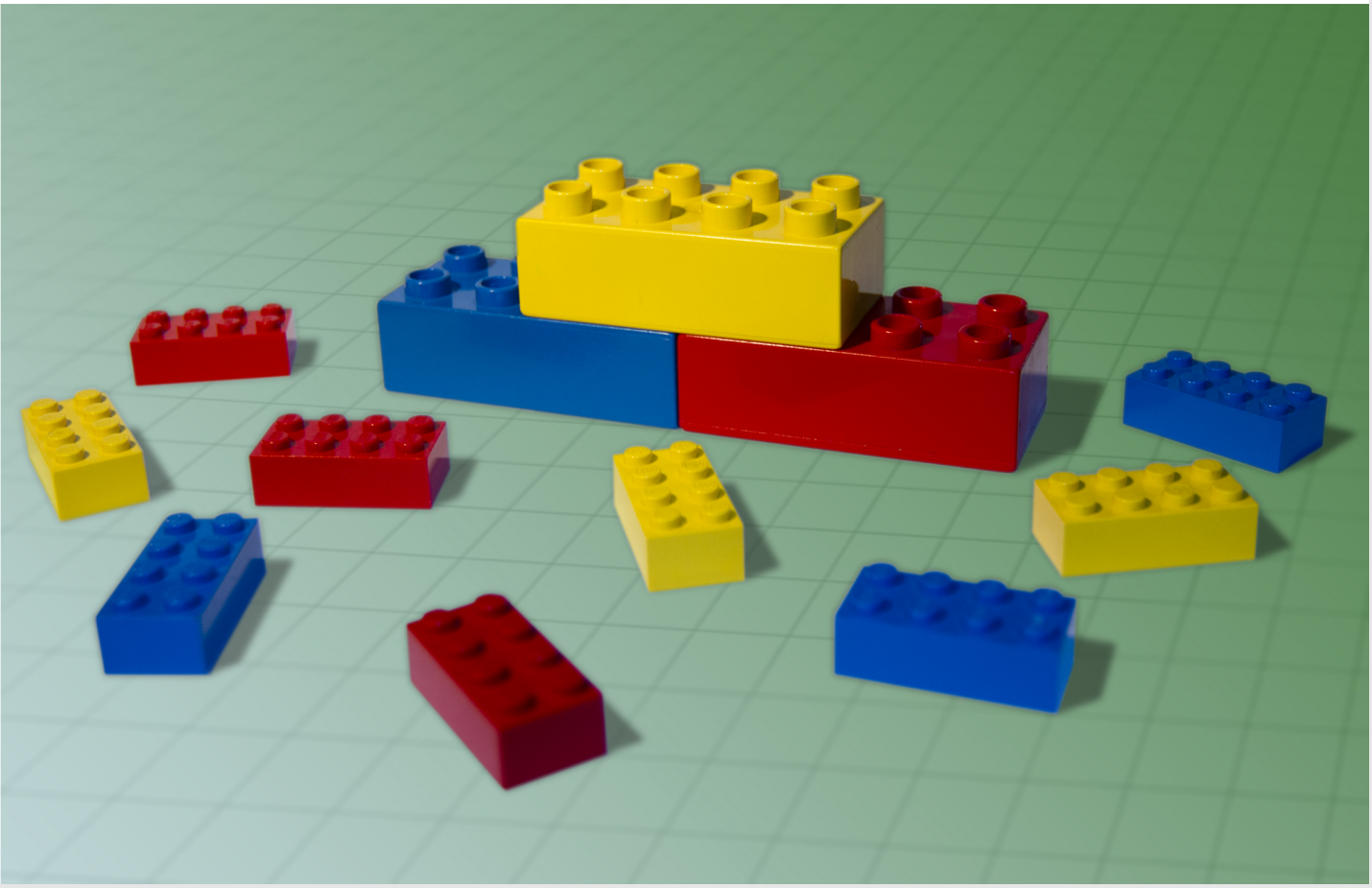
TU Clausthal



Constanze Deiters

Beschreibung und konsistente Komposition von Bausteinen fr den Architekturentwurf von Softwaresystemen

SSE-Dissertation 11

Software Systems Engineering

Department of Informatics
Chair of Prof. Dr. Andreas Rausch

Beschreibung und konsistente Komposition von Bausteinen fr den Architekturentwurf von Softwaresystemen

D o c t o r a l T h e s i s
(D i s s e r t a t i o n)

to be awarded the degree of
Doctor rerum naturalium
(Dr. rer. nat.)

submitted by
Constanze Deiters
from

approved by the Department of Informatics,
Clausthal University of Technology

2015

Dissertation Clausthal, SSE-Dissertation 11, 2015

Chairperson of the Board of Examiners
Prof. Dr. Michael Kolonko

Chief Reviewer
Prof. Dr. Andreas Rausch

2. Reviewer
Prof. Dr. Stefan Biffl

Date of oral examination:

Thesis text comes here. . .

Beschreibung und konsistente Komposition von Bausteinen für den Architekturentwurf von Softwaresystemen

---

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat)

vorgelegt von
Constanze Deiters
aus Hildesheim

genehmigt von der
Fakultät für Mathematik/Informatik und Maschinenbau
der Technischen Universität Clausthal

2015

Dissertation Clausthal, SSE-Dissertation 11, 2015

Vorsitzender der Promotionskommission
Prof. Dr. Sven Hartmann

Hauptberichterstatter
Prof. Dr. Andreas Rausch

Mitberichterstatter
Prof. Dr. Jörg P. Müller

Tag der mündlichen Prüfung: 13. April 2015

Kurzfassung

Die Architektur eines Softwaresystems legt die grundlegenden Eigenschaften, Konzepte und Strukturen des Systems fest. Beim Entwurf der Softwarearchitektur wird auf wiederverwendbare Architekturkonzepte, wie z.B. Muster oder Referenzarchitekturen, zurückgegriffen. Diese in dieser Arbeit als Architekturbausteine bezeichneten Architekturkonzepte beschreiben bewährte Lösungskonzepte für verschiedene Entwurfsprobleme.

Architekturbausteine stellen an ihre Anwendung Bedingungen, die über die von ihnen definierten Richtlinien hinsichtlich Struktur und Verhalten hinausgehen. Eine Softwarearchitektur ist nur dann konsistent zu den auf sie angewendeten Architekturbausteinen, wenn deren Bedingungen von der Architektur eingehalten werden. Um die Qualität der Softwarearchitektur zu sichern, müssen die Bedingungen nicht nur bei der Anwendung an sich, sondern über die vollständige Entwicklungszeit hinweg gültig sein. Mit der Größe und Komplexität eines Softwaresystems steigt potentiell auch die Anzahl der angewendeten Architekturbausteine und das Risiko für die Verletzung ihrer Bedingungen. Für den bausteinbasierten Architekturentwurf ist daher ein flexibler Mechanismus erforderlich, der die Anwendung von Bausteinen mit der Überwachung ihrer Bedingungen kombiniert.

In dieser Arbeit wird daher ein umfassendes Konzept erarbeitet, das Softwarearchitekten bei der Erstellung von großen und komplexen Softwaresystemen basierend auf Architekturbausteinen unterstützten soll. Grundlage bildet dabei eine Beschreibungssprache für bausteinbasierte Softwarearchitekturen sowie für Architekturbausteine und ihre Bedingungen. Darauf aufbauend wird ein Mechanismus zur Anwendung von Architekturbausteinen definiert, der die Einhaltung der Bedingungen sicherstellt.

Für die Beschreibung von Softwarearchitekturen und Architekturbausteinen werden zwei voneinander unabhängige Teilmetamodelle definiert, die durch einen Integrationsteil zum Nachvollziehen der Bausteinanwendungen verbunden werden. Modelle dieses Metamodells werden sowohl durch eine semiformale Sprache beschrieben als auch durch prädikatenlogische Strukturen formal repräsentiert. Dazu passend werden die Bedingungen der Architekturbausteine durch prädikatenlogische Sätze formuliert. Das erarbeitete Konzept für den bausteinbasierten Architekturentwurf wird anhand eines Beispielsystems ausgewertet. Hierfür werden die Architekturbausteine, die in diesem Beispielsystem zur Anwendung kommen, entsprechend der entwickelten Sprache beschrieben und formal repräsentiert. Zugleich werden Bedingungen für ihre konsistente Anwendung festgelegt. Ergänzend wird ein Entwurf für eine Werkzeugunterstützung des bausteinbasierten Architekturentwurfs entwickelt.

Die Anwendung auf das Beispielsystem hat gezeigt, dass der in dieser Arbeit entwickelte bausteinbasierte Architekturentwurf praktisch einsetzbar ist. Dabei veranschaulicht die Formalisierung der unterschiedlichen Architekturbausteine und ihrer Bedingungen die Flexibilität und Fähigkeit des Ansatzes, verschiedene Architekturkonzepte beschreiben zu können. Zudem demonstriert das Beispiel die Eignung des Ansatzes, die Einhaltung der Bedingungen über die vollständige Entwicklungszeit hinweg zu überwachen. Insgesamt bildet diese Arbeit damit die konzeptionelle Grundlage für eine flexible Werkzeugunterstützung zur systematischen Anwendung von wiederverwendbaren Architekturkonzepten.

Danksagung

Nach Jahren intensiver Arbeit habe ich nun das Ziel erreicht und diese Arbeit fertiggestellt. Auf diesem Weg haben mich viele Menschen begleitet und direkt oder indirekt unterstützt. Ich möchte daher an dieser Stelle die Gelegenheit nutzen, mich bei allen zu bedanken; vor allem denen, die im Folgenden nicht noch einmal namentlich genannt werden: Vielen, vielen Dank!

Zunächst danke ich meinem Doktorvater Prof. Dr. Andreas Rausch, der mich dazu ermutigt hat, mich mit dem Thema des bausteinbasierten Architekturentwurfs in dieser Arbeit auseinanderzusetzen. Diskussionen mit ihm und seine kritischen Anmerkungen haben mich immer wieder zu neuen Blickwinkeln angeregt und zu neuen Denkanstößen geführt. Besonders bedanken möchte ich mich für die Unterstützung in den letzten zwei Jahren, in denen wir uns teilweise mehrmals pro Woche über diese Arbeit ausgetauscht haben. Des Weiteren bedanke ich mich für die lehr- und abwechslungsreiche Zeit am Lehrstuhl, in der ich in Projekten und bei der Unterstützung in der Lehre viele Erfahrungen sammeln konnte.

Des Weiteren danke ich Prof. Dr. Jörg P. Müller für seine Bereitschaft, das Zweitgutachten für diese Arbeit zu übernehmen. Zudem haben mir seine Anmerkungen geholfen, diese Arbeit zu verbessern.

Meinen Kollegen am Lehrstuhl danke ich für die gute Zusammenarbeit in Forschung, Lehre und den vielfältigen anderen Aufgaben, die am Lehrstuhl angefallen sind. Besonders die vielen, oft humoristischen Erlebnisse auf Kolloquien, in der Küche oder am Kickertisch haben den Arbeitsalltag immer wieder aufgelockert, die Gemeinschaft gestärkt und werden mir unvergesslich bleiben. Insbesondere bedanke ich mich bei Dirk Herrling für das Korrekturlesen verschiedener Kapitel dieser Arbeit.

Mein ganz besonderer Dank gebührt Sebastian Herold. In den vergangenen Jahren konnte ich mit ihm jederzeit über verschiedene Fragestellungen dieser Arbeit diskutieren. Zudem hat Sebastian umfangreiche Teile dieser Arbeit Korrektur gelesen. Über das fachliche hinaus war er mir in den letzten Jahren eine große moralische Unterstützung. Danke, dass du immer für mich da warst und mich immer wieder motiviert hast.

Meiner Familie und insbesondere meinen Eltern danke ich für die fortwährende Unterstützung während meiner kompletten akademischen Ausbildung. Meiner Schwester Viktoria danke ich überdies für das Korrekturlesen einiger Teile dieser Arbeit.

Inhaltsverzeichnis

Einordnung und Verwandte Arbeiten

Lösungskonzept

Umsetzung und Ergebnisse

Anhang

1. Einleitung

Inhalt

1.1. Motivation

Eine gute und sorgfältig konzipierte Softwarearchitektur legt die Basis für ein hochwertiges Softwaresystem. Nach der Anforderungserhebung ist sie das erste Ergebnis im Entwicklungsprozess hin zur Realisierung eines Softwaresystems [Som10]. In ihrer Funktion als grundlegendes Rahmenwerk für ein Softwaresystem hat die Softwarearchitektur einen großen Einfluss auf das später entwickelte System und auf dessen Qualität [BCK03, MB01]. Mängel im Design der Softwarearchitektur führen u.a. zu einem fehlerhaften oder schlecht wartbaren System. Darüber hinaus erhöhen sich Kosten und Aufwand für die Behebung von Fehlern, je später sie während des Entwicklungsprozesses oder sogar erst zur Laufzeit aufgedeckt werden [ZC09, KM07, Wes02].

Mit zunehmender Größe und Komplexität der Softwarearchitektur eines zu entwickelnden Systems steigt auch das Risiko für Fehler. Gleichzeitig gewinnen Qualitätsanforderungen wie Wartbarkeit an Relevanz, womit dem Architekturentwurf insgesamt eine wachsende Bedeutung zukommt. Zur Vereinfachung des Architekturentwurfs werden wiederverwendbare Architekturelemente, wie Komponenten, in Kombination mit wiederverwendbaren Architekturkonzepten eingesetzt. Gerade der Entwurf großer und komplexer Softwaresystemen kann durch solche Architekturkonzepte unterstützt werden [BMR+96].

Wiederverwendbare Architekturkonzepte resultieren aus den Erfahrungen, die Softwarearchitekten über Jahre hinweg gesammelt haben. Diese Konzepte konservieren das Wissen und dienen als bewährte Lösungsschablonen für verschiedene Problemsituationen. Zu solchen Architekturkonzepten zählen neben Architekturmustern [BMR+96, Cop96, Sha96, Fow02, GS94, GHJV94] auch Architekturprinzipien [GKRS12, Bal11, BRSV00] und Referenzarchitekturen [BCK03, Fow02]. Sie definieren Richtlinien, wie Architekturelemente (Komponenten, Konnektoren und weitere) auf struktureller Ebene oder in Bezug auf ihr Verhalten zusammenspielen. Die häufige Verwendung dieser Konzepte in der Praxis [HA08, Mic09, SAB10] sowie ihre Anzahl und ihr Variantenreichtum [Ris00] zeugen davon, dass sie in der Architekturgemeinde als wichtiger Bestandteil akzeptiert und für die Architekturerstellung von Bedeutung sind. Solche Konzepte werden in der vorliegenden Arbeit als die wesentlichen Artefakte des Architekturentwurfs verstanden und als *Architekturbausteine* bezeichnet.

Bei der Entwicklung einer Softwarearchitektur sind Architekturbausteine nicht nur aufgrund ihrer Funktion als wiederverwendbare Lösungskonzepte von Bedeutung. Die Vorgaben, die ein Architekturbaustein hinsichtlich Struktur und Verhalten macht, beeinflussen verschiedene Qualitätseigenschaften eines Softwaresystems [BCK03, Cru09]. Dementsprechend werden

Architekturbausteine nicht nur aufgrund ihrer Unterstützung bei der Umsetzung von funktionalen Aspekten ausgewählt, sondern auch im Hinblick auf nichtfunktionale Anforderungen [ZZGL08, HMRS07, AW02, GY01].

Alleine die Verwendung von entsprechenden Architekturbausteinen stellt nicht sicher, dass die Architektur eines Softwaresystems die gewünschten Qualitätseigenschaften aufweist. Durch seine Anwendung in einer Softwarearchitektur kann ein Architekturbaustein mit anderen Architekturbausteinen kombiniert werden. In Folge der Kombination vermischen sich die Strukturen und Verhalten, die die einzelnen Architekturbausteine implizieren. Unter Umständen werden durch die Kombination – oder auch schon durch die Anwendung an sich – unbemerkt Voraussetzungen für die Qualitätseigenschaften verletzt, die einer der beteiligten Architekturbausteine mit sich bringt. Als Konsequenz weicht die erstellte Softwarearchitektur von den Vorgaben ab und erfüllt nicht mehr die verlangten (Qualitäts-)Anforderungen.

Für die Realisierung einer Softwarearchitektur mit möglichst hoher Qualität sollte sichergestellt werden, dass die Voraussetzungen für die Qualitätseigenschaften erhalten bleiben. Die Menge der Architekturbausteine, ihre Variabilität bei der Anwendung und ihre Kombinationsmöglichkeiten untereinander schließen die Vorgabe fester Kombinationsmuster aus. Zur Unterstützung des bausteinbasierten Architekturwurfs ist vielmehr ein flexibler Mechanismus erforderlich. Dieser Mechanismus muss bei der Anwendung von Architekturbausteinen die Einhaltung der Voraussetzungen für die Qualitätseigenschaften sicherstellen.

Im Bereich der Architekturbeschreibungssprachen oder der Beschreibung und Komposition von Architektur- oder Designmustern adressieren viele Arbeiten Teile des beschriebenen Problems, doch fehlen ihnen oftmals wesentliche Aspekte. Viele dieser Ansätze lassen die Möglichkeit vermissen, Architekturbausteine in einer Form zu beschreiben, die nach ihrer Anwendung in der Architektur zudem noch als Einheiten identifizierbar sind. Die Identifizierbarkeit von angewendeten Bausteinen ist aber hinsichtlich der Überwachung auf Erhaltung der Eigenschaften relevant. Des Weiteren besitzen viele Arbeiten einen nur eingeschränkten Kombinationsbegriff für Muster oder vergleichbare Konzepte, der eine flexible Kombination nicht erlaubt oder eine Evolution der Softwarearchitektur erschwert.

1.2. Ziele und Ergebnisse der Arbeit

Ziel dieser Arbeit ist die Erarbeitung eines umfassenden Konzepts, um Softwarearchitekten bei der Erstellung der Architektur von großen und komplexen Softwaresystemen zu unterstützen. Die Erkenntnisse dieser Arbeit sollen dabei den systematischen und korrekten Einsatz von wiederverwendbaren Architekturkonzepten, die als Architekturbausteine erfasst werden, erleichtern. Wesentliche Aspekte dieses Konzepts hin zu einem bausteinbasierten Architekturentwurf und Ergebnisse der vorliegenden Arbeit sind:

- *Formale Beschreibung von Architekturmodellen und Architekturbausteinen*

  Eine formale Beschreibung erlaubt die präzise Modellierung von Architekturen und Architekturbausteinen. Zudem ist sie Grundlage für die Realisierung einer Werkzeugunterstützung des bausteinbasierten Architekturentwurfs. In dieser Arbeit wird zunächst das Metamodell einer semiformalen Beschreibungssprache durch UML-Klassendiagramme definiert. Für Modelle dieser Sprache wird im weiteren Verlauf eine Abbildung auf prädikatenlogische Strukturen angegeben. Um die Unabhängigkeit von Architekturen und Architekturbausteinen zu unterstützen, beschreibt die verwendete Sprache diese durch

unterschiedliche Teilsprachen getrennt voneinander. Die Distanz zwischen Architektur- und Bausteinbeschreibung wird durch einen dritten Teilbereich der Beschreibungssprache überbrückt. Dieser stellt Sprachmittel zur Verfügung, um die Anwendung eines Architekturbausteins in einer Softwarearchitektur eindeutig nachvollziehen zu können.

- *Formale Formulierung und Überprüfung von Bedingungen für die Anwendung von Architekturbausteinen*

  Zur Übertragung der mit einem Architekturbaustein assoziierten Qualitätseigenschaften auf eine Softwarearchitektur, stellt dieser Architekturbaustein Bedingungen an seine Anwendung. Eine formale Formulierung präzisiert diese zumeist informell beschriebenen Bedingungen. Die Bedingungen werden dabei im Kontext eines Architekturbausteins und unabhängig von einer konkreten Architekturbeschreibungssprache formuliert. Zur Auswertung der Bedingungen werden daher zudem Abbildungen zwischen Baustein- und Architekturkonzepten definiert. Durch eine Formulierung mit prädikatenlogischen Ausdrücken können die Bedingungen direkt über einem formalen Architekturmodell ausgewertet werden.

- *Operationalisierung der Anwendung von Architekturbausteinen*

  Durch eine Operationalisierung wird die Durchführung einer Bausteinanwendung formal und eindeutig definiert. Ausgangspunkt und Ergebnis der definierten Funktionen sind die mathematischen Strukturen, die die Architekturmodelle formal repräsentieren. Schließlich ist die Operationalisierung auch Grundlage für eine (teil-)automatisierte Werkzeugunterstützung des bausteinbasierten Architekturentwurfs.

Ergebnis der Arbeit ist damit ein durchgängiger Ansatz, der im Gegensatz zu anderen aktuellen Forschungsansätzen nicht nur einzelne Aspekte der Problemstellung adressiert. Dieser Ansatz ermöglicht neben Mustern auch andere wiederverwendbare Architekturkonzepte als Architekturbausteine zu erfassen und zu formalisieren und darüber hinaus konsistent anzuwenden und miteinander zu kombinieren.

1.3. Struktur der Arbeit

Nachfolgend wird ein Überblick über die Struktur und den Inhalt dieser Arbeit gegeben. Kapitel 2 *(Einführung in Grundbegriffe der Softwarearchitektur)* führt in die Grundlagen der Softwarearchitektur ein und erläutert die verschiedenen Arten von Architekturkonzepten, die in dieser Arbeit als Architekturbausteine bezeichnet werden. In Kapitel 3 *(Verwandte Arbeiten)* werden andere Forschungsansätze betrachtet, die Gemeinsamkeiten mit der vorliegenden Arbeit aufweisen. Dabei wird sowohl auf Beschreibungssprachen für Softwarearchitekturen und Muster als auch auf Möglichkeiten zur Konsistenzprüfung von Architekturen eingegangen. Kapitel 4 *(Problemstellung am Beispiel)* führt ein Beispielsystem ein, an dem in den folgenden Kapiteln die erarbeiteten Konzepte des bausteinbasierten Architekturentwurfs illustriert werden. Außerdem wird an diesem Beispiel die Problemstellung dieser Arbeit erläutert und die zu lösenden Teilprobleme identifiziert. Das Kapitel schließt mit der Formulierung der in dieser Arbeit adressierten Forschungsfragen.

Kapitel 5 *(Konzept des bausteinbasierten Architekturentwurfs)* erläutert die konzeptionellen Grundlagen dieser Arbeit und ordnet die Verwendung von Architekturbausteinen in einen Architekturentwurfsprozess ein. In Kapitel 6 *(Beschreibung von bausteinbasierten Architekturen)*

wird ein gemeinsames Metamodell zur Beschreibung von Architekturmodellen und Architekturbausteinen definiert. Zudem bietet dieses Metamodell die Möglichkeit, zu beschreiben, wo und wie ein Architekturbaustein in einer Architektur angewendet wurde. Auch auf die Formulierung der Bedingungen, die ein Architekturbaustein an seine Anwendung stellt, wird in diesem Kapitel eingegangen. Die verschiedenen Aspekte der Beschreibung werden dabei fortlaufend am in Kapitel 4 eingeführten Beispielsystem veranschaulicht. Wie das zuvor definierte Metamodell und darauf basierende Modelle durch prädikatenlogische Strukturen formalisiert werden, wird in Kapitel 7 *(Formalisierung des bausteinbasierten Architekturentwurfs)* festgelegt. Anschließend werden auf Basis dieser formalen Modelle Funktionen zur Operationalisierung des bausteinbasierten Architekturentwurfs spezifiziert.

Die in den vorherigen Kapiteln erarbeiteten Konzepte werden in Kapitel 8 *(Fallstudie zur Anwendung des bausteinbasierten Architekturentwurfs)* auf das Beispielsystem angewandt und ausgewertet. Hierzu werden zunächst die im Beispielsystem verwendeten Architekturbausteine formalisiert. Im Anschluss werden verschiedene Entwicklungsschritte im Entwurf des Beispielsystems vollzogen. Deren Ergebnisse werden hinsichtlich der Einhaltung der Bedingungen untersucht, die die verwendeten Architekturbausteine an ihre Anwendung stellen. Kapitel 9 *(Entwurf einer Werkzeugunterstützung)* präsentiert den Entwurf für ein Werkzeug zur teilautomatisierten Unterstützung des bausteinbasierten Architekturentwurfs. Abschließend werden in Kapitel 10 *(Fazit)* die Ergebnisse dieser Arbeit zusammengefasst und diskutiert.

Zusätzlich finden sich im Anhang ergänzende Informationen zu den vorherigen Kapiteln. Anhang A *(Integrierte Metamodelle)* stellt die beiden Teilmetamodelle für Architektur und Bausteine, die in Kapitel 6 über verschiedene Diagramme verteilt sind, jeweils in einem einzigen Diagramm integriert dar. In Anhang B *(Ausformulierte Eigenschaftsbedingungen der im Beispielsystem verwendeten Architekturbausteine)* werden die Eigenschaftsbedingungen ausführlich als prädikatenlogische Sätze notiert, die in Abschnitt 8.2 nur abkürzend eingeführt wurden. Anhang C *(Formale Repräsentation der Architekturbausteine des Beispielsystems)* enthält die vollständige formale Repräsentation der im Beispielsystem verwendeten Architekturbausteine.

2. Einführung in Grundbegriffe der Softwarearchitektur

Inhalt

Gegenstand dieser Arbeit ist ein Konzept zum Entwurf von Softwarearchitekturen, der wesentlich auf Architekturkonzepten wie Mustern, Prinzipien und Referenzarchitekturen basiert. Dieses Kapitel soll daher eine Einführung in die grundlegenden Begriffe dieser Arbeit geben. Abschnitt 2.1 gibt zunächst einen Überblick über Grundlagen der Softwarearchitektur und erläuert, wie der Begriff der Softwarearchitektur in dieser Arbeit verstanden wird. Verschiedene Architekturkonzepte – die Bausteine zum Architekturentwurf in dieser Arbeit – werden in Abschnitt 2.2 vorgestellt.

2.1. Was ist eine Softwarearchitektur?

Die Anfänge des Gebiets der Softwarearchitektur reichen zurück in die späten 60er Jahre des 20. Jahrhunderts. Im Tagungsband einer 1969 von der *North Atlantic Treaty Organization* (NATO) organisierten Konferenz wird das erste Mal der Begriff der Softwarearchitektur verwendet [KOS06, BR70]. In den Folgejahren wurden durch verschiedene Autoren wegbereitende Arbeiten veröffentlicht. Mit dem Verständnis eines Moduls als Umsetzung des Prinzips des *Information Hidings* führte Parnas in [Par72] ein Konzept ein, das als Vorläufer des Komponentenkonzepts verstanden wird. Ein weiterer Meilenstein ist das Buch *The Mythical Man Month* [Bro75], in dem der Autor Brooks Konzepte zu Entwurf und Organisation von Softwaresystemen beschreibt.

Den Durchbruch als eigenständiges Forschungsgebiet erlangte die Softwarearchitektur allerdings erst Anfang der 90er Jahre. Als Anstoß dieser Entwicklung wird vor allem die Veröffentlichung *Foundations for the Study of Software Architecture* von Perry und Wolf [PW92] betrachtet [KOS06]. Im Jahr 2000 wurden die bisherigen Erkenntnisse schließlich im ANSI/IEEE-Standard 1471-2000 *Recommended Practice for Architectural Description of Software-intensive Systems* [1471] festgehalten, der im Jahr 2011 durch den ISO/IEC/IEEE-Standard 42010-2011 *Systems and Software Engineering - Architecture Description* [42010] eine Überarbeitung erfuhr.

Trotz dieser beiden Industriestandards gibt es nicht die eine Definition des Begriffs der *Softwarearchitektur*. Davon zeugen die über 100 verschiedenen Definitionen dieses Begriffs, die das Software Engineering Institut der Carnegie Mellon Universität zusammengetragen hat [SEI].

Gemeinsam ist vielen Definitionen aber die Auffassung, dass eine Softwarearchitektur aus Elementen – die oft als *building blocks* oder Komponenten bezeichnet werden – und Beziehungen zwischen diesen Elementen besteht. Dieses Verständnis gibt auch der ANSI/IEEE-Standard 1471-2000 wieder, der eine Softwarearchitektur als die grundlegende Organisation eines Softwaresystems bezeichnet. Außerdem bezieht diese Definition die Umgebung des Systems und die Prinzipien der Architekturentwicklung mit ein:

> *„[Software architecture is] the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."*
>
> [ANSI/IEEE 1471-2000]

Die überarbeitete Definition im ISO/IEC/IEEE-Standard 42010-2011 abstrahiert die vorherige Definition und stellt statt des Organisationsaspekts die Konzepte und Eigenschaften eines Systems in den Vordergrund. Zudem wird statt von Komponenten allgemein nur noch von Elementen eines Softwaresystems gesprochen:

> *„[Software architecture is] the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution."*
>
> [ISO/IEC/IEEE 42010-2011]

Der Architekturbegriff in dieser Arbeit ordnet sich zwischen diesen beiden Definitionen ein. Zunächst legt die Architektur eines Softwaresystems die grundlegenden Eigenschaften, Konzepte und Strukturen des Systems fest. Elementare Elemente in der Beschreibung einer Architektur sind dabei Komponenten und die Beziehungen zwischen Komponenten. Wesentlich, vor allem für den Entwurf der Architektur, sind hingegen die in dieser Arbeit als Architekturbausteine bezeichneten Architekturkonzepte.

Die zentralen Beschreibungselemente einer Softwarearchitektur sind Komponenten. Komponenten sind modulare Elemente, die als Einheit zur Komposition dienen und in ihrer Umgebung austauschbar sind [Szy02, Obj11]. Sie kapseln ihre interne Realisierung und definieren durch Schnittstellen, die das nach außen sichtbare Verhalten einer Komponente beschreiben, Interaktionspunkte zu ihrer Umgebung. Hierbei wird zwischen angebotenen und benötigten Schnittstellen unterschieden. Über angebotene Schnittstellen stellt eine Komponente die von ihr realisierte Funktionalität zur Verfügung. Hingegen legen benötigte Schnittstellen fest, welche Funktionalität eine Komponente von ihrer Umgebung für die Realisierung (eines Teils) ihrer Funktionalität benötigt. Darüber hinaus sind Komponenten strukturierende Elemente, die als Einheiten der hierarchischen Dekomposition eines Softwaresystems dienen [Obj11, GKRS12].

In der Beschreibung einer Architektur werden Komponenten mit ihren Eigenschaften wie Schnittstellen sowie ihren gegenseitigen Beziehungen und Interaktionen festgehalten. Das Modell einer konkreten Softwarearchitektur wird dabei durch Sichten aus unterschiedlichen Perspektiven beschrieben. Jede Sicht stellt einen ausgewählten Aspekt in den Mittelpunkt der Beschreibung und abstrahiert von für die Sicht nicht relevanten Details. Für die Dokumentation einer Softwarearchitektur wird i.d.R. eine Kombination sich gegenseitig ergänzender Sichten verwendet. Unterschiedliche Autoren haben diesbezüglich verschiedene Standards definiert. Die eingesetzten Sichten variieren dabei ebenso wie deren Bezeichnung oder die Details, die beschrieben werden sollen [Kru95, HNS00, CBB+10, arc42]. Gemeinsam ist diesen und weiteren Arbeiten eine Kombination von Sichten, so dass insgesamt strukturelle, Verhaltens- und Verteilungsaspekte betrachten werden [RH06].

2.2. Was sind Architekturkonzepte?

Im Architekturentwurf sehen sich die Softwarearchitekten regelmäßig mit ähnlichen Problemstellungen konfrontiert. So ähneln sich beispielsweise Struktur und Verhalten von Systemen für Webshops unabhängig von der konkreten Geschäftsdomäne. Architekten können daher das Wissen und die Erfahrungen, die sie in dem einen System gesammelt haben, auf ein anderes übertragen. Durch eine standardisierte Beschreibung, die von der konkreten Problemstellung abstrahiert, steht dieses Wissen als wiederverwendbares Architekturkonzept auch anderen Softwarearchitekten zur Verfügung. Unterschieden werden diese Architekturkonzepte hier in Muster (Abschnitt 2.2.1), Prinzipien (Abschnitt 2.2.2) und Referenzarchitekturen (Abschnitt 2.2.3).

2.2.1. Muster

Eine Art der Architekturprinzipien sind Muster [GHJV94, BMR+96, Fow02, RH06]. Ein Muster abstrahiert von konkreten Elementen wie Komponenten, Klassen oder Objekten und beschreibt deren generelle Eigenschaften sowie deren Verantwortlichkeiten, Beziehungen und Interaktionen. Die Richtlinien, die ein Muster für Struktur und Verhalten definiert, dienen als Hilfestellung bei der Realisierung einer Anwendung. Dabei kann ein Muster in der Regel unabhängig von einer bestimmten Anwendungsdomäne eingesetzt werden. Durch ihre weite Verbreitung und meistens ausführliche Dokumentation bieten Muster Entwicklern zudem ein Vokabular, um Probleme und Lösungen für einen Systementwurf auf abstrakterer Ebene zu diskutieren. Zusätzlich zu den 25 ursprünglich von der *Gang of Four* [GHJV94] beschriebenen Designmustern wurden im Laufe der Jahre viele weitere Softwaremuster und deren Varianten definiert und beschrieben. Etwa 700 verschiedene Softwaremuster und ihre Varianten werden in [Ris00] aufgelistet.

In der Softwareentwicklung kommen verschiedene Arten von Mustern zum Einsatz. Für den Architekturentwurf sind besonders Architektur- und Designmuster von Interesse. Ein Architekturmuster beschreibt Funktionalität von Komponenten und Subsystemen sowie strukturellen und verhaltensbasierte Zusammenhänge zwischen ihnen. Designmuster beschreiben hingegen den Aufbau und das Zusammenspiel von Komponenten auf Implementierungsebene mit Klassen, Schnittstellen und deren Interaktionen. So schlägt z.B. das *Observermuster* [GHJV94] vor, die konkreten Klassen für das Subject und den Observer als Spezialisierung jeweils einer abstrakten Klasse oder eines Interfaces zu realisieren. Diese Implementierungsrichtlinien kann man ignorieren und nur die beiden Rollen, die das Muster mit Subject und Observer definiert, und deren Interaktionen betrachten. Eine solche abstrakte Version des Observermusters kann nun auch auf Komponenten angewendet werden. Ebenso wie das Observermuster können auch andere Designmuster abstrahiert und auf Architekturebene angewendet werden. Zudem bestehen Architekturmuster teilweise sogar aus Designmustern. Beispielsweise enthält das durch [BMR+96] als Architekturmuster eingestufte *Model-View-Controller*-Muster das Observermuster.

2.2.2. Prinzipien

Prinzipien [GKRS12, Bal11] beschreiben querschnittliche Eigenschaften und Richtlinien. In einem System angewendete Prinzipien sollten dabei nicht nur auf Teil, sondern idealerweise auf alle Elemente des Systems oder des jeweiligen Subsystems zutreffen. Ähnlich wie bei den Mustern kommen viele Prinzipien aus dem implementierungsnahen Entwurf. Aber auch hier

lassen sich verschiedene Vertreter abstrahieren und auf Architekturebene anwenden.

Beispiele für Prinzipien sind *Information Hiding* [Par72], *Separation of Concerns* [Eva04] oder *konzeptionelle Integrität* [Bal11]. Das Prinzip des Information Hidings besagt, dass die Interna einer Komponente gekapselt sind und nur ihre für den externen Zugriff definierten Schnittstellen nach außen sichtbar sind. Ziel des Prinzips *Separation of Concerns* ist es, unterschiedliche Funktionalitäten möglichst auch auf unterschiedliche Komponenten oder Subsysteme aufzuteilen und nicht durch eine einzelne Komponente zu realisieren. Beispielsweise sollten technische Belange und Aspekte der Anwendungslogik voneinander getrennt werden. Dieses Prinzip kann auch auf die Schnittstellen einer Komponente angewendet werden [Bal11]: Realisiert eine Komponente einen zusammenhängenden Aufgabenkomplex, sollten die Schnittstellen entsprechend der Einzelaufgaben aufgeteilt werden. Die konzeptionelle Integrität verlangt, dass Entwurfsentscheidungen im System einheitlich eingehalten werden. Zum Beispiel sollten für dieselbe Aufgabe nicht unterschiedliche Schnittstellen definiert werden.

2.2.3. Referenzarchitekturen

Eine Referenzarchitektur [RH06, Som10] ist eine Abstraktion einer standardisierten Softwarearchitektur. Sie definiert ein zumeist domänenspezifisches Grundgerüst für ein Softwaresystem. Eingesetzt wird eine Referenzarchitektur bereits zu Beginn des Architekturentwurfs und dient als Grundlage für den weiteren Architekturentwurf. Im Gegensatz zu Architekturmustern definiert eine Referenzarchitektur im Allgemeinen bereits das Grundgerüst für ein komplettes System und nicht nur für Teile des Systems. Eine Referenzarchitektur wird dabei oft in eine funktionale, eine logische und eine technische Ebene unterteilt. Der Fokus einer funktionalen Referenzarchitektur liegt auf dem Funktionsumfang eines Systems und seiner Aufteilung in Funktionsbereiche. Hingegen definiert eine logische Referenzarchitektur den strukturellen Aufbau inklusive abstrakter Kommunikationsbeziehungen eines Systems. Hierfür werden z.B. Schnittstellen benannt und ihre Merkmale umrissen, aber noch nicht detailliert festgelegt. Eine technische Referenzarchitektur konzentriert sich auf die technische Infrastruktur und die zu verwendenden Basistechnologien. Dies schließt u.a. die Festlegung einer Programmiersprache mit ein.

Ein Einsatzgebiet von Referenzarchitekturen ist der Entwurf von Softwaresystemen als Teil einer Anwendungslandschaft z.B. in Unternehmen oder Behörden. Der Einsatz einer gemeinsamen Referenzarchitektur reduziert vor allem bei Systemen mit einer ähnlichen Grundstruktur den Aufwand für die Architekturerstellung. Statt für jedes System eine Architektur von Grund auf neu zu erstellen, wird die Referenzarchitektur nur auf das konkrete System angepasst. Gleichzeitig erleichtert eine einheitliche Architektur die Wartung, auch wenn die Systeme möglicherweise von unterschiedlichen Auftragnehmern realisiert werden. Eine solche Referenzarchitektur wird z.B. mit der Register Factory®[BVA12] für die elektronischen Register im Bundesverwaltungsamt eingesetzt. Nicht nur Unternehmen oder Behörden, die ihre Softwaresysteme von anderen entwickeln lassen, definieren ggf. Richtlinien in Form von Referenzarchitekturen. Auch die entwickelnden Unternehmen konservieren die Erfahrungen ihrer Mitarbeiter oft in entsprechenden Standards. So definiert beispielsweise das IT-Beratungs- und Dienstleistungsunternehmen Capgemini sd&m mit Quasar [EKN+12] ebenfalls eine Referenzarchitektur.

In Referenzarchitekturen kommen oft auch (Architektur-)Muster zur Anwendung. Beispiele für solche Referenzarchitekturen mit angewendeten Mustern sind die 3-Schichten-Architektur [Fow02] und der Aufbau eines Compilers [RH06]. Die 3-Schichten-Architektur kommt für gewöhnlich bei Informationssystemen zum Einsatz. Sie unterteilt ein Softwaresystem logisch in

die drei Schichten Präsentation, Anwendungslogik und Datenhaltung, wobei das Schichtenmuster [BMR+96] zum Einsatz kommt. Für den Bau eines Compilers hat sich die Unterteilung in einzelne Einheiten etabliert. Jede dieser Einheiten ist für eine der Aufgaben lexikalische Analyse, syntaktische Analyse, semantische Analyse und Generierung verantwortlich. Die Einheiten und ihre Kommunikationskanäle zu benachbarten Einheiten werden dabei durch das Pipe-and-Filter-Muster [BMR+96] realisiert.

3. Verwandte Arbeiten

Inhalt

Dieses Kapitel gibt einen Überblick über Arbeiten, die Gemeinsamkeiten mit dem bausteinbasierten Architekturentwurf aufweisen. Zunächst handelt es sich dabei um Ansätze zur Beschreibung sowohl von Softwaresystemen und Architekturen (Abschnitt 3.1) als auch von Design- oder Architekturmuster (Abschnitt 3.2). Ein wesentlicher Aspekt im bausteinbasierten Entwurf ist zudem die Konsistenz zwischen Bausteinen und der resultierenden Architektur. Verwandte Arbeiten zur Überprüfung der Konsistenz werden in Abschnitt 3.3 betrachtet.

3.1. Beschreibung von Softwaresystemen und Architekturen

3.1.1. Unified Modeling Language (UML)

Die *Unified Modeling Language* (UML) [Obj11] ist eine Modellierungssprache, die von der *Object Management Group* (OMG) gepflegt wird. Ursprünglich wurde sie entwickelt, um Struktur und Verhalten von objektorientierten Systemen zu beschreiben. Seit der Version 2.0 unterstützt sie zudem die Modellierung von komponentenbasierten Architekturen. Um die strukturellen und verhaltensbasierten Aspekte eines Systems zu fassen, bietet die UML verschiedene Notations- sowie Diagrammarten an. Neben einer grafischen Notation als Teil der UML Spezifikation bietet die OMG mit dem Format *XML Metadata Interchange* (XMI) [Obj14d] eine textuelle Repräsentation von UML-Modellen an. Darüber hinaus existieren verschiedene Arbeiten, die UML-Modelle und da besonders Klassendiagramme durch formale Sprachen repräsentieren [AP03, SZ08, EWKM10].

Design- oder Architekturmuster werden strukturell oft durch UML-Klassendiagramme beschrieben. Dies kann auf verschiedene Umstände zurück geführt werden. Zum einen haben viele

Designmuster einen objektorientierten Charakter, der mit der Darstellung als Klassendiagramm harmoniert. Zum anderen ist UML weit verbreitet und verstanden. Außerdem wurden bereits die ersten Dokumentationen von Designmustern durch [GHJV94] mit einer ähnlichen Notation illustriert. Ein Klassendiagramm definiert die Rollen eines Musters als Typen, die durch Klassen oder Interfaces dargestellt werden. Zusätzlich werden Beziehungen zwischen den Rollen durch Assoziationen oder Vererbung und Eigenschaften der Rollen durch Attribute oder Methodensignaturen festgelegt. Ein Muster wird dadurch instanziiert, indem Klassen des Anwendungsmodells die Typen des Musters spezialisieren. Allerdings vermischen sich auf diese Weise Vererbungsbeziehungen, die durch die Anwendungsdomäne motiviert sind, mit denen für die Musterinstanziierung. Konzeptionell wäre hier eine Trennung wünschenswert (*separation of concerns*). Ein weiterer Nachteil ist die nicht existente Möglichkeit, die Teilnehmer einer Musterinstanz logisch zusammenzufassen.

Mit Kollaborationen (*Collaboration*) bietet die UML einen Mechanismus, um explizit Muster zu definieren. Eine Kollaboration ist ein strukturiertes Element, das die Struktur einer Menge von kollaborierenden Elementen, den Rollen, beschreibt. Die Anwendung einer Kollaboration wird als eigenes Element (*CollaborationUse*) repräsentiert. In dessen Kontext werden die Rollen einer Kollaboration den Elementen eines Modells durch Bindungen zugewiesen. Kollaborationen trennen somit die Definition von der Anwendung eines Musters und erlauben die Identifikation von einzelnen Instanzen inklusive der durchgeführten Bindungen. Allerdings schränken Kollaborationen die Typen, die als Rollen betrachtet werden können, stark ein.

3.1.2. Model Driven Architecture (MDA)

Die modellgetriebene Softwareentwicklung [SVEH07] beschreibt ein generelles Konzept zur Generierung von Anwendungen oder Anwendungsteilen aus Modellen. Ziel ist es, ein Ausgangsmodell durch geeignete Transformationsschritte bis hin zu Programmcode immer weiter zu verfeinern und manuelle Änderungen auf einem möglichst abstrakten Niveau durchzuführen. Die *Model Driven Architecture* (MDA) [Obj14a, KWB03] der OMG ist eine Realisierung dieses Konzepts.

Für die verschiedenen Abstraktionsebenen definiert die MDA unterschiedliche Arten von Modellen. Das *Platform Independet Model* (PIM) beschreibt die Logik eines Anwendungssystems unabhängig von plattformspezifischen Informationen wie einer Technologie oder Programmiersprache. Durch einen Transformationsschritt wird das PIM in das *Platform Specific Model* (PSM) überführt. Das PSM verfeinert die Systembeschreibung um Details der Zielplattform (z.B. Java-spezifische Datentypen, wenn die Zielsprache Java ist). Durch einen weiteren Transformationsschritt wird aus dem PSM Programmcode generiert.

Die MDA stellt keine direkte Unterstützung für die Beschreibung oder Anwendung von Mustern zur Verfügung. Allerdings kann die Transformation zur Anwendung von Mustern genutzt werden. Passende Transformationsregeln können im Ausgangsmodell Strukturen identifizieren, die auf Muster hinweisen. Diese können dann entsprechend übersetzt und dabei z.B. um weitere Details angereichert werden. Solche Transformationsregeln können allerdings auch zu falsch positiven Musteridentifikationen führen und damit ein Ergebnis erzeugen, das nicht beabsichtigt ist. Auch sind Transformationsregeln recht starr und können eine mögliche Variabilität bei der Anwendung eines Musters nicht berücksichtigen.

3.1.3. Architecture Description Languages (ADLs)

Architectural Description Languages (ADLs) sind zumeist textuelle Sprachen mit wohldefinierter Syntax und Semantik, um Struktur und Verhalten von Softwarearchitekturen zu beschreiben. Ein breiter Konsens besteht darin, dass die Grundelemente einer ADL Komponenten, Konnektoren und Konfigurationen sind [MT00]. Komponenten werden als Elemente der Verarbeitung und der Speicherung von Daten betrachtet und definieren durch Ports und/oder Interfaces, wie mit ihnen interagiert werden kann. Konnektoren modellieren Interaktionen zwischen Komponenten. Konfigurationen beschreiben das Zusammenspiel von Komponenten und Konnektoren in Systemen oder Subsystemen. Verschiedene ADLs bieten zudem die Möglichkeit, eine Konfiguration durch eine Komponente oder einen Konnektor zu abstrahieren und damit die Komplexität der Beschreibung zu reduzieren [MT00]. Die Fähigkeit, Muster als eigenständige Elemente zu beschreiben, unterscheidet sich in den verschiedenen ADLs [Cle96, KA07]. ADLs, die eine solche Möglichkeit nicht bieten, werden im Folgenden nicht betrachtet.

Aesop von Garlan, Allen und Ockerbloom [GAO94] ist keine ADL im eigentlichen Sinn, sondern eine Entwicklungsumgebung, um Architekturen zu modellieren, denen jeweils genau ein Muster zugrunde liegt. Die zugrunde liegende Sprache ermöglicht die Beschreibung von Mustern durch Ableiten der musterspezifischen Sprachelemente von den Basiselementen aus *Aesop*. Ob, und wenn ja wie, eine Kombination von Mustern möglich ist, beschreiben die Autoren ebenso wenig wie die Möglichkeiten, Bedingungen an die Muster und ihre Instanzen zu stellen.

Die ADL *Wright* [All97] ermöglicht durch das Sprachelement Style die Definition von Mustern. Ein Style kann Typen für Komponenten, für Konnektoren und für Interaktionspunkte zwischen Konnektoren und Komponenten definieren. Im Style formulierte Bedingungen stellen sicher, dass die Verwendung des Styles innerhalb einer Systemkonfiguration korrekt erfolgt. Konzeptionell kann eine Systemkonfiguration nur auf einem Style basieren. Soll ein System basierend auf verschiedenen Mustern beschrieben werden, muss zuvor ein entsprechend kombinierter Style definiert werden. *Wright* bietet allerdings die Möglichkeit der hierarchischen Komposition, bei der eine Komponente eine Systemkonfiguration repräsentiert. Auch wenn dies vom Autor nicht explizit dargestellt wird, kann auf diese Weise ein System beschrieben werden, dessen Teilsysteme auf unterschiedlichen Styles basieren. Damit ist zumindest eine eingeschränkte Kombination von Mustern möglich.

In *Armani* von Monroe [Mon99] definiert ein Style ebenso wie in *Wright* Typen für die verschiedenen Beschreibungselemente der ADL. Im Unterschied zu *Wright* besitzt *Armani* ein Multitypkonzept. Dies erlaubt den Beschreibungselementen wie Komponenten oder Konnektoren gleichzeitig verschiedene Typen anzunehmen. Ebenfalls durch eine Typbeziehung wird dargestellt, auf welchen Styles ein durch eine Systemkonfiguration beschriebenes System basiert. Dadurch ist die Kombination verschiedener Styleinstanzen direkt auf Systemebene möglich und erfordert nicht die Neudefinition eines kombinierten Styles. Wie auch bei *Wright* können außerdem durch hierarchische Komposition Systeme beschrieben werden, deren Teilsysteme sich in den zugrunde liegenden Styles unterscheiden. *Armani* bietet allerdings keine Möglichkeit, in der Systemkonfiguration zwischen verschiedenen Styleinstanzen und ihren jeweiligen Elementen zu unterscheiden. Somit können keine instanzspezifischen oder instanzübergreifenden Bedingungen über Styles formuliert werden.

ADLs wie *AO-ADL* [PFT11] oder *AspectualACME* [GCB+06] nutzen aspektorientierte Konzepte. Mit Aspekten beschreiben die genannten Sprachen querschnittliche Eigenschaften, die die Interaktionen zwischen Komponenten betreffen. Dazu werden Aspekte zunächst auch als gewöhnliche Komponenten beschrieben. Ob eine Komponente als Aspekt betrachtet wird, wird

durch die Rolle bestimmt, die sie im Rahmen einer Interaktion einnimmt. Die Rolle einer Komponente wird in der Systemkonfiguration durch den Konnektor festgelegt, über den sie mit anderen Komponenten verbunden ist. Abläufe innerhalb von Komponenten bleiben zumindest mit den genannten Ansätzen unbeeinflusst. Dementsprechend können mit diesen Ansätzen in erster Linie Muster modelliert werden, die Interaktionen zwischen Elementen beschreiben. Eher strukturell orientierte Muster oder Muster, die interne Verhaltensprotokolle einzelner Elemente definieren, sind nur eingeschränkt modellierbar.

3.2. Beschreibung von Mustern

3.2.1. Formale Beschreibungen

In seiner Arbeit betrachtet Abd-Allah [AB96, AA96] Softwaresysteme mit sogenannten heterogenen Architekturen und deren Komposition. Als heterogen werden dabei Systeme verstanden, deren Teilsysteme Architekturen aufweisen, die jeweils auf unterschiedlichen Mustern basieren. Zur Formalisierung nutzt Abd-Allah die Sprache Z [Spi92], die auch die Definition von muster- und systemweiten Bedingungen erlaubt. Teilsysteme und damit die instanziierten Muster eines Systems werden in der Arbeit von Abd-Allah nur durch verbindende Konnektoren komponiert. Eine überlappende Kombination von Teilsystemen und Mustern ist nicht vorgesehen.

Im Fokus des Ansatzes von Mikkonen [Mik98] liegt vor allem die Beschreibung der Interaktionen in Designmustern. Entsprechend reduziert sind die strukturellen Sprachelemente, die sich auf Klassen und Beziehungen beschränken. Die strukturelle Kombination von Mustern erfolgt durch eine Art von Mehrfachvererbung, die zwei Klassen durch eine neue kombinierte ersetzt. Das Verhalten einer Klasse wird entsprechend der Vererbungsbeziehung umdefiniert. Die Instanziierung von Mustern erfolgt ebenfalls über einen Vererbungsmechanismus. In der Arbeit wird der Fall, dass eine „Rolle“ von mehreren Elementen übernommen wird, nicht berücksichtigt. Neben dem sprachlich stark eingeschränkten Metamodell bietet der Ansatz von Mikkonen keine Möglichkeit für die Formulierung von Bedingungen.

Dong [Don02, ACDL99] beschreibt die Struktur von Designmustern und Designmodellen durch prädikatenlogische Fakten, dessen Relationssymbole aus OO-Sprachelementen abgeleitet sind (z.B. Class, AbstractClass, inherits). Die Instanziierung eines Musters ist als Ersetzungsfunktion über den Parameternamen der Fakten definiert, die dieses Muster repräsentieren. Muster oder Modelle werden durch Vereinigung komponiert. Dabei werden die Namen von ausgewählten Elementen ersetzt, die im Schnitt der zu komponierenden Modelle liegen sollen. Global definierte Bedingungen der Art „C ist entweder eine Klasse oder eine abstrakte Klasse“ sollen eine grundlegende Konsistenz von komponierten Modellen sicherstellen.

Der von Dong entwickelte Ansatz zur Beschreibung von Mustern wird auch von Zitouni [ZBS11, Zit12] genutzt. Zitouni kombiniert die Musterbeschreibung von Dong mit LOTOS [BB87], einer Sprache zur Beschreibung von verteilten Systemen. LOTOS ist darauf ausgerichtet, die Interaktionen in einem System zu beschreiben. Die Verhaltensfokussierung zeigt sich auch darin, dass sowohl ein System als auch seine Subsysteme als Prozesse betrachtet werden. Der Ansatz von Zitouni wird insgesamt ähnlich motiviert wie die vorliegende Arbeit: Aus einer Bibliothek von Mustern werden Kandidaten ausgewählt und auf ein Softwaresystem angewendet. Bei Zitouni soll dabei am Ende Java-Programmcode generiert werden. Allerdings beschreibt er in seinen Veröffentlichungen in erster Linie die von ihm verwendeten Ansätze und lässt relevante Fragen unbeantwortet. So wird nicht erläutert, wie der Beschreibungsansatz von Dong mit LOTOS kombiniert wird. Auch geht Zitouni nicht darauf ein, wie in seinem

kombinierten Ansatz die Anwendung und Kombination von Mustern erfolgt.

Der Ansatz von Taibi [Tou07] ist ähnlich zu dem zuvor beschriebenen Ansatz von Dong. Taibi verwendet für die Struktur von Mustern und Modellen auch jeweils eine objektbasierte Sprache, die auf der Prädikatenlogik basiert. Die grundlegenden Konzepte für die Komposition und Instanziierung von Mustern sind sehr ähnlich zu denen von Dong. Ein wesentlicher Nachteil beider Arbeiten ist die nicht vorhandene Kennzeichnung von Musterinstanzen im Ergebnismodell. Damit ist es nicht möglich, Bedingungen zu formulieren, die sich sowohl auf das Zusammenspiel der Rollen eines Musters als auch auf das Zusammenspiel verschiedener Musterinstanzen beziehen.

Bayley und Zhu [BZ07, BZ08b, BZ08a, BZ10] nutzen GEBNF (G*raphically Extended Backus Naur Form*) [ZS06], um durch UML-Diagramme beschriebene Modelle zu formalisieren. Designmuster beschreiben sie durch eine Menge von Bedingungen über Modellen. Für die Bedingungen nutzen sie einen Formalismus, der der Prädikatenlogik ähnlich ist und um eigene Notationselemente ergänzt wurde. Der Ansatz von Bayley und Zhu ist weniger konstruktiv, sondern stärker auf die Überprüfung eines Modells in Hinsicht auf die korrekte Verwendung eines Musters ausgelegt. Entsprechend existiert kein explizites Konzept für die Instanziierung von Mustern. Der Ansatz erlaubt das Überlappen von Mustern im Modell. Zudem existiert ein Konzept, um aus bestehenden Musterbeschreibungen ein neues Muster zu kombinieren, wobei auch semantische Aspekte berücksichtigt werden. Bedingungen eines Musters beziehen sich nur auf den Kontext seiner Anwendung. Querbeziehungen zwischen verschiedenen Musteranwendungen können nur definiert werden, wenn ggf. durch Komposition ein neues Muster definiert wird, in das entsprechende Bedingungen integriert werden. Allerdings können so nicht verschiedene Anwendungen desselben Musters berücksichtigt werden.

3.2.2. Erweiterung der UML

Die UML als Ausgangspunkt für die Beschreibung von Mustern bietet den Vorteil, dass je nach Art und Umfang der Erweiterung bestehende Modellierungs- und Metamodellierungswerkzeuge verwendet werden können. Die UML kann dazu auf verschiedene Arten erweitert werden. Durch die Definition von Profilen kann eine leichtgewichtige Erweiterung vorgenommen werden. Komplexer ist hingegen eine Änderung des UML-Metamodells.

Elaasar, Briand und Labiche [EBL06] erweitern zur Modellierung von Mustern die auf der Metametamodellebene (M3) befindliche *Meta Object Facility* (MOF) [Obj14b]. Diese *Epattern* genannte Erweiterung realisieren die Autoren auf Basis von *Ecore* aus dem *Eclipse Modeling Framework* (EMF) [SBPM08]. Mit diesem Ansatz werden Muster auf der Metamodellebene (M2) und ihre Instanzen auf der Modellebene (M1) modelliert. Dies hat allerdings den Nachteil, dass die Einführung eines neuen Musters die Anpassung des Metamodells (M2) eines Systemmodells (M1) nach sich zieht. Jedes als EPattern repräsentierte Muster kann eine beliebige Menge von als EConstraint repräsentierte Bedingungen besitzen. Diese Bedingungen werden z.B. mit OCL formuliert und beziehen sich auf eine Musterinstanz.

Im Zentrum der gemeinsamen Arbeiten von Zdun, Kamal und Avgeriou [ZA08, KAZ08] stehen sogenannte *Architekturprimitive*. Architekturprimitive sind charakteristische Elemente eines Muster, die sich unter Umständen auch in anderen Mustern wiederfinden. Um Architekturprimitive zu modellieren, erweitern die Autoren die UML durch ein Profil mit Stereotypen für die Metamodellelemente Component, Interface, Port, Connector und Package. Eine Architekturprimitive wird dabei durch eine Menge von Stereotypen repräsentiert, allerdings wird ein Muster nicht explizit durch Architekturprimitive modelliert. Um sicherzustellen, dass die

Stereotypen im Modell konsistent eingesetzt werden und dementsprechend eine Architekturprimitive korrekt modelliert wird, werden OCL-Bedingungen formuliert. Überlappen sich die Realisierungen von jeweils der gleichen Architekturprimitiven, können die zusammengehörigen Modellelemente eventuell nicht eindeutig identifiziert werden. Dies kann zu falschen Ergebnissen bei der Auswertung der Bedingungen führen. Auch fehlt ein Instanzbegriff. Ohne einen solchen können keine Bedingungen formuliert werden, die das Zusammenspiel verschiedener Realisierungen von Architekturprimitiven betreffen.

3.2.3. MDA-basierte Ansätze

Im Mittelpunkt von MDA-basierten Ansätzen steht die automatische Instanziierung von Mustern, wozu geeignete Transformationsregeln definiert werden müssen. Um bestehende Werkzeuge nutzen zu können, wird als Modellierungssprache für die MDA-spezifischen Ansätze oft UML mit ggf. ergänzenden Profilen eingesetzt.

Kajsa, Majtas und Navrat [KMN11] spezifizieren Muster mit Hilfe eines UML-Profils. Dieses Profil definiert Stereotypen, die Rollen für verschiedene Realisierungsvarianten der Muster repräsentieren. Zusätzlich enthält das Profil eine Menge von OCL-Bedingungen, um die Konsistenz der Musterinstanzen sicherzustellen. Der Designprozess von einem PIM über ein PSM zum Programmcode erfolgt mit Unterstützung eines Werkzeug. Dabei können sowohl im PIM als auch im PSM Rollen eines Musters durch Stereotypen gekennzeichnet werden. Basierend auf den Stereotypen werden im Transformationsschritt vom PIM zum PSM z.B. für die Musterrealisierung notwendige Methoden ergänzt. Zusätzlich werden in diesem Transformationsschritt unvollständig gekennzeichnete Muster erkannt und die entsprechenden Elemente um passende Default-Stereotypen erweitert. Dieser Ansatz weist einige Schwachstellen auf. Designelemente können zwar als Rollen gekennzeichnet werden, allerdings besteht keine Möglichkeit Musterinstanzen voneinander abzugrenzen. Damit ist es auch nicht möglich, instanzspezifische Bedingungen auszuwerten.

3.2.4. Aspektorientierte Beschreibungen

Die aspektorientierte Softwareentwicklung (AOSD) ergänzt objektorientierte oder komponentenbasierte Beschreibungstechniken um das Beschreibungskonzept der Aspekte [FECA04]. Das Ziel der Aspektorientierung ist es, querschnittliche Eigenschaften, wie z.B. Logging oder Fehlerbehandlung, modular als Aspekte zu beschreiben. Außerhalb der Aspektorientierung wird solche Funktionalität für jedes einzelne betroffene Modellelement (z.B. Klasse oder Komponente) erneut beschrieben. Durch Aspekte werden diese Funktionalitäten nur noch einmalig generisch und dazu unabhängig von dem jeweiligen Modellelement formuliert. Sogenannte „Pointcuts“ spezifizieren, an welcher Stelle Aspekte in den Ablauf integriert werden. Umgesetzt wird das aspektorientierte Konzept bereits seit einiger Zeit in verschiedenen Programmiersprachen, wie z.B. durch AspectJ [KHH+01] als Erweiterung von Java [Ora13].

Muster haben Gemeinsamkeiten mit Aspekten. Das von Mustern implizierte Verhalten kann auch querschnittlich sein und trägt dann auch nur indirekt zur Kernfunktionalität eines Systems bei. Zudem kann ein Muster mehrere Klassen oder Komponenten betreffen. Denn ein Muster definiert vielfach mehrere Rollen und ebenso können in einer Softwarearchitektur mehrere Instanzen eines Musters auftreten. Hier können Aspekte möglicherweise zu einer Modularisierung beitragen.

Hachani und Bardou [HB02, HB03] modellieren die beiden syntaktisch ähnlichen Muster

Strategy und Visitor mit Aspekten. Dabei repräsentieren sie jeden konkreten Visitor durch einen Aspekt, was ihnen erlaubt, die generische Methode accept mit einem treffenderen Namen zu versehen. Hingegen definieren sie nur einen Strategy-Aspekt. Da sie ihr prinzipielles Vorgehen nicht dokumentieren und sich die Modellierungen der beiden ähnlichen Muster unterscheiden, ist das generelle Konzept der Autoren nicht zu erkennen. Nach eigener Aussage setzen sie pro Instanz eines Musters genau einen Aspekt ein, um die Nachvollziehbarkeit eines Muster zu verbessern.

Piveta und Zancanella stellen in [PZ03] eine Lösung zur Implementierung eines Designmusters mit AspectJ vor. Das Muster an sich repräsentieren sie durch einen abstrakten Aspekt, der die Variablen und Methoden zur Realisierung dessen Funktionalität definiert. Aus dem abstrakten Aspekt leitet sich ein konkreter Aspekt ab, der die Parameter einer Instanz dieses Musters bestimmt. Zudem implementiert der konkrete Aspekt die Methoden des Musters und definiert den Pointcut für die Ausführung der Methoden. Sie illustrieren ihren Ansatz am Beispiel des Observermusters, wobei sie nach [GHJV94] zwischen abstrakten und konkreten Subject und Observern unterscheiden. Dadurch, dass ein abstrakter Aspekt nur die Datentypen der abstrakten Subjects und Observer kennt, erreichen sie eine Trennung zwischen Definition und Instanz des Musters. Ein ähnliches Vorgehen haben Hannemann und Kiczales für verschiedene Muster gewählt [HK02].

Das grundlegende Konzept hinter Aspekten findet sich auch in der Modellierungssprache *LayOM* von Bosch [Bos98, Bos99] wieder, wobei die entsprechenden Modellierungselemente nicht Aspekte, sondern Layer genannt werden. Jeder Layer beschreibt wiederverwendbares Verhalten wie z.B. ein Muster. Anschaulich umschließt ein Layer ein Objekt eines *LayOM*-Modells und manipuliert die Interaktionen dieses Objekts mit anderen. Durch einen Layer verändert sich das Objekt an sich nicht und kann parallel ohne einen Layer oder kombiniert mit anderen Layern verwendet werden.

Charakteristisch für aspektorientierte Ansätze ist die Trennung zwischen dem querschnittlichen Verhalten der Muster und der eigentlichen Kernfunktionalität des Systems. Die Verhaltensbeschreibung des Systems wird durch Aspekte übersichtlich, weil generische Anteile als eigene modulare Einheiten ausgelagert sind. Auf der anderen Seite können diese generischen Anteile auch Teil des Kernverhaltens einer Klasse oder Komponente sein, wodurch deren Beschreibung lückenhaft erscheint. Beide Ansätze lassen sich bedingt nutzen, wenn Muster in erster Linie querschnittliche Funktionalität definieren. Aspekte können diese Funktionalität modular beschreiben. Anders verhält es sich bei Mustern, die grundlegende Strukturen, wie z.B. bei einem mehrschichtigen System, festlegen. Auch berücksichtigt keiner der Ansätze die Formulierung von zusätzlichen Bedingungen über die modellierten Elemente oder Muster.

3.2.5. Rollenbasierte Beschreibungen

Rollenbasierte Ansätze betrachteten verschiedene syntaktische Elemente oder verschiedene Verhaltensaspekte einer Musterbeschreibung als Rollen. Die Beschreibung von Mustern erfolgt durch die Kombination von geeigneten Rollen. Bei der Instanziierung eines Musters werden die Rollen den Elementen eines Architektur- oder Designmodells nur zugeordnet; diese Modellelemente sind keine Instanzen der Musterrollen. Die Zuordnung von Rollen überträgt dabei syntaktische oder semantische Eigenschaften auf das entsprechende Modellelement.

Muster werden von Riehle [Rie00] durch Rollenmodelle modelliert, die aus einer Menge von Rollentypen bestehen. Ein Rollentyp definiert dabei das Verhalten einer Rolle, die wiederum einen beobachtbaren Verhaltensaspekt eines Objekts repräsentiert. Zwischen den Rollentypen

eines Rollenmodells können objektorientierte Beziehungen wie Assoziationen oder Vererbung bestehen. Zusätzlich wird für die Rollentypen paarweise festgelegt, in wie weit diese im Kontext der Anwendung ihres Rollenmodells kombinierbar sind [Rie97]. Klassenmodelle modellieren die Anwendung von Rollenmodellen und bestehen neben den entsprechenden Rollenmodellen aus einem Klassendiagramm. Ein Klassenmodell kann somit Rollenmodelle kombinieren, wobei jede Klasse des Klassenmodells Rollen eines oder unterschiedlicher Rollenmodelle kombinieren kann. Für Klassenmodelle führt Riehle sowohl eine grafische Repräsentation als auch eine textuelle Spezifikation ein. Erstere ähnelt der Notation von UML-Klassendiagrammen, die um Elemente zur Darstellung der Rollenmodelle ergänzt ist. Die textuelle Notation ist an die Syntax der Programmiersprache Java angelehnt. Riehle definiert zwar Rollen als Verhaltensaspekte eines Objekts, beschreibt in seiner Arbeit allerdings weder das Verhalten von Rollen noch ein Konzept für deren Kombination. Außer der Einschränkung der Kombinierbarkeit von Rollentypen bietet der Ansatz keine Möglichkeit, um Bedingungen über Muster und deren Elemente zu definieren.

Im Gegensatz zu anderen Ansätzen betrachten Kim und Carrington [KC09] nicht nur Rollen, die von Objekten ausgefüllt werden. Neben solchen als Klassenrollen bezeichneten Rollentypen definieren sie Attributrollen, Operationenrollen, Parameterrollen und eine Menge verschiedener Beziehungsrollen. Ein Muster wird durch *Object-Z* [DR00] basierend auf verschiedenen Rollen spezifiziert. Zusätzlich werden mit *Object-Z* musterspezifische Bedingungen formuliert. Als Bindeglied zwischen Mustern und deren Instanzen werden Bindungsmodelle eingesetzt, die zu den Rollentypen kompatible Bindungstypen enthalten. Muster an sich können durch einen Ableitungsmechanismus und eventueller Umbenennung von Rollen komponiert werden. Ob auch eine Kombination von Musterinstanzen durch überlappende Instanziierung möglich ist, geht aus [KC09] nicht hervor. Ebenfalls nicht nachvollziehbar ist, ob eine muster- und instanzübergreifende Formulierung von Bedingungen möglich ist.

3.2.6. Pattern Languages

Sogenannte Pattern Languages [Zim95, BMR+96, KB03, AZ05] sind keine Sprachen, um Muster zu beschreiben. Muster können im Gegenteil als ihr Vokabular betrachtet werden. Pattern Languages beschreiben vielmehr die Beziehungen, in denen die einzelnen Muster zueinander stehen können. So kann ein Muster ein anderes enthalten oder eine Menge von Muster wird häufig in Kombination eingesetzt. Einige Pattern Languages ordnen die Muster unabhängig von ihren Beziehungen zusätzlich verschiedenen Kategorien zu. Diese Kategorien liegen teilweise orthogonal zueinander und ergeben sich z.B. aus dem groben Einsatzzweck oder dem Bereich einer Architektur, in dem sie häufig eingesetzt werden.

Kamal und Avgeriou [KA10] gehen mit ihrer Arbeit über die üblichen Pattern Languages hinaus. Statt zwischen Mustern haben sie die Beziehungen zwischen den einzelnen Bestandteilen verschiedener Muster untersucht. Der Fokus ihrer Untersuchungen lag auf der Kombination von Mustern und wie sich dabei deren Bestandteile zueinander verhalten. So wurde identifiziert, dass Bestandteile eines Musters durch Kombination z.B. vermischt werden oder zukünftig interagieren.

Für den bausteinbasierten Architekturentwurf sind Pattern Languages in Hinsicht auf die Beziehung zwischen den Mustern von Interesse. Mit einer entsprechenden Pattern Language kann definiert werden, welche Muster wie miteinander zu kombinieren sind, um möglichst die Konsistenz der Architektur zu wahren. Besonders der Ansatz von Kamal und Avgeriou kann durch detaillierte Richtlinien zur Konsistenz beitragen.

3.3. Überprüfung der Konsistenz von Architekturen

3.3.1. Statische Abhängigkeitsanalysen

Verfahren zur statischen Abhängigkeitsanalyse untersuchen eine Datenmenge (z.B. Programmcode) hinsichtlich vorhandener Beziehungen und Abhängigkeiten. Je nach Verfahren kann dabei gleichzeitig eine Quantifizierung oder auch eine Bewertung der Ergebnisse erfolgen.

Die *Design Structure Matrix* (DSM) [EB12] ist eine allgemeine Methode zur Darstellung und Analyse von Beziehungen zwischen Elementen eines Systems. Diese Methode basiert auf einer quadratischen Matrix, deren Zeilen und Spalten die zu untersuchenden Elemente repräsentieren (z.B. Komponenten oder Klassen eines Softwaresystems). Die Einträge der Matrix bewerten den Grad der Abhängigkeit von jeweils zwei Elementen. Dabei ist die Anwendung von DSMs nicht auf Softwaresysteme beschränkt, sondern kann auch in anderen Bereichen eingesetzt werden [Bro01]. DSMs können genutzt werden, um Regeln und Bedingungen an ein System auf Basis von analysierbaren Beziehungen auszuwerten. Da jede DSM nur jeweils eine bestimmte Beziehungsart betrachten kann, ist es für komplexere Bedingungen notwendig, DSMs mit unterschiedlichem Fokus zu kombinieren.

Das Ziel des *Software Reflexion Modeling* [MNS95] ist es, zwei Modelle mit unterschiedlichem Abstraktionsniveau (z.B. eine logische Softwarearchitektur und ein Quellcodemodell) miteinander zu vergleichen. Dazu werden manuell Abbildungen zwischen Elementen der beiden Modelle definiert. Aus diesen Informationen wird ein Reflexion Model generiert, das aufzeigt, wo und wo nicht die beiden zu vergleichenden Modelle konform zueinander sind. Das Konzept des *Software Reflexion Modeling* wird in verschiedenen Werkzeugen zur Konsistenzprüfung eingesetzt. Werkzeuge wie *Dependometer* [Dep] oder *SonarJ* und *Sotograph* [hello] untersuchen ein Zielmodell hinsichtlich der Einhaltung horizontaler und vertikaler Schichten oder Teilsysteme. Die Arten von Regeln, die dadurch überprüft werden können, sind entsprechend beschränkt. Das *SAVE*-Werkzeug [SAVE, LM08] ist flexibler in Bezug auf die Struktur der logischen Architektur und erlaubt es, beliebige Strukturen zu definieren. Die Autoren planen zudem die Erweiterung des Werkzeugs, um neben der Struktur auch das Verhalten von Softwaresystemen überprüfen zu können. Ähnlich flexibel ist *ConQAT* [ConQAT, DHHJ10], dass auch die Modellierung von hierarchisch komponierten Komponenten in der logischen Architektur zulässt. Keines dieser Werkzeuge unterstützt zunächst Muster im Allgemeinen. Es ist aber vorstellbar, dass einige strukturelle Aspekte von Mustern mit flexibleren Werkzeugen wie *ConQAT* überprüft werden können. Allerdings fehlen insgesamt die Möglichkeiten, um Verhalten zu untersuchen sowie komplexere Bedingungen von Mustern zu überprüfen.

3.3.2. Anfragesprachen und Constraint Languages

Anfragesprachen werden genutzt, um Informationen aus einer Quelle wie z.B. einer Datenbank abzurufen. Dazu wird in der Anfragesprache ein Ausdruck formuliert, der die gewünschten Daten beschreibt. Das Ergebnis einer Anfrage kann ein einzelnes Element, eine Menge von Elementen oder auch zusammengesetzte Strukturen aus verschiedenen Arten von Elementen sein. Mit Hilfe einer Anfrage kann auch die Einhaltung von Bedingungen geprüft werden. Hierzu wird eine Anfrage so formuliert, dass sie all jene Elemente liefert, die eine Bedingung verletzen. Ist die Ergebnismenge leer, wird die zugehörige Bedingung erfüllt. Ähnlich kann auch die Auswertung eines Ausdrucks einer Constraint Language verstanden werden: Eine Bedingung ist wahr, wenn keine Elemente existieren, die sie verletzen.

Einer der bekanntesten Vertreter solcher Sprachen ist die von der OMG definierte *Object Constraint Language* (OCL) [Obj14c]. Sie gehört letztendlich zu beiden der eingangs erläuterten Sprachkategorien. Diese Sprache wurde zunächst als Constraint Language konzipiert, kann aber seit Version 2.0 auch zur Formulierung von Anfragen genutzt werden. Die OCL ist eine deklarative Sprache mit einer prädikatenlogischen Grundlage [WK04], die als Ergänzung der UML gedacht ist. Formuliert werden OCL-Bedingungen im Kontext eines Modellelements und werden im Kontext einer Instanz dieses Elements ausgewertet. Es ist daher nicht möglich, eine OCL-Bedingung in dem Modell auszuwerten, in dem sie formuliert wurde.

Eine weitere Sprache, die auf Modellen arbeitet, ist in dem Framework *EMF-IncQuery* [BURV11, BHH+12] enthalten, welches der Auswertung von EMF-Modellen dient. Das Konzept hinter der Anfragesprache von *EMF-IncQuery* basiert auf Graphmustern. Diese Sprache bietet einige Eigenschaften direkt an, die in anderen Sprachen teilweise erst umgesetzt werden müssen. So kann auf Elemente einer geordneten Menge direkt über einen Index zugegriffen werden oder die transitive Hülle kann alleine durch Angabe von * bestimmt werden.

Die *Dependency Constraint Language* (DCL) [TV09] ist eine deklarative Constraint Language, um die modulare Organisation eines Systems zu beschränken. Hierzu werden zunächst Module definiert, die die Klassen der Java-Implementierung des vorliegenden Systems zusammenfassen. In DCL werden verschiedene Arten von Abhängigkeiten in objektorientierten Systemen (Methoden- oder Variablenzugriff, Objekterzeugung, Vererbung usw.) durch eigene Sprachelemente repräsentiert. Allerdings ist die Sprache auf diese Sprachelemente und damit die Bedingungen auf die Beschreibungen von Abhängigkeiten beschränkt.

Ähnlich wie DCL wurden viele Anfragesprachen und Constraint Languages für die Auswertung von Quellcode und nicht von Modellen entwickelt. Hierzu zählen auch die *Structural Constraint Language* (SCL) [DH06] und die Anfragesprache *JQuery* [JV03, Vol06]. Beide Sprachen haben die Ausdrucksmächtigkeit der Prädikatenlogik und folgen auch mit ihrer Syntax bekannten Logikprogrammiersprachen (*JQuery* stärker als SCL). Im Gegensatz zu anderen (Logik-)Sprachen (z.B. Prolog) ist *JQuery* zudem typisiert. In [DH09] wurde *JQuery* bereits genutzt, um die Einhaltung von Architekturregeln ähnlich zu den Bedingungen im Kontext von Mustern zu überprüfen. Dazu wurden Anfragen formuliert, die Verletzungen der Regeln im Programmcode identifizieren.

Speziell für die Konsistenzprüfung von Architekturen und ihrer Realisierung durch Designmodelle oder Programmcode haben sowohl Mens [Men00] als auch Herold [Her11, HR13, HR14] jeweils ein Framework konzipiert. Beide nutzen Logikprogrammierung und repräsentieren Architektur und Design oder Code durch eine logische Faktenbasis. Während das Framework von Mens Bedingungen über objektorientieren Konstrukten formuliert, ermöglicht das Framework von Herold zusätzlich auch auf komponentenbasierte Konzepte Bezug zu nehmen. Mit beiden Arbeiten können Bedingungen formuliert und ausgewertet werden, die auch durch Muster impliziert werden. Allerdings bietet keines der beiden Frameworks Konzepte an, um Muster an sich zu beschreiben und anzuwenden.

4. Problemstellung am Beispiel

Inhalt

In einem Softwareentwicklungsprojekt übernimmt oftmals einer der erfahrensten Entwickler die Rolle des Softwarearchitekten und ist für die Erstellung der Architektur verantwortlich [Kru99]. Aufgrund seiner Erfahrung erkennt er viele Probleme bereits im Vorfeld und kann sie vermeiden. Allerdings ist nicht an jedem Entwicklungsprojekt ein erfahrener Entwickler (Abschnitt 4.1) beteiligt oder die Art des zu entwickelnden Systems ist den Beteiligten bisher nicht vertraut. Unabhängig von der Erfahrung kann es durch die Vielfalt der möglichen Architekturalternativen zu Inkonsistenzen kommen, die sich im schlimmsten Fall im fertigen System durch Fehlverhalten auswirken [BCK03].

Ein erster Schritt in Richtung Vermeidung von Fehlern und Inkonsistenzen ist die Verwendung von standardisierten Architekturlösungen – den Architekturbausteinen (Abschnitt 4.2). Zur Erfüllung der Anforderungen an das Softwaresystem werden diese miteinander kombiniert. Aber alleine die Verwendung solcher Architekturbausteine führt noch nicht zu einer fehlerfreien Architektur. Wie das Beispiel in diesem Kapitel zeigt (Abschnitt 4.3), ist zudem die korrekte Integration eines Bausteins in eine bestehende Architektur relevant. Denn nur durch eine korrekte Integration können die speziellen Eigenschaften eines Architekturbausteins erhalten bleiben. Neben der Frage der Erhaltung dieser Bausteineigenschaften lassen sich an dem Beispiel weitere Fragestellungen und Probleme identifizieren, die für einen bausteinbasierten Architekturentwurf gelöst werden müssen (Abschnitt 4.4).

4.1. Ausgangssituation des Beispielszenarios

Die Problemstellung dieser Arbeit und die resultierenden Fragestellungen werden mit Hilfe eines fiktiven Szenarios verdeutlicht. Im Folgenden wird zunächst die Ausgangssituation in dem Szenario dargestellt: Ein kleines mittelständisches Unternehmen hat seine IT-Landschaft im Laufe der letzten 20 Jahre immer wieder um neue Produkte erweitert. Diese Produktvielfalt hat allerdings dazu geführt, dass mittlerweile für fast jeden Geschäftsprozess ein anderes System oder Programm verwendet werden muss. Zusätzlich haben sich in den verschiedenen Abteilungen des Unternehmens unterschiedliche Dateiformate für die Ablage der anfallenden Daten etabliert. Um Informationen aus verschiedenen Abteilungen zusammenzuführen, werden

diese von den Mitarbeitern oft per Hand von einem Dokument in ein anderes übertragen. Bei diesem manuellen Prozess ist es schon mehrmals zu gravierenden Fehlern gekommen.

Die aus der IT-Infrastruktur resultierenden Probleme hat mittlerweile auch der Unternehmensinhaber erkannt und daher vor kurzem einen jungen Softwareentwickler eingestellt. Dieser sollte zunächst verschiedene Programme mit individuellen Erweiterungen versehen, um zumindest an einigen Stellen manuelle Prozesse zu automatisieren. Aber auch diese Zwischenlösung weist noch viele Defizite auf, so dass der Unternehmensinhaber mit der Einführung eines einzigen unternehmensweiten Systems alle abteilungsspezifischen Insellösungen ersetzen möchte. Dabei bevorzugt er eine Eigenentwicklung, um allen Besonderheiten des Unternehmens gerecht zu werden. Seinen Softwareentwickler beauftragt er, dieses System zu entwerfen.

Abgesehen von den funktionalen Details hat der Entwickler die groben Anforderungen an das zu erstellende System schnell erfasst. So sollen alle Daten zentral in einer gemeinsamen Datenbank abgelegt werden. Jede Abteilung erhält eine für ihre Aufgaben angepasste Programmoberfläche, mit der die Mitarbeiter auf alle relevanten Daten zugreifen können. Außerdem sollen in Zukunft mobile Endgeräte eingesetzt werden, um Mitarbeitern im weitläufigen Produktionsgelände den Zugriff auf das System einfacher zu ermöglichen. Damit sich die einzelnen Abteilungen nicht wie bisher über die Veränderung verschiedener Daten benachrichtigen müssen, soll das Programm außerdem Mitarbeiter auf für sie relevante Datenaktualisierungen hinweisen. Zudem soll das System so aufgebaut werden, dass zukünftige Weiterentwicklungen einfach und flexibel möglich sind.

4.2. Architekturbausteine für das Beispielsystem

Unser Softwareentwickler hat nun eine grobe Vorstellung des neuen Systems gewonnen, das er komponentenbasiert entwickeln möchte. Für die verschiedenen sich ergebenen Entwurfsprobleme möchte er auf bereits bewährte und standardisierte Lösungen zurückgreifen. In Architekturbausteinen[1], die sowohl auf komponentenbasierte Systeme anwendbare Entwurfsmuster als auch verschiedene Architekturprinzipien beschreiben, findet er die gewünschten Standardlösungen. Unter Berücksichtigung der Anforderungen wählt er aus einem umfangreichen Katalog von Architekturbausteinen mehrere für das Grundgerüst des zu entwickelnden Systems aus: den Schichtenbaustein (Abschnitt 4.2.1), den Observerbaustein (Abschnitt 4.2.2), den Fassadebaustein (Abschnitt 4.2.3) und die Hierarchische Komposition (Abschnitt 4.2.4).

4.2.1. Schichtenbaustein

Das Schichtenmuster (engl. layers pattern) [BMR+96, CBB+10], hier als *Schichtenbaustein* adaptiert, ist ein strukturelles Architekturmuster zur logischen Unterteilung eines Softwaresystems. Softwarelemente wie Klassen oder Komponenten werden in logischen Einheiten, den *Schichten*, zusammengefasst. Innerhalb eines Systems stehen die verschiedenen Schichten in einem unidirektionalen Nutzungs- oder Abhängigkeitsverhältnis zueinander.

[1] Architekturbausteine im Sinne dieses Ansatzes beinhalten neben der Beschreibung von Struktur und Verhalten der Lösung auch weitere indirekte Eigenschaften und Bedingungen, die für die Gültigkeit dieser Eigenschaften eingehalten werden müssen (siehe Abschnitt 5.1.1).

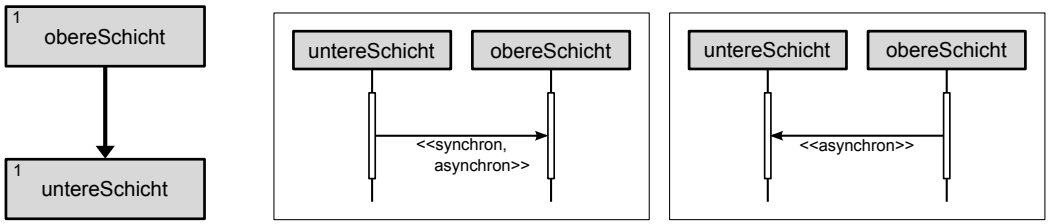


Abbildung 4.1: Schematische Darstellung der Struktur (a) des Schichtenbausteins und des durch erlaubte Kommunikationsarten und -richtungen repräsentierte Verhalten (b).

Verwendung

Ein Ziel der Schichtung ist die Trennung von Bereichen mit unterschiedlichem Abstraktionsniveau und deren hierarchische Anordnung. Die in einer Schicht zusammengefassten Elemente zeichnen sich durch einen ähnlichen Abstraktionsgrad in Bezug auf Implementierungsdetails aus. Eine Schicht bündelt beispielsweise verschiedene Datenbankzugriffe und stellt diese anderen hierarchisch oberhalb angeordneten Schichten zur Verfügung. Die nutzenden Schichten können über definierte Schnittstellen auf die Daten zugreifen, ohne Informationen zu besitzen, wo und wie diese Daten hinterlegt sind.

Struktur

In einem mehrschichtigen System besteht zwischen den einzelnen Schichten basierend auf ihrem Abstraktionsgrad eine hierarchische Ordnung. Dabei wird die abstrakteste Schicht als die oberste und die mit dem niedrigsten Abstraktionsgrad als die unterste Schicht betrachtet. Zugriff darf nur von oben nach unten erfolgen, wobei je nach Ausprägung des Bausteins der Zugriff nur direkt auf die nächsttiefere (*strikte Schichtung*) oder auf alle unterhalb erfolgen darf (*nicht-strikte Schichtung*). Da die einzelnen Schichten mitunter Funktionalitäten für die nächsthöheren bereitstellen, spricht man auch davon, dass die obere von der verwendeten unteren abhängt.

Abhängigkeiten zwischen zwei Schichten können beispielsweise durch die Nutzung von Typen entstehen. Innerhalb einer Schicht definierte Typen können in einer anderen spezialisiert oder generalisiert werden oder sie können zur Typisierung, z.B. von Methodenargumenten oder Rückgabewerten, verwendet werden [Her11]. Die Schicht, die die Typen nutzt, ist somit abhängig von der Typdefinition in der bereitstellenden Schicht.

Abbildung 4.1a veranschaulicht die Struktur des Schichtenmusters: Schichten werden durch Rechtecke und die Nutzungs- oder Abhängigkeitsbeziehung durch einen Pfeil dargestellt. Die logisch unterste Schicht wird dabei auch in der Abbildung unten dargestellt. Wie auch in der Abbildung dargestellt, wird hier angenommen, dass der Schichtenbaustein an sich aus nur zwei Schichten (der oberen und der unteren) besteht. Ein System mit mehreren Schichten wird durch entsprechend häufige Anwendungen des Schichtenbausteins erzeugt.

Verhalten

Der Schichtenbaustein definiert kein spezifisches Verhalten im Sinne eines Protokolls, welches die Interaktionen zwischen den Elementen zweier Schichten beschreibt. Es beschränkt hingegen

die Art und Richtung von Interaktionen und die jeweils interagierenden Schichten[2]. Die Position innerhalb der Schichtenhierarchie gibt zunächst an, welche andere Schicht B eine Schicht A benutzen darf. Durch die Nutzung von B ist A von B abhängig, wenn die Korrektheit von A auf der Korrektheit und Existenz von B beruht [CBB+10]. Nutzen kann eine Schicht eine andere auf verschiedene Weisen. Eine Schicht A nutzt beispielsweise eine Schicht B, wenn A eine Methode von B aufruft. Das korrekte Verhalten von A ist abhängig von dem in B implementierten Verhalten. Allerdings stellt ein asynchroner Aufruf eine Ausnahme dar [Her11]. Im Gegensatz zu einem synchronen Methodenaufruf ist die aufrufende Schicht nicht von der Methodenausführung abhängig und kann somit nicht, z.B. durch eine Endlosschleife der aufgerufenen Schicht, blockiert werden.

Abhängigkeiten im Sinne der oben genannten dürfen im Schichtenbaustein nur in Pfeilrichtung der Nutzungsbeziehung erfolgen. Einzig asynchrone Aufrufe bilden hier eine Ausnahme und sind auch in die entgegengesetzte Richtung erlaubt (siehe Abb. 4.1b). Andernfalls wäre es auch nicht möglich, Informationen wie Benachrichtigungen oder Fehlermeldungen nach oben weiterzuleiten.

Spezifische Eigenschaften

Die Einteilung der Architektur eines Softwaresystems in geeignete Schichten sorgt für eine klare Trennung von Verantwortlichkeiten. Zudem besitzt eine solche Architektur eine klar hierarchisch gegliederte Struktur. Zyklische Abhängigkeiten können hierdurch nicht auftreten, was die Wartung des Systems vereinfacht. Zusätzlich erleichtern die azyklische sowie die modulare Struktur den Test von Schichten und ihrer Elemente.

Nachteilig kann sich die Schichtung hingegen auf die Weiterleitung von Informationen auswirken. Aufrufe müssen unter Umständen über viele Schichten hinweg delegiert werden. Zudem erfordert die Ergänzung eines neuen über viele Schichten verlaufenden Aufrufs, dass das System statt an wenigen an vielen Stellen verändert werden muss.

Beispiele für die Verwendung

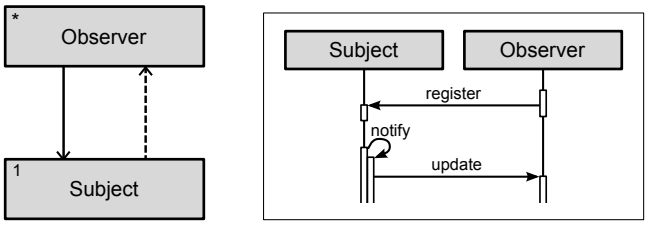
Im Bereich von Softwarearchitekturen ist eine der populärsten Anwendungen des Schichtenmusters die 3-Schichten-Referenzarchitektur für Informationssysteme [Fow02]. Diese unterteilt ein Softwaresystem von unten nach oben in die Schichten Datenhaltung, Anwendungslogik und Präsentation.

In der Informatik insgesamt das wohl bekannteste Beispiel für das Schichtungsprinzip ist das ISO-OSI-7-Schichtenmodell [Tan03] der Kommunikationsprotokolle in Computernetzwerken. Dieses Modell unterteilt die Netzwerkkommunikation beginnend bei der hardwarenahen Bitübertragungsschicht bis zur Anwendungsebene.

4.2.2. Observerbaustein

Das dem *Observerbaustein* zugrundeliegende Oberservermuster [GHJV94] – oder auch als *publisher-subscriber*-Muster [BMR+96] beschrieben – ist ein Verhaltensmuster, das der Benachrichtigung von Objekten über Zustandsveränderungen dient. Ändert sich der Zustand eines Objekts (*Subject*) werden alle daran interessierten Objekte (*Observer*) informiert. Dabei ist das Subject

[2] Präziser formuliert erfolgt die Interaktion zwischen Elementen, die zu den jeweiligen Schichten gehören. Daher wird dies im Folgenden auch angenommen, wenn von einer Interaktion zwischen Schichten gesprochen wird.



(a) (b)

Abbildung 4.2: Schematische Darstellung von Struktur (a) und Verhalten (b) des Observerbausteins.

nur für die Benachrichtigung und eventuelle Weitergabe der veränderten Informationen an die Observer zuständig; das Subject braucht den Aufbau der Observer und die Auswirkungen von Änderungen nicht zu kennen.

Verwendung

Ziel des Musters ist es, den Bereich einer Datenänderung von den Systembereichen zu entkoppeln, auf die Änderungen eine Auswirkung haben. Folgendes Beispiel veranschaulicht die Verwendung des Musters: Die Messdaten einer Wetterstation sollen auf unterschiedlichen Endgeräten automatisch aktualisierend angezeigt werden. Da sich die Anzeigemöglichkeiten von Endgerät zu Endgerät deutlich unterscheiden, existiert für jedes Endgerät eine spezifische Ansicht. Ändern sich nun die Messdaten, müssen diese Information an die entsprechenden Endgeräte verteilt und jede Ansicht angepasst werden.

Eine Möglichkeit besteht darin, diese Aufgabe der Datenverwaltung zu übertragen, die Datenänderungen aktiv an die Ansichten weiterreicht. Dies führt allerdings zu einer engen Kopplung der Ansichten an die Datenverwaltung, erschwert damit die Wiederverwendung und ist unflexibel in Bezug auf Erweiterungen um weitere Ansichten. Eine flexiblere Lösung bietet die Entkopplung von Daten und Ansicht: Die verschiedenen Ansichten werden unabhängig für die unterschiedlichen Endgeräte erstellt und registrieren sich bei der Datenverwaltung als Observer. Für die Datenverwaltung ist die Art des Endgerätes unerheblich. Bei einer Datenänderung werden die registrierten Observer lediglich benachrichtigt und übernehmen das Abrufen der Daten und Aktualisieren der Ansicht selbst.

Struktur

Strukturell besteht der Observerbaustein aus Observer und Subject (siehe Abb. 4.2a), wobei ein Subject von einer beliebigen Anzahl von Observern beobachtet werden kann. Eine Abhängigkeit besteht zwischen Observer und Subject zum einen durch die Registrierung des Observers beim Subject, zum anderen durch die Benachrichtigung bei Datenänderung und Austausch der veränderten Daten. Durch eine klare Definition der Schnittstellen und eines Verhaltensprotokoll wird die Kopplung zwischen beiden reduziert und die Flexibilität für Erweiterungen gewahrt.

Verhalten

Die Kommunikation zwischen Observer und Subject ist klar definiert (siehe Abb. 4.2b). Um über Änderungen informiert zu werden, registriert sich ein Observer zunächst beim Subject

(register). Verändert sich der Zustand des Subjects, wird es darüber informiert (notify) und gibt diese Information an alle registrierten Observer weiter (update).

Die Art der Weitergabe der veränderten Daten von Subject zu Observer legt der Baustein nicht fest. Zum einen können die Daten direkt bei der Benachrichtigung durch update übermittelt werden (*Push*) oder durch den Observer nach der Benachrichtigung abgerufen werden (*Pull*). Ebenfalls lässt der Observerbaustein die Art der Kommunikation offen. So können die Aufrufe sowohl synchron als auch asynchron erfolgen.

Spezifische Eigenschaften

Subject und Observer sind nur lose aneinander gekoppelt. Dadurch können beide unabhängig voneinander modifiziert werden, was die Modularität und damit Erweiterbarkeit und Austauschbarkeit erhöht. Die Unabhängigkeit ist allerdings nicht mehr gewährleistet, wenn sich die Reaktionen eines Observers auf das Subject auswirken. Beispielsweise könnte die Benachrichtigung durch update am Observer eine Aktionsfolge auslösen, die wiederum zu Änderungen am vom Subject verwalteten Daten führt. Daraufhin würde das Subject durch notify wieder informiert werden und den Observer abermals durch update benachrichtigen. Wird ein solcher Kreislauf nicht unterbrochen, entsteht eine Endlosschleife, die das System blockieren kann.

Der Baustein legt nicht fest, ob die Kommunikation synchron oder asynchron erfolgt. In bestimmten Anwendungsfällen kann es allerdings erforderlich sein, dass z.B. der Zustand des Subjects nicht wieder verändert wird, bevor der Observer auf die aktuellen Änderungen reagiert hat. Zudem können hohe Änderungsraten zu einer hohen Kommunikationslast zwischen Subject und Observer führen.

Beispiele für die Verwendung

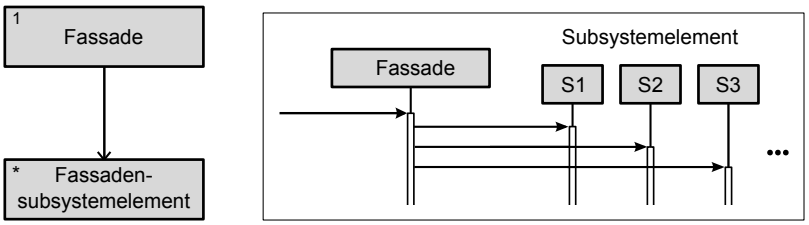
Das Observermuster findet auch in verschiedenen anderen Mustern Verwendung, z.B. im Architekturmuster Model-View-Controller (MVC) [BMR+96]. Dieses Muster unterteilt eine Anwendung in Daten (Model), Ansicht (View) und Steuerungsbereich (Controller). Änderungen tauschen die verschiedenen Bereiche mit Hilfe des Observermusters aus.

4.2.3. Fassadebaustein

Das *Fassademuster* (engl. facade pattern) [GHJV94], hier als *Fassadebaustein* adaptiert, ist ein Strukturmuster. Einem als Subsystem bezeichnetem Geflecht aus Softwareelementen wird ein neues Softwareelement - die *Fassade* - vorgeschaltet. Interaktionen von außen auf das Subsystem erfolgen unter Verwendung der Fassade. Die Fassade interagiert dann mit den für die Interaktion notwendigen Elementen des Subsystems.

Verwendung

Ziel des Fassadebausteins ist es, eine einheitliche und vereinfachte Zugriffsmöglichkeit auf ein Subsystem anzubieten. Eine Fassade fasst z.B. wiederkehrende und komplexe Abläufe, die der Interaktion mit verschiedenen Elementen des Subsystems bedürfen, zusammen. Diese Abläufe müssen so nicht mehr durch verschiedene Elemente implementiert werden, sondern finden sich zentral in der Fassade. Ein Element, das ursprünglich mit einem oder mehreren Elementen des Subsystems kommunizierte, interagiert so nur noch mit der Fassade und ist vom restlichen Subsystem entkoppelt. Betrachtet man einen einzelnen Ablauf im Detail, ist durch die Fassade



(a) (b)

Abbildung 4.3: Schematische Darstellung von Struktur (a) und Verhalten (b) des Fassadebausteins.

ein zusätzlicher Interaktionspartner hinzugekommen. Gleichzeitig hat sich allerdings das Abhängigkeitsgeflecht verschoben. Ohne die Fassade besteht eine m-zu-n-Abhängigkeit zwischen den nutzenden Elementen und den Elementen des Subsystems. Stattdessen hängen nun die nutzenden Elemente nur noch von der Schnittstelle der Fassade ab und zugleich hängt von den Elementen des Subsystems mit der Fassade nur ein einziges Element ab. Insgesamt geht mit der Nutzung einer Fassade daher z.B. eine bessere Wartbarkeit, eine geringere Coderedundanz und eine reduzierte Gefahr von inkonsistenten Implementierungen von an sich identischen Abläufen einher.

Struktur

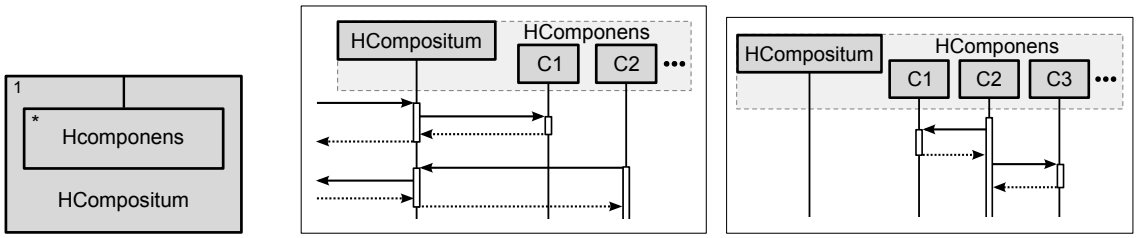
Der Fassadebaustein besteht aus einer *Fassade* und einem Fassadensubsystem (siehe Abb. 4.3a). Ein Subsystem besteht aus einer beliebigen Menge von Softwareelementen (*Fassadensubsystemelemente*) und einem beliebig komplexen Beziehungsgeflecht zwischen diesen Softwareelementen. Die Fassade kapselt das Subsystem und dient Elementen, die das Subsystem nutzen, als Zugriffspunkt. Abhängigkeiten bestehen dadurch nur zwischen Nutzern und Fassade und zwischen der Fassade und dem Subsystem. Über vom Subsystem ausgehende Abhängigkeiten, z.B. in Form von Interaktionen, auf die Umgebung macht dieser Baustein keine Aussagen. Äquivalent zu der Kapselung der eingehenden Interaktionen ist die Einführung einer Fassade für ausgehende Interaktionen denkbar.

Verhalten

Eine Fassade bietet Funktionalität an, die sich aus Funktionen des von ihr gekapselten Subsystems zusammensetzt. Daher führen Aufrufe an der Fassade immer zu Aufrufen durch die Fassade an das Subsystem (siehe Abb. 4.3b). Beschreibt man das Verhalten der Fassade generisch, so sind sowohl die Anzahl der von der Fassade ausgehenden Aufrufe als auch die jeweils beteiligten Subsystemelemente variabel. Lediglich die Richtung und Wege von möglichen Interaktionen und damit verbundener Abhängigkeiten werden durch den Baustein festgelegt (von Fassade auf das Subsystem sowie von Nutzern nur zur Fassade und nicht direkt auf das Subsystem).

Spezifische Eigenschaften

Je nach Variante erlaubt der Fassadebaustein Nutzern weiterhin den direkter Zugriff auf das Subsystem an der Fassade vorbei. Um eine weitestgehende Unabhängigkeit zwischen Nutzern



(a) (b)

Abbildung 4.4: Schematische Darstellung von Struktur (a) und Verhalten (b) der Hierarchischen Komposition.

und Subsystem zu erreichen, kann der Fassade ein exklusiver Zugriff eingeräumt und sämtlicher Fremdzugriff verboten werden.

Beispiele für die Verwendung

Eingesetzt wird das Fassademuster zum Beispiel in der Java API Swing. Die Klasse JOptionPane [Ora13] bietet als Fassade Methoden für die Erstellung verschiedener, aber einfacher Dialogfenster an. Der Nutzer benötigt auf diese Weise keine Kenntnis über die verschiedenen GUI-Klassen (z.B. für Schaltflächen oder Layoutmanager), die zur Erzeugung des Dialogs notwendig sind. Gestaltungsmöglichkeiten bestehen nur durch wenige Parameter, wodurch die Komplexität der Schnittstelle gering gehalten wird.

4.2.4. Baustein Hierarchische Komposition

Die *hierarchische Komposition*, die hier ebenfalls als Architekturbaustein interpretiert wird, ist kein Muster im klassischen Sinn – anders als dies bei den Ausgangsmustern der zuvor beschriebenen Architekturbausteinen der Fall ist. Es handelt sich hierbei vielmehr um ein Konzept zur Wiederverwendung von Softwareelementen. Strukturierende Softwareelemente, wie z.B. Komponenten, werden dabei aus anderen Softwareelementen aufgebaut. Dieses Konzept ist bereits in vielen Komponentenmodellen [MT00, HB07] und Beschreibungssprachen für komponentenbasierte Architekturen [Obj11] verankert.

Verwendung

Der Mechanismus der hierarchischen Komposition erlaubt es, ein neues Softwareelement unter Verwendung bereits existierender Softwareelemente zusammenzusetzen. Ungeachtet seiner Komplexität ist dabei ein solches zusammengesetztes Softwareelement für seine Nutzer nicht als Ansammlung von vielen Softwareelementen sichtbar. Für einen Nutzer bildet es hingegen ausschließlich eine für sich abgeschlossene Einheit. Dieses Prinzip erlaubt somit Wiederverwendung bei gleichzeitiger Reduktion der Komplexität.

Struktur

Im Wesentlichen besteht dieser Baustein aus dem zusammengesetzten Softwareelement *HCompositum* und einer beliebigen Menge von darin enthaltenen Softwareelementen (*HComponens*). Die *HComponens* werden zu einer beliebig komplexen Struktur miteinander kombiniert und

bilden das HCompositum. Die Darstellung in Abbildung 4.4a spiegelt diesen Aspekt der hierarchischen Strukturierung abstrakt wieder.

Verhalten

Das Verhalten des Bausteins der hierarchischen Komposition ist unspezifisch. Ein HComponens kann mit anderen Softwareelementen desselben HCompositums beliebig interagieren. Ebenfalls sind Interaktionen mit Softwareelementen außerhalb des HCompositums möglich, allerdings sind hierbei zwei Varianten zu unterscheiden. Mit einer konkreten hierarchischen Komposition kann gleichzeitig auch das Prinzip des *Information Hidings* [Par72], d.h. der Kapselung der HComponens, umgesetzt werden. Direkte Interaktionen zwischen Elementen außerhalb des HCompositums und einem HComponens sind dann nicht möglich. Interaktionen von außerhalb mit einem HComponens erfolgen über das HCompositum, von wo diese an die inneren HComponens weitergereicht und verarbeitet werden (siehe links in Abb. 4.4b). Antworten oder vom HComponens ausgehende Interaktionen erfolgen über den umgekehrten Weg. Ein HComponens kann nur mit anderen HComponens desselben HCompositums direkt ohne das HCompositum interagieren (siehe rechts in Abb. 4.4b). Ohne Information Hiding kann ein HComponens auch direkt mit Softwareelementen außerhalb des HCompositums interagieren.

Spezifische Eigenschaften

Hierarchische Komposition ohne Information Hiding ist lediglich ein Strukturierungskonzept. In Verbindung mit Information Hiding bildet das HCompositum hingegen eine Kapsel, die den HComponens die in den vorherigen Abschnitten Struktur und Verhalten erläuterten Einschränkungen bzgl. ihrer Interaktionspartner auferlegt.

Beispiele für die Verwendung

Das der UML [Obj11] zugrunde liegende Komponentenmodell setzt hierarchische Komposition mit Information Hiding um. Als StructuredClassifier besitzt eine Komponente (Component) durch Komposition sogenannte Parts vom Metatyp Property. Diese Parts können durch ihre Eigenschaft als TypedElement wiederum mit einem von Type abgeleiteten Metatyp Component getypt sein. Die hierarchische Komposition wird demzufolge durch die Zusammensetzung von komponentengetypten Elementen zu einer Komponente realisiert. Component ist zudem ein EncapsulatedClassifier, welcher Ports besitzt. Ports definieren Interaktionspunkte zwischen einem EncapsulatedClassifier und seiner Umgebung. Diese zusätzlichen Interaktionspunkte verbergen die Interna einer Komponente (Information Hiding).

4.3. Entwicklung des Beispielsystems

In der vorherigen Arbeitsphase wurden die Architekturbausteine für das neue Unternehmenssystem durch den jungen Entwickler ausgewählt, auf deren Basis er nun die Architektur erstellt. In mehreren Entwicklungsschritten wendet er die Bausteine an und nimmt weitere Verfeinerungen an der Architektur vor. Anders als von ihm erwartet, verläuft der Architekturentwurf nicht reibungslos. An mehreren Stellen kommt es zu Konflikten, auf die er dank der Eigenschaftsbedingungen der Bausteine schnell aufmerksam wird.

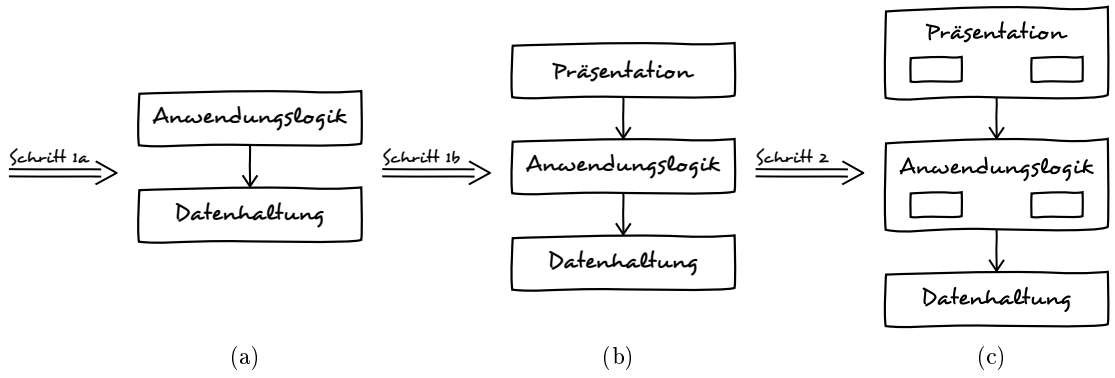


Abbildung 4.5: Schritt 1 und 2 der Entwicklung: Unterteilung des Systems in die drei Schichten Datenhaltung, Anwendungslogik und Präsentation (a, b); zudem werden Präsentation und Anwendungslogik unter Anwendung der hierarchischen Komposition aus verschiedenen Subelementen zusammengesetzt (c).

Entwicklung: Trennung der Systemteile

Zwei Anforderungen an das neue Unternehmenssystem sind die zentrale Speicherung der Daten und die verschiedenen Sichten von unterschiedlichen Endgeräten aus auf das System. Beide Anforderungen legen eine Unterteilung des Systems nahe, bei der sowohl die Daten als auch die Präsentation vom Rest des Systems entkoppelt sind. Die zentrale Datenhaltung bietet mehrere Vorteile. Zum einen liegen die Daten nicht in verschiedenen Anwendungen und damit redundant vor, was zu Dateninkonsistenzen führen kann. Zum anderen werden die Datenbankzugriffe an einer Stelle gebündelt, wodurch der Zugriff für die verschiedenen Anwendungen von den technischen Details abstrahiert angeboten werden kann. Die separate Präsentation entkoppelt die Darstellung von der Anwendungslogik und erlaubt somit, dass trotz der unterschiedlichen Oberflächen für verschiedene Abteilungen oder Endgeräte nur ein Anwendungskern realisiert werden muss.

Der Entwickler entscheidet sich daher zunächst für eine dreischichtige Architektur. Hierfür wendet er in zwei aufeinander folgenden Entwicklungsschritten jeweils den Schichtenbaustein an. In **Entwicklungsschritt 1a** erstellt er zunächst die beiden Schichten, die die Datenhaltung und die Anwendungslogik beinhalten (siehe Abb. 4.5a). Die untere Schicht bildet die Datenhaltung, die die Zugriffe auf die Daten verwaltet und die technischen Details der Kommunikation mit dem eigentlichen Datenbankserver für die restlichen Teile des Unternehmenssystems verbirgt. Die mittlere Schicht enthält die Anwendungslogik und verarbeitet die Daten. In **Entwicklungsschritt 1b** wird die Präsentation hinzugefügt, die verschiedene Benutzerschnittstellen auf das System zur Verfügung stellt. Bei dieser Anwendung des Schichtenbausteins ist die Anwendungslogik die untere Schicht, während die Präsentation die obere Schicht ist (vgl. Abb. 4.5b). Zwischen den Schichten der beiden Bausteinanwendungen besteht jeweils von oben nach unten eine unidirektionale Nutzungsbeziehung. Dabei hängen die Schichten von den jeweils darunter liegenden Schichten und deren bereitgestellter Funktionalität ab.

Jede der drei Schichten soll als Komponente realisiert werden. Um die Komplexität der Präsentation zu verringern, unterteilt der Entwickler diese Komponente durch die Anwendung der hierarchischen Komposition. Jede Benutzerschnittstelle fasst er mit ihrer jeweiligen Funktionalität in einer eigenen Komponente zusammen. Statt die Präsentation durch die neuen Komponenten zu ersetzen, erlaubt die hierarchische Komposition diese in der Präsentation zu verwenden.

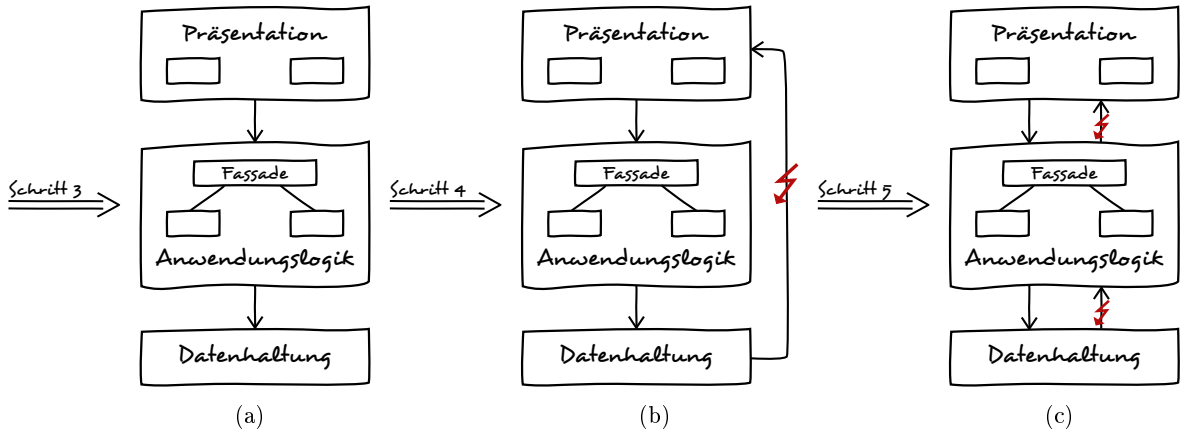


Abbildung 4.6: Schritt 3 bis 5 der Entwicklung: In der Anwendungslogik wird eine Fassade integriert (a). Die direkte Kommunikation von Datenhaltung zu Präsentation führt zu einem Zyklus (b); der Weg über die Anwendungslogik führt ebenfalls zu Zyklen (c).

Ähnlich unterteilt der Entwickler auch die verschiedenen Verarbeitungs- und Verwaltungsprozesse und nutzt die dabei entstandenen Komponenten in der Anwendungslogik (siehe **Entwicklungsschritt 2** in Abb. 4.5c).

Sowohl die abteilungsspezifischen Benutzerschnittstellen auf das System als auch die Oberflächenanwendungen für die unterschiedlichen Endgeräte nutzen Komponenten in der Anwendungslogik. Dabei werden immer wieder identische und teilweise komplexe Interaktionen mit verschiedenen Komponenten der Anwendungslogik durchgeführt. Um die Interaktionen an dieser Stelle zu vereinfachen, führt der Entwickler in **Entwicklungsschritt 3** eine Fassade ein. Zugriffe von der Präsentation auf die Anwendungslogik erfolgen nun nur noch über die Komponente, die die Fassade realisiert, und nicht mehr direkt auf die verschiedenen Komponenten. Zugleich erhöht die Einführung der Fassade die Unabhängigkeit zwischen den beiden betroffenen Schichten und sorgt damit für eine leichtere Wartbarkeit.

Entwicklung: Änderungsaktualisierung der Präsentation

Eine weitere Anforderung sieht vor, dass sich die Anzeigen der verschiedenen Endgeräte bei Datenänderungen aktualisieren sollen. Zu diesem Zweck möchte der Entwickler in der Präsentation eine Funktion realisieren, die aus der Datenhaltung heraus aufgerufen wird. Denn einen direkten Funktionsaufruf hält er für die schnellste und einfachste Möglichkeit eines Datenaustauschs, um eine zeitnahe Darstellung von Datenaktualisierungen zu ermöglichen. In der Architektur drückt er dies durch die Einziehung einer Kommunikationsverbindung von der Datenhaltung zur Präsentation aus (siehe **Entwicklungsschritt 4** in Abb. 4.6b). Bei einer anschließenden Betrachtung seiner Architektur stellt der Entwickler allerdings fest, dass durch den letzten Schritt eine Bedingung des Schichtenbausteins verletzt ist: Durch die Kommunikationsverbindung entsteht ein Zyklus, der alle drei Schichten einbezieht.

In der Hoffnung, der Lösung seines Problems näher zu kommen, studiert unser Entwickler noch einmal die Beschreibung des Schichtenbausteins. Dabei liest er, dass der Schichtenbaustein auch in einer Variante mit einer strikten Hierarchie existiert, die untersagt, dass Schichten bei der Kommunikation übersprungen werden. Daher lässt er die Kommunikation zwischen Datenhaltung und Präsentation durch die Anwendungslogik weiterreichen (siehe **Entwicklungsschritt**

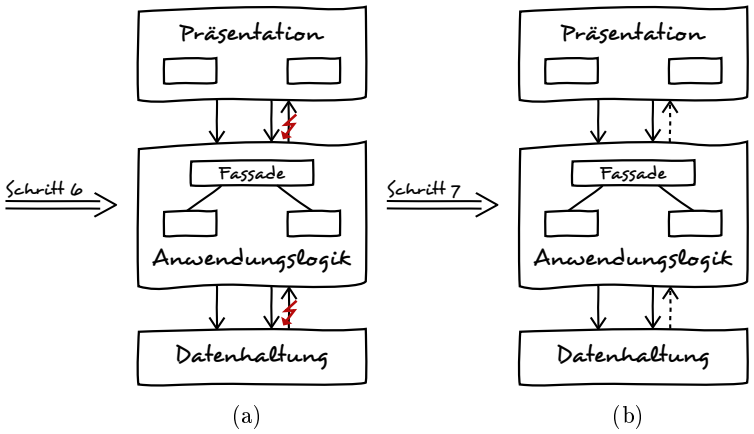


Abbildung 4.7: Schritt 6 und 7 der Entwicklung: Die Kombination von Schichtung und Observer behebt die Zyklen zunächst nicht (a); asynchrone Kommunikation löst die zyklische Abhängigkeit auf (b).

5 in Abb. 4.6c). Hierdurch entfällt zwar der ursprüngliche, große Zyklus, der alle Schichten umfasste, es entstehen so allerdings bidirektionale Abhängigkeiten zwischen zwei jeweils benachbarten Schichten. Diese bidirektionalen Abhängigkeiten erzeugen jedoch wiederum Zyklen. Dieses Mal existieren die Zyklen zwischen jeweils zwei benachbarten Schichten. Sie widersprechen aber ebenfalls der Eigenschaft Zyklenfreiheit des Schichtenbausteins.

In **Entwicklungsschritt 6** integriert der Entwickler sowohl zwischen den Schichten Datenhaltung und Anwendungslogik als auch zwischen den Schichten Anwendungslogik und Präsentation den Observerbaustein. Eine Eigenschaft des Observerbausteins ist die losere Kopplung von Observer und Subject, wodurch der Entwickler hofft, die zyklische Abhängigkeit zwischen den Schichten zu beheben. Im Verhaltensprotokoll des Observerbausteins verläuft die Kommunikation vom Observer zum Subject (Registrierung, registry) als auch vom Subject zum Observer (Aktualisierungsaufforderung, update). Durch diese bidirektionale Kommunikation entsteht im Observerbaustein zwangsläufig ein Zyklus zwischen den Beteiligten. Kombiniert mit dem Schichtenbaustein entsteht dieser Zyklus zwischen zwei Schichten und verletzt somit weiterhin die Bedingung der Zyklenfreiheit (siehe Abb. 4.7a).

Betrachtet man allerdings genauer, wie die Verbindungen in den Bausteinen definiert sind (siehe Abschnitt 4.2), so lässt sich dieser vermeintliche Konflikt auflösen. Laut Abschnitt 4.2.2 schränkt der Observerbaustein nicht ein, ob die Kommunikation zwischen Observer und Subject synchron oder asynchron erfolgt. Asynchrone Kommunikation erzeugt keine Abhängigkeit zwischen den Kommunikationspartner und ist daher im Schichtenbaustein entgegen der Nutzungsbeziehung erlaubt (siehe Abschnitt 4.2.1). Demzufolge führt der Einsatz von asynchroner Kommunikation nicht zu einem Zyklus. Die Zyklen zwischen jeweils zwei Schichten können durch den Entwickler also aufgelöst werden, indem er festlegt, dass die Aktualisierungsaufforderung nur asynchron an den Observer gesendet werden darf (siehe **Entwicklungsschritt 7** in Abb. 4.7b).

Skizze der Systemgrobarchitektur

In mehreren Schritten hat unser Entwickler die Grobarchitektur für das neue Unternehmenssystem erstellt. Die Skizze in Abbildung 4.8 zeigt einen Ausschnitt des Systems, in dem sich

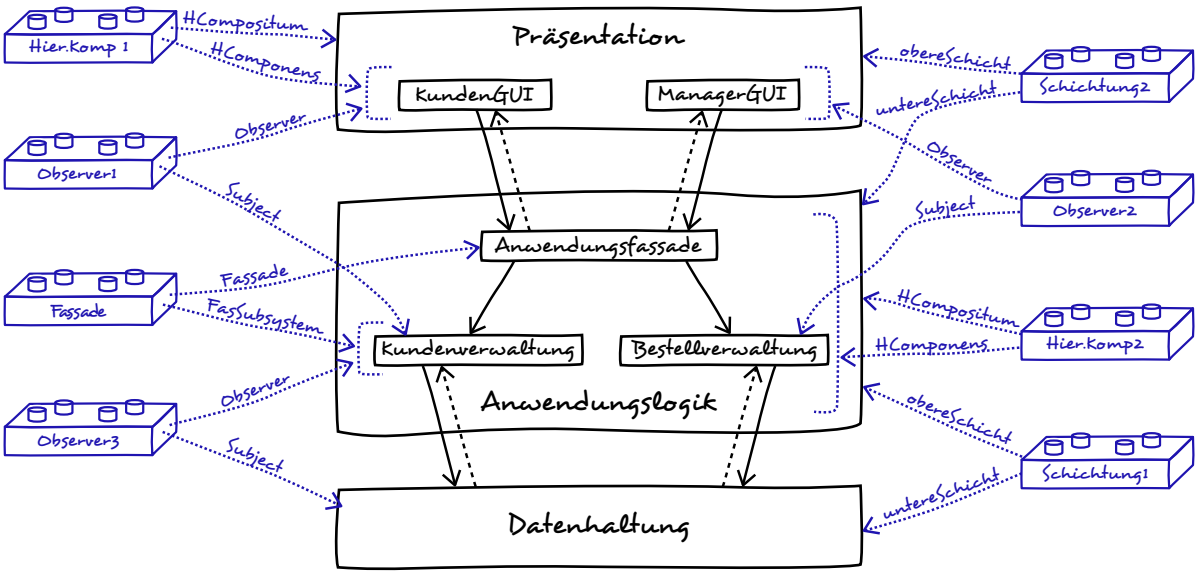


Abbildung 4.8: Skizze der Grobarchitektur des Beispielsystems mit Visualisierung der verwendeten Architekturbausteine und beispielhafte Ergänzung von Systemdetails.

alle in den Entwicklungsschritten angewendeten Bausteine wiederfinden. Allerdings wurde die Darstellung des internen Aufbaus von Präsentation und Anwendungslogik auf nur jeweils zwei Komponenten (in Anwendungslogik zusätzlich zur Fassade) reduziert. Dieser Ausschnitt wird im weiteren Verlauf dieser Arbeit zur Illustration verwendet. Innerhalb der Präsentation werden die Komponenten KundenGUI und ManagerGUI genutzt; in der Anwendungslogik sind die Anwendungsfassade und die zur Datenverwaltung vorgesehenen Komponenten Kundenverwaltung und Bestellungsverwaltung dargestellt.

In dieser Skizze sind ebenfalls die Anwendungen von Schichtenbaustein, Fassadenbaustein, Observerbaustein und der Hierarchischen Komposition mit Verweis auf die beteiligten Elemente der Architektur verdeutlicht. Während in den vorherigen Entwicklungsschritten die Bausteinanwendung nur schematisch erfolgte, sind nun die beteiligten Elemente genauer identifiziert. So tritt bei der Observeranwendung zwischen Präsentation und Anwendungslogik nicht die Präsentation als Observer auf, sondern ihre GUI-Elemente, die durch KundenGUI und ManagerGUI realisiert werden. Ebenso ist das Subject nicht die Anwendungslogik als Ganzes, sondern ihre durch Kundenverwaltung und Bestellungsverwaltung realisierten Verwaltungselemente. Da somit nun zwei Subjects existieren, ergänzt der Entwickler eine weitere Anwendung des Observerbausteins, so dass jede Anwendung auf jeweils eine der beiden Verwaltungselemente als Subject verweist (Observeranwendungen Observer1 und Observer2). Analog wird die Observerrolle bei der Anwendung zwischen Anwendungslogik und Datenhaltung statt durch Anwendungslogik durch Kundenverwaltung und Bestellungsverwaltung ausgefüllt (Observer3). Letztere bilden zudem das Fassadensubsystem der Fassadenanwendung Fassade in der Anwendungslogik.

4.4. Forschungsfragen und Lösungsfelder

In diesem Kapitel wurde an einem Beispiel erläutert, welche Schwierigkeiten bei der Erstellung einer Softwarearchitektur auftreten können. Aufgrund mangelnder Erfahrung sind dem beobachteten Entwickler verschiedene Fehler unterlaufen. So wurde zunächst durch die Verwendung

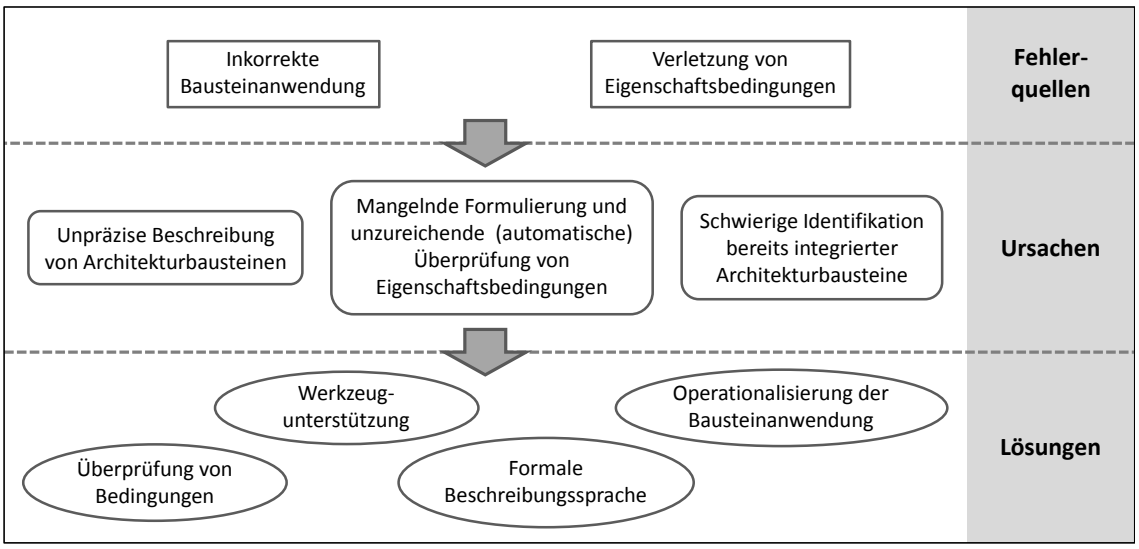


Abbildung 4.9: Übersicht über identifizierte Probleme, deren Ursachen und Lösungen.

einer Aufrufbeziehung zwischen Datenhaltung und Präsentation ein Zyklus unter Beteiligung mehrerer Schichten erzeugt. Aufgrund des hierarchischen Aufbaus des Schichtenbausteins darf allerdings kein Zyklus auftreten. Auch wenn das zuvor betrachtete Beispiel konstruiert ist, so treten die gezeigten Probleme auch in realen Softwaresystemen auf [DDHR09, DH09]. Während das Beispiel klein und überschaubar ist, sind viele reale Softwaresysteme deutlich größer und komplexer, wodurch auch die Häufigkeit für Fehler steigen kann. Kommen zudem mehr Architekturbausteine zum Einsatz, nehmen gleichzeitig die Anzahl der einzuhaltenden Bedingungen und damit mögliche Fehlerquellen zu.

Die begangenen Fehler führten in dem Beispiel zu Verletzungen von Bedingungen, die die Eigenschaften der verwendeten Architekturbausteine sicherstellen sollen. Zusätzlich zur fehlerhaften Kombination von Bausteinen miteinander oder mit ergänzenden Architekturelementen wie Aufrufbeziehungen ist eine inkorrekte Verwendung von Bausteinen denkbar. Denn die Frage, wie ein Baustein korrekt angewendet wird, ist aus der Beschreibung in der Literatur nicht immer eindeutig zu beantworten. Eine falsche Bausteinanwendung kann ebenfalls zu Verletzungen der Bedingungen eines Bausteins führen. Neben persönlichen Faktoren wie Unerfahrenheit lassen sich diese Fehler vor allem auf folgende Ursachen zurückführen (siehe auch Abb. 4.9):

- *Unpräzise Beschreibung von Architekturbausteinen und ihrer Eigenschaftsbedingungen*

  Die Architekturbausteine und vor allem die Bedingungen für ihre Eigenschaften sind nicht ausreichend klar und formal beschrieben. Hierdurch besteht nicht nur die Gefahr einer Fehlinterpretation, sondern auch eine Unterstützung durch softwarebasierte Werkzeuge wird erschwert.

- *Schwierige Identifikation bereits integrierter Architekturbausteine*

  Bereits in der Architektur verwendete Bausteine können mangels einer entsprechenden Notation nicht ohne weiteres identifiziert werden. Dadurch kann nur schwer erkannt werden, an welcher Stelle der aktuell betrachtete Architekturbaustein Konflikte mit bereits integrierten Bausteinen hervorruft.

- *Mangelnde und unzureichende (automatische) Überprüfung von Eigenschaftsbedingungen*

Bei der Integration eines Architekturbausteins oder eines ergänzenden Architekturelements werden die Eigenschaftsbedingungen aller verwendeter Bausteine nicht getestet. Konflikte werden nicht erkannt und führen zu Fehlern in der fertigen Software oder müssen später kostenintensiv behoben werden.

Alle oben genannten Punkte begünstigen Inkonsistenzen in der zu erstellenden Architektur. Ein Verfahren, das den Entwurf einer Architektur basierend auf Architekturbausteinen vollzieht, sollte daher alle diese Punkte adressieren, um das Risiko von Fehlern zu verringern. Daher steht die Lösung dieser Punkte im Zentrum dieser Arbeit. Zusammengefasst werden sie durch die folgenden drei Forschungsfragen:

Forschungsfrage 1:

Wie können Architekturbausteine präzise beschrieben werden?

Forschungsfrage 2:

Wie lassen sich in einem Architekturmodell angewendete Architekturbausteine inklusive ihrer Bestandteile eindeutig identifizieren?

Forschungsfrage 3:

Wie können die Eigenschaftsbedingungen eines Architekturbausteins möglichst universell formuliert und ihre Einhaltung in einem Architekturmodell überprüft werden?

Um die Forschungsfragen in der vorliegenden Arbeit zu beantworten, werden verschiedene Maßnahmen verfolgt. Im Einzelnen sind dies (siehe auch Abb. 4.9):

- *Formale Beschreibung von Architekturbausteinen*

  Eine formale Beschreibungssprache präzisiert die Beschreibung eines Architekturbausteins und lässt wenig Raum für Interpretationen durch den Anwender des Bausteins. Zudem ermöglicht eine formale Beschreibung die Realisierung einer Werkzeugunterstützung. Die Beschreibungssprache muss dabei mächtig genug sein, um alle architekturrelevanten Bestandteile und Aspekte eines Bausteins zu spezifizieren. Außerdem muss sie Beschreibungselemente bereithalten, um die Bestandteile einer aus Bausteinen erstellten Architektur eindeutig der zugehörigen Bausteinanwendung zuzuordnen.

- *Sicherstellung der konsistenten Bausteinanwendung durch Überprüfung der Eigenschaftsbedingungen*

  Die Anwendung jedes Architekturbausteins muss gewissen Konsistenzbedingungen genügen. Darüber hinaus stellen viele Bausteine weitere Bedingungen an eine korrekte Anwendung (Eigenschaftsbedingungen). Die Konsistenzprüfung wertet mit einem dafür geeigneten Mechanismus die Gültigkeit dieser Bedingungen aus und lokalisiert existierende Inkonsistenzen.

- *Operationalisierung der Bausteinanwendung*

  Eine Operationalisierung definiert formal und eindeutig, wie ein Baustein und seine Bestandteile in einer Architektur integriert werden. Darüber hinaus ermöglicht sie die Automatisierung der Architekturerstellung basierend auf Architekturbausteinen.

- *Werkzeugunterstützung*

  Die Verwendung eines Softwarewerkzeuges kann einen Entwickler oder Softwarearchitekten vor allem bei der Überprüfung der Eigenschaftsbedingungen unterstützen. Bei einem auftretenden Konflikt kann dieser sofort seine vorherigen Änderungen korrigieren und damit Verletzungen von Eigenschaftsbedingungen vermeiden. Darüber hinaus kann ein solches Werkzeug dafür Sorge tragen, dass der Architekturbaustein auch wie vorgesehen angewendet wird. Auch kann damit recht einfach die gesamte Historie der bisherigen Architekturentwicklung dokumentiert werden.

5. Konzept des bausteinbasierten Architekturentwurfs

Inhalt

Ziel des bausteinbasierten Architekturentwurfs (BBAE) ist es, die Erstellung der Architektur eines Softwaresystems durch die Kombination von bewährten Konzepten und Mustern, den Architekturbausteinen, zu realisieren. Das Beispiel im vorherigen Kapitel hat gezeigt, dass es trotz bewährter Lösungen zu unerwünschten Konstellationen oder Inkonsistenzen kommen kann. Im Beispiel hat sich auch herausgestellt, dass neben der klaren Definition von Struktur und Verhalten oft weitere Bedingungen an die Anwendung eines Architekturbausteins gestellt werden. Diese Bedingungen werden teilweise nur informell beschrieben und können bei der Verwendung des Bausteins leicht übersehen werden. Durch eine gründliche manuelle Prüfung wurden im Beispiel Fehler aufgedeckt, die auf die Nichtbeachtung dieser Bedingungen zurückgehen. Für ein relativ kleines Beispiel, das aus einer überschaubaren Menge von oft verwendeten Architekturbausteinen basiert, ist eine Überprüfung noch verhältnismäßig einfach zu vollziehen. Mit zunehmender Größe und Komplexität ist allerdings eine explizite Definition dieser Bedingungen und ein strukturiertes Vorgehen für die Überprüfung notwendig.

In diesem Kapitel wird skizziert, wie die vorliegende Arbeit diese Herausforderungen adressiert. Die grundlegenden Konzepte des Ansatzes, u.a. der Begriff des Architekturbausteins oder der Ablauf der Kombination von Architekturbausteinen, werden in Abschnitt 5.1 erläutert. Abschnitt 5.2 betrachtet hingegen den Lebenszyklus von Architekturbausteinen und die Anwendung von Bausteinen im Kontext eines Architekturentwurfsprozesses.

5.1. Das Konzept im Überblick

Die zentralen Artefakte des in dieser Arbeit entwickelten Ansatzes sind die Architekturbausteine (Abschnitt 5.1.1). Neben Struktur und Verhalten definieren sie Bedingungen, die die Einhaltung wesentlicher Eigenschaften sicherstellen. Diese Bedingungen können dabei nicht durch die Struktur- und Verhaltensbeschreibung alleine ausgedrückt werden, sondern erfordern ein erweitertes Beschreibungskonzept. Die Architektur eines Softwaresystems wird durch die Anwendung verschiedener Architekturbausteine und deren Kombination konstruiert. Dabei kann die Anwendung eines Architekturbausteins sowohl bestehende Elemente der Architektur miteinbeziehen als auch die Erzeugung neuer Architekturelemente bewirken (Abschnitt 5.1.2).

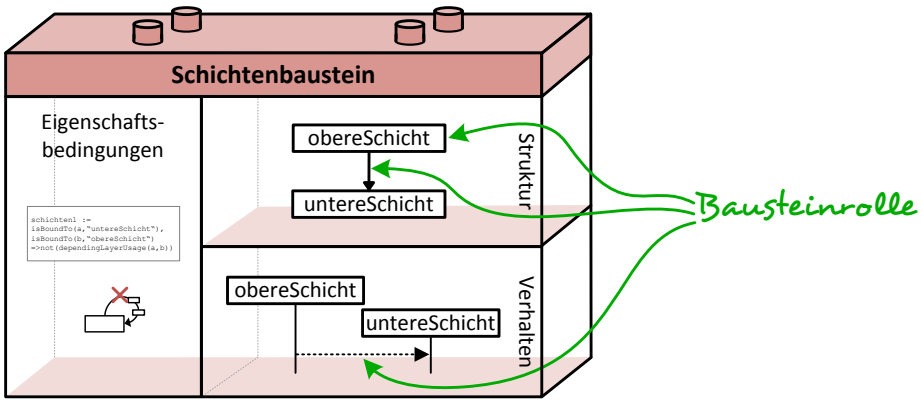


Abbildung 5.1: Skizzierte Beschreibung eines Architekturbausteins am Beispiel des Schichtenbausteins.

Jede Erweiterung muss den Bedingungen der bisher kombinierten Architekturbausteinen genügen. Hierzu werden die Bedingungen aller bisher angewendeter Bausteine in einem iterativen Entwicklungsprozess überprüft (Abschnitt 5.1.3).

5.1.1. Das Prinzip des Architekturbausteins

Softwarearchitekten greifen bei ihrer Arbeit oft auf bewährte Lösungen zurück [Som10, Fow02, BMR+96, GKRS12, Bal11, RH06]. Zu solchen Lösungen zählen Muster, Referenzarchitekturen und Architekturprinzipien (siehe auch Abschnitt 2.2), die bei Bedarf an entsprechenden Stellen in der Architektur eingesetzt werden. Die verschiedenen Muster und Prinzipien haben sich über viele Jahre hinweg aus der Erfahrung von Softwareentwicklern und -architekten herausgebildet. Dabei wurden auch ihre Vor- und Nachteile deutlich und damit auch, wie gut sie immer wiederkehrende Anforderungen erfüllen oder auch, welchen sie widersprechen. In dem in dieser Arbeit vorgestellten Ansatz werden diese Lösungen als *Architekturbausteine* bezeichnet.

Ein Architekturbaustein macht Vorgaben über die Struktur und das Verhalten eines Teils einer Architektur. Er umfasst eine generalisierte Beschreibung von Struktur und Verhalten, die erst durch die Anwendung des Architekturbausteins an den Kontext der Architektur angepasst wird. Wesentliche Bestandteile dieser Beschreibungen sind verschiedene strukturelle und verhaltensbasierte Beschreibungselemente, die als *Rollen* gekennzeichnet sind. In der Strukturbeschreibung werden die strukturellen Rollen und ihre Beziehungen zueinander festgelegt. Beispielsweise enthält das Schichtenmuster (siehe Abschnitt 4.2.1) die beiden Rollen obereSchicht und untereSchicht und eine Nutzungsbeziehung zwischen diesen Rollen, die das Auftreten von Abhängigkeiten auf von der oberen zur unteren Schicht beschränkt (vgl. Abb. 5.1). Ähnlich werden in der Verhaltensbeschreibung die Interaktionen zwischen den Rollen und die Abfolge der Interaktionen festgelegt. Dabei werden die Interaktionen selber auch als Rollen betrachtet. Bei der Anwendung eines Bausteins werden seine Rollen von Architekturelementen, wie Komponenten oder auch Nachrichten, übernommen.

Ein Architekturbaustein umfasst nicht nur die schablonenartige Beschreibung von Struktur und Verhalten. Weitere wesentliche Bestandteile eines Architekturbausteins sind die Bedingungen, die für die Wahrung von speziellen Eigenschaften verantwortlich sind (siehe Abb. 5.1). Diese *Eigenschaftsbedingungen* werden in dieser Arbeit im Gegensatz zu gängigen Musterbeschreibungen explizit und formal formuliert. Ein Beispiel für eine solche Eigenschaft ist die im

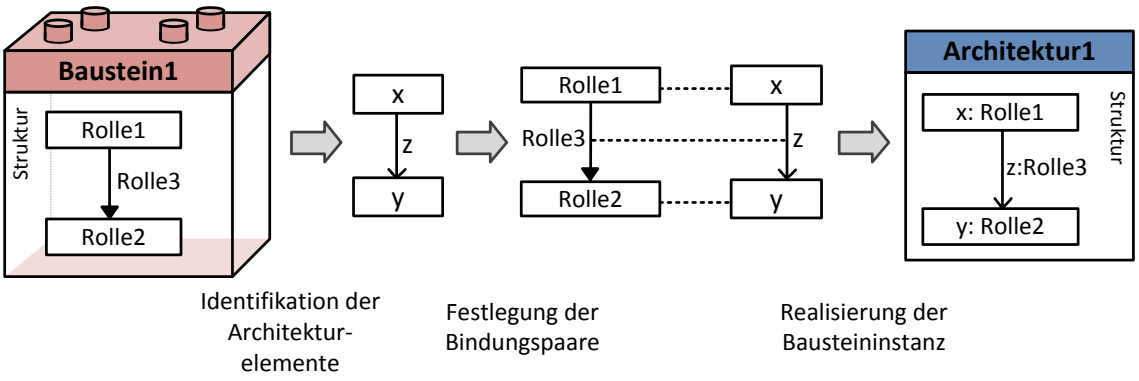


Abbildung 5.2: Ablauf einer Bausteininstanziierung.

Schichtenbaustein geforderte Zyklenfreiheit (vgl. Abschnitt 4.2.1). Im Fall des Schichtenbausteins bezieht sich ein Zyklus auf Nutzungsbeziehungen zwischen den Schichten des Bausteins. Für eine gültige Anwendung dürfen diese Beziehungen nur in bestimmte Richtungen zwischen den Schichten erfolgen und sie dürfen u.a. keinen Zyklus bilden. Zur Gewährleistung der Zyklenfreiheit werden daher in einer Bedingung die Art und Richtung von Nutzungsbeziehungen zwischen den einzelnen Schichten beschränkt. Neben explizit beschriebener Gestalt und Verhalten eines Bausteins sind oftmals seine Eigenschaften für die Auswahl eines auf ein Problem oder eine Anforderung passenden Bausteins maßgeblich. Die Einhaltung der Eigenschaftsbedingungen ist daher nicht nur aus Konsistenzgründen wichtig. Auch für die Erfüllung von Anforderungen, z.B. nichtfunktionalen Anforderungen hinsichtlich Modularität oder Wartbarkeit eines Systems, ist die Einhaltung der Eigenschaftsbedingungen relevant.

5.1.2. Softwarearchitektur aus Bausteinen

Beim bausteinbasierten Architekturentwurf eines komponentenbasierten Systems werden nicht nur einzelne Elemente wie Komponenten oder Schnittstellen betrachtet und zusammengefügt. Zusätzlich werden mit Architekturbausteinen Einheiten angewendet, die als eine zusammenhängende und möglicherweise komplexe Menge von Architekturelementen betrachtet werden können. Im Rahmen einer solchen als *Bausteininstanziierung* bezeichneten Anwendung werden die Rollen eines Bausteins an korrespondierende Architekturelemente gebunden. Wie in Abbildung 5.2 illustriert, erfolgt eine Bausteininstanziierung in mehreren Schritten. Die Bausteinelemente Rolle1, Rolle2 und Rolle3 sollen an die Architekturelemente x, y bzw. z gebunden werden. Diese Zuordnung von einzelnen Architekturelementen und Bausteinrollen, die aneinander gebunden werden sollen, wird durch *Bindungspaare* repräsentiert. Bevor die Bindung erfolgt, werden die Architekturelemente identifiziert und, sofern sie bisher nicht existieren, erstellt. Schließlich werden die neuen Elemente in die Architekturbeschreibung integriert und die Anwendung des Architekturbausteins damit vollzogen. Die Informationen über die Bausteininstanz und ihre enthaltenen Bindungspaare werden im Rahmen der Architekturbeschreibung dokumentiert.

Die Anzahl der Rollen, die an ein Architekturelement gebunden werden können, ist nicht beschränkt. Es kann sowohl an mehreren Bindungen derselben Bausteininstanz als auch an verschiedenen Bausteininstanzen von einem oder unterschiedlichen Architekturbausteine beteiligt sein. Durch ein Architekturelement mit Bindungen in unterschiedlichen Bausteininstanzen werden die beteiligten Bausteininstanzen kombiniert. Dieser Vorgang der *Instanzkomposition*

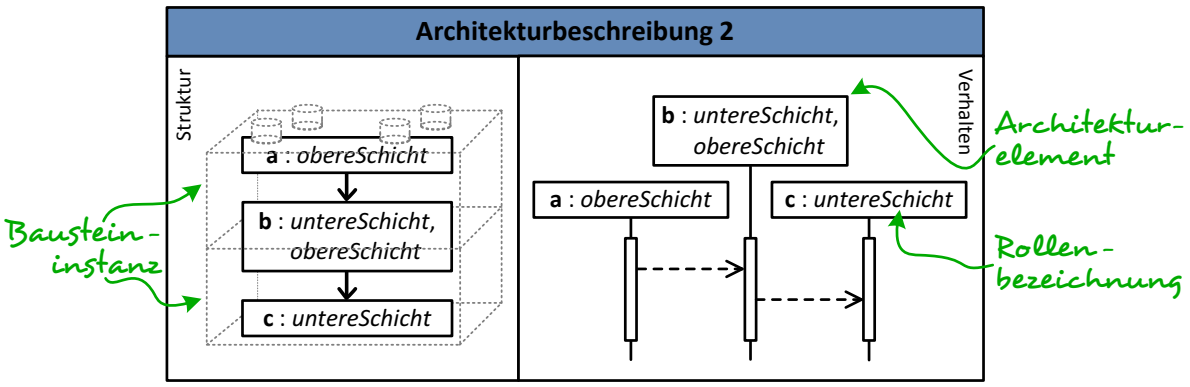


Abbildung 5.3: Skizzierte Beschreibung einer aus Bausteinen komponierten Architektur.

kann in Abbildung 5.3 nachvollzogen werden. Die dort gezeigte Architektur wurde aus zwei Instanzen des Schichtenbausteins erstellt. Eine Instanz betrifft die beiden Architekturelemente a und b, an die die Rollen obereSchicht bzw. untereSchicht gebunden wurden. In einer anderen Bausteininstanz wurden diese beiden Rollen an b bzw. c gebunden. Architekturelement b ist an beiden Bausteininstanzen beteiligt und ermöglicht so die Komposition der beiden Instanzen.

5.1.3. Überprüfung von Eigenschaftsbedingungen im Architekturentwurf

Ein möglicher Ausgangspunkt der Architekturerstellung ist eine einzelne, initiale Bausteininstanz. Darauf aufbauend wird die Architektur schrittweise durch weitere Bausteininstanzen mittels der in Abschnitt 5.1.2 erläuterten Instanzkomposition erweitert. Abbildung 5.4 stellt diese schrittweise Architekturerstellung exemplarisch unter Verwendung des Schichtenbausteins und des Observerbausteins dar. In den ersten beiden Schritten wurde zunächst jedes Mal der Schichtenbaustein instanziiert. Daraus resultieren Architekturbeschreibung1 und im nächsten Schritt Architekturbeschreibung2 (siehe Architekturbeschreibung2 auch in Abbildung 5.3). Beide Architekturen sind hinsichtlich der gewünschten Zyklenfreiheit des Schichtenbausteins (notiert durch ✔) korrekt. Die Architekturbeschreibung 3a entsteht durch eine dritte Instanziierung des Schichtenbausteins, in der c an obereSchicht und a an untereSchicht gebunden wird. Bereits hier, zwei Schritte nach der initialen Architektur, entsteht ein Zyklus, der alle drei Architekturelemente a, b und c einbezieht. Dieser Zyklus verletzt die Eigenschaftsbedingung des Schichtenbausteins (dargestellt durch ✘).

Wird dieser Fehler nicht behoben, enthält jede auf Architekturbeschreibung 3a aufbauende Architektur diesen Zyklus und ist bezüglich der Eigenschaftsbedingung ungültig. Erfolgt erst am Ende der Architekturerstellung eine Überprüfung, können frühe Fehler oder Konflikte möglicherweise die Revidierung der gesamten Architektur bedingen. Je früher Fehler und Konflikte in der Entwicklung aufgedeckt werden, desto geringer ist der Aufwand, um sie zu beseitigen. Es ist daher nicht ausreichend, die Gültigkeit aller Eigenschaftsbedingungen erst am Ende des Architekturentwurfs zu überprüfen. Deshalb werden, wie auch in Abbildung 5.4 gezeigt, nach jedem Entwicklungsschritt die Eigenschaftsbedingungen aller Architekturbausteine überprüft, die in der resultierenden Architektur verwendet wurden. Treten Verletzungen auf, kann der Softwarearchitekt an dieser Stelle über weitere Maßnahmen entscheiden. So kann er den zum Konflikt führenden Schritt zurücknehmen und mit einer anderen Bausteininstanziierung fortfahren (siehe Architekturbeschreibung 3b durch Anwendung des Observerbausteins in Abb. 5.4).

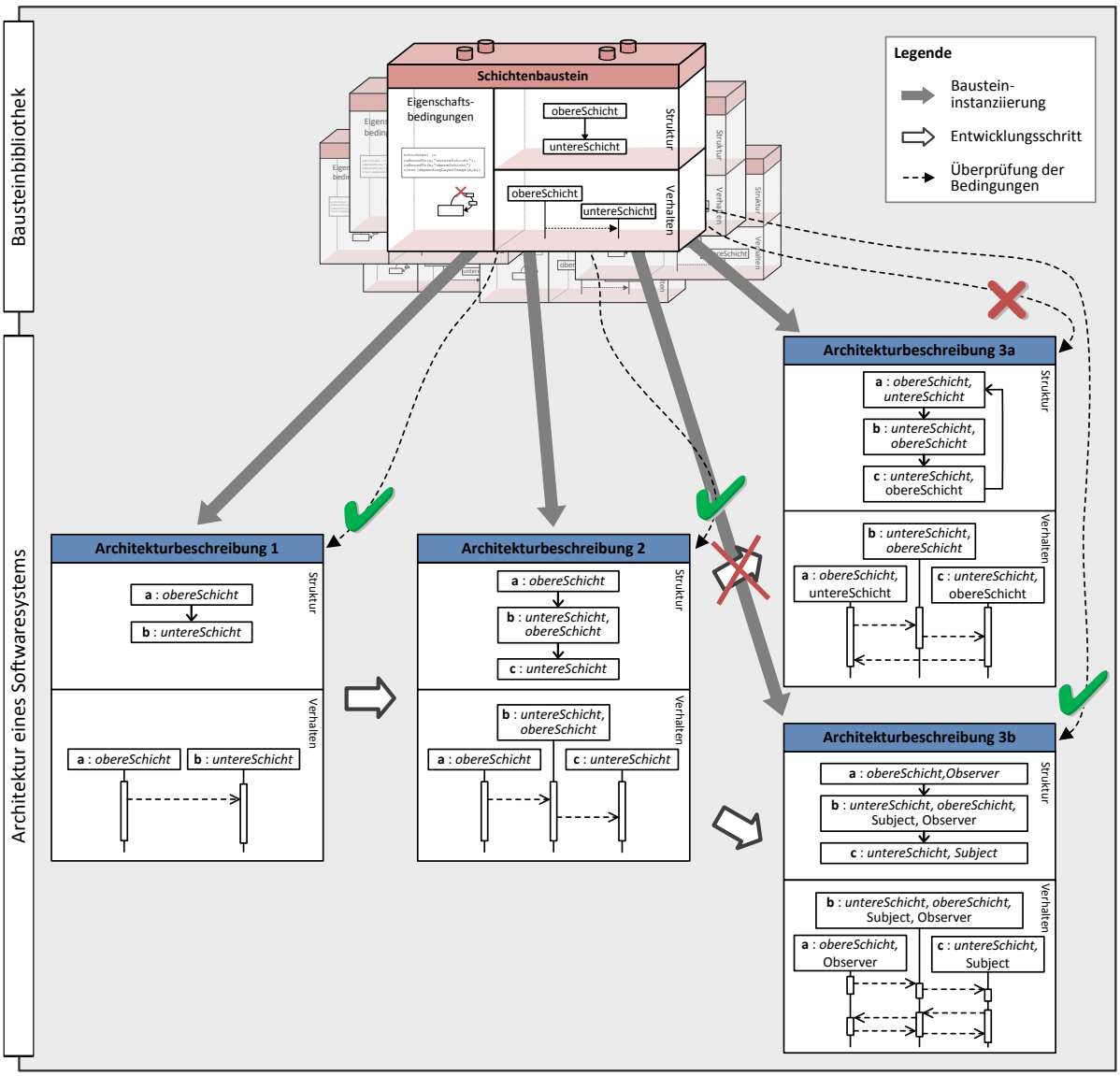


Abbildung 5.4: Schrittweise Instanziierung und Komposition von Architekturbausteinen mit Überprüfung der Eigenschaftsbedingungen.

Er kann eine Verletzung aber auch zunächst ignorieren, da sie beispielsweise durch weitere Entwicklungsschritte wieder eliminiert wird. Je nach Situation sind die Integration weiterer Bausteininstanzen oder die Verfeinerung von Teilen der Architektur mögliche konfliktlösende Erweiterungen. So führte z.B. die Festlegung auf ausschließlich asynchrone Interaktion zwischen bestimmten Architekturelementen in Schritt 7 des Beispiels aus Abschnitt 4.3 zur Behebung des zuvor aufgetretenen Konflikts.

Die Eigenschaftsbedingungen werden im Rahmen der Bausteinbeschreibung auf Basis der Elemente formuliert, die die Bausteinbeschreibungssprache zur Verfügung stellt. Ihre Gültigkeit wird hingegen auf den Architekturmodellen und damit basierend auf den Sprachelementen der Architektur überprüft. Für die Überprüfung der Eigenschaftsbedingungen muss die Distanz zwischen den Modellen überwunden werden. Dies geschieht durch die Bindungen, die im

Rahmen der Bausteininstanziierung erzeugt wurden. Sie stellen den Zusammenhang zwischen den Bausteinelementen auf der einen Seite und den Architekturelementen auf der anderen Seite her. Durch die Bindungen können die für die Überprüfungen notwendigen Kontexte in der Architektur ermittelt werden. Im Schichtenbaustein wird die Eigenschaftsbedingung zur Zyklenfreiheit basierend auf den Rollen obereSchicht und untereSchicht formuliert. Im Beispiel aus Abbildung 5.4 wird dann überprüft, ob es zwischen a in der Rolle obereSchicht und b in der Rolle untereSchicht einen Zyklus gibt. Über die Bindungen ist zudem die Identifikation der Bausteininstanzen möglich, die an einer Verletzung einer Eigenschaftsbedingung beteiligt sind. Diese Information kann bei der Behebung der Verletzung hilfreich sein.

5.2. Einordnung in einen Architekturentwurfsprozess

Betrachtet man den üblichen Prozess der Softwareentwicklung, so steht am Anfang eines jeden Softwareentwicklungsprojekts die Identifikation der verschiedenen Anforderungen an das spätere Softwaresystem [Som10]. Neben funktionalen Aspekten haben vor allem die erhobenen nichtfunktionalen Anforderungen Einfluss auf die Architektur des Systems [Som10]. Unternehmen oder Organisationen, die regelmäßig Softwareentwicklungsprojekte durchführen, definieren oft Standards in Form einer Rahmenarchitektur oder anderweitiger architektureller Vorgaben, die u.a. die Umsetzung verschiedener Anforderungen unterstützen [Fow02, BCK03]. Diese Standards sind globales Wissen, dass jeweils lokal in den einzelnen Projekten genutzt wird. Der Wissenstransfer erfolgt dabei oft nicht nur einseitig in die Projekte. Erfahrungen und neue Erkenntnisse, die in einem Projekt gewonnen werden, können in das globale Wissen einfließen, um dieses zu verbessern oder zu ergänzen.

Das Konzept eines Architekturbausteins und sein Lebenszyklus integrieren sich in einen solchen Entwicklungsprozess. Abbildung 5.5 skizziert einen Entwicklungsprozess, in dessen Mittelpunkt die Architekturbausteine stehen. Diese stellen dabei den zentralen Bestandteil des globalen Wissens dar und werden formalisiert in einer Bibliothek zusammengefasst. Anforderungen, die an ein Softwaresystem gestellt werden, werden von den verschiedenen Architekturbausteinen in unterschiedlichem Umfang unterstützt. Daher wählt der Architekt die Bausteine für das in seinem Projekt zu entwerfende System aus der globalen Bibliothek in Einklang mit den Anforderungen aus (Abb. 5.5, Schritt 2). Die gewählten Architekturbausteine werden nun schrittweise mit der im Abschnitt 5.1.2 beschriebenen Instanzkomposition zusammengefügt (Abb. 5.5, Schritt 3). Dabei werden nach jedem Schritt die Eigenschaftsbedingungen aller Bausteine hinsichtlich ihrer Gültigkeit geprüft (Abb. 5.5, Schritt 4). Bei erfolgreicher Überprüfung fährt der Architekt mit der Komposition weiterer Bausteine fort. Hingegen wählt er bei einem negativen Prüfungsergebnis gegebenenfalls alternative Architekturbausteine aus der Datenbank aus (Abb. 5.5, Schritt 5).

Verantwortlich für die Erstellung und Pflege der Bausteinbibliothek ist ein erfahrener Architekt in der Rolle des Bausteiningenieurs. Seine Aufgabe besteht zum einen darin, die in der Literatur häufig nur informell dokumentierten Bausteine präzise und formal zu beschreiben (Abb. 5.5, Schritt 1). Eine präzise Beschreibung ermöglicht es, die Bausteine ohne Missverständnisse im Architekturentwurf von verschiedenen Projekten einzusetzen. Besonderheiten oder an die Verwendung geknüpfte Bedingungen finden sich zudem nur teilweise und dann oft nur informell in der Literatur und beruhen daher stärker auf den eigenen Erfahrungen des Bausteiningenieurs. Neben der Erstellung von formalen Bausteinbeschreibungen untersucht der Bausteiningenieur Architekturen, die mit dem bausteinbasierten Ansatz entwickelt worden sind.

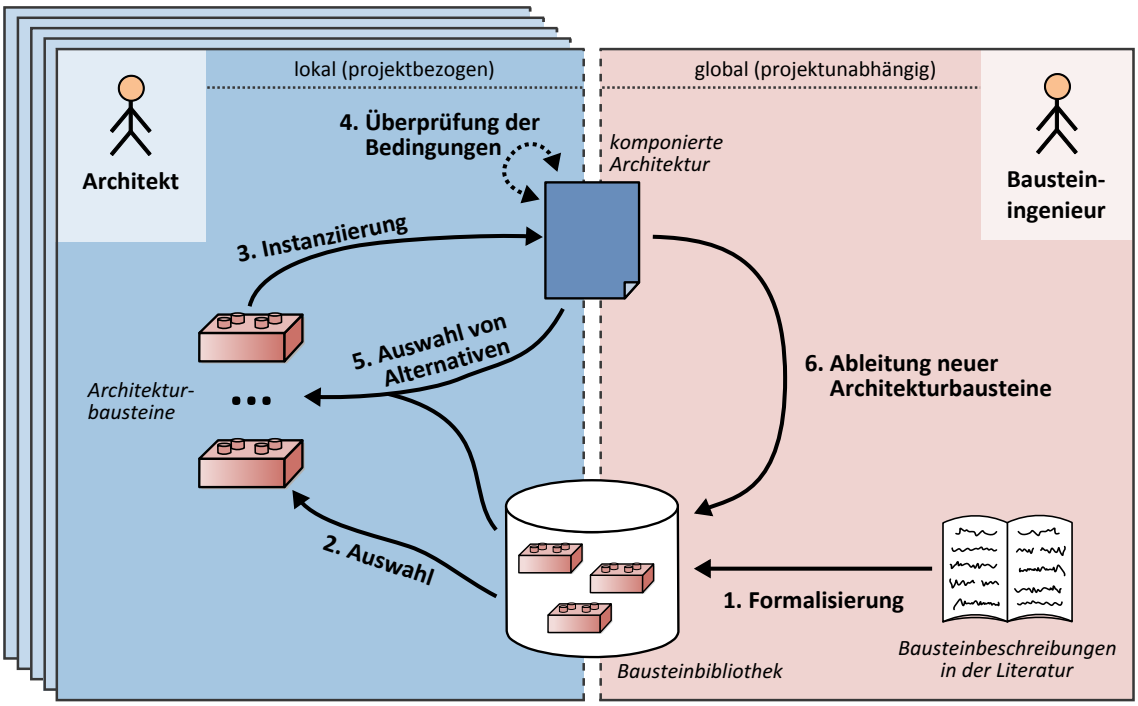


Abbildung 5.5: Architekturbausteine im Zentrum eines Architekturentwurfsprozesses.

Möglicherweise können diese komplette Architektur oder zusammenhängende Teile davon als neue Bausteine in die Datenbank aufgenommen werden (Abb. 5.5, Schritt 6). Beispielsweise ist das Ergebnis des Beispielszenarios aus Abschnitt 4.3 – die dreischichtige Architektur kombiniert mit zwei Observermustern – ein Kandidat für die Aufnahme als neuer Architekturbaustein in die Bibliothek.

Der beschriebene Ablauf illustriert eine mögliche Integration der in dieser Arbeit entwickelten Konzepte in einen Architekturentwurfsprozess. Die Definition eines konkreten Prozesses ist jedoch nicht Ziel dieser Arbeit. Vielmehr stehen die Architekturbausteine, ihre Beschreibung und die Frage, wie aus Architekturbausteinen eine Architektur erstellt werden kann, im Mittelpunkt der Arbeit und der folgenden Kapitel.

6. Beschreibung von bausteinbasierten Architekturen

Inhalt

Sprache ist die Grundlage für die Darstellung, Weitergabe und Verarbeitung von Informationen und Sachverhalten. Um die Wahrscheinlichkeit für Missverständnisse möglichst gering zu halten, ist eine eindeutige und wiederholbare Interpretation der durch die Sprache ausgedrückten Informationen wesentlich. Vor allem für die automatische Verarbeitung ist eine eindeutige Interpretation von besonderer Bedeutung. Erreicht wird dies durch eine klare Definition von Syntax und Semantik der Sprache. Wichtig für eine Sprache ist außerdem ihre Ausdrucksmächtigkeit. So sollte sie die notwendigen Sprachelemente aufweisen, um alle ihrem Einsatzzweck entsprechenden Informationen darstellen zu können.

In der Softwareentwicklung werden Informationen vielfach in Form von Modellen wiedergegeben.Die Grundlage der jeweils verwendeten Modellierungssprache bildet in vielen Fällen ein semiformales Metamodell [RH06, KWB05]. Ein solches Metamodell definiert die zur Verfügung stehenden Sprachelemente und oftmals semantische Teilaspekte der Sprache. Dazu wird es gegebenenfalls durch formal oder informell formulierte Bedingungen und Erläuterungen ergänzt.

Den im bausteinbasierten Architekturentwurf zum Einsatz kommenden Beschreibungen liegt ebenfalls ein Metamodell zugrunde. Im folgenden Kapitel werden die verschiedenen Bereiche des Metamodells vorgestellt. Diese werden durch Diagramme dargestellt, die sich Teilen der grafischen Notation von UML-Klassendiagrammen bedienen. Ergänzende Informationen und Einschränkungen werden informell erläutert und durch prädikatenlogische Ausdrücke formalisiert. Im Hinblick auf eine Automatisierung des bausteinbasierten Entwurfs und der Überprüfung werden die Eigenschaftsbedingungen der Bausteine ebenfalls durch prädikatenlogische Ausdrücke definiert. Für die Verwendung in prädikatenlogischen Ausdrücken werden Klassen des Metamodells durch identisch benannte unäre Prädikate abgebildet, während Beziehungen zwischen Klassen durch mehrstellige Prädikate ausgedrückt werden. Eine Besonderheit bilden die in den Diagrammen genutzten Vererbungsbeziehungen, die jeweils als Teilmengenbeziehungen zwischen den Mengen von Instanzen zweier Klassen verstanden werden.[3]

Das Metamodell für die Beschreibungen im bausteinbasierten Architekturentwurf ist in drei Teile unterteilt. Abschnitt 6.1 gibt zunächst einen Überblick über diese Teile und ihren Zusammenhang. Im Anschluss werden die Teilmetamodelle für die Architekturbeschreibung (Abschnitt 6.2) und die Bausteinbeschreibung (Abschnitt 6.3) vorgestellt. Die Verbindung von Baustein und Architektur entsteht im bausteinbasierten Architekturentwurf durch den Vorgang der Instanzkomposition. Die Formulierung dieser Informationen erfolgt im Teilmetamodell für die Integrationsbeschreibung (Abschnitt 6.4). Wesentlich für eine erfolgreiche Instanzkomposition von Architekturbausteinen ist die Einhaltung ihrer Eigenschaftsbedingungen. Deren Formulierung wird in Abschnitt 6.5 erläutert. Die erarbeiteten Konzepte werden in den einzelnen Abschnitten nicht nur theoretisch beschrieben sondern auch an Beispielen illustriert.

Die Abfolge, in der die einzelnen Metamodellteile in diesem Kapitel vorgestellt werden, spiegelt nur bedingt die Reihenfolge bei der Erstellung der Beschreibungen im bausteinbasierten Architekturentwurfs wider. Im Allgemeinen wird im bausteinbasierten Architekturentwurf z.B. nicht zuerst eine Architektur erstellt und im Anschluss die Rollen der Bausteine identifiziert. Stattdessen existieren zu Beginn die Bausteinbeschreibungen und gegebenenfalls eine initiale Architektur. Architekturelemente werden entweder im Rahmen einer Bausteininstanziierung erstellt – wenn an sie Bausteinrollen gebunden werden sollen aber sie zuvor noch nicht existieren – oder sie werden individuell durch den Architekten hinzugefügt.

6.1. Überblick über Beschreibung und Metamodell

Das vollständige Metamodell, das als Grundlage für die Beschreibungen im bausteinbasierten Entwurf dient, ist in drei logische getrennte, aber miteinander verbundene Teilmetamodelle unterteilt (vgl. Abbildung 6.1). Der erste Teil stellt die Mittel für die Beschreibung von Struktur und Verhalten einer komponentenbasierten Architektur zur Verfügung. Komponenten und Verbindungen zwischen Komponenten bilden die zentralen Elemente dieser Architekturbeschreibung. Hinzu kommen Elemente zur Detaillierung von Verbindungen und zum hierarchischen Aufbau von Komponenten. Das Metamodell enthält zudem Beschreibungselemente zur Modellierung von Verhalten, das auf dem Austausch von Nachrichten im System basiert.

Für die Beschreibung von Architekturbausteinen ist das zweite Teilmetamodell zuständig. Es wäre zwar naheliegend, für die Beschreibung eines Bausteins dasselbe Metamodell wie für die resultierende Architektur zu verwenden. Dem widerspricht aber der allgemeine Charakter

[3] Weitere Details zur Abbildung der im Folgenden grafisch definierten Beschreibungssprache auf eine Repräsentation durch prädikatenlogische Konzepte können dem Abschnitt 7.2.1 entnommen werden.

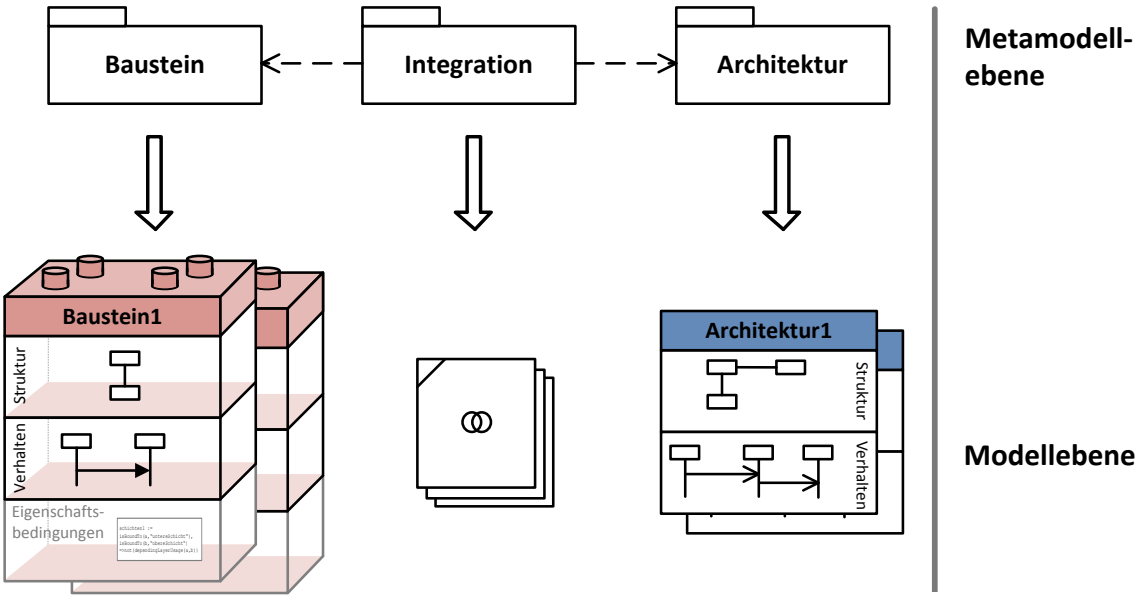


Abbildung 6.1: Übersicht über die Modell- und Metamodellebene der Beschreibungen des bausteinbasierten Ansatzes.

eines Architekturbausteins. Die Beschreibung eines Architekturbausteins abstrahiert von seiner konkreten Umsetzung im Kontext einer Architektur und ihrer Beschreibung. Zudem sind die Sprachelemente einer Architekturbeschreibungssprache oftmals durch Details einer Technologie oder einer Programmiersprache beeinflusst. Von beiden Aspekten soll die Sprache zur Beschreibung eines Architekturbausteins unabhängig sein. Nichtsdestotrotz weisen beide Teilmetamodelle Ähnlichkeiten auf. So stehen im Zentrum der Strukturbeschreibung eines Architekturbausteins Entitätenrollen und Konnektorrollen, die Parallelen zu Komponenten und ihren Verbindungen aufweisen. Auch wurde für die Verhaltensbeschreibung der Architekturbausteine ebenfalls ein nachrichtenbasierter Ansatz gewählt.

Die beiden Teilmetamodelle für die Bausteinbeschreibung auf der einen Seite und der Architekturbeschreibung auf der anderen Seite stehen in keinem direkten Bezug zueinander. Eine Beziehung zwischen beiden Teilen ist allerdings notwendig, um Rückschlüsse über gebundene Bausteine zu ermöglichen. So sind Informationen darüber von Interesse, welcher Architekturbaustein an welcher Stelle in einer Softwarearchitektur instanziiert wurde und welche Architekturelemente dabei an welche Bausteinrollen gebunden wurden. Erforderlich sind diese Informationen vor allem bei der Überprüfung der Eigenschaftsbedingungen der Bausteine. Zwar werden die Eigenschaftsbedingungen im Rahmen der Bausteinbeschreibung definiert, sie werden aber auf Basis eines Architekturmodells auf ihre Einhaltung hin untersucht. Zur Herstellung dieses Bezugs zwischen Baustein- und Architekturbeschreibung dient der dritte Teil mit dem Metamodellanteil für die Integrationsbeschreibung. Das Metamodell für die Integrationsbeschreibung definiert die Sprachmittel, um Bausteininstanzen und die in diesem Zusammenhang vollzogenen Bindungen von Bausteinrollen an Architekturelemente zu beschreiben. Die logische Trennung von Architektur- und Bausteinmetamodell führt zu einer Unabhängigkeit zwischen diesen und erlaubt den Austausch eines oder beider Metamodellteile. Ein Austausch eines der beiden Teilmetamodelle ist mindestens dann möglich, wenn zu den Rollentypen kompatible Architekturelemente existieren. Zudem muss das Metamodell der verbindenden Integrationsbeschreibung

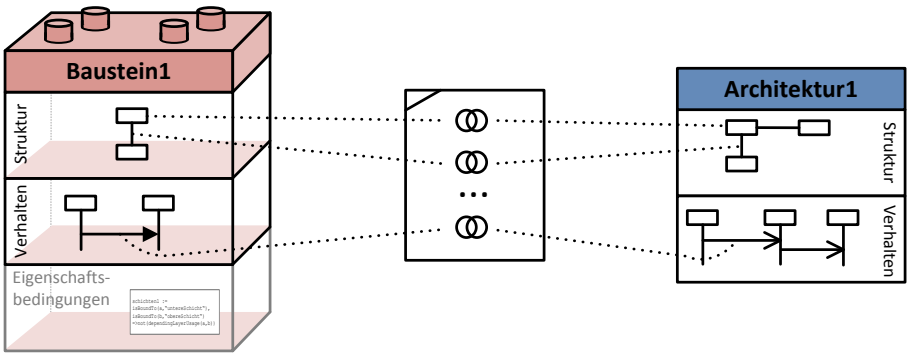


Abbildung 6.2: Abstraktes Beispiel der Modellierung einer Bausteininstanziierung.

entsprechend angepasst werden.

Abbildung 6.2 illustriert an einem abstrakten Beispiel, wie die drei Teile eines Modells auf Basis des Metamodells zueinander in Beziehung stehen. Dabei besteht sowohl ein Architekturbaustein als auch eine Architekturbeschreibung aus einer Menge von strukturellen und verhaltensbasierten Elemente. Beim Baustein werden einige dieser Elemente besonders hervorgehoben und als Rollen bezeichnet. Eine Rolle wird bei einer Instanziierung des Bausteins an ein oder mehrere Architekturelemente gebunden. Dabei wird im Metamodell der Integrationsbeschreibung festgelegt, zwischen welchem Rollentyp und welchem Architekturelementtyp eine Bindung prinzipiell möglich ist. Dies ist in der Abbildung durch die Verbindungen zwischen „Kästen“, „Linien“ und „Pfeilen“ angedeutet. Jede dieser Bindungen wird als eigenständiges Element repräsentiert. Im Metamodell wird dabei für jedes Paar aus Architekturelementtyp und Rollentyp, das einander gebunden werden darf, ein eigener Bindungstyp definiert.

6.2. Architekturbeschreibung

In der Forschungs- und Anwendungsgemeinschaft herrscht ein breiter Konsens darüber, aus welchen grundlegenden Elementen eine Architektur besteht und die damit auch durch eine Architekturbeschreibungssprache repräsentiert werden müssen [MT00]. Die Konzepte unterscheiden sich geprägt durch Technologie oder Anwendungsdomäne allerdings im Detail. So entstanden verschiedene Architekturbeschreibungssprachen (ADLs), die sich aufgrund verschiedener Zielsetzungen teilweise stark unterscheiden (vgl. auch die Übersicht in Abschnitt 3.1). Die grundlegenden Konzepte zur Architekturbeschreibung hatten auch Einfluss auf die im Jahr 2005 veröffentlichte Version 2.0 der UML Spezifikation [Obj05]. Im Gegensatz zu früheren Versionen definiert die UML nun auch Beschreibungselemente und Diagrammarten, um explizit Softwarearchitekturen zu modellieren. Um allerdings die Ansichten der verschiedenen Beteiligten ausreichend zu berücksichtigen, erlaubt das Metamodell der UML viele Freiheiten. Daher lehnt sich das in diesem Kapitel beschriebene Architekturmetamodell nur grob an die UML an. Einige Aspekte des Metamodells basieren zudem auf dem Modell aus [Her11]. Dem Verhaltensteil des Metamodells liegt hingegen kein bereits existierendes Modell zugrunde, wobei verschiedene Beschreibungssprachen ebenfalls auf nachrichten- und szenariobasierten Konzepten beruhen [MT00].

In diesem Kapitel werden zunächst grundlegende Konzepte wie der Aufbau einer Architektur (Abschnitt 6.2.1) und die Benennung ihrer Elemente (Abschnitt 6.2.2) eingeführt. Des Weiteren werden neben verschiedenen Typen für Architekturelemente (Abschnitt 6.2.3) sowohl struktur-

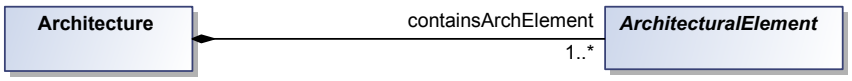


Abbildung 6.3: Architektur und Architekturelemente.

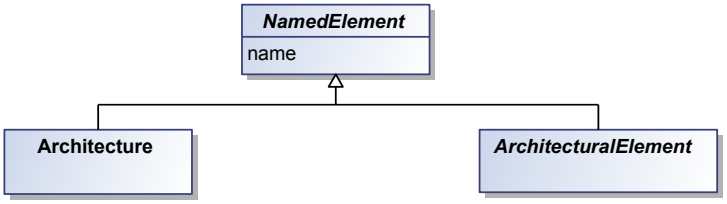


Abbildung 6.4: Benannte Elemente in der Architekturbeschreibung.

beschreibende (Abschnitt 6.2.4) als auch verhaltensbeschreibende (Abschnitt 6.2.6) Elemente definiert. Im Anschluss an die Erläuterung der struktur- und verhaltensbeschreibenden Elemente wird deren Verwendung am in Abschnitt 4.3 eingeführtem Beispielsystem illustriert (Abschnitt 6.2.5 bzw. Abschnitt 6.2.7). Die Ausarbeitung einer umfassenden Architekturbeschreibungssprache ist nicht Gegenstand dieser Arbeit. Daher erhebt das hier verwendete Metamodell keinen Anspruch auf Vollständigkeit, sondern beschränkt sich auf einen Ausschnitt, der für die Darstellung der Konzepte des bausteinbasierten Architekturentwurfs benötigt wird.

6.2.1. Architektur und Architekturelemente

Eine Architektur eines Softwaresystems wird durch ein Element vom Typ Architecture repräsentiert. Wie in Abbildung 6.3 dargestellt, besteht eine Architektur aus einer beliebigen Menge von Architekturelementen (ArchitecturalElement, containsArchElement). Der abstrakte Typ ArchitecturalElement fasst als Oberklasse die verschiedenen Arten von Architekturelementen, z.B. zur Struktur- und Verhaltensbeschreibung, zusammen. Entsprechend sind alle der im folgenden vorgestellten Elemente der Architekturbeschreibung Spezialisierungen von ArchitecturalElement.

6.2.2. Benennung von Beschreibungselementen

Namen von Elementen dienen ihrer eindeutigen Identifikation innerhalb eines abgegrenzten Namensraums. Dazu besitzen benannte Elemente das Attribut name. Ein Paket (Package), das die Typen eines Systems logisch organisiert, definiert einen Namensraum. Zudem bilden komplexe Typen wiederum einen Namensraum für ihre Bestandteile. Namensräume können auch ineinander geschachtelt werden. Obwohl die Namen lokal in ihrem Namensraum eindeutig sind, so können sie global betrachtet mehrfach verwendet werden. Um die benannten Elemente auch global eindeutig bestimmen zu können, kann zusätzlich zum Namen der Pfad durch die Hierarchie der Namensräume (vollqualifizierender Name) angegeben werden. Wie Abbildung 6.4 zeigt, können sämtliche Elemente der Architekturbeschreibung benannt werden.

6.2.3. Typen in der Architekturbeschreibung

Ein wesentliches Metaelement vieler Programmier- und Beschreibungssprachen sind Typen. Typen fassen Eigenschaften zusammen, die auf eine Menge von Elementen zutreffen. Durch die Zuweisung eines Typs an ein typisierbares Element werden die durch den Typ definierten

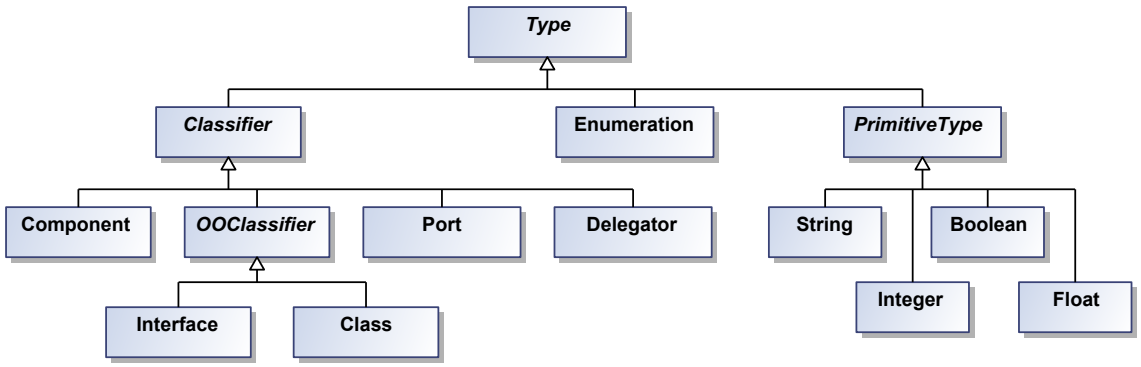


Abbildung 6.5: Typen der Architekturbeschreibung und ihre Hierarchie (nach [Her11]).

Eigenschaften auf das typisierbare Element übertragen (Typisierung). Die UML-Spezifikation beschreibt einen Typen als Beschränkung eines Bereichs von Werten, die ein entsprechend getyptes Element repräsentieren kann [Obj11].

Typhierarchie

Unterteilen lassen sich Typen (Type) in primitive Typen (PrimitiveType), Enumerationen (Enumeration) und komplexe Typen (Classifier) (vgl. Abb. 6.5). Primitive Typen wie Boolean, Integer oder String spielen in der Architekturbeschreibung nur eine untergeordnete Rolle. Sie werden in erster Linie zur Typisierung von Attributen und Parametern eingesetzt. Enumerationen sind Aufzählungstypen, die den Wertebereich eines Attributs oder Parameters auf eine Menge an Werten eingrenzen. Anders als z.B. in der UML werden sie im vorliegenden Ansatz als einfache Typen betrachtet, die kein Verhalten definieren.

Im Gegensatz zu primitiven Typen sind komplexe Typen aus anderen Typen zusammengesetzt und können sowohl eine durch Attribute und Methodensignaturen definierte Struktur als auch Verhalten besitzen. Komplexe Typen (genauer Typarten[4]) in komponentenbasierten Systemen sind Komponenten (Component), Ports (Port), Delegatoren (Delegator) und die objektorientierten Typen Schnittstelle (Interface) und Klasse (Class). In der Hierarchie zwischen den objektorientierten Typen und dem komplexen Obertypen Classifier wurde zusätzlich der abstrakte Typ OOClassifier eingeführt, um die Gemeinsamkeiten von Schnittstellen und Klassen zusammenzufassen.

Der abstrakte Typ OOClassifier, der hier von [Her11] übernommen wurde, existiert in der UML nicht. Die objektorientierten Typen sind dort direkt Spezialisierungen von Classifier. Ein weiterer Unterschied zur UML besteht darin, dass hier und in [Her11] der Typ Component direkt von Classifer abgeleitet wird; die UML sieht diesen Typ als Spezialisierung von Class. Zudem werden hier die beiden speziellen Typen Port und Delegator eingeführt, die weder in der UML noch [Her11] eine Entsprechung besitzen.

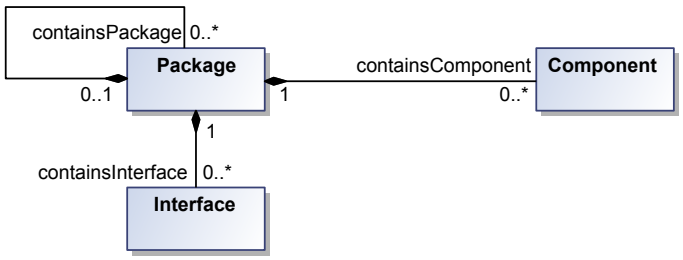


Abbildung 6.6: Logische Organisation von Typen (nach [Her11]).

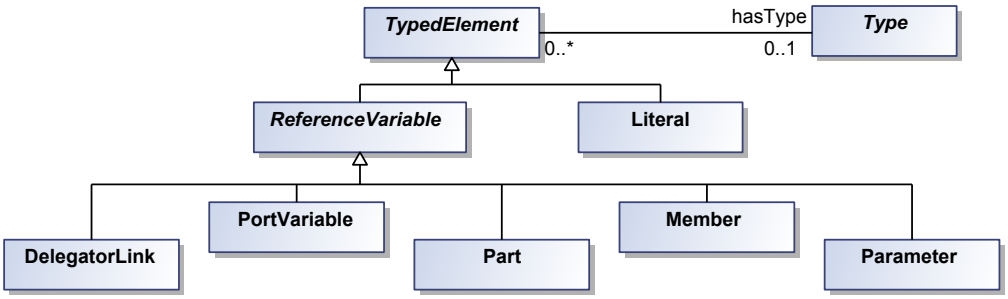


Abbildung 6.7: Typisierbare Elemente der Architekturbeschreibung (nach [Her11]).

Logische Organisation von Typen

Zur Strukturierung des Systems werden Typen wie Komponenten und Schnittstellen in Paketen (Package) gruppiert (siehe Abb. 6.6). Jede Komponente oder Schnittstelle kann dabei immer nur in genau einem Paket enthalten sein (siehe Multiplizität von containsComponent bzw. containsInterface). Zwei identisch benannte und ggf. identisch implementierte Komponenten, die in unterschiedlichen Paketen liegen, werden auch als zwei unterschiedliche Komponenten betrachtet. Außer Komponenten und Schnittstellen können Pakete ebenfalls andere Pakete enthalten (containsPkg). Diese Schachtelung von Paketen ermöglicht eine hierarchische Strukturierung von Namensräumen.

Typisierbare Elemente

Verschiedene Elemente der Architekturbeschreibung können durch die Zuweisung eines von der Oberklasse Type abgeleiteten Typs typisiert werden (hasType). Zusammengefasst werden diese typisierbaren Elemente durch die abstrakte Oberklasse TypedElement. Wie im Metamodellausschnitt aus Abbildung 6.7 dargestellt, gehören zu den typisierbaren Elementen unter anderem verschiedene Arten von Referenzvariablen (ReferenceVariable). Hierzu zählen Attribute (Member) von Klassen oder Schnittstellen und Ein- oder Ausgabeparameter (Parameter) von Methoden. Weitere Referenzvariablen sind auch die inneren Bestandteile einer Komponente (Part) und die Portvariablen (PortVariable) sowie Delegationskonnektoren (DelegatorLink) von Parts. Weitere typisierte Elemente sind Literale (Literal), die der Darstellung der Werte von primitiven Typen dienen.

Bei der Zuordnung von Typen zu typisierbaren Elementen sind Einschränkungen hinsichtlich

[4] Komponente, Schnittstelle, Klasse oder Port bezeichnen zunächst nur eine Typart. Genau genommen ist erst ein entsprechend definiertes Element mit Name und Eigenschaften ein Typ. Solange die Bedeutung aus dem Kontext ersichtlich ist, wird im Folgenden für beides der Begriff Typ verwendet.

typisierbares Element	Typen
Literal	PrimitiveType, Enumeration
Member	PrimitiveType, OOClassifier, Enumeration
Parameter	PrimitiveType, OOClassifier, Enumeration
Part	Component
PortVariable	Port
DelegatorLink	Delegator

Abbildung 6.8: Einschränkung der Zuordnung zwischen Typen und typisierbaren Elementen.

der Kompatibilität zu beachten (vgl. Abb. 6.8). Literale sind bereits entsprechend ihrer Definition auf primitive Typen eingeschränkt. Attributen, Parametern und lokalen Variablen können keine Komponententypen, sondern nur primitive Typen oder objektorientierte Typen zugewiesen werden. Hingegen können Komponentenbestandteile nur mit einem Komponententyp und Portinstanzen nur mit Ports typisiert werden.

Verglichen mit der UML wurde in [Her11] die Hierarchie der typisierbaren Elemente vereinfacht. Letztendlich wurde aus der vergleichbaren UML-Hierarchie ausschließlich die Entität Parameter übernommen. Hingegen wurden Member und Part als Entitäten in das Modell aufgenommen, die in der UML lediglich als Attribute der dortigen Entitäten Classifier und Component aufgeführt sind.

Objektorientierte Typen

Abbildung 6.9 zeigt den Ausschnitt des Metamodells zur Beschreibung des strukturellen Aufbaus und des Zusammenhangs von objektorientierten Typen. Objektorientierte Typen (OOClassifier), also sowohl Klassen als auch Schnittstellen, können eine Menge von Methodensignaturen (Signature) besitzen (hasSignature). Eine Signatur legt neben der Definition des Namens einer Methode eine Menge von Parametern (Parameter) fest (hasParameter), die diese Methode zur Ausführung benötigt. Zur Spezifikation ihres Rückgabewerts besitzt eine Methode maximal einen Parameter mit Namen „return“.

Objektorientierte Typen können andere objektorientierte Typen erweitern (extends). Allerdings können jeweils nur gleichartige Typen erweitert werden, d.h., dass Klassen nur Klassen und Schnittstellen nur Schnittstellen erweitern können. Durch die Erweiterung erben die erweiterten Typen die von den übergeordneten Typen definierten Signaturen (inheritsSignature). Signaturen, die ein Typ besitzt, definiert er somit entweder selber (definesSignature) oder erbt sie. Klassen erben zudem Signaturen von Schnittstellen, die sie implementieren (implements). Die Beziehungen zwischen OOClassifier und Signature entspricht einer Teilmengenbeziehung. Die Teilmenge der Tupel OOClassifier $\times$ Signature, die der Menge hasSignature entspricht, ist identisch zu der Vereinigung der Mengen inheritsSignature und definesSignature (hasSignature $=$ inheritsSignature $\cup$ definesSignature). Bedingung $\mathbf{AC_1}$ formuliert diesen Sachverhalt als prädikatenlogische Aussage:

$$\mathbf{AC_1} := \forall o \forall s : \texttt{OOClassifier}(o) \wedge \texttt{Signature}(s) \wedge (\texttt{hasSignature}(o,s) \leftrightarrow \texttt{inheritsSignature}(o,s) \vee \texttt{definesSignature}(o,s))$$

Klassen besitzen nicht nur Signaturen von Methoden, sondern implementieren durch Methodenrümpfe (MethodBody, implementsSignature) auch das Verhalten dieser Signaturen. Eine

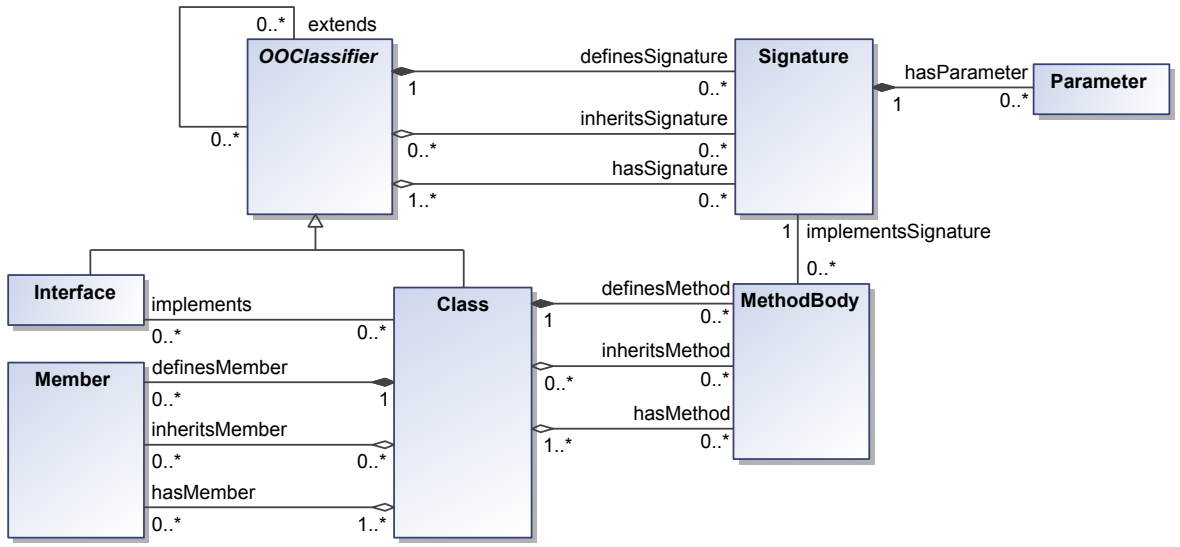


Abbildung 6.9: Struktur objektorientierter Typen (nach [Her11]).

Klasse muss dabei zu jeder Signatur, die ihr durch hasSignature zugeordnet wird, eine Implementierung in Form eines Methodenrumpfs besitzen. Die Methodenrümpfe, die eine Klasse besitzt (hasMethod), definiert sie entweder selbst (definesMethod) oder erbt sie von durch sie erweiterten Klassen (inheritsMethod). Ähnlich wie bei den Signaturen eines Typs gilt für die Beziehung zwischen Klassen und Methodenrümpfen hasMethod = inheritsMethod ∪ definesMethod und entsprechend ist die Bedingung $\mathbf{AC_2}$ formuliert:

$$\mathbf{AC_2} := \forall c \forall m : \texttt{Class}(c) \wedge \texttt{Method}(m) \wedge (\texttt{hasMethod}(c,m) \leftrightarrow \texttt{inheritsMethod}(c,m) \vee \texttt{definesMethod}(c,m))$$

Zusätzlich zu Signaturen kann eine Klasse auch Attribute (Member) definieren. Attribute als strukturelle Eigenschaften einer Klasse sind getypte Elemente, die ebenfalls an erweiternde Klassen vererbt werden können. Äquivalent zu den Beziehungen zu Methodenrümpfen gilt hasMember = inheritsMember ∪ definesMember und die Bedingung $\mathbf{AC_3}$:

$$\mathbf{AC_3} := \forall c \forall m : \texttt{Class}(c) \wedge \texttt{Member}(m) \wedge (\texttt{hasMember}(c,m) \leftrightarrow \texttt{inheritsMember}(c,m) \vee \texttt{definesMember}(c,m))$$

Der hier vorgestellte Ausschnitt des Metamodell, der der Beschreibung der Struktur objektorientierter Typen dient, entspricht bis auf wenige Änderungen dem aus [Her11]. Beide Modelle weisen einige Gemeinsamkeiten aber auch Unterschiede zu dem entsprechenden Teil der UML auf. Attribute von objektorientierten Typen werden in der UML durch das Element Property beschrieben, das in seiner Bedeutung allerdings deutlich umfangreicher ist als das hier verwendete Element Member. Auch Signature und MethodBody finden durch die Elemente Operation bzw. Behaviour eine Entsprechung in der UML.

6.2.4. Struktur von Komponenten und komponentenbasierten Systemen

In einem komponentenbasierten System sind Komponenten (Component, siehe Abb. 6.10) die modularen Einheiten, die zur Realisierung der Funktionalität des Systems miteinander verbunden werden. Komponenten werden hier zunächst in zwei Arten von Komponenten unterteilt,

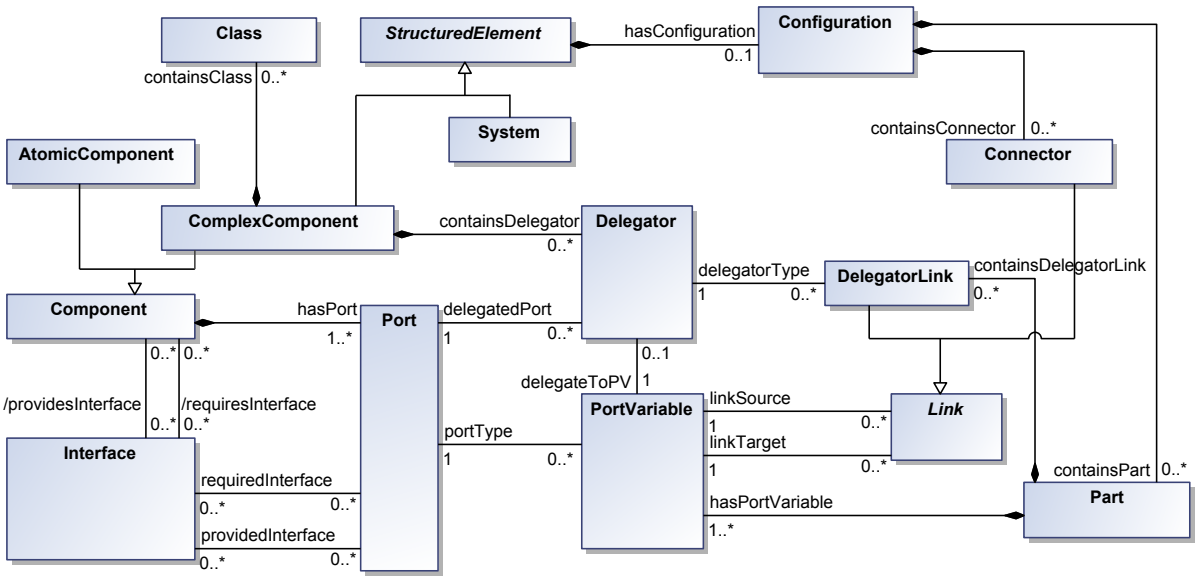


Abbildung 6.10: Struktur von Komponenten und Systemen.

nämlich in atomare (AtomicComponent) und komplexe (ComplexComponent) Komponenten. Eine *atomare Komponente* wird ausschließlich durch Klassen gebildet (containsClass). Zur Laufzeit bilden die Instanzen dieser Klassen ein Geflecht interagierender Objekte, das die Funktionalität dieser Komponente umsetzt. Interaktionen zwischen zwei Komponenteninstanzen werden durch Interaktionen zwischen Objekten realisiert, die Teile der jeweiligen Objektgeflechte sind. *Komplexe Komponenten* werden intern hingegen durch die Verknüpfung von atomaren oder komplexen Komponenten gebildet. Auch einer komplexen Komponente liegt zur Laufzeit daher letztendlich ein Objektgeflecht zugrunde, welches als Verknüpfung der Objektgeflechte der enthaltenen atomaren Komponenteninstanzen betrachtet werden kann. Die Laufzeitsicht auf eine Komponente oder ein komponentenbasiertes System wird im Folgenden aber nicht betrachtet, sondern ausschließlich die Sicht auf deren statische Struktur. So zeigt Abbildung 6.10 den Metamodellausschnitt für die Beschreibung des Aufbaus einer Komponente, wobei sowohl die innere Struktur als auch die von außen sichtbaren Eigenschaften berücksichtigt werden.

Ports und Schnittstellen

Komponenten verbergen die Details ihrer internen Struktur vor ihrer Umgebung. Sie können daher ungeachtet der Komplexität ihrer internen Realisierung einheitlich verwendet werden. Nach außen sichtbar (Blackbox-Sicht) sind lediglich die Definitionen der Interaktionsspunkte (Port, siehe Abb. 6.10) einer Komponente (hasPort). Sowohl atomare als auch komplexe Komponenten können Ports als Bestandteile besitzen. Wie mit einer Komponente kommuniziert werden kann, legen die mit den Ports assoziierten Schnittstellen (Interface) fest. Hierbei wird zwischen angebotenen und benötigten Schnittstellen unterschieden. Angebotene Schnittstellen (providedInterface) spezifizieren Funktionalität, die die Komponente zur Verfügung stellt. Benötigte Schnittstellen (requiredInterface) spezifizieren hingegen die für die Realisierung der Komponente benötigte Funktionalität. Die abgeleiteten Eigenschaften providesInterface und requiresInterface kumulieren die Schnittstellen der einzelnen Ports einer Komponente. Damit die Zuständigkeit bei der Verarbeitung von Interaktionen eindeutig ist, darf die Menge der Sig-

naturen der Schnittstellen an einem Port keine zwei gleich benannten Elemente enthalten:

$$\begin{aligned}\mathbf{AC_4} := \forall p \forall i \forall j \forall s \forall t \forall m \forall n : & \texttt{Port}(p) \wedge \texttt{Interface}(i) \wedge \texttt{Interface}(j) \wedge \\ & (\texttt{requiredInterface}(p,i) \vee \texttt{providedInterface}(p,i)) \wedge \\ & (\texttt{requiredInterface}(p,j) \vee \texttt{providedInterface}(p,j)) \wedge \\ & \texttt{Signature}(s) \wedge \texttt{Signature}(t) \wedge \texttt{hasSignature}(i,s) \wedge \texttt{hasSignature}(j,t) \wedge \\ & \texttt{name}(s,m) \wedge \texttt{name}(t,n) \wedge (m \equiv n) \rightarrow \\ & (s \equiv t)\end{aligned}$$

Konfiguration

Zusätzlich zu den nach außen sichtbaren Eigenschaften, die auch atomare Komponenten aufweisen, besitzt komplexe Komponenten eine innere Struktur. Bestandteile der inneren Struktur sind Parts (Part, siehe Abb. 6.10) und Konnektoren (Connector). Zusammengefasst wird die innere Struktur einer komplexen Komponente durch ihre Konfiguration (Configuration), die aus den Parts und Konnektoren der Komponente besteht (containsPart bzw. containsConnector). Neben der inneren Struktur von Komponenten wird ebenfalls die Struktur von Systemen (System) durch eine Konfiguration beschrieben. Aufgrund dieser Gemeinsamkeit werden sowohl Komponenten als auch Systeme als strukturierte Elemente (StructuredElement) verstanden, die beide eine Konfiguration besitzen (hasConfiguration).

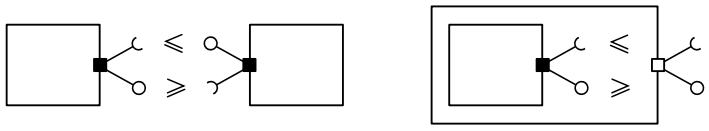
Ein System wird als eine abgeschlossene Einheit betrachtet. Im Unterschied zu Komponenten besitzt es daher nach außen keine Ports und Schnittstellen. Soll ein Menge von interagierenden Systemen betrachtet werden, können diese Systeme als Komponenten modelliert werden.

Parts und Portvariable

Parts sind benannte und komponentengetypte Elemente, aus denen eine komplexe Komponente aufgebaut ist. Sie interagieren mit anderen Parts und besitzen zu diesem Zweck definierte Interaktionspunkte, die Portvariablen (PortVariable, hasPortVariable, siehe Abb. 6.10). Portvariablen sind Konkretisierungen genau der Ports, die der Komponententyp des Parts definiert. Ein Port ist demzufolge der Typ einer Portvariablen (portType). Formal betrachtet, ist die Beziehung zwischen Port und Portvariable eine Teilmenge der Beziehung zwischen Typ und typisierbarem Element ($\text{portType} \subseteq \text{hasType}$). Die Anzahl der Portvariablen, durch die ein Port pro Part konkretisiert wird, entspricht der Anzahl der von dem Port ausgehenden Delegatoren. Jede der inneren Portvariablen wird auf diese Weise von außen als Interaktionspunkt nutzbar, wobei das konkrete Element durch die äußere Portvariable gekapselt wird. Zur Erhaltung der Konsistenz zwischen Typ und getyptem Element muss jeder Part mindestens eine Portvariable von jedem Porttypen des eigenen Komponententyps besitzen ($\mathbf{AC_5}$). Umgekehrt darf ein Part keine Portvariable besitzen, dessen Porttyp nicht durch den Komponententyp des Parts definiert ist ($\mathbf{AC_6}$):

$$\begin{aligned}\mathbf{AC_5} := \forall q \forall c \forall p : & \texttt{Part}(q) \wedge \texttt{hasType}(q,c) \wedge \texttt{Component}(c) \wedge \texttt{Port}(p) \wedge \texttt{hasPort}(c,p) \rightarrow \\ & \exists v : \texttt{PortVariable}(v) \wedge \texttt{hasPortVariable}(q,v) \wedge \texttt{portType}(v,p)\end{aligned}$$

$$\begin{aligned}\mathbf{AC_6} := \forall q \forall c \forall v : & \texttt{Part}(q) \wedge \texttt{hasType}(q,c) \wedge \texttt{Component}(c) \wedge \texttt{PortVariable}(v) \wedge \\ & \texttt{hasPortVariable}(q,v) \rightarrow \\ & \exists p : \texttt{Port}(p) \wedge \texttt{portType}(v,p) \wedge \texttt{hasPort}(c,p)\end{aligned}$$



(a) (b)

Abbildung 6.11: Kompatibilität von Portvariablen bei Konnektoren (a) und Delegatoren (b).

