Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Konzeptioneller Entwurf und Implementierung einer Methode zur automatisierten Verfeinerung von Topologien

Arthur Kaul

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Dr. h.c. Frank Leymann
Betreuer/in:	M.Sc. Jasmin Guth, M.Sc. Karoline Saatkamp
Beginn am:	1. März 2017
Beendet am:	29. September 2017
CR-Nummer:	C.0, D.2.2, D.2.10, D.2.11, D.2.13, H.4.2

Kurzfassung

Im Bereich moderner Cloud- und serviceorientierter Architekturen, muss in sich ständig ändernden Systemumgebungen, die Systemarchitektur schnell und flexibel an die neuen Anforderungen angepasst und erneut bereitgestellt werden. Daher werden die Systemarchitekturen in diesen Bereichen durch eine Zusammenstellung von abstrakten Komponenten und Services beschrieben, die bei sich ändernden Anforderungen angepasst und in neue ausführbare Systemarchitekturen verfeinert werden können. Eine große Herausforderung bei der Verfeinerung stellt die automatische Zusammenstellung von zueinander kompatiblen ausführbaren Services und Komponenten dar, die alle Anforderungen erfüllen. Um diese Herausforderung zu bewältigen wurde ein Verfeinerungsmodell für die Verfeinerung von abstrakten Systemarchitekturen in ausführbare konkrete Systemarchitekturen entwickelt. Das Verfeinerungsmodell dient als Grundlage für ein semi-automatisiertes Verfeinerungsverfahren von Systemarchitekturen, welche durch Topologiemodelle dargestellt und beschrieben werden. Das Verfeinerungsverfahren wird in einem Architectural Refinement (ArchRef) Tool als Prototyp implementiert und kann für die Modellierung von Topologiemodellen und ihre anschließende automatisierte Verfeinerung verwendet werden.

Schlagwörter: Systemarchitekturen, Topologien, Verfeinerung, Modelltransformation

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

ADL Architecture Description Language. 33

AM Ausgangsmodell. 29

CCIM Cloud Computation Independent Model. 37

CIM Computation Independent Model. 36

CPIM Cloud Provider Independent Model. 37

CPSM Cloud Provider Specific Model. 37

CSP Communicating Sequential Processes. 34

CWM Comman Warehouse Metamodel. 35

DMMN Declarative Application Management Modelling and Notation. 41

GReAT Graph Rewriting and Transformation Language. 38

MDA Model-Driven Architecture. 35

MOF Meta Object Facility. 27

PIM Plattform Independent Model. 36

PM Plattform Modell. 36

PSM Plattform Specific Model. 36

QoS Quality of Service. 115

TOSCA Topology and Orchestration Specification for Cloud Computing. 25

TR Transformationsregel. 29

XMI XML Metadata Interchange. 35

XML Extensible Markup Language. 33

ZM Zielmodell. 29

1 Einleitung

In der heutigen Zeit steigt die Bedeutung der Gestaltung von wiederverwendbaren, integrativen und erweiterbaren Architekturen in Entwurfs- und Entwicklungsphasen von Informationssystemen. Dies hat zur Folge, dass insbesondere Systemarchitekten vor der Herausforderung stehen, die Wiederverwendbarkeit von Systemarchitekturen zu erhöhen, um somit die Abhängigkeit von spezifischen Technologien und Anbietern zu verringern. Des Weiteren kann durch die Ausgestaltung von Architekturen, die von Anbietern und Technologien unabhängig sind, der Aufwand bei der Migration und Integration von Informationssystemen verringert und ein Vendor-Lock-in-Effekt verhindert werden [Nel09]. Dieser Effekt spielt vor allem dann eine wichtige Rolle, wenn Unternehmen sich dazu entscheiden, die Systemarchitektur mittels Produkten eines Cloud-Anbieters oder anderen Drittanbietern zu realisieren [DWC10][Lea09].

Eine weitere zunehmende Herausforderung für Systemarchitekten stellt die Identifikation und Auswahl von einzelnen Bestandteilen, die in einer Systemarchitektur verwendet werden können dar. Dabei sollen möglichst alle Anforderungen, die an ein Informationssystem gestellt werden, durch die Systemarchitektur wiedergegeben werden. Dieser Auswahlprozess wird zunehmend durch die steigende Anzahl von Anbietern und Produkten in Bereichen wie Cloud-Computing, Web Services und Microservices erschwert [Li+10]. Allerdings ist nicht nur die Auswahl von zweckmäßigen Bestandteilen von Informationssystemarchitekturen entscheidend, ob die Anforderungen eines Informationssystems erfüllt werden oder nicht, sondern auch die Kombination der jeweiligen ausgewählten Bestandteile und die spätere Verteilung auf unterschiedliche Plattformen und Anbieter [SB94].

Um die Herausforderungen in den Entwurfs- und Entwicklungsphasen zu bewältigen, durchlaufen die Systemarchitekten einen Verfeinerungsprozess, in dem sie abstrakte Architekturen anfertigen. Diese sind von Anbietern und Technologien unabhängig und können daher leicht angepasst und wiederverwendet werden [Mac+06]. Anschließend werden diese schrittweise rekursiv bis zu einer ausführbaren Systemarchitektur verfeinert, die in der realen Welt ausgeführt werden kann. Die Architektur wird mit jedem Verfeinerungsschritt spezifischer und geht von einer abstrakten Systemarchitektur in eine oder mehrere spezifische Systemdesignlösungen über [EM01][VL03]. Da einzelne Bestandteile oder Technologien einer Systemarchitektur einen Lebenszyklus besitzen und sich innerhalb diesem weiterentwickeln oder von anderen Produkten und Technologien ersetzt werden, müssen Teile der Systemarchitektur entfernt, hinzugefügt oder ersetzt werden. Deshalb muss der Verfeinerungsprozess von Systemarchitekturen mehrfach im Laufe des Lebenszyklus von Systemarchitekten wiederholt werden [Bal11b]. Da es sich um einen immer wiederkehrenden Prozessablauf handelt, lohnt es sich, diesen

zu automatisieren, um somit Zeit und Kosten einzusparen. Daher soll in dieser Arbeit ein graphbasiertes Konzept (vgl. Kap. 5) und eine Methode (vgl. Kap. 6) erarbeitet werden, mit denen der Verfeinerungsprozess von Systemarchitekturen, die mittels Topologien beschrieben und dargestellt werden, automatisiert werden kann um somit die Systemarchitekten in den Entwurfs- und Entwicklungsphasen zu unterstützen.

1.1 Problemstellung

Zu Beginn der Entwicklung von neuen Systemen sowie bei Anpassungen von bestehenden, wird zu Beginn ein Grobentwurf der neuen bzw. angepassten Systemarchitektur modelliert [Bal11a]. Damit diese Architektur in einer Laufzeitumgebung aufgesetzt werden kann, muss sie solange schrittweise verfeinert, konkretisiert und spezifiziert werden, bis ausreichend Informationen für eine automatisierte Bereitstellung und Ausführung in einer Laufzeitumgebung, verfügbar sind. In den Bereichen des Cloud-Computings und der serviceorientierten Architekturen stellt sich die Frage mit welchen Services von welchen Anbietern die Architektur überhaupt realisiert werden kann und welche bei der Umsetzung verwendet werden können, damit alle Anforderungen, die an das neue oder anzupassende Informationssystem bestehen, auch erfüllt werden [LNZ04]. Um eine geeignete Lösung zu finden, muss eine große Anzahl an unterschiedlichen Bestandteilen und Kombinationsmöglichkeiten betrachtet werden. Dabei ist es des Öfteren der Fall, dass einige Lösungsmöglichkeiten nicht sofort oder überhaupt nicht von den Softwarearchitekten entdeckt werden und dadurch auch nicht in den Architekturentscheidungsprozess miteinbezogen werden. Um dies zu vermeiden, soll der Prozess der Verfeinerung von Topologien automatisiert werden. Dadurch sollen möglichst alle realisierbaren Kombinationsmöglichkeiten einer Systemarchitektur, die alle Anforderungen erfüllen, einem Systemarchitekten aufgezeigt werden.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, einen Verfeinerungsprozess zu entwickeln, der möglichst automatisiert durchgeführt werden kann und durch den die, im vorherigen Kapitel genannten Probleme gelöst werden können. Das erarbeitete Konzept zur automatisierten Verfeinerung soll nicht nur auf Topologien, die den Schwerpunkt dieser Arbeit darstellen, anwendbar sein, sondern auch auf andere graphbasierte Modelle. Dadurch soll gewährleistet werden, dass zu einem späteren Zeitpunkt eine leichte und schnelle Adaption des Verfahrens vorgenommen werden kann. Das entwickelte Verfahren soll in einem Prototyp mittels einer Webanwendung umgesetzt werden. Des Weiteren sollen folgende Fragestellungen näher betrachtet und untersucht werden:

- Welche graphbasierten Modelle existieren in der Literatur um eine Systemarchitektur zu beschreiben und formal zu definieren? Und welche Ansätze existieren bereits in der automatischen Transformation und Verfeinerung von graphbasierten Modellen?

- Anhand welchem Verfeinerungsmodell können Topologiemodelle automatisiert verfeinert werden und welche Schritte und Phasen müssen bei einem automatisierten Verfeinerungsprozess durchlaufen werden?

1.3 Gliederung

Die vorliegende Arbeit ist in acht Kapitel unterteilt. Im Folgenden wird ein kurzer Überblick zu den einzelnen Kapiteln gegeben und die Inhalte von diesen aufgezeigt.

Im ***ersten Kapitel*** wird die Thematik vorgestellt sowie die Problemstellungen näher erläutert. Zudem wurden die Ziele, die mit dieser Arbeit verfolgt und untersucht werden genau formuliert.

Im ***zweiten Kapitel*** werden die grundlegenden Begriffe wie der Architektur-, Design-, Modell- und Metamodellbegriff erläutert und voneinander abgegrenzt. Anschließend werden Graphen definiert und die relevanten Ausprägungsformen von diesen anhand von Beispielen aufgezeigt. Abschließend wird der Transformations- und Verfeinerungsbegriff sowie dessen unterschiedliche Ausprägungsformen definiert und voneinander abgegrenzt.

Das ***dritte Kapitel*** zeigt verwandte Ansätze auf, die in Bezug zu dieser Arbeit stehen und sich mit dem Beschreiben und Darstellen von Systemarchitekturen sowie mit der Transformation und Verfeinerung von Modellen von Systemarchitekturen befassen.

Das ***vierte Kapitel*** beschreibt und formalisiert das angepasste Topologie Metamodell von Breitenbücher [Bre16], welches zur Beschreibung von Systemarchitekturen in dieser Arbeit verwendet wird.

Im ***fünften Kapitel*** wird ein Lösungskonzept vorgestellt, auf dessen Grundlage ein semi-automatisiertes Verfeinerungsverfahren für Systemarchitekturen entwickelt wird. In diesem Konzept wird das Level Graph Metamodell, welches die einzelnen Bestandteile und Beziehungen eines Level Graph Modells beschreibt, im Detail erläutert und definiert.

Das ***sechste Kapitel*** beschreibt die einzelnen Schritte der Methodik für die semi-automatisierte Verfeinerung von Topologiemodellen anhand des zuvor erarbeiteten Level Graph Metamodell. Die Methodik besteht dabei aus fünf wesentlichen Einzelschritten, die bei einem Verfeinerungsprozess durchlaufen werden.

Im ***siebten Kapitel*** wird die Implementierung des entwickelten ArchRef Tool Prototyps vorgestellt. Dabei werden die einzelnen Bestandteile des Prototyps und die zur Umsetzung verwendeten Technologien, anhand einer Systemarchitektur erläutert.

Im abschließenden ***achten Kapitel*** wird ein Gesamtüberblick über die Ergebnisse in einem Fazit wiedergegeben sowie ein Ausblick auf mögliche weitere Forschungen.

2 Grundlagen

In diesem Kapitel soll ein gemeinsames, einheitliches Verständnis über grundlegende Begriffe, Konzepte und Methoden, die für diese Arbeit von Relevanz sind, herbeigeführt werden. Dazu werden Begriffe wie Architektur, Design, Modell und Metamodell genauer definiert und voneinander abgegrenzt. Anschließend werden die wichtigsten Formen von Graphen definiert und ihre Strukturen anhand von Beispielen aufgezeigt. Des Weiteren wird der Transformations- und Verfeinerungsbegriff präzisiert und ein Bezug zur Transformation von Modellen hergestellt.

2.1 Architektur- und Designbegriff

Der Begriff der Architektur stammt aus dem Griechischen und wird vor allem im Bauwesen als die Kunst bzw. Wissenschaft verstanden Bauwerke zu konstruieren oder zu gestalten [Dud06][MW83]. Unter dem Begriff des Bauwerkes wird im Bereich der Informationstechnologie die Konstruktionszeichnungen und Beschreibungen von Informationstechnologiesystemen, welche z.B. in Systemspezifikationen enthalten sind verstanden [BMS13]. Architekturen sind im Normalfall zweckgebunden, weshalb die angefertigten Konstruktionen einen bestimmten Zweck dienen sollten, damit diese als Architekturen bezeichnet werden können [Sou66]. Ein weiterer Gesichtspunkt von Architekturen ist, dass diese den subjektiven Wahrnehmungen und den Umwelteinflüssen des Urhebers, dem sogenannten Architekten, unterliegen und durch diesen letztendlich beeinflusst werden [FS13]. Dadurch kann die Vielfältigkeit und Heterogenität der Lösungsansätze sehr hoch sein. Im Bereich der Informationstechnologie kann diese hohe Heterogenität und Vielfältigkeit von Architekturlösungen zu schwerwiegenden Problemen führen. Eine Verschmelzung von Informationssystemen von mehreren verschiedenen Unternehmen kann z.B. nicht ohne Weiteres durchgeführt werden, wenn diese nicht auf Standards beruhen, sondern aus individuellen Technologien von unterschiedlichen Anbietern bestehen, die zudem nicht kompatibel zueinander sind [Kel02]. Diese spezifischen Architekturlösungen werden in der Literatur als Designlösungen bezeichnet und stellen jeweils eine eigenständige Möglichkeit dar, ein und das selbe Problem zu lösen und somit den gleichen Zweck zu erfüllen [RH06]. Daher könnte zum Beispiel eine Referenzarchitektur von einer Webanwendung sowohl auf Basis von Amazon Web Services [Amac] als auch mittels Azure Cloud Services [Web] umgesetzt werden.

2.1.1 Systemarchitektur

Im Informationstechnologiebereich ist häufig die Rede von Architekturen, wenn Begriffe wie Anwendungssystemarchitekturen, Cloud-Computing Systemarchitekturen, Softwaresystemarchitekturen oder weitere Begriffe dieser Art erwähnt werden [Fow02][HNS00]. Der erste Teil dieser Terminologien identifiziert meist den Bereich, in dem die Architektur eingesetzt wird oder die Technologien, auf denen die Architektur basiert. Bei dem zweiten Teil der Begrifflichkeiten handelt es sich um den Kernbegriff, den alle Begrifflichkeiten gemeinsam haben, dem Begriff des Systems. Daraus lässt sich herleiten, dass ein System durch eine Architektur im Informationstechnologiebereich beschrieben und dargestellt werden soll. Diese speziellen Musterformen von Architekturen fallen unter den Oberbegriff der Systemarchitektur. Um definieren zu können, was explizit unter einer Systemarchitektur zu verstehen ist, ist es wichtig nachdem der Begriff der Architektur definiert wurde, den Begriff des Systems zu konkretisieren. Die nachfolgende Definition von Rich Hillard [Hil00] soll als Basis für die Herleitung der Eigenschaften einer Systemarchitektur dienen:

> „Architecture [is] the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.“ [Hil00]

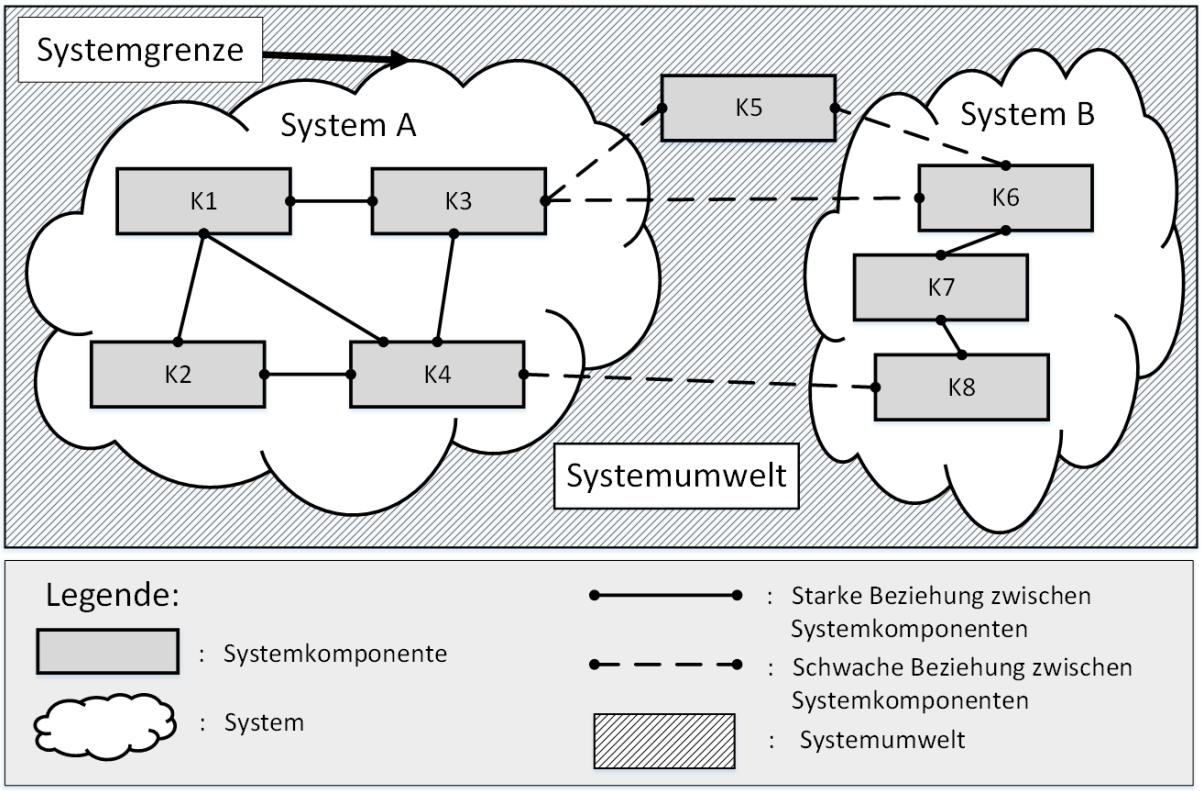

Abbildung 2.1: System und Systemumwelt (vgl. [Böh02])

Aus dieser Aussage von Rich Hillard [Hil00] wird unter der Bezeichnung System, die Gesamtheit von Komponenten verstanden, welche miteinander verbunden sind und somit in Beziehung zueinanderstehen. Durch die Komponenten und Beziehungen soll die Struktur und das Verhalten von einem System wiedergegeben und ein System von anderen Systemen in der Umwelt abgegrenzt werden [Böh02]. In diesem Zusammenhang wird häufig die Abbildung 2.1 in der Literatur zur Verdeutlichung und Abgrenzung eines Systems von der Systemumwelt aufgezeigt. Bei dieser Abgrenzung gehören Komponenten mit starken Beziehungen untereinander zu einem System und Komponenten mit schwachen Beziehungen werden unterschiedlichen Systemen oder der Systemumwelt zugeordnet.

2.1.2 Perspektiven von Systemarchitekturen

In der Praxis werden Systemarchitekturen meist nicht nur durch eine einzige Darstellungsform repräsentiert, sondern durch mehrere unterschiedliche Formen [Cle+02]. Diese Darstellungsformen betrachten das gleiche System aus unterschiedlichen Blickwinkeln und setzen dabei den Schwerpunkt auf verschiedene Aspekte eines Systems. Bei ER-Modellen liegt der Fokus z.B. auf der Struktur und Organisation der Daten in einem System [Vos00]. Die verschiedenen Ausprägungsformen der Darstellung eines Systems können anhand zweier Dimensionen in unterschiedliche Kategorien differenziert werden. Bei der ersten Dimension handelt es sich um die Dimension der Perspektive. In dieser werden die Systemarchitekturen anhand des Blickwinkels oder der Sicht kategorisiert, die ein Betrachter auf die Systemarchitektur hat. Dabei kann nach Jacobson, Booch und Rumbaugh [RJB04] die Dimension der Perspektive von Systemarchitekturen in eine strukturelle Perspektive und in eine Verhaltensperspektive unterteilt werden:

- ***Strukturelle Perspektive:***
  Die strukturelle Perspektive dient im Wesentlichen dazu die Funktionalitäten von einzelnen Komponenten sowie die jeweiligen Schnittstellen zwischen den unterschiedlichen Komponenten aufzuzeigen. Dadurch soll den Betrachtern ein Verständnis von der Struktur und den Abhängigkeiten zwischen Komponenten vermittelt werden. In dieser Perspektive wird die Systemarchitektur zu einem festen Zeitpunkt betrachtet und daher auch als statische Perspektive bezeichnet. Diese Perspektive wird z.B. durch Klassen-, Paket-, Blockdiagramme, Topologien (vgl. Kap. 3) und weitere dargestellt [RJB04].
- ***Verhaltensperspektive:***
  In der Verhaltensperspektive wird, im Gegensatz zu der strukturellen Perspektive, das Verhalten einer Systemarchitektur nicht nur zu einem Zeitpunkt betrachtet, sondern über einen Zeitraum. Daher wird diese Perspektive auch als dynamische Perspektive bezeichnet. Sie beschreibt insbesondere die internen Abläufe von den einzelnen Komponenten sowie die Abläufe der Interaktionen zwischen den Komponenten. Dabei kann das System zu unterschiedlichen Zeitpunkten verschiedene Strukturen aufweisen. Dieses Phänomen

wird auch als Mobilität von Systemarchitekturen bezeichnet [MT10]. Die Verhaltensperspektive kann z.B. durch Kollaborations-, Sequenz-, Aktivitäts-, Zustandsdiagramme usw. dargestellt werden [RJB04].

In dieser Arbeit wird die automatisierte Verfeinerung von Systemarchitekturen der strukturellen Perspektive untersucht. Daher ist die Verhaltensperspektive nicht relevant für die Arbeit und wird in den nachfolgenden Kapiteln nicht berücksichtigt.

2.1.3 Abstraktionsebenen von Systemarchitekturen

Anhand der zweiten Dimension, der Abstraktionsdimension, werden die Systemarchitekturen der strukturellen Perspektive und der Verhaltensperspektive zusätzlich anhand ihres Abstraktionsgrades differenziert. Dabei variiert die Anzahl an Abstraktionsebenen von Systemarchitekturen je nachdem, wie stark oder schwach die Abstraktionsabstufungen oder Verfeinerungsschritte definiert werden, um von einer abstrakten Architektur zur einer spezifischen Architektur in einem Verfeinerungsprozess zu gelangen. In der Literatur werden meist drei bis vier verschiedene Abstraktionsebenen für eine Systemarchitekturverfeinerung als ausreichend angesehen [SNH95][GPR07]. Soni et. al. [SNH95] beschreibt vier unterschiedliche Abstraktionsebenen (vgl. Tab. 2.1), anhand derer eine Verfeinerung und Differenzierung der

Systemarchitekturebenen			
Ebene	**Bedeutung**	**Zielgruppe**	**Abstraktions-grad**
Konzeptionelle Architektur	Benutzt Domänen spezifische Komponenten und Konnektoren und dient zur Vermittlung eines Gesamtverständnisses der statischen und dynamischen Strukturen eines Systems.	Endnutzer, Systemarchitekt, Manager	hoch
Ausführbare Architektur	Die ausführbare Architektur betrachtet die konzeptionelle Architektur in unterschiedlichen Ausführungsumgebungen und Konfigurationsmöglichkeiten.	Entwickler, Systemarchitekt	mittel
Modul Architektur	Die Modul Architektur dient dazu, die einzelnen Module einer ausführbaren Architektur in Bezug auf Konsistenz der Schnittstellen zwischen Modulen zu analysieren.	Entwickler, Systemarchitekt	mittel
Programmcode Architektur	Konfiguration und Management der Modul Architektur in einer spezifisch aufgebauten Systemumgebung basierend auf Produkten von Herstellern.	Entwickler, Infrastrukturmanager, Systemarchitekt	gering

Tabelle 2.1: Systemarchitekturebenen nach Soni et. al. (vgl. [SNH95])

Systemarchitekturen in Bezug auf ihren Abstraktionsgrad durchgeführt werden kann. Die Abstraktionsebenen können anschließend verwendet werden, um einen Verfeinerungsprozess aufzusetzen und dadurch Systemarchitekturen mit unterschiedlichen Abstraktionsgraden zu erstellen [SNH95].

Dieser Ansatz von Soni et. al. [SNH95] wird im Folgenden angepasst und soll als Grundlage für die spätere Verfeinerung von Systemarchitekturen anhand von Abstraktionsebenen dienen. Da der in Kapitel 6 entwickelte Verfeinerungsprozess keinen Programmcode generiert, wird auf eine Programmcode Architekturebene verzichtet. Damit ergeben sich folgende drei angepasste Abstraktionsebenen, welche für diese Arbeit von Relevanz sind und wie folgt definiert werden [SNH95]:

- ***Konzeptionelle Systemarchitektur:***
  In einer konzeptionellen Systemarchitektur wird die grobe Verteilung von Systemkomponenten aufgezeigt und welche dieser Komponenten wie miteinander in Beziehung stehen. Sie dient dazu einen Gesamtüberblick über ein Informationssystem zu vermitteln, um dadurch wesentliche Anforderungen und Eigenschaften, die an das System gestellt werden, identifizieren zu können. Bei den konzeptionellen Systemarchitekturen handelt es sich um abstrakte Systemarchitekturen, welche unabhängig von Technologien und Anbietern sind. Eine Komponente von dieser Architektur könnte z.B. eine ganz allgemeingültige Datenbank- oder Serverkomponente sein.
- ***Individuelle Systemarchitektur:***
  In einer individuellen Systemarchitektur werden detaillierte Informationen von einzelnen Komponenten und Beziehungen der konzeptionellen Systemarchitektur entweder in einer abstrakten oder spezifischen Form ausgedrückt. Eine abstrakte individuelle Systemarchitektur beschreibt diese detaillierten Informationen ausschließlich mit Komponenten die unabhängig von spezifischen Technologien und Anbietern sind. Wohingegen eine spezifische individuelle Systemarchitektur zusätzlich Aussagen darüber trifft mit welcher Technologie einzelne Komponenten und die Beziehungen zwischen diesen umgesetzt werden. Dennoch sind die Komponenten weiterhin unabhängig von bestimmten Anbietern, um die Unabhängigkeit von den Architekturen zumindest teilweise zu erhalten und den Vendor-Lock In Effekt auf die unterste Abstraktionsebene zu verlagern. Somit kann die individuelle Systemarchitekturebene als eine gemischte Abstraktionsebene angesehen werden in der sowohl abstrakte Bestandteile als auch spezifische Bestandteile vorkommen können. Eine Komponente von dieser Architektur könnte z.B. eine ganz allgemeingültige relationale Datenbank Komponente oder SQL Datenbank Komponente darstellen.
- ***Ausführbare Systemarchitektur:***
  Die ausführbare Systemarchitektur ist eine spezifische individuelle Systemarchitektur, die zusätzlich mit Deployment Informationen angereichert wurde und dessen Bestandteile abhängig von Technologie und Anbieter sind. Diese deploybaren Systemarchitekturen können zum Beispiel mit dem Topology and Orchestration Specification for Cloud

Computing (TOSCA) Standard [Tos] beschrieben werden. Für den TOSCA Standard existieren Werkzeuge, wie z.B. WINERY [Kop+13] zum Modellieren von deploybaren Topologien und OpenTosca [Bin+14] zum Deployen und Ausführen der modellierten Topologien. Eine Komponente von dieser Systemarchitektur könnte z.B. eine Amazon EC2 Komponente mit einer spezifischen Konfiguration sein. In dieser Arbeit wird in dem entwickelten ArchRef Tool Prototyp (vgl. Kap. 7) in den Datenstrukturen auf die Deployment Informationen zur Vereinfachung verzichtet.

2.2 Modell und Metamodell

Im Folgenden sollen die beiden Begriffe Modell und Metamodell voneinander abgegrenzt werden. Dazu wird der Modell- und Metamodellbegriff erläutert, um anschließend eine Abgrenzung zwischen diesen Begriffen festlegen zu können.

2.2.1 Modellbegriff

In der Literatur gibt es eine Vielzahl an verschiedenen Definitionen, die den Fokus auf unterschiedliche Eigenschaften und Aspekte eines Modells legen. Bei den meisten Definitionen wird ein Modell als eine Abbildung, Darstellung oder Beschreibung von einem Objekt angesehen [KWB03][Sei03]. Nach MDA Guide wird die Spezifikation eines Systems als Modell bezeichnet, indem ein System durch eine Kombination von Zeichnungen und Texten repräsentiert und aus unterschiedlichen Perspektiven beschrieben wird [MMB+03] (vgl. Kap. 2.1.2). Demnach werden Systemarchitekturen durch Modelle repräsentiert und stellen die Konstruktionszeichnungen von Informationssystemarchitekturen dar. Nach Selic [Sel03] weist ein Modell fünf Eigenschaften auf, anhand derer eine Bewertung der Qualität von Modellen durchgeführt werden kann. Im Folgenden werden die Eigenschaften nach Selic [Sel03] beschrieben und auf dem System Begriff angepasst:

- ***Abstrakt:***
  Nur die wichtigsten Faktoren und Eigenschaften von dem zu erschaffenden oder modellierenden System sollten durch ein Modell wiedergegeben werden. Das bedeutet, dass wichtige Aspekte des Systems durch das Modell hervorgehoben und irrelevante vernachlässigt werden sollen.
- ***Verständlich:***
  Jeder Akteur in der spezifischen Domäne, in der das Modell umgesetzt werden soll, soll das Modell mit all seinen zugehörigen Texten und Zeichnungen verstehen können.
- ***Genau:***
  Das Modell soll die Realität so genau wie möglich wiedergeben und keine falschen Aussagen über das zu erschaffende oder zu beschreibende System beinhalten. Zudem

dürfen keine zentralen Aussagen im Modell über das beschriebene System ausgeklammert werden.

- ***Prädiktiv:***
  Durch ein Modell sollen Aussagen und Schlussfolgerungen über das Verhalten der zu erschaffenden Systeme in der Realität abgeleitet und somit vorhergesagt werden können.
- ***Günstig:***
  Der Aufwand ein Modell zu erstellen und anschließend zu analysieren soll deutlich geringer sein, als die eigentliche Implementierung des zu erschaffenden Systems in der Realität, damit sich dieser lohnt.

Einige der Eigenschaften von Modellen weisen negative Wechselwirkungen auf. Das bedeutet, wenn ein Modell zu abstrakt ist, könnte es unter Umständen von den Akteuren nicht mehr verstanden werden, weil wichtige Aspekte weggelassen wurden. Das hat zur Folge, dass bei der Erstellung von Modellen Kompromisse bei der Erfüllung aller Eigenschaften eingegangen werden müssen. Dabei können je nach Verwendungszweck des Modells, die Eigenschaften unterschiedlich stark gewichtet werden. Zum Beispiel spielt bei dem Zweck, der Vermittlung eines einheitlichen Gesamtverständnisses über die Struktur einer Systemarchitektur an alle relevanten Stakeholder, die Abstraktion eine übergeordnete Rolle gegenüber der Genauigkeit von einem Modell. Wohingegen bei der Implementierung und Ausführung von einer Systemarchitektur, die Genauigkeit von einem Modell eine wichtigere Rolle gegenüber der Abstraktion von einem Modell darstellt.

2.2.2 Metamodellbegriff

Nachdem die Hauptmerkmale und der Zweck eines Modells geklärt wurden, bleibt noch die Frage offen welche Rolle und Zweck ein Metamodell erfüllt und von welcher Bedeutung ein Metamodell für ein Modell ist. Diese offenen Punkte sollen im Folgenden erarbeitet und geklärt werden. Das Metamodell wird in der Literatur häufig als ein Modell von anderen Modellen bezeichnet. Die kurze prägnante Formulierung der OMG Group im MDA Guide trifft es sehr gut, demnach ist ein Metamodell:

> „A Model of Models“ [MMB+03].

Der Grund warum ein Metamodell eingeführt wurde, ist das häufig Aussagen über ein Modell mit der gleichen Sprache gemacht werden, mit der auch das Modell beschrieben wird. Dieser Umstand kann dazu führen, dass nicht mehr klar differenziert werden kann, ob sich Aussagen auf das Modell an sich beziehen oder ob eine Aussage sich darauf bezieht wie ein Modell zu beschreiben ist. Dieses Phänomen wird auch in der Literatur als „Lügner-Paradoxon“ bezeichnet [Háj98]. Um paradoxe Aussagen in einem Modell zu vermeiden muss festgelegt werden, dass Aussagen die zur Beschreibung des eigentlichen Modells nötig sind, der Modell Ebene zugeordnet werden und Aussagen darüber, wie die Modelle zu beschreiben sind einer separaten Metamodell Ebene zugeordnet werden [ÁES01]. Somit wird durch die Einführung

von zwei klar voneinander getrennten Modell Ebenen erreicht, dass die Aussagen voneinander durch die Verwendung von unterschiedlichen Sprachräume abgegrenzt werden. Doch dies löst das Problem nicht vollständig, sondern verschiebt es eine Ebene nach oben. Nun stellt sich die Frage wie können Aussagen, über Aussagen im Metamodell getroffen werden. Dazu wird die gleiche Lösung rekursiv auf das Metamodell angewandt und es wird ein Meta-Metamodell eingeführt [MOF02]. Dies führt zu einer hierarchischen Struktur von mehreren Modell Ebenen, dieses Phänomen wird auch im Bereich der Logik und Mathematik als „Tarski Hierarchie“ bezeichnet [Tar35]. Jedoch stellt sich noch die Frage wie viele unterschiedliche Modell Ebenen benötigt werden um Modelle definieren zu können. Dafür wird als Referenz das Meta Object Facility (MOF) Modell [MOF02] von der OMG Group herangezogen, bei dem vier verschiedene Modell Ebenen als ausreichend angesehen werden. Bei diesen vier Ebenen handelt es sich um ein Meta-Metamodell, ein Metamodell, ein Modell und eine Ebene für die Instanzen von Modellen in der realen Welt. In dieser Arbeit wird lediglich die Modell und Metamodell Ebene für die Beschreibung der unterschiedlichen Modelle verwendet, die für die Verfeinerung und Darstellung von Systemarchitekturen benötigt werden.

2.3 Graphen

Es existieren bereits viele Ansätze in der Literatur für die Erstellung von deklarativen Modellen, um dadurch Systemarchitekturen aus unterschiedlichsten Bereichen zu beschreiben, wie Petri-Netze [Abe13], UML-Diagramme [Obj] und ER-Modelle [Vos00] und noch weitaus mehr. Wobei viele dieser Ansätze auch in einen einfachen Graphen umgeformt und durch diesen beschrieben werden können (vgl. Kap. 3). Da das vorgeschlagene Konzept in Kapitel 5 sich ebenfalls auf einen graphbasierten Ansatz zurückführen lässt, wird in diesem Abschnitt die Basis von einem Graphen definiert sowie ein kurzer Überblick über die relevanten Formen von existierenden Graphen anhand von Beispielen aufgezeigt.

In der Graphentheorie wird die Basis von endlichen Graphen $G = (V, E)$ als ein Tupel von zwei Mengen bezeichnet, wobei folgende Eigenschaften für diese Mengen gelten [Wes+01]:

(i) V ist eine nicht leere endliche Menge der Knoten in einem Graphen G, wobei $v \in V$ genau einen Knoten in dem Graphen G repräsentiert

(ii) $E = V \times V$ ist eine möglicherweise leere endliche Menge von Kanten in einem Graphen G, wobei $e = (v_\mathrm{i}, v_\mathrm{j}) \in E$ genau eine Kante in dem Graphen G ist, wenn $v_\mathrm{i} \in V \wedge v_\mathrm{j} \in V$

(iii) Des Weiteren gilt $V \bigcap E = \emptyset$, da kein Element, das eine Kante e im Graphen G repräsentiert auch ein Knoten v im Graphen G sein kann

Auf dieser Basis von einem Graphen lassen sich unterschiedlichste Graph Strukturen ableiten, je nachdem welche Einschränkungen auf der Knotenmenge V und auf der Kantenmenge E gemacht werden. So können Kanten zwischen Knoten z.B. gerichtete oder gewichtete Kanten im Graphen darstellen. Eine Übersicht der wichtigsten Strukturen und Formen von

Graphen kann aus der Abbildung 2.2 entnommen werden. Zwei mögliche spezielle Formen von Graphen wären z.B. hierarchische oder k-partite Graphen. Diese Strukturen werden auch bei der Definition des Level Graph Metamodell (vgl. Kap. 5) aufgegriffen. Im weiteren Verlauf dieser Arbeit wird angenommen, dass alle Mengen die definiert werden endliche Mengen sind und damit auch jeder weitere Graph der definiert wird einen endlichen Graphen darstellt.

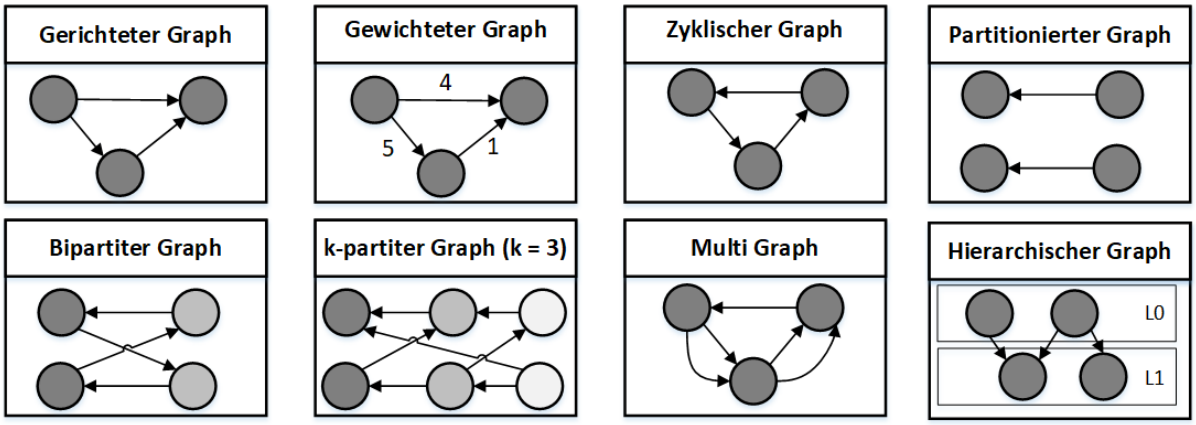

Abbildung 2.2: Strukturen und Formen von Graphen

2.4 Transformation- und Verfeinerungsbegriff

Unter dem Transformationsbegriff wird im Bereich von Systemarchitekturen, die Umwandlung von einem oder mehreren Ausgangsmodelle (AMs) in ein oder mehrere andere Zielmodelle (ZMs), anhand von sogenannten Transformationsregeln (TRs) verstanden [All07][SK03]. Ein analoges Verständnis gibt es auch in dem Bereich der Graph Transformation. Dort wird die Umwandlung von einem oder mehreren Graphen, in ein oder mehrere andere Graphen durch Transformationsfunktionen unter dem Transformationsbegriff verstanden [ERR06]. Diese Analogie ist von Bedeutung für die erarbeitete Methode zur Transformation von Systemarchitekturen, da in dieser eine Systemarchitektur durch ein Topologiemodell beschrieben wird, welches durch einen Graphen dargestellt werden kann (vgl. Kap. 4). Das Topologiemodell wird anhand eines weiteren Verfeinerungsmodell, den Level Graphen (vgl. Kap. 5), welches die Transformationsregeln festlegt und ebenfalls einen Graphen darstellt, verfeinert. Dabei lässt sich der Transformationsbegriff anhand verschiedener Faktoren in diverse Kategorien aufteilen und spezifizieren. Einer dieser Faktoren ist die Transformationsrichtung, in die ein Ausgangsmodell transformiert wird. Diese kann nach Mens et. al. [MVG06] in eine horizontale und vertikale Transformationsrichtung unterschieden werden und wird in dieser Arbeit wie folgt definiert (vgl. Abb. 2.3) [All07][SK03][MVG06]:

- ***Horizontale Transformation***
  Bei der horizontalen Transformation werden Modelle innerhalb einer Abstraktionsebene umgewandelt. Das bedeutete, dass bei einer Transformation von einem Ausgangsmodell in ein Zielmodell beide Modelle den gleichen Abstraktionsgrad haben. Jedoch könnten sich die Strukturen gänzlich unterscheiden und die Zielmodelle auf andere Metamodelle basieren als die Ausgangsmodelle. Bei dieser Art der Transformation handelt es sich um eine exogene Transformation, da die Grenze von dem Metamodell des Ausgangsmodells bei der Transformation überschritten wird und sich damit die Metamodelle von Ausgangsmodell und Zielmodell unterscheiden [Jou+08]. Dies wäre zum Beispiel der Fall, wenn bei einer Migration ein System welches auf einer .Net Plattform betrieben wird auf eine J2EE Plattform migriert werden soll. Der andere Fall der horizontalen Transformation wäre, die Transformation innerhalb eines Metamodells. Das bedeutete, dass sowohl der Sprachraum als auch der Abstraktionsgrad des Ausgangsmodells und des Zielmodells gleich sind. In diesem Fall könnte es sich zum Beispiel um die Transformation in eine andere Designlösung der Systemarchitektur mit dem gleichen Abstraktionsgrad handeln, wobei beide Designlösungen durch ein Modell dargestellt werden, welches von ein und demselben Metamodell abgeleitet wurde. Die Horizontale Transformation ist nicht Gegenstand dieser Arbeit, da sich der Abstraktionsgrad bei der Transformation von Modellen nicht verändert und daher diese keine Verfeinerungstransformation darstellt.
- ***Vertikale Transformation***
  Bei der vertikalen Transformation befindet sich das Ausgangsmodell auf einer anderen Abstraktionsebene als das Zielmodell [Men+06]. Dabei können zwei Fälle unterschieden werden. Der erste Fall wäre, wenn der Abstraktionsgrad des Ausgangsmodells kleiner als der Abstraktionsgrad des Zielmodells ist. In diesem Fall würde es sich um eine Form der Abstraktions- oder Generalisierungstransformation handeln. Diese wird meistens durch Weglassen von Informationen oder durch Zusammenziehen von Gemeinsamkeiten durchgeführt. Ein Anwendungsfall dieser Transformation ist das Auffinden von allgemeingültigen Entwurfsmustern von Systemarchitekturen, die zur Lösung von bestimmten wiederkehrende Probleme verwendet werden können. Der zweite Fall der vertikalen Transformation wäre, wenn das Ausgangsmodell einen höheren Abstraktionsgrad besitzt als das Zielmodell. In diesem Fall handelt es sich um eine Verfeinerungstransformation bei der das Ausgangsmodell durch Informationen angereichert und die Struktur des Systems verfeinert wird, innerhalb der Verfeinerungstransformation können wiederum unterschiedliche Arten der Verfeinerungstransformation existieren (vgl. Kap. 5.7) [Men+06][MVG06]. Die Verfeinerungstransformation ist Kernbestandteil dieser Arbeit.

Es können auch Kombinationen dieser beiden Transformationsrichtungen vorkommen. Zum Beispiel kann eine Transformation vertikal und horizontal sein, wenn das Ausgangsmodell von einer höheren Abstraktionsebene, in ein Zielmodell einer tieferen Abstraktionsebene verfeinert wird und beide Modelle auf unterschiedlichen Metamodellen basieren. Die Abbildung 2.3 stellt diesen Fall durch die rechts unten abgebildete diagonale Transformationsrichtung dar und bezeichnet diese als exogenen Verfeinerungstransformation. Ein anderer Faktor anhand

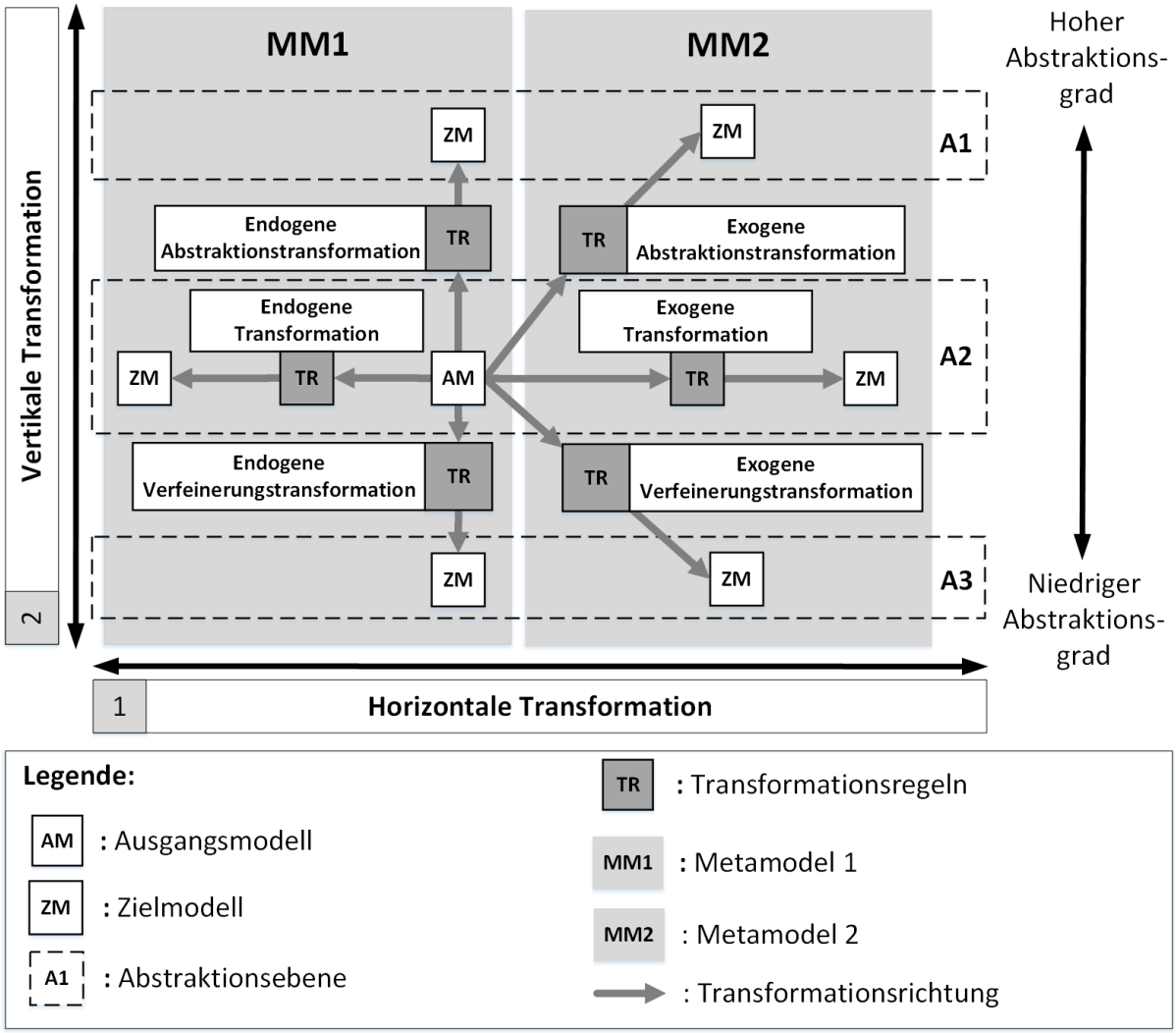

Abbildung 2.3: Transformationsrichtungen

dem die Transformationsarten eingeteilt werden können, ist die Anzahl von Ausgangs- und Zielmodellen, die in dem Transformationsprozess entstehen oder verwendet werden [MVG06]. Dabei lassen sich folgende vier Kategorien bilden, die 1-zu-1, die 1-zu-N, die N-zu-1 und die N-zu-N Transformation [Men+06]. Diese Formen wiederum können in den Einzelnen aufgeführten Transformationsrichtungen vorkommen und je nach Transformationsrichtung eine unterschiedliche Bedeutung haben. So stellt eine 1-zu-N Transformation mit vertikaler Transformationsrichtung von einem Ausgangsmodell in eine tiefere Abstraktionsebene, eine Verfeinerung dar, wohingegen eine 1-zu-N Transformation in horizontaler Transformationsrichtung eine Spaltung von dem Ausgangsmodell darstellt [MVG06]. Diese Arbeit befasst sich insbesondere mit der vertikalen endogenen Modell-zu-Modell Verfeinerungstransformation von graphbasierten Modellen, daher wird im weiteren Verlauf der Schwerpunkt auf diese Transformationsart gelegt (vgl. Abb. 2.3).

3 Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten und Konzepte vorgestellt die sich mit dem Darstellen und Beschreiben von Systemarchitekturen mittels Modellierungssprachen oder Architecture Desciprion Languages (ADLs) befassen. Anschließend werden Modell-zu-Modell Transformationsverfahren, die sich insbesondere mit dem Transformieren von graphbasierten Modellen auseinandersetzen näher betrachtet.

3.1 Architecture Description Languages und Modellierungssprachen

Ziel von ADLs und Modellierungssprachen sind, die Struktur und das Verhalten von System- oder Softwarearchitekturen durch eine formale Syntax oder eine graphische Notation zu beschreiben [GMW10][HM08]. Die ADLs werden meistens mit einer formalen Syntax beschrieben. Dabei kann auch auf bereits existierende Standards aufgesetzt werden, damit keine neue Syntax definiert werden muss, wie z.B. auf Extensible Markup Language (XML) [W3ca]. Die Modellierungssprachen werden dagegen meistens durch graphische Notationen repräsentiert. Wobei die graphischen Notationen durch Transformation in eine formale Sprache umgewandelt werden kann und umgekehrt. In diesen Fällen spricht man auch von einer Modell-zu-Text Transformation [CH06]. Die formalen Sprachen eignen sich insbesondere für Auswertungen und Analysen von Architekturen durch verschiedene bereits bekannte Graph Algorithmen, unter der Voraussetzung, dass sich die formal definierte Syntax auf eine Graph Datenstruktur abbilden lässt [CH03]. In der Literatur existieren eine Vielzahl von ADLs und Modellierungssprachen zum Beschreiben von Systemarchitekturen, wobei zu meist jede davon den Schwerpunkt auf einen bestimmten Anwendungsbereich legt und spezifische Ziele verfolgt. Im Folgenden werden einige dieser ADLs und Modellierungssprachen vorgestellt und dem graphbasierten Ansatz, der in dieser Arbeit vorgeschlagen wird gegenübergestellt.

π-ADL

Die π-ADL wurde zum Beschreiben von Systemarchitekturen entwickelt, die dynamisches und mobiles Verhalten aufweisen und ihre Strukturen über die Zeit hinweg verändern können [Oqu04]. Dabei basiert die π-ADL auf der höher geordneten formalen π-Calculus Sprache und daher wird auch die π-ADL durch eine formale Syntax beschrieben [San93]. Da jedoch die

formale Sprache von π-ADL auch Bestandteile wie Komponenten und Konnektoren beinhaltet, kann sie in einen Graphen umgewandelt und somit durch graphbasierte Algorithmen verarbeitet werden. Der Vorteil bei π-ADL gegenüber dem Ansatz dieser Masterarbeit ist, dass durch sie auch dynamische Veränderungen und die Mobilität von Systemarchitekturen wiedergespiegelt werden, was mit dem ArchRef Tool Prototyp (vgl. Kap. 7) aktuell noch nicht möglich ist.

Unified Modelling Langage (UML)

UML ist eine Modellierungssprache welche durch die OMG standardisiert wurde und sich insbesondere bei dem Entwerfen von objektorientierten Softwarearchitekturen etabliert und durchgesetzt hat [Obj]. Dabei wird durch UML nicht nur eine Darstellungsform von einem Architekturdiagramm unterstützt, sondern eine ganze Reihe an unterschiedlichen Diagrammtypen, wie Sequenzdiagramme, Use Case Diagramme, Paketdiagramme und viele weitere, die sowohl die strukturelle Perspektive als auch die Verhaltensperspektive einer Systemarchitektur auf unterschiedlichsten Abstraktionsebenen wiederspiegeln können (vgl. Kap. 2.1.2 u. Kap. 2.1.3) [RJB04]. Das UML Modell basiert auf dem MOF Modell welches ebenfalls von der OMG entwickelt und standardisiert wurde. Dabei stellt jedes Element von dem UML Modell eine Instanz von einer Klasse dar, was auch der Grund dafür ist das UML verstärkt in dem Bereich der objektorientierten Softwarearchitekturen eingesetzt wird [MOF02]. Da die graphischen Diagramme der UML Sprache alle in Form von Graphen dargestellt werden, lassen sich diese genau wie das Topologiemodell dieser Arbeit (vgl. Kap. 4), auf eine Graph Datenstruktur mappen und verarbeiten. Jedoch liegt der Fokus bei den Topologiemodellen nicht auf Softwarearchitekturen, sondern viel mehr auf der Zusammenstellung von Service Komponenten im Bereich der Cloud-Computing Architekturen.

Wright

Die Wright ADL ist ebenso wie die π-ADL eine ADL, die eine abstrakte Systemarchitektur durch eine formale Sprache beschreibt. Dabei basiert jedoch die Wright ADL auf einer Reihe von Communicating Sequential Processes (CSP) Notationen [MT00]. Diese Notationen stellen eine Menge von statischen Prüfungen dar, die dazu verwendet werden die Konsistenz und Vollständigkeit von einer Systemarchitektur zu überprüfen [All97]. Dabei setzt Wright den Fokus insbesondere auf die Analyse des Verhaltens von Systemarchitekturen und kann genutzt werden um Deadlocks in den Systemarchitekturen zu entdecken [MT00].

ACME

Aufgrund der großen Anzahl an unterschiedlichen ADLs wurde ACME entwickelt um eine Architektur, die mit einer ADL (A) beschrieben wurde in eine andere ADL (B), die auf einer anderen Syntax als ADL (A) basiert, zu transformieren und somit den Austausch von ADLs zu

ermöglichen [GMW10]. Daher wird ACME auch verstärkt in dem Bereich der horizontalen Transformation eingesetzt um Modelle von unterschiedlichen Metamodellen austauschbar zu machen und weniger in dem Bereich der Verfeinerungstransformation. ACME basiert auf der Grundlage, dass alle Architekturen im Kern zwei gemeinsame Bestandteile haben. Diese Kernbestandteile werden bei ACME als Komponenten und Konnektoren bezeichnet und sind die Hauptbestandteile von einer Systemarchitektur (vgl. 2.1.1) [GMW10]. Aus diesem Grund könnte ACME auch für eine graphbasierte Transformation verwendet werden, da sich die Komponenten auf die Knoten und die Konnektoren auf die Kanten eines Graphen mappen lassen. ACME kann sowohl durch eine formale Syntax dargestellt werden, wie auch durch eine graphische Notation, die z.B. mit ACME Studio [Acm] erstellt werden kann.

xADL

Bei der xADL handelt es sich um eine ADL, die von der University of California, Irvine entwickelt wurde und mit den Softwarearchitekturen beschrieben werden können [DHT01]. Anders als die vorherigen ADLs basiert xADL nicht auf einer formalen eigenen Syntax Sprache, sondern auf einer Menge von definierten XML-Schemen, in denen z.B. die Strukturelemente einer Systemarchitektur, wie Komponenten und Konnektoren, definiert werden. Diese können unabhängig von Tools oder Architektur Style verwendet werden [Xad][DHT02]. Auch die ArchRef Implementierung in dieser Arbeit beruht auf einem XML-Schema [Kau17]. Der Vorteil bei einer XML-Schema Definition ist, dass diese flexibel bei Bedarf erweitert werden kann und eine plattformunabhängige strukturierte Sprache darstellt. Dadurch kann Interoperabilität und der Austausch von Systemarchitekturmodellen zwischen Systemen die auf unterschiedlichen Plattformen laufen gewährleistet werden [Xad].

Petri-Netze

Petri-Netze sind bipartite gerichtete Graphen mit zwei Arten von Knoten, den Plätzen und den Transitionen, wobei eine Kante in einem Petri-Netz immer einen Platz mit einer Transition oder eine Transition mit einem Platz verbindet [Sta78]. Durch Petri-Netze können eine große Anzahl verschiedener Systeme beschrieben werden wie z.B. parallele, verteilte, nebenläufige oder nicht-deterministische Systeme [Bau90]. Zudem werden Petri-Netze auch verstärkt im Bereich vom Entwerfen von Business Prozessen eingesetzt und dort für Business Process Mining Ansätze verwendet [HS04]. Der Kompatibilitäts-Level Graph von dem Level Graph Modell in dieser Arbeit (vgl. Kap. 5.4) könnte durch Anpassungen auch als ein Petri-Netz modelliert werden.

3.2 Modell-zu-Modell Transformation

Im Bereich der Modell-zu-Modell Transformation existiert eine ganze Reihe von unterschiedlichen Ansätzen. In dieser Arbeit wird der Fokus auf Ansätze gelegt, die sich mit der graphbasierten Modell-zu-Modell Transformation beschäftigen. Der Model-Driven Architecture (MDA) Ansatz der OMG, ist wohl einer der bekanntesten Ansätze aus der Modell-zu-Modell Transformation und basiert auf einer Reihe von Standards, wie UML [Obj], Comman Warehouse Metamodel (CWM) [Poo+02], MOF [MOF02] und XML Metadata Interchange (XMI) [Xmi]. Da dieser Ansatz, die Basis für einige der nachfolgenden vorgestellten Transformationsmethoden ist, wird dieser zunächst im Detail betrachtet.

Model-Driven Architecture (MDA)

Das MDA Modell besteht aus nachfolgenden drei Modell Ebenen die bei einem Architekturverfeinerungsprozess durchlaufen werden [GPR07]:

- ***Computation Independent Model (CIM):***
  Durch das CIM werden die funktionalen Anforderungen, welche das Gesamtsystem erfüllen muss dargestellt. Diese Darstellungen sind vollständig unabhängig von den Technologien oder Plattformen, die für die spätere Implementierung des Gesamtsystems verwendet werden. Die CIMs sind oft das Ergebnis von einem Requirements Engineering Prozess im Unternehmen und beschreiben die zukünftigen fachlichen Prozessabläufe und Anforderungen anhand von Use Case Diagramme. CIMs stellen die oberste Modell Ebene des MDA Transformationsansatzes dar und dienen als Ausgangspunkt für eine Verfeinerung von Architekturen [MDA08].
- ***Plattform Independent Model (PIM):***
  Die PIMs beschreiben die Struktur und die Funktionen der einzelnen Komponenten einer Systemarchitektur. Dabei muss bei der Erstellung von PIMs darauf geachtet werden, dass die erhobenen funktionalen Anforderungen des zugehörigen CIMs vollständig abgedeckt werden. Die PIMs beschreiben, genau wie die CIMs, nicht auf welcher spezifischen Plattform die Systemarchitektur umgesetzt wird. Allerdings wird die Systemarchitektur nicht rein aus einer funktionalen Perspektive beschrieben, sondern vielmehr aus einer technischen. Die PIMs stellen die mittlere Ebene des MDA Ansatzes dar [GPR07].
- ***Plattform Specific Model (PSM):***
  Die unterste Ebene vom MDA Modell wird durch die PSMs abgebildet. Diese Modelle beschreiben letztendlich mit welchen Technologien und auf welchen spezifischen Plattformen die Gesamtarchitektur umgesetzt wird. Diese Modelle können nur auf einem spezifischen Plattform Model eingesetzt werden und weisen daher eine geringe Portabilität auf, sind jedoch detailliert genug beschrieben um aus ihnen Programmcode für den Implementierungsprozess zu generieren [GPR07].

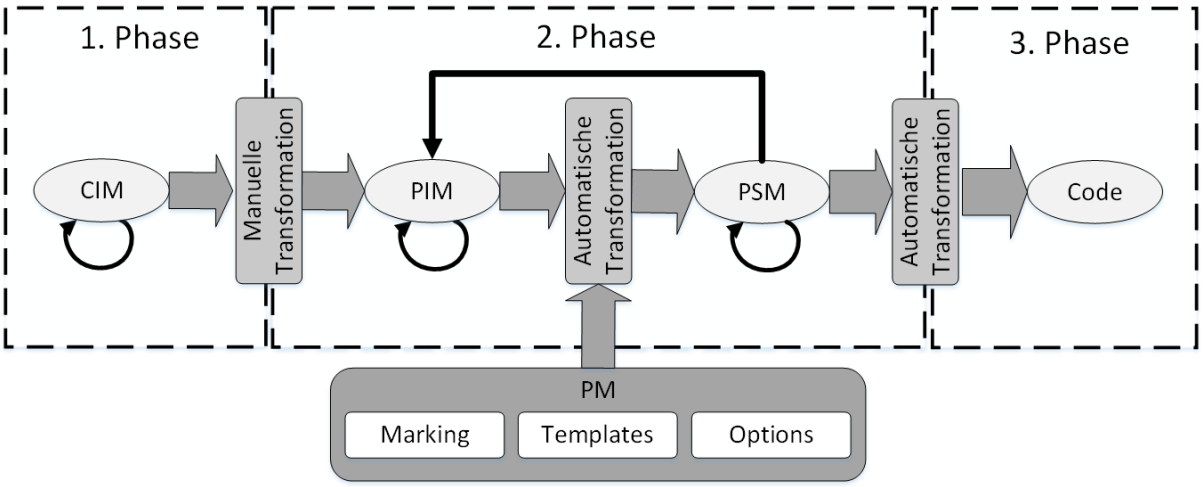

Abbildung 3.1: Model-Driven-Architecture Verfeinerungsprozess (vgl. [GPR07])

Die Abbildung 3.1 stellt den Verfeinerungsprozess, welcher auf dem MDA Modell basiert dar. Dabei startet dieser nach dem MDA Modell mit einem CIM als Ausgangspunkt und besteht aus drei wesentlichen Verfeinerungsphasen, die wiederum ein oder mehrere Transformationsschritte beinhalten können. Das CIM Modell wird in der ersten Verfeinerungsphase durch einen oder mehrere manuellen Transformationsschritte in ein weiteres detaillierteres CIM der ersten Modell Ebene überführt oder in ein PIM der zweiten Modell Ebene umgewandelt. Durch die Umwandlung in ein PIM beginnt die zweite Verfeinerungsphase. In der zweiten Phase wird das entstandene PIM aus der ersten Phase entweder durch einen oder mehrere automatische oder manuelle Transformationsschritte in ein weiteres PIM verfeinert oder mit Hilfe von einem Plattform Modell (PM) automatisch in ein PSM überführt. Dabei wird unter einem PM folgendes verstanden:

- ***Plattform Model (PM):***
  Das PM dient zur Unterstützung für einen automatischen Transformationsprozess in der zweiten Phase um ein PIM automatisch in ein PSM zu überführen. Es definiert Regeln, Markierungen oder Templates, die für die Transformation eines PIM, in ein PSM verwendet werden können. Daraus folgt, dass jede spezifische Plattform ein eigenes separates zugehöriges Plattform Model für eine automatische Transformation definieren muss [MDA08].

Die zweite Verfeinerungsphase kann mehrmals wiederholt werden. Bei einer Wiederholung der zweiten Verfeinerungsphase wird das am Ende entstandene PSM der vorherigen Verfeinerungsphase zu einem PIM in der nächsten Verfeinerungsphase. In der letzten Verfeinerungsphase wird das PSM durch einen automatischen Transformationsschritt in Programmcode umgewandelt (vgl. Abb. 3.1).

MODAClouds

Der MODAClouds Ansatz basiert auf dem MDA Ansatz [Mod]. MODAClouds wurde jedoch speziell für das Entwerfen von Multi-Cloud Anwendungen entwickelt, die später in mehreren verschiedenen Cloudumgebungen von diversen Anbietern deployed und betrieben werden sollen [Ard+12]. Dadurch soll das Problem von dem Vendor-Lock-In Effekt an einen Cloud-Anbieter reduziert werden. Der MODAClouds Ansatz unterscheidet ebenfalls wie der MDA Ansatz drei Modell Ebenen, die bei der Verfeinerung von einer Cloud-Anwendungssystemarchitektur durchlaufen werden [Mod]: (1) Cloud Computation Independent Model (CCIM), (2) Cloud Provider Independent Model (CPIM), (3) Cloud Provider Specific Model (CPSM). Die Bedeutung der einzelnen Modelle für den Verfeinerungsprozess kann dabei aus dem MDA-Ansatz analog abgeleitet werden und auf das Cloud-Computing Paradigma adaptiert werden. MODAClouds stellt eine Reihe von Tools zur Verfügung mit denen nicht nur Multi-Cloud Architekturen erstellt werden können, sondern diese auch noch anschließend automatisiert in eine Cloud Laufzeitumgebung deployed und überwacht sowie zur Laufzeit analysiert werden [DN+17][Mod]. Das in Kapitel 6 entwickelte Verfeinerungsverfahren dieser Arbeit kann ebenfalls für Verfeinerungen im Bereich von Cloud-Computing verwendet werden.

MOLA

Der MOLA Ansatz lässt sich ebenfalls auf den MDA-Ansatz der OMG zurückführen, jedoch beschreibt MOLA den Transformationsprozess nicht anhand von Modell Ebenen, sondern stellt viel mehr eine separate graphische Modell Transformationssprache dar. Mit dieser Sprache kann definiert werden, welche Strukturen einer Systemarchitektur wie in andere Strukturen überführt und transformiert werden können [KBC05]. Dabei besteht MOLA aus nachfolgenden zwei wesentlichen Bestandteilen [KBC05]:

1.) ***Statements:***
Die Statements, ausgenommen von Schleifen werden eingeschlossen in graue abgerundete Rechtecke graphisch dargestellt, daher werden sie auch als graphische Statements bezeichnet. Diese können dabei elementare Transformationsstatements wie Zuweisungen oder Transformationsregeln darstellen oder aber auch Schleifen. Wobei in einer Schleife wiederum mehrere Sequenzen von Statements enthalten sein können [KBC05].

2.) ***MOLA Programme:***
Ein MOLA Programm ist eine Sequenz von graphischen Statements, die durch gestrichelte, gerichtete Linien miteinander verbunden werden und in der Sequenz ausgeführt werden. Durch Programme wird definiert wie ein Modell in ein anderes Modell auf Basis eines Metamodells transformiert werden kann [KBC05].

Der in dieser Arbeit vorgestellte Verfeinerungsprozess, basiert auf Level Graph Modelle die ebenfalls als eine Art graphische Transformationssprache für Topologiemodelle angesehen werden können (vgl. Kap. 5).

GReAT

Bei Graph Rewriting and Transformation Language (GReAT) handelt es sich ebenfalls, wie bei MOLA [KBC05] um eine Transformationssprache, wobei die Graphen, die in den Transformationsprozess eingegeben und ausgegeben werden durch UML Diagramme dargestellt werden [Bal+07]. Dabei lässt sich GReAT auf einem Graph-Muster-Match Ansatz zurückführen. In GReAT gibt es zwei Graphen, die für eine Verfeinerung benötigt werden. Einen Muster-Graphen, welcher in eine andere Struktur transformiert werden soll und einen Host-Graphen, welcher festlegt wie die Muster-Graphen transformiert werden können. Dabei wird in dem Host-Graphen nach einem Matching von dem gesamten Muster-Graphen gesucht [ERR06]. Es können nur dann Muster-Graphen durch den Host Graph verfeinert werden, wenn jedes Element in dem Muster-Graphen auch ein zugehöriges Element in dem Host Graphen besitzt [ERR06][Bal+07]. Des Weiteren werden eine ganze Reihe von Transformationsregeln definiert wie ein Matching von einem Muster in einem Host-Graphen gefunden werden kann und welche Arten von Kontrollflussstrukturen, wie z.B. Hierarchien, Sequenzen, Bedingungen oder Rekursionen verwendet werden können, um die Transformation der Muster-Graphen zu beeinflussen. Da GReAT auf Graphen, die durch UML beschrieben werden, basiert ist es insbesondere für die Transformation von Softwarearchitekturen geeignet [Kar+03]. In dieser Arbeit werden ebenfalls wie bei GReAT Transformationsregeln definiert, anhand derer die Verfeinerungstransformation durchgeführt wird. Die benötigten Transformationsregeln für den Level Graph Ansatz dieser Arbeit (vgl. Kap. 5) werden in den Ableitungen sowie Verfeinerungs- und Kompatibilitäts-Level Graphen definiert (vgl. Kap. 5.5, Kap. 5.7 und Kap. 5.4). Zudem kann der entworfene Algorithmus des entwickelten Verfeinerungsverfahren in Kapitel 6.4 ebenfalls als ein Muster-Match Algorithmus betrachtet werden, der nach den Bestandteilen von Topologiemodellen in einem Level Graphen sucht und diese dann anschließend verfeinert.

ATL

Der ATL Ansatz befasst sich mit der Transformation von Modellen die auf unterschiedlichen Metamodellen basieren, wobei jedoch die Metamodelle jeweils Instanzen von demselben Meta-Metamodell darstellen wie auch das ATL Modell [JK05]. Dabei wird mit ATL eine Datei definiert, in der deklarative und imperative Transformationsregeln definiert werden und somit ein Ausgangsmodell in ein Zielmodell anhand von diesen definierten Regeln durchgeführt werden kann [JK05]. Jedoch liegt der Schwerpunkt bei ATL im Bereich der horizontalen Transformation und nicht in im Bereich der heterogenen vertikalen Transformation (vgl. Kap. 2.4 und vgl. Abb. 2.3) [Jou+08]. In dieser Arbeit dient das Level Graph Modell als eine Art ATL Datei, welches definiert wie Topologiemodelle in welche andere Topologiemodelle überführt werden können.

ADMentor

Der ADMentor Ansatz von Olaf Zimmermann et al. [Zim+15] befasst sich mit der Erstellung von Systemdesignlösungen durch sukzessives Lösen von Architekturentscheidungsproblemen. Dieser Ansatz basiert dabei auf einem Problem Space und ein Solution Space. In dem Problem Space werden Probleme die bei der Architekturerstellung auftreten können durch Probleme definiert. Diese könnten durch Auswahl einer Option gelöst werden, wobei die Auswahl einer Option wiederum zu einem anderen Architekturproblem führen könnte. Dieses müsste wiederum durch eine weitere Option gelöst werden bis eine passende Lösung gefunden wurde. Der Solution Space stellt den Lösungsraum dar, in dem die Systemarchitekturen erstellt werden. Dabei können Problemereignisse und Optionsereignisse bei der Modellierung von Systemarchitekturen in dem Lösungsraum instanziiert werden. Diese sollen dem Systemarchitekten helfen Architekturentscheidungen zu treffen. Bei dem Level Graph Ansatz in dieser Arbeit könnten die abstrakten Topologiemodelle (vgl. Kap. 4) als ein Architektur Entscheidungsproblem modelliert werden, indem die erwarteten Eigenschaften (vgl. Kap. 4.6) ausgewählte Optionen für ein Problem darstellen und durch ein Level Graph Modell welches als ein Entscheidungsbaum modelliert wird aufgelöst werden.

4 Metamodell von Topologien

In dieser Arbeit wird für das deklarative Modell zur Beschreibung von Systemarchitekturen ein Metamodell von Topologien verwendet, welches auf der Topologieschicht des Declarative Application Management Modelling and Notation (DMMN) Modell von Breitenbücher [Bre16] basiert.

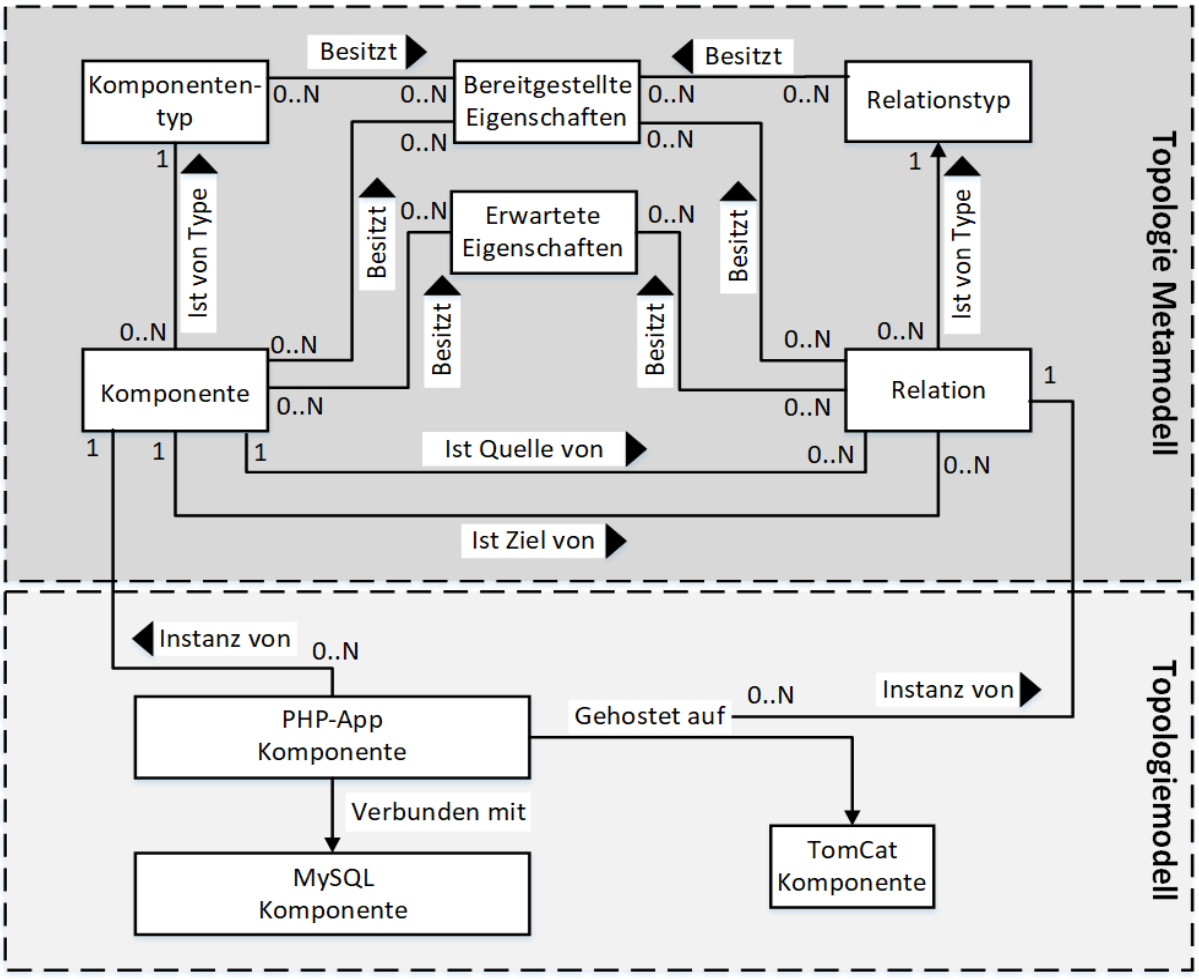

Abbildung 4.1: Metamodell und Modell von Topologien (vgl. [Bre16])

Die Abbildung 4.1 stellt in dem oberen Teil die Ebene des Topologie Metamodells mit seinen Entitäten und Beziehungen zwischen diesen Entitäten als ER-Modell dar. Dabei identifizieren

die Kanten in der Topologie Metamodell Ebene, die Beziehungen sowie die Kardinalitäten zwischen den Entitäten des Topologie Metamodells. Zum Beispiel ist eine Komponente genau einem Komponententypen zugeordnet und umgekehrt kann ein Komponententyp keine oder mehreren Komponenten zugeordnet werden. Die jeweiligen Beziehungen der einzelnen Entitäten des Metamodells werden in den nachfolgenden Kapiteln nochmals angeführt. Die Metamodell Ebene des Topologie Metamodells drückt aus, wie Topologiemodelle zu beschreiben sind. Die Topologiemodell Ebene ist in der Abbildung 4.1 unterhalb dem Topologie Metamodell abgebildet. In dieser ist ein exemplarisches Topologiemodell mit einer PHP-App Komponente, die über eine „Verbunden mit“ Relation mit einer MySQL Komponente und mit einer weiteren „Gehostet auf“ Relation mit einer TomCat Komponente verbunden ist. Dabei stellt jeder der Knoten in dem Topologiemodell eine Instanz von einer Komponente und jede Kante eine Instanz von einer Relation des Topologie Metamodell dar.

Im Folgenden werden die einzelnen Bestandteile und Beziehungen zwischen diesen formal definiert. Für alle Definitionen in dieser Arbeit gilt, dass π_{i} die Projektion des i-ten Elementes von einem Tupel ist und $\sum^{+}$ die Menge aller Zeichen die im Unicode vorkommen können.

4.1 Topologie

Eine Topologie stellt einen gerichteten, möglicherweise zyklischen und nicht zusammenhängenden Multigraphen dar und beschreibt eine Systemarchitektur anhand der strukturellen Perspektive, durch Komponenten und Relationen zwischen diesen Komponenten. Eine Topologie wird formal wie folgt definiert, wobei das 19-Tupel von Topologien nach Breitenbücher [Bre16] auf ein 9-Tupel zur Vereinfachung reduziert wird. Des Weiteren wird das Eigenschaftselement des 19-Tupel in bereitgestellte Eigenschaften und erwartete Eigenschaften im Tupel aufgeteilt, um dadurch die jeweilige Semantik der Eigenschaften für den späteren Verfeinerungsprozess besser differenzieren zu können (vgl. Kap. 6.3). Zudem wird bei allen Definitionen in dieser Arbeit auf die Eindeutigkeit der Namenselemente der einzelnen Tupel, wie es der Fall bei Breitenbücher [Bre16] ist verzichtet, da diese nicht von Relevanz für die Verfeinerung sind und lediglich als verständliche Bezeichnung der einzelnen Bestandteile für die Benutzer dienen:

Definition 4.1.1 (Topologie)
Sei $T = \{t_1, t_2, ..., t_{\mathrm{j}}, ..., t_{\mathrm{n}} | n \in \mathbb{N}\}$ die Menge aller Topologien, dann wird eine Topologie $t_{\mathrm{j}} \in T$, durch folgenden Tupel mit neun verschiedenen Elementen repräsentiert, wobei sechs davon Mengen und drei davon Abbildungen sind:

$$t_{\mathrm{j}} = (K_{\mathrm{t_j}}, R_{\mathrm{t_j}}, KT_{\mathrm{t_j}}, RT_{\mathrm{t_j}}, EE_{\mathrm{t_j}}, BE_{\mathrm{t_j}}, typ_{\mathrm{t_j}}, erwartet_{\mathrm{t_j}}, bereitgestellt_{\mathrm{t_j}}) \tag{4.1}$$

Die einzelnen Mengen Elemente des Tupels werden dabei wie folgt definiert:

1.) K_{t_j} ist die Menge aller Komponenten, die in der Topologie t_j vorkommen, wobei $k_i \in K_{t_j}$ genau eine Komponente in der Topologie t_j repräsentiert (vgl. Def. 4.2.1).

2.) R_{t_j} ist die Menge aller Relationen in der Topologie t_j, wobei $r_i \in R_{t_j}$ genau eine Relation zwischen zwei Komponenten in der Topologie t_j repräsentiert (vgl. Def. 4.4.1).

3.) KT_{t_j} ist die Menge aller Komponententypen in der Topologie t_j, wobei jedes $kt_i \in KT_{t_j}$ die Semantik für jede $k_i \in K_{t_j}$ beschreibt, die diesem Komponententypen durch die typ_{t_j} Abbildung zugeordnet ist (vgl. Def. 4.3.1).

4.) RT_{t_j} ist die Menge aller Relationstypen in der Topoloie t_j, wobei jeder $rt_i \in RT_{t_j}$ die Semantik für jede $r_i \in R_{t_j}$ beschreibt, die diesem Relationstypen durch die typ_{t_j} Abbildung zugeordnet ist (vgl. Def. 4.5.1).

5.) EE_{t_j} ist die Menge aller erwarteten Eigenschaften, die von den Modellierern einer Topologie t_j erwartet werden. Wobei jedes $ee_i \in EE_{t_j}$ eine erwartete Eigenschaft beschreibt, die von allen Komponenten k_i und Relationen r_i in der Topologie t_j erwartet wird, die dieser erwarteten Eigenschaft durch die $erwartet_{t_j}$ Abbildung zugeordnet sind (vgl. Def. 4.6.1).

6.) BE_{t_j} ist die Menge aller bereitgestellten Eigenschaften, die von der Topologie t_j zur Verfügung gestellt werden. Wobei jedes $be_i \in BE_{t_j}$ eine bereitgestellte Eigenschaft beschreibt, die einer oder mehreren Komponenten k_i, Relationen r_i, Komponententyp kt_i und Relationstyp rt_i über die $bereitgestellt_{t_j}$ Abbildung zugeordnet wird (vgl. Def. 4.6.1).

Und die einzelnen Abbildungen des Tupels werden wie nachfolgend definiert:

7.) typ_{t_j} ist die Abbildung, die jeder Komponente k_i genau einen Komponententypen kt_i und jeder Relation r_i genau einen Relationstypen rt_i zuordnet (vgl. Def. 4.7.1).

8.) $erwartet_{t_j}$ ist die Abbildung, die jeder Komponente k_i und jeder Relation r_i keine oder mehrere erwartete Eigenschaften $EET \subset EE_{t_j} \cup \emptyset$ zuordnet (vgl. Def. 4.8.2).

9.) $bereitgestellt_{t_j}$ ist die Abbildung, die jeder Komponente k_i, Relation r_i Komponententyp kt_i und Relationstyp rt_i keine oder mehrere bereitgestellte Eigenschaften $BET \subset BE_{t_j} \cup \emptyset$ zuordnet (vgl. Def. 4.8.1).

Die Abbildung 4.2 stellt die graphische Notation einer gesamten Topologie, die in dieser Arbeit verwendet wird dar. In dieser ist exemplarisch ein Topologiemodell einer einfachen Webanwendung, die aus einer MyPHPApp Anwendungskomponente, die vom PHP_App Komponententyp ist und einer Own_MySQL Komponente, die vom Komponententyp MySQL_DB ist, abgebildet. Dabei sind die Komponenten der Topologie durch eine PHP_MySQL Relation, die vom PHP_MySQL Konnektor Relationstyp ist, miteinander verbunden. Auf der rechten Seite der Abbildung 4.2 ist das Topologiemodell in seiner kompakten Darstellungsform abgebildet. In

der Kompaktform wird auf die Bezeichnungen von den Topologieelementen verzichtet und lediglich die ID der jeweiligen Elemente angegeben.

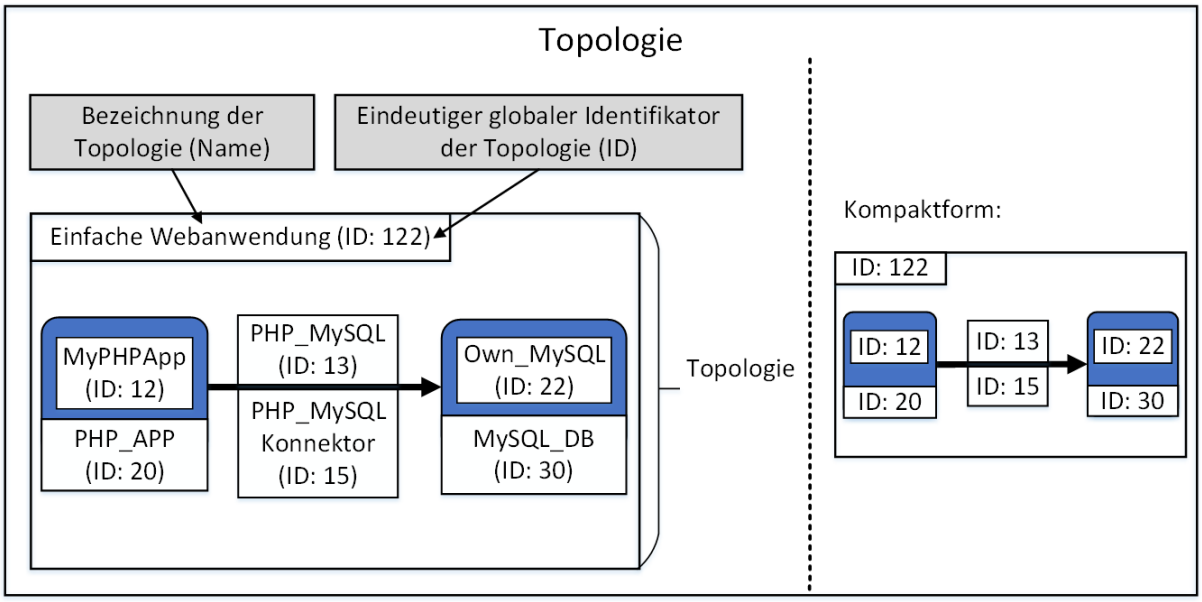

Abbildung 4.2: Notation von Topologiemodellen

Da die Elemente des Topologie Tupels nur grob definiert wurden, werden diese im Folgenden nacheinander im Detail definiert und ihre zugehörige graphische Notation aufgezeigt.

4.2 Komponente

Eine Komponente repräsentiert genau einen Knoten in einem Topologie Graphen. Jede Komponente in einer Topologie ist genau einem Komponententyp von der Topologie zugeordnet und es existiert mindestens eine Komponente in einer Topologie. Zudem ist eine Komponente Quelle oder Ziel von keiner oder mehreren Relationen (vgl. Abb. 4.1). Die Zuordnung der Komponenten zu den Komponententypen wird über die Typ Abbildung in der Topologie abgebildet (vgl. Def. 4.7.1). Komponenten können sowohl abstrakte als auch spezifische Bestandteile in einer Topologie repräsentieren. Eine spezifische Komponente könnte z.B. einen Webservice oder einen Cloud-Service von einem bestimmten Anbieter darstellen, wie z.B. der DynamoDB Service von Amazon [Amaa]. Eine abstrakte Komponente in einer Topologie könnte eine allgemeine Kategorie von den spezifischen Komponenten darstellen, wie z.B. eine Datenbank Komponente oder eine Load Balancer Komponente [Feh+14]. Formal werden die Komponenten wie folgt nach [Bre16] definiert:

Definition 4.2.1 (Komponente)
Sei $K_{t_j} = \{k_1, k_2, ..., k_i, ..., k_n | n \in \mathbb{N}\}$ die nicht leere Menge aller Komponenten in einer Topologie $t_j \in T$, dann wird eine Komponente $k_i \in K_{t_j}$ durch folgenden Tupel definiert:

$$\begin{gathered} K_{t_j} \subseteq \sum^+ \times \sum^+ \\ k_i = (id, name) \in K_{t_j} \end{gathered} \tag{4.2}$$

1.) $\pi_1(k_i) = id$, wobei $id \in \sum^+$ ein eindeutiger globaler Identifikator von k_i ist.

2.) $\pi_2(k_i) = name$, wobei $name \in \sum^+$ eine Bezeichnung für k_i ist.

4.3 Komponententyp

Ein Komponententyp beschreibt die Semantik von einer Komponente in einer Topologie, wobei jeder Komponententyp keiner oder mehreren Komponenten in der Topologie zugeordnet sein kann (vgl. Abb. 4.1). Die Zuordnung der Komponententypen zu den Komponenten wird über die Typ Abbildung in der Topologie abgebildet (vgl. Def. 4.7.1). Zum Beispiel hat der Komponententyp ApplicationServer für eine MyOwnServer Komponente die Bedeutung, dass es sich um einen Application Server bei dieser Komponente handelt. Formal werden die Komponententypen wie folgt nach [Bre16] definiert:

Definition 4.3.1 (Komponententyp)
Sei $KT_{t_j} = \{kt_1, kt_2, ..., kt_i, ..., kt_n | n \in \mathbb{N}\}$ die Menge aller Komponententypen in einer Topologie t_j, dann wird ein Komponententyp $kt_i \in KT_{t_j}$ durch folgenden Tupel definiert:

$$\begin{gathered} KT_{t_j} \subseteq \sum^+ \times \sum^+ \\ kt_i = (id, name) \in KT_{t_j} \end{gathered} \tag{4.3}$$

1.) $\pi_1(kt_i) = id$, wobei $id \in \sum^+$ ein eindeutiger globaler Identifikator von kt_i ist.

2.) $\pi_2(kt_i) = name$, wobei $name \in \sum^+$ eine Bezeichnung für kt_i ist, welche die Semantik des Komponententyp identifiziert.

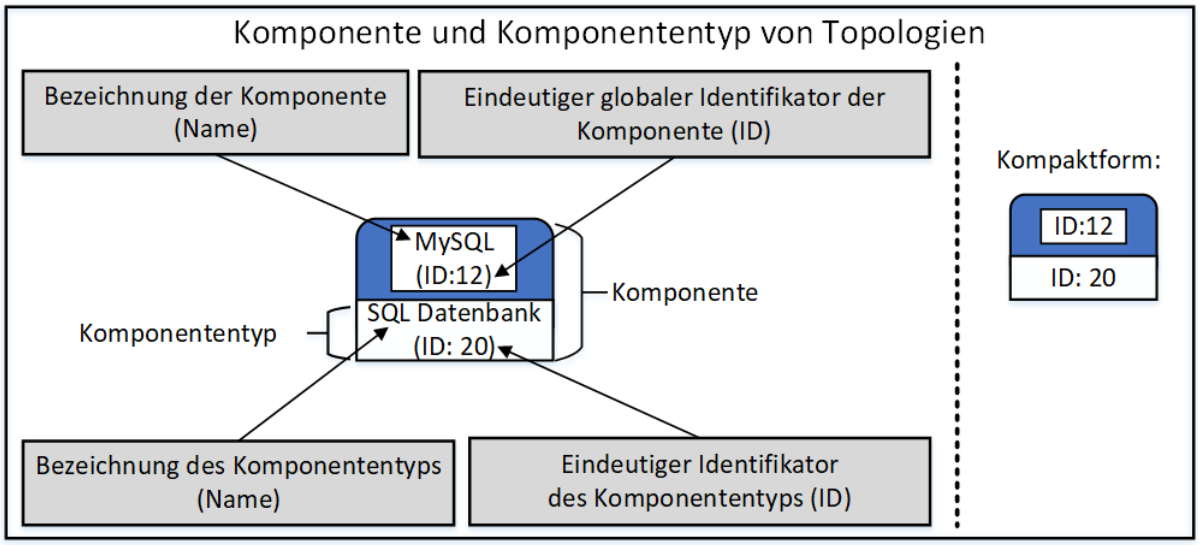

Abbildung 4.3: Notation Komponente und Komponententyp

In der Abbildung 4.3 ist die graphische Notation, die für Komponenten und Komponententypen in Topologien verwendet wird abgebildet. Dabei werden Komponenten durch blaue, oben abgerundete Rechtecke dargestellt und in der Mitte von diesem werden Name sowie ID von einer Komponente in einem weißen Rechteck angegeben. Der Name und die ID des Komponententypen von der Komponente werden in einem weißen Rechteck unten an die Komponente angehängt angegeben. In dem Beispiel in der Abbildung 4.3 handelt es sich um eine MySQL (ID:12) Komponente, die von dem Komponententyp SQL Datenbank (ID:20) ist. Auf der rechten Seite der Abbildung 4.3 ist die Kompaktform von Komponenten und ihren Komponententypen abgebildet.

4.4 Relation

Eine Relation repräsentiert genau eine gerichtete Kante in einem Topologie Graphen. Jede Relation in einer Topologie ist genau einem Relationstyp von der Topologie zugeordnet und hat genau eine Zielkomponente und genau eine Quellkomponente in der Topologie (vgl. Abb. 4.1). Die Zuordnung der Relationen zu den Relationstypen wird über die Typ Abbildung in der Topologie abgebildet (vgl. Def. 4.7.1). Eine Relation könnte z.B. ein JDBC-MySQL Konnektor in einer Topologie repräsentieren, die von einem JDBC-Konnektor Relationstypen ist. Diese Relation könnte als eine Verbindung zwischen einer Java-Anwendungskomponente und einer MySQL-Datenbank Komponente verwendet werden. Dabei können Relationen einen sehr spezifischen Relationstypen wie in dem zuvor genannten Beispiel repräsentieren, aber auch allgemein gültige Relationstypen wie zum Beispiel einen HostedOn Typen darstellen. Formal werden die Relationen wie folgt nach [Bre16] definiert:

Definition 4.4.1 (Relation)
Sei $R_{t_j} = \{r_1, r_2, ..., r_i, ..., r_n | n \in \mathbb{N}\}$ die Menge aller Relationen in einer Topologie t_j, dann wird eine Relation $r_j \in R_{t_j}$ durch folgenden Tupel definiert:

$$\begin{aligned} R_{t_j} &\subseteq \sum^* \times \sum^* \times K_{t_j} \times K_{t_j} \\ r_i &= (id, name, k_q, k_z) \in R_{t_j} \end{aligned} \tag{4.4}$$

1.) $\pi_1(r_i) = id$, wobei $id \in \sum^*$ ein eindeutiger globaler Identifikator von r_i ist.

2.) $\pi_2(r_i) = name$, wobei $name \in \sum^*$ eine Bezeichnung für r_i ist

3.) $\pi_3(r_i) = k_q$, wobei $k_q \in K_{t_j}$ die Quellkomponente für $r_i \in R_{t_j}$ ist.

4.) $\pi_4(r_i) = k_z$, wobei $k_z \in K_{t_j}$ die Zielkomponente für $r_i \in R_{t_j}$ ist.

4.5 Relationstyp

Ein Relationstyp beschreibt die Semantik von einer Relation in einer Topologie, wobei jeder Relationstyp keiner oder mehreren Relationen in der Topologie zugeordnet sein kann (vgl. Abb. 4.1). Zum Beispiel hat der Relationstyp Datenbank Konnektor für eine MyJDBC Konnektor Relation die Bedeutung, dass es sich um eine Datenbankverbindung bei dieser Relation handelt. Die Zuordnung der Relationstypen zu den Relationen in der Topologie wird über die Typ Abbildung abgebildet (vgl. Def. 4.7.1). Formal werden die Relationstypen wie folgt nach [Bre16] definiert:

Definition 4.5.1 (Relationstyp)
Sei $RT_{t_j} = \{rt_1, rt_2, ..., rt_i, ..., rt_n | n \in \mathbb{N}\}$ die Menge aller Relationstypen in einer Topologie t_j, dann wird ein Relationstyp $rt_i \in RT_{t_j}$ durch folgenden Tupel definiert:

$$\begin{aligned} RT_{t_j} &\subseteq \sum^* \times \sum^* \\ rt_i &= (id, name) \in RT_{t_j} \end{aligned} \tag{4.5}$$

1.) $\pi_1(rt_i) = id$, wobei $id \in \sum^*$ ein eindeutiger globaler Identifikator von rt_i ist.

2.) $\pi_2(r_i) = name$, wobei $name \in \sum^*$ eine Bezeichnung für rt_i ist, welche die Semantik des Relationstyp identifiziert.

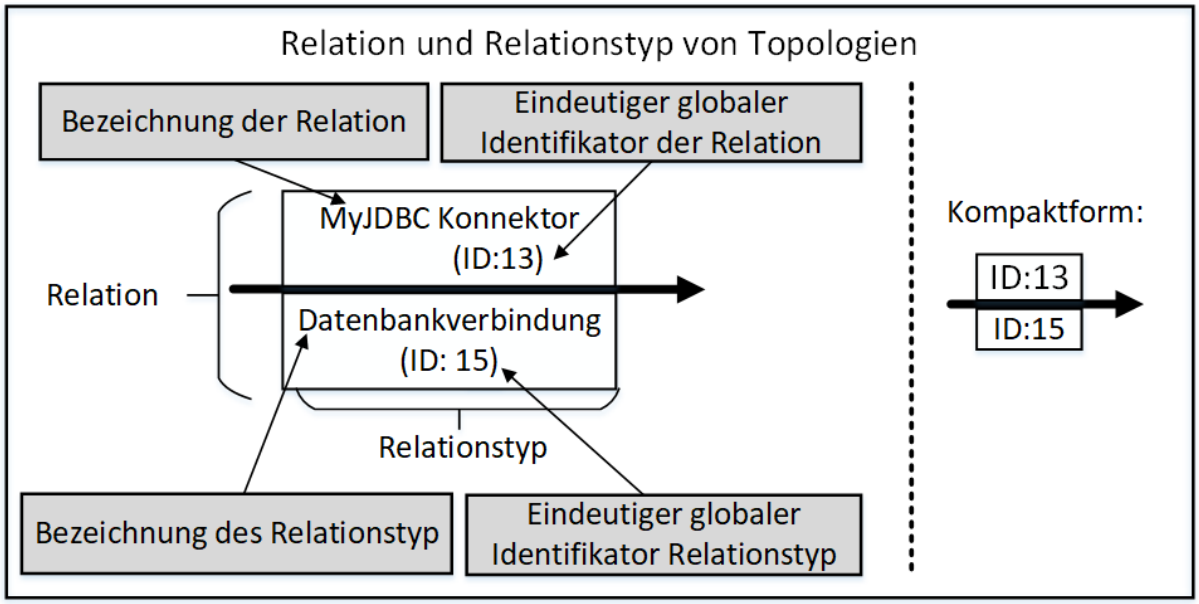

Abbildung 4.4: Notation Relation und Relationstyp

In der Abbildung 4.4 ist die graphische Notation, die für Relationen und Relationstypen in Topologien verwendet wird abgebildet. Dabei werden die Relationen durch gerichtete Pfeile zwischen den Komponenten dargestellt, wobei in dem oberen weißen Rechteck auf dem Pfeil der Name sowie die ID der Relation angegeben werden und unterhalb des Pfeiles der Name und die ID des Relationstyps der Relation angegeben werden. In dem Beispiel in der Abbildung 4.4 handelt es sich um eine MyJDBC Konnektor (ID:13) Relation, die von dem Relationstyp Datenbankverbindung (ID:15) ist. Auf der rechten Seite der Abbildung 4.3 ist die Kompaktform von Relationen und ihren Relationstypen abgebildet.

4.6 Erwartete und bereitgestellte Eigenschaften

Die Menge der Eigenschaftselemente aus dem DMMN Modell von Breitenbücher [Bre16] wird in eine Menge von erwarteten Eigenschaftselemente und eine Menge von bereitgestellten Eigenschaftselemente aufgeteilt. Bei diesen beiden Mengen handelt es sich um zwei disjunkte Mengen, dadurch soll die Semantik der jeweiligen Eigenschaften für den Verfeinerungsprozess unverkennbar voneinander abgegrenzt werden (vgl. Kap. 6.3). Des Weiteren wird in dieser Arbeit zur Vereinfachung angenommen, dass die Eigenschaftselemente nur Name-Wert Paare vom Datentyp String darstellen. Daher wird auf ein Datentypelement bei den Definitionen von den Tupeln der jeweiligen Eigenschaften verzichtet. In dem Verfeinerungsprozess werden Eigenschaften, die den gleichen Namen haben als übereinstimmende Eigenschaften angesehen. Dies führt dazu, dass über den Namen der Eigenschaften in dem Verfeinerungsprozess ein Matching durchgeführt werden kann. Im Kapitel 6.3 wird das Matching der erwarteten und

bereitgestellten Eigenschaften, welches innerhalb der entwickelten Verfeinerungsmethodik (vgl. Kap. 6) durchgeführt wird im Detail anhand von einem einfachen Beispiel erläutert.

Erwartete Eigenschaften

Eine erwartete Eigenschaft ist eine Eigenschaft, die ein Modellierer von einer Komponente oder Relation in der Topologie erwartet. Jede erwartete Eigenschaft wird von einem Modellierer keiner oder mehreren unterschiedlichen Komponenten oder Relationen zugeordnet. Dabei können keine oder mehrere erwartete Eigenschaften von einer Komponente oder Relation erwartet werden (vgl. Abb. 4.1). Bei den Modellierern kann es sich zum Beispiel um Systemarchitekten oder Benutzer des Systems handeln. Die Zuordnung der erwarteten Eigenschaften zu den jeweiligen Komponenten und Relationen in einer Topologie wird über die erwartete Eigenschaft Abbildung abgebildet (vgl. Kap. 4.8). Eine erwartete Eigenschaft könnte z.B. die Größe des erwarteten Arbeitsspeichers einer Datenbank Komponente sein. Formal werden erwartete Eigenschaften wie folgt nach [Bre16] angepasst definiert:

Definition 4.6.1 (Erwartete Eigenschaft)
Sei $EE_{t_j} = \{ee_1, ee_2, ..., ee_i, ..., ee_n | n \in \mathbb{N}\}$ die Menge aller erwarteten Eigenschaften in einer Topologie t_j, dann wird eine erwartete Eigenschafte $ee_i \in EE_t$ durch folgenden Tupel definiert:

$$\begin{gathered} EE_{t_j} \subseteq \sum^+ \times \sum^+ \times \sum^+ \\ ee_i = (id, name, wert) \in EE_{t_j} \end{gathered} \tag{4.6}$$

1.) $\pi_1(ee_i) = id$, wobei $id \in \sum^+$ ein eindeutiger globaler Identifikator von ee_i ist.

2.) $\pi_2(ee_i) = name$, wobei $name \in \sum^+$ eine Bezeichnung für ee_i ist.

3.) $\pi_3(ee_i) = wert$, wobei $wert \in \sum^+$ der Wert von ee_i ist.

Bereitgestellte Eigenschaften

Eine bereitgestellte Eigenschaft ist eine Eigenschaft, die von einer Komponente, einer Relation, einen Komponententyp oder einen Relationstyp, der gesamten Topologie zur Verfügung gestellt wird. Jede bereitgestellte Eigenschaft wird durch einen Modellierer keiner oder mehreren unterschiedlichen Komponenten, Relationen, Komponententypen oder Relationstypen in der Topologie zugeordnet und jede Komponente, Relation, Komponententyp oder Relationstyp kann keine oder mehrere Eigenschaften der Topologie bereitstellen (vgl. Abb. 4.1). Bei den Modellierern kann es sich zum Beispiel um Entwickler und Anbieter von einer Komponente oder Relation handeln. Die Zuordnung der bereitgestellten Eigenschaften zu den jeweiligen Komponenten, Relationen, Komponententypen und Relationstypen in einer Topologie wird über die bereitgestellte Eigenschaft Abbildung abgebildet (vgl. Def. 4.8.1). Eine bereitgestellte

Eigenschaft könnte z.B. der Anbieter sein von dem eine Komponente zur Verfügung gestellt wird. Formal werden die bereitgestellten Eigenschaften wie folgt nach [Bre16] angepasst definiert:

Definition 4.6.2 (Bereitgestellte Eigenschaft)
Sei $BE_{t_j} = \{be_1, be_2, ..., be_i, ..., be_n | n \in \mathbb{N}\}$ die Menge aller bereitgestellten Eigenschaften in einer Topologie t_j, dann wird eine bereitgestellte Eigenschafte $be_i \in BE_t$ durch folgenden Tupel definiert:

$$\begin{aligned} BE_{t_j} &\subseteq \sum^+ \times \sum^+ \times \sum^+ \\ be_i &= (id, name, wert) \in BE_{t_j} \end{aligned} \tag{4.7}$$

1.) $\pi_1(be_i) = id$, wobei $id \in \sum^+$ ein eindeutiger globaler Identifikator von be_i ist.

2.) $\pi_2(be_i) = name$, wobei $name \in \sum^+$ eine Bezeichnung für be_i ist.

3.) $\pi_3(be_i) = wert$, wobei $wert \in \sum^+$ der Wert von be_i ist.

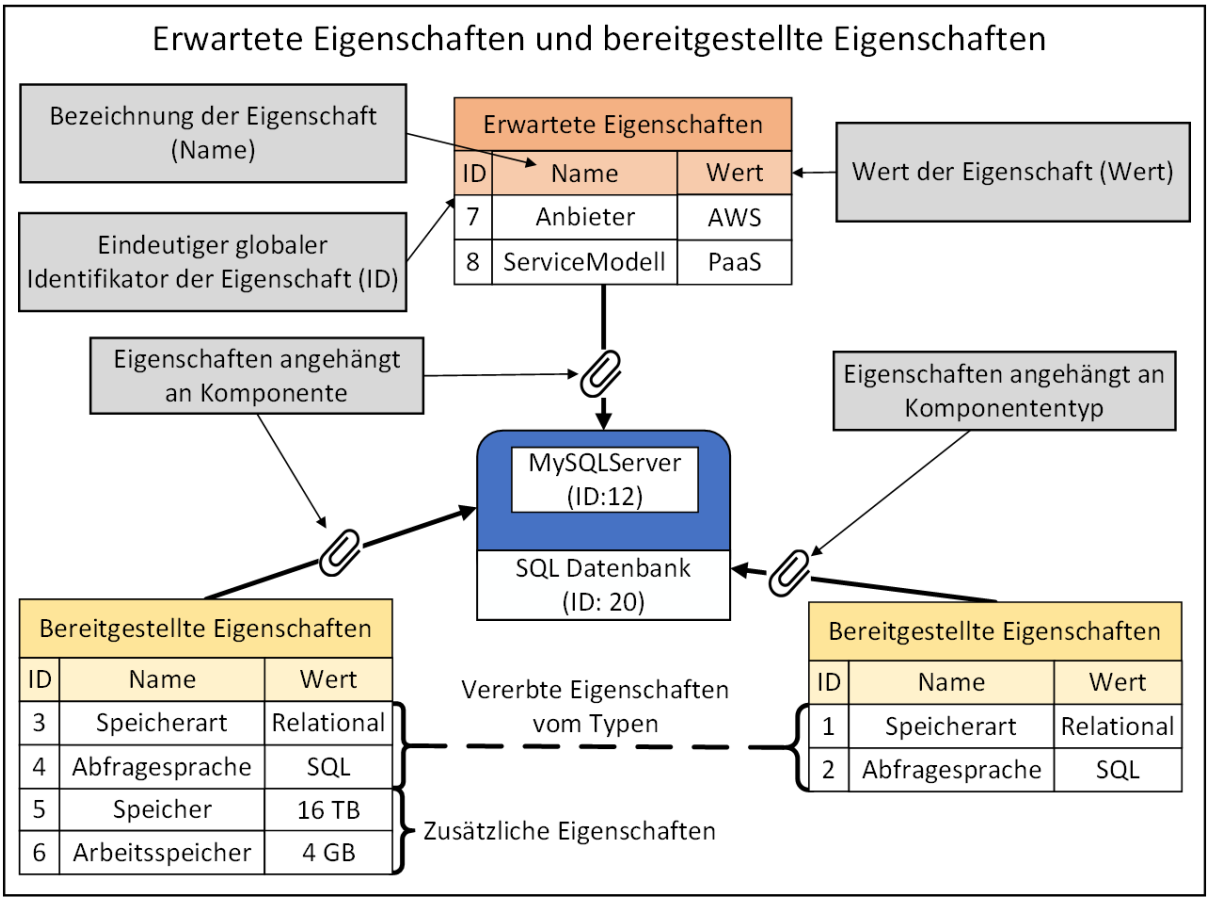

Abbildung 4.5: Notation erwartete und bereitgestellte Eigenschaften

Die Abbildung 4.5 zeigt die graphische Notation, für erwartete und bereitgestellte Eigenschaften von den Entitäten einer Topologie. Dabei werden die Eigenschaften wenn sie für Erklärungen benötigt werden, als Listen von Name-Wert Paare angehängt an die jeweiligen Topologie oder Level Graph Entitäten (vgl. Kap. 5.2) denen sie zugeordnet sind, dargestellt. In der Abbildung 4.5 sind als Beispiel die erwarteten und bereitgestellten Eigenschaften von einer MySQLServer Komponente (ID: 12), die vom Komponententyp SQL Datenbank (ID: 20) ist, abgebildet. Dabei wurde der MySQLServer Komponete (ID: 12), die erwarteten Eigenschaft Anbieter (ID: 7) mit dem Wert AWS und ServiceModell (ID: 8) mit dem Wert PaaS zugeordnet. Diese erwarteten Eigenschaften haben für den Verfeinerungsprozess die Bedeutung, dass der Modellierer von der MySQLServer Komponente (ID: 12) erwartet, dass diese durch ein PaaS Angebot von Amazon Web Services realisiert wird. Zudem erbt die MySQLServer Komponente (ID: 12) die bereitgestellten Eigenschaften Speicherart (ID: 1) und Abfragesprache (ID: 2) von dem zugeordneten SQL Datenbank (ID: 20) Komponententyp. Dabei kann eine Komponente noch zusätzliche weitere individuelle bereitgestellte Eigenschaften zu den bereits geerbten Eigenschaften des Komponententypen aufweisen. In dem aufgezeigten Beispiel wären die zusätzlichen individuellen bereitgestellten Eigenschaften der MySQLServer Komponente (ID: 12), die Eigenschaft Speicher (ID: 5) mit dem Wert 16 TB und Arbeitsspeicher (ID: 6) mit dem Wert 4 GB.

4.7 Typ Abbildung

Die Typ Abbildung ordnet jeder Komponente in der Topologie, genau einen Komponententypen der Topologie und jeder Relation in der Topologie, genau einen Relationstypen der Topologie zu. Formal wird die Typ Abbildung wie folgt nach [Bre16] angepasst definiert, wobei auf die Zuordnung von einem nicht definierten Typ verzichtet wird, da in dieser Arbeit zur Vereinfachung angenommen wird, dass jede Komponente und Relation genau einem Typen zugeordnet ist und dieser existiert und bekannt ist:

Definition 4.7.1 (Typ Abbildung)
Sei die Menge aller Elemente einer Topologie $TE_{t_j} = \{te_1, te_2, ..., te_i, ..., te_n | n \in \mathbb{N}\}$, die Vereinigung der Mengen aller Komponenten K_{t_j} und der Menge aller Relationen R_{t_j} einer Topologie t_j, dann ist $te_i \in TE_{t_j}$ genau ein Topologieelement dieser Vereinigung:

$$\begin{aligned} TE_{t_j} &= K_{t_j} \cup R_{t_j} \\ te_i &\in TE_{t_j} \end{aligned} \tag{4.8}$$

Und sei die Menge aller Typen einer Topologie $TET_{t_j} = \{tet_1, tet_2, ..., tet_i, ..., tet_n | n \in \mathbb{N}\}$, die Vereinigung der Menge aller Komponententypen KT_{t_j} und der Menge aller Relationstypen RT_{t_j} einer Topologie t_j, dann ist $tet_i \in TET_{t_j}$ genau ein Topologietyp dieser Vereinigung:

$$\begin{gathered} TET_{t_j} = KT_{t_j} \cup RT_{t_j} \\ tet_i \in TET_{t_j} \end{gathered} \tag{4.9}$$

Dann wird die Typ Abbildung einer Topologie typ_{t_j} wie folgt definiert:

$$typ_{t_j} : TE_{t_j} \to TET_{t_j}, te_i \mapsto tet_i \tag{4.10}$$

4.8 Eigenschafts Abbildungen

Die Eigenschaft Abbildung nach [Bre16] wird in dieser Arbeit in zwei separate Abbildungen aufgeteilt. Die $bereitgestellt_{t_j}$ Abbildung ordnet jeder Komponente, Relation, Komponententyp und Relationstyp in einer Topologie, keine oder mehrere bereitgestellte Eigenschaften zu. Wohingegen die $erwartet_{t_j}$ Abbildung jeder Komponente und Relation in einer Topologie, keine oder mehrere erwartete Eigenschaft zuordnet. Formal werden diese zwei Abbildungen wie folgt definiert:

Sei $ET_{t_j} = \{et_1, et_2, ...et_i, ..., et_n | n \in \mathbb{N}\}$ die Vereinigung der Mengen aller Topologieelemente TE_{t_j} und der Menge aller Topologietypen TET_{t_j} einer Topologie t_j, dann ist $et_i \in ET_{t_j}$ genau ein Eigenschaftsträger dieser Vereinigung:

$$\begin{gathered} ET_{t_j} = TE_{t_j} \cup TET_{t_j} \\ et_i \in ET_{t_j} \end{gathered} \tag{4.11}$$

Definition 4.8.1 (Bereitgestellte Eigenschaften Abbildung)

$$bereitgestellt_{t_j} : ET_{t_j} \to \mathcal{P}(BE_{t_j}) \tag{4.12}$$

Definition 4.8.2 (Erwartete Eigenschaften Abbildung)

$$erwartet_{t_j} : TE_{t_j} \to \mathcal{P}(EE_{t_j}) \tag{4.13}$$

5 Lösungskonzept

Damit eine Topologie automatisiert verfeinert werden kann, wird zusätzlich zu dem Topologiemodell ein weiteres Modell benötigt in welchem definiert wird, wie eine Topologie in eine andere Topologie transformiert werden kann. Dazu wurde in einem Lösungskonzept ein erster Entwurf von diesem Modell ausgearbeitet. Dieses Konzept wird im Folgenden im Detail definiert sowie seine Bestandteile, Strukturen und Eigenschaften beschrieben und ihre Bedeutung für die Verfeinerung erläutert. Das ausgearbeitete Verfeinerungsmodell des Lösungskonzepts wird als Level Graph Metamodell bezeichnet.

5.1 Grundkonzept

Die Grundidee für eine schrittweise Verfeinerung von Topologiemodellen anhand von Abstraktionsebenen ist, dass zu Beginn der Entwicklung von neuen Systemarchitekturen, diese zunächst durch ein abstraktes Topologiemodell auf einer konzeptionellen Abstraktionsebene beschrieben werden (vgl. Kap. 2.1.3). Diese abstrakten Topologiemodelle beinhalten dabei ganz allgemeingültige Komponenten, wie z.B. eine Datenbank Komponente oder eine Anwendungskomponente und allgemeingültige Relationen zwischen diesen Komponenten wie z.B. einen APP_DB_Konnektor oder eine ConnectedTo Abhängigkeitsbeziehung. Das erstellte abstrakte Topologiemodell dient als Ausgangsmodell für einen Verfeinerungsprozess.

Die Abbildung 5.1 stellt den Ablauf des Verfeinerungsprozesses schematisch dar. In der linken Hälfte ist die abstrakte Topologie, die von System- oder Softwarearchitekten modelliert oder falls eine abstrakte Topologie bereits existiert, ausgewählt und angepasst wird, abgebildet (vgl. Abb. 5.1 (1)). Diese abstrakte Topologie wird mit erwarteten Eigenschaften, welche z.B. Anforderungen an die Systemarchitektur darstellen, angereichert und an einen Topologie Transformator übergeben (vgl. Abb. 5.1 (2)). Dieser Transformator führt eine Verfeinerung und Konkretisierung, anhand eines Verfeinerungsmodells durch und generiert neue spezifische Topologiemodelle. Diese Topologiemodelle erfüllen alle erwarteten Eigenschaften der abstrakten Topologie durch bereitgestellte Eigenschaften (vgl. Kap. 6.3) und beinhalten dabei spezifische Ausprägungsformen der einzelnen Bestandteile von der abstrakten Topologie, wie zum Beispiel eine EC2 und eine DynamoDB Service Komponente von Amazon Web Services [Amab] oder eine JDBC_SQL_DB_Konntektor Relation. Allerdings kann ein spezifisches Topologiemodell auch Bestandteile der abstrakten Topologie enthalten wenn dafür keine speziellen Ausprägungsformen existieren, wie zum Beispiel eine allgemeingültige „HostedOn“ Relation.

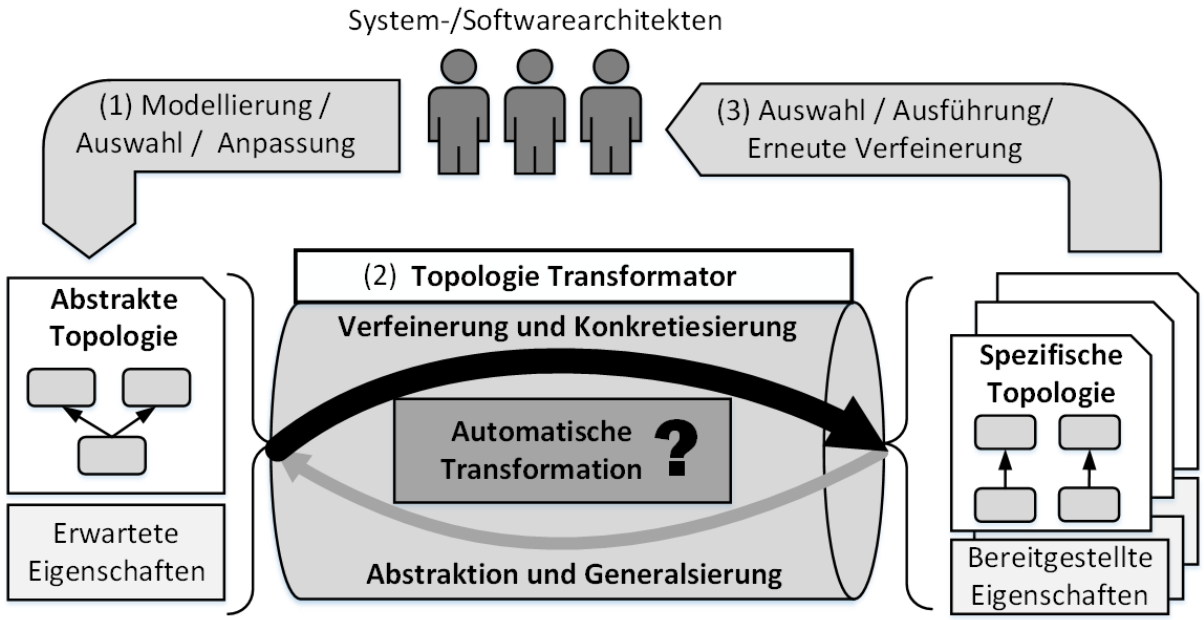

Abbildung 5.1: Schematischer Ablauf des Verfeinerungsprozess

Der Systemarchitekt kann daraufhin eine spezifische Topologie auswählen und ausführen lassen. Falls die spezifische Topologie keine ausführbare Topologie ist, kann die spezifische Topologie erneut angepasst werden und als neue abstrakte Topologie für einen weiteren Verfeinerungsschritt verwendet werden (vgl. Abb. 5.1 (3)). Um die Transformation zu automatisieren wird innerhalb des Topologie Transformator ein Algorithmus für die Verfeinerung von Topologiemodellen definiert und ausgeführt (vgl. Kap. 6.4). Dazu wurde ein Level Graph Metamodell für Topologiemodelle erarbeitet, welches als Verfeinerungsmodell in dem Transformator dient. In der Abbildung 5.2 werden die vier identifizierten Grundprinzipien, welche für die Verfeinerung von Topologien von Bedeutung sind anhand eines abstrakten Beispiels aufgezeigt.

Das erste Prinzip für die Verfeinerung ist, die Repräsentation von Topologieelementen in dem Verfeinerungsmodell. Dieses wird im oberen Teil der Abbildung 5.2 durch, die „repräsentiert durch“ Beziehungen angedeutet. Dabei wird jede Komponente und Relation einer Topologie durch einen Knoten in dem Verfeinerungsmodell repräsentiert. In der Abbildung 5.2 wird z.B. die Komponente K1 mit dem Komponententyp KT1 durch einen Knoten mit dem gleichen Komponententypen KT1 und die Relation R1 mit dem Relationstyp RT1 durch einen Knoten mit dem gleichen Relationstyp RT1 in der Ausgangsabstraktionsebene des Verfeinerungsmodells (2) repräsentiert. Das Repräsentationsprinzip wird in dieser Arbeit durch das Topologie Metamodell (vgl. Kap. 4) und das Level Graph Metamodell festgelegt (vgl. Kap. 5.2).

Dabei wird durch die schwarzen Kanten zwischen den Knoten innerhalb der Ausgangsabstraktionsebene und Zielabstraktionsebene des Verfeinerungsmodell (2) in der Abbildung 5.2, die Kompatibilität zwischen Knoten, zwischen Knoten und Fragmenten sowie zwischen Fragmen-

ten im Verfeinerungsmodell identifiziert. Zum Beispiel gibt die Kante zwischen den Knoten KT1 und RT1 in der Ausgangsabstraktionsebenen des Verfeinerungsmodells (2) an, dass eine beliebige Komponente von dem Komponententyp KT1 eine gültige Quell Komponente von einer beliebigen Relation vom Relationstyp RT1 ist.

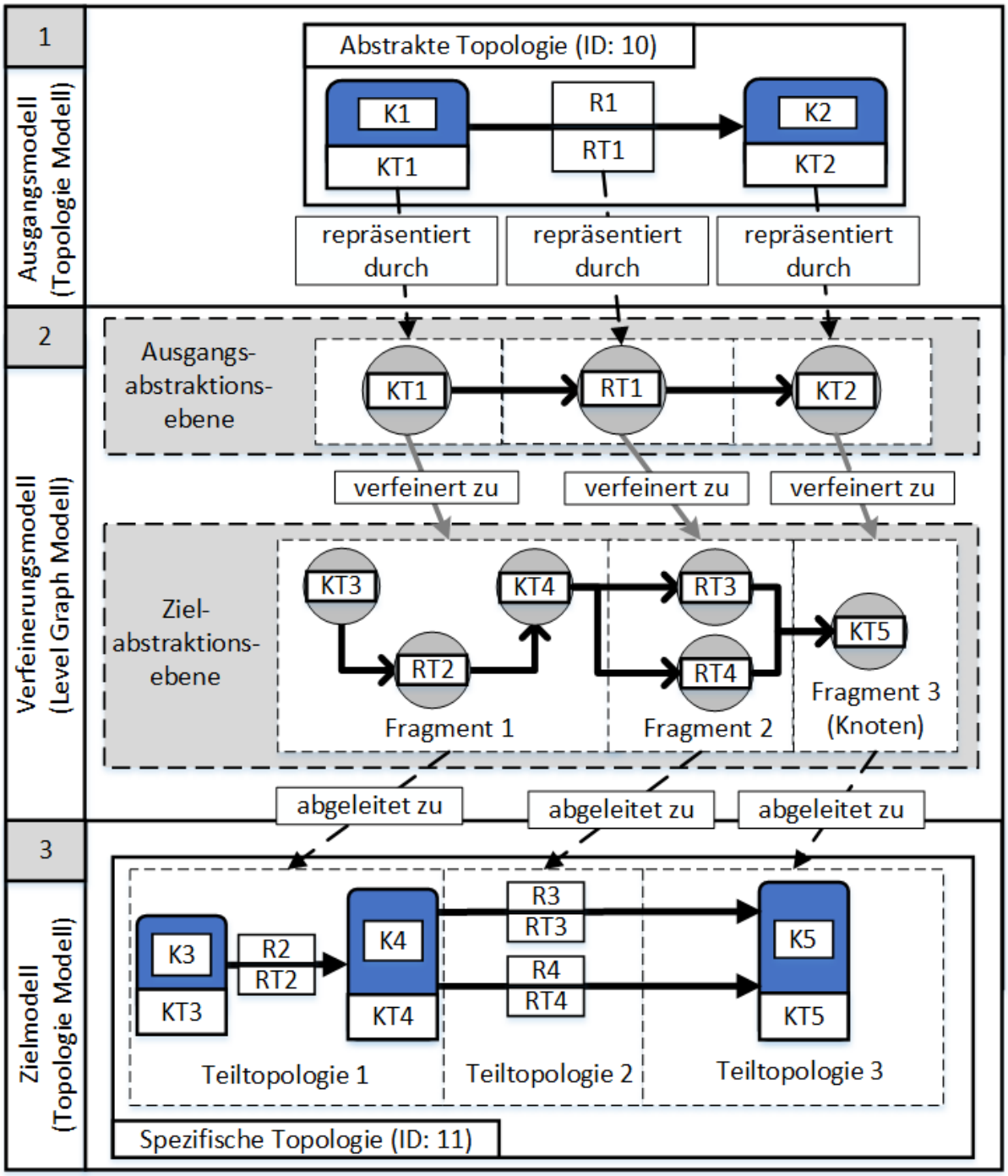

Abbildung 5.2: Grundprinzipien der Verfeinerung von Topologiemodelle

Wohingegen die Kante zwischen den Knoten RT1 und KT2 angibt, dass eine beliebige Komponente vom Komponententyp KT2 eine gültige Zielkomponente, einer beliebigen Relation vom Relationstypen RT1 ist. Dieses Prinzip ist das Kompatibilitätsprinzip, welches im Detail durch den Kompatibilitätslevel-Graphen definiert wird (vgl. Kap. 5.4).

Das dritte Prinzip wird in der mittleren Schicht in der Abbildung 5.2 (2) durch das Verfeinerungsmodell mit den „verfeinert zu“ Beziehungen angedeutet. Daher wird dieses Prinzip auch als Verfeinerungsprinzip bezeichnet. Die grauen „verfeinert zu“ Kanten geben dabei an, in welche Knoten oder Fragmente der Zielabstraktionsebene, die Knoten der Ausgangsabstraktionsebene verfeinert werden können. In diesem Beispiel könnte der Knoten KT1 der Ausgangsabstraktionsebene in ein Fragment 1 verfeinert werden, welches aus drei Knoten KT3, RT2 und KT4 besteht, die miteinander verbunden sind und der Knoten KT2 auf der rechten Seite der Ausgangsabstraktionsebene könnte in einen anderen Knoten KT5 verfeinert werden. Die Verfeinerungsprinzipien werden in dem Kapitel 5.7 durch den Verfeinerungs-Level Graphen im Detail vorgestellt. Nachdem alle Elemente der abstrakten Topologie (ID: 10) in der Abstraktionsebene gefunden und in die einzelnen Knoten oder Fragmente verfeinert sowie auf Kompatibilität zueinander geprüft wurden, muss ein Topologiemodell aus der gefundenen Lösung generiert werden.

Die Ableitung von einer spezifischen Topologie, aus der gefundenen Lösung der Zielabstraktionsebene vom Verfeinerungsmodell stellt das letzte Prinzip dar. Dieses Prinzip wird in der Abbildung 5.2 im unteren Teil zwischen der Verfeinerungsmodell (2) und Zielmodell (3) Ebene durch die „abgeleitet zu“ Beziehungen angedeutet. Dabei wird jedes generierte Fragment in eine Teiltopologie abgeleitet und mit den anderen kompatiblen Teiltopologien miteinander verbunden. Zum Beispiel wird das Fragment 1 mit den Knoten KT3, RT2 und KT4 in eine geschlossene Teiltopologie 1 abgeleitet. Diese Teiltopologie 1 besteht dabei aus einer Komponente K3 vom Komponententyp KT3, die mit einer Relation R2 vom Relationstyp RT2 mit einer anderen Komponente K4 vom Komponententypen KT4 verbunden ist. Dabei stellt eine geschlossene Teiltopologie ein Topologiemodell dar, in dem jede Relation eine Quell- und Zielkomponente hat, wohingegen eine offene Teiltopologie ein Topologiemodell darstellt, die mindestens eine Relation ohne Quell- und eine Relation ohne Zielkomponente besitzt. Dadurch können offene Teiltopologien dazu verwendet werden zwei geschlossene Teiltopologien miteinander zu verbinden. Die unterschiedlichen Ableitungsmöglichkeiten von Topologien werden im Kap. 5.5 im Detail vorgestellt. Dabei stellt das Ableitungsprinzip das Gegenstück zu dem Repräsentationsprinzip dar.

Durch diese vier Grundprinzipien kann die abstrakte Topologie mit der ID 10 im aufgezeigten Beispiel auf Basis des Verfeinerungsmodells mit den gemeinsamen verwendeten Komponententypen und Relationstypen in eine spezifische Topologie mit der ID 11 verfeinert werden. Bei dieser Art der Modell Transformation handelt es sich um eine endogene vertikale Verfeinerungstransformation, die im Kapitel 2.4 erläutert wurde. In den folgenden Kapiteln werden die oben erwähnten vier Grundprinzipien genauer ausformuliert, damit anschließend der Verfeinerungsprozess und der Verfeinerungsalgorithmus für Topologiemodelle definiert werden kann.

5.2 Metamodell des Level Graphen

Im Folgenden wird das Metamodell des Level Graphen definiert, anhand dessen später die Eigenschaften und Strukturen von Level Graph Modellen abgeleitet werden können. Dabei werden die Entitäten, die für die Beschreibung des Algorithmus sowie für weitere Definitionen benötigt werden formal im Detail definiert, bei den restlichen Entitäten wird auf eine detaillierte formale Definition an dieser Stelle verzichtet. Um vorab einen Überblick über die Entitäten des Level Graph Metamodell sowie dessen Beziehungen zueinander zu vermitteln, wird das Level Graph Metamodell in der Abbildung 5.3 gesamtheitlich als ER-Modell dargestellt. In diesem sind auch die Basiselemente B_{lg_j}, die auch im Topologie Metamodell (vgl. Kap. 4 i.v.m Abb. 4.1) verwendet werden, in der oberen Mitte mit einem hellgrauen Rechteck gekennzeichnet. Die Menge der Basiselemente B_{lg_j} ist die Vereinigung der Menge der Komponententypen KT_{lg_j} (vgl. Def. 4.3.1), der Menge der Relationstypen RT_{lg_j} (vgl. Def. 4.5.1) und der Menge der bereitgestellten Eigenschaften BE_{lg_j} (vgl. Def. 4.6.2). Allerdings gehört die Menge der erwarteten Eigenschaften nicht zu den Basiselementen, des Level Graph Metamodell, da diese lediglich den Komponenten und Relationen in einer Topologie zugeordnet werden und in dem Verfeinerungsmodell somit nicht benötigt werden. Das Level Graph Metamodell legt fest, wie

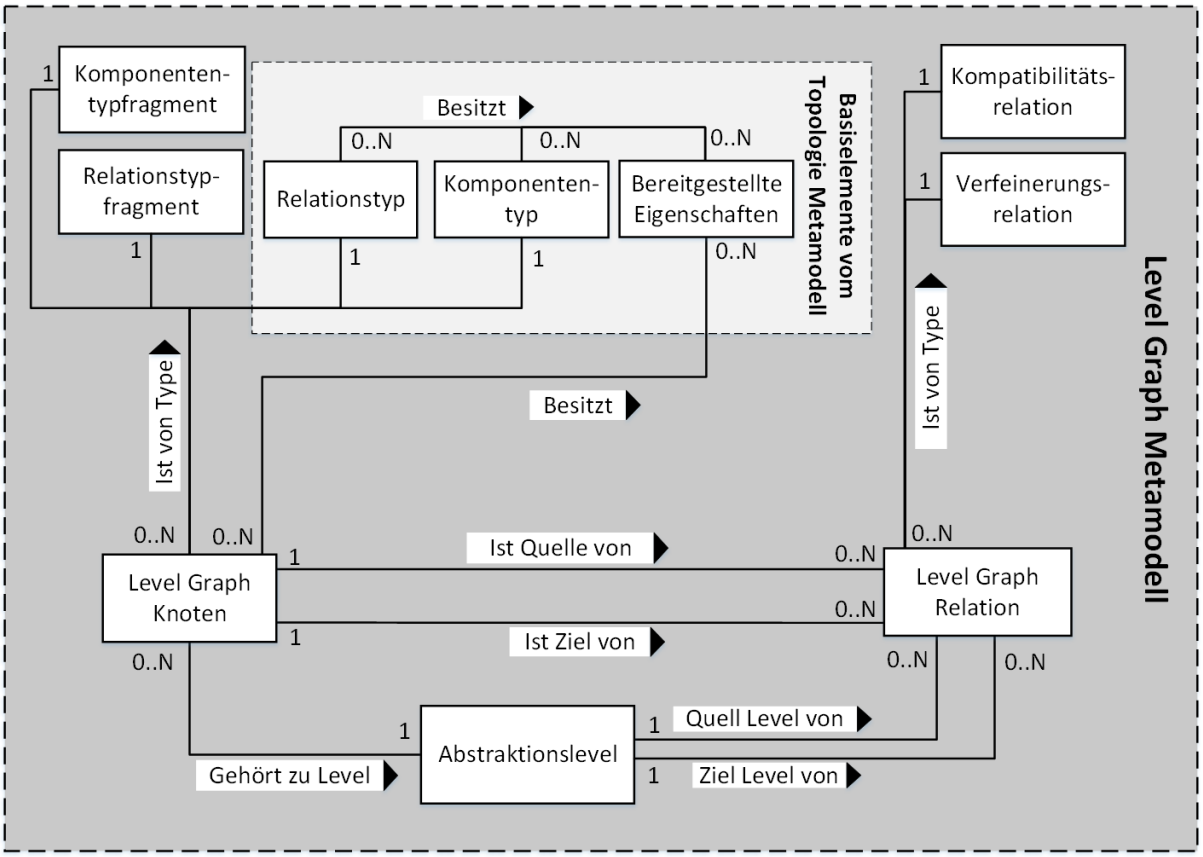

Abbildung 5.3: Metamodell von Level Graphen

valide Level Graph Modelle darzustellen sind und welche Beziehungen zwischen den einzelnen Entitäten mit welchen Kardinalitäten existieren. Die einzelnen Entitäten und die jeweiligen Beziehungen dieser werden in den nachfolgenden Abschnitten im Detail definiert. Daher wird an dieser Stelle auf eine detaillierte Erklärung verzichtet.

5.2.1 Level Graph

Ein Level Graph stellt einen gerichteten, gefärbten, gewichteten, möglicherweise zyklischen und nicht zusammenhängenden Graphen mit mindestens zwei Abstraktionslevel dar. Wobei die Level Graph Knoten genau einem Abstraktionslevel zugeordnet sind. Durch diese Zuordnung wird eine hierarchische Struktur in dem Level Graphen definiert. Level Graphen sollen dazu dienen, die Abstraktionsabstufungen zwischen den Komponententypen und Relationstypen festzulegen. Durch diese Abstufung kann anschließend abgeleitet werden, welche abstrakte Komponententypen (vgl. Kap. 4.3) und Relationstypen (vgl. Kap. 4.5) in einem Abstraktionslevel, in welche spezifischen anderen Komponententypen und Relationstypen eines niedrigeren Abstraktionslevel verfeinert werden können. Allerdings ist es auch möglich, dass abstrakte Komponententypen und Relationstypen in tieferen Abstraktionslevel wiederholt vorkommen können. Dadurch soll ermöglicht werden, dass abstrakte Komponententypen und Relationstypen, die sowohl in abstrakten Topologien und spezifischen Topologien vorkommen, durchgehend verfeinert werden können. Zum Beispiel könnte eine MyHostedOn Relation von einem HostedOn Relationstyp, sowohl in einem konzeptionellen Topologiemodell, als auch in einem ausführbaren Topologiemodell vorkommen. Allerdings führt eine Verwendung von abstrakten Typen in tieferen Abstraktionslevel dazu, dass nicht mehr unverkennbar die Abstraktionsgrade von Komponententypen und Relationstypen anhand des Level Graphen abgeleitet werden können. Formal wird ein Level Graph wie folgt definiert:

Definition 5.2.1 (Level Graph)
Sei $LG = \{lg_1, lg_2, ..., lg_\text{j}, ..., lg_\text{n} | n \in \mathbb{N}\}$ die Menge aller Level Graphen, dann wird ein Level Graph $lg_\text{j} \in LG$, durch folgenden 11-Tupel repräsentiert, wobei acht Elemente davon Mengen von Entitäten und drei davon Abbildungen zwischen den Entitäten repräsentieren:

$$\begin{gathered} lg_\text{j} = (AL_{\text{lg}_\text{j}}, LGK_{\text{lg}_\text{j}}, LGR_{\text{lg}_\text{j}}, KT_{\text{lg}_\text{j}}, RT_{\text{lg}_\text{j}}, \\ KTF_{\text{lg}_\text{j}}, RTF_{\text{lg}_\text{j}}, BE_{\text{lg}_\text{j}}, level_{\text{lg}_\text{j}}, typ_{\text{lg}_\text{j}}, bereitgestellt_{\text{lg}_\text{j}}) \\ lg_\text{j} \in LG \end{gathered} \tag{5.1}$$

Für die Basiselemente $B_{\text{lg}_\text{j}} = BE_{\text{lg}_\text{j}} \cup KT_{\text{lg}_\text{j}} \cup RT_{\text{lg}_\text{j}}$ des Level Graphen können die Definitionen, die bereits beim Topologie Metamodell für die einzelnen Teilmengen der Basiselemente verwendet wurden analog auf den Level Graphen angepasst werden (vgl. Def. 4.3.1, Def. 4.5.1 und Def. 4.6.2). Daher wird in den weiteren Abschnitten darauf verzichtet diese im Detail für den Level Graphen erneut zu definieren. Dies gilt auch für die Typ und bereitgestellte

Eigenschaft Abbildung, die in den Kapiteln 4.7 und 4.8 für das Topologie Metamodell bereits definiert wurden. Dabei ist jedoch anzumerken, dