dokumente können also mit dieser Technik beliebig strukturiert werden. Um die Bedeutung der Strukturelemente festzulegen, wird RDF eingesetzt. Somit kann durch Annotationen der Typ einer Information gekennzeichnet werden. Da die Bedeutung der RDF-Beschreibungen nicht zwingenderweise einheitlich definiert ist, werden die Zusammenhänge mittels Ontologien festgelegt. Durch Ontologien können die Zusammenhänge der Begriffe definiert und mittels Inferenzregeln abgeleitet werden. Durch Softwareagenten soll es möglich werden, automatisch nach bestimmten Informationen zu suchen. Letztendlich sollen die Grenzen des virtuellen Netzes fallen und auch Dinge der physischen Welt mit URIs adressiert und mit RDF-Beschreibungen annotiert werden können. Dies entspricht der Vorstellung des Ubiquitous Computing (vgl. auch Abschnitt 2.1).

Neben dem World Wide Web und dem Semantischen Web gibt es auch Webservices und semantische Webservices. Webservices sind im Gegensatz zu den üblichen Webseiten nicht auf die Interaktion mit einem Benutzer ausgelegt, sondern auf die automatische und maschinelle Verarbeitung (siehe Abschnitt 3.2.3). Auch hier stellt sich die Frage, wie Webservices miteinander interagieren können, die unabhängig und ohne Wissen voneinander entwickelt werden. Wie im Fall von eHome-Diensten müssen hier gemeinsame Schnittstellen gegeben sein. Für Webservices bedeutet dies, dass die ausgetauschten Nachrichten interpretiert werden können müssen und dass das Verhalten der Webservices zueinander passen muss. Ein kompatibles Verhalten bedeutet wiederum, dass der Ablauf von gesendeten und empfangenen Nachrichten, d. h. das Protokoll, passen muss. Für die Beschreibung von Webservices wird die Sprache *WSDL* [Wor07] verwendet, die jedoch hauptsächlich syntaktische Informationen beinhaltet. Zur Ergänzung von Webservice-Beschreibungen durch semantische Informationen gibt es verschiedene Ansätze. Ein Beispiel dafür ist OWL-S.

OWL-S ist ein auf OWL basierender Ansatz, mit dem Eigenschaften und Funktionalitäten von Webservices semantisch beschrieben werden können, um das Finden, Verwenden, Komponieren und Überwachen bestimmter Webservices im Semantischen Web zu ermöglichen [Wor04a]. OWL-Spezifikationen setzen sich aus drei Bestandteilen zusammen, einem *Service Profile*, einem *Service Model* und einem *Service Grounding*.

Das *Service Profile* beinhaltet grundlegende Informationen über einen Dienst. Unter anderem werden darin die Schnittstellen sowie Vorbedingungen, Effekte und Attribute des Dienstes festgelegt. Das *Service Model* spezifiziert das Verhalten des Dienstes. Dazu stehen verschiedene Kontrollstrukturen zur Verfügung, z. B. für bedingte Anweisungen und Schleifen. Das *Service Grounding* macht Angaben darüber, wie der Dienst von anderen Diensten angesprochen werden kann.

Listing 5.7 zeigt ein Beispiel für ein *Process Model* in OWL-S, das das Verhalten eines Dienstes zur Personalisierung einer Umgebung beschreibt. Das Beispiel zeigt die Beschreibung eines Weckdienstes, der sich sequentiell aus mehreren Teilprozessen zusammensetzt. Die ersten beiden atomaren Prozesse sollen die

```
1 <process:CompositeProcess rdf:ID="WakeUp">
2     <process:composedOf>
3         <process:Sequence>
4             <process:components rdf:parseType="Collection">
5                 <process:AtomicProcess rdf:about="#BlindsUp"/>
6                 <process:AtomicProcess rdf:about="#BrewCoffee"/>
7                 <process:CompositeProcess rdf:about="#PlayMusic"/>
8             </process:components>
9         </process:Sequence>
10    </process:composedOf>
11 </process:CompositeProcess>
```

Listing 5.7: Beispiel eines OWL-S Service Model

Rollladen in der Umgebung hochfahren (Zeile 5) bzw. die Kaffeemaschine aktivieren (Zeile 6), der dritte Prozess ist ein zusammengesetzter Prozess, der Musik abspielen soll (Zeile 7).

Diese Prozesse beschreiben jedoch noch keine konkreten Webservices, sondern sind abstrakte Beschreibungen. Die Prozessbeschreibung ist daher auch nicht als ausführbares Programm zu verstehen, sondern stellt lediglich eine Beschreibung des Dienstverhaltens in Bezug auf andere Dienste dar, damit eine entsprechende Dienstkomposition ermöglicht wird. Dabei müssen die verschiedenen Teilprozesse nicht notwendigerweise von mehreren verschiedenen Diensten realisiert werden, es reicht ggfs. ein einzelner Dienst, sofern er den spezifizierten Ablauf realisiert.

Bei der Umsetzung des Semantischen Web und semantischer Webservices müssen ähnliche Herausforderungen adressiert werden wie bei der Entwicklung von eHome-Systemen. Das Umfeld ist heterogen, dennoch müssen Dienste möglichst interoperabel sein, damit der Ansatz sinnvoll eingesetzt werden kann.

Ein wesentlicher Unterschied zum Ansatz in dieser Arbeit ist die Verteilung, die im Web gegeben ist. In dieser Arbeit werden die eHome-Dienste auf einem zentralen Residential Gateway verwaltet und ausgeführt. Für die semantische Betrachtung hat dies allerdings prinzipiell keine Auswirkungen. Das Semantische Web ist jedoch nicht auf eine spezifische Domäne festgelegt, wie dies bei eHome-Diensten der Fall ist. Aufgrund der größeren Allgemeinheit ist ein Abgleich über semantische Beschreibungen deutlich schwerer zu erreichen, da kaum zu erwarten ist, dass man sich auf ein globales semantisches Modell in Form einer entsprechenden Ontologie wird einigen können. Daher muss das Problem der Integration von Ontologien behandelt werden, was mit weiteren Herausforderungen verbunden ist [HRK08a, HRK08b, HRK09]. Durch die Einschränkung des Anwendungsgebiets auf eHome-System in der vorliegenden Arbeit ist zu erwarten, dass die Entwicklung einer Ontologie für die semantische Dienstbeschreibung leichter zu erreichen ist.

Ein deutlicher Unterschied zwischen Webservices und den eHome-Diensten in dieser Arbeit liegt außerdem in der Art der Kommunikation. Während Webservices über das Versenden von Nachrichten miteinander kommunizieren, findet die Kommunikation hier über Methodenaufrufe statt, da die Dienste auf Basis von OSGi realisiert werden. Dies ändert jedoch nichts an der grundsätzlichen Problematik, da es für die Kommunikation unerheblich ist, ob inkompatible Schnittstellen vorliegen oder inkompatible Nachrichtenformate und -protokolle. Für eine Lösung des Problems werden im Fall von Webservices Mediatordienste benötigt, die in der Lage sind, Nachrichten des einen Dienstes in passende Nachrichten für einen anderen Dienst zu übersetzen und dabei auch mögliche Inkompatibilitäten im Protokoll zu berücksichtigen.

Im Fall der in dieser Arbeit betrachteten eHome-Dienste werden stattdessen Adapterkomponenten generiert, die zwischen unterschiedlichen Schnittstellen vermitteln und Methodenaufrufe der einen Schnittstelle auf Methodenaufrufe der anderen Schnittstelle abbilden. Aus der unterschiedlichen Realisierung der Dienste und ihrer Kommunikation ergibt sich daher auch eine unterschiedliche Realisierung der Lösung.

Das Verhalten von Diensten wird in dieser Arbeit anders als in OWL-S nicht betrachtet. Die einzelnen Operationen der Dienstschnittstellen werden semantisch annotiert, der Ablauf der Kommunikation wird jedoch nicht spezifiziert. Dies würde in Bezug auf die adressierten Basisfunktionalitäten im eHome keinen wesentlichen Zusatznutzen bieten, da es sich bei diesen Basisfunktionalitäten meist um wenig komplexe Abläufe handelt. Üblicherweise können Geräte ein- und ausgeschaltet oder in weitere Betriebszustände versetzt werden, oder es ist die stufenlose Regulierung eines bestimmten Parameters möglich. Dabei bestehen kaum Abhängigkeiten zwischen den Operationen, die in einer Verhaltensspezifikation explizit betrachtet werden müssten.

Amigo-Projekt

Das *Amigo-Projekt* [Ami08] ist ein von der Europäischen Union gefördertes Forschungsprojekt, das von 2004 bis 2008 durchgeführt wurde. Daran beteiligt waren verschieden europäische Unternehmen und Forschungsinstitutionen, unter anderem *Philips*, *France Telecom*, die *Fraunhofer-Gesellschaft*, das *Europäische Microsoft Innovations Center (EMIC)* und das *Institut National de Recherche en Informatique et en Automatique (INRIA)*. Amigo steht für *Ambient Intelligence for the Networked Home Environment*. Ziel dieses Forschungsprojekts war es, eine offene, standardisierte Architektur für *Middlewares* zu schaffen, die eine Plattform für interoperable Dienste in vernetzten Wohnumgebungen bietet [Ami04]. Bisher sind Heimautomatisierung, Unterhaltungselektronik, PCs und mobile Kommunikation voneinander getrennte Bereiche. Durch die Vernetzung dieser Bereiche soll eine Infrastruktur geschaffen werden, die eine integrierte Nutzung all dieser Bereiche ermöglicht. Der Amigo-Ansatz soll dabei den Charakter einer Referenzarchitektur haben, die Integration und Interoperabilität auf einer höheren semantischen Ebene erlaubt [GMB+05].

Zur Einbindung der eigentlichen Hardware sind in der Amigo-Architektur verschiedene Abstraktionsschichten vorgesehen, die Plattformebene, die Middlewareebene und die Anwendungsebene. Auf allen Ebenen tritt Heterogenität in Erscheinung. Die verschiedenen heterogenen Gerätestandards werden durch generische Treiber verkapselt, die dem sogenannten *Domotic Service Model* genügen müssen [GVR+07] und die Gerätefunktionalitäten als Dienste zur Verfügung stellen. Anwendungsdienste, die den Top-Level-Diensten in der vorliegenden Arbeit entsprechen, können dann auf diese zurückgreifen. Dieses Vorgehen zur Adressierung von Heterogenität ist zunächst syntaxbasiert.

Ein semantische Erweiterung der Amigo-Middleware wird in [BGI05] beschrieben. Darin wird ein Ansatz zur verhaltensbasierten Dienstkomposition eingeführt. Die Zielsetzung ist, Dienste entsprechend einer benutzerspezifischen abstrakten Verhaltensspezifikation, einer sogenannten *User Tasks*, zu komponieren. Das vom Benutzer gewünschte Verhalten soll mittels der verfügbaren Dienste realisiert werden. Dazu müssen die Verhaltensspezifikationen der Dienste mit denen der abstrakten Beschreibung abgeglichen werden. Die syntaxbasierten Mechanismen der Amigo-Middleware wurden entsprechend erweitert, sodass der spezifizierte Gesamtprozess automatisch aus den Teilprozessen der verfügbaren Dienste zusammengesetzt werden kann. So wird aus der abstrakten User Task ein konkreter Ablauf, der ausführbar ist. Dies geschieht in drei Schritten:

1. Für jeden atomaren Prozess der User Task werden zunächst semantisch passende Dienste gesucht, die diesen Prozess realisieren.
2. Wenn für alle atomaren Prozesse passende Dienste gefunden wurden, wird als nächstes der zusammengesetzte Prozess betrachtet. Dabei kann es vorkommen, dass ein gefundener Dienst gleich mehrere atomare Prozesse der User Task abdeckt. Der Gesamtablauf der User Task muss letztendlich aus den verfügbaren konkreten Dienste nachgebildet werden. Dieser Vorgang wird *Konversationsmatching* genannt.
3. Abschließend liegt ein konkreter Plan zu der User Task vor. Dieser wird durch das Aufrufen der konkreten Operationen zur Ausführung gebracht. Dabei kann es vorkommen, dass aufgrund von Kontextänderungen die Komposition neu durchgeführt werden muss.

Zur Ermittlung passender Dienste im ersten Schritt werden der Diensttyp, Ein- und Ausgabeparameter sowie Vor- und Nachbedingungen analysiert, die in einer Ontologie definiert und zur Dienstbeschreibung verwendet werden. Bei der Suche nach passenden Diensten werden die Dienstsignaturen, d. h. die angebotenen und die benötigten Dienstfunktionalitäten, miteinander verglichen, und es wird überprüft, welcher Grad von Übereinstimmung besteht. Der in [BPG+08] beschriebene Ansatz zum Vergleich der Ontologiekonzepte unterscheidet vier verschiedene Übereinstimmungsgrade, die in einer Arbeit von Paolucci et al. eingeführt wurden [PKPS02]. Die vier Übereinstimmungsgrade sind:

1. *Exact:* Die Konzepte sind äquivalent oder das benötigte Konzept ist eine direkte Unterklasse des angebotenen.
2. *Plugin:* Das angebotene Konzept umfasst das benötigte.
3. *Subsumes:* Das benötigte Konzept umfasst das angebotene.
4. *Fail:* Keines der Konzepte umfasst das jeweils andere.

Weitere Diensteigenschaften wie Qualitätsattribute können verwendete werden, um eine Rangfolge bei mehreren passenden Kandidaten zu erstellen. Für die semantische Beschreibung werden im Amigo-Projekt mehrere Ontologien verschiedener Abstraktionsebenen eingesetzt. Diese decken unterschiedliche Bereiche ab, wie z. B. Unterhaltungselektronik, Gebäudetechnik oder auch Kontext- und Qualitätsaspekte.

Für das *Konversationsmatching*, d. h. die Komposition des zusammengesetzten Prozesses, der in Form einer User Task spezifiziert ist, wird ein Algorithmus zur verhaltensbasierten Dienstkomposition verwendet [BGI07]. Für die Beschreibung der User Tasks wie auch der Dienste wird Amigo-S[4] verwendet, eine Beschreibungssprache, die auf OWL-S basiert. Amigo-S ist für die Anwendung in eHomes ausgelegt und umfasst einige entsprechende Erweiterungen, z. B. in Bezug auf Geräte und Kontextinformationen. Für das Konversationsmatching werden die Amigo-S-Beschreibungen in endliche Automaten transformiert. Die Automaten der im ersten Schritt gefundenen konkreten Dienstkandidaten werden zu einem globalen Automaten kombiniert. Der endliche Automat der abstrakten User Task wird dann durchlaufen und für jeden Schritt wird ein semantisch äquivalenter Schritt des globalen Automaten gesucht. So entsteht ein Ergebnisautomat, der die abstrakte User Task durch einen Ablauf aus konkreten Dienstfunktionalitäten nachbildet. Dieser dient als Grundlage für die eigentliche Ausführung der User Task.

Abbildung 5.22 zeigt eine Übersicht der Architektur des Amigo-Projekts. Als Grundlage dient zunächst die im unteren Teil dargestellte Infrastruktur zur Anbindung der Geräte, die unterschiedlichen Standards entsprechen können. Mehrere Abstraktionsschichten dienen dazu, die Details der konkreten Hardware zu verkapseln, sodass diese auf der Ebene der Dienste nicht mehr betrachtet werden müssen. Im oberen Teil der Abbildung ist die Dienstkomposition dargestellt. Die User Task auf der linken Seite wird in einen endlichen Automaten transformiert, der im Konversationsmatching mit den endlichen Automaten der verfügbaren, semantisch passenden Dienste abgeglichen wird. Das Ergebnis ist ein Ablaufplan aus konkreten Diensten, der auf der rechten Seite dargestellt ist.

Ein wesentlicher Unterschied zwischen der semantischen Komposition in dieser Arbeit und dem im Amigo-Projekt entwickelten Ansatz besteht in der Vorgehens-

[4] In [BGI07] wird für die Beschreibungssprache statt Amigo-S der Name COCOA-L verwendet.

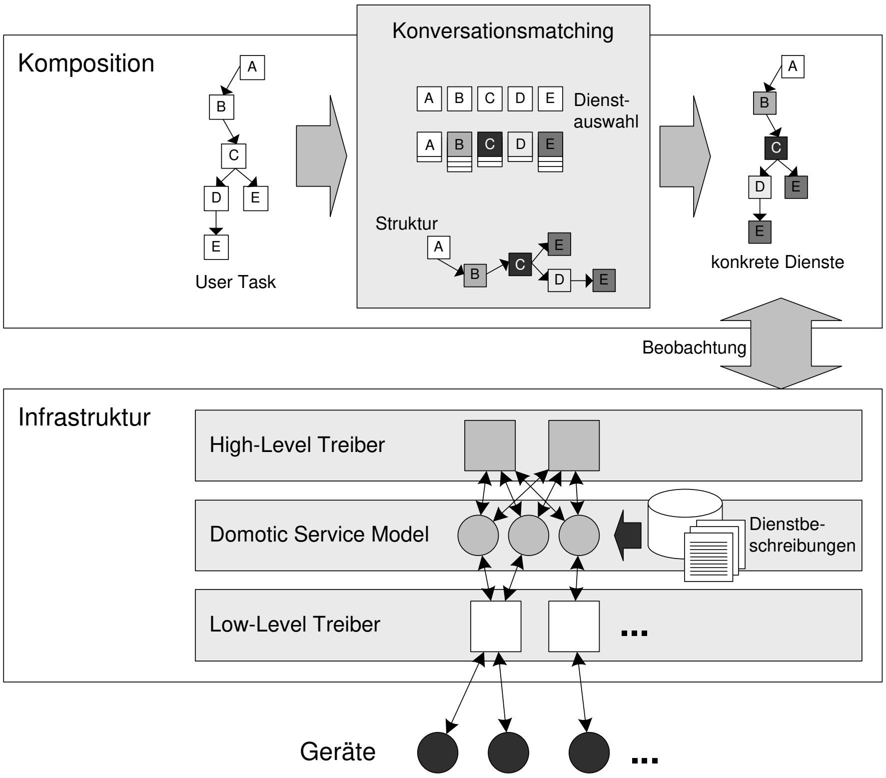


Abbildung 5.22: Übersicht der Architektur des Amigo-Projekts.

weise. In dieser Arbeit basiert die Adaption auf semantisch beschriebenen Dienstschnittstellen. Der Grund liegt darin, dass die eHome-Dienste auf Basis von Java und OSGi implementiert werden. Im Amigo-Projekt werden Dienste teils zwar auch als OSGi-Bundles oder .NET-Komponenten realisiert, für die Komposition werden diese jedoch als Webservices verkapselt und mit einer semantischen Beschreibung des Dienstverhaltens in Amigo-S versehen. Diese Beschreibung ist maßgeblich für die Komposition. Somit entfallen einige Problemstellungen dieser Arbeit, wie etwa die Generierung von Adapterkomponenten. Gemeinsam ist beiden Ansätzen jedoch die Verwendung einer semantischen Beschreibung als Grundlage der Dienstkomposition.

Ein damit zusammenhängender Unterschied besteht in der Art, wie Benutzerwünsche erfasst werden. In Amigo werden dazu die User Tasks, d. h. abstrakte Verhaltensspezifikationen, verwendet. Im Gegensatz dazu werden in dieser Arbeit vom Benutzer Top-Level-Dienste ausgewählt. Zur Erfüllung der Abhängigkeiten von Top-Level-Diensten werden dann die semantisch beschriebenen Schnittstellen verwendet. Der Nutzer kann im Ansatz dieser Arbeit nur aus den verfügbaren Top-Level-Diensten auswählen, kann also keine Wünsche spezifizieren, zu denen kein Top-Level-Dienst existiert. Allerdings können Top-Level-Dienste parametrisiert werden. So ist eine Anpassung des Dienstes an besondere Wünsche möglich, zumindest in dem Maße, in dem es vom Dienstentwickler vorgesehen wurde.

Bei der Verwendung von User Tasks stellt sich die Frage, wer diese spezifiziert. In [VRV05] wird vorgeschlagen, Bibliotheken mit vordefinierten User Tasks zu verwenden. Dann können allerdings auch nur bereits in Form von User Tasks spezifizierte Anforderungen ausgewählt werden. Dies stellt ein Problem dar, wenn z. B. Abläufe, wie der in Listing 5.7 spezifizierte Weckdienst, an die Vorstellungen des Benutzers angepasst werden sollen. Umgekehrt wird es jedoch kaum sinnvoll sein, die Spezifikation der User Tasks dem Benutzer selbst zu überlassen. Die Verwendung einer Sprache wie Amigo-S ist dazu zu kompliziert. Auch wenn die Spezifikation durch geeignete Werkzeuge unterstützt würde, wäre ein solches Vorgehen vermutlich zu zeitaufwendig und fehleranfällig, als dass der Benutzer seine Anforderungen in dieser Form spezifizieren wollte.

Somit ergibt sich durch die Verwendung von User Tasks in der Praxis kein erkennbarer Vorteil. Eher als Nachteil kann die Ausdrucksstärke der Spezifikationssprache betrachtet werden. Bei der konkreten Implementierung eines Dienstes bieten sich deutlich mehr Möglichkeiten, als die wenigen Konstrukte einer Spezifikationssprache wie Amigo-S bieten können.

Bei der semantischen Komposition können auch Inkompatibilitäten hinsichtlich der Parameter von Operationen auftreten. Dies wird in dieser Arbeit adressiert (vgl. Abschnitt 5.5) und ist für die Adaption unerlässlich. Im Amigo-Projekt werden für das Matching wie auch im hier vorgestellten Ansatz die semantischen Konzepte betrachtet, es erfolgt aber keine weitere Betrachtung möglicher Inkompatibilitäten auf syntaktischer Ebene. Diese können jedoch auftreten, da die Verhaltensspezifikationen der verfügbaren Dienste konkrete Operationen enthalten. Daher kann es nötig sein, dass eine Konvertierung zwischen unterschiedlichen Typen nötig ist. Wie der Übergang von den semantischen Konzepten der Spezifikationen auf die syntaktischen Schnittstellen der konkreten Dienste realisiert werden soll, wird in den beschriebenen Ansätzen nicht angesprochen. Dieser

Vorgang ist für die praktische Anwendung aber erforderlich und wird in dieser Arbeit durch die Adaptergenerierung adressiert.

Eine Werkzeugunterstützung zur Interaktion zur Laufzeit wie in der vorliegenden Arbeit ist in Amigo nicht vorgesehen. Der Prozess der semantischen Dienstkomposition kann nicht vom Benutzer beeinflusst werden, so ist z. B. keine interaktive Auswahl unter den passenden, durch Treiber angebundenen Geräten möglich. In [KKNP08] wird ein Werkzeug namens *VantagePoint* vorgestellt, das im Rahmen des Amigo-Projekts entwickelt wurde. Dieses Werkzeug erlaubt das Testen semantischer Beschreibungen intelligenter Umgebungen. Die semantischen Konzepte können dazu mit den Entitäten der simulierten Umgebung assoziiert werden, wodurch die semantische Modellierung unterstützt wird. Eine Benutzerinteraktion zur Laufzeit, wie sie durch den Ansatz dieser Arbeit unterstützt wird, ist damit jedoch nicht möglich. VantagePoint zielt vielmehr auf die Unterstützung bei der Dienstentwicklung ab. Dabei wird jedoch nur die semantische Ebene berücksichtigt, die Beschreibung der konkreten Dienstfunktionalität durch semantische Verhaltensbeschreibungen wird nicht explizit adressiert. Dazu wurde im Rahmen dieser Arbeit der Service-Editor entwickelt (siehe Abschnitt 6.3.1), der genau für die Aufgabe der semantischen Dienstbeschreibung entworfen wurde.

Domotic OSGi Gateway

Das *Domotic OSGi Gateway (DOG)* ist ein Forschungsprojekt an der Politecnico di Torino in Italien mit dem Ziel, ein OSGi-basiertes Gateway für die Realisierung verteilter, intelligenter Anwendungen für Wohnumgebungen zu entwickeln [BCC08b]. Diese Umgebungen werden *Intelligent Domotic Environments* genannt. Dabei sollen die verschiedenen Standards und Infrastrukturen durch DOG gebündelt werden, um einen einheitlichen Zugriff zu ermöglichen, und es sollen durch semantische Beschreibungen und semantisches Schlussfolgern komplexe Dienste realisiert werden.

Grundlage für diesen Ansatz ist *DogOnt* [BC08], eine Ontologie, die mit OWL realisiert wurde und durch semantische Schlussfolgerungen folgende Fragen beantworten soll:

- Wo befindet sich ein Gerät in der Umgebung?

- Welche Funktionalitäten bietet ein Gerät an?
- Was wird für die technische Interaktion mit dem Gerät benötigt?
- Welche Konfigurationen kann das Gerät annehmen?
- Wie ist die Umgebung strukturiert?
- Wie ist die Umgebung architektonisch aufgebaut und möbliert?

Ein Ausschnitt der Ontologie ist in Abbildung 5.23 dargestellt. Der Umgebungsbeschreibung dienen die Konzepte Building Environment und Building Thing als Grundlage. Für die Interaktion mit Geräten gibt es die Konzepte Functionality und State, die Funktionalitäten bzw. Zustände modellieren. Die beschriebenen Funktionalitäten sind solche, die direkt von den Geräten angeboten werden, z. B. die Steuerung einer Lampe oder die Regulierung der Lautstärke eines Geräts. Die Zustände stellen Attribute der Geräte dar, wie z. B. die Helligkeit einer Lampe oder die Lautstärke eines Lautsprechers.

Die Architektur von DOG ist in vier sogenannte Ringe unterteilt [BCC08a], die unterschiedliche Abstraktionsebenen repräsentieren. Die erste Ebene ist *Ring 0*, der direkt auf der OSGi Service Plattform aufsetzt und grundlegende Bibliotheken zur Verfügung stellt. Der *Ring 1* stellt Schnittstellen für die verschiedenen Infrastrukturen im Haus wie KNX oder X10 bereit. Damit können Nachrichten zwischen den DOG-Bundles und den Gateways auf Netzwerkebene ausgetauscht werden. *Ring 2* implementiert das sogenannte *House Model*, die wichtigste Komponente in DOG. Das House Model verwaltet ein Modell der Umgebung auf Basis der DogOnt-Ontologie und ermöglicht es anderen Komponenten, Anfragen zu stellen. Eine weitere Komponente verteilt Nachrichten an die unteren Ringe, damit diese an die entsprechenden Netzwerke der Umgebung verteilt werden. Schließlich gibt es noch *Ring 3*, der die Schnittstelle nach außen darstellt. Diese kann direkt über Java oder über eine Webschnittstelle mittels XML-RPC[5] angesprochen werden.

Zur Laufzeit werden die über die Schnittstellen eingehenden Befehle durch Abfragen in SPARQL[6], einer Abfragesprache für RDF, an die Ontologie validiert und dann an die tieferen Architekturebenen weitergeleitet. Um geräteübergreifende

[5] http://ws.apache.org/xmlrpc/
[6] http://www.w3.org/TR/rdf-sparql-query/

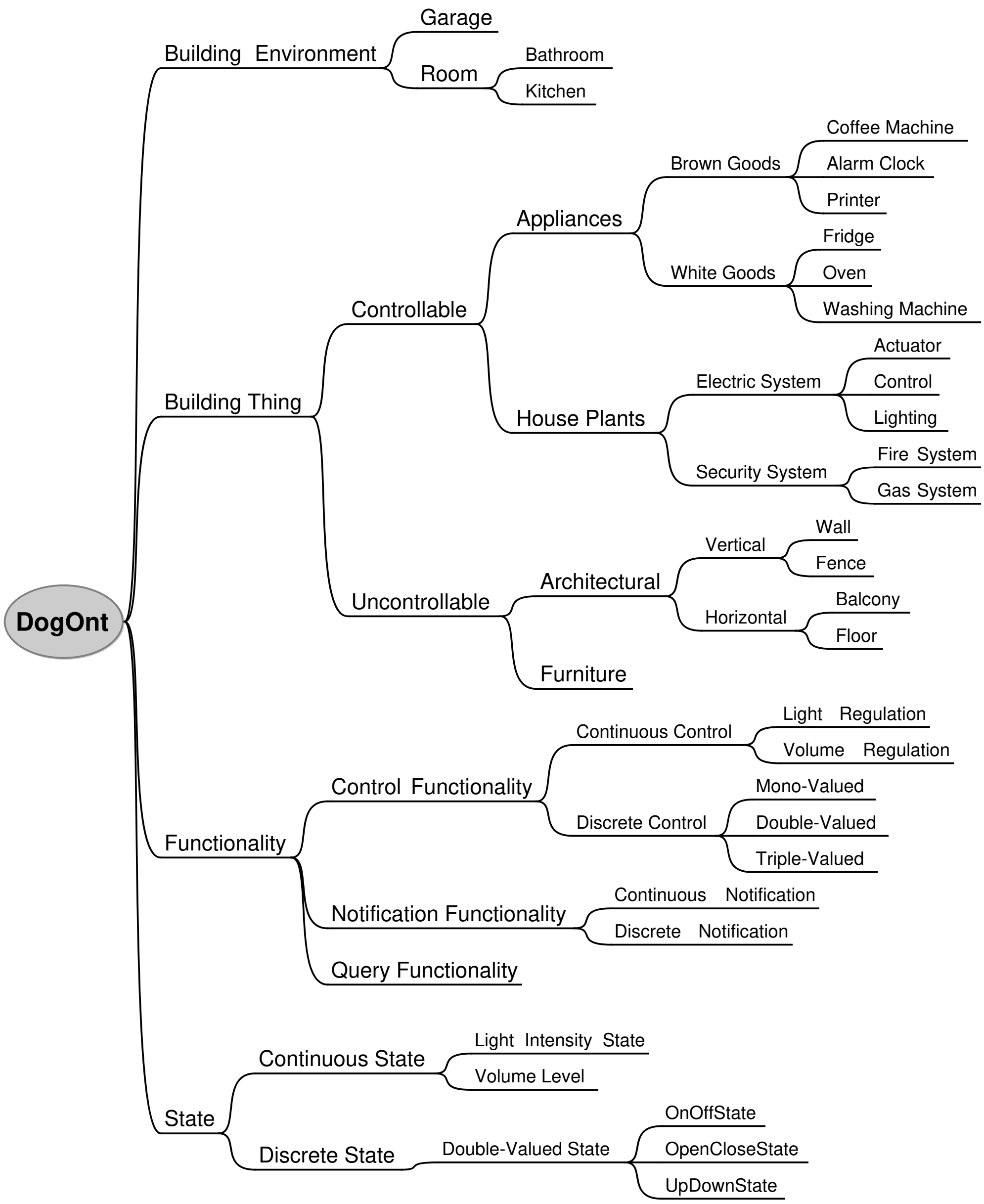


Abbildung 5.23: Ausschnitt der DogOnt-Ontologie. (in Anlehnung an [BC08])

Funktionalitäten zu realisieren, können Regeln in der Regelsprache Turtle[7] spezifiziert werden [BCC08c].

Der DOG-Ansatz ist wie auch diese Arbeit auf das spezifische Anwendungsfeld eHomes ausgerichtet. Auch in DOG geht es darum, eine Interoperabilität mittels semantischer Beschreibungen herzustellen. Dabei wird in DOG hauptsächlich die Heterogenität der Hardwareumgebung und der entsprechenden Infrastrukturen adressiert. Auch im Amigo-Projekt (siehe oben) wurde von verschiedenen Hardwarestandards abstrahiert, was durch generische Gerätetreiber für die Amigo-Middleware erreicht wurde. Bei diesem Vorgehen handelt es sich um eine übliche Abstraktionsschicht, die ohne semantische Mechanismen auskommt.

Die semantische Modellierung setzt in Amigo erst oberhalb der Hardware-Abstraktionsschicht an. In DOG werden die semantischen Konzepte jedoch genau für die semantische Beschreibung der Umgebung und der Geräte verwendet. Daher ist die Ontologie DogOnt auch speziell auf diese Aufgabe abgestimmt, was auch in dem in Abbildung 5.23 dargestellten Ausschnitt deutlich wird. Die Konzepte in DogOnt sind entsprechend konkret und orientieren sich an realen Geräten.

In der vorliegenden Arbeit wird die Integration der verschiedenen Hardwarestandards nicht betrachtet. Es wird vorausgesetzt, dass alle zur Verfügung stehenden Geräte durch entsprechende Treiberdienste eingebunden werden. Wie die Treiberdienste mit der Infrastruktur und der eigentlichen Hardware kommunizieren, ist dabei nicht Gegenstand der Arbeit.

Wie im DOG-Ansatz geht die eHome-Ontologie in dieser Arbeit zunächst von den Basisdiensten, die konkrete Gerätefunktionalitäten verkapseln, aus. Daraus ergibt sich ein ähnlicher Aufbau der Ontologie, z. B. Functionality und State in DogOnt gegenüber Capability und Attribute im hier vorgestellten Ansatz. Darüber hinaus können hier jedoch auch abstraktere Funktionalitäten hinzugefügt werden, die dann bereits innerhalb der Ontologie mittels Regeln auf konkretere Funktionalitäten abgebildet werden.

Die Abbildung der DogOnt-Konzepte auf die syntaktische Ebene wird in den spezifischen Netzwerktreibern in Ring 1 der Architektur vorgenommen. Wie dies

[7] `http://www.w3.org/TeamSubmission/turtle/`

genau geschieht, ist in der Literatur nicht beschrieben. Je nachdem, welche Infrastruktur unterstützt werden soll, muss hier ein Entwickler die entsprechenden Abbildungsmechanismen implementieren. Eine Werkzeugunterstützung, wie der Service-Editor in dieser Arbeit (siehe Abschnitt 6.3.1), ist für diese Aufgabe nicht verfügbar.

Eine automatische Generierung von Adaptern, wie sie in dieser Arbeit vorgestellt wurde, ist daher ebenfalls nicht vorgesehen. Die Transformation muss in den Netzwerktreibern stattfinden und ist dementsprechend von einem Entwickler für jede zu unterstützende Infrastruktur zunächst manuell durchzuführen. Allerdings muss in DOG auch nicht zwischen unterschiedlichen syntaktischen Schnittstellen transformiert werden, sondern ausschließlich von der semantischen Ebene auf die syntaktische Ebene, z. B. um die in Turtle spezifizierten geräteübergreifenden Funktionalitäten zu realisieren.

5.9 Zusammenfassung

In diesem Kapitel wurde ein Ansatz zur semantischen Adaption von eHome-Systemen vorgestellt. Das Ziel der semantischen Adaption ist die Unterstützung einer flexiblen Dienstkomposition in einem heterogenen Umfeld, wie es in eHomes gegeben ist. Dazu wurde als Grundlage der Komposition eine semantische Dienstbeschreibung auf Basis einer domänenspezifischen Ontologie eingeführt. Auf diese Weise können syntaktische Inkompatibilitäten überbrückt werden.

Zunächst wurde das Problem der Heterogenität in eHomes erläutert sowie die sich daraus ergebenden Herausforderungen in Bezug auf die Dienstkomposition. Als Grundlage für den vorgestellten Lösungsansatz wurden Konzepte der semantischen Modellierung, Ontologien und verschiedene Adaptionsarten eingeführt. Auf dieser Basis wurde der semantische Teilprozess, der die semantische Adaption umsetzt, und seine Einbettung in den kontinuierlichen SCD-Prozess beschrieben. Die vier Phasen des semantischen Teilprozesses sind die *Semantikdefinition* in Form einer eHome-Ontologie, die *semantische Abbildung* zur Dienstbeschreibung, das *semantische Matching* auf Basis der Dienstbeschreibungen und die *Adaptierung*, die mittels automatisch generierter Adapter eine Komposition heterogener Dienste ermöglicht. Diese Phasen wurde im Detail erläutert.

Abschließend wurden die in der vorliegenden Arbeit entwickelten semantischen Konzepte mit einigen verwandten Arbeiten verglichen. Dabei wurden die wesentlichen Unterschiede herausgearbeitet und diskutiert. Häufig ergaben sich Unterschiede aus der Art der Dienstimplementierung und ihrer Kommunikation. In einigen Arbeiten wird das Verhalten von Diensten in Form von Kommunikationsprotokollen betrachtet. Das Dienstverhalten wird in der vorliegenden Arbeit nicht in die semantischen Betrachtungen einbezogen. Stattdessen steht die Generierung von Adaptern im Vordergrund, die es ermöglichen, syntaktisch inkompatible Dienste komponieren zu können. Eine Adaptierung syntaktischer Inkonsistenzen sowie die Unterstützung semantischer Beschreibungen durch Werkzeuge wird in den verwandten Arbeiten nicht vorgestellt.

Kapitel 6

Realisierung

Dieses Kapitel bietet eine Übersicht der im Rahmen dieser Arbeit entstandenen Werkzeuge. Zunächst wird eine Übersicht des prototypischen Gesamtsystems gegeben. Im nächsten Abschnitt folgt eine kurze Erläuterung einiger technischer Grundlagen, die für die Entwicklung der Werkzeuge von Bedeutung sind. Im Hauptteil des Kapitels werden alle Werkzeuge, die zur Unterstützung der vorgestellten Ansätze benötigt werden, im Detail beschrieben, gefolgt von einer Beschreibung der wesentlichen Aspekte, die für die Implementierung von Diensten für den Prototypen relevant sind. Die Anwendbarkeit der entwickelten Werkzeuge wurde in verschiedenen Testumgebungen analysiert. Nach einer kurzen Beschreibung dieser Testumgebungen folgt schließlich eine Übersicht über den Umfang der Implementierung.

6.1 Systemübersicht

Abbildung 6.1 zeigt eine konzeptuelle Übersicht eines eHome-Systems, wie es den Überlegungen dieser Arbeit zu Grunde liegt. Einzelne Elemente der Abbildung finden sich auch in Abbildung 4.1 wieder, wenngleich hier die Schichtung anders dargestellt ist. Auf der obersten Ebene ist die Benutzerschnittstelle des eHome-Systems, die durch interaktive Werkzeuge zur Administration und Steuerung realisiert wird, dargestellt. Alle entwickelten Werkzeuge basieren technisch

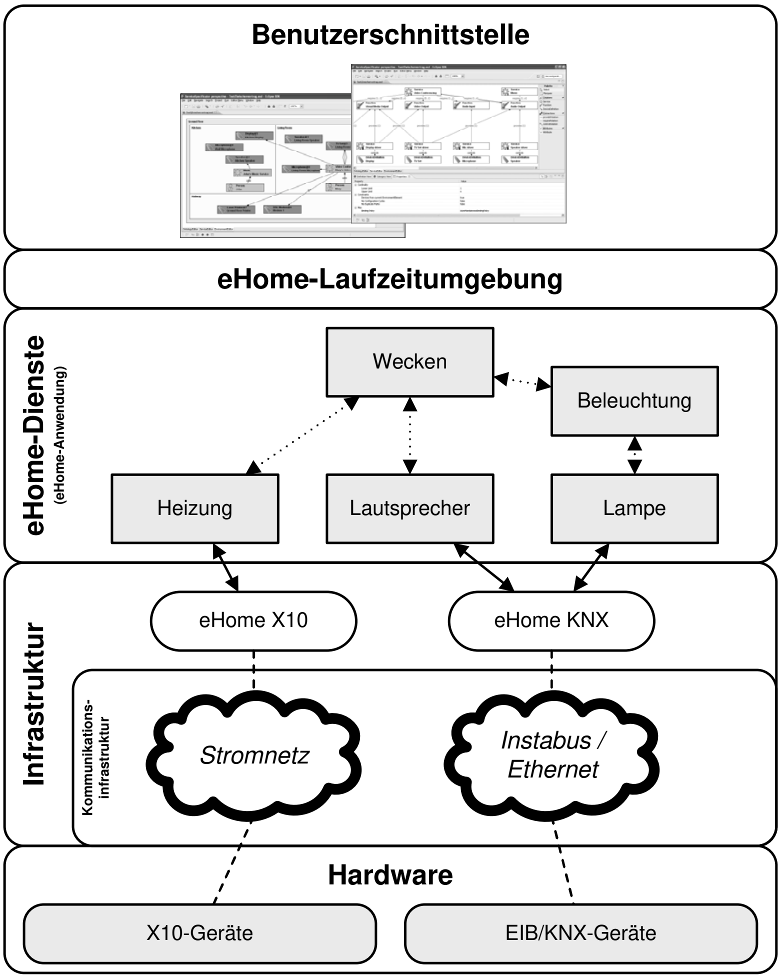


Abbildung 6.1: Übersicht des realisierten Systems.

auf der Eclipse Rich Client Platform, die in Abschnitt 6.2 beschrieben wird. Die Werkzeuge dienen der Steuerung der eHome-Laufzeitumgebung, die darunter dargestellt ist und für Konfigurierung, Deployment und Laufzeitmanagement der eHome-Dienste zuständig ist. Dies geschieht durch Ausführung des SCD-Prozesses. Auf dieser Ebene wird auch das zentrale graphbasierte Datenmodell – das *eHome-Modell* – verwaltet, welches alle zur Ausführung des SCD-Prozesses relevanten Informationen umfasst. Die deployten Instanzen der eHome-Dienste sind in der nächsten Ebene dargestellt. Zuvor müssen die Implementierungen aller im eHome genutzten Dienste, die in Form von OSGi-Bundles vorliegen, geladen werden. Dann kann die Laufzeitumgebung über die jeweilige eHome

Service Factory die benötigten Instanzen erzeugen (vgl. Abschnitt 4.4.3). Die Dienstabhängigkeiten werden durch Bindungen zwischen passenden Dienstinstanzen aufgelöst. Dies wird nicht durch die Dienste selbst sondern von der Laufzeitumgebung unter Anwendung des Entwurfsmusters Dependency Injection vorgenommen. Unterhalb der Dienstebene befindet sich die Ebene der Infrastruktur, die für die Anbindung der vorhandenen Hardware im eHome zuständig ist. Die Anbindung dieser Infrastruktur durch Treiberdienste, die die Funktionalitäten der einzelnen Geräte im eHome-System verfügbar machen, erfolgt hier über verschiedene Hilfsdienste. So können z. B. unterschiedliche Kommunikationsinfrastrukturen, wie etwa das Stromnetz für X10-Geräte oder Ethernet für EIB/KNX-Geräte (siehe auch Abschnitt 2.3), durch solche Hilfsdienste verkapselt werden. Die eigentlichen Geräte werden dann über Signale, die über die Kommunikationsinfrastruktur übermittelt werden, angesteuert.

6.2 Technische Grundlagen

Eine wesentliche technische Grundlage der entwickelten Werkzeuge stellt die Softwareplattform *Eclipse* dar [dRW04]. Eclipse ist eine integrierte Entwicklungsumgebung (engl. *Integrated Development Environment, IDE*), die aus der Entwicklungsumgebung *VisualAge for Java* von IBM hervorgegangen ist, nachdem der Quellcode von IBM freigegeben wurde. Das frühere Plug-In-System von Eclipse wurde in der Version 3.0 durch die eigens entwickelte OSGi-Implementierung *Eclipse Equinox* ersetzt.

Durch Einführung des OSGi-basierten Plug-In-Systems in Eclipse wurde zum einen OSGi stärker verbreitet und zum anderen entwickelte sich Eclipse von einer reinen Entwicklungsumgebung zu einer flexiblen Plattform, mit der sich verschiedenste graphische Editorwerkzeuge entwickeln lassen. Diese Plattform wird daher *Eclipse Rich Client Platform (RCP)* genannt [ML05]. Auf Basis der RCP lassen sich durch Hinzufügen von Plug-Ins (d. h. OSGi-Komponenten für Equinox) auf einfache Weise komplexe Anwendungen erstellen. Dazu stehen einerseits die zahlreichen bereits für Eclipse vorhandenen Plug-Ins zur Verfügung, andererseits können neue, anwendungsspezifische Plug-Ins entwickelt und hinzugefügt werden. Damit wird die einfache Wiederverwendung bestehender Komponenten ermöglicht, was den Entwicklungsaufwand einer RCP-Anwendung verringert.

Zur Realisierung der graphischen Benutzeroberfläche kommt in Eclipse nicht die *Swing*-Bibliothek aus den *Java Foundation Classes (JFC)* von Sun Microsystems zum Einsatz, sondern das *Standard Widget Toolkit (SWT)*, das von IBM eigens für Eclipse entwickelt wurde [SHNM04]. Im Unterschied zu Swing, dessen grafische Komponenten direkt von Java dargestellt werden, basiert SWT auf den nativen Grafikelementen des jeweils zugrunde liegenden Betriebssystems. Dadurch ist eine performantere Darstellung möglich, allerdings zum Preis einer verminderten Plattformunabhängigkeit, da für jedes Betriebssystem eine jeweils passende SWT-Bibliothek erforderlich ist. Um die Entwicklung grafischer Editoren auf Basis der Eclipse RCP zu erleichtern, bietet Eclipse das *Graphical Editing Framework (GEF)* an [MDG+04, EEHT05]. Dieses kann als Plug-In zur Eclipse RCP hinzugefügt werden und unterstützt die Entwicklung grafischer Editoren auf Basis beliebiger abstrakter Modelle. Dabei kommt das Entwurfsmuster *Model-View-Controller (MVC)* [BMR+98, FFSB04, Ree03] zur Anwendung. Das MVC-Muster ist ein Architekturmuster, das für interaktive Anwendungen eingesetzt wird und die Bereiche Datenmodell, Darstellung und Steuerung voneinander separiert. Die Aufteilung in diese drei Komponenten vereinfacht die spätere Wartung und erlaubt außerdem das einfache Austauschen einzelner Komponenten. So können z. B. auch mehrere Darstellungen derselben Daten realisiert werden.

In Abbildung 6.2 ist die Anwendung des MVC-Musters im Zusammenhang mit Eclipse GEF und einem Datenmodell in Fujaba dargestellt. Die Controller-Komponenten werden in Eclipse durch sogenannte *EditParts* realisiert. Sie stellen die Verbindung zwischen den Darstellungselementen, für die SWT verwendet wird, und den zugehörigen Modellelementen her. Damit durch Benutzeraktionen im Editor Veränderungen ausgelöst werden können, werden zu den EditParts verschiedene *Policies* definiert, die eine Reaktion auf die Events, die durch die Benutzerschnittstelle ausgelöst werden, implementieren. Die Policies erstellen dann sogenannte *Command*-Objekte, die von GEF weiterverarbeitet werden. Das Ausführen der Commands kann wiederum zu Änderungen im Modell führen. Änderungen im Modell führen dann zu weiteren Events, die eine Benachrichtigung der zugehörigen EditParts bewirken, sodass alle dem Modellelement zugehörigen Darstellungen aktualisiert werden können.

Das Modell wird nicht in GEF selbst verwaltet, es können vielmehr beliebige externe Modelle angebunden werden. Diese können mit dem *Eclipse Modeling Framework (EMF)* erstellt werden, ebenso ist aber auch die Anbindung z. B. eines

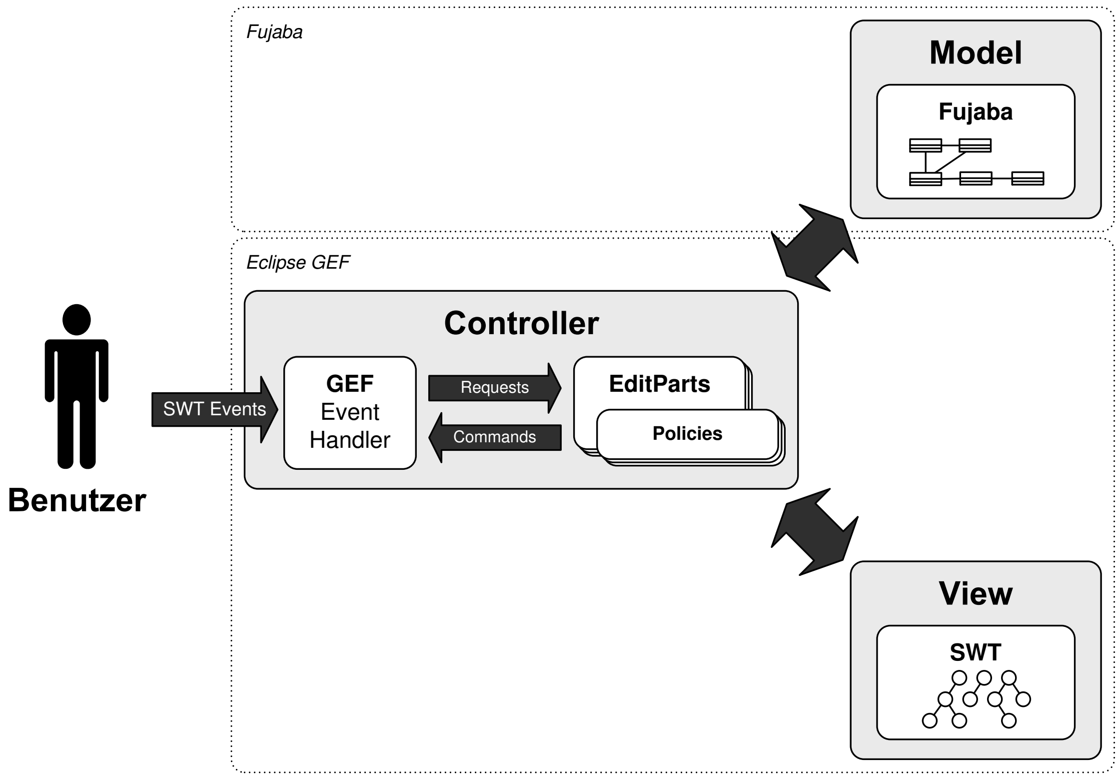


Abbildung 6.2: Anwendung des MVC-Musters mit GEF und Fujaba.

Fujaba-Modells möglich, wie es auch in den hier vorgestellten Werkzeugen der Fall ist. Fujaba und GEF werden bereits in verschiedenen anderen Projekten in Kombination eingesetzt. In [BD06] wird z. B. ein graphisches Werkzeug zum Management dynamischer Aufgabennetze beschrieben. Auch in Projekten des Lehrstuhls für Informatik 3 wurde schon ein solcher Ansatz verfolgt [Mos09]. Damit das Fujaba-Modell aus dem Werkzeug heraus angebunden werden kann, müssen die Modellelemente „beobachtbar“ sein. Nur dann können die EditParts, die Controller-Elemente in GEF, auf Veränderungen des Modells reagieren. Dazu werden in [Mos09] die Templates von Fujabas *CodeGen2*, das für die Codegenerierung zuständig ist, modifiziert. In dieser Arbeit wird stattdessen *CoObRA* (**Co**ncurrent **Ob**ject **R**eplication fr**A**mework) [SZN04, Sch03] verwendet, ein Rahmenwerk zur Unterstützung von Versionierung und Persistenz objektorientierter Datenstrukturen. Es wird von Fujaba selbst verwendet und kann auf einfache Weise auch in mit Fujaba entwickelte Anwendungen integriert werden. Durch den Einsatz von CoObRA können die generierten Klassen des Datenmodells entspre-

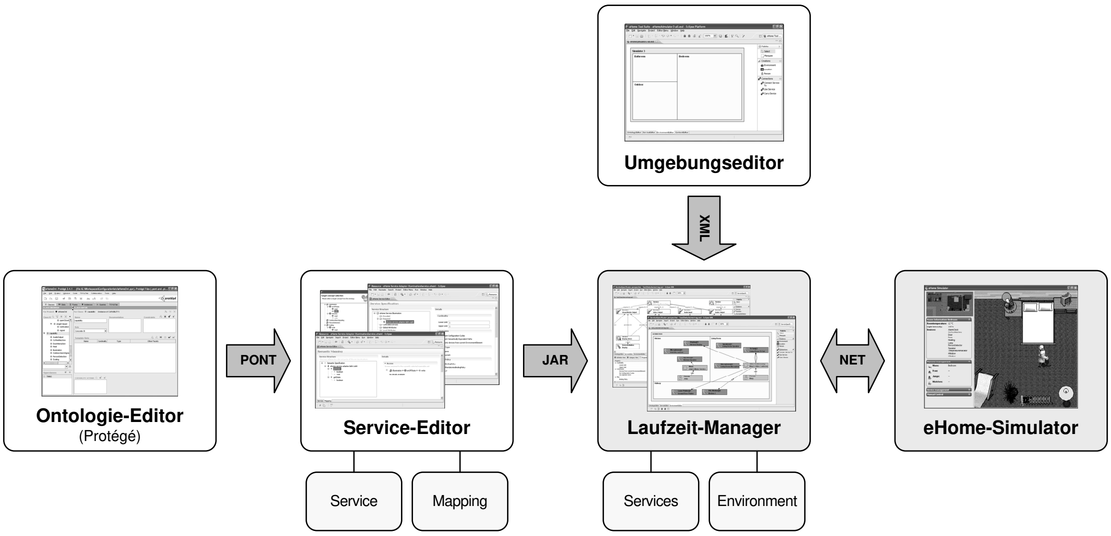


Abbildung 6.3: Zusammenspiel der entwickelten Werkzeuge.

chend der *JavaBeans*-Spezifikation [Ham97] mittels des Listener/Observer-Entwurfsmusters [GHJV95] überwacht werden. So kann das Modell zusammen mit GEF verwendet werden, ohne dass in den Mechanismus der Codegenerierung eingegriffen werden muss.

Die OSGi-Implementierung Eclipse Equinox wurde bereits anfangs dieses Abschnitts erwähnt. Sie dient seit Eclipse 3.0 als Grundlage aller Eclipse Plug-Ins. In dieser Arbeit wird Equinox auch zur Realisierung der eHome-Dienste verwendet, die als OSGi-Bundles implementiert werden. Darüber hinaus wurde Equinox auch für die Entwicklung des Service-Editors (siehe Abschnitt 6.3.1) eingesetzt, der als Plug-In in die Eclipse Entwicklungsumgebung integriert werden kann und so schon während der Dienstentwicklung zur Spezifikation der Dienste zur Verfügung steht.

6.3 Werkzeugunterstützung

In diesem Abschnitt werden die Werkzeuge beschrieben, die zur Umsetzung des kontinuierlichen SCD-Prozesses (vgl. Abschnitt 4.2) verwendet werden. Bis auf den Ontologie-Editor wurden alle Werkzeuge speziell für die jeweiligen Aufgaben im Rahmen dieser Arbeit entwickelt. Abbildung 6.3 zeigt eine Übersicht der verschiedenen Werkzeuge und wie sie miteinander zusammenhängen.

Als *Ontologie-Editor* wurde *Protégé*, ein verbreiteter Editor zur Ontologieentwicklung, eingesetzt (vgl. auch Abschnitt 5.4.3). Die mit Protégé erstellte framebasierte eHome-Ontologie wird, wie in Abschnitt 5.4 beschrieben, für die semantische Dienstspezifikation benötigt. Sie wird von Protégé in einer Datei mit der Endung `.pont` gespeichert, die vom Service-Editor, der für die Erstellung von Dienstspezifikationen entwickelt wurde, eingelesen wird. Auf diese Weise stehen die Konzepte der Ontologie für die semantische Beschreibung der eHome-Dienste zur Verfügung.

6.3.1 Service-Editor

Der *Service-Editor* wurde als Werkzeug zur Dienstspezifikation entwickelt. Diese setzt sich aus den Beschreibungen für den kontinuierlichen SCD-Prozess und aus der semantischen Dienstbeschreibung zusammen. Die Spezifikation, die für den SCD-Prozess benötigt wird, wurde in Abschnitt 4.3 beschrieben. Dabei handelt es sich um Angaben zum Diensttyp, zu den angebotenen und benötigten Funktionalitäten, zur Bindungsstrategie, zu den Bindungstypen und die Auswahl der anzuwendenden Bindungsbeschränkungen. Zur semantischen Dienstbeschreibung werden die Schnittstellen der angebotenen und benötigten Funktionalitäten auf die mit dem Ontologie-Editor erstellte eHome-Ontologie abgebildet. Der konzeptuelle Hintergrund dieser Abbildung wurde in Abschnitt 5.5 erläutert.

Technisch wurde der Service-Editor als Plug-In für die Eclipse-Entwicklungsumgebung realisiert und kann daher zusammen mit Eclipse gestartet und ausgeführt werden. Dadurch ist es auf einfache Weise möglich, schon bei der Implementierung der Dienste auch die Spezifikation zu erstellen. Eine nachträgliche Spezifikation einer bereits bestehenden Implementation ist jedoch ebenfalls möglich. Die Dienste werden als OSGi-Bundles exportiert, die in Form von Java-Archiven mit der Dateiendung `.jar` gespeichert werden. Die mit dem Service-Editor erstellten Spezifikationen werden in ein XML-Format exportiert, das in einer Datei mit der Endung `.ehxml` gespeichert und dem Java-Archiv hinzugefügt wird. Implementation und Spezifikation sind also gemeinsam in einer `.jar`-Datei

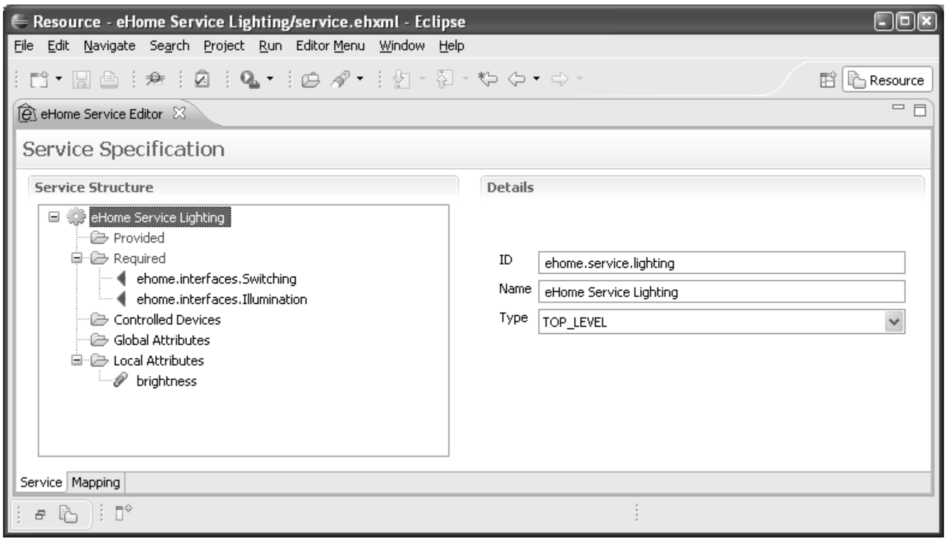


Abbildung 6.4: Bearbeiten einer Dienstspezifikation im Service-Editor.

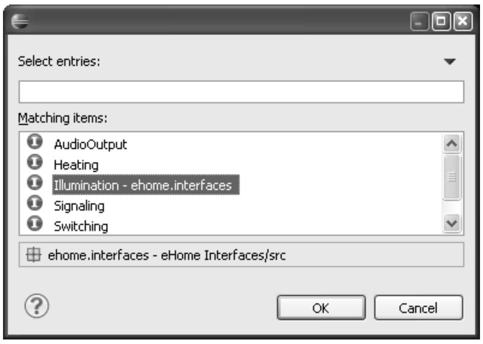


Abbildung 6.5: Auswahl einer Schnittstelle.

verkapselt. Diese stellt den für das Deployment vorbereiteten eHome-Dienst zur Verfügung, der dann im Laufzeit-Manager geladen werden kann.

Das Erstellen einer neuen Dienstspezifikation erfolgt über einen entsprechenden Eclipse-Wizard. Dieser ermöglicht es, eHome-Dienst-Projekten eine neue `.ehxml`-Datei zur Spezifikation des Dienstes hinzuzufügen. Zur Bearbeitung der Spezifikation wird dann der Service-Editor geöffnet. Wie die Hauptansicht des Editors aussieht, ist in Abbildung 6.4 zu sehen. Im dargestellten Beispiel wird der Lichtdienst eHome Service Lighting spezifiziert. Dieser ist ein Top-Level-Dienst und bietet daher anderen Diensten keine Funktionalitäten an. Der Abschnitt

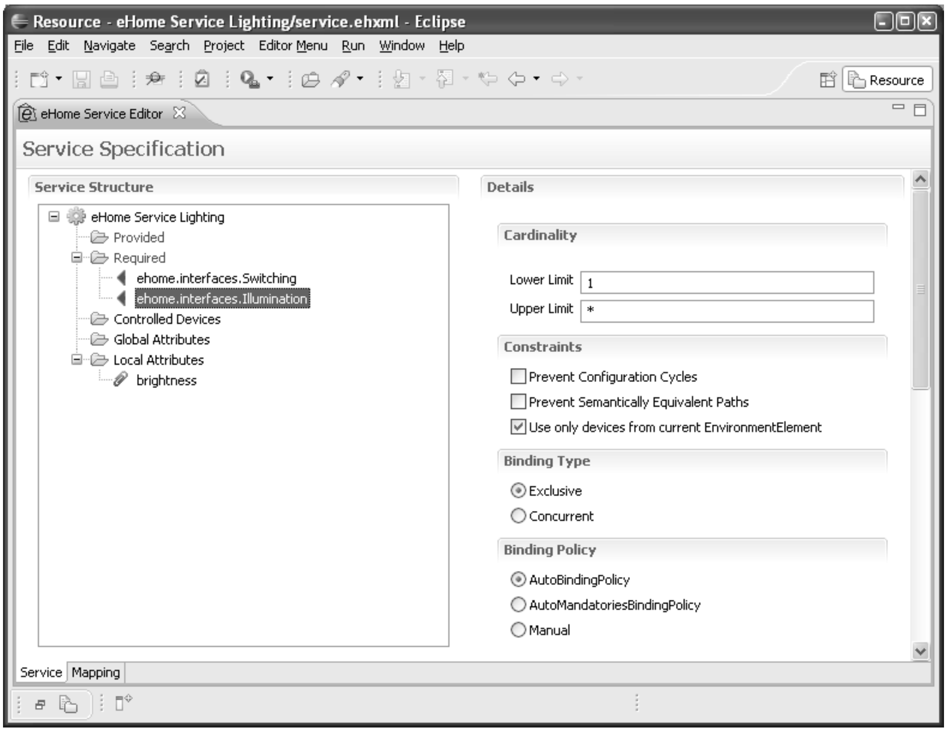


Abbildung 6.6: Spezifikation einer benötigten Funktionalität.

Provided auf der linken Seite der Abbildung ist aus diesem Grund leer. Unter dem Abschnitt Required finden sich die benötigten Funktionalitäten des Dienstes. Dies sind im Beispiel die zwei Schnittstellen `ehome.interfaces.Switching` und `ehome.interfaces.Illumination`, die zuvor über den Dialog aus Abbildung 6.5 ausgewählt wurden. Die Switching-Funktionalität wird vom Lichtdienst benötigt, damit das Licht über einen Schalter ein- und ausgeschaltet werden kann. Über die Illumination-Funktionalität werden die Lampen angebunden, die die eigentliche Beleuchtung realisieren. Zusätzlich kann der Lichtdienst noch über ein lokales Attribut brightness parametrisiert werden. Das Attribut erlaubt es, die Lichthelligkeit für dimmbare Lampen zu justieren.

Die einzelnen angebotenen und benötigten Funktionalitäten können ebenfalls näher spezifiziert werden. In Abbildung 6.6 ist zu sehen, welche Angaben für die benötigte Funktionalität Illumination gemacht werden. Auf der rechten Seite des Editors sind die Details zu sehen. Unter der Rubrik *Cardinality* werden die Kardinalitätsgrenzen eingegeben. In diesem Fall ist 1 bis * angegeben, es muss

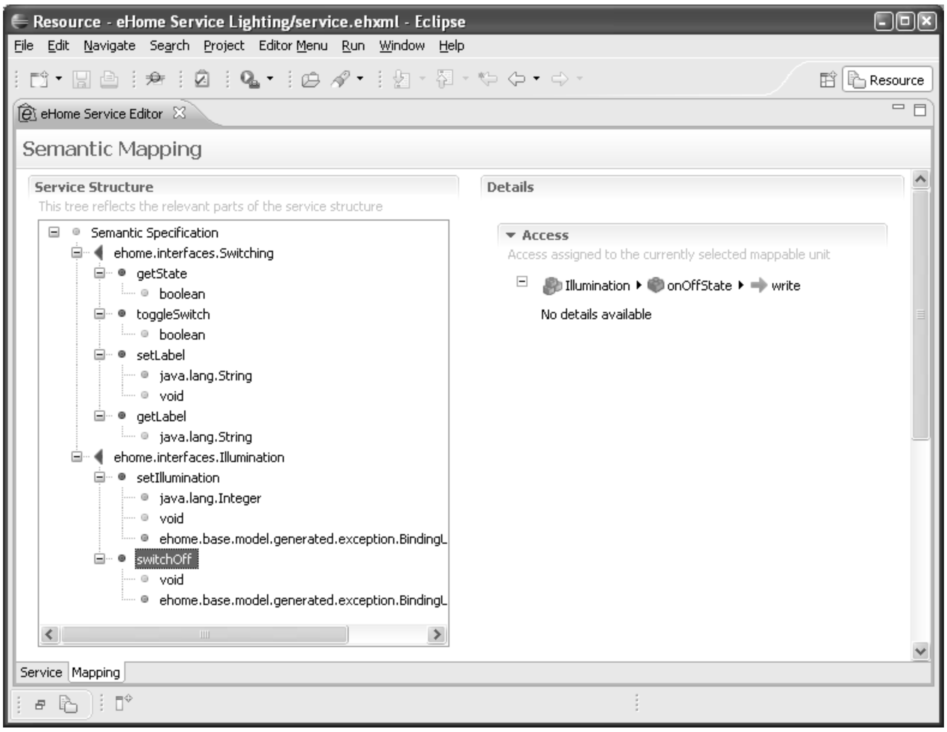


Abbildung 6.7: Spezifikation der semantischen Abbildung.

also mindestens ein Anbieter der Beleuchtungsfunktionalität verfügbar sein, es können aber auch beliebig viele zusätzlich gebunden werden. Unter *Constraints* können die vordefinierten Bindungsbeschränkungen ausgewählt werden. In diesem Fall wurde ausgewählt, dass für die Bindung der Beleuchtungsfunktionalität nur lokale Ressourcen verwendet werden sollen. In der Rubrik *Binding Type* wird zwischen exklusiven und nebenläufigen Bindungen unterschieden. Im Beispiel wurde eine exklusive Bindung festgelegt. Abschließend wird unter *Binding Policy* noch die anzuwendende Bindungsstrategie festgelegt. Für die Funktionalität Illumination wurde hier die automatische Bindungsstrategie gewählt, sodass entsprechende Ressourcen automatisch gebunden werden, sobald sie in der Umgebung zur Verfügung stehen.

Für die semantische Spezifikation des Dienstes kann von der *Service*-Seite des Editors auf die *Mapping*-Seite umgeschaltet werden. Dazu dienen die Reiter am unteren Ende des Editorfensters. In der *Mapping*-Seite werden die Abbildungen zwischen den syntaktischen Elementen der Dienstschnittstellen und den seman-

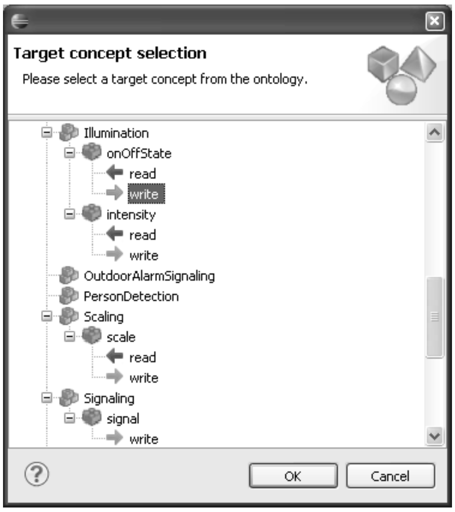


Abbildung 6.8: Auswahl einer Capability aus der eHome-Ontologie.

tischen Konzepten der Ontologie bearbeitet. In Abbildung 6.7 ist dieser Teil des Editors dargestellt. Auf der linken Seite sind die zuvor ausgewählten Schnittstellen des Dienstes zu sehen. Wenn diese markiert werden, erscheinen auf der rechten Seite die Details der Abbildung. Im dargestellten Beispiel ist die Abbildung für die Methode `switchOff` der Schnittstelle `ehome.interfaces.Illumination` angezeigt. Diese wird auf den Access write des Attribute onOffState der Capability Illumination abgebildet. Die anderen Operationen der Dienstschnittstellen können auf dieselbe Weise spezifiziert werden. Das Annotieren von Operationen mit Tags aus der Ontologie ist ebenfalls möglich. Abbildung 6.8 zeigt den Dialog, in dem die Konzepte der eHome-Ontologie ausgewählt werden können. Entsprechend der Darstellungsontologie werden die Konzepte hierarchisch in einer Baumstruktur angezeigt und können vom Benutzer selektiert werden.

6.3.2 Umgebungseditor

Für die Ausführung des kontinuierlichen SCD-Prozesses wird neben den spezifizierten Diensten auch eine Umgebungsspezifikation benötigt (vgl. auch Abbildung 4.2). Diese wird mit dem *Umgebungseditor* erstellt, der es erlaubt, die physikalische Struktur der eHome-Umgebung zu spezifizieren. Abbildung 6.9 zeigt die Oberfläche des Umgebungseditors. Darin kann der Grundriss des eHomes

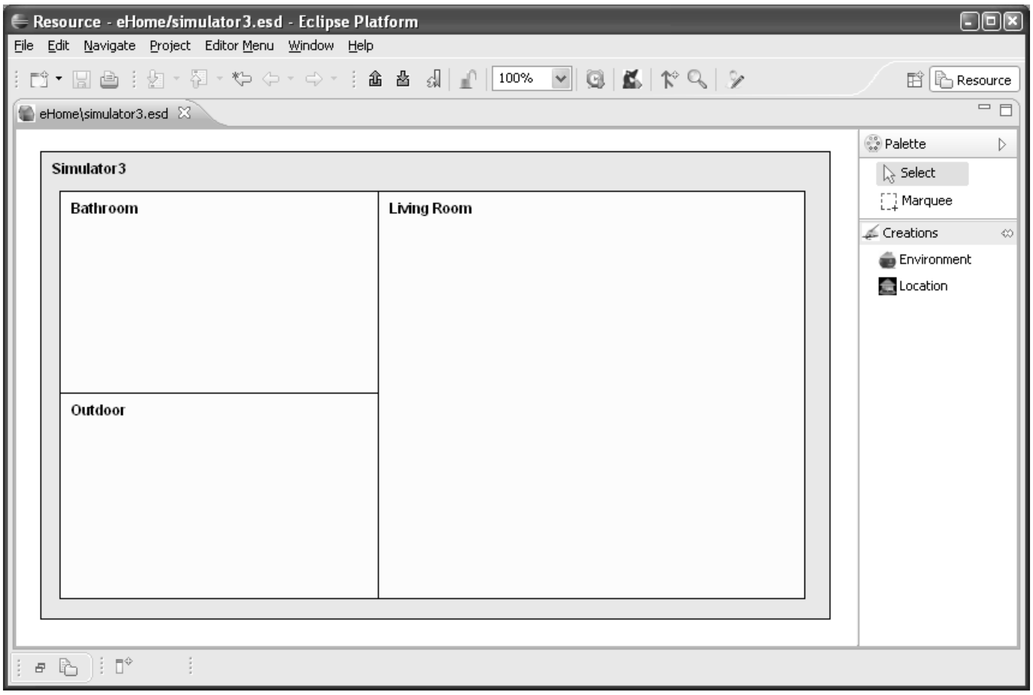


Abbildung 6.9: Hauptfenster des Umgebungseditors.

festgelegt werden. Dieser dient als Grundlage für den räumlichen Kontext von Personen, Geräten und Diensten in der Umgebung. Auf der rechten Seite des Editors ist eine Palette zu sehen. Der Benutzer kann aus dieser Palette Elemente selektieren und diese dann auf der Arbeitsfläche des Editors instanziieren. Für die Zwecke dieser Arbeit werden zunächst nur Umgebungen und Räume verwendet. Die Umgebungsspezifikation wird wie auch die Dienstspezifikation in einem XML-Format gespeichert und kann so vom Laufzeit-Manager eingelesen werden. Dieser ermöglicht die Visualisierung und die Interaktion zur Laufzeit.

6.3.3 Laufzeit-Manager

Der *Laufzeit-Manager* ist das Werkzeug, das für die Interaktion mit dem eHome-System zur Laufzeit verwendet wird. Im Hauptfenster wird die zuvor eingelesene Umgebungsspezifikation dargestellt. Diese stellt die statischen Kontextinformationen dar. Die als Java-Archiv exportierten eHome-Dienste können nun geladen werden und stehen dann für die Verwendung im eHome zur Verfügung.

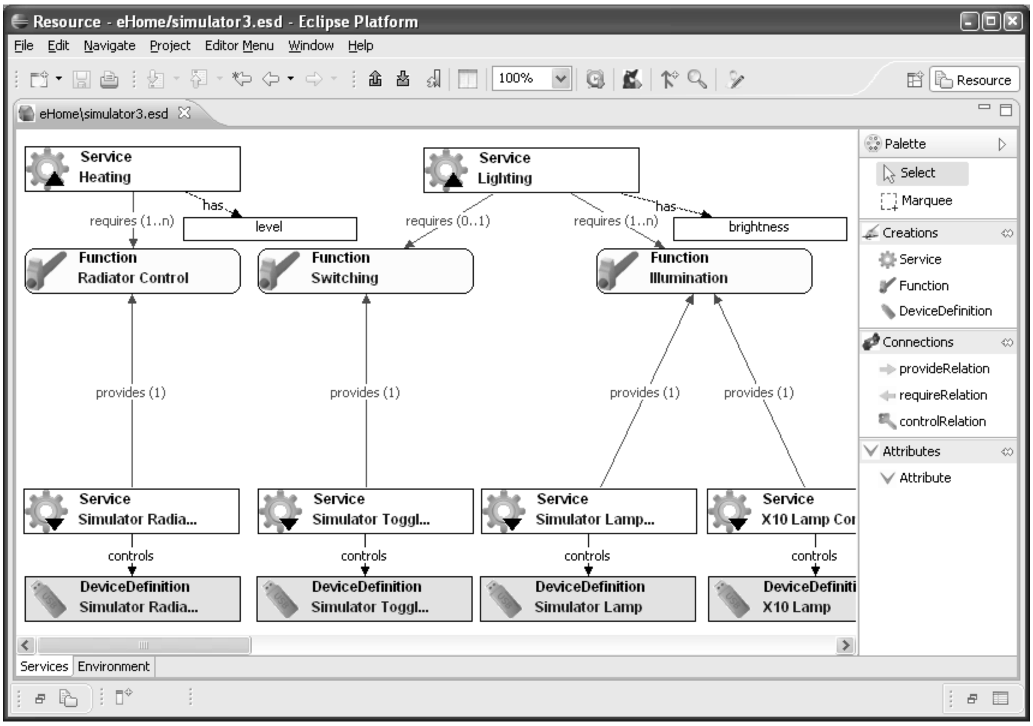


Abbildung 6.10: Dienstübersicht im Laufzeit-Manager.

Beim Laden der Dienste werden deren Spezifikationen eingelesen, und alle benötigten Daten über den Dienst werden im eHome-Modell erfasst.

Die *Services*-Seite des Laufzeit-Managers ist in Abbildung 6.10 dargestellt und bietet eine Übersicht aller geladenen Dienste und ihrer Abhängigkeiten. In der Abbildung ist z. B. unter anderem der Dienst Lighting zu sehen, der wie oben beschrieben die Funktionalitäten Switching und Illumination benötigt. In der dargestellten Konstellation wird die Funktionalität Switching durch den Dienst Simulator Toggle Switch Control angeboten, Illumination wird durch Simulator Lamp Control und X10 Lamp Control angeboten. Die *Services*-Ansicht dient der Übersicht über die geladenen Dienste und deren Abhängigkeiten. Die Bearbeitung der Dienstbeschreibungen wird mit dem oben beschriebenen Service-Editor durchgeführt.

Die wichtigste Ansicht des Laufzeit-Managers ist die in Abbildung 6.11 dargestellte *Environment*-Seite. Diese ermöglicht das Spezifizieren von Personen und Geräten, sowie das Instanziieren von Diensten. Dabei können die Elemente mit-

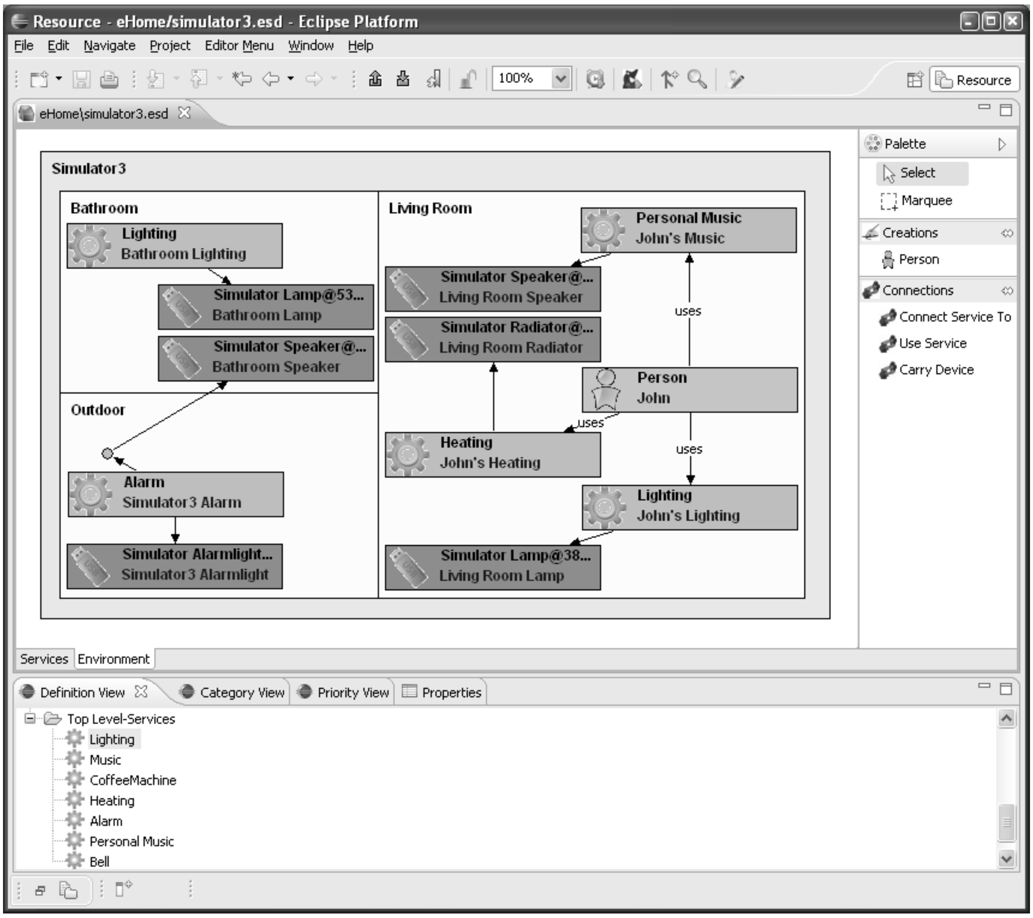


Abbildung 6.11: Hauptfenster des Laufzeit-Managers.

tels *Drag-and-Drop*-Funktionalität in der grafischen Darstellung der Umgebung „abgelegt" werden, was die räumliche Zuordnung festlegt. Zur Auswahl der Elemente dient zum einen die Palette auf der rechten Seite des Editorfensters und zum anderen eine Ansicht der geladenen Top-Level-Dienste, die unter *Definition View* am unteren Rand der Abbildung zu sehen ist. Diese Dienste können per *Drag-and-Drop* den Räumen zugewiesen werden, was eine Instanziierung des jeweiligen Dienstes bewirkt. Die Dienstinstanzen werden dann im Laufzeit-Manager angezeigt und können auch innerhalb der Umgebung verschoben werden. Die Zuordnung von Personen und Top-Level-Diensten ermöglicht es, Dienste als personenbezogene Dienste zu nutzen. Basisdienste werden zusammen mit dem jeweils gesteuerten Gerät angezeigt. Der Übersichtlichkeit halber werden integrierende Dienste in einer reduzierten Darstellung als kleine Kreise dargestellt, deren Informationen bei Bedarf eingeblendet werden können.

Durch die Interaktion mit dem Werkzeug spezifiziert der Benutzer seine Anforderungen und kann diese jederzeit anpassen. Dies entspricht der Spezifizierungsphase des kontinuierlichen SCD-Prozesses. Das Werkzeug stellt die momentan konfigurierte und deployte Situation des eHome-Systems dar und bietet so neben den Interaktionsmöglichkeiten auch eine Visualisierung des aktuellen Betriebszustands. Dies gibt dem Benutzer die Möglichkeit, das Verhalten des Systems zu verstehen und ggfs. seinen Bedürfnissen anzupassen.