Konnektoren

Konnektoren (Connector, siehe Abb. 6.10) sind ungerichtete Verbindungselemente (Link). Sie beschreiben ebenso wie Delegationskonnektoren (siehe unten) Interaktionswege zwischen Parts. Dazu verbinden sie jeweils zwei Parts über deren Portvariablen miteinander. Dabei können die Enden des Verbindungselements durch ihre verschiedenen Bezeichner (linkSource, linkTarget) unterschieden werden. Ein Konnektor ist nur dann zulässig, wenn die Porttypen der jeweiligen Portvariablen über paarweise kompatible Schnittstellen verfügen. Das heißt, dass der Typ von einer der beiden Portvariablen genau die Schnittstellen oder eine ihrer Spezialisierungen anbietet, die der Typ der jeweils anderen Portvariablen benötigt. Abbildung 6.11a skizziert diesen Zusammenhang. Das Symbol $\leq$ drückt die Gleichheit oder Spezialisierung zwischen zwei Schnittstellen aus, wobei es mit der offenen Seite zu der spezielleren Schnittstelle weist. Das Prädikat `pvConCompatible(e, f)` fasst zusammen, wann die Portvariablen `e` und `f` *verbindungskompatible* Porttypen besitzen:

$$\begin{aligned}
&\texttt{pvConCompatible}(\texttt{e},\texttt{f}) := \texttt{PortVariable}(\texttt{e}) \wedge \texttt{PortVariable}(\texttt{f}) \wedge \\
&\quad \forall \texttt{p} \forall \texttt{q} : \texttt{portType}(\texttt{e},\texttt{p}) \wedge \texttt{Port}(\texttt{p}) \wedge \texttt{portType}(\texttt{f},\texttt{q}) \wedge \texttt{Port}(\texttt{q}) \wedge \\
&\quad (\forall \texttt{i} : \texttt{Interface}(\texttt{i}) \wedge \texttt{providedInterface}(\texttt{p},\texttt{i}) \rightarrow \\
&\quad\ \exists \texttt{u} : \texttt{Interface}(\texttt{u}) \wedge \texttt{requiredInterface}(\texttt{q},\texttt{u}) \wedge ((\texttt{i} \equiv \texttt{u}) \vee \texttt{extends}(\texttt{i},\texttt{u}))) \wedge \\
&\quad (\forall \texttt{j} : \texttt{Interface}(\texttt{j}) \wedge \texttt{providedInterface}(\texttt{q},\texttt{j}) \rightarrow \\
&\quad\ \exists \texttt{v} : \texttt{Interface}(\texttt{v}) \wedge \texttt{requiredInterface}(\texttt{p},\texttt{v}) \wedge ((\texttt{j} \equiv \texttt{v}) \vee \texttt{extends}(\texttt{j},\texttt{v})))
\end{aligned} \tag{6.1}$$

Somit muss jeder Konnektor an seinen Endpunkten (linkSource, linkTarget) zwei Portvariablen mit verbindungskompatiblen Porttypen besitzen:

$$\begin{aligned}
\mathbf{AC_7} := {} & \forall \texttt{c} \forall \texttt{p} \forall \texttt{q} : \texttt{Connector}(\texttt{c}) \wedge \texttt{linkSource}(\texttt{c},\texttt{p}) \wedge \texttt{PortVariable}(\texttt{p}) \wedge \\
& \texttt{linkTarget}(\texttt{c},\texttt{q}) \wedge \texttt{PortVariable}(\texttt{q}) \rightarrow \\
& \texttt{pvConCompatible}(\texttt{p},\texttt{q})^{1}
\end{aligned}$$

[1] (6.1)

Außerdem kann zwischen einem Paar von Portvariablen nur ein einziger Konnektor verlaufen:

$$\begin{aligned}
\mathbf{AC_8} := {} & \forall \texttt{c} \forall \texttt{d} \forall \texttt{p} \forall \texttt{q} : \texttt{Connector}(\texttt{c}) \wedge \texttt{linkSource}(\texttt{c},\texttt{p}) \wedge \texttt{PortVariable}(\texttt{p}) \wedge \\
& \texttt{linkTarget}(\texttt{c},\texttt{q}) \wedge \texttt{PortVariable}(\texttt{q}) \wedge \texttt{Connector}(\texttt{d}) \wedge \\
& ((\texttt{linkSource}(\texttt{d},\texttt{p}) \wedge \texttt{linkTarget}(\texttt{d},\texttt{q})) \vee (\texttt{linkSource}(\texttt{d},\texttt{q}) \wedge \texttt{linkTarget}(\texttt{d},\texttt{p}))) \\
& \rightarrow (\texttt{c} \equiv \texttt{d})
\end{aligned}$$

Konnektoren verbinden Parts über deren Portvariablen miteinander. Dabei müssen sich sowohl

die verbundenen Parts als auch der Konnektor in derselben Konfiguration befinden:

$$\begin{aligned}\mathbf{AC_9} := \forall c \forall p \forall q \forall s \forall t : &\texttt{Connector}(c) \wedge \texttt{linkSource}(c,p) \wedge \texttt{PortVariable}(p) \wedge \\ &\texttt{linkTarget}(c,q) \wedge \texttt{PortVariable}(q) \wedge \\ &\texttt{hasPortVariable}(s,p) \wedge \texttt{Part}(s) \wedge \texttt{hasPortVariable}(t,q) \wedge \texttt{Part}(t) \rightarrow \\ &\exists o : \texttt{Configuration}(o) \wedge \texttt{containsPart}(o,s) \wedge \\ &\texttt{containsPart}(o,t) \wedge \texttt{containsConnector}(o,c)\end{aligned}$$

Interaktionen innerhalb eines Parts erfolgen intern. Daher existieren keine Konnektoren, die zwei Portvariablen desselben Parts miteinander verbinden:

$$\begin{aligned}\mathbf{AC_{10}} := \forall c \forall p \forall q \forall s \forall t : &\texttt{Connector}(c) \wedge \texttt{PortVariable}(p) \wedge \texttt{linkSource}(c,p) \wedge \\ &\texttt{PortVariable}(q) \wedge \texttt{linkTarget}(c,q) \wedge \\ &\texttt{Part}(s) \wedge \texttt{hasPortVariable}(s,p) \wedge \texttt{Part}(t) \wedge \texttt{hasPortVariable}(t,q) \rightarrow \\ &\neg(s \equiv t)\end{aligned}$$

Delegatoren

Für die Verbindung der Parts im Inneren einer komplexen Komponente mit den nach außen sichtbaren Ports sind die Delegatoren (Delegator, siehe Abb. 6.10) einer Komponente (containsDelegator) zuständig. Sie ermöglichen Interaktionen zwischen der Umgebung einer Komponente und den sie realisierenden inneren Bestandteilen. Ein Delegator verläuft von einem Port der umgebenen Komponente (delegatingPort) zu einer Portvariablen eines Parts der Komponente (delegateToPV):

$$\begin{aligned}\mathbf{AC_{11}} := \forall d \forall p \forall q \forall r : &\texttt{Delegator}(d) \wedge \texttt{delegatedPort}(d,p) \wedge \texttt{Port}(p) \wedge \\ &\texttt{delegateToPV}(d,q) \wedge \texttt{PortVariable}(q) \wedge \texttt{hasPortVariable}(r,q) \wedge \texttt{Part}(r) \rightarrow \\ &\exists c \exists o : \texttt{Component}(c) \wedge \texttt{containsDelegator}(c,d) \wedge \texttt{hasConfiguration}(c,o) \wedge \\ &\texttt{Configuration}(o) \wedge \texttt{containsPart}(o,r)\end{aligned}$$

Analog zu Konnektoren zwischen Parts müssen bei einer Delegation der beteiligte Port und der Typ der beteiligten Portvariablen bezüglich ihrer Schnittstellen kompatibel sein. Dies ist gegeben, wenn im Fall von angebotenen Schnittstellen der Typ einer Portvariablen dieselben oder spezialisierende Schnittstellen wie der delegierende Port anbietet. Im Fall von benötigten Schnittstellen verhält sich dies gegensätzlich; der delegierende Port muss dieselben oder spezialisierende Schnittstellen benötigen wie der Typ der Portvariablen. Anschaulich skizziert Abbildung 6.11b die Zusammenhänge zwischen den Schnittstellen. Ein Port `p` und eine Portvariable `v` können folglich nur dann durch einen Delegator verbunden werden, wenn ihre Schnittstellen den genannten Voraussetzungen genügen. Das Prädikat $\texttt{popvDelCompatible}(p,v)$ fasst dies formal zusammen:

$$\begin{aligned}&\texttt{popvDelCompatible}(p,v) := \\ &\quad \exists q : \texttt{Port}(p) \wedge \texttt{PortVariable}(v) \wedge \texttt{portType}(v,q) \wedge \texttt{Port}(q) \wedge \\ &\quad (\forall i : \texttt{Interface}(i) \wedge \texttt{providedInterface}(p,i) \rightarrow \\ &\qquad \exists j : \texttt{Interface}(j) \wedge \texttt{providedInterface}(q,j) \wedge ((i \equiv j) \vee \texttt{extends}(j,i))) \wedge \\ &\quad (\forall j : \texttt{Interface}(j) \wedge \texttt{providedInterface}(q,j) \rightarrow \\ &\qquad \exists i : \texttt{Interface}(i) \wedge \texttt{providedInterface}(p,i) \wedge ((i \equiv j) \vee \texttt{extends}(j,i))) \wedge \\ &\quad (\forall j : \texttt{Interface}(j) \wedge \texttt{requiredInterface}(q,j) \rightarrow \\ &\qquad \exists i : \texttt{Interface}(i) \wedge \texttt{requiredInterface}(p,i) \wedge ((i \equiv j) \vee \texttt{extends}(i,j))) \wedge \\ &\quad (\forall i : \texttt{Interface}(i) \wedge \texttt{requiredInterface}(p,i) \rightarrow \\ &\qquad \exists j : \texttt{Interface}(j) \wedge \texttt{requiredInterface}(q,j) \wedge ((i \equiv j) \vee \texttt{extends}(i,j)))\end{aligned} \tag{6.2}$$

Für Port p und Portvariable v aller Delegatoren d muss nun popvDelCompatible(p,v) gelten, damit die Kompatibilität zwischen p und v sichergestellt ist:

[1](6.2)

$$\mathbf{AC_{12}} := \forall d \forall p \forall v : \texttt{Delegator}(d) \wedge \texttt{delegatedPort}(d,p) \wedge \texttt{Port}(p) \wedge \texttt{delegateToPV}(d,v) \wedge \texttt{PortVariable}(v) \rightarrow \texttt{popvDelCompatible}(p,v)^{1}$$

Da Interaktionen nur in atomaren Komponenten verarbeitet werden können, darf eine Folge von Verbindungselementen nicht an einem Part mit einem komplexen Komponententyp enden. Jeder komplexe Komponententyp muss daher für jeden seiner Porttypen einen Delegator besitzen, der diesen Port nach innen weiter delegiert:

$$\mathbf{AC_{13}} := \forall c \forall p : \texttt{ComplexComponent}(c) \wedge \texttt{Port}(p) \wedge \texttt{hasPort}(c,p) \rightarrow \exists d : \texttt{Delegator}(d) \wedge \texttt{containsDelegator}(c,d) \wedge \texttt{delegatedPort}(d,p)$$

Delegationskonnektoren

Ähnlich zu Ports, die zu Portvariablen konkretisiert werden, werden Delegatoren im Kontext von Parts zu Delegationskonnektoren (DelegatorLink, containsDelegatorLink, siehe Abb. 6.10) konkretisiert. Äquivalent gilt auch für die Typbeziehung zwischen Delegator und Delegationskonnektor, dass diese eine Teilmenge der Beziehung zwischen Typ und typisierbarem Element ist (delegatorType $\subseteq$ hasType). Ebenfalls ähnlich zu Ports und Portvariablen impliziert die Typbeziehung zwischen Part und Komponente einige Bedingungen in Bezug auf Delegatoren und Delegationskonnektoren. So muss jeder Part mindestens einen Delegationskonnektor pro Delegatortyp des Komponententyps aufweisen ($\mathbf{AC_{14}}$). Außerdem darf er keinen Delegationskonnektor besitzen, der nicht auf einen Delegator des Komponententyps verweist ($\mathbf{AC_{15}}$):

$$\mathbf{AC_{14}} := \forall p \forall c \forall d : \texttt{Part}(p) \wedge \texttt{Component}(c) \wedge \texttt{hasType}(p,c) \wedge \texttt{Delegator}(d) \wedge \texttt{containsDelegator}(c,d) \rightarrow \exists e : \texttt{DelegatorLink}(e) \wedge \texttt{containsDelegatorLink}(p,e) \wedge \texttt{delegatorType}(e,d)$$

$$\mathbf{AC_{15}} := \forall p \forall e \forall c : \texttt{Part}(p) \wedge \texttt{DelegatorLink}(e) \wedge \texttt{containsDelegatorLink}(p,e) \wedge \texttt{Component}(c) \wedge \texttt{hasType}(p,c) \rightarrow \exists d : \texttt{Delegator}(d) \wedge \texttt{containsDelegator}(c,d) \wedge \texttt{delegatorType}(e,d)$$

Delegationskonnektoren verlaufen zwischen zwei Portvariablen. Bei einer der beiden handelt es sich um eine Portvariable, die den delegierenden Port konkretisiert. Die andere Portvariable entspricht dem Ziel des Delegators, der durch den Delegationskonnektor konkretisiert wird.

$$\mathbf{AC_{16}} := \forall e \forall r \forall q \forall s \forall c : \texttt{DelegatorLink}(e) \wedge \texttt{linkSource}(e,r) \wedge \texttt{PortVariable}(r) \wedge \texttt{linkTarget}(e,q) \wedge \texttt{PortVariable}(q) \wedge \texttt{Part}(s) \wedge \texttt{hasPortVariable}(s,r) \wedge \texttt{containsDelegatorLink}(s,e) \wedge \texttt{hasType}(s,c) \wedge \texttt{Component}(c) \rightarrow \exists d \exists p : \texttt{Delegator}(d) \wedge \texttt{Port}(p) \wedge \texttt{delegatorType}(e,d) \wedge \texttt{delegatedPort}(d,p) \wedge \texttt{delegateToPV}(d,q) \wedge \texttt{portType}(r,p) \wedge \texttt{hasPort}(c,p)$$

Zwischen „inneren“ und „äußeren“ Portvariablen besteht eine 1-zu-1-Abbildung. Daher dürfen keine zwei Delegationskonnektoren innerhalb eines Parts dieselbe Portvariable als Endpunkt

besitzen:

$$\mathbf{AC_{17}} := \forall c \forall d \forall p \forall q : \texttt{DelegatorLink}(c) \wedge \texttt{DelegatorLink}(d) \wedge \texttt{Part}(p) \wedge \texttt{containsDelegatorLink}(p,c) \wedge \texttt{containsDelegatorLink}(p,d) \wedge \texttt{PortVariable}(q) \wedge \texttt{linkEnd}(c,q)^{1} \wedge \texttt{linkEnd}(d,q) \rightarrow (c \equiv d) \quad {}^{1(6.3)}$$

Für obige Definition wurde die Überprüfung, ob eine Portvariable `e` ein Ende der Verbindung `l` ist, durch das Prädikat `linkEnd(l,e)` verkürzt:

$$\texttt{linkEnd}(l,e) := \texttt{Link}(l) \wedge \texttt{PortVariable}(e) \wedge (\texttt{linkSource}(l,e) \vee \texttt{linkTarget}(l,e)) \tag{6.3}$$

Für die Kompatibilität zwischen innerer und äußerer Portvariable bezüglich eines Delegationskonnektors gilt ähnliches wie zwischen Port und Portvariable einer Delegation. Die beiden Portvariablen müssen *delegationskompatibel* sein. Dementsprechend müssen ihre Porttypen Schnittstellen aufweisen, die in bereits oben genannter Weise zueinander in Beziehung stehen:

$$\mathbf{AC_{18}} := \forall d \forall a \forall b : \texttt{DelegatorLink}(d) \wedge \texttt{linkSource}(d,a) \wedge \texttt{PortVariable}(a) \wedge \texttt{linkTarget}(d,b) \wedge \texttt{PortVariable}(b) \rightarrow \texttt{pvDelCompatible}(a,b)^{1} \quad {}^{1(6.4)}$$

Das Prädikat `pvDelCompatible(a,b)` zur Überprüfung der Delegationskompatibilität zweier Portvariablen lässt sich basierend auf der Kompatibilität zwischen Portvariable und Port einer Delegation (siehe Gleichung 6.2) definieren:

$$\texttt{pvDelCompatible}(a,b) := \texttt{PortVariable}(a) \wedge \texttt{PortVariable}(b) \wedge \exists p : \texttt{portType}(a,p) \wedge \texttt{Port}(p) \wedge \texttt{popvDelCompatible}(p,b) \tag{6.4}$$

Für die Sicherstellung der Kompatibilität zwischen „innerer“ und „äußerer“ Portvariable ist es nicht zwingend erforderlich, die Bedingung $\mathbf{AC_{18}}$ aufzustellen. Diese wird bereits indirekt durch zwei andere Bedingungen zugesichert. Durch Bedingung $\mathbf{AC_{16}}$ ist zunächst sicher gestellt, dass ein Delegationskonnektor nur zwischen zwei Portvariablen besteht, die bereits durch den Delegator, den dieser Delegationskonnektor konkretisiert, in Beziehung stehen. Die Kompatibilität zwischen der „inneren“ Portvariablen und dem Porttyp der „äußeren“ Portvariablen des konkretisierten Delegators wird schließlich durch Bedingung $\mathbf{AC_{12}}$ gewährleistet.

Im Unterschied zu Konnektoren besitzt ein Delegationskonnektor eine Richtung. Zu deren Bestimmung werden die beiden von Link geerbten Attribute linkSource und linkTarget genutzt. Dabei wird festgelegt, dass linkSource auf die Portvariable des „äußeren“ Parts verweist und linkTarget auf die Portvariable des „inneren“ Parts:

$$\mathbf{AC_{19}} := \forall e \forall s \forall c \forall d \forall r \forall q \forall p \forall t \forall o : \texttt{DelegatorLink}(e) \wedge \texttt{Part}(s) \wedge \texttt{containsDelegatorLink}(s,e) \wedge \texttt{Component}(c) \wedge \texttt{hasType}(s,c) \wedge \texttt{Delegator}(d) \wedge \texttt{delegatorType}(e,d) \wedge \texttt{containsDelegator}(c,d) \wedge \texttt{Port}(r) \wedge \texttt{delegatedPort}(d,r) \wedge \texttt{hasPort}(c,r) \wedge \texttt{PortVariable}(q) \wedge \texttt{portType}(q,r) \wedge \texttt{hasPortVariable}(s,q) \wedge \texttt{PortVariable}(p) \wedge \texttt{delegateToPV}(d,p) \wedge \texttt{Part}(t) \wedge \texttt{hasPortVariable}(t,p) \wedge \texttt{Configuration}(o) \wedge \texttt{containsPart}(o,t) \wedge \texttt{hasConfiguration}(c,o) \rightarrow \texttt{linkSource}(e,q) \wedge \texttt{linkTarget}(e,p)$$

Vergleich des Metamodells mit [Her11] und der UML

Außer im Umfang unterscheidet sich der hier vorgestellte Metamodellausschnitt zur strukturellen Beschreibung von Komponenten und Systemen konzeptionell vor allem in zwei Punkten von der UML und von [Her11]. Einer dieser Punkte ist die Trennung der Definition eines Interaktionspunktes als Teil einer Komponentenspezifikation (Port) von seiner Nutzung als Teil eines Parts (PortVariable). Dies ermöglicht es, die Enden eines Konnektors und die zugehörigen Parts eindeutig und ohne Umwege zu identifizieren. Für diesen Zweck nutzen die UML als auch [Her11] den jeweiligen Part als Kontext des Ports. Eng verbunden mit dem vorherigen Konzept ist auch die Trennung von Delegator und seiner Nutzung innerhalb eines Parts. Hierdurch ist nicht nur der delegierende Porttyp, sondern die konkrete Portvariable identifizierbar.

Die von [Her11] eingeführte Spezialisierung von Ports in angebotene und benötigte Ports wurde nicht übernommen. Dieses Konzept erlaubt keine Mischung von benötigten und angebotenen Schnittstellen an einem gemeinsamen Port, sondern erfordert ihre Unterteilung in benötigte und angebotene Schnittstellen. Dies widerspricht dem Verständnis eines Ports als bidirektionalem Interaktionspunkt im vorliegenden Ansatz.

Ein weiterer wesentlicher Unterschied besteht in der Modellierung von Konnektoren und Delegationskonnektoren. Während die UML und [Her11] diesen Unterschied lediglich durch einen Wahrheitswert spezifizieren, werden im vorliegendem Ansatz zwei eigenständige Elemente definiert. Die Unterteilung in zwei verschiedene Elemente ist nicht zuletzt deshalb erforderlich, weil die beiden Konnektorarten jeweils teilweise unterschiedliche Architekturelemente miteinander verbinden.

Hingegen wurde aus [Her11] das Konzept eines eigenen strukturellen Elements zur Definition einer Systemkonfiguration übernommen, das die UML nicht kennt. Die Modellierung eines Systems ist in der UML nichtsdestotrotz z.B. durch die Definition eines Stereotyps für das Komponentenelement mit entsprechenden Einschränkungen möglich.

6.2.5. Beispiel: Strukturelle Architekturbeschreibung

In den vorangegangenen Abschnitten wurden die Bereiche des Metamodells zur Architekturbeschreibung eingeführt, die für die Modellierung der strukturellen Anteile vorgesehen sind. Ihre Verwendung wird im folgenden Abschnitt an einem Beispiel erläutert. Als Beispiel dient dabei das in Abschnitt 4.3 entwickelte System, das bisher zunächst informell beschrieben wurde. Hierbei handelt es sich um ein dreischichtiges System aus Präsentation, Anwendungslogik und Datenhaltung, wobei die beiden erst genannten Schichten durch weitere GUI-Elemente bzw. Elemente zur Datenverwaltung verfeinert werden. Zum Zweck der Änderungsaktualisierung wurde in Abbildung 4.8 zwischen den Schichten jeweils ein Observermechanismus vorgesehen, die im Folgenden detailliert werden.

Präsentation

Die Präsentationsschicht des Systems wird durch ein Element vom Typ der komplexen Komponente Präsentation realisiert. Abbildung 6.12 zeigt den Aufbau dieser Komponente in der Whitebox-Sicht. Dies bedeutet, dass sowohl ihre nach außen sichtbaren Elemente als auch ihre interne Struktur sichtbar sind. Nach außen sichtbar sind die drei Ports gui_kobs_port, gui_bobs_port und gui_al_port. Die ersten beiden Ports realisieren jeweils Observerfunktionalität, weswegen beide die Schnittstelle SubjectIF benötigen und die Schnittstelle ObserverIF anbieten. Trotz der identischen Schnittstellen liegt der Fokus der beiden Observer auf unterschiedlichen Informationen.

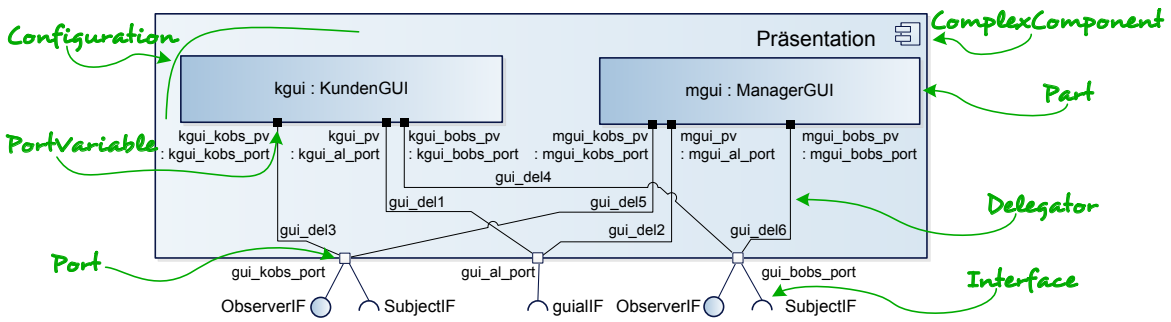


Abbildung 6.12: Aufbau der Komponente Präsentation.

(a) (b)

Abbildung 6.13: Aufbau der Komponenten KundenGUI (a) und ManagerGUI (b).

Während gui_kobs_port zur Reaktion auf Änderungen von Kundendaten dient, ist kgui_bobs_port für die Reaktion auf Änderungen von Bestellungsdaten gedacht. Der Port gui_al_port benötigt hingegen nur die Schnittstelle guiaIIF.

Die Konfiguration der Komponente besteht aus zwei Parts. kgui vom Typ KundenGUI ist für die Anwendungsoberfläche zur Pflege und Anzeige von Kundendaten zuständig, während wgui vom Typ ManagerGUI Zugriff auf sämtliche Daten im System ermöglicht. Untereinander sind diese beiden Parts in der Komponente nicht verbunden. Für ihre Aufgaben benötigen die Parts allerdings Informationen von außerhalb der umgebenden Komponente. Um die notwendigen Interaktionen zu ermöglichen, sind beide Parts durch Delegatoren an ihren jeweiligen Portvariablen mit den verschiedenen Ports von Präsentation verbunden. So verläuft z.B. der Delegator del3 von der mit kgui_kobs_port getypten Portvariablen kgui_kobs_pv von kgui zum Port gui_kobs_port. Die Konsistenzbedingung bei Verwendung von Delegatoren erfordert Schnittstellenkompatibilität zwischen den Porttypen an den jeweiligen Konnektorenden. Wie die Beschreibung der Komponententypen KundenGUI und ManagerGUI in Abbildung 6.13 zeigt, ist die geforderte Kompatibilität gegeben, da die Ports von Präsentation sogar jeweils dieselben Schnittstellen benötigen und anbieten wie die Porttypen der durch die Delegatoren verbundenen Portvariablen.

Anwendungslogik

Die Komponente Anwendungslogik bildet den Typen des Parts, der die Schicht Anwendungslogik des Beispielsystems realisiert. Ihr Aufbau ist in Abbildung 6.14 in der Whitebox-Sicht dargestellt. Die Definition ihrer Ports al_kobs_port, al_gui_port und al_bobs_port ist komplementär zu den drei Ports der Komponente Präsentation. Zur Realisierung der Observerfunktionalität bieten die beiden Ports al_kobs_port und al_bobs_port jeweils die Schnittstelle SubjectIF an und benötigen beide die Schnittstelle ObserverIF. Der Port al_gui_port bietet hingegen die Schnittstelle guiaIIF an. Zur Interaktion mit der Datenhaltung besitzt die Komponente Anwendungslogik zwei weitere Ports. al_dataobs_port weist Eigenschaften für die Realisierung von Observerfunktiona-

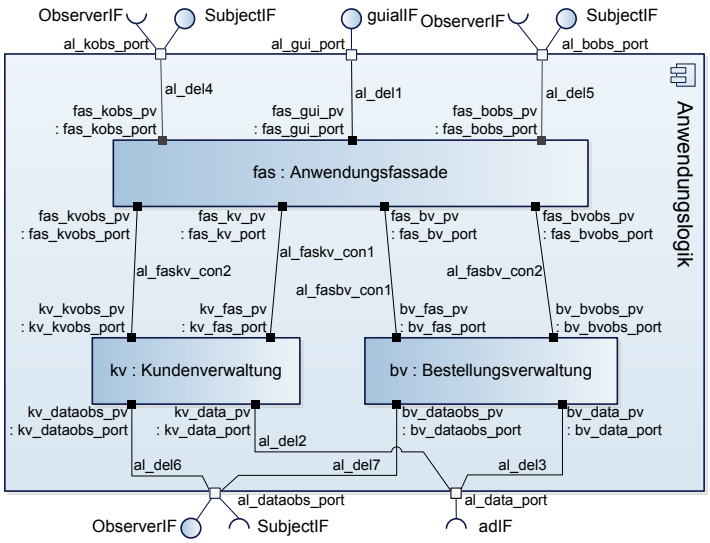


Abbildung 6.14: Aufbau der Komponente Anwendungslogik.

lität auf: er bietet die Schnittstelle OberserverIF an und benötigt die Schnittstelle SubjectIF. Zusätzlich existiert der Port al_data_port mit der Schnittstelle adIF zum Datenaustausch.

Die innere Struktur der Komponente Anwendungslogik ist die komplexeste der in diesem Beispiel gezeigten Komponenten. Aufgaben dieser Komponente sind die Verwaltung der Daten des Systems und die Steuerung von Abläufen. Zu diesem Zweck enthält sie zunächst die beiden unterschiedlich Parts kv und bv, die jeweils für einen anderen Bereich verantwortlich sind. Ableiten lässt sich der Verantwortungsbereich aus dem Namen des jeweiligen Komponententyps (Kundenverwaltung bzw. Bestellungsverwaltung). Interaktionen über die Ports al_kobs_port, al_bobs_port und al_gui_port zu diesen Parts werden durch eine Fassade gekapselt; diese Funktion übernimmt der Part fas vom Typ Anwendungsfassade. Daher sind die beiden Verwaltungsparts durch insgesamt vier Konnektoren mit dem Fassadenpart fas verbunden. Einer dieser Konnektoren ist al_faskv_con1, der von der Portvariablen kv_kvobs_pv vom Typ kv_kvobs_port am Part kv zur Portvariablen fas_kvobs_pv vom Typ fas_kvobs_port verläuft.

Die Parts im Inneren der Komponente sind mit den Ports der Komponente durch Delegatoren verbunden. Der Fassadenpart fas ist über seine Portvariablen fas_kobs_pv, fas_gui_pv und fas_bobs_pv durch Delegatoren mit jeweils einem der korrespondierenden Ports al_kobs_port, al_gui_port bzw. al_bobs_port verbunden. In die beiden Ports al_dataobs_port und al_data _port laufen jeweils pro Verwaltungspart ein Delegator ein. So verlaufen z.B. zum Port al_dataobs_port die beiden Delegatoren al_del6 und al_del7. Dabei geht das jeweils andere Ende von Portvariable kv_dataobs_pv des Parts kv bzw. von Portvariable bv_dataobs_pv des Parts bv aus.

Abbildung 6.15 gibt eine Übersicht über die Komponententypen der eben als Bestandteile von Anwendungslogik beschriebenen Parts. Im Vergleich mit Abbildung 6.14 sind u.a. die Kompatibilitäten zwischen den verschiedenen Ports zu erkennen. Diese erlauben es, Konnektoren zwischen den Portvariablen oder im Fall von Delegatoren zwischen Portvariablen und Ports zu etablieren. So stimmen die Schnittstellenspezifikationen der Ports kv_dataobs_port und bv_dataobs_port bzw. kv_data_port und bv_data_port der Verwaltungskomponententypen mit der des Ports al_dataobs_port bzw. al_data_port der Anwendungslogik überein. Ebenfalls identisch sind die Schnittstellenspezifikationen der Ports al_kobs_port, al_gui_port und al_bobs_port der Anwendungslogik mit denen der korrespondierenden Ports fas_kobs_port, fas_gui_port und fas_kobs_port

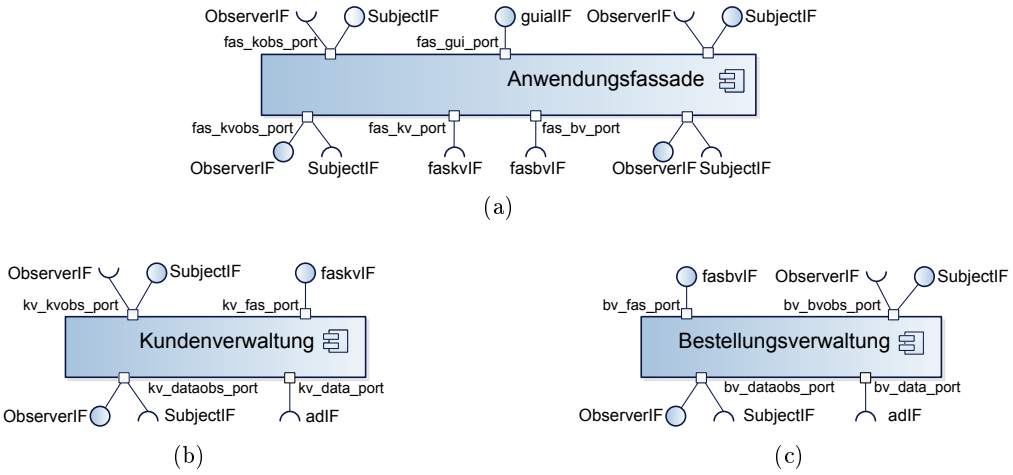


Abbildung 6.15: Aufbau der Komponenten Anwendungsfassade (a), Kundenverwaltung (b) und Bestellungsverwaltung (c).

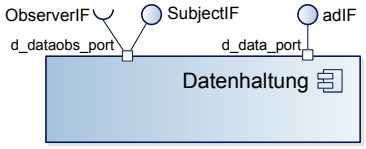


Abbildung 6.16: Aufbau der Komponente Datenhaltung.

der Fassadenkomponente. Die Fassadenkomponente besitzt außerdem in Richtung ihres Subsystems die Ports fas_kvobs_port, fas_kv_port, fas_bv_port und fas_bvobs_port. Deren Schnittstellen sind komplementär zu den Schnittstellen jeweils eines Ports von Kundenverwaltung und Bestellungsverwaltung.

Datenhaltung

Abbildung 6.16 zeigt die Komponente Datenhaltung. Sie ist der Typ des Parts, der die Datenhaltungsschicht in dem Beispielsystem repräsentiert. Wie zu sehen ist, handelt es sich hierbei um eine atomare Komponente, da sie bezüglich des vorliegenden Komponentenmodells keine innere Struktur besitzt. Der Port d_dataobs_port stellt die Schnittstellen für die Observerfunktionalität zur Verfügung. Er ist komplementär zu der Spezifikation des Ports al_dataobs_port der Komponente Anwendungslogik. Ebenso passt der Port d_data_port mit der angebotenen Schnittstelle adIF zu dem Port al_data_port der Anwendungslogik, die ebendiese Schnittstelle benötigt.

Systemkonfiguration

Abbildung 6.17 zeigt die Systemkonfiguration des Beispielsystems system in einer hierarchischen Whitebox-Sicht. Die drei Schichten des Systems sind jeweils als ein Part der Konfiguration modelliert. Die Präsentationsschicht wird durch den Part namens gui vom Typ Präsentation abgebildet. Die drei Ports der Komponente Präsentation werden jeweils durch zwei Portvariablen konkretisiert. Hierbei sind jeweils zwei Portvariable von demselben Porttyp; dargestellt wird dies durch ein Rechteck um jeweils die beiden Portvariablen, die denselben Porttypen besitzen. Für jede durch einen Konnektor an den Port angebundene Portvariable eines „inneren“ Parts

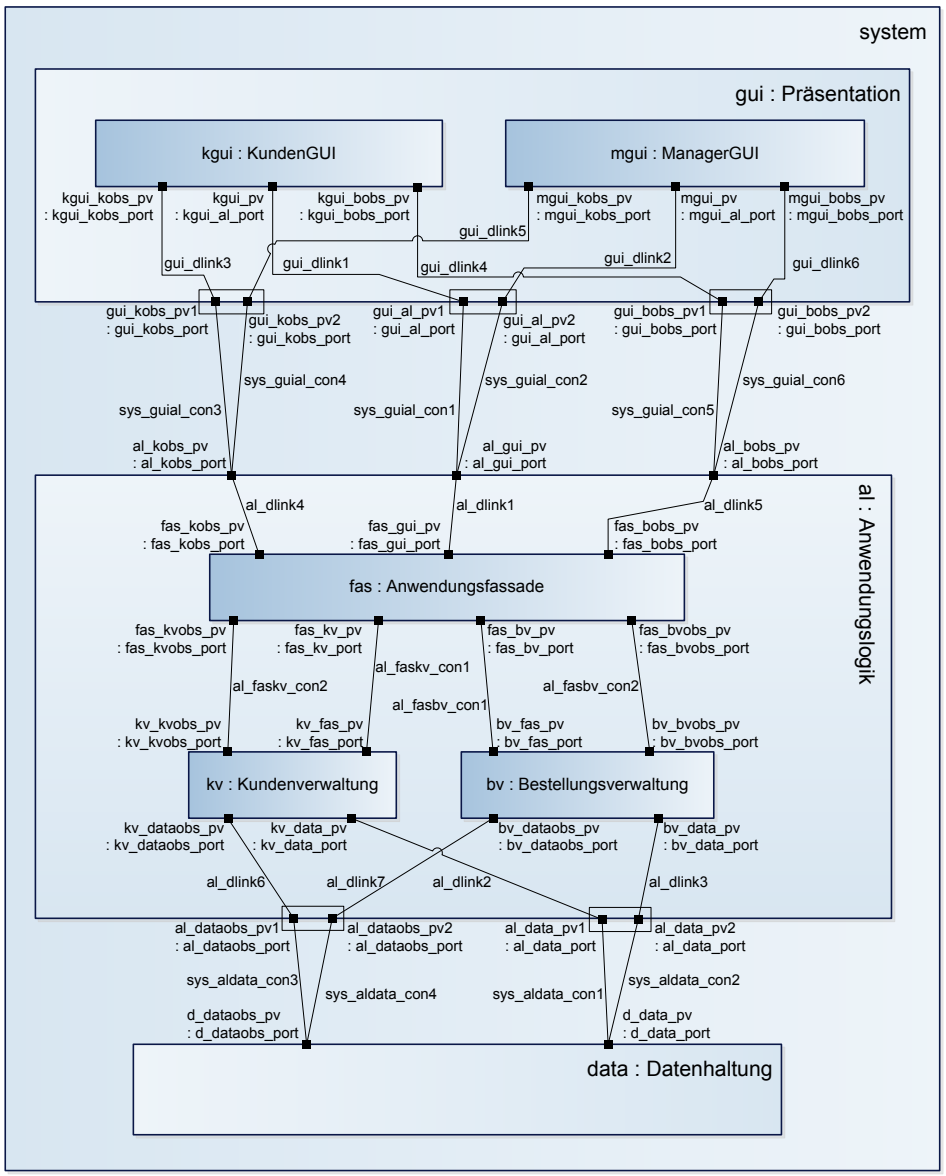


Abbildung 6.17: Aufbau des Gesamtsystems.

wird eine Portvariable am „äußeren“ Part erzeugt. Die Delegatoren des Komponententyps, die zwischen seinen Ports und den Portvariablen seiner Parts verlaufen, werden ebenfalls konkretisiert. Dies geschieht durch die Delegationskonnektoren gui_dlink1 bis gui_dlink6 zwischen den „inneren“ und „äußeren“ Portvariablen.

Äquivalent werden auch die Ports und Delegatoren der Komponente Anwendungslogik für den Part al konkretisiert, der die Anwendungslogikschicht repräsentiert. Pro Portvariable auf der Seite von gui läuft jeweils einer der Konnektoren sys_guial_con1 bis sys_guial_con6 aus. Dabei laufen jeweils zwei dieser Konnektoren an jeder Portvariable von al ein.

Die Verbindungsstruktur zwischen al und dem Part data vom Typ Datenhaltung ist ähnlich zu derjenigen zwischen gui und al konkretisiert. Die Anzahl der beteiligten Portvariablen von al entspricht der Anzahl der im Typ Anwendungslogik durch Delegatoren an die Ports al_dataobs_port

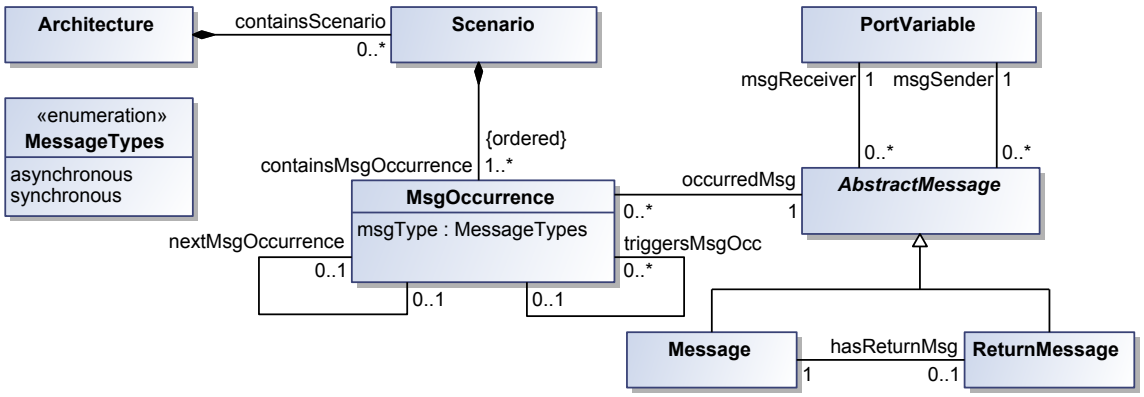


Abbildung 6.18: Verhalten in einem komponentenbasierten System.

und al_data_port angebundenen Portvariablen. Entsprechend laufen in den beiden Portvariablen von data jeweils zwei der Konnektoren sys_aldata_con1 bis sys_aldata_con4 zusammen ein.

6.2.6. Szenariobasierte Verhaltensbeschreibung

Die Verhaltensbeschreibung eines komponentenbasierten Systems erfolgt in dieser Arbeit durch den Einsatz von Szenarios (Scenario) als Teil der Architektur (containsScenario). Ein Szenario ist ein Ausschnitt aus dem Ablauf eines Systems. Es zeigt mit einer festen Abfolge von Interaktionen, wie sich das System in bestimmten Situationen verhält. Während der Laufzeit eines Systems können die Szenarios beliebig häufig und in beliebiger Reihenfolge auftreten, allerdings müssen nicht zwingend alle definierten Szenarios durchlaufen werden. Der formale Aufbau eines Szenarios wird durch den Metamodellausschnitt in Abbildung 6.18 ebenso definiert wie die Modellierung von Interaktionen.

Nachrichten und Antwortnachrichten

Interaktionen werden in Nachrichten (Message, siehe Abb. 6.18) und Antwortnachrichten (ReturnMessage) unterschieden. Antwortnachrichten stehen immer im Bezug zu einer zuvor gesendeten Nachricht, die eine Antwort erwartet (hasReturnMsg). Die gemeinsamen Eigenschaften der beiden Nachrichtenarten werden im Metamodell durch die gemeinsame Oberklasse AbstractMessage modelliert. Informell werden Parts als die interagierenden Elemente betrachtet, wobei der Nachrichtenaustausch korrekterweise über ihre Portvariablen erfolgt. Jede Nachricht oder Antwortnachricht besitzt sowohl genau eine Portvariable als Empfänger (msgReceiver) als auch genau eine als Sender (msgSender). Zwischen welchen Portvariablen dabei Nachrichten und Antwortnachrichten ausgetauscht werden können, wurde bereits in der Strukturbeschreibung festgelegt. So werden durch die Verbindungselemente (Konnektoren und Delegationskonnektoren), über die zwei Portvariablen unterschiedlicher Parts miteinander verbunden sind, Interaktionswege vorgegeben. Außerdem sind Interaktionen zwischen den Portvariablen eines Parts erlaubt, der durch eine atomare Komponente getypt ist. Diese Interaktionen abstrahieren Abläufe, die innerhalb von atomaren Komponenten durch deren Klassen definiert sind und im vorliegenden Architekturmodell nicht modelliert werden. Dass Interaktionen nur über die festgelegten

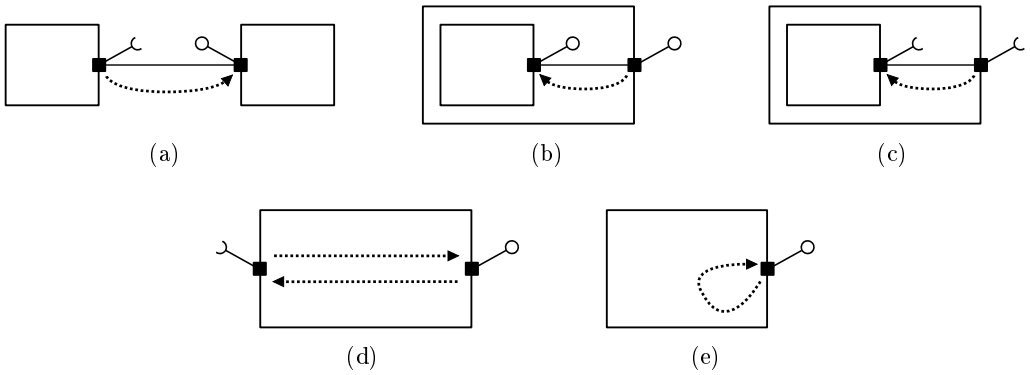


Abbildung 6.19: Schematische Darstellung der Senderichtung von Nachrichten bei Konnektoren (a), Delegationskonnektoren bei angebotenenen (b) und benötigten (c) Schnittstellen und innerhalb von Parts mit atomaren Komponententyp (d,e).

Interaktionswege erfolgen, stellt $\mathbf{AC_{20}}$ sicher:

$$\begin{aligned}\mathbf{AC_{20}} := {} & \forall \mathtt{m} \forall \mathtt{p} \forall \mathtt{q} : \mathtt{PortVariable}(\mathtt{p}) \wedge \mathtt{PortVariable}(\mathtt{q}) \wedge \mathtt{AbstractMessage}(\mathtt{m}) \wedge \\ & \mathtt{msgSender}(\mathtt{p},\mathtt{m}) \wedge \mathtt{msgReceiver}(\mathtt{q},\mathtt{m}) \rightarrow \\ & (\exists \mathtt{l} : \mathtt{Link}(\mathtt{l}) \wedge \mathtt{linkEnd}(\mathtt{l},\mathtt{p}) \wedge \mathtt{linkEnd}(\mathtt{l},\mathtt{q})) \vee \\ & (\exists \mathtt{r} \exists \mathtt{a} : \mathtt{Part}(\mathtt{r}) \wedge \mathtt{AtomicComponent}(\mathtt{a}) \wedge \mathtt{hasType}(\mathtt{r},\mathtt{a}) \wedge \\ & \mathtt{hasPortVariable}(\mathtt{r},\mathtt{p}) \wedge \mathtt{hasPortVariable}(\mathtt{r},\mathtt{q}))\end{aligned}$$

Die Botschaft, die durch eine Nachricht oder Antwortnachricht übermittelt wird, entspricht dem Namen, den sie als NamedElement besitzt. Welche Nachrichten eine Portvariable senden oder empfangen kann, ist durch die Schnittstellen ihres Porttyps definiert. Sowohl in einer Schnittstelle des Senders als auch in einer Schnittstelle des Empfängers muss eine passende Signatur existieren. Eine Signatur wird dann als passend betrachtet, wenn die Namen von Signatur und Nachricht identisch sind. Wann dementsprechend eine Schnittstelle und eine Nachricht kompatibel sind, fasst das Prädikat ifMsgCompatible für die nachfolgenden Bedingungen zusammen:

$$\begin{aligned}\mathtt{ifMsgCompatible}(\mathtt{i},\mathtt{m}) := {} & \mathtt{Interface}(\mathtt{i}) \wedge \mathtt{Message}(\mathtt{m}) \wedge \\ & \exists \mathtt{s} \exists \mathtt{a} \exists \mathtt{b} : \mathtt{Signature}(\mathtt{s}) \wedge \mathtt{hasSignature}(\mathtt{i},\mathtt{s}) \wedge \\ & \mathtt{name}(\mathtt{m},\mathtt{a}) \wedge \mathtt{name}(\mathtt{s},\mathtt{b}) \wedge (\mathtt{a} \equiv \mathtt{b})\end{aligned} \tag{6.5}$$

Je nach Art der Verbindung zwischen den interagierenden Portvariablen ist die Richtung der Nachricht vorgegeben und für Sender und Empfänger müssen bestimmte Kriterien gelten. Wird eine Nachricht zwischen zwei durch einen Konnektor verbundenen Portvariablen ausgetauscht, müssen Empfänger und Sender eine Schnittstelle anbieten bzw. benötigen, die zu der Nachricht kompatibel ist (vgl. die schematische Darstellung in Abb. 6.19a). Dies trifft auch auf Nachrichten zu, die entlang von Delegationskonnektoren ausgetauscht werden (siehe Abb. 6.19b und 6.19c). Auch hier müssen die Schnittstellen zu den Nachrichten passende Signaturen aufweisen. In diesen drei genannten Fällen besitzt der Porttyp der sendenden Portvariable die gleiche oder generellere Schnittstelle der beiden Interaktionspartner (vgl. auch Abbildung 6.11a in Abschnitt 6.2.4). Die Überprüfung der Senderseite ist daher ausreichend, weil die Gültigkeit der in Abschnitt 6.2.4 definierten Bedingungen zur Verbindungs- ($\mathbf{AC_7}$) und Delegationskompatibilität ($\mathbf{AC_{12}}$) von Portvariablen die Existenz einer passenden Signatur auf Empfängerseite

sicherstellt. Ob der Austausch einer Nachricht in den drei genannten Fällen (a), (b) und (c) in korrekter Weise modelliert ist, wird durch die drei Bedingungen $\mathbf{AC_{21}}$, $\mathbf{AC_{22}}$ bzw. $\mathbf{AC_{23}}$ sichergestellt:

$$\begin{aligned}\mathbf{AC_{21}} := {} & \forall m \forall p \forall q \exists t \exists c : \texttt{Message}(m) \wedge \texttt{PortVariable}(p) \wedge \texttt{PortVariable}(q) \wedge \neg(p \equiv q) \wedge \\ & \texttt{msgSender}(m,p) \wedge \texttt{msgReceiver}(m,q) \wedge \texttt{Port}(t) \wedge \texttt{hasType}(p,t) \wedge \\ & \texttt{Connector}(c) \wedge \texttt{linkEnd}(c,p)^{1} \wedge \texttt{linkEnd}(c,q) \rightarrow \\ & \exists i : \texttt{Interface}(i) \wedge \texttt{requiredInterface}(t,i) \wedge \texttt{ifMsgCompatible}(i,m)^{2}\end{aligned} \quad {}^{1(6.3),}\,{}^{2(6.5)}$$

$$\begin{aligned}\mathbf{AC_{22}} := {} & \forall m \forall p \forall q \exists t \exists d : \texttt{Message}(m) \wedge \texttt{PortVariable}(p) \wedge \texttt{PortVariable}(q) \wedge \neg(p \equiv q) \wedge \\ & \texttt{msgSender}(m,p) \wedge \texttt{msgReceiver}(m,q) \wedge \texttt{Port}(t) \wedge \texttt{hasType}(p,t) \wedge \\ & \texttt{DelegatorLink}(d) \wedge \texttt{linkSource}(c,p) \wedge \texttt{linkTarget}(c,q) \rightarrow \\ & \exists i : \texttt{Interface}(i) \wedge \texttt{providedInterface}(t,i) \wedge \texttt{ifMsgCompatible}(i,m)^{1}\end{aligned} \quad {}^{1(6.5)}$$

$$\begin{aligned}\mathbf{AC_{23}} := {} & \forall m \forall p \forall q \exists t \exists d : \texttt{Message}(m) \wedge \texttt{PortVariable}(p) \wedge \texttt{PortVariable}(q) \wedge \neg(p \equiv q) \wedge \\ & \texttt{msgSender}(m,p) \wedge \texttt{msgReceiver}(m,q) \wedge \texttt{Port}(t) \wedge \texttt{hasType}(p,t) \wedge \\ & \texttt{DelegatorLink}(d) \wedge \texttt{linkTarget}(c,p) \wedge \texttt{linkSource}(c,q) \rightarrow \\ & \exists i : \texttt{Interface}(i) \wedge \texttt{requiredInterface}(t,i) \wedge \texttt{ifMsgCompatible}(i,m)^{1}\end{aligned} \quad {}^{1(6.5)}$$

Etwas anders stellt sich dies bei Nachrichten zur Abstraktion der innere Abläufe in Parts mit atomaren Komponententyp dar (vgl. die schematische Darstellung in Abb. 6.19d). Hier wird eine unspezifizierte Menge von Interaktionen auf Klassenebene durch eine Nachricht abgekürzt. Die sendende Portvariable besitzt so im Allgemeinen keine Schnittstelle mit einer zu der Nachricht passenden Signatur. Allerdings muss die empfangende Portvariable eine passende Schnittstelle aufweisen, wobei die Art der Schnittstelle irrelevant ist. Handelt es sich bei der Schnittstelle um eine angebotene Schnittstelle, wird die Nachricht in der Folge vom Part selber verarbeitet; bei einer benötigten Schnittstelle wird diese Nachricht weitergereicht.

$$\begin{aligned}\mathbf{AC_{24}} := {} & \forall m \forall p \forall q \exists t \exists b \exists a : \texttt{Message}(m) \wedge \texttt{PortVariable}(p) \wedge \texttt{PortVariable}(q) \wedge \\ & \neg(p \equiv q) \wedge \texttt{msgSender}(m,p) \wedge \texttt{msgReceiver}(m,q) \wedge \texttt{Port}(t) \wedge \texttt{hasType}(q,t) \wedge \\ & \texttt{Part}(b) \wedge \texttt{AtomicComponent}(a) \wedge \texttt{hasType}(b,a) \wedge \\ & \texttt{hasPortVariable}(b,p) \wedge \texttt{hasPortVariable}(b,q) \rightarrow \\ & \exists i : \texttt{Interface}(i) \wedge (\texttt{requiredInterface}(t,i) \vee \texttt{providedInterface}(t,i)) \wedge \\ & \texttt{ifMsgCompatible}(i,m)^{1}\end{aligned} \quad {}^{1(6.5)}$$

Ebenfalls möglich sind Nachrichten, deren Sender auch dem Empfänger entspricht (Abb. 6.19e). Allerdings handelt es sich hierbei dann auch um eine Abstraktion von klassenbasierten Abläufen einer atomaren Komponente. Daher muss die betroffene Portvariable zu einem Part mit atomarem Komponententypen gehören und der Porttyp eine Schnittstelle mit zur Nachricht passender Signatur anbieten:

$$\begin{aligned}\mathbf{AC_{25}} := {} & \forall m \forall p \exists t \exists b : \texttt{Message}(m) \wedge \texttt{PortVariable}(p) \wedge \texttt{Port}(t) \wedge \texttt{hasType}(p,t) \wedge \\ & \texttt{msgReceiver}(m,p) \wedge \texttt{msgSender}(m,p) \wedge \texttt{Part}(b) \wedge \texttt{hasPortVariable}(b,p) \rightarrow \\ & \exists a \exists i : \texttt{AtomicComponent}(a) \wedge \texttt{hasType}(b,a) \wedge \texttt{Interface}(i) \wedge \\ & \texttt{providedInterface}(t,i) \wedge \texttt{ifMsgCompatible}(i,m)^{1}\end{aligned} \quad {}^{1(6.5)}$$

Zwischen Antwortnachrichten und den Signaturen der Porttypen der interagierenden Portvariablen besteht hingegen kein weiterer Zusammenhang. Allerdings sind Antwortnachrichten eng an Nachrichten gebunden und können daher nur auf denselben Wegen in umgekehrter Richtung

erfolgen, wie die zugehörigen Nachrichten.

$$\mathbf{AC_{26}} := \forall r \forall p \forall q : \texttt{ReturnMessage}(r) \wedge \texttt{PortVariable}(p) \wedge \texttt{PortVariable}(q) \wedge \texttt{msgSender}(r,p) \wedge \texttt{msgReceiver}(r,q) \rightarrow \exists m : \texttt{Message}(m) \wedge \texttt{hasReturnMsg}(m,r) \wedge \texttt{msgSender}(m,q) \wedge \texttt{msgReceiver}(m,p)$$

Nachrichtenvorkommen und ihre Reihenfolge

In der Verhaltensbeschreibung kann eine Nachricht oder Antwortnachricht beliebig oft auftreten. Jedes Auftreten wird dabei durch ein Nachrichtenvorkommen (MsgOccurence, siehe Abb. 6.18) der zugehörigen Nachricht oder Antwortnachricht (occurredMsg) dargestellt. Die Art der Nachrichtenübermittlung kann durch das Attribut msgType für jedes Nachrichtenvorkommen unabhängig von der jeweiligen Nachricht als synchron oder asynchron festgelegt werden. Ein Szenario fasst schließlich eine Menge von Nachrichtenvorkommen zusammen (containsMsgOccurrence) und ordnet (ordered) diese entsprechend ihrer Abfolge. Über die Eigenschaft nextMsgOccurrence kann der Vorgänger oder Nachfolger eines Nachrichtenvorkommens innerhalb des Szenarios bestimmt werden. Zwei Vorkommen, auf die sich die Eigenschaft nextMsgOccurrence bezieht, müssen sich innerhalb desselben Szenarios befinden:

$$\mathbf{AC_{27}} := \forall o \forall p : \texttt{MsgOccurrence}(o) \wedge \texttt{MsgOccurrence}(p) \wedge \texttt{nextMsgOccurrence}(o,p) \rightarrow \exists s : \texttt{Scenario}(s) \wedge \texttt{containsMsgOccurrence}(s,o) \wedge \texttt{containsMsgOccurrence}(s,p)$$

Basierend auf der Eigenschaft nextMsgOccurrence kann das transitive Prädikat nextMsgOccurrence* definiert werden, um auszudrücken, dass ein Nachrichtenvorkommen irgendwann nach einem anderen Nachrichtenvorkommen auftritt:

$$\texttt{nextMsgOccurrence}^*(a,b) := \texttt{nextMsgOccurrence}(a,b) \vee (\exists z : \texttt{nextMsgOccurrence}(a,z) \wedge \texttt{nextMsgOccurrence}^*(z,b)) \tag{6.6}$$

Wie oben erläutert, steht eine Antwortnachricht in einem direkten Bezug zu einer Nachricht. Dieser Zusammenhang spiegelt sich auch in der Folge der Nachrichtenvorkommen in einem Szenario wieder: Dem Vorkommen einer Nachricht mit zugeordneter Antwortnachricht muss irgendwann im Ablauf auch ein Vorkommen der Antwortnachricht folgen ($\mathbf{AC_{28}}$). Umgekehrt muss jedem Vorkommen einer Antwortnachricht irgendwann das Vorkommen einer Nachricht vorausgegangen sein ($\mathbf{AC_{29}}$):

$$\mathbf{AC_{28}} := \forall m \forall o \exists r : \texttt{Message}(m) \wedge \texttt{MsgOccurrence}(o) \wedge \texttt{occurredMsg}(o,m) \wedge \texttt{ReturnMessage}(r) \wedge \texttt{hasReturnMsg}(m,r) \rightarrow \exists p : \texttt{MsgOccurrence}(p) \wedge \texttt{occurredMsg}(p,r) \wedge \texttt{nextMsgOccurrence}^*(o,p)^{1(6.6)}$$

$$\mathbf{AC_{29}} := \forall r \forall p \exists m : \texttt{ReturnMessage}(r) \wedge \texttt{MsgOccurrence}(p) \wedge \texttt{occurredMsg}(p,r) \wedge \texttt{Message}(m) \wedge \texttt{hasReturnMsg}(m,r) \rightarrow \exists o : \texttt{MsgOccurrence}(o) \wedge \texttt{occurredMsg}(o,m) \wedge \texttt{nextMsgOccurrence}^*(o,p)^{1(6.6)}$$

Das Vorkommen von Nachrichten und zugehörigen Antwortnachrichten ist aneinander gekoppelt. Daher muss die Anzahl der jeweiligen Vorkommen innerhalb eines Szenarios übereinstim-

men:

$$\begin{aligned}\mathbf{AC_{30}} := {} & \forall s \forall m \forall r \exists u \exists v : \texttt{Scenario}(s) \wedge \texttt{Message}(m) \wedge \texttt{ReturnMessage}(r) \wedge \\ & \texttt{hasReturnMsg}(m,r) \wedge \texttt{count}((\texttt{MsgOccurrence}(o) \wedge \texttt{occurredMsg}(o,m) \wedge \\ & \qquad \texttt{constainsMsgOccurrence}(s,o)), u)^1 \wedge \\ & \texttt{count}((\texttt{MsgOccurrence}(p) \wedge \texttt{occurredMsg}(p,r) \wedge \\ & \qquad \texttt{constainsMsgOccurrence}(s,p)), v) \rightarrow \\ & (u \equiv v)\end{aligned} \quad {}^{1(6.21)}$$

Eine weitere Einschränkung bezüglich der Abfolge von Nachrichtenvorkommen in einem Szenario wird durch das Attribut msgType impliziert. Bei synchroner Kommunikation wartet der Sender auf die Abarbeitung der von ihm ausgelösten Interaktion und kann solange, bis diese ausgeführt ist, an keiner weiteren Interaktion teilnehmen [TS06]. Entsprechend muss nach einem synchronen Nachrichtenvorkommen einer Nachricht mit Antwortnachricht als nächste Interaktion mit Beteiligung der sendenden Portvariablen ein Vorkommen der Antwortnachricht folgen:

$$\begin{aligned}\mathbf{AC_{31}} := {} & \forall o \forall m \exists r \exists p : \texttt{MsgOccurrence}(o) \wedge \texttt{Message}(m) \wedge \texttt{occurredMessage}(o,m) \wedge \\ & \texttt{ReturnMessage}(r) \wedge \texttt{hasReturnMessage}(m,r) \wedge \texttt{msgType}(o,\texttt{"synchronous"}) \wedge \\ & \texttt{PortVariable}(p) \wedge \texttt{msgSender}(m,p) \rightarrow \\ & \exists t : \texttt{MsgOccurrence}(t) \wedge \texttt{occurredMsg}(t,r) \wedge \texttt{nextMsgOccAtPV}(o,t,p)^1\end{aligned} \quad {}^{1(6.7)}$$

Dabei ist das eben genutzte Prädikat $\texttt{nextMsgOccAtPV}(s,t,p)$ wahr, wenn bezogen auf die Portvariable p die beiden Vorkommen einer Nachricht oder Antwortnachricht s und t direkt aufeinander folgen. Die Portvariable p kann bei beiden zugehörigen Nachrichten oder Antwortnachrichten sowohl als Sender als auch als Empfänger auftreten.

$$\begin{aligned}\texttt{nextMsgOccAtPV}(s,t,p) := {} & \texttt{MsgOccurrence}(s) \wedge \texttt{MsgOccurrence}(t) \wedge \\ & \texttt{nextMsgOccurrence}^*(s,t)^1 \wedge \texttt{PortVariable}(p) \wedge \\ & \forall m \forall n : \texttt{AbstractMessage}(m) \wedge \texttt{occurredMsg}(s,m) \wedge \\ & (\texttt{msgSender}(m,p) \vee \texttt{msgReceiver}(m,p)) \wedge \texttt{AbstractMessage}(n) \wedge \\ & \texttt{occurredMsg}(t,n) \wedge (\texttt{msgSender}(n,p) \vee \texttt{msgReceiver}(n,p)) \wedge \\ & (\forall u \forall v : \texttt{AbstractMessage}(u) \wedge \texttt{MessageOccurrence}(v) \wedge \\ & \texttt{occurredMsg}(v,u) \wedge \neg(s \equiv v) \wedge \neg(t \equiv v) \wedge \\ & \texttt{nextMsgOccurrence}^*(s,v) \wedge \texttt{nextMsgOccurrence}^*(v,t) \rightarrow \\ & \neg(\texttt{msgSender}(v,p) \vee \texttt{msgReceiver}(v,p)))\end{aligned} \quad {}^{1(6.6)} \quad (6.7)$$

Häufig ist ein Vorkommen einer Nachricht oder Antwortnachricht dafür verantwortlich, dass ein anderes Vorkommen erfolgt. In der Abfolge eines Szenarios folgt das ausgelöste Vorkommen nicht zwangsläufig direkt auf das auslösende. Es können z.B. verschiedene komplexe Abläufe, die nacheinander ablaufen, durch ein Vorkommen ausgelöst werden. Die Betrachtung der Abfolge alleine ist daher nicht ausreichend, um Ursache und Wirkung abzubilden. Ein solches Zusammenwirken zweier Vorkommen wird durch triggersMsgOcc modelliert. Dabei kann es für ein Vorkommen immer maximal einen Auslöser geben, der aber mitunter beliebige viele Vorkommen auslösen kann. Damit ein Vorkommen Auslöser eines anderen sein kann, muss dieses vor dem ausgelösten erfolgen und der Empfänger des auslösenden muss Sender des ausgelösten sein:

$$\begin{aligned}\mathbf{AC_{32}} := {} & \forall o \forall p : \texttt{MsgOccurrence}(o) \wedge \texttt{MsgOccurrence}(p) \wedge \texttt{triggersMsgOcc}(o,p) \rightarrow \\ & \texttt{nextMsgOccurrence}^*(o,p)^1 \wedge \\ & \exists q : \texttt{PortVariable}(q) \wedge \texttt{msgOccReceiver}(o,q) \wedge \texttt{msgOccSender}(p,q)\end{aligned} \quad {}^{1(6.6)},\ {}^{2(6.9)},\ {}^{3(6.8)}$$

Die Prädikate `msgOccSender(o, p)` und `msgOccReceiver(o, p)` identifizieren, ob zu einem Nachrichtenvorkommen `o` eine Nachricht existiert, deren Sender bzw. Empfänger eine Portvariable `p` ist:

$$\begin{aligned}\texttt{msgOccSender}(\texttt{o},\texttt{p}) := \texttt{MessageOccurrence}(\texttt{o}) \wedge \texttt{PortVariable}(\texttt{p}) \wedge \\ \exists \texttt{m} : \texttt{AbstractMessage}(\texttt{m}) \wedge \texttt{occurredMsg}(\texttt{o},\texttt{m}) \wedge \texttt{msgSender}(\texttt{m},\texttt{p})\end{aligned} \tag{6.8}$$

$$\begin{aligned}\texttt{msgOccReceiver}(\texttt{o},\texttt{p}) := \texttt{MessageOccurrence}(\texttt{o}) \wedge \texttt{PortVariable}(\texttt{p}) \wedge \\ \exists \texttt{m} : \texttt{AbstractMessage}(\texttt{m}) \wedge \texttt{occurredMsg}(\texttt{o},\texttt{m}) \wedge \texttt{msgReceiver}(\texttt{m},\texttt{p})\end{aligned} \tag{6.9}$$

6.2.7. Beispiel: Verhalten in der Architekturbeschreibung

In diesem Abschnitt wird das Verhalten des Beispielsystems basierend auf dem im vorherigen Abschnitt erläuterten Metamodell modelliert. Die folgende Verhaltensbeschreibung beschränkt sich dabei mit einem einzelnen Szenario auf einen Ausschnitt des Systemverhaltens. Abbildung 6.20 zeigt dieses Szenario in Form eines Diagramms, dessen Notation an die von bekannten Diagrammarten zur Verhaltensbeschreibung (u.a. UML-Sequenzdiagramme [Obj11]) angelehnt ist.

Im oberen Bereich des Diagramms sind die Parts notiert, die in diesem Szenario miteinander interagieren. Zusätzlich sind die Portvariablen der Parts abgebildet; allerdings nur diejenigen, über die die dargestellten Interaktionen erfolgen. Der Kontext eines Szenarios ist nicht auf die Parts der Konfiguration einer Komponente oder eines Systems beschränkt. Interaktionen werden auch zwischen Parts aus unterschiedlichen Hierarchieebenen modelliert, solange diese über einen Delegationskonnektor verbunden sind. In Szenariodiagrammen werden Parts daher ähnlich wie in der Systemkonfiguration in Abbildung 6.17 in einer hierarchischen Whitebox-Sicht detailliert. Es werden nicht nur die Parts und ihre Portvariablen, sondern auch die Parts des jeweiligen Komponententyps dargestellt.

Die senkrechten gestrichelten Linien, die im Diagramm ausgehend von den Portvariablen eingezogen sind, repräsentieren die Portvariable an dem jeweiligen oberen Ende. Zwischen diesen Linien verlaufen die durch Pfeile dargestellten Vorkommen von Nachrichten oder Antwortnachrichten eines Szenarios, wobei die Pfeilspitze jeweils zum Empfänger zeigt. Start- und Endpunkt der Pfeile sind entsprechend des Senders bzw. Empfängers der zu dem Nachrichtenvorkommen zugehörigen Nachricht oder Antwortnachricht platziert. Die Symbole zur Kennzeichnung der Art eines Vorkommens sind analog zu denen in UML-Sequenzdiagrammen [Obj11] gewählt. Während Vorkommen von Nachrichten durch Pfeile mit einer durchgängigen Linie dargestellten werden, sind Vorkommen von Antwortnachrichten durch Pfeile mit einer unterbrochenen Linie gekennzeichnet. Zudem wird eine synchrone Interaktion durch eine ausgemalte Pfeilspitze ($\longrightarrow\!\!\blacktriangleright$) und eine asynchrone durch eine einfache Pfeilspitze ($\longrightarrow\!\!>$) markiert. Bei zwei Vorkommen, die direkt aufeinander folgen und bei denen der Empfänger des ersten mit dem Sender des zweiten übereinstimmt, wird implizit angenommen, dass das erste Vorkommen das zweite auslöst. Werden durch ein Vorkommen mehrere andere Vorkommen ausgelöst, wird dies durch einen grauen Balken dargestellt. Dieser wird entlang der senkrechten Linie eingezeichnet, die die Portvariable repräsentiert, die Empfänger des auslösenden Vorkommens ist. Er verläuft dabei von dem auslösenden Vorkommen bis zum letzten ausgelösten Vorkommen.

Das abgebildete Szenario beschreibt zunächst die Registrierung der beiden GUI-Parts kgui und mgui als Observer an dem Part kv zur Kundenverwaltung und die Registrierung von kv

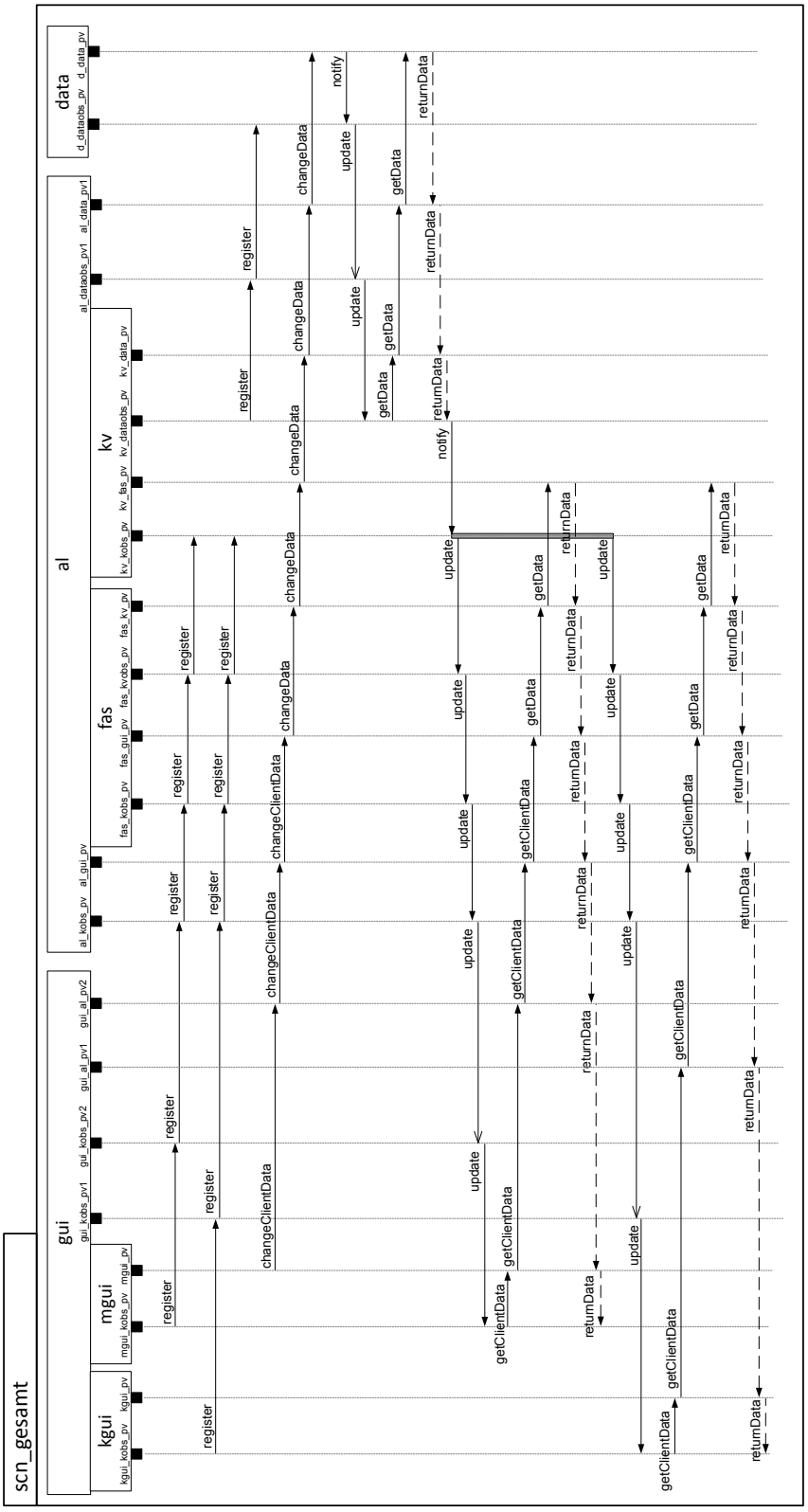


Abbildung 6.20: Szenario: Änderung von Kundendaten und Benachrichtigung der registrierten Observer.

wiederum als Observer an dem Part data zur Datenhaltung. Keine der Registrierungen erfolgt durch ein einzelnes Nachrichtenvorkommen, sondern jeweils durch eine Folge von Nachrichtenvorkommen entlang der vorgegebenen Interaktionswege (vgl. die Verbindungselemente in der Systemkonfiguration). So verläuft das erste Vorkommen einer register-Nachricht entlang eines Delegationskonnektors zwischen mgui_kobs_pv und gui_kobs_pv2 und das zweite entlang eines Konnektors zwischen gui_kobs_pv2 und al_kobs_pv. Das Vorkommen der register-Nachricht zwischen den Portvariablen fas_kobs_pv und fas_kvobs_pv verläuft hingegen nicht entlang eines Verbindungselements. Bei dem Typ Anwendungsfassade des Parts fas handelt es sich um eine atomare Komponente, deren interner Ablauf durch eine Nachricht abstrahiert wird.

Im Anschluss wird die Änderung von Kundendaten modelliert, die durch mgui initiiert wird. Dem Vorkommen einer changeClientData-Nachricht ausgehend von der Portvariable mgui_pv folgen weitere Vorkommen einer Nachricht jeweils gleichem Namens. Von dem Part fas wird nun ein Vorkommen einer anders benannten Nachricht an kv weitergereicht. Die unterschiedlichen Namen der Nachrichten weisen an dieser Stelle darauf hin, dass fas unter anderem für kv als Fassade agiert. Als Fassade versteckt dieser Part die Elemente des Fassadensubsystems sowie deren Schnittstellen und bietet stattdessen eine Schnittstelle mit ggf. deutlich verschiedenen Signaturen an.

Die Änderung der Daten in der Datenhaltung durch changeData löst das Vorkommen der notify-Nachricht zwischen d_data_pv und d_dataobs_pv aus. Diese löst wiederum die Benachrichtigung von kv als Observer von data über eine Kaskade von update-Vorkommen aus. Dieser Part aktualisiert sich durch das Initiieren eines Vorkommen der Nachricht getData. Der Erhalt der aktuellen Daten wird durch die Kaskade von Vorkommen von Antwortnachrichten returnData repräsentiert. Im Anschluss benachrichtigt kv seine beiden Observer kgui und mgui durch entsprechende Kaskaden von Nachrichtenvorkommen. Die beiden Observer fragen wiederum die geänderten Daten bei kv ab. Sowohl die Benachrichtigung von mgui als auch die von kgui wird durch das eine notify-Vorkommen im Architekturszenario ausgelöst. Entsprechend verläuft ein senkrechter grauer Balken von der Pfeilspitze des notify-Nachrichtenvorkommens bis zu dem Pfeil, der die zweite update-Kaskade zur Benachrichtigung von kgui startet.

6.3. Beschreibung von Architekturbausteinen

Ein Architekturbaustein wird beim bausteinbasierten Architekturentwurf als eine zusammenhängende Einheit betrachtet. Vergleichbar mit einer Schablone definiert er unter anderem Vorgaben für Struktur und Verhalten einer (Teil-)Architektur. Wesentliche Elemente eines Architekturbausteins sind seine Rollen, die bei der Anwendung eines Bausteins (=Bausteininstanziierung) einem oder mehreren Architekturelementen zugewiesen werden.

Im Folgenden wird der Metamodellteil für die Beschreibung von Architekturbausteinen vorgestellt. Dieses Metamodell ist von dem in Abschnitt 6.2 erläuterten Metamodell der Architekturbeschreibungssprache unabhängig. Das Metamodell zur Bausteinbeschreibung umfasst sowohl Elemente zur strukturellen Beschreibung (Abschnitt 6.3.1) als auch zur Verhaltensbeschreibung von Architekturbausteinen (Abschnitt 6.3.3). Die Abschnitte 6.3.2 und 6.3.4 illustrieren schließlich die Beschreibung von Architekturbausteinen anhand jeweils eines im Beispielsystem verwendeten Bausteins.

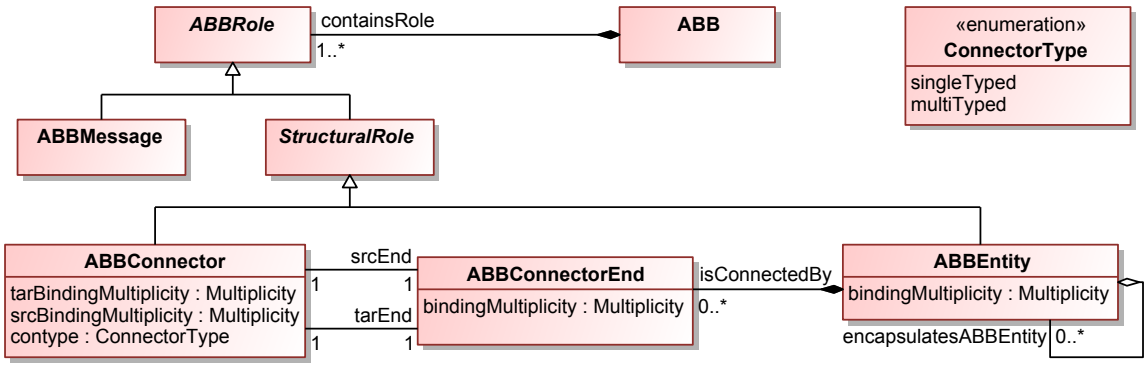


Abbildung 6.21: Rollen und Struktur eines Architekturbausteins.

6.3.1. Rollen und Struktur eines Architekturbausteins

Die zentralen Elemente eines Architekturbausteins (ABB) sind seine Rollen. Bei der Instanziierung eines Bausteins sind dies die Elemente, die an Architekturelemente gebunden werden und somit eine Entsprechung in der Architektur finden. Der Metamodellausschnitt in Abbildung 6.21 zeigt u.a. die Rollenhierarchie, die einem Baustein zugrunde liegt. Die Rollen eines Bausteins (ABBRole, containsRole) decken sowohl strukturelle als auch verhaltensbasierte Aspekte ab. Verhalten in Form einzelner Nachrichten wird durch Nachrichtenrollen (ABBMessage) spezifiziert, die zwischen den Entitätenrollen (ABBEntity) eines Baustein ausgetauscht werden. Enitätenrollen sind ebenfalls wie Konnektorrollen (ABBConnector) strukturelle Rollen (StructuralRole). Strukturelle Rollen und ihre Beziehungen zueinander bilden die Struktur eines Bausteins. Jede Konnektorrolle verbindet jeweils zwei Entitätenrollen miteinander. Als Anknüpfungspunkte der Konnektorrollen dienen Konnektorrollenenden (ABBConnectorEnd) an den jeweiligen Entitätenrollen (isConnectedBy). Über die beiden Bezeichner srcEnd und tarEnd können die beiden Enden einer Konnektorrolle unterschieden werden. Da eine Konnektorrolle zwei Entitätenrollen miteinander verbindet, dürfen deren Enden weder identisch sein noch zu ein und derselben Entitätenrolle gehören. Eine Konnektorrolle und die mit ihr indirekt verbundenen Entitätenrollen müssen zudem zu demselben Architekturbaustein gehören:

$$\begin{aligned}\mathbf{BC_1} := {} & \forall a \forall c \forall e \forall f \forall s \forall t : \texttt{ABB}(a) \wedge \texttt{ABBConnector}(c) \wedge \texttt{ABBConnectorEnd}(e) \wedge \\ & \texttt{srcEnd}(c,e) \wedge \texttt{ABBConnectorEnd}(f) \wedge \texttt{tarEnd}(c,f) \wedge \texttt{ABBEntity}(s) \wedge \\ & \texttt{isConnectedBy}(s,e) \wedge \texttt{ABBEntity}(t) \wedge \texttt{isConnectedBy}(t,f) \rightarrow \\ & \neg(e \equiv f) \wedge \neg(s \equiv t) \wedge \\ & \texttt{containsRole}(a,c) \wedge \texttt{containsRole}(a,s) \wedge \texttt{containsRole}(a,t)\end{aligned}$$

In Analogie zu den hierarchischen Strukturen in komponentenbasierten Systemen ist es auch möglich, Entitätenrollen ineinander zu schachteln. So kann eine Entitätenrolle beliebig viele Entitätenrollen enthalten (encapsulatesABBEntity), die untereinander oder mit der umgebenen Entitätenrolle wieder durch Konnektorrollen verbunden sein können. Innere Entitätenrollen müssen auch in irgendeiner Weise mit der umgebenen Entitätenrolle interagieren können. Daher wird gefordert, dass jede innere Entitätenrolle entweder direkt oder indirekt über andere innere Entitätenrollen mit der umgebenen Entitätenrolle durch eine Konnektorrolle verbunden ist:

$$\mathbf{BC_2} := \forall a \forall i : \texttt{ABBEntity}(a) \wedge \texttt{ABBEntity}(i) \wedge \texttt{encapsulatesABB}(a,i) \rightarrow \texttt{connectedABBEntities}^*(a,i)^{1} \qquad {}^{1}(6.11)$$

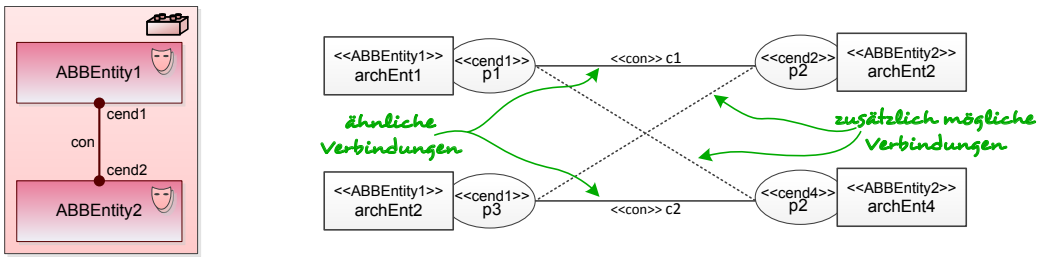


Abbildung 6.22: Ähnlichkeit von Verbindungen.

mit

$$\begin{aligned}&\texttt{connectedABBEntities}(x,y) := \texttt{ABBEntity}(x) \wedge \texttt{ABBEntity}(y) \wedge \\ &\quad \exists c : \texttt{ABBConnector}(c) \wedge \texttt{srcEnd}(c,a) \wedge \texttt{ABBConnectorEnd}(a) \wedge \texttt{tarEnd}(c,b) \wedge \\ &\quad \texttt{ABBConnectorEnd}(b) \wedge \texttt{isConnectedBy}(x,a) \wedge \texttt{isConnectedBy}(y,b)\end{aligned} \tag{6.10}$$

Die transitive Erweiterung von `connectedABBEntities` erfolgt durch die rekursive Definition:

[1](6.10)

$$\begin{aligned}&\texttt{connectedABBEntities}^*(x,y) := \texttt{connectedABBEntities}(x,y)^1 \vee \\ &\quad (\exists z : \texttt{connectedABBEntities}(x,z) \wedge \\ &\quad \texttt{connectedABBEntities}^*(z,y) \wedge \neg(x \equiv y))\end{aligned} \tag{6.11}$$

Je nach Architekturbaustein ist es möglich, dass im Zuge einer Bausteininstanziierung eine Entitätenrolle mehrfach gebunden wird. In der Architektur übernimmt folglich nicht nur ein einziges Architekturelement, sondern verschiedene Architekturelemente diese Rolle. Wie viele Bindungen für eine Entitätenrolle erlaubt sind, gibt das Attribut bindingMultiplicity der Klasse ABBEntity an. Im Zusammenspiel mit der Konnektorrolle ist die alleinige Angabe einer Multiplizität an der Entitätenrolle nicht ausreichend, um die möglichen Topologien von gültigen Instanziierungen eines Bausteins zu spezifizieren. Aus diesem Grund besitzen auch die Konnektorrollenenden ein Attribut bindingMultiplicity. Die in diesem Attribut hinterlegte Häufigkeit ist anders als bei Entitätenrollen definiert. Sie gilt nicht pro Bausteininstanz, sondern pro Architekturelement, das an die Entitätenrolle an diesem Konnektorrollenende gebunden ist. Zusätzlich besitzt auch die Konnektorrolle die beiden Multiplizitätsattribute (srcBindingMultiplicity und tarBindingMultiplicity), die wiederum im Kontext eines Konnektorrollenendes zu betrachten sind. Diese beiden Multiplizitäten sind den Multiplizitäten an Assoziationsenden in UML-Klassendiagrammen ähnlich. Sie legen fest, wie viele gebundene Konnektorrollenenden dem betrachteten gebundenem Konnektorrollenende gegenüberliegen und mit letzterem durch gebundene Konnektorrollen verbunden sind.

Des Weiteren besitzt eine Konnektorrolle das Attribut contype vom Typ ConnectorType, das die Werte singleTyped oder multiTyped annehmen kann. Dieses Attribut macht Aussagen über die Ähnlichkeit von architekturellen Verbindungselementen, an die eine Konnektorrolle im Rahmen einer Bausteininstanz gebunden wird. Die Ähnlichkeit von Verbindungselementen leitet sich aus der kreuzweisen Kompatibilität der Architekturelemente an ihren Enden ab. Wenn ein Verbindungselement an eine Konnektorrolle gebunden ist, sind seine Enden jeweils an eines der beiden Konnektorrollenenden der Konnektorrolle gebunden. Zwei oder mehr Verbindungselemente sind nun ähnlich, wenn jedes an das erste Konnektorrollenende gebundene Ende mit jedem an das zweite Konnektorrollenende gebundene Ende kompatibel und somit verbindbar ist. Ist eine Konnektorrolle als singleTyped definiert, müssen pro Bausteininstanz all jene Ver-

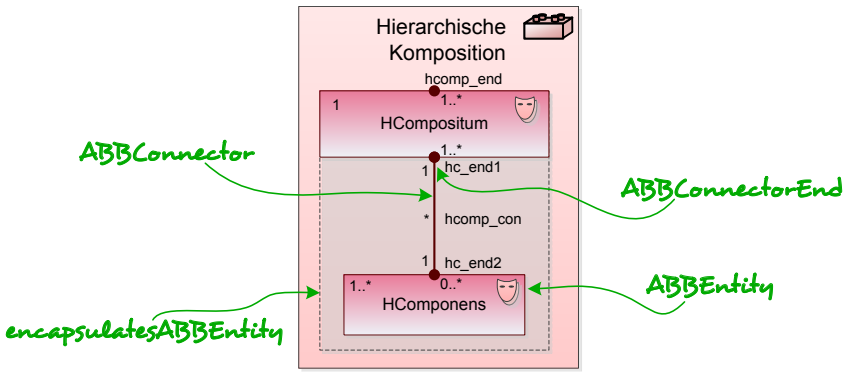


Abbildung 6.23: Beispiel für die graphische Strukturbeschreibung eines Architekturbausteins.

bindungselemente, an die diese Konnektorrolle gebunden ist, ähnlich zueinander sein. Ist eine Konnektorrolle hingegen multiTyped, so können diese Verbindungselemente auch unähnlich sein. Diese Unterscheidung ist notwendig, da es Fälle gibt, in denen die Enden der gebundenen Verbindungselemente typkompatibel sein müssen (siehe auch Beispiel in Abschnitt 6.3.2).

Abstrahieren lässt sich der Begriff der Ähnlichkeit mit Hilfe eines Graphen, in dem die Knoten als die Endpunkte der Verbindungselemente interpretiert werden. Die Menge der Knoten wird entsprechend des gebundenen Konnektorrollenendes in zwei Partitionen unterteilt. Zwischen zwei Knoten aus den beiden Partitionen wird im Graph genau dann eine Kante eingezogen, wenn diese Knoten kompatibel sind. Entsteht auf diese Weise ein vollständig bipartiter Graph, sind alle Endpunkte kreuzweise kompatibel und somit die Verbindungselemente ähnlich. Abbildung 6.22 veranschaulicht das folgende Beispiel. Angenommen, es existiert ein Architekturbaustein mit den Entitätenrollen ABBEntity1 und ABBEntity2, der Konnektorrolle con sowie den Konnektorrollenenden cend1 und cend2. In einer Instanz dieses Bausteins wird ABBEntity1 an die Architekturelemente archEnt1 und archEnt3 sowie ABBEntity2 an archEnt2 und archEnt4 gebunden. Entsprechend werden auch die Konnektorrollenenden cend1 und cend2 an p1 und p3 bzw. an p2 und p4 gebunden. archEnt1 und archEnt2 sind über p1 und p2 durch das Verbindungselement c1 und die Architekturelemente archEnt3 und archEnt4 über p3 und p4 durch c2 verbunden. Die beiden Verbindungselemente c1 und c2 sind dabei an con gebunden. c1 und c2 werden nun als ähnlich bezeichnet, wenn es möglich ist, archEnt1 mit archEnt4 über p1 und p4 sowie archEnt3 mit archEnt2 über p3 und p2 zu verbinden.

6.3.2. Beispiel: Strukturelle Bausteinbeschreibung

Im vorherigen Abschnitt wurde das Metamodell zur Beschreibung der Struktur von Architekturbausteinen beschrieben. Auf Basis dieses Metamodells werden die verschiedenen Architekturbausteine modelliert. Exemplarisch wird im Folgenden die Beschreibung des Bausteins Hierarchisches Komposition gezeigt (siehe Abbildung 6.23).

Generell wird ein Baustein durch einen Kasten mit dem Stereotypensymbol visualisiert. Innerhalb des Bausteinkastens stehen die Kästchen, die die Entitätenrollen des Architekturbausteins repräsentieren. Diese werden zusätzlich durch das Stereotypensymbol gekennzeichnet. In der linken oberen Ecke wird die Multiplizität der jeweiligen Entitätenrolle angezeigt. Konnektorrollen werden durch Linien zwischen den durch sie verbundenen Entitätenrollen dargestellt. An den Enden der Konnektorrollen sind ihre Multiplizitäten notiert. Hingegen gibt der Wert mittig an der Konnektorrolle an, ob es sich um eine singleTyped (1) oder multiTyped

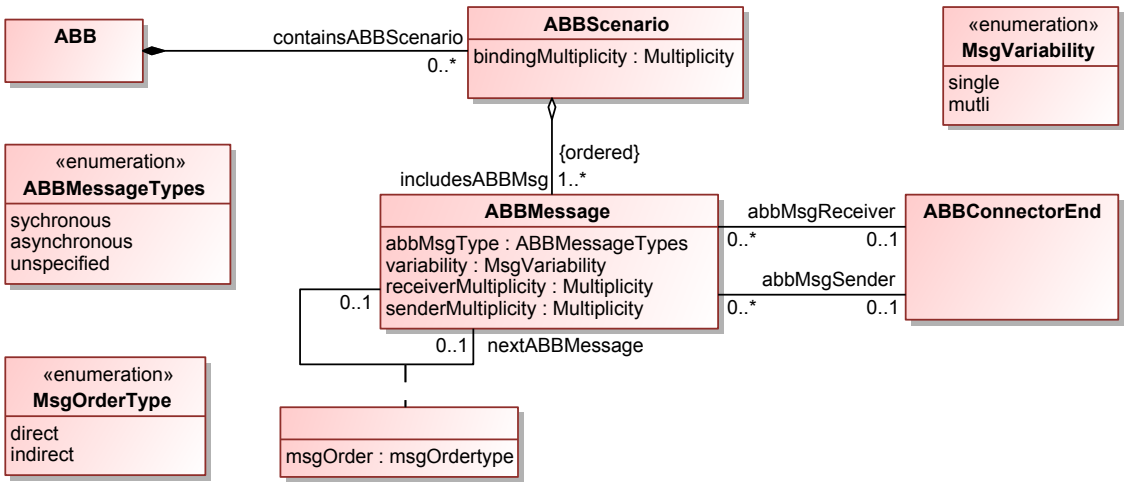


Abbildung 6.24: Metamodell der Verhaltensbeschreibung eines Architekturbausteins.

(*) Konnektorrolle handelt. Konnektorrollenenden werden durch die ausgefüllten Kreise an den Endpunkten von Konnektorrollen visualisiert. Der Wert ihres Multiplizitätsattributs findet sich innerhalb des Kastens für die Entitätenrolle am Ende der entsprechenden Konnektorrolle wieder.

Der in Abbildung 6.23 dargestellte Baustein der Hierarchischen Komposition besitzt die Entitätenrollen HCompositum und HComponens. Die Entitätenrolle HComponens ist in HCompositum geschachtelt. Dies ist durch den Kasten mit gestrichelten Kanten dargestellt, auf dessen Rand die Entitätenrolle HCompositum und in dessen Inneren HComponens platziert ist. Wie durch den entsprechenden Wert angegeben, unterscheidet sich die Bindungshäufigkeit der beiden Entitätenrollen. Während pro Bausteininstanz genau ein HCompositum gebunden wird, kann es beliebig viele Bindungen von HComponens geben. Die Multiplizitäten der Konnektorrollenenden werden im Kontext der Bindung der zugehörigen Entitätenrolle ausgewertet. So werden die beiden Konnektorrollenenden hc_end0 und hc_end1 an HCompositum beliebig häufig aber mindestens einmal gebunden. Indessen muss das Konnektorrollenende hc_end2 nicht im Kontext jeder Bindung von HComponens gebunden werden, wobei die Häufigkeit nach oben nicht beschränkt ist. An beiden Enden der Konnektorrolle ist eine Multiplizität von 1 angebenen. Somit kann jedes Element, das an das Konnektorrollenende hc_end1 oder hc_end2 gebunden ist, mit immer nur einem Element verbunden sein, das an die jeweils andere Rolle gebunden ist. Dabei brauchen die Verbindungen zwischen den gebundenen Konnektorrollenenden einer Bausteininstanz nicht ähnlich zueinander sein (multiTyped).

6.3.3. Verhalten eines Architekturbausteins

Das Verhalten eines Architekturbausteins wird ebenso wie seine Struktur basierend auf seinen Rollen beschrieben. Abbildung 6.24 zeigt das Metamodell, das in dieser Arbeit zur Beschreibung des Bausteinverhaltens verwendet wird. Ausgangspunkt sind die Nachrichtenrollen (ABBMessage), die die Interaktionen zwischen Entitätenrollen beschreiben. Die zwischen den Entitätenrollen ausgetauschten Botschaften entsprechen dabei dem Namen der jeweiligen Nachrichtenrolle. Modelliert werden als Sender (abbMsgSender) und Empfänger (abbMsgReceiver) allerdings nicht die Entitätenrollen selbst, sondern deren Konnektorrollenenden (ABBConnectorEnd). Vielmehr ist dasjenige Konnektorrollenende Sender oder Empfänger, das die Entitä-

tenrolle mit derjenigen Konnektorrolle verbindet, über die diese Interaktion stattfinden soll. Entsprechend können Nachrichtenrollen nur zwischen Konnektorrollenenden angegeben werden, die im selben Baustein über eine Konnektorrolle miteinander in Beziehung stehen. Zudem müssen Konnektorrolle und die übermittelte Nachrichtenrolle zu demselben Baustein gehören:

$$\begin{aligned}\mathbf{BC_3} := \forall m \forall s \forall t : &\texttt{ABBMessage}(m) \wedge \texttt{ABBConnectorEnd}(s) \\ &\texttt{ABBConnectorEnd}(t) \wedge \texttt{abbMsgSender}(m,s) \wedge \texttt{abbMsgReceiver}(m,t) \rightarrow \\ &\exists c \exists a : \texttt{ABBConnector}(c) \wedge ((\texttt{srcEnd}(c,s) \wedge \texttt{tarEnd}(c,t)) \vee \\ &\qquad\qquad (\texttt{srcEnd}(c,t) \wedge \texttt{tarEnd}(c,s))) \wedge \\ &\texttt{ABB}(a) \wedge \texttt{containsRole}(a,m) \wedge \texttt{containsRole}(a,c)\end{aligned}$$

Eine Nachrichtenrolle muss allerdings nicht gleichzeitig sowohl einen Sender als auch einen Empfänger besitzen (siehe Multiplizität 0..1). Eine der beiden Seiten kann undefiniert sein. In solchen Fällen ist es für den Baustein und sein Verhalten unerheblich, von wem die Nachricht gesendet wurde oder wer diese erhält. Allerdings ist es nicht erlaubt, dass sowohl Sender als auch Empfänger undefiniert sind. Zudem muss die Entitätenrolle, zu der das sendende oder empfangende Konnektorrollenende gehört, zu demselben Baustein gehören wie die ausgetauschte Nachrichtenrolle:

$$\begin{aligned}\mathbf{BC_4} := \forall m : &\texttt{ABBMessage}(m) \rightarrow \\ &\exists e \exists b \exists a : \texttt{ABBConnectorEnd}(e) \wedge (\texttt{abbMsgSender}(m,e) \vee \texttt{abbMsgReceiver}(m,e)) \wedge \\ &\texttt{ABBEntity}(b) \wedge \texttt{isConnectedBy}(b,e) \wedge \\ &\texttt{ABB}(a) \wedge \texttt{containsRole}(a,m) \wedge \texttt{containsRole}(a,b)\end{aligned}$$

Die Eigenschaft abbMsgType einer Nachrichtenrolle legt die Art des Nachrichtenaustauschs fest, auf die eine an sie gebundene Interaktion erfolgen soll. Zwei mögliche Werte sind asynchronous und synchronous. Darüber hinaus ist der Wert unspecified möglich, der die Art der Interaktion nicht festlegt und beide Möglichkeiten erlaubt.

