Masterarbeit

Eike-Christian Ramcke

Universelle Modellanalyse von Anwendungslandschaften basierend auf dem ArchiMate Standard

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Technik und Informatik
Department Informatik

Hamburg University of Applied Sciences
Faculty of Computer Science and Engineering
Department of Computer Science

Eike-Christian Ramcke

Universelle Modellanalyse von Anwendungslandschaften basierend auf dem ArchiMate Standard

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Ulrike Steffens
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 06. Dezember 2018

Eike-Christian Ramcke

Thema der Arbeit

Universelle Modellanalyse von Anwendungslandschaften basierend auf dem ArchiMate Standard

Stichworte

Anwendungslandschaft, Enterprise Architecture, ArchiMate, model analysis

Kurzzusammenfassung

Anwendungslandschaften von Unternehmen sind historisch gewachsen, durch Abhängigkeiten vieler Anwendungen verflochten, heterogen und komplex. Im Rahmen von Enterprise Architecture (EA) werden Modelle genutzt, um unternehmerische oder technische Entscheidungen zu unterstützen und Aussagen über Zustände und Systemeigenschaften in Anwendungslandschaften zu bestimmen bzw. vorherzusagen.

Diese Masterthesis adaptiert die EA-Modellierungssprache *Multi-Attribute Prediction (MAP)*, die an der KTH Stockholm im Department *Industrial Information and Control Systems* entwickelt wurde, und überträgt die Analyse-Algorithmen auf den ArchiMate Standard. Mit *MAP* können Wahrscheinlichkeiten über Qualitätseigenschaften wie Anwendungsgröße, Datengenauigkeit, Kopplung, Verfügbarkeit und Kosten über die Struktur eines Modells berechnet werden. Die Ergebnisse sind zum Abschluss der Berechnung als Eigenschaften auf Knoten im Modell sichtbar.

Analyse Your Enterprise (AYE) ist eine Web-Anwendung zur Modellierung und Berechnung von ArchiMate-Modellen, die im Zuge dieser Arbeit entwickelt wurde. *AYE* kann ArchiMate-Modelle auch ohne weitere manuelle Angaben analysieren. Die in *AYE* implementierten Algorithmen zur Analyse von Verfügbarkeit, Kopplung und Kosten lösen Zyklen auf, erlauben es die Analysen zu erweitern oder einzuschränken und können mit der großen Anzahl an semantischen Knoten und Kanten aus ArchiMate umgehen. Neben den aus *MAP* übertragenen Analysen sollen zukünftig weitere hinzu kommen.

Eike-Christian Ramcke

Title of Thesis

Universal model analysis of application landscapes based on the ArchiMate standard

Keywords

Applications landscape, Enterprise Architecture, ArchiMate, model analysis

Abstract

Application landscapes are grown historically. They are interlaced with many dependencies, are heterogeneously and complex. Enterprise Architecture (EA) models are used for business and technical decisions. Moreover, they determine or predict system states and attributes in application landscapes.

This master thesis adapts the EA-model-language *Multi-Attribute Prediction (MAP)*, developed by KTH Stockholm at the Department *Industrial Information and Control Systems* and maps the analyse-algorithms to the ArchiMate standard. *MAP* features automate analysis of quality attributes from the model structure in five areas: application size, data accuracy, coupling, availability and costs. The results are shown as probability values in the model elements.

Analyse Your Enterprise (AYE) is a web-application for modelling and analysing ArchiMates models, which was developed within this thesis. *AYE* can compute the quality attributes without any manual inputs. The algorithms developed in *AYE* for availability, coupling and costs resolve cycles in the model structure. They also leave the opportunity to extend or restrict this analysis and they allow to use a huge amount of semantic elements and relationships from ArchiMate. In the future *AYE* will be developed further and additional analyse-algorithms could be added.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen

ADM Architecture Development Method.

AYE Analyse Your Enterprise.

BPMN Business Process Model and Notation.

CORS Cross-Origin Resource Sharing.

EA Enterprise Architecture.

EAAT Enterprise Architecture Analysis Tool.

EAM Enterprise Architecture Management.

ERM Entity-Relationship-Modell.

HAW Hochschule für Angewandte Wissenschaften.

HAWAI HAW Laboratory for Architecture and IT-Management.

ICC Informatik Compute Cloud.

JWT JSON Web Token.

MAP Multi-Attribute Prediction.

MAPL Multi-Attribute Prediction Language.

MOM Message Oriented Middleware.

ORM Object Relational Mapping.

UML Unified Modeling Language.

Danksagung

An dieser Stelle möchte ich mich zunächst bei Frau Prof. Dr. Ulrike Steffens und bei Herrn Prof. Dr. Olaf Zukunft für die Betreuung und Unterstützung während der Erstellung dieser Arbeit bedanken.

Weiterhin möchte ich mich insbesondere bei Herrn Prof. Dr. Mathias Lövehagen Ekstedt von der KTH Stockholm für die Möglichkeit der Forschung am Department *Industrial Information and Control Systems* bedanken.

Ein besonderer Dank gilt meiner Lebensgefährtin Louisa Spahl, die mich seelisch unterstützte und gerade im Endspurt auf viel gemeinsame Zeit verzichten musste.

Ebenfalls geht ein großes Dankeschön an meine Eltern, die mich in jeglicher Hinsicht unterstützten.

1 Einleitung

Heutige Unternehmen sind im hohen Maße abhängig von ihrer eingesetzten Software [MKG04]. In den letzten Jahrzehnten hat sich die Anzahl der Systeme in Unternehmen erhöht und Anwendungen sind stärker miteinander vernetzt. Solche Anwendungslandschaften können aus vielen hundert oder gar tausend Systemen bestehen, die eng miteinander gekoppelt sind. Der Einsatz unterschiedlicher Technologien aus mehreren Jahrzehnten führt zu hoher Heterogenität. Mangelnde zentrale Steuerung von Systemen führte zu Software in verschiedenen Abteilungen, die den gleichen Zweck erfüllen. Das Management von Softwaresystemen und ihrer Umgebung ist zu einem komplexen Geschäft geworden [Eng+08].

1.1 Motivation

Der Einsatz von *EA* unterstützt Prinzipien, Methoden und Modelle um eine ganzheitliche Sicht auf Design und Realisierung von organisatorischen Strukturen, Geschäftsprozessen, Informationssystemen und Infrastruktur zu erhalten [Lan17]. Dabei geht EA nicht ins Detail von kleinteiligen Lösungen in einzelnen Domänen, sondern fokussiert sich auf die wesentlichen Prozesse im Unternehmen, in der IT und ihrer Weiterentwicklung [FAW07]. Die ganzheitliche Sicht vereint alle Domänen und Stakeholder mit unterschiedlichen Schwerpunkten und Sichten auf die Architektur.

Der Einsatz von Modellen, Sichten, Visualisierungen und Analysen soll bei der Zusammenbringung und Kommunikation verschiedener Domänen im Unternehmen helfen. Ein Verständnis über die Unternehmensarchitektur kann unternehmerische und technische Entscheidungen unterstützen, indem u.a. strategische Ziele und Anforderungen transparent und überprüfbar werden. Inzwischen gibt es diverse Tools, Frameworks und Modellierungssprachen für das Enterprise Architecture Management (EAM) [RZM14].

Diese Arbeit beschäftigt sich mit der Analyse in einer frühen Planungsphase für Soll-Unternehmensarchitekturen. Das Ziel ist die Vorhersage ausgewählter Eigenschaften von zukünftigen Systemen im Kontext dieser Architekturen, die als Modell beschrieben sind. Die Modellanalyse soll Auswirkungen von Veränderungen aufdecken, bevor diese in Auftrag gegeben werden.

Die akademische EA-Modellierungssprache *MAP* ist ein Versuch, System- und Qualitätseigenschaften über strukturelle Informationen und Attribute auf Knoten zu berechnen, um Schwachstellen und Probleme möglichst früh aufzudecken. *MAP* orientiert sich an ArchiMate, einer populären und durch die *Open Group* standardisierte EA-Modellierungssprache [Ope17]. ArchiMate ist durch die syntaktische Nähe zu *Unified Modeling Language (UML)* in der Praxis beliebt und wird von vielen Tools für den unternehmerischen Einsatz unterstützt [RZM14].

1.2 Zielsetzung

Im Rahmen dieser Masterarbeit soll das Konzept der Modellierungssprache *MAP* für den praktischen Einsatz tauglich gemacht werden. Dazu sollen die Analysen aus *MAP* in einem dazu entwickelten Tool (namentlich *Analyse Your Enterprise* oder kurz *AYE*) zur direkten Analyse ohne Umwege mit ArchiMate-Modellen ausführbar werden. Ohne ein solches Tool müsste der Anwender das bestehende ArchiMate-Modell einer Unternehmensarchitektur für *MAP* neu modellieren, da *MAP* auf einem eigenen konzeptionellen und speziell für die enthaltenen Analysemechanismen optimiertes Metamodell basiert.

ArchiMate besteht im Gegensatz zu *MAP* aus wesentlich mehr Ebenen, Elementen und Beziehungen. Die möglichen Strukturen eines Modells aus *MAP* sind optimiert für die verwendeten Analysen, daher lassen sich diverse Semantiken nicht in *MAP* modellieren und somit auch nicht analysieren. Diese zusätzlichen Möglichkeiten der Modellierung aus ArchiMate soll *AYE* für die eigenen Analysen nutzen um noch brauchbarere Ergebnisse zu liefern.

Diese Arbeit beschränkt sich in der Betrachtung auf drei Analysen: Kopplung, Verfügbarkeit und Kosten, die sich an dem Vorbild *MAP* orientieren. Das Tool *AYE* soll erweiterbar gestaltet werden, um die Integration weiterer Analysen zu vereinfachen.

1.3 Aufbau der Arbeit

Diese Arbeit ist in fünf Kapitel gegliedert. Im Anschluss an diese Einführung, in Kapitel *2 Grundlagen*, werden die Modellierungssprachen ArchiMate und *MAP* erklärt und deren Unterschiede, Analysemechanismen sowie Strukturen genauer beschrieben.

Kapitel *3 Projekt* geht auf Design und Umsetzung der Anwendung *AYE* ein, die im Rahmen dieser Masterarbeit entwickelt wurde.

Die in *AYE* entworfenen und verwendeten Algorithmen zur Analyse von Kopplung, Verfügbarkeit und Kosten werden in Kapitel *4 Modellanalyse* vorgestellt.

Kapitel *5*, *Zusammenfassung und Fazit*, fasst die Inhalte dieser Arbeit zusammen und bewertet das Ergebnis. Der Ausblick erläutert Erweiterungsmöglichkeiten und schließt die Arbeit ab.

2 Grundlagen

Die Grundlagen befassen sich zunächst mit der zugrundeliegenden Modellierungssprache *ArchiMate* und gehen anschließend auf die Möglichkeiten zur Modellanalyse ein. Das Unterkapitel Modellanalyse legt den Fokus auf die Modellsprache *MAP*, dessen Sprachmodell gewisse Ähnlichkeiten mit *ArchiMate* besitzt und verschiedene Analysemechanismen beinhaltet.

2.1 ArchiMate

ArchiMate [Ope17] ist eine von der *Open Group* standardisierte Modellierungssprache für Unternehmensarchitekturen. *ArchiMate* orientiert sich an der Syntax des UML Komponentendiagramms und geht dabei einen Schritt weiter und ergänzt das Element *Komponente* um weitere semantische Elementtypen, die sich vom Stereotyp und symbolisch unterscheiden. Gegliedert sind die Knoten und Beziehungen auf Ebenen und in Aspekten, wie Abbildung 2.1 zeigt.

2.1.1 Ebenen

Den *ArchiMate Core* bilden die Ebenen *Business Layer*, *Application Layer* und *Technology Layer*. Alle weiteren Ebenen sind als Erweiterungen zu *ArchiMate* hinzugekommen und erst seit Version 3.x im Framework fest verankert.

Der *Business Layer* konzentriert sich auf organisatorische Dienste, die durch Geschäftsprozesse und Personen realisiert werden. Der *Application Layer* unterstützt den *Business Layer* durch Applikationen, Dienste und Software. Der *Technology Layer* modelliert Speicher, Netzwerke, Hardware, Betriebssystem etc.

ArchiMate Full ergänzt vier weitere Ebenen. Senkrecht zu den übrigen Ebenen erstreckt sich der *Motivation Layer*. Mit diesem können Beweggründe für Designentscheidungen

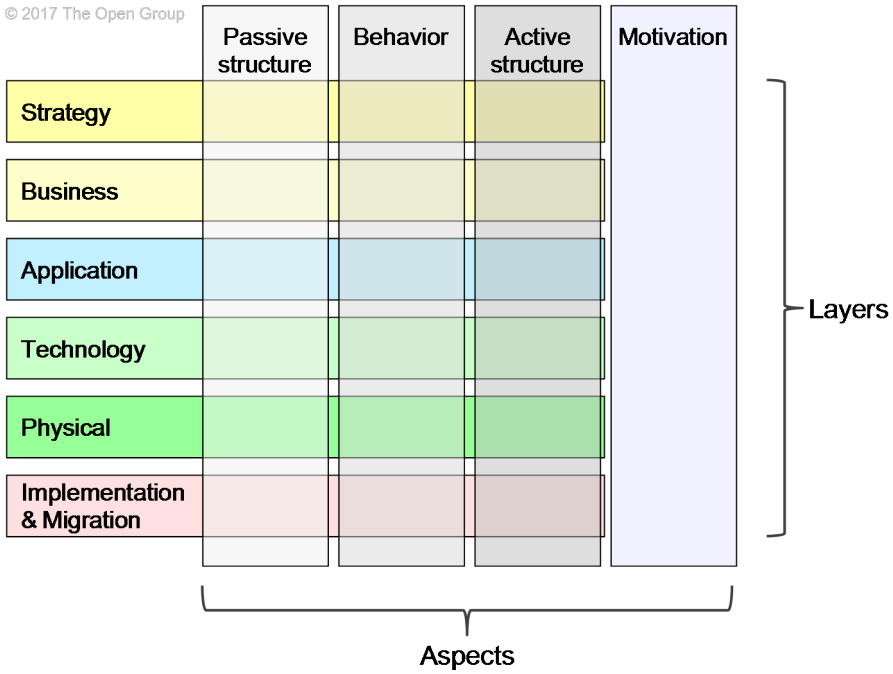


Abbildung 2.1: *Ebenen in ArchiMate Full* [Ope17]

oder Gründe für Veränderungen an der Unternehmensarchitektur modelliert werden. Der *Strategy Layer* organisiert die abstrakte Einteilung von Ressourcen und Capabilities. Der *Physical Layer* wird eng verzahnt mit dem *Technology Layer* und erweitert diesen um physikalische Elemente wie Equipment, Fabrik, verteilte Netzwerke und Material. Der *Implementation and Migration Layer* ergänzt Elemente, um das Vorgehen zur Veränderung der Unternehmensarchitektur zu modellieren (zum Beispiel Arbeitspakete oder Arbeitsergebnisse).

2.1.2 Aspekte

ArchiMate unterteilt alle Elemente in Aspekte. Der *Active Structure* Aspekt repräsentiert aktive strukturelle Elemente wie beispielsweise Mitarbeiter, Komponenten einer Anwendung oder Geräte - somit einen aktiven Gegenstand (Nomen). Der *Behavior*-Aspekt enthält Elemente mit Verhalten bzw. Funktionalität, zum Beispiel Geschäftsprozesse, Funktionen, Events oder Dienste - Elemente, die Veränderungen herbeiführen (Verben). Der *Passive Structure* Aspekt enthält Objekte, auf denen die Veränderungen (*Behavior*) ausgeführt werden, wie beispielsweise Dokumente, Datenobjekte oder physikalische Gegenstände.

2.1.3 Elemente

Abbildung 2.2 stellt alle im *ArchiMate Full Framework* enthaltenen Elemente dar. Die angedeuteten Spalten auf den Ebenen unterteilen die Aspekte (*Active Structure*, *Behavior* und *Passive Structure*).

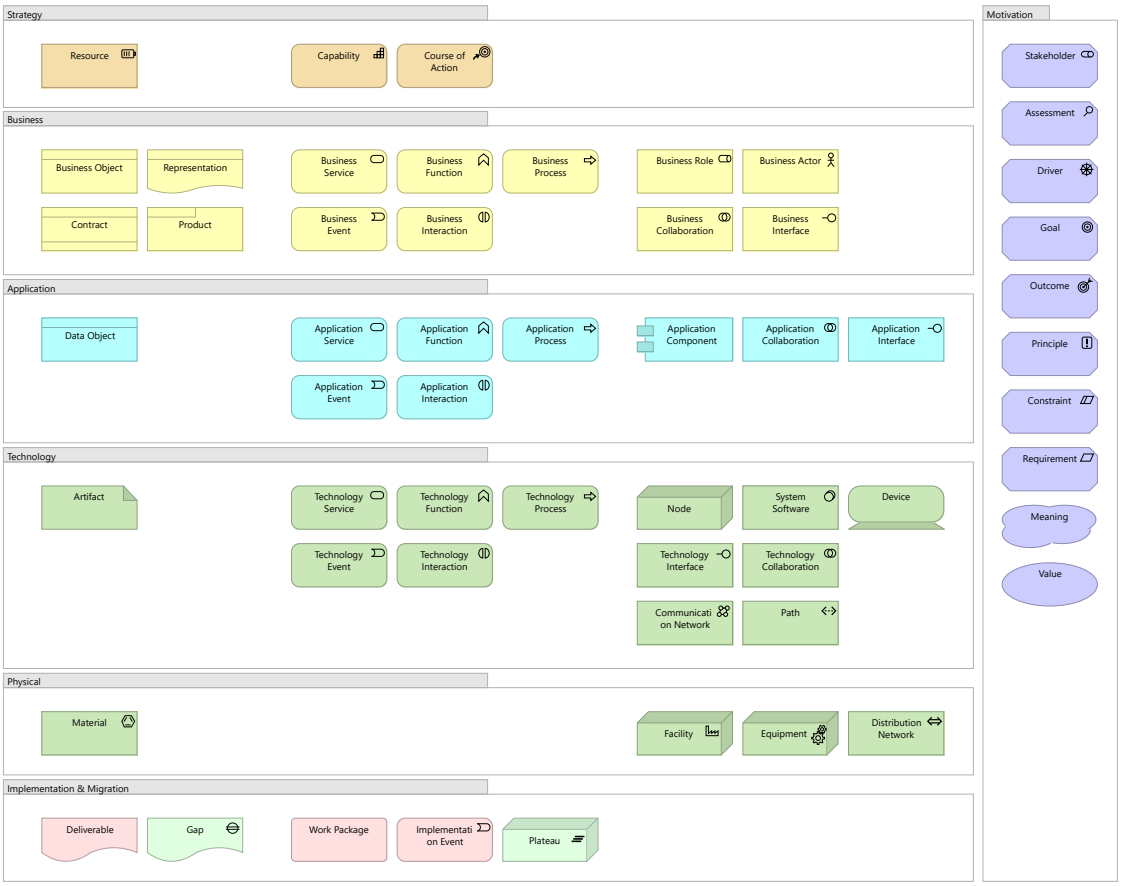


Abbildung 2.2: *Elemente in ArchiMate Full*

2.1.4 Beziehungen

ArchiMate versucht einen Zusammenhalt zwischen verschiedenen Domänen und Sichten einer Unternehmensarchitektur auszudrücken. Es wurden dafür bekannte Konzeptionen aus anderen Modellierungssprachen entsprechender Domäne übernommen. Beziehungen wie *Composition*, *Association* und *Specialization* stammen aus *UML*. *Triggering* ist typisch bei Modellen zur Darstellung von Geschäftsprozessen wie beispielsweise *Business*

Process Model and Notation (BPMN).

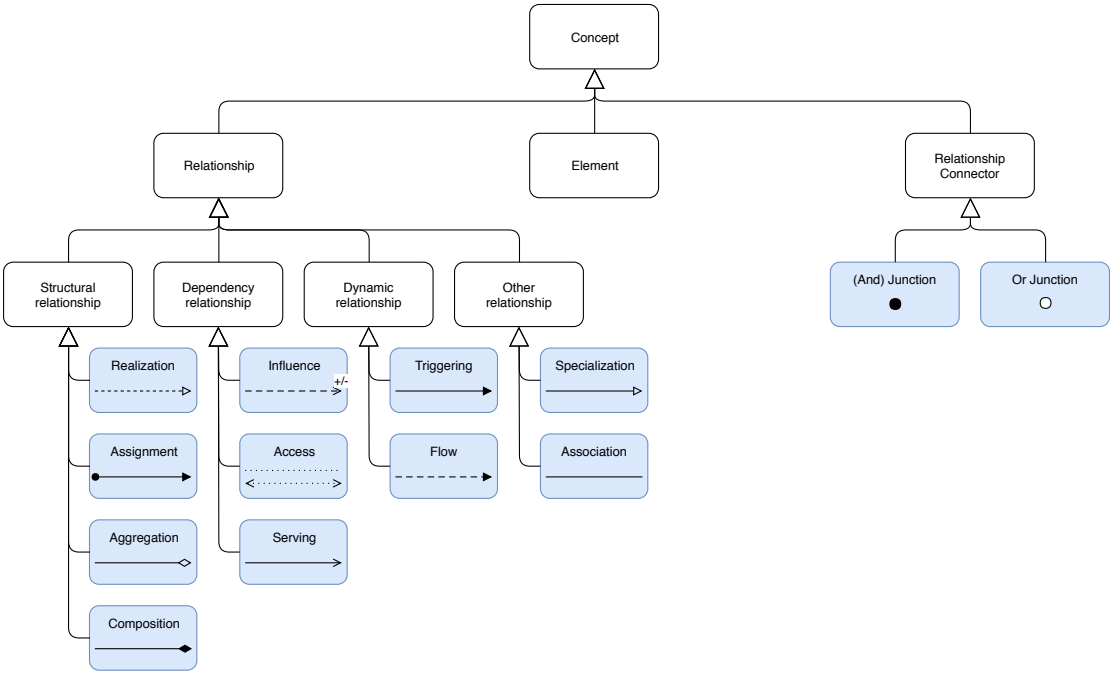


Abbildung 2.3: *Beziehungen in ArchiMate*

Abbildung 2.3 gibt einen Überblick über alle Beziehungen im Gesamtkonzept von *ArchiMate*. Relationen unterteilen sich, wie hier dargestellt, in vier Kategorien: *Structural*, *Dependency*, *Dynamic* und *Other* Relationship.

Dependency- und *Structural Relationships* besitzen eine geordnete Liste zum Ausdruck von Stärke im Zusammenhalt (siehe Abbildung 2.4). *Influence* stellt dabei den schwächsten Zusammenhalt zwischen Elementen dar und *Composition* den stärksten.

Anhand dieses geordneten Zusammenhalts können Beziehungen abgeleitet werden: Zwei *Dependency-* oder *Structural Relationships* $r : R$ und $s : S$ sind zulässige Beziehungen zwischen Elementen a, b und c indem $r(a, b)$ und $s(b, c)$ in Relation stehen. Damit leitet sich eine *Structural Relationship* $t : T$ mit $t(a, c)$ ab, die den Typ T der schwächsten Beziehung von R und S erhält.

Auch für *Dynamic Relationships* gibt es Ableitungen:

Flow-Beziehung:

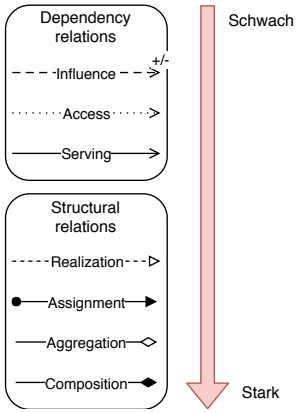


Abbildung 2.4: *Stärke von Beziehungen in ArchiMate*

- Gibt es eine *Flow*-Beziehung r von Element a zu Element b und eine *Structural Relationship* von Element c zu Element a, dann leitet sich daraus eine *Flow*-Beziehung r von Element c zu Element b ab.
- Gibt es eine *Flow*-Beziehung r von Element a zu Element b und eine *Structural Relationship* von Element d zu Element b, dann leitet sich daraus eine *Flow*-Beziehung r von Element a zu Element d ab.

Triggering-Beziehung:

- Gibt es eine *Triggering*-Beziehung r von Element a zu Element b und eine *Assignment*-Beziehung von Element c zu Element a, dann leitet sich daraus eine *Triggering*-Beziehung r von Element c zu Element b ab.
- Gibt es eine *Triggering*-Beziehung r von Element a zu Element b und eine *Assignment*-Beziehung von Element d zu Element b, dann leitet sich daraus eine *Triggering*-Beziehung r von Element a zu Element d ab.
- Gibt es eine *Triggering*-Beziehung r von Element a zu Element b und eine *Triggering*-Beziehung von Element b zu Element c, dann leitet sich daraus eine *Triggering*-Beziehung r von Element a zu Element c ab (*Transitivität*).

2.1.5 Eigenschaften

ArchiMate spezifiziert im *Exchange File Format* [Ope16] für jedes Element und jede Beziehung eine Liste aus Key-Value Paaren, um Eigenschaften zu hinterlegen. Dies er-

möglicht den generischen Austausch von Daten über verschiedene Tools hinweg, die das *Exchange File Format* unterstützen. Über diesen Weg können außerdem modellbasierte Analysen realisiert werden [Lan17], die auch *AYE* nutzt. ArchiMate selbst besitzt keinen spezifizierten Mechanismus für Analysen [När12].

Der Datentyp jeder Eigenschaft kann angegeben werden: *String*, *Boolean*, *Currency*, *Date*, *Time* und *Number*. Strukturierte Datentypen sind im *Exchange File Format* nicht zulässig und müssen in eine entsprechende Repräsentation transformiert werden.

2.1.6 Metamodell

ArchiMate basiert auf einem konzeptionellen Metamodell, das eine Struktur bei der Verknüpfung von Elementen und Relationen vorgibt. Damit ist es nicht möglich jede Elementart über eine beliebige Beziehung mit einem beliebigen weiteren Element zu verknüpfen.

Jede Ebene besitzt in ArchiMate letztlich ihr eigenes Metamodell, basierend auf einem generischen Metamodell-Konzept (Meta-Metamodell). Ebenen untereinander werden über erlaubte *Cross-Layer Dependencies* beschrieben. Die Metamodelle jeder Ebene plus die *Cross-Layer Dependencies* ergeben zusammen das umfassende Metamodell von ArchiMate.

Abbildung 2.5 zeigt als Beispiel das Metamodell des *Application Layers*. Die Elemente und Beziehungen sind abstrakt und nicht instanziiert, somit als Generalisierung zu verstehen. Die Beziehungen dürfen in einem konkreten und instanziierten Modell nur die Arten von Beziehungen zwischen den Elementarten im Metamodell nutzen.

Das Metamodell des *Application Layers* (Abbildung 2.5) stellt zur besseren Übersicht nicht alle Relationen dar: *Association*-Beziehungen dürfen zwischen jedem Element uneingeschränkt verwendet werden. Außerdem darf jedes Element zu einem Element gleichen Typs eine *Composition*-, *Aggregation*- und *Specialization*-Beziehung zuordnen. Auch abgeleitete Beziehungen (wie unter *2.1.4 Beziehungen* beschrieben) dürfen zusätzlich verwendet werden.

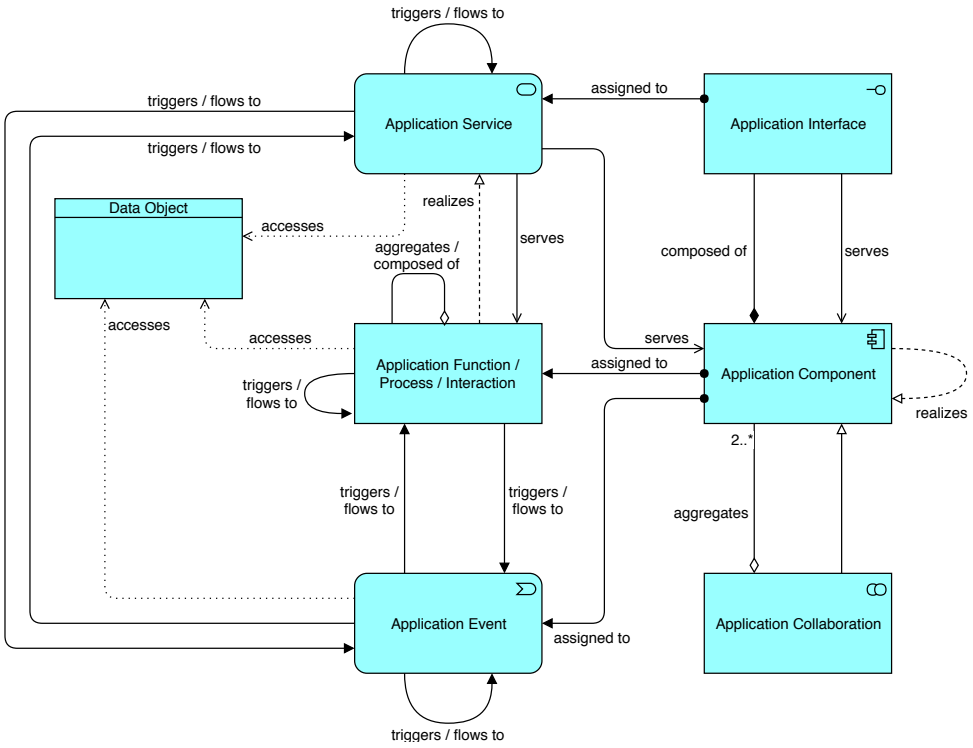


Abbildung 2.5: *Application Layer Metamodel*

2.1.7 Sichten

ArchiMate legt den Fokus auf eine ganzheitliche Struktur einer Unternehmensarchitektur. Modellierte Elemente und Beziehungen sind daher ganzheitlich definiert. Durch Sichten (*Views*) lassen sich Ausschnitte aus ausgewählten Elementen und Beziehungen getrennt betrachten. Sichten können zum Beispiel bestimmte Tätigkeitsfelder von Stakeholdern beschreiben oder im Architekturdesign Resultate aus formellen Interviews darstellen oder Geschäftsprozesse getrennt betrachten, etc.

Mit Hilfe von *Viewpoints* lassen sich in *ArchiMate* Sichten klassifizieren. Eine Klassifikation besteht aus einer festgelegten Menge an Elementen aus dem Metamodell und einem Namen des *Viewpoints*. *ArchiMate* gibt bereits einige *Viewpoints* vor, die optional benutzt werden können.

2.1.8 TOGAF

Die *Open Group* entwickelte neben ArchiMate auch das EA-Framework *TOGAF* [Har11]. Wie im Bereich EA üblich [BBL12], unterteilt *TOGAF* die Unternehmensarchitektur in drei Ebenen: Geschäftsarchitektur, Informationssystemarchitektur und Technologiearchitektur, die sich um weitere Ebenen erweitern lassen. Dabei bietet *TOGAF* mit *Architecture Development Method (ADM)* ein iteratives Vorgehen zur Modellierung der Architektur an, das in Abbildung 2.6 dargestellt ist.

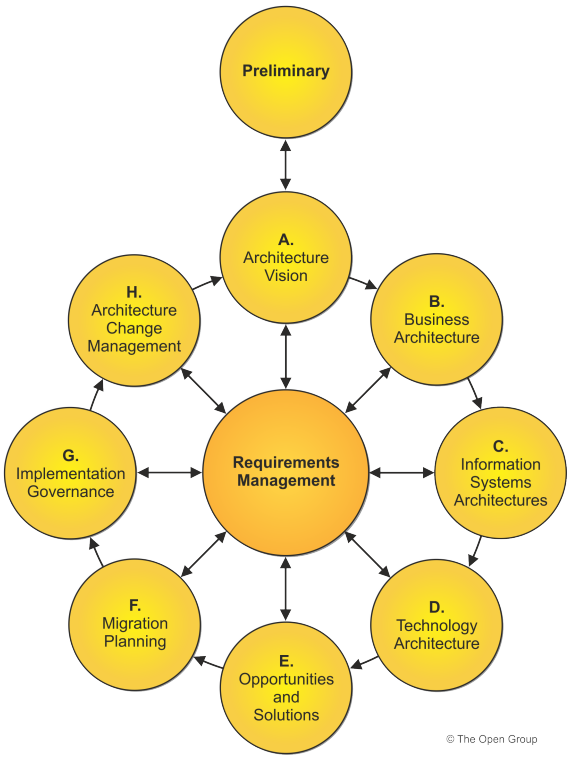


Abbildung 2.6: *Architecture Development Method (ADM) aus TOGAF [Har11]*

In der *Preliminary* Phase wird die Grundlage bestehend aus Team, Organisation, eingesetzte Architekturprinzipien und Tools bestimmt. Phase A legt die Architekturvision fest, die mithilfe des *Strategy-* und *Motivation-Layers* modelliert werden kann. Phase B bis D beschreiben die Geschäfts-, Anwendungs- und Technologiearchitektur im Ist- und Soll-Zustand, die ebenfalls durch *ArchiMate Core* Ebenen repräsentierbar sind. Phase E bis G beschreiben das Festlegen, Dokumentieren, Umsetzen und Überwachen des geplanten Vorhabens. In Phase G wird die Architektur produktiv eingesetzt und konsolidiert, sowie Anforderungen für einen erneuten Prozessdurchlauf gesammelt.

2.2 Modellanalyse

EA-Modelle können benutzt werden um ein grundlegendes Verständnis von der Unternehmensarchitektur zu erhalten, die in der Regel überaus komplex ist [Lag10]. Analysen können die Strukturen der Modelle untersuchen und komplexe Muster vereinfachen oder in greifbare Zahlen zusammenfassen. Die Modelle können außerdem mit Statistiken oder weiteren Modellen kombiniert werden, zum Beispiel ein Ist-Modell zu einem geplanten Soll-Modell [BMS09].

Die Modellanalyse unterstellt als Annahme, dass EA Modelle mehr aggregierte Informationen enthalten als in die Modellierung geführt wurden. Eine Unternehmensarchitektur lässt Schlussfolgerungen zu oder zeigt Konsequenzen im Unternehmen auf, zum Beispiel, unter welchen Bedingungen bestimmte Systeme oder Geschäftsprozesse ausfallen können.

Ein generelles Problem bei EA Modellen ist die Ungewissheit über die korrekte und angemessene Abbildung [Joh+07]. Zum Beispiel: Basiert das Modell auf den aktuellsten Informationen? Sind alle Systeme noch in Benutzung? Laufen die Prozesse wirklich so ab wie dargestellt? Ist der Detailgrad des Modells ausgewogen? Diese Arten von Fragen sollen EA Frameworks wie beispielsweise *TOGAF* beantworten [Har11]. Die Modellanalyse vertraut darauf, dass Modelle mit besten Wissen und Gewissen modelliert wurden und baut auf diesen auf.

2.2.1 Metamodelle

Wird ein Modell oder die Modellbildung selbst zum Gegenstand der Modellierung, spricht man von Metamodellen [AK03]. Metamodelle sind somit die Abstraktion von Modellen.

Metamodellierung stellt eine universell einsetzbare Methode dar um Modellsprachen zu definieren und zu beschreiben. Von zentraler Bedeutung ist dabei Struktur und Verhalten der Instanz eines Metamodells. Vorteile ergeben sich gerade in der Beschreibung von komplexen ganzheitlichen Informationssystemarchitekturen durch die einheitliche und formalisierte Darstellung.

Die *Object Management Group (OMG)* spezifizierte mit der *Meta Object Facility (MOF)* [OMG16] eine gemeinsame Grundlage für objektorientierte Metamodelle: Unter anderem ein Meta-Metamodell und Datenformate zum Austausch von Metamodellen. Durch

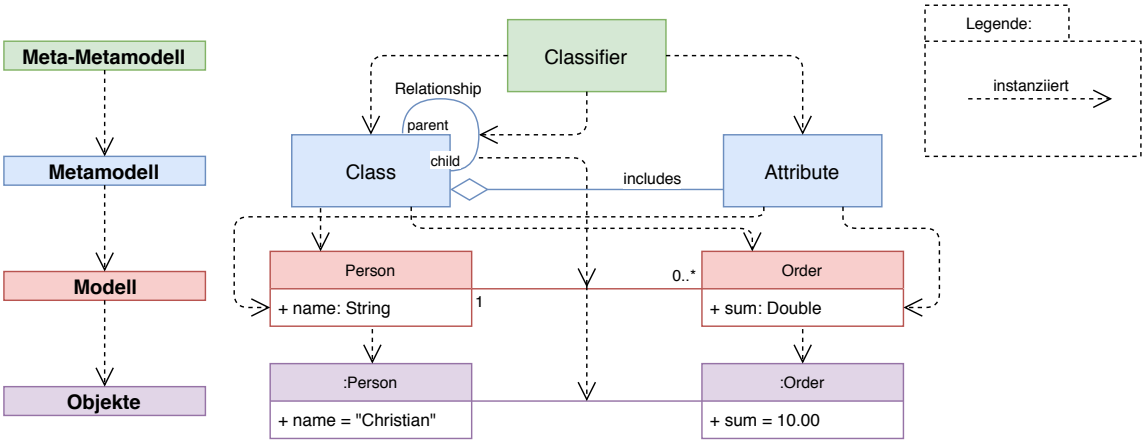


Abbildung 2.7: *Metamodellierung an einer exemplarischen Modellsprache*

die gemeinsame Herkunft unterliegt *UML* der *MOF* Spezifikation [OMG17]. *ArchiMate* hingegen nähert sich *MOF* an und strebt ein kompatibles Metamodell zukünftig an [Arm+13; Ope17]. Dadurch ergeben sich Gemeinsamkeiten wie zum Beispiel Notation von Elementen und Relationen in beiden Modellsprachen.

Abbildung 2.7 zeigt ein Beispiel einer exemplarischen Modellsprache, die über mehrere Ebenen der Abstraktion durch Modelle gegliedert ist und deren unterste Ebene konkrete Objekte darstellt.

Für die automatisierte Analyse auf Modellen haben Metamodelle außerdem den Vorteil, dass sie die Komplexität einschränken und semantisch nicht sinnvolle Verbindungen ausschließen. Andererseits erlauben Metamodelle nur eine eingeschränkte Analysemöglichkeit, wenn dafür benötigte Zusammenhänge nicht beschrieben werden können.

2.2.2 Multi-Attribute Prediction (MAP)

Multi-Attribute Prediction (MAP) - in einigen Publikationen auch *Multi-Attribute Prediction Language (MAPL)* genannt - ist eine Modellierungssprache zur Analyse von nichtfunktionalen Qualitätseigenschaften in Unternehmensarchitekturen. Die Sprache versucht mehrere Attribute für Qualität in einem Metamodell zu kombinieren und anhand der Modellstruktur zu analysieren. Die Grundlage für *MAP* legen mehrere Publikationen, unter anderem [NJN07; När+08; När12] und [NBE14].

In der finalen Version 2.3x von *MAP* wurden die folgenden Attribute zur Analyse von Qualitätseigenschaften implementiert: *Application Coupling*, *Application Size*, *Data Accuracy*, *Service Availability* und *Service Cost*. Zusätzlich können mit *Requirement Utility* verschiedene Trade-offs zwischen diesen Eigenschaften modelliert und analysiert werden. Die Evaluation der Analysen wurde durch Masterstudierende zwischen 2006 und 2014 in diversen Fallstudien mit Industriepartnern der *KTH Royal Institute of Technology* durchgeführt [Lag+17]. Außerdem wurde *MAP* in diversen Lehrveranstaltungen erfolgreich eingesetzt.

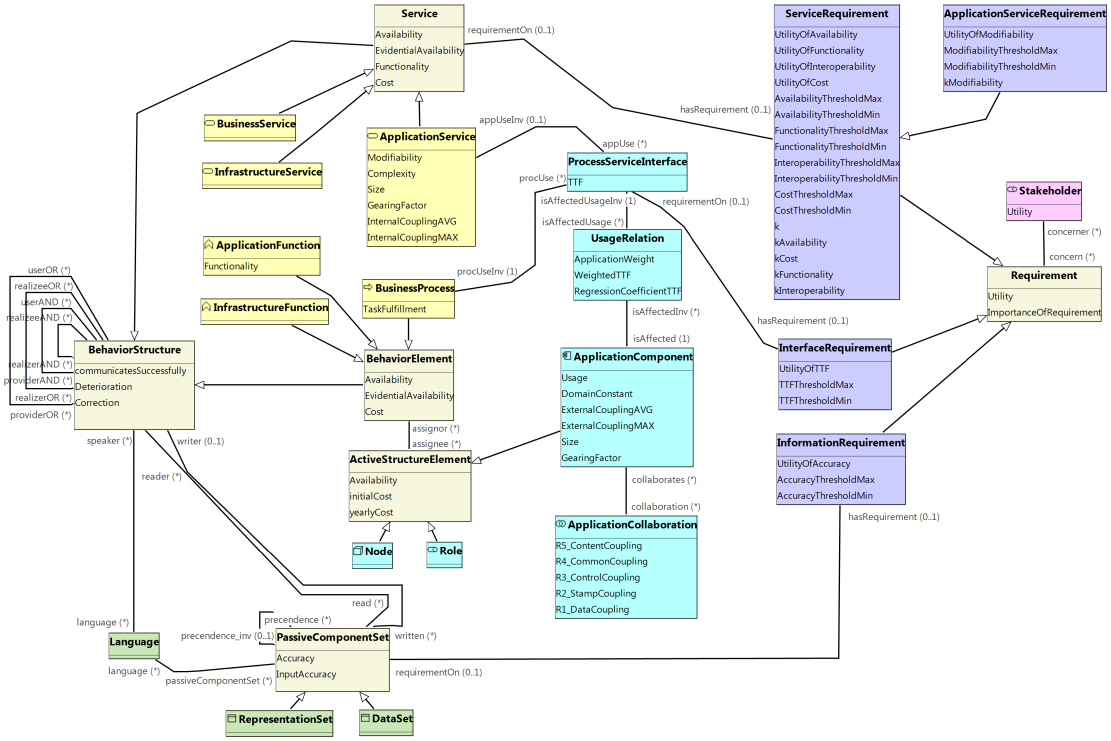


Abbildung 2.8: *MAP* class diagram Metamodell [LE14]

Das Metamodell von *MAP* (Abbildung 2.8) zeigt mit den verwendeten Typen an Elementen die Nähe zu ArchiMate. Neben vier Ebenen (*Business Layer*, *Application Layer*, *Technology Layer* und *Motivation Layer*) sind alle Aspekte (*Active Structure*, *Behavior* und *Passive Structure*) vorhanden. *MAP* besitzt insgesamt deutlich weniger Element-Typen und mögliche Relationen als ArchiMate. Dafür enthält *MAP* unter anderem die *Redundancy*-Relation um Redundanzen abzubilden.

Die Elemente *BehaviorStructure*, *BehaviorElement*, *ActiveStructureElement*, *PassiveComponentSet*, *Service* und *Requirement* (chiffongelber Hintergrund) sind abstrakt und stehen

zum Modellieren nicht zur Verfügung. Die Beziehungen zu den Subelementen sind somit hier im Metamodell auch als Generalisierung zu verstehen.

MAP verwendet die Werte von Eigenschaften, die sich auf Elementen befinden. Es gibt drei Arten an Eigenschaften: *Initial* als festen Wert, *Evidence* für benutzerdefinierte Werte und *Derived* für berechnete Werte. *Evidence* kann deterministisch (als einzelne Zahl) oder auch stochastisch (als vollständige statistische Wahrscheinlichkeitsverteilung) dargestellt werden.

Die folgenden Analysen aus *MAP* sollen jeweils anhand eines simplen Beispielmodells genauer erläutert werden.

2.2.2.1 Coupling

Kopplung (Coupling) beschreibt die Bindungsstärke durch gegenseitige Abhängigkeiten zwischen Systemen oder Komponenten [IEE90].

Die Analyse von Kopplung in *MAP* gibt die Anzahl an verbundenen *ApplicationComponent*-Elementen als Eigenschaft `coupling` an. Abbildung 2.9 zeigt dies in einem Beispielmodell.

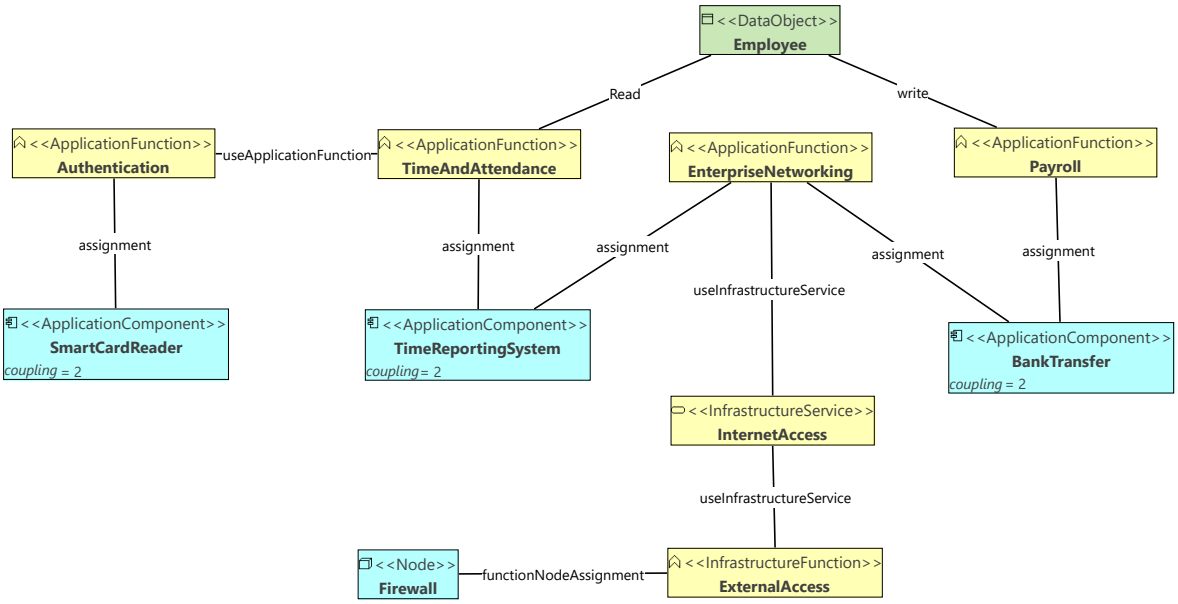


Abbildung 2.9: *Analyse von Koppling (Coupling) mit MAP*

Die Metrik der Coupling-Analyse von *MAP* geht auf *Fenton und Melton* zurück [FM90] und sollte ursprünglich mit einem Gewichtungsfaktor die Kopplungsart unterschiedlich in die Berechnung einbeziehen. Daher enthält das Metamodell (Abbildung 2.8) Eigenschaften wie `CouplingMAX` und `CouplingAVG` (für Durchschnitt). Diese wurden in den späteren Versionen von *MAP* zu einer einzigen *Coupling*-Eigenschaft auf Elementen zusammengefasst. Dadurch unterscheidet der Algorithmus in *MAP* drei verschiedene Kopplungsarten nach dem IEEE Standard [IEE90] und speichert diese in getrennten Mengen [Lan17]:

Usage dependency: Eine *ApplicationComponent* benutzt Dienste oder Funktionen anderer *ApplicationComponent*-Elemente. Diese Abhängigkeit ist im Beispielmodell zwischen *SmartCardReader* und *TimeReportingSystem* vorhanden.

Resource dependency: Eine *ApplicationComponent* teilt sich Infrastruktur Ressourcen mit anderen *ApplicationComponent*-Elementen. Diese Abhängigkeit ist im Beispielmodell zwischen *TimeReportingSystem* und *BankTransfer* über *EnterpriseNetworking* vorhanden.

Data dependency: Sowohl die *ApplicationComponent* als auch andere *ApplicationComponent*-Elemente lesen oder schreiben auf gemeinsamen Datenobjekten. Diese Abhängigkeit ist im Beispielmodell zwischen *TimeReportingSystem* und *BankTransfer* über *EmployeeData* vorhanden.

Der *coupling*-Wert wird schließlich als Vereinigungsmenge dieser Kopplungsarten zusammengefasst: $coupling = |usage \cup resource \cup data|$. Sind zwei Anwendungen über unterschiedliche Kopplungsarten gebunden, dann erhöht sich der `coupling`-Wert somit nicht. Nur die Bindung zu anderen Anwendungen, Diensten oder Schnittstellen, unabhängig von der Kopplungsart, erhöht diesen Wert.

2.2.2.2 Availability

Verfügbarkeit (Availability) in *MAP* wird als zeitabhängige Prozentangabe auf Elementen angegeben. Der *Availability*-Wert wird über das Zusammenspiel mit abhängigen Elementen und deren Verfügbarkeitswerten bestimmt. Abbildung 2.10 zeigt dies an einem Beispielmodell. Die Berechnung geht auf die Konzepte *Fault Tree Analysis (FTA)* und *Reliability Block Analysis (RBD)* zurück [RH04; Joh88].

Abbildung 2.10: *Analyse von Verfügbarkeit (Availability) mit MAP*

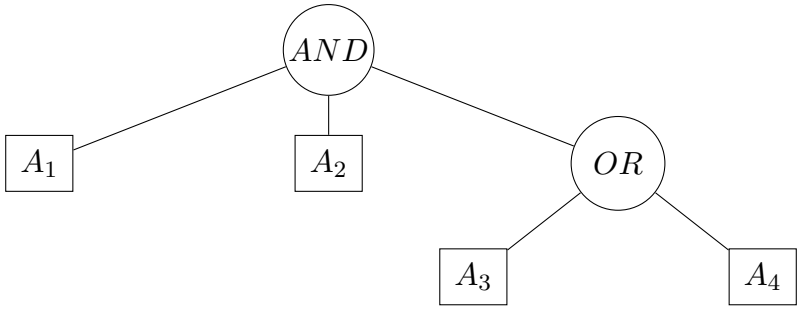
Da die Verfügbarkeitswerte voneinander abhängen, müssen spätestens die äußeren Blattelemente im Modell über einen *Availability*-Wert verfügen. Es gibt dazu zwei Möglichkeiten um diesen anzugeben:

- Aktive Elemente (in *MAP* sind das *BusinessRole*, *ApplicationComponent*, *SystemSoftware*, *Node* und *Device*) enthalten zwei Eingaben über *Evidence*: *timeBetweenFailures* (TBF) und *timeToRepair* (TTR) um Ausfälle zu beschreiben. Aus diesen lässt sich die Verfügbarkeit folgendermaßen berechnen:

$$A_{avg} = \frac{TBF}{TBF + TTR}$$

- Alternativ kann eine beobachtete Verfügbarkeit (*observedAvailability*) als Prozentangabe über *Evidence* angegeben werden, wie es bei den Elementen *PayPal* und *VISA* im Beispielmodell der Fall ist.

Bei der weiteren Berechnung wird rekursiv ein logischer *Fault Tree* aufgebaut [Lag+17]. Dabei sind Kindknoten eines Elements, die über eine *Redundancy*-Relation verbunden sind, logisch OR- und ansonsten AND-verknüpft.



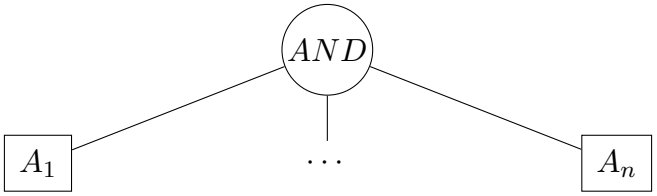
$$A_{avg} = A_1 \wedge A_2 \wedge (A_3 \vee A_4)$$

Die AND-Verknüpfung beschreibt am konkreten Beispiel Systeme, die von mehreren Komponenten abhängig sind. Fällt eine Komponente durch einen Fehler aus, dann kann das System nicht verwendet werden. OR verknüpfte Systeme sind semantisch redundant und das System kann weiterhin verwendet werden, sofern mindestens eine Komponente verfügbar ist.

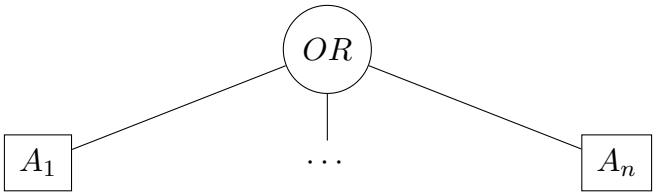
Betrachtet man das Element *BookSales* aus dem Beispielmodell, dann ergibt sich daraus folgender logischer Ausdruck:

$$A_{BookSales} = A_{Inventory} \wedge (A_{VISA} \vee A_{Paypal})$$

AND und OR verknüpfte Terme des logischen Ausdrucks werden für jedes Element wie folgt berechnet [Joh+13]:



$$A_{avg} = \prod_{i=1}^{n} A_i$$



$$A_{avg} = \coprod_{i=1}^{n} A_i = 1 - \prod_{i=1}^{n}(1 - A_i)$$

Betrachtet man wieder das Element *BookSales* aus dem Beispielmodell, dann ergibt sich daraus folgende Formel zur Berechnung von *Availability*:

$$A_{avg} = 0.97 * (1 - (1 - 0.97) * (1 - 0.95)) \approx 0.9685$$

Auch wenn in *MAP* Zyklen modelliert werden können, verhindert das Metamodell im Bereich der betrachteten Elemente für *Availability* die Modellierung von Zyklen. Der *Fault Tree* wird somit nur aus Elementarten gebildet, die im *Availability*-Metamodell existieren.

2.2.2.3 Service Cost

Die Kosten (Cost) Analyse beschreibt die Kostenverteilung über die Unternehmensarchitektur. Teilen sich zum Beispiel Dienste ein System, dann teilen sich diese auch die Kosten. Umgekehrt kostet ein Geschäftsprozess soviel wie seine enthaltenen Elemente als Summe, sofern diese nicht wiederum durch andere Prozesse geteilt sind.

Ähnlich wie bei der Verfügbarkeitsanalyse müssen die Kosten auf aktiven Elementen (für die Kostenanalyse in *MAP* nur *BusinessRole*, *ApplicationComponent* und *Node*) per Benutzereingabe als *Evidence* angegeben werden. Unterschieden werden zwei Arten von Kosten:

Capex (capital expenditure) sind einmalige Investitionsausgaben. Beispiel: Der Kauf eines Servers.

Opex (operational expenditure) sind laufende Betriebskosten. Beispiel: Die monatliche Miete für die Büroräume.

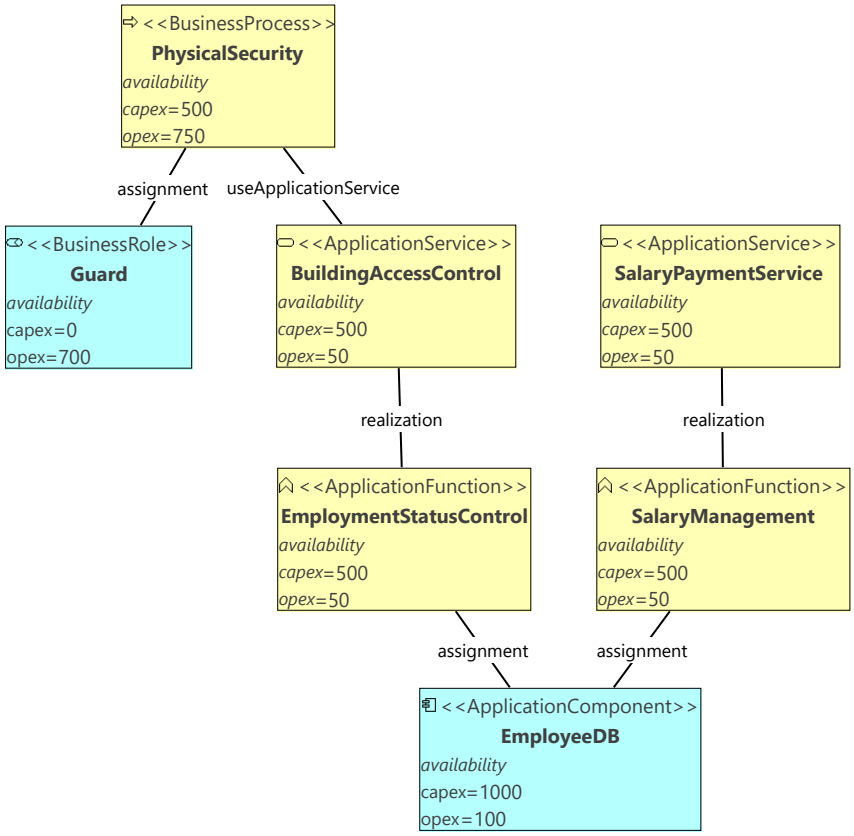


Abbildung 2.11: *Analyse von Kosten (Cost) mit MAP*

Abbildung 2.11 zeigt die Verteilung der Kosten an einem Beispielmodell. Hier teilen sich *EmploymentStatusControl* und *SalaryManagement* die Kosten der *EmployeeDB* und der Geschäftsprozess *PhysicalSecurity* trägt die gesamten Kosten von *Guard* und *BuildingAccessControl*.

Die Berechnung verläuft analog zur *Availability*-Analyse (siehe 2.2.2.2 Availability). Die *AND*-Verknüpfungen summieren die *Capex*- und *Opex*-Werte jeweils. Die rekursive Weitergabe der Werte an Elternknoten wird geteilt. Auch das Metamodell zur Kostenanalyse lässt keine Zyklen zu.

2.2.2.4 Weitere Analysen

MAP enthält noch drei weitere Analysen, die nicht in das Projekt *AYE* übertragen wurden: *Application Size*, *Data Accuracy* und *Requirement Utility*.

Application Size

Die Größe einer Anwendung wird in dieser Analyse anhand der Programmzeilen als Eigenschaft `linesOfCode` auf *ApplicationComponent* dargestellt. Dahinter liegt die Annahme, dass umfangreichere Anwendungen schwieriger zu warten sind als kleinere. Der berechnete `linesOfCode`-Wert soll dabei eine quantitative Aussage über die Komplexität der Anwendung geben.

Die Eigenschaft `linesOfCode` ist abgeleitet aus *Function Points*, basierend auf der Spezifikation von [Int10], und der verwendeten Programmiersprache in Form eines *Gearing Faktors*. Die *Function Points* sind ebenfalls abgeleitet aus gelesenen und geschriebenen *DataObjects*, sowie Interaktionen durch *BusinessRoles* über Schnittstellen auf die Anwendung. Berechnet wird `linesOfCode` als Produkt aus *Function Points* und *Gearing Faktor*.

Application Size wurde letztlich nicht für das Projekt *AYE* transformiert, weil die Berechnung der *Functions Points* nicht nachvollziehbar genug aus den Veröffentlichungen [Joh+13; Lag+17], sowie über das *Enterprise Architecture Analysis Tool (EAAT)* hervorging. Darüber hinaus lieferte *EAAT* nur stochastische Ergebnisse, obwohl die Veröffentlichungen deterministische Werte zeigen.

Data Accuracy

Die Analyse zur Integrität der Daten (*Data Accuracy*) versucht im Datenfluss durch eine Unternehmensarchitektur einen Wert für Korrektheit von Datenobjekten oder Repräsentationen abzuleiten. Die Annahme ist, dass *Behavior*-Elemente durch lesende Zugriffe auf einem Objekt und schreibende Zugriffe auf einem anderen Objekt die Korrektheit der Daten positiv wie auch negativ beeinflussen. *Behavior*-Elemente können zum Beispiel Geschäftsprozesse oder Funktionen von Anwendungen und Infrastruktur sein.

Behavior-Elemente haben entsprechend einen `correction` (Korrektur) und einen `deterioration` (Verschlechterung) -Wert in Prozent (zwischen 0 und 1), der angegeben werden kann (über *Evidence*). Abbildung 2.12 zeigt das Verhalten der Analyse an einem Beispiel.

Ähnlich zu der Verfügbarkeitsanalyse wird ein beobachteter Wert für die Korrektheit als `observedAccuracy` angegeben. Eine *Source-Target* Beziehung bestimmt den Datenfluss.

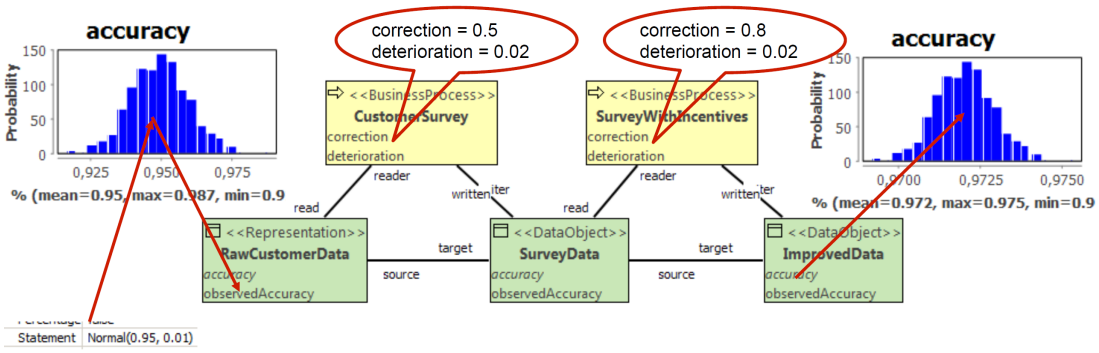


Abbildung 2.12: *Analyse der Datenintegrität (Data Accuracy) mit MAP [Lag+17]*

Der `accuracy`-Wert auf dem Element *ImprovedData* im Beispielmodell (Abbildung 2.12) berechnet sich wie folgt: Durch die rekursive Berechnung steht der `accuracy`-Wert mit 0,956 auf *SurveyData* bereits fest. Auf dem Element *ImprovedData* wird nun folgende Berechnung ausgeführt:

$$Acc_{ImprovedData} = 0.956 * (1 - 0.02) + (1 - 0.956) * 0.8 \approx 0.972$$

Die Analyse zur Datenintegrität wurde nicht im Projekt *AYE* verwirklicht, weil das ArchiMate Metamodell keine *Source-Target* Relation zur Modellierung des Datenflusses definiert. Darüber hinaus bietet ArchiMate keinen sinnvollen Ersatz für diese Relation an. Die einzige mögliche Relationsart ist die ungerichtete Assoziation, die zwischen Element-Kombinationen aus *Data Object* und *Representation* zulässig ist.

Requirement Utility

Requirement Utility ermöglicht die Modellierung von Trade-offs zwischen zwei bestehenden Analysen (*Application Coupling*, *Application Size*, *Data Accuracy*, *Service Availability* und *Service Cost*). In *MAP* gibt es zu jeder Analyse ein eigenes *Constraint*-Element, zum Beispiel *AvailabilityConstraint* zur Angabe der minimalen und maximalen Verfügbarkeit. Darüber hinaus können Ziele und Anforderungen (übereinstimmend mit den ArchiMate-Elementen *Goal* und *Requirement* aus dem *Motivation-Layer*) modelliert werden.

Anforderungen (*Requirement*-Element) verbinden immer genau zwei *Constraint*-Elemente und erzeugen eine Kosten-Nutzen Funktion (wie im Beispielmodell unter Abbildung 2.13). Ziele (*Goal*-Elemente) können beliebig viele Anforderungen verknüpfen. Die Gewichtung

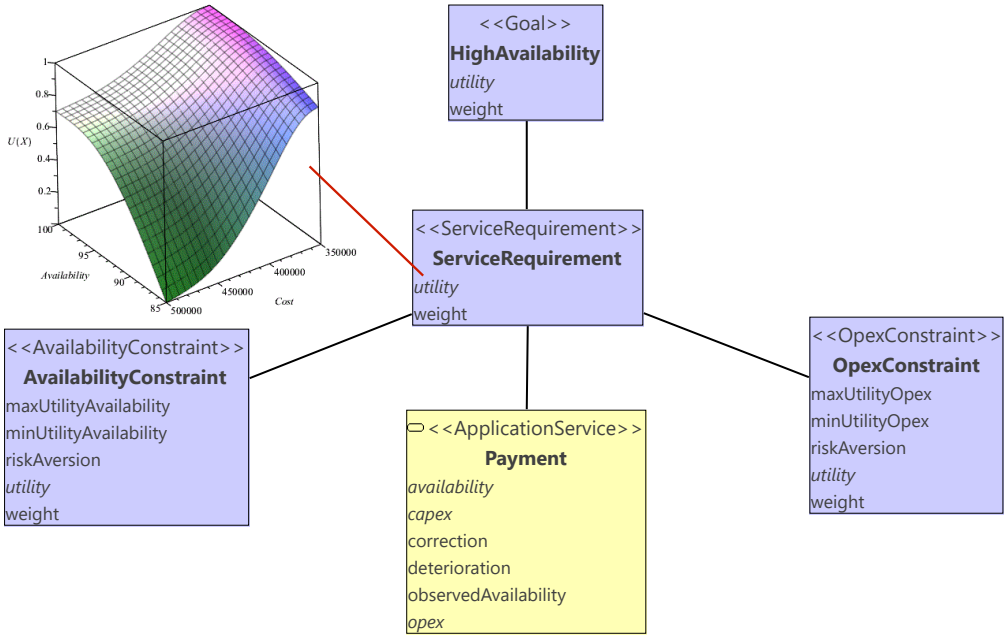


Abbildung 2.13: *Analyse von Requirement Utility mit MAP*

(*weight*) als Prozentangabe zwischen 0 und 1 bestimmt die Priorität der Eigenschaften im Verhältnis zu den Nachbarknoten.

Die *Utility*-Eigenschaft leitet den Nutzen aus diesen Angaben ab. Fällt beispielsweise die Verfügbarkeit unter den minimalen Wert einer Bedingung wird der Nutzen als 0 gewertet. Ab dem maximalen Wert bleibt der Nutzen konstant. Die Steigung dazwischen kann über die *riskAversion* abgestimmt werden: Null bedeutet neutral gegenüber einem Risiko und somit ist die Steigung linear. Negative Werte drücken eine Zuneigung zum Risiko aus und die Steigung folgt der Exponentialfunktion. Positive Werte sind demnach dem Risiko abgeneigt und folgen der Grenzfunktion.

Tiefer gehende Details zu der Berechnung und Theorie finden sich in den Veröffentlichungen [Ost+13; Lag+17]. Die Umsetzung von *Requirement Utility* wurde für das Projekt *AYE* nicht in Betracht gezogen.

Theoretische Analysen

Darüber hinaus existieren weitere theoretische Konzepte für Analysen, die in *MAP* nicht implementiert wurden. Zum Beispiel Modifizierbarkeit (*Modifiability*) und Interoperabilität (*Interoperability*) [Joh+13]. Diese Qualitätseigenschaften besitzen viele Einfluss-

faktoren, die nicht mit dem ArchiMate-ähnlichen Metamodell vereinbar sind und daher eine eigene Modellsprache benötigen [Lag10]. Kopplung und Anwendungsgröße sind zwei Einflüsse, die für eine Modifizierbarkeitsanalyse relevant sind und mit *MAP* analysiert werden können [Lag+17].

3 Projekt

Das Projekt *Analyse Your Enterprise (AYE)* ist im Rahmen des Masterstudiums in Zusammenarbeit mit dem *HAW Laboratory for Architecture and IT-Management (HAWAI)* und dem Department *Industrial Information and Control Systems* der *KTH Royal Institute of Technology* in Stockholm entstanden. Das Ziel war, die praktische Anwendbarkeit der Analysen von *Multi-Attribute Prediction (MAP)* auf Basis von *ArchiMate* in einem webbasierten Tool nutzbar zu machen. Durch *AYE* soll gezeigt werden, dass sich die Analysen für den praktischen Einsatz eignen. Sofern ein ArchiMate-Modell existiert, soll ohne zusätzlichem Aufwand eine Analyse von implementierten Qualitätseigenschaften möglich sein. *MAP*-Modelle können bereits über ein (nicht webbasiertes) Tool modelliert und analysiert werden: *Enterprise Architecture Analysis Tool (EAAT)* [EAA13]. Der Vorteil für den Anwender von *AYE* gegenüber *EAAT* soll im direkten Analysieren bestehender ArchiMate Modelle liegen, ohne ein neues Modell dafür erstellen zu müssen.

Dieses Kapitel gibt zunächst einen Überblick über die Anforderungen an *AYE* und geht anschließend tiefer auf Design und Umsetzung des Projektes ein.

3.1 Anforderungen

Die Anforderungen sind auf Basis von gemeinsamen Diskussionen und Gesprächen innerhalb der HAWAI-Arbeitsgruppe über längere Zeit entstanden. *AYE* wurde dabei als Prototyp entwickelt und realisiert daher nicht alle Anforderungen bzw. einige nur teilweise.

A1) AYE soll auf dem Metamodell von ArchiMate 3.x basieren
MAP orientiert sich am älteren ArchiMate 2.x Metamodell, weil Version 3.x erst 2016 erschien [Ope17]. Die neuere Version fügt im wesentlichen neue Ebenen hinzu (*Strategy Layer* und *Physical Layer*), sowie diverse Relationen zwischen den neuen Elementen in

den neuen Ebenen. Außerdem ist das *Exchange File Format* für ArchiMate entsprechend erweitert worden [Ope16] und bleibt dabei weiterhin kompatibel zu ArchiMate 2.x Modellen. Für das Ziel einer praktischen Anwendbarkeit ist die Umsetzung des ArchiMate 3.x Metamodells daher sinnvoll um möglichst viele unterschiedliche ArchiMate-Modelle zu unterstützen.

A2) AYE überlässt dem Benutzer, welche Elemente und Relationen in die Berechnung von Analysen einfließen

Am Department *Industrial Information and Control Systems* der *KTH* Stockholm wurde zur Modellierung von *MAP* das *EAAT* [EAA13] entwickelt. Bestandteil von *EAAT* ist das Programm *Class Modeler* als Entwicklungsumgebung, über das Metamodelle mit Analysemechanismen implementiert werden können. Über diesen Weg ist es auch möglich das Metamodell von *MAP* zu erweitern.

In *AYE* ist das Metamodell schon von der Vision her unveränderbar. Dadurch ist die Anpassbarkeit von Analysen zwar eingeschränkt, dafür lassen sich die betroffenen Elemente und Relationen einfacher auf die Bedürfnisse anpassen. Am Beispiel Kopplung würde dies bedeuten, dass die Berechnung auch menschliche Schnittstellen (im Sinne von Daten von System *A* ausdrucken und über ein Formular in System *B* eintragen) einbeziehen kann.

AYE soll daher zu jeder Analyse einstellbare Listen über Elemente und Relationen enthalten, die in die Berechnung einfließen, um die Analysen auf einfache Weise zu individualisieren.

A3) AYE soll zumindest die Analyse für Kopplung und Verfügbarkeit implementieren

Zumindest Kopplung und Verfügbarkeit sollten über *AYE* zu berechnen sein. Zum Projektabschluss konnte auch die Kostenanalyse umgesetzt werden.

A4) Existierende ArchiMate-Modelle müssen in AYE importiert werden können

Eine Import-Funktion ist für AYE essentiell, damit ein Benutzer möglichst schnell an Analyseergebnisse mit *AYE* kommen kann. *ArchiMate* Modelle können in verschiedenen Tools erstellt werden, zum Beispiel *Archi*, *Visual Paradigm* und *Enterprise Studio*. Gemeinsam exportieren sie das *ArchiMate Open Exchange Format* [Ope16]. *AYE* soll somit dieses Dateiformat prüfen und importieren können.

A5) AYE soll nach dem Import eines Modells ohne weitere Benutzereingaben zu Analyseergebnissen kommen

Nach dem Import eines ArchiMate-Modells soll *AYE* durch Vergabe von Standardwerten die Berechnungen direkt ausführbar machen. Betroffen sind insbesondere die Verfügbarkeits- und Kostenanalyse. Auch mit Standardwerten lassen sich dadurch Auswirkungen ablesen.

A6) Für jedes Modell lässt sich einstellen, welche Analysen berechnet werden sollen

Die Analysen von *AYE* nutzen die Eigenschaften (siehe 2.1.5 Eigenschaften) um Analyseergebnisse zu speichern. Jede Eigenschaft muss nach ArchiMate Spezifikation als *Property Definition* für ein Modell hinterlegt werden [Ope17]. *AYE* soll Vorlagen für *Property Definitions* anbieten, um Analysemechanismen aktivieren zu können. Auch das Entfernen der Definitionen zum Deaktivieren soll möglich sein.

A7) Die Analyseberechnungen in AYE sollen parallel ausgeführt werden und jeweils maximal 2 Sekunden dauern

AYE soll als Webanwendung implementiert werden, in der mehrere Benutzer gleichzeitig Modelle auswerten können. Bei größeren Modellen kann die Berechnung auch länger dauern, daher sollen die Analysealgorithmen in unabhängigen und parallel arbeitenden Worker-Prozessen durchgeführt werden. Bezugnehmend auf Anforderung *A14* ist das Produktivsystem ein Cloud-System, das im Hintergrund mit einem containerbasierten Kubernetes-Cluster arbeitet und die Skalierung von Prozessen erlaubt. Diese Möglichkeiten soll *AYE* nutzen.

A8) Mit AYE können ArchiMate-Modelle modelliert werden

Die Analysen auf Modellen erlauben es, Probleme in der Zusammenarbeit mit anderen Systemen frühzeitig zu erkennen. Damit der Benutzer verschiedene Konstellationen ausprobieren kann, sollte *AYE* zumindest Veränderungen am Modell erlauben.

A9) In AYE werden vom Metamodell nicht erlaubte Elemente und Relationen rot markiert

Diese Anforderung ist relativ früh im Entwicklungsprozess aufgenommen worden. Importiert der Benutzer ein fehlerhaftes Modell, das nicht dem ArchiMate-Metamodell entspricht, dann sollen nicht erlaubte Elemente und Relationen farblich markiert werden. Mit dem *ArchiMate Open Exchange Format* [Ope16] Version 3.x ist der Import nicht mehr möglich, wenn die Schema-Datei zur Überprüfung angewendet wird. Dennoch ist diese Anforderung möglicherweise relevant, wenn *AYE* Funktionen zur Modellierung anbietet.

A10) AYE erlaubt die Erweiterung des ArchiMate-Metamodells durch die Definition von Element- und Relationstypen für bestimmte Analysen
Einige Analysen von *MAP* benötigen zusätzliche Typen von Relationen, zum Beispiel *Data Accuracy* (siehe Abschnitt 2.2.2.4). ArchiMate hat Relationen fest ins Metamodell verankert und bietet nicht die Möglichkeit eigene Relationen zu definieren. Somit ist auch das *Open Exchange Format* ohne Erweiterbarkeit von Relationen aufgebaut.

Die Möglichkeit der Definition von Elementen und Relationen, die ihr Verhalten von den vorgegebenen Vertretern aus ArchiMate erben, würde eine Möglichkeit zur Erweiterung darstellen, jedoch die Austauschmöglichkeiten per *Open Exchange Format* einschränken. Aufgrund der Einschränkungen wurde diese Anforderung noch nicht in *AYE* umgesetzt.

A11) AYE muss einen Service zur Authentifizierung HAWAI-übergreifend nutzen
AYE soll nach Anforderung *A14* einen serviceorientierten Ansatz nutzen. Damit nicht jeder Service Funktionalitäten zur Autorisierung und Authentifizierung implementieren muss, bietet *Single Sign-On (SSO)* die Möglichkeit eine Benutzerentität über mehrere bekannte Services aufrecht zu erhalten [New15].

Daraus entstand die Idee, diesen Vorteil auf die gesamte Arbeitsgruppe *HAWAI* zu erweitern. Damit die typischen Funktionalitäten (registrieren, einloggen, Passwort vergessen) nicht durch jede Anwendung im *HAWAI*-Kontext wiederholt implementiert werden müssen, soll ein projektübergreifender Service zum Autorisieren und Authentifizieren für die Arbeitsgruppe erstellt und genutzt werden.

A12) AYE muss in der bestehenden Anwendungslandschaft integriert sein
Die Arbeitsgruppe *HAWAI* besitzt (Stand Dezember 2018) noch keine eigene Anwendungslandschaft. Frühe Visionen aus gemeinsamen Diskussionen streben die Realisierung

einer Landschaft mit heterogenen Anwendungen und Schnittstellen an. Im Hinblick auf eine zukünftig lauffähige Anwendungslandschaft soll *AYE* eine Schnittstelle anbieten, die gemessene Werte (zum Beispiel Verfügbarkeit oder Kosten) einzelner Anwendungen entgegennehmen kann, um diese in den Analysen zu berücksichtigen.

Diese Anforderung schließt außerdem mit ein, dass weitere Tools unabhängig von der Anwendungslandschaft mit *AYE* arbeiten können. Tools wie zum Beispiel ein Dashboard als zentrale Toolbox mit Dateiverwaltungsfunktion, über die Dateien mit enthaltenen Modellen importiert und exportiert werden können [Jäc18].

A13) AYE soll nach dem Microservice-Architekturstil entwickelt werden
Microservices erhielten mit der Veröffentlichung des gleichnamigen Artikels von Martin Fowler (2014) [FL14] in der *HAWAI*-Arbeitsgruppe viel Aufmerksamkeit.

Die Aufteilung der Komponenten in unabhängige Services unterstützt die Zusammenarbeit von mehreren Personen an dem Projekt. Außerdem lassen sich Microservices flexibel skalieren, wodurch eine Beschleunigung der Analyse-Prozesse möglich wird. Die Architektur von *AYE* soll sich daher Microservice-Architekturstil orientieren.

Das Ziel ist, die Komponenten serviceorientiert zu gestalten, leichtgewichtige Schnittstellen (wie zum Beispiel *REST/HTTP* und *Messaging*) einzusetzen, die Datenhaltung zu dezentralisieren und automatisierte Auslieferungsprozesse wie *Continuous Integration/Delivery* zu nutzen.

A14) AYE soll auf der Informatik Compute Cloud (ICC) der Hochschule für Angewandte Wissenschaften (HAW) Hamburg bereitgestellt werden
Während der Entwicklung von *AYE* wurde an der HAW Hamburg ein *Kubernetes*-Cluster[1] (*ICC*) realisiert. Dieser soll für eine Produktiv-Umgebung genutzt werden.

3.2 Design

In diesem Kapitel sollen Architektur und Designentscheidungen erläutert werden.

[1] Kubernetes ist ein Container-Orchestrierungssystem. Mit Kubernetes können virtualisierte Container automatisiert bereitgestellt, skaliert und verwaltet werden.

3.2.1 Datenmodell

Die anfängliche Basis für die Entwicklung von *AYE* stellt das Entity-Relationship-Modell (ERM) dar. Die Datenverwaltung muss im Wesentlichen alle ArchiMate Entitäten abbilden können.

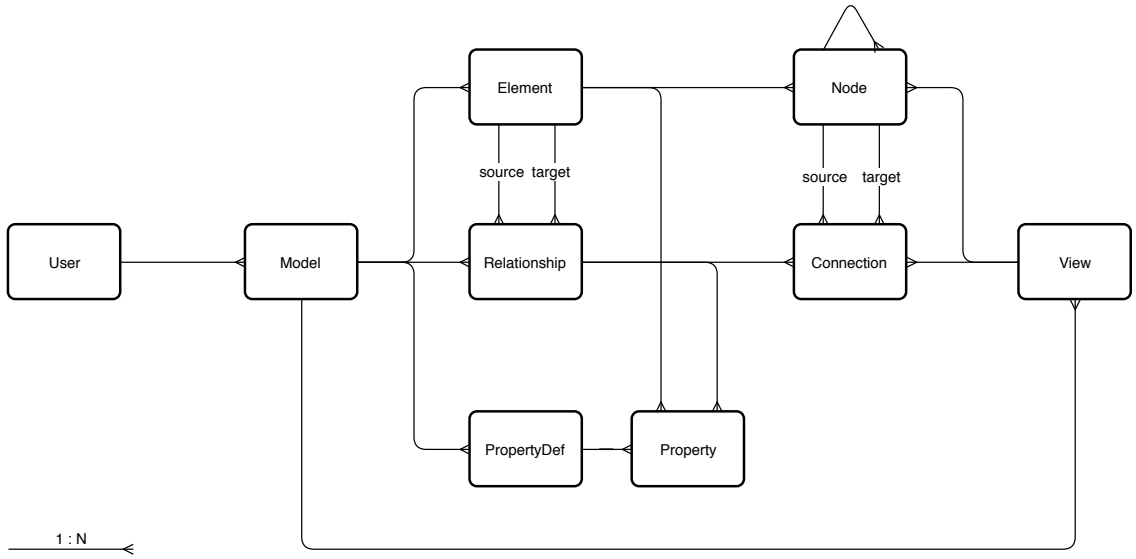


Abbildung 3.1: *ArchiMate Datenmodell*

Abbildung 3.1 zeigt das ERM in Martin-Notation. Die Entitäten wurden aus der ArchiMate Spezifikation [Ope17] und der *ArchiMate Open Exchange Format* Schemadatei [Ope16] abgeleitet.

Die Entitäten *Model*, *Element*, *Relationship* auf der linken Seite bilden das ganzheitliche Modell ab, das in mehrere Sichten (*View*) mit den entsprechenden Gegenstücken *Node* und *Connection* auf der rechten Seite zusammengesetzt ist. So lassen sich Elemente und Relationen eines Modells auf mehreren Sichten darstellen. Attribute zur Positionierung (x und y) für die Ausgabe befinden sich daher nur in *Node* und *Connection*. *Property Definition* (als Entität kurz *PropertyDef*) bestimmt, welche Eigenschaften mit Bezeichnung und Typ in Elementen und Relationen verwendet werden dürfen. *Propery* enthält somit die eigentlichen Analysewerte.

Folgende Attribute und relationale Zusammenhänge wurden für die Entitäten (aus Abbildung 3.1) zugeordnet. Unterstrichene Attribute stellen den Primärschlüssel dar, kursive Attribute den Fremdschlüssel:

User	UserID, Username, Firstname, Lastname, EMail, Role
Model	ModelID, *UserID*, Name, Documentation
Element	ElementID, *ModelID*, Label, Type, Documentation
Relationship	RelationshipID, *ModelID*, *SourceElementID*, *TargetElementID*, Label, Type, Documentation
View	ViewID, *ModelID*, Name, Viewpoint, Documentation
Node	NodeID, *ParentNodeID*, *ViewID*, *ElementID*, X, Y, Width, Height, FillColor, LineColor
Connection	ConnectionId, *ViewID*, *RelationshipID*, *SourceNodeID*, *TargetNodeID*, Type, LineColor
PropertyDef	PropertyDefID, *ModelID*, Name, Type, Documentation
Property	PropertyID, *ElementID*, *RelationshipID*, Value

Das Attribut *Type* speichert die Typenbezeichnung direkt als String, zum Beispiel *ApplicationComponent* in der Element-Entität oder *Composition* in der Relationship-Entität. Das Attribut *Documentation* ist zur Angabe von freien Text.

ArchiMate definiert für *Connections* noch Biegungspunkte (*Bendpoints*), um den Linienverlauf über bestimmte zweidimensionale Positionen abzubilden. Diese Entität wurde für den Prototypen weggelassen.

3.2.2 Komponenten und Services

Im nächsten Schritt wurden die Komponenten auf oberster Ebene definiert. Abbildung 3.2 zeigt diese als UML-Komponentendiagramm. Jede Komponente arbeitet als unabhängiger Service mit eigener Laufzeitumgebung.

Im Zentrum steht der *AYE Analyse Service*, der über eine Schnittstelle die Funktionalitäten zur Modellierung und Analyse von ArchiMate erlauben soll. Die Daten dazu speichert der Analyse-Service in einer beliebigen Datenbank ab, auf die kein anderer Service Zugriff hat.

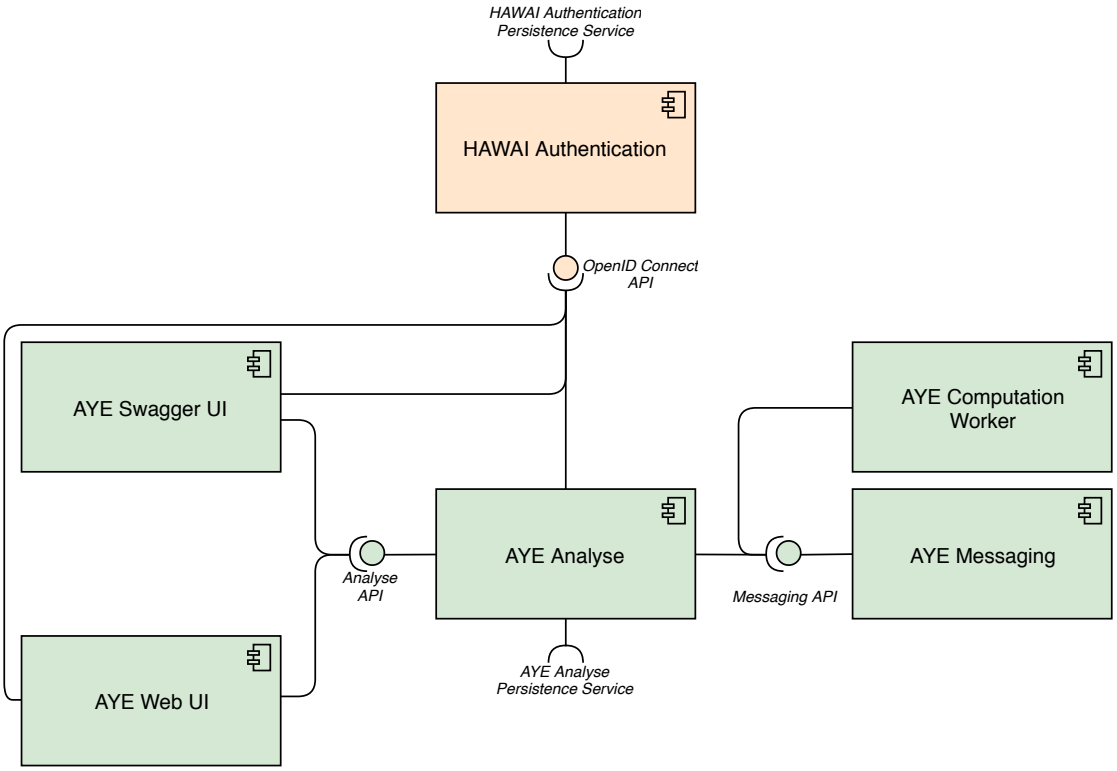


Abbildung 3.2: *Services und Komponenten von AYE*

Wird eine Analyse zu einem Modell über die *Analyse API* angefordert, soll für jede Analyse (Verfügbarkeit, Kopplung oder Kosten) eine Nachricht an den *AYE Messaging* Service gesendet werden. Diese Nachricht enthält einen Graph mit den Knoten und Kanten, die zur Lösung nötig sind. Dieser Graph muss daher nicht das ganze Modell abbilden, für einige Analysen können bestimmte Knoten und Kanten ausgelassen werden.

Der *AYE Computation Worker* lauscht auf der *Messaging API* und konsumiert Nachrichten, sofern freie Kapazitäten zur Berechnung verfügbar sind. Die in den Nachrichten enthaltenen Graphen werden für Transformationen und Berechnungen allein im Arbeitsspeicher gehalten. Auf einem Produktivsystem wird dieser Worker skaliert, dadurch laufen die Berechnungen parallel ab. Das Ergebnis der Berechnung stellt eine Liste aus Eigenschaften (Property-Entität) mit Werten dar, die der *AYE Analyse* Service durch Lauschen auf der *Messaging API* entgegen nimmt und speichert.

Der *AYE Swagger UI* Service stellt eine Benutzeroberfläche zur Dokumentation der *Analyse API* zu Verfügung, in der außerdem einzelne Operationen der Schnittstelle direkt

ausgeführt werden können. Die Dokumentation wurde im Entwicklungsprozess parallel mit der *AYE Analyse API* umgesetzt, um mit den Operationen experimentieren zu können. Zum Einsatz kommt dabei das Open Source Tool *Swagger UI* [Sma16].

Die *AYE Web UI* realisiert die Benutzeroberfläche zur Modellierung und Analyse als Single-Page-Webanwendung, die Komponenten anhand der Daten über die *Analyse API* nachladen kann.

Der *HAWAI Authentication* Service soll nach Anforderung A11 auch von anderen Projekten der *HAWAI*-Arbeitsgruppe verwendet werden können. Die REST/HTTP-Schnittstelle implementiert den OpenID Connect Standard [Ope14] und ermöglicht so die zentrale Autorisierung und Authentifizierung per *Single Sign-On (SSO)* für alle Services der Arbeitsgruppe. Entsprechend erhält der *HAWAI Authentication* Service die Datenhoheit über User-Entitäten und möglichen weiteren Entitäten wie Rollen und Berechtigungen.

3.2.3 Schnittstellen

Während der OpenID Connect Standard bereits eine Struktur vorgibt sind die Schnittstellen für die *Analyse API* und die Struktur der Nachricht für die *Messaging API* zu definieren.

Analyse API

Die *Analyse API* ist eine REST/HTTP-Schnittstelle und bietet Operationen zu den Entitäten aus Abbildung 3.1 an. Die Request- und Response-Nachrichten sind immer im JSON-Format. Alle verwendeten HTTP-Methoden und ihre Bedeutung werden im folgenden erklärt:

GET Ruft Tupel ab.

POST Fügt neue Tupel hinzu.

PUT Ändert ein angegebenes Tupel oder fügt es hinzu, wenn es nicht existiert.

PATCH Ändert angegebene Attribute eines bestehenden Tupels.

DELETE Löscht ein angegebenes Tupel.

OPTIONS Gibt erlaubte Operationen der Schnittstelle zurück. Dies ist wichtig im Umgang mit Cross-Origin Resource Sharing (CORS) während der Entwicklung.

Die Struktur der URI für Request-Nachrichten ist wie folgt aufgebaut:

POST	/tasks/analyse/{modelId}
GET|POST	/models/
GET|PUT|PATCH|DELETE	/models/{modelId}
GET|POST	/models/{modelId}/views/
GET|PUT|PATCH|DELETE	/models/{modelId}/views/{viewId}
GET|POST	/models/{modelId}/elements/
GET|PUT|PATCH|DELETE	/models/{modelId}/elements/{elementId}
GET|POST	/models/{modelId}/relationships/
GET|PUT|PATCH|DELETE	/models/{modelId}/relationships/{relationshipId}
GET|POST	/models/{modelId}/propertydefs/
GET|PUT|PATCH|DELETE	/models/{modelId}/propertydefs/{propertydefId}
GET|POST	/models/{modelId}/views/{viewId}/nodes/
GET|PUT|PATCH|DELETE	/models/{modelId}/views/{viewId}/nodes/{nodeId}
GET|POST	/models/{modelId}/views/{viewId}/connections/
GET|PUT|PATCH|DELETE	/models/{modelId}/views/{viewId}/connections/{connectionId}
GET|POST	/models/{modelId}/elements/{elementId}/properties/
GET|PUT|PATCH|DELETE	/models/{modelId}/elements/{elementId}/properties/{propertyId}

Attribute, die nicht in der URI eine Erwähnung finden, werden im HTTP-Header der Request-Nachricht als *Form Data* übertragen.

Die Analyse eines Modells wird über die `POST`-Operation der URI `/tasks/analyse/{modelId}` gestartet. Der Import eines Modells erfolgt über die `file`-Eigenschaft im HTTP-Header in der `POST`-Operation der URI `/models/`.

Die Schnittstelle antwortet mit den folgenden *HTTP Status Codes* in der Response-Nachricht:

200 Erfolgreich (`GET|POST|PUT|PATCH`)

202 Akzeptiert (`POST` nur bei `/tasks/analyse/{modelId}`)

204 Erfolgreich ohne Inhalt (`DELETE`)

400 Ungültige Anfrage (`POST|PUT|PATCH`)

403 Verboten (`GET|POST|PUT|PATCH|DELETE`)

404 Nicht gefunden (bei allen Operationen möglich)

409 Konflikt (`POST|PUT|PATCH`)

423 Gesperrt (`DELETE`)

Die Dokumentation der *Analyse API* ist online[2] abrufbar und enthält zu jeder Operation genauere Informationen.

Messaging API

Die *Messaging API* gehört zu einem Service, der eine Message Oriented Middleware (MOM) realisiert. Die Nachrichten sind im *GraphML*-Format aufgebaut, um Strukturen von Graphen zu beschreiben.

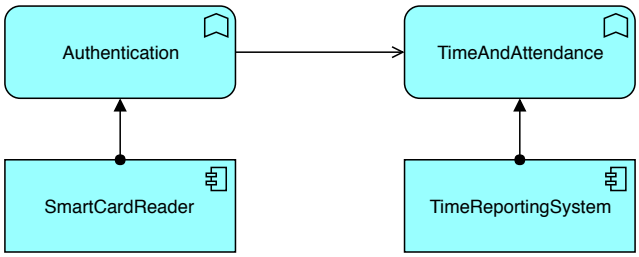


Abbildung 3.3: *Beispielmodell zur Übertragung mit GraphML*

Ein Beispiel-Graph (Abbildung 3.3) aus vier Elementen und drei Relationen für die Kopplungsanalyse würde wie folgt beschrieben werden:

[2] Dokumentation der *Analyse API* kann unter https://analyse-your-enterprise.de/api/doc/ abgerufen werden.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
5   http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
6   <key id="model" for="graph" attr.name="model" attr.type="string"/>
7   <key id="label" for="node" attr.name="label" attr.type="string"/>
8   <key id="type" for="node" attr.name="type" attr.type="string"/>
9   <key id="coupling_def" for="graph" attr.name="coupling" attr.type="string"/>
10  <graph id="Coupling" edgedefault="undirected">
11    <data key="model">modelId</data>
12    <data key="coupling_def">propertydefId</data>
13    <node id="TimeAndAttendance">
14      <data key="label">TimeAndAttendance</data>
15      <data key="type">ApplicationFunction</data>
16    </node>
17    <node id="Authentication">
18      <data key="label">Authentication</data>
19      <data key="type">ApplicationFunction</data>
20    </node>
21    <node id="SmartCardReader">
22      <data key="label">SmartCardReader</data>
23      <data key="type">ApplicationComponent</data>
24    </node>
25    <node id="TimeReportingSystem">
26      <data key="label">TimeReportingSystem</data>
27      <data key="type">ApplicationComponent</data>
28    </node>
29    <edge id="1" source="Authentication" target="TimeAndAttendance">
30      <data key="type">Serving</data>
31    </edge>
32    <edge id="2" source="SmartCardReader" target="Authentication">
33      <data key="type">Assignment</data>
34    </edge>
35    <edge id="3" source="TimeReportingSystem" target="TimeAndAttendance">
36      <data key="type">Assignment</data>
37    </edge>
38  </graph>
39 </graphml>
```

Der Graph wird über die *GraphML*-Elemente `graph`, `node` (Knoten) und `edge` (Kante) beschrieben. Die `key`-Elemente (Zeile 6 bis 9) deklarieren die anwendungsspezifischen

`data`-Einträge (zum Beispiel in Zeile 11, 12 oder 14, 15), die innerhalb der *GraphML*-Elemente zur Beschreibung des Graphen verwendet werden.

Das `data`-Element `model` (Zeile 11) enthält den Identifikator des Modells auf dem die Analyse durchgeführt werden soll. Über das `data`-Element `coupling_def` (Zeile 12) kann die Analyseart zugeordnet werden. `type`-Elemente geben an, welche Stereotypen die Knoten und Kanten in Bezug auf *ArchiMate* besitzen.

Die Verwendung der `key`- bzw. `data`-Elemente wird dynamisch vom *AYE Analyse* Service aufgebaut, so dass auch mehrere Analysearten angefordert werden können. Auch die Angabe von Eigenschaften (wie beispielsweise `observedAvailability` in der Verfügbarkeitsanalyse) ist über diesen Weg möglich.

Als Ergebnis gibt der *AYE Computation* Service eine Liste mit Eigenschaften und Werten im JSON-Format zurück an die *Messaging API*. Die Antwort-Nachricht in Bezug auf den Beispiel-Graph (Abbildung 3.3) wäre wie folgt strukturiert:

```
1 [
2   {'element': 'SmartCardReader', 'propertydef': '[id]', 'value': 1},
3   {'element': 'TimeReportingSystem', 'propertydef': '[id]', 'value': 1}
4 ]
```

Der Identifikator im Feld `propertydef` wurde gekürzt und bezieht sich auf die *Property Definition* für Kopplung.

OpenID Connect API

Die *OpenID Connect API* ist eine REST/HTTP-Schnittstelle für Benutzer- und Ressourcen-Verwaltung [Bih15]. Ein Webservice kann den Benutzer auf diesen Service umleiten um die Autorisierung und Authentifizierung durchzuführen. Im Anschluss leitet der Service den Benutzer zurück auf die angefragte Seite, während der Service einen Token für die laufende Session erhält, mit dem erlaubte Ressourcen über mehrere bekannte Services abrufbar sind. Ist eine Session noch aktiv, beispielsweise beim Öffnen eines neuen Browser-Tabs, dann sind keine Benutzereingaben nötig. Die Weiterleitungen greifen direkt den Session-Token ab.

Abbildung 3.4 zeigt im UML-Sequenzdiagramm den nach *OpenID-Connect* [Ope14] implementierten Ablauf zwischen dem *AYE Web UI* Service und dem *HAWAI Authentication* Service. Stellvertretend zum *AYE Web UI* Service kann im Ablauf auch jeder andere

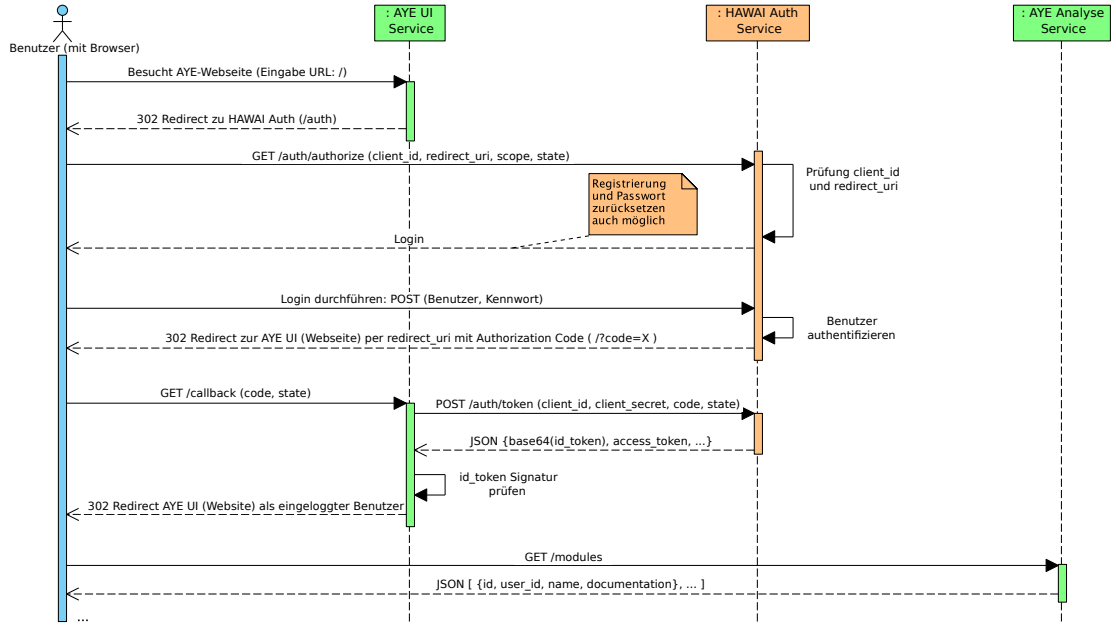


Abbildung 3.4: *Interaktion mit der OpenID Connect API*

Client stehen (zum Beispiel geht der *AYE Swagger UI* Service identisch vor). Auch für den *AYE Analyse* Service kann ein anderer Ressourcen-Service eingesetzt werden.

3.3 Umsetzung

Dieses Kapitel geht auf die eingesetzten Technologien, aufgetauchten Herausforderungen und Lösungen in der Umsetzung der einzelnen Services ein.

3.3.1 Services

AYE Analyse Service

Der *AYE Analyse Service* ist mit *PHP 7.x* und *Phalcon PHP* als Framework realisiert. *Phalcon PHP* ist in C geschrieben und wird als Erweiterung zu *PHP* in die Laufzeitumgebung eingebunden.

Phalcon PHP bietet eine sogenannte Micro-Architektur an, mit der REST/HTTP-Schnittstellen leichtgewichtig entwickelt werden können. Gleichzeitig ist auch *Object Relational*

Mapping (ORM) mit relationalen und dokumentenbasierten Datenbanken möglich. In diesem Fall arbeitet der Analyse Service mit einer *MySQL* Datenbank zusammen.

Die Überprüfung des Benutzertokens wird über ein Event beim Zugriff über die Schnittstelle auf bestimmte Routen gelöst. Bei einer Anfrage wird das mitgelieferte *JSON Web Token (JWT)* entschlüsselt und die darin enthaltenen Benutzerinformationen (User-Entität aus Abbildung 3.1) gegenüber der *OpenID Connect API* überprüft. Erlaubte aber dem *Analyse* Service unbekannte User werden im gleichen Schritt angelegt.

Neben den spezifischen Funktionalitäten der Schnittstelle, die nach Entität in eigenen Controller-Klassen gekapselt sind, gibt es noch die folgenden Komponenten: *GraphML*, *Messaging*, *JWT* und *ModelImport*. Diese werden von den Controller-Klassen angesprochen. Die *GraphML*-Komponente verpackt ein Modell in das GraphML-Format. Die *Messaging* Komponente kann Nachrichten an die *Messaging API* senden und von dieser konsumieren. Die JWT-Komponente extrahiert Benutzerinformationen und prüft die Befugnisse des Benutzers gegenüber der *OpenID Connect API*. Die *ModelImport* Komponente liest eine XML-Datei im *ArchiMate Open Exchange Format* ein, prüft diese auf das korrekte ArchiMate Metamodell und speichert die enthaltenen Elemente.

Der *AYE Analyse* Service kann auf zwei verschiedene Arten gestartet werden: Als HTTP-Service um die REST/HTTP API zur Verfügung zu stellen oder als Konsolen-Anwendung, die auf der *Messaging API* lauscht. In beiden Möglichkeiten stehen die identischen Komponenten sowie das *ORM* zur Verfügung.

AYE Computation Worker Service

Der *AYE Computation Worker Service* ist mit *Python 3.x* realisiert und verwendet zur Verarbeitung von Graphen die Bibliothek *NetworkX* [Sch08]. Mit NetworkX können GraphML-Daten eingelesen und Graphenobjekte direkt erzeugt werden. Außerdem stellt NetworkX diverse Operationen und Algorithmen der Graphentheorie zur Verfügung. Die Graphen bleiben im Hauptspeicher für die gesamte Berechnungsdauer.

Die Berechnung wird je nach den angegebenen *PropertyDef*-Attribut in der GraphML Nachricht in einem eigenen Prozess gestartet. Die Arbeitsweisen der einzelnen Analysealgorithmen werden in Kapitel 4 Modellanalyse detailliert beschrieben.

AYE Messaging Service

Der Messaging Service arbeitet mit *Rabbit MQ* als *Message Broker*. Nachrichten werden somit über das *Advanced Message Queuing Protocol (AMQP)* ausgetauscht.

Zum Einsatz kommen lediglich zwei Queues: Die Task-Queue nimmt GraphML Nachrichten zur Berechnung von Analysen an, die der *AYE Computation Worker* Service abholt. Die Properties-Queue nimmt JSON Nachrichten mit den Lösungen einer Analyse entgegen, die der *AYE Analyse* Service konsumiert.

AYE Web UI Service

Die Benutzeroberfläche (UI) wurde mit *Angular 6.x* und *TypeScript* realisiert. Das Layout der Seite hält sich an das *Material Design* von Google [Goo17].

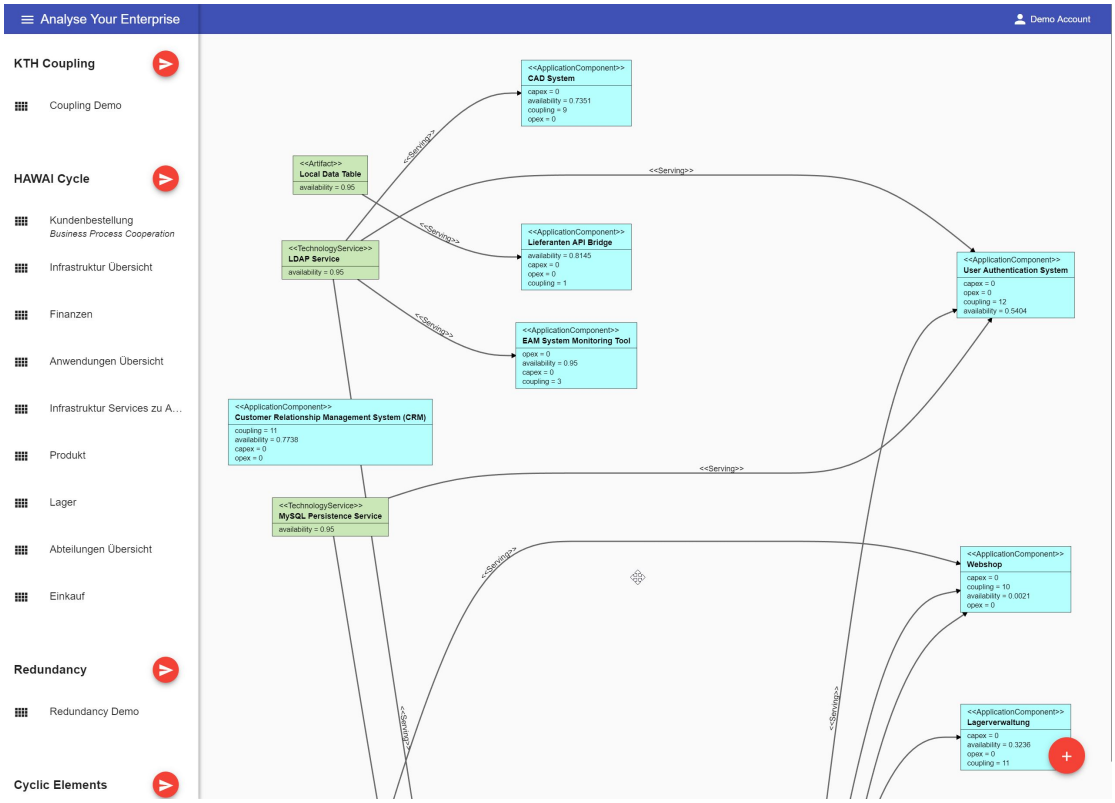


Abbildung 3.5: *Screenshot der AYE Benutzeroberfläche*

Eine Herausforderung lag in der Darstellung von ArchiMate-Modellen mit Interaktionsmöglichkeiten auf Elementen und Relationen in einem HTML5 Canvas, die mit *TypeScript* zusammenarbeiten. Nach dem Versuch einer Eigenentwicklung und Experimenten

mit verschiedenen Bibliotheken, darunter *JointJS*, *GoJS* und *DS.js*, konnte mit *NGX-Graph 5.x* eine passende Auswahl getroffen werden. *NGX-Graph* hat zwei Einschränkungen, die in zukünftigen Versionen laut Entwickler nachgebessert werden sollen: Die Positionierung auf dem HTML5 Canvas kann nicht angegeben werden, die Elemente positionieren sich automatisiert. Außerdem lässt sich das Klick- und Move-Event eines Elements nicht unterscheiden.

Der *AYE Web UI* Service holt sich mit dem Login des Benutzers ein JWT über die *OpenID Connect API* und interagiert anschließend bei jeder Aktion des Anwenders mit der *Analyse API*, wodurch das JWT immer mitgegeben wird. Das JWT aktualisiert sich in regelmäßigen Abständen im Hintergrund neu (refresh). Ohne Aktivität des Benutzers läuft das Token und somit die Session nach einiger Zeit aus. Diese Funktionalitäten wurden in zwei zentralen Komponenten gekapselt. Eine dieser Komponenten kümmert sich um die Interaktionen mit der *OpenID Connect API* und dem Aktualisieren des JWT. Die andere Komponente ist für die *Analyse API* zuständig und hängt das JWT an jede Anfrage.

AYE Swagger UI Service

Dieser Service verwendet *Swagger UI 2.x* [Sma16] zur aufbereiteten Dokumentation und Ausführung einzelner Operationen der *Analyse API*.

Swagger UI benötigt Erweiterungen zur Autorisierung und Authentifizierung, die eine Zusammenarbeit mit der *OpenID Connect API* ermöglichen (siehe Abbildung 3.4). Dadurch lassen sich die Operationen auf der *Analyse API* ausführen, die immer ein JWT voraussetzen.

HAWAI Authentication Service

Dieser Service verwendet *Keycloak*, eine Open Source Software von Red Hat, die *OpenID Connect* als Web-Schnittstelle realisiert. Über einen *Java Wildfly Application Server* stellt dieser Service neben der REST/HTTP-Schnittstelle auch eine Weboberfläche bereit, in der Benutzer, Rollen, erlaubte Clients und vieles mehr verwaltet und konfiguriert werden können. Da der *HAWAI Authentication Service* auch den Login- und Registrierungsprozess implementiert, kann pro Client dafür eine eigene Optik gestaltet und integriert werden.

3.3.2 Stand der Entwicklung

Das Projekt AYE konnte zu einem großen Teil realisiert werden. Die Benutzeroberfläche enthält nicht alle Funktionalitäten, die die *Analyse API* zur Verfügung stellt. Zum Beispiel gibt es keine Möglichkeit zur Modellierung von ArchiMate Modellen. Diese müssen über die Import-Funktion hinzugefügt werden. Außerdem lassen sich über die Benutzeroberfläche keine Elemente und Relationen ausschließen. Im Folgenden ein Überblick über die zu Anfang aufgestellten Anforderungen (siehe 3.1 Anforderungen) und deren Fertigstellungsgrad:

Anforderung	Beschreibung	Zustand
A1	AYE arbeitet mit dem ArchiMate 3.x Metamodell.	100%
A2	Die Algorithmen arbeiten mit einstellbaren Elementen und Relationen. Die Einstellungen können jedoch nicht auf der Benutzeroberfläche bearbeitet werden.	75%
A3	Kopplung, Verfügbarkeit und Kosten wurden umgesetzt.	100%
A4	Ein Import von ArchiMate Modellen ist möglich.	100%
A5	AYE verwendet Standardwerte für alle Analysen.	100%
A6	Die gewünschten Analysearten können auf einem Modell angegeben werden.	100%
A7	Die Analysen arbeiten parallel über mehrere Worker-Prozesse und jeder Worker erstellt einen Unterprozess für jede entgegengenommene Analyseart.	100%
A8	Die Analyse API unterstützt die Modellierung von ArchiMate, die Benutzeroberfläche bietet jedoch keine Funktionen dafür an.	50%
A9	Bei einem Import wird das Metamodell geprüft. Sind nicht erlaubte Elemente und Beziehungen enthalten, wird kein Modell in AYE angelegt. Es wurden keine Markierungen implementiert.	0%
A10	Erweiterungsmöglichkeiten des ArchiMate-Metamodells wurden nicht umgesetzt.	0%
A11	Wurde als *HAWAI Authentication Service* umgesetzt.	100%

Anforderung	Beschreibung	Zustand
A12	Ausgewählte Eigenschaften können über die *Analyse API* gesetzt werden. Beim Verändern des Modells können sich die technischen Schlüssel ändern, deswegen sollte hier noch ein manuell eintragbarer Identifikator gesetzt werden können.	80%
A13	AYE wurde serviceorientiert entwickelt und verwendet das *Continuous Integration/Delivery* System von *GitLab*.	100%
A14	AYE läuft produktiv auf der ICC der HAW Hamburg[3]	100%

[3] *Analyse Your Enterprise (AYE)* kann unter https://analyse-your-enterprise.de/ mit dem Demo-Zugang (Benutzername: *demo*, Kennwort: *demo*) ausprobiert werden.

4 Modellanalyse

Dieses Kapitel betrachtet die Analyse-Algorithmen von *AYE* und geht auf besondere Muster in Modellen ein, die sich zum Beispiel durch Zyklen oder lange Ketten aus Elementen äußern. Außerdem werden Erweiterungsmöglichkeiten, Abwandlungen und Grenzen dieser Analysen diskutiert.

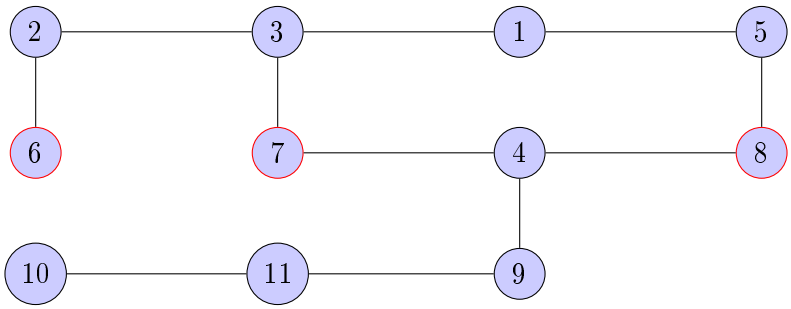
4.1 Algorithmen

Die Algorithmen wurden anhand der Beschreibungen aus den Veröffentlichungen [Joh+13; Lag+17] und Experimenten mit *EAAT* für ArchiMate abgeleitet und neu implementiert.

Jeder Algorithmus arbeitet mit Graphen, die bereits eingeschränkt wurden und in denen unbeteiligte Elemente und Relationen nicht vorkommen können.

4.1.1 Coupling

Die Kopplungsanalyse verwendet einen ungerichteten Graphen als Abstraktion eines ArchiMate-Modells. Der folgende abgeleitete Graph zeigt das Beispielmodell aus Abbildung 2.9:



Die rot umrandeten Knoten (6, 7 und 8) beschreiben Elemente, die Informationen austauschen und sich somit gegenseitig koppeln können, beispielsweise Anwendungen. Im folgenden werden diese Knoten *Kopplungselemente* genannt.

Schwarz umrandete Knoten leiten Informationen weiter, beispielsweise Funktionen der beteiligten Anwendungen.

Der Algorithmus bildet eine Menge aus geordneten Paaren aller Kombinationen der *Kopplungselemente* (rot markiert). Bezugnehmend auf den Beispielgraphen würde die Menge wie folgt aussehen:

$$C(s,t) = \{(6,7),(6,8),(7,8)\}$$

In einer Iteration über die Menge $C(s,t)$ wird ein Weg zwischen Quelle s und Ziel t gesucht, der nicht über ein *Kopplungselement* führt. Existiert ein Weg, dann wird der Kopplungswert auf Quelle und Ziel um 1 erhöht.

Die Wegfindung wird über den Dijkstra-Algorithmus [Dij59] gelöst mit Kantengewichtungen von 1. Die Suche nach den kürzesten Wegen verhindert gefundene Wege mit Zyklen zwischen s und t. Der Algorithmus terminiert somit in jedem Fall und findet ein deterministisches Ergebnis.

Algorithm 1: Kopplung

Input: G = ungerichteter Graph

Output: Liste der Kopplungselemente e mit Kopplungswert *coupling*

$e \leftarrow$ Kopplungselemente in G
while e **do**
  $coupling(e) \leftarrow 0$ ▷ Kopplungselemente e initialisieren
end

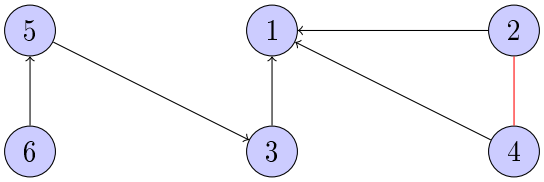
$C(s,t) \leftarrow combinations(e)$
while *C(s, t)* **do**
  $E \leftarrow e \setminus \{s,t\}$ ▷ Kopplungselemente ohne s und t
  $H \leftarrow G \setminus E$ ▷ Teilgraph H ohne Kopplungselemente E
  if $shortest\_path(s,t)$ in H **then**
    $coupling(s) \leftarrow coupling(s) + 1$
    $coupling(t) \leftarrow coupling(t) + 1$
end

Das Ergebnis des Beispielgraphen würde wie folgt aussehen:

$$properties = \{coupling(6) = 2, coupling(7) = 2, coupling(8) = 2\}$$

4.1.2 Availability

Die Verfügbarkeitsanalyse verwendet einen gerichteten Graphen als Abstraktion von ArchiMate-Modellen. Der folgende abgeleitete Graph G zeigt das Beispielmodell aus Abbildung 2.10:



Die rot markierte Kante kennzeichnet eine Redundanz zwischen Knoten 2 und 4. Aus Graph G wird zunächst ein zyklenfreier Graph H ohne Redundanzbeziehungen erzeugt. Graph H des Beispielgraphen würde daher keine rot markierte Kante enthalten.

Redundanzbeziehungen verfolgen
Die beiden Knoten einer Redundanzbeziehung (Quelle und Ziel) werden bis zu einem verzweigenden oder zusammenführenden Knoten bzw. Wurzelknoten verfolgt. Dieser letzte Knoten in der Reihe wird ungerichtet in einer Liste abgelegt. Da eine Redundanzbeziehung ungerichtet ist, wird diese immer durch zwei Tupel beschrieben. Zum Beispiel: `redundant` $= \{(2, 4), (4, 2)\}$.

Die Verfolgung der Redundanzen ist notwendig, weil die Semantik in verketteten Elementen verloren geht. Dies zeigt das konkrete Beispiel in Abbildung 4.1.

Gegeben sei eine `observedAvailability` von *0,95* in den Elementen *Inventory Service*, *VISA* und *PayPal*. Die Komponenten *VISA* und *PayPal* reichen ihren Verfügbarkeitswert an die realisierten Services weiter, da keine *OR*-Verknüpfung vorliegt. Auf *BookSales* kommt es daher zu folgender Berechnung:

$$A_{avg} = 0.95 * 0.95 * 0.95 \approx 0.8574$$

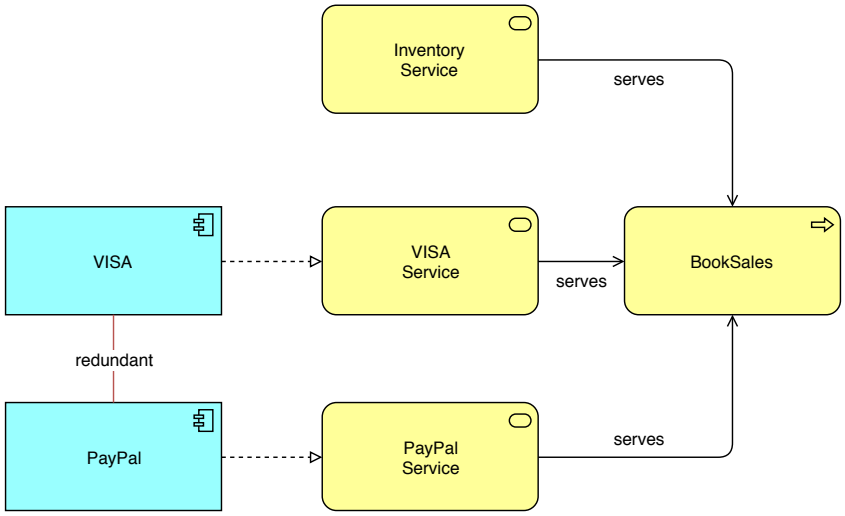


Abbildung 4.1: *Semantik einer Redundanzbeziehung bei verketteten Elementen*

Verschieben sich die Redundanzbeziehungen auf Knoten, die mehrere ausgehende Kanten oder deren Nachfolger eingehende Kanten zusammenführen, dann kann die Semantik erhalten werden. Im Modellbeispiel (Abbildung 4.1) würde sich die Redundanzbeziehung von *VISA* und *PayPal* auf *VISA Service* und *PayPal Service* verschieben. Für BookSales ergibt sich nun die folgende Berechnung:

$$A_{avg} = 0.95 * (1 - (1 - 0.95) * (1 - 0.95)) \approx 0.9476$$

Zyklen auflösen

Die Zyklen in Graph H werden mithilfe der Tiefensuche über Querkanten erkannt. Jeder gefundene zyklische Weg wird durch einen neuen Knoten abgebildet, der als Attribut den zyklischen Weg enthält, zum Beispiel `cycle` $= \{1, 2, 3, 1\}$. Eingehende und ausgehende Kanten aus dem Zyklus werden zum neuen Knoten umgeleitet und die zyklischen Knoten entfernt. Die Suche nach Zyklen wird solange fortgesetzt, bis kein Zyklus in Graph H auffindbar ist.

Fault Tree erzeugen

Der rekursive Aufruf zur Erstellung eines *Fault Trees* auf jedem Knoten (siehe 2.2.2.2 Availability) beginnt bei den Quellknoten ohne Eingangskanten $d_H^-(v) = 0$. Jeder besuchte Knoten trägt den eigenen Verfügbarkeitswert `availability` beim Nachfolger in eine `child`-Liste ein. Ist ein `observedAvailability` Attribut vorhanden, wird

dessen Wert eingetragen. Ist kein Verfügbarkeitswert existent (typischerweise bei einem Quellknoten), wird ein Standardwert genommen. Haben sich alle Kindelemente beim Nachfolger gemeldet, wird der *Fault Tree* erzeugt und der neue Verfügbarkeitswert berechnet. Anschließend setzt sich die Rekursion fort.

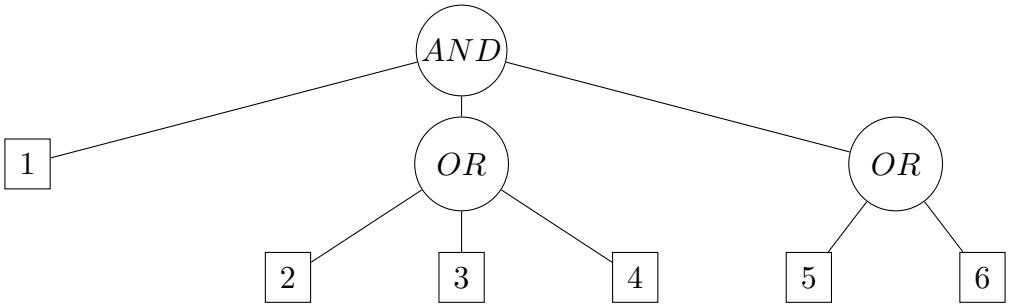
Der *Fault Tree* kann anhand der Kindelemente eines besuchten Knotens und der Liste aus Redundanzbeziehungen (`redundant`) erzeugt und die Verfügbarkeit berechnet werden. Angenommen die Redundanzliste enthält folgende Werte:

$$\texttt{redundant} = \{(3,4),(4,3),(5,6),(6,5),(2,3),(3,2),(4,10),(10,4),(3,11),(11,3)\}$$

Die Kindelemente auf dem besuchten Knoten haben sich wie folgt eingetragen (zweiter Wert im Tupel gibt die Verfügbarkeit an):

$$\texttt{children} = \{(1,0.6),(2,0.6),(3,0.6),(4,0.6),(5,0.5),(6,0.5)\}$$

Der *Fault Tree* ergibt sich beim Durchlaufen der Kindliste. Kindelement 2 wird zum Beispiel in der Redundanzmenge gefunden und die Nachfolger sind: $(2,3,4,10,11)$. Elemente 10 und 11 sind keine Kindelemente, somit bleiben $(2,3,4)$. Für jedes Kindelement ergibt sich folgender *Fault Tree* mit Berechnung:



$$A_{avg} = 1 \wedge (2 \vee 3 \vee 4) \wedge (5 \vee 6)$$

$$A_{avg} = 0.6 * (1 - (1 - 0.6) * (1 - 0.6) * (1 - 0.6)) * (1 - (1 - 0.5) * (1 - 0.5)) \approx 0.4212$$

Die Knoten, die einen Zyklus abbilden, berechnen ihren rekursiv erhaltenen Verfügbarkeitswert über die AND-Verknüpfung. Zum Beispiel: Enthält der Knoten einen Verfügbarkeitswert von 0.95 und einen zyklischen Weg über drei Knoten $\{1,2,3,1\}$, dann ergibt sich daraus folgende Rechnung:

$$A_{avg} = 0.95^3 \approx 0.8574$$

Ist die Rekursion abgeschlossen, hat jedes Element in Graph H einen Verfügbarkeitswert. Anschließend müssen diese Werte auf Graph G übertragen und die Knoten zur Abbildung von Zyklen aufgelöst werden.

Algorithmus

Im nun folgenden Algorithmus zur Berechnung von Verfügbarkeit wurden Funktionen vereinfacht um den Ablauf übersichtlich zu halten:

`replaceNodes(cyclicNodes, nodeName):` Ersetzt die in `cyclicNodes` enthaltenen Knoten durch einen neuen Knoten, identifizierbar durch den `nodeName`. Die eingehenden und ausgehenden Kanten der `cyclicNodes` werden auf den neuen Knoten übertragen.

`buildFaultTree(node.children, redundantEdges):` Erzeugt einen *Fault Tree* aus den Kind-Elementen eines Knotens und den Redundanzknoten, wie im letzten Abschnitt beschrieben. Das Ergebnis ist eine Menge aus AND-verknüpften Knoten und Untermengen, die OR-verknüpft sind. Zum Beispiel: $\{1, 2, \{3, 4, 5\}, \{6, 7\}\}$.

Algorithm 2: Availability (Teil 1)

Input: G = gerichteter Graph

Output: Elementliste e mit Verfügbarkeitswerten *availability*

$redundantEdges \leftarrow \emptyset$

$H \leftarrow G$ ▷ Erzeugt eine Kopie von G auf H

$RemoveRedundantEdges()$

$ReplaceCyclesWithNodes()$

$AggregateThroughGraph()$

$AssignToOriginalGraph()$

Function `RemoveRedundantEdges()`:

```
redundant ← ∅
foreach edge in H.edges() do
    if edge of type redundant then
        H.edges() ← H.edges() \ {edge}        ▷ Entferne edge aus H
        redundant ← redundant ∪ {edge}
end
foreach edge in redundant do
    source, target ← edge                     ▷ edge ist ein Tupel aus (source, target)
    source ← LastBranchNode(source)
    target ← LastBranchNode(target)
    redundantEdges ← redundantEdges ∪ {(source, target)}
end
return
```

end

Function `LastBranchNode(node)`:

```
successor ← H.successor(node)
if H.outDegree(node) = 1 ∧ H.inDegree(successor) = 1 then
    return LastBranchNode(successor)
return node
```

end

Algorithm 2: Availability (Teil 2)

```
Function ReplaceCyclesWithNodes():
    while cycle ← H.findCycle do
        cyclicNodes ← {∀ node in cycle}
        nodeName ← randomString()
        H.addNode(nodeName, type ← 'cyclic', cyclicNodes)
        H.replaceNodes(cyclicNodes, nodeName)
    end
    return
end

Function AggregateThroughGraph():
    leafNodes ← {∀x in H.nodes() if H.outDegree(x) > 0 ∧ H.inDegree(x) = 0}
    foreach node in leafNodes do
        AggregateThroughGraphRec(node)
    end
    return
end

Function AggregateThroughGraphRec(node):
    if H.inDegree(node) = |node.children| then
        if node.observedAvailability then
            node.availability ← node.observedAvailability
        else if |node.children| > 0 then
            node.availability ← ComputeAvailability(node)
        else
            node.availability ← DEFAULT
        end
    foreach successor in H.successor(node) do
        successor.children ← node.availability
        AggregateThroughGraphRec(successor)
    end
    return
end
```

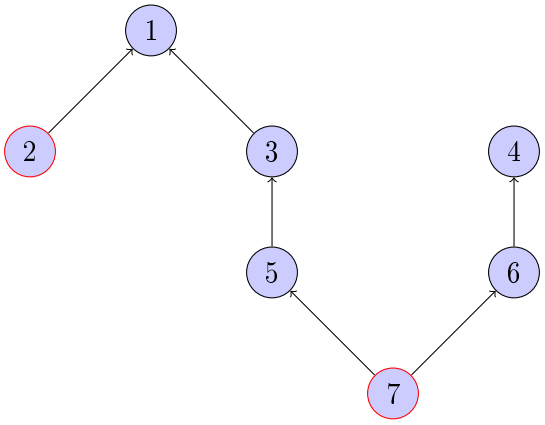
Algorithm 2: Availability (Teil 3)

```
Function ComputeAvailability(node):
    faultTree ← BuildFaultTree(node.children, redundantEdges)
    result ← 1
    foreach andElem in faultTree do
        if andElem of type Set then
            subResult ← 1
            foreach orElem in andElem do
                subResult ← subResult * (1 − orElem.availability)
            end
            result ← result * (1 − subResult)
        else
            result ← andElem.availability
        end
    end
    if node of type cyclic then
        result ← res^|node.cyclicNodes|
    return result
end

Function AssignToOriginalGraph():
    foreach node in H.nodes() do
        if node of type cyclic then
            foreach nodeCyclic in node.cyclicNodes do
                G.nodes(nodeCyclic).availability ← node.availability
            end
        else
            G.nodes(node).availability = node.availability
        end
    end
    return
end
```

4.1.3 Service Cost

Die Kostenanalyse verwendet einen gerichteten Graphen als Abstraktion von ArchiMate-Modellen. Der folgende abgeleitete Graph G zeigt das Beispielmodell aus Abbildung 2.11:



Die rot umrandeten Knoten (2 und 7) beschreiben Elemente, die einmalige Kosten (*Capex*) und laufende Betriebskosten (*Opex*) vorgeben. Schwarz umrandete Knoten (beispielsweise Schnittstellen, Funktionen oder Geschäftsprozesse) übernehmen Kosten über die Verzweigungen ihrer Kind-Elemente.

Aus Graph G wird wie in der Verfügbarkeitsanalyse ein zyklenfreier Graph H erzeugt (siehe 4.1.2 Availability).

Der Algorithmus beginnt rekursiv bei den Quellknoten ohne Eingangskanten $d_H^-(v) = 0$. Jeder besuchte Knoten teilt den eigenen *Capex*- bzw. *Opex*-Wert durch die Anzahl ausgehender Kanten und gibt diesen Wert an die Elternknoten weiter. Wenn keine Kostenangaben verfügbar sind, wird ein festgelegter Standardwert genommen. Beim Elternknoten werden diese Werte in entsprechende `capex` bzw. `opex` Listen geschrieben. Wird einer dieser Elternknoten besucht, summieren sich die Kostenwerte der jeweiligen Listen.

Knoten, die einen Zyklus abbilden, summieren die Kosten der am Zyklus beteiligten Knoten und teilen diese anschließend durch die Anzahl zyklischer Knoten plus ausgehender Kanten. Das Ergebnis wird an die Elternknoten weitergereicht. Ein Beispiel findet sich in Abbildung 4.2.

Die Berechnung von *Capex* und *Opex* für den *Excel Converter Service* berechnet sich hier wie folgt:

$$capex = \frac{10000 + 1000 + 1000}{3 + 2} = 2400$$

$$opex = \frac{500 + 100 + 100}{3 + 2} = 140$$

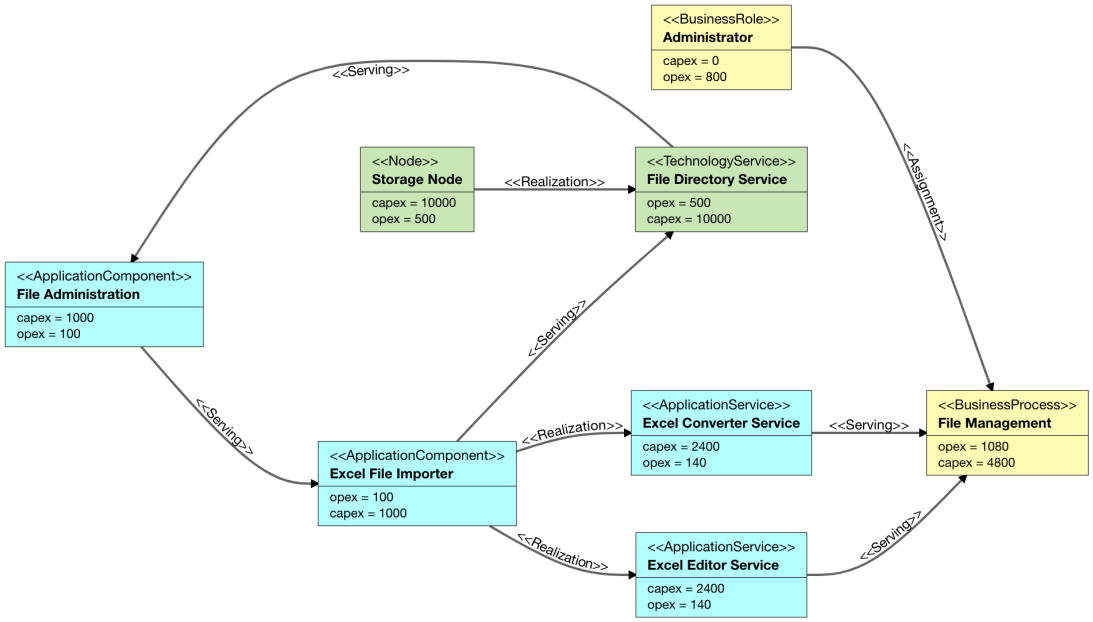


Abbildung 4.2: *Kostenanalyse mit zyklischen Knoten*

Algorithm 3: Service Cost (Teil 1)

Input: G = gerichteter Graph

Output: Elementliste e mit Kosten *capex* und *opex*

$e \leftarrow$ Kostenelemente in G ▷ Z.B. *Node*, *ApplicationComponent* oder *BusinessRole*

$H \leftarrow G$ ▷ Erzeugt eine Kopie von G auf H

$ReplaceCyclesWithNodes()$ ▷ Identisch mit *Availability*-Algorithmus

$AggregateThroughGraph()$

$AssignToOriginalGraph()$

Function `AggregateThroughGraph()`**:**

- $leafNodes \leftarrow \{\forall x$ in $H.nodes()$ if $H.outDegree(x) > 0 \wedge H.inDegree(x) = 0\}$
- **foreach** *node* in $leafNodes$ **do**
  - $AggregateThroughGraphRec(node)$
- **end**
- **return**

end

Function `AggregateThroughGraphRec(`*node*`)`**:**

- **if** $H.inDegree(node) = |node.children|$ **then**
  - **if** *node.type* in e **then**
    - **if** $\neg node.capex$ **then** $node.capex \leftarrow$ CAPEX-DEFAULT
    - **if** $\neg node.opex$ **then** $node.opex \leftarrow$ OPEX-DEFAULT
    - $node.capex \leftarrow node.observedAvailability$
  - **else if** $|node.children| > 0 \vee$ *node* in *cyclicNodes* **then**
    - $node.capex = sum(node.children.capex)$
    - $node.opex = sum(node.children.opex)$
  - **foreach** *successor* in $H.successor(node)$ **do**
    - $successor.children.capex \leftarrow ComputeParentCosts(capex, node)$
    - $successor.children.opex \leftarrow ComputeParentCosts(opex, node)$
    - $AggregateThroughGraphRec(successor)$
  - **end**
- **return**

end

Algorithm 3: Service Cost (Teil 2)

```
Function ComputeParentCosts(costType, node):
    if costType = capex then
        cost ← node.capex
    else
        cost ← node.opex
    end
    if node in cyclicNodes then
        foreach nodeInG in node.cyclicNodes do
            if nodeInG.type in e then
                if costType = capex then
                    cost ← cost + sum(nodeInG.capex)
                else
                    cost ← cost + sum(nodeInG.opex)
                end
            end
        return cost / (H.outDegree(node) + |node.cyclicNodes|)
    return cost / H.outDegree(node)
end

Function AssignToOriginalGraph():
    foreach node in H.nodes() do
        if node of type cyclic then
            foreach nodeInG in node.cyclicNodes do
                if nodeInG.type not in e then
                    nodeInG.capex ← node.capex
                    nodeInG.opex ← node.opex
            end
        else
            G.nodes(node).capex = node.capex
            G.nodes(node).opex = node.opex
        end
    end
    return
end
```

4.2 Evaluierung

In diesem Unterkapitel werden einige Muster und für die Algorithmen kritische Situationen in konkreten Modellen überprüft. Mit den meisten dieser Konstellationen können die Algorithmen umgehen.

4.2.1 Lose Kopplung

Bei der Modellierung von loser Kopplung kommunizieren abhängige Elemente über eine Schnittstelle oder Service. Am Beispielmodell von *MAP* (Abbildung 2.9) lässt sich ablesen, dass *ApplicationFunctions* Informationen bzw. Daten ungerichtet weitergeben. Dieses Verhalten lässt sich in *AYE* ebenfalls beobachten, beispielsweise in Abbildung 4.3. Die Anwendungen auf der rechten Seite zeichnen sich durch hohe Kopplungswerte aus.

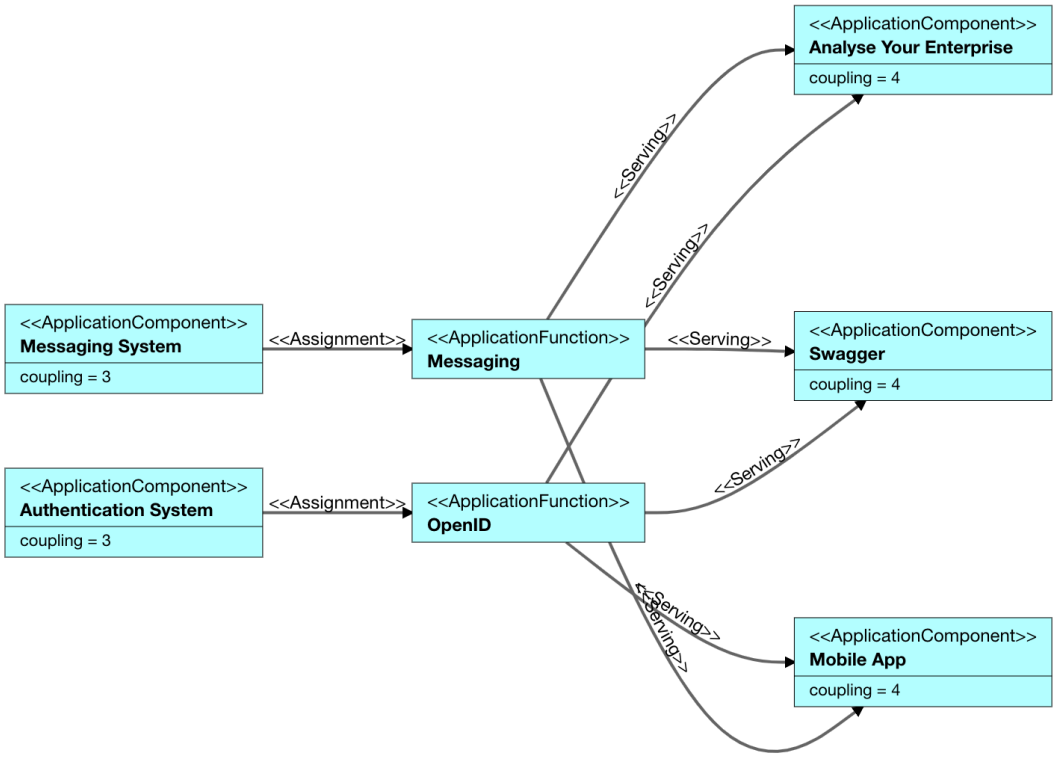


Abbildung 4.3: *Kopplung über ApplicationFunction*

Anders ist dies bei der Verwendung von *ApplicationInterface* bzw. *ApplicationService* statt einer *ApplicationFunction* (siehe Abbildung 4.4).

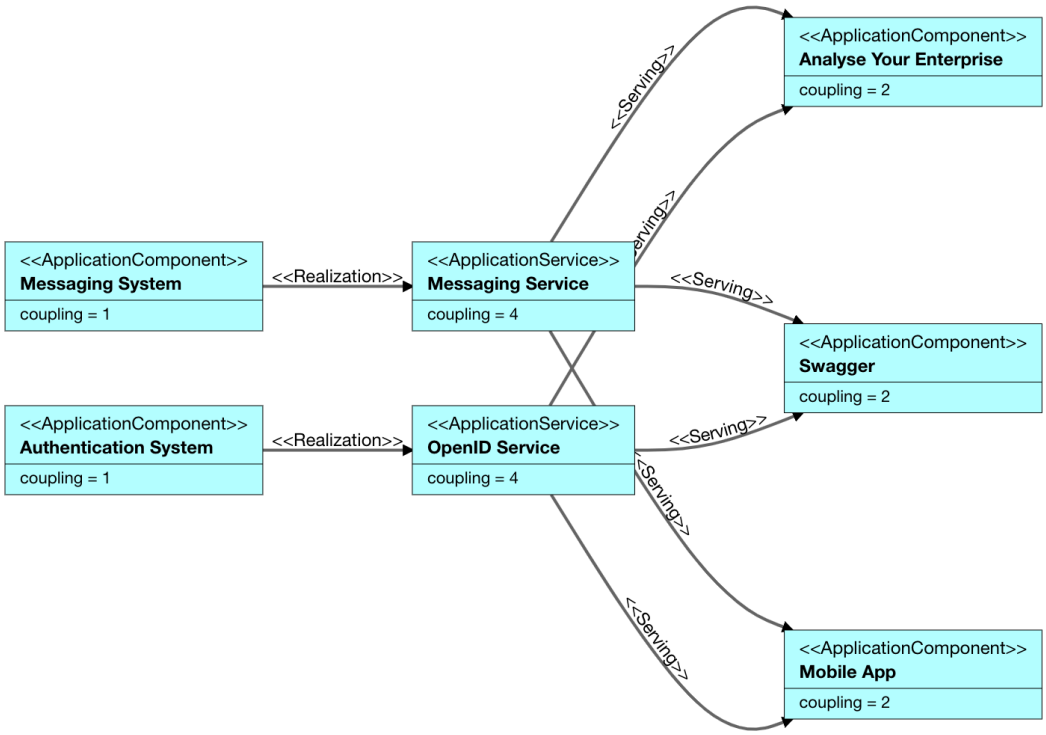


Abbildung 4.4: *Lose Kopplung über ApplicationService*

Nach der größtenteils umgesetzten Anforderung A2 können die Elemente eingestellt werden, die Kopplungselemente darstellen. Voreingestellt sind die folgenden Elementarten von *ArchiMate*:

- ApplicationComponent
- ApplicationInterface
- ApplicationService
- BusinessInterface
- BusinessService
- TechnologyInterface
- TechnologyService

Lose Kopplung lässt sich somit ohne weitere Einstellungen auf den drei Ebenen von *ArchiMate Core* abbilden.

4.2.2 Ableitungen von Redundanzbeziehungen

Redundanzbeziehungen kommen in der Verfügbarkeitsanalyse vor. Sie sind nicht Bestandteil von ArchiMate und wurden in Hinblick auf eine Erweiterungsmöglichkeit des ArchiMate-Metamodells für *AYE* (Anforderung A10) provisorisch als Assoziation mit Bezeichnung `redundant` hinzugefügt. Die Redundanzbeziehung spielt eine zentrale Rolle in der Verfügbarkeitsanalyse, da sie die *OR*-Verknüpfung repräsentiert.

Bei der Übertragung in die Verfügbarkeitsanalyse von *AYE* stellt sie eine Herausforderung dar, denn durch die vielen Modellierungsmöglichkeiten verliert die Redundanzbeziehung ihre Bedeutung (siehe Beispiel aus Abbildung 4.1).

Die in *AYE* gewählte Lösung ist die Verschiebung der Redundanzbeziehung im Algorithmus - es wird somit eine Redundanzbeziehung abgeleitet. Die Ableitung wird nur dann vorgenommen, wenn keine Verzweigungen oder Zusammenführungen vorliegen.

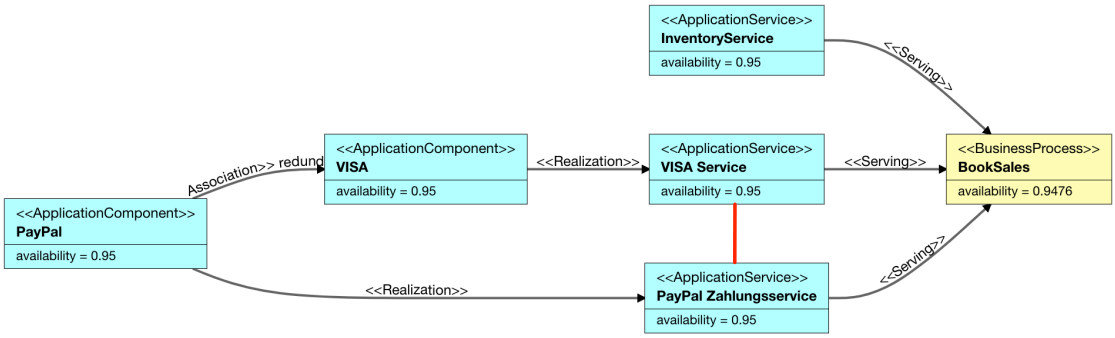


Abbildung 4.5: *Abgeleitete Redundanzbeziehung*

Abbildung 4.5 zeigt ein Positivbeispiel aus *AYE*, in dem die abgeleitete Beziehung rot markiert wurde und zu einer *OR*-Verknüpfung im *Fault Tree* von *BookSales* führt.

Verzweigungen kommen in größeren Modellen häufiger vor, deswegen ist die Wahrscheinlichkeit hoch, dass sich die Bedeutung der Redundanzbeziehung verlieren kann. Abbildung 4.6 zeigt einen fast identischen Graphen. Durch die Verzweigung auf dem *PayPal*-Element in Richtung *PayPal Rechnungen* und *PayPal Zahlungsservice* kann die Redundanzbeziehung nicht weiter verschoben werden und auf BookSales ergibt sich lediglich eine *AND*-Verknüpfung.

Eine mögliche Lösung ist das kontinuierliche Ableiten der Redundanzbeziehungen in Richtung der Wurzeln des Graphen, bis eine Zusammenführung gefunden wird. Dadurch

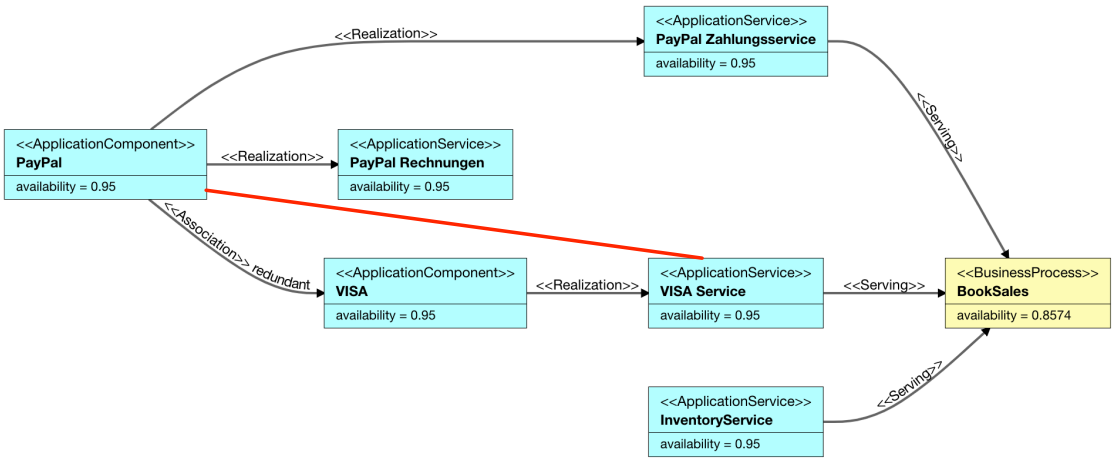


Abbildung 4.6: *Verschachtelte Zyklen*

würden im Beispielgraphen (Abbildung 4.6) folgende Redundanzbeziehungen entstehen: *(PayPal Rechnungen, VISA), (PayPal, VISA Service), (PayPal Rechnungen, VISA Service), (PayPal Zahlungsservice, VISA), (PayPal Zahlungsservice, VISA Service)*. Von allen diesen abgeleiteten Beziehungen führt nur die *(PayPal Zahlungsservice, VISA Service)*-Beziehung zu einer *OR*-Verknüpfung im *Fault Tree* von *BookSales*.

Die andere mögliche Lösung ist das direkte Setzen von Redundanzbeziehungen vor Zusammenführungen. Dadurch bleibt die Berechnung für den Modellierenden nachvollziehbar. Zudem müssen Redundanzen in der finalen Version von *AYE*, genauso wie in *MAP*, speziell für die Analyse gesetzt werden, denn ArchiMate kennt diese Art Beziehung nicht. Die Verschiebung der Redundanzen ist somit eine simple Zwischenlösung.

4.2.3 Überlappende Zyklen

Die Algorithmen lösen Zyklen auf, indem sie die beteiligten Knoten zusammenfassen. Zusammengefasste Knoten können anschließend erneut an einem Zyklus beteiligt sein und weitere Zusammenfassungen sind nötig.

Dieses Szenario zeigt Abbildung 4.7 am Beispiel einer Verfügbarkeitsanalyse aus *AYE*.

Die zwei überlappenden Zyklen müssen vom Algorithmus nacheinander aufgelöst werden. Bei der Betrachtung des Elements *Invoice UI* ergibt sich folgende Rechnung:

$$A_{InvoiceUI} = (0.95 * 0.95 * 0.95)^6 = 0.8574^6 \approx 0.3972$$

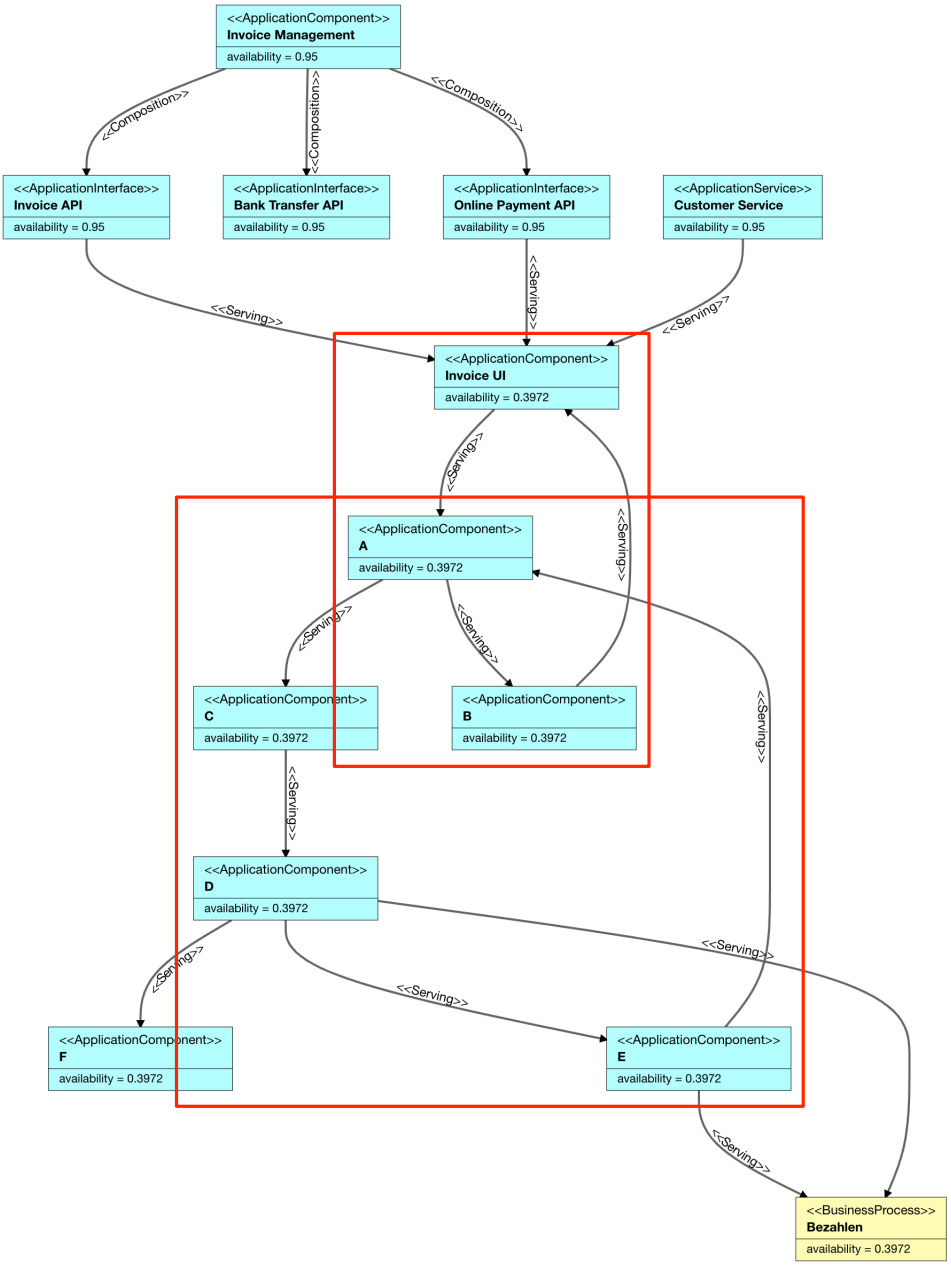


Abbildung 4.7: *Zwei überlappende Zyklen*

Die drei Eingangskanten aus *Invoice API*, *Online Payment API* und *Customer Service* zu je 0.95 als Verfügbarkeitswert werden zunächst multipliziert. Da *Invoice UI* in einem Zyklus aus insgesamt 6 Teilnehmern liegt, trifft erneut die *AND*-Verknüpfung zu und der zugeordnete Verfügbarkeitswert von 0.8574 wird mit der Anzahl im Zyklus enthaltener

Knoten potenziert.

Der Algorithmus kann somit Zyklen korrekt auflösen und berechnen.

4.2.4 Zyklen in Redundanzbeziehungen

Redundanzbeziehungen können sich ebenfalls in Zyklen befinden, auch wenn dies in konkreten Modellen eher selten vorkommen wird. Die Auflösung dieser Situation soll dennoch betrachtet werden.

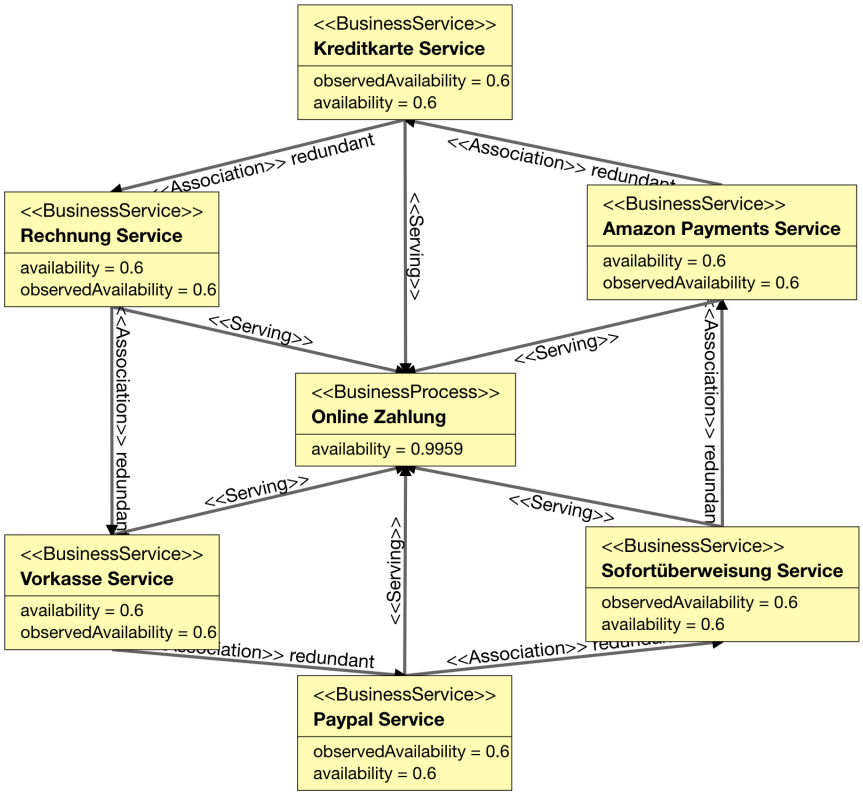


Abbildung 4.8: *Zyklus aus redundanten Elementen*

Abbildung 4.8 zeigt einen Rendundanz-Zyklus um ein Element herum. Der Algorithmus erzeugt eine Menge aus redundanten Beziehungen und leitet gemeinsam mit den Kind-Elementen von *Online Zahlung* einen *Fault Tree* ab. Wird diese Ableitung auf Basis einer Menge von *OR*-Verknüpfungen geführt, dann wird automatisch eine Kante ignoriert. Auf *Online Zahlung* ergibt sich damit folgende Rechnung:

$$A_{OnlineZahlung} = (1-(1-0.6)*(1-0.6)*(1-0.6)*(1-0.6)*(1-0.6)*(1-0.6)) \approx 0.9959$$

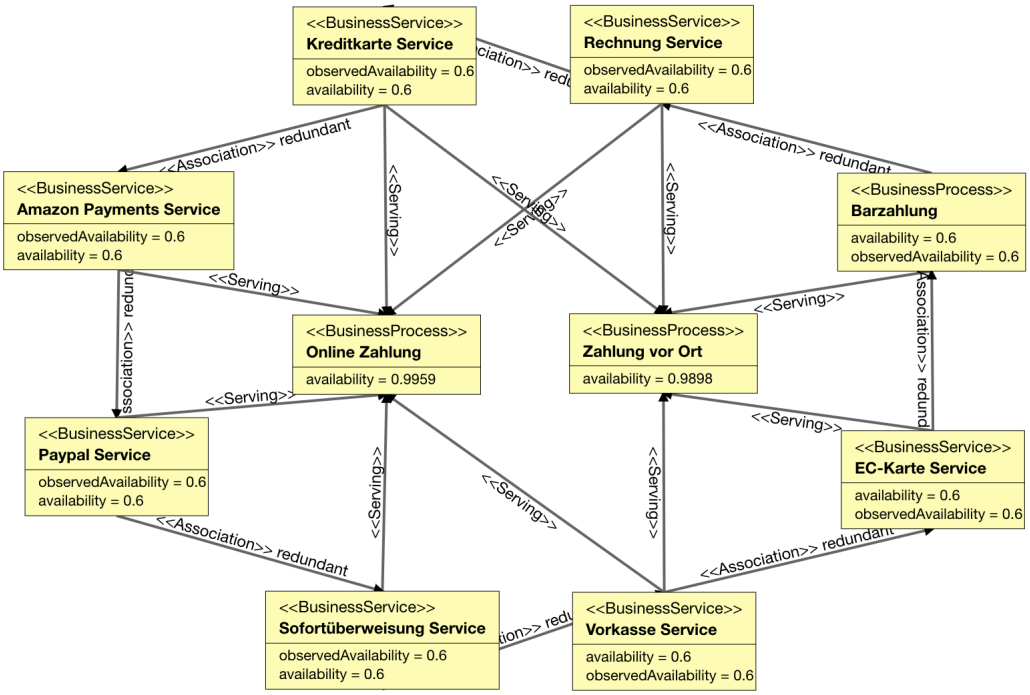


Abbildung 4.9: *Zyklus mit Überschneidungen*

Abbildung 4.9 besitzt nun zwei Elemente um einen Zyklus herum, die sich gegenseitig überschneiden. Für die zentralen Elemente ist der Zyklus nicht relevant und keine Kante wird ignoriert. Für das Element *Online Zahlung* bleibt die Berechnung identisch, da die selben Kind-Elemente zueinander redundant sind. Für *Zahlung vor Ort* ergibt sich folgende Berechnung:

$$A_{ZahlungvorOrt} = (1-(1-0.6)*(1-0.6)*(1-0.6)*(1-0.6)*(1-0.6)) \approx 0.9898$$

4.2.5 HAWAICycle Beispielmodell

HAWAICycle ist ein umfangreicheres ArchiMate-Modell, dass die Unternehmensarchitektur eines ausgedachten Produktionsunternehmens für Fahrräder abbildet. Das Modell umfasst 170 Elemente und 245 Relationen.

In der Entwicklung von *AYE* wurde es verwendet, um die Ergebnisse und Laufzeiten der Algorithmen zu überprüfen. Da sich das *HAWAICycle*-Modell auf mehrere Sichten

verteilt und somit die Nachvollziehbarkeit der Berechnungen schwieriger wird, wurden nur Plausibilitätsprüfungen der Analysen durchgeführt.

Die Laufzeitmessungen hingegen konnten Aufschluss geben, in welchem Rahmen sich die Algorithmen bewegen. Alle anderen Beispielmodelle, die in dieser Arbeit verwendet wurden, hatten eine maximale Berechnungszeit von einer Millisekunde ohne Skalierung auf einer Node (Skylake Core i5 Prozessor mit 2,9 GHz, 16 GB LPDDR3-2133 Ram).

Die durchschnittlichen Laufzeiten bei diesem Modell für Kopplung lagen bei 1100, für Verfügbarkeit bei 220 und für Kosten bei 410 Millisekunden. Gemessen wurden nur die Algorithmen selbst ohne Übertragungswege.

Die schlechte Berechenbarkeit der Kopplungsanalyse ist im Dijkstra-Algorithmus begründet. Der Algorithmus ist jedoch nicht darauf angewiesen den kürzesten Weg zu finden. Es würde reichen, wenn ein einfacher Weg gefunden wird. Einfache Wege haben jedoch den Nachteil, dass sie sich in Zyklen verfangen können und somit ist ein Ergebnis nicht sichergestellt. An dieser Stelle kann die Auflösung von Zyklen erneut eingesetzt werden, die dieses Problem löst und die Laufzeit auf das Niveau der anderen Algorithmen bringt.

5 Zusammenfassung und Fazit

Diese Arbeit befasste sich zunächst mit den Modellierungssprachen *ArchiMate* und *Multi-Attribute Prediction (MAP)*. Mit ArchiMate lassen sich Unternehmensarchitekturen über eine Vielzahl an Elementen und sieben getrennte Layer detailliert modellieren. Das Metamodell von ArchiMate ist entsprechend umfangreich, besitzt selbst jedoch keinen spezifizierten Mechanismus für Analysen. *MAP* orientiert sich von der Syntax an ArchiMate und bietet im Gegensatz nur eine kleine Menge an Elementen und Relationen an, die für enthaltene Analysemechanismen nötig sind.

Mit dem Projekt *Analyse Your Enterprise (AYE)* sollte ArchiMate um entsprechende Analysemechanismen erweitert werden, die *MAP* bereits anbietet. Das Ziel war eine webbasierte Anwendung für den praktischen Einsatz. ArchiMate Modelle sollten dafür importierbar und lediglich durch Auswahl der gewünschten Analyseart brauchbare Ergebnisse ausgeben. Zu den auswählbaren Analysen gehören im entwickelten Prototyp: Kopplung, Verfügbarkeit und Kosten. Durch den Umfang von ArchiMate und die generische Berechnung der Analysen können diese Layer-übergreifend arbeiten und so zum Beispiel Kopplung über menschliche Schnittstellen berücksichtigen.

Für detailliertere Analysen sollten dem Anwender Funktionalitäten angeboten werden, um Element- und Beziehungsarten im Modell einzubeziehen oder auszuschließen. Die Schnittstellen von *AYE* erlauben außerdem, Werte aus Metriken externer Systeme zu übernehmen.

Die Architektur von *AYE* ist serviceorientiert und erweiterbar aufgebaut. Zudem wurde das Projekt über die *Informatik Compute Cloud (ICC)*, einem Kubernetes-Cluster der *Hochschule für Angewandte Wissenschaften (HAW)*, öffentlich zur Verfügung gestellt[1]. Dadurch sollen weitere Projekte und Abschlussarbeiten in Zusammenarbeit mit der Forschungsgruppe *HAW Laboratory for Architecture and IT-Management (HAWAI)* motiviert werden.

[1] *Analyse Your Enterprise (AYE)* kann unter https://analyse-your-enterprise.de/ mit dem Demo-Zugang (Benutzername: *demo*, Kennwort: *demo*) abgerufen werden.

Die Modellanalyse wurde durch die entwickelten Algorithmen dargestellt, die im Rahmen des ArchiMate Metamodells die Analyse von *MAP* realisieren. Die Kopplungsanalyse setzt zum Finden abhängiger Elemente auf den Dijkstra-Algorithmus. Die Verfügbarkeitsanalyse löst Zyklen auf, leitet Redundanzbeziehungen ab und baut rekursiv für jedes Element einen *Fault Tree* auf, über den das Ergebnis berechnet wird. Die Kostenanalyse löst ebenfalls Zyklen auf und multipliziert bei Verzweigung bzw. teilt bei Zusammenführung rekursiv die Kosten in Richtung der Wurzeln des Modells.

Die Herausforderungen lagen in der Auflösung von Zyklen und Ableitung von Redundanzbeziehungen. Die Evaluation wurde über kritische Situationen und Beispielmodelle geführt. Darüber hinaus wurde eine größere Unternehmensarchitektur als Beispielmodell eines virtuellen Herstellers von Fahrrädern (HAWAICycle) erstellt, um Laufzeiten zu messen.

5.1 Fazit

Der Prototyp von *AYE* ermöglicht es, ein bestehendes ArchiMate-Modell ohne viel Aufwand auf Kopplung, Verfügbarkeit und Kosten zu analysieren. Die Ergebnisse der Analysen sind einfache Indikatoren, die bei der Modellierung einer Unternehmensarchitektur helfen können, indem sie auf ungewollte Auswirkungen frühzeitig aufmerksam machen. Der praktische Nutzen von *AYE* entfaltet sich erst, wenn über die Anwendungsoberfläche eine Modellierung möglich wird. Erst dann können diese Auswirkungen von Veränderungen am Modell sofort sichtbar werden.

Die Veröffentlichungen zu *MAP* [Joh+13; Lag+17; Joh+16; LJE15; LE14; När+08; NJN07; Ost+13] ließen wenig Aufschluss über die verwendeten Algorithmen zu. Mit dieser Arbeit wurden drei Lösungen vorgestellt, die Modelle mit graphentheoretischen Methoden lösen und eine Vorlage für weitere Analysemöglichkeiten anbieten. Die Algorithmen sind auch auf andere Modellierungssprachen anwendbar, denn im Kern sind es gerichtete und ungerichtete Graphen, die nach Typen von Knoten filtern und Werte aggregieren.

In den Algorithmen liegt noch viel Optimierungspotential, so lässt sich bei der Kopplungsanalyse auf den Dijkstra-Algorithmus verzichten und bei der Verfügbarkeitsanalyse ist ein besserer Umgang mit Ableitungen von Redundanzbeziehungen möglich.

ArchiMate ist in frühen Planungsphasen von Ist- und Soll-Architekturen, zum Beispiel im gemeinsamen Einsatz mit *Architecture Development Method (ADM)* von *TOGAF* in

Phase A bis D, eine geeignete Modellierungssprache. Zusammen mit den hier vorgestellten Analysen können damit frühzeitig Auswirkungen auf Kopplung, Verfügbarkeit und Kosten festgestellt und angepasst werden.

Für komplexere Analysemöglichkeiten ist das festgelegte Metamodell eher ungeeignet. Durch den Fokus auf die Modellierung benutzt ArchiMate allgemeinere Semantiken bei Elementen und Relationen, die Analysen zusätzlich erschweren. Zum Beispiel benötigt die Analyse von *Modifizierbarkeit* [Lag10] ein abgestimmtes Metamodell mit Elementen, die eine konkretere Bedeutung besitzen. Bei Modifizierbarkeit wäre dies zum Beispiel *Architect Team* oder *Developer Team*, die in ArchiMate beide als *Business Role* modelliert werden würden. Jedoch müssen sich diese Elemente im Metamodell unterscheiden um unterschiedliche Eigenschaften mit Auswirkungen auf die Modifizierbarkeit getrennt betrachten zu können.

Das ArchiMate-Metamodell schränkt auch Analysemöglichkeiten von *MAP* ein, denn einige benötigte Zusammenhänge können nicht beschrieben werden. Zum Beispiel verwendet die Verfügbarkeitsanalyse eine zusätzliche Relationsart, die Redundanzbeziehung. Mit der spezifizierten Möglichkeit *Properties* in ArchiMate zu definieren (PropertyDef) ist eine flexible Möglichkeit geschaffen worden. Diese könnte in *AYE* um ein ähnliches Konzept für Elemente und Relationen eingesetzt werden. Dies würde das Importieren von ArchiMate-Modellen weiterhin möglich machen und weitere Analysemöglichkeiten erlauben, die nur wenige zusätzlichen Elemente und Relationen benötigen. Für die Anwender würde dies bedeuten, dass sie das Modell um zusätzliche Informationen anreichern müssten.

5.2 Weitere potentielle Forschungsarbeiten

AYE setzt einen Grundbaustein, der für weitere Arbeiten genutzt werden kann. Neben möglichen Definitionen von Elementen und Relationen zur Erweiterung des ArchiMate-Metamodells finden sich in *MAP* weitere Analysearten, die integriert werden könnten, zum Beispiel die Utility-Analyse oder Datenkorrektheit. Darüber hinaus arbeitet *AYE* mit deterministischen Werten, die durch stochastische Ergebnisse ergänzt werden können.

Bei der Verfügbarkeitsanalyse kann außerdem das kontinuierliche Ableiten von Redundanzbeziehungen in Richtung Wurzelknoten verfolgt werden. Hier sind Seiteneffekte zu

erwarten, die eine genauere Evaluation erfordern.

In größeren Modellen verliert sich für den Anwender schnell die Übersicht und die Nachvollziehbarkeit, durch welche Einflüsse bestimmte Werte zustande kamen. Eine interaktivere Darstellung der Einflussfaktoren zur besseren Transparenz der Berechnung könnte hilfreich sein.

Allgemein ist *AYE* offen für weitere Analysen und begrüßt jede Weiterentwicklung.

Literatur

[AK03] Colin Atkinson und Thomas Kühne. "Kühne, T.: Model-driven development: a metamodeling foundation. IEEE Software 20(5), 36-41". In: *Software, IEEE* 20 (Okt. 2003), S. 36–41. DOI: 10.1109/MS.2003.1231149.

[Arm+13] Chris Armstrong u. a. *Using the ArchiMate Language with UML*. White Paper. Sep. 2013.

[BBL12] Stefan Bente, Uwe Bombosch und Shailendra Langade. *Collaborative Enterprise Architecture: Enriching EA with Lean, Agile, and Enterprise 2.0 Practices*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[Bih15] Charles Bihis, Hrsg. *Mastering OAuth 2.0*. Birmingham, UK: Packt Publishing, Dez. 2015. ISBN: 978-1-78439-540-7.

[BMS09] Sabine Buckl, Florian Matthes und Christian M. Schweda. "Classifying Enterprise Architecture Analysis Approaches". In: *Enterprise Interoperability*. Hrsg. von Raul Poler, Marten van Sinderen und Raquel Sanchis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 66–79. ISBN: 978-3-642-04750-3.

[Dij59] Edsger Wybe Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1.1 (Dez. 1959), S. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390.

[EAA13] EAAT. *EAAT User Manual: A guide how to use Class Modeler and Object Modeler*. Manual. Abgerufen am 29. Oktober 2018. Stockholm, Schweden: Dept. of Industrial Information und Control Systems, KTH Royal Institute of Technology, Aug. 2013. URL: https://www.kth.se/polopoly_fs/1.428432!/Menu/general/column-content/attachment/EAAT_manual.pdf.

[Eng+08] Gregor Engels u. a. *Quasar Enterprise: Anwendungslandschaften serviceorientiert gestalten*. 1. Auflage. Heidelberg: dpunkt.verlag GmbH, 2008. ISBN: 978-3-89864-506-5.

[FAW07] Ronny Fischer, Stephan Aier und Robert Winter. "A Federated Approach to Enterprise Architecture Model Maintenance". In: *Enterprise Modelling and Information Systems Architectures* 2 (Jan. 2007), S. 14–22. DOI: 10.18417/emisa.2.2.2.

[FL14] Martin Fowler und James Lewis. *Microservices*. Abgerufen am 30. September 2018. März 2014. URL: https://martinfowler.com/articles/microservices.html.

[FM90] Norman E. Fenton und Austin Melton. "Deriving structurally based software measures." In: *Journal of Systems and Software* 12.3 (1990), S. 177–187.

[Goo17] Google Inc. *Material Design*. Abgerufen am 15. Juni 2017. 2017. URL: https://material.io/.

[Har11] Van Haren. *TOGAF Version 9.1*. 10. Auflage. Van Haren Publishing, 2011. ISBN: 978-9087536794.

[IEE90] IEEE. "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (Dez. 1990), S. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[Int10] International Function Point Users Group (IFPUG). *Function Point Counting Practices Manual, Release 4.3.1*. Spezifikation. 2010.

[Jäc18] Andreas Jäckel. "Konzeption und Umsetzung einer Portalseite und Integrationsumgebung für die HAWAI Microservices". Bachelor Thesis. Hamburg: Hochschule für Angewandte Wissenschaften Hamburg, Okt. 2018.

[Joh+07] Pontus Johnson u. a. "Enterprise Architecture Analysis with Extended Influence Diagrams". In: *Information Systems Frontiers* 9.2 (2007), S. 163–180. ISSN: 1572-9419.

[Joh+13] Pontus Johnson u. a. *IT Management with Enterprise Architecture*. Techn. Ber. Abgerufen am 30. Oktober 2018. Dept. of Industrial Information und Control Systems, KTH Royal Institute of Technology, Aug. 2013. URL: https://www.kth.se/nse/research/resilient-information-and-control-systems/projects/the-multi-attribute-prediction-map-class-diagram-1.387306.

[Joh+16] Pontus Johnson u. a. "Modeling and Analyzing Systems-of-Systems in the Multi-Attribute Prediction Language (MAPL)". In: *2016 IEEE/ACM 4th International Workshop on Software Engineering for Systems-of-Systems (SESoS)*. Mai 2016, S. 1–7. DOI: 10.1109/SESoS.2016.009.

[Joh88] Barry W. Johnson, Hrsg. *Design and Analysis of Fault Tolerant Digital Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1988. ISBN: 0-201-07570-9.

[Lag+17] Robert Lagerström u. a. "Automated Probabilistic System Architecture Analysis in the Multi-Attribute Prediction Language (MAPL): Iteratively Developed using Multiple Case Studies". In: *Complex Systems Informatics and Modeling Quarterly* (Juli 2017), S. 38–68. DOI: 10.7250/csimq.2017-11.03.

[Lag10] Robert Lagerström. "Enterprise Systems Modifiability Analysis: An Enterprise Architecture Modeling Approach for Decision Making". Diss. Stockholm: KTH Royal Institute of Technology, Apr. 2010.

[Lan17] Marc Lankhorst. *Enterprise Architecture at Work: Modelling, Communication and Analysis*. 4. Auflage. Berlin/Heidelberg: Springer-Verlag, 2017. ISBN: 978-3-662-53932-3.

[LE14] Robert Lagerström und Mathias Ekstedt. "Extending a general theory of software to engineering". In: *GTSE*. Hrsg. von Pontus Johnson, Michael Goedicke und Ivar Jacobson. ACM, 2014, S. 36–39. ISBN: 978-1-4503-2850-0.

[LJE15] Robert Lagerström, Pontus Johnson und Mathias Ekstedt. "Search-Based Design of Large Software Systems-of-Systems". In: *2015 IEEE/ACM 3rd International Workshop on Software Engineering for Systems-of-Systems*. Mai 2015, S. 44–47. DOI: 10.1109/SESoS.2015.15.

[MKG04] Nigel Melville, Kenneth Kraemer und Vijay Gurbaxani. "Review: Information Technology and Organizational Performance: An Integrative Model of It Business Value". In: *MIS Q.* 28.2 (Juni 2004), S. 283–322. ISSN: 0276-7783.

[När+08] Per Närman u. a. "Using Enterprise Architecture Models for System Quality Analysis". In: *2008 12th International IEEE Enterprise Distributed Object Computing Conference*. Sep. 2008, S. 14–23. DOI: 10.1109/EDOC.2008.26.

[När12] Per Närman. "Enterprise Architecture for Information System Analysis: Modeling and assessing data accuracy, availability, performance and application usage". Diss. Stockholm: KTH Royal Institute of Technology, Sep. 2012.

[NBE14] Per Närman, Markus Buschle und Mathias Ekstedt. "An enterprise architecture framework for multi-attribute infor mation systems analysis". In: *Software and Systems Modeling* 13.3 (2014), S. 1085–1116. ISSN: 16191366.

[New15] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. 1. Auflage. Berlin: O'Reilly Media, Feb. 2015. ISBN: 978-1-491-95035-7.

[NJN07] Per Närman, Pontus Johnson und Lars Nordström. "Enterprise Architecture: A Framework Supporting System Quality Analysis". In: *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*. Okt. 2007, S. 130–130. DOI: 10.1109/EDOC.2007.39.

[OMG16] OMG. *Meta Object Facility (MOF) Core Specification, Version 2.5.1*. Spezifikation. Object Management Group, Nov. 2016. URL: http://www.omg.org/spec/MOF/2.5.1.

[OMG17] OMG. *Unified Modeling Language (UML) Specification, Version 2.5.1*. Spezifikation. Object Management Group, Dez. 2017. URL: http://www.omg.org/spec/UML/2.5.1.

[Ope14] OpenID Foundation. *OpenID Connect*. Abgerufen am 16. November 2018. Feb. 2014. URL: https://openid.net/connect/.

[Ope16] The Open Group. *The Open Group ArchiMate Open Exchange Format*. Abgerufen am 01. Oktober 2018. Dez. 2016. URL: http://www.opengroup.org/xsd/archimate/3.0/.

[Ope17] The Open Group. *ArchiMate 3.0.1 Specification*. Spezifikation. Abgerufen am 02. Oktober 2018. 2017. URL: http://pubs.opengroup.org/architecture/archimate3-doc/.

[Ost+13] Magnus Osterlind u. a. "Enterprise Architecture Evaluation Using Utility Theory". In: *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC* (Sep. 2013), S. 347–351. DOI: 10.1109/EDOCW.2013.45.

[RH04] Marvin Rausand und Arnljot Høyland. *System Reliability Theory: Models, Statistical Methods and Applications*. Hoboken, New York: Wiley-Interscience, 2004.

[RZM14] Sascha Roth, Marin Zec und Florian Matthes. *Enterprise Architecture Tool Survey 2014. State-of-the-Art and Future Development*. Techn. Ber. Technische Universität München, 2014.

[Sch08] Daniel A. Schult. "Exploring network structure, dynamics, and function using NetworkX". In: *In Proceedings of the 7th Python in Science Conference (SciPy)*. Abgerufen am 21. November 2018. 2008, S. 11–15.

[Sma16] SmartBear Software. *Swagger UI*. Abgerufen am 16. November 2018. Sep. 2016. URL: https://swagger.io/tools/swagger-ui/.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Ort	Datum	Unterschrift im Original