6.4 Anwendungsumgebungen

Zum Testen und Auswerten der in dieser Arbeit entwickelten Konzepte wurden verschiedene Testszenarien realisiert und ausgeführt. Einige der implementierten Beispieldienste wurden bereits in Abschnitt 2.4 beschrieben. Diese Beispieldienste realisieren exemplarisch typische eHome-Funktionalitäten. Da im Rahmen des Projekts keine reale Wohnumgebung zur Verfügung stand, wurden stattdessen Testumgebungen zur Simulation von eHomes entwickelt. Die darin verwendeten Testgeräte werden durch entsprechende Treiberdienste angesteuert, sodass sich im Vergleich zu realen Geräten in einem Haushalt kein qualitativer Unterschied ergibt. Im Verlauf des Projekts wurde mehre solcher Testumgebungen verwendet.

Bereits in der Arbeit von NORBISRATH [Nor07] wurden zwei Testumgebungen eingeführt. Die erste Umgebung ist der sogenannte *X10-Demonstrator*, der in Abbildung 6.12 zu sehen ist. Er wurde, einem Puppenhaus nachempfunden, aus Holz gebaut und besteht aus mehreren Räumen. Als Residential Gateway wurden drei PCs hinter dem Demonstrator montiert, die neben der Ausführung von Diensten auch für die Anbindung der Geräte des Demonstrators verwendet wurden. Der X10-Demonstrator enthält verschiedene Geräte, die über X10-Technik (vgl. Abschnitt 2.3 und [Sma09a]) angesteuert werden können. Unter anderem wurden X10-Schalter, X10-Lampenfassungen und X10-Bewegungsmelder verwendet. Zusätzlich wurden Touchscreens, Webcams, Aktivboxen und Kopfhö-

Abbildung 6.12: Der X10-Demonstrator. (Quelle: [Nor07])

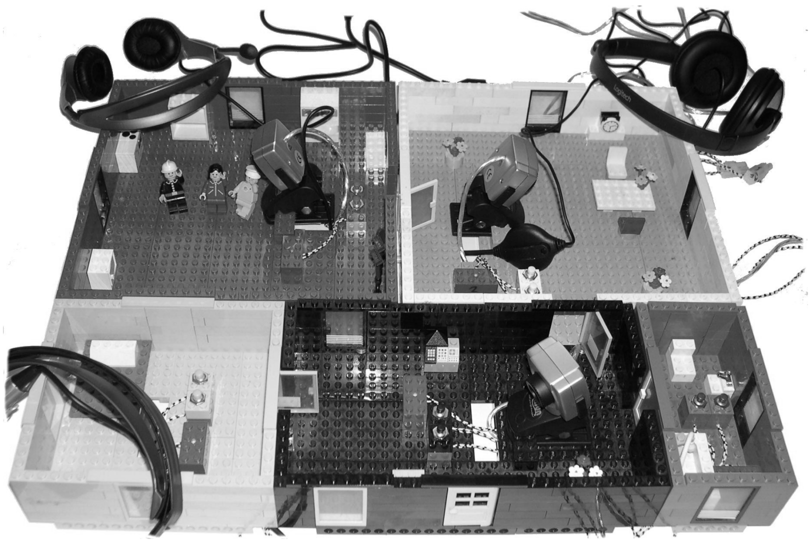


Abbildung 6.13: Der Lego-Demonstrator. (Quelle: [Nor07])

Abbildung 6.14: Der eHome-Simulator.

rer angeschlossen. Diese können direkt von den PCs aus angesteuert werden, während die X10-Geräte über ein X10-Gateway mit den PCs verbunden sind.

Eine zweite Testumgebung aus der Arbeit von NORBISRATH ist der *Lego-Demonstrator*, der in Abbildung 6.13 dargestellt ist. Dieser ist aus Lego-Steinen zusammengesetzt und mit LED-Lampen, Tastern und ebenfalls Webcams ausgestattet. Zusätzlich werden Kopfhörer für die Audioausgabe verwendet. Als Residential Gateway wurde für den Lego-Demonstrator ein Laptop eingesetzt. Die Lego-Bauteile wurden über ein spezielles Steuermodul angesteuert, das über die USB-Schnittstelle mit dem Laptop verbunden werden konnte. Die Webcams und Kopfhörer wurden ebenfalls mittels USB an den Laptop angeschlossen. Aufgrund der geringen Größe und dem als Residential Gateway verwendeten Laptop konnte der Lego-Demonstrator leichter transportiert und modifiziert werden als der X10-Demonstrator.

Um die Flexibilität der durchführbaren Tests weiter zu erhöhen, wurde im Rahmen dieser Arbeit und der Arbeit von ARMAÇ eine weitere Testumgebung entwickelt, der *eHome-Simulator* [AR07]. Dieser ist im Gegensatz zu den beiden oben beschriebenen Demonstratoren vollständig in Software realisiert.

Die simulierte Umgebung und die darin enthaltenen Personen und Geräte können durch ein interaktives grafisches Werkzeug zur Laufzeit gesteuert und modifiziert werden. Auf diese Weise ist es möglich, weitergehende Szenarien zu simulieren, die im Rahmen der Hardware-Demonstratoren nicht umsetzbar waren. So ist es im eHome-Simulator beispielsweise auch möglich, Geräte wie Heizungen, eine Kaffeemaschine oder eine vollautomatische Personenerkennung zu simulieren. Auch die Steuerung von Fenstern und Türen durch das eHome-System ist im Simulator möglich.

Im Hauptfenster des Simulators ist die simulierte eHome-Umgebung grafisch dargestellt. In der Mitte des Ausschnitts ist eine Person zu sehen, am unteren Bildrand befinden sich ein Heizkörper und eine Lampe. Im oberen Teil sind eine Kaffeemaschine und ein Lautsprecher zu sehen. Die Person kann interaktiv durch die Umgebung navigiert werden, es ist außerdem möglich, weitere Personen hinzuzufügen. Eine Modifikation der Geräteumgebung durch Hinzufügen, Entfernen oder Verschieben von Geräten wird ebenfalls unterstützt. Auf diese Weise können mit dem eHome-Simulator alle in Abschnitt 4.1.1 erläuterten Einflussfaktoren auf Mobilität und Dynamik in eHomes simuliert werden.

Am linken Rand des eHome-Simulators befindet sich ein Bereich mit Informationen zum aktuellen Umgebungsstatus und einer Gesamtübersicht der ausgewählten eHome-Umgebung. Darunter sind verschiedene Werkzeuge für die Interaktion mit der Umgebung angeordnet. Zum einen finden sich dort Werkzeuge zur Verwaltung der Personen und der Geräte, zum anderen werden auch Interaktionsgeräte aus der simulierten Umgebung selbst, wie z. B. Schalter und Regler, in diesem Bereich angezeigt. Die Anbindung der eHome-Laufzeitumgebung an den Simulator geschieht über eine Netzwerkschnittstelle. Auf diese Weise können beide Applikationen auf unterschiedlichen Rechnern betrieben werden. Die Anbindung mehrerer Simulator-Instanzen wurde im Rahmen dieser Arbeit nicht durchgeführt, dies wäre aber aufgrund der verwendeten Infrastruktur mit geringem Aufwand realisierbar.

Auf Basis des eHome-Simulators wurden unter anderem drei Testszenarien umgesetzt, die den in Abschnitt 2.4 vorgestellten typischen Anwendungsfeldern von Funktionalitäten in eHomes entsprechen. Dabei konnten die in dieser Arbeit entwickelten und implementierten Konzepte erprobt werden.

Das *Komfortszenario* wurde durch Dienste zur Personalisierung der unmittelbaren Umgebung von Nutzern umgesetzt. Dazu wurden ein *Musikdienst* zum Abspielen einer personalisierten Musikauswahl, ein *Lichtdienst* für eine individuelle Beleuchtung, sowie ein *Heizungsdienst* für eine benutzerspezifische Temperierung der Umgebung erstellt. Diese Szenarien konnten zusammen mit den in Abschnitt 6.3 beschriebenen Werkzeugen und dem eHome-Simulator erfolgreich umgesetzt werden.

Als Beispiel für das *Sicherheitsszenario* wurde ein Alarmdienst entwickelt. Dieser verwendet Glasbruchsensoren an den Fenstern der Umgebung, sowie Bewegungsmelder und Sensoren an den Türen, um damit eine Einbruchssituation zu detektieren. Wird ein Einbruch erkannt, so reagiert der Alarmdienst mit verschiedenen Maßnahmen. Zunächst werden die Fenster und Türen verriegelt und es wird ein Alarmsignal über einen außen angebrachten Signalgeber ausgelöst. Der Signalgeber erzeugt ein akustisches Signal und verfügt außerdem über ein rotes Blinklicht. Zusätzlich werden die Lampen und Lautsprecher im eHome aktiviert, sodass auch dort ein akustisches Signal ausgeben werden kann. Die Lampen werden zum Blinken gebracht, um den Eindringling in seinem Vorhaben zu stören, und zusätzlich Aufmerksamkeit bei eventuellen Passanten oder Nachbarn zu erzeugen.

Für das Szenario der *Unterstützung von Senioren* wurde der in Abschnitt 2.4.3 erwähnte Klingeldienst realisiert. Durch einen Taster am Eingang des eHomes können sich Besucher bemerkbar machen. Dazu kann die übliche Variante eines akustischen Signals, das die Bewohner aufmerksam macht, umgesetzt werden. Es besteht aber aufgrund der Dienstspezifikation auch die Möglichkeit, mehrere semantisch passende Varianten einzusetzen, z. B. die Benachrichtigung der Bewohner durch ein visuelles Signal, das im eHome-Simulator durch blinkende Lampen realisiert werden kann. In diesem Szenario kommt die semantische Komposition über abstrakte Funktionalitäten zum Einsatz, die eine höhere Flexibilität bei der Realisierung von Top-Level-Diensten und damit den vom Benutzer gewünschten Funktionalitäten ermöglicht.

6.5 Umfang der Implementierung

Bei der Implementierung der Konzepte dieser Arbeit wurden unterschiedliche Vorgehensweisen angewandt. Während die Anwendungslogik der Laufzeitumgebung im Wesentlichen in einem modellgetriebenen Ansatz umgesetzt wurde, sind andere Teile, wie die eHome-Dienste, klassisch in Java implementiert worden. Bei der Entwicklung eines Dienstes können einige der benötigten Klassen mit nur geringfügigen Anpassungen aus Vorlagen übernommen werden. Die wesentliche Arbeit besteht darin, die Anwendungslogik des Dienstes zu implementieren. Die Werkzeuge wurden ebenfalls klassisch in Java entwickelt, wobei auf die Eclipse RCP und GEF aufgebaut wurde (vgl. Abschnitt 6.2).

Im Folgenden wird eine Übersicht über den Umfang der implementierten Konzepte gegeben. Dazu wird jeweils die Anzahl der Codezeilen betrachtet, die nicht leer sind und keine Kommentare beinhalten. Dabei wird zwischen den Codezeilen insgesamt (*LoC Gesamt*) und den Codezeilen in Methodenrümpfen (*LoC Methoden*) unterschieden. Zusätzlich wird die Anzahl der Klassen und die Anzahl der definierten Methoden betrachtet.

	LoC Gesamt	**LoC Methoden**	**Klassen**	**Methoden**
Service-Editor	3.661	2.303	46	229
Laufzeit-Manager	22.267	13.308	295	1.646
eHome-Simulator	9.289	6.196	98	708
Gesamt	35.217	21.807	439	2.583

Tabelle 6.1: Umfang der Werkzeug-Implementierungen.

Tabelle 6.1 zeigt eine Übersicht des Umfangs der implementierten Werkzeuge. Hier fällt auf, dass die Implementierung des Service-Editors vergleichsweise kompakt ausfällt, während der Laufzeit-Manager mit über 20.000 Codezeilen das umfangreichste Projekt ist. Dies liegt darin begründet, dass der Laufzeit-Manager die Darstellung und Interaktion mit dem eHome-Modell realisiert. Daher müssen für alle Modellelemente entsprechende *EditPart*-Klassen als Controller erzeugt werden. Zusätzlich müssen Policy-Klassen für die Interaktion und Klassen für die grafische Darstellung erstellt werden. Für die verschiedenen Views des Editors müssen jeweils weitere Klassen implementiert werden. Dazu kommen Dialoge und die Anwendungslogik zum Einlesen der Dienst- und Umgebungsspezifikationen, die von dem Werkzeug verarbeitet werden müssen. Insgesamt ergeben sich also zahlreiche Aspekte, die der Laufzeit-Manager realisieren muss. Der große Umfang an Quellcode liegt also zum einen in der Architektur des MVC-Musters in GEF begründet und zum anderen auch in der umfassenden Funktionalität des Werkzeugs. Ein weiterer, ganz praktischer Aspekt ist, dass der Laufzeit-Manager eine längere Entwicklungsgeschichte aufzuweisen hat. So wie auch der eHome-Simulator, der ebenfalls vergleichsweise umfangreich ausfällt, wurde mit der Entwicklung des Laufzeit-Managers bereits in einer frühen Phase der Arbeit begonnen. Aus diesem Grund wurden im Verlauf der Entwicklung immer neue Funktionen realisiert, die zum Umfang der Implementierung beigetragen haben.

	LoC Gesamt	**LoC Methoden**	**Klassen**	**Methoden**
SCD-Prozess	30.427	24.017	90	1.236
davon generiert	*29.459*	*23.541*	*73*	*1.141*
Semantische Adaption	2.303	1.341	24	105
Simulatoranbindung	698	508	7	34
Gesamt	33.428	25.866	121	1.375

Tabelle 6.2: Umfang der Implementierungen der Laufzeitumgebung.

Aus Tabelle 6.2 geht der Umfang der Laufzeitumgebung hervor. Darin ist die Implementierung des eigentlichen SCD-Prozesses und der strukturellen Adaption, des Teilprozesses zur semantischen Adaption, sowie die damit zusammenhängende Infrastrukturfunktionalität zusammengefasst. Zusätzlich ist in der Tabelle noch die Implementierung der Anbindung des eHome-Simulators als Anwendungsumgebung angegeben. Darin ist im Wesentlichen die Netzwerkkommunikation mit dem Simulator realisiert. Die Simulatoranbindung ist der kleinste Teil der Laufzeitumgebung und gehört nicht zum eigentlichen Kern des Systems, da die Anbindung unterschiedlicher Umgebungen unterschiedlich gelöst werden kann. Der entsprechende Teil ist daher je nach Art der vorliegenden Infrastruktur ggfs. auszutauschen.

Die Mechanismen zur semantischen Adaption sind direkt in Java implementiert worden, während für die Realisierung des SCD-Prozesses zur strukturellen Adaption ein modellgetriebener Ansatz mit Fujaba verfolgt wurde. Die semantische Adaption konnte recht kompakt realisiert werden. Die Dienstspezifikation wird vom Entwickler vorgenommen und ist bereits durch den Service-Editor als

Werkzeug abgedeckt (siehe Tabelle 6.1). Zur Laufzeit muss für die semantische Adaption zum einen die Semantic Registry realisiert werden, damit bei der Konfigurierung auch semantische Matches gefunden werden können. Dies konnte auf Basis der Dienstspezifikationen effizient implementiert werden. Zum anderen muss die Adaptergenerierung als zweite wesentliche Aufgabe zur Laufzeit gelöst werden. Dazu werden JET-Templates eingesetzt, die Vorlagen des zu erzeugenden Adaptercodes beinhalten (vgl. Abschnitt 5.7.4). Durch die Verwendung von Templates konnte auch hier die Implementierung kompakt gelöst werden.

Die Realisierung des SCD-Prozesses und der strukturellen Adaption ist wesentlich umfangreicher. Ein nicht zu vernachlässigender Faktor dabei ist die Verwendung von Fujaba zur Codegenerierung. Der Anteil des generierten Codes ist ebenfalls in Tabelle 6.2 angegeben. Der allergrößte Teil der Implementierung des SCD-Prozesses ist in Fujaba realisiert und wird durch Codegenerierung erzeugt. Einige weitere Hilfsklassen wurden von Hand implementiert und stellen den Rest dar. Der mit Fujaba generierte Quellcode ist nicht auf Lesbarkeit und Effizienz optimiert. Daher ergibt sich hier eine hohe Anzahl an Codezeilen, die aber bei einer händischen Implementierung durchaus geringer ausfallen könnte. Auffällig ist auch, dass die Anzahl der in Fujaba spezifizierten Klassen mit 90 im Vergleich zu z. B. der semantischen Adaption mit 24 Klassen nicht übermäßig hoch ist, während die Anzahl der Codezeilen im selben Vergleich einen viel höheren Faktor aufweist. Der von Fujaba generierte Code besteht also aus deutlich umfangreicheren Klassen, mit einer höheren Anzahl von Codezeilen pro Klasse. Ein direkter Vergleich mit der Implementierung der semantischen Adaption ist jedoch nicht möglich, da beide Teilsysteme ganz unterschiedliche Funktionen realisieren und daher auch naturgemäß unterschiedlich umfangreich sind. Dies muss bei der Betrachtung der Zahlen berücksichtigt werden.

	LoC Gesamt	**LoC Methoden**	**Klassen**	**Methoden**
Top-Level-Dienste	199	134	3	12
Integrierende Dienste	96	44	3	9
Basisdienste	135	52	5	15

Tabelle 6.3: Durchschnittswerte des Implementierungsumfangs von Beispieldiensten.

Abschließend sind in Tabelle 6.3 die Daten einiger der implementierten Beispieldienste aufgeführt. Um einen Vergleich zu ermöglichen, sind die Durchschnittswerte pro Dienst angegeben. Dabei wurden acht Top-Level-Dienste, vier integrierende Dienste und zwölf Basisdienste zugrunde gelegt. Die Top-Level-Dienste sind im Vergleich am umfangreichsten. Sie haben insgesamt die meisten Codezeilen und auch die umfangreicheren Methoden, was an dem Wert *LoC Methoden* abgelesen werden kann. Dies liegt darin begründet, dass Top-Level-Dienste die eigentlichen vom Nutzer gewünschten Funktionalitäten implementieren. Somit realisieren sie eine umfangreichere Anwendungslogik als typische integrierende Dienste. Diese realisieren häufig einfache Zusatzfunktionalitäten oder Abwandlungen der zugrunde liegenden Basisfunktionalität. Denkbar wären jedoch auch umfangreichere integrierende Dienste, die einen großen Abstraktionsschritt realisieren. Solche Dienste wurden jedoch im Rahmen der Beispiele dieser Arbeit nicht entwickelt. Schließlich werden noch die Basisdienste untersucht. In der Tabelle wurden dazu Treiberdienste für den eHome-Simulator betrachtet. Die Basisdienste umfassen üblicherweise weniger Codezeilen als Top-Level-Dienste, jedoch mehr als integrierende Dienste. Insbesondere enthalten sie mehr Methoden, die aber weniger umfangreich sind. Der Grund dafür ist, dass die Basisdienste den Zugriff auf die Geräte im eHome verkapseln und damit die verschiedenen Operationen zum Ansteuern dieser Geräte an die zugrunde liegende Infrastruktur weiterreichen müssen. Da die Operationen jeweils durch Methoden realisiert werden, ergibt sich daraus die erhöhte Anzahl. Weil aber keine komplexe Anwendungslogik realisiert werden muss, ist der Umfang der einzelnen Methoden jeweils gering.

6.6 Zusammenfassung

In diesem Kapitel wurden die im Rahmen der vorliegenden Arbeit entwickelten Werkzeuge vorgestellt. Die Werkzeuge decken den Gesamtprozess von der Ontologieentwicklung und der Dienstentwicklung über die Installation eines spezifischen eHomes bis hin zur Laufzeitphase, die durch den kontinuierlichen SCD-Prozess umgesetzt wird, ab.

Zunächst wurde eine Übersicht des Systemaufbaus beschrieben, und es wurden einige Grundlagen erläutert, die für die Entwicklung der Werkzeuge von Bedeutung sind. Insbesondere wurden die Eclipse Rich Client Platform (RCP) und das Graphical Editing Framework (GEF) vorgestellt, und es wurde die Anwendung des MVC-Musters mit GEF und Fujaba beschrieben.

Die einzelnen Werkzeuge wurden näher vorgestellt. Der Service-Editor wird zur Dienstspezifikation verwendet. Darin werden alle Informationen über einen Dienst festgelegt, die zur Dienstkomposition im kontinuierlichen SCD-Prozess benötigt werden, einschließlich der semantischen Dienstbeschreibung. Der Umgebungseditor legt die statischen Kontextinformationen der eHome-Umgebung fest, es wird insbesondere der Grundriss des Gebäudes für den räumlichen Kontext festgelegt. Der Laufzeit-Manager wird zur Visualisierung und Interaktion mit dem eHome-System zur Laufzeit verwendet. Darin kann insbesondere die Auswahl der Dienste, deren Zuordnung zu Räumen oder Personen sowie die Dienstkomposition modifiziert werden.

Des Weiteren wurden die im eHome-Projekt verwendeten Anwendungsumgebungen vorgestellt. Für diese Arbeit wurde dabei in erster Linie auf den neu entwickelten eHome-Simulator zurückgegriffen, der eine virtuelle eHome-Umgebung zur Verfügung stellt. Damit konnten die in Abschnitt 2.4 beschriebenen Szenarien in einer Beispielimplementierung getestet werden.

Zusätzlich wurden einige Daten zum Umfang der Implementierungen der Werkzeuge, der Laufzeitumgebung und der Dienste präsentiert. Dabei wurde auch auf den jeweiligen Entwicklungsprozess und die verwendeten Plattformen und Werkzeuge, wie z. B. Fujaba, eingegangen.

Kapitel 7

Schlussbemerkungen

In diesem abschließenden Kapitel werden die Themen der vorliegenden Arbeit zusammengefasst. Nach einer Übersicht der Ergebnisse erfolgt eine Bewertung des Beitrags der Arbeit in Bezug auf die wissenschaftlichen Fragestellungen. In einem Ausblick werden einige weiterführende Themen aufgezeigt, die auf der Grundlage dieser Arbeit Potential für zukünftige Arbeiten bieten.

7.1 Zusammenfassung

Im Rahmen der vorliegenden Arbeit wurden Konzepte für eine adaptive Softwareinfrastruktur für eHomes entwickelt. Ziel dieser Konzepte ist es, zum einen die Entwicklung von eHome-Diensten durch eine Unterstützung auf der Ebene der Laufzeitumgebung zu erleichtern. Zum anderen soll eine Adaptivität des Systems zur Laufzeit gewährleistet werden, die den Anforderungen eines eHome-Systems entspricht.

In der Arbeit wurden zwei zentrale Bereiche untersucht, die strukturelle Adaption und die semantische Adaption. Die *strukturelle Adaption* adressiert Änderungen der Benutzeranforderungen und Änderungen des Kontextes zur Laufzeit, auf die das eHome-System entsprechend reagieren muss. Damit wird insbesondere die *Mobilität* und *Dynamik* in eHomes berücksichtigt, die sowohl von den

Personen im eHome als auch von den dort genutzten Geräten ausgeht. Die *semantische Adaption* adressiert die Heterogenität im Umfeld von eHomes, die insbesondere auch die eHome-Dienste betrifft. Als Grundlage der Dienstkomposition dienen im Ansatz der vorliegenden Arbeit anstatt syntaktischer Schnittstellen semantische Beschreibungen der Dienstfunktionalitäten.

Strukturelle Adaption

Eine Berücksichtigung dynamischer Veränderungen ist essentiell für die Anwendung von eHome-Systemen. Der in [Nor07] entwickelte SCD-Prozess wurde zuvor als einmaliger Vorgang zur Einrichtung eines eHome-Systems betrachtet, der nach dem Deployment und dem Start des Systems abgeschlossen ist. Die Weiterentwicklung zum *kontinuierlichen SCD-Prozess* im Rahmen dieser Arbeit bietet neue Mechanismen zur dynamischen Adaption, sodass spätere Anpassungen der Anforderungen zur Laufzeit berücksichtigt werden können.

Funktionalitäten wie kontextbezogene Dienstbindungen müssen nicht mehr in den Diensten selbst implementiert werden, diese Querschnittsfunktionalitäten werden stattdessen durch die Laufzeitumgebung realisiert. Dadurch reduziert sich der Entwicklungsaufwand und es kann eine redundante Mehrfachentwicklung mit parallelen, zueinander inkonsistenten Implementierungen vermieden werden. Der neue kontinuierliche SCD-Prozess umfasst die Schritte *Spezifizierung*, *Konfigurierung* und *Deployment*, die nun zur Laufzeit unterstützt werden.

Damit eine automatisierte Verarbeitung zur Laufzeit möglich ist, wurde die Spezifikation der Dienste erweitert. Durch die Einführung von *Bindungsstrategien* kann das Verhalten bei der Konfigurierung festgelegt werden. So ist eine automatische oder manuelle Konfigurierung möglich, ebenso kann festgelegt werden, bis zu welchem Grad die Dienstabhängigkeiten zu erfüllen sind. Bindungsstrategien ermöglichen es, eine automatische Reaktion des Systems auf dynamische Änderungen herbeizuführen. Dies ist erforderlich, da der Nutzer nicht bei jeder Veränderung manuell interagieren möchte. Daher war es wichtig, eine automatische Rekonfigurierung zu ermöglichen.

Die neu eingeführten *Bindungsbeschränkungen* ermöglichen eine genauere Festlegung darüber, welche Dienste für eine Bindung in Frage kommen. Neben den benötigten Funktionalitäten und den zugehörigen Bindungskardinalitäten können so z. B. auch Kontextbedingungen mit einbezogen werden. Bindungsbeschränkungen werden durch Graphmuster in Fujaba realisiert und können daher auf alle modellierten Entitäten des eHome-Modells Bezug nehmen. Durch Bindungsbeschränkungen ist beispielsweise die einfache Realisierung personenbezogener Dienste möglich.

Die Einführung *nebenläufiger Bindungen* ermöglicht eine effizientere Nutzung von Ressourcen im eHome. Dienste, deren angebotene Funktionalitäten bereits gebunden sind, können nun von weiteren Diensten genutzt werden, wenn die momentan tatsächlich genutzten Bindungen den Dienst noch nicht vollständig ausschöpfen. Viele Ressourcen können nicht parallel genutzt werden und bieten ihre Funktionalitäten daher immer nur einem Dienst gleichzeitig an. Auf der anderen Seite werden aber diese Funktionalitäten häufig gar nicht dauerhaft genutzt. Zur Auflösung dieses Konflikts haben sich nebenläufige Bindungen als nützliches Konzept erwiesen. Durch die zusätzliche Verwendung von *Prioritäten* kann eine Rangordnung der nutzenden Dienste für individuelle Funktionalitäten festgelegt werden. Dies ist besonders für sicherheitsrelevante Dienste eine wichtige Erweiterung.