Das Verhalten eines Architekturbausteins wird maßgeblich durch die Abfolge von Nachrichtenrollen festgelegt. Hierfür ist die Eigenschaften nextABBMessage verantwortlich, die jeder Nachrichtenrolle einen Nachfolger oder einen Vorgänger zuweist. Zwei über diese Eigenschaft verbundene Nachrichtenrollen müssen zum selben Baustein gehören:

$$\begin{aligned}\mathbf{BC_5} := \forall m \forall n : &\texttt{ABBMessage}(m) \wedge \texttt{ABBMessage}(n) \wedge \texttt{nextABBMessage}(m,n) \rightarrow \\ &\exists b : \texttt{ABB}(b) \wedge \texttt{containsRole}(b,m) \wedge \texttt{containsRole}(b,n)\end{aligned}$$

Eine Ordnung über den Nachrichtenrollen gibt zunächst nur eine zeitliche Abfolge an, macht aber keine Aussagen über die Beziehung zwischen zwei Nachrichtenrollen. Wenn in einem Baustein festgelegt ist, dass eine Nachrichtenrolle b auf eine Nachrichtenrolle a folgt, kann dies auf zwei Arten interpretiert werden: Zum einen kann zwischen a und b ein direkter kausaler Zusammenhang bestehen, indem a der Auslöser für b ist. Im Fall einer Instanziierung des Bausteins muss ein solcher Zusammenhang auch für die gebundenen Nachrichtenrollen gelten. Zum anderen kann die Reihenfolge lediglich bedeuten, dass b irgendwann auf a folgt. Wenn der Baustein und sein Verhalten instanziiert werden, kann dann zwischen den gebundenen Nachrichtenrollen beliebiges geschehen. Modelliert werden diese beiden unterschiedlichen Beziehungen zwischen Nachrichtenrollen durch eine zusätzliche Eigenschaft im Zusammenhang mit der Definition der Abfolge (msgOrder). Diese Eigenschaft kann einen der beiden Werte direct für den direkten und auslösenden Zusammenhang und indirect für den losen Zusammenhang annehmen. Vollständig

wird somit eine Abfolge von Nachrichtenrollen durch das Triple aus Nachrichtenrolle, Nachfolger und Art der Abfolge abgebildet. Das dazu verwendete Prädikat nextABBMessage/3 wird durch ein zweistelliges abgekürzt, wenn im betrachteten Zusammenhang die Art der Abfolge nicht relevant ist:

$$\texttt{nextABBMessage}(\texttt{a},\texttt{b}) := \exists \texttt{c} : \texttt{nextABBMessage}(\texttt{a},\texttt{b},\texttt{c}) \tag{6.12}$$

Wie erläutert, werden Konnektorrollenenden als Sender und Empfänger von Nachrichtenrollen modelliert. Sind zwei Nachrichtenrollen als direkt aufeinanderfolgend definiert, müssen der Empfänger der ersten und der Sender der zweiten zu derselben Entitätenrolle gehören:

$$\begin{aligned}\mathbf{BC_6} := {} & \forall \texttt{m} \forall \texttt{n} \exists \texttt{s} \exists \texttt{t} : \texttt{ABBMessage}(\texttt{m}) \wedge \texttt{ABBMessage}(\texttt{n}) \wedge \\ & \texttt{nextABBMessage}(\texttt{m},\texttt{n},\texttt{"direct"}) \wedge \texttt{ABBConnectorEnd}(\texttt{s}) \wedge \\ & \texttt{abbMsgReceiver}(\texttt{m},\texttt{s}) \wedge \texttt{ABBConnectorEnd}(\texttt{t}) \wedge \texttt{abbMsgSender}(\texttt{m},\texttt{t}) \rightarrow \\ & \exists \texttt{e} : \texttt{ABBEntity}(\texttt{e}) \wedge \texttt{isConnectedBy}(\texttt{e},\texttt{s}) \wedge \texttt{isConnectedBy}(\texttt{e},\texttt{t})\end{aligned}$$

Die verschiedenen Elemente eines Bausteins werden in der Beschreibung in der Einzahl betrachtet. Dies geschieht unabhängig davon, wie oft diese gebunden werden dürfen. So ist z.B. *ein* Konnektorrollenende Sender *einer* Nachrichtenrolle, auch wenn dieses Konnektorrollenende bei der Instanziierung des zugehörigen Bausteins mehrfach gebunden wird. Es könnten sowohl alle Architekturelemente, die an das Konnektorrollenende gebundenen sind, Ausgangspunkt einer an die Nachrichtenrolle gebundenen Interaktion sein oder auch nur eines. Um ausdrücken zu können, wie viele gebundene Sender als auch Empfänger eine gebundene Nachrichtenrolle aufweisen kann, existieren die beiden Attribute senderMultiplicity und receiverMultiplicity.

Eine Nachrichtenrolle kann Stellvertreter für unterschiedliche Interaktionen sein. In manchen Bausteinen wird mit einer Nachrichtenrolle eine konkrete Botschaft assoziiert. Interaktionen, die an eine solche Nachrichtenrolle gebunden werden, sollten in ihrem Verhalten untereinander äquivalent sein. Um Aussagen über die Verhaltensäquivalenz zu machen, besitzt eine Nachrichtenrolle das Attribut variability vom Typ MsgVariability. Durch seine Werte gibt die Nachrichtenvariabilität an, ob die gebundenen Interaktionen eine Verhaltensäquivalenz aufweisen müssen (single) oder sich unterscheiden können (multi). Ein Anhaltspunkt für Verhaltensäquivalenz kann z.B. durch die Identität von ausgetauschten Botschaften gegeben sein.

Die Nachrichtenrollen eines Bausteins werden schließlich durch ein Bausteinszenario (ABBScenario) gebündelt (includesABBMsg). Ein Bausteinszenario repräsentiert die Verhaltensbeschreibung eines Architekturbausteins, vom dem jeder Baustein beliebig viele enthalten kann (containsABBScenario). Da ein Bausteinszenario auf den zugehörigen Baustein begrenzt ist, muss jede Nachrichtenrolle, die in einem Bausteinszenario auftritt, auch zu demselben Baustein wie das Szenario gehören:

$$\begin{aligned}\mathbf{BC_7} := {} & \forall \texttt{m} \forall \texttt{s} : \texttt{ABBMessage}(\texttt{m}) \wedge \texttt{ABBScenario}(\texttt{s}) \wedge \texttt{includesABBMsg}(\texttt{s},\texttt{m}) \rightarrow \\ & \exists \texttt{b} : \texttt{ABB}(\texttt{b}) \wedge \texttt{containsABBScenario}(\texttt{b},\texttt{s}) \wedge \texttt{containsRole}(\texttt{b},\texttt{m})\end{aligned}$$

Eine Nachrichtenrolle, die in keinem Bausteinszenario enthalten ist, verletzt zwar $\mathbf{BC_7}$ nicht, kann aber auf diese Weise nicht zu dem Bausteinverhalten beitragen. Daher wird außerdem gefordert, dass jede Nachrichtenrolle eines Bausteins auch in einem Bausteinszenario vorkommt:

$$\begin{aligned}\mathbf{BC_8} := {} & \forall \texttt{m} \forall \texttt{b} : \texttt{ABBMessage}(\texttt{m}) \wedge \texttt{ABB}(\texttt{b}) \wedge \texttt{includesRole}(\texttt{b},\texttt{m}) \rightarrow \\ & \exists \texttt{s} : \texttt{ABBScenario}(\texttt{s}) \wedge \texttt{containsABBScenario}(\texttt{b},\texttt{s}) \wedge \texttt{includesABBMsg}(\texttt{s},\texttt{m})\end{aligned}$$

Solange eine Nachrichtenrolle nicht die letzte oder die erste in der Abfolge eines Bausteinszenarios ist, existiert immer auch ein Nachfolger bzw. ein Vorgänger. Dies impliziert, dass es pro Bausteinszenario auch jeweils nur genau eine erste und genau eine letzte Nachrichtenrolle in einer Abfolge geben kann. Entsprechend stehen zwei unterschiedliche Nachrichtenrollen in einer transitiven Nachfolgebeziehung zueinander:

$$\begin{aligned}\mathbf{BC_9} := \forall s \forall m \forall n : &\texttt{ABBScenario}(s) \wedge \texttt{ABBMessage}(m) \wedge \texttt{ABBMessage}(n) \wedge \\ &\texttt{includesABBMsg}(s,m) \wedge \texttt{includesABBMsg}(s,n) \wedge \neg(m \equiv n) \rightarrow \\ &\texttt{nextABBMessage}^{\star}(m,n)^{1} \vee \texttt{nextABBMessage}^{\star}(n,m)\end{aligned} \quad {}^{1}(6.13)$$

Dabei ist $\texttt{nextABBMessage}^{\star}(m,n)$ als transitive Variante des Prädikats $\texttt{nextABBMessage}(m,n)$ definiert und identifiziert, ob eine Nachrichtenrolle n irgendwann nach einer Nachrichtenrolle m in der Abfolge auftritt:

$$\begin{aligned}\texttt{nextABBMessage}^{\star}(m,n) := &\texttt{ABBMessage}(m) \wedge \texttt{ABBMessage}(n) \wedge \\ &(\texttt{nextABBMessage}(m,n) \vee \\ &(\exists x : \texttt{nextABBMessage}(m,x) \wedge \texttt{nextABBMessage}^{\star}(x,n)))\end{aligned} \quad (6.13)$$

Wie bei einigen strukturellen Elementen eines Architekturbausteins kann auch für ausgewählte verhaltensbezogene Elemente bestimmt werden, wie oft diese im Fall einer Instanziierung des Bausteins gebunden werden dürfen. Dies wird bereits bei einem Bausteinszenario als gröbsten modellierten Verhaltensaspekt durch das Attribut bindingMultiplicity festgelegt. Bei einer mehrwertigen Multiplizität tritt das durch das Bausteinszenario modellierte Verhaltensmuster entsprechend häufig in der Verhaltensbeschreibung der Architektur auf. Die Häufigkeit, mit der eine Nachrichtenrolle gebunden wird, hängt zunächst von der Bindungshäufigkeit des Bausteinszenarios ab, das diese Nachrichtenrolle enthält. In dessen Bindungskontext gelten die bereits erläuterten Attribute für die Sender- und Empfängerhäufigkeit (senderMultiplicity bzw. receiverMultiplicity) einer Nachrichtenrolle. Diese legen fest, wie viele der gebundenen Konnektorrollenenden im Kontext einer gebundenen Nachrichtenrolle als Sender oder Empfänger auftreten. Wie häufig eine Nachrichtenrolle gebunden wird, wird nicht explizit festgelegt. Die Bindungshäufigkeit der Nachrichtenrolle ergibt sich aus der Bindungshäufigkeit von Sender und Empfänger.

6.3.4. Beispiel: Verhalten eines Architekturbausteins

Im vorangegangenen Abschnitt wurde das Metamodell für die Verhaltensbeschreibung eines Architekturbausteins eingeführt. Dessen Verwendung wird in diesem Abschnitt nun anhand der Verhaltensbeschreibung eines Architekturbausteins des Beispielsystems vorgestellt. Dabei wird das Verhalten mit Hilfe einer grafischen Notation visualisiert.

Die graphische Darstellung (vgl. Abb. 6.25) ähnelt auf den ersten Blick der Darstellung des Architekturverhaltens, wobei einige Unterschiede bestehen. Die Entitätenrollen sind mit dem Stereotypensymbol  versehen und am oberen Rand angeordnet. An ihnen befinden sich die Konnektorrollenenden, die ausgehend von • durch senkrechte Linien repräsentiert werden. Zwischen diesen Linien sind die Nachrichtenrollen als Pfeile dargestellt. Eine ausgemalte Pfeilspitze ($\longrightarrow\!\!\blacktriangleright$) visualisiert dabei eine synchrone und eine einfache Pfeilspitze ($\longrightarrow$) eine asynchrone Nachrichtenrolle. Ist die Nachrichtenart nicht festgelegt (unspecified), wird die Pfeilspitze nicht ausgemalt ($\longrightarrow\!\!\triangleright$). Ein unbekannter Sender oder Empfänger wird an der entsprechenden Seite des Pfeils durch einen ausgemalten Kreis symbolisiert. Am Anfang und am Ende eines Pfeils

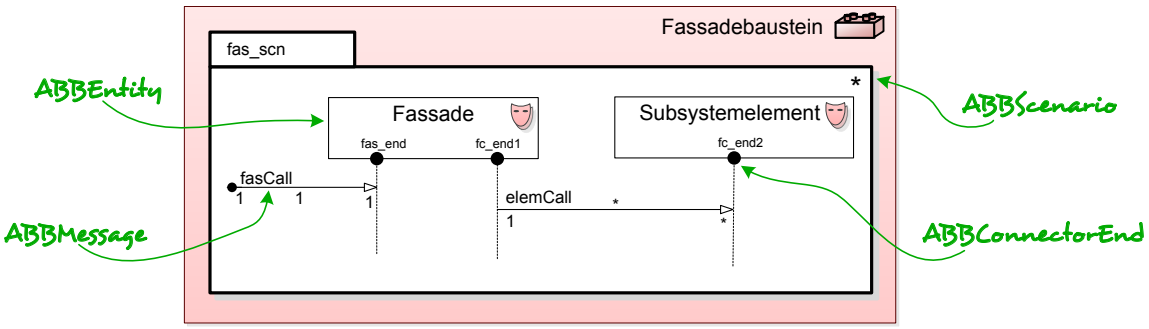


Abbildung 6.25: Beispiel für die graphische Verhaltensbeschreibung eines Architekturbausteins.

sind Sender- bzw. Empfängermultiplizität notiert, während der Wert in der Mitte eines Pfeils die Nachrichtenvariabilität angibt.

Als Beispiel wird hier die Verhaltensbeschreibung des Fassadebausteins betrachtet. Im Fassadebaustein kapselt die Fassade die Funktionalität der Elemente des Fassadensubsystems. Interaktionen, die von dritten ausgehen, erfolgen zunächst mit der Fassade und werden von dieser an die entsprechenden Subsystemelemente weitergereicht. Das Bausteinszenario in Abbildung 6.25 beschreibt dieses abstrakte Verhalten. Die initiale Interaktion ist durch die Nachrichtenrolle fasCall dargestellt, die ausgehend von einer unbekannten Quelle am Konnektorrollenende fas_end der Entitätenrolle Fassade eingeht. Pro gebundenem Bausteinszenario kann es nur je einen Empfänger und einen Sender dieser Nachrichtenrolle geben (1 an den Pfeilenden). Implizit kann diese Nachrichtenrolle dadurch auch nur an eine Interaktion im Kontext des Bausteinszenarios gebunden werden. Als Folge von fasCall erfolgt ausgehend von fc_end1 die Nachrichtenrolle elemCall mit Ziel fc_con2 an der Entitätenrolle Subsystemelement. Diese Nachrichtenrolle kann pro gebundenem Bausteinszenario an Interaktionen mit verschiedenen Empfängern gebunden werden (* an der Pfeilspitze). Folglich hängt die Bindungshäufigkeit dieser Nachrichtenrolle von der Anzahl der Empfänger ab. Zudem können sich diese Interaktionen in Bezug auf die Verhaltensäquivalenz unterscheiden (Eigenschaft multi durch * an der Pfeilmitte).

6.4. Integrationsbeschreibung

In den vorherigen Abschnitten wurden Metamodelle für die Beschreibung sowohl von Architekturbausteinen als auch von Architekturen definiert. Baustein- und Architekturbeschreibungen stehen als Modelle auf Basis dieser beiden Metamodelle durch explizit notierte Bausteininstanzen in Beziehung. Die Verwaltung der Bausteininstanzen und der in ihrem Kontext erfolgten Bindungen von Bausteinrollen an Architekturelemente ist Aufgabe des Metamodells der Integrationsbeschreibung. Es stellt die notwendigen Beschreibungsmittel zur Verfügung, um sowohl strukturelle als auch verhaltensbasierte Baustein- und Architekturelemente aufeinander abbilden zu können (Abschnitt 6.4.1). Ergänzt wird das zunächst durch Klassendiagramme definierte Metamodell durch zahlreiche Bedingungen, um die syntaktische Korrektheit eines auf dem Metamodell basierenden Modells sicherzustellen (Abschnitt 6.4.2). Veranschaulicht wird die Verwendung des Metamodells anhand der Bausteininstanziierungen (Abschnitt 6.4.3), die im Rahmen der Entwicklung des Beispielsystems erfolgen.

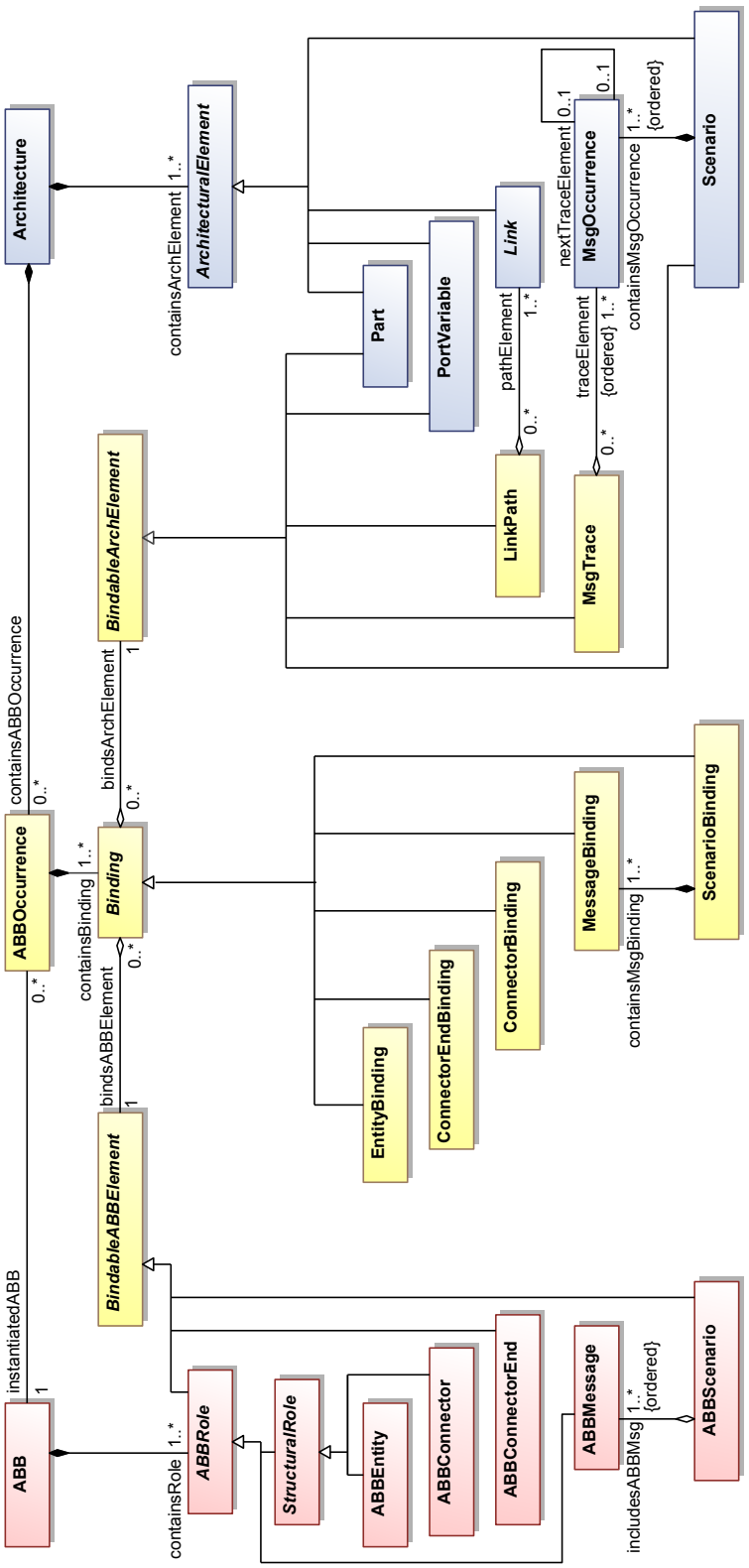


Abbildung 6.26: Integrationsteil des Metamodells (gelbe Elemente) und dessen Einbettung zwischen den Teilmetamodellen für die Baustein- (rote Elemente) und Architekturbeschreibung (blaue Elemente).

6.4.1. Metamodell der Integrationsbeschreibung

Das der Integrationsbeschreibung zugrunde liegende Metamodell definiert Elemente mit unterschiedlicher Zielsetzung. Es enthält sowohl Elemente für die Beschreibung von Bindungen als auch für die Zusammenfassung der in einer Instanziierung erfolgten Bindungen. Auf diese Weise verbindet es die Metamodelle von Architektur und Baustein (siehe Abbildung 6.26). Beim bausteinbasierten Architekturentwurf wird eine Architektur (Architecture) aus verschiedenen Bausteinen (ABB) durch Anwenden der Instanzkomposition zusammengefügt. Pro Instanziierungsvorgang wird genau eine Instanz (ABBOccurence) genau eines Architekturbausteins (instantiatedABB) erstellt. Dabei kann ein Baustein jedoch an insgesamt beliebig vielen Instanziierungsvorgängen beteiligt sein. Eine Architektur besteht so aus der Komposition einer Menge von Instanzen (containsABBOccurrence) beliebig vieler Bausteine. Allerdings ist die Instanziierung eines Bausteins architekturspezifisch, d.h., dass eine konkrete Bausteininstanz nur in genau einer Architektur (Architecture) integriert sein kann.

Im Rahmen einer Bausteininstanziierung werden die bindbaren Elemente des Bausteins (BindableABBElement) verschiedenen Architekturelementen zugeordnet. Neben den Rollen eines Bausteins (ABBRole) können weitere Teile eines Baustein als bindbar deklariert werden. Die Zuordnungen zwischen Baustein- und Architekturelementen werden durch Bindungen (Binding) verwaltet (bindsABBElement, bindsArchElement). Bindungen werden wiederum durch Bausteinvorkommen ABBOccurrence zusammengefasst (containsBinding), die jeweils eine Bausteininstanz repräsentieren. In ihrer Gesamtheit erlauben diese Elemente der Integrationsbeschreibung die Nachverfolgbarkeit einer Bausteininstanziierung inklusive aller Rollenbindungen. Bindbare Bausteinelemente können allerdings nicht an beliebige Architekturelemente gebunden werden. So ist zunächst die Menge der Typen von Architekturelementen, die an Bindungen partizipieren darf, eingeschränkt. Zusammengefasst wird sie unter dem abstrakten Element BindableArchElement. Des Weiteren werden verschiedene Arten von Bindungen als Spezialisierung des abstrakten Obertypen Binding definiert. Diese legen fest, welche Typen von bindbaren Bausteinelementen an welche Typen von bindbaren Architekturelementen gebunden werden können. Darüber hinaus werden bezüglich der Bindungen und der durch sie einander zugeordneten Elemente einige Einschränkungen vorgenommen. So sind an jeder Bindung nur genau ein bindbares Bausteinelement und auch nur genau ein bindbares Architekturelement beteiligt. Zudem darf ein Paar aus bindbarem Bausteinelement und bindbarem Architekturelement nur maximal ein Mal pro Bausteininstanz aneinander gebunden werden.

Im vorliegenden Metamodell zur Integrationsbeschreibung werden verschiedene Arten von Bindungen unterschieden. EntityBinding bindet Entitätenrollen (ABBEntity) eines Bausteins an die Parts, die diese Rollen in der Architektur übernehmen sollen. Dabei kann eine Entitätenrolle pro Bausteininstanz an so vielen Bindungen beteiligt sein, wie dies durch ihre Multiplizität angegeben ist. Hingegen wird eine Konnektorrolle (ABBConnector) durch eine Konnektorbindung (ConnectorBinding) nicht an ein einzelnes Architekturelement, sondern an eine Menge von Verbindungen (Link) gebunden. Diese Menge von Verbindungen wird durch das kumulierende Integrationselement Verbindungspfad (LinkPath, pathElement) repräsentiert. Ein Verbindungspfad kann sowohl aus einem einzelnen Verbindungselement, wie einem Konnektor oder einem Delegationskonnektor, bestehen oder auch eine komplexere Struktur besitzen. Dabei muss er nicht zwingend aus einer linearen Abfolge von Verbindungselementen bestehen, sondern kann auch Verzweigungen enthalten. An den Enden eines solchen Verbindungspfads stehen Parts, an die jeweils eine der Entitätenrollen gebunden sein muss, die durch die betrachtete Konnektorrolle im Baustein verbunden wurden. Die Endpunkte eines Verbindungspfades sind Portvaria-

blen, die durch Konnektorrollenendbindungen (ConnectorEndBinding) an Konnektorrollenenden (ABBConnectorEnd) gebunden sind. Auch hier müssen bei der Bindung die im Baustein definierten Strukturen erhalten bleiben. So kann ein Konnektorrollenende einer Konnektorrolle nur an eine Portvariable gebunden werden, die sich am Ende eines an die Konnektorrolle gebundenen Verbindungspfades befindet. Und weiterhin kann diese Portvariable nur zu einem Part gehören, der an die Entitätenrolle an dem Konnektorrollenende gebunden ist.

Im Baustein ist der Weg, über den eine Nachrichtenrolle (ABBMessage) weitergereicht werden kann, durch eine Konnektorrolle vorgegeben. Auf Architekturseite ist eine Konnektorrolle an einen Verbindungspfad aus beliebig vielen Verbindungselementen gebunden. Eine Interaktion, an die eine Nachrichtenrolle gebunden wird (MessageBinding), erfolgt entsprechend entlang eines solchen Verbindungspfads. Diese Interaktion ist ähnlich wie der Verbindungspfad unterteilt. Pro Element des Verbindungspfads erfolgt nur die Teilinteraktion, die zwischen den durch das Verbindungselement verbundenen Parts stattfindet. Daher wird eine Nachrichtenrolle ebenfalls wie eine Konnektorrolle an ein kumulierendes Integrationselement gebunden. Bei diesem kumulierenden Integrationselement handelt es sich um eine Nachrichtenspur (MsgTrace). Eine Nachrichtenspur repräsentiert (traceElement) eine geordnete Menge von Nachrichtenvorkommen (MsgOccurrence). Über diese Menge aus Nachrichtenvorkommen kann mit Hilfe von nextTraceElement iteriert werden. Eigenschaften einer Nachrichtenrolle, die im Baustein festgelegt wurden, müssen bei ihrer Bindung beachtet werden. So kann z.B. im Baustein die Art einer Nachrichtenrolle (synchron oder asynchron) festgelegt werden. In einer Nachrichtenspur, die an diese Nachrichtenrolle gebunden ist, muss die Art jedes Nachrichtenvorkommens dazu konform sein. Auch müssen die Konnektorrollenenden, an die die Endpunkte einer Nachrichtenspur gebunden sind, mit Sender und Empfänger der Nachrichtenrolle übereinstimmen.

Ebenso wichtig für die Verhaltensbeschreibung wie die Nachrichtenrollen an sich, ist ihr durch ein Bausteinszenario (ABBScenario) definierter Kontext. Ein Bausteinszenario wird im Rahmen einer Bausteininstanziierung durch eine Szenariobindung (ScenarioBinding) an ein Architekturszenario (Scenario) gebunden. Die Bindung eines Bausteinszenarios an ein Architekturszenario führt nicht dazu, dass jedes Nachrichtenvorkommen des Architekturszenarios gebunden wird. Vielmehr kann ein gebundenes Architekturszenario auch Nachrichtenvorkommen besitzen, die zu keiner Nachrichtenspur zusammengefasst werden und somit nicht an eine Nachrichtenrolle gebunden sind. Auf der anderen Seite müssen alle Nachrichtenrollen des Bausteinszenarios innerhalb eines Architekturszenarios gebunden sein. Dabei müssen sie u.a. auch die im Baustein festgelegte Reihenfolge einhalten. Die Nachrichtenbindungen, die im Kontext einer Szenariobindung erfolgen, sind dieser Szenariobindung zugeordnet (containsMsgBinding). So sind auch in dem Fall, in dem ein Bausteinszenario mehrmals an ein Architekturszenario gebunden ist, die zusammengehörigen Nachrichtenbindungen identifizierbar.

Die Trennung von Architektur- und Bausteinbeschreibung durch einen separaten Integrationsteil entkoppelt diese beiden Beschreibungen voneinander. Diese Unabhängigkeit erlaubt es, die Architektur- oder Bausteinsprache einfacher durch andere zu ersetzen. Änderungen der einen Beschreibungssprache haben keine Auswirkungen auf die jeweils andere Beschreibungssprache. Ansatzpunkt für den Austausch sind jeweils die generischen Typen BindableABBElement und BindableArchElement, die unabhängig von der eingesetzten Sprache bestehen bleiben. Angepasst werden müssen hingegen die erlaubten Bindungspartner der konkreten Bindungen und die jeweiligen Einschränkungen. Allerdings ist es für die Ausgestaltung der konkreten Bindungen notwendig, dass die verwendeten Sprachen jeweils semantisch kompatible Elemente zu den hier vorgestellten bindbaren Bausteinelementen bzw. bindbaren Architekturelementen aufweisen.

6.4.2. Konsistenzbedingungen der Integrationsbeschreibung

Das in Abbildung 6.26 dargestellte Metamodell der Integrationsbeschreibung zeigt verschiedene Beziehungen zwischen den einzelnen Metamodellklassen durch Assoziationen und Multiplizitäten auf. Diese und weitere Informationen, die in Klassendiagrammen darstellbar sind, reichen in einigen Fällen nicht aus, um die Anforderungen an eine gültige Instanz dieses Metamodells auszudrücken. Daher wird das Metamodell um Bedingungen ergänzt, die im Folgenden durch prädikatenlogische Ausdrücke formuliert werden. Für ein gültiges Modell sind alle der in diesem Abschnitt erläuterten und definierten Bedingungen $\mathbf{IC}_i$ zu erfüllen.

Für die Formulierung der Bedingungen werden verschiedene Hilfsprädikate mit unterschiedlichem Charakter eingeführt. Einige bieten eine abkürzende Schreibweise für eine prädikatenlogische Formel an während andere benötigte mathematische Konzepte abbilden. Als abkürzende Prädikate werden die folgenden verwendet.

$\texttt{isBoundToBy}(\texttt{b},\texttt{a},\texttt{e})$ führt die Aussage, dass ein Bindungselement $\texttt{e}$ ein bindbares Bausteinelement $\texttt{b}$ mit einem bindbaren Architekturelement $\texttt{a}$ verbindet, in einem Prädikat zusammen:

$$\begin{aligned}\texttt{isBoundToBy}(\texttt{b},\texttt{a},\texttt{e}) := \texttt{BindableABBElement}(\texttt{b}) \land \texttt{BinableArchElement}(\texttt{a}) \land \\ \texttt{Binding}(\texttt{e}) \land \texttt{bindsABBElement}(\texttt{e},\texttt{b}) \land \texttt{bindsArchElement}(\texttt{e},\texttt{a})\end{aligned} \tag{6.14}$$

$\texttt{isBoundTo}(\texttt{b},\texttt{a})$ ist wahr, wenn in einer beliebigen Bausteininstanz ein Bausteinelement $\texttt{b}$ an ein Architekturelement $\texttt{a}$ gebunden ist:

$$\begin{aligned}\texttt{isBoundTo}(\texttt{b},\texttt{a}) := \texttt{BindableABBElement}(\texttt{b}) \land \texttt{BindableArchElement}(\texttt{a}) \land \\ \exists \texttt{e} : \texttt{Binding}(\texttt{e}) \land \texttt{isBoundToBy}(\texttt{b},\texttt{a},\texttt{e})\end{aligned} \tag{6.15}$$

$\texttt{isBoundToByInOccurrence}(\texttt{b},\texttt{a},\texttt{e},\texttt{x})$ erweitert das Prädikat aus 6.14 um einen Parameter $\texttt{x}$ für die Bausteininstanz der Bindung und ist wahr, wenn ein bindbares Bausteinelement $\texttt{b}$ durch das Bindungselement $\texttt{e}$ an das bindbare Architekturelement $\texttt{a}$ in der Bausteininstanz $\texttt{x}$ gebunden wird:

$$\begin{aligned}\texttt{isBoundToByInOccurrence}(\texttt{b},\texttt{a},\texttt{e},\texttt{x}) := \\ \texttt{BindableABBElement}(\texttt{b}) \land \texttt{BindableArchElement}(\texttt{a}) \land \texttt{Binding}(\texttt{e}) \land \\ \texttt{ABBOccurence}(\texttt{x}) \land \texttt{isBoundToBy}(\texttt{b},\texttt{a},\texttt{e}) \land \texttt{containsBinding}(\texttt{x},\texttt{e})\end{aligned} \tag{6.16}$$

$\texttt{isBoundToInOccurrence}(\texttt{b},\texttt{a},\texttt{x})$ ist wahr, wenn ein bindbares Bausteinelement $\texttt{b}$ an das bindbare Architekturelement $\texttt{a}$ in der Bausteininstanz $\texttt{x}$ gebunden wird:

$$\begin{aligned}\texttt{isBoundToInOccurrence}(\texttt{b},\texttt{a},\texttt{x}) := \texttt{BindableABBElement}(\texttt{b}) \land \\ \texttt{BindableArchElement}(\texttt{a}) \land \texttt{ABBOccurence}(\texttt{x}) \land \\ \exists \texttt{e} : \texttt{Binding}(\texttt{e}) \land \texttt{isBoundToInOccurrence}(\texttt{b},\texttt{a},\texttt{e},\texttt{x})\end{aligned} \tag{6.17}$$

$\texttt{isBoundByInOccurrence}(\texttt{b},\texttt{e},\texttt{x})$ ist wahr, wenn ein bindbares Bausteinelement $\texttt{b}$ durch die Bindung $\texttt{e}$ in der Bausteininstanz $\texttt{x}$ gebunden wird:

$$\begin{aligned}\texttt{isBoundByInOccurrence}(\texttt{b},\texttt{e},\texttt{x}) := \texttt{BindableABBElement}(\texttt{b}) \land \texttt{Binding}(\texttt{e}) \land \\ \texttt{ABBOccurence}(\texttt{x}) \land \texttt{containsBinding}(\texttt{x},\texttt{e}) \land \texttt{bindsABBElement}(\texttt{e},\texttt{b})\end{aligned} \tag{6.18}$$

$\texttt{isBoundInOccurrence}(\texttt{b},\texttt{x})$ ist wahr, wenn ein bindbares Bausteinelement $\texttt{b}$ in der Bausteininstanz $\texttt{x}$ gebunden wird:

$$\begin{aligned}\texttt{isBoundInOccurrence}(\texttt{b},\texttt{x}) := \texttt{BindableABBElement}(\texttt{b}) \land \texttt{ABBOccurence}(\texttt{x}) \land \\ \exists \texttt{e} : \texttt{Binding}(\texttt{e}) \land \texttt{isBoundByInOccurrence}(\texttt{b},\texttt{e},\texttt{x})\end{aligned} \tag{6.19}$$

Die folgenden Hilfsprädikate bilden mathematische Konzepte ab. Sie werden nicht als Abkürzung einer prädikatenlogischen Formel formuliert; stattdessen wird die Menge der Tupel definiert, für die das Prädikat als wahr ausgewertet wird.

`containsRangeValue(n, v)` ist wahr, wenn ein ganzzahliger Wert `v` innerhalb eines Intervalls `n` liegt, welches den Wertebereich einer Multiplizität repräsentiert:

$$\texttt{containsRangeValue} = \{(\texttt{n}, \texttt{v}) \mid \texttt{v} \in \texttt{n}\} \tag{6.20}$$

`count(P, v)` ist ein Prädikat der Prädikatenlogik zweiter Stufe. Die Anzahl der gültigen Belegungen der freien Variablen in der prädikatenlogischen Formel im ersten Parameter `P` wird mit dem zweiten Parameter `v` unifiziert. Die zugrundliegende Menge ist wie folgt definiert:

$$\texttt{count} = \{(\texttt{P}, \texttt{v}) \mid |\texttt{P}| = \texttt{v}\} \tag{6.21}$$

Allgemeine Bedingungen

Die unterschiedlichen Elemente eines Architekturbausteins werden bei seiner Instanziierung an Architekturelemente gebunden. Allerdings darf ein strukturelles Bausteinelement pro Bausteininstanz nur maximal ein Mal an dasselbe Architekturelement gebunden werden:

$$\begin{aligned}\mathbf{IC_1} := \forall \texttt{o} \forall \texttt{a} \forall \texttt{b} \forall \texttt{x} \forall \texttt{y} : \; & \texttt{ABBOccurrence}(\texttt{o}) \wedge \texttt{Binding}(\texttt{x}) \wedge \texttt{Binding}(\texttt{y}) \wedge \\ & (\texttt{StructuralRole}(\texttt{a}) \vee \texttt{ABBConnectorEnd}(\texttt{a})) \wedge \\ & \texttt{BindableArchElement}(\texttt{b}) \wedge \texttt{isBoundToByInOccurrence}(\texttt{a},\texttt{b},\texttt{x},\texttt{o})^{1} \wedge \\ & \texttt{isBoundToByInOccurrence}(\texttt{a},\texttt{b},\texttt{y},\texttt{o}) \rightarrow (\texttt{x} \equiv \texttt{y})\end{aligned} \quad {}^{1(6.16)}$$

Durch die unterschiedlichen Bindungstypen wird impliziert, welcher Bausteinelementtyp an welchen Architekturelementtyp gebunden wird. Allerdings ermöglicht die Definition des Metamodells durch das Klassendiagramm in Abbildung 6.26 bisher auch nicht konforme Zuordnungen. Durch die folgenden Bedingungen werden diese nicht konformen Zuordnungen untersagt:

$$\begin{aligned}\mathbf{IC_2} := \forall \texttt{x} \forall \texttt{a} \forall \texttt{b} : \; & \texttt{EntityBinding}(\texttt{x}) \wedge \texttt{bindsABBElement}(\texttt{x},\texttt{a}) \wedge \texttt{bindsArchElement}(\texttt{x},\texttt{b}) \rightarrow \\ & \texttt{ABBEntity}(\texttt{a}) \wedge \texttt{Part}(\texttt{b})\end{aligned}$$

$$\begin{aligned}\mathbf{IC_3} := \forall \texttt{x} \forall \texttt{a} \forall \texttt{b} : \; & \texttt{ConnectorBinding}(\texttt{x}) \wedge \texttt{bindsABBElement}(\texttt{x},\texttt{a}) \wedge \texttt{bindsArchElement}(\texttt{x},\texttt{b}) \rightarrow \\ & \texttt{ABBConnector}(\texttt{a}) \wedge \texttt{LinkPath}(\texttt{b})\end{aligned}$$

$$\begin{aligned}\mathbf{IC_4} := \forall \texttt{x} \forall \texttt{a} \forall \texttt{b} : \; & \texttt{ConnectorEndBinding}(\texttt{x}) \wedge \texttt{bindsABBElement}(\texttt{x},\texttt{a}) \wedge \texttt{bindsArchElement}(\texttt{x},\texttt{b}) \rightarrow \\ & \texttt{ABBConnectorEnd}(\texttt{a}) \wedge \texttt{PortVariable}(\texttt{b})\end{aligned}$$

$$\begin{aligned}\mathbf{IC_5} := \forall \texttt{x} \forall \texttt{a} \forall \texttt{b} : \; & \texttt{MessageBinding}(\texttt{x}) \wedge \texttt{bindsABBElement}(\texttt{x},\texttt{a}) \wedge \texttt{bindsArchElement}(\texttt{x},\texttt{b}) \rightarrow \\ & \texttt{ABBMessage}(\texttt{a}) \wedge \texttt{MsgTrace}(\texttt{b})\end{aligned}$$

$$\begin{aligned}\mathbf{IC_6} := \forall \texttt{x} \forall \texttt{a} \forall \texttt{b} : \; & \texttt{ScenarioBinding}(\texttt{x}) \wedge \texttt{bindsABBElement}(\texttt{x},\texttt{a}) \wedge \texttt{bindsArchElement}(\texttt{x},\texttt{b}) \rightarrow \\ & \texttt{ABBScenario}(\texttt{a}) \wedge \texttt{Scenario}(\texttt{b})\end{aligned}$$

In einer Bausteininstanz können nur bindbare Elemente des instanziierten Bausteins gebunden werden:

[1](6.22) $\mathbf{IC_7} := \forall o \forall e \forall a : \texttt{ABBOccurence}(o) \wedge \texttt{Binding}(e) \wedge \texttt{containsBinding}(o,e) \wedge$
$\texttt{BindableABBElement}(a) \wedge \texttt{bindsABBElement}(e,a) \rightarrow$
$\exists b : \texttt{ABB}(b) \wedge \texttt{instantiatedABB}(o,b) \wedge$
$((\texttt{ABBRole}(a) \wedge \texttt{containsRole}(b,a)) \vee$
$(\texttt{ABBConnectorEnd}(a) \wedge \texttt{hasEnd}(c,a)^1 \wedge \texttt{ABBConnector}(c) \wedge \texttt{containsRole}(b,c)) \vee$
$(\texttt{ABBScenario}(a) \wedge \texttt{containsABBScenario}(b,a)))$

Hierbei ist das Prädikat $\texttt{hasEnd}(c, x)$ wahr, wenn das Konnektorrollenende x ein Ende der Konnektorrolle c ist:

$$\texttt{hasEnd}(c,x) := \texttt{ABBConnector}(c) \wedge \texttt{ABBConnectorEnd}(x) \wedge (\texttt{srcEnd}(c,x) \vee \texttt{tarEnd}(c,x)) \tag{6.22}$$

Bindung der Entitätenrollen

Wie oft eine Entitätenrolle pro Instanz ihres Bausteins gebunden werden darf, legt ihr Attribut bindingMultiplicity fest. Diese Art der Multiplizität variiert für jede Instanz der Metaklasse ABBEntity und kann daher nicht in dem Klassendiagramm des Metamodells global eingeschränkt werden.

[1](6.18), [2](6.21), [3](6.20) $\mathbf{IC_8} := \forall b \forall x \forall y \exists n \exists v :$
$\texttt{ABBOccurence}(x) \wedge \texttt{ABBEntity}(y) \wedge \texttt{instantiatedABB}(x,b) \wedge \texttt{ABB}(b) \wedge$
$\texttt{containsRole}(b,y) \wedge \texttt{bindingMultiplicity}(y,n) \wedge$
$\texttt{count}((\texttt{EntityBinding}(z) \wedge \texttt{isBoundByInOccurence}(y,z,x)^1), v)^2 \rightarrow$
$\texttt{containsRangeValue}(n,v)^3$

Die Bausteinbeschreibungssprache ermöglicht die Festlegung, dass Entitätenrollen hierarchisch ineinander geschachtelt werden können. Wird ein Baustein mit geschachtelten Entitätenrollen gebunden, so ist dies auch in der Architektur zu berücksichtigen. Der Komponententyp des Parts, an den die „äußere“ Entitätenrolle gebunden ist, muss einen Part besitzen, an den die „innere“ Entitätenrolle gebunden ist:

[1](6.17) $\mathbf{IC_9} := \forall o \forall x \forall y \forall p \forall q : \texttt{ABBOccurrence}(o) \wedge \texttt{ABBEntity}(x) \wedge \texttt{ABBEntity}(y) \wedge$
$\texttt{encapsulatesABBEntity}(x,y) \wedge \texttt{Part}(p) \wedge \texttt{isBoundToInOccurrence}(x,p,o)^1 \wedge$
$\texttt{Part}(q) \wedge \texttt{isBoundToInOccurrence}(y,q,o) \rightarrow$
$\exists c \exists d : \texttt{Component}(c) \wedge \texttt{hasType}(p,c) \wedge \texttt{Configuration}(d) \wedge$
$\texttt{hasConfiguration}(c,d) \wedge \texttt{containsPart}(d,q)$

Bindung der Konnektorrollenenden

Konnektorrollenenden, die an einer Entitätenrolle hängen, werden ebenfalls gebunden. Anders als bei der Bindung von Entitätenrollen legt die Multiplizität des Konnektorrollenendes die mögliche Bindungshäufigkeit nicht pro Bausteininstanz fest. Diese Multiplizität wird im Kontext jeder Bindung der übergeordneten Entitätenrolle betrachtet. In der folgenden Bedingung

werden die Bindungen kontextabhängig gezählt und mit dem Wert des Multiplizitätenattributs verglichen:

$$\begin{aligned}\mathbf{IC_{10}} := \forall o \forall a \forall e \forall q \exists n \exists v : & \texttt{ABBOccurrence}(o) \wedge \texttt{ABBEntity}(a) \wedge \\ & \texttt{ABBConnectorEnd}(e) \wedge \texttt{Part}(q) \wedge \texttt{isBoundToInOccurrence}(a,q,o) \wedge \\ & \texttt{isConnectedBy}(a,e) \wedge \texttt{bindingMultiplicity}(e,n) \wedge \\ & \texttt{count}((\texttt{PortVariable}(p) \wedge \texttt{hasPortvariable}(q,p) \wedge \\ & \qquad \texttt{isBoundToInOccurrence}(e,p,o)), v)^1 \rightarrow \\ & \texttt{containsRangeValue}(n,v)^2\end{aligned}$$

Ein Konnektorrollenende ist ein Teil einer Entitätenrolle. Die Existenz einer Bindung zwischen Konnektorrollenende und Portvariable setzt daher auch eine Bindung zwischen den übergeordneten Elementen (Entitätenrolle und Part) voraus:

$$\begin{aligned}\mathbf{IC_{11}} := \forall o \forall t \forall e \forall p \forall a \forall q : & \texttt{ABBOccurrence}(o) \wedge \texttt{ABBConnectorEnd}(e) \wedge \\ & \texttt{PortVariable}(p) \wedge \texttt{isBoundToInOccurrence}(e,p,o)^1 \wedge \texttt{ABBEntity}(a) \wedge \\ & \texttt{isConnectedBy}(a,e) \wedge \texttt{Part}(q) \wedge \texttt{hasPortVariable}(q,p) \rightarrow \\ & \exists s : \texttt{EntityBinding}(s) \wedge \texttt{isBoundToByInOccurence}(a,q,s,o)\end{aligned}$$

[1](6.17), [2](6.16)

Bindung der Konnektorrollen

Konnektorrollen verbinden über Konnektorrollenenden jeweils zwei Entitätenrollen miteinander. Wird bei einer Instanziierung eines Bausteins eine Konnektorrolle an einen Verbindungspfad gebunden, so müssen auch die Konnektorrollenenden an die Enden des Verbindungspfades gebunden werden:

$$\begin{aligned}\mathbf{IC_{12}} := \forall o \forall c \forall e \forall l : & \\ & \texttt{ABBOccurrence}(o) \wedge \texttt{ABBConnector}(c) \wedge \texttt{ABBConnectorEnd}(e) \wedge \texttt{LinkPath}(l) \wedge \\ & \texttt{isBoundToInOccurrence}(c,l,o)^1 \wedge \texttt{hasEnd}(c,e)^2 \rightarrow \\ & \exists p : \texttt{PortVariable}(p) \wedge \texttt{isBoundToInOccurrence}(e,p,o) \wedge \texttt{linkEnd}(l,p)^3\end{aligned}$$

[1](6.17), [2](6.22), [3](6.3)

Anders herum müssen ebenfalls auch alle Enden des Verbindungspfads an Konnektorrollenenden der zugehörigen Konnektorrolle gebunden sein:

$$\begin{aligned}\mathbf{IC_{13}} := \forall o \forall c \forall l \forall p : & \\ & \texttt{ABBOccurrence}(o) \wedge \texttt{ABBConnector}(c) \wedge \texttt{LinkPath}(l) \wedge \texttt{PortVariable}(p) \wedge \\ & \texttt{isBoundToInOccurrence}(c,l,o)^1 \wedge \texttt{linkEnd}(l,p)^2 \rightarrow \\ & \exists e : \texttt{ABBConnectorEnd}(e) \wedge \texttt{isBoundToInOccurrence}(e,p,o) \wedge \texttt{hasEnd}(c,e)^3\end{aligned}$$

[1](6.17), [2](6.3), [3](6.22)

Wie häufig eine Konnektorrolle insgesamt gebunden werden kann, ist nicht explizit festgelegt. Eingeschränkt wird dies zum einen durch die unterschiedlichen im Baustein definierten Multiplizitäten für Entitätenrollen und Konnektorrollenenden. Zum anderen besitzt eine Konnektorrolle die beiden Attribute srcBindingMultiplicity und tarBindingMultiplicity. Diese Multiplizitäten sind im Kontext der Portvariablen auszuwerten, die an die Enden der Konnektorrolle gebunden sind. Pro Portvariable werden hierzu die auslaufenden Verbindungspfade gezählt, die an die betrachtete Konnektorrolle gebunden sind. Für Portvariablen, die an das srcEnd der Konnektorrolle gebunden sind, muss die Anzahl zu srcBindingMultiplicity konform sein. Für an tarEnd

gebundene Portvariablen gilt analog tarBindingMultiplicity.

[1](6.17), [2](6.3), [3](6.21), [4](6.20)

$$\mathbf{IC_{14}} := \forall o \forall c \forall e \forall p \exists n \exists v : \texttt{ABBOccurrence}(o) \wedge \texttt{ABBConnector}(c) \wedge \texttt{ABBConnectorEnd}(e) \wedge \texttt{PortVariable}(p) \wedge \texttt{isBoundToInOccurrence}(e,p,o)^1 \wedge ((\texttt{srcEnd}(c,e) \wedge \texttt{srcBindingMultiplicity}(c,n)) \vee (\texttt{tarEnd}(c,e) \wedge \texttt{tarBindingMultiplicity}(c,n))) \wedge \texttt{count}((\texttt{LinkPath}(l) \wedge \texttt{isBoundToInOccurrence}(c,l,o) \wedge \texttt{linkEnd}(l,p)^2), v)^3 \rightarrow \texttt{containsRangeValue}(n,v)^4$$

Eine Konnektorrolle kann sowohl als singleTyped als auch als multiTyped definiert sein. Dieses Attribut legt fest, in wie weit sich die Verbindungspfade ähneln, die in einer Bausteininstanz an dieselbe Konnektorrolle gebunden werden. Entsprechend der Ausführungen in Abschnitt 6.3.1 sind Verbindungspfade ähnlich zueinander (singleTyped), wenn ihre jeweils gegenüberliegenden Endpunkte beliebig miteinander verbunden werden können. Übertragen auf die Architekturbeschreibung bedeutet dies, dass die Portvariablen an den Enden der Verbindungspfade kompatibel zueinander sein müssen. Zu unterscheiden sind zwei Fälle: Zum einen kann eine Konnektorrolle zwei ineinander geschachtelte Entitätenrollen miteinander verbinden. In diesem Fall haben die an diese Konnektorrolle gebundenen Verbindungspfade eine delegierende Funktion. Bei Ähnlichkeit von delegierenden Verbindungspfaden müssen die Portvariablen an ihren Enden delegationskompatibel sein (vgl. Abschnitt 6.2.4). Im zweiten Fall verbindet die Konnektorrolle zwei Entitätenrollen auf gleicher Ebene. Damit die Verbindungspfade ähnlich zueinander sind, müssen hier die Portvariablen verbindungskompatibel sein.

[1](6.17), [2](6.23), [3](6.24), [4](6.25)

$$\mathbf{IC_{15}} := \forall o \forall c \forall k \forall l : \texttt{ABBOccurrence}(o) \wedge \texttt{ABBConnector}(c) \wedge \texttt{LinkPath}(k) \wedge \texttt{LinkPath}(l) \wedge \texttt{contype}(c, \texttt{"singleTyped"}) \wedge \texttt{isBoundToInOccurrence}(c,k,o)^1 \wedge \texttt{isBoundToInOccurrence}(c,l,o) \wedge \neg(k \equiv l) \rightarrow (\texttt{encapsulatedABBConnector}(c)^2 \wedge \texttt{similarDelLinkPath}(k,l)^3) \vee (\neg\texttt{encapsulatedABBConnector}(c) \wedge \texttt{similarConLinkPath}(k,l)^4)$$

Eine Konnektorrolle kann zwischen zwei Entitätenrollen verlaufen, von denen die eine die andere enthält. Trifft dies auf dies auf eine Konnektorrolle c zu, ist encapsulatedABBConnector(c) mit nachfolgender Definition wahr:

[1](6.22)

$$\texttt{encapsulatedABBConnector}(c) := \texttt{ABBConnector}(c) \wedge \exists a \exists b \exists s \exists t : \texttt{ABBConnectorEnd}(a) \wedge \texttt{ABBConnectorEnd}(b) \wedge \neg(a \equiv b) \wedge \texttt{hasEnd}(c,a)^1 \wedge \texttt{hasEnd}(c,b) \wedge \texttt{ABBEntity}(s) \wedge \texttt{ABBEntity}(t) \wedge \texttt{isConnectedBy}(s,a) \wedge \texttt{isConnectedBy}(b,t) \wedge \texttt{encapsulatesABBEntity}(s,t) \tag{6.23}$$

Bei einem delegierenden Verbindungspfad sind die Portvariablen an dessen Enden delegationskompatibel. Zwei delegierende Verbindungspfade sind nun ähnlich, wenn ihre Portvariablen auch kreuzweise delegationskompatibel sind:

[1](6.26), [2](6.4)

$$\texttt{similarDelLinkPath}(k,l) := \texttt{LinkPath}(k) \wedge \texttt{LinkPath}(l) \wedge \forall p \forall q \forall s \forall t : \texttt{PortVariable}(p) \wedge \texttt{PortVariable}(q) \wedge \texttt{PortVariable}(s) \wedge \texttt{PortVariable}(t) \wedge \texttt{linkPathEnd}(k,p)^1 \wedge \texttt{linkPathEnd}(k,q) \wedge \neg(p \equiv q) \wedge \texttt{linkPathEnd}(l,s) \wedge \texttt{linkPathEnd}(l,t) \wedge \neg(s \equiv t) \wedge \texttt{pvDelCompatible}(p,q)^2 \wedge \texttt{pvDelCompatible}(s,t) \wedge \texttt{pvDelCompatible}(p,t) \wedge \texttt{pvDelCompatible}(s,q) \tag{6.24}$$

Zwei Verbindungspfade, dessen Enden verbindungskompatibel sind, sind ähnlich, wenn die Enden auch kreuzweise verbindungskompatibel sind:

$$\begin{aligned}&\texttt{similarConLinkPath}(\texttt{k},\texttt{l}) := \texttt{LinkPath}(\texttt{k}) \wedge \texttt{LinkPath}(\texttt{l}) \wedge \\ &\quad \forall \texttt{p} \forall \texttt{q} \forall \texttt{s} \forall \texttt{t} : \texttt{PortVariable}(\texttt{p}) \wedge \texttt{PortVariable}(\texttt{q}) \wedge \texttt{PortVariable}(\texttt{s}) \wedge \\ &\quad \texttt{PortVariable}(\texttt{t}) \wedge \texttt{linkPathEnd}(\texttt{k},\texttt{p})^{1} \wedge \texttt{linkPathEnd}(\texttt{k},\texttt{q}) \wedge \neg(\texttt{p} \equiv \texttt{q}) \wedge \\ &\quad \texttt{linkPathEnd}(\texttt{l},\texttt{s}) \wedge \texttt{linkPathEnd}(\texttt{l},\texttt{t}) \wedge \neg(\texttt{s} \equiv \texttt{t}) \wedge \\ &\quad \texttt{pvConCompatible}(\texttt{p},\texttt{q})^{2} \wedge \texttt{pvConCompatible}(\texttt{s},\texttt{t}) \wedge \\ &\quad \texttt{pvConCompatible}(\texttt{p},\texttt{t}) \wedge \texttt{pvConCompatible}(\texttt{s},\texttt{q})\end{aligned} \tag{6.25}$$

[1](6.26), [2](6.1)

Die beiden vorigen Prädikate machen Aussagen über die Enden eines Verbindungspfads. Um eine Portvariable p zu identifizieren, die Ende eines Verbindungspfades e ist, wird das Prädikat $\texttt{linkPathEnd}(\texttt{p},\texttt{e})$ definiert. Dies ist der Fall, wenn es kein zu e gehörendes Verbindungselement k gibt, das von p wegführt oder das von einer anderen Portvariablen q des Parts, zu dem e gehört, wegführt:

$$\begin{aligned}&\texttt{linkPathEnd}(\texttt{e},\texttt{p}) := \texttt{LinkPath}(\texttt{e}) \wedge \texttt{PortVariable}(\texttt{p}) \wedge \\ &\quad \exists \texttt{l} : \texttt{Link}(\texttt{l}) \wedge \texttt{linkEnd}(\texttt{l},\texttt{p})^{1} \wedge \texttt{pathElement}(\texttt{e},\texttt{l}) \wedge \\ &\quad (\neg \exists \texttt{k} : \texttt{Link}(\texttt{k}) \wedge \texttt{pathElement}(\texttt{e},\texttt{k}) \wedge \neg(\texttt{k} \equiv \texttt{l}) \wedge \\ &\quad\ (\texttt{linkEnd}(\texttt{k},\texttt{p}) \vee \\ &\quad\quad (\exists \texttt{q} \exists \texttt{b} : \texttt{PortVariable}(\texttt{q}) \wedge \texttt{hasPortVariable}(\texttt{b},\texttt{q}) \wedge \texttt{Part}(\texttt{b}) \wedge \\ &\quad\quad\ \texttt{hasPortVariable}(\texttt{b},\texttt{p}) \wedge \neg(\texttt{q} \equiv \texttt{p}) \wedge \texttt{linkEnd}(\texttt{q},\texttt{k}))))\end{aligned} \tag{6.26}$$

[1](6.3)

Eine multiTyped Konnektorrolle erlaubt sowohl Ähnlichkeit als auch Unähnlichkeit, so dass in diesem Fall keine Einschränkungen zu beachten sind.

In einem Baustein ist eine Konnektorrolle ein direktes Verbindungselement zwischen zwei Entitätenrollen. Hingegen kann ein Verbindungspfad, an den eine Konnektorrolle gebunden wird, deutlich komplexer sein und aus einer beliebigen Anzahl an Verbindungselementen und Verzweigungen bestehen. Die Portvariablen an den Enden des Verbindungspfades sind an die Konnektorrollenenden der Konnektorrolle des Verbindungspfades gebunden. Unabhängig von der Komplexität des Verbindungspfades dürfen keine anderen als die gebundenen Enden existieren:

$$\begin{aligned}\mathbf{IC_{16}} := &\forall \texttt{o} \forall \texttt{l} \forall \texttt{p} : \texttt{ABBOccurrence}(\texttt{o}) \wedge \texttt{LinkPath}(\texttt{l}) \wedge \texttt{PortVariable}(\texttt{p}) \wedge \\ &\texttt{isBoundToInOccurrence}(\texttt{c},\texttt{l},\texttt{o})^{1} \wedge \texttt{ABBConnector}(\texttt{c}) \wedge \texttt{linkPathEnd}(\texttt{l},\texttt{p})^{2} \rightarrow \\ &\exists \texttt{e} : \texttt{ABBConnectorEnd}(\texttt{e}) \wedge (\texttt{srcEnd}(\texttt{c},\texttt{e}) \vee \texttt{tarEnd}(\texttt{c},\texttt{e})) \wedge \\ &\texttt{isBoundToInOccurrence}(\texttt{e},\texttt{p},\texttt{o})\end{aligned}$$

[1](6.17), [2](6.26)

Bindung der Nachrichtenrollen

Eine Nachrichtenrolle wird an eine Nachrichtenspur gebunden. Eine Nachrichtenspur besteht aus einer zusammenhängenden Folge von Nachrichtenvorkommen. Jedem Nachrichtenvorkommen in einer Nachrichtenspur geht bis auf das erste Nachrichtenvorkommen immer ein anderes vorher ($\mathbf{IC_{17}}$), sowie bis auf das letzte Nachrichtenvorkommen immer eines nachfolgt ($\mathbf{IC_{18}}$):

$$\begin{aligned}\mathbf{IC_{17}} := &\forall \texttt{t} \forall \texttt{o} : \texttt{MsgTrace}(\texttt{t}) \wedge \texttt{MsgOccurrence}(\texttt{o}) \wedge \texttt{traceElement}(\texttt{t},\texttt{o}) \rightarrow \\ &(\exists \texttt{k} : \texttt{MsgOccurrence}(\texttt{k}) \wedge \texttt{traceElement}(\texttt{t},\texttt{k}) \wedge \texttt{nextTraceElement}(\texttt{k},\texttt{o})) \vee \\ &\texttt{beginningOfMsgTrace}(\texttt{o},\texttt{t})^{1}\end{aligned}$$

[1](6.27)

[1](6.28)

$$\mathbf{IC_{18}} := \forall t \forall o : \texttt{MsgTrace}(t) \wedge \texttt{MsgOccurrence}(o) \wedge \texttt{traceElement}(t,o) \rightarrow (\exists k : \texttt{MsgOccurrence}(k) \wedge \texttt{traceElement}(t,k) \wedge \texttt{nextTraceElement}(o,k)) \vee \texttt{endOfMsgTrace}(o,t)^1$$

Die beiden Prädikate $\texttt{beginningOfMsgTrace}(o,t)$ und $\texttt{endOfMsgTrace}(o,t)$ stellen fest, ob ein Nachrichtenvorkommen o Anfang bzw. Ende einer Nachrichtenspur t ist:

$$\texttt{beginningOfMsgTrace}(o,t) := \texttt{MsgTrace}(t) \wedge \texttt{MsgOccurrence}(o) \wedge \texttt{traceElement}(t,o) \wedge \neg\exists p : \texttt{MessageOccurrence}(p) \wedge \texttt{traceElement}(t,p) \wedge \texttt{nextTraceElement}(p,o) \tag{6.27}$$

$$\texttt{endOfMsgTrace}(o,t) := \texttt{MsgTrace}(t) \wedge \texttt{MsgOccurrence}(o) \wedge \texttt{traceElement}(t,o) \wedge \neg\exists p : \texttt{MessageOccurrence}(p) \wedge \texttt{traceElement}(t,p) \wedge \texttt{nextTraceElement}(o,p) \tag{6.28}$$

Der Zusammenhang innerhalb einer Nachrichtenspur definiert sich nicht alleine darüber, dass für jedes Nachrichtenvorkommen ein Vorgänger oder Nachfolger definiert ist. Aufeinanderfolgende Nachrichtenvorkommen müssen zudem im Empfänger des ersten und im Sender des zweiten übereinstimmen:

[1](6.9), [2](6.8)

$$\mathbf{IC_{19}} := \forall t \forall a \forall b : \texttt{MsgTrace}(t) \wedge \texttt{MsgOccurrence}(a) \wedge \texttt{traceElement}(t,a) \wedge \texttt{MsgOccurrence}(b) \wedge \texttt{traceElement}(t,b) \wedge \texttt{nextTraceElement}(a,b) \rightarrow \exists p : \texttt{PortVariable}(p) \wedge \texttt{msgOccReceiver}(a,p)^1 \wedge \texttt{msgOccSender}(b,p)^2$$

Die Bindung eines Bausteinszenarios erfolgt vollständig an ein Architekturszenario und verteilt sich nicht über verschiedene Architekturszenarios. Dementsprechend müssen die Nachrichtenvorkommen einer Nachrichtenspur alle zu demselben Architekturszenario gehören:

$$\mathbf{IC_{20}} := \forall t \forall o \forall p : \texttt{MsgTrace}(t) \wedge \texttt{MsgOccurrence}(o) \wedge \texttt{MsgOccurrence}(p) \wedge \texttt{traceElement}(t,o) \wedge \texttt{traceElement}(t,p) \rightarrow \exists s : \texttt{Scenario}(s) \wedge \texttt{containsMsgOccurrence}(s,o) \wedge \texttt{containsMsgOccurrence}(s,p)$$

Eine Nachrichtenrolle hat als Sender und Empfänger jeweils ein Konnektorrollenende. Wird eine Nachrichtenrolle an eine Nachrichtenspur gebunden, müssen ebenfalls Sender und Empfänger der Nachrichtenrolle an die jeweiligen Enden der Nachrichtenspur gebunden werden. Da eine Nachrichtenrolle auch einen im Baustein nicht spezifizierten Sender oder Empfänger besitzen kann, wird diese Bedingung für Sender ($\mathbf{IC_{21}}$) und Empfänger ($\mathbf{IC_{22}}$) aufgeteilt:

[1](6.17), [2](6.29)

$$\mathbf{IC_{21}} := \forall o \forall n \forall m \exists s \exists t : \texttt{ABBOccurrence}(o) \wedge \texttt{ABBMessage}(n) \wedge \texttt{MsgTrace}(m) \wedge \texttt{isBoundToInOccurrence}(n,m,o)^1 \wedge \texttt{ABBConnectorEnd}(s) \wedge \texttt{abbMsgSender}(n,s) \wedge \texttt{PortVariable}(t) \wedge \texttt{msgTraceStart}(m,t)^2 \rightarrow \texttt{isBoundToInOccurrence}(s,t,o)$$

[1](6.17), [2](6.30)

$$\mathbf{IC_{22}} := \forall o \forall n \forall m \exists s \exists t : \texttt{ABBOccurrence}(o) \wedge \texttt{ABBMessage}(n) \wedge \texttt{MsgTrace}(m) \wedge \texttt{isBoundToInOccurrence}(n,m,o)^1 \wedge \texttt{ABBConnectorEnd}(s) \wedge \texttt{abbMsgReceiver}(n,s) \wedge \texttt{PortVariable}(t) \wedge \texttt{msgTraceEnd}(m,t)^2 \rightarrow \texttt{isBoundToInOccurrence}(s,t,o)$$

In den vorangegangenen Bedingungen wurden die beiden Prädikate $\texttt{msgTraceStart}(m,s)$ und

$\texttt{msgTraceEnd}(\texttt{m},\texttt{e})$ eingesetzt, um die sendende Portvariable des ersten Nachrichtenvorkommens bzw. die empfangende Portvariable des letzten Nachrichtenvorkommens der Nachrichtenspur $\texttt{m}$ zu identifizieren:

$$\begin{aligned}\texttt{msgTraceStart}(\texttt{m},\texttt{s}) := {} & \texttt{MsgTrace}(\texttt{m}) \wedge \texttt{PortVariable}(\texttt{s}) \wedge \\ & \exists \texttt{o} : \texttt{MsgOccurrence}(\texttt{o}) \wedge \texttt{traceElement}(\texttt{m},\texttt{o}) \wedge \texttt{msgOccSender}(\texttt{o},\texttt{s})^{1} \wedge \\ & \neg\exists \texttt{p} : \texttt{MsgOccurrence}(\texttt{p}) \wedge \texttt{traceElement}(\texttt{m},\texttt{p}) \wedge \texttt{nextMsgOccurrence}^{*}(\texttt{p},\texttt{o})^{2}\end{aligned} \tag{6.29}$$

[1](6.8), [2](6.6)

$$\begin{aligned}\texttt{msgTraceEnd}(\texttt{m},\texttt{e}) := {} & \texttt{MsgTrace}(\texttt{m}) \wedge \texttt{PortVariable}(\texttt{e}) \wedge \\ & \exists \texttt{o} : \texttt{MsgOccurrence}(\texttt{o}) \wedge \texttt{traceElement}(\texttt{m},\texttt{o}) \wedge \texttt{msgOccReceiver}(\texttt{a},\texttt{e})^{1} \wedge \\ & \neg\exists \texttt{p} : \texttt{MsgOccurrence}(\texttt{p}) \wedge \texttt{traceElement}(\texttt{m},\texttt{p}) \wedge \texttt{nextMsgOccurrence}^{*}(\texttt{o},\texttt{p})^{2}\end{aligned} \tag{6.30}$$

[1](6.9), [2](6.6)

Sofern eine Nachrichtenrolle keinen undefinierten Sender oder Empfänger hat, wird sie im Baustein entlang einer Konnektorrolle ausgetauscht. Ähnliches muss auch für Verbindungspfade und Nachrichtenspuren gelten, die an ein solches Paar von Konnektorrolle und Nachrichtenrolle gebunden sind. Zunächst müssen Verbindungspfad und Nachrichtenspur in derselben Bausteininstanz gebunden sein. Dann muss für jede gebundene Nachrichtenspur ein entsprechender Verbindungspfad existieren, an welchem die Nachrichtenspur entlang verläuft. Eine Nachrichtenspur verläuft entlang eines Verbindungspfads, wenn jedes ihrer Nachrichtenvorkommen entlang eines Verbindungselements des Verbindungspfades erfolgt. Ausgenommen von dieser Bedingung sind Stellen, an denen ein Nachrichtenvorkommen innerhalb eines atomar getypten Parts verläuft. Hier weist ein Verbindungspfad eine entsprechende Lücke auf. Nachrichtenrollen mit einem undefinierten Ende betrifft die folgende Bedingung nicht. Sie erfüllen bereits die Vorbedingung nicht, da sie nicht in Beziehung zu einer Konnektorrolle stehen:

$$\begin{aligned}\mathbf{IC_{23}} := {} & \forall \texttt{o} \forall \texttt{m} \forall \texttt{t} \exists \texttt{c} : \texttt{ABBOccurrence}(\texttt{o}) \wedge \texttt{ABBMessage}(\texttt{m}) \wedge \texttt{MsgTrace}(\texttt{t}) \wedge \\ & \texttt{isBoundToInOccurrence}(\texttt{m},\texttt{t},\texttt{o})^{1} \wedge \texttt{ABBConnector}(\texttt{c}) \wedge \texttt{abbMsgOverABBCon}(\texttt{m},\texttt{c})^{2} \rightarrow \\ & \exists \texttt{l} : \texttt{LinkPath}(\texttt{l}) \wedge \texttt{isBoundToInOccurrence}(\texttt{c},\texttt{l},\texttt{o}) \wedge \\ & \texttt{msgTraceAlongLinkPath}(\texttt{t},\texttt{l})^{3}\end{aligned}$$

[1](6.17), [2](6.31), [3](6.32)

In der vorangegangenen Bedingung kommen verschiedene Prädikate zur Abkürzung zum Einsatz, die im Folgenden erläutert werden. $\texttt{abbMsgOverABBCon}(\texttt{m},\texttt{c})$ ist wahr, wenn eine Nachrichtenrolle $\texttt{m}$ in einem Baustein entlang einer Konnektorrolle $\texttt{c}$ erfolgt. Hierzu wird überprüft, ob die Konnektorrollenenden Sender und Empfänger der Nachrichtenrolle sind:

$$\begin{aligned}\texttt{abbMsgOverABBCon}(\texttt{m},\texttt{c}) := {} & \texttt{ABBMessage}(\texttt{m}) \wedge \texttt{ABBConnector}(\texttt{c}) \wedge \\ & \exists \texttt{s} \exists \texttt{a} \exists \texttt{b} : \texttt{ABB}(\texttt{s}) \wedge \texttt{containsRole}(\texttt{s},\texttt{m}) \wedge \texttt{containsRole}(\texttt{s},\texttt{c}) \wedge \\ & \texttt{ABBConnectorEnd}(\texttt{a}) \wedge \texttt{ABBConnectorEnd}(\texttt{b}) \wedge \texttt{srcEnd}(\texttt{c},\texttt{a}) \wedge \texttt{tarEnd}(\texttt{c},\texttt{b}) \wedge \\ & ((\texttt{abbMsgSender}(\texttt{m},\texttt{a}) \wedge \texttt{abbMsgReceiver}(\texttt{m},\texttt{b})) \vee \\ & \ (\texttt{abbMsgSender}(\texttt{m},\texttt{b}) \wedge \texttt{abbMsgReceiver}(\texttt{m},\texttt{a})))\end{aligned} \tag{6.31}$$

Das Prädikat $\texttt{msgTraceAlongLinkPath}(\texttt{t},\texttt{l})$ ist wahr, wenn alle Nachrichtenvorkommen der Nachrichtenspur $\texttt{t}$ entweder entlang eines Verbindungselements von $\texttt{l}$ laufen oder durch einen atomar getypten Part verlaufen:

$$\begin{aligned}\texttt{msgTraceAlongLinkPath}(\texttt{t},\texttt{l}) := {} & \texttt{MsgTrace}(\texttt{t}) \wedge \texttt{LinkPath}(\texttt{l}) \wedge \\ & \forall \texttt{o} : \texttt{MsgOccurence}(\texttt{o}) \wedge \texttt{traceElement}(\texttt{t},\texttt{o}) \wedge \\ & ((\exists \texttt{k} : \texttt{Link}(\texttt{k}) \wedge \texttt{pathElement}(\texttt{l},\texttt{k}) \wedge \texttt{msgOccAlongLink}(\texttt{o},\texttt{k})^{1}) \vee \\ & \ (\exists \texttt{p} : \texttt{Part}(\texttt{p}) \wedge \texttt{msgOccThroughPart}(\texttt{o},\texttt{p})^{2}))\end{aligned} \tag{6.32}$$

[1](6.33), [2](6.34)

Ob ein Nachrichtenvorkommen $\texttt{o}$ entlang einer Verbindung $\texttt{l}$ erfolgt, überprüft das Prädikat

msgOccAlongLink(o,l). Dies trifft zu, wenn die Portvariablen, die Sender und Empfänger von o sind, ebenfalls die Endpunkte von l sind:

$$\begin{aligned}\texttt{msgOccAlongLink}(\texttt{o},\texttt{l}) := {} & \texttt{MsgOccurrence}(\texttt{o}) \wedge \texttt{Link}(\texttt{l}) \wedge \\ & \exists \texttt{a} \exists \texttt{b} : \texttt{PortVariable}(\texttt{a}) \wedge \texttt{PortVariable}(\texttt{b}) \wedge \texttt{msgOccSender}(\texttt{o},\texttt{a})^{1} \wedge \\ & \texttt{msgOccReceiver}(\texttt{o},\texttt{b})^{2} \wedge \texttt{linkEnd}(\texttt{l},\texttt{a})^{3} \wedge \texttt{linkEnd}(\texttt{l},\texttt{b})\end{aligned} \tag{6.33}$$

[1](6.8), [2](6.9), [3](6.3)

Das Prädikat msgOccThroughPart(o, p) wertet schließlich aus, ob ein Nachrichtenvorkommen o durch einen atomar getypten Part p weitergereicht wird. Dies ist der Fall, wenn Sender und Empfänger von o Portvariablen eines atomar getypten Parts sind und zwischen ihnen kein Verbindungselement existiert:

$$\begin{aligned}\texttt{msgOccThroughPart}(\texttt{o},\texttt{p}) := {} & \texttt{MsgOccurrence}(\texttt{o}) \wedge \texttt{Part}(\texttt{p}) \wedge \\ & \exists \texttt{a} \exists \texttt{b} : \texttt{PortVariable}(\texttt{a}) \wedge \texttt{PortVariable}(\texttt{b}) \wedge \texttt{hasPortVariable}(\texttt{p},\texttt{a}) \wedge \\ & \texttt{hasPortVariable}(\texttt{p},\texttt{b}) \wedge \texttt{msgOccSender}(\texttt{o},\texttt{a})^{1} \wedge \texttt{msgOccReceiver}(\texttt{o},\texttt{b})^{2} \wedge \\ & \exists \texttt{c} : \texttt{AtomicComponent}(\texttt{c}) \wedge \texttt{hasType}(\texttt{p},\texttt{c}) \wedge \\ & \neg \exists \texttt{l} : \texttt{Link}(\texttt{l}) \wedge \texttt{linkEnd}(\texttt{l},\texttt{a})^{3} \wedge \texttt{linkEnd}(\texttt{l},\texttt{b})\end{aligned} \tag{6.34}$$

[1](6.8), [2](6.9), [3](6.3)

In der Verhaltensbeschreibung eines Architekturbausteins kann für eine Nachrichtenrolle festgelegt werden, ob sie synchron oder asynchron erfolgen soll. Ist die Art des Nachrichtenaustauschs festgelegt, müssen auch alle Nachrichtenvorkommen einer Nachrichtenspur, an die diese Nachrichtenrolle gebunden ist, dazu konform sein:

$$\begin{aligned}\mathbf{IC_{24}} := {} & \forall \texttt{o} \forall \texttt{m} \forall \texttt{t} : \texttt{ABBOccurrence}(\texttt{o}) \wedge \texttt{ABBMessage}(\texttt{m}) \wedge \texttt{MsgTrace}(\texttt{t}) \wedge \\ & \texttt{isBoundToInOccurrence}(\texttt{m},\texttt{t},\texttt{o})^{1} \wedge \texttt{abbMsgType}(\texttt{m},\texttt{"synchronous"}) \rightarrow \\ & \forall \texttt{s} : \texttt{MsgOccurrence}(\texttt{s}) \wedge \texttt{traceElement}(\texttt{t},\texttt{s}) \wedge \texttt{msgType}(\texttt{s},\texttt{"synchronous"})\end{aligned}$$

[1](6.17)

$$\begin{aligned}\mathbf{IC_{25}} := {} & \forall \texttt{o} \forall \texttt{m} \forall \texttt{t} : \texttt{ABBOccurrence}(\texttt{o}) \wedge \texttt{ABBMessage}(\texttt{m}) \wedge \texttt{MsgTrace}(\texttt{t}) \wedge \\ & \texttt{isBoundToInOccurrence}(\texttt{m},\texttt{t},\texttt{o})^{1} \wedge \texttt{abbMsgType}(\texttt{m},\texttt{"asynchronous"}) \rightarrow \\ & \forall \texttt{s} : \texttt{MsgOccurrence}(\texttt{s}) \wedge \texttt{traceElement}(\texttt{t},\texttt{s}) \wedge \texttt{msgType}(\texttt{s},\texttt{"asynchronous"})\end{aligned}$$

[1](6.17)

Bindung der Bausteinszenarios

Das Verhalten eines Architekturbausteins wird wesentlich durch Bausteinszenarios bestimmt, welche die Nachrichtenrollen zusammenfassen. Im Rahmen einer Bausteininstanziierung werden die Bausteinszenarios an Szenarios der Architekturbeschreibung gebunden. Wie oft jedes Bausteinszenario für eine gültige Instanziierung gebunden werden muss, gibt das Attribut bindingMultiplicity an:

$$\begin{aligned}\mathbf{IC_{26}} := {} & \forall \texttt{a} \forall \texttt{o} \forall \texttt{s} \exists \texttt{m} \exists \texttt{n} \exists \texttt{v} : \texttt{ABB}(\texttt{a}) \wedge \texttt{ABBOccurrence}(\texttt{o}) \wedge \texttt{instantiatedABB}(\texttt{o},\texttt{a}) \wedge \\ & \texttt{ABBScenario}(\texttt{s}) \wedge \texttt{containsABBScenario}(\texttt{a},\texttt{s}) \wedge \texttt{bindingMultiplicity}(\texttt{s},\texttt{n}) \wedge \\ & \texttt{count}(\texttt{ScenarioBinding}(\texttt{b}) \wedge \texttt{isBoundByInOccurrence}(\texttt{s},\texttt{b},\texttt{o})^{1}),\texttt{v})^{2} \rightarrow \\ & \texttt{containsRangeValue}(\texttt{n},\texttt{v})\end{aligned}$$

[1](6.18), [3](6.21), [4](6.20)

Ein Bausteinszenario muss vollständig gebunden werden. Dies ist der Fall, wenn jede seiner Nachrichtenrollen im Kontext einer Szenariobindung ebenfalls gebunden wurde:

$$\begin{aligned}\mathbf{IC_{27}} := {} & \forall \texttt{s} \forall \texttt{b} \forall \texttt{m} : \texttt{ABBScenario}(\texttt{s}) \wedge \texttt{ScenarioBinding}(\texttt{b}) \wedge \texttt{bindsABBElement}(\texttt{b},\texttt{s}) \wedge \\ & \texttt{ABBMessage}(\texttt{m}) \wedge \texttt{includesABBMsg}(\texttt{s},\texttt{m}) \rightarrow \\ & \exists \texttt{n} : \texttt{MessageBinding}(\texttt{n}) \wedge \texttt{bindsABBElement}(\texttt{n},\texttt{m}) \wedge \texttt{containsMsgBinding}(\texttt{s},\texttt{n})\end{aligned}$$

Die Bindung einer Nachrichtenrolle kann nur einer Szenariobindung zugeordnet werden, wenn diese Nachrichtenrolle auch in diesem Bausteinszenario enthalten ist:

$$\begin{aligned}\mathbf{IC_{28}} := \forall a \forall b \forall m : &\texttt{ScenarioBinding}(a) \wedge \texttt{MessageBinding}(b) \wedge \\ &\texttt{containsMsgBinding}(a,b) \wedge \texttt{ABBMessage}(m) \wedge \texttt{bindsABBElement}(b,m) \rightarrow \\ &\exists s : \texttt{ABBScenario}(s) \wedge \texttt{bindsABBElement}(a,s) \wedge \texttt{includesABBMsg}(s,m)\end{aligned}$$

Szenariobindungen und die ihnen zugeordneten Nachrichtenbindungen müssen in derselben Bausteininstanz erfolgen:

$$\begin{aligned}\mathbf{IC_{29}} := \forall s \forall m : &\texttt{ScenarioBinding}(s) \wedge \texttt{MessageBinding}(m) \wedge \texttt{containsMsgBinding}(s,m) \rightarrow \\ &\exists o : \texttt{ABBOccurrence}(o) \wedge \texttt{containsBinding}(o,s) \wedge \texttt{containsBinding}(o,m)\end{aligned}$$

Ein Bausteinszenario wird an ein Architekturszenario gebunden und nicht auf verschiedene Architekturszenarios aufgeteilt. Daher müssen sich alle Nachrichtenspuren, an die die Nachrichtenrollen eines Bausteinszenarios im Kontext einer Szenariobindung gebundenen sind, in dem gebundenen Architekturszenario befinden:

$$\begin{aligned}\mathbf{IC_{30}} := \forall u \forall v \forall a \forall b \exists t : &\texttt{MsgTrace}(u) \wedge \texttt{MsgTrace}(v) \wedge \texttt{MessageBinding}(a) \wedge \\ &\texttt{bindsArchElement}(a,u) \wedge \texttt{MessageBinging}(b) \wedge \texttt{bindsArchElement}(b,v) \wedge \\ &\texttt{ScenarioBinding}(t) \wedge \texttt{containsMsgBinding}(t,a) \wedge \texttt{containsMsgBinding}(t,b) \rightarrow \\ &\exists s : \texttt{Scenario}(s) \wedge \texttt{bindsArchElement}(t,s) \wedge \\ &\texttt{containsMsgTrace}(s,u)^1 \wedge \texttt{containsMsgTrace}(s,v)\end{aligned} \quad {}^1(6.35)$$

Das dabei verwendete Prädikat $\texttt{containsMsgTrace}(o,t)$ ist wahr, wenn alle Nachrichtenvorkommen der Nachrichtenspur t auch in dem Architekturszenario s enthalten sind:

$$\begin{aligned}\texttt{containsMsgTrace}(o,t) := &\texttt{Scenario}(s) \wedge \texttt{MsgTrace}(t) \wedge \\ &\forall o : \texttt{MsgOccurrence}(o) \wedge \texttt{traceElement}(t,o) \rightarrow \\ &\texttt{containsMsgOccurrence}(s,o)\end{aligned} \quad (6.35)$$

Nachrichtenrollen schränken ihre Bindungshäufigkeit ein. Sie legen fest, wie viele Sender oder Empfänger insgesamt für die an sie gebundenen Nachrichtenspuren pro Szenariobindung existieren dürfen. Diese Multiplizitäten sind in den Attributen senderMultiplicity und receiverMultiplicity hinterlegt. Ihre Einhaltung wird durch die Bedingungen $\mathbf{IC_{31}}$ bzw. $\mathbf{IC_{32}}$ sichergestellt:

$$\begin{aligned}\mathbf{IC_{31}} := \forall s \forall m \exists a \exists u \exists v : &\texttt{ScenarioBinding}(s) \wedge \texttt{ABBMessage}(m) \wedge \texttt{ABBScenario}(a) \wedge \\ &\texttt{includesABBMsg}(a,m) \wedge \texttt{bindsABBElement}(s,a) \wedge \texttt{senderMultiplicity}(m,u) \wedge \\ &\texttt{count}(\texttt{PortVariable}(p) \wedge \texttt{traceStartOfBoundABBMsgInScenBinding}(p,m,s)^1, v)^2 \rightarrow \\ &\texttt{containsRangeValue}(u,v)^3\end{aligned} \quad \begin{matrix}{}^1(6.36),\\ {}^2(6.21),\\ {}^3(6.20)\end{matrix}$$

$$\begin{aligned}\mathbf{IC_{32}} := \forall s \forall m \exists a \exists u \exists v : &\texttt{ScenarioBinding}(s) \wedge \texttt{ABBMessage}(m) \wedge \texttt{ABBScenario}(a) \wedge \\ &\texttt{includesABBMsg}(a,m) \wedge \texttt{bindsABBElement}(s,a) \wedge \texttt{receiverMultiplicity}(m,u) \wedge \\ &\texttt{count}(\texttt{PortVariable}(p) \wedge \texttt{traceEndOfBoundABBMsgInScenBinding}(p,m,s)^1, v)^2 \rightarrow \\ &\texttt{containsRangeValue}(u,v)^3\end{aligned} \quad \begin{matrix}{}^1(6.37),\\ {}^2(6.21),\\ {}^3(6.20)\end{matrix}$$

In den vorangegangenen Bedingungen wurden die beiden Ausdrücke $\texttt{traceStartOfBoundABB-MsgInScenBinding}(p, m, s)$ und $\texttt{traceEndOfBoundABBMsgInScenBinding}(p, m, s)$ verwendet. Diese sind wahr, wenn eine Portvariable p der Start bzw. das Ende einer Nachrichtenspur

ist, die im Kontext einer Szenariobindung s an eine Nachrichtenrolle m gebunden ist:

$^{1}(6.29)$, $^{2}(6.38)$

$$\begin{aligned}&\texttt{traceStartOfBoundABBMsgInScenBinding}(\texttt{p},\texttt{m},\texttt{s}) :=\\ &\quad \texttt{PortVariable}(\texttt{p}) \wedge \texttt{ABBMessage}(\texttt{m}) \wedge \texttt{ScenarioBinding}(\texttt{s}) \wedge\\ &\quad \exists \texttt{t} : \texttt{MsgTrace}(\texttt{t}) \wedge \texttt{msgTraceStart}(\texttt{p},\texttt{t})^{1} \wedge\\ &\quad \texttt{abbMsgBoundToTraceInScenBinding}(\texttt{m},\texttt{t},\texttt{s})^{2}\end{aligned} \tag{6.36}$$

$^{1}(6.30)$, $^{2}(6.38)$

$$\begin{aligned}&\texttt{traceEndOfBoundABBMsgInScenBinding}(\texttt{p},\texttt{m},\texttt{s}) :=\\ &\quad \texttt{PortVariable}(\texttt{p}) \wedge \texttt{ABBMessage}(\texttt{m}) \wedge \texttt{ScenarioBinding}(\texttt{s}) \wedge\\ &\quad \exists \texttt{t} : \texttt{MsgTrace}(\texttt{t}) \wedge \texttt{msgTraceEnd}(\texttt{p},\texttt{t})^{1} \wedge\\ &\quad \texttt{abbMsgBoundToTraceInScenBinding}(\texttt{m},\texttt{t},\texttt{s})^{2}\end{aligned} \tag{6.37}$$

Das Prädikat $\texttt{abbMsgBoundToTraceInScenBinding}(\texttt{m},\texttt{t},\texttt{s})$ ist wahr, wenn eine Nachrichtenrolle m im Kontext einer Szenariobindung s an eine Nachrichtenspur t gebunden ist:

$$\begin{aligned}&\texttt{abbMsgBoundToTraceInScenBinding}(\texttt{m},\texttt{t},\texttt{s}) :=\\ &\quad \texttt{ABBMessage}(\texttt{m}) \wedge \texttt{MsgTrace}(\texttt{t}) \wedge \texttt{Scenario}(\texttt{s}) \wedge\\ &\quad \exists \texttt{n} : \texttt{MsgBinding}(\texttt{n}) \wedge \texttt{containsMsgBinding}(\texttt{s},\texttt{n}) \wedge\\ &\quad \texttt{bindsABBElement}(\texttt{n},\texttt{m}) \wedge \texttt{bindsArchElement}(\texttt{n},\texttt{t})\end{aligned} \tag{6.38}$$

Über den Nachrichtenrollen in einem Bausteinszenario ist eine Ordnung definiert. Diese Ordnung muss sich auch in der Abfolge der Nachrichtenspuren wiederfinden, die im Kontext einer Szenariobindung an diese Nachrichtenrollen gebunden sind. Eine Nachrichtenrolle kann innerhalb einer Szenariobindung auch an verschiedene Nachrichtenspuren gebunden sein. Daher müssen zwei aufeinander folgende Nachrichtenspuren entweder an dieselbe Nachrichtenrolle oder an zwei aufeinander folgende gebunden sein:

$^{1}(6.39)$, $^{2}(6.12)$

$$\begin{aligned}\mathbf{IC_{33}} :=\ &\forall \texttt{a} \forall \texttt{b} \forall \texttt{c} \forall \texttt{s} \forall \texttt{t} \forall \texttt{m} \forall \texttt{n} : \texttt{ScenarioBinding}(\texttt{a}) \wedge \texttt{MessageBinding}(\texttt{b}) \wedge\\ &\texttt{MessageBinding}(\texttt{c}) \wedge \texttt{containsMsgBinding}(\texttt{a},\texttt{b}) \wedge \texttt{containsMsgBinding}(\texttt{a},\texttt{c}) \wedge\\ &\texttt{MsgTrace}(\texttt{s}) \wedge \texttt{MsgTrace}(\texttt{t}) \wedge \texttt{nextMsgTrace}(\texttt{s},\texttt{t})^{1} \wedge \texttt{bindsArchElement}(\texttt{b},\texttt{s}) \wedge\\ &\texttt{bindsArchElement}(\texttt{c},\texttt{t}) \wedge \texttt{ABBMessage}(\texttt{m}) \wedge \texttt{ABBMessage}(\texttt{n}) \wedge\\ &\texttt{bindsABBElement}(\texttt{b},\texttt{m}) \wedge \texttt{bindsABBElement}(\texttt{c},\texttt{n}) \rightarrow\\ &\texttt{nextABBMsg}(\texttt{m},\texttt{n})^{2} \vee (\texttt{m} \equiv \texttt{n})\end{aligned}$$

Das Prädikat $\texttt{nextMsgTrace}(\texttt{s},\texttt{t})$ identifiziert zwei aufeinander folgende Nachrichtenspuren s und t. Beide werden innerhalb derselben Szenariobindung an zwei Nachrichtenrollen des entsprechenden Szenarios gebunden. Die Reihenfolge wird dabei über die Nachrichtenvorkommen bestimmt, aus denen die Nachrichtenspuren bestehen und über die bereits eine Ordnung definiert ist:

$^{1}(6.27)$, $^{2}(6.6)$

$$\begin{aligned}&\texttt{nextMsgTrace}(\texttt{s},\texttt{t}) := \texttt{MsgTrace}(\texttt{s}) \wedge \texttt{MsgTrace}(\texttt{t}) \wedge\\ &\quad \exists \texttt{a} \exists \texttt{b} \exists \texttt{c} \exists \texttt{m} \exists \texttt{n} : \texttt{ScenarioBinding}(\texttt{a}) \wedge \texttt{MessageBinding}(\texttt{b}) \wedge\\ &\quad \texttt{MessageBinding}(\texttt{c}) \wedge \texttt{containsMsgBinding}(\texttt{a},\texttt{b}) \wedge\\ &\quad \texttt{containsMsgBinding}(\texttt{a},\texttt{c}) \wedge \texttt{MessageOccurrence}(\texttt{m}) \wedge\\ &\quad \texttt{MessageOccurrence}(\texttt{n}) \wedge \texttt{beginningOfMsgTrace}(\texttt{m},\texttt{s})^{1} \wedge\\ &\quad \texttt{beginningOfMsgTrace}(\texttt{n},\texttt{t}) \wedge \texttt{nextMsgOccurrence}^{*}(\texttt{m},\texttt{n})^{2} \wedge\\ &\quad \neg \exists \texttt{u} \exists \texttt{d} \exists \texttt{o} : \texttt{MsgTrace}(\texttt{u}) \wedge \texttt{MessageBinding}(\texttt{d}) \wedge\\ &\quad \texttt{containsMsgBinding}(\texttt{a},\texttt{d}) \wedge\\ &\quad \texttt{MessageOccurrence}(\texttt{o}) \wedge \texttt{beginningOfMsgTrace}(\texttt{o},\texttt{u}) \wedge\\ &\quad \texttt{nextMsgOccurrence}^{*}(\texttt{m},\texttt{o}) \wedge \texttt{nextMsgOccurrence}^{*}(\texttt{o},\texttt{n})\end{aligned} \tag{6.39}$$

Folgen in einem Bausteinszenario zwei Nachrichtenrollen aufeinander, kann dies unterschiedliche Bedeutungen haben. So kann ein direkter Bezug zwischen ihnen bestehen, indem die eine als Auslöser der anderen angesehen wird. Es kann sich aber auch lediglich um eine lose zeitliche Beziehung handeln, die aussagt, dass eine Nachrichtenrolle irgendwann nach der anderen erfolgt. Im Baustein wird die Art der Abfolge pro Vorgänger/Nachfolger-Paar durch das Attribut msgOrder mit den Werten direct und indirect definiert. Werden zwei Nachrichtenspuren im Kontext einer Szenariobindung an zwei direkt aufeinander folgende Nachrichtenrollen gebunden, muss die erste Nachrichtenspur Auslöser der zweiten sein. Allerdings können sich zwei direkt aufeinander folgende Nachrichtenrollen in den Konnektorrollenenden als Empfänger der ersten und Sender der zweiten unterscheiden. Beide Konnektorrollenende müssen dann aber zu derselben Entitätenrolle gehören. Zwei Nachrichtenspuren, die an zwei solche Nachrichtenrollen gebunden sind, müssen an Portvariablen desselben Part enden bzw. starten. Zudem muss zwischen Endpunkt der ersten und Startpunkt der zweiten eine Folge von sich auslösenden Nachrichtenvorkommen existieren:

$$
\begin{aligned}
\mathbf{IC_{34}} := {} & \forall a \forall b \forall c \forall s \forall t \forall m \forall n : \texttt{ScenarioBinding}(a) \wedge \texttt{MessageBinding}(b) \wedge \\
& \texttt{MessageBinding}(c) \wedge \texttt{containsMsgBinding}(a,b) \wedge \texttt{containsMsgBinding}(a,c) \wedge \\
& \texttt{MsgTrace}(s) \wedge \texttt{MsgTrace}(t) \wedge \texttt{bindsArchElement}(b,s) \wedge \\
& \texttt{bindsArchElement}(c,t) \wedge \texttt{ABBMessage}(m) \wedge \texttt{ABBMessage}(n) \wedge \\
& \texttt{bindsABBElement}(b,m) \wedge \texttt{bindsABBElement}(c,n) \wedge \texttt{nextABBMsg}(m,n,\texttt{"direct"}) \rightarrow \\
& \exists e \exists f : \texttt{MsgOccurrence}(e) \wedge \texttt{MsgOccurrence}(f) \wedge \\
& \texttt{endOfMsgTrace}(e,s)^{1} \wedge \texttt{beginningOfMsgTrace}(f,t)^{2} \wedge \\
& (\texttt{triggersMsgOcc}(e,f) \vee \\
& \quad (\exists p \exists q \exists r : \texttt{Part}(p) \wedge \texttt{PortVariable}(q) \wedge \texttt{PortVariable}(r) \wedge \\
& \quad \texttt{msgOccReceiver}(e,q)^{3} \wedge \texttt{msgOccSender}(f,r)^{4} \wedge \texttt{hasPortVariable}(p,q) \wedge \\
& \quad \texttt{hasPortVariable}(p,r) \wedge \texttt{triggersMsgOcc}^{\star}(e,f)^{5}))
\end{aligned}
$$

[1](6.28), [2](6.27), [3](6.9), [4](6.8), [5](6.40)

Ob zwei Nachrichtenvorkommen sich auslösen, wird durch die Beziehung triggersMsgOcc festgelegt. Das Prädikat $\texttt{triggersMsgOcc}^{\star}(e,f)$ erweitert diese Beziehung transitiv:

$$
\begin{aligned}
\texttt{triggersMsgOcc}^{\star}(e,f) := {} & \texttt{triggersMsgOcc}(e,f) \vee \\
& (\exists g : \texttt{triggersMsgOcc}(e,g) \wedge \texttt{triggersMsgOcc}^{\star}(g,f))
\end{aligned}
\tag{6.40}
$$

6.4.3. Beispiel: Instanziierung von Bausteinen

Das Metamodell zur Integrationsbeschreibung stellt die Beschreibungsmittel zur Verfügung, um die Zusammenhänge zwischen der Beschreibung von Bausteinen und einer aus diesen Bausteinen erstellten Architektur aufzuzeigen. Bisher wurden das Beispielsystem (Abschnitte 6.2.5 und 6.2.7) und verschiedene, dort verwendete Bausteine (siehe Abschnitte 6.3.2 und 6.3.4) entsprechend der jeweiligen Metamodelle beschrieben. Der vorliegende Abschnitt erläutert die Verbindung zwischen der Architektur des Beispielsystems und den im Beispielsystem verwendeten Architekturbausteinen. Dadurch ist es möglich, die Bausteininstanzen, aus denen das Beispielsystem maßgeblich besteht, und die in diesem Zusammenhang vorgenommenen Rollenbindungen nachzuvollziehen.

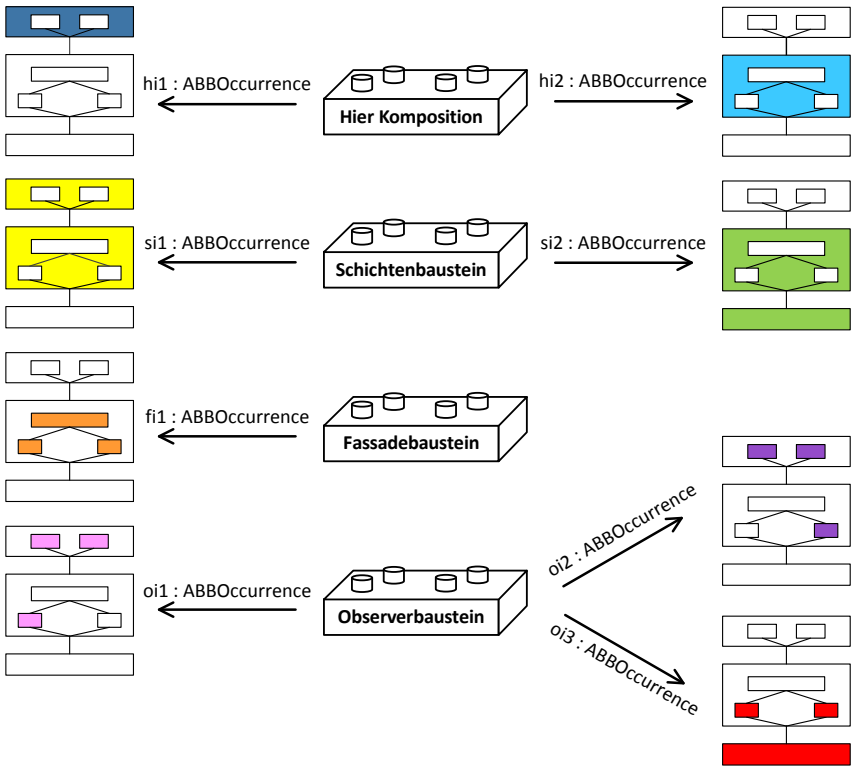


Abbildung 6.27: Überblick über die verschiedenen Bausteininstanzen im Beispielsystem.

Bausteininstanzen im Beispielsystem

Abbildung 6.27 gibt zunächst auf Strukturebene einen Überblick über die im Beispielsystem verwendeten Architekturbausteine und deren Instanzen. Jeder Bausteininstanz ist eine schematische Darstellung des Beispielsystems zugeordnet. In dieser sind jene Parts farblich gekennzeichnet, an die die Entitätenrollen des jeweiligen Bausteins in der betrachteten Instanz gebunden sind. So findet der Baustein Hierarchische Komposition jeweils bei der Hierarchisierung der Parts gui und al Verwendung (hi1 und hi2). Ebenso wird der Schichtenbaustein zwei Mal instanziiert und umfasst zum einen gui und al (si1) und zum anderen al und data (si2). Der Fassadebaustein wird ein Mal innerhalb von al verwendet (fi1). Vom Pull-Observerbaustein existieren hingegen drei Instanzen im Beispielsystem. Zunächst wird durch zwei Instanzen die Beobachtung der beiden Verwaltungspart in al durch kgui und mgui realisiert (oi1 und oi2). Die dritte Instanz umfasst die Verwaltungsparts als Beobachter der Daten aus data (oi3).

Bindung der strukturellen Bausteinaspekte im Beispielsystem

Die schematische Darstellung gibt keine Informationen über die Entitätenrollen, die an den jeweiligen Part gebunden wurde. Diese Information enthält Abbildung 6.28, die die bereits aus Abbildung 6.17 bekannte Systemkonfiguration mit ergänzenden Annotationen zeigt. Jeder Part wurde um Angaben über die an ihn gebundenen Entitätenrollen und über die Bausteininstanz, in der diese Bindungen jeweils erfolgten, ergänzt. Zudem kennzeichnen farbige, quaderförmige Symbole die verschiedenen Bindungen der Entitätenrollen, wobei gleichfarbige Symbole auf gebundene Entitätenrollen derselben Bausteininstanz hinweisen. Die Farben der Symbole korre-

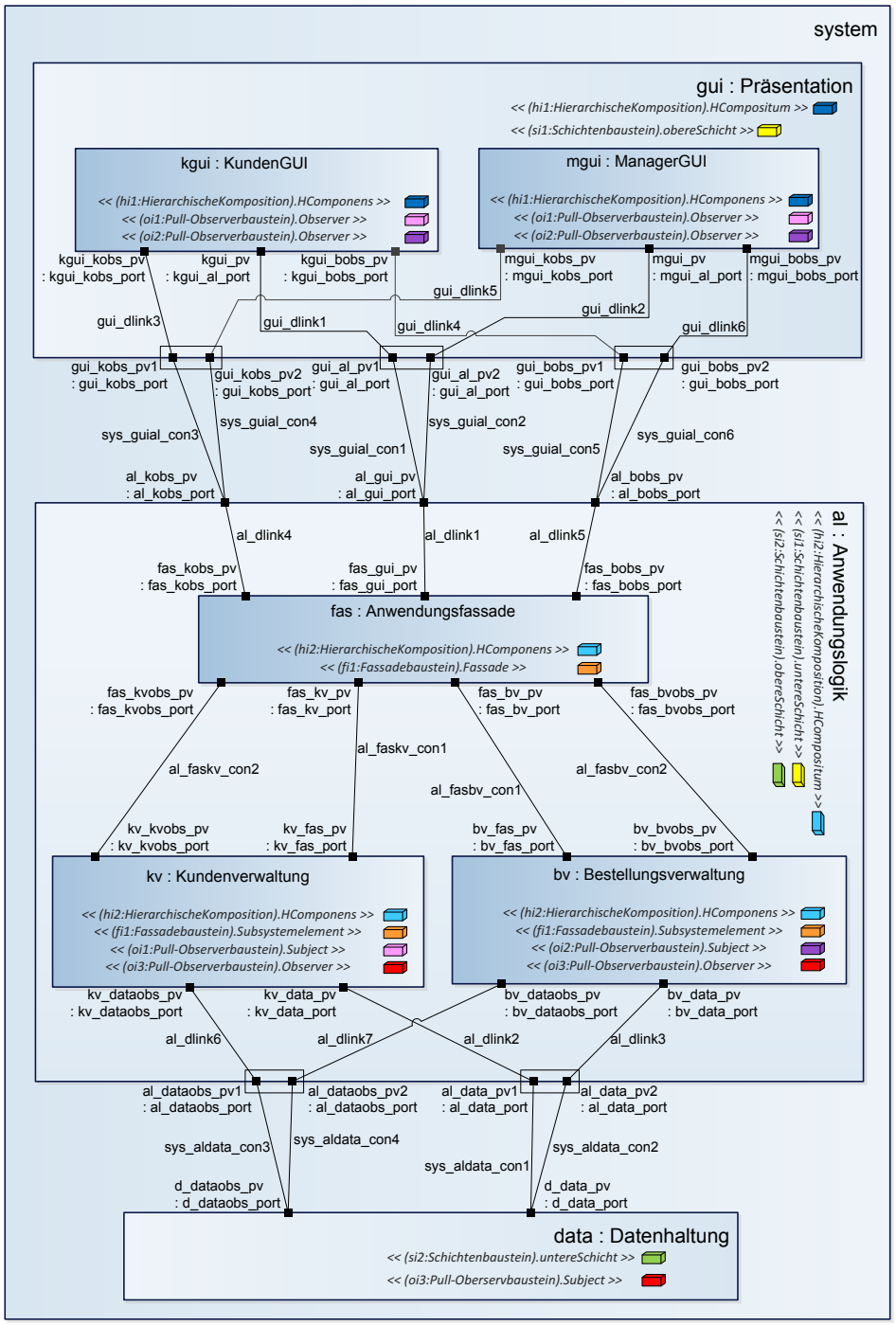


Abbildung 6.28: Systemkonfiguration des Beispielsystems mit Ergänzung der gebundenen Entitätenrollen der jeweiligen Bausteininstanzen.

lieren dabei mit den Farben der Instanzen aus Abbildung 6.27. So vereint kgui drei verschiedene Rollen: Zum einen ist dieser Part durch die Bausteininstanz hi1 () an die Rolle HComponens gebunden und zum anderen agiert sie durch die Bindungen in den Bausteininstanzen oi1 () und oi2 () in zwei verschiedenen Kontexten jeweils als Observer.

Abbildung 6.29 listet die Verbindungspfade auf, an die die Konnektorrollen in den Bausteininstanzen des Beispielsystems gebunden sind. In der Instanz oi1 () des Observerbausteins ist z.B. die Konnektorrolle obs_con an den Verbindungspfad gebunden Dieser besteht aus den vier Verbindungselementen gui_dlink3, sys_guial_con3, al_dlink4 und al_faskv_con2. Durch diesen Verbindungspfad werden die Parts kgui und kv verbunden, an die in derselben Bausteininstanz die Entitätenrollen Observer bzw. Subject gebunden wurden (vgl. Abbildung 6.28).

Die Abbildungen 6.28 und 6.29 zeigen lediglich die Bindungen jeweils einer Art von Bausteinrolle. Abbildung 6.30 visualisiert hingegen die Bindungen aller strukturellen bindbaren Bausteinelemente für die Instanz oi1 des Pull-Observerbausteins. Zum Beispiel erfolgen die Bindungen der Entitätenrollen Observer und Subject an die Parts kgui und kv durch die Entitätenbindungen eb2 bzw. eb3. Die Konnektorrolle obs_con wird durch die Konnektorbindung cb1 an einen Verbindungspfad aus den vier Verbindungselementen gui_dlink3, sys_guial_con3, al_dlink4 und al_faskv_con2 gebunden. Schließlich werden die Konnektorrollenenden von obs_con an die Portvariablen kgui_kobs_pv und kv_kvobs pv der Parts kgui bzw. kv als Enden des Verbindungspfads gebunden.

Bindung der verhaltensspezifischen Bausteinaspekte im Beispielsystem

Die Bindung von verhaltensspezifischen Elementen des Pull-Observerbausteins wird in Abbildung 6.31 dargestellt. Für diese Illustration wurde ein Ausschnitt des in Abbildung 6.20 gezeigten Architekturszenario gewählt. In diesem Ausschnitt wird die Datenhaltung (data) ignoriert und entsprechend die mit diesem Part erfolgenden Nachrichtenvorkommen. Dieses Auslassen wird durch einen grauen Kasten gekennzeichnet. Zur besseren Orientierung sind zusätzlich noch einmal die Bindungen der Konnektorrollenenden (oi1_ceb1 bis oi1_ceb6) an die verschiedenen Portvariablen aufgeführt. Die Bindungen der Entitätenrollen Observer an kgui und mgui sowie Subject an kv können ebenso wie die Bindung der Konnektorrollen in der zuvor beschriebenen Abbildung 6.30 nachvollzogen werden.

Die bindbaren Elemente der Verhaltensbeschreibung eines Bausteins sind seine Bausteinszenarios und die in diesen enthaltenen Nachrichtenrollen. Der Pull-Observerbaustein definiert ein Bausteinszenario, das im vorliegenden Beispiel mit der Szenariobindung oi1_sb1 an das abgebildete Architekturszenario gebunden wird. Im Kontext dieser Szenariobindung erfolgen die Nachrichtenbindungen oi1_mb1 bis oi1_mb7 der unterschiedlichen Nachrichtenrollen. Jede Nachrichtenrolle, die eine Sender- oder Empfängermultiplizität größer als 1 aufweist, wird ein Mal pro gebundenem Observer-Subject-Paar und damit insgesamt jeweils zwei Mal gebunden. Nicht zutreffend ist dies für die Nachrichtenrolle notify, die sowohl nur einen Sender als auch nur einen Empfänger pro Szenariobindung besitzen kann. Sie bildet auch in anderer Hinsicht eine Ausnahme. Während die anderen Nachrichtenrollen an eine mehrelementige Nachrichtenspur gebunden sind, ist die Nachrichtenrolle notify in diesem Fall durch oi1_mb3 an eine Nachrichtenspur mit nur einem Element gebunden (Nachrichtenvorkommen notify zwischen den Portvariablen kv_dataobs_pv und kv_kobs_pv). Die Nachrichtenrolle register wird an zwei unterschiedliche und sich nicht überschneidende Nachrichtenspuren aus Nachrichtenvorkommen von register-Nachrichten gebunden. Durch die Nachrichtenbindung oi1_mb1 erfolgt dies an eine Nachrichtenspur, die an der Portvariablen mgui_kobs_pv beginnt und an kv_kobs_pv endet. oi1_mb2 bindet die-

Baustein	Bausteininstanz	Konnektorrolle	Verbindungspfad
Schichtenbaustein	si1	schichten_con	{sys_guial_con1}
			{sys_guial_con2}
			{sys_guial_con3}
			{sys_guial_con4}
			{sys_guial_con5}
			{sys_guial_con6}
	si2	schichten_con	{sys_aldata_con1}
			{sys_aldata_con2}
			{sys_aldata_con3}
			{sys_aldata_con4}
Hierarchische Komposition	hi1	hcomp_con	{gui_dlink1}
			{gui_dlink2}
			{gui_dlink3}
			{gui_dlink4}
			{gui_dlink5}
			{gui_dlink6}
	hi2	hcmp_con	{al_dlink1}
			{al_dlink2}
			{al_dlink3}
			{al_dlink4}
			{al_dlink5}
			{al_dlink6}
			{al_dlink7}
Pull-Observerbaustein	oi1	obs_con	{gui_dlink3, sys_guial_con3, al_dlink4, al_faskv_con2}
			{gui_dlink5, sys_guial_con4, al_dlink4, al_faskv_con2}
		modell_con	{gui_dlink1, sys_guial_con1, al_dlink1, al_faskv_con1}
			{gui_dlink2, sys_guial_con2, al_dlink1, al_faskv_con1}
	oi2	obs_con	{gui_dlink4, sys_guial_con5, al_dlink5, al_fasbv_con2}
			{gui_dlink6, sys_guial_con6, al_dlink5, al_fasbv_con2}
		modell_con	{gui_dlink1, sys_guial_con1, al_dlink1, al_fasbv_con1}
			{gui_dlink2, sys_guial_con2, al_dlink1, al_fasbv_con1}
	oi3	obs_con	{al_dlink6, sys_aldata_con3}
			{al_dlink7, sys_aldata_con4}
		modell_con	{al_dlink2, sys_aldata_con1}
			{al_dlink3, sys_aldata_con2}
Fassadebaustein	fi1	fassade_con	{al_faskv_con1}
			{al_faskv_con2}
			{al_fasbv_con1}
			{al_fasbv_con2}

Abbildung 6.29: Übersicht über die Bindungen der Konnektorrollen im Beispielsystem.

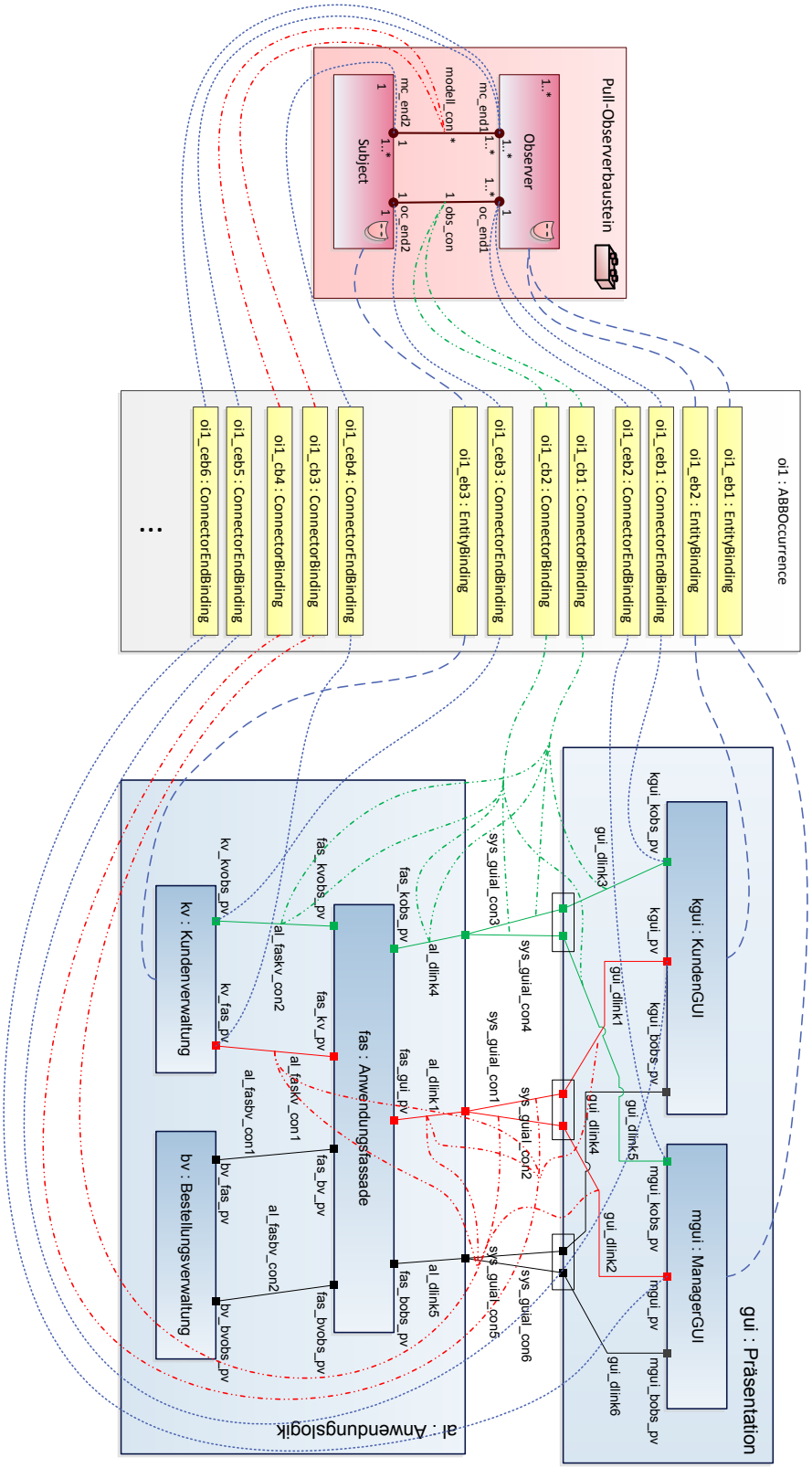


Abbildung 6.30: Visualisierung der Bindung von strukturellen Bausteinelementen bei einer Instanziierung des Pull-Observerbausteins.

se Nachrichtenrolle außerdem an eine Nachrichtenspur zwischen kgui_kobs_pv und kv_kobs_pv. Die beiden Nachrichtenspuren, an die durch die Nachrichtenbindungen oi1_mb4 und oi1_mb5 die Nachrichtenrolle update gebunden ist, starten beide an der Portvariablen kv_kobs_pv des an Subject gebundenen Parts kv und enden an mgui_kobs_pv bzw. kgui_kobs_pv. In dem vorliegenden Beispiel bestehen die bisher betrachteten Nachrichtenspuren alle aus Nachrichtenvorkommen zu jeweils identisch benannten Nachrichten. Die Bezeichnungen der beiden Nachrichtenspuren, an die die Nachrichtenrolle getModell gebunden ist, unterscheiden sich hingegen. Sie enthalten beide zunächst vier Nachrichtenvorkommen von verschiedenen mit getClientData benannten Nachrichten und dann jeweils zwei unterschiedliche getData-Vorkommen. Dabei starten die beiden Nachrichtenspuren mit mgui_kobs_pv und kgui_kobs_pv an unterschiedlichen Portvariablen, enden aber beide an kv_fas_pv.

Im Vergleich mit der Abbildung, die die Bindung der strukturellen Bausteinelemente für die Bausteininstanz oi1 illustriert, sind die Verbindungspfade identifizierbar, über die die Nachrichtenspuren erfolgen. Es ist erkennbar, dass der Zusammenhang zwischen Konnektorrolle und der darüber erfolgenden Nachrichtenrolle auch bei den gebundenen Architekturelementen eingehalten wird.

6.5. Beschreibung der Eigenschaftsbedingungen von Bausteinen

Ein Architekturbaustein wird nicht nur durch seine Struktur und sein Verhalten charakterisiert, sondern besitzt darüber hinaus weitere Eigenschaften. Diese speziellen Eigenschaften werden oftmals nur informell dokumentiert und sind an Bedingungen gebunden. Solche Eigenschaftsbedingungen betreffen z.B. Beziehungen zwischen den Elementen eines Architekturbausteins oder machen Vorgaben für die Verwendung eines Bausteins im Zusammenspiel mit anderen Bausteinanwendungen. Bei der Beschreibung verschiedener Eigenschaftsbedingungen werden abstrakte Begriffe wie Abhängigkeit oder erlaubte Nutzung verwendet. Diese Begriffe betreffen Beziehungen zwischen den Rollen des jeweiligen Bausteins und werden auf die Architekturelemente, an die diese Rollen gebunden werden, übertragen.

Eigenschaftsbedingungen lassen sich nur bedingt durch eine reine Struktur- oder Verhaltensbeschreibung ausdrücken, wie sie in Abschnitt 6.3 in den dort vorgestellten Sprachmitteln zur Struktur- und Verhaltensbeschreibung vorgestellt wurde. Eine Integration dieser Begriffe würde aufgrund ihrer unterschiedlichen Gestalt zu teilweise aufwändigen Erweiterungen des aktuellen Metamodells führen. Außerdem kann die Beschreibung eines neuen Bausteins eine wiederholte Anpassung des Metamodells erfordern. Zudem wird nicht das Maß an Flexibilität geboten, das für die teils komplexen Begriffe notwendig ist. Daher werden diese Bedingungen auf Basis prädikatenlogischer Ausdrücke formuliert. Da die Eigenschaftsbedingungen Teil der Bausteinbeschreibung sind, können die für die Formulierung verwendbaren Prädikate nicht aus dem gesamten Metamodell abgeleitet werden. Neben Prädikaten, die auf dem Bausteinmetamodell basieren, kommen nur noch die generischen Teile von Integrations- und Architekturmetamodell zum Einsatz. Die Gestalt einiger Bedingungen erfordert allerdings z.B. auf konkrete Sprachelemente der Architekturbeschreibung Bezug zu nehmen. Zur Wahrung der Unabhängigkeit zwischen Architektur- und Bausteinbeschreibung kommen sogenannte Adapterausdrücke (Abschnitt 6.5.1) zum Einsatz. Die prädikatenlogisch formulierten Eigenschaftsbedingungen weisen alle ein gemeinsames Schema auf (Abschnitt 6.5.2). Aufbauend auf diesem Schema können die Eigenschaftsbedingungen abkürzend und übersichtlich dargestellt werden. In Abschnitt 6.5.3 wird die Eigenschaftsbedingungen eines im Beispielsystem verwendeten Architekturbausteins

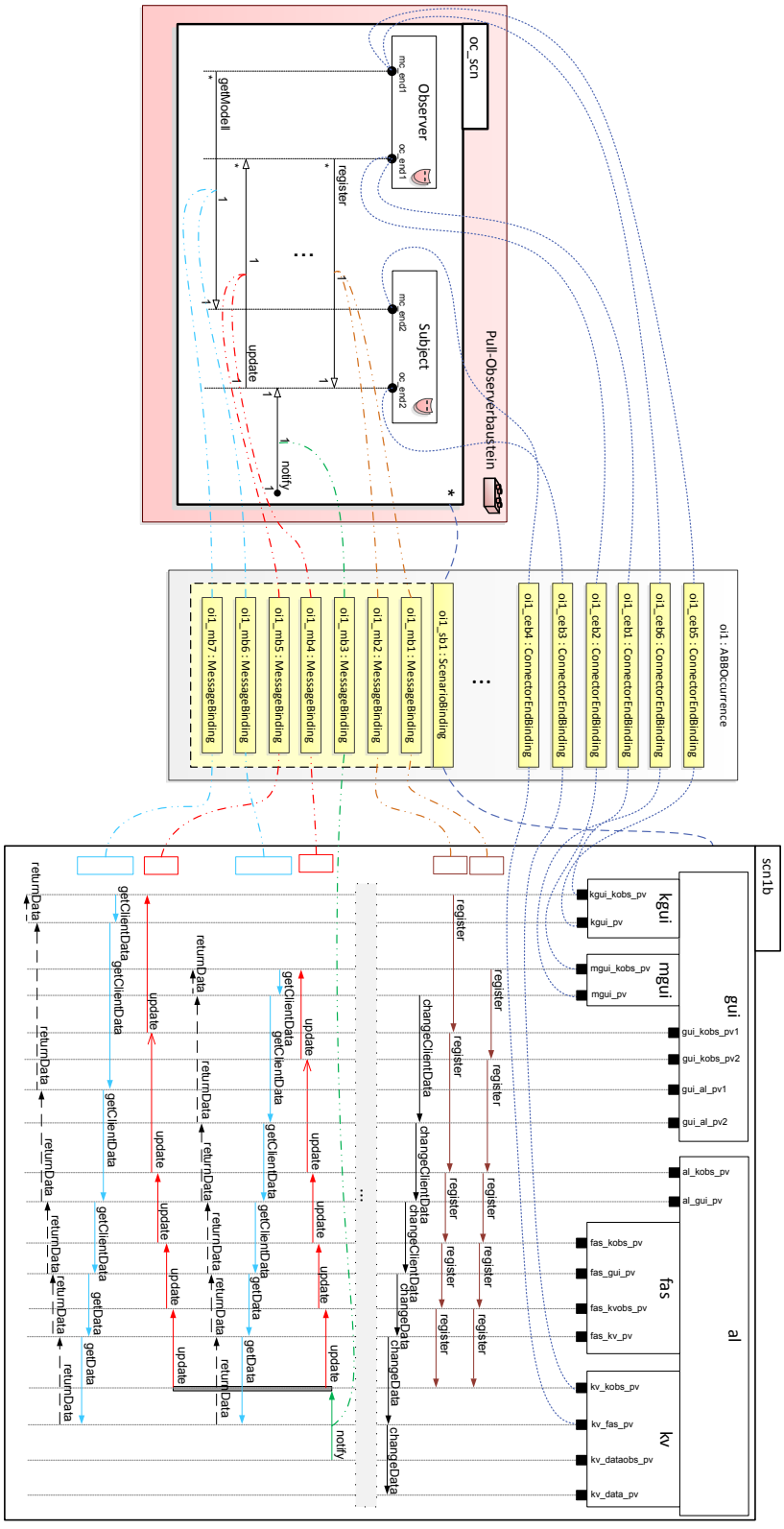


Abbildung 6.31: Visualisierung der Bindung von verhaltenspezifischen Rollen bei einer Instanziierung des Pull-Observerbausteins.

formuliert. Neben der ausführlichen Definition wird ebenfalls die abkürzende Schreibweise eingeführt.

6.5.1. Adapterausdruck

Adapterausdrücke stellen die in der informellen Beschreibung von Bausteinbedingungen verwendeten abstrakten Begriffe der formalen Beschreibung zur Verfügung. Außerdem konkretisieren sie diese Begriffe, indem sie die Semantik der Begriffe im Kontext der Architekturbeschreibungssprache durch Kombination von Architekturbeschreibungselementen definieren. Dabei soll auch ihr Name weitestgehend von der konkreten Architekturbeschreibungssprache abstrahieren.

Für ihre Verwendung in Eigenschaftsbedingungen werden Adapterausdrücke durch Prädikate repräsentiert. Diese *Adapterprädikate* werden wie andere Prädikate auch durch eine prädikatenlogische Regel definiert, die in diesem Fall folgende Gestalt hat:

```
Adapterprädikat\n := Definition des Architektursachverhalts
```

Der abstrakte Begriff, der mit Hilfe eines Adapterausdrucks formuliert werden soll, wird durch das Prädikat auf der linken Seite einer prädikatenlogischen Regel repräsentiert (`Adapterprädikat`). Dieses Prädikat kann eine beliebige Anzahl `n` von Variablen besitzen, die auf Elemente vom Typ boundArchElement verweisen. Die rechte Seite der Regel (`Definition des Architektursachverhalts`) definiert die Semantik des abstrakten Begriffs. Der hierzu verwendete prädikatenlogische Ausdruck kann dabei nur Prädikate enthalten, die Sprachelemente aus dem Metamodell der Architekturbeschreibung oder Adapterausdrücke repräsentieren. Außerdem können Prädikate verwendet werden, die zur Abkürzung eines Ausdrucks aus erlaubten Prädikaten definiert sind.

Anhand des Architekturbausteins Schichtenmuster wird im Folgenden der Begriff des Adapterausdrucks veranschaulicht. Dieser Baustein impliziert eine Abhängigkeitsbeziehung zwischen den Architekturelementen, an die seine beiden Entitätenrollen gebunden sind. Diese Beziehung schränkt ein, in welche Richtung und auf welche Art Abhängigkeiten z.B. in Form von Aufrufen oder Nutzungen zwischen den Elementen bestehen dürfen. So darf nur das Architekturelement, das die obere Schicht repräsentiert, von dem Architekturelement abhängen, das die untere Schicht repräsentiert. In die umgekehrte Richtung sind nur Nutzungen erlaubt, die zu keiner Abhängigkeit führen, wie z.B. asynchrone Call-Back-Aufrufe. Der hier verwendete Begriff der Abhängigkeit ist ein abstrakter Begriff, der mit Hilfe des Adapterprädikats `dependingLayerUsage\2` konkretisiert wird. Zwei Architekturelemente können auf unterschiedliche Weise voneinander abhängen. Ein Beispiel ist ein synchroner Methodenaufruf, durch den eine Warteabhängigkeit entsteht. Alle Nutzungsbeziehungen, die im Rahmen des Schichtenmusters als Abhängigkeit identifiziert worden sind und ggf. bereits durch einen Adapterausdruck konkretisiert wurden, können schließlich durch einen einziges (Adapter-)Prädikat zusammengefasst werden. Ein Architekturelement `x` hängt nun bezüglich der Schichtung von einem Architekturelement `y` ab, wenn eine identifizierte Art von Abhängigkeit zwischen diesen beiden Elementen besteht. Dies ist u.a. dann der Fall, wenn es eine synchrone Interaktion gibt, die von `x` nach `y` erfolgt:

```
dependingLayerUsage(x,y) :=
    containsSyncInteraction(x,y) ∨ kindOfUsing1(x,y) ∨ ...
```

Die durch einen Adapterausdruck konkretisierten abstrakten Begriffe können in der Beschreibung verschiedener Architekturbausteine auftreten. Daher ist ein Adapterausdruck nicht auf die

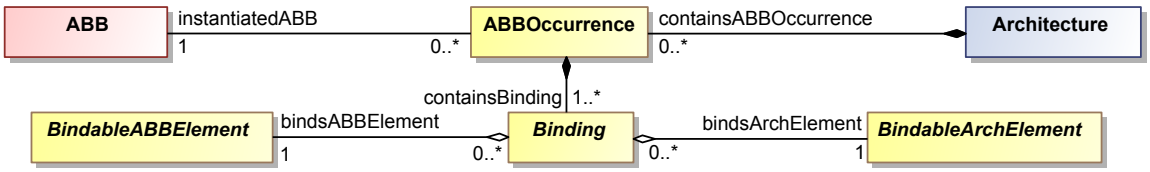


Abbildung 6.32: Generischer Anteil des Metamodells.

Verwendung in einem bestimmten Architekturbaustein beschränkt, sondern ist global verfügbar und kann von unterschiedlichen Architekturbausteinen genutzt werden. Erfordert ein neu zu formalisierender Baustein die Definition eines neuen Adapterausdrucks, wird dieser in die bestehende Menge der existierenden Adapterausdrücke integriert. Allerdings darf die Definition bestehender Adapterausdrücke nicht beliebig verändert werden, um Konflikte zu vermeiden.

6.5.2. Formulierung von Eigenschaftsbedingungen

Eigenschaftsbedingungen sind Teil der Beschreibung von Architekturbausteinen. Sie sollen daher unabhängig von der konkreten Architekturbeschreibung formuliert werden. In einem prädikatenlogischen Ausdruck, der eine Eigenschaftsbedingung wiedergibt, ist daher nur eine eingeschränkte Menge von Prädikaten zulässig. Hierbei handelt es sich um Prädikate, die

- den generischen Teil des querschnittlichen Metamodells abbilden (vgl. Abbildung 6.32)
- die Beschreibungselemente des Metamodell zur Beschreibung von Bausteinen abbilden (vgl. Abschnitt 6.3.1 und 6.3.4)
- einen Adapterausdruck (vgl. Abschnitt 6.5.1) repräsentieren
- auf Basis der zuvor genannten Arten von zulässigen Prädikaten definiert wurden

Eine Eigenschaftsbedingung muss in allen Instanzen des Bausteins gültig sein. Daher bezieht sie sich im Regelfall auch nur auf eine Instanz und macht keine instanzübergreifenden Aussagen. Entsprechend wird eine Eigenschaftsbedingung über all denjenigen Architekturelementen ausgewertet, die in diesen Instanzen an Bausteinelemente gebunden sind. Die prädikatenlogisch formulierten Eigenschaftsbedingungen nehmen daher außer auf den eigentlichen Baustein und seine bindbaren Elemente auch auf die Instanzen und Bindungen Bezug.

Aus der Betrachtung verschiedener Bausteinbedingungen lässt sich ein wiederkehrendes Schema ableiten, das eine Eigenschaftsbedingung zunächst in vier Bereiche unterteilt (vgl. Abbildung 6.33). Im ersten Bereich wird notiert, von welchem Architekturbaustein *alle* Instanzen betrachtet werden. Der zweite Bereich legt dann fest, welche Bausteinelemente für diese Bedingung relevant sind. Im Folgenden werden die Architekturelemente betrachtet, an die diese Bausteinelemente in der gerade betrachteten Bausteininstanz gebunden sind. In beiden Bereichen werden die Elemente durch allquantifizierende prädikatenlogische Ausdrücke identifiziert. Diese Ausdrücke werden durch die Nennung der Bezeichner der jeweiligen Bausteinelemente abkürzt. Der dritte Bereich ist nicht immer belegt. In diesem können die zu betrachtenden Elemente weiter eingrenzt und weitere Vorbedingungen gestellt werden. Der vierte Bereich umfasst schließlich die Eigenschaft, die gelten muss.

6.5.3. Beispiel: Eigenschaftsbedingung eines Architekturbausteins

In diesen Abschnitt wird am Beispiel des Pull-Observerbausteins die Formulierung einer Eigenschaftsbedingung verdeutlicht. Diese Bedingung wird zunächst vollständig in einen prädikaten-

ABBC *<Name ABB-Bedingung>*

<betrachteter Baustein> 
<betroffene Bausteinrollen und ihre Bindungen> 
<zusätzliche Vorbedingungen>
<zu geltende Eigenschaft>

Abbildung 6.33: Schema der Bausteinbedingungen.

logischen Satz übersetzt. Anschließend wird das Schema aus Abbildung 6.33 genutzt, um die Bedingung abkürzend darzustellen.

Der Pull-Observerbaustein definiert zwei Eigenschaftsbedingungen. Das Ziel der ersten Bedingung ist es, sicherzustellen, dass die Benachrichtigung der Observer als zentraler Verhaltensaspekt des Observerbausteins überhaupt erfolgt. Hierfür muss eine Interaktion, die an die Nachrichtenrolle notify gebunden ist, durch eine vorangehende Interaktion ausgelöst werden:

$$
\begin{aligned}
&\mathbf{ABBC}_{\textit{Pull-Observer1}} := \\
&\forall o \forall p : \texttt{ABBOccurrence}(o) \wedge \texttt{ABB}(p) \wedge \texttt{instantiatedABB}(o,p) \wedge \\
&\qquad \texttt{name}(p, \texttt{"Pull-Observerbaustein"}) \wedge \\
&\forall m : \texttt{Binding}(m) \wedge \texttt{containsBinding}(o,m) \wedge \\
&\forall r : \texttt{ABBMessage}(r) \wedge \texttt{name}(r, \texttt{"notify"}) \wedge \texttt{bindsABBElement}(m,r) \wedge \\
&\forall a : \texttt{BindableArchElement}(a) \wedge \texttt{bindsArchElement}(m,a) \rightarrow \\
&\texttt{boundHasActivatingMsg}(a)
\end{aligned}
$$

In der Bedingung werden zunächst alle Instanzen `o` des Architekturbausteins `Pull-Observerbaustein` betrachtet. Mit Hilfe aller Bindungen `m` dieser Instanz werden dann die Architekturelemente `a` identifiziert, an die die Nachrichtenrolle `notify` gebunden ist. Damit die Eigenschaftsbedingung gültig ist, muss schließlich auf alle diese Architekturelemente das Adapterprädikat `boundHasActivatingMsg(a)` zutreffen. Dabei identifiziert das Adapterprädikat `boundHasActivatingMsg(a)`, ob die Interaktion `a` einen Auslöser besitzt:

$$
\begin{aligned}
&\texttt{boundHasActivatingMsg}(a) := \texttt{MsgTrace}(a) \wedge \qquad [1](6.27) \\
&\quad \exists b \exists c : \texttt{MsgOccurrence}(b) \wedge \texttt{MsgOccurrence}(c) \wedge \qquad \text{(AP 1)} \\
&\quad \texttt{beginningOfMsgTrace}(b,a)^{1} \wedge \texttt{triggersMsgOcc}(c,b)
\end{aligned}
$$

Vollständig formuliert umfasst die Eigenschaftsbedingung $\mathbf{ABBC}_{\textit{Pull-Observer1}}$ mehrere Zeilen. Durch Verwendung des abkürzenden Schemas kann die Anzahl der zu notierenden Ausdrücke deutlich reduziert werden. Um die linke Seite der Bedingung auszudrücken, werden lediglich die Namen des Architekturbausteins und der betroffenen Bausteinrolle aufgeführt. Auf der rechten Seite der Bedingung wird der Parameter durch den Namen der Bausteinrolle ersetzt, die er repräsentiert.

ABBC$_{Pull\text{-}Observer1}$

"Pull-Observerbaustein"	
"notify"	

boundHasActivatingMsg("notify")

7. Formalisierung des bausteinbasierten Architekturentwurfs

Inhalt

In Kapitel 6 wurde das Metamodell für Modelle des bausteinbasierten Architekturentwurfs in Form von Klassendiagrammen definiert. Beispiele für Modelle dieses Metamodells wurden ebenfalls durch Diagramme dargestellt. Diese Modelle müssen Konsistenzbedingungen des Metamodells ebenso wie Eigenschaftsbedingungen der Bausteine einhalten. Beide Arten von Bedingungen wurden durch prädikatenlogische Sätze ausgedrückt. Um die Bedingungen zu überprüfen, müssen sowohl Metamodell als auch die Modelle basierend auf prädikatenlogischen Begriffen repräsentiert werden. Abschnitt 7.1 bietet dazu eine Übersicht über einige syntaktische und semantische Grundlagen der Prädikatenlogik, wie diese auch in verschiedenen Lehrbüchern vermittelt werden [EFT92, Sch00]. Das Metamodell und die Modelle des bausteinbasierten Architekturentwurfs werden auf Begriffe der Prädikatenlogik, wie Signatur oder Struktur, abgebildet. Dabei gibt Abschnitt 7.2 nicht nur die formale Repräsentation an sich an, sondern erläutert auch die Abbildung zwischen Modellen und Prädikatenlogik.

Ein Modell wird im Rahmen eines Entwicklungsschritts u.a. durch die Instanziierung eines Architekturbausteins verändert. Änderungen am Modell können durch Funktionen, die auf der formalen Repräsentation basieren, ebenfalls formal beschrieben werden (Abschnitt 7.3). Als Änderungsfunktionen werden neben der Instanziierung von Architekturbausteien auch Funktionen für das Hinzufügen oder Entfernen von Modellteilen definiert. Abschließend werden in Abschnitt 7.4 die in diesem Kapitel eingeführten formalen Konzepte an einem Beispiel erläutert. Dabei wird ein Modell formal repräsentiert, sowie ein Entwicklungsschritt ausgehend von diesem Modell durchgeführt.

7.1. Prädikatenlogische Grundlagen der formalen Beschreibung

Prädikatenlogische Ausdrücke werden nach festgelegten Regeln auf Basis verschiedener Zeichen und Symbole gebildet. Die Menge der Zeichen umfasst $\forall$, $\exists$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\neg$, (,) und $\equiv$. Hinzu kommen Variablensymbole v_0, v_1, v_2, ... und die Symbole der Signatur.

Definition 1 (Signatur $\mathcal{S}$ der Prädikatenlogik)
Eine *Signatur* $\mathcal{S} := \mathcal{F} \dot{\cup} \mathcal{R}$ besteht aus

- einer abzählbaren Menge von *Funktionssymbolen* $\mathcal{F}$ und
- einer abzählbaren Menge von *Relationssymbolen* $\mathcal{R}$.

Funktionssymbole besitzen eine Stelligkeit von $n \geq 0$, wobei 0-stellige Funktionssymbole auch als Konstanten bezeichnet werden. Die Stelligkeit von Relationssymbolen ist $n \geq 1$.

Die Prädikatenlogik unterscheidet Terme und Ausdrücke, die basierend auf diesen Zeichen und Symbolen induktiv gebildet werden. Ausdrücke bauen dabei auf Termen auf.

Definition 2 ($\mathcal{S}$-Term, $\mathcal{S}$-Ausdruck, $\mathcal{S}$-Satz)
Sei $\mathcal{S}$ eine Signatur. Als $\mathcal{S}$-*Term* bezeichnet wird

- jede Variable v_0, v_1, v_2, ...,
- jede Konstante aus $\mathcal{S}$ und
- jedes $\mathrm{f}(t_1, \ldots, t_n)$ mit $n \geq 1$-stelligem Funktionssymbol $\mathrm{f} \in \mathcal{F}$ und $\mathcal{S}$-Termen $t_1, \ldots, t_n$.

Ein $\mathcal{S}$-*Ausdruck* ist

- $t_1 \equiv t_2$ für $\mathcal{S}$-Terme t_1 und t_2,
- $\mathrm{R}(t_1, \ldots, t_n)$ für ein n-stelliges Relationssymbol $\mathrm{R} \in \mathcal{R}$ und $\mathcal{S}$-Terme $t_1, \ldots, t_n$,
- $\neg\varphi$ für $\mathcal{S}$-Ausdruck φ,
- $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$ und $(\varphi \leftrightarrow \psi)$ für $\mathcal{S}$-Ausdrücke φ und ψ und
- $\forall \mathrm{x} : \varphi$ und $\exists \mathrm{x} : \varphi$ für $\mathcal{S}$-Ausdruck φ und Variable x.

Enthält ein $\mathcal{S}$-*Ausdruck* φ keine freien Variablen, wird φ auch als $\mathcal{S}$-Satz bezeichnet.

Die Auswertung von prädikatenlogischen Aussagen, d.h. die Bestimmung des Wahrheitswertes einer Aussage, erfolgt auf Basis einer mathematischen Struktur. Hierzu werden zunächst die Funktions- und Relationssymbole der Signatur auf Funktionen bzw. Relationen über einer gemeinsamen Trägermenge abgebildet. Diese Abbildung wird auch als Interpretation der Signatursymbole bezeichnet.

Definition 3 ($\mathcal{S}$-Struktur)
Sei $\mathcal{S}$ eine Signatur. Eine $\mathcal{S}$-*Struktur* $\mathfrak{S} := (\mathcal{A}, \alpha)$ besteht aus

- einer nichtleeren *Trägermenge* $\mathcal{A}$ und
- einer auf $\mathcal{S}$ definierten Abbildung α, die
  - jedes n-stellige Funktionensymbol $\mathrm{f} \in \mathcal{F}$ auf eine n-stellige *Funktion* $\alpha(\mathrm{f}) : \mathcal{A}^n \rightarrow \mathcal{A}$ und
  - jedes n-stellige Relationssymbol $\mathrm{R} \in \mathcal{R}$ auf eine n-stellige *Relation* $\alpha(\mathrm{R}) \subseteq \mathcal{A}^n$ abbildet.

Anstelle von $\alpha(\mathrm{f})$ und $\alpha(\mathrm{R})$ wird auch $\mathrm{f}^{\mathfrak{S}}$ bzw. $\mathrm{R}^{\mathfrak{S}}$ als abkürzende Schreibweise vereinbart. Die Mengen $\mathcal{F}^{\mathfrak{S}} := \{\mathrm{f}^{\mathfrak{S}} \mid \mathrm{f} \in \mathcal{F}\}$ und $\mathcal{R}^{\mathfrak{S}} := \{\mathrm{R}^{\mathfrak{S}} \mid \mathrm{R} \in \mathcal{R}\}$ werden analog notiert.

Neben den Symbolen der Signatur kann ein prädikatenlogischer Ausdruck Variablen enthalten. Der Wahrheitswert eines Ausdrucks mit Variablen hängt dann von den konkreten Werten ab, die diesen Variablen zugewiesen werden. Um einen prädikatenlogischen Ausdruck auswerten (interpretieren) zu können, muss neben einer Struktur folglich eine Belegung der Variablen existieren.

Definition 4 ($\mathcal{S}$-Interpretation)
Sei $\mathcal{S}$ eine Signatur. Eine $\mathcal{S}$-*Interpretation* $\mathfrak{I} := (\mathfrak{S}, \beta)$ besteht aus

- einer $\mathcal{S}$-Struktur $\mathfrak{S}$ und
- einer Belegungsfunktion $\beta : \{v_i \mid i \in \mathbb{N}\} \to \mathcal{A}$.

Durch eine $\mathcal{S}$-Interpretation wird ein $\mathcal{S}$-Ausdruck bezüglich seines (Wahrheits-)Wert ausgewertet. Hierzu wird zunächst der in $\mathcal{A}$ liegende Wert jedes $\mathcal{S}$-Terms bestimmt durch

- $\mathfrak{I}(x) := \beta(x)$ für Variablen x aus dem Definitionsbereich von β und
- $\mathfrak{I}(\mathrm{f}(t_1, \ldots, t_n)) := \mathrm{f}^{\mathfrak{S}}(\mathfrak{I}(t_1), \ldots, \mathfrak{I}(t_n))$ für Terme $t_1, \ldots, t_n$ und ein Funktionssymbol $\mathrm{f} \in \mathcal{F}$ der Signatur $\mathcal{S}$.

Der Wahrheitswert eines Ausdrucks wird induktiv über seinen Aufbau definiert; hierbei seien φ und ψ $\mathcal{S}$-Ausdrücke, t_i $\mathcal{S}$-Terme, $\mathrm{R} \in \mathcal{R}$ ein Relationssymbol der Signatur $\mathcal{S}$ und x eine Variable:

- $\mathfrak{I}(t_1 \equiv t_2) = \begin{cases} wahr & \text{falls } \mathfrak{I}(t_1) \equiv \mathfrak{I}(t_2) \\ falsch & \text{sonst} \end{cases}$
- $\mathfrak{I}(\mathrm{R}(t_1, \ldots, t_n)) = \begin{cases} wahr & \text{falls } (\mathfrak{I}(t_1), \ldots, \mathfrak{I}(t_n)) \in \mathrm{R}^{\mathfrak{S}} \\ falsch & \text{sonst} \end{cases}$
- $\mathfrak{I}(\neg\varphi) = \begin{cases} wahr & \text{falls } \mathfrak{I}(\varphi) = false \\ falsch & \text{sonst} \end{cases}$
- $\mathfrak{I}(\varphi \wedge \psi) = \begin{cases} wahr & \text{falls } \mathfrak{I}(\varphi) = true \text{ und } \mathfrak{I}(\psi) = true \\ falsch & \text{sonst} \end{cases}$
- $\mathfrak{I}(\varphi \vee \psi) = \begin{cases} wahr & \text{falls } \mathfrak{I}(\varphi) = true \text{ oder } \mathfrak{I}(\psi) = true \\ falsch & \text{sonst} \end{cases}$
- $\mathfrak{I}(\forall \mathrm{x} : \varphi) = \begin{cases} wahr & \text{falls für alle } a \in \mathcal{A} \text{ gilt: } \mathfrak{I}_{[x/a]}(\varphi) = true \\ falsch & \text{sonst} \end{cases}$
- $\mathfrak{I}(\exists \mathrm{x} : \varphi) = \begin{cases} wahr & \text{falls ein } a \in \mathcal{A} \text{ existiert mit: } \mathfrak{I}_{[x/a]}(\varphi) = true \\ falsch & \text{sonst} \end{cases}$

Die Beziehung zwischen einer $\mathcal{S}$-Interpretation $\mathfrak{I}$ und einem in dieser als *wahr* ausgewerteten $\mathcal{S}$-Ausdruck φ wird als Modellbeziehung bezeichnet. Abhängig von zusätzlichen Voraussetzungen kann die Modellbeziehung auf Mengen von $\mathcal{S}$-Ausdrücken und auf $\mathcal{S}$-Strukturen ausgedehnt werden.

Definition 5 (Modell von)
Gilt für einen $\mathcal{S}$-Ausdruck φ die Gleichung $\mathfrak{I}(\varphi) = wahr$, wird die $\mathcal{S}$-Interpretation $\mathfrak{I}$ als *Modell von* φ bezeichnet. Dies wird ebenfalls als $\mathfrak{I}$ *erfüllt* φ oder φ *gilt bei* $\mathfrak{I}$ gesprochen und als $\mathfrak{I} \models \varphi$ notiert.
Die Modellbeziehung kann auch auf eine Menge von $\mathcal{S}$-Ausdrücken Φ erweitert werden, wenn $\mathfrak{I} \models \varphi$ für jedes $\varphi \in \Phi$ gilt. Notiert wird dies als $\mathfrak{I} \models \Phi$.
Ist φ ein $\mathcal{S}$-Satz, ist die Belegung β unerheblich. Die $\mathcal{S}$-Struktur $\mathfrak{S}$ kann dann auch als *Modell von* φ bezeichnet werden ($\mathfrak{S} \models \varphi$). Für eine Menge Φ von $\mathcal{S}$-Sätzen gilt entsprechend $\mathfrak{S} \models \Phi$, wenn $\mathfrak{S} \models \varphi$ für jedes $\varphi \in \Phi$ gilt.

Zu einer Menge von $\mathcal{S}$-Sätzen kann eine Menge von $\mathcal{S}$-Strukturen existieren, die im prädikatenlogischen Sinn alle Modelle dieser Satzmenge sind. Eine solche Menge von Modellen wird als Modellklasse bezeichnet:

Definition 6 (Modellklasse, Axiomensystem)
Für eine Menge Φ von $\mathcal{S}$-Sätzen ist

$$\mathfrak{K} := \{\mathfrak{S} \mid \mathfrak{S} \text{ ist } \mathcal{S}\text{-Struktur und } \mathfrak{S} \models \Phi\}$$

als die Modellklasse $\mathfrak{K}$ von Φ definiert.
Die Menge Φ wird auch als *Axiomensystem* der Modellklasse $\mathfrak{K}$ bezeichnet.

7.2. Formale Beschreibung von bausteinbasierten Architekturen

Metamodell und Modelle des bausteinbasierten Architekturentwurfs können mit den in Abschnitt 7.1 erläuterten Grundlagen der Prädikatenlogik repräsentiert werden. Die Abbildung von Modellen auf prädikatenlogische Konzepte wurde bereits in verschiedenen Ansätzen durchgeführt [SZ08, Her11]. In der vorliegenden Arbeit erfolgt die Abbildung von Metamodell und Modell ähnlich zu der in [Her11]. Weil sich u.a. das Metamodell der Arbeit von [Her11] von dem hier vorliegenden Metamodell unterscheidet, werden zusätzlich einige Ergänzungen vorgenommen. Darüber hinaus wird das vorliegende Metamodell logisch in drei Bereiche unterteilt, die auch in der prädikatenlogischen Repräsentation berücksichtigt werden. Signaturen repräsentieren dabei die durch das Metamodell festgelegten Sprachelemente. Zusätzlich bildet ein Axiomensystem die im Metamodell definierten Bedingungen zwischen den Sprachelementen ab (Abschnitt 7.2.1). Ein Modell wird durch eine Struktur repräsentiert. Eine solche Struktur ist über derjenigen Signatur definiert ist, die das Metamodell des Modells repräsentiert. Zudem muss diese Struktur das Axiomensystem erfüllen (Abschnitt 7.2.2), das die Bedingungen des jeweiligen Metamodells abbildet. Eigenschaftsbedingungen eines Architekturbausteins werden auf ähnliche Weise wie die Bedingungen des Metamodells formuliert. Diese müssen durch Strukturen erfüllt werden, die bausteinkonsistente Modelle repräsentieren (Abschnitt 7.2.3).

7.2.1. Repräsentation des Metamodells

Ein Metamodell definiert Entitätentypen und modelliert u.a. durch Assoziationen Beziehungen zwischen diesen Entitätentypen. Die Entitätentypen und explizit modellierten Beziehungen bilden die Sprachelemente des Metamodells. Formal werden sie hier durch Relationssymbole einer Signatur repräsentiert.

Das Metamodell für den bausteinbasierten Architekturentwurf (*BBAE*) wurde in Kapitel 6 in drei Teilmetamodelle unterteilt. Dabei ist jedes Element des kompletten Metamodells genau einem Teilmetamodell zugeteilt. Eine entsprechende Aufteilung erfolgt auch bei der formalen Repräsentation des Metamodells durch Signaturen.

Definition 7 (Signaturen des Metamodells)
Die den Teilmetamodellen zugeordneten Signaturen werden wie folgt definiert, wobei $\mathcal{F}$ und $\mathcal{R}$ eine Menge von Funktionssymbolen bzw. von Relationssymbolen bezeichnen:

- $\mathcal{S}_\mathrm{A} := \mathcal{F}_\mathrm{A} \dot\cup \mathcal{R}_\mathrm{A}$ für den Architekturteil
- $\mathcal{S}_\mathrm{B} := \mathcal{F}_\mathrm{B} \dot\cup \mathcal{R}_\mathrm{B}$ für den Bausteinteil

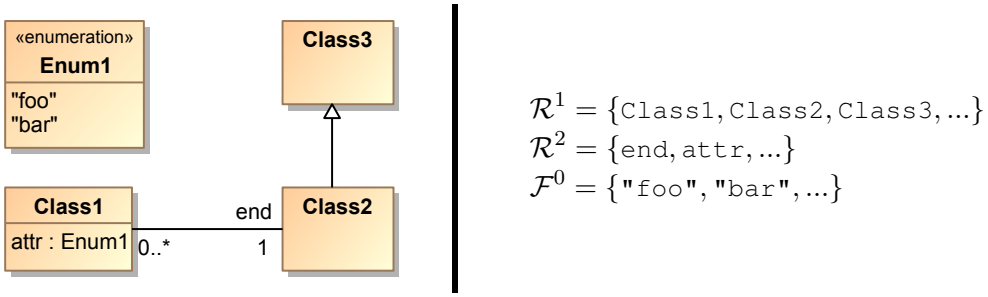


Abbildung 7.1: Beispiel für die Abbildung eines Klassendiagramms auf Symbole einer Signatur.

- $\mathcal{S}_{\mathrm{I}} := \mathcal{F}_{\mathrm{I}} \dot{\cup} \mathcal{R}_{\mathrm{I}}$ für den Integrationsteil

Die syntaktischen Elemente des vollständigen Metamodells werden durch die disjunkte Vereinigung der Teilsignaturen repräsentiert:

$$\mathcal{S}_{\mathrm{BBAE}} := \mathcal{S}_{\mathrm{A}} \dot{\cup} \mathcal{S}_{\mathrm{B}} \dot{\cup} \mathcal{S}_{\mathrm{I}}$$

Für die Definition des Metamodells wurden in Kapitel 6 Klassendiagramme verwendet. Aus diesen Diagrammen werden die konkreten Symbole der Signaturen ableitet. Dabei kommen in erster Linie unäre und binäre Relationssymbole sowie 0-stellige Funktionssymbole (Konstantensymbole) zum Einsatz. Anders als in [Her11] werden in einigen Fällen zusätzlich auch ternäre Relationssymbole verwendet.

1) *Unäre Relationssymbole* repräsentieren Entitätentypen (`<NameEntitätentyp>`).
2) *Binäre Relationssymbole* werden abgeleitet aus
   - Assoziationen zwischen zwei Entitätentypen, wenn ein benanntes Assoziationsende existiert (`<NameAssoziationsende>`)
   - Attributen von Entitätentypen (`<NameAttribut>`)
3) *Ternäre* oder ggf. auch *n-näre Relationssymbole* repräsentieren Assoziationen mit zusätzlichen Eigenschaften (vgl. `nextABBMessage` in Abschnitt 6.3.3).
4) *Konstantensymbole* bilden Werte von Attributen wie Namen in die Signatur ab.

Abbildung 7.1 veranschaulicht die Zuordnung anhand eines Beispiels: Verschiedene Elemente eines Klassendiagramms werden auf Symbole einer Signatur und mit unterschiedlicher Stelligkeit abgebildet.

Die durch Symbole einer Signatur repräsentierten Elemente sind nur ein Aspekt des Metamodells, der durch prädikatenlogische Konzepte formalisiert wird. Zusätzlich definiert ein Metamodell Bedingungen bezüglich einzelner Elemente oder bezüglich der Beziehung zwischen Elementen. Ein Teil dieser Bedingungen hat generischen Charakter und ist direkt aus den Klassendiagrammen ableitbar, die zur Beschreibung des Metamodells genutzt wurden:

1) *Vererbungsbeziehungen* zwischen Entitätentypen werden in den Klassendiagrammen des Metamodells explizit modelliert. Ein Element vom Typ des abgeleiteten Entitätentyps ist ebenfalls vom Typ des Superentitätentyps. Generisch wird dieser Sachverhalt durch Bedingungen der Art

$$\mathbf{CV}_i := \forall \mathtt{x} : \texttt{<Entitätentyp>}(\mathtt{x}) \rightarrow \texttt{<SuperEntitätentyp>}(\mathtt{x})$$

formuliert.
Für das Beispiel in Abbildung 7.1 wird $\forall x : \texttt{Class2}(x) \rightarrow \texttt{Class3}(x)$ verlangt.

2) Eine *Assoziation* setzt verschiedene Entitätentypen in Beziehung zueinander. Der Bezug einer Assoziation zu ihren Entitätentypen geht durch ihre Abbildung auf ein Relationssymbol allerdings verloren. Um dennoch den *Typbezug* sicherzustellen, werden entsprechende Bedingungen formuliert:

$$\mathbf{CT}_i := \forall x \forall y : \texttt{<Assoziationsende>}(x,y) \rightarrow \texttt{<Entitätentyp1>}(x) \wedge \texttt{<Entitätentyp2>}(y)$$

Dabei wird festgelegt, dass die erste Tupelkomponente des Relationssymbols <Assoziationsende> für das Element auf der gegenüberliegenden Seite des Assoziationsende steht und die zweite für das Element direkt am benannten Assoziationsende.
Aus dem Beispiel in Abbildung 7.1 wird die Bedingung $\forall x \forall y : \texttt{end}(x,y) \rightarrow \texttt{Class1}(x) \wedge \texttt{Class2}(y)$ abgeleitet.

3) Über die *Multiplizitäten* an den Assoziationsenden wird angegeben, wie viele Elemente der gegenüberliegenden Entitätentypen jeweils in Beziehung zueinander stehen können. Dabei kann eine Multiplizität durch einen einzelnen Wert oder durch ein Intervall festgelegt werden. Während [Her11] an dieser Stelle vier verschiedene Fälle von Multiplizitäten betrachtet, wird hier bei der Formulierung der Bedingungen verallgemeinert und nur zwischen Einzelwerten und Intervallen unterschieden.

- Wird die Multiplizität durch einen einzelnen Wert bestimmt, wird die zugehörige Bedingung nach folgendem Schema formuliert:

[1] (6.21)

$$\mathbf{CM}_i := \forall x \exists v : \texttt{<Entitätentyp1>}(x) \wedge \texttt{count}(\texttt{<Entitätentyp2>}(y) \wedge \texttt{<Assoziationsende>}(x,y), v)^{1} \rightarrow (\texttt{<MultiplizitätsWert>} \equiv v)$$

Für das Beispiel lautet die entsprechende Bedingung
$\forall x \exists v : \texttt{Class1}(x) \wedge \texttt{count}(\texttt{Class2}(y) \wedge \texttt{end}(x,y), v) \rightarrow (1 \equiv v)$.

- Erstreckt sich eine Multiplizität über ein Intervall, erfolgt die Formulierung der Bedingung nach folgendem Schema:

[1] (6.21), [2] (6.20)

$$\mathbf{CM}_i := \forall x \exists v : \texttt{<Entitätentyp1>}(x) \wedge \texttt{count}(\texttt{<Entitätentyp2>}(y) \wedge \texttt{<Assoziationsende>}(x,y), v)^{1} \rightarrow \texttt{containsRangeValue}(\texttt{<MultiplizitätsIntervall>}, v)^{2}$$

Eine entsprechende Bedingung für die mehrwertige Multiplizität aus Abbildung 7.1 lautet $\forall y \exists v : \texttt{Class2}(y) \wedge \texttt{count}(\texttt{Class1}(x) \wedge \texttt{end}(x,y), v) \rightarrow \texttt{containsRangeValue}(\texttt{[0,*)}, v)$.

4) Im Gegensatz zu [Her11] werden im Metamodell dieser Arbeit auch Enumerationen verwendet. *Enumerationen* sind spezielle Datentypen, die individuell definiert werden können und eine endliche Menge von Werten festlegen. Ein Attribut vom Typ einer Enumeration kann nur einen seiner Werte annehmen. Diese Werteinschränkungen implizieren Bedingungen über allen Entitätentypen, die mit Enumerationen getypte Attribute besitzen. Solche Bedingungen sind nach folgendem Muster aufgebaut:

$$\mathbf{CE}_i := \forall x \forall v : \texttt{<Entitätentyp>}(x) \wedge \texttt{<Attributname>}(x,v) \rightarrow (v \equiv \texttt{<Enumerationswert1>}) \vee (v \equiv \texttt{<Enumerationswert2>}) \vee \ldots$$

Für das Attribut aus dem Beispiel wird folglich die Bedingung $\forall x \forall v : \texttt{Class1}(x) \wedge \texttt{attr}(x,v) \rightarrow (v \equiv \texttt{"foo"}) \vee (v \equiv \texttt{"bar"})$ abgeleitet.

Bedingungen, die nach den obigen Schemata formuliert werden, erhalten eine eindeutige Bezeichnung. Diese Bezeichnung enthält die Kategorie der Bedingung (**V**ererbung, **T**ypbezug, **M**ultiplizität und **E**numeration) und wird um eine fortlaufende Nummer ergänzt. Der Bezeichnung wird zudem je nach zugehörigem Teilmetamodell (**A**rchitektur, **B**austein, **I**ntegration) ein entsprechender Buchstabe vorangestellt ($\mathbf{ACV}_i$, $\mathbf{BCV}_i$, $\mathbf{ICV}_i$ und $\mathbf{ACT}_i$, $\mathbf{BCT}_i$, $\mathbf{ICT}_i$ sowie $\mathbf{ACM}_i$, $\mathbf{BCM}_i$, $\mathbf{ICM}_i$ und $\mathbf{ACE}_i$, $\mathbf{BCE}_i$, $\mathbf{ICE}_i$). Ergänzend zu den aus den Klassendiagrammen ableitbaren Bedingungen wurden in Kapitel 6 im Rahmen der jeweiligen Teilmetamodelle weitere Bedingungen definiert. Diese Bedingungen $\mathbf{AC_1}$, ..., $\mathbf{AC_{32}}$, $\mathbf{BC_1}$, ..., $\mathbf{BC_9}$ und $\mathbf{IC_1}$, ..., $\mathbf{IC_{34}}$ wurden bereits als prädikatenlogische Sätze formuliert. Jedes gültige Modell des bausteinbasierten Entwurfs muss alle diese Bedingungen erfüllen. Bezeichnet wird die Menge der Bedingungen als Axiomensystem.

Definition 8 (Axiomensysteme des Metamodells)
Die Axiomensysteme über die Teilmetamodelle sind wie folgt definiert, wobei die Indexmengen alle Bedingungen des jeweiligen Bedingungtyps indizieren:

- Die Bedingungen des Architekturteils Φ_A sind eine Menge von $\mathcal{S}_\mathrm{A}$-Sätzen:

$$\Phi_\mathrm{A} := \{\mathbf{ACV}_i \mid i \in \mathbb{I}_{ACV}\} \dot\cup \{\mathbf{ACT}_i \mid i \in \mathbb{I}_{ACT}\} \dot\cup \{\mathbf{ACM}_i \mid i \in \mathbb{I}_{ACM}\} \dot\cup \{\mathbf{ACE}_i \mid i \in \mathbb{I}_{ACE}\} \dot\cup \{\mathbf{AC}_i \mid i \in \mathbb{I}_{AC}\}$$

- Die Bedingungen des Bausteinteils Φ_B sind eine Menge von $\mathcal{S}_\mathrm{B}$-Sätzen:

$$\Phi_\mathrm{B} := \{\mathbf{BCV}_i \mid i \in \mathbb{I}_{BCV}\} \dot\cup \{\mathbf{BCT}_i \mid i \in \mathbb{I}_{BCT}\} \dot\cup \{\mathbf{BCM}_i \mid i \in \mathbb{I}_{BCM}\} \dot\cup \{\mathbf{BCE}_i \mid i \in \mathbb{I}_{BCE}\} \dot\cup \{\mathbf{BC}_i \mid i \in \mathbb{I}_{BC}\}$$

- Die Bedingungen des Integrationsteils Φ_I sind eine Menge von $\mathcal{S}_\mathrm{BBAE}$-Sätzen:

$$\Phi_\mathrm{I} := \{\mathbf{ICV}_i \mid i \in \mathbb{I}_{ICV}\} \dot\cup \{\mathbf{ICT}_i \mid i \in \mathbb{I}_{ICT}\} \dot\cup \{\mathbf{ICM}_i \mid i \in \mathbb{I}_{ICM}\} \dot\cup \{\mathbf{ICE}_i \mid i \in \mathbb{I}_{ICE}\} \dot\cup \{\mathbf{IC}_i \mid i \in \mathbb{I}_{IC}\}$$

Die Menge aller Bedingungen des Metamodells wird durch Φ_M zusammengefasst:

$$\Phi_\mathrm{M} := \Phi_\mathrm{A} \dot\cup \Phi_\mathrm{B} \dot\cup \Phi_\mathrm{I}$$

7.2.2. Repräsentation eines Modells

Allgemein wird ein Modell auf Basis der im Metamodell definierten Sprachelemente formuliert und muss die Bedingungen erfüllen, die durch das Metamodell vorgegeben werden. Hier wird ein Modell formal durch eine Struktur abgebildet, deren zugrunde liegende Signatur aus den Sprachelementen des Metamodells abgeleitet ist. Eine Struktur, die ein Modell repräsentiert, muss konform zum Metamodell des Modells sein. Dies ist gegeben, wenn die Struktur das Axiomensystem erfüllt, das aus den Bedingungen des Metamodells gebildet wird.

Im bausteinbasierten Architekturentwurf besteht ein (Gesamt-)Modell aus der Beschreibung von Architekturbausteinen, der Architektur eines Softwaresystems und der zugehörigen Integrationsbeschreibung. Eine entsprechende Repräsentation erfolgt durch eine Struktur über der Signatur $\mathcal{S}_\mathrm{BBAE}$. Als zentraler und wiederverwendbarer Teil dieses Ansatzes wird ein einzelner Baustein ebenfalls als eigenständiges Modell betrachtet und durch eine Struktur über der Signatur $\mathcal{S}_\mathrm{B}$ formalisiert.

Definition 9 (Repräsentation eines Architekturbausteins)
Eine formale Repräsentation eines Architekturbausteins[5] ist eine $\mathcal{S}_\mathfrak{B}$-Struktur $\mathfrak{B} := (\mathcal{A}_\mathfrak{B}, \alpha_\mathfrak{B})$ mit $\mathcal{S}_\mathfrak{B} = \mathcal{F}_\mathfrak{B} \dot\cup \mathcal{R}_\mathfrak{B}$, für die gilt, dass

- die Relation, auf die das Relationssymbol $\texttt{ABB} \in \mathcal{R}_\mathfrak{B}$ durch $\alpha_\mathfrak{B}$ abgebildet wird, einelementig ist ($|\texttt{ABB}^\mathfrak{B}| = 1$) und
- die Struktur $\mathfrak{B}$ ein Modell des Axiomensystems $\Phi_\mathfrak{B}$ ($\mathfrak{B} \models \Phi_\mathfrak{B}$) ist.

Die Architektur eines Systems wird im bausteinbasierten Architekturentwurf basierend auf beliebig vielen verschiedenen Architekturbausteinen erstellt. Die Menge der insgesamt für den Entwurf zur Verfügung stehenden Bausteine wird als *Bausteinkatalog* bezeichnet. Ein Bausteinkatalog ist eine Teilmenge aller gültigen Bausteinmodelle und damit formal eine Teilmenge der Modellklasse des Axiomensystems $\Phi_\mathfrak{B}$, deren Elemente $\mathcal{S}_\mathfrak{B}$-Strukturen $\mathfrak{B}_i$ sind. Für den formalen Bausteinkatalog ist zudem sicherzustellen, dass sich die Namensräume der enthaltenen Bausteine nicht überschneiden. Daher wird die als Bausteinkatalog bezeichnete Teilmenge zusätzlich bezüglich der Trägermengen $\mathcal{A}_{\mathfrak{B}_i}$ der enthaltenen $\mathfrak{B}_i$ eingeschränkt. Diese Einschränkung ist notwendig, da die Formalisierung eines Gesamtmodells ebenfalls durch eine Struktur realisiert wird. Deren Trägermenge besteht dann u.a. aus der Vereinigung der Trägermengen der Bausteinmodelle. Überschneidungen der Namensräume können dazu führen, dass das Axiomensystem $\Phi_\mathfrak{B}$ formal nicht mehr erfüllt wird.

Definition 10 (Bausteinkatalog)
Ein Bausteinkatalog $\mathfrak{K}_\mathfrak{B}$ ist zunächst eine Teilmenge der Menge aller Bausteine $\mathfrak{B}$ (wobei diese Menge wiederum Teilmenge der Modellklasse von $\Phi_\mathfrak{B}$ ist):

$$\mathfrak{K}_\mathfrak{B} \subseteq \{\mathfrak{B} \mid \mathfrak{B} \text{ ist } \mathcal{S}_\mathfrak{B}\text{-Struktur}, \mathfrak{B} \models \Phi_\mathfrak{B} \text{ und } |\texttt{ABB}^\mathfrak{B}| = 1\}$$

Ist $\mathfrak{K}_\mathfrak{B}$ ein Bausteinkatalog, dann müssen des Weiteren für zwei beliebige Bausteine $\mathfrak{B}_i \in \mathfrak{K}_\mathfrak{B}$ und $\mathfrak{B}_j \in \mathfrak{K}_\mathfrak{B}$ mit $\mathfrak{B}_i = (\mathcal{A}_{\mathfrak{B}_i}, \alpha_{\mathfrak{B}_i})$ und $\mathfrak{B}_j = (\mathcal{A}_{\mathfrak{B}_j}, \alpha_{\mathfrak{B}_j})$ sowie $i \neq j$ folgende Bedingungen gelten (*beachte*: $x^\mathfrak{B}$ entspricht $\alpha_\mathfrak{B}(x)$):

- Jede Konstante (0-stelliges Funktionssymbole) $\mathrm{c} = \mathrm{f}^0 \in \mathcal{F}_\mathfrak{B} \subseteq \mathcal{S}_\mathfrak{B}$ wird durch $\alpha_{\mathfrak{B}_i}$ und $\alpha_{\mathfrak{B}_j}$ auf identische Elemente in $\mathcal{A}_{\mathfrak{B}_i}$ und $\mathcal{A}_{\mathfrak{B}_j}$ abgebildet:
$$\forall \mathrm{c} \in \mathcal{F}_\mathfrak{B} : \mathrm{c}^{\mathfrak{B}_i} = \mathrm{c}^{\mathfrak{B}_j}$$
- Jedes n-stellige Relationssymbol $\mathrm{R} \in \mathcal{R}_\mathfrak{B} \subseteq \mathcal{S}_\mathfrak{B}$ wird durch $\alpha_{\mathfrak{B}_i}$ und $\alpha_{\mathfrak{B}_j}$ auf schnittfreie Relationen $\mathrm{R}^{\mathfrak{B}_i} \subseteq (\mathcal{A}_{\mathfrak{B}_i})^n$ und $\mathrm{R}^{\mathfrak{B}_j} \subseteq (\mathcal{A}_{\mathfrak{B}_j})^n$ abgebildet:
$$\forall \mathrm{R} \in \mathcal{R}_\mathfrak{B} : \mathrm{R}^{\mathfrak{B}_i} \cap \mathrm{R}^{\mathfrak{B}_j} = \emptyset$$
- Die Trägermengen $\mathcal{A}_{\mathfrak{B}_i}$ und $\mathcal{A}_{\mathfrak{B}_j}$ sind disjunkt bis auf die Abbildung der Konstanten aus $\mathcal{S}_\mathfrak{B}$:
$$\mathcal{A}_{\mathfrak{B}_i} \cap \mathcal{A}_{\mathfrak{B}_j} = \mathrm{c}^{\mathfrak{B}_i} = \mathrm{c}^{\mathfrak{B}_j}$$

Das Gesamtmetamodell ist derart gestaltet, dass das Teilmetamodell für die Architekturbeschreibung ebenso unabhängig wie das Teilmetamodell für die Bausteinbeschreibung betrachtet werden kann. Auch wenn es nicht dem Konzept des bausteinbasierten Entwurfs entspricht, ist

[5] Abkürzend wird ggf. statt von der Repräsentation eines Architekturbausteins auch nur von einem Architekturbaustein gesprochen, wenn eine Unterscheidung zwischen Modell und Repräsentation aus dem Kontext hervorgeht.

die Modellierung einer Architektur ohne eine Bausteininstanz gültig bezüglich des Gesamtmetamodells. Die formale Repräsentation eines Architekturmodells erfolgt ähnlich zu der eines Bausteinmodells.

Definition 11 (Repräsentation eines Architekturmodells)
Eine formale Repräsentation eines Architekturmodells[6] ist eine $\mathcal{S}_{\mathrm{A}}$-Struktur $\mathfrak{A} := (\mathcal{A}_{\mathfrak{A}}, \alpha_{\mathfrak{A}})$, für die gilt, dass

- die Relation, auf die das Relationssymbol $\texttt{Architecture} \in \mathcal{R}_{\mathrm{A}}$ der Signatur $\mathcal{S}_{\mathrm{A}}$ durch $\alpha_{\mathfrak{A}}$ abgebildet wird, einelementig ist ($|\texttt{Architecture}^{\mathfrak{A}}| = 1$) und
- die Struktur $\mathfrak{A}$ ein Modell des Axiomensystems Φ_{A} ($\mathfrak{A} \models \Phi_{\mathrm{A}}$) ist.

Entsprechend der vorangegangenen Definition liegen die formalen Repräsentationen von Bausteinen und die formale Repräsentation der Architektur eines Gesamtmodells als separate formale Modelle vor. Der Integrationsteil eines Gesamtmodells bildet hingegen kein eigenständiges formales Modell in Form einer Struktur. Er wird lediglich als Teil des Gesamtmodells durch eine spezifische Trägermenge $\mathcal{A}_{\mathfrak{I}}$ und eine entsprechende Abbildung $\alpha_{\mathfrak{I}}$ repräsentiert. Das Tupel $(\mathcal{A}_{\mathfrak{I}}, \alpha_{\mathfrak{I}})$ erfüllt allerdings nicht die Kriterien, die an eine Struktur gestellt werden. Es wird zwar eine Trägermenge und eine Abbildung definiert, aber die Zielmenge der Abbildung erstreckt sich auch über die Trägermengen von Architektur und Bausteinen. Dies ist auf den verbindenden Charakter des Integrationsteils zurückzuführen (vgl. auch die Darstellung des Metamodells als Klassendiagramm in Abbildung 6.26).

Definition 12 (Repräsentation eines (gültigen) Gesamtmodells des bausteinbasierten Architekturentwurfs und Modellklasse des Gesamtmodells)
Eine formale Repräsentation eines Gesamtmodells[7] des bausteinbasierten Architekturentwurfs ist eine $\mathcal{S}_{\mathrm{BBAE}}$-Struktur $\mathfrak{M} := (\mathcal{A}_{\mathfrak{M}}, \alpha_{\mathfrak{M}})$ mit

- einer Trägermenge $\mathcal{A}_{\mathfrak{M}}$ als disjunkte Vereinigung der Trägermengen aller Bausteinmodelle des Bausteinkatalogs $\mathcal{A}_{\mathfrak{B}} = \left(\bigcup_{\mathfrak{B}_i \in \mathfrak{K}_{\mathfrak{B}}} \mathcal{A}_{\mathfrak{B}_i}\right)$, des Architekturmodells $\mathcal{A}_{\mathfrak{A}}$ und des Integrationsanteils $\mathcal{A}_{\mathfrak{I}}$:

  $$\mathcal{A}_{\mathfrak{M}} := (\mathcal{A}_{\mathfrak{B}} \dot{\cup} \mathcal{A}_{\mathfrak{A}}) \cup \mathcal{A}_{\mathfrak{I}}$$

  wobei der Schnitt der Trägermenge $\mathcal{A}_{\mathfrak{I}}$ eingeschränkt ist durch:
  - $\mathcal{A}_{\mathfrak{A}} \cap \mathcal{A}_{\mathfrak{I}} = \{a \,|\, a \in \mathcal{A}_{\mathfrak{A}} \wedge\ a \in \texttt{BindableArchElement}^{\mathfrak{I}}\}$
  - $\mathcal{A}_{\mathfrak{B}} \cap \mathcal{A}_{\mathfrak{I}} = \{a \,|\, a \in \mathcal{A}_{\mathfrak{B}} \wedge\ a \in \texttt{BindableABBElement}^{\mathfrak{I}}\}$
- und einer kumulierten Abbildung der Signatursymbole $x \in \mathcal{S}_{\mathrm{BBAE}}$

  $$\alpha_{\mathfrak{M}}(x) := \begin{cases} \alpha_{\mathfrak{A}}(x) & \text{für } x \in \mathcal{S}_{\mathrm{A}} \\ \bigcup\limits_{\mathfrak{B}_i \in \mathfrak{K}_{\mathfrak{B}}} \alpha_{\mathfrak{B}_i}(x) & \text{für } x \in \mathcal{S}_{\mathrm{B}} \\ \alpha_{\mathfrak{I}}(x) & \text{für } x \in \mathcal{S}_{\mathrm{I}} \end{cases}$$

  mit Signatursymbolen des Architekturteils ($\mathcal{S}_{\mathrm{A}}$), des Bausteinteils ($\mathcal{S}_{\mathrm{B}}$) und des Inte-

[6] Abkürzend wird statt von der Repräsentation eines Architekturmodells auch nur von einem Architekturmodell oder kürzer von einer Architektur gesprochen, wenn eine Unterscheidung zwischen Modell und Repräsentation aus dem Kontext hervorgeht.

grationsteils ($\mathcal{S}_\mathrm{I}$). Die Abbildung $\alpha_\mathfrak{I}$ über den Signatursymbolen $\mathcal{S}_\mathrm{I} = \mathcal{F}_\mathrm{I} \dot\cup \mathcal{R}_\mathrm{I}$ bildet

- jedes n-stellige Funktionssymbol $\mathrm{f} \in \mathcal{F}_\mathrm{I}$ auf eine n-stellige Funktion $\alpha(\mathrm{f}) : (\mathcal{A}_\mathfrak{I})^n \in \mathcal{A}_\mathfrak{I}$,
- jedes 1-stellige Relationssymbol $\mathrm{R} \in \mathcal{R}_\mathrm{I}$ auf eine 1-stellige Relation $\alpha(\mathrm{R}) \subseteq \mathcal{A}_\mathfrak{I}$ und
- jedes $n \geq 2$-stellige Relationssymbol $\mathrm{R} \in \mathcal{R}_\mathrm{I}$ auf eine $n \geq 2$-stellige Relation $\alpha(\mathrm{R}) \subseteq (\mathcal{A}_\mathfrak{M})^n$ ab.

Ist die Struktur $\mathfrak{M}$ darüber hinaus ein Modell des Axiomensystems Φ_M ($\mathfrak{M} \models \Phi_\mathrm{M}$), dann wird sie als *gültiges Gesamtmodell* bezeichnet. Die Menge aller gültigen Gesamtmodelle ist entsprechend die Modellklasse

$$\mathfrak{K}_\mathfrak{M} := \{\mathfrak{M} \,|\, \mathfrak{M} \models \Phi_\mathrm{M}\}$$

7.2.3. Gültigkeit der Eigenschaftsbedingungen

Eigenschaftsbedingungen beschreiben Voraussetzungen, die gelten müssen, damit bestimmte bausteinspezifische Eigenschaften gewahrt sind. Zu diesem Zweck kann jeder Architekturbaustein $\mathfrak{B}$ eine unbeschränkte Anzahl von Eigenschaftsbedingungen definieren. Wie in Abschnitt 6.5 gezeigt, werden Eigenschaftsbedingungen als prädikatenlogische Sätze formuliert. Formal handelt es sich dabei um Sätze über der Signatur $\mathcal{S}_\mathrm{BBAE}$, die Aussagen über das vollständige Gesamtmodell machen. Gekennzeichnet werden Eigenschaftsbedingungen mit Bezeichnern der Form $\mathrm{ABBC}_{\mathfrak{B},i}$. Diese Bezeichner verweisen zum einen auf den zugehörigen Architekturbaustein $\mathfrak{B}$ und sind zum anderen pro Baustein fortlaufend nummeriert.

Definition 13 (Eigenschaftsbedingungen eines Architekturbausteins)
Die Eigenschaftsbedingungen eines Architekturbausteins $\mathfrak{B}$ aus $\mathfrak{K}_\mathfrak{B}$ sind durch eine Menge von prädikatenlogischen $\mathcal{S}_\mathrm{BBAE}$-Sätzen $\Phi_{\mathrm{ABB}_\mathfrak{B}}$ definiert. Mit einer Indexmenge $\mathbb{I}_{\mathrm{ABBC}_\mathfrak{B}}$ gilt dann für Baustein $\mathfrak{B}$:

$$\Phi_{\mathrm{ABB}_\mathfrak{B}} := \{\mathrm{ABBC}_{\mathfrak{B},i} \mid i \in \mathbb{I}_{\mathrm{ABBC}_\mathfrak{B}}\}$$

Die Menge der Eigenschaftsbedingungen aller Bausteine $\mathfrak{B}$ aus $\mathfrak{K}_\mathfrak{B}$ ist dann

$$\Phi_\mathrm{ABB} := \bigcup_{\mathfrak{B} \in \mathfrak{K}_\mathfrak{B}} \Phi_{\mathrm{ABB}_\mathfrak{B}}$$

Ein Gesamtmodell, das gültig bezüglich des in dieser Arbeit definierten Metamodells ist, muss laut Definition 12 alle Bedingungen aus Φ_M erfüllen. Diese Bedingungen gelten unabhängig von der im Gesamtmodell enthaltenen Bausteininstanzen. Sobald das Gesamtmodell Bausteininstanzen enthält, müssen auch die Eigenschaftsbedingungen der zu den jeweiligen Instanzen gehörigen Bausteinen gelten. Ein solches *bausteinkonsistentes* Gesamtmodell sollte das Ergebnis des bausteinbasierten Architekturentwurfs sein.

[7] Abkürzend wird statt von der Repräsentation des Gesamtmodells auch nur von dem Gesamtmodell an sich gesprochen, wenn eine Unterscheidung zwischen Modell und Repräsentation aus dem Kontext hervorgeht.

Definition 14 (Bausteinkonsistentes Gesamtmodell)
Ein gültiges Gesamtmodell $\mathfrak{M}$ ($\in \mathfrak{K}_{\mathfrak{M}}$) ist *konsistent* zu den Bausteinen seines Bausteinkatalogs $\mathfrak{K}_{\mathfrak{B}}$, wenn

$$\mathfrak{M} \models \Phi_{\text{ABB}} \text{ für alle } \mathfrak{B} \in \mathfrak{K}_{\mathfrak{B}}$$

gilt.

7.3. Funktionen zur Änderung von Modellen

Im Basiskonzept des bausteinbasierten Architekturentwurfs wird die Architektur eines Systems durch die Instanziierung eines Bausteins erweitert. Allerdings können mit einer Bausteininstanziierung auch umfangreichere Änderungen an der Architektur einhergehen, die sich nicht nur auf Ergänzungen beschränken. Es können auch Aufteilungen und Umgestaltungen von Architekturelementen oder Teilen der Architektur erfolgen. Solche Änderungen führen dann ggf. auch zu Löschungen und anschließenden Neuerstellungen. Dabei wird nicht nur der Architekturteil verändert. Auch der Integrationsteil des Gesamtmodells muss entsprechend angepasst werden. Denn Löschungen im Architekturteil können bestehende Bindungen vorheriger Instanziierungen betreffen. Um ein konsistentes Ergebnismodell zu erhalten, müssen bestehende Bindungen berücksichtigt und ggf. geeignet angepasst werden.

Im Folgenden werden die verschiedenen Funktionen des bausteinbasierten Architekturentwurfs formal beschrieben. Hierbei werden durch verschiedene Parameter sowohl der zu instanziierende Baustein und dazugehörige Bindungspaare als auch im Modell durchzuführende Änderungen festgelegt. Wie und wo sich die Parameter auf das Ausgangsmodell auswirken, wird durch die Funktionen an sich definiert. In Abschnitt 7.3.1 wird zunächst eine Funktion für eine additive Bausteininstanziierung definiert und erläutert. Im Rahmen dieser Funktion kann ein Gesamtmodell nur erweitert werden; Löschungen sind nicht möglich. Aufbauend auf dieser zentralen Funktion ist eine weitere, ebenfalls additive Variante definiert. Diese Funktion bietet statt der Instanziierung eines Bausteins die Möglichkeit, eine bestehende Bausteininstanz um zusätzliche Bindungen zu erweitern. Um am Ausgangsmodell hingegen umfangreichere Änderungen mit Löschungen ausführen zu können, wird in Abschnitt 7.3.2 eine subtraktive Funktion angegeben. Mit Hilfe dieser Funktion können sowohl komplette Bausteininstanzen als auch einzelne Bindungen oder Architekturelement aus einem Gesamtmodell entfernt werden. Im bausteinbasierten Architekturentwurf wird ein Modell schrittweise entwickelt. Ein solcher Entwicklungsschritt kombiniert die Anwendung der zuvor genannten Änderungsfunktionen (Abschnitt 7.3.3).

7.3.1. Erstellen und Erweitern einer Bausteininstanz

Ausgangspunkt einer Bausteininstanziierung ω_{add} ist ein Gesamtmodell $\mathfrak{M}$. Zulässig ist auch ein Gesamtmodell, in dem die Trägermengen $\mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}}$ des Architekturteils und $\mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}}$ des Integrationsteils leer sind. Ein solches Gesamtmodell enthält ausschließlich einen Bausteinkatalog. Möglich ist auch ein leeres Gesamtmodell ohne Bausteinkatalog; dies widerspricht allerdings dem Konzept des bausteinbasierten Entwurfs, da dann die Instanziierung eines Bausteins nicht durchführbar ist.

Durch ω_{add} kann nur ein Baustein $\mathfrak{B}$ aus dem Bausteinkatalog von $\mathfrak{M}$ instanziiert werden. Im Zentrum der Instanziierung von $\mathfrak{B}$ steht die Bindung seiner bindbaren Bausteinelemente

an entsprechende Architekturelemente. Welche Bausteinelemente dabei an welche Architekturelemente gebunden werden sollen, wird durch Bindungspaare als Parameter der Funktion festgelegt. Die Architekturelemente dieser Bindungspaare $\mathfrak{P}$ existieren entweder bereits im Ausgangsmodell oder werden im Zuge der Bausteininstanziierung explizit als Architekturerweiterung ergänzt. Eine *Architekturerweiterung* enthält dabei nicht unbedingt nur die als Bindungspartner benötigten Architekturelemente. Sie kann darüber hinaus weitere Architekturelemente enthalten, die zur Sicherung der Konsistenz zum Metamodell notwendig sind. Definiert wird eine Architekturerweiterung $\Delta\mathfrak{A}$ als Tupel aus einer Trägermenge und einer Abbildung der Signatursymbole aus $\mathcal{S}_\mathrm{A}$. Angemerkt sei, dass dieses Tupel keine $\mathcal{S}_\mathrm{A}$-Struktur und damit entsprechend Definition 11 auch kein Architekturmodell ist.

Das Integrationsmetamodell definiert mit Verbindungspfaden und Nachrichtenspuren zwei Arten von bindbaren Architekturelementen (vgl. Abschnitt 6.4.1). Bei diesen Elementen handelt es sich um *kumulierenden Elemente*, die jeweils gleichartige Architekturelemente zusammenfassen. Durch sie kann eine Menge von Architekturelementen an ein entsprechendes Bausteinelement gebunden werden. Derartige Zuordnungen $\mathfrak{Z}_\mathrm{L}$ und $\mathfrak{Z}_\mathrm{M}$ können im Kontext einer Bausteininstanziierung ebenfalls neu hinzukommen.

Definition 15 (Bausteininstanziierung: Definitions- und Wertebereich)
Die Instanziierungsfunktion ω_add ist eine Funktion, die ein Tupel $(\mathfrak{M}, \mathfrak{B}, \Delta\mathfrak{A}, \mathfrak{Z}_\mathrm{L}, \mathfrak{Z}_\mathrm{M}, \mathfrak{P})$ auf ein Gesamtmodell $\mathfrak{M}'$ abbildet:

$$\omega_\mathrm{add} : (\mathfrak{M}, \mathfrak{B}, \Delta\mathfrak{A}, \mathfrak{Z}_\mathrm{L}, \mathfrak{Z}_\mathrm{M}, \mathfrak{P}) \mapsto \mathfrak{M}'$$

Für die Parameter der Instanziierungsfunktion gilt:

- $\mathfrak{M} := (\mathcal{A}_\mathfrak{M}, \alpha_\mathfrak{M})$ ist als $\mathcal{S}_\mathrm{BBAE}$-Struktur mit Trägermenge $\mathcal{A}_\mathfrak{M}$ und Abbildung $\alpha_\mathfrak{M}$ ein Gesamtmodell.
- $\mathfrak{B} := (\mathcal{A}_\mathfrak{B}, \alpha_\mathfrak{B})$ ist ein Baustein des Bausteinkatalogs $\mathfrak{K}_\mathfrak{B}$ (=Modellklasse der Bausteine) und damit eine $\mathcal{S}_\mathrm{B}$-Struktur mit Trägermenge $\mathcal{A}_\mathfrak{B}$ und Abbildung $\alpha_\mathfrak{B}$. Ebenso wie jedes andere Element des Bausteinkatalogs ist $\mathfrak{B}$ in $\mathfrak{M}$ enthalten.
- $\Delta\mathfrak{A} := (\mathcal{A}_{\Delta\mathfrak{A}}, \alpha_{\Delta\mathfrak{A}})$ sei als Architekturerweiterung bezeichnet. Die Abbildung $\alpha_{\Delta\mathfrak{A}}$ bildet die Signatur $\mathcal{S}_\mathrm{A}$ ab und ist über den Trägermengen $\mathcal{A}_{\Delta\mathfrak{A}}$ und $\mathcal{A}_{\mathfrak{M}_\mathfrak{A}}$ definiert (mit $\mathcal{A}_{\mathfrak{M}_\mathfrak{A}}$ als Trägermenge des Architekturteils aus $\mathfrak{M}$ und $\mathcal{A}_{\mathfrak{M}_\mathfrak{A}} \subseteq \mathcal{A}_\mathfrak{M}$). Dabei gilt für Funktionssymbole $\mathrm{f} \in \mathcal{F}_\mathrm{A} \subseteq \mathcal{S}_\mathrm{A}$: $\alpha_{\Delta\mathfrak{A}}(\mathrm{f}) \in \mathcal{A}_{\Delta\mathfrak{A}} \cup \mathcal{A}_{\mathfrak{M}_\mathfrak{A}}$ und n-stellige Relationssymbole $\mathrm{R} \in \mathcal{R}_\mathrm{A} \subseteq \mathcal{S}_\mathrm{A}$: $\alpha_{\Delta\mathfrak{A}}(\mathrm{R}) \in (\mathcal{A}_{\Delta\mathfrak{A}} \cup \mathcal{A}_{\mathfrak{M}_\mathfrak{A}})^n$.
- $\mathfrak{Z}_\mathrm{L}$ ist eine Menge von Paaren (u, V) mit einem $u \in \Delta\mathrm{I}$ aus einer Menge von Integrationselementen und einer Menge V. Elemente in V entstammen der Trägermenge $\mathcal{A}_{\mathfrak{M}_\mathfrak{A}}$ oder $\mathcal{A}_{\Delta\mathfrak{A}}$ und sind auch in den Relationen $\texttt{Link}^\mathfrak{M}$ bzw. $\texttt{Link}^{\Delta\mathfrak{A}}$ enthalten, auf die $\alpha_\mathfrak{M}$ bzw. $\alpha_{\Delta\mathfrak{A}}$ das Relationssymbol $\texttt{Link} \in \mathcal{S}_\mathrm{A}$ abbildet. Für V gilt zusammengefasst also $V \subseteq \texttt{Link}^\mathfrak{M} \cup \texttt{Link}^{\Delta\mathfrak{A}} \subseteq \mathcal{A}_{\mathfrak{M}_\mathfrak{A}} \cup \mathcal{A}_{\Delta\mathfrak{A}}$. Für die Tupelmenge $\mathfrak{Z}_\mathrm{L}$ gilt demnach:
$$\mathfrak{Z}_\mathrm{L} \subseteq \Delta\mathrm{I} \times \mathbb{P}(\texttt{Link}^\mathfrak{M} \cup \texttt{Link}^{\Delta\mathfrak{A}})$$
- $\mathfrak{Z}_\mathrm{M}$ ist analog zu $\mathfrak{Z}_\mathrm{L}$ eine Menge von Paaren (t, N) mit einem $t \in \Delta\mathrm{I}$ aus einer Menge von Integrationselementen und einer Menge N. Elemente in N entstammen der Trägermenge $\mathcal{A}_{\mathfrak{M}_\mathfrak{A}}$ oder $\mathcal{A}_{\Delta\mathfrak{A}}$ und sind auch in den Relationen $\texttt{MsgOccurrence}^\mathfrak{M}$ bzw. $\texttt{MsgOccurrence}^{\Delta\mathfrak{A}}$ enthalten, auf die $\alpha_\mathfrak{M}$ bzw. $\alpha_{\Delta\mathfrak{A}}$ das Relationssymbol $\texttt{MsgOccurrence} \in \mathcal{S}_\mathrm{A}$ abbildet. Für N gilt zusammengefasst also $N \subseteq \texttt{MsgOccurrence}^\mathfrak{M} \cup$

$\texttt{MsgOccurrence}^{\Delta\mathfrak{A}} \subseteq \mathcal{A}_{\mathfrak{M}_\mathfrak{A}} \cup \mathcal{A}_{\Delta\mathfrak{A}}$. Für die Tupelmenge $\mathfrak{Z}_\mathrm{M}$ gilt demnach:

$$\mathfrak{Z}_\mathrm{M} \subseteq \Delta\mathrm{I} \times \mathbb{P}(\texttt{MsgOccurrence}^{\mathfrak{M}} \cup \texttt{MsgOccurrence}^{\Delta\mathfrak{A}})$$

- $\mathfrak{P}$ ist eine Menge von Paaren (b, a) mit b aus der Trägermenge $\mathcal{A}_\mathfrak{B}$ des Bausteins $\mathfrak{B}$, wobei b ebenfalls Element der Relation $\texttt{BindableABBElement}^{\mathfrak{M}}$ ($\subseteq \mathcal{A}_\mathfrak{B}$) ist, auf die die Abbildung $\alpha_\mathfrak{M}$ des Gesamtmodells $\mathfrak{M}$ das Relationssymbol $\texttt{BindableABBElement} \in \mathcal{S}_\mathrm{B}$ abbildet. a ist entweder ein Element der Trägermenge $\mathcal{A}_\mathfrak{M}$ und zudem in der Relation $\texttt{BindableArchElement}^{\mathfrak{M}}$ enthalten oder entstammt der Trägermenge $\mathcal{A}_{\Delta\mathfrak{A}}$ oder ist ein Element der Menge $\Delta\mathrm{I}$ von Integrationselementen. Für die Menge $\mathfrak{P}$ gilt demzufolge:

$$\mathfrak{P} \subseteq \texttt{BindableABBElement}^{\mathfrak{M}} \times (\texttt{BindableArchElement}^{\mathfrak{M}} \cup \mathcal{A}_{\Delta\mathfrak{A}} \cup \Delta\mathrm{I})$$

- $\mathfrak{M}' := (\mathcal{A}_{\mathfrak{M}'}, \alpha_{\mathfrak{M}'})$ ist ebenso wie $\mathfrak{M}'$ eine $\mathcal{S}_\mathrm{BBAE}$-Struktur mit Trägermenge $\mathcal{A}_{\mathfrak{M}'}$ und Abbildung $\alpha_{\mathfrak{M}'}$. $\mathfrak{M}'$ ist zudem Element der Modellklasse $\mathfrak{K}_\mathfrak{M}$ der gültigen Gesamtmodelle. Somit gilt $\mathfrak{M}' \models \Phi_\mathrm{M}$.

In obigen Definitionen sei $\Delta\mathrm{I}$ eine Teilmenge des Universums $\mathcal{U}_\mathrm{I}$ über den Elementen des Integrationsteils mit $\Delta\mathrm{I} = \{x \,|\, (x, y) \in \mathfrak{Z}_\mathrm{L}\} \dot\cup \{x \,|\, (x, y) \in \mathfrak{Z}_\mathrm{M}\}$. Zudem gelte $\Delta\mathrm{I} \cap \mathcal{A}_\mathfrak{M} = \emptyset$.

Architekturelemente, die im Zuge einer Bausteininstanziierung neu erstellt werden, werden der Instanziierungsfunktion durch eine Trägermenge $\mathcal{A}_{\Delta\mathfrak{A}}$ übergeben. Durch eine Abbildung $\alpha_{\Delta\mathfrak{A}}$ wird ebenfalls festgelegt, wie die Symbole der Signatur $\mathcal{S}_\mathrm{A}$ auf Relationen über den neuen Architekturelementen aus $\mathcal{A}_{\Delta\mathfrak{A}}$ und den bestehenden aus der Trägermenge $\mathcal{A}_\mathfrak{M}$ des Gesamtmodells abgebildet werden. Zusätzliche Beachtung finden die Elemente in den Relationen $\texttt{Part}^{\Delta\mathfrak{A}}$, $\texttt{PortVariable}^{\Delta\mathfrak{A}}$ und $\texttt{Scenario}^{\Delta\mathfrak{A}}$. Diese Relationen repräsentieren den Teil der Elemente aus $\mathcal{A}_{\Delta\mathfrak{A}}$, an die Bausteinelemente gebunden werden können.

Definition 16 (Bausteininstanziierung: Einfügen der Architekturerweiterung $\Delta\mathfrak{A}$)
Sei $\mathfrak{M} := (\mathcal{A}_\mathfrak{M}, \alpha_\mathfrak{M})$ das Gesamtmodell vor der Ausführung der Instanziierungsfunktion ω_add und $\Delta\mathfrak{A} := (\mathcal{A}_{\Delta\mathfrak{A}}, \alpha_{\Delta\mathfrak{A}})$ die Architekturerweiterung im Rahmen der Instanziierung. Für den Architekturteil des neuen Gesamtmodells $\mathfrak{M}' := (\mathcal{A}_{\mathfrak{M}'}, \alpha_{\mathfrak{M}'})$ gilt:

- $\mathcal{A}_{\mathfrak{M}_\mathfrak{A}} \subseteq \mathcal{A}_\mathfrak{M}$ sei die Trägermenge des Architekturteils von $\mathfrak{M}$. Dann ist der Architekturteil der Trägermenge $\mathcal{A}_{\mathfrak{M}'}$ von $\mathfrak{M}'$ definiert durch:

$$\mathcal{A}_{\mathfrak{M}'_\mathfrak{A}} := \mathcal{A}_{\mathfrak{M}_\mathfrak{A}} \cup \mathcal{A}_{\Delta\mathfrak{A}}$$

- $\mathcal{S}_\mathrm{A}$ sei der Architekturteil der Signatur von $\mathfrak{M}$ und $\mathfrak{M}'$. Die Abbildung $\alpha_{\mathfrak{M}'}$ von $\mathfrak{M}'$ ist für $\mathrm{R} \in \mathcal{S}_\mathrm{A}$ wie folgt definiert:

$$\mathrm{R}^{\mathfrak{M}'} := \mathrm{R}^{\mathfrak{M}} \cup \mathrm{R}^{\Delta\mathfrak{A}}$$

Des Weiteren sei die Menge $\Delta\texttt{BindableArchElement}_{\Delta\mathfrak{A}} \subseteq \mathcal{A}_{\Delta\mathfrak{A}}$ definiert als

$$\Delta\texttt{BindableArchElement}_{\Delta\mathfrak{A}} := \texttt{Part}^{\Delta\mathfrak{A}} \cup \texttt{PortVariable}^{\Delta\mathfrak{A}} \cup \texttt{Scenario}^{\Delta\mathfrak{A}}$$

Kumulierende Elemente der Integrationsbeschreibung bieten die Möglichkeit Bausteinelemente indirekt an eine Menge von Architekturelemente zu binden. Durch den Parameter $\mathfrak{Z}_\mathrm{L}$ werden der Instanziierungsfunktion Paare übergeben, deren erste Komponente für einen neuen Verbindungspfad steht und deren zweite Komponente eine Menge von zu kumulierenden Verbindungselementen enthält. Analog ist der Parameter $\mathfrak{Z}_\mathrm{M}$ für Nachrichtenspur und Nachrichtenvorkommen aufgebaut. Der Teil des Integrationsmetamodells, der die Typen der kumu-

lierenden Elemente und ihre Beziehung definiert, wird durch die Relationssymbole `LinkPath`, `pathElement`, `MsgTrace` und `traceElement` der Signatur $\mathcal{S}_\mathrm{I}$ repräsentiert. Die Abbildung $\alpha_{\mathfrak{M}'}$ dieser Relationssymbole wird entsprechend ergänzt, so dass die Zuordnungen aus $\mathfrak{Z}_\mathrm{L}$ und $\mathfrak{Z}_\mathrm{M}$ auch in die resultierenden Relationen einfließen.

Definition 17 (Bausteininstanziierung: Einfügen der kumulierenden Elemente der Integrationsbeschreibung $\mathfrak{Z}_\mathrm{L}$ und $\mathfrak{Z}_\mathrm{M}$)
Sei $\mathfrak{M} := (\mathcal{A}_\mathfrak{M}, \alpha_\mathfrak{M})$ das Gesamtmodell vor der Ausführung der Instanziierungsfunktion ω_add. Für den Ausschnitt der kumulierenden Elemente des Integrationsteil des neuen Gesamtmodells $\mathfrak{M}' := (\mathcal{A}_{\mathfrak{M}'}, \alpha_{\mathfrak{M}'})$ gilt:

- $\mathcal{A}_{\mathfrak{M}'_\mathfrak{I}}$ sei die Trägermenge des Integrationsteils von $\mathfrak{M}'$. Dann ist die Menge der kumulierenden Elemente $\Delta\mathrm{I} = \{x \mid (x, y) \in \mathfrak{Z}_\mathrm{L}\} \dot\cup \{x \mid (x, y) \in \mathfrak{Z}_\mathrm{M}\}$ in der Differenz der Trägermengen $\mathfrak{M}'$ und $\mathfrak{M}$ enthalten:

$$\Delta\mathrm{I} \subseteq \mathcal{A}_{\mathfrak{M}'_\mathfrak{I}} \setminus \mathcal{A}_{\mathfrak{M}_\mathfrak{I}}$$

- Die Abbildung $\alpha_{\mathfrak{M}'}$ ist für Relationssymbole $\mathrm{R} \in \{\texttt{LinkPath}, \texttt{MsgTrace}, \texttt{pathElement}, \texttt{traceElement}\} \subseteq \mathcal{S}_\mathrm{I}$ definiert als

$$\mathrm{R}^{\mathfrak{M}'} := \mathrm{R}^{\mathfrak{M}} \dot\cup \Delta\mathrm{R}$$

  mit

  $\Delta\texttt{LinkPath} := \{u \mid (u, V) \in \mathfrak{Z}_\mathrm{L}\}$
  $\Delta\texttt{MsgTrace} := \{t \mid (t, N) \in \mathfrak{Z}_\mathrm{M}\}$
  $\Delta\texttt{pathElement} := \{(u, e) \mid (u, V) \in \mathfrak{Z}_\mathrm{L} \text{ und } e \in V\}$
  $\Delta\texttt{traceElement} := \{(t, e) \mid (t, N) \in \mathfrak{Z}_\mathrm{M} \text{ und } e \in N\}$

Der letzte Parameter der Instanziierungsfunktion enthält die Bindungspaare aus Bausteinelementen des zu instanziierenden Bausteins $\mathfrak{B}$ und bindbaren Architekturelementen. Mögliche bindbare Architekturelemente können verschiedenen Mengen entstammen. Sie können zum einen bereits in der Trägermenge $\mathcal{A}_\mathfrak{M}$ des alten Gesamtmodells $\mathfrak{M}$ enthalten sein. Zum anderen können sie auch aus Mengen stammen, die erst während des Instanziierungsvorgangs ergänzt werden. Bei der letzteren Art von Mengen handelt es sich um die Trägermenge $\mathcal{A}_{\Delta\mathfrak{A}}$ der Architekturerweiterung und der Menge der neuen kumulierenden Integrationselemente $\Delta\mathrm{I}$. Pro Bindungspaar wird der Trägermenge $\mathcal{A}_{\mathfrak{M}'}$ ein neues Element hinzugefügt, das diese Bindung repräsentiert. Die Abbildung $\alpha_{\mathfrak{M}'}$ wird entsprechend angepasst, so dass die neuen Bindungselemente u.a. auch Teil der Relation $\texttt{Binding}^{\mathfrak{M}'}$ sind. Die Trägermenge $\mathcal{A}_{\mathfrak{M}'}$ wird außerdem um ein Element erweitert, das die Bausteininstanz an sich repräsentiert und in der Relation $\texttt{ABBOccurrence}^{\mathfrak{M}'}$ enthalten ist. Daneben wird $\alpha_{\mathfrak{M}'}$ für alle weiteren Relationssymbole $\in \mathcal{S}_\mathrm{I}$ definiert, die für Instanziierung und Bindung relevant sind.

Definition 18 (Bausteininstanziierung: Erzeugen und Einfügen der Elemente der Bausteininstanz)
Sei $\Delta\texttt{Binding} \subseteq \mathcal{U}_\mathrm{I}$ eine Teilmenge des Universums $\mathcal{U}_\mathrm{I}$ über Elementen des Integrationsteils mit $|\Delta\texttt{Binding}| = |\mathfrak{P}|$. Zudem gelte: $\Delta\texttt{Binding} \cap \mathcal{A}_\mathfrak{M} = \emptyset$ und $\Delta\texttt{Binding} \cap \Delta\mathrm{I} = \emptyset$.
Ferner existiere eine bijektive Abbildung $f : \mathfrak{P} \to \Delta\texttt{Binding}$, die jedem $p \in \mathfrak{P}$ ein $c \in \Delta\texttt{Binding}$ zuordnet.

Sei $\mathfrak{M} := (\mathcal{A}_{\mathfrak{M}}, \alpha_{\mathfrak{M}})$ das Gesamtmodell vor der Ausführung der Instanziierungsfunktion ω_{add}. Für den Integrationsteil (mit Ausnahme der kumulierenden Elemente) des neuen Gesamtmodells $\mathfrak{M}' := (\mathcal{A}_{\mathfrak{M}'}, \alpha_{\mathfrak{M}'})$ gilt dann:

- Die Abbildung $\alpha_{\mathfrak{M}'}$ des Gesamtmodells $\mathfrak{M}'$ ist für jedes Relationssymbol $\mathrm{R} \in \mathcal{S}_{\mathrm{I}} \setminus \{\texttt{LinkPath}, \texttt{MsgTrace}, \texttt{pathElement}, \texttt{traceElement}\}$ definiert als $\mathrm{R}^{\mathfrak{M}'} := \mathrm{R}^{\mathfrak{M}} \dot{\cup} \Delta\mathrm{R}$ mit

  $\Delta\texttt{Binding}$ wie oben definiert

  $\Delta\texttt{EntityBinding} := \{c \mid (b,a) = p \in \mathfrak{P},\ f(p) = c \text{ und } b \in \texttt{ABBEntity}^{\mathfrak{B}}\}$

  $\Delta\texttt{ConnectorBinding} := \{c \mid (b,a) = p \in \mathfrak{P},\ f(p) = c \text{ und } b \in \texttt{ABBConnector}^{\mathfrak{B}}\}$

  $\Delta\texttt{ConnectorEndBinding} := \{c \mid (b,a) = p \in \mathfrak{P},\ f(p) = c \text{ und } b \in \texttt{ABBConnectorEnd}^{\mathfrak{B}}\}$

  $\Delta\texttt{MessageBinding} := \{c \mid (b,a) = p \in \mathfrak{P},\ f(p) = c \text{ und } b \in \texttt{ABBMessage}^{\mathfrak{B}}\}$

  $\Delta\texttt{ScenarioBinding} := \{c \mid (b,a) = p \in \mathfrak{P},\ f(p) = c \text{ und } b \in \texttt{ABBScenario}^{\mathfrak{B}}\}$

  $\Delta\texttt{containsMsgBinding} := \{(u,v) \mid (u,s) \in \Delta\texttt{bindsABBElement}, (s,m) \in \texttt{includesABBMsg}^{\mathfrak{B}} \text{ und } (v,m) \in \Delta\texttt{bindsABBElement}\}$

  $\Delta\texttt{bindsABBElement} := \{(c,b) \mid (b,a) = p \in \mathfrak{P} \text{ und } f(p) = c\}$

  $\Delta\texttt{bindsArchElement} := \{(c,a) \mid (b,a) = p \in \mathfrak{P} \text{ und } f(p) = c\}$

  $\Delta\texttt{ABBOccurrence} := \{o\} \subseteq \mathcal{U}_{\mathrm{I}} \setminus (\Delta\texttt{Binding} \cup \Delta\mathrm{I}) \text{ und } |\Delta\texttt{ABBOccurrence}| = 1$

  $\Delta\texttt{containsBinding} := \{(o,c) \mid o \in \Delta\texttt{ABBOccurrence} \text{ und } c \in \Delta\texttt{Binding}\}$

  $\Delta\texttt{instantiatedABB} := \{(o,b) \mid o \in \Delta\texttt{ABBOccurrence} \text{ und } b \in \texttt{ABB}^{\mathfrak{B}}\}$

  $\Delta\texttt{containsABBOccurrence} := \{(a,o) \mid o \in \Delta\texttt{ABBOccurrence} \text{ und } a \in \texttt{Architecture}^{\mathfrak{M}_{\mathfrak{A}}}\}$

- Die Trägermenge $\mathcal{A}_{\mathfrak{M}'_{\mathfrak{I}}}$ des Integrationsteils von $\mathfrak{M}'$ ist definiert durch

$$\mathcal{A}_{\mathfrak{M}'_{\mathfrak{I}}} := \mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}} \dot{\cup} \Delta\mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}}$$

  mit

$$\Delta\mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}} := \Delta\texttt{BindableArchElement} \dot{\cup} \Delta\texttt{Binding} \dot{\cup} \Delta\texttt{ABBOccurrence}$$

  wobei $\Delta\texttt{BindableArchElement}$ definiert ist als

$$\Delta\texttt{BindableArchElement} := \Delta\texttt{BindableArchElement}_{\Delta\mathfrak{A}} \cup \Delta\mathrm{I}$$

Auf Basis der formalen Beschreibung kann auch der in Abschnitt 5.1.2 informell eingeführte Begriff der *Instanzkomposition* formal gefasst werden. Die Instanzkomposition bezeichnet die Kombination von Bausteininstanzen, wobei Bausteinelemente unterschiedlicher Instanzen an dieselben Architekturelemente gebunden werden. Dies erfolgt durch eine Bausteininstanziierung ω_{add}, wenn in den durch $\mathfrak{P}$ festgelegten Bindungspaaren Architekturelemente vorkommen, die bereits im Ausgangsmodell $\mathfrak{M}$ an Bindungen beteiligt sind.

Definition 19 (Instanzkomposition)
Eine durch eine Instanziierungsfunktion ω_{add} mit dem Parametertupel $(\mathfrak{M}, \mathfrak{B}, \Delta\mathfrak{A}, \mathfrak{Z}_{\mathrm{L}}, \mathfrak{Z}_{\mathrm{M}}, \mathfrak{P})$

ausgeführte Bausteininstanziierung ist eine *Instanzkomposition*, wenn

$$\{a \mid (b, a) \in \mathfrak{P}\} \subseteq \{a \mid (c, a) \in \texttt{bindsArchElement}^{\mathfrak{M}}\}$$

gilt.

Neben der Instanziierung eines Architekturbausteins ist auch die Erweiterung einer bestehenden Bausteininstanz möglich. Die Erweiterung hat viele Gemeinsamkeiten mit der Instanziierung und unterscheidet sich nur in zwei Details von dieser. Entsprechend ähnlich wird auch die Funktion ω_{ext} definiert. Statt eines Parameters für den zu instanziierenden Baustein besitzt ω_{ext} einen Parameter für die zu erweiternde Bausteininstanz. Der zweite Unterschied betrifft die Relationen $\texttt{ABBOccurrence}^{\mathfrak{M}}$ und $\texttt{containsABBOccurrence}^{\mathfrak{M}}$, die im Fall von ω_{ext} nicht um ein neues Element ergänzt werden.

Definition 20 (Erweiterung einer Bausteininstanz)
Die Erweiterung einer Bausteininstanz ω_{ext} bildet ein Tupel $(\mathfrak{M}, o, \Delta\mathfrak{A}, \mathfrak{Z}_{\text{L}}, \mathfrak{Z}_{\text{M}}, \mathfrak{P})$ auf ein Gesamtmodell $\mathfrak{M}'$ ab:

$$\omega_{\text{ext}} : (\mathfrak{M}, o, \Delta\mathfrak{A}, \mathfrak{Z}_{\text{L}}, \mathfrak{Z}_{\text{M}}, \mathfrak{P}) \mapsto \mathfrak{M}'$$

Für die Parameter von ω_{ext} gilt:

- o ist ein Element aus $\texttt{ABBOccurrence}^{\mathfrak{M}} \subseteq \mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}}$.
- Die Parameter $\mathfrak{M}$, $\Delta\mathfrak{A}$, $\mathfrak{Z}_{\text{L}}$, $\mathfrak{Z}_{\text{M}}$, $\mathfrak{P}$ und $\mathfrak{M}'$ sind identisch zu den Parameter von ω_{add} in Definition 15 definiert.

Definition 16 (Einfügen von neuen Architekturelementen aus $\Delta\mathfrak{A}$) und Definition 17 (Einfügen von kumulierenden Elementen aus $\mathfrak{Z}_{\text{L}}$ und $\mathfrak{Z}_{\text{M}}$) für ω_{add} werden auf ω_{ext} übertragen.
Definition 18 (Erzeugen und Einfügen der Elemente der Bausteininstanz) für ω_{add} wird wie folgt auf ω_{ext} angepasst:

- Für Relationssymbole $\text{R} \in \{\texttt{ABBOccurrence}, \texttt{containsABBOccurrence}\} \subseteq \mathcal{S}_{\text{I}}$ ist die Abbildung $\alpha_{\mathfrak{M}'}$ des Gesamtmodells $\mathfrak{M}'$ als $\text{R}^{\mathfrak{M}'} := \text{R}^{\mathfrak{M}} \dot{\cup} \Delta\text{R}$ definiert mit

  $\Delta_{\texttt{ABBOccurrence}} := \emptyset$

  $\Delta_{\texttt{containsABBOccurrence}} := \emptyset$

- Die Relationen $\text{R}^{\mathfrak{B}}$ mit den Relationssymbolen $\text{R} \in \{\texttt{ABB}, \texttt{ABBEntity}, \texttt{ABBConnector}, \texttt{ABBConnectorEnd}, \texttt{ABBMessage}, \texttt{includesABBMsg}, \texttt{ABBScenario}\} \subseteq \mathcal{S}_{\text{B}}$ sind für ω_{ext} durch entsprechende Relationen $\text{R}^{\mathfrak{M}}$ zu ersetzen.

7.3.2. Entfernen von Modellteilen

Das Entfernen von Teilen eines Gesamtmodells $\mathfrak{M}$ wird durch die subtraktive Funktion ω_{sub} beschrieben. Diese Funktion ermöglicht es, sowohl komplette Bausteininstanzen $\delta\mathfrak{O}$ als auch einzelne Bindungen $\delta\mathfrak{P}$ zu entfernen. In beiden Fällen sind nicht nur die angegebenen Elemente an sich, sondern ggf. weitere Elemente des Integrationsteils zu löschen. Auch Architekturelemente können durch Anwendung dieser Funktion aus dem Gesamtmodell entfernt werden. Eine solche *Architekturreduktion* $\delta\mathfrak{A}$ wird analog zu einer Architekturerweiterung ebenfalls durch ein Tupel aus Trägermenge und Abbildung angegeben.

Definition 21 (Entfernen von Modellteilen: Definitions- und Wertebereich)
Die Funktion ω_{sub} ist eine Funktion, die ein Tupel $(\mathfrak{M}, \delta\mathfrak{A}, \delta\mathfrak{P}, \delta\mathfrak{O})$ auf eine Struktur $\mathfrak{M}'$ abbildet:

$$\omega_{\text{sub}} : (\mathfrak{M}, \delta\mathfrak{A}, \delta\mathfrak{P}, \delta\mathfrak{O}) \mapsto \mathfrak{M}'$$

Für die Parameter der Funktion ω_{sub} gilt:

- $\mathfrak{M} := (\mathcal{A}_{\mathfrak{M}}, \alpha_{\mathfrak{M}})$ und $\mathfrak{M}' := (\mathcal{A}_{\mathfrak{M}'}, \alpha_{\mathfrak{M}'})$ sind durch $\mathcal{S}_{\text{BBAE}}$-Strukturen mit Trägermenge $\mathcal{A}_{\mathfrak{M}}$ bzw. $\mathcal{A}_{\mathfrak{M}'}$ und Abbildung $\alpha_{\mathfrak{M}}$ bzw. $\alpha_{\mathfrak{M}'}$ repräsentierte Gesamtmodelle.
- $\delta\mathfrak{A} := (\mathcal{A}_{\delta\mathfrak{A}}, \alpha_{\delta\mathfrak{A}})$ sei als Architekturreduktion bezeichnet. Die Trägermenge $\mathcal{A}_{\delta\mathfrak{A}}$ ist Teilmenge der Architekturelemente des Gesamtmodells $\mathfrak{M}$ ($\mathcal{A}_{\delta\mathfrak{A}} \subseteq \mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}}$). Die Abbildung $\alpha_{\delta\mathfrak{A}}$ bildet die Signatur $\mathcal{S}_{\text{A}}$ ab und ist über der Trägermenge $\mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}}$ definiert (mit $\mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}}$ als Trägermenge des Architekturteils aus $\mathfrak{M}$ und $\mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}} \subseteq \mathcal{A}_{\mathfrak{M}}$). Dabei gilt für Funktionssymbole $\mathrm{f} \in \mathcal{F}_{\text{A}} \subseteq \mathcal{S}_{\text{A}}$: $\alpha_{\delta\mathfrak{A}}(\mathrm{f}) \in \mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}}$ und n-stellige Relationssymbole $\mathrm{R} \in \mathcal{R}_{\text{A}} \subseteq \mathcal{S}_{\text{A}}$: $\alpha_{\delta\mathfrak{A}}(\mathrm{R}) \in (\mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}})^n$.
- $\delta\mathfrak{P}$ ist eine Menge von Elementen p mit $p \in \texttt{Binding}^{\mathfrak{M}} \subseteq \mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}}$.
- $\delta\mathfrak{O}$ ist eine Menge von Elementen o mit $o \in \texttt{ABBOccurrence}^{\mathfrak{M}} \subseteq \mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}}$.

Durch den Parameter $\delta\mathfrak{A}$ werden Architekturelemente der Trägermenge $\mathcal{A}_{\mathfrak{M}}$ bestimmt, die durch die Funktion ω_{sub} gelöscht werden. Die Reduktion der Trägermenge wirkt sich zusätzlich auf die Relationen aus, auf die $\alpha_{\mathfrak{M}}$ die Relationssymbole der Signatur $\mathcal{S}_{\text{A}}$ abbildet. Wird ein Architekturelement der Trägermenge entfernt, so darf dieses in keiner der Relationen nach Anwendung von ω_{sub} noch existieren. Die Abbildung $\alpha_{\delta\mathfrak{A}}$ legt daher die zu löschenden Elemente der Relationen fest. Entsprechend bildet $\alpha_{\delta\mathfrak{A}}$ die Relationssymbole aus $\mathcal{S}_{\text{A}}$ auf Relationen über $\mathcal{A}_{\mathfrak{M}}$ ab.

Definition 22 (Entfernen von Modellteilen: Durchführen der Architekturreduktion $\delta\mathfrak{A}$)
Sei $\mathfrak{M} := (\mathcal{A}_{\mathfrak{M}}, \alpha_{\mathfrak{M}})$ eine $\mathcal{S}_{\text{BBAE}}$-Struktur vor und $\mathfrak{M}' := (\mathcal{A}_{\mathfrak{M}'}, \alpha_{\mathfrak{M}'})$ eine $\mathcal{S}_{\text{BBAE}}$-Struktur nach der Ausführung der Funktion ω_{sub}.
Mit der Architekturreduktion $\delta\mathfrak{A} := (\mathcal{A}_{\delta\mathfrak{A}}, \alpha_{\delta\mathfrak{A}})$ gilt für den Architekturteil des neuen Gesamtmodells:

- $\mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}} \subseteq \mathcal{A}_{\mathfrak{M}}$ sei die Trägermenge des Architekturteils von $\mathfrak{M}$. Dann ist der Architekturteil der Trägermenge $\mathcal{A}_{\mathfrak{M}'}$ von $\mathfrak{M}'$ definiert durch:

$$\mathcal{A}_{\mathfrak{M}'_{\mathfrak{A}}} := \mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}} \setminus \mathcal{A}_{\delta\mathfrak{A}}$$

- $\mathcal{S}_{\text{A}}$ sei der Architekturteil der Signatur von $\mathfrak{M}$ und $\mathfrak{M}'$. Die Abbildung $\alpha_{\mathfrak{M}'}$ von $\mathfrak{M}'$ ist für $\mathrm{R} \in \mathcal{S}_{\text{A}}$ wie folgt definiert:

$$\mathrm{R}^{\mathfrak{M}'} := \mathrm{R}^{\mathfrak{M}} \setminus \mathrm{R}^{\delta\mathfrak{A}}$$

Die Funktion ω_{sub} bietet die Möglichkeit, sowohl einzelne Bindungen als auch komplette Bausteininstanzen aus dem Gesamtmodell zu entfernen. Die zu löschenden Bindungen und Bausteininstanzen werden der Funktion durch die beiden Parameter $\delta\mathfrak{P}$ bzw. $\delta\mathfrak{O}$ übergeben. Basierend auf diesen Parametern wird sowohl die neue Trägermenge $\mathcal{A}_{\mathfrak{m}'_{\mathfrak{I}}}$ aus $\mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}}$ abgeleitet als auch die Abbildung $\alpha_{\mathfrak{M}}$ für Relationssymbole des Integrationsteils angepasst. So werden aus der Trägermenge $\mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}}$ u.a. die Elemente aus den Mengen $\delta\mathfrak{P}$ und $\delta\mathfrak{O}$ entfernt. Die Elemente aus $\delta\mathfrak{P}$ und $\delta\mathfrak{O}$ werden ebenfalls aus den Relationen $\texttt{Binding}^{\mathfrak{M}}$ bzw. $\texttt{ABBOccurrence}^{\mathfrak{M}}$ entfernt. Darüber hinaus haben die entfernten Bindungen und Bausteininstanzen Einfluss auf

andere Relationen, auf die die Relationssymbole der Signatur $\mathcal{S}_{\mathrm{I}}$ abgebildet werden. So führt z.B. das Entfernen einer Bausteininstanz zu Veränderungen in den Relationen $\texttt{Binding}^{\mathfrak{M}}$ und $\texttt{containsBinding}^{\mathfrak{M}}$, da mit dem Wegfall der Bausteininstanz auch die zugehörigen Bindungen zu entfernen sind. Aber auch kumulierende Elemente, die nach der Löschung von Bindungen an keiner weiteren Bindung mehr beteiligt sind, werden gelöscht.

Definition 23 (Entfernen von Modellteilen: Entfernen von Bindungspaaren $\delta\mathfrak{P}$ und Bausteininstanzen $\delta\mathfrak{O}$)
Sei $\mathfrak{M} := (\mathcal{A}_{\mathfrak{M}}, \alpha_{\mathfrak{M}})$ eine $\mathcal{S}_{\mathrm{BBAE}}$-Struktur vor und $\mathfrak{M}' := (\mathcal{A}_{\mathfrak{M}'}, \alpha_{\mathfrak{M}'})$ eine $\mathcal{S}_{\mathrm{BBAE}}$-Struktur nach der Ausführung der Funktion ω_{sub}.
Für den Integrationsteils des neuen Gesamtmodells $\mathfrak{M}' := (\mathcal{A}_{\mathfrak{M}'}, \alpha_{\mathfrak{M}'})$ gilt dann:

- Die Abbildung $\alpha_{\mathfrak{M}'}$ des Gesamtmodells $\mathfrak{M}'$ ist für jedes Relationssymbol $\mathrm{R} \in \mathcal{S}_{\mathrm{I}}$ definiert als $\mathrm{R}^{\mathfrak{M}'} := \mathrm{R}^{\mathfrak{M}} \setminus \delta\mathrm{R}$ mit

  $\delta\texttt{ABBOccurrence} := \delta\mathfrak{O}$

  $\delta\texttt{Binding} := \delta\mathfrak{P} \cup \{c \mid (o,c) \in \texttt{containsBinding}^{\mathfrak{M}} \text{ und } o \in \delta\texttt{ABBOccurrence}\}$

  $\delta\texttt{EntityBinding} := \texttt{EntityBinding}^{\mathfrak{M}} \cap \delta\texttt{Binding}$

  Analog zu $\delta\,\texttt{EntityBinding}$ sind $\delta\,\texttt{ConnectorBinding}$, $\delta\,\texttt{ConnectorEndBinding}$, $\delta\texttt{MessageBinding}$ und $\delta\texttt{ScenarioBinding}$ definiert.

  $\delta\texttt{containsMsgBinding} := \{(u,v) \mid (u,v) \in \texttt{containsMsgBinding}^{\mathfrak{M}} \text{ und } v \in \delta\texttt{MessageBinding}\}$

  $\delta\texttt{LinkPath} := \{u \mid \forall (c,u) \in \texttt{bindsArchElement}^{\mathfrak{M}} : c \in \delta\texttt{ConnectorBinding}\}$

  $\delta\texttt{MsgTrace} := \{t \mid \forall (c,t) \in \texttt{bindsArchElement}^{\mathfrak{M}} : c \in \delta\texttt{MessageBinding}\}$

  $\delta\texttt{pathElement} := \{(u,e) \mid (u,e) \in \texttt{pathElement}^{\mathfrak{M}} \text{ und } u \in \delta\texttt{LinkPath}\}$

  $\delta\texttt{traceElement} := \{(t,e) \mid (t,e) \in \texttt{traceElement}^{\mathfrak{M}} \text{ und } t \in \delta\texttt{MsgTrace}\}$

  $\delta\texttt{BindableArchElement} := \delta\texttt{LinkPath} \cup \delta\texttt{MsgTrace}$

  $\delta\texttt{bindsABBElement} := \{(c,b) \mid (c,b) \in \texttt{bindsABBElement}^{\mathfrak{M}} \text{ und } c \in \delta\texttt{Binding}\}$

  $\delta\texttt{bindsArchElement} := \{(c,a) \mid (c,a) \in \texttt{bindsArchElement}^{\mathfrak{M}} \text{ und } c \in \delta\texttt{Binding}\}$

  $\delta\texttt{containsBinding} := \{(o,c) \mid (o,c) \in \texttt{containsBinding}^{\mathfrak{M}} \text{ und } c \in \delta\texttt{Binding}\}$

  $\delta\texttt{instantiatedABB} := \{(o,b) \mid (o,b) \in \texttt{instantiatedABB}^{\mathfrak{M}} \text{ und } o \in \delta\texttt{ABBOccurrence}\}$

  $\delta\texttt{containsABBOccurrence} := \{(a,o) \mid (a,o) \in \texttt{containsABBOccurrence}^{\mathfrak{M}} \text{ und } o \in \delta\texttt{ABBOccurrence}\}$

- Die Trägermenge $\mathcal{A}_{\mathfrak{M}'_{\mathfrak{I}}}$ des Integrationsteils von $\mathfrak{M}'$ ist definiert durch

  $$\mathcal{A}_{\mathfrak{M}'_{\mathfrak{I}}} := \mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}} \dot{\cup}\, \delta\mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}}$$

  mit

  $$\delta\mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}} := \delta\texttt{ABBOccurrence} \cup \delta\texttt{Binding} \cup \delta\texttt{BindableArchElement}$$

7.3.3. Kombination von Änderungsfunktionen

In den Abschnitten 7.3.1 und 7.3.2 wurden Funktionen zur Änderung eines Gesamtmodells definiert. Diese Änderungsfunktionen können miteinander kombiniert und zu Entwicklungs-

schritten des bausteinbasierten Entwurfs zusammengefasst werden. Zur Kombination von Änderungsfunktionen wird das Gesamtmodell, das das Ergebnis der ersten Funktion ist, der zweiten Funktion als Parameter übergeben.

Definition 24 (Kombination von Änderungsfunktionen)
Seien ω' und ω'' zwei Änderungsfunktionen mit $\omega' : (\mathfrak{M}, t') \mapsto \mathfrak{M}'$ und $\omega'' : (\mathfrak{M}', t'') \mapsto \mathfrak{M}''$, wobei t' und t'' weitere Parameter von ω' bzw. ω'' und $\mathfrak{M}$, $\mathfrak{M}'$ sowie $\mathfrak{M}''$ Gesamtmodelle sind. Dann ist eine Funktion ω mit $(\mathfrak{M}, t', t'') \mapsto \mathfrak{M}''$ wie folgt als Kombination von ω' und ω'' definiert:

$$\omega(\mathfrak{M}, t', t'') = \omega''(\mathfrak{M}', t'') \circ \omega'(\mathfrak{M}, t') = \omega''(\omega'(\mathfrak{M}, t'), t'')$$

Abkürzend wird dies ohne Parameter durch $\omega = \omega'' \circ \omega'$ notiert.

Die Kombination von Änderungsfunktionen wird bzgl. der Häufigkeit einzelner Funktionen und deren Reihenfolge eingeschränkt. So kann maximal ein Architekturbaustein pro Entwicklungsschritt instanziiert werden. Entsprechend darf ω_{add} maximal einmal angewendet werden. Destruktive Änderungen durch ω_{sub} können nur zu Beginn vor einer Bausteininstanziierung durchgeführt werden. Abschließend können dann beliebig viele Ergänzungen durch ω_{ext} erfolgen.

Definition 25 ((Konsistenter) Entwicklungsschritt)
Ein Entwicklungsschritt ω_{BBAE} ist eine Kombination der Änderungsfunktionen ω_{add}, ω_{ext} und ω_{sub} und mit $m, n \in \mathbb{N}_0$ sowie $o \in \{0, 1\}$ mit $n + o \geq 1$ definiert als

$$\omega_{\text{BBAE}} := (\omega_{\text{ext}})^n \circ (\omega_{\text{add}})^o \circ (\omega_{\text{sub}})^m$$

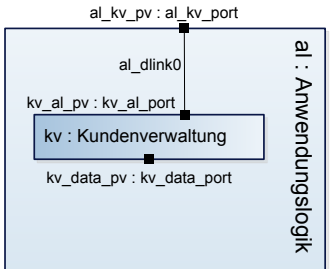
und bildet auf ein Gesamtmodell $\mathfrak{M}$ ab, das gültig ($\mathfrak{M} \models \Phi_{\text{BBAE}}$) sein muss.
Ist $\mathfrak{M}$ zudem bausteinkonsistent ($\mathfrak{M} \models \Phi_{\text{ABB}}$), wird der Entwicklungsschritt als *bausteinkonsistent* oder kurz als *konsistent* bezeichnet.

Als letzte Funktion in einem Entwicklungsschritt ω_{BBAE} wird die Erweiterung ω_{ext} oder eine Bausteininstanziierung ω_{add} angewendet. Beide Funktionen bilden gemäß Definition 15 auf ein Gesamtmodell $\mathfrak{M}$ ab, das gültig ($\mathfrak{M} \models \Phi_{\text{BBAE}}$) ist. Demzufolge ist das Ergebnis eines Entwicklungsschritts ebenfalls mindestens gültig.

7.4. Beispiel für eine formale Bausteininstanziierung

In den vorherigen Abschnitten dieses Kapitels wurden verschiedene Aspekte des bausteinbasierten Architekturentwurfs formal definiert. Zum einen wurde festgelegt, wie ein Modell und das Metamodell des bausteinbasierten Architekturentwurfs formal repräsentiert werden. Zum anderen wurden Funktionen definiert, mit deren Hilfe eine Veränderung eines solchen formalen Modells möglich ist. Die Verwendung dieser Formalismen wird im Folgenden an einem Entwicklungsschritt des einführenden Beispiels aus Abschnitt 4.3 skizziert. Betrachtet wird dabei der dritte Entwicklungsschritt, in dem der Fassadebaustein instanziiert wird. Es werden sowohl die zu kombinierenden Änderungsfunktionen und ihre Parameter als auch das formale Ausgangsmodell des Entwicklungsschritts angegeben. Dabei wird allerdings nicht das vollständige Beispielsystem wiedergegeben, sondern lediglich Bereiche, die für das Beispiel relevant sind.

Abbildung 7.2a zeigt einen Ausschnitt aus der Struktur des Beispielsystems vor dem Entwicklungsschritt 3. Es besteht aus einem Part `al`, dessen Komponententyp einen Part `kv` enthält.



(a)

Binding	ABBElement	ArchElement
hi2_eb1	HCompositum	al
hi2_eb2	HComponens	kv
hi2_cb1	hcomp_con	{al_dlink0}
hi2_ceb1	hcomp_end0	al_kv_pv
hi2_ceb2	hcomp_end1	al_kv_pv
hi2_ceb3	hcomp_end2	kv_al_pv

(b)

Abbildung 7.2: Ausschnitt aus Struktur (a) und Bindungen (b) des Beispielsystems vor Entwicklungsschritt 3.

Verbunden sind diese Parts durch einen Delegationskonnektor `al_dlink0`. Des Weiteren listet Abbildung 7.2b die Bindungen auf, die zu einer vorher erfolgten Instanziierung des Bausteins Hierarchische Komposition gehören und an denen die gezeigten Elemente beteiligt sind.

Das Ausgangsmodell ist ein Gesamtmodell $\mathfrak{M}$ aus $\mathfrak{K}_{\mathfrak{M}}$ mit einer Trägermenge $\mathcal{A}_{\mathfrak{M}}$ und einer Abbildung $\alpha_{\mathfrak{M}}$. Die Trägermenge ist in $\mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}}$ für die Architekturteile, $\mathcal{A}_{\mathfrak{M}_{\mathfrak{B}}}$ für die Bausteinteile (im Folgenden nicht dargestellt) und $\mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}}$ für die Integrationsteile geteilt. $\mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}}$ und $\mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}}$ entsprechen für den Ausschnitt aus Abbildung 7.2a folgenden Mengen:

$$\mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}} = \{\texttt{al}, \texttt{kv}, \texttt{Anwendungslogik}, \texttt{Kundenverwaltung}, \texttt{al\_dlink0}, \texttt{al\_kv\_pv}, \texttt{kv\_al\_pv}, \texttt{kv\_data\_pv}, ...\}$$

$$\mathcal{A}_{\mathfrak{M}_{\mathfrak{I}}} = \{\texttt{hi2}, \texttt{hi2\_eb1}, \texttt{hi2\_eb2}, \texttt{hi2\_cb1}, \texttt{hi2\_ceb1}, \texttt{hi2\_ceb2}, \texttt{hi2\_ceb3}, ...\}$$

Die Abbildung $\alpha_{\mathfrak{M}}$ bildet die Signaturen $\mathcal{S}_{\mathrm{A}}$ und $\mathcal{S}_{\mathrm{I}}$ in die Trägermenge $\mathcal{A}_{\mathfrak{M}}$ ab. Für einen Teil der Signaturen ist die Abbildung nachfolgend in Auszügen dargestellt:

$$\texttt{Part}^{\mathfrak{M}} = \{\texttt{al}, \texttt{kv}, ...\}$$

$$\texttt{Component}^{\mathfrak{M}} = \{\texttt{Anwendungslogik}, \texttt{Kundenverwaltung}, ...\}$$

$$\texttt{hasType}^{\mathfrak{M}} = \{(\texttt{al}, \texttt{Anwendungslogik}), (\texttt{kv}, \texttt{Kundenverwaltung}), ...\}$$

$$\texttt{PortVariable}^{\mathfrak{M}} = \{\texttt{al\_kv\_pv}, \texttt{kv\_al\_pv}, \texttt{kv\_data\_pv}, ...\}$$

$$\texttt{hasPortVariable}^{\mathfrak{M}} = \{(\texttt{al}, \texttt{al\_kv\_pv}), (\texttt{kv}, \texttt{kv\_al\_pv}), (\texttt{kv}, \texttt{kv\_data\_pv}), ...\}$$

$$\texttt{DelegatorLink}^{\mathfrak{M}} = \{\texttt{al\_dlink0}, ...\}$$

$$\texttt{linkSource}^{\mathfrak{M}} = \{(\texttt{al\_dlink0}, \texttt{al\_kv\_pv}), ...\}$$

$$\texttt{linkTarget}^{\mathfrak{M}} = \{(\texttt{al\_dlink0}, \texttt{kv\_al\_pv}), ...\}$$

$$\texttt{LinkPath}^{\mathfrak{M}} = \{\texttt{lp1}, ...\}$$

$$\texttt{pathElement}^{\mathfrak{M}} = \{(\texttt{lp1}, \{\texttt{al\_dlink0}\})\}$$

$$\texttt{ABBOccurrence}^{\mathfrak{M}} = \{\texttt{hi2}, ...\}$$

$$\texttt{Binding}^{\mathfrak{M}} = \{\texttt{hi2\_eb1}, \texttt{hi2\_eb2}, \texttt{hi2\_cb1}, \texttt{hi2\_ceb1}, \texttt{hi2\_ceb2}, ...\}$$

$$\texttt{bindsArchElement}^{\mathfrak{M}} = \{(\texttt{hi2\_eb1}, \texttt{al}), (\texttt{hi2\_eb2}, \texttt{kv}), (\texttt{hi2\_cb1}, \texttt{lp1}), (\texttt{hi2\_ceb1}, \texttt{al\_kv\_pv}), (\texttt{hi2\_ceb3}, \texttt{kv\_al\_pv}), ...\}$$

Im Beispiel wird durch die Einführung der Fassade im Entwicklungsschritt 3 der Zugriffspunkt `kv_al_pv` des Parts `kv` gekapselt. Dadurch ersetzt letztendlich der Part `fas` als Fassade den Delegationskonnektor `al_dlink0`. Infolgedessen wird das andere Ende von `al_dlink0` an

al (al_kv_pv) durch einen neuen Zugriffspunkt ersetzt, der nun mit fas über einen Delegationskonnektor verbunden ist. Der Entwicklungsschritt 3 setzt sich dementsprechend aus der Anwendung der drei verschiedenen Änderungsfunktionen ω_{ext}, ω_{add} und ω_{sub} zu der Funktion ω_{E3} zusammen:

$$\omega_{\text{E3}} = \omega_{\text{ext}} \circ \omega_{\text{add}} \circ \omega_{\text{sub}}$$

Dabei werden zuerst Modellelemente durch die subtraktive Änderungsfunktion ω_{sub} entfernt. Im Anschluss wird durch ω_{add} die Bausteininstanziierung des Entwicklungsschritt durchgeführt und schließlich werden durch ω_{ext} bestehende Bausteininstanzen ergänzt.

Im Folgenden werden die Parameter der einzelnen Änderungsfunktionen des Entwicklungsschritts 3 in Auszügen dargestellt. Nicht berücksichtigt wird dabei der Parameter für das Gesamtmodell, der oben initial beschrieben wurde und jeweils als Ergebnis der inneren Funktion an die äußere weitergereicht wird.

- Die subtraktive Änderungsfunktion ω_{sub} besitzt neben dem Parameter $\mathfrak{M}$ für das zu verändernde Gesamtmodell die Parameter $\delta\mathfrak{A}$ (Architekturreduktion), $\delta\mathfrak{P}$ (zu entfernende Bindungspaare) und $\delta\mathfrak{O}$ (zu entfernende Bausteininstanzen). Im Beispiel werden diese wie folgt belegt:

  $\delta\mathfrak{A}$ ist als Tupel mit $\delta\mathcal{A} = \{\texttt{al\_dlink0}, \texttt{al\_kv\_pv}, ...\}$ und einer Abbildung $\delta\alpha$ u.a. durch $\delta\,\texttt{PortVariable}^{\mathfrak{M}} = \{\texttt{al\_kv\_pv}, ...\}$ und $\delta\,\texttt{hasPortVariable}^{\mathfrak{M}} = \{(\texttt{al}, \texttt{al\_kv\_pv}), ...\}$ definiert

  $\delta\mathfrak{P} = \{(\texttt{hcomp\_end0}, \texttt{al\_kv\_pv}), (\texttt{hcomp\_end1}, \texttt{al\_kv\_pv}), (\texttt{hcomp\_end2}, \texttt{kv\_al\_pv}), (\texttt{hcomp\_con}, \texttt{al\_dlink0}), ...\}$

  $\delta\mathfrak{O} = \emptyset$

- Die Bausteininstanziierung ω_{add} besitzt neben $\mathfrak{M}$ die Parameter $\mathfrak{B}$ (zu instanziierender Baustein), $\Delta\mathfrak{A}$ (zu ergänzende Architekturelemente), $\mathfrak{Z}_{\text{L}}$ sowie $\mathfrak{Z}_{\text{M}}$ (kumulierende Integrationselemente) und $\mathfrak{P}$ (Bindungspaare). Belegt werden diese Parameter im Beispiel folgendermaßen:

  $\mathfrak{B} = \texttt{Fassadebaustein}$

  $\Delta\mathfrak{A}$ mit $\Delta\mathcal{A} = \{\texttt{fas}, \texttt{Anwendungsfassade}, \texttt{fas\_kv\_pv}, \texttt{fas\_kv\_port}, \texttt{al\_faskv\_con2}, ...\}$

  $\mathfrak{Z}_{\text{L}} = \{(\texttt{lp2}, \{\texttt{al\_faskv\_con1}\}), ...\}$

  $\mathfrak{Z}_{\text{M}} = \{...\}$

  $\mathfrak{P} = \{(\texttt{Fassade}, \texttt{fas}), (\texttt{Subsystemelement}, \texttt{kv}), (\texttt{fassade\_con}, \texttt{lp2}), ...\}$

- Die Parameter der Erweiterungsfunktions ω_{ext} sind mit denen der Bausteininstanziierung bis auf den zweiten Parameter identisch. Statt eines zu instanziierenden Bausteins erwartet die Erweiterungsfunktion eine Bausteininstanz o. Die Parameter werden im Beispiel folgendermaßen belegt:

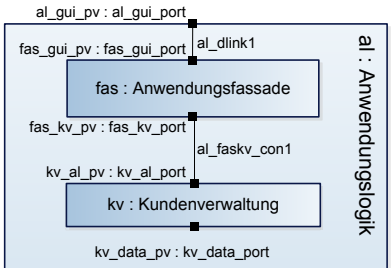
  $o = \texttt{hi2}$

  $\Delta\mathfrak{A}$ mit $\Delta\mathcal{A} = \{\texttt{al\_gui\_pv}, \texttt{al\_gui\_port}, \texttt{fas\_gui\_pv}, \texttt{fas\_gui\_port}, \texttt{al\_dlink1}, ...\}$

  $\mathfrak{Z}_{\text{L}} = \{(\texttt{lp3}, \{\texttt{al\_dlink2}\}), ...\}$

  $\mathfrak{Z}_{\text{M}} = \{...\}$

  $\mathfrak{P} = \{(\texttt{hcomp\_con}, \texttt{lp3}), (\texttt{hcomp\_end0}, \texttt{al\_gui\_pv}), (\texttt{HComponens}, \texttt{fas}), (\texttt{hcomp\_end1}, \texttt{al\_gui\_pv}), (\texttt{hcomp\_end2}, \texttt{fas\_gui\_pv}), ...\}$



(a)

Binding	ABBElement	ArchElement
hi2_eb1	HCompositum	al
hi2_eb2	HComponens	kv
hi2_eb4	HComponens	fas
hi2_cb1	hcomp_con	{al_dlink1}
hi2_ceb1	hcomp_end0	al_gui_pv
hi2_ceb2	hcomp_end1	al_gui_pv
hi2_ceb3	hcomp_end2	fas_gui_pv
fi1_eb1	Fassade	fas
fi1_eb2	Subsystemelement	kv
fi1_cb1	fassade_con	{al_faskv_con1}
fi1_ceb1	fc_end1	fas_kv_pv
fi1_ceb2	fc_end2	kv_al_pv

(b)

Abbildung 7.3: Ausschnitt aus Struktur (a) und Bindungen (b) des Beispielsystems nach Entwicklungsschritt 3.

Ergebnis des Entwicklungsschritt 3 ist ein neues Gesamtmodell $\mathfrak{M}'$, in das eine Instanz des Fassadebausteins integriert wurde. Abbildung 7.3a stellt einen Ausschnitt dieses Modells dar. Zusätzlich werden die Bindungen, an denen die im Diagramm dargestellten Elemente beteiligt sind, in Abbildung 7.3b aufgelistet. Gemäß der Werte, die den Parameter der Änderungsfunktionen im Erweiterungsschritts ω_{E3} zugeordnet wurden, setzt sich die Trägermenge $\mathcal{A}_{\mathfrak{M}'}$ folgendermaßen zusammen:

$$\mathcal{A}_{\mathfrak{M}'_{\mathfrak{A}}} = \mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}} \setminus \{\texttt{al\_dlink0}, \texttt{al\_kv\_pv}, ...\} \cup \{\texttt{fas}, \texttt{Anwendungsfassade}, \texttt{al\_dlink1}, \texttt{al\_faskv\_con1}, \texttt{al\_gui\_pv}, \texttt{fas\_gui\_pv}, \texttt{fas\_kv\_pv}, ...\}$$

$$\mathcal{A}_{\mathfrak{M}'_{\mathfrak{I}}} = \mathcal{A}_{\mathfrak{M}_{\mathfrak{A}}} \setminus \{\texttt{hi2\_cb1}, \texttt{hi2\_ceb1}, \texttt{hi2\_ceb2}, ...\} \cup \{\texttt{hi2\_eb4}, \texttt{hi2\_cb1}, \texttt{hi2\_ceb1}, \texttt{hi2\_ceb2}, \texttt{fi1}, \texttt{fi1\_eb1}, \texttt{fi1\_eb2}, \texttt{fi1\_cb1}, \texttt{fi1\_ceb1}, \texttt{fi1\_ceb2}, ...\}$$

Entsprechend wurde auch die Abbildung $\alpha_{\mathfrak{M}'}$ geändert, die die Signaturen $\mathcal{S}_{\mathrm{A}}$ und $\mathcal{S}_{\mathrm{I}}$ in die Trägermenge $\mathcal{A}'_{\mathfrak{M}}$ abbildet. Der zuvor bereits betrachtete Teil der Signaturen stellt sich nach ω_{E3} wie folgt dar:

$$\texttt{Part}^{\mathfrak{M}'} = \texttt{Part}^{\mathfrak{M}} \cup \{\texttt{fas}, ...\}$$

$$\texttt{Component}^{\mathfrak{M}'} = \texttt{Component}^{\mathfrak{M}} \cup \{\texttt{Anwendungsfassade}, ...\}$$

$$\texttt{hasType}^{\mathfrak{M}'} = \texttt{hasType}^{\mathfrak{M}} \cup \{(\texttt{fas}, \texttt{Anwendungsfassade}), ...\}$$

$$\texttt{PortVariable}^{\mathfrak{M}'} = \texttt{PortVariable}^{\mathfrak{M}} \setminus \{\texttt{al\_kv\_pv}, ...\} \cup \{\texttt{al\_gui\_pv}, \texttt{fas\_gui\_pv}, \texttt{fas\_kv\_pv}, ...\}$$

$$\texttt{hasPortVariable}^{\mathfrak{M}'} = \texttt{hasPortVariable}^{\mathfrak{M}} \setminus \{(\texttt{al}, \texttt{al\_kv\_pv}), ...\} \cup \{(\texttt{al}, \texttt{al\_gui\_pv}), (\texttt{fas}, \texttt{fas\_gui\_pv}), (\texttt{fas}, \texttt{fas\_kv\_pv}), ...\}$$

$$\texttt{DelegatorLink}^{\mathfrak{M}'} = \texttt{DelegatorLink}^{\mathfrak{M}} \setminus \{\texttt{al\_dlink1}, ...\} \cup \{\texttt{al\_dlink1} ...\}$$

$$\texttt{Connector}^{\mathfrak{M}'} = \texttt{Connector}^{\mathfrak{M}} \setminus \{...\} \cup \{\texttt{al\_faskv\_con1} ...\}$$

$$\texttt{linkSource}^{\mathfrak{M}'} = \texttt{linkSource}^{\mathfrak{M}} \setminus \{(\texttt{al\_dlink0}, \texttt{al\_kv\_pv}), ...\} \cup \{(\texttt{al\_dlink1}, \texttt{al\_gui\_pv}), (\texttt{al\_faskv\_con1}, \texttt{fas\_kv\_pv}), ...\}$$

$$\texttt{linkTarget}^{\mathfrak{M}'} = \texttt{linkTarget}^{\mathfrak{M}} \setminus \{(\texttt{al\_dlink0}, \texttt{kv\_al\_pv}), ...\} \cup \{(\texttt{al\_dlink1}, \texttt{fas\_gui\_pv}), (\texttt{al\_faskv\_con1}, \texttt{kv\_al\_pv}), ...\}$$

$$\texttt{LinkPath}^{\mathfrak{M}'} = \texttt{LinkPath}^{\mathfrak{M}} \setminus \{\texttt{lp1}, ...\} \cup \{\texttt{lp2}, \texttt{lp3}, ...\}$$

$$\texttt{pathElement}^{\mathfrak{M}'} = \texttt{pathElement}^{\mathfrak{M}} \setminus \{(\texttt{lp1}, \{\texttt{al\_dlink0}\}), ...\} \cup \{(\texttt{lp2}, \{\texttt{al\_faskv\_con1}\}), (\texttt{lp3}, \{\texttt{al\_dlink1}\})\}$$

$$\texttt{ABBOccurrence}^{\mathfrak{M}'} = \texttt{ABBOccurrence}^{\mathfrak{M}} \cup \{\texttt{fi1}, ...\}$$

$$\texttt{Binding}^{\mathfrak{M}'} = \texttt{Binding}^{\mathfrak{M}} \setminus \{\texttt{hi2\_cb1}, \texttt{hi2\_ceb1}, \texttt{hi2\_ceb2}, \texttt{hi2\_ceb3}, ...\} \cup \{\texttt{hi2\_eb4}, \texttt{hi2\_cb1}, \texttt{hi2\_ceb1}, \texttt{hi2\_ceb2}, \texttt{hi2\_ceb3}, \texttt{fi1\_eb1}, \texttt{fi1\_eb2}, \texttt{fi1\_ceb1}, \texttt{fi1\_cb1}, \texttt{fi1\_ceb2}, ...\}$$

$$\texttt{bindsArchElement}^{\mathfrak{M}'} = \texttt{bindsArchElement}^{\mathfrak{M}} \setminus \{(\texttt{hi2\_cb1}, \texttt{lp1}), (\texttt{hi2\_ceb1}, \texttt{al\_kv\_pv}), (\texttt{hi2\_ceb3}, \texttt{kv\_al\_pv}), ...\} \cup \{(\texttt{hi2\_eb4}, \texttt{fas}), (\texttt{hi2\_cb1}, \texttt{lp3}), (\texttt{hi2\_ceb1}, \texttt{al\_gui\_pv}), (\texttt{hi2\_ceb3}, \texttt{fas\_gui\_pv}), (\texttt{fi1\_eb1}, \texttt{fas}), (\texttt{fi1\_eb2}, \texttt{al}), (\texttt{fi1\_cb1}, \texttt{lp2}), (\texttt{fi1\_ceb1}, \texttt{fas\_kv\_pv}), (\texttt{fi1\_ceb2}, \texttt{kv\_al\_pv}), ...\}$$

8. Fallstudie zur Anwendung des bausteinbasierten Architekturentwurfs

Inhalt

In den vorherigen Kapiteln wurden verschiedene Aspekte des bausteinbasierten Architekturentwurfs ausgearbeitet. Es wurde ein Metamodell definiert, mit dem neben Architekturen auch Architekturbausteine beschrieben werden können (siehe Kapitel 6). Zusätzlich führt dieses Metamodell durch entsprechende Elemente die Metamodellteile für Architekturen und Bausteine zusammen. Dieser Integrationsteil des Metamodells ermöglicht es, die in einer Architektur integrierten Bausteininstanzen zu identifizieren. Für Modelle auf Basis dieses Metamodells wurde des Weiteren eine Abbildung auf mathematische Strukturen eingeführt (siehe Abschnitt 7.2). Diese Strukturen bilden die Grundlage, um die als prädikatenlogische Sätze formulierten Eigenschaftsbedingungen (siehe Abschnitt 6.5) auszuwerten. Wesentlich hierfür sind die Integrationsaspekte des Metamodells, die eine Identifikation der Bausteininstanzen in der Architektur erst ermöglichen. Schließlich wurde mit der Operationalisierung (siehe Abschnitt 7.3) die Voraussetzung für eine strukturierte und automatisierbare Erstellung einer bausteinbasierten Architektur geschaffen.

Dieses Kapitel zeigt die Anwendung des bausteinbasierten Entwurfs am Beispielsystem[8] aus Abschnitt 4.3. Dabei dient die Entwicklung des Beispielsystems gleichzeitig als Fallstudie, deren Ziele in Abschnitt 8.1 erläutert werden. Im Rahmen der Fallstudie werden zunächst die im Beispielsystem verwendeten Architekturbausteine inklusive ihrer Eigenschaftsbedingungen beschrieben (Abschnitt 8.2). Anschließend werden verschiedene Entwicklungsschritte des Beispielsystems durchgeführt und die resultierenden Modelle hinsichtlich ihrer Konsistenz zu den Eigenschaftsbedingungen untersucht (Abschnitt 8.3). Die Fallstudie wird abschließend noch einmal hinsichtlich der zuvor definierten Ziele betrachtet (Abschnitt 8.4).

[8] Das Beispielsystem besteht am Ende seines Entwurfs aus einem dreischichtigen Informationssystem. Zwischen seinen Schichten, die teilweise hierarchisch komponiert sind, wird der Observer realisiert, während die Anwendungsschicht eine Fassade enthält.

8.1. Ziele der Fallstudie

Diese Fallstudie illustriert die Anwendung des bausteinbasierten Architekturentwurfs. Sie untersucht die praktische Einsatzfähigkeit des Ansatzes und dient der Evaluation des Ansatzes hinsichtlich der gestellten Forschungsfragen. In Anlehnung an die in Abschnitt 4.4 formulierten Forschungsfragen stehen bei der Untersuchung besonders folgende Punkte im Fokus:

- Beschreibung der Architekturbausteine
- Identifizierung von angewendeten Architekturbausteinen und ihrer Bestandteile
- Formulierung von Eigenschaftsbedingungen der Architekturbausteine
- Überprüfung der Eigenschaftsbedingungen von angewendeten Architekturbausteinen

Ziel der Fallstudie ist der Nachweis, dass diese Punkte durch den in dieser Arbeit vorgestellten Ansatz umfassend adressiert werden und der Ansatz praktisch einsetzbar ist. Eine abschließende Betrachtung erfolgt in Abschnitt 8.4.

8.2. Beschreibung der Architekturbausteine des Beispielsystems

Das Beispielsystem aus Kapitel 4 besteht maßgeblich aus der Anwendung von vier Architekturbausteinen. Im Folgenden werden Struktur, Verhalten und die Eigenschaftsbedingungen dieser Bausteine modelliert. Ausschnitte einzelner Architekturbausteine wurden bereits in Abschnitt 6.3 und in Abschnitt 6.5 zur Erläuterung des Metamodells bzw. der Formulierung von Eigenschaftsbedingungen vorgestellt. Diese Ausschnitte werden an dieser Stelle noch einmal aufgegriffen und somit alle Bausteine vollständig beschrieben. Struktur und Verhalten der Bausteine werden mit der in dieser Arbeit eingeführten grafischen Notation dargestellt. Ergänzend dazu ist die formale Repräsentation jedes Architekturbausteins in Anhang C notiert. Die Eigenschaftsbedingungen der Bausteine werden im Folgenden in der abkürzenden Schreibweise dargestellt. Die ausführliche Definition findet sich jeweils in Anhang B. Bei den im Beispielsystem verwendeten Bausteinen handelt es sich um den Schichtenbaustein (Abschnitt 8.2.1), den Pull-Observerbaustein (Abschnitt 8.2.2), den Fassadebaustein (Abschnitt 8.2.3) und die Hierarchische Komposition (Abschnitt 8.2.4).

8.2.1. Schichtenbaustein

Der Schichtenbaustein ist ein rein strukturell orientierter Baustein und definiert kein spezifisches Verhalten. Daher beschränkt sich die folgende Beschreibung auch nur auf seine Struktur. Abbildung 8.1 zeigt den Schichtenbaustein mit seinen beiden Entitätenrollen obereSchicht und untereSchicht. Beide Entitätenrollen können, wie durch ihre Multiplizitäten angezeigt, durch eine Bausteininstanziierung nur jeweils ein einziges Mal instanziiert werden. Ihre Konnektorrollenenden sc_end1 bzw. sc_end2 können hingegen jeweils beliebig oft pro Bindung der Entitätenrolle gebunden werden. Verbunden sind die beiden Rollen über ihre Konnektorrollenenden durch die Konnektorrolle schichten_con. Die Konnektorrolle darf in beide Richtungen zu beliebig vielen Endpunkten verlaufen und ist multiTyped.

Zwischen den beiden Entitätenrollen des Schichtenbausteins besteht eine hierarchische Nutzungs- und Abhängigkeitsbeziehung. Diese erlaubt Abhängigkeiten lediglich ausgehend von der oberen Schicht zur unteren Schicht, nicht aber umgekehrt. Wird der Baustein instanziiert und

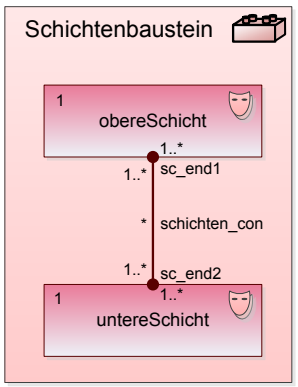


Abbildung 8.1: Bausteine des Beispielsystems: Struktur des Schichtenbausteins.

die Entitätenrollen gebunden, müssen u.a. die Interaktionen zwischen den jeweils gebundenen Architekturelementen dieser Einschränkung gehorchen. Es dürfen demzufolge weder direkte noch transitive Interaktionen zwischen von einem an untereSchicht gebundenem Architekturelement zu einem an obereSchicht gebundenem Architekturelement erfolgen, die zu einer Nutzungs- oder Abhängigkeitsbeziehung führen. Wann eine Interaktion eine nicht erlaubte Nutzung oder Abhängigkeit darstellt, wird im Rahmen der Eigenschaftsbedingung festgelegt. Insgesamt ist diese Eigenschaftsbedingung des Schichtenmusters abkürzend wie folgt definiert:

ABBC$_{Schichten1}$

`"Schichtenbaustein"`
`"obereSchicht", "untereSchicht"`
`¬dependingLayerUsage("untereSchicht","obereSchicht")`

Das Adapterprädikat `dependingLayerUsage(b, a)` legt fest, welche Arten der Nutzung zu einer Abhängigkeit führen. An dieser Stelle wird lediglich eine synchrone Interaktion als eine solche Nutzung definiert. Weitere Beziehungen, die nicht zugelassen werden sollen, können durch Disjunktion in der folgenden Definition des Adapterprädikats ergänzt werden:

$$\texttt{dependingLayerUsage}(b,a) := \texttt{containsSyncInteraction}(b,a)^{1} \qquad \text{(AP 2)}$$

[1] (AP 3)

Bei dem Prädikat `containsSyncInteraction(b,a)` handelt es sich ebenfalls um ein Adapterprädikat. Es überprüft, ob zwei Architekturelemente `b` und `a` durch eine synchrone Interaktion voneinander abhängen. Dabei müssen die beiden Architekturelemente nicht direkt miteinander interagieren. Es werden auch Folgen von Interaktionen betrachtet, die ihrer Ursprung in `a` haben und irgendwann `b` erreichen. In einer Folge reicht die Existenz einer einzelnen synchronen

Interaktion aus, um das Kriterium für Abhängigkeit zu erfüllen:

[1](AP 4), [2](AP 6)

$$\begin{aligned}&\texttt{containsSyncInteraction}(\texttt{b},\texttt{a}) := \texttt{syncInteraction}(\texttt{b},\texttt{a})^1 \vee \\ &\quad(\exists \texttt{c} : \texttt{Part}(\texttt{c}) \wedge \texttt{syncInteraction}(\texttt{b},\texttt{c}) \wedge \texttt{interaction}^*(\texttt{c},\texttt{a})^2) \vee \\ &\quad(\exists \texttt{c} : \texttt{Part}(\texttt{c}) \wedge \texttt{interaction}^*(\texttt{b},\texttt{c}) \wedge \texttt{syncInteraction}(\texttt{c},\texttt{a})) \vee \\ &\quad(\exists \texttt{c} \exists \texttt{d} : \texttt{Part}(\texttt{c}) \wedge \texttt{Part}(\texttt{d}) \wedge \\ &\quad\ \texttt{interaction}^*(\texttt{b},\texttt{c}) \wedge \texttt{syncInteraction}(\texttt{c},\texttt{d}) \wedge \texttt{interaction}^*(\texttt{d},\texttt{a}))\end{aligned} \tag{AP 3}$$

Eine synchrone Interaktion liegt vor, wenn zwischen zwei Portvariablen der beiden Parts `b` und `a` ein synchrones Nachrichtenvorkommen modelliert ist. Zusammengefasst wird dies durch das Adapterprädikat $\texttt{syncInteraction}(\texttt{b},\texttt{a})$:

$$\begin{aligned}&\texttt{syncInteraction}(\texttt{b},\texttt{a}) := \texttt{Part}(\texttt{a}) \wedge \texttt{Part}(\texttt{b}) \wedge \\ &\quad\exists \texttt{p} \exists \texttt{q} : \texttt{PortVariable}(\texttt{p}) \wedge \texttt{Portvariable}(\texttt{q}) \wedge \\ &\quad\texttt{hasPortVariable}(\texttt{a},\texttt{p}) \wedge \texttt{hasPortVariable}(\texttt{b},\texttt{q}) \wedge \\ &\quad\exists \texttt{m} \exists \texttt{n} \exists \texttt{v} : \texttt{MessageOccurrence}(\texttt{m}) \wedge \texttt{Message}(\texttt{n}) \wedge \texttt{occurredMsg}(\texttt{m},\texttt{n}) \wedge \\ &\quad\texttt{msgSender}(\texttt{n},\texttt{q}) \wedge \texttt{msgReceiver}(\texttt{n},\texttt{p}) \wedge \texttt{msgType}(\texttt{m},\texttt{"synchronous"})\end{aligned} \tag{AP 4}$$

Fast identisch zu AP 4 ist $\texttt{interaction}(\texttt{b},\texttt{a})$ formuliert, das die Existenz einer beliebigen Interaktion zwischen zwei Parts `b` und `a` überprüft:

$$\begin{aligned}&\texttt{interaction}(\texttt{b},\texttt{a}) := \texttt{Part}(\texttt{a}) \wedge \texttt{Part}(\texttt{b}) \wedge \\ &\quad\exists \texttt{p} \exists \texttt{q} : \texttt{PortVariable}(\texttt{p}) \wedge \texttt{Portvariable}(\texttt{q}) \wedge \\ &\quad\texttt{hasPortVariable}(\texttt{a},\texttt{p}) \wedge \texttt{hasPortVariable}(\texttt{b},\texttt{q}) \wedge \\ &\quad\exists \texttt{m} \exists \texttt{n} \exists \texttt{v} : \texttt{MessageOccurrence}(\texttt{m}) \wedge \texttt{Message}(\texttt{n}) \wedge \texttt{occurredMsg}(\texttt{m},\texttt{n}) \wedge \\ &\quad\texttt{msgSender}(\texttt{n},\texttt{q}) \wedge \texttt{msgReceiver}(\texttt{n},\texttt{p})\end{aligned} \tag{AP 5}$$

Durch AP 4 und AP 5 kann nur auf direkte Interaktion zwischen `b` und `a` geprüft werden. Ob eine beliebig lange Folge von Interaktionen existiert, kann das Adapterprädikat $\texttt{interaction}^*(\texttt{b},\texttt{a})$ als transitive Variante von $\texttt{interaction}(\texttt{b},\texttt{a})$ feststellen:

[1](AP 5)

$$\begin{aligned}&\texttt{interaction}^*(\texttt{b},\texttt{a}) := \texttt{Part}(\texttt{a}) \wedge \texttt{Part}(\texttt{b}) \wedge (\texttt{interaction}(\texttt{b},\texttt{a})^1 \vee \\ &\quad(\exists \texttt{c} : \texttt{Part}(\texttt{c}) \wedge \texttt{interaction}(\texttt{b},\texttt{c}) \wedge \texttt{interaction}^*(\texttt{c},\texttt{a})))\end{aligned} \tag{AP 6}$$

In der strikten Variante sind Interaktionen nur zwischen benachbarten Schichten erlaubt. Das Überspringen einzelner oder mehrerer Schichten der Hierarchie ist untersagt. Um eine entsprechende Eigenschaftsbedingung aufstellen zu können, muss zunächst geklärt werden, wie im bausteinbasierten Entwurf ein mehrschichtiges System gebildet wird. Denn die im Schichtenbaustein vorliegende Interpretation des Schichtenmusters betrachtet lediglich zwei Schichten. Eine mehrschichtige Hierarchie ist erst durch eine mehrfache Instanziierung des Bausteins möglich. Hierbei werden die Rollen obereSchicht und untereSchicht in unterschiedlichen Instanziierungen an denselben Part gebunden. Um sicherzustellen, dass keine Schichten der Hierarchie übersprungen werden, wird jeder Part, der an die Entitätenrolle obereSchicht gebunden ist, und alle mit ihm verbundenen Parts untersucht. Ein verbundener Part darf keine untereSchicht in derselben Schichtenhierarchie wie der Ausgangspart sein oder muss sich in derselben Bausteininstanz befinden.

ABBC$_{Schichten2}$

"Schichtenbaustein" 
"obereSchicht" 
$\forall x : \texttt{BindableArchElement}(x) \wedge$ $\texttt{boundABBEntitiesAreConnected}(\texttt{"obereSchicht"}, x)$
$\neg\texttt{isLowerLayerOfLL}^*(x, \texttt{"obereSchicht"})$

Das Adapterprädikat $\texttt{isLowerLayerOf}(x,y)$ stellt fest, ob Part x untere Schicht zu einem Part y ist. Hierzu wird geprüft, ob beide Parts in derselben Bausteininstanz des Schichtenbausteins an die jeweiligen Entitätenrollen gebunden sind:

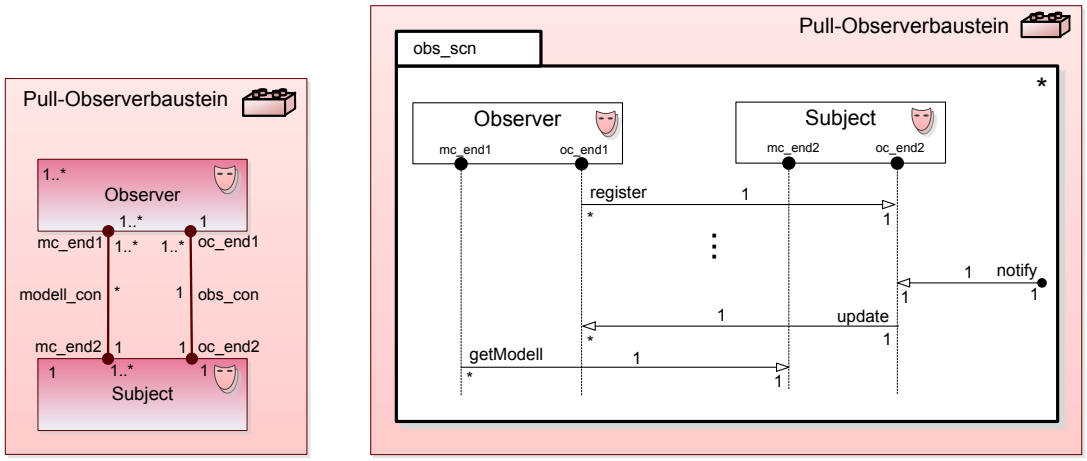
$$\begin{aligned}\texttt{isLowerLayerOf}(x,y) := \texttt{Part}(x) \wedge \texttt{Part}(y) \wedge \\ \exists b \exists o \exists c \exists d \exists m \exists n : \texttt{ABB}(b) \wedge \texttt{name}(b, \texttt{"Schichtenbaustein"}) \wedge \\ \texttt{ABBOccurrence}(o) \wedge \texttt{instantiatedABB}(o,b) \wedge \texttt{ABBEntity}(c) \wedge \\ \texttt{name}(c, \texttt{"untereSchicht"}) \wedge \texttt{ABBEntity}(d) \wedge \texttt{name}(d, \texttt{"obereSchicht"}) \wedge \\ \texttt{containsRole}(b,c) \wedge \texttt{containsRole}(b,d) \wedge \texttt{Binding}(m) \wedge \\ \texttt{Binding}(n) \wedge \texttt{containsBinding}(o,m) \wedge \texttt{containsBinding}(o,n) \wedge \\ \texttt{bindsArchElement}(m,x) \wedge \texttt{bindsArchElement}(n,y) \wedge \\ \texttt{bindsABBElement}(m,c) \wedge \texttt{bindsABBElement}(n,d)\end{aligned} \tag{AP 7}$$

Die oben erläuerte Schichtenbeziehung zwischen zwei Parts x und y wird basierend auf AP 7 durch $\texttt{isLowerLayerOfLL}^*(x,y)$ transitiv definiert. Hierbei wird allerdings die direkte Beziehung, wenn x und y in derselben Bausteininstanz gebunden sind, ausgenommen.

$$\begin{aligned}\texttt{isLowerLayerOfLL}^*(x,y) := \\ \exists z : \texttt{isLowerLayerOf}(x,z)^{1\,(\text{AP 7})} \wedge \texttt{isLowerLayerOfLL}^*(z,y)\end{aligned} \tag{AP 8}$$

8.2.2. Pull-Observerbaustein

Abbildung 8.2a zeigt die Struktur des Pull-Observerbausteins zu sehen. Der Pull-Observerbaustein ist eine Erweiterung des Observerbausteins. Beide besitzen die Entitätenrollen Observer und Subject, die durch eine Konnektorrolle obs_con miteinander verbunden werden. Neben der Benachrichtigung über Änderungen, die über obs_con erfolgen, wird beim Pull-Observer außerdem auch der Zugriff auf die geänderten Daten über modell_con modelliert. Durch eine Bausteininstanziierung darf bei beiden das Subject nur ein einziges Mal gebunden werden, während die Entitätenrolle Observer mehrfach auftreten darf. Diese Einschränkungen sind durch die Multiplizitäten an den Entitätenrollen dokumentiert. Die Konnektorrollenenden oc_end1 und oc_end2, die über obs_con verbunden sind, dürfen jeweils nur ein Mal pro gebundener Entitätenrolle auftreten. Die Konnektorrolle obs_con ist singleTyped und darf pro gebundenem oc_end2 an beliebig vielen ausgehenden Verbindungen zu oc_end1-Elementen gebunden sein. Anders herum darf über obs_con jedes an Observer gebundene Architekturelement allerdings nur mit einem gebundenem Subject verbunden sein (1 an dem Konnektorrollenende). Die multiTyped Konnektorrolle modell_con weist dieselben Eingangs- und Ausgangsmultiplizitäten auf wie obs_con. Hingegen dürfen ihre Konnektorrollenenden mc_end1 und mc_end2 beliebig oft im Kontext einer Entitätenrollenbindung gebunden werden.



(a) (b)

Abbildung 8.2: Bausteine des Beispielsystems: Struktur (a) und Verhalten (b) des Pull-Observerbausteins.

Von den Architekturbausteinen des Beispielsystems definiert der Pull-Observerbaustein das umfangreichste Verhalten. Dies wird durch das Bausteinszenario obs_scn modelliert (vgl. Abb. 8.2b). Zu Beginn registrieren sich eine Menge von an Observer gebundene Elemente am gebundenen Subject. Dies erfolgt durch an die Nachrichtenrolle register gebundene Interaktionen, die von mehreren an oc_end1 gebundenen Elementen ausgehen können (* am Pfeilanfang). Alle diese Interaktionen müssen an einem einzigen an oc_end2 gebundenen Element am gebundenen Subject eintreffen. Irgendwann nach register erfolgt notify. Zwischen diesen beiden Nachrichtenrollen besteht kein direkter Zusammenhang; notify wird nicht durch register ausgelöst. Lediglich die zeitliche Reihenfolge ist vorgegeben. Dieser Sachverhalt wird durch senkrechte Pünktchen (⋮) zwischen den zugehörigen Pfeilen symbolisiert. notify erfolgt dabei an oc_end2 von unbekannter Quelle ausgehend ein Mal und löst update aus. Die Nachrichtenrolle update kann beliebige an oc_end1 gebundene Empfänger haben. Daraufhin wird an dem Konnektorrollenende mc_end1 die Nachrichtenrolle getModell ausgelöst. Bei allen Nachrichtenrollen dieses Bausteinszenarios ist die Art des Nachrichtenaustauschs nicht festgelegt. Hingegen müssen die Interaktionen, die an jeweils eine der Nachrichtenrollen gebunden sind, verhaltensäquivalent sein (Wert single durch 1 an der Pfeilmitte).

Der Pull-Observerbaustein definiert zwei Eigenschaftsbedingungen. Die erste wurde bereits in Abschnitt 6.5.3 detailliert als ausformulierter prädikatenlogischer Ausdruck vorgestellt. Der Vollständigkeit halber wird diese hier noch einem in der abkürzenden Form inklusive der Definition des verwendeten Adapterprädikats aufgeführt. Diese Bedingung überprüft, ob eine an die Nachrichtenrolle notify gebundene Interaktion durch eine andere Interaktion ausgelöst wird:

ABBC$_{Pull\text{-}Observer1}$

```
"Pull-Observerbaustein"
"notify"

boundHasActivatingMsg("notify")
```

Das verwendete Adapterprädikat boundHasActivatingMsg(a) identifiziert, ob die Interaktion a einen Auslöser besitzt:

$$\begin{aligned}\texttt{boundHasActivatingMsg}(\texttt{a}) := {} & \texttt{MsgTrace}(\texttt{a}) \wedge \\ & \exists \texttt{b} \exists \texttt{c} : \texttt{MsgOccurrence}(\texttt{b}) \wedge \texttt{MsgOccurrence}(\texttt{c}) \wedge \\ & \texttt{beginningOfMsgTrace}(\texttt{b},\texttt{a})^{1} \wedge \texttt{triggersMsgOcc}(\texttt{c},\texttt{b})\end{aligned} \qquad {}^{1(6.27)} \quad \text{(AP 9)}$$

Die zweite Eigenschaftsbedingung des Pull-Observerbausteins stellt sicher, dass auch alle zuvor registrierten Observer bei der Benachrichtigung berücksichtigt werden. Dazu werden alle Architekturelemente betrachtet, die an die Entitätenrolle Observer gebunden sind und sich zuvor durch das Aussenden einer an register gebundenen Interaktion registriert haben. Diese Architekturelemente müssen jeweils durch eine an update gebundene Interaktion benachrichtigt werden, sobald eine beliebige an notify gebundene Interaktion erfolgt:

ABBC$_{\textit{Pull-Observer2}}$

Pull-Observerbaustein 
"register", "Observer" 
boundABBEntityIsABBMsgSender("Observer","register") ∧ ∃"notify" : nextBoundABBMsg*("register","notify")
∃"update" : nextBoundABBMsg("notify","update") ∧ boundABBEntityIsABBMsgReceiver("Observer","update")

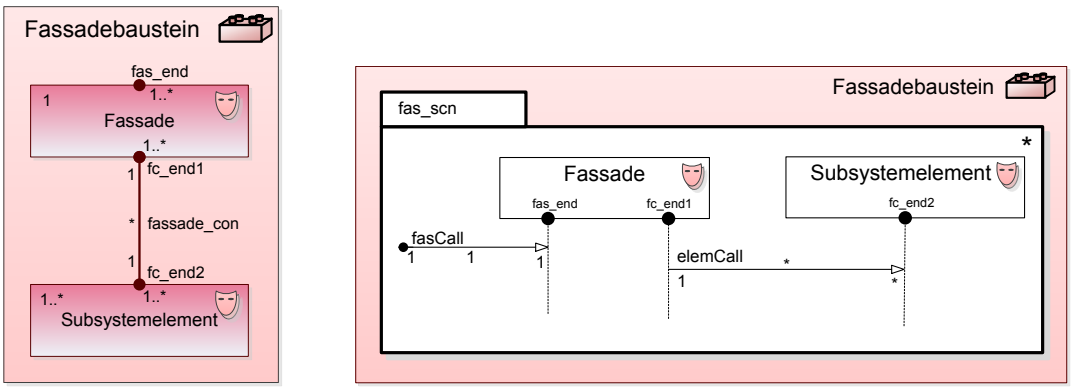
In der Formulierung dieser Eigenschaftsbedingung wurden drei Adapterprädikate eingesetzt. Das erste Adapterprädikat boundABBEntityIsABBMsgSender(b, a) stellt fest, ob das Architekturelement b der Sender der Interaktion a ist. Im Kontext dieser Eigenschaftsbedingung bezieht sich b auf Architekturelemente, die an Entitätenrollen wie Observer gebunden werden können, und a auf an register bindbare Elemente. Dementsprechend ist das Prädikat wahr, wenn ein Part b eine Portvariable besitzt, von der eine Nachrichtenspur a ausgeht.

$$\begin{aligned}\texttt{boundABBEntityIsABBMsgSender}(\texttt{b},\texttt{a}) := {} & \texttt{Part}(\texttt{b}) \wedge \texttt{MsgTrace}(\texttt{a}) \wedge \\ & \exists \texttt{o} \exists \texttt{m} \exists \texttt{p} : \texttt{MsgOccurrence}(\texttt{o}) \wedge \texttt{beginningOfMsgTrace}(\texttt{o},\texttt{a})^{1} \wedge \\ & \texttt{PortVariable}(\texttt{p}) \wedge \texttt{hasPortVariable}(\texttt{b},\texttt{p}) \wedge \\ & \texttt{Message}(\texttt{m}) \wedge \texttt{occurredMsg}(\texttt{o},\texttt{m}) \wedge \texttt{msgSender}(\texttt{m},\texttt{p})\end{aligned} \qquad {}^{1(6.27)} \quad \text{(AP 10)}$$

Analog zu AP 10 ist boundABBEntityIsABBMsgReceiver(b, a) definiert. Hier wird überprüft, ob ein Part b eine Portvariable als Empfänger eines Nachrichtenvorkommens a besitzt:

$$\begin{aligned}\texttt{boundABBEntityIsABBMsgReceiver}(\texttt{b},\texttt{a}) := {} & \texttt{Part}(\texttt{b}) \wedge \texttt{MsgTrace}(\texttt{a}) \wedge \\ & \exists \texttt{o} \exists \texttt{m} \exists \texttt{p} : \texttt{MsgOccurrence}(\texttt{o}) \wedge \texttt{endOfMsgTrace}(\texttt{o},\texttt{a})^{1} \wedge \\ & \texttt{PortVariable}(\texttt{p}) \wedge \texttt{hasPortVariable}(\texttt{b},\texttt{p}) \wedge \\ & \texttt{Message}(\texttt{m}) \wedge \texttt{occurredMsg}(\texttt{o},\texttt{m}) \wedge \texttt{msgReceiver}(\texttt{m},\texttt{p})\end{aligned} \qquad {}^{1(6.28)} \quad \text{(AP 11)}$$

Das Adapterprädikat nextBoundABBMsg*(a, c) testet hingegen, ob die beiden an Nachrichtenrollen gebundenen Elemente a und c irgendwann nacheinander erfolgen. Nachrichtenrollen werden gemäß des Integrationteils des Metamodells an Nachrichtenspuren gebunden, die kumulierende Integrationselemente sind und eine Folge von Nachrichtenvorkommen zusammenfassen. Nachrichtenspuren werden hier als nacheinander folgend betrachtet, wenn ihre jeweiligen ersten



(a) (b)

Abbildung 8.3: Bausteine des Beispielsystems: Struktur (a) und Verhalten (b) des Fassadebausteins.

Nachrichtenvorkommen zeitlich nacheinander erfolgen:

^{1}(6.27), ^{2}(6.6)

$$\begin{aligned}\texttt{nextBoundABBMsg}^*(\texttt{a},\texttt{c}) := {} & \texttt{MsgTrace}(\texttt{a}) \wedge \texttt{MsgTrace}(\texttt{c}) \wedge \\ & \exists \texttt{o} \exists \texttt{p} : \texttt{MsgOccurrence}(\texttt{o}) \wedge \texttt{beginningOfMsgTrace}(\texttt{o},\texttt{a})^1 \wedge \\ & \texttt{MsgOccurrence}(\texttt{p}) \wedge \texttt{beginningOfMsgTrace}(\texttt{p},\texttt{c}) \wedge \\ & \texttt{nextMsgOccurrence}^*(\texttt{o},\texttt{p})^2\end{aligned} \tag{AP 12}$$

Hingegen muss in der Variante $\texttt{nextBoundABBMsg}(\texttt{a},\texttt{c})$ das letzte Nachrichtenvorkommen der ersten Nachrichtenspur $\texttt{a}$ direkt vor dem ersten Nachrichtenvorkommen der zweiten Nachrichtenspur $\texttt{c}$ erfolgen:

^{1}(6.28), ^{2}(6.27)

$$\begin{aligned}\texttt{nextBoundABBMsg}^*(\texttt{a},\texttt{c}) := {} & \texttt{MsgTrace}(\texttt{a}) \wedge \texttt{MsgTrace}(\texttt{c}) \wedge \\ & \exists \texttt{o} \exists \texttt{p} : \texttt{MsgOccurrence}(\texttt{o}) \wedge \texttt{endOfMsgTrace}(\texttt{o},\texttt{a})^1 \wedge \\ & \texttt{MsgOccurrence}(\texttt{p}) \wedge \texttt{beginnigOfMsgTrace}(\texttt{p},\texttt{c})^2 \wedge \\ & \texttt{nextMsgOccurrence}(\texttt{o},\texttt{p})\end{aligned} \tag{AP 13}$$

8.2.3. Fassadebaustein

Die Struktur des Fassadebaustein ist in Abbildung 8.3a dargestellt. Seine beiden Entitätenrollen Fassade und Subsystemelement sind durch die Konnektorrolle fassade_con verbunden. Während bei einer Bausteininstanziierung Fassade genau ein Mal gebunden wird, muss es mindestens eine Bindung der Rolle Subsystemelement geben. Zwischen einem Fassade-Element und den Subsystemelement-Elementen dürfen beliebig viele fassade_con-Verbindungen verlaufen, die sich nicht ähnlich sein müssen. Allerdings darf jeweils nur eine Verbindung pro Endpunkt existieren (Multiplizität 1 an den Konnektorrollenenden fc_end1 und fc_end2). Pro gebundener Entitätenrolle dürfen die beiden Konnektorrollenenden hingegen mehrmals gebunden werden. Darüber hinaus besitzt die Entitätenrolle Fassade ein Konnektorrollenende fas_end ohne verbindende Konnektorrolle mit der Multiplizität 1..*.

Im Fassadebaustein kapselt die Fassade die Funktionalität der Elemente des Fassadensubsystems. Interaktionen, die von dritten ausgehen, erfolgen zunächst mit der Fassade und werden von dieser an die entsprechenden Subsystemelemente weitergereicht. Das Bausteinszenario

in Abbildung 8.3b beschreibt dieses abstrakte Verhalten[9]. Die initiale Interaktion ist durch die Nachrichtenrolle fasCall dargestellt, die ausgehend von einer unbekannten Quelle am Konnektorrollenende fas_end der Entitätenrolle Fassade eingeht. Pro gebundenem Bausteinszenario kann es nur je einen Empfänger und einen Sender dieser Nachrichtenrolle geben (1 an den Pfeilenden). Implizit kann diese Nachrichtenrolle dadurch auch nur an eine Interaktion im Kontext des Bausteinszenarios gebunden werden. Als Folge von fasCall erfolgt ausgehend von fc_end1 die Nachrichtenrolle elemCall mit Ziel fc_con2 an der Entitätenrolle Subsystemelement. Diese Nachrichtenrolle kann pro gebundenem Bausteinszenario an Interaktionen mit verschiedenen Empfängern gebunden werden (* an der Pfeilspitze). Folglich hängt die Bindungshäufigkeit dieser Nachrichtenrolle von der Anzahl der Empfänger ab. Zudem können sich diese Interaktionen in Bezug auf die Verhaltensäquivalenz unterscheiden (Eigenschaft multi durch * an der Pfeilmitte).

Die Zugriffsbeschränkung auf die Subsystemelemente einer Fassade können je nach Ausprägung unterschiedlich strikt ausfallen. Im vorliegenden Fall sind Zugriffe von außen mit Einschränkungen auf die Subsystemelemente erlaubt. Zugriffe dürfen allerdings nicht über die Zugriffspunkte erfolgen, über die das Fassadenelement selber zugreift und die bei der Instanziierung des Bausteins an `fc_end2` gebunden sind. Anderen Subsystemelemente derselben Fassadeninstanz ist der Zugriff gestattet.

ABBC$_{Fassade1}$

`"Fassadebaustein"` 
`"fc_end2"` 
$\forall$ `k` : `BindableArchElement(k)` $\wedge$ `isOppositeToBoundCEnd(k,"fc_end2")`
($\exists$ `"Fassade"` : `belongsToBoundABBEntity(k,"Fassade")`)$\vee$ ($\exists$ `"Subsystemelement"` : `belongsToBoundABBEntity(k,"Subsystemelement")`)

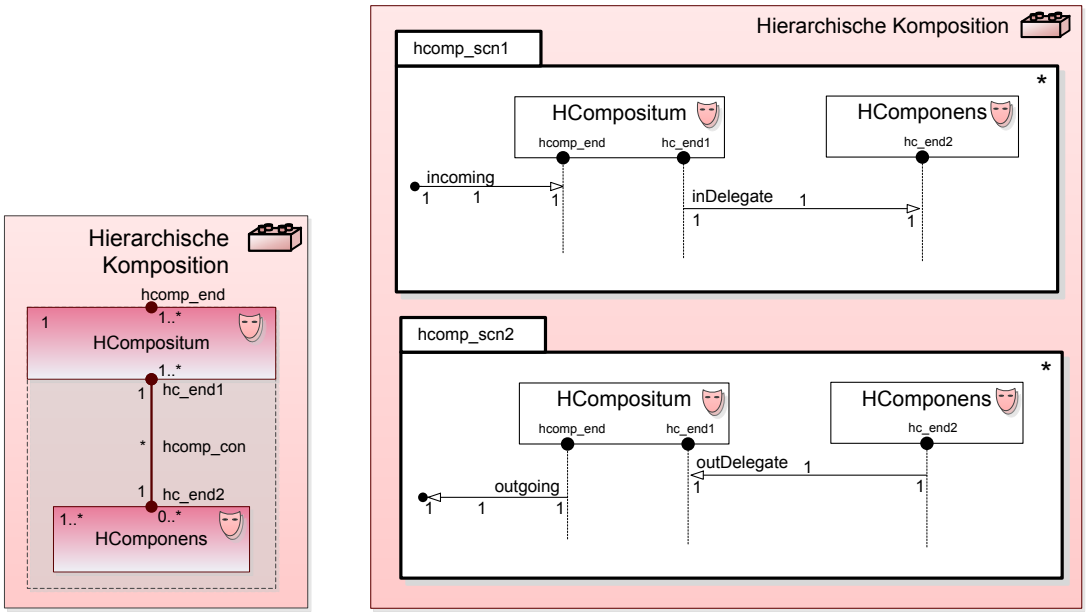
Diese Eigenschaftsbedingung setzt das Adapterprädikat `isOppositeToBoundCEnd(k, e)` ein, um festzustellen, ob die Architekturelemente `k` und `e` miteinander verbunden. Da es sich hierbei um Portvariablen handelt, muss für die Gültigkeit dieses Prädikats ein Verbindungselement zwischen diesen beiden existieren:

$$\texttt{isOppositeToBoundCEnd(k,e)} := \texttt{PortVariable(k)} \wedge \texttt{PortVariable(e)} \wedge \exists \texttt{l} : \texttt{Link(l)} \wedge \texttt{linkEnd(l,k)}^{1} \wedge \texttt{linkEnd(l,e)} \wedge \neg(\texttt{k} \equiv \texttt{e}) \quad \text{(AP 14)}$$

[1](6.3)

Das Adapterprädikat `belongsToBoundABBEntity(k, e)` ist wahr, wenn `k` eine Portvariable eines Parts `e` ist:

$$\texttt{belongsToBoundABBEntity(k,e)} := \texttt{PortVariable(k)} \wedge \texttt{Part(e)} \wedge \texttt{hasPortVariable(e,k)} \quad \text{(AP 15)}$$



(a) (b)

Abbildung 8.4: Bausteine des Beispielsystems: Struktur (a) und Verhalten (b) der Hierarchischen Komposition.

8.2.4. Hierarchische Komposition

In Abbildung 8.4a ist die Struktur der Hierarchischen Komposition modelliert[10]. Dieser Baustein besitzt die Entitätenrollen HCompositum und HComponens, wobei HComponens in HCompositum geschachtelt ist. Wie durch den entsprechenden Wert angegeben, unterscheidet sich die Bindungshäufigkeit der beiden Entitätenrollen. Während pro Bausteininstanz genau ein HCompositum gebunden wird, kann es beliebig viele Bindungen von HComponens geben. Die Multiplizitäten der Konnektorrollenenden werden im Kontext der Bindung der zugehörigen Entitätenrolle ausgewertet. So werden die beiden Konnektorrollenenden hc_end0 und hc_end1 an HCompositum beliebig häufig aber mindestens einmal gebunden. Indessen muss das Konnektorrollenende hc_end2 nicht im Kontext jeder Bindung von HComponens gebunden werden, wobei die Häufigkeit nach oben nicht beschränkt ist. An beiden Enden der Konnektorrolle ist eine Multiplizität von 1 angegeben. Somit kann jedes Element, das an das Konnektorrollenende hc_end1 oder hc_end2 gebunden ist, mit immer nur einem Element verbunden sein, das an die jeweils andere Rolle gebunden ist. Dabei brauchen die Verbindungen zwischen den gebundenen Konnektorrollenenden einer Bausteininstanz nicht ähnlich zueinander sein (multiTyped).

Bei der Hierarchischen Komposition kapselt das äußere Element, das HCompositum, die inneren Elemente, die HComponens. Ein- und ausgehende Interaktionen mit an HComponens gebundenen Architekturelementen erfolgen über das umgebende HCompositum. Das Verhalten der Hierarchischen Komposition wird durch zwei Bausteinszenarios beschrieben, wobei eines eingehende und das andere ausgehende Interaktionen betrachtet (vgl. Abb. 8.4b). Das erste

[9] Das Verhalten des Fassadebaustein wurde bereits als Beispiel für die Verhaltensbeschreibung eines Architekturbausteins in Abschnitt 6.3.4 erläutert.

[10] Die Hierarchische Komposition wurde bereits in Abschnitt 6.3.2 als Beispiel für die Strukturbeschreibung eines Architekturbausteins erläutert.

Szenario (hcomp_scn1) beschreibt eingehende Interaktion und beginnt mit der Nachrichtenrolle incoming. incoming geht von einer unbekannten Quelle aus und an hcomp_end am HCompositum ein. Dies löst die Nachrichtenrolle inDelegate von hc_end1 zu hc_end2 aus. Entsprechend der angegebenen Multiplizitäten können beide Nachrichtenrollen pro gebundenem Bausteinszenario nur mit jeweils genau einem Empfänger und Sender gebunden werden. Vom HComponens ausgehende Interaktionen werden durch das zweite Szenario (hcomp_scn2) beschrieben. Hier wird der Ablauf durch die Nachrichtenrolle outDelegate, die von hc_end2 ausgeht, initiiert. Diese an hc_end1 eingehende Nachrichtenrolle löst an hcomp_end die Nachrichtenrolle outgoing aus. outgoing hat ein dem Baustein nicht bekanntes Ziel. Auch in diesem Bausteinszenario können die Nachrichtenrollen nur mit jeweils genau einem Sender und Empfänger gebunden werden.

Die Hierarchische Komposition definiert zwei Eigenschaftsbedingungen. Zunächst wird verlangt, dass alle entsprechenden Architekturelemente, die innerhalb eines an HCompositum gebundenem Architekturelements liegen, an die Entitätenrolle HComponens in derselben Bausteininstanz gebunden sind:

ABBC$_{HierKomposition1}$

`"HierarchischeKomposition"`
`"HCompositum"`
$\forall$`k : BindableArchElement(k)` $\wedge$ `boundABBEntityEncapsulates("HCompositum",k)`
$\exists$`"HComponens" : (k` $\equiv$ `"HComponens")`

Ob ein Architekturelement `i`, das einem Part entspricht, innerhalb eines anderen Parts `o` liegt, stellt das Adapterprädikat `boundABBEntityEncapsulates(o, i)` fest. Ein Part enthält allerdings nicht direkt einen anderen Part. Vielmehr muss dazu `i` Teil der Konfiguration des Komponententyps von `o` sein:

$$\begin{aligned}&\texttt{boundABBEntityEncapsulates}(o,i) := \texttt{Part}(o) \wedge \texttt{Part}(i) \wedge \\ &\quad \exists c \exists d : \texttt{Component}(c) \wedge \texttt{hasType}(o,c) \wedge \\ &\quad \texttt{Configuration}(d) \wedge \texttt{hasConfiguration}(c,d) \wedge \texttt{containsPart}(d,i)\end{aligned} \tag{AP 16}$$

Die zweite Eigenschaftsbedingung der Hierarchischen Komposition betrifft den Zugriff auf einen Teil der Elemente und ähnelt damit der Eigenschaftsbedingung **ABBC**$_{Fassade1}$ des Fassadebausteins. Diese Ähnlichkeit liegt in der Tatsache begründet, dass beide Bausteine eine Art Kapselung mit allerdings unterschiedlicher Zielsetzung realisieren. Während sich bei der Fassade die Einschränkung von externen Zugriffen nur auf bestimmte Interaktionspunkte bezieht, umfasst sie bei der Hierarchischen Komposition das jeweils komplette HComponens. So dürfen mit einem HComponens nur andere HComponens derselben Bausteininstanz oder das zugehörige HCompositum interagieren. Es erfolgt keine Unterscheidung hinsichtlich der Interaktionspunkte.

ABBC$_{HierKomposition2}$

HierarchischeKomposition	
"HComponens"	
$\forall$k : BindableArchElement(k) $\wedge$ boundABBEntitiesAreConnected("HComponens",k)	
($\exists$"HComponens" : (k $\equiv$ "HComponens")) $\vee$ ($\exists$"HCompositum" : (k $\equiv$ "HCompositum"))	

Das Adapterprädikat boundABBEntitiesAreConnected(a,b) ist wahr, wenn die beiden Architekturelemente a und b Parts sind, die über ein Verbindungselement miteinander verbunden sind:

$$\begin{aligned}\texttt{boundABBEntitiesAreConnected(a,b)} := {} & \texttt{Part(a)} \wedge \texttt{Part(b)} \wedge \\ & \exists \texttt{l} \exists \texttt{p} \exists \texttt{q} : \texttt{Link(l)} \wedge \texttt{linkEnd(l,p)} \wedge \texttt{linkEnd(l,q)} \wedge \neg(\texttt{p} \equiv \texttt{q}) \wedge \\ & \texttt{hasPortVariable(a,p)} \wedge \texttt{hasPortVariable(b,q)}\end{aligned} \tag{AP 17}$$

8.3. Identifikation von Verletzungen im Beispielsystem

In Abschnitt 4.3 wurde in mehreren Schritten die Architektur für ein Beispielsystem erstellt. Verschiedene Inkonsistenzen, die dabei auftraten, wurden nur durch Zufall aufgedeckt. Der folgende Abschnitt überträgt das Konzept des bausteinbasierten Architekturentwurfs auf dieses Beispielsystem und seine Entwicklungsschritte. Jeder Entwicklungsschritt wird nun systematisch vollzogen, indem dem konstruktiven Vorgang direkt eine Überprüfung des Ergebnisses folgt. Die zu überprüfenden Eigenschaftsbedingungen der Architekturbausteine wurden in dem vorherigen Abschnitt durch prädikatenlogische Sätze formuliert. Diese werden auf Basis der formalen Repräsentation des Beispielsystems ausgewertet. Um die Auswertung nachvollziehen zu können, wird in den folgenden Unterabschnitten die formale Repräsentation der jeweiligen Entwicklungsversion notiert. Dabei wird die Darstellung auf die Relationen und Elemente des formalen Modells reduziert, die die für die Auswertung relevanten Details abbilden.

Die folgenden Abschnitte 8.3.1 bis 8.3.4 erläutern die Anwendung des bausteinbasierten Architekturentwurfs auf die Entwicklung des Beispielsystems. Es werden an dieser Stelle nur die Entwicklungsschritte 4 bis 7 betrachtet. Diese Auswahl wurde getroffen, weil die Ergebnisse der Schritte 4 bis 6 jeweils Verletzungen von Eigenschaftsbedingungen aufweisen. Mit Entwicklungsschritt 7 wurde anschließend wieder ein bausteinkonsistentes Modell realisiert. Der vorausgehende Entwicklungsschritt 3 wurde bereits ansatzweise in Abschnitt 7.4 wiedergegeben.

8.3.1. Entwicklungsschritt 4

Vor Entwicklungsschritt 4 handelt es sich bei dem Beispielsystem um ein dreischichtiges System. Die drei Schichten zur Präsentation, Anwendungslogik und Datenhaltung werden durch Parts repräsentiert, die jeweils nur mit den in der Schichtenhierarchie benachbarten interagieren können. Die beiden Parts des Systems, die die Schichten zur Präsentation bzw. Anwendungslogik repräsentieren, sind zudem durch komplexe Komponenten getypt. In Entwicklungsschritt 4

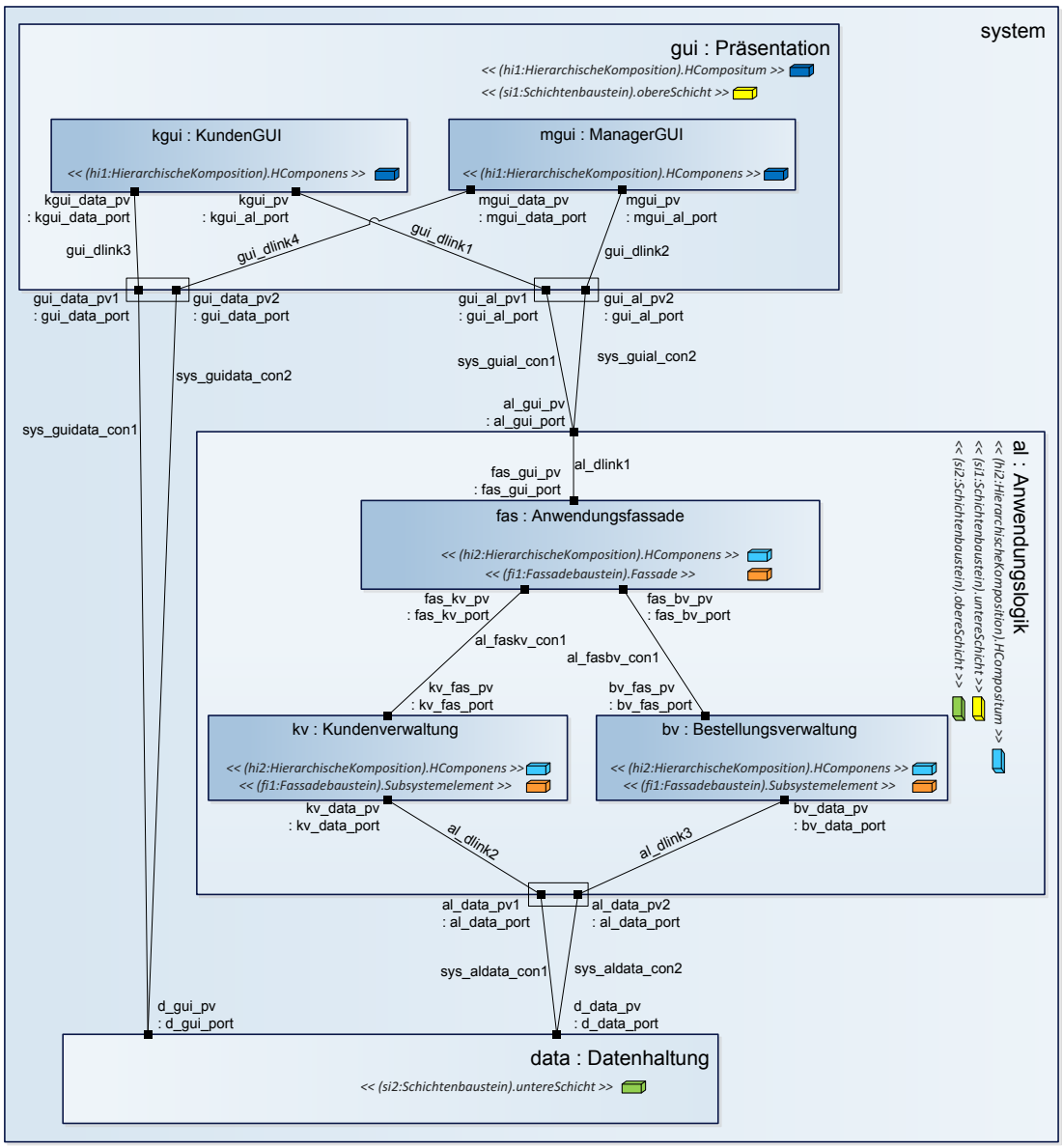


Abbildung 8.5: Systemkonfiguration des Beispielsystems nach Entwicklungsschritt 4.

wird den Parts der Schichten Datenhaltung und Präsentation ermöglicht, direkt miteinander zu interagieren. Es wird kein Architekturbaustein instanziiert, sondern lediglich eine Architekturerweiterung durchgeführt.

Aufbau des Beispielsystems

Die Struktur des Beispielsystems system ist in Abbildung 8.5 dargestellt. Durch Entwicklungsschritt 4 sind keine neuen Parts hinzugekommen. Stattdessen wird eine Verbindung zwischen den Parts data und kgui sowie data und mgui realisiert. Hierzu werden die Konnektoren sys_guidata_con1 und sys_guidata_con2 zwischen data und gui sowie die Delegationskonnektoren gui_dlink_dlink3 und gui_dlink4 zwischen gui und kgui bzw. mgui erstellt. Als Anknüpfungspunkte

dieser vier Verbindungselemente werden zudem mehrere Portvariablen an den entsprechenden Parts erstellt.

Formal wird das Beispielsystem nach Entwicklungsschritt 4 durch $\mathfrak{M}^4$ repräsentiert. Im Folgenden wird ein Teil der Relationen von $\mathfrak{M}^4$ notiert. Dieser Auszug enthält die wesentlichen Informationen, um im Anschluss die Überprüfung der Eigenschaftsbedingungen nachvollziehen zu können.

$\texttt{System}^{\mathfrak{M}^4} = \texttt{System}^{\mathfrak{M}^3} = \{\texttt{system}\}$

$\texttt{Component}^{\mathfrak{M}^4} = \texttt{Component}^{\mathfrak{M}^3} = \{\texttt{Präsentation}, \texttt{KundenGUI}, \texttt{ManagerGUI}, \texttt{Anwendungslogik}, \texttt{Anwendungsfassade}, \texttt{Kundenverwaltung}, \texttt{Bestellungsverwaltung}, \texttt{Datenhaltung}\}$

$\texttt{Part}^{\mathfrak{M}^4} = \texttt{Part}^{\mathfrak{M}^3} = \{\texttt{gui}, \texttt{kgui}, \texttt{mgui}, \texttt{al}, \texttt{fas}, \texttt{kv}, \texttt{bv}, \texttt{data}\}$

$\texttt{Configuration}^{\mathfrak{M}^4} = \texttt{Configuration}^{\mathfrak{M}^3} = \{\texttt{sys\_conf}, \texttt{gui\_conf}, \texttt{al\_conf}\}$

$\texttt{hasConfiguration}^{\mathfrak{M}^4} = \texttt{hasConfiguration}^{\mathfrak{M}^3} = \{(\texttt{system}, \texttt{sys\_conf}), (\texttt{Präsentation}, \texttt{gui\_conf}), (\texttt{Anwendungslogik}, \texttt{al\_conf})\}$

$\texttt{containsPart}^{\mathfrak{M}^4} = \texttt{containsPart}^{\mathfrak{M}^3} = \{(\texttt{sys\_conf}, \texttt{gui}), (\texttt{sys\_conf}, \texttt{al}), (\texttt{sys\_conf}, \texttt{data}), (\texttt{gui\_conf}, \texttt{kgui}), (\texttt{gui\_conf}, \texttt{mgui}), (\texttt{al\_conf}, \texttt{fas}), (\texttt{al\_conf}, \texttt{kv}), (\texttt{al\_conf}, \texttt{bv})\}$

$\texttt{Connector}^{\mathfrak{M}^4} = \{\texttt{sys\_guial1}, \texttt{sys\_guial2}, \texttt{sys\_guidata\_con1}, \texttt{sys\_guidata\_con2}, \texttt{sys\_aldata\_con1}, \texttt{sys\_aldata\_con2}, \texttt{al\_faskv\_con1}, \texttt{al\_fasbv\_con1}\}$

$\texttt{containsConnector}^{\mathfrak{M}^4} = \{(\texttt{sys\_conf}, \texttt{sys\_guial1}), (\texttt{sys\_conf}, \texttt{sys\_guial2}), (\texttt{sys\_conf}, \texttt{sys\_guidata\_con1}), (\texttt{sys\_conf}, \texttt{sys\_guidata\_con2}), (\texttt{sys\_conf}, \texttt{sys\_aldata\_con1}), (\texttt{sys\_conf}, \texttt{sys\_aldata\_con2}), (\texttt{al\_conf}, \texttt{al\_faskv\_con1}), (\texttt{al\_conf}, \texttt{al\_fasbv\_con1})\}$

$\texttt{DelegatorLink}^{\mathfrak{M}^4} = \texttt{DelegatorLink}^{\mathfrak{M}^3} = \{\texttt{gui\_dlink1}, \texttt{gui\_dlink2}, \texttt{gui\_dlink3}, \texttt{gui\_dlink4}, \texttt{al\_dlink1}, \texttt{al\_dlink2}, \texttt{al\_dlink3}\}$

$\texttt{containsDelegatorLink}^{\mathfrak{M}^4} = \texttt{containsDelegatorLink}^{\mathfrak{M}^3} = \{(\texttt{al}, \texttt{al\_dlink1}), (\texttt{al}, \texttt{al\_dlink2}), (\texttt{al}, \texttt{al\_dlink3}), (\texttt{gui}, \texttt{gui\_dlink1}), (\texttt{gui}, \texttt{gui\_dlink2}), (\texttt{gui}, \texttt{gui\_dlink3}), (\texttt{gui}, \texttt{gui\_dlink4})\}$

$\texttt{PortVariable}^{\mathfrak{M}^4} = \{\texttt{gui\_data\_pv1}, \texttt{kgui\_data\_pv}, \texttt{kgui\_pv}, \texttt{al\_gui\_pv}, \texttt{fas\_kv\_pv}, ...\}$

$\texttt{hasPortVariable}^{\mathfrak{M}^4} = \{(\texttt{gui}, \texttt{gui\_data\_pv1}), (\texttt{gui}, \texttt{gui\_data\_pv2}), (\texttt{gui}, \texttt{gui\_al\_pv1}), (\texttt{gui}, \texttt{gui\_al\_pv2}), (\texttt{kgui}, \texttt{kgui\_data\_pv}), (\texttt{kgui}, \texttt{kgui\_pv}), (\texttt{mgui}, \texttt{mgui\_data\_pv}), (\texttt{mgui}, \texttt{mgui\_pv}), (\texttt{al}, \texttt{al\_gui\_pv}), (\texttt{al}, \texttt{al\_data\_pv1}), (\texttt{al}, \texttt{al\_data\_pv2}), (\texttt{fas}, \texttt{fas\_gui\_pv}), (\texttt{fas}, \texttt{fas\_kv\_pv}), (\texttt{fas}, \texttt{fas\_bv\_pv}), (\texttt{kv}, \texttt{kv\_fas\_pv}), (\texttt{bv}, \texttt{bv\_fas\_pv}), (\texttt{kv}, \texttt{kv\_data\_pv}), (\texttt{bv}, \texttt{bv\_data\_pv}), (\texttt{data}, \texttt{d\_gui\_pv}), (\texttt{data}, \texttt{d\_data\_pv})\}$

$\texttt{linkSource}^{\mathfrak{M}^4} = \{(\texttt{sys\_aldata\_con1}, \texttt{al\_data\_pv1}), (\texttt{sys\_aldata\_con2}, \texttt{al\_data\_pv2}), (\texttt{sys\_guidata\_con1}, \texttt{gui\_data\_pv1}), (\texttt{sys\_guidata\_con2}, \texttt{gui\_data\_pv2}), (\texttt{sys\_guial1}, \texttt{gui\_al\_pv1}), (\texttt{sys\_guial2}, \texttt{gui\_al\_pv2}), (\texttt{al\_faskv\_con1}, \texttt{fas\_kv\_pv}), (\texttt{al\_fasbv\_con1}, \texttt{fas\_bv\_pv}), (\texttt{gui\_dlink1}, \texttt{gui\_al\_pv1}), (\texttt{gui\_dlink2}, \texttt{gui\_al\_pv2}), (\texttt{gui\_dlink3}, \texttt{gui\_data\_pv1}), (\texttt{gui\_dlink4}, \texttt{gui\_data\_pv2}), (\texttt{al\_dlink1}, \texttt{al\_gui\_pv}), (\texttt{al\_dlink2}, \texttt{al\_data\_pv1}), (\texttt{al\_dlink3}, \texttt{al\_data\_pv2})\}$

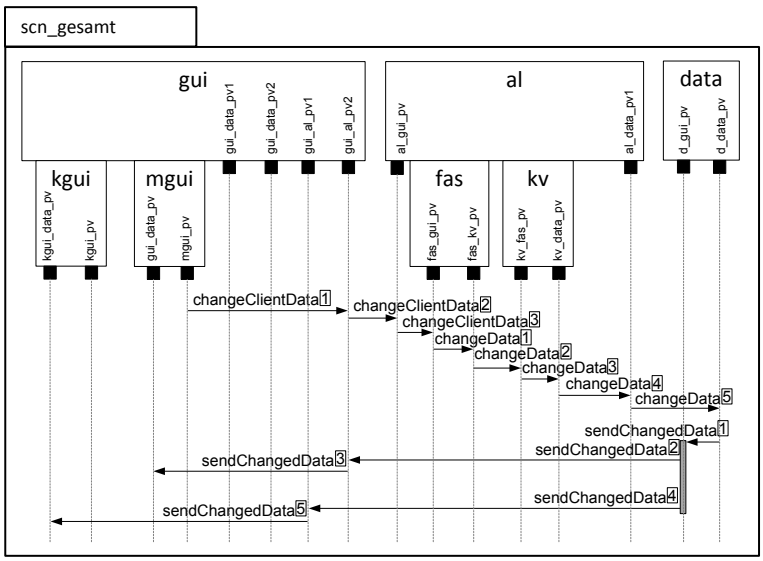


Abbildung 8.6: Szenario scn_gesamt des Beispielsystems nach Entwicklungsschritt 4.

$$\texttt{linkTarget}^{\mathfrak{M}^4} = \{(\texttt{sys\_aldata\_con1}, \texttt{d\_data\_pv}), (\texttt{sys\_aldata\_con2}, \texttt{d\_data\_pv}),$$
$$(\texttt{sys\_guidata\_con1}, \texttt{d\_gui\_pv}), (\texttt{sys\_guidata\_con2}, \texttt{d\_gui\_pv}),$$
$$(\texttt{sys\_guial1}, \texttt{al\_gui\_pv}), (\texttt{sys\_guial2}, \texttt{al\_gui\_pv}),$$
$$(\texttt{al\_faskv\_con1}, \texttt{kv\_fas\_pv}), (\texttt{al\_fasbv\_con1}, \texttt{bv\_fas\_pv}), (\texttt{gui\_dlink1}, \texttt{kgui\_pv}),$$
$$(\texttt{gui\_dlink2}, \texttt{mgui\_pv}), (\texttt{gui\_dlink3}, \texttt{kgui\_data\_pv}), (\texttt{gui\_dlink4}, \texttt{mgui\_data\_pv}),$$
$$(\texttt{al\_dlink1}, \texttt{fas\_gui\_pv}), (\texttt{al\_dlink2}, \texttt{kv\_data\_pv}), (\texttt{al\_dlink3}, \texttt{bv\_data\_pv})\}$$

Abbildung 8.6 zeigt das Szenario scn_gesamt des Beispielsystems. Das Szenario modelliert den Ablauf einer von mgui ausgehenden Änderungen in der Datenhaltung und der Weitergabe der geänderten Daten an kgui und mgui. Dabei erfolgt der Austausch der geänderten Daten direkt zwischen data und den beiden gui-Elementen durch sendChangedData2 und sendChangedData3 bzw. sendChangedData4 und sendChangedData5.

Dieses Szenario wird durch $\mathfrak{M}^4$ u.a. mit den folgenden Relationen formal repräsentiert. Dabei werden ebenso wie im Fall der Strukturbeschreibung verschiedene Relationen ausgelassen. Zudem werden nicht alle Elemente, die im Diagramm notiert sind, aufgeführt.

$$\texttt{Scenario}^{\mathfrak{M}^4} = \texttt{Scenario}^{\mathfrak{M}^3} = \{\texttt{scn\_gesamt}\}$$

$$\texttt{MsgOccurrence}^{\mathfrak{M}^4} = \{\texttt{changeClientData1}, \texttt{changeClientData2}, \texttt{changeClientData3},$$
$$\texttt{changeData1}, \texttt{changeData2}, \texttt{changeData3}, \texttt{changeData4}, \texttt{changeData5},$$
$$\texttt{sendChangedData1}, \texttt{sendChangedData2}, \texttt{sendChangedData3}, \texttt{sendChangedData4},$$
$$\texttt{sendChangedData5}, \texttt{getClientData1}, ...\}$$

$$\texttt{nextMsgOccurrence}^{\mathfrak{M}^4} = \{(\texttt{changeData5}, \texttt{sendChangedData1}),$$
$$(\texttt{sendChangedData1}, \texttt{sendChangedData2}),$$
$$(\texttt{sendChangedData2}, \texttt{sendChangedData3}), ...\}$$

$$\texttt{triggersMsgOccurence}^{\mathfrak{M}^4} = \{(\texttt{changeData5}, \texttt{sendChangedData1}),$$
$$(\texttt{sendChangedData1}, \texttt{sendChangedData2}),$$
$$(\texttt{sendChangedData1}, \texttt{sendChangedData4}), ...\}$$

$\texttt{msgType}^{\mathfrak{M}^4} = \{(\texttt{changeClientData2}, \texttt{"synchronous"}), (\texttt{changeData5}, \texttt{"synchronous"}),$
$(\texttt{sendChangedData2}, \texttt{"synchronous"}), (\texttt{sendChangedData4}, \texttt{"synchronous"}), ...\}$

$\texttt{occurredMsg}^{\mathfrak{M}^4} = \{(\texttt{changeClientData1}, \texttt{changeClientData\_msg1}),$
$(\texttt{changeData1}, \texttt{changeData\_msg1}), (\texttt{changeData2}, \texttt{changeData\_msg2}), ...\}$

$\texttt{msgSender}^{\mathfrak{M}^4} = \{(\texttt{changeClientData\_msg1}, \texttt{mgui\_pv}), (\texttt{changeData\_msg1}, \texttt{fas\_gui\_pv}),$
$(\texttt{changeData\_msg2}, \texttt{fas\_kv\_pv}), ...\}$

$\texttt{msgReceiver}^{\mathfrak{M}^4} = \{(\texttt{changeClientData\_msg1}, \texttt{fas\_gui\_pv}),$
$(\texttt{changeData\_msg1}, \texttt{fas\_kv\_pv}), (\texttt{changeData\_msg2}, \texttt{kv\_fas\_pv}), ...\}$

Instanzen der Architekturbausteine und Überprüfung ihrer Eigenschaftsbedingungen

Nach der Durchführung von Entwicklungsschritt 4 besteht das Beispielsystem ebenso wie zuvor aus fünf Bausteininstanzen, die auf drei verschiedene Architekturbausteine zurückgehen. Verbunden mit diesen Bausteininstanzen sind verschiedene Bindungen zwischen bindbaren Bausteinelementen und bindbaren Architekturelementen. Zumindest die Bindung der Entitätenrollen mit den dazugehörigen Bausteininstanzen kann in Abbildung 8.5 nachvollzogen werden. Mit Hilfe der Bindungen können im Anschluss die Eigenschaftsbedingungen der drei verwendeten Bausteine getestet werden.

Bausteininstanzen und Bindungen zählen zum Integrationsteil, der als Teil von $\mathfrak{M}^4$ definiert ist. Der Integrationsteil wird durch die folgenden Relationen formal repräsentiert. Dabei werden aus Gründen der Übersichtlichkeit weder alle Relationen noch alle Elemente der notierten Relationen aufgeführt. Dokumentiert sind vor allem die Relationen und Elemente mit Bezug zu den bindbaren Bausteinelementen, die für die Überprüfung der Eigenschaftsbedingungen der drei verwendeten Bausteine relevant sind.

$\texttt{ABBOccurrence}^{\mathfrak{M}^4} = \texttt{ABBOccurrence}^{\mathfrak{M}^3} = \{\texttt{si1}, \texttt{si2}, \texttt{hi1}, \texttt{hi2}, \texttt{fi1}\}$

$\texttt{instantiatedABB}^{\mathfrak{M}^4} = \texttt{instantiatedABB}^{\mathfrak{M}^3} = \{(\texttt{si1}, \texttt{"Schichtenbaustein"}),$
$(\texttt{si2}, \texttt{"Schichtenbaustein"}), (\texttt{hi1}, \texttt{"HierarchischeKomposition"}),$
$(\texttt{hi2}, \texttt{"HierarchischeKomposition"}), (\texttt{fi1}, \texttt{"Fassadebaustein"})\}$

$\texttt{bindsABBElement}^{\mathfrak{M}^4} = \texttt{bindsABBElement}^{\mathfrak{M}^3} = \{(\texttt{si1\_eb1}, \texttt{"obereSchicht"}),$
$(\texttt{si1\_eb2}, \texttt{"untereSchicht"}), (\texttt{si2\_eb1}, \texttt{"obereSchicht"}),$
$(\texttt{si2\_eb2}, \texttt{"untereSchicht"}), (\texttt{hi1\_eb1}, \texttt{"Hcompositum"}), (\texttt{hi1\_eb2}, \texttt{"Hcomponens"}),$
$(\texttt{hi1\_eb3}, \texttt{"Hcomponens"}), (\texttt{hi2\_eb1}, \texttt{"Hcompositum"}), (\texttt{hi2\_eb2}, \texttt{"Hcomponens"}),$
$(\texttt{hi2\_eb3}, \texttt{"Hcomponens"}), (\texttt{hi2\_eb4}, \texttt{"Hcomponens"}), (\texttt{fi1\_eb1}, \texttt{"Fassade"}),$
$(\texttt{fi1\_eb2}, \texttt{"Subsystemelement"}), (\texttt{fi1\_eb3}, \texttt{"Subsystemelement"}),$
$(\texttt{fi1\_ceb1}, \texttt{"fc\_end2"}), (\texttt{fi1\_ceb2}, \texttt{"fc\_end2"}), ...\}$

$\texttt{bindsArchElement}^{\mathfrak{M}^4} = \texttt{bindsArchElement}^{\mathfrak{M}^3} = \{(\texttt{si1\_eb1}, \texttt{gui}), (\texttt{si1\_eb2}, \texttt{al}),$
$(\texttt{si2\_eb1}, \texttt{al}), (\texttt{si2\_eb2}, \texttt{data}), (\texttt{hi1\_eb1}, \texttt{gui}), (\texttt{hi1\_eb2}, \texttt{kgui}), (\texttt{hi1\_eb3}, \texttt{mgui}),$
$(\texttt{hi2\_eb1}, \texttt{al}), (\texttt{hi2\_eb2}, \texttt{kv}), (\texttt{hi2\_eb3}, \texttt{bv}), (\texttt{hi2\_eb4}, \texttt{fas}), (\texttt{fi1\_eb1}, \texttt{fas}),$
$(\texttt{fi1\_eb2}, \texttt{kv}), (\texttt{fi1\_eb3}, \texttt{bv}), (\texttt{fi1\_ceb1}, \texttt{kv\_fas\_pv}), (\texttt{fi1\_ceb2}, \texttt{bv\_fas\_pv}), ...\}$

Die Auswertung der Eigenschaftsbedingungen erfolgt auf Basis von $\mathfrak{M}^4$. In den logischen Sätzen, durch die die Eigenschaftsbedingungen formuliert sind, werden die Variablen mit Werten aus der Trägermenge $\mathcal{A}_{\mathfrak{M}^4}$ belegt und der Wahrheitswert dieser Sätze bestimmt. Eine Eigenschaftsbedingung ist dann gültig, wenn jede mögliche Belegung als wahr bewertet wird.

Abbildung 8.7: Übersicht über die Gültigkeit der Eigenschaftsbedingungen in $\mathfrak{M}^4$.

Hingegen reicht eine Belegung mit negativer Bewertung aus, damit eine Eigenschaftsbedingung nicht erfüllt wird. Abbildung 8.7 gibt einen Überblick darüber, welche Eigenschaftsbedingungen in $\mathfrak{M}^4$ gültig sind. Während die Bedingungen sowohl für den Fassadebaustein als auch für die Hierarchische Komposition eingehalten werden, werden die beiden des Schichtenbausteins verletzt.

Nicht aufgeführt sind Eigenschaftsbedingungen von Architekturbausteinen, die nicht instanziiert wurden. Diese werden ebenfalls als wahr ausgewertet. Denn wenn ein Baustein bisher nicht instanziiert wurde, ist der Ausdruck $\forall o \forall p$: `ABBOccurrence(o)` $\wedge$ `instantiatedABB(o,p)` $\wedge$ `ABB(p)` $\wedge$ `name(p,"BausteinXYZ")` falsch. Dieser Ausdruck ist Teil der linken Seite der Implikation, aus der die Bedingung besteht. Unabhängig von der rechten Seite der Implikation ist damit die Eigenschaftsbedingung immer wahr.

Die erste Eigenschaftsbedingung $\mathbf{ABBC}_{Schichten1}$ des Schichtenbausteins überprüft $\mathfrak{M}^4$ auf die Existenz von zyklischen Abhängigkeiten zwischen den Schichten einer Bausteininstanz. Sie schlägt fehl für zwei verschiedene Belegungen ihrer Variablen:

	Bausteininstanz `o`	„obere Schicht“ `a`	„untere Schicht“ `b`	...
Belegung 1	`si1`	`gui`	`al`	...
Belegung 2	`si2`	`al`	`data`	...

Wie der vorherigen Übersicht zu entnehmen ist, betrifft die Verletzung der Bedingungen beide Instanzen des Schichtenbausteins. Für beide Belegungen schlägt die Bedingung fehl, weil der Ausdruck $\neg$`dependingLayerUsage(b,a)` (AP 2) auf der rechten Seite der Implikation als falsch ausgewertet wird. Gemäß der Definition des Prädikats führen (transitive) Interaktionen mit synchronen Anteilen zwischen `b` und `a` zu diesem Ergebnis. Wesentlich ist dabei die Auswertung der Prädikate `containsSyncInteraction\2` (AP 3) und `syncInteraction\2` (AP 4) im Kontext der beiden oben notierten Belegungen. Betrachtet man die Belegungen der beiden Prädikate im Detail, lassen sich die verantwortlichen synchronen Interaktionen identifizieren. Von den auch in Abbildung 8.6 dargestellten Nachrichtenvorkommen sind `changeClientData2`, `changeData5`, `sendChangedData2` und `sendChangedData4` Teil dieser Belegungen.

Die zweite Eigenschaftsbedingung des Schichtenbausteins überprüft $\mathfrak{M}^4$ auf die Einhaltung der strikten Hierarchie. Diese schlägt fehl für die folgende Belegung:

	Bausteininstanz `o`	„obere Schicht“ `a` in `o`	„untere Schicht“ `x` nicht in `o`	...
Belegung	`si1`	`gui`	`data`	...

Diese Belegung erfüllt zunächst die linke Seite der Implikation von $\mathbf{ABBC}_{Schichten2}$. Damit wird u.a. bestätigt, dass `gui` eine „obere Schicht“ im Kontext der Bausteininstanz `si1` ist. Außerdem wird der Ausdruck `boundABBEntitiesAreConnected(a, x)` mit obiger Belegung als wahr ausgewertet. Für `boundABBEntitiesAreConnected(gui,data)` gibt es zwei gültige Belegungen, die die Konnektoren `sys_guidata_con1` und `sys_guidata_con2` enthalten. Die Ungültigkeit der Eigenschaftsbedingung folgt dann aus der negativen Auswertung des Ausdrucks $\neg$ `isLowerLayerOfLL*(x, a)` (AP 8) auf der rechten Seite der Implikation. Transitiv

betrachtet ist der Part `data` nämlich eine untere Schicht von `gui`. In Bausteininstanz `si2` wird der Part `data` als „untere Schicht“ zu dem Part `al` festgelegt. Der Part `al` ist wiederum durch die Bausteininstanz `si1` die „untere Schicht“ von `gui`.

8.3.2. Entwicklungsschritt 5

Entwicklungsschritt 5 integriert keine weiteren Bausteininstanzen in das Beispielsystem, sondern besteht aus einer Architekturreduktion kombiniert mit einer Architekturerweiterung. Die im vorherigen Entwicklungsschritt eingezogene Verbindung zwischen den Schichten zur Präsentation und Datenhaltung wird wieder zurückgenommen (Architekturreduktion). Die Interaktionen, die über diese Verbindung erfolgen sollten, werden nun entlang der Schichtenhierarchie über bereits bestehende Verbindung gereicht. Diese Verbindungen werden um entsprechende Merkmale ergänzt (Architekturerweiterung).

Aufbau des Beispielsystems

In Abbildung 8.8 ist die Struktur des Beispielsystems nach Entwicklungsschritt 5 zu sehen. In der dargestellten Sicht entspricht die Struktur wieder dem Stand von vor Entwicklungsschritt 4. Die Änderungen finden sich in nicht dargestellten Details. Zuvor waren die Verbindungselemente unidirektional, denn es war nur möglich, in der Schichtenhierarchie von oben nach unten zu interagieren. Nun sollen Interaktionen in beide Richtungen erfolgen, weswegen auch in der Struktur entsprechende Änderungen notwendig sind. Zwar werden keinen neuen (unidirektionale) Verbindungselemente ergänzt, aber die Definition ihrer Endpunkte wird verändert. Wie Abbildung 8.9 zeigt, werden die Ports an den Komponententypen der Parts um Schnittstellen erweitert. Stellvertretend zeigt die Abbildung den Komponententyp Kundenverwaltung. Nach Entwicklungsschritt 4 benötigt dessen Port kv_data_port nur die Schnittstelle adIF (vgl. Abb. 8.9a). Entwicklungsschritt 5 hat die angebotene Schnittstelle daIF ergänzt (vgl. Abb. 8.9b) und damit Interaktionen in beide Richtungen ermöglicht.

Das Beispielsystem nach Entwicklungsschritt 5 wird durch $\mathfrak{M}^5$ formal repräsentiert. Im Folgenden werden die Änderungen der Relationen von $\mathfrak{M}^5$ im Vergleich zu denen von $\mathfrak{M}^4$ notiert.

$$\texttt{System}^{\mathfrak{M}^5} = \texttt{System}^{\mathfrak{M}^4}$$

$$\texttt{Component}^{\mathfrak{M}^5} = \texttt{Component}^{\mathfrak{M}^4}$$

$$\texttt{Part}^{\mathfrak{M}^5} = \texttt{Part}^{\mathfrak{M}^4}$$

$$\texttt{Configuration}^{\mathfrak{M}^5} = \texttt{Configuration}^{\mathfrak{M}^4}$$

$$\texttt{hasConfiguration}^{\mathfrak{M}^5} = \texttt{hasConfiguration}^{\mathfrak{M}^4}$$

$$\texttt{containsPart}^{\mathfrak{M}^5} = \texttt{containsPart}^{\mathfrak{M}^4}$$

$$\texttt{Connector}^{\mathfrak{M}^5} = \texttt{Connector}^{\mathfrak{M}^4} \setminus \{\texttt{sys\_guidata\_con1}, \texttt{sys\_guidata\_con2}\}$$

$$\texttt{containsConnector}^{\mathfrak{M}^5} = \texttt{containsConnector}^{\mathfrak{M}^4} \setminus \{(\texttt{sys\_conf}, \texttt{sys\_guidata\_con1}), (\texttt{sys\_conf}, \texttt{sys\_guidata\_con2})\}$$

$$\texttt{DelegatorLink}^{\mathfrak{M}^5} = \texttt{DelegatorLink}^{\mathfrak{M}^4} \setminus \{\texttt{gui\_dlink3}, \texttt{gui\_dlink4}\}$$

$$\texttt{containsDelegatorLink}^{\mathfrak{M}^5} = \texttt{containsDelegatorLink}^{\mathfrak{M}^4} \setminus \{(\texttt{gui}, \texttt{gui\_dlink3}), (\texttt{gui}, \texttt{gui\_dlink4})\}$$

$$\texttt{PortVariable}^{\mathfrak{M}^5} = \texttt{PortVariable}^{\mathfrak{M}^4} \setminus \{\texttt{gui\_data\_pv1}, \texttt{gui\_data\_pv2}, \texttt{d\_gui\_pv}\}$$

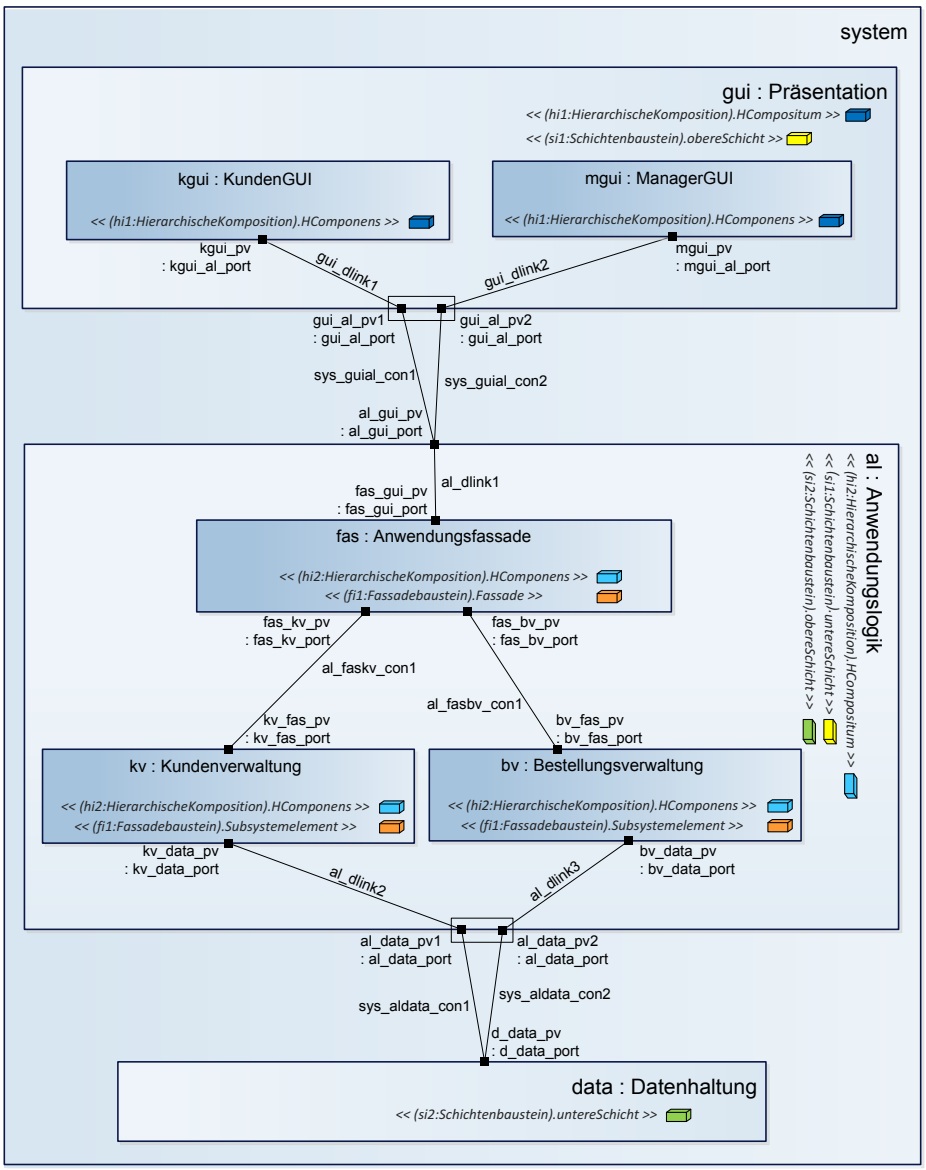


Abbildung 8.8: Systemkonfiguration des Beispielsystems nach Entwicklungsschritt 5.

$$\texttt{hasPortVariable}^{\mathfrak{M}^5} = \texttt{hasPortVariable}^{\mathfrak{M}^4} \setminus \{(\texttt{gui},\texttt{gui\_data\_pv1}), (\texttt{gui},\texttt{gui\_data\_pv2}), (\texttt{data},\texttt{d\_gui\_pv})\}$$

$$\texttt{linkSource}^{\mathfrak{M}^5} = \texttt{linkSource}^{\mathfrak{M}^4} \setminus \{(\texttt{sys\_guidata\_con1},\texttt{gui\_data\_pv1}), (\texttt{sys\_guidata\_con2},\texttt{gui\_data\_pv2}), (\texttt{gui\_dlink3},\texttt{gui\_data\_pv1}), (\texttt{gui\_dlink4},\texttt{gui\_data\_pv2})\}$$

$$\texttt{linkTarget}^{\mathfrak{M}^5} = \texttt{linkTarget}^{\mathfrak{M}^4} \setminus \{(\texttt{sys\_guidata\_con1},\texttt{d\_gui\_pv}), (\texttt{sys\_guidata\_con2},\texttt{d\_gui\_pv}), (\texttt{gui\_dlink3},\texttt{kgui\_data\_pv}), (\texttt{gui\_dlink4},\texttt{mgui\_data\_pv})\}$$

Ein Ausschnitt des Architekturszenarios scn_gesamt nach Entwicklungsschritt 5 ist in Abbildung 8.10 zu sehen. Mit diesem Entwicklungsschritt werden die direkten Interaktionen zwischen

Abbildung 8.9: Unterschied in der Komponentenbeschreibung zwischen Entwicklungsschritt 4 (a) und 5 (b).

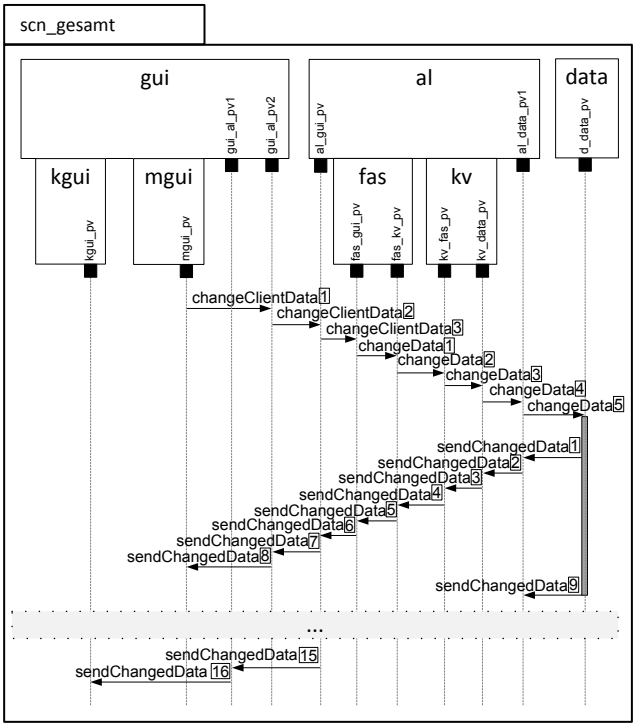


Abbildung 8.10: Szenario scn_gesamt des Beispielsystems nach Entwicklungsschritt 5.

data und gui zur Weiterreichung von geänderten Daten entfernt. Diese Interaktionen werden durch Folgen von Nachrichtenvorkommen ersetzt, die durch die verschiedenen Parts hindurch gereicht werden statt sie zu umgehen. Eine dieser Folgen wird durch die Nachrichtenvorkommen sendChangedData1 bis sendChangedData8 modelliert.

Die Relationen, die die verhaltensbasierten Informationen von $\mathfrak{M}^5$ repräsentieren, werden im Vergleich zu $\mathfrak{M}^4$ teilweise verändert:

$$\texttt{Scenario}^{\mathfrak{M}^5} = \texttt{Scenario}^{\mathfrak{M}^4}$$

$$\begin{aligned}\texttt{MsgOccurrence}^{\mathfrak{M}^5} = \texttt{MsgOccurrence}^{\mathfrak{M}^4} \setminus \{&\texttt{sendChangedData1}, \texttt{sendChangedData2},\\ &\texttt{sendChangedData3}, \texttt{sendChangedData4}, \texttt{sendChangedData5}\} \cup\\ \{&\texttt{sendChangedData1}, \texttt{sendChangedData2}, \texttt{sendChangedData3}, \texttt{sendChangedData4},\\ &\texttt{sendChangedData5}, \texttt{sendChangedData6}, \ldots, \texttt{sendChangedData16}\}\end{aligned}$$

$$\texttt{nextMsgOccurrence}^{\mathfrak{M}^5} = \texttt{nextMsgOccurrence}^{\mathfrak{M}^4} \setminus \{(\texttt{changeData5}, \texttt{sendChangedData1}), (\texttt{sendChangedData1}, \texttt{sendChangedData2}), (\texttt{sendChangedData2}, \texttt{sendChangedData3}), ...\} \cup \{(\texttt{changeData5}, \texttt{sendChangedData1}), (\texttt{sendChangedData1}, \texttt{sendChangedData2}), (\texttt{sendChangedData2}, \texttt{sendChangedData3}), ...\}$$

$$\texttt{triggersMsgOccurence}^{\mathfrak{M}^5} = \texttt{triggersMsgOccurence}^{\mathfrak{M}^4} \setminus \{(\texttt{changeData5}, \texttt{sendChangedData1}), (\texttt{sendChangedData1}, \texttt{sendChangedData2}), (\texttt{sendChangedData1}, \texttt{sendChangedData4}), ...\} \cup \{(\texttt{changeData5}, \texttt{sendChangedData1}), (\texttt{sendChangedData1}, \texttt{sendChangedData2}), (\texttt{changeData5}, \texttt{sendChangedData9}), ...\}$$

$$\texttt{msgType}^{\mathfrak{M}^5} = \texttt{msgType}^{\mathfrak{M}^4} \setminus \{(\texttt{sendChangedData2}, \texttt{"synchronous"}), (\texttt{sendChangedData4}, \texttt{"synchronous"}), ...\} \cup \{(\texttt{sendChangedData1}, \texttt{"synchronous"}), (\texttt{sendChangedData7}, \texttt{"synchronous"}), ...\}$$

$$\texttt{occurredMsg}^{\mathfrak{M}^5} = \texttt{occurredMsg}^{\mathfrak{M}^4} \setminus \{(\texttt{sendChangedData1}, \texttt{sendChangedData\_msg1}), (\texttt{sendChangedData2}, \texttt{sendChangedData\_msg2}), ...\} \cup \{(\texttt{sendChangedData1}, \texttt{sendChangedData\_msg1}), (\texttt{sendChangedData2}, \texttt{sendChangedData\_msg2}), \ldots, (\texttt{sendChangedData16}, \texttt{sendChangedData\_msg16}), ...\}$$

$$\texttt{msgSender}^{\mathfrak{M}^5} = \texttt{msgSender}^{\mathfrak{M}^4} \setminus \{(\texttt{sendChangedData\_msg1}, \texttt{d\_data\_pv}), (\texttt{sendChangedData\_msg2}, \texttt{d\_gui\_pv}), ...\} \cup \{(\texttt{sendChangedData\_msg1}, \texttt{d\_data\_pv}), (\texttt{sendChangedData\_msg2}, \texttt{al\_data\_pv}), ...\}$$

$$\texttt{msgReceiver}^{\mathfrak{M}^5} = \texttt{msgReceiver}^{\mathfrak{M}^4} \setminus \{(\texttt{sendChangedData\_msg1}, \texttt{d\_gui\_pv}), (\texttt{sendchangedData\_msg2}, \texttt{gui\_al\_pv2}), ...\} \cup \{(\texttt{sendChangedData\_msg1}, \texttt{al\_data\_pv}), (\texttt{sendChangedData\_msg2}, \texttt{kv\_data\_pv}), ...\}$$

Instanzen der Architekturbausteine und Überprüfung ihrer Eigenschaftsbedingungen

Durch Entwicklungsschritt 5 kommen keine neuen Bausteininstanzen im Beispielsystem hinzu. Allerdings werden durch die Änderungen in Struktur und Verhalten einige Bausteininstanzen verändert. Davon betroffen sind Bindungen von Konnektorrollen, Konnektorrollenenden und Nachrichtenrollen. Durch den Wegfall der Delegationskonnektoren gui_dlink3 und gui_dlink4 werden auch deren Bindungen an die Konnektorrolle hcomp_con der Hierarchischen Komposition entfernt. Zusätzlich werden die Portvariablen als Endpunkte dieser Delegationskonnektoren sowie ihre Bindungen entfernt. Dasselbe trifft auf die Interaktionen und deren Bindungen zu, die über die entfernten Verbindungselemente erfolgten. Hinzugekommen sind in der Verhaltensbeschreibung hingegen Nachrichtenvorkommen, die zu Nachrichtenspuren zusammengefasst neu gebunden werden.

Der Integrationsteil von $\mathfrak{M}^5$ wird nach Entwicklungsschritt 5 entsprechend der Änderungen durch die folgenden Relationen repräsentiert:

$$\texttt{ABBOccurrence}^{\mathfrak{M}^5} = \texttt{ABBOccurrence}^{\mathfrak{M}^4}$$

$$\texttt{instantiatedABB}^{\mathfrak{M}^5} = \texttt{instantiatedABB}^{\mathfrak{M}^4}$$

$\mathbf{ABBC}_{Schichten1}$	✗	$\mathbf{ABBC}_{Schichten2}$	✓
$\mathbf{ABBC}_{HierKomposition1}$	✓	$\mathbf{ABBC}_{HierKomposition2}$	✓
$\mathbf{ABBC}_{Fassade1}$	✓		

Abbildung 8.11: Übersicht über die Gültigkeit der Eigenschaftsbedingungen in $\mathfrak{M}^5$.

$$\texttt{traceElement}^{\mathfrak{M}^5} = \texttt{traceElement}^{\mathfrak{M}^4} \setminus \{(\texttt{mt10}, \{\texttt{sendChangedData2}\}), (\texttt{mt11}, \{\texttt{sendChangedData3}\}), ...\} \cup \{(\texttt{mt10}, \{\texttt{sendChangedData2}\}), (\texttt{mt15}, \{\texttt{sendChangedData7}\}), ...\}$$

$$\texttt{bindsABBElement}^{\mathfrak{M}^5} = \texttt{bindsABBElement}^{\mathfrak{M}^4} \setminus \{(\texttt{hi1\_cb3}, \texttt{hcomp\_con}), (\texttt{hi1\_ceb5}, \texttt{hc\_end2}), (\texttt{hi1\_ceb6}, \texttt{hc\_end1}), (\texttt{hi1\_ceb7}, \texttt{hc\_end0}), (\texttt{hi1\_mb3}, \texttt{"incoming"}), (\texttt{hi1\_mb4}, \texttt{"incoming"}), (\texttt{hi1\_mb5}, \texttt{"inDelegate"}), ...\} \cup \{(\texttt{hi2\_mb6}, \texttt{"incoming"}), (\texttt{hi1\_mb3}, \texttt{"incoming"}), (\texttt{hi2\_mb12}, \texttt{"outDelegate"}), ...\}$$

$$\texttt{bindsArchElement}^{\mathfrak{M}^5} = \texttt{bindsArchElement}^{\mathfrak{M}^4} \setminus \{(\texttt{hi1\_cb3}, \texttt{gui\_dlink3}), (\texttt{hi1\_ceb5}, \texttt{kgui\_data\_pv}), (\texttt{hi1\_ceb6}, \texttt{gui\_data\_pv1}), (\texttt{hi1\_ceb7}, \texttt{gui\_data\_pv1}), (\texttt{hi1\_mb3}, \texttt{mt10}), (\texttt{hi1\_mb4}, \texttt{mt11}), (\texttt{hi1\_mb5}, \texttt{mt11}), ...\} \cup \{(\texttt{hi2\_mb6}, \texttt{mt10}), (\texttt{hi1\_mb3}, \texttt{mt15}), (\texttt{hi2\_mb12}, \texttt{mt15}), ...\}$$

In Entwicklungsschritt 5 werden die gleichen Eigenschaftsbedingungen auf $\mathfrak{M}^5$ ausgewertet wie zuvor nach Entwicklungsschritt 4. Wie Abbildung 8.11 zeigt, ist im Unterschied zum vorherigen Schritt nun auch die Eigenschaftsbedingung $\mathbf{ABBC}_{Schichten2}$ gültig. Nur noch die Bedingung $\mathbf{ABBC}_{Schichten1}$ wird nicht erfüllt. Die Belegungen, die zu einer negativen Auswertung von $\mathbf{ABBC}_{Schichten1}$ führen, sind identisch mit denen aus dem vorherigen Schritt (vgl. Abschnitt 8.3.1):

	Bausteininstanz `o`	„obere Schicht“ `a`	„untere Schicht“ `b`	...
Belegung 1	`si1`	`gui`	`al`	...
Belegung 2	`si2`	`al`	`data`	...

Ursachen für dieses Ergebnis sind wie auch nach Entwicklungsschritt 4 synchrone Nachrichtenvorkommen. Allerdings handelt es sich nicht um dieselbe Menge wie zuvor. So werden die Nachrichtenvorkommen zwischen data und gui, die vorher beteiligt waren, im aktuellen Entwicklungsschritt gelöscht. Andererseits werden neue Nachrichtenvorkommen ergänzt, von denen nun einige zu der Verletzung der Bedingung beitragen. Um die betreffenden Nachrichtenvorkommen zu identifizieren, ist die Auswertung des Ausdrucks $\neg\,\texttt{dependingLayerUsage}(\texttt{b}, \texttt{a})$ (AP 2) in den Kontexten der obigen Belegungen zu betrachten. Hierzu werden wiederum die Prädikate `containsSyncInteraction\2` (AP 3) und `syncInteraction\2` (AP 4) mit entsprechenden Parametern ausgewertet. Auf diesem Weg lassen sich neben weiteren die Nachrichtenvorkommen changeClientData2, changeData5, sendChangedData1 und sendChangedData7 als beteiligte ermitteln. In Abbildung 8.10 kann dieses Ergebnis nachvollzogen werden.

8.3.3. Entwicklungsschritt 6

In Entwicklungsschritt 6 wird dreimal der Pull-Observerbaustein instanziiert. Zudem werden die Ergänzungen aus dem vorherigen Entwicklungsschritt durch eine Architekturreduktion wieder zurückgenommen. Im Rahmen der Bausteininstanziierungen werden im System keine neuen Parts erstellt. Allerdings werden u.a. verschiedene Verbindungselemente und Portvariablen auf

der strukturellen Ebene ergänzt. Zudem wird die Möglichkeit zur bidirektionalen Interaktion über die Verbindungselemente aus Entwicklungsschritt 5 wieder revidiert. Dies führt zusätzlich zu dem instanziierten Bausteinverhalten zu weiteren Änderungen am Verhalten des Beispielsystems.

Gemäß Definition 25 aus Abschnitt 7.3.3 handelt es sich bei Entwicklungsschritt 6 nicht um einen einzelnen Entwicklungsschritt. Entsprechend der Definition kann formal immer nur eine Instanz eines Architekturbausteins pro Entwicklungsschritt gebildet werden. Korrekterweise erfolgen somit drei Entwicklungsschritte, in denen jeweils eine Instanz des Pull-Observerbausteins gebildet wird. Um das Beispiel kompakt zu halten, werden diese drei Entwicklungsschritte hier als Einheit betrachtet.

Aufbau des Beispielsystems

Im Vergleich mit dem Ergebnis von Entwicklungsschritt 5 lassen sich in der in Abbildung 8.12 dargestellten Struktur des Beispielsystems verschiedene Änderungen identifizieren. Es werden mehrere Delegationskonnektoren und Konnektoren sowie Portvariablen für deren Endpunkte ergänzt. Zum Beispiel besitzen die Parts kgui und mgui nun drei statt zuvor einer Portvariablen. Hinzugekommen sind in diesen beiden Fällen die Portvariablen kgui_kobs_pv und kgui_bobs_pv bzw. mgui_kobs_pv und mgui_bobs_pv. In diesen Portvariablen enden die hinzugefügten Delgationskonnektoren gui_dlink3, gui_dlink4, gui_dlink5 und gui_dlink6.

Im Zuge dieses Entwicklungsschritts wird die zuvor eingeführte Bidirektionalität der Verbindungselemente wieder rückgängig gemacht. In Abbildung 8.13 kann dies an der Komponente Kundenverwaltung nachvollzogen werden. Im Unterschied zur vorherigen Version in Abbildung 8.9b fehlt dem Port kv_data_port nun die angebotene Schnittstelle daIF und dem Port kv_fas_port die benötigte Schnittstelle kvfasIF. Stattdessen besitzt diese Komponente die beiden neuen Ports kv_dataobs_port sowie kv_kvobs_port. Diese Ports besitzen die Schnittstellen ObserverIF und SubjectIF, die die möglichen Interaktionen im Zuge des Observerbausteins definieren.

Formal wird das Beispielsystem nach Entwicklungsschritt 6 durch $\mathfrak{M}^6$ repräsentiert. Im Vergleich zu $\mathfrak{M}^5$ werden dessen Relationen für die strukturellen Architekturaspekte wie folgt verändert:

$$\texttt{System}^{\mathfrak{M}^6} = \texttt{System}^{\mathfrak{M}^5}$$

$$\texttt{Component}^{\mathfrak{M}^6} = \texttt{Component}^{\mathfrak{M}^5}$$

$$\texttt{Part}^{\mathfrak{M}^6} = \texttt{Part}^{\mathfrak{M}^5}$$

$$\texttt{Configuration}^{\mathfrak{M}^6} = \texttt{Configuration}^{\mathfrak{M}^5}$$

$$\texttt{hasConfiguration}^{\mathfrak{M}^6} = \texttt{hasConfiguration}^{\mathfrak{M}^5}$$

$$\texttt{containsPart}^{\mathfrak{M}^6} = \texttt{containsPart}^{\mathfrak{M}^5}$$

$$\texttt{Connector}^{\mathfrak{M}^6} = \texttt{Connector}^{\mathfrak{M}^5} \cup \{\texttt{sys\_guial\_con3}, \texttt{sys\_guial\_con4}, \texttt{sys\_guial\_con5}, \texttt{sys\_guial\_con6}, \texttt{al\_faskv\_con2}, \texttt{al\_fasbv\_con2}, \texttt{sys\_aldata\_con3}, \texttt{sys\_aldata\_con4}\}$$

$$\texttt{containsConnector}^{\mathfrak{M}^6} = \texttt{containsConnector}^{\mathfrak{M}^5} \cup \{(\texttt{sys\_conf}, \texttt{sys\_guial\_con3}), (\texttt{sys\_conf}, \texttt{sys\_guidata\_con2}), (\texttt{al\_conf}, \texttt{al\_faskv\_con2}), ...\}$$

$$\texttt{DelegatorLink}^{\mathfrak{M}^6} = \texttt{DelegatorLink}^{\mathfrak{M}^5} \cup \{\texttt{gui\_dlink3}, \texttt{gui\_dlink4}, \texttt{gui\_dlink5}, \texttt{gui\_dlink6}, \texttt{al\_dlink4}, \texttt{al\_dlink5}, \texttt{al\_dlink6}, \texttt{al\_dlink7}\}$$

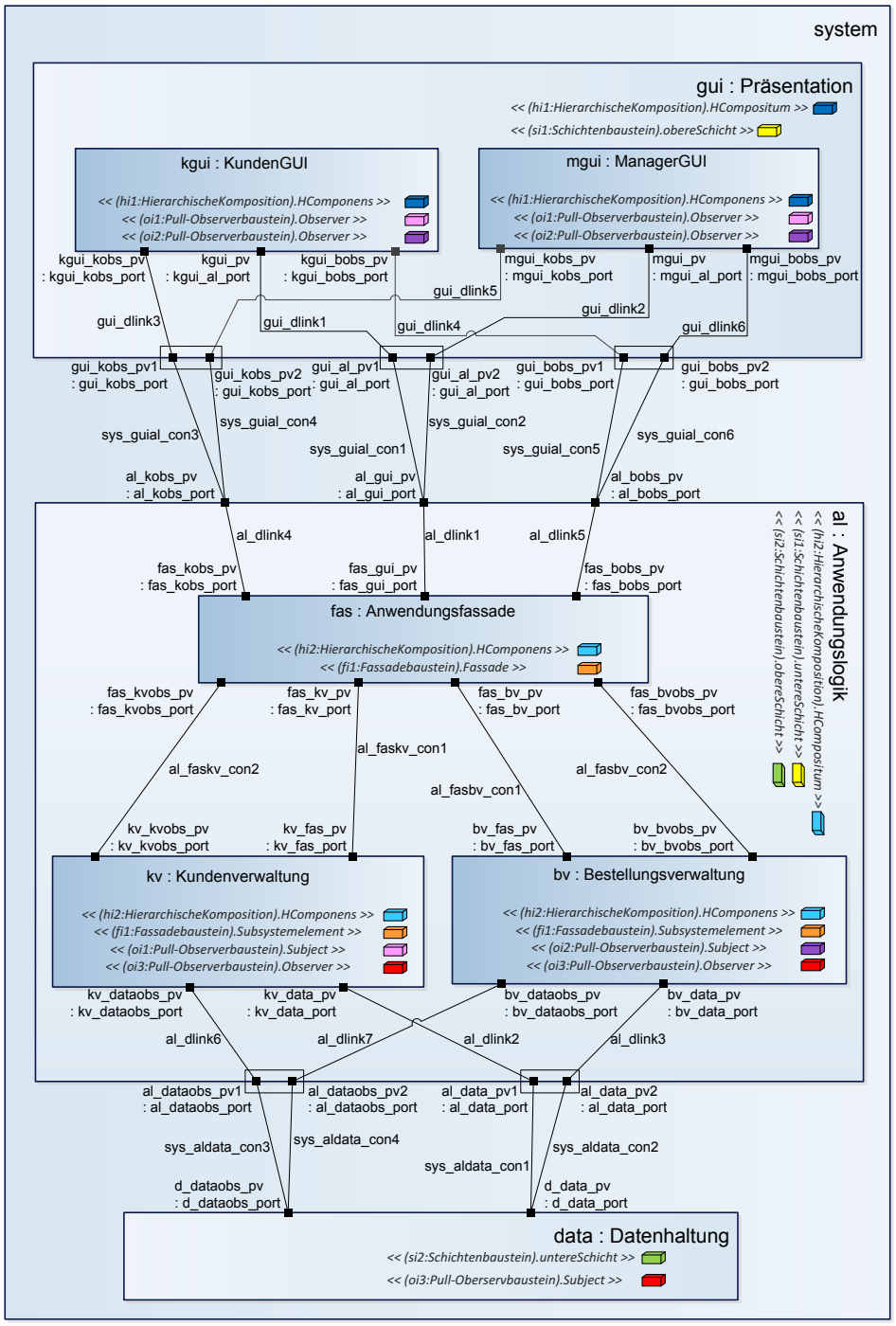


Abbildung 8.12: Systemkonfiguration des Beispielsystems nach Entwicklungsschritt 6 (siehe auch Abb. 6.28 in Abschnitt 6.4.3).

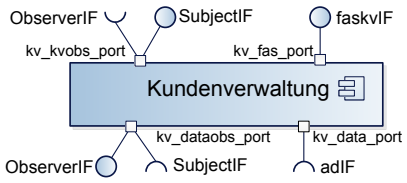


Abbildung 8.13: Komponentenbeschreibung nach Entwicklungsschritt 6 (siehe auch Abb. 6.15b in Abschnitt 6.2.5).

$$\texttt{containsDelegatorLink}^{\mathfrak{M}^6} = \texttt{containsDelegatorLink}^{\mathfrak{M}^5} \cup \{(\texttt{gui},\texttt{gui\_dlink3}), (\texttt{gui},\texttt{gui\_dlink4}), (\texttt{al},\texttt{al\_dlink4}), (\texttt{al},\texttt{al\_dlink5}), ...\}$$

$$\texttt{PortVariable}^{\mathfrak{M}^6} = \texttt{PortVariable}^{\mathfrak{M}^5} \cup \{\texttt{kgui\_kobs\_pv}, \texttt{gui\_kobs\_pv1}, \texttt{al\_kobs\_pv}, \texttt{fas\_kobs\_pv}, \texttt{fas\_kvobs\_pv}, \texttt{kv\_kvobs\_pv}, \texttt{kv\_dataobs\_pv}, \texttt{al\_dataobs\_pv}, \texttt{d\_dataobs\_pv}, ...\}$$

$$\texttt{hasPortVariable}^{\mathfrak{M}^6} = \texttt{hasPortVariable}^{\mathfrak{M}^5} \cup \{(\texttt{kgui},\texttt{kgui\_kobs\_pv}), (\texttt{gui},\texttt{gui\_kobs\_pv1}), (\texttt{al},\texttt{al\_kobs\_pv}), (\texttt{fas},\texttt{fas\_kobs\_pv}), (\texttt{fas},\texttt{fas\_kvobs\_pv}), ...\}$$

$$\texttt{linkSource}^{\mathfrak{M}^6} = \texttt{linkSource}^{\mathfrak{M}^5} \cup \{(\texttt{gui\_dlink3},\texttt{gui\_kobs\_pv1}), (\texttt{sys\_guial\_con3},\texttt{gui\_kobs\_pv1}), (\texttt{al\_dlink4},\texttt{al\_kobs\_pv}), (\texttt{al\_faskv\_con2},\texttt{fas\_kvobs\_pv}), (\texttt{al\_dlink6},\texttt{al\_dataobs\_pv1}), ...\}$$

$$\texttt{linkTarget}^{\mathfrak{M}^6} = \texttt{linkTarget}^{\mathfrak{M}^5} \cup \{(\texttt{gui\_dlink3},\texttt{kgui\_kobs\_pv}), (\texttt{sys\_guial\_con3},\texttt{al\_kobs\_pv}), (\texttt{al\_dlink4},\texttt{fas\_kobs\_pv}), (\texttt{al\_faskv\_con2},\texttt{kv\_kvobs\_pv}), (\texttt{al\_dlink6},\texttt{kv\_dataobs\_pv}), ...\}$$

Abbildung 8.14 zeigt einen Ausschnitt des Architekturszenarios scn_gesamt nach Entwicklungsschritt 6. Die Nachrichtenvorkommen sendChangedData1 bis sendChangedData16 wurden im Vergleich zu dem vorherigen Entwicklungsschritt entfernt. Diese Nachrichtenfolgen dienten der Weitergabe von geänderten Daten und werden durch Verhaltensmuster ersetzt, die dem Verhalten des Observerbausteins entsprechen: Datenänderungen lösen nun Nachrichtenvorkommen mit der Bezeichnung notify aus. Diese stoßen ihrerseits Änderungsbenachrichtigungen durch update-Nachrichtenvorkommen an. Wenn eine Benachrichtigung einen der beiden Parts kgui oder mgui erreicht, fordern diese die geänderten Daten u.a. durch getClientData an. Am Anfang des Architekturszenarios steht außerdem die Registrierung durch register-Nachrichtenvorkommen. Diese werden durch Parts ausgelöst, die in den verschiedenen Bausteininstanzen an die Entitätenrolle Observer gebundenen sind.

Entsprechend der oben skizzierten Änderungen werden auch die Relationen angepasst, die in $\mathfrak{M}^6$ den Verhaltensaspekt der Architekturbeschreibung repräsentieren:

$$\texttt{Scenario}^{\mathfrak{M}^6} = \texttt{Scenario}^{\mathfrak{M}^5}$$

$$\texttt{MsgOccurrence}^{\mathfrak{M}^6} = \texttt{MsgOccurrence}^{\mathfrak{M}^5} \setminus \{\texttt{sendChangedData1}, \texttt{sendChangedData2}, \texttt{sendChangedData3}, \texttt{sendChangedData4}, ..., \texttt{sendChangedData16}\} \cup \{\texttt{register1}, \texttt{register2}, \texttt{notify1}, \texttt{notify2}, \texttt{update1}, \texttt{update2}, \texttt{update3}, \texttt{update8}, \texttt{getClientData1}, \texttt{getClientData2}, \texttt{getData1}, \texttt{getData2}, ...\}$$

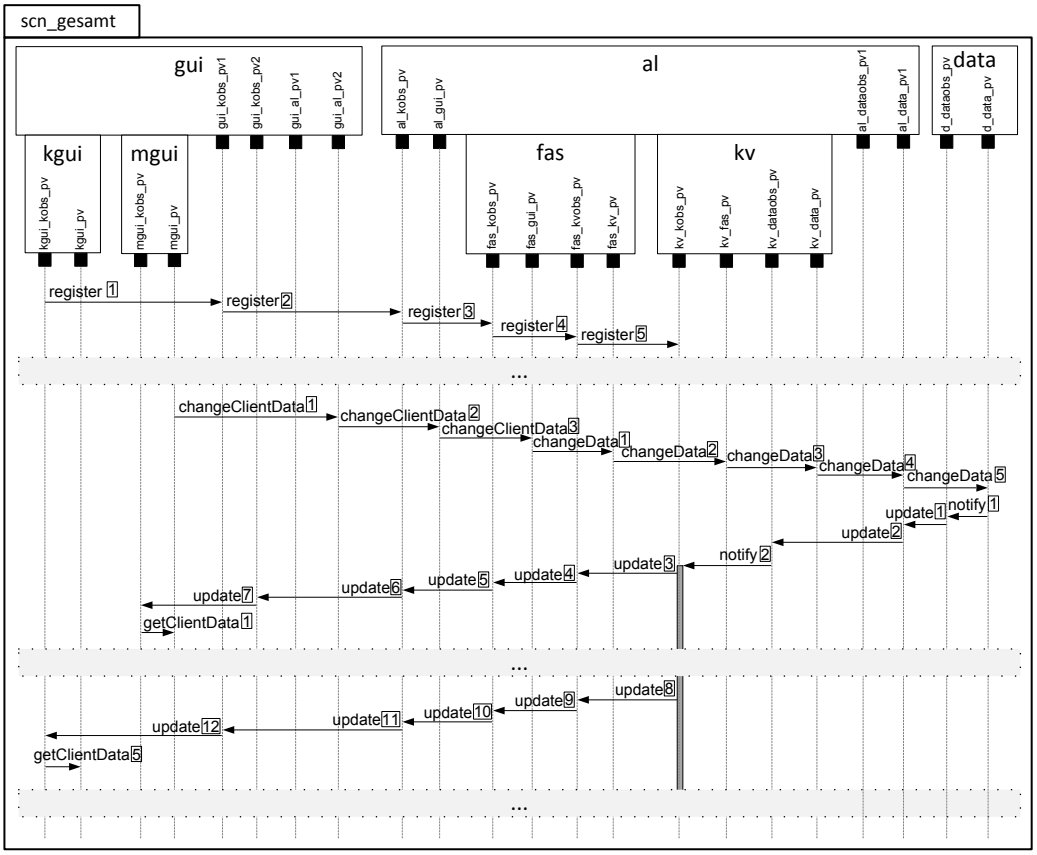


Abbildung 8.14: Szenario scn_gesamt des Beispielsystems nach Entwicklungsschritt 6.

$$\texttt{nextMsgOccurrence}^{\mathfrak{M}^6} = \texttt{nextMsgOccurrence}^{\mathfrak{M}^5} \setminus \{(\texttt{changeData5}, \texttt{sendChangedData1}), (\texttt{sendChangedData1}, \texttt{sendChangedData2}), (\texttt{sendChangedData2}, \texttt{sendChangedData3}), \ldots\} \cup \{(\texttt{changeData5}, \texttt{notify1}), (\texttt{notify1}, \texttt{update1}), (\texttt{update2}, \texttt{notify2}), (\texttt{notify2}, \texttt{update3}), (\texttt{update3}, \texttt{update4}), (\texttt{update7}, \texttt{getClientData1}), \ldots\}$$

$$\texttt{triggersMsgOccurence}^{\mathfrak{M}^6} = \texttt{triggersMsgOccurence}^{\mathfrak{M}^5} \setminus \{(\texttt{changeData5}, \texttt{sendChangedData1}), (\texttt{sendChangedData1}, \texttt{sendChangedData2}), (\texttt{changeData5}, \texttt{sendChangedData9}), (\texttt{sendChangedData9}, \texttt{getClientData1}), \ldots\} \cup \{(\texttt{changeData5}, \texttt{notify1}), (\texttt{notify2}, \texttt{update3}), (\texttt{notify2}, \texttt{update8}), \ldots\}$$

$$\texttt{msgType}^{\mathfrak{M}^6} = \texttt{msgType}^{\mathfrak{M}^5} \setminus \{(\texttt{sendChangedData1}, \texttt{"synchronous"}), (\texttt{sendChangedData7}, \texttt{"synchronous"}), \ldots\} \cup \{(\texttt{register2}, \texttt{"synchronous"}), (\texttt{update1}, \texttt{"synchronous"}), (\texttt{update6}, \texttt{"synchronous"}), (\texttt{update11}, \texttt{"synchronous"}), \ldots\}$$

$$\texttt{occurredMsg}^{\mathfrak{M}^6} = \texttt{occurredMsg}^{\mathfrak{M}^5} \setminus \{(\texttt{sendChangedData1}, \texttt{sendChangedData\_msg1}), \ldots (\texttt{sendChangedData2}, \texttt{sendChangedData\_msg16}), \ldots\} \cup \{(\texttt{register1}, \texttt{register\_msg1}), (\texttt{notify1}, \texttt{notify\_msg1}), (\texttt{notify2}, \texttt{notify\_msg2}), (\texttt{update3}, \texttt{update\_msg3}), (\texttt{update8}, \texttt{update\_msg3}), (\texttt{getClientData1}, \texttt{getClientData\_msg1}), \ldots\}$$

$$\texttt{msgSender}^{\mathfrak{M}^6} = \texttt{msgSender}^{\mathfrak{M}^5} \setminus \{(\texttt{sendChangedData\_msg1}, \texttt{d\_data\_pv}), (\texttt{sendChangedData\_msg2}, \texttt{al\_data\_pv}), ...\} \cup \{(\texttt{register\_msg1}, \texttt{kgui\_kobs\_pv}), (\texttt{notify\_msg1}, \texttt{d\_data\_pv}), (\texttt{notify\_msg2}, \texttt{kv\_dataobs\_pv}), (\texttt{update\_msg3}, \texttt{kv\_kobs\_pv}), (\texttt{getClientData\_msg1}, \texttt{mgui\_kobs\_pv}), ...\}$$

$$\texttt{msgReceiver}^{\mathfrak{M}^6} = \texttt{msgReceiver}^{\mathfrak{M}^5} \setminus \{(\texttt{sendChangedData\_msg1}, \texttt{al\_data\_pv}), (\texttt{sendChangedData\_msg2}, \texttt{kv\_data\_pv}), ...\} \cup \{(\texttt{register\_msg1}, \texttt{gui\_kobs\_pv1}), (\texttt{notify\_msg1}, \texttt{d\_dataobs\_pv}), (\texttt{notify\_msg2}, \texttt{kv\_kobs\_pv}), (\texttt{update\_msg3}, \texttt{fas\_kvobs\_pv}), (\texttt{getClientData\_msg1}, \texttt{mgui\_pv}), ...\}$$

Instanzen der Architekturbausteine und Überprüfung ihrer Eigenschaftsbedingungen

Wie bereits eingangs erwähnt wurde, werden mit Entwicklungsschritt 6 drei Instanzen des Pull-Observerbausteins gebildet. Im Kontext dieser Bausteininstanzen werden verschiedene, neue Bindungselemente erstellt. Wie auch der Abbildung 8.12 zu entnehmen ist, sind die Entitätenrollen des Observerbausteins an verschiedene Parts gebunden. Zudem werden aufgrund der Bausteininstanzen neue Architekturelemente ergänzt und an ein oder mehrere der bindbaren Bausteinelemente gebunden. So sind alle neuen Verbindungselemente Teil von Verbindungspfaden, an die die Konnektorrollen des Observerbausteins gebunden sind. Analog gehören alle neuen Nachrichtenvorkommen zu gebundenen Nachrichtenspuren. In diesem Entwicklungsschritt werden nur wenige Bindungen entfernt. Hierbei handelt es sich nur um Bindungen von entfernten Nachrichtenvorkommen, die an Nachrichtenrollen der Hierarchischen Komposition gebunden waren.

Im Folgenden ist ein Auszug der Relationen notiert, die entsprechend den Integrationsteil von $\mathfrak{M}^6$ repräsentieren:

$$\texttt{ABBOccurrence}^{\mathfrak{M}^6} = \texttt{ABBOccurrence}^{\mathfrak{M}^5} \cup \{\texttt{oi1}, \texttt{oi2}, \texttt{oi3}\}$$

$$\texttt{instantiatedABB}^{\mathfrak{M}^6} = \texttt{instantiatedABB}^{\mathfrak{M}^5} \cup \{(\texttt{oi1}, \texttt{"Pull-Observerbaustein"}), (\texttt{oi2}, \texttt{"Pull-Observerbaustein"}), (\texttt{oi3}, \texttt{"Pull-Observerbaustein"})\}$$

$$\texttt{traceElement}^{\mathfrak{M}^5} = \texttt{traceElement}^{\mathfrak{M}^4} \setminus \{(\texttt{mt10}, \{\texttt{sendChangedData2}\}), (\texttt{mt15}, \{\texttt{sendChangedData7}\}), ...\} \cup \{(\texttt{mt20}, \{\texttt{notify2}\}), (\texttt{mt21}, \{\texttt{update3}, \texttt{update4}, \texttt{update5}, \texttt{update6}, \texttt{update7}\}), ...\}$$

$$\texttt{bindsABBElement}^{\mathfrak{M}^5} = \texttt{bindsABBElement}^{\mathfrak{M}^4} \setminus \{(\texttt{hi2\_mb6}, \texttt{"incoming"}), (\texttt{hi1\_mb3}, \texttt{"incoming"}), (\texttt{hi2\_mb12}, \texttt{"outDelegate"}), ...\} \cup \{(\texttt{oi1\_mb3}, \texttt{"notify"}), (\texttt{oi1\_mb4}, \texttt{"update"}), (\texttt{hi2\_mb6}, \texttt{"outDelegate"}), ...\}$$

$$\texttt{bindsArchElement}^{\mathfrak{M}^5} = \texttt{bindsArchElement}^{\mathfrak{M}^4} \setminus \{(\texttt{hi2\_mb6}, \texttt{mt10}), (\texttt{hi1\_mb3}, \texttt{mt15}), (\texttt{hi2\_mb12}, \texttt{mt15}), ...\} \cup \{(\texttt{oi1\_mb3}, \texttt{mt20}), (\texttt{oi1\_mb4}, \texttt{mt21}), (\texttt{hi2\_mb6}, \texttt{mt20}), ...\}$$

In Entwicklungsschritt 6 wird ein neuer Architekturbaustein instanziiert. Dadurch werden auf dem Gesamtmodell $\mathfrak{M}^6$ zusätzlich auch die Eigenschaftsbedingungen des Pull-Observerbausteins ausgewertet. Eine Übersicht über die Ergebnisse der Auswertung aller Eigenschaftsbedingungen über $\mathfrak{M}^6$ gibt Abbildung 8.15. Wie der Abbildung zu entnehmen ist, werden die neuen Bedingungen eingehalten, während $\mathbf{ABBC}_{Schichten1}$ weiterhin verletzt wird. Auch dieses Mal sind beide Instanzen des Schichtenbausteins von der Verletzung betroffen. Ebenso wie

$\mathbf{ABBC}_{Schichten1}$	✗	$\mathbf{ABBC}_{Schichten2}$	✓
$\mathbf{ABBC}_{HierKomposition1}$	✓	$\mathbf{ABBC}_{HierKomposition2}$	✓
$\mathbf{ABBC}_{Fassade1}$	✓	$\mathbf{ABBC}_{Pull\text{-}Observer1}$	✓
$\mathbf{ABBC}_{Pull\text{-}Observer2}$	✓		

Abbildung 8.15: Übersicht über die Gültigkeit der Eigenschaftsbedingungen in $\mathfrak{M}^6$.

in Entwicklungsschritt 5 werden in Entwicklungsschritt 6 Nachrichtenvorkommen entfernt, die zuvor zu der Verletzung von $\mathbf{ABBC}_{Schichten1}$ beitrugen. Aber auch dieses Mal werden Nachrichtenvorkommen ergänzt, die nun zum Teil ursächlich für die Verletzung sind. Diese Nachrichtenvorkommen lassen sich durch die Auswertung der Prädikate `containsSyncInteraction\2` (AP 3) und `syncInteraction\2` (AP 4) bestimmen, die in der Definition des Prädikats `dependingLayerUsage\2` (AP 2) verwendet werden. Letzteres Prädikat ist wiederum wesentlich für die Definition der Eigenschaftsbedingung. Wie zuvor sind auch in $\mathfrak{M}^6$ die Nachrichtenvorkommen changeClientData2 und changeData5 mit an der Ungültigkeit von $\mathbf{ABBC}_{Schichten1}$ beteiligt. Darüber hinaus können die neuen Nachrichtenvorkommen update1, update6 und update11 als Ursachen ermittelt werden.

8.3.4. Entwicklungsschritt 7

Entwicklungsschritt 7 nimmt nur geringe Veränderungen am Beispielsystem vor. Es wird kein Architekturbaustein instanziiert. Außerdem werden weder strukturelle Elemente noch Elemente zur Verhaltensbeschreibung ergänzt oder gelöscht. Mit diesem Entwicklungsschritt wird lediglich die Art der Interaktion einiger Nachrichtenvorkommen geändert. Da aus Sicht der Operationalisierung eine Änderung durch Löschen und Ergänzen vollzogen wird, handelt es sich bei diesem Entwicklungsschritt daher trotzdem um eine Kombination aus Architekturreduktion und Architekturerweiterung.

Da sich die Änderungen auf das Verhalten beziehen, stimmt die Struktur des Beispielsystems weiterhin mit der nach Entwicklungsschritt 6 (siehe Abb. 8.12) überein. Auch die formale Repräsentation durch $\mathfrak{M}^7$ ist für die strukturelle Architekturbeschreibung identisch mit $\mathfrak{M}^6$. Keine der entsprechenden Relationen wird verändert.

In den vorangegangenen Entwicklungsschritten wurden sämtliche Nachrichtenvorkommen im Architekturszenario scn_gesamt als synchron modelliert. Die Synchronität hat allerdings in jedem vorherigem Entwicklungsschritt zur Verletzung der Eigenschaftsbedingung $\mathbf{ABBC}_{Schichten1}$ geführt. Wie Abbildung 8.16 zeigt, wird die Interaktionsart der Nachrichtenvorkommen notify1, notify2 und update1 bis update12 geändert. Diese sollen nun asynchron erfolgen. Die Relation `msgType` ist daher die einzige in der formalen Repräsentation des Beispielsystems, die durch Entwicklungsschritt 7 verändert wird:

$$\begin{aligned}\texttt{msgType}^{\mathfrak{M}^7} = \texttt{msgType}^{\mathfrak{M}^6} \setminus \{&(\texttt{notify1},\texttt{"synchronous"}), (\texttt{notify2},\texttt{"synchronous"}),\\ &(\texttt{update1},\texttt{"synchronous"}), \ldots, (\texttt{update12},\texttt{"synchronous"}), \ldots\} \cup\\ \{&(\texttt{notify1},\texttt{"asynchronous"}), (\texttt{notify2},\texttt{"asynchronous"}),\\ &(\texttt{update1},\texttt{"asynchronous"}), \ldots, (\texttt{update12},\texttt{"asynchronous"})\}\end{aligned}$$

Dadurch, dass keine bindbaren Architekturelemente gelöscht oder ergänzt werden, erfolgen auch keine Veränderungen am Integrationsteil des Beispielsystems. Für die Relationen, die diesen Bereich repräsentieren, ist $\mathfrak{M}^7$ identisch mit $\mathfrak{M}^6$. Die Veränderungen durch Entwicklungsschritt 7 haben somit nur geringe Auswirkungen auf das Beispielsystem und seine formale

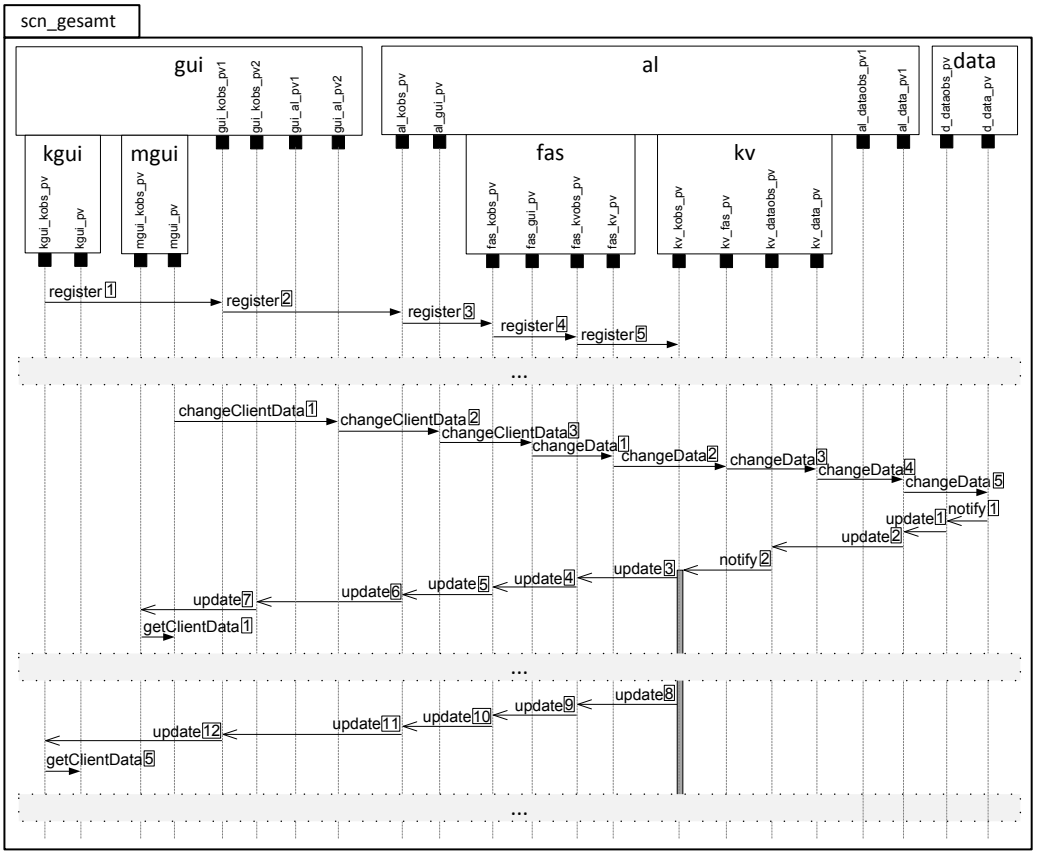


Abbildung 8.16: Szenario scn_gesamt des Beispielsystems nach Entwicklungsschritt 7.

$\mathbf{ABBC}_{Schichten1}$	✔	$\mathbf{ABBC}_{Schichten2}$	✔
$\mathbf{ABBC}_{HierKomposition1}$	✔	$\mathbf{ABBC}_{HierKomposition2}$	✔
$\mathbf{ABBC}_{Fassade1}$	✔	$\mathbf{ABBC}_{Pull\text{-}Observer1}$	✔
$\mathbf{ABBC}_{Pull\text{-}Observer2}$	✔		

Abbildung 8.17: Übersicht über die Gültigkeit der Eigenschaftsbedingungen in $\mathfrak{M}^7$.

Repräsentation. Indessen führt dieser Entwicklungsschritt dazu, dass die Überprüfung aller Eigenschaftsbedingungen erfolgreich ist (vgl. Abb. 8.17). Nachrichtenvorkommen, die in der Schichtenhierarchie von unten nach oben erfolgen, werden nun als asynchron modelliert. Dadurch bestehen keine zyklischen Abhängigkeiten mehr zwischen den Schichten, die zuvor zu ungültigen Eigenschaftsbedingungen führten.

8.4. Betrachtung der Ergebnisse der Fallstudie

In den vorangegangenen Abschnitten wurde die Anwendung des bausteinbasierten Architekturentwurfs dargestellt. Dazu wurden zunächst Struktur- und Verhaltensvorgaben verschiedener Architekturbausteine beschrieben. Anschließend wurde das Beispielsystem in mehreren Entwicklungsschritten entworfen, wobei nach jedem Schritt die Gültigkeit der Eigenschaftsbedingungen der Architekturbausteine überprüft wurde. Im Folgenden wird rückblickend betrachtet,

in wieweit die in Abschnitt 8.1 aufgeführten Ziele erreicht wurden:

- *Beschreibung der Architekturbausteine*

  Im Rahmen der Fallstudie wurden vier verschiedene Architekturbausteine beschrieben und formalisiert. Durch ihre Unterschiede stellt jeder dieser Architekturbausteine andere Anforderungen an die Beschreibungssprache. Während der Schichtenbaustein keine Verhaltensbeschreibung besitzt, ist der Observerbaustein stark durch sein Verhalten geprägt. Trotz ihrer Unterschiede können sowohl Struktur als auch Verhalten aller Architekturbausteine ohne Einschränkungen beschrieben und formalisiert werden.

- *Identifizierung von angewendeten Architekturbausteinen und ihrer Bestandteile*

  Die vier Architekturbausteine wurden im Beispielsystem zusammen achtmal instanziiert. Jede dieser Bausteininstanzen kann jederzeit eindeutig im Architekturmodell identifiziert werden. Dies trifft ebenfalls auf die gebundenen Elemente des Architekturbausteins zu. Die Bindungen der Bausteinelemente lassen sich zudem eindeutig einer Bausteininstanz zuordnen.

- *Formulierung von Eigenschaftsbedingungen der Architekturbausteine*

  Für die vier Architekturbausteine wurden insgesamt sieben Eigenschaftsbedingungen durch prädikatenlogische Sätze formuliert. Diese machen Aussagen sowohl über die Struktur- als auch über die Verhaltensbeschreibung des Architekturmodells. Darüber hinaus beziehen sie sich nicht nur auf eine einzelne Instanz des sie definierenden Architekturbausteins. Einige berücksichtigen weitere Instanzen oder Architekturelemente, die nicht einer bestimmten Bausteininstanz zugeordnet sein müssen. Die Kombination aus der zugrunde liegenden Beschreibungssprache und der Prädikatenlogik ermöglicht demzufolge die Formulierung unterschiedlicher Eigenschaftsbedingungen.

- *Überprüfung der Eigenschaftsbedingungen von angewendeten Architekturbausteinen*

  Die Überprüfung der Eigenschaftsbedingungen ist Teil jedes Entwicklungsschritts. Für die Auswertung wird sowohl auf die in die Architektur integrierten Bausteininstanzen als auch auf die Architekturelemente, die in dem Kontext der jeweiligen Instanz an Bausteinelemente gebunden sind, zurückgegriffen. Dadurch liefert die Auswertung der Bedingungen neben einem Wahrheitswert auch die Architekturelemente, die im Fall einer Verletzung für ebendiese verantwortlich sind.

Die Fallstudie wurde anhand des in dieser Arbeit entwickelten Ansatzes durchgeführt. Wie die obige Aufstellung zeigt, wurden alle aus den Forschungsfragen abgeleiteten Punkte in der Fallstudie konkretisiert. Dies stützt die Tatsache, dass der erarbeitete Ansatz Lösungen für alle gestellten Forschungsfragen enthält. Ergänzend zeigt das folgende Kapitel, dass sich der Ansatz auch im Rahmen einer Werkzeugunterstützung implementieren lässt. Eine Diskussion aller Ergebnisse dieser Arbeit erfolgt am Schluss in Abschnitt 10.2.

9. Entwurf einer Werkzeugunterstützung

Inhalt

Sowohl die Modellierung einer Softwarearchitektur als auch die Überprüfung der verschiedenen Bedingungen, die an ein Modell gestellt werden, sind komplexe Aufgaben. Der Einsatz eines geeigneten Werkzeugs unterstützt diese Aufgaben. In diesem Kapitel wird daher ein solches Werkzeug entworfen, das den bausteinbasierten Architekturentwurf begleiten soll. In Abschnitt 9.1 wird das Oberflächenkonzept für das Werkzeug vorgestellt. Abschnitt 9.2 beschreibt in einer Komponenten- und einer Laufzeitsicht die Architektur dieses Werkzeugs. Als Mechanismus, der die Überprüfung der Bedingungen durchführt, wird ein Wissenrepräsentations- und Anfragesystem vorgeschlagen. In Abschnitt 9.3 wird mit PowerLoom ein solches vorgestellt.

9.1. Nutzersicht auf das Werkzeug

Die Modellierung von Architekturen und Architekturbausteinen ist eine Aufgabe, die besonders durch eine geeignete grafische Oberfläche unterstützt werden kann. Auch die Auswertung der Konformitätsprüfung profitiert von einer guten Darstellung der Ergebnisse. Im Folgenden werden entsprechende Oberflächenkonzepte für das Werkzeug zur Unterstützung des bausteinbasierten Architekturentwurfs vorgestellt.

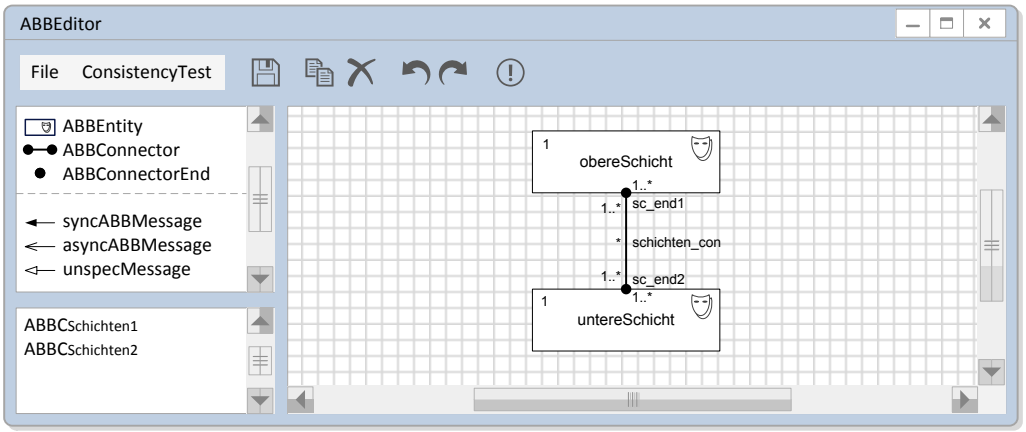


Abbildung 9.1: Oberflächenkonzept des Bausteineditors.

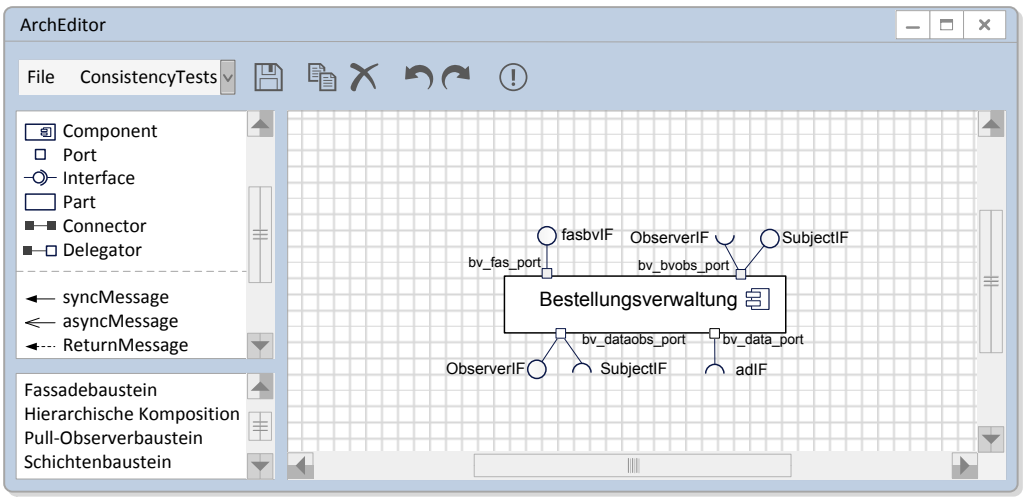


Abbildung 9.2: Oberflächenkonzept des Architekturmodelleditors.

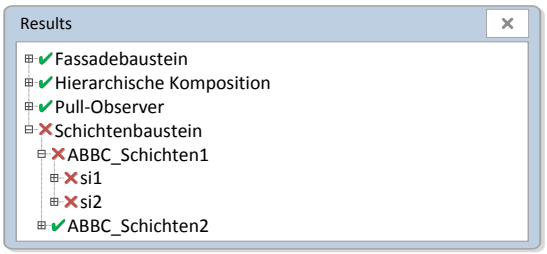


Abbildung 9.3: Oberflächenkonzept der Ergebnisanzeige der Konformitätsprüfung.

Abbildung 9.1 zeigt das Oberflächenkonzept für den Bausteineditor. Zentrales Element ist die Zeichenfläche des Editors, in dem die Modellelemente arrangiert werden. Die möglichen Modellelemente werden im linken oberen Fensterbereich aufgelistet und können per Drag&Drop auf die Zeichenfläche gezogen werden. Eigenschaftsbedingungen, die für den aktuell modellierten Architekturbaustein angelegt wurden, werden im linken unteren Fensterbereich aufgelistet. Diese werden in einem eigenen Dialogfenster entsprechend des Schemas aus Abschnitt 6.5.2 formuliert. Der Zugriff auf verschiedene Funktionen, wie das Abspeichern von Bausteinen oder das Aufrufen der Konformitätsprüfung, erfolgt über die Menuleiste.

Ähnlich zum Bausteineditor ist die Oberfläche des Architektureditors in Abbildung 9.2 aufgebaut. Auch hier ist die Zeichenfläche das zentrale Element, neben der im linken oberen Bereich die verfügbaren Modellelemente aufgelistet sind. Zusätzlich befindet sich unterhalb der Modellelemente ein Auswahlbereich zur Anwendung der verschiedenen im Werkzeug hinterlegten Architekturbausteine.

In Abbildung 9.3 ist das Oberflächenkonzept für die Ergebnisausgabe der Konformitätsprüfung dargestellt. In dem gezeigten Fall werden die Ergebnisse abgebildet, die die Überprüfung der Eigenschaftsbedingungen der Architekturbausteine aus dem Beispielsystem nach Entwicklungsschritt 5 liefert. Die Ereignisanzeige erfolgt baumartig, mit den jeweils überprüften Architekturbausteinen auf oberster Ebene. In der nächsten Ebene werden die verschiedenen Eigenschaftsbedingungen der Bausteine aufgelistet und darunter die Bausteininstanzen, in denen diese Bedingungen geprüft wurden. Bei Bedingungsprüfungen mit negativem Ergebnis werden in den weiteren Ebenen Details zu den Verletzungen angegeben, indem die betreffenden Elemen-

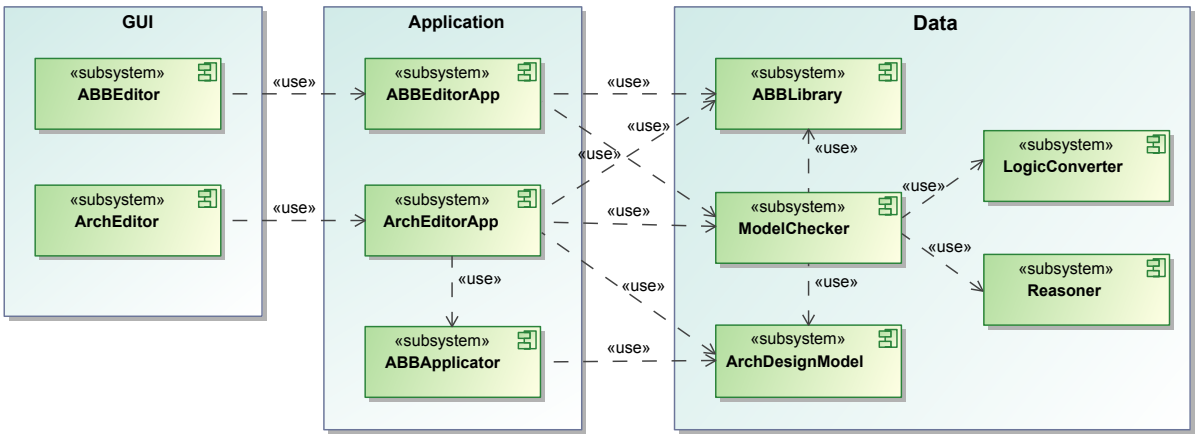


Abbildung 9.4: Überblick über den strukturellen Aufbau des Werkzeugs in der Komponentensicht.

te aufgelistet werden. Von diesen Einträgen in der Baumansicht kann direkt zu den Elementen im Editor navigiert werden. Nach einer durchgeführten Prüfung erhalten auch die Elemente im Editor eine Markierung, wenn sie an der Verletzung einer Bedingung beteiligt sind.

9.2. Architektur einer Werkzeugunterstützung

Die Architektur eines Softwaresystems kann durch Sichten aus verschiedenen Perspektiven beschrieben werden. Dabei liegt der Fokus jeweils auf einem ausgewählten Aspekt der Beschreibung und abstrahiert von für die Sicht nicht relevanten Details. Für die Dokumentation einer Softwarearchitektur wird i.d.R. eine Kombination sich gegenseitig ergänzender Sichten verwendet. Unterschiedliche Autoren haben diesbezüglich verschiedene Standards definiert. Die eingesetzten Sichten variieren dabei ebenso wie deren Bezeichnung oder die Details, die beschrieben werden sollen [Kru95, HNS00, CBB+10, arc42]. Gemeinsam ist diesen und anderen Arbeiten, dass sie jeweils eine Sicht mit strukturellen, mit Verhaltens- und mit Verteilungsaspekten festlegen [RH06]. In Anlehnung daran wird im Folgenden die Architektur einer Werkzeugunterstützung durch eine statische Komponentensicht (Abschnitt 9.2.1) und durch eine verhaltensbasierte Laufzeitsicht (Abschnitt 9.2.2) beschrieben.

9.2.1. Komponentensicht

Die Komponentensicht zeigt eine statische Strukturansicht auf die Architektur des Werkzeugs. Logisch ist die Architektur in die drei Schichten GUI, Application und Data unterteilt. Die Komponenten zur Realisierung des Werkzeugs sind jeweils einer dieser Schichten zugeordnet. Abbildung 9.4 zeigt in einer Komponentensicht die Komponenten inklusive der Zuordnung zu den Schichten und die Beziehungen zwischen den Komponenten.

Die Schicht GUI

Die Schicht GUI repräsentiert die Benutzeroberfläche des Werkzeugs und enthält die beiden Komponenten ABBEditor und ArchEditor.

ABBEditor. Diese Komponente realisiert einen grafischen Editor, um Architekturbausteine mit Hilfe von Diagrammen zu modellieren. Einem Benutzer werden, wie in einem CASE-

Tool üblich, die verschiedenen im Metamodell definierten Elemente zur Modellierung eines Architekturbausteins zur Verfügung gestellt. Des Weiteren kann der Benutzer für den Baustein Eigenschaftsbedingungen formulieren. Der Benutzer kann sich zudem alle im Werkzeug hinterlegten Architekturbausteine anzeigen lassen, davon einzelne löschen oder bearbeiten und einen neu erstellten oder bearbeiteten abspeichern. Außerdem kann vom Benutzer eine Prüfung auf Konformität hinsichtlich der Metamodellbedingungen angestoßen werden. Die Komponente ABBEditor realisiert nur die Oberfläche, für die Ausführung der Anwendungslogik nutzt sie die Komponente ABBEditorApp in der Schicht Application.

ArchEditor. Diese Komponente realisiert einen grafischen Editor, um Softwarearchitekturen mit Hilfe von Diagrammen zu modellieren. Ein Benutzer kann dabei sowohl auf die im Architekturmetamodell definierten Modellierungselemente zurückgreifen, als auch Architekturbausteine aus der Bausteinbibliothek anwenden. Für das erstellte Architekturmodell kann der Benutzer jederzeit eine Überprüfung sowohl hinsichtlich der Metamodellbedingungen als auch der Eigenschaftsbedingungen der verwendeten Architekturbausteine initiieren. Für die Ausführung der Anwendungslogik greift diese Komponente auf die ArchEditorApp in der Schicht Application zurück.

Die Schicht Application

Die Schicht Application enthält mit den Komponenten ABBEditorApp und ArchEditorApp die Anwendungslogik der beiden Editoren. Außerdem umfasst sie mit der Komponente ABBApplicator die Funktionalität zur Instanziierung eines Architekturbausteins.

ABBEditorApp. Die Aufgabe dieser Komponente ist es, die Anwendungslogik für den ABBEditor zu realisieren. Sie setzt die Funktionalität um, die notwendig ist, um einen neuen Architekturbaustein zu bearbeiten. Für die Änderungsoperationen von Architekturbausteinen nutzt ABBEditorApp wiederum die Komponente ABBLibrary der Schicht Data. Die Prüfung eines Architekturbaustein hinsichtlich seiner Konformität zu den Metamodellbedingungen reicht diese Komponente an ModelChecker in der Schicht Data weiter.

ArchEditorApp. Diese Komponente realisiert die Anwendungslogik hinter dem ArchEditor. Sie bietet dem ArchEditor Funktionen, um ein Architekturmodell zu bearbeiten. Dazu greift sie auf die Komponente ArchDesignModel der Schicht Data zurück. Die Anwendung von Architekturbausteinen reicht sie unter Weitergabe des betreffenden Bausteins an die Komponente ABBApplicator weiter. Welche Architekturbausteine zur Anwendung verfügbar sind, wird in der durch ABBLibrary repräsentierten Bausteindatenbank abgefragt. Für die Überprüfung eines Architekturmodells sowohl hinsichtlich der Metamodellbedingungen als auch der Eigenschaftsbedingungen der verwendeten Architekturbausteine nutzt sie die Komponente ModelChecker.

ABBApplicator. Diese Komponente setzt die Instanziierung von Architekturbausteinen um, indem sie dafür sorgt, dass die notwendigen Elemente des Integrationsteils angelegt werden. Sie wird dazu von der ArchEditorApp aufgerufen und erhält von dieser die notwendigen Informationen wie den zu instanziierenden Baustein und die Bindungspaare. Ebenfalls realisiert sie die Entfernung von Bausteininstanzen. Veränderungen im Integrationsteil des Architekturmodells reicht sie zur Persistierung an ArchDesignModel in der Schicht Data weiter.

Die Schicht Data

Die Schicht Data verwaltet in den Komponenten ABBLibrary und ArchDesignModel die Datenmodelle für Architekturbausteine bzw. Architekturmodelle und bietet Zugriff auf persistierte Modelle. Zudem beherbergt sie mit ModelChecker die Komponente zur Steuerung der Konformitätsprüfung der Modelle. Die Konformitätsprüfung wird durch die Komponente Reasoner auf den durch LogicConverter umgewandelten Modellen durchgeführt.

ABBLibrary. Diese Komponente verwaltet die Bausteinbibliothek und enthält das Datenmodell, um Architekturbausteine inklusive ihrer Eigenschaftsbedingungen zu repräsentieren. Im Datenmodell integriert sind dabei auch die Metamodellbedingungen des Bausteinteils. ABBLibrary stellt über eine Schnittstelle Funktionen zur Verfügung, um sowohl einzelne Bausteine als auch die Bibliothek an sich zu bearbeiten. Eine weitere Schnittstelle erlaubt den Zugriff auf alle enthaltenen Bedingungen.

ArchDesignModel. Diese Komponente verwaltet das aktuell zu bearbeitende Architekturmodell inklusive der Informationen zu in diesem Architekturmodell angewendeten Architekturbausteinen. ArchDesignModel enthält das Datenmodell, um sowohl den Architekturteil als auch den Integrationsteil eines Modells zu repräsentieren. Zudem enthält diese Komponente die Bedingungen dieser beiden Metamodellteile. Die Komponente bietet verschiedene Schnittstellen an. Funktionen zur Bearbeitung des Architekturteils und des Integrationsteils werden durch zwei verschiedene Schnittstellen zur Verfügung gestellt. Eine dritte Schnittstelle erlaubt den Zugriff auf die Bedingungen der beiden Metamodellteile.

ModelChecker. Der Komponente ModelChecker fällt die Steuerung der verschiedenen Konformitätsprüfungen zu. Zum Initiieren der Prüfungen stellt sie eine Schnittstelle zur Verfügung. Je nach Art der Prüfung fragt sie die zu überprüfenden Bedingungen an den benachbarten Komponenten ABBLibrary und ArchDesignModel ab. Vor der Prüfung müssen Modelle und Bedingungen in die Sprache des verwendeten Logiksystems übersetzt werden. Hierzu nutzt ModelChecker die Komponente LogicConverter. Für die Prüfung werden die logischen Fakten, in die die Modelle umgewandelt wurden, und die Bedingungen an die Komponente Reasoner weitergereicht.

LogicConverter. Die Aufgabe dieser Komponente ist die Umwandlung von Modellen und Bedingungen in die Sprache des zur Überprüfung verwendeten Logiksystems. Dazu implementiert sie eine Menge von Übersetzungsregeln. Die Kapselung dieser Funktionalität in einer eigenen Komponente erlaubt die einfache Bereitstellung von Übersetzungen für verschiedene Logiksysteme.

Reasoner. Diese Komponente kapselt die Schnittstelle zum Logiksystem, das für die eigentliche Überprüfung der Bedingungen verwendet wird. Die dazu benötigte Faktenbasis und die zu prüfenden Bedingungen werden dem Reasoner über seine Schnittstelle vom Aufrufer zur Verfügung gestellt.

9.2.2. Laufzeitsicht

Die Laufzeitsicht beschreibt die Interaktionen der Komponenten des Werkzeugs zur Laufzeit. Exemplarisch wird in diesem Abschnitt auf die Instanziierung eines Architekturbausteins mit anschließenden Konformitätsprüfungen eingegangen.

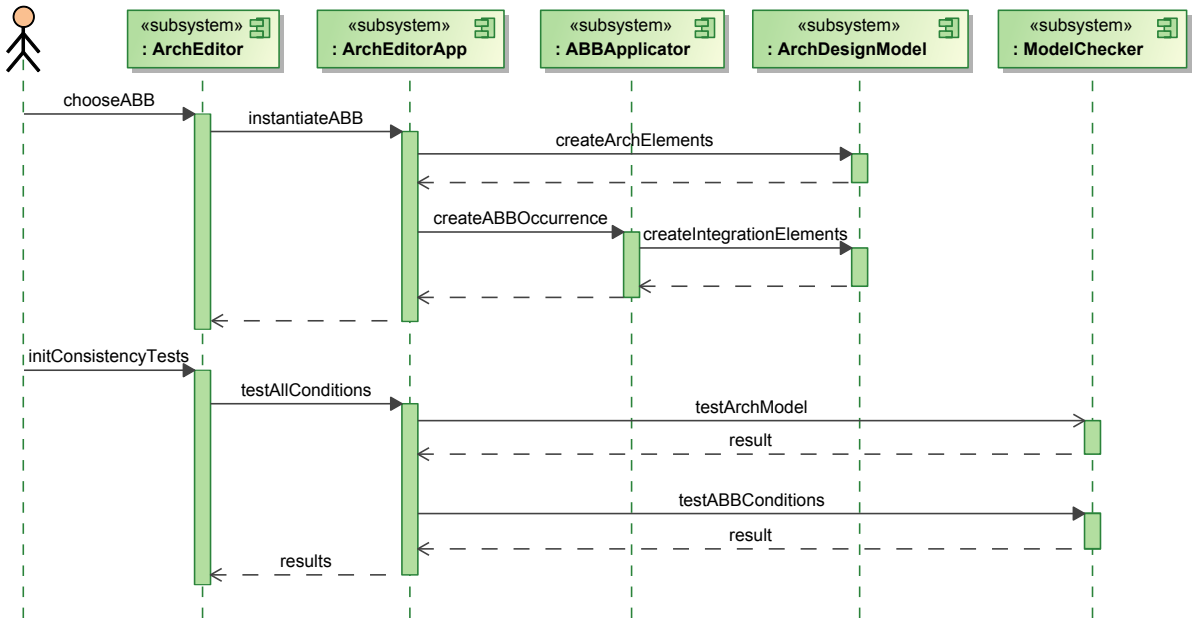


Abbildung 9.5: Ablauf der Instanziierung eines Architekturbausteins mit anschließender Konformitätsprüfung.

Instanziierung eines Architekturbausteins

Abbildung 9.5 skizziert den Ablauf der Instanziierung eines Architekturbausteins mit anschließender Konformitätsprüfung. Initiiert wird der Ablauf durch den Benutzer. Dieser wählt einen Architekturbaustein zur Anwendung aus und bestimmt die Architekturelemente, an die die Rollen des Bausteins gebunden werden sollen (chooseABB). Durch die Betätigung einer Schaltfläche auf der durch ArchEditor zur Verfügung gestellten Nutzeroberfläche wird der Ablauf im Werkzeug gestartet. Daraufhin wird an ArchEditorApp die Methode instantiateABB mit allen notwendigen Daten aufgerufen. Übergebene Daten beinhalten den zu instanziierenden Architekturbaustein sowie die durch den Nutzer festgelegten Bindungspaare. Architekturelemente, die im Zuge der Bausteininstanziierung angelegt werden sollen, werden im Datenmodel des Werkzeugs durch einen Aufruf von createElements an ArchDesignModel mit entsprechenden Parametern erzeugt. Die Erstellung der Bausteininstanz an sich wird durch einen Aufruf an ABBApplicator initiiert (createABBOccurrence). Mit Hilfe der übergebenen Daten bestimmt diese Komponente die Elemente des Integrationsteils, die durch createIntegrationElements im Datenmodell des Architekturmodells erstellt werden.

Im Anschluss initiiert der Benutzer die Überprüfung des Architekturmodells hinsichtlich Konformität zu den Metamodellbedingungen des Architektur- und Integrationsteils sowie zu den Eigenschaftsbedingungen der verwendeten Architekturbausteine (initConsistencyTests). Die Komponente ArchEditor gibt dies durch einen entsprechenden Aufruf an ArchEditorApp weiter (testAllConditions). Nacheinander werden nun die beiden Aufrufe testArchModel und testABBConditions an ModelChecker ausgeführt, um die Metamodellbedingungen bzw. die Eigenschaftsbedingungen der Bausteine zu testen. Die Ergebnisse der beiden Aufrufe werden gebündelt an den Aufrufer ArchEditor von testAllConditions zurückgegeben.

Die Abläufe in der Schicht Data, die durch testArchModel und testABBConditions ausgelöst werden, werden im Folgenden detaillierter betrachtet.

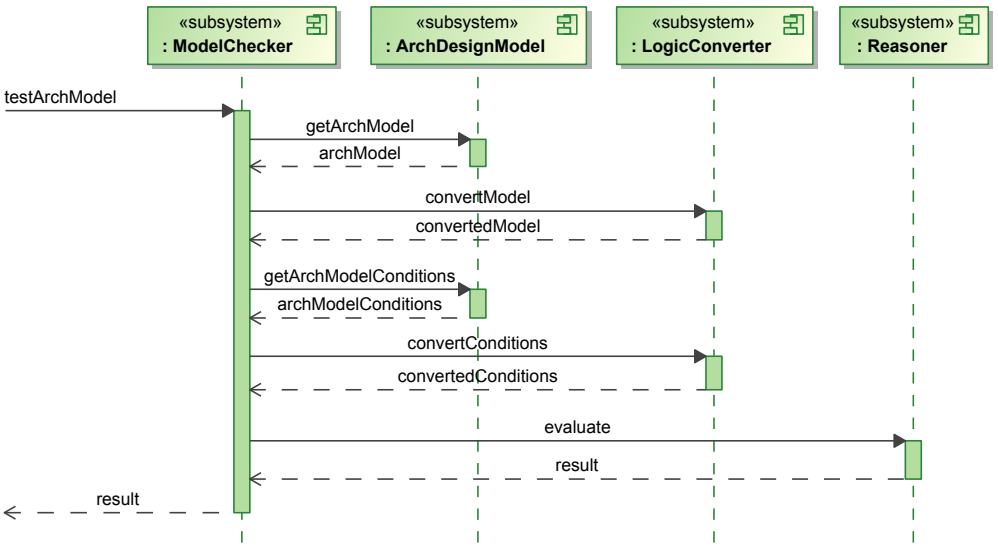


Abbildung 9.6: Ablauf der Prüfung eines Architekturmodells hinsichtlich der Konformität zu den Metamodellbedingungen.

Prüfung der Metamodellbedingungen

Der Ablauf zur Prüfung eines Architekturmodells hinsichtlich der Konformität zu den Metamodellbedingungen wird in Abbildung 9.6 skizziert. Der Aufruf von testArchModel an ModelChecker löst verschiedene aufeinanderfolgende Aktionen aus. Zunächst wird das Architekturmodel abgerufen (getArchModell an ArchDesignModel) und in logische Fakten übersetzt (convertModel an LogicConverter). Ebenso werden die Metamodellbedingungen ausgelesen (getArchModelConditions) und in die Syntax des Logiksystems übersetzt (convertConditions). Mit dem Aufruf von evaluate werden das übersetzte Modell und die übersetzten Bedingungen an den Reasoner weitergereicht. Das von der Komponente Reasoner ermittelte Ergebnis wird schließlich vom ModelChecker an den ursprünglichen Aufrufer von testArchModel zurückgereicht.

Prüfung der Eigenschaftsbedingungen

In Abbildung 9.7 ist der Ablauf zur Überprüfung eines Architekturmodells hinsichtlich der Eigenschaftsbedingungen skizziert. Dieser ist dem Ablauf zur Überprüfung der Metamodellbedingungen sehr ähnlich. Beide Abläufe unterschieden sich lediglich in den Bedingungen, die überprüft werden. Statt die Metamodellbedingungen auszulesen, ruft dieser Ablauf durch getABBConditions die Eigenschaftsbedingungen der Bausteine von ABBLibrary ab. Die Übersetzung von Modell und Bedingungen ebenso wie die Überprüfung erfolgt analog zum vorherigen Ablauf zur Überprüfung der Metamodellbedingungen.

9.3. Wissensrepräsentations- und Anfragesystem

Das Logiksystem hinter dem Werkzeug zum bausteinbasierten Entwurf dient der Überprüfung der im Werkzeug erstellten Modelle. Welches Logiksystem dabei zum Einsatz kommt, ist nicht festgelegt und kann durch die modulare Architektur des Werkzeugs mit wenig Aufwand ausgetauscht werden. An dieser Stelle soll das Wissensrepräsentations- und Logiksystem PowerLoom

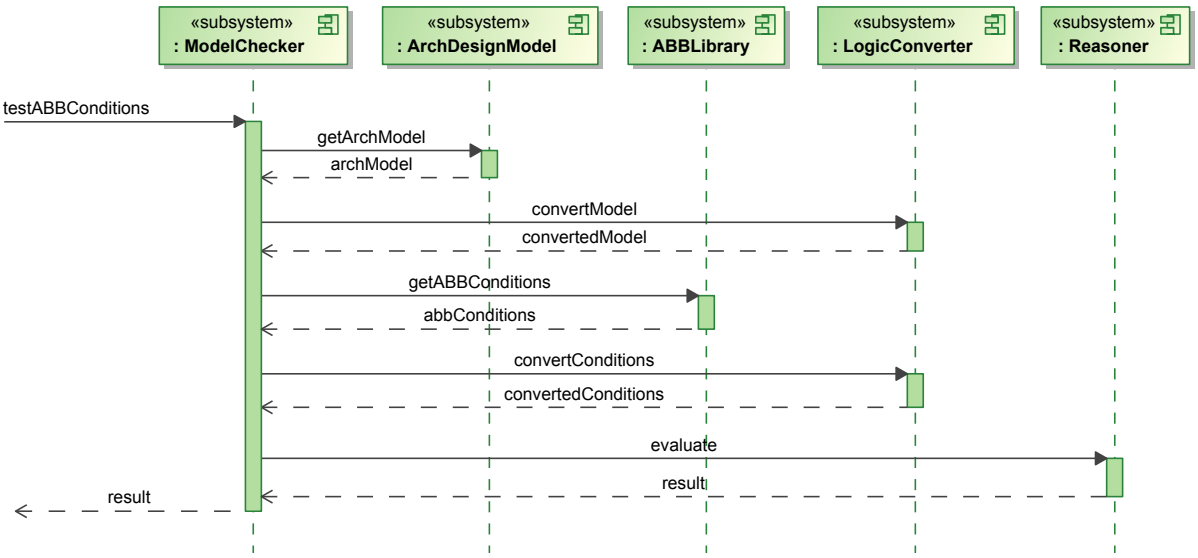


Abbildung 9.7: Ablauf der Prüfung eines Architekturmodells hinsichtlich der Konformität zu den Eigenschaftsbedingungen der verwendeten Architekturbausteine.

[PLm] vorgestellt werden. PowerLoom wurde im Rahmen dieser Arbeit bereits in frühen Phasen verwendet, um die entwickelten Konzepte zu evaluieren. Neben einer Kommandozeileneingabe bietet PowerLoom auch APIs für Lisp, C++ und Java an und bringt damit gute Voraussetzungen für die Integration in ein Werkzeug mit.

In PowerLoom wird Wissen durch eine deklarative, logikbasierte Sprache repräsentiert. Die verwendete Sprache basiert auf der Prädikatenlogik erster Stufe und weist darüber hinaus einige Besonderheiten wie z.B. typisierbare Prädikate auf. Als Syntax nutzt diese Sprache eine Präfixnotation, die auf das *Knowledge Interchange Format* (KIF) [GF+92] zurückgeht.

In PowerLoom werden zwei Arten von Prädikaten unterschieden. Sogenannte Konzepte repräsentieren einstellige Prädikate, während der Begriff Relation für mehrstellige Prädikate genutzt wird. Konzepte und Relationen müssen vor ihrer Verwendung durch `defconcept` bzw. `defrelation` deklariert werden. Die Stelligkeit von Relationen wird durch die Anzahl der Parameter festgelegt, wobei ein Parameter durch die Angabe eines zuvor definierten Konzepts typisiert werden kann. Konzepte können als Spezialisierung eines anderen Konzepts deklariert werden. Auf diese Weise lassen sich z.B. auch OO-Typhierarchien aufbauen, ohne entsprechende Bedingungen definieren zu müssen. Das folgende Beispiel zeigt die Deklaration der Konzepte `BindableArchElement`, `Part` und `PortVariable`, wobei die beiden letzteren im Metamodell aus Abschnitt 6.4.1 als Subtypen des ersten definiert wurden. Die Relation `hasPortVariable` repräsentiert die gleichbenannte Assoziation des Metamodells. Ihr Definitionsbereich wird durch die Typisierung der beiden Variablen auf Tupel aus Parts und Portvariablen eingeschränkt.

```
(defconcept BindableArchElement)
(defconcept Part (BindableArchElement))
(defconcept PortVariable (BindableArchElement))
(defrelation hasPortVariable ((?p Part) (?v PortVariable)))
```

Mit Hilfe von logischen Regeln können aus dem in der Faktenbasis gespeicherten Wissen

neue Informationen abgeleitet werden. In Verbindung mit der Definition eines Konzepts oder einer Relation können Regeln zur Abkürzung von Ausdrücken genutzt werden. In der Eigenschaftsbedingung $\mathbf{ABBC}_{Schichten1}$ wird das Prädikat `interaction` verwendet. Dieses gibt an, ob zwischen zwei Parts eine Interaktion erfolgt (siehe auch die Definition des entsprechenden Adapterprädikats AP 5). In PowerLoom wird dafür zunächst eine Relation `interaction` definiert und anschließend die zugehörige Ableitungsregel angegeben:

```
(defrelation interaction ((?b Part) (?a Part)))
(defrule interaction_rule
  (=> (exists (?p ?q)
      (and (PortVariable ?p) (PortVariable ?q)
        (hasPortVariable ?a ?p) (hasPortVariable ?b ?q)
        (exists (?m ?n ?v)
          (and (MessageOccurrence ?m) (Message ?n) (occurredMsg ?m ?n)
            (msgSender ?n ?q) (msgReceiver ?n ?p)))))
  (interaction ?b ?a)))
```

Bausteine und Architekturmodelle werden durch eine Menge von Fakten auf Basis der Konzepte und Relationen repräsentiert. In PowerLoom werden Fakten durch das Schlüsselwort `assert` der Faktenbasis hinzugefügt. Folgendes Beispiel repräsentiert die Tatsache, dass ein Architekturbaustein `schichtenBaustein` mit den beiden Entitätenrollen `obereSchicht` und `untereSchicht` existiert:

```
(assert (ABB schichtenBaustein))
(assert (ABBEntity obereSchicht))
(assert (ABBEntity untereSchicht))
(assert (containsRole schichtenBaustein obereSchicht))
(assert (containsRole schichtenBaustein untereSchicht))
```

Die zu prüfenden Metamodellbedingungen oder Eigenschaftsbedingungen der Bausteine werden auf der Faktenbasis, die das zu überprüfende Modell repräsentiert, ausgewertet. Statt lediglich den Wahrheitswert der Bedingungen zu bestimmen, können durch geeignete Anfragen die Elemente ermittelt werden, die zu einem negativen Ergebnis führen. Eine Anfrage zur Ermittlung möglicher Verletzungen der Eigenschaftsbedingung $\mathbf{ABBC}_{Schichten1}$ wird im Folgenden gestellt. Im dargestellten Fall wird von PowerLoom ein Ergebnis ermittelt. Existiert hingegen kein Ergebnis, ist die Eigenschaftsbedingung gültig.

```
(retrieve all (?o ?a ?b ?p ?m ?n ?r ?s)
  (and (ABBOccurrence ?o) (ABB ?p) (instantiatedABB ?o ?p)
    (name ?p "Schichtenbaustein")
    (Binding ?m) (containsBinding ?o ?m)
    (Binding ?n) (containsBinding ?o ?n)
    (ABBEntity ?r) (name ?r "obere_Schicht") (bindsABBElement ?m ?r)
    (ABBEntity ?s) (name ?s "untere_Schicht") (bindsABBElement ?n ?s)
    (BindableArchElement ?a) (bindsArchElement ?m ?a)
    (BindableArchElement ?b) (bindsArchElement ?n ?b)
    (dependingLayerUsage ?b ?a)))

There is 1 solution:
```

```
  #1: ?o=si2, ?a=al, ?b=data, ?p=schichtenBaustein, ?m=si2_eb1,
      ?n=si2_eb2, ?r=obereSchicht, ?s=untereSchicht
```

Um die Anfrage zu verkürzen, kann eine stellvertretende Relation `schichten1` deklariert werden, der durch eine Regel der oben angefragte Ausdruck zugeordnet wird. Die Deklaration der Relation und damit auch die Ausgabe der Ergebnisse kann so auf relevante Parameter beschränkt werden:

```
(retrieve all (schichten1 ?o ?a ?b))

There is 1 solution:
  #1: ?o=si2, ?a=al, ?b=data
```

10. Fazit

Inhalt

In dieser Arbeit wurden verschiedene Aspekte eines bausteinbasierten Architekturentwurfs betrachtet. Dabei wurden verschiedene Ergebnisse erarbeitet (Abschnitt 10.1), die zusammen den in dieser Arbeit vorgestellten Ansatz formen. In Abschnitt 10.2 werden die Ergebnisse in Bezug zu den gestellten Forschungsfragen diskutiert. Zudem werden die Einschränkungen des Ansatzes betrachtet und ein Ausblick gegeben.

10.1. Beiträge der Arbeit

Das Ergebnis dieser Arbeit ist ein Ansatz zum Architekturentwurf basierend auf als Architekturbausteinen bezeichneten Architekturkonzepten. Bei der Erarbeitung dieses Ansatzes wurden verschiedene Teilergebnisse erreicht, die im Folgenden beschrieben werden:

- *Rollenkonzept für die Beschreibung von Architekturbausteinen*

  Die zentralen Elemente eines Architekturbausteins werden als Rollen verstanden. Auf ihrer Basis beschreibt ein Architekturbaustein die von ihm definierten Richtlinien für eine Softwarearchitektur. Dabei unterscheidet ein Architekturbaustein verschiedene strukturelle als auch verhaltensbasierte Rollen. Im Rahmen einer Bausteininstanz als Anwendung eines Architekturbausteins werden diese Rollen von Architekturelementen übernommen.

- *Beschreibungssprache für Architekturmodelle, Architekturbausteine und die Anwendung von Architekturbausteinen*

  Grundlage des entwickelten Ansatzes bildet eine Beschreibungssprache, die sowohl Architekturmodelle und Architekturbausteine als auch die Anwendung von Architekturbausteinen beschreibt. Das Metamodell dieser Sprache wurde zunächst semiformal durch UML-Klassendiagramme definiert. Es berücksichtigt dabei sowohl strukturelle als auch verhaltensbasierte Aspekte. Logisch wird das Metamodell in drei Teile unterteilt, wobei jeweils ein Teil für die Beschreibung von Architekturmodellen bzw. von Architekturbausteinen zuständig ist. Der Bausteinteil repräsentiert dabei durch entsprechend gekennzeichnete Elemente die Rollen eines Architekturbausteins. Der dritte Teil des Metamodells, der Integrationsteil, dient als Bindeglied zwischen den beiden anderen ansonsten unabhängigen Teilen. Er enthält die Beschreibungselemente für die Beschreibung einer Bausteininstanz. Hierzu zählen neben der Repräsentation einer Bausteininstanz auch die Bindungen, die zwischen Rollen des Bausteins und Architekturelementen im Rahmen der Bausteininstanz erfolgen.

- *Formalisierung der Beschreibung*

  Eine Formalisierung von Architekturmodellen inklusive der Integrationsbeschreibung als

auch von Architekturbausteinen wird durch eine Abbildung auf mathematische Strukturen realisiert. Die Signaturen dieser Strukturen leiten sich aus dem Metamodell ab. Diese Strukturen bieten die Grundlage, um prädikatenlogische Aussagen über den Modellen zu formulieren.

- *Beschreibung und Formalisierung von Bedingungen für die Anwendung von Architekturbausteinen*

  Es werden zwei Arten von Bedingungen für die Anwendung von Architekturbausteinen beschrieben. Unter der ersten Art sind generische Bedingungen zu verstehen, die im Rahmen des Metamodells zur Beschreibung von Bausteinanwendungen definiert sind. Sie gelten für die Anwendung aller Architekturbausteine. Die zweite Art von Bedingungen wird in diesem Ansatz als Eigenschaftsbedingung bezeichnet. Eigenschaftsbedingungen sind im Gegensatz zu den Bedingungen des Metamodells bausteinspezifisch und stehen im Mittelpunkt des bausteinbasierten Architekturentwurfs. Sie dienen der Überprüfung, ob spezifische Eigenschaften eines Architekturbausteins durch die Architektur, auf die er angewendet ist, eingehalten werden. Formalisiert werden beide Arten von Bedingungen durch prädikatenlogische Sätze. Um die formale Formulierung von Eigenschaftsbedingungen zu erleichtern, wurde ein vereinfachtes Schema eingeführt.

- *Überprüfung von Bedingungen für die Anwendung von Architekturbausteinen*

  Die Bedingungen für die Anwendung von Architekturbausteinen werden auf Architekturmodellen überprüft. Architekturmodelle werden als mathematische Strukturen repräsentiert und Bedingungen als prädikatenlogische Sätze beschrieben, so dass für die Überprüfung ein Logiksystem verwendet werden kann. Durch die Umformulierung von Bedingungen als Anfragen können zudem die Elemente ermittelt werden, die die Bedingungen verletzten. Als Logiksystem wurde in dieser Arbeit PowerLoom eingesetzt.

- *Operationalisierung der Anwendung von Architekturbausteinen und der Änderung von Architekturmodellen*

  Der Ansatz bietet sowohl für die Instanziierung von Architekturbausteinen als auch für Änderungen am Architekturmodell eine operationelle Beschreibung. Die hierzu definierten Funktionen basieren dabei auf den mathematischen Strukturen, die Architekturmodelle und Bausteine formal repräsentieren.

- *Formalisierung mehrerer Architekturbausteine und ihrer Eigenschaftsbedingungen*

  Im Rahmen der Fallstudie wurden die vier Architekturbausteine Schichtenbaustein, Pull-Observerbaustein, Fassadebaustein und Hierarchische Komposition mit dem erarbeiteten Beschreibungsansatz beschrieben und formalisiert. Zusätzlich wurden für jeden dieser Architekturbausteine Eigenschaftsbedingungen formuliert und durch prädikatenlogische Sätze ausgedrückt. Einige der Eigenschaftsbedingungen beziehen sich dabei nicht nur auf eine Instanz des Bausteins, sondern machen Aussagen über das Zusammenspiel verschiedener Bausteininstanzen.

- *Entwurf einer Werkzeugunterstützung*

  Für die Unterstützung des bausteinbasierten Architekturentwurfs wurde ein Werkzeug entworfen. Dieses Werkzeug bietet einen Editor sowohl für die Modellierung von Architekturbausteinen als auch für die Modellierung von Architekturen. Der Editor für die Modellierung von Architekturen unterstützt dabei den Nutzer bei der Anwendung von Architekturbausteinen. Für die Überprüfung von Bedingungen auf Architekturmodellen

und Bausteinen nutzt das Werkzeug ein logikbasiertes System; in dieser Arbeit wird das Wissensrepräsentation- und Anfragesystem PowerLoom eingesetzt.

Die oben aufgeführten Ergebnisse sind in den vorangegangenen Kapiteln dieser Arbeit ausführlicher beschrieben. Zudem werden viele der genannten Punkte durch Beispiele illustriert.

10.2. Diskussion und Ausblick

In Abschnitt 4.4 wurden drei Forschungsfragen formuliert. Diese Fragen wurden aus den Problemen abgeleitet, die bei dem Architekturentwurf eines Beispielsystems aufgetreten waren. Im Folgenden werden die Ergebnisse dieser Arbeit in Bezug zu den Forschungsfragen gestellt und diskutiert. Hierzu werden zunächst noch einmal die Forschungsfragen wiederholt:

Forschungsfrage 1:

Wie können Architekturbausteine präzise beschrieben werden?

Forschungsfrage 2:

Wie lassen sich in einem Architekturmodell angewendete Architekturbausteine inklusive ihrer Bestandteile eindeutig identifizieren?

Forschungsfrage 3:

Wie können die Eigenschaftsbedingungen eines Architekturbausteins möglichst universell formuliert und ihre Einhaltung in einem Architekturmodell überprüft werden?

Auf Forschungsfrage 1 gibt diese Arbeit mit der in Abschnitt 6.3 definierten Beschreibungssprache für Architekturbausteine eine Antwort. Diese rollenbasierte Beschreibungssprache erlaubt es, sowohl Struktur als auch Verhalten eines Architekturbausteins präzise zu beschreiben. Ergänzende Bedingungen über dem Metamodell der Sprache stellen die korrekte Verwendung der Sprache sicher. Es kann zudem präzise festgelegt werden, wie oft jede Rolle bei der Instanziierung des Bausteins gebunden werden darf. Diese Beschreibungssprache wurde aus der Analyse von Architekturkonzepten (u.a. aus [GHJV94] und [BCK03]) abgeleitet, die in der Praxis relevant und verbreitet sind. Ihre Tauglichkeit wurde zudem im Rahmen der Fallstudie an unterschiedlichen Architekturbausteinen gezeigt.

Es ist allerdings nicht auszuschließen, dass Architekturkonzepte existieren, die erweiterte Beschreibungskonzepte benötigen. Beispielsweise kann das Architekturkonzept der serviceorientierten Schnittstelle [HRB+08] nur indirekt beschrieben werden. Eine serviceorientierte Schnittstelle zeichnet sich dadurch aus, dass sie nur serviceorientierte Methoden enthält. Serviceorientierte Methoden können als Parameter nur primitive Datentypen oder Transferobjektklassen [Fow02] besitzen. Darüber hinaus realisieren diese Schnittstellen eine Kopiersemantik, was heißt, dass nur Kopien von Objekten weitergereicht werden. Die Bausteinsprache dieser Arbeit besitzt allerdings keine Sprachmittel, um objektorientierte Typen oder Parameter von Nachrichten zu beschreiben. Nichtsdestotrotz kann dieses Architekturkonzept beschrieben werden, indem die Einschränkungen bezüglich der Schnittstellen in Form von Eigenschaftsbedingungen formuliert werden. Dabei werden Adapterprädikate eingesetzt, die die Typeinschränkungen der Bausteinbeschreibung auf abstrakte Weise zur Verfügung stellen.

Forschungsfrage 2 wird durch den Integrationsteils des Metamodells adressiert. Dieser erlaubt es, jede Instanz eines Architekturbausteins zu ermitteln. Zudem können alle Bindungen, die im

Rahmen einer Bausteininstanz erfolgen, und die auf beiden Seiten beteiligten Elemente eindeutig identifiziert werden. Die Nützlichkeit dieses Konzepts zeigt sich darin, dass Eigenschaftsbedingungen trotz ihrer allgemeinen Formulierung im Kontext einer konkreten Bausteininstanz ausgewertet werden können.

Die dritte Forschungsfrage stellt zwei Teilfragen. Der erste Teil beschäftigt sich dabei mit der Formulierung von Eigenschaftsbedingungen. Als zugrundeliegender Formalismus wurde dazu die Prädikatenlogik gewählt, auf der auch verschiedene andere Sprachen zur Beschreibung von Bedingungen basieren (vgl. Abschnitt 3.3.2). Eine solche Sprache ist abgesehen von generischen Sprachelementen – z.B. zum Vergleich oder Abzählen von Elementen – auf die durch ihre Signatur definierten Sprachelemente eingeschränkt. Die Signatur besteht in diesem Fall aus Relationssymbolen, die aus dem Metamodell für die Bausteinbeschreibung und der Integrationsbeschreibung abgeleitet werden. Die Ausdrucksfähigkeit der Sprache zur Formulierung der Eigenschaftsbedingungen wird zusätzlich durch Adapterprädikate indirekt auf die Architekturbeschreibungssprache erweitert. Adapterprädikate formulieren Sachverhalte auf Basis der aus dem Architekturmetamodell abgeleiteten Relationssymbole und können in Eigenschaftsbedingungen verwendet werden. Allerdings wird die durch den Integrationsteil propagierte Trennung von Architektur- und Bausteinbeschreibung durch die Adapterprädikate aufgeweicht. Beispielsweise könnte die Architekturbeschreibungssprache so minimal angepasst werden, dass der Integrationsteil davon unberührt bleibt. Die Definitionen der Adapterprädikate können hingegen von den Änderungen betroffen sein und müssen angepasst werden. Je nach Umfang der Änderungen und der definierten Adapterprädikate müssen ggf. sogar Eigenschaftsbedingungen neu formuliert werden.

Der zweite Teil von Forschungsfrage 3 fragt danach, wie die Einhaltung von Eigenschaftsbedingungen im Architekturmodell überprüft werden kann. Ein wesentlicher Aspekt hierfür ist die formale Repräsentation von Architekturmodellen als mathematische Strukturen und die prädikatenlogische Formulierung von Eigenschaftsbedingungen. Dadurch kann für die Überprüfung ein logikbasiertes System, wie das in dieser Arbeit verwendete PowerLoom, eingesetzt werden. Nachteilig verhält sich hierbei allerdings die Komplexität der prädikatenlogischen Resolution, die zu den NP-vollständigen Problemen gehört. Bei überschaubaren Architekturen mit wenigen Bausteinen kann eine Überprüfung noch mit geringem Zeitaufwand in einem entsprechenden Modellierungswerkzeug auch während des Architekturentwurfs durchgeführt werden (vgl. auch die Auswertungen in [Her11]). Mit zunehmendem Umfang des Architekturmodells und Anzahl der Bedingungen steigt die benötigte Rechenzeit stark an, so dass eine Überprüfung von der eigentlichen Modellierung entkoppelt werden muss.

An dieser Stelle bietet sich Potential für weitere Arbeiten. Zunächst können Messreihen an Softwaresystemen unterschiedlicher Größe durchgeführt werden, die Aufschluss über die tatsächlich benötigte Rechenzeit einer realistischen Menge an Bedingungen geben. Ergänzend kann untersucht werden, ob, und wenn ja wie, die Auswertung insgesamt effizienter gestaltet werden kann. Hierfür bietet ggf. auch die Formulierung der Bedingungen Optimierungsmöglichkeiten.

Der in dieser Arbeit erstellte Ansatz zum bausteinbasierten Entwurf wurde in Kapitel 8 auf ein Beispielsystem angewendet. Dabei handelt es sich um ein realistisches, wenn auch nur um ein kleines Softwaresystem. Die Anwendung auf ein komplexes System kann zusätzliche Erkenntnisse liefern. Zudem sind entsprechende empirische Untersuchungen anzustellen, um den Nutzen des Ansatzes in realen Systemen zu evaluieren. Erst dann kann endgültig beurteilt werden, ob z.B. Softwarearchitekturen mit dem bausteinbasierten Architekturentwurf effizienter entworfen werden können und dabei weniger Inkonsistenzen entstehen. Hierfür ist auch eine umfangreiche Werkzeugunterstützung hilfreich, die den in dieser Arbeit vorgestellten Entwurf

implementiert. Zur Unterstützung der Bedienbarkeit sollte auch das vorgeschlagene Schema zur vereinfachten Formulierung von Eigenschaftsbedingungen verwendet werden.

Das Konzept zur Überprüfung von Eigenschaftsbedingungen in Architekturmodellen geht davon aus, dass die Bausteininstanzen mit Hilfe der Elemente des Integrationsmodells gekennzeichnet sind. Es ist denkbar, dieses Konzept nicht nur im Kontext des hier vorgestellten Architekturentwurfs zu betrachten. Vorstellbar ist eine Erweiterung, mit deren Hilfe auch ungekennzeichnete Bausteininstanzen in Architekturmodellen identifizierbar sind. Auf diese Weise kann das Überprüfungskonzept auch zur Qualitätssicherung von Softwaresystemen eingesetzt werden, die nicht nach dem vorliegenden Ansatz entwickelt wurden.

In Abschnitt 5.2 wurde skizziert, wie sich der bausteinbasierte Entwurf in den generellen Ablauf des Architekturentwurfs integriert. Hierbei wurde bereits das Ableiten neuer Architekturbausteine aus einer entworfenen Architektur erwähnt. Eine Frage, die sich an einen solchen Ableitungsmechanismus stellt, ist z.B. die, wie sich dabei aus den Eigenschaftsbedingungen von kombinierten Bausteininstanzen die Eigenschaftsbedingungen für einen neuen Architekturbaustein ableiten lassen.

Anhang

A. Integrierte Metamodelle

A.1. Teilmetamodell der Bausteinbeschreibung

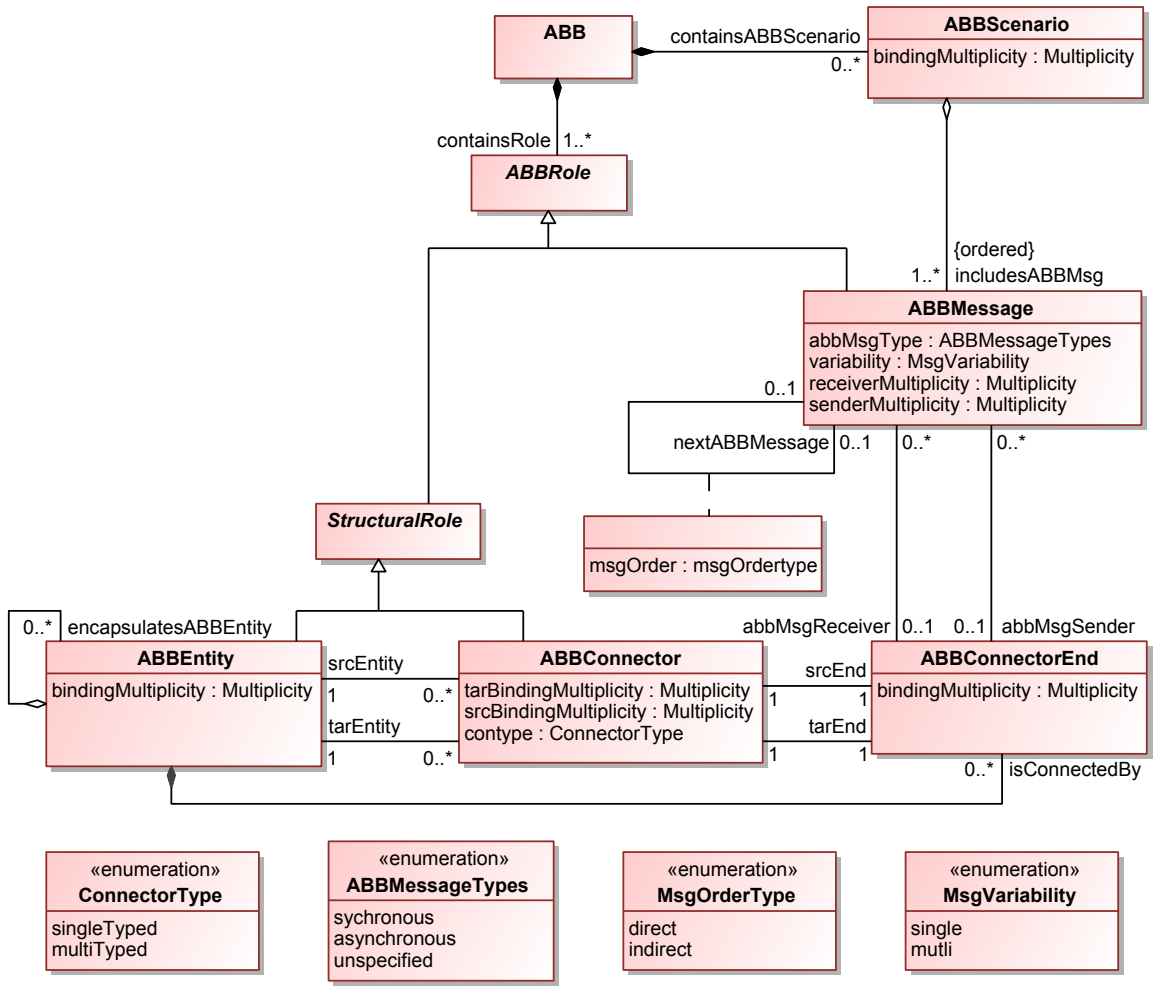


Abbildung A.1: Integriertes Metamodell zur Bausteinbeschreibung.

A.2. Teilmetamodell der Architekturbeschreibung

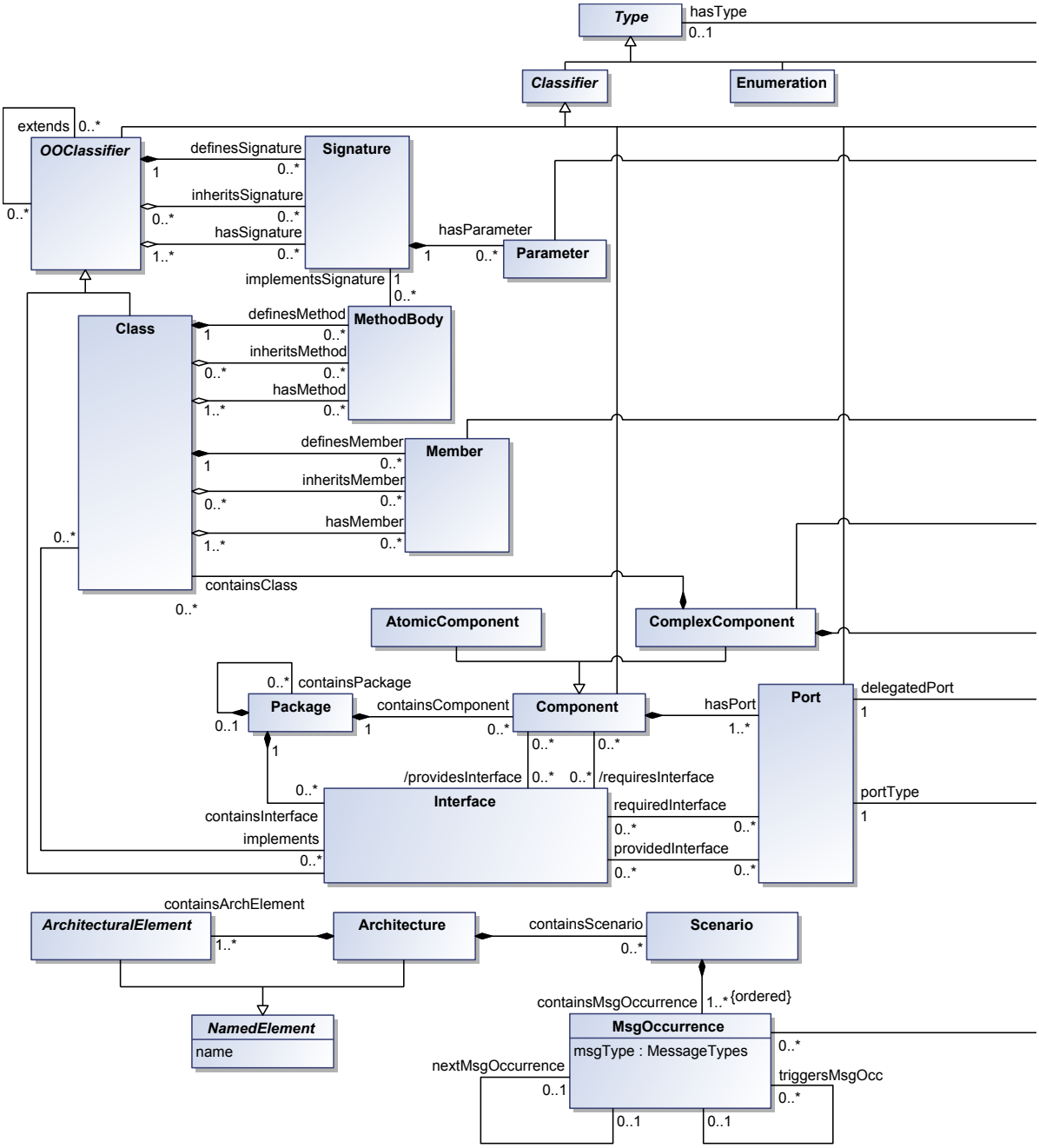
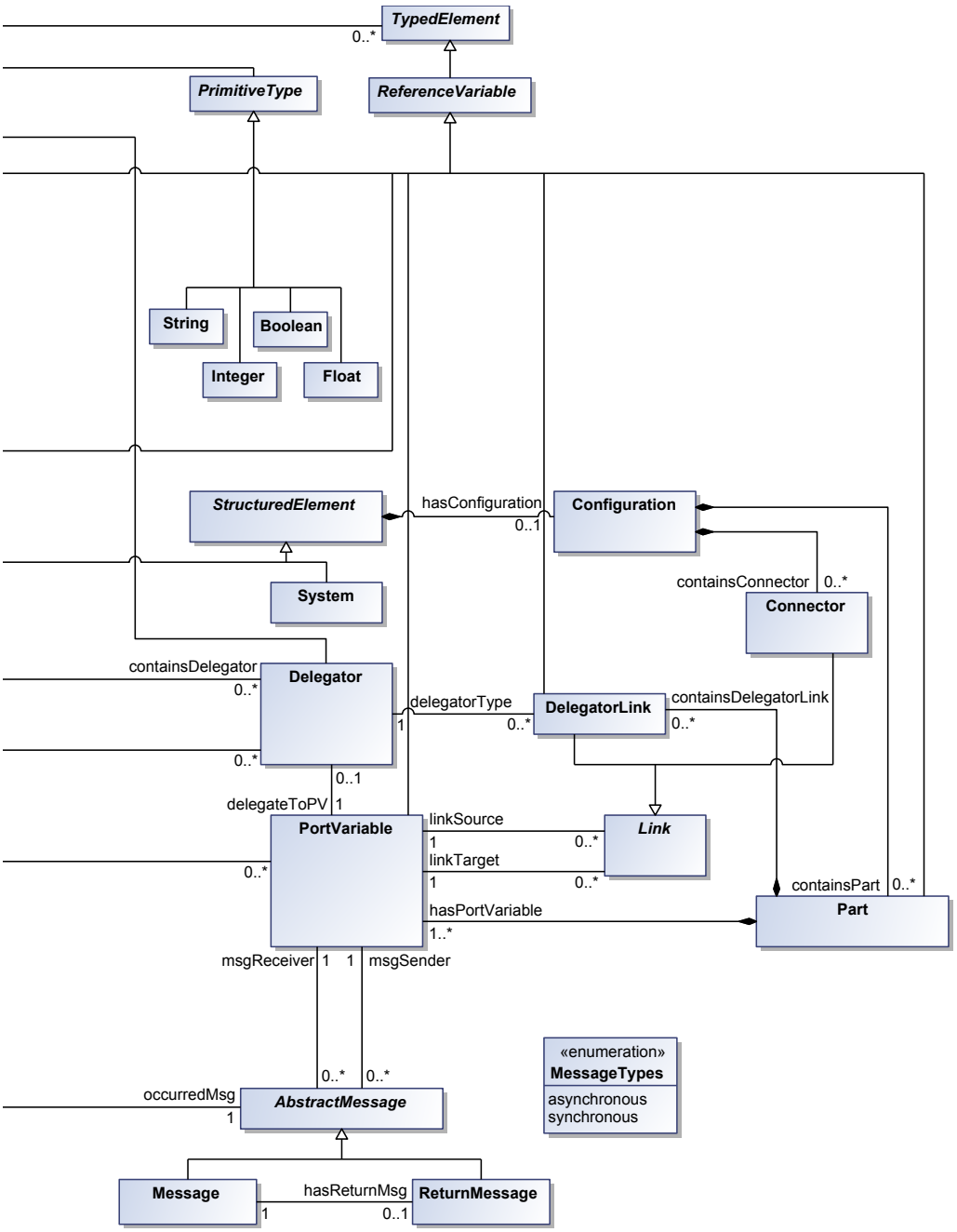


Abbildung A.2: Integriertes Metamodell zur Architekturbeschreibung.


TypedElement
0..*
PrimitiveType
ReferenceVariable
String
Boolean
Integer
Float
StructuredElement
hasConfiguration
0..1
Configuration
containsConnector
0..*
Connector
System
containsDelegator
0..*
Delegator
delegatorType
1
DelegatorLink
containsDelegatorLink
0..*
0..*
0..1
delegateToPV
1
PortVariable
linkSource
1
0..*
Link
linkTarget
1
0..*
0..*
containsPart
0..*
Part
hasPortVariable
1..*
msgReceiver
1
1
msgSender
0..*
0..*
occurredMsg
1
AbstractMessage
«enumeration»
MessageTypes
asynchronous
synchronous
Message
hasReturnMsg
1
0..1
ReturnMessage

B. Ausformulierte Eigenschaftsbedingungen der im Beispielsystem verwendeten Architekturbausteine

Im Folgen finden sich die Eigenschaftsbedingungen als ausformulierte prädikatenlogische Ausdrücke, die in Abschnitt 8.2 lediglich abkürzend aufgeführt wurden.

Schichtenbaustein

Der Schichtenbaustein besitzt zunächst die Eigenschaftsbedingung $\mathbf{ABBC}_{Schichten1}$ zur Überprüfung, dass keine gebundene untere Schicht (transitiv) von einer gebundenen oberen Schicht derselben Bausteininstanz abhängt:

$\mathbf{ABBC}_{Schichten1} :=$
$\forall o \forall p : \texttt{ABBOccurrence}(o) \wedge \texttt{ABB}(p) \wedge \texttt{instantiatedABB}(o,p) \wedge$
$\quad \texttt{name}(p, \texttt{"Schichtenbaustein"}) \wedge$
$\forall m \forall n : \texttt{Binding}(m) \wedge \texttt{containsBinding}(o,m) \wedge \texttt{Binding}(n) \wedge \texttt{containsBinding}(o,n) \wedge$
$\forall r \forall s : \texttt{ABBEntity}(r) \wedge \texttt{name}(r, \texttt{"obereSchicht"}) \wedge \texttt{bindsABBElement}(m,r) \wedge$
$\quad \texttt{ABBEntity}(s) \wedge \texttt{name}(s, \texttt{"untereSchicht"}) \wedge \texttt{bindsABBElement}(n,s) \wedge$
$\forall a \forall b : \texttt{BindableArchElement}(a) \wedge \texttt{bindsArchElement}(m,a) \wedge$
$\quad \texttt{BindableArchElement}(b) \wedge \texttt{bindsArchElement}(n,b) \rightarrow$
$\neg \texttt{dependingLayerUsage}(b,a)$[1]

[1] (AP 2)

Darüber hinaus stellt die Eigenschaftsbedingung $\mathbf{ABBC}_{Schichten2}$ sicher, dass die strikte Hierarchie eingehalten wird. Aus einer oberen Schicht heraus darf nicht auf die untere Schicht der eigenen unteren Schicht zugegriffen werden:

$\mathbf{ABBC}_{Schichten2} :=$
$\forall o \forall p : \texttt{ABBOccurrence}(o) \wedge \texttt{ABB}(p) \wedge \texttt{instantiatedABB}(o,p) \wedge$
$\quad \texttt{name}(p, \texttt{"Schichtenbaustein"}) \wedge$
$\forall m : \texttt{Binding}(m) \wedge \texttt{containsBinding}(o,m) \wedge$
$\forall r : \texttt{ABBEntity}(r) \wedge \texttt{name}(r, \texttt{"obereSchicht"}) \wedge \texttt{bindsABBElement}(m,r) \wedge$
$\forall a : \texttt{BindableArchElement}(a) \wedge \texttt{bindsArchElement}(m,a) \wedge$
$\forall x : \texttt{BindableArchElement}(x) \wedge \texttt{boundABBEntitiesAreConnected}(a,x)$[1] $\rightarrow$
$\neg \texttt{isLowerLayerOfLL}^{\star}(x,a)$[2]

[1] (AP 8),
[2] (AP 17)

Pull-Observerbaustein

Der Pull-Observerbaustein besitzt die beiden Eigenschaftsbedingungen $\mathbf{ABBC}_{PullObserver1}$ und $\mathbf{ABBC}_{Pull\text{-}Observer2}$. Die erste testet, ob die gebundene Nachrichtenrolle `notify` einen Auslöser besitzt.

[1] (AP 9)

$$\mathbf{ABBC}_{Pull\text{-}Observer1} :=$$
$$\forall o \forall p : \texttt{ABBOccurrence}(o) \wedge \texttt{ABB}(p) \wedge \texttt{instantiatedABB}(o,p) \wedge$$
$$\texttt{name}(p, \texttt{"Pull-Observerbaustein"}) \wedge$$
$$\forall m : \texttt{Binding}(m) \wedge \texttt{containsBinding}(o,m) \wedge$$
$$\forall r : \texttt{ABBMessage}(r) \wedge \texttt{name}(r, \texttt{"notify"}) \wedge \texttt{bindsABBElement}(m,r) \wedge$$
$$\forall a : \texttt{BindableArchElement}(a) \wedge \texttt{bindsArchElement}(m,a) \rightarrow$$
$$\texttt{boundHasActivatingMsg}(a)^{1}$$

Die zweite stellt sicher, dass auch alle registrierten Observer bei einer Änderung benachrichtigt werden.

[1] (AP 10), [2] (AP 12), [3] (AP 13), [4] (AP 11)

$$\mathbf{ABBC}_{Pull\text{-}Observer2} :=$$
$$\forall o \forall p : \texttt{ABBOccurrence}(o) \wedge \texttt{ABB}(p) \wedge \texttt{instantiatedABB}(o,p) \wedge$$
$$\texttt{name}(p, \texttt{"Pull-Observerbaustein"}) \wedge$$
$$\forall m \forall n : \texttt{Binding}(m) \wedge \texttt{containsBinding}(o,m) \wedge$$
$$\texttt{Binding}(n) \wedge \texttt{containsBinding}(o,n) \wedge$$
$$\forall r \forall e : \texttt{ABBMessage}(r) \wedge \texttt{name}(r, \texttt{"register"}) \wedge \texttt{bindsABBElement}(m,r) \wedge$$
$$\texttt{ABBEntity}(e) \wedge \texttt{name}(e, \texttt{"Observer"}) \wedge \texttt{bindsABBElement}(n,e) \wedge$$
$$\forall a \forall b : \texttt{BindableArchElement}(a) \wedge \texttt{bindsArchElement}(m,a) \wedge$$
$$\texttt{BindableArchElement}(b) \wedge \texttt{bindsArchElement}(n,b) \wedge$$
$$\texttt{boundABBEntityIsABBMsgSender}(b,a)^{1} \wedge$$
$$\exists l : \texttt{Binding}(l) \wedge \texttt{containsBinding}(o,l) \wedge$$
$$\exists s : \texttt{ABBMessage}(s) \wedge \texttt{name}(s, \texttt{"notify"}) \wedge \texttt{bindsABBElement}(l,s) \wedge$$
$$\exists c : \texttt{BindableArchElement}(c) \wedge \texttt{bindsArchElement}(l,c) \wedge$$
$$\texttt{nextBoundABBMsg}^{*}(a,c)^{2} \rightarrow$$
$$\exists k : \texttt{Binding}(k) \wedge \texttt{containsBinding}(o,k) \wedge$$
$$\exists t : \texttt{ABBMessage}(t) \wedge \texttt{name}(t, \texttt{"update"}) \wedge \texttt{bindsABBElement}(k,t) \wedge$$
$$\exists d : \texttt{BindableArchElement}(d) \wedge \texttt{bindsArchElement}(k,d) \wedge$$
$$\texttt{nextBoundABBMsg}(c,d)^{3} \wedge \texttt{boundABBEntityIsABBMsgReceiver}(b,d)^{4}$$

Hierarchische Komposition

Der Baustein Hierarchische Komposition besitzt zwei Eigenschaftsbedingungen, die durch $\mathbf{ABBC}_{HierKomposition1}$ und $\mathbf{ABBC}_{HierKomposition2}$ definiert sind. Die erste prüft zunächst, ob alle Elemente innerhalb eines gebundenen `HCompositum` auch als `HComponens` gebunden sind.

$\mathbf{ABBC}_{HierKomposition1} :=$ [1](AP 16)

$$
\begin{aligned}
&\forall o \forall p : \texttt{ABBOccurrence}(o) \wedge \texttt{ABB}(p) \wedge \texttt{instantiatedABB}(o,p) \wedge \\
&\qquad \texttt{name}(p, \texttt{"HierarchischeKomposition"}) \wedge \\
&\forall m : \texttt{Binding}(m) \wedge \texttt{containsBinding}(o,m) \wedge \\
&\forall r : \texttt{ABBEntity}(r) \wedge \texttt{name}(r, \texttt{"HCompositum"}) \wedge \texttt{bindsABBElement}(m,r) \wedge \\
&\forall a : \texttt{BindableArchElement}(a) \wedge \texttt{bindsArchElement}(m,a) \wedge \\
&\forall k : \texttt{BindableArchElement}(k) \wedge \texttt{boundABBEntityEncapsulates}(a,k)^1 \rightarrow \\
&\exists c \exists l \exists s : \texttt{BindableArchElement}(c) \wedge \texttt{Binding}(l) \wedge \\
&\texttt{bindsArchElement}(l,c) \wedge \texttt{containsBinding}(o,l) \wedge \\
&\texttt{ABBEntity}(s) \wedge \texttt{bindsABBElement}(l,s) \wedge \texttt{name}(s, \texttt{"HComponens"}) \wedge (k \equiv c)
\end{aligned}
$$

Die zweite Bedingung stellt sicher, dass auf ein gebundenes `HComponens` kein Element zugreift, das außerhalb des zugehörigen gebundenen `HCompositum` liegt.

$\mathbf{ABBC}_{HierKomposition2} :=$ [1](AP 17)

$$
\begin{aligned}
&\forall o \forall p : \texttt{ABBOccurrence}(o) \wedge \texttt{ABB}(p) \wedge \texttt{instantiatedABB}(o,p) \wedge \\
&\qquad \texttt{name}(p, \texttt{"HierarchischeKomposition"}) \wedge \\
&\forall m : \texttt{Binding}(m) \wedge \texttt{containsBinding}(o,m) \wedge \\
&\forall r : \texttt{ABBEntity}(r) \wedge \texttt{name}(r, \texttt{"HComponens"}) \wedge \texttt{bindsABBElement}(m,r) \wedge \\
&\forall a : \texttt{BindableArchElement}(a) \wedge \texttt{bindsArchElement}(m,a) \wedge \\
&\forall k : \texttt{BindableArchElement}(k) \wedge \texttt{boundABBEntitiesAreConnected}(a,k)^1 \rightarrow \\
&\exists c \exists l \exists s : \texttt{BindableArchElement}(c) \wedge \texttt{Binding}(l) \wedge \\
&\texttt{bindsArchElement}(l,c) \wedge \texttt{containsBinding}(o,l) \wedge \\
&\texttt{ABBEntity}(s) \wedge \texttt{bindsABBElement}(l,s) \wedge (k \equiv s) \wedge \\
&(\texttt{name}(s, \texttt{"HComponens"}) \vee \texttt{name}(s, \texttt{"HCompositum"}))
\end{aligned}
$$

Fassadebaustein

Die Eigenschaftsbedingung $\mathbf{ABBC}_{Fassade1}$ des Fassadebausteins überprüft, dass der Zugriff auf die Fassadenschnittstellen der Subsystemelemente nur durch andere Subsystemelemente oder die Fassade erfolgt.

[1](AP 14), [2](AP 15)

$$
\begin{aligned}
&\mathbf{ABBC}_{Fassade1} := \\
&\forall o \forall p : \texttt{ABBOccurrence}(o) \wedge \texttt{ABB}(p) \wedge \texttt{instantiatedABB}(o,p) \wedge \\
&\qquad \texttt{name}(p, \texttt{"Fassadebaustein"}) \wedge \\
&\forall m : \texttt{Binding}(m) \wedge \texttt{containsBinding}(o,m) \wedge \\
&\forall r : \texttt{ABBConnectorEnd}(r) \wedge \texttt{name}(r, \texttt{"fc\_end2"}) \wedge \texttt{bindsABBElement}(m,r) \wedge \\
&\forall a : \texttt{BindableArchElement}(a) \wedge \texttt{bindsArchElement}(m,a) \wedge \\
&\forall b : \texttt{BindableArchElement}(b) \wedge \texttt{isOppositeToBoundCEnd}(b,r)^{1} \rightarrow \\
&\exists c \exists l \exists s : \texttt{BindableArchElement}(c) \wedge \texttt{Binding}(l) \wedge \\
&\texttt{bindsArchElement}(l,c) \wedge \texttt{containsBinding}(o,l) \wedge \\
&\texttt{ABBEntity}(s) \wedge \texttt{bindsABBElement}(l,s) \wedge \texttt{belongsToBoundABBEntity}(b,s)^{2} \wedge \\
&(\texttt{name}(s, \texttt{"Fassade"}) \vee \texttt{name}(s, \texttt{"Subsystemelement"}))
\end{aligned}
$$

C. Formale Repräsentation der Architekturbausteine des Beispielsystems

Schichtenbaustein

Der Schichtenbaustein wird durch die Struktur $\mathfrak{B}_{Schichten}$ mit einer Abbildung der Trägermenge auf folgende Relationen repräsentiert[11]:

$\texttt{ABB}^{\mathfrak{B}} = \{\texttt{schichtenabb}\}$

$\texttt{containsRole}^{\mathfrak{B}} = \{(\texttt{schichtenabb}, \texttt{obereSchicht}), (\texttt{schichtenabb}, \texttt{untereSchicht}), (\texttt{schichtenabb}, \texttt{schichten\_con})\}$

$\texttt{ABBRole}^{\mathfrak{B}} = \{\texttt{obereSchicht}, \texttt{untereSchicht}, \texttt{schichten\_con}\}$

$\texttt{StructuralRole}^{\mathfrak{B}} = \{\texttt{obereSchicht}, \texttt{untereSchicht}, \texttt{schichten\_con}\}$

$\texttt{ABBConnector}^{\mathfrak{B}} = \{\texttt{schichten\_con}\}$

$\texttt{tarBindingMultiplicity}^{\mathfrak{B}} = \{(\texttt{schichten\_con}, \texttt{1..*})\}$

$\texttt{srcBindingMultiplicity}^{\mathfrak{B}} = \{(\texttt{schichten\_con}, \texttt{1..*})\}$

$\texttt{contype}^{\mathfrak{B}} = \{(\texttt{schichten\_con}, \texttt{"multiTyped"})\}$

$\texttt{ABBConnectorEnd}^{\mathfrak{B}} = \{\texttt{sc\_end1}, \texttt{sc\_end2}\}$

$\texttt{bindingMultiplicity}^{\mathfrak{B}} = \{(\texttt{obereSchicht}, \texttt{1..*}), (\texttt{untereSchicht}, \texttt{1..*}), (\texttt{sc\_end1}, \texttt{1..*}), (\texttt{sc\_end2}, \texttt{1..*})\}$

$\texttt{srcEnd}^{\mathfrak{B}} = \{(\texttt{schichten\_con}, \texttt{sc\_end1})\}$

$\texttt{tarEnd}^{\mathfrak{B}} = \{(\texttt{schichten\_con}, \texttt{sc\_end2})\}$

$\texttt{ABBEntity}^{\mathfrak{B}} = \{\texttt{obereSchicht}, \texttt{untereSchicht}\}$

$\texttt{encapsulatesABBEntity}^{\mathfrak{B}} = \emptyset$

$\texttt{isConnectedBy}^{\mathfrak{B}} = \{(\texttt{obereSchicht}, \texttt{sc\_end1}), (\texttt{untereSchicht}, \texttt{sc\_end2})\}$

$\texttt{ABBScenario}^{\mathfrak{B}} = \emptyset$

$\texttt{containsABBScenario}^{\mathfrak{B}} = \emptyset$

$\texttt{ABBMessage}^{\mathfrak{B}} = \emptyset$

$\texttt{includesABBMsg}^{\mathfrak{B}} = \emptyset$

$\texttt{variability}^{\mathfrak{B}} = \emptyset$

$\texttt{abbMsgTypes}^{\mathfrak{B}} = \emptyset$

$\texttt{receiverMultiplicity}^{\mathfrak{B}} = \emptyset$

$\texttt{senderMultiplicity}^{\mathfrak{B}} = \emptyset$

$\texttt{abbMsgReceiver}^{\mathfrak{B}} = \emptyset$

$\texttt{abbMsgSender}^{\mathfrak{B}} = \emptyset$

[11] Aus Gründen der Übersichtlichkeit wird hier und in den folgenden Abschnitten jeweils auf das Subskript von $\mathfrak{B}$ bei den notierten Relationen verzichtet.

$\texttt{nextABBMessage}^{\mathfrak{B}} = \emptyset$

$\texttt{hasName}^{\mathfrak{B}} = \{(\texttt{schichtenabb}, \texttt{"Schichtenbaustein"}),$
$(\texttt{obereSchicht}, \texttt{"obereSchicht"}), (\texttt{untereSchicht}, \texttt{"untereSchicht"}), ...\}$

Pull-Observerbaustein

Der Pull-Observerbaustein wird durch die Struktur $\mathfrak{B}_{Pull-Observer}$ mit einer Abbildung der Trägermenge auf folgende Relationen repräsentiert:

$\texttt{ABB}^{\mathfrak{B}} = \{\texttt{observerabb}\}$

$\texttt{containsRole}^{\mathfrak{B}} = \{(\texttt{observerabb}, \texttt{observer}), (\texttt{observerabb}, \texttt{subject}),$
$(\texttt{observerabb}, \texttt{obs\_con}), (\texttt{observerabb}, \texttt{modell\_con}), (\texttt{observerabb}, \texttt{register}),$
$(\texttt{observerabb}, \texttt{notify}), (\texttt{observerabb}, \texttt{update}), (\texttt{observerabb}, \texttt{getModell})\}$

$\texttt{ABBRole}^{\mathfrak{B}} = \{\texttt{observer}, \texttt{subject}, \texttt{obs\_con}, \texttt{modell\_con}, \texttt{register},$
$\texttt{notify}, \texttt{update}, \texttt{getModell}\}$

$\texttt{StructuralRole}^{\mathfrak{B}} = \{\texttt{observer}, \texttt{subject}, \texttt{obs\_con}, \texttt{modell\_con}\}$

$\texttt{ABBConnector}^{\mathfrak{B}} = \{\texttt{obs\_con}, \texttt{modell\_con}\}$

$\texttt{tarBindingMultiplicity}^{\mathfrak{B}} = \{(\texttt{obs\_con}, \texttt{1..*}), (\texttt{modell\_con}, \texttt{1..*})\}$

$\texttt{srcBindingMultiplicity}^{\mathfrak{B}} = \{(\texttt{obs\_con}, \texttt{1}), (\texttt{modell\_con}, \texttt{1})\}$

$\texttt{contype}^{\mathfrak{B}} = \{(\texttt{obs\_con}, \texttt{"singleTyped"}), (\texttt{model\_con}, \texttt{"multiTyped"})\}$

$\texttt{ABBConnectorEnd}^{\mathfrak{B}} = \{\texttt{oc\_end1}, \texttt{oc\_end2}, \texttt{mc\_end1}, \texttt{mc\_end2}\}$

$\texttt{bindingMultiplicity}^{\mathfrak{B}} = \{(\texttt{observer}, \texttt{1..*}), (\texttt{subject}, \texttt{1}), (\texttt{oc\_end1}, \texttt{1}), (\texttt{oc\_end2}, \texttt{1}),$
$(\texttt{mc\_end1}, \texttt{1..*}), (\texttt{mc\_end2}, \texttt{1..*}), (\texttt{obs\_con}, *)\}$

$\texttt{srcEnd}^{\mathfrak{B}} = \{(\texttt{obs\_con}, \texttt{oc\_end1}), (\texttt{modell\_con}, \texttt{mc\_end1})\}$

$\texttt{tarEnd}^{\mathfrak{B}} = \{(\texttt{obs\_con}, \texttt{oc\_end2}), (\texttt{modell\_con}, \texttt{mc\_end2})\}$

$\texttt{ABBEntity}^{\mathfrak{B}} = \{\texttt{observer}, \texttt{subject}\}$

$\texttt{encapsulatesABBEntity}^{\mathfrak{B}} = \emptyset$

$\texttt{isConnectedBy}^{\mathfrak{B}} = \{(\texttt{observer}, \texttt{oc\_end1}), (\texttt{observer}, \texttt{mc\_end1}),$
$(\texttt{subject}, \texttt{oc\_end2}), (\texttt{subject}, \texttt{mc\_end2})\}$

$\texttt{ABBScenario}^{\mathfrak{B}} = \{\texttt{obs\_scn}\}$

$\texttt{containsABBScenario}^{\mathfrak{B}} = \{(\texttt{observerabb}, \texttt{obs\_scn})\}$

$\texttt{ABBMessage}^{\mathfrak{B}} = \{\texttt{register}, \texttt{notify}, \texttt{update}, \texttt{getModell}\}$

$\texttt{includesABBMsg}^{\mathfrak{B}} = \{(\texttt{obs\_con}, \texttt{register}), (\texttt{obs\_con}, \texttt{notify}),$
$(\texttt{obs\_con}, \texttt{update}), (\texttt{obs\_con}, \texttt{getModell})\}$

$\texttt{variability}^{\mathfrak{B}} = \{(\texttt{register}, \texttt{"single"}), (\texttt{notify}, \texttt{"single"}),$
$(\texttt{update}, \texttt{"single"}), (\texttt{getModell}, \texttt{"single"})\}$

$\texttt{abbMsgTypes}^{\mathfrak{B}} = \{(\texttt{register}, \texttt{"unspecified"}), (\texttt{notify}, \texttt{"unspecified"}),$
$(\texttt{update}, \texttt{"unspecified"}), (\texttt{getModell}, \texttt{"unspecified"})\}$

$\texttt{receiverMultiplicity}^{\mathfrak{B}} = \{(\texttt{register}, \texttt{1}), (\texttt{notify}, \texttt{1}), (\texttt{update}, *), (\texttt{getModell}, \texttt{1})\}$

$\texttt{senderMultiplicity}^{\mathfrak{B}} = \{(\texttt{register}, *), (\texttt{notify}, \texttt{1}), (\texttt{update}, \texttt{1}), (\texttt{getModell}, *)\}$

$\texttt{abbMsgReceiver}^{\mathfrak{B}} = \{(\texttt{register}, \texttt{oc\_end2}), (\texttt{notify}, \texttt{oc\_end2}), (\texttt{update}, \texttt{oc\_end1}), (\texttt{getModell}, \texttt{mc\_end2})\}$

$\texttt{abbMsgSender}^{\mathfrak{B}} = \{(\texttt{register}, \texttt{oc\_end1}), (\texttt{update}, \texttt{oc\_end2}), (\texttt{getModell}, \texttt{mc\_end1})\}$

$\texttt{nextABBMessage}^{\mathfrak{B}} = \{(\texttt{register}, \texttt{notify}, \texttt{"indirect"}), (\texttt{notify}, \texttt{update}, \texttt{"direct"}), (\texttt{update}, \texttt{getModell}, \texttt{"direct"})\}$

$\texttt{hasName}^{\mathfrak{B}} = \{(\texttt{observerabb}, \texttt{"Observerbaustein"}), (\texttt{observer}, \texttt{"Observer"}), (\texttt{subject}, \texttt{"Subject"}), (\texttt{register}, \texttt{"register"}), (\texttt{notify}, \texttt{"notify"}), (\texttt{update}, \texttt{"update"}), (\texttt{getModell}, \texttt{"getModell"}), ...\}$

Fassadebaustein

Der Fassadebaustein wird durch die Struktur $\mathfrak{B}_{Fassade}$ mit einer Abbildung der Trägermenge auf folgende Relationen repräsentiert:

$\texttt{ABB}^{\mathfrak{B}} = \{\texttt{fassadeabb}\}$

$\texttt{containsRole}^{\mathfrak{B}} = \{(\texttt{fassadeabb}, \texttt{fassade}), (\texttt{fassadeabb}, \texttt{subsyselem}), (\texttt{fassadeabb}, \texttt{fassade\_con}), (\texttt{fassadeabb}, \texttt{fasCall}), (\texttt{fassadeabb}, \texttt{elemCall})\}$

$\texttt{ABBRole}^{\mathfrak{B}} = \{\texttt{fassade}, \texttt{subsyselem}, \texttt{fassade\_con}, \texttt{fasCall}, \texttt{elemCall}\}$

$\texttt{StructuralRole}^{\mathfrak{B}} = \{\texttt{fassade}, \texttt{subsyselem}, \texttt{fassade\_con}\}$

$\texttt{ABBConnector}^{\mathfrak{B}} = \{\texttt{fassade\_con}\}$

$\texttt{tarBindingMultiplicity}^{\mathfrak{B}} = \{(\texttt{fassade\_con}, 1)\}$

$\texttt{srcBindingMultiplicity}^{\mathfrak{B}} = \{(\texttt{fassade\_con}, 1)\}$

$\texttt{contype}^{\mathfrak{B}} = \{(\texttt{fassade\_con}, \texttt{"multiTyped"})\}$

$\texttt{ABBConnectorEnd}^{\mathfrak{B}} = \{\texttt{fas\_end}, \texttt{fc\_end1}, \texttt{fc\_end2}\}$

$\texttt{bindingMultiplicity}^{\mathfrak{B}} = \{(\texttt{fassade}, 1), (\texttt{subsyselem}, 1..*), (\texttt{fas\_end}, 1..*), (\texttt{fc\_end1}, 1..*), (\texttt{fc\_end2}, 1..*), (\texttt{fas\_scn}, *)\}$

$\texttt{srcEnd}^{\mathfrak{B}} = \{(\texttt{fassade\_con}, \texttt{fc\_end1})\}$

$\texttt{tarEnd}^{\mathfrak{B}} = \{(\texttt{fassade\_con}, \texttt{fc\_end2})\}$

$\texttt{ABBEntity}^{\mathfrak{B}} = \{\texttt{fassade}, \texttt{subsyselem}\}$

$\texttt{encapsulatesABBEntity}^{\mathfrak{B}} = \emptyset$

$\texttt{isConnectedBy}^{\mathfrak{B}} = \{(\texttt{fassade}, \texttt{fas\_end}), (\texttt{fassade}, \texttt{fc\_end1}), (\texttt{subsyselem}, \texttt{fc\_end2})\}$

$\texttt{ABBScenario}^{\mathfrak{B}} = \{\texttt{fas\_scn}\}$

$\texttt{containsABBScenario}^{\mathfrak{B}} = \{(\texttt{fassadeabb}, \texttt{fas\_scn})\}$

$\texttt{ABBMessage}^{\mathfrak{B}} = \{\texttt{fasCall}, \texttt{elemCall}\}$

$\texttt{includesABBMsg}^{\mathfrak{B}} = \{(\texttt{fas\_scn}, \texttt{fasCall}), (\texttt{fas\_scn}, \texttt{elemCall})\}$

$\texttt{variability}^{\mathfrak{B}} = \{(\texttt{fasCall}, \texttt{"single"}), (\texttt{elemCall}, \texttt{"multi"})\}$

$\texttt{abbMsgTypes}^{\mathfrak{B}} = \{(\texttt{fasCall}, \texttt{"unspecified"}), (\texttt{elemCal}, \texttt{"unspecified"})\}$

$\texttt{receiverMultiplicity}^{\mathfrak{B}} = \{(\texttt{fasCall}, 1), (\texttt{elemCall}, *)\}$

$\texttt{senderMultiplicity}^{\mathfrak{B}} = \{(\texttt{fassCall}, 1), (\texttt{elemCall}, 1)\}$

$\texttt{abbMsgReceiver}^{\mathfrak{B}} = \{(\texttt{fasCall}, \texttt{fas\_end}), (\texttt{elemCall}, \texttt{fc\_end2})\}$

$\texttt{abbMsgSender}^{\mathfrak{B}} = \{(\texttt{elemCall}, \texttt{fc\_end1})\}$

$\texttt{nextABBMessage}^{\mathfrak{B}} = \{(\texttt{fasCall}, \texttt{elemCall}, \texttt{"direct"})\}$

$\texttt{hasName}^{\mathfrak{B}} = \{(\texttt{fassadeabb}, \texttt{"Fassadebaustein"}), (\texttt{fassade}, \texttt{"Fassade"}),$
$(\texttt{subsyselem}, \texttt{"Subsystemelement"}), ...\}$

Hierarchische Komposition

Der Baustein Hierarchische Komposition wird durch die Struktur $\mathfrak{B}_{HierKomposition}$ mit einer Abbildung der Trägermenge auf folgende Relationen repräsentiert:

$\texttt{ABB}^{\mathfrak{B}} = \{\texttt{hcompabb}\}$

$\texttt{containsRole}^{\mathfrak{B}} = \{(\texttt{hcompabb}, \texttt{hcompositum}), (\texttt{hcompabb}, \texttt{hcomponens}),$
$(\texttt{hcompabb}, \texttt{hcomp\_con}), (\texttt{hcompabb}, \texttt{incoming}), (\texttt{hcompabb}, \texttt{inDelegate}),$
$(\texttt{hcompabb}, \texttt{outDelegate}), (\texttt{hcompabb}, \texttt{outgoing})\}$

$\texttt{ABBRole}^{\mathfrak{B}} = \{\texttt{hcompositum}, \texttt{hcomponens}, \texttt{hcomp\_con}, \texttt{incoming}, \texttt{inDelegate},$
$\texttt{outDelegate}, \texttt{outgoing}\}$

$\texttt{StructuralRole}^{\mathfrak{B}} = \{\texttt{hcompositum}, \texttt{hcomponens}, \texttt{hcomp\_con}\}$

$\texttt{ABBConnector}^{\mathfrak{B}} = \{\texttt{hcomp\_con}\}$

$\texttt{tarBindingMultiplicity}^{\mathfrak{B}} = \{(\texttt{hcmop\_con}, 1)\}$

$\texttt{srcBindingMultiplicity}^{\mathfrak{B}} = \{(\texttt{hcmop\_con}, 1)\}$

$\texttt{contype}^{\mathfrak{B}} = \{(\texttt{hcmop\_con}, \texttt{"multiTyped"})\}$

$\texttt{ABBConnectorEnd}^{\mathfrak{B}} = \{\texttt{hcomp\_end}, \texttt{hc\_end1}, \texttt{hc\_end2}\}$

$\texttt{bindingMultiplicity}^{\mathfrak{B}} = \{(\texttt{hcompositum}, 1), (\texttt{hcomponens}, 1..*), (\texttt{hcomp\_end}, 1..*),$
$(\texttt{hc\_end1}, 1..*), (\texttt{hc\_end2}, 0..*), (\texttt{hcomp\_scn1}, *), (\texttt{hcomp\_scn2}, *)\}$

$\texttt{srcEnd}^{\mathfrak{B}} = \{(\texttt{hcomp\_con}, \texttt{hc\_end1})\}$

$\texttt{tarEnd}^{\mathfrak{B}} = \{(\texttt{hcomp\_con}, \texttt{hc\_end2})\}$

$\texttt{ABBEntity}^{\mathfrak{B}} = \{\texttt{hcompositum}, \texttt{hcomponens}\}$

$\texttt{encapsulatesABBEntity}^{\mathfrak{B}} = \{(\texttt{hcompositum}, \texttt{hcomponens})\}$

$\texttt{isConnectedBy}^{\mathfrak{B}} = \{(\texttt{hcompositum}, \texttt{hcomp\_end}), (\texttt{hcompositum}, \texttt{hc\_end1}),$
$(\texttt{hcomponens}, \texttt{hc\_end2})\}$

$\texttt{ABBScenario}^{\mathfrak{B}} = \{\texttt{hcomp\_scn1}, \texttt{hcomp\_scn2}\}$

$\texttt{containsABBScenario}^{\mathfrak{B}} = \{(\texttt{hcompabb}, \texttt{hcomp\_scn1}), (\texttt{hcompabb}, \texttt{hcomp\_scn2})\}$

$\texttt{ABBMessage}^{\mathfrak{B}} = \{\texttt{incoming}, \texttt{inDelegate}, \texttt{outDelegate}, \texttt{outgoing}\}$

$\texttt{includesABBMsg}^{\mathfrak{B}} = \{(\texttt{hcomp\_scn1}, \texttt{incoming}), (\texttt{hcomp\_scn1}, \texttt{inDelegate}),$
$(\texttt{hcomp\_scn2}, \texttt{outDelegate}), (\texttt{hcomp\_scn2}, \texttt{outgoing})\}$

$\texttt{variability}^{\mathfrak{B}} = \{(\texttt{incoming}, \texttt{"single"}), (\texttt{inDelegate}, \texttt{"single"}),$
$(\texttt{outDelegate}, \texttt{"single"}), (\texttt{outgoing}, \texttt{"single"})\}$

$\texttt{abbMsgTypes}^{\mathfrak{B}} = \{(\texttt{incoming}, \texttt{"unspecified"}), (\texttt{inDelegate}, \texttt{"unspecified"}),$
$(\texttt{outDelegate}, \texttt{"unspecified"}), (\texttt{outgoing}, \texttt{"unspecified"})\}$

$\texttt{receiverMultiplicity}^{\mathfrak{B}} = \{(\texttt{incoming}, 1), (\texttt{inDelegate}, 1), (\texttt{outDelegate}, 1),$
$(\texttt{outgoing}, 1)\}$

$\texttt{senderMultiplicity}^{\mathfrak{B}} = \{(\texttt{incoming},1), (\texttt{inDelegate},1), (\texttt{outDelegate},1),$
$(\texttt{outgoing},1)\}$

$\texttt{abbMsgReceiver}^{\mathfrak{B}} = \{(\texttt{incoming},\texttt{hcomp\_end}), (\texttt{inDelegate},\texttt{hc\_end2}),$
$(\texttt{outDelegate},\texttt{hc\_end1})\}$

$\texttt{abbMsgSender}^{\mathfrak{B}} = \{(\texttt{inDelegate},\texttt{hc\_end1}), (\texttt{outDelegate},\texttt{hc\_end2}),$
$(\texttt{outgoing},\texttt{hcomp\_end})\}$

$\texttt{nextABBMessage}^{\mathfrak{B}} = \{(\texttt{incoming},\texttt{inDelegate},\texttt{"direct"}),$
$(\texttt{outDelegate},\texttt{outgoing},\texttt{"direct"})\}$

$\texttt{hasName}^{\mathfrak{B}} = \{(\texttt{hcompabb},\texttt{"HierarchischeKomposition"}),$
$(\texttt{hcompositum},\texttt{"HCompositum"}), (\texttt{hcompenens},\texttt{"HComponens"}),$
$(\texttt{incoming},\texttt{"incoming"}), (\texttt{inDelegate},\texttt{"inDelegate"}),$
$(\texttt{outDelegate},\texttt{"outDelegate"}), (\texttt{outgoing},\texttt{"outgoing"}), ...\}$

Literaturverzeichnis

[1471] ANSI/IEEE 1471-2000. *Recommended Practice for Architectural Description of Software-intensive Systems*, 2000.

[42010] ISO/IEC/IEEE 42010-2011. *Systems and Software Engineering - Architecture Description*, 201.

[AA96] Ahmed Abd-El-Shafy Abd-Allah. *Composing Heterogeneous Software Architectures*. Dissertation, University of Southern California, 1996.

[AB96] Ahmed Abd-Allah und Barry Boehm. *Models for Composing Heterogeneous Software Architectures*. Technical Report USC-CSE-96-505, Center for Software Engineering, University of Southern California, 1996.

[ACDL99] P. Alencar, D. Cowan, J. Dong, und C. Lucena. *An Evolutionary Approach to Structural Design Composition*. Technischer Bericht CS-99-16, Computer Science Department, University of Waterloo, 1999.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. Dissertation, no. cmu-cs-97-144, Carnegie Mellon University, 1997.

[AP03] Nuno Amálio und Fiona Polack. *Comparison of Formalisation Approaches of UML Class Constructs in Z and Object-Z*. In ZB 2003: Formal Specification and Development in Z and B, 339–358. Springer, 2003.

[arc42] *arc42 - Ressourcen für Softwarearchitekten*. http://www.arc24.de.

[AW02] Ivan Araujo und Michael Weiss. *Linking Patterns and Non-Functional Requirements*. In Proceedings of the 9th Conference on Pattern Language of Programs (PLoP 2002), Monticello Illinois (USA), 2002.

[AZ05] Paris Avgeriou und Uwe Zdun. *Architectural Patterns Revisited - a Pattern Language*. In Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLoP), 1–39, Irsee, Germany, 2005.

[Bal11] Helmut Balzert. *Lehrbuch der Softwaretechnik*. Spektrum Akademischer Verlag, 2011.

[BB87] Tommaso Bolognesi und Ed Brinksma. *Introduction to the ISO Specification Language LOTOS*. Computer Networks and ISDN Systems - Special Issue: Protocol Specification and Testing, 14(1): 25–59, 1987.

[BCK03] Len Bass, Paul Clements, und Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman, Amsterdam, 2003.

[BHH+12] Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, und Dániel Varró. *Integrating Efficient Model Queries in State-of-the-art EMF Tools*. In Objects, Models, Components, Patterns, 1–8. Springer, 2012.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, und Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, Band 1 von Software Design Patterns. Wiley & Sons, 1. Auflage, 1996.

[Bos98] Jan Bosch. *Design Patterns as Language Constructs.* Journal of object-oriented Programming, 11: 18–32, 1998.

[Bos99] J. Bosch. *Superimposition: a Component Adaptation Technique.* Information and Software Technology, 41(5): 257–273, 1999.

[BR70] J.N. Buxton und Brian Randell (Editoren). *Software Engineering Techniques: Report on a Conference sponsored by the NATO Science Committee.* Scientific Affairs Division, NATO, 1970.

[Bro75] Fred Brooks. *The Mythical Man-Month.* Addison-Wesley, 1975.

[Bro01] Tyson R. Browning. *Applying the Design Structure Matrix to System Decomposition and Integration Problems: a Review and New Directions.* IEEE Transactions on Engineering Management, 48(3): 292–306, 2001.

[BRSV00] Klaus Bergner, Andreas Rausch, Marc Sihling, und Alexander Vilbig. *Putting the Parts Together - Concepts, Description Techniques, and Development Process for Componentware.* In Proceedings of the 33rd Hawaii International Conference on System (HICSS), Band 8, Washington, DC, USA, 2000. IEEE Computer Society.

[BVA12] Bundesverwaltungsamt (BVA). *Register Factory.* White paper, www.register-factory.de, 2012.

[BURV11] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, und Dániel Varró. *A Graph Query Language for EMF Models.* In Theory and Practice of Model Transformations, 167–182. Springer, 2011.

[BZ07] Ian Bayley und Hong Zhu. *Formalising Design Patterns in Predicate Logic.* In 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM), 25–36, 2007.

[BZ08a] Ian Bayley und Hong Zhu. *On the Composition of Design Patterns.* In The 8th International Conference on Quality Software (QSIC '08), 27–36. IEEE, 2008.

[BZ08b] Ian Bayley und Hong Zhu. *Specifying behavioural features of design patterns.* Technical Report TR-08-01, Department of Computing, Oxford Brookes University, Oxford, UK, 2008.

[BZ10] Ian Bayley und Hong Zhu. *A Formal Language of Pattern Compositions.* In Proceedings of the 2nd International Conference on Pervasive Patterns (PATTERNS), 1–6, Lisbon, Portugal, 2010.

[CBB+10] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, und Judith Stafford. *Documenting Software Architectures: Views and Beyond.* Addison-Wesley Professional, 1. Auflage, 2010.

[Cle96] Paul C. Clements. *A Survey of Architecture Description Languages.* In Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD '96), 16. IEEE Computer Society, 1996.

[ConQAT] ConQAT Projekt Webseite. https://www.cqse.eu/en/products/conqat.

[Cop96] James O Coplien. *Software Patterns.* SIGS Press, 1996.

[Cru09] David Bettencourt da Cruz. *POSAAM - Eine Methode zu mehr Systematik und Expertenunabhängigkeit in der qualitativen Architekturbewertung.* Dissertation, Technische Universität München, Fakultät für Informatik, 2009.

[DDHR09] Constanze Deiters, Patrick Dohrmann, Sebastian Herold, und Andreas Rausch. *Rule-Based Architectural Compliance Checks for Enterprise Architecture Management.* In Enterprise Distributed Object Computing Conference (EDOC), 183–192. IEEE International, 2009.

[Dep] Dependometer Projekt Webseite. http://source.valtech.com/display/dpm/Dependometer.

[DH06] Hou Daqing und H. James Hoover. *Using SCL to Specify and Check Design Intent in Source Code.* Software Engineering, IEEE Transactions on, 32(6): 404–423, 2006.

[DH09] Constanze Deiters und Sebastian Herold. *Konformität zwischen Code und Architektur - logikbasierte Überprüfung von Architekturregeln.* OBJEKTspektrum, 04/09: 54–59, 2009.

[DHHJ10] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, und Elmar Juergens. *Flexible Architecture Conformance Assessment With ConQAT.* In ACM/IEEE 32nd International Conference on Software Engineering (ICSE), Band 2, 247–250, 2010.

[Don02] Jing Dong. *Design Component Contracts: Modeling and Analysis of Pattern-Based Composition.* Dissertation, Computer Science Department, University of Waterloo, 2002.

[DR00] Roger W. Duke und Gordon Rose. *Formal object-oriented specification using Object-Z.* Macmillan, 2000.

[EB12] Steven D. Eppinger und Tyson R. Browning. *Design Structure Matrix Methods and Applications.* MIT Press, 2012.

[EBL06] Maged Elaasar, Lionel Briand, und Yvan Labiche. *A Metamodeling Approach to Pattern Specification.* In Model Driven Engineering Languages and Systems, number 4199 in LNCS, 484–498. Springer Berlin / Heidelberg, 2006.

[EFT92] Heinz-Dieter Ebbinghaus, Jörg Flum, und Wolfgang Thomas. *Einführung in die mathematische Logik.* BI-Wissenschaftsverlag, Mannheim; Leipzig; Wien; Zürich, 3., vollst. überarb. und erw. aufl.. Auflage, 1992.

[EKN+12] Gregor Engels, Marion Kremer, Thomas Nötzold, Thomas Wolf, Karl Prott, Jörg Hohwiller, Alexander Hofmann, Andreas Seidl, Diethelm Schlegel, und Oliver F. Nandico. *Quasar 3.0 - A Situational Approach to Software Engineering.* Capgemini sd&m, 2012. http://www.de.capgemini.com/resource-file-access/resource/pdf/quasar3.0.pdf.

[Eva04] Eric Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2004.

[EWKM10] L Efrizoni, WMN Wan-Kadir, und R Mohamad. *Formalization of UML class diagram using description logics.* In Information Technology (ITSim), 2010 International Symposium in, Band 3, 1168–1173. IEEE, 2010.

[FECA04] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, und Mehmet Akşit. *Aspect-Oriented Software Development.* Addison Wesley, 2004.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional, 2002.

[GAO94] David Garlan, Robert Allen, und John Ockerbloom. *EXploiting Style in Architectural Design Environments.* SIGSOFT Softw. Eng. Notes, 19(5): 175–188, 1994.

[GCB+06] Alessandro Garcia, Christina Chavez, Thais Batista, Claudio Sant'anna, Uirá Kulesza, Awais Rashid, und Carlos Lucena. *On the Modular Representation of Architectural Aspects.* In Volker Gruhn und Flavio Oquendo (Editoren), Software Architecture, number 4344 in LNCS, 82–97. Springer Berlin Heidelberg, 2006.

[GF+92] Michael R. Genesereth, Richard E. Fikes, et al. *Knowledge interchange format-version 3.0: Reference manual.* Technischer Bericht Logic-92-1, Computer Science Department, Stanford University, 1992.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

[GKRS12] Mahbouba Gharbi, Arne Koschel, Andreas Rausch, und Gernot Starke. *Basiswissen für Softwarearchitekten.* dpunkt.verlag, 2012.

[GS94] David Garlan und Mary Shaw. *An Introduction to Software Architecture.* Technischer Bericht CMU-CS-94-166, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1994.

[GY01] Daniel Gross und Eric Yu. *From Non-Functional Requirements to Design through Patterns.* Requirements Engineering, 6(1): 18–36, 2001.

[HA08] Neil B. Harrison und Paris Avgeriou. *Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation.* In Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA), 147–156, Washington, DC, USA, 2008. IEEE Computer Society.

[HB02] Ouafa Hachani und Daniel Bardou. *Using Aspect-Oriented Programming for Design Patterns Implementation.* In Workshop on Reuse in Object-Oriented Information Systems Design (OOIS), 2002.

[HB03] Ouafa Hachani und Daniel Bardou. *On Aspect-oriented Technology and Object-oriented Design Patterns.* In Workshop on Analysis of Aspect-Oriented Software, 2003.

[HB07] Petr Hnětynka und Tomáš Burěs. *Advanced Features of Hierarchical Component Models.* In Proceedings of the 10th International Conference on Information System Implementation and Modeling, 3–10, 2007.

[hello] hello2morrow. https://www.hello2morrow.com/products/sonarj.

[Her11] Sebastian Herold. *Architectural Compliance in Component-Based Systems. Foundations, Specification, and Checking of Architectural Rules.* Dissertation, TU Clausthal, 2011.

[HK02] Jan Hannemann und Gregor Kiczales. *Design pattern implementation in Java and AspectJ.* In Proceedings of OOPSLA '02, Band 37, 161–173. ACM, 2002.

[HMRS07] Sebastian Herold, Andreas Metzger, Andreas Rausch, und Heiko Stallbaum. *Towards Bridging the Gap between Goal-Oriented Requirements Engineering and Compositional Architecture Development.* In Proceedings of the 2nd Workshop on SHAring and Reusing architectural Knowledge - Architecture, Rationale, and

Design Intent (SHARK-ADI), Washington, DC, USA, 2007. IEEE Computer Society.

[HNS00] Christine Hofmeister, Robert Nord, und Dilip Soni. *Applied Software Architecture.* Addison-Wesley Professional, 2000.

[HR13] Sebastian Herold und Andreas Rausch. *Complementing Model-Driven Development for the Detection of Software Architecture Erosion.* In 5th International Workshop on Modeling in Software Engineering (MiSE), 24–30. IEEE, 2013.

[HR14] Sebastian Herold und Andreas Rausch. *A Rule-Based Approach to Architecture Conformance Checking as a Quality Management Measure.* In Relating System Quality and Software Architecture, 181–207. Elsevier / Morgan Kaufman, 2014.

[HRB+08] Sebastian Herold, Andreas Rausch, Alexander Bösl, Jan Ebell, Christian Linsmeier, und Detlef Peters. *A Seamless Modeling Approach for Service-Oriented Information Systems.* In Proceedings of the 5th International Conference on Information Technology: New Generations (ITNG), 438–446. IEEE Computer Society, 2008.

[JV03] Doug Janzen und Kris de Volder. *Navigating and Querying Code Without Getting Lost.* In Proceedings of the 2nd international conference on Aspect-oriented software development, 178–187. ACM, 2003.

[KA07] Ahmad Waqas Kamal und Paris Avgeriou. *An Evaluation of ADLs on Modeling Patterns for Software Architecture.* In Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering techniques (RISE), Lecture Notes in Computer Science. Springer, 2007.

[KA10] Ahmad Kamal und Paris Avgeriou. *Mining Relationships between the Participants of Architectural Patterns.* In Muhammad Babar und Ian Gorton (Editoren), Software Architecture, Band 6285 von Lecture Notes in Computer Science, 401–408. Springer Berlin / Heidelberg, 2010.

[KAZ08] Ahmad Waqas Kamal, Paris Avgeriou, und Uwe Zdun. *Modeling Architectural Patterns Variants.* In Proceedings of 13th European Conference on Pattern Languages of Programs (EuroPLoP), 1–23, Irsee, Germany, 2008.

[KB03] Saluka R Kodituwakku und Peter Bertok. *Pattern categories: a mathematical approach for organizing design patterns.* In Proceedings of the 2002 Conference on Pattern Languages of Programs, 63–73. Australian Computer Society, 2003.

[KC09] Soon-Kyeong Kim und David Carrington. *A Formalism to Describe Design Patterns Based on Role Concepts.* Formal Aspects of Computing, 21(5): 397–420, 2009.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, und William G. Griswold. *An Overview of AspectJ.* In Jørgen Lindskov Knudsen (Editor), ECOOP 2001 – Object-Oriented Programming, Band 2072 von Lecture Notes in Computer Science, 327–354. Springer Berlin Heidelberg, 2001.

[KM07] Tom King und Joe Marasco. *What Is the Cost of a Requirement Error?* http://www.stickyminds.com/article/what-cost-requirement-error, 2007.

[KMN11] Peter Kajsa, Lubomir Majtas, und Pavol Navrat. *Design Pattern Instantiation Directed by Concretization and Specialization.* Computer Science and Information Systems, 8(1): 41–72, 2011.

[KOS06] Philippe Kruchten, Henk Obbink, und Judith Stafford. *The Past, Present, and Future for Software Architecture.* IEEE Software, 23(2): 22–30, 2006.

[Kru95] Philippe Kruchten. *The 4+ 1 View Model of Architecture.* Software, IEEE, 12(6): 42–50, 1995.

[Kru99] Philippe Kruchten. *The Software Architect, and the Software Architecture Team.* In Patrick Donohoe (Editor), Software Architecture, 565–583. Kluwer Academic Publishers, Boston, 1999.

[KWB03] Anneke Kleppe, Jos Warmer, und Wim Bast. *MDA Explained - The Model Driven Architecture: Practice and Promise.* Addison-Wesley Professional, Boston, 2003.

[KWB05] Anneke G. Kleppe, Jos Warmer, und Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Professional, 2005.

[LM08] Mikael Lindvall und Dirk Muthig. *Bridging the Software Architecture Gap.* Computer, 41(6): 98–101, 2008.

[MB01] Ruth Malan und Dana Bredemeyer. *Software Architecture: Central Concerns, Key Decisions.* Bredemeyer Consulting White Paper 4/28/05, 2001.

[Men00] Kim Mens. *Automating Architectural Conformance Checking by Means of Logic Meta Programming.* Dissertation, Vrije Universiteit Brussel, 2000.

[Mic09] Microsoft Patterns & Practices Team. *Microsoft Application Architecture Guide.* Microsoft Press, 2. Auflage, 2009.

[Mik98] Tommi Mikkonen. *Formalizing Design Patterns.* In Proceedings of the 20th International Conference on Software Engineering, 115–124, 1998.

[MNS95] Gail C. Murphy, David Notkin, und Kevin Sullivan. *Software Reflexion Models: Bridging the Gap Between Source and High-level Models.* ACM SIGSOFT Software Engineering Notes, 20(4): 18–28, 1995.

[Mon99] Robert T. Monroe. *Rapid Development of Custom Software Architecture Design Environments.* Dissertation, no. cmu-cs-99-161, Carnegie Mellon University, 1999.

[MT00] Nenad Medvidovic und Richard N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages.* IEEE Transactions on Software Engineering, 26(1): 70–93, 2000.

[Obj05] Object Management Group (OMG). *UML® superstructure specification, Version 2.0*, 2005.

[Obj11] Object Management Group (OMG). *UML® superstructure specification, Version 2.4*, 2011.

[Obj14a] Object Management Group (OMG). *MDA Guide, Version 2.0*, 2014.

[Obj14b] Object Management Group (OMG). *Meta Object Facility (MOF®) Core Specification Version 2.4.2*, 2014.

[Obj14c] Object Management Group (OMG). *Object Constraint Language (OCL), Version 2.4*, 2014.

[Obj14d] Object Management Group (OMG). *XML Metadata Interchange (XMI)®, Version 2.4.2*, 2014.

[Ora13] Oracle. *Java™ Platform, Standard Edition 7, API Specification*, 2013.

[PLm] University of Southern California. *PowerLoom®Knowledge Representation & Reasoning System.* http://www.isi.edu/isd/LOOM/PowerLoom.

[Par72] David L. Parnas. *On the Criteria to Be Used in Decomposing Systems into Modules.* Communications of the ACM, 15(12): 1053–1058, 1972.

[PFT11] Mónica Pinto, Lidia Fuentes, und José María Troya. *Specifying aspect-oriented architectures in AO-ADL.* Information and Software Technology, 53(11): 1165–1182, 2011.

[PW92] Dewayne E. Perry und Alexander L. Wolf. *Foundations for the study of software architecture.* ACM SIGSOFT Software Engineering Notes, 17(4): 40–52, 1992.

[PZ03] Eduardo Kessler Piveta und Luis Carlos Zancanella. *Observer Pattern using Aspect-Oriented Programming.* In 3nd Latin American Conference on Pattern Languages of Programming (SugarLoafPLOP), 2003.

[RH06] Ralf Reussner und Wilhelm Hasselbring (Editoren). *Handbuch der Software-Architektur.* dpunkt Verlag, 1. Auflage, 2006.

[Rie97] Dirk Riehle. *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose.* Technischer Bericht 97-1-1, Ubilab, Union Bank of Switzerland, Zürich, 1997.

[Rie00] Dirk Riehle. *Framework Design: A Role Modeling Approach.* Diessertation, No. 13509, ETH Zürich, Zürich, Switzerland, 2000.

[Ris00] Linda Rising. *The Pattern Almanac 2000.* Addison-Wesley Longman, Amsterdam, 2000.

[SAB10] Klaas-Jan Stol, Paris Avgeriou, und Muhammad Ali Babar. *Identifying Architectural Patterns Used in Open Source Software: Approaches and Challenges.* In Proceedings of the 14th International Conference on Evaluation and Assessment of Software Engineering (EASE), Keele University, UK, 2010.

[SAVE] SAVE (Software Architecture Visualisation and Evaluation) Projekt Webseite. http://www.fc-md.umd.edu/save.

[SBPM08] David Steinberg, Frank Budinsky, Marcelo Paternostro, und Ed Merks. *EMF: Eclipse Modeling Framework.* Addison Wesley, Upper Saddle River, NJ, 2008.

[Sch00] Uwe Schöning. *Logik für Informatiker.* Spektrum Akademischer Verlag, Heidelberg, Berlin, 5. Auflage, 2000.

[Sha96] Mary Shaw. *Some Patterns for Software Architectures.* In John M. Vlissides, James O. Coplien, und Norman L. Kerth (Editoren), Pattern Languages of Program Design, Band 2, 255–269. Addison-Wesley, 1996.

[SEI] Software Engineering Institute, Carnegie Mellon Universtiy. *Community Software Architecture Definitions.* http://www.sei.cmu.edu/architecture/start/glossary/community.cfm.

[Som10] Ian Sommerville. *Software Engineering.* Addison-Wesley, Boston, 2010.

[Spi92] J Michael Spivey. *The Z notation.* Prentice Hall, 1992.

[SVEH07] Thomas Stahl, Markus Völter, Sven Efftinge, und Arno Haase. *Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management.* dpunkt.verlag, 2. Auflage, 2007.

[SZ08] Lijun Shan und Hong Zhu. *A Formal Descriptive Semantics of UML.* In Formal Methods and Software Engineering, 375–396. Springer, 2008.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-oriented Programming.* Pearson Education, 2002.

[Tan03] Andrew S. Tanenbaum. *Computernetzwerke.* Pearson Studium, 4. Auflage, 2003.

[Tou07] Toufik Taibi. *An Integrated Approach to Design Patterns Formalization.* In Toufik Taibi (Editor), Design Pattern Formalization Techniques, 1–19. IGI Global, 2007.

[TS06] Andrew S. Tanenbaum und Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition).* Prentice Hall, 2006.

[TV09] Ricardo Terra und Marco Tulio Valente. *A Dependency Constraint Language to Manage Object-oriented Software Architectures.* Software: Practice and Experience, 39(12): 1073–1094, 2009.

[Vol06] Kris de Volder. *JQuery: a Generic Code Browser With a Declarative Configuration Language.* In Practical Aspects of Declarative Languages, 88–102. Springer, 2006.

[Wes02] J. Christopher Westland. *The Cost of Errors in Software Development: Evidence from Industry.* Journal of Systems and Software, 62(1): 1–9, 2002.

[WK04] Jos Warmer und Anneke Kleppe. *Object Constraint Language 2.0.* mitp-Verlag, 2004.

[ZA08] Uwe Zdun und Paris Avgeriou. *A Catalog of Architectural Primitives for Modeling Architectural Patterns.* Information and Software Technology, 50(9-10): 1003–1034, 2008.

[ZBS11] Abdelhafid Zitouni, Mahmoud Boufaida, und Lionel Seinturier. *A Contract-Based Language to Specify Design Components.* International Journal of Software Engineering and Its Applications, 5(3): 71–88, 2011.

[ZC09] Michael Zhivich und Robert K. Cunningham. *The Real Cost of Software Errors.* Security & Privacy, IEEE, 7(2): 87–90, 2009.

[Zim95] Walter Zimmer. *Relationships between design patterns.* Pattern languages of program design, 1: 345–364, 1995.

[Zit12] Abdelhafid Zitouni. *Rigorous Description Of Design Components Functionality: An Approach Based Contract.* International Journal of Computer Science Issues, 9(3): 187–196, 2012.

[ZS06] Hong Zhu und Lijun Shan. *Well-formedness, Consistency and Completeness of Graphic Models.* Proceedings of UKSIM, 47–53, 2006.

[ZZGL08] Olaf Zimmermann, Uwe Zdun, Thomas Gschwind, und Frank Leymann. *Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method.* In Proceedings of 7th Working IEEE/IFIP Conference on Software Architecture (WICSA), 157–166, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

Abbildungsverzeichnis

Die Architektur eines Softwaresystems legt die grundlegenden Eigenschaften, Konzepte und Strukturen des Systems fest. Beim Entwurf der Softwarearchitektur wird auf wiederverwendbare Architekturkonzepte, wie z.B. Muster oder Referenzarchitekturen, zurückgegriffen. Diese in dieser Arbeit als Architekturbausteine bezeichneten Architekturkonzepte beschreiben bewährte Lösungskonzepte für verschiedene Entwurfsprobleme.
Architekturbausteine stellen an ihre Anwendung Bedingungen, die über die von ihnen definierten Richtlinien hinsichtlich Struktur und Verhalten hinausgehen. Eine Softwarearchitektur ist nur dann konsistent zu den auf sie angewendeten Architekturbausteinen, wenn deren Bedingungen von der Architektur eingehalten werden. Um die Qualität der Softwarearchitektur zu sichern, müssen die Bedingungen nicht nur bei der Anwendung an sich, sondern über die vollständige Entwicklungszeit hinweg gültig sein. Mit der Größe und Komplexität eines Softwaresystems steigt potentiell auch die Anzahl der angewendeten Architekturbausteine und das Risiko für die Verletzung ihrer Bedingungen. Für den bausteinbasierten Architekturentwurf ist daher ein flexibler Mechanismus erforderlich, der die Anwendung von Bausteinen mit der Überwachung ihrer Bedingungen kombiniert.
In dieser Arbeit wird daher ein umfassendes Konzept erarbeitet, das Softwarearchitekten bei der Erstellung von großen und komplexen Softwaresystemen basierend auf Architekturbausteinen unterstützten soll. Grundlage bildet dabei eine Beschreibungssprache für bausteinbasierte Softwarearchitekturen sowie für Architekturbausteine und ihre Bedingungen. Darauf aufbauend wird ein Mechanismus zur Anwendung von Architekturbausteinen definiert, der die Einhaltung der Bedingungen sicherstellt. Für die Beschreibung von Softwarearchitekturen und Architekturbausteinen werden zwei voneinander unabhängige Teilmetamodelle definiert, die durch einen Integrationsteil zum Nachvollziehen der Bausteinanwendungen verbunden werden. Modelle dieses Metamodells werden sowohl durch eine semiformale Sprache beschrieben als auch durch prädikatenlogische Strukturen formal repräsentiert. Dazu passend werden die Bedingungen der Architekturbausteine durch prädikatenlogische Sätze formuliert. Das erarbeitete Konzept für den bausteinbasierten Architekturentwurf wird anhand eines Beispielsystems ausgewertet. Hierfür werden die Architekturbausteine, die in diesem Beispielsystem zur Anwendung kommen, entsprechend der entwickelten Sprache beschrieben und formal repräsentiert. Zugleich werden Bedingungen für ihre konsistente Anwendung festgelegt. Ergänzend wird ein Entwurf für eine Werkzeugunterstützung des bausteinbasierten Architekturentwurfs entwickelt.
Die Anwendung auf das Beispielsystem hat gezeigt, dass der in dieser Arbeit entwickelte bausteinbasierte Architekturentwurf praktisch einsetzbar ist. Dabei veranschaulicht die Formalisierung der unterschiedlichen Architekturbausteine und ihrer Bedingungen die Flexibilität und Fähigkeit des Ansatzes, verschiedene Architekturkonzepte beschreiben zu können. Zudem demonstriert das Beispiel die Eignung des Ansatzes, die Einhaltung der Bedingungen über die vollständige Entwicklungszeit hinweg zu überwachen. Insgesamt bildet diese Arbeit damit die konzeptionelle Grundlage für eine flexible Werkzeugunterstützung zur systematischen Anwendung von wiederverwendbaren Architekturkonzepten.