Semantische Adaption

Die geringe Standardisierung der Technologien, die in eHomes Anwendung finden, spiegelt sich auch auf der Ebene der eHome-Dienste wider. Dienste werden daher nicht gemäß eines globalen Standards entwickelt, sodass mit syntaktischen Inkompatibilitäten zu rechnen ist. Um dieser Problematik zu begegnen, wurde im Rahmen der vorliegenden Arbeit die *semantische Adaption* eingeführt, die die Komposition von eHome-Diensten auf eine semantische Grundlage stellt. Entscheidend für die Dienstkomposition ist somit nicht mehr in erster Linie eine gemeinsame syntaktische Schnittstelle, sondern die *semantische Übereinstimmung* der Dienstfunktionalitäten.

Damit eine Dienstkomposition auf Basis semantischer Beschreibungen möglich ist, muss eine einheitliche Definition der semantischen Konzepte der Anwendungsdomäne vorliegen. Dazu wird in dieser Arbeit eine *eHome-Ontologie* verwendet, die zunächst erstellt werden muss. Dienste können dann mit Bezug auf diese Ontologie semantisch beschrieben werden. Die Erfassung aller relevanten

Konzepte in einer Ontologie ist grundsätzlich ein nicht triviales Problem, da die Sachverhalte häufig auf verschiedenste Weisen modelliert werden können. Die Einschränkung auf den Bereich eHomes und die typischen Basisfunktionalitäten erleichtert dabei das Vorgehen und macht einen Lösungsansatz möglich.

Nachdem eine Ontologie spezifiziert wurde, können eHome-Dienste *semantisch beschrieben* werden. Dazu werden die syntaktischen Elemente ihrer Schnittstellen auf die semantischen Konzepte der Ontologie abgebildet. Diese Beschreibung wurde als weiterer Bestandteil in die Dienstspezifikation aufgenommen.

Die semantische Beschreibung dient zur Laufzeit als Grundlage für das *semantische Matching*, d. h. das Auffinden zueinander passender Dienste im Rahmen der Konfigurierung im kontinuierlichen SCD-Prozess. Da die Interaktion und Kommunikation unter den Diensten auf syntaktischer Basis erfolgt, wird darüber hinaus ein Mechanismus zur Adaptierung benötigt.

In dieser Arbeit wurde ein Ansatz zur *automatischen Adaptergenerierung* auf Basis der semantischen Dienstbeschreibungen vorgestellt. Die semantische Spezifikation dient damit nicht nur zum semantischen Matching, sondern darüber hinaus als Grundlage der Adaption, die die spätere Komposition der Dienste erst ermöglicht. Die Adapterkomponenten müssen dynamisch erzeugt werden, da sich die Komposition der Dienste im Zuge der strukturellen Adaption erst zur Laufzeit ergibt und sich außerdem laufend verändern kann.

Werkzeugunterstützung

Die in dieser Arbeit entwickelten Konzepte werden durch entsprechende Werkzeuge umgesetzt. Während der Dienstentwicklung wird der *Service-Editor* für die Spezifikation verwendet. Dieser wurde als Plug-In für die Entwicklungsumgebung Eclipse entwickelt und steht so bereits während der Dienstimplementierung zur Verfügung. Damit ist eine integrierte Entwicklung und Spezifikation möglich.

Als Grundlage für die semantische Dienstbeschreibung dient eine Ontologie, die mit *Protégé*, einem verbreiteten Werkzeug zur Modellierung von Wissen und Ontologien, erstellt wird. Es musste daher für diesen Zweck kein eigenes Werkzeug in dieser Arbeit entwickelt werden.

Für die Ausführungsphase wurde der *Laufzeit-Manager* als Werkzeug zur Interaktion mit dem eHome-System entwickelt. Damit ist eine Kontrolle des momentanen Zustands der Konfiguration möglich, so wie eine manuelle Anpassung. Die Möglichkeit des Benutzers zur Interaktion ist ein wichtiger Aspekt dieser Arbeit, da Benutzer die Kontrolle über ihre Wohnumgebungen behalten möchten. Ein System, das automatische Dienstkompositionen erstellt, auf die der Nutzer keinerlei Einfluss hat und die er im ungünstigsten Fall auch nicht nachvollziehen kann, wird keine Verbreitung finden.

7.2 Bewertung

Die in dieser Arbeit entwickelten Lösungskonzepte erfüllen die Zielsetzungen aus Abschnitt 1.2. Der bestehende Ansatz im eHome-Projekt konnte zu einer Laufzeitumgebung für eHome-Dienste weiterentwickelt werden, die Mechanismen zur Unterstützung von Kontextbezug und einer personenbezogenen Konfigurierung der Dienste zur Verfügung stellt.

Der neu eingeführte kontinuierliche SCD-Prozess ermöglicht nun eine *dynamische Adaption*. Die *Laufzeitphase* wurde in den vorherigen Arbeiten im eHome-Projekt nicht betrachtet, diese birgt jedoch große Herausforderungen. Die Anpassung der Dienstkomposition zur Laufzeit wird nun ermöglicht und die Dienstentwicklung konnte durch die Einführung von Middleware-Funktionalitäten in der Laufzeitumgebung erleichtert werden.

Durch den semantischen Teilprozess konnte die Laufzeitumgebung um semantische Funktionalitäten erweitert werden, was eine *höhere Flexibilität* bei der Dienstkomposition und eine *Überwindung syntaktischer Inkompatibilitäten* ermöglicht.

Die Entwicklung geeigneter *Werkzeuge* zur Unterstützung des neuen Prozesses war ein weiteres gesetztes Ziel. Auch dieses konnte erreicht werden. Die in Abschnitt 6.3 vorgestellten Werkzeuge decken die verschiedenen Phasen des Gesamtprozesses ab und ermöglichen eine Unterstützung der Entwickler und Anwender bei den jeweiligen Aufgabenstellungen. Die gesetzten Ziele konnten somit erreicht werden.

Graphbasierter Ansatz

Aus wissenschaftlicher Sicht liefern die Ergebnisse dieser Arbeit in verschiedener Hinsicht einen Beitrag. Ein wichtiger Aspekt ist der Einsatz *graphbasierter Techniken* in einer serviceorientierten Middleware. Dadurch haben sich für diese Arbeit verschiedene Implikationen ergeben.

Sowohl die Modellierung der Struktur als auch die Modellierung des Verhaltens der Laufzeitumgebung konnten auf einer abstrakteren Modellebene realisiert werden, der letztendliche Quellcode wurde automatisch generiert. Damit ergeben sich die Vorteile der modellgetriebenen Softwareentwicklung, z. B. die bessere Übersicht über das entwickelte System und seine Bestandteile sowie die Möglichkeit einer späteren Portierung. Auch die deklarative, visuelle Vorgehensweise bietet Vorteile, z. B. beim Spezifizieren gesuchter Graphmuster in Story-Diagrammen.

Das *globale Graphmodell* des eHome-Systems, das der Laufzeitumgebung zugrunde liegt, bietet den Vorteil, dass bei der Umsetzung der Lösungen auf alle relevanten Daten einfach zugegriffen werden kann. Informationen über die eHome-Umgebung, den Kontext von Personen und Geräten, die geladenen eHome-Dienste und die vom Benutzer erzeugten Dienstinstanzen, all dies ist im eHome-Modell erfasst und kann zur Verarbeitung genutzt werden. Relevante Daten müssen somit nicht aus den unterschiedlichsten Quellen zusammengefügt werden, was für die Realisierung des SCD-Prozesses ein wichtiger Faktor war.

Ein weiterer Vorteil des graphbasierten Ansatzes ergab sich für die *Werkzeugentwicklung*. Die Verwendung der Eclipse Rich Client Platform (RCP) und des Graphical Editing Frameworks (GEF) ermöglicht die Entwicklung von Werkzeugen für beliebige zugrunde liegende Modelle. Das eHome-Modell konnte damit neben der Verwendung für die Anwendungslogik der Laufzeitumgebung auch für die entwickelten Werkzeuge genutzt werden. Für die darzustellenden Elemente des Modells wurden dazu passende View- und Controller-Komponenten entwickelt.

Die eigentliche Implementierung des Editors wurde von Hand durchgeführt, denkbar wäre aber ein verbesserter Entwicklungsprozess, der das Grundgerüst des Editors automatisch generieren lässt, so wie es mit UPGRADE und PROGRES möglich ist (vgl. Abschnitt 3.3.1). So könnte der graphbasierte Ansatz noch besser für die Werkzeugentwicklung ausgenutzt werden. Für die spätere Anwendung der Werkzeuge ist dies jedoch nicht von Bedeutung.

Dynamische Adaptergenerierung

Ein weiterer wissenschaftlicher Beitrag dieser Arbeit ist die Entwicklung eines *Laufzeitmechanismus zur automatischen Adaptergenerierung*. Dabei wurde auf die Meta-Programmierung und Templates zur Codegenerierung sowie auf eine semantische Dienstspezifizierung zurückgegriffen.

Das Problem der Komposition heterogener Dienste wird in verschiedenen Projekten betrachtet. In vielen Ansätzen wird dabei jedoch hauptsächlich auf Mechanismen zur semantischen Inferenz eingegangen, die zum Auffinden passender Dienstkompositionen genutzt werden können. Die Übertragung auf die syntaktische Ebene wird aber häufig ausgeklammert. Gerade diese Übertragung wird in dieser Arbeit betrachtet.

Ein Lösungsansatz zur dynamischen Erzeugung von Adapterkomponenten wurde in dieser Arbeit vorgestellt. Zunächst wird auf Basis semantischer Dienstbeschreibungen, die die syntaktischen Schnittstellenelemente der Dienstimplementierungen auf semantische Ontologiekonzepte abbilden, der für die Adaption benötigte Quellcode generiert. Dabei wird auf Code-Templates zurückgegriffen, in denen aus den unterschiedlichen syntaktischen Elemente und deren semantischer Zuordnung die benötigten Methoden erzeugt werden. Die Verwendung von Templates hat dabei den Vorteil, dass nicht die konstruktive Erstellung des Codes implementiert werden muss, sondern der Code in gewohnter Struktur formuliert werden kann, wobei die einzelnen Elemente ggfs. durch das zugrunde liegende Match und die daran beteiligten Schnittstellen parametrisiert sind.

Der generierte Adaptercode muss danach kompiliert und in das laufende System integriert werden. Dies wird durch die Verwendung der Bibliothek Javassist realisiert. Diese ermöglicht die *strukturelle Reflection* in Java, einem Ansatz zur Meta-Programmierung. So können neue Klassen und Methoden erzeugt und in das laufende System eingebunden werden. Dieser Ansatz zeigt neue Möglichkeiten auf, Dienste, die wie in OSGi über direkte Methodenaufrufe miteinander kommunizieren, mittels Adapterkomponenten zu koppeln.

Aspektorientierte Dienstüberwachung

Die Verwendung aspektorientierter Programmierung zur *Laufzeitüberwachung der Dienstkommunikation* ist ein weiterer Aspekt dieser Arbeit. Häufig zitierte Beispiele für die Nutzung aspektorientierter Programmierung sind Querschnittsfunktionalitäten wie das Logging. Aspekte können aber auch zur Überwachung einer Anwendung genutzt werden. Ein Beispiel dafür zeigt diese Arbeit, in der die aspektorientierte Programmierung für die Überwachung der Kommunikation und Interaktion zwischen eHome-Diensten genutzt wird.

So konnten *nebenläufige Bindungen* und die *Priorisierung* von Diensten umgesetzt werden. Das sogenannte *Load-Time Weaving* bindet die Aspekte erst beim Laden einer Komponente ein. Mit Eclipse Equinox und AspectJ können so auch für OSGi-Bundles Aspekte erzeugt werden, die beim Laden der Bundles eingebunden werden. Der entwickelte Ansatz zum Management der Dienstinteraktion bietet zudem Erweiterungsmöglichkeiten und ist auf andere Anwendungsbereiche übertragbar. Es ist z. B. denkbar, ein ähnliches Vorgehen zur Gewährleistung bestimmter Qualitätsattribute von Diensten anzuwenden.

Anwendungsdomäne eHomes

Es gibt eine Vielzahl von Forschungsansätzen, die sich aus unterschiedlichen Richtungen mit adaptiven Infrastrukturen befassen. Meist handelt es sich um serviceorientierte Ansätze, die die Abhängigkeiten zwischen Diensten verwalten. Dabei werden unter anderem auch Verteilungsinfrastrukturen eingesetzt oder agentenbasierte Ansätze verfolgt. Komponentenplattformen wie z. B. OSGi oder Webservice-Architekturen dienen häufig als Grundlage für die entwickelten Lösungen.

Viele Ansätze gehen jedoch nicht auf spezifische Anforderungen der späteren Einsatzbereiche ein. In dieser Arbeit wurde die *Anwendungsdomäne eHomes* betrachtet. So konnten Besonderheiten, wie die Art der Dienste oder die Kontextinformationen, die in eHomes von Bedeutung sind, berücksichtigt werden. Viele serviceorientierte Ansätze ermöglichen das dynamische Suchen und Binden von Diensten, stellen allerdings wenig Unterstützung für die Nutzung von Kontextinformationen zur Verfügung. Ein allgemeiner serviceorientierte Ansatz bietet für die Entwicklung von eHome-Systemen keine ausreichende Unterstützung. Der Kontextbezug von Diensten ist eine wesentliche Voraussetzung, sodass ein geeignetes Kontextmodell zur Verfügung gestellt werden muss. Dies wurde in dieser Arbeit durch das eHome-Modell umgesetzt.

Bindungsbeschränkungen erlauben es, auf die Elemente des eHome-Modells Bezug zu nehmen, sodass eine kontextbezogene Dienstkomposition ermöglicht wird. Auch für die semantische Dienstbeschreibung und die Adaption war der Fokus auf eHomes eine wichtige Voraussetzung. Die Modellierung der Ontologie wird durch die Einschränkung auf die Anwendungsdomäne eHomes und die typischen Basisfunktionalitäten erleichtert. Die Abbildbarkeit der syntaktischen Schnittstellenelemente hängt davon ab, wie gut die Ontologie die real vorkommenden Sachverhalte modelliert. Ein klar abgegrenztes Anwendungsgebiet ist dabei sehr hilfreich. Ein allgemeiner Ansatz zur Adaption von Diensten müsste hingegen zunächst sicherstellen, dass alle denkbaren Funktionalitäten einer Dienstschnittstelle eine Entsprechung in der Ontologie haben, was nur schwer realisierbar ist.

Werkzeugunterstützung

Schließlich ist die Werkzeugunterstützung zur manuellen Einflussnahme auf die Konfiguration zur Laufzeit ein wichtiger Aspekt dieser Arbeit. Die implementierten Werkzeugprototypen integrieren alle entwickelten Ansätze zu einem Gesamtprozess. Viele in der Literatur beschriebenen Ansätze basieren auf einem vollautomatischen Vorgehen. Im Bereich von eHomes ist dies jedoch nicht praxisbezogen, da die Bewohner die Kontrolle über ihre Umgebung behalten wollen. Daher muss die Möglichkeit der Interaktion in den Lösungsansatz einbezogen werden, wie es in dieser Arbeit umgesetzt wurde.

Eine Besonderheit ist die Unterstützung der Dienstspezifikation durch den Service-Editor. Dies ist insbesondere für die semantische Abbildung von Bedeutung. Häufig wird für die Entwicklung von Diensten keine gezielte Unterstützung angeboten. Gerade für die Dienstentwicklung ist dies aber wichtig, denn nur wenn es leicht möglich ist, neu entwickelte Dienste semantisch zu beschreiben, werden Ansätze wie die in dieser Arbeit vorgestellte semantische Adaption auch zur Anwendung kommen können.

Zur Entwicklung der Werkzeuge selbst wurden verschiedene Techniken eingesetzt. Zum einen wurden als Rahmenwerke die Eclipse Rich Client Platform und das Graphical Editing Framework verwendet, sodass auf viele bereits implementierte Standardkomponenten zurückgegriffen werden konnte. Die Architektur und die Anwendungslogik der Laufzeitumgebung wurden graphbasiert in Fujaba modelliert. Wie oben bereits erwähnt wäre hier eine weitere Unterstützung für die Entwicklung des Editors nützlich gewesen, damit die den Modellelementen entsprechenden View- und Controller-Elemente zumindest zum Teil hätten automatisch erzeugt werden können. Eine derartige Erweiterung für Fujaba könnte den Entwicklungsaufwand für grafische Editoren auf Basis von Fujaba-Modellen reduzieren.

7.3 Ausblick

Im Folgenden wird ein Ausblick auf zukünftige Erweiterungen der in dieser Arbeit vorgestellten Konzepte gegeben. Der Lösungsansatz wurde bisher in einer prototypischen Implementierung realisiert und anhand des eHome-Simulators getestet. Die Konzepte haben sich dabei als realisierbar erwiesen und konnten erfolgreich angewandt werden. Der kontinuierliche SCD-Prozess wird durch die entwickelten Werkzeuge abgedeckt.

Dennoch gibt es unterschiedliche Bereiche, die in dieser Arbeit nicht behandelt wurden. Diese bieten Potential für eine weitere Forschung im Bereich der Softwareunterstützung für eHomes und könnten als Folgearbeiten auf den bisher erarbeiteten Ergebnissen aufsetzen und diese ergänzen.

Spezifikation von Top-Level-Diensten als Workflows

Bisher ist im Ansatz der vorliegenden Arbeit vorgesehen, dass alle Dienste in Java implementiert und als OSGi-Bundles zur Verfügung gestellt werden. Die Dienstspezifikationen dienen als Beschreibungen dieser Implementierungen. In einer denkbaren Erweiterung des Ansatzes könnte aber auch die Anwendungslogik von Diensten in einer deklarativen Spezifikation festgelegt werden. Diese könnte ein höheres Abstraktionsniveau aufweisen als die Implementierung in einer Programmiersprache wie Java und könnte damit die Entwicklung von Diensten vereinfachen.

Insbesondere für die Entwicklung von Top-Level-Diensten erscheint ein solches Vorgehen sinnvoll. Eine mögliche Erweiterung wäre daher die Spezifikation von Top-Level-Diensten in Form von *Workflows*. Anstatt die gewünschten Abläufe zu implementieren, könnten diese durch geeignete Werkzeuge spezifiziert werden. Dabei könnten sogar direkt die semantischen Konzepte aus der eHome-Ontologie verwendet werden. Anstatt die Operationen der benötigten Funktionalitäten entsprechend einer syntaktischen Schnittstelle aufzurufen, könnten in den Workflows die Konzepte der Ontologie zugrunde gelegt werden. So wäre eine weitere Abstraktion möglich.

Denkbar ist, dass die Entwicklung, durch Werkzeuge unterstützt, soweit vereinfacht werden kann, dass auch die Nutzer selbst neue Dienste spezifizieren können. Dies würde den Bewohnern die Möglichkeit eröffnen, nicht nur bestehende Dienste den eigenen Wünschen entsprechend zu parametrisieren, sondern, den individuellen Anforderungen gemäß, ganz neue Dienste zu erstellen.

Frei definierbare Bindungsbeschränkungen

Die Dienstentwicklung betreffend wäre eine Flexibilisierung der in dieser Arbeit eingeführten Bindungsbeschränkungen eine mögliche Erweiterung. Bisher können zur Spezifikation von Diensten nur vordefinierte Bindungsbeschränkungen ausgewählt werden. Diese müssen zuvor in Fujaba spezifiziert worden sein und können dann in der Dienstbeschreibung ausgewählt werden.

Frei definierbare Bindungsbeschränkungen würden eine höhere Flexibilität bei der Dienstentwicklung ermöglichen. Dazu sollte die graphbasierte Spezifikation der Bindungsbeschränkungen beibehalten werden, da so die Vorteile des graphischen eHome-Modells ausgenutzt werden können. Denkbar wäre, dass der Dienstentwickler unter Verwendung von Fujaba eigene Bindungsbeschränkungen entwickelt, die dann nachträglich hinzugefügt werden könnten. Möglich wäre aber auch die Entwicklung eines einfacher zu bedienenden Werkzeugs, das eine graphische Bearbeitung von Bindungsbeschränkungen ermöglicht. Ein solches Werkzeug könnte aus Fujaba abgeleitet werden.

Semantisches Matching von Teilschnittstellen

Das semantische Matching erlaubt in der bisherigen Realisierung die Komposition von Diensten, die nur einen Teil einer in der Ontologie beschriebenen Funktionalität abdecken. Dies ist nötig, da nicht alle Schnittstellen den gesamten Umfang der in der Ontologie vorgesehenen Funktionalität zur Verfügung stellen können. Es ist jedoch bisher nicht möglich, eine benötigte Funktionalität durch die angebotenen Funktionalitäten *mehrerer* Dienste zu erfüllen. Dies kann in der Praxis auftreten, wenn z. B. ein Dienst eine Funktionalität in ihrem gesamten in der Ontologie spezifizierten Umfang benötigt, diese Funktionalität aber nur durch die Kombination mehrerer anbietender Dienste vollständig erfüllt werden kann. Dann wäre es sinnvoll, alle entsprechenden Dienste zu binden, um die benötigte Funktionalität so zu erfüllen, sofern es keine anderen Möglichkeiten einer Bindung gibt.

Berücksichtigung von Qualitätsattributen

Eine Erweiterung, die ebenfalls die Dienstkomposition betrifft, ist die detailliertere Betrachtung von Funktionalitäten. In dieser Arbeit wurde eine semantische Grundlage für die Dienstkomposition geschaffen. Funktionalitäten werden nicht mehr in erster Linie durch syntaktische Dienstschnittstellen repräsentiert, sondern durch die semantische Beschreibung und die in der Ontologie definierten Konzepte.

Funktionalitäten könnten aber durch *weitere Attribute* näher beschrieben werden, die dann bei der Dienstkomposition berücksichtigt würden und so bessere Matches ermöglichen könnten. Eine mögliche Erweiterung wäre z. B. die Betrachtung von Qualitätsattributen bei der Dienstkomposition. Nicht alle Dienste benötigen eine Funktionalität in gleicher Qualitätsausprägung. Während für die Ausgabe eines Signaltons ein kleiner Lautsprecher, wie er etwa in einem Handy integriert ist, durchaus genügt, ist für das Abspielen einer Sinfonie eine höhere Wiedergabequalität wünschenswert. Entsprechend dem jeweiligen Zweck ist daher die richtige Auswahl der verfügbaren Ressourcen entscheidend. Für den Nutzer ist es unbefriedigend, wenn die vorhandenen Ressourcen nicht zweckmäßig gebunden werden und er ggfs. regelmäßig manuell nachkonfigurieren muss. Die Auszeichnung angebotener und benötigter Funktionalitäten mit Qualitätsattributen könnte daher zur Verbesserung der Auswahl geeigneter Dienste während der Konfigurierung beitragen.

Weiterführende Evaluation

Die Anbindung realer Wohnumgebungen an den bisher realisierten Lösungsansatz würde eine weiterführende Auswertung der entwickelten Konzepte ermöglichen. Die in der Abgrenzung in Abschnitt 1.2 beschriebenen Bereiche, die in dieser Arbeit ausgespart wurden, stellen den Kontext des hier entwickelten Systems dar. Diese Aspekte müssen für die praktische Anwendung ebenfalls betrachtet werden.

Es muss also z. B. der Bereich der *automatischen Kontexterfassung* näher untersucht werden. Die zahlreichen Mechanismen zur Lokalisierung von Personen und Objekten in eHomes müssen an die Laufzeitumgebung angebunden werden, um die strukturelle Adaption bei Bedarf automatisch durchzuführen. Eine Integration solcher Mechanismen ist über das eHome-Modell möglich.

Ebenso muss die Anbindung der verschiedenen *Hardware- und Infrastrukturtechnologien* realisiert werden. Die Steuerung der Geräte wird durch Treiberdienste verkapselt. Tiefer liegende Abstraktionsschichten wurden jedoch in dieser Arbeit nicht betrachtet. Sinnvoll erscheint die Entwicklung von Komponenten zur Abstraktion von Details der verwendeten Infrastrukturen, z. B. der X10- oder KNX-Technologie. Dies erleichtert die Entwicklung von Treiberdiensten.

Die Entwicklung neuer Ansätze, die diese Aspekte unterstützen, erfordert weitere Forschung. Schließlich muss auch eine weiterführende Analyse der Anwendungen im eHome, die durch Top-Level-Dienste realisiert werden, stattfinden. Es stellt sich die Frage, welche Dienste von den Menschen angenommen werden und welche Funktionalitäten weniger gewünscht sind. Endgültige Antworten dazu kann jedoch nicht die Softwaretechnik oder Informatik im Allgemeinen geben. Diese Fragestellungen werden erst nach einer breiten Verfügbarkeit und Anwendung von eHomes in der Praxis abschließend zu beantworten sein.

Abbildungsverzeichnis

Listingverzeichnis

Literaturverzeichnis

[ADB+99] G. D. ABOWD, A. K. DEY, P. J. BROWN, N. DAVIES, M. SMITH und P. STEGGLES: *Towards a Better Understanding of Context and Context-Awareness*. In: H.-W. GELLERSEN (Herausgeber): *Handheld and Ubiquitous Computing – First International Symposium, HUC'99*, Band 1707 der Reihe *LNCS*, Seiten 304–307. Springer-Verlag, 1999.

[AE08] I. ARMAÇ und D. EVERS: *Client Side Personalization of Smart Environments*. In: *SAM '08: Proceedings of the 1st International Workshop on Software Architectures and Mobility*, Seiten 57–59. ACM Press, 2008.

[AEH+99] M. ANDRIES, G. ENGELS, A. HABEL, B. HOFFMANN, H. KREOWSKI, S. KUSKE, D. PLUMP, A. SCHÜRR und G. TAENTZER: *Graph Transformation for Specification and Programming*. Science of Computer Programming, 34(1):1–54, April 1999.

[AEHS06] M. ALIA, F. ELIASSEN, S. HALLSTEINSEN und E. STAV: *MADAM: Towards a Flexible Planning-based Middleware*. In: *International Conference on Software Engineering (ICSE), Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems (SEAMS'06)*, Seite 96. ACM Press, 2006.

[Ait03] E. AITENBICHLER: *Ortungssysteme für mobile Endgeräte*. In: K. R. DITTRICH, W. KÖNIG, A. OBERWEIS, K. RANNENBERG und W. WAHLSTER (Herausgeber): *INFORMATIK 2003 – Innovative Informatikanwendungen, Beiträge der 33. Jahrestagung der Gesellschaft für Informatik e. V.*, Band 34 der Reihe *LNI*, Seiten 202–206. Gesellschaft für Informatik, 2003.

[AM00] G. D. ABOWD und E. D. MYNATT: *Charting Past, Present, and Future Research in Ubiquitous Computing*. ACM Transactions on Computer-Human Interaction (TOCHI), 7(1):29–58, ACM Press, 2000.

[Ami04] AMIGO PROJECT: *Amigo – Short project description*. Technischer Bericht IST-004182, Amigo Project, 2004.

[Ami08] AMIGO PROJECT: *Amigo Project – Homepage*. `http://www.hitech-projects.com/euprojects/amigo/`, 2008.

[APPR09] I. ARMAÇ, A. PANCHENKO, M. PETTAU und D. RETKOWITZ: *Privacy-Friendly Smart Environments*. In: K. AL-BEGAIN (Herausgeber): *Third International Conference on Next Generation Mobile Applications, Services and Technologies (NGMAST 2009)*, Seiten 425–431. IEEE Computer Society, 2009.

[AR07] I. ARMAÇ und D. RETKOWITZ: *Simulation of Smart Environments*. In: *Proceedings of the IEEE International Conference on Pervasive Services 2007 (ICPS'07)*, Seiten 257–266. IEEE Computer Society, 2007.

[AR08] I. ARMAÇ und D. ROSE: *Privacy-Friendly User Modelling for Smart Environments*. In: *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous 2008)*, Seiten 1–6, 2008.

[Arm08] I. ARMAÇ: *Protecting the Privacy of Mobile eHome Users*. In: *Proceedings des gemeinsamen Workshops der Graduiertenkollegs 2008*, Seiten 11–12. GITO, 2008.

[Awa09] AWARE HOME RESEARCH INITIATIVE, GEORGIA INSTITUTE OF TECHNOLOGY: *Aware Home Research Initiative – Homepage*. `http://awarehome.imtc.gatech.edu/`, 2009.

[Bal05] H. BALZERT: *UML 2 kompakt*. Spektrum Akademischer Verlag, 2. Auflage, 2005.

[BB02] G. BANAVAR und A. BERNSTEIN: *Software Infrastructure and Design Challenges for Ubiquitous Computing Applications*. Communications of the ACM, 45(12):92–96, ACM Press, Dezember 2002.

[BBEL07] A. BOTTARO, J. BOURCIER, C. ESCOFFIER und P. LALANDA: *Context-Aware Service Composition in a Home Control Gateway*. In: *IEEE International Conference on Pervasive Services (ICPS'07)*, Seiten 223–231. IEEE Computer Society, 2007.

[BBG$^+$06] S. BECKER, A. BROGI, I. GORTON, S. OVERHAGE, A. ROMANOVSKY und M. TIVOLI: *Towards an Engineering Approach to Component Adaptation*. In: R. H. REUSSNER, J. A. STAFFORD und C. A. SZYPERSKI (Herausgeber): *Architecting Systems with Trustworthy Components*, Band 3938 der Reihe *LNCS*, Seiten 193–215. Springer-Verlag, 2006.

[BC08] D. BONINO und F. CORNO: *DogOnt – Ontology Modeling for Intelligent Domotic Environments*. In: A. SHETH, S. STAAB, M. DEAN, M. PAOLUCCI, D. MAYNARD, T. FININ und K. THIRUNARAYAN (Herausgeber): *The Semantic Web – ISWC 2008, 7th International Semantic Web Conference*, Band 5318 der Reihe *LNCS*, Seiten 790–803. Springer-Verlag, Oktober 2008.

[BCC08a] D. BONINO, E. CASTELLINA und F. CORNO: *DOG: An Ontology-Powered OSGi Domotic Gateway*. In: *20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'08)*, Seiten 157–160. IEEE Computer Society, November 2008.

[BCC08b] D. BONINO, E. CASTELLINA und F. CORNO: *The DOG Gateway: Enabling Ontology-based Intelligent Domotic Environments*. IEEE Transactions on Consumer Electronics, 54(4):1656–1664, IEEE Consumer Electronics Society, November 2008.

[BCC08c] D. BONINO, E. CASTELLINA und F. CORNO: *Uniform Access to Domotic Environments through Semantics*. In: A. GANGEMI, J. KEIZER, V. PRESUTTI und H. STOERMER (Herausgeber): *Proceedings of the 5th Workshop on Semantic Web Applications and Perspectives (SWAP2008)*. CEUR Workshop Proceedings, Dezember 2008.

[BCGR08] M. BROY, M. V. CENGARLE, H. GRÖNNIGER und B. RUMPE: *Modular Description of a Comprehensive Semantics Model for the UML (Version 2.0)*. Informatik-Bericht 2008-06, Technische Universität Braunschweig, Carl-Friedrich-Gauss-Fakultät, Oktober 2008.

[BCGR09a] M. BROY, M. V. CENGARLE, H. GRÖNNIGER und B. RUMPE: *Considerations and Rationale for a UML System Model*, Seiten 43–60. In: K. LANO [Lan09], 2009.

[BCGR09b] M. BROY, M. V. CENGARLE, H. GRÖNNIGER und B. RUMPE: *Definition of the System Model*, Seiten 61–93. In: K. LANO [Lan09], 2009.

[BD06] T. BUCHMANN und A. DOTOR: *Building Graphical Editors with GEF and Fujaba*. In: H. GIESE und B. WESTFECHTEL (Herausgeber): *Proceedings of the Fujaba Days 2006*, Band tr-ri-06-275, Seiten 47–51. Universität Paderborn, 2006.

[BDH+98] M. BROY, A. DEIMEL, J. HENN, K. KOSKIMIES, F. PLÁŠIL, G. POMBERGER, W. PREE, M. STAL und C. SZYPERSKI: *What characterizes a (software) component?* Software – Concepts & Tools, 19(1):49–56, Springer-Verlag, Juni 1998.

[BG06] A. BOTTARO und A. GÉRODOLLE: *Extended Service Binder: Dynamic Service Availability Management in Ambient Intelligence*. FRCSS'06: Workshop on Future Research Challenges for Software and Services, Communications of the EASST, April 2006.

[BGI05] S. BEN MOKHTAR, N. GEORGANTAS und V. ISSARNY: *Ad Hoc Composition of User Tasks in Pervasive Computing Environments*. In: T. GSCHWIND, U. ASSMANN und O. NIERSTRASZ (Herausgeber): *Software Composition – 4th International Workshop, SC 2005*, Band 3628 der Reihe *LNCS*, Seiten 31–46. Springer-Verlag, September 2005.

[BGI07] S. BEN MOKHTAR, N. GEORGANTAS und V. ISSARNY: *COCOA: COnversation-based service COmposition in pervAsive computing environments with QoS support*. Journal of Systems and Software, 80(12):1941–1955, Elsevier, Dezember 2007.

[Böh99] B. BÖHLEN: *Basisschicht eines Rahmenwerks für graphbasierte Anwendungen*. Diplomarbeit, RWTH Aachen, 1999.

[BH07] A. BOTTARO und R. S. HALL: *Dynamic Contextual Service Ranking*. In: M. LUMPE und W. VANDEPERREN (Herausgeber): *6th Internatio-*

nal Symposium on Software Composition, SC 2007, Band 4829 der Reihe *LNCS*, Seiten 126–140. Springer-Verlag, März 2007.

[BHL01] T. BERNERS-LEE, J. HENDLER und O. LASSILA: *The Semantic Web*. Scientific American, 284(5):34–43, Verlagsgruppe Georg von Holtzbrinck, 2001.

[BJSW02] B. BÖHLEN, D. JÄGER, A. SCHLEICHER und B. WESTFECHTEL: *UPGRADE: A Framework for Building Graph-Based Interactive Tools*. GraBaTs 2002, Graph-Based Tools, 1st International Conference on Graph Transformation (ICGT), Electronic Notes in Theoretical Computer Science, 72(2):91–101, Elsevier, 2002.

[BME+07] G. BOOCH, R. A. MAKSIMCHUK, M. ENGLE, B. YOUNG, J. CONALLEN und K. HOUSTON: *Object Oriented Analysis and Design with Applications*. Addison-Wesley Longman, 3. Auflage, 2007.

[BMR+98] F. BUSCHMANN, R. MEUNIER, H. ROHNERT, P. SOMMERLAD und M. STAL: *Pattern-orientierte Software-Architektur: Ein Pattern-System*. Addison-Wesley, 1998.

[BPG+08] S. BEN MOKHTAR, D. PREUVENEERS, N. GEORGANTAS, V. ISSARNY und Y. BERBERS: *EASY: Efficient semAntic Service discoverY in pervasive computing environments with QoS and context support*. Journal of Systems and Software, 81(5):785–808, Elsevier, Mai 2008.

[BQvS07] T. H. F. BROENS, D. A. C. QUARTEL und M. J. VAN SINDEREN: *Towards a Context Binding Transparency*. In: *Dependable and Adaptable Networks and Services – Proceedings of the 13th Open European Summer School and IFIP TC6.6 Workshop, EUNICE 2007, Enschede, The Netherlands*, Band 4606 der Reihe *LNCS*, Seiten 9–16. Springer-Verlag, Juli 2007.

[Bru07] S. BRUNN: *Amerikanischer Stil*. Technology Review, Seiten 10–11, Heise Zeitschriften Verlag, März 2007.

[BSG+06] J. BIZER, S. SPIEKERMANN, O. GÜNTHER et al.: *TAUCIS – Technikfolgenabschätzung Ubiquitäres Computing und Informationelle Selbstbestimmung*. Studie im Auftrag des Bundesministeriums für Bildung und Forschung, Unabhängiges Landeszentrum für Daten-schutz Schleswig-Holstein und Institut für Wirtschaftsinformatik der Humboldt-Universität zu Berlin, Juli 2006.

[Bun08] BUNDESMINISTERIUM FÜR BILDUNG UND FORSCHUNG: *AAL – Altersgerechte Assistenzsysteme für ein gesundes und unabhängiges Leben, Ambient Assisted Living*. Broschüre, Bundesministerium für Bildung und Forschung (BMBF), Bonn und Berlin, 2008.

[BvHvS06] T. H. F. BROENS, A. T. VAN HALTEREN und M. J. VAN SINDEREN: *Infrastructural Support for Dynamic Context Bindings*. In: *Proceedings of the First European Conference on Smart Sensing and Context (EuroSSC'06)*, Band 4272 der Reihe *LNCS*, Seiten 82–97. Springer-Verlag, Oktober 2006.

[CCHW05] A. COLYER, A. CLEMENT, G. HARLEY und M. WEBSTER: *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley, Januar 2005.

[CE00] K. CZARNECKI und U. W. EISENECKER: *Generative Programming, Methods, Tools, and Applications*. Addison-Wesley Longman, 2000.

[CH03] H. CERVANTES und R. S. HALL: *Automating Service Dependency Management in a Service-Oriented Component Model*. In: I. CRNKOVIC, H. SCHMIDT, J. STAFFORD und K. WALLNAU (Herausgeber): *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, Seiten 379–382. IEEE Press, Mai 2003.

[CH04a] H. CERVANTES und R. S. HALL: *A Framework for Constructing Adaptive Component-Based Applications: Concepts and Experiences*. In: *Component-Based Software Engineering, 7th International Symposium, CBSE 2004*, Band 3054 der Reihe *LNCS*, Seiten 130–137. Springer-Verlag, 2004.

[CH04b] H. CERVANTES und R. S. HALL: *Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model*. In: *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, Seiten 614–623. IEEE Press, Juli 2004.

[Chi00] S. CHIBA: *Load-Time Structural Reflection in Java*. In: E. BERTINO (Herausgeber): *ECOOP 2000 – Object-Oriented Programming, 14th European Conference Sophia Antipolis and Cannes, France, June 12-16, 2000 Proceedings*, Band 1850 der Reihe *LNCS*, Seiten 313–336. Springer-Verlag, 2000.

[Chi09] S. CHIBA: *Javassist – Homepage*. `http://www.csg.is.titech.ac.jp/~chiba/javassist/`, 2009.

[CJB99] B. CHANDRASEKARAN, J. R. JOSEPHSON und V. R. BENJAMINS: *What Are Ontologies, and Why Do We Need Them?* IEEE Intelligent Systems, 14(1):20–26, IEEE Educational Activities Department, 1999.

[dCYG08] C. A. DA COSTA, A. C. YAMIN und C. F. R. GEYER: *Toward a General Software Infrastructure for Ubiquitous Computing*. IEEE Pervasive Computing, 7(1):64–73, IEEE Press, Januar 2008.

[Deu09] DEUTSCHE TELEKOM AG, T-COM ZENTRALE: *T-Com Haus – Homepage*. `http://t-com-haus.i-dmedia.com/`, 2009.

[dRW04] J. DES RIVIÈRES und J. WIEGAND: *Eclipse: A Platform for Integrating Development Tools*. IBM Systems Journal, 43(2):371–383, IBM Corporation, 2004.

[EEHT05] K. EHRIG, C. ERMEL, S. HÄNSGEN und G. TAENTZER: *Generation of Visual Editors as Eclipse Plug-Ins*. In: *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, Seiten 134–143. ACM Press, 2005.

[EEKR99] H. EHRIG, G. ENGELS, H. KREOWSKI und G. ROZENBERG (Herausgeber): *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, Band 2. World Scientific, Singapur, 1999.

[EF03] A. EBERHART und S. FISCHER: *Web Services – Grundlagen und praktische Umsetzung mit J2EE und .NET*. Carl Hanser Verlag, 2003.

[EFLR98] A. EVANS, R. FRANCE, K. LANO und B. RUMPE: *Developing the UML as a Formal Modelling Notation*. In: P. MULLER und J. BÉZIVIN (Herausgeber): *The Unified Modeling Language – Workshop*

UML'98: Beyond the Notation. Ecole Superieure Mulhouse, Université de Haute-Alsace, 1998.

[EFLR99] A. EVANS, R. B. FRANCE, K. LANO und B. RUMPE: *The UML as a Formal Modeling Notation*. In: *«UML»'98 Beyond the Notation – First International Workshop Mulhouse, France*, Band 1618 der Reihe *LNCS*, Seiten 336–348. Springer-Verlag, 1999.

[EG01] W. K. EDWARDS und R. E. GRINTER: *At Home with Ubiquitous Computing: Seven Challenges*. In: *UbiComp'01: Proceedings of the 3rd International Conference on Ubiquitous Computing*, Seiten 256–272. Springer-Verlag, 2001.

[EH07] C. ESCOFFIER und R. S. HALL: *Dynamically Adaptable Applications with iPOJO Service Components*. In: M. LUMPE und W. VANDERPERREN (Herausgeber): *Software Composition – 6th International Symposium, SC 2007*, Band 4829 der Reihe *LNCS*, Seiten 113–128. Springer-Verlag, 2007.

[EHL07] C. ESCOFFIER, R. S. HALL und P. LALANDA: *iPOJO: an Extensible Service-Oriented Component Framework*. In: *IEEE International Conference on Services Computing (SCC 2007)*, Seiten 474–481. IEEE Computer Society, 2007.

[EKS+04] A. ECKHARDT, A. KEEL, A. SCHÖNENBERGER, F. BUFFON und M. OBERHOLZER: *Telemedizin – Studie des Zentrums für Technologiefolgen-Abschätzung*. Bericht TA 49/2004, Zentrum für Technologiefolgen-Abschätzung beim Schweizerischen Wissenschafts- und Technologierat, Birkenweg 61, CH-3003 Bern, Schweiz, September 2004.

[ELFR99] A. EVANS, K. LANO, R. FRANCE und B. RUMPE: *Meta-Modeling Semantics of UML*. In: H. KILOV, B. RUMPE und I. SIMMONDS (Herausgeber): *Behavioral Specifications of Businesses and Systems*. Kluver Academic Publisher, 1999.

[Eur02] EUROPÄISCHE KOMMISSION: *The Sixth Framework Programme in brief*. Bericht, European Commission, Community Research and Development Information Service (CORDIS), Dezember 2002.

[Eur03] EUROPÄISCHE KOMMISSION: *IST 2003 – The Opportunities Ahead*. Bericht, European Commission, Directorate-General Information Society, 2003.

[FFSB04] E. FREEMAN, E. FREEMAN, K. SIERRA und B. BATES: *Head First Design Patterns*. O'Reilly Media, 1. Auflage, Oktober 2004.

[FM05] E. FLEISCH und F. MATTERN (Herausgeber): *Das Internet der Dinge – Ubiquitous Computing und RFID in der Praxis: Visionen, Technologien, Anwendungen, Handlungsanleitungen*. Springer-Verlag, 2005.

[FMRS07] C. FUSS, C. MOSLER, U. RANGER und E. SCHULTCHEN: *The Jury is still out: A Comparison of AGG, Fujaba, and PROGRES*. In: T. MARGARIA, J. PADBERG, G. TAENTZER, K. EHRIG und H. GIESE (Herausgeber): *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, Band 6, Seite 14. Electronic Communications of the EASST, 2007.

[FNTZ98] T. FISCHER, J. NIERE, L. TORUNSKI und A. ZÜNDORF: *Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language*. In: G. ENGELS und G. ROZENBERG (Herausgeber): *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, Band 1764 der Reihe *LNCS*, Seiten 296–309. Springer-Verlag, November 1998.

[For82] C. L. FORGY: *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*. Artificial Intelligence, 19:17–37, 1982.

[Fow04] M. FOWLER: *Inversion of Control Containers and the Dependency Injection pattern*. `http://martinfowler.com/articles/injection.html`, 2004.

[FR07] R. FRANCE und B. RUMPE: *Model-driven Development of Complex Software: A Research Roadmap*. In: *Future of Software Engineering 2007 (FOSE'07) at ICSE,* Seiten 37–54. IEEE Computer Society, Mai 2007.

[Fra09] FRAUNHOFER INHAUS-ZENTRUM: *inHaus – Innovationszentrum der Fraunhofer-Gesellschaft – Homepage*. `http://www.inhaus-zentrum.de/`, 2009.

[Fri03] E. FRIEDMAN-HILL: *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Oktober 2003.

[Fro08] R. FROTSCHER: *Adaptives Kontextmanagement für dynamische eHomes*. Bachelorarbeit, FH Aachen, Standort Jülich, Oktober 2008.

[GFC04] A. GÓMEZ-PÉREZ, M. FERNÁNDEZ-LÓPEZ und O. CORCHO: *Ontological Engineering: With Examples from the Areas of Knowledge Management, e-Commerce and the Semantic Web*. Springer-Verlag, 2004.

[GHJV95] E. GAMMA, R. HELM, R. JOHNSON und J. VLISSIDES: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1995.

[GMB$^+$05] N. GEORGANTAS, S. B. MOKHTAR, Y. BROMBERG, V. ISSARNY, J. KALAOJA, J. KANTAROVITCH, A. GERODOLLE und R. MEVISSEN: *The Amigo Service Architecture for the Open Networked Home Environment*. In: *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, Seiten 295–296. IEEE Press, 2005.

[GMF$^+$02] J. H. GENNARI, M. A. MUSEN, R. W. FERGERSON, W. E. GROSSO, M. CRUBZY, H. ERIKSSON, N. F. NOY und S. W. TU: *The Evolution of Protégé: An Environment for Knowledge-Based Systems Development*. International Journal of Human-Computer Studies, 58:89–123, 2002.

[GPR06] V. GRUHN, D. PIEPER und C. RÖTTGERS: *MDA – Effektives Software-Engineering mit UML 2 und Eclipse*. Xpert.press. Springer-Verlag, 2006.

[GR07] K. GEIHS und R. REICHLE: *Development of Self-Adaptive Services*. In: *14th Workshop of the HP Software University Association (HPSUA)*, Juli 2007.

[Gra90] R. GRABHERR: *Wohnen im Computer – Haustechnik von morgen: Alptraum oder traumhaft?* Computer Live, Seiten 10–14, Markt & Technik Verlag, Juli 1990.

[Gru93] T. R. GRUBER: *A Translation Approach to Portable Ontology Specifications*. Knowledge Acquisition, 5(2):199–220, Academic Press Ltd., 1993.

[Gua98] N. GUARINO: *Formal Ontology and Information Systems*. In: N. GUARINO (Herausgeber): *Formal Ontology in Information Systems. Proceedings of FOIS'98, Trento, Italy, 6-8 June 1998*, Seiten 3–15, Amsterdam, Niederlande, 1998. IOS Press.

[GVR+07] A. GÉRODOLLE, M. VALLÉE, I. ROUSSAKI, D. TSESMETZIS, I. PAPAIOANNOU, M. ANAGNOSTOU, E. NAROSKA, G. THOMSON, S. BIANCO, N. GEORGANTAS, S. B. MOKHTAR, V. ISSARNY et al.: *Amigo overall middleware: Final prototype implementation and documentation*. Deliverable D3.5, Amigo Project, Dezember 2007.

[Ham97] G. HAMILTON: *JavaBeans – Version 1.01-A*. Spezifikation, Sun Microsystems Inc., August 1997.

[HC03] R. S. HALL und H. CERVANTES: *Gravity: Supporting Dynamically Available Services in Client-Side Applications*. In: *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Seiten 379–382. ACM Press, 2003.

[Hec06] R. HECKEL: *Graph Transformation in a Nutshell*. Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004), Electronic Notes in Theoretical Computer Science, 148(1):187–198, Elsevier, 2006.

[Hei98] G. T. HEINEMAN: *A Model for Designing Adaptable Software Components*. In: *Proceedings of the 22nd Annual International Computer Software and Applications Conference (COMPSAC '98)*, Seiten 121–127. IEEE Press, 1998.

[HMNS03] U. HANSMANN, L. MERK, M. S. NICKLOUS und T. STOBER: *Pervasive Computing – The Mobile World*. Springer-Verlag, 2. Auflage, 2003.

[Hof09] M. HOFFMANN: *Ontologiebasierte Komposition von eHome-Diensten*. Bachelorarbeit, RWTH Aachen, September 2009.

[Hom05] HOMEPLUG POWERLINE ALLIANCE, INC.: *HomePlug 1.0 Technology White Paper*. Technischer Bericht, Februar 2005.

[Hou09] HOUSE_N RESEARCH GROUP, DEPARTMENT OF ARCHITECTURE, MASSACHUSETTS INSTITUTE OF TECHNOLOGY: *MIT House_n – Homepage*. http://architecture.mit.edu/house_n/, 2009.

[Höp09] F. HÖPFLINGER (Herausgeber): *Age Report 2009 – Einblicke und Ausblicke zum Wohnen im Alter*. Seismo Verlag, Zürich, Schweiz, 2009.

[HR00] D. HAREL und B. RUMPE: *Modeling Languages: Syntax, Semantics and All That Stuff – Part I: The Basic Stuff*. Technischer Bericht MCS00-16, Faculty of Mathematics and Computer Science, Weizmann Institute of Science, Israel, Oktober 2000.

[HR04] D. HAREL und B. RUMPE: *Meaningful Modeling: What's the Semantics of "Semantics"?* IEEE Computer, 37(10):64–72, IEEE Computer Society, 2004.

[HRK08a] T. HEER, D. RETKOWITZ und B. KRAFT: *Algorithm and Tool for Ontology Integration based on Graph Rewriting*. In: A. SCHÜRR, M. NAGL und A. ZÜNDORF (Herausgeber): *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007)*, Band 5088 der Reihe *LNCS*, Seiten 577–582. Springer-Verlag, 2008.

[HRK08b] T. HEER, D. RETKOWITZ und B. KRAFT: *Incremental Ontology Integration*. In: J. CORDEIRO und J. FILIPE (Herausgeber): *ICEIS (3-1), 10th International Conference on Enterprise Information Systems (ICEIS 2008)*, Seiten 13–20, 2008.

[HRK09] T. HEER, D. RETKOWITZ und B. KRAFT: *Tool Support for the Integration of Light-Weight Ontologies*. In: J. FILIPE und J. CORDEIRO (Herausgeber): *Enterprise Information Systems, 10th International Conference, ICEIS 2008, Barcelona, Spain, June 12-16, 2008, Revised Selected Papers*, Band 19 der Reihe *LNBIP*, Seiten 175–187. Springer-Verlag, 2009.

[HS05] M. N. HUHNS und M. P. SINGH: *Service-Oriented Computing: Key Concepts and Principles*. IEEE Internet Computing, 9(1):75–81, 2005.

[Int05] INTERNATIONAL TELECOMMUNICATION UNION (ITU): *ITU Internet Reports 2005: The Internet of Things*. Executive Summary, International Telecommunication Union (ITU), November 2005.

[Int06] S. S. INTILLE: *The Goal: Smart People, Not Smart Homes*. In: *Proceedings of the International Conference on Smart Homes and Health Telematics*. IOS Press, 2006.

[IST01] IST ADVISORY GROUP (ISTAG), EUROPEAN COMMISSION: *Scenarios for Ambient Intelligence in 2010*. Bericht, European Commission, Community Research and Development Information Service (CORDIS), Februar 2001.

[JSW00] D. JÄGER, A. SCHLEICHER und B. WESTFECHTEL: *AHEAD: A Graph-Based System for Modeling and Managing Development Processes*. In: M. NAGL, A. SCHÜRR und M. MÜNCH (Herausgeber): *Applications of Graph Transformations with Industrial Relevance – International Workshop AGTIVE'99*, Band 1779 der Reihe *LNCS*, Seiten 325–340. Springer-Verlag, 2000.

[KBW07] P. KUNZMANN, F.-P. BURKARD und F. WIEDMANN: *dtv-Atlas zur Philosophie*. Deutscher Taschenbuch Verlag, 13. Auflage, 2007.

[KHH+01] G. KICZALES, E. HILSDALE, J. HUGUNIN, M. KERSTEN, J. PALM und W. G. GRISWOLD: *An Overview of AspectJ*. In: J. L. KNUDSEN (Herausgeber): *ECOOP 2001 – Object-Oriented Programming, 15th European Conference*, Band 2072 der Reihe *LNCS*, Seiten 327–354. Springer-Verlag, 2001.

[Kir05] M. KIRCHHOF: *Integrierte Low-Cost eHome-Systeme – Prozesse und Infrastrukturen*. Dissertation, RWTH Aachen, Dezember 2005.

[KKNP08] J. KANTOROVITCH, J. KALAOJA, I. NISKANEN und T. PIIRAINEN: *Towards a Better Understanding of Semantic Ontology-Based Home Service Modelling*. In: *22nd International Conference on Advanced Information Networking and Applications (AINA 2008)*, Seiten 26–31. IEEE Computer Society, März 2008.

[KL89] M. KIFER und G. LAUSEN: *F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme*. ACM SIGMOD Record, 18(2):134–146, ACM Press, 1989.

[KLM+97] G. KICZALES, J. LAMPING, A. MENDHEKAR, C. MAEDA, C. LOPES, J.-M. LOINGTIER und J. IRWIN: *Aspect-Oriented Programming*. In: M. AKŞIT und S. MATSUOKA (Herausgeber): *ECOOP'97 – Object-Oriented Programming, 11th European Conference*, Band 1241 der Reihe *LNCS*, Seiten 220–242. Springer-Verlag, 1997.

[KNNZ00] H. J. KÖHLER, U. NICKEL, J. NIERE und A. ZÜNDORF: *Integrating UML Diagrams for Production Control Systems*. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, Seiten 241–251. ACM Press, 2000.

[KR97] H. KILOV und B. RUMPE: *Summary of ECOOP'97 Workshop on Precise Semantics of Object-Oriented Modeling Techniques*. In: J. BOSCH und S. MITCHELL (Herausgeber): *Object-Oriented Technology – ECOOP'97 Workshop Reader*, Band 1357 der Reihe *LNCS*. Springer-Verlag, 1997.

[KR05] B. KRAFT und D. RETKOWITZ: *Operationale Semantikdefinition für Konzeptuelles Regelwissen*. In: L. WEBER und F. SCHLEY (Herausgeber): *Forum Bauinformatik 2005*, Seiten 173–182. Lehrstuhl Bauinformatik, BTU Cottbus, 2005.

[KR06a] B. KRAFT und D. RETKOWITZ: *Graph Transformations for Dynamic Knowledge Processing*. In: E. ROBICHAUD (Herausgeber): *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS'06)*, Seiten 1–10. IEEE Computer Society, 2006.

[KR06b] B. KRAFT und D. RETKOWITZ: *Rule-Dependencies for Visual Knowledge Specification in Conceptual Design*. In: H. RIVARD (Herausgeber): *Proceedings of the 11th International Conference on Computing in Civil and Building Engineering (ICCCBE-XI)*, Seiten 1–12. ACSE, 2006.

[Kra07] B. KRAFT: *Semantische Unterstützung des konzeptuellen Gebäudeentwurfs*. Dissertation, RWTH Aachen, März 2007.

[Kul09] S. Kulle: *Dynamische Bindungsverwaltung zur bedarfsorientierten Ressourcennutzung von eHome-Diensten*. Diplomarbeit, RWTH Aachen, April 2009.

[Lan09] K. Lano (Herausgeber): *UML 2 Semantics and Applications*. John Wiley & Sons, 2009.

[LFJ07] X. Li, Y. Fan und F. Jiang: *A Classification of Service Composition Mismatches to Support Service Mediation*. In: *The 6th International Conference on Grid and Cooperative Computing (GCC 2007)*, Seiten 315–321. IEEE Press, August 2007.

[LW05] K.-K. Lau und Z. Wang: *A Taxonomy of Software Component Models*. In: *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, Seiten 88–95. IEEE Computer Society, 2005.

[LW07] K.-K. Lau und Z. Wang: *Software Component Models*. IEEE Transactions on Software Engineering, 33(10):709–724, IEEE Computer Society, 2007.

[Mat03] F. Mattern (Herausgeber): *Total vernetzt – Szenarien einer informatisierten Welt*. Springer-Verlag, 2003.

[MDG$^+$04] B. Moore, D. Dean, A. Gerber, G. Wagenknecht und P. Vanderheyden: *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. IBM Corporation, 1. Auflage, Februar 2004.

[Mey97] B. Meyer: *Object-Oriented Software Construction*. Prentice Hall, 2. Auflage, 1997.

[Min75] M. Minsky: *Minsky's frame system theory*. In: *TINLAP '75: Proceedings of the 1975 workshop on Theoretical issues in natural language processing*, Seiten 104–116. Association for Computational Linguistics, 1975.

[ML05] J. McAffer und J. Lemieux: *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. The Eclipse Series. Addison-Wesley, Oktober 2005.

[MLM+06] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown und R. Metz: *Reference Model for Service Oriented Architecture 1.0*. Organization for the Advancement of Structured Information Standards (OASIS), Oktober 2006.

[MM01] J. Miller und J. Mukerji: *Model Driven Architecture (MDA)*. Document Number: ormsc/2001-07-01, Object Management Group (OMG), Juli 2001.

[MM03] J. Miller und J. Mukerji: *MDA Guide Version 1.0.1*. Document Number: omg/2003-06-01, Object Management Group (OMG), Juni 2003.

[Mos09] C. Mosler: *Graphbasiertes Reengineering von Telekommunikationssystemen*. Dissertation, RWTH Aachen, Juni 2009.

[Moz05] M. C. Mozer: *Lessons from an Adaptive Home*. In: D. Cook und R. Das (Herausgeber): *Smart environments: Technologies, protocols, and applications*, Seiten 273–294. John Wiley & Sons, 2005.

[MSKC04] P. K. McKinley, S. M. Sadjadi, E. P. Kasten und B. H. C. Cheng: *A Taxonomy of Compositional Adaptation*. Technischer Bericht MSU-CSE-04-17, Department of Computer Science, Michigan State University, East Lansing, Michigan, USA, Mai 2004.

[MSS08] S. Müller, M. Santi und A. Sixsmith: *Eliciting user requirements for Ambient Assisted Living: Results of the SOPRANO project*. In: P. Cunningham und M. Cunnighamn (Herausgeber): *Collaboration and the Knowledge Economy: Issues, Applications, Case Studies, Part 1*, Seiten 81–88. IOS Press, 2008.

[Nag90] M. Nagl: *Softwaretechnik – Methodisches Programmieren im Großen*. Springer-Verlag, 1990.

[Nag96] M. Nagl (Herausgeber): *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, Band 1170 der Reihe *LNCS*. Springer-Verlag, 1996.

[NARS06] U. Norbisrath, I. Armaç, D. Retkowitz und P. Salumaa: *Modeling eHome Systems*. In: S. Terzis (Herausgeber): *Proceedings of the 4th International Workshop on Middleware for Pervasive and Ad-Hoc*

Computing (MPAC 2006) held at the 7th International Middleware Conference, Seiten 1–6. ACM Press, 2006.

[NLLP04] L. M. NI, Y. LIU, Y. C. LAU und A. P. PATIL: *LANDMARC: Indoor Location Sensing Using Active RFID*. Wireless Networks, 10(6):701–710, Springer Netherlands, November 2004.

[NMA06] U. NORBISRATH, C. MOSLER und I. ARMAÇ: *The eHomeConfigurator Tool Suite*. In: R. MEERSMAN, Z. TARI und P. HERRERO (Herausgeber): *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, Band 4278 der Reihe *LNCS*, Seiten 1315–1324. Springer-Verlag, 2006.

[NNZ00] U. NICKEL, J. NIERE und A. ZÜNDORF: *The FUJABA Environment*. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, Seiten 742–745. ACM Press, 2000.

[Nor07] U. NORBISRATH: *Konfigurierung von eHome-Systemen*. Dissertation, RWTH Aachen, 2007.

[NSSK05] U. NORBISRATH, P. SALUMAA, E. SCHULTCHEN und B. KRAFT: *Fujaba-Based Tool Development for eHome Systems*. Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004), Electronic Notes in Theoretical Computer Science, 127(1):89–99, Elsevier Science, März 2005.

[Obj09a] OBJECT MANAGEMENT GROUP: *OMG Unified Modeling Language (OMG UML), Infrastructure – Version 2.2*. Document Number: formal/2009-02-04, Object Management Group (OMG), Februar 2009.

[Obj09b] OBJECT MANAGEMENT GROUP: *OMG Unified Modeling Language (OMG UML), Superstructure – Version 2.2*. Document Number: formal/2009-02-02, Object Management Group (OMG), Februar 2009.

[OR23] C. K. OGDEN und I. A. RICHARDS: *The Meaning of Meaning*. Kegan Paul, London, 1923.

[OSG04] OSGI ALLIANCE: *Listeners Considered Harmful: The „Whiteboard“ Pattern – Revision 2.0*. Technischer Bericht, August 2004.

[OSG07a] OSGI ALLIANCE: *About the OSGi Service Platform – Revision 4.1*. Juni 2007.

[OSG07b] OSGI ALLIANCE: *OSGi Service Platform – Core Specification – Release 4, Version 4.1*. Technischer Bericht, April 2007.

[OSG07c] OSGI ALLIANCE: *OSGi Service Platform – Service Compendium – Release 4, Version 4.1*. Technischer Bericht, April 2007.

[OSG09] OSGI ALLIANCE: *OSGi Alliance – Homepage*. http://www.osgi.org/Main/HomePage, 2009.

[Pap03] M. P. PAPAZOGLOU: *Service-Oriented Computing: Concepts, Characteristics and Directions*. In: *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE'03)*, Seiten 3–12. IEEE Computer Society, 2003.

[Phi03] PHILIPS RESEARCH: *365 days Ambient Intelligence research in HomeLab*. Bericht, Mai 2003.

[Pie09] M. PIENKOS: *Semantische Spezifikation und Adaption von eHome-Diensten*. Diplomarbeit, RWTH Aachen, Mai 2009.

[PKPS02] M. PAOLUCCI, T. KAWAMURA, T. PAYNE und K. SYCARA: *Semantic Matching of Web Services Capabilities*. In: *ISWC'02: Proceedings of the 1st International Semantic Web Conference on The Semantic Web*, Band 2342 der Reihe *LNCS*, Seiten 333–347. Springer-Verlag, 2002.

[PTDL07] M. P. PAPAZOGLOU, P. TRAVERSO, S. DUSTDAR und F. LEYMANN: *Service-Oriented Computing: State of the Art and Research Challenges*. IEEE Computer, 40(11):38–45, IEEE Computer Society, 2007.

[RAN09] D. RETKOWITZ, I. ARMAÇ und M. NAGL: *Towards Mobility Support in Smart Environments*. In: *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, Seiten 603–608. Knowledge Systems Institute Graduate School, 3420 Main Street, Skokie, Illinois 60076, USA, 2009.

[RBD+09] R. ROUVOY, P. BARONE, Y. DING, F. ELIASSEN, S. HALLSTEINSEN, J. LORENZO, A. MAMELLI und U. SCHOLZ: *MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments*. In: B. H. C. CHENG, R. DE LEMOS, H. GIESE, P. INVERARDI und J. MAGEE (Herausgeber): *Software Engineering for Self-Adaptive Systems*, Band 5525 der Reihe *LNCS*, Seiten 164–182. Springer-Verlag, 2009.

[Ree03] T. REENSKAUG: *The Model-View-Controller (MVC) – Its Past and Present*. Technischer Bericht, Universität Oslo, August 2003.

[Ret05] D. RETKOWITZ: *Regelbasierte Wissensdefinition und Analyse zur Unterstützung des konzeptuellen Entwurfs*. Diplomarbeit, RWTH Aachen, Oktober 2005.

[Ret07] D. RETKOWITZ: *Dynamic Composition of eHome Software*. In: *Dagstuhl „zehn plus eins“ – Zehn Informatik-Graduiertenkollegs und ein Informatik-Forschungskolleg stellen sich vor. 4.–6. Juni 2007*. Verlagshaus Mainz GmbH, Süsterfeldstr. 83, 52072 Aachen, 2007.

[RK09] D. RETKOWITZ und S. KULLE: *Dependency Management in Smart Homes*. In: T. SENIVONGSE und R. OLIVEIRA (Herausgeber): *Distributed Applications and Interoperable Systems, 9th IFIP WG 6.1 International Conference (DAIS 2009)*, Band 5523 der Reihe *LNCS*, Seiten 143–156. Springer-Verlag, 2009.

[Roz97] G. ROZENBERG (Herausgeber): *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*, Band 1. World Scientific, Singapur, 1997.

[RP08] D. RETKOWITZ und M. PIENKOS: *Ontology-based Configuration of Adaptive Smart Homes*. In: F. TAÏANI und R. CERQUEIRA (Herausgeber): *Proceedings of the 7th Workshop on Reflective and Adaptive Middleware (ARM'08) held at the 9th International Middleware Conference*, Seiten 11–16. ACM Press, 2008.

[RS08] D. RETKOWITZ und M. STEGELMANN: *Dynamic Adaptability for Smart Environments*. In: R. MEIER und S. TERZIS (Herausgeber): *Distributed Applications and Interoperable Systems, 8th IFIP WG 6.1*

International Conference (DAIS 2008), Band 5053 der Reihe *LNCS*, Seiten 154–167. Springer-Verlag, 2008.

[Rum04a] B. RUMPE: *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer-Verlag, August 2004.

[Rum04b] B. RUMPE: *Modellierung mit UML: Sprache, Konzepte und Methodik*. Springer-Verlag, Mai 2004.

[Sch91] A. SCHÜRR: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Dissertation, RWTH Aachen, 1991.

[Sch03] C. SCHNEIDER: *CASE Tool Unterstützung für die Delta-basierte Replikation und Versionierung komplexer Objektstrukturen*. Diplomarbeit, TU Braunschweig, April 2003.

[SGM02] C. SZYPERSKI, D. GRUNTZ und S. MURER: *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 2. Auflage, 2002.

[Sha03] N. SHADBOLT: *Ambient Intelligence*. IEEE Intelligent Systems, 18(4):2–3, IEEE Computer Society, 2003.

[SHNM04] M. SCARPINO, S. HOLDER, S. NG und L. MIHALKOVIC: *SWT/JFace in Action – GUI Design with Eclipse 3.0*. Manning Publications Co., 2004.

[Sma09a] SMART HOME SYSTEMS, INC.: *How X10 Works – X10 Theory*. `http://www.smarthomeusa.com/info/x10theory/`, 2009.

[Sma09b] SMARTHOME PADERBORN E. V.: *SmartHome Paderborn – Homepage*. `http://www.smarthomepaderborn.de/`, 2009.

[SN99] F. STEIMANN und W. NEJDL: *Modellierung und Ontologie*. Universität Hannover, Institut für Rechnergestützte Wissensverarbeitung, Lange Laube 3, 30159 Hannover, 1999.

[Spi92] J. M. SPIVEY: *The Z Notation: A Reference Manual*. Prentice Hall International, 2. Auflage, 1992.

[Ste08] M. STEGELMANN: *Spezifikation und Komposition von Diensten in dynamischen eHome-Systemen*. Diplomarbeit, RWTH Aachen, Januar 2008.

[SWZ99] A. SCHÜRR, A. WINTER und A. ZÜNDORF: *The PROGRES approach: Language and environment*, Seiten 487–550. Band 2 der Reihe H. EHRIG et al. [EEKR99], 1999.

[SZN04] C. SCHNEIDER, A. ZÜNDORF und J. NIERE: *CoObRA – a small step for development tools to collaborative environments*. In: J. GRUNDY, R. WELLAND und H. STOECKLE (Herausgeber): *Workshop on Directions in Software Engineering Environments – Workshop at ICSE 2004, 26th International Conference on Software Engineering*, Seiten 20–27, Mai 2004.

[Tek05] U. TEKER: *Realisierung und Evaluation eines Indoor-Lokalisierungssystems mittels WLAN*. Diplomarbeit, Universität Bremen, November 2005.

[Uni09] UNIVERSITY OF COLORADO AT BOULDER: *The Adaptive House – Homepage*. `http://www.cs.colorado.edu/~mozer/house/`, 2009.

[US08] A. UMBACH-DANIEL und T. SCHUMANN: *Pilotprojekt für das Heimautomations- und Meldesystem der Adhoco AG zur Unterstützung des selbstständigen Wohnens im Alter*. Zusammenfassung der sozialwissenschaftlichen Begleitstudie im Auftrag der Age Stiftung, November 2008.

[Völ04] M. VÖLTER: *Metamodellierung*. Technischer Bericht, Ingenieurbüro für Softwaretechnologie Markus Völter, Ziegelaecker 11, 89520 Heidenheim, 2004.

[VRV05] M. VALLÉE, F. RAMPARANY und L. VERCOUTER: *Flexible Composition of Smart Device Services*. In: L. T. YANG, J. MA, M. TAKIZAWA und T. K. SHIH (Herausgeber): *Proceedings of the 2005 International Conference on Pervasive Systems and Computing, PSC 2005, Las Vegas, Nevada, June 27-30, 2005*, Seiten 165–171. CSREA Press, Juni 2005.

[WB98] M. WEISER und J. S. BROWN: *The Coming Age of Calm Technology*. In: P. J. DENNING und R. M. METCALFE (Herausgeber): *Beyond Calculation: The Next Fifty Years of Computing*. Springer-Verlag, 1998.

[Wei91] M. WEISER: *The Computer for the 21st Century*. Scientific American, 265(3):94–104, September 1991.

[WGB99] M. WEISER, R. GOLD und J. S. BROWN: *The origins of ubiquitous computing research at PARC in the late 1980s*. IBM Systems Journal, 38(4):693–696, IBM Corporation, 1999.

[WHFG92] R. WANT, A. HOPPER, V. FALCÃO und J. GIBBONS: *The Active Badge Location System*. ACM Transactions on Information Systems, 10(1):91–102, ACM Press, Januar 1992.

[WHKL08] G. WÜTHERICH, N. HARTMANN, B. KOLB und M. LÜBKEN: *Die OSGi Service Platform: Eine Einführung mit Eclipse Equinox*. dpunkt.verlag, Heidelberg, 1. Auflage, 2008.

[WNR+06] H. H. WANG, N. NOY, A. RECTOR, M. MUSEN, T. REDMOND, D. RUBIN, S. TU, T. TUDORACHE, N. DRUMMOND, M. HORRIDGE und J. SEIDENBERG: *Frames and Ontology Side by Side*. 9th International Protégé Conference, Juli 2006.

[Wor04a] WORLD WIDE WEB CONSORTIUM (W3C): *OWL-S: Semantic Markup for Web Services*. http://www.w3.org/Submission/OWL-S/, November 2004.

[Wor04b] WORLD WIDE WEB CONSORTIUM (W3C): *Web Services Architecture*. http://www.w3.org/TR/ws-arch/, Februar 2004.

[Wor07] WORLD WIDE WEB CONSORTIUM (W3C): *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. http://www.w3.org/TR/wsdl20/, Juni 2007.

[Xue08] C. XUE: *Laufzeitkontrolle von Diensten in dynamischen eHome-Systemen*. Diplomarbeit, RWTH Aachen, November 2008.

[Zün96] A. ZÜNDORF: *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme*. Dissertation, RWTH Aachen, 1996.

[ZVE06] ZVEI – ZENTRALVERBAND ELEKTROTECHNIK- UND ELEKTRONIKINDUSTRIE E. V. (Herausgeber): *KNX – Handbuch Haus- und Gebäudesystemtechnik*. KNX Association, 5. Auflage, 2006.

Index

Lebenslauf

Daniel Retkowitz

Geburtsdatum	12. Mai 1979
Geburtsort	Münster, Westfalen
Staatsangehörigkeit	deutsch
seit 11/2005	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Informatik 3 (Softwaretechnik) der RWTH Aachen
10/2005	Studienabschluss als Diplom-Informatiker
09/2001 – 07/2002	Auslandsstudium an der Chalmers Tekniska Högskola in Göteborg, Schweden
10/1999 – 10/2005	Studium der Informatik mit Nebenfach Betriebswirtschaftslehre an der RWTH Aachen
07/1998 – 07/1999	Zivildienst im Malteser Krankenhaus St. Josef in Hamm
05/1998	Abitur
1989 – 1998	Immanuel-Kant-Gymnasium in Münster-Hiltrup
1985 – 1989	Kardinal-von-Galen Grundschule in Drensteinfurt

Aachener Informatik-Berichte

This is the list of all technical reports since 1987. To obtain copies of reports please consult http://aib.informatik.rwth-aachen.de/ or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: biblio@informatik.rwth-aachen.de

1987-01 * Fachgruppe Informatik: Jahresbericht 1986
1987-02 * David de Frutos Escrig, Klaus Indermark: Equivalence Relations of Non-Deterministic Ianov-Schemes
1987-03 * Manfred Nagl: A Software Development Environment based on Graph Technology
1987-04 * Claus Lewerentz, Manfred Nagl, Bernhard Westfechtel: On Integration Mechanisms within a Graph-Based Software Development Environment
1987-05 * Reinhard Rinn: Über Eingabeanomalien bei verschiedenen Inferenzmodellen
1987-06 * Werner Damm, Gert Döhmen: Specifying Distributed Computer Architectures in AADL*
1987-07 * Gregor Engels, Claus Lewerentz, Wilhelm Schäfer: Graph Grammar Engineering: A Software Specification Method
1987-08 * Manfred Nagl: Set Theoretic Approaches to Graph Grammars
1987-09 * Claus Lewerentz, Andreas Schürr: Experiences with a Database System for Software Documents
1987-10 * Herbert Klaeren, Klaus Indermark: A New Implementation Technique for Recursive Function Definitions
1987-11 * Rita Loogen: Design of a Parallel Programmable Graph Reduction Machine with Distributed Memory
1987-12 J. Börstler, U. Möncke, R. Wilhelm: Table compression for tree automata
1988-01 * Gabriele Esser, Johannes Rückert, Frank Wagner Gesellschaftliche Aspekte der Informatik
1988-02 * Peter Martini, Otto Spaniol: Token-Passing in High-Speed Backbone Networks for Campus-Wide Environments
1988-03 * Thomas Welzel: Simulation of a Multiple Token Ring Backbone
1988-04 * Peter Martini: Performance Comparison for HSLAN Media Access Protocols
1988-05 * Peter Martini: Performance Analysis of Multiple Token Rings
1988-06 * Andreas Mann, Johannes Rückert, Otto Spaniol: Datenfunknetze
1988-07 * Andreas Mann, Johannes Rückert: Packet Radio Networks for Data Exchange
1988-08 * Andreas Mann, Johannes Rückert: Concurrent Slot Assignment Protocol for Packet Radio Networks
1988-09 * W. Kremer, F. Reichert, J. Rückert, A. Mann: Entwurf einer Netzwerktopologie für ein Mobilfunknetz zur Unterstützung des öffentlichen Straßenverkehrs
1988-10 * Kai Jakobs: Towards User-Friendly Networking
1988-11 * Kai Jakobs: The Directory - Evolution of a Standard
1988-12 * Kai Jakobs: Directory Services in Distributed Systems - A Survey
1988-13 * Martine Schümmer: RS-511, a Protocol for the Plant Floor

1988-14 * U. Quernheim: Satellite Communication Protocols - A Performance Comparison Considering On-Board Processing

1988-15 * Peter Martini, Otto Spaniol, Thomas Welzel: File Transfer in High Speed Token Ring Networks: Performance Evaluation by Approximate Analysis and Simulation

1988-16 * Fachgruppe Informatik: Jahresbericht 1987

1988-17 * Wolfgang Thomas: Automata on Infinite Objects

1988-18 * Michael Sonnenschein: On Petri Nets and Data Flow Graphs

1988-19 * Heiko Vogler: Functional Distribution of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers

1988-20 * Thomas Welzel: Einsatz des Simulationswerkzeuges QNAP2 zur Leistungsbewertung von Kommunikationsprotokollen

1988-21 * Th. Janning, C. Lewerentz: Integrated Project Team Management in a Software Development Environment

1988-22 * Joost Engelfriet, Heiko Vogler: Modular Tree Transducers

1988-23 * Wolfgang Thomas: Automata and Quantifier Hierarchies

1988-24 * Uschi Heuter: Generalized Definite Tree Languages

1989-01 * Fachgruppe Informatik: Jahresbericht 1988

1989-02 * G. Esser, J. Rückert, F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik

1989-03 * Heiko Vogler: Bottom-Up Computation of Primitive Recursive Tree Functions

1989-04 * Andy Schürr: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language

1989-05 J. Börstler: Reuse and Software Development - Problems, Solutions, and Bibliography (in German)

1989-06 * Kai Jakobs: OSI - An Appropriate Basis for Group Communication?

1989-07 * Kai Jakobs: ISO's Directory Proposal - Evolution, Current Status and Future Problems

1989-08 * Bernhard Westfechtel: Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control

1989-09 * Peter Martini: High Speed Local Area Networks - A Tutorial

1989-10 * P. Davids, Th. Welzel: Performance Analysis of DQDB Based on Simulation

1989-11 * Manfred Nagl (Ed.): Abstracts of Talks presented at the WG '89 15th International Workshop on Graphtheoretic Concepts in Computer Science

1989-12 * Peter Martini: The DQDB Protocol - Is it Playing the Game?

1989-13 * Martine Schümmer: CNC/DNC Communication with MAP

1989-14 * Martine Schümmer: Local Area Networks for Manufactoring Environments with hard Real-Time Requirements

1989-15 * M. Schümmer, Th. Welzel, P. Martini: Integration of Field Bus and MAP Networks - Hierarchical Communication Systems in Production Environments

1989-16 * G. Vossen, K.-U. Witt: SUXESS: Towards a Sound Unification of Extensions of the Relational Data Model

1989-17 * J. Derissen, P. Hruschka, M.v.d. Beeck, Th. Janning, M. Nagl: Integrating Structured Analysis and Information Modelling
1989-18 A. Maassen: Programming with Higher Order Functions
1989-19 * Mario Rodriguez-Artalejo, Heiko Vogler: A Narrowing Machine for Syntax Directed BABEL
1989-20 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Graph-based Implementation of a Functional Logic Language
1990-01 * Fachgruppe Informatik: Jahresbericht 1989
1990-02 * Vera Jansen, Andreas Potthoff, Wolfgang Thomas, Udo Wermuth: A Short Guide to the AMORE System (Computing Automata, MOnoids and Regular Expressions)
1990-03 * Jerzy Skurczynski: On Three Hierarchies of Weak SkS Formulas
1990-04 R. Loogen: Stack-based Implementation of Narrowing
1990-05 H. Kuchen, A. Wagener: Comparison of Dynamic Load Balancing Strategies
1990-06 * Kai Jakobs, Frank Reichert: Directory Services for Mobile Communication
1990-07 * Kai Jakobs: What's Beyond the Interface - OSI Networks to Support Cooperative Work
1990-08 * Kai Jakobs: Directory Names and Schema - An Evaluation
1990-09 * Ulrich Quernheim, Dieter Kreuer: Das CCITT - Signalisierungssystem Nr. 7 auf Satellitenstrecken; Simulation der Zeichengabestrecke
1990-11 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Lazy Narrowing in a Graph Machine
1990-12 * Kai Jakobs, Josef Kaltwasser, Frank Reichert, Otto Spaniol: Der Computer fährt mit
1990-13 * Rudolf Mathar, Andreas Mann: Analyzing a Distributed Slot Assignment Protocol by Markov Chains
1990-14 A. Maassen: Compilerentwicklung in Miranda - ein Praktikum in funktionaler Programmierung (written in german)
1990-15 * Manfred Nagl, Andreas Schürr: A Specification Environment for Graph Grammars
1990-16 A. Schürr: PROGRESS: A VHL-Language Based on Graph Grammars
1990-17 * Marita Möller: Ein Ebenenmodell wissensbasierter Konsultationen - Unterstützung für Wissensakquisition und Erklärungsfähigkeit
1990-18 * Eric Kowalewski: Entwurf und Interpretation einer Sprache zur Beschreibung von Konsultationsphasen in Expertensystemen
1990-20 Y. Ortega Mallen, D. de Frutos Escrig: A Complete Proof System for Timed Observations
1990-21 * Manfred Nagl: Modelling of Software Architectures: Importance, Notions, Experiences
1990-22 H. Fassbender, H. Vogler: A Call-by-need Implementation of Syntax Directed Functional Programming
1991-01 Guenther Geiler (ed.), Fachgruppe Informatik: Jahresbericht 1990
1991-03 B. Steffen, A. Ingolfsdottir: Characteristic Formulae for Processes with Divergence
1991-04 M. Portz: A new class of cryptosystems based on interconnection networks

1991-05 H. Kuchen, G. Geiler: Distributed Applicative Arrays
1991-06 * Ludwig Staiger: Kolmogorov Complexity and Hausdorff Dimension
1991-07 * Ludwig Staiger: Syntactic Congruences for w-languages
1991-09 * Eila Kuikka: A Proposal for a Syntax-Directed Text Processing System
1991-10 K. Gladitz, H. Fassbender, H. Vogler: Compiler-based Implementation of Syntax-Directed Functional Programming
1991-11 R. Loogen, St. Winkler: Dynamic Detection of Determinism in Functional Logic Languages
1991-12 * K. Indermark, M. Rodriguez Artalejo (Eds.): Granada Workshop on the Integration of Functional and Logic Programming
1991-13 * Rolf Hager, Wolfgang Kremer: The Adaptive Priority Scheduler: A More Fair Priority Service Discipline
1991-14 * Andreas Fasbender, Wolfgang Kremer: A New Approximation Algorithm for Tandem Networks with Priority Nodes
1991-15 J. Börstler, A. Zündorf: Revisiting extensions to Modula-2 to support reusability
1991-16 J. Börstler, Th. Janning: Bridging the gap between Requirements Analysis and Design
1991-17 A. Zündorf, A. Schürr: Nondeterministic Control Structures for Graph Rewriting Systems
1991-18 * Matthias Jarke, John Mylopoulos, Joachim W. Schmidt, Yannis Vassiliou: DAIDA: An Environment for Evolving Information Systems
1991-19 M. Jeusfeld, M. Jarke: From Relational to Object-Oriented Integrity Simplification
1991-20 G. Hogen, A. Kindler, R. Loogen: Automatic Parallelization of Lazy Functional Programs
1991-21 * Prof. Dr. rer. nat. Otto Spaniol: ODP (Open Distributed Processing): Yet another Viewpoint
1991-22 H. Kuchen, F. Lücking, H. Stoltze: The Topology Description Language TDL
1991-23 S. Graf, B. Steffen: Compositional Minimization of Finite State Systems
1991-24 R. Cleaveland, J. Parrow, B. Steffen: The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems
1991-25 * Rudolf Mathar, Jürgen Mattfeldt: Optimal Transmission Ranges for Mobile Communication in Linear Multihop Packet Radio Networks
1991-26 M. Jeusfeld, M. Staudt: Query Optimization in Deductive Object Bases
1991-27 J. Knoop, B. Steffen: The Interprocedural Coincidence Theorem
1991-28 J. Knoop, B. Steffen: Unifying Strength Reduction and Semantic Code Motion
1991-30 T. Margaria: First-Order theories for the verification of complex FSMs
1991-31 B. Steffen: Generating Data Flow Analysis Algorithms from Modal Specifications
1992-01 Stefan Eherer (ed.), Fachgruppe Informatik: Jahresbericht 1991
1992-02 * Bernhard Westfechtel: Basismechanismen zur Datenverwaltung in strukturbezogenen Hypertextsystemen
1992-04 S. A. Smolka, B. Steffen: Priority as Extremal Probability
1992-05 * Matthias Jarke, Carlos Maltzahn, Thomas Rose: Sharing Processes: Team Coordination in Design Repositories

1992-06 O. Burkart, B. Steffen: Model Checking for Context-Free Processes
1992-07 * Matthias Jarke, Klaus Pohl: Information Systems Quality and Quality Information Systems
1992-08 * Rudolf Mathar, Jürgen Mattfeldt: Analyzing Routing Strategy NFP in Multihop Packet Radio Networks on a Line
1992-09 * Alfons Kemper, Guido Moerkotte: Grundlagen objektorientierter Datenbanksysteme
1992-10 Matthias Jarke, Manfred Jeusfeld, Andreas Miethsam, Michael Gocek: Towards a logic-based reconstruction of software configuration management
1992-11 Werner Hans: A Complete Indexing Scheme for WAM-based Abstract Machines
1992-12 W. Hans, R. Loogen, St. Winkler: On the Interaction of Lazy Evaluation and Backtracking
1992-13 * Matthias Jarke, Thomas Rose: Specification Management with CAD
1992-14 Th. Noll, H. Vogler: Top-down Parsing with Simultaneous Evaluation on Noncircular Attribute Grammars
1992-15 A. Schuerr, B. Westfechtel: Graphgrammatiken und Graphersetzungssysteme(written in german)
1992-16 * Graduiertenkolleg Informatik und Technik (Hrsg.): Forschungsprojekte des Graduiertenkollegs Informatik und Technik
1992-17 M. Jarke (ed.): ConceptBase V3.1 User Manual
1992-18 * Clarence A. Ellis, Matthias Jarke (Eds.): Distributed Cooperation in Integrated Information Systems - Proceedings of the Third International Workshop on Intelligent and Cooperative Information Systems
1992-19-00 H. Kuchen, R. Loogen (eds.): Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages
1992-19-01 G. Hogen, R. Loogen: PASTEL - A Parallel Stack-Based Implementation of Eager Functional Programs with Lazy Data Structures (Extended Abstract)
1992-19-02 H. Kuchen, K. Gladitz: Implementing Bags on a Shared Memory MIMD-Machine
1992-19-03 C. Rathsack, S.B. Scholz: LISA - A Lazy Interpreter for a Full-Fledged Lambda-Calculus
1992-19-04 T.A. Bratvold: Determining Useful Parallelism in Higher Order Functions
1992-19-05 S. Kahrs: Polymorphic Type Checking by Interpretation of Code
1992-19-06 M. Chakravarty, M. Köhler: Equational Constraints, Residuation, and the Parallel JUMP-Machine
1992-19-07 J. Seward: Polymorphic Strictness Analysis using Frontiers (Draft Version)
1992-19-08 D. Gärtner, A. Kimms, W. Kluge: pi-Redˆ+ - A Compiling Graph-Reduction System for a Full Fledged Lambda-Calculus
1992-19-09 D. Howe, G. Burn: Experiments with strict STG code
1992-19-10 J. Glauert: Parallel Implementation of Functional Languages Using Small Processes
1992-19-11 M. Joy, T. Axford: A Parallel Graph Reduction Machine
1992-19-12 A. Bennett, P. Kelly: Simulation of Multicache Parallel Reduction

1992-19-13 K. Langendoen, D.J. Agterkamp: Cache Behaviour of Lazy Functional Programs (Working Paper)

1992-19-14 K. Hammond, S. Peyton Jones: Profiling scheduling strategies on the GRIP parallel reducer

1992-19-15 S. Mintchev: Using Strictness Information in the STG-machine

1992-19-16 D. Rushall: An Attribute Grammar Evaluator in Haskell

1992-19-17 J. Wild, H. Glaser, P. Hartel: Statistics on storage management in a lazy functional language implementation

1992-19-18 W.S. Martins: Parallel Implementations of Functional Languages

1992-19-19 D. Lester: Distributed Garbage Collection of Cyclic Structures (Draft version)

1992-19-20 J.C. Glas, R.F.H. Hofman, W.G. Vree: Parallelization of Branch-and-Bound Algorithms in a Functional Programming Environment

1992-19-21 S. Hwang, D. Rushall: The nu-STG machine: a parallelized Spineless Tagless Graph Reduction Machine in a distributed memory architecture (Draft version)

1992-19-22 G. Burn, D. Le Metayer: Cps-Translation and the Correctness of Optimising Compilers

1992-19-23 S.L. Peyton Jones, P. Wadler: Imperative functional programming (Brief summary)

1992-19-24 W. Damm, F. Liu, Th. Peikenkamp: Evaluation and Parallelization of Functions in Functional + Logic Languages (abstract)

1992-19-25 M. Kesseler: Communication Issues Regarding Parallel Functional Graph Rewriting

1992-19-26 Th. Peikenkamp: Charakterizing and representing neededness in functional loginc languages (abstract)

1992-19-27 H. Doerr: Monitoring with Graph-Grammars as formal operational Models

1992-19-28 J. van Groningen: Some implementation aspects of Concurrent Clean on distributed memory architectures

1992-19-29 G. Ostheimer: Load Bounding for Implicit Parallelism (abstract)

1992-20 H. Kuchen, F.J. Lopez Fraguas, J.J. Moreno Navarro, M. Rodriguez Artalejo: Implementing Disequality in a Lazy Functional Logic Language

1992-21 H. Kuchen, F.J. Lopez Fraguas: Result Directed Computing in a Functional Logic Language

1992-22 H. Kuchen, J.J. Moreno Navarro, M.V. Hermenegildo: Independent AND-Parallel Narrowing

1992-23 T. Margaria, B. Steffen: Distinguishing Formulas for Free

1992-24 K. Pohl: The Three Dimensions of Requirements Engineering

1992-25 * R. Stainov: A Dynamic Configuration Facility for Multimedia Communications

1992-26 * Michael von der Beeck: Integration of Structured Analysis and Timed Statecharts for Real-Time and Concurrency Specification

1992-27 W. Hans, St. Winkler: Aliasing and Groundness Analysis of Logic Programs through Abstract Interpretation and its Safety

1992-28 * Gerhard Steinke, Matthias Jarke: Support for Security Modeling in Information Systems Design

1992-29 B. Schinzel: Warum Frauenforschung in Naturwissenschaft und Technik

1992-30 A. Kemper, G. Moerkotte, K. Peithner: Object-Orientation Axiomatised by Dynamic Logic

1992-32 * Bernd Heinrichs, Kai Jakobs: Timer Handling in High-Performance Transport Systems

1992-33 * B. Heinrichs, K. Jakobs, K. Lenßen, W. Reinhardt, A. Spinner: Euro-Bridge: Communication Services for Multimedia Applications

1992-34 C. Gerlhof, A. Kemper, Ch. Kilger, G. Moerkotte: Partition-Based Clustering in Object Bases: From Theory to Practice

1992-35 J. Börstler: Feature-Oriented Classification and Reuse in IPSEN

1992-36 M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, Y. Vassiliou: Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis

1992-37 * K. Pohl, M. Jarke: Quality Information Systems: Repository Support for Evolving Process Models

1992-38 A. Zuendorf: Implementation of the imperative / rule based language PROGRES

1992-39 P. Koch: Intelligentes Backtracking bei der Auswertung funktional-logischer Programme

1992-40 * Rudolf Mathar, Jürgen Mattfeldt: Channel Assignment in Cellular Radio Networks

1992-41 * Gerhard Friedrich, Wolfgang Neidl: Constructive Utility in Model-Based Diagnosis Repair Systems

1992-42 * P. S. Chen, R. Hennicker, M. Jarke: On the Retrieval of Reusable Software Components

1992-43 W. Hans, St.Winkler: Abstract Interpretation of Functional Logic Languages

1992-44 N. Kiesel, A. Schuerr, B. Westfechtel: Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications

1993-01 * Fachgruppe Informatik: Jahresbericht 1992

1993-02 * Patrick Shicheng Chen: On Inference Rules of Logic-Based Information Retrieval Systems

1993-03 G. Hogen, R. Loogen: A New Stack Technique for the Management of Runtime Structures in Distributed Environments

1993-05 A. Zündorf: A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES

1993-06 A. Kemper, D. Kossmann: Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis

1993-07 * Graduiertenkolleg Informatik und Technik (Hrsg.): Graduiertenkolleg Informatik und Technik

1993-08 * Matthias Berger: k-Coloring Vertices using a Neural Network with Convergence to Valid Solutions

1993-09 M. Buchheit, M. Jeusfeld, W. Nutt, M. Staudt: Subsumption between Queries to Object-Oriented Databases

1993-10 O. Burkart, B. Steffen: Pushdown Processes: Parallel Composition and Model Checking

1993-11 * R. Große-Wienker, O. Hermanns, D. Menzenbach, A. Pollacks, S. Repetzki, J. Schwartz, K. Sonnenschein, B. Westfechtel: Das SUKITS-Projekt: A-posteriori-Integration heterogener CIM-Anwendungssysteme

1993-12 * Rudolf Mathar, Jürgen Mattfeldt: On the Distribution of Cumulated Interference Power in Rayleigh Fading Channels
1993-13 O. Maler, L. Staiger: On Syntactic Congruences for omega-languages
1993-14 M. Jarke, St. Eherer, R. Gallersdoerfer, M. Jeusfeld, M. Staudt: ConceptBase - A Deductive Object Base Manager
1993-15 M. Staudt, H.W. Nissen, M.A. Jeusfeld: Query by Class, Rule and Concept
1993-16 * M. Jarke, K. Pohl, St. Jacobs et al.: Requirements Engineering: An Integrated View of Representation Process and Domain
1993-17 * M. Jarke, K. Pohl: Establishing Vision in Context: Towards a Model of Requirements Processes
1993-18 W. Hans, H. Kuchen, St. Winkler: Full Indexing for Lazy Narrowing
1993-19 W. Hans, J.J. Ruz, F. Saenz, St. Winkler: A VHDL Specification of a Shared Memory Parallel Machine for Babel
1993-20 * K. Finke, M. Jarke, P. Szczurko, R. Soltysiak: Quality Management for Expert Systems in Process Control
1993-21 M. Jarke, M.A. Jeusfeld, P. Szczurko: Three Aspects of Intelligent Cooperation in the Quality Cycle
1994-01 Margit Generet, Sven Martin (eds.), Fachgruppe Informatik: Jahresbericht 1993
1994-02 M. Lefering: Development of Incremental Integration Tools Using Formal Specifications
1994-03 * P. Constantopoulos, M. Jarke, J. Mylopoulos, Y. Vassiliou: The Software Information Base: A Server for Reuse
1994-04 * Rolf Hager, Rudolf Mathar, Jürgen Mattfeldt: Intelligent Cruise Control and Reliable Communication of Mobile Stations
1994-05 * Rolf Hager, Peter Hermesmann, Michael Portz: Feasibility of Authentication Procedures within Advanced Transport Telematics
1994-06 * Claudia Popien, Bernd Meyer, Axel Kuepper: A Formal Approach to Service Import in ODP Trader Federations
1994-07 P. Peters, P. Szczurko: Integrating Models of Quality Management Methods by an Object-Oriented Repository
1994-08 * Manfred Nagl, Bernhard Westfechtel: A Universal Component for the Administration in Distributed and Integrated Development Environments
1994-09 * Patrick Horster, Holger Petersen: Signatur- und Authentifikationsverfahren auf der Basis des diskreten Logarithmusproblems
1994-11 A. Schürr: PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems
1994-12 A. Schürr: Specification of Graph Translators with Triple Graph Grammars
1994-13 A. Schürr: Logic Based Programmed Structure Rewriting Systems
1994-14 L. Staiger: Codes, Simplifying Words, and Open Set Condition
1994-15 * Bernhard Westfechtel: A Graph-Based System for Managing Configurations of Engineering Design Documents
1994-16 P. Klein: Designing Software with Modula-3
1994-17 I. Litovsky, L. Staiger: Finite acceptance of infinite words

1994-18 G. Hogen, R. Loogen: Parallel Functional Implementations: Graphbased vs. Stackbased Reduction

1994-19 M. Jeusfeld, U. Johnen: An Executable Meta Model for Re-Engineering of Database Schemas

1994-20 * R. Gallersdörfer, M. Jarke, K. Klabunde: Intelligent Networks as a Data Intensive Application (INDIA)

1994-21 M. Mohnen: Proving the Correctness of the Static Link Technique Using Evolving Algebras

1994-22 H. Fernau, L. Staiger: Valuations and Unambiguity of Languages, with Applications to Fractal Geometry

1994-24 * M. Jarke, K. Pohl, R. Dömges, St. Jacobs, H. W. Nissen: Requirements Information Management: The NATURE Approach

1994-25 * M. Jarke, K. Pohl, C. Rolland, J.-R. Schmitt: Experience-Based Method Evaluation and Improvement: A Process Modeling Approach

1994-26 * St. Jacobs, St. Kethers: Improving Communication and Decision Making within Quality Function Deployment

1994-27 * M. Jarke, H. W. Nissen, K. Pohl: Tool Integration in Evolving Information Systems Environments

1994-28 O. Burkart, D. Caucal, B. Steffen: An Elementary Bisimulation Decision Procedure for Arbitrary Context-Free Processes

1995-01 * Fachgruppe Informatik: Jahresbericht 1994

1995-02 Andy Schürr, Andreas J. Winter, Albert Zündorf: Graph Grammar Engineering with PROGRES

1995-03 Ludwig Staiger: A Tight Upper Bound on Kolmogorov Complexity by Hausdorff Dimension and Uniformly Optimal Prediction

1995-04 Birgitta König-Ries, Sven Helmer, Guido Moerkotte: An experimental study on the complexity of left-deep join ordering problems for cyclic queries

1995-05 Sophie Cluet, Guido Moerkotte: Efficient Evaluation of Aggregates on Bulk Types

1995-06 Sophie Cluet, Guido Moerkotte: Nested Queries in Object Bases

1995-07 Sophie Cluet, Guido Moerkotte: Query Optimization Techniques Exploiting Class Hierarchies

1995-08 Markus Mohnen: Efficient Compile-Time Garbage Collection for Arbitrary Data Structures

1995-09 Markus Mohnen: Functional Specification of Imperative Programs: An Alternative Point of View of Functional Languages

1995-10 Rainer Gallersdörfer, Matthias Nicola: Improving Performance in Replicated Databases through Relaxed Coherency

1995-11 * M.Staudt, K.von Thadden: Subsumption Checking in Knowledge Bases

1995-12 * G.V.Zemanek, H.W.Nissen, H.Hubert, M.Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology

1995-13 * M.Staudt, M.Jarke: Incremental Maintenance of Externally Materialized Views

1995-14 * P.Peters, P.Szczurko, M.Jeusfeld: Oriented Information Management: Conceptual Models at Work

1995-15 * Matthias Jarke, Sudha Ram (Hrsg.): WITS 95 Proceedings of the 5th Annual Workshop on Information Technologies and Systems

1995-16 * W.Hans, St.Winkler, F.Saenz: Distributed Execution in Functional Logic Programming

1996-01 * Jahresbericht 1995

1996-02 Michael Hanus, Christian Prehofer: Higher-Order Narrowing with Definitional Trees

1996-03 * W.Scheufele, G.Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates

1996-04 Klaus Pohl: PRO-ART: Enabling Requirements Pre-Traceability

1996-05 Klaus Pohl: Requirements Engineering: An Overview

1996-06 * M.Jarke, W.Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools

1996-07 Olaf Chitil: The Sigma-Semantics: A Comprehensive Semantics for Functional Programs

1996-08 * S.Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics

1996-09 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP96 - Fifth International Conference on Algebraic and Logic Programming

1996-09-0 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP 96 - Fifth International Conference on Algebraic and Logic Programming: Introduction and table of contents

1996-09-1 Ilies Alouini: An Implementation of Conditional Concurrent Rewriting on Distributed Memory Machines

1996-09-2 Olivier Danvy, Karoline Malmkjær: On the Idempotence of the CPS Transformation

1996-09-3 Víctor M. Gulías, José L. Freire: Concurrent Programming in Haskell

1996-09-4 Sébastien Limet, Pierre Réty: On Decidability of Unifiability Modulo Rewrite Systems

1996-09-5 Alexandre Tessier: Declarative Debugging in Constraint Logic Programming

1996-10 Reidar Conradi, Bernhard Westfechtel: Version Models for Software Configuration Management

1996-11 * C.Weise, D.Lenzkes: A Fast Decision Algorithm for Timed Refinement

1996-12 * R.Dömges, K.Pohl, M.Jarke, B.Lohmann, W.Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models

1996-13 * K.Pohl, R.Klamma, K.Weidenhaupt, R.Dömges, P.Haumer, M.Jarke: A Framework for Process-Integrated Tools

1996-14 * R.Gallersdörfer, K.Klabunde, A.Stolz, M.Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996

1996-15 * H.Schimpe, M.Staudt: VAREX: An Environment for Validating and Refining Rule Bases

1996-16 * M.Jarke, M.Gebhardt, S.Jacobs, H.Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization

1996-17 Manfred A. Jeusfeld, Tung X. Bui: Decision Support Components on the Internet

1996-18 Manfred A. Jeusfeld, Mike Papazoglou: Information Brokering: Design, Search and Transformation

1996-19 * P.Peters, M.Jarke: Simulating the impact of information flows in networked organizations

1996-20 Matthias Jarke, Peter Peters, Manfred A. Jeusfeld: Model-driven planning and design of cooperative information systems

1996-21 * G.de Michelis, E.Dubois, M.Jarke, F.Matthes, J.Mylopoulos, K.Pohl, J.Schmidt, C.Woo, E.Yu: Cooperative information systems: a manifesto

1996-22 * S.Jacobs, M.Gebhardt, S.Kethers, W.Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

1996-23 * M.Gebhardt, S.Jacobs: Conflict Management in Design

1997-01 Michael Hanus, Frank Zartmann (eds.): Jahresbericht 1996

1997-02 Johannes Faassen: Using full parallel Boltzmann Machines for Optimization

1997-03 Andreas Winter, Andy Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

1997-04 Markus Mohnen, Stefan Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

1997-05 * S.Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

1997-06 Matthias Nicola, Matthias Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

1997-07 Petra Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

1997-08 Dorothea Blostein, Andy Schürr: Computing with Graphs and Graph Rewriting

1997-09 Carl-Arndt Krapp, Bernhard Westfechtel: Feedback Handling in Dynamic Task Nets

1997-10 Matthias Nicola, Matthias Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

1997-11 * R. Klamma, P. Peters, M. Jarke: Workflow Support for Failure Management in Federated Organizations

1997-13 Markus Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

1997-14 Roland Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

1997-15 George Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

1998-01 * Fachgruppe Informatik: Jahresbericht 1997

1998-02 Stefan Gruner, Manfred Nagel, Andy Schürr: Fine-grained and Structure-Oriented Document Integration Tools are Needed for Development Processes

1998-03 Stefan Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

1998-04 * O. Kubitz: Mobile Robots in Dynamic Environments

1998-05 Martin Leucker, Stephan Tobies: Truth - A Verification Platform for Distributed Systems

1998-06 * Matthias Oliver Berger: DECT in the Factory of the Future

1998-07 M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects

1998-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien

1998-10 * M. Nicola, M. Jarke: Performance Modeling of Distributed and Replicated Databases

1998-11 * Ansgar Schleicher, Bernhard Westfechtel, Dirk Jäger: Modeling Dynamic Software Processes in UML

1998-12 * W. Appelt, M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web

1998-13 Klaus Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation

1999-01 * Jahresbericht 1998

1999-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version

1999-03 * R. Gallersdörfer, M. Jarke, M. Nicola: The ADR Replication Manager

1999-04 María Alpuente, Michael Hanus, Salvador Lucas, Germán Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing

1999-05 * W. Thomas (Ed.): DLT 99 - Developments in Language Theory Fourth International Conference

1999-06 * Kai Jakobs, Klaus-Dieter Kleefeld: Informationssysteme für die angewandte historische Geographie

1999-07 Thomas Wilke: CTL+ is exponentially more succinct than CTL

1999-08 Oliver Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures

2000-01 * Jahresbericht 1999

2000-02 Jens Vöge, Marcin Jurdzinski A Discrete Strategy Improvement Algorithm for Solving Parity Games

2000-03 D. Jäger, A. Schleicher, B. Westfechtel: UPGRADE: A Framework for Building Graph-Based Software Engineering Tools

2000-04 Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach

2000-05 Mareike Schoop: Cooperative Document Management

2000-06 Mareike Schoop, Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling

2000-07 * Markus Mohnen, Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages

2000-08 Thomas Arts, Thomas Noll: Verifying Generic Erlang Client-Server Implementations

2001-01 * Jahresbericht 2000

2001-02 Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces

2001-03 Thierry Cachat: The power of one-letter rational languages

2001-04 Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus
2001-05 Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages
2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic
2001-07 Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem
2001-08 Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling
2001-09 Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10 Achim Blumensath: Axiomatising Tree-interpretable Structures
2001-11 Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung
2002-01 * Jahresbericht 2001
2002-02 Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems
2002-03 Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages
2002-04 Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting
2002-05 Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines
2002-06 Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata
2002-07 Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities
2002-08 Markus Mohnen: An Open Framework for Data-Flow Analysis in Java
2002-09 Markus Mohnen: Interfaces with Default Implementations in Java
2002-10 Martin Leucker: Logics for Mazurkiewicz traces
2002-11 Jürgen Giesl, Hans Zantema: Liveness in Rewriting
2003-01 * Jahresbericht 2002
2003-02 Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03 Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05 Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06 Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07 Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 * Fachgruppe Informatik: Jahresbericht 2003

2004-02 Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic

2004-03 Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting

2004-04 Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming

2004-05 Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming

2004-06 Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming

2004-07 Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination

2004-08 Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information

2004-09 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity

2004-10 Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules

2005-01 * Fachgruppe Informatik: Jahresbericht 2004

2005-02 Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"

2005-03 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions

2005-04 Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem

2005-05 Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06 Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information

2005-07 Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks

2005-08 Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut

2005-09 Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures

2005-10 Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11 Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-12 Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems

2005-13 Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14 Felix C. Freiling, Sukumar Ghosh: Code Stabilization

2005-15 Uwe Naumann: The Complexity of Derivative Computation

2005-16 Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

2005-17 Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18 Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"

2005-19 Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers

2005-20 Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.

2005-21 Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited

2005-22 Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins

2005-23 Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves

2005-24 Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks

2006-01 * Fachgruppe Informatik: Jahresbericht 2005

2006-02 Michael Weber: Parallel Algorithms for Verification of Large Systems

2006-03 Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

2006-04 Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation

2006-05 Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F

2006-06 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color

2006-07 Thomas Colcombet, Christof Löding: Transforming structures by set interpretations

2006-08 Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs

2006-09 Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking

2006-10 Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed

2006-11 Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers

2006-12 Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning

2006-13	Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities
2006-14	Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"
2006-15	Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices
2006-16	Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness
2006-17	Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines
2007-01 *	Fachgruppe Informatik: Jahresbericht 2006
2007-02	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations
2007-03	Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase
2007-04	Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation
2007-05	Uwe Naumann: On Optimal DAG Reversal
2007-06	Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking
2007-07	Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications
2007-08	Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches
2007-09	Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption
2007-10	Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes
2007-11	Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke
2007-12	Uwe Naumann: An L-Attributed Grammar for Adjoint Code
2007-13	Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs
2007-14	Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes
2007-15	Volker Stolz: Temporal assertions for sequential and concurrent programs
2007-16	Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks
2007-17	René Thiemann: The DP Framework for Proving Termination of Term Rewriting
2007-18	Uwe Naumann: Call Tree Reversal is NP-Complete

2007-19 Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control

2007-20 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems

2007-21 Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains

2007-22 Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets

2008-01 * Fachgruppe Informatik: Jahresbericht 2007

2008-02 Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing

2008-03 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, Harald Zankl: Maximal Termination

2008-04 Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler

2008-05 Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations

2008-06 Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs

2008-07 Alexander Nyßen, Horst Lichter: The MeDUSA Reference Manual, Second Edition

2008-08 George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs

2008-09 George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs

2008-10 George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time

2008-11 George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows

2008-12 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages

2008-13 Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs

2008-14 Bastian Schlich: Model Checking of Software for Microcontrollers

2008-15 Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves

2008-16 Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study

2008-17 Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving

2008-18 Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems

2008-19 Dirk Wilking: Empirical Studies for the Application of Agile Methods to Embedded Systems

2009-02	Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications
2009-03	Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices
2009-04	Daniel Klünder: Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation
2009-05	George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs
2009-06	George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete
2009-07	Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I
2009-08	Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs
2009-09	Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem
2009-10	Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm
2009-11	Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs
2009-12	Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes
2009-13	Martin Zimmermann: Time-optimal Winning Strategies for Poset Games
2009-14	Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)
2009-15	Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems
2009-16	George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs
2009-17	Carsten Kern: Learning Communicating and Nondeterministic Automata
2009-18	Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies
2010-02	Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03	Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering
2010-04	René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme
2010-07	George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms
2010-08	Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting
2010-09	George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs

2010-10 Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut

2010-11 Martin Zimmermann: Parametric LTL Games

2010-12 Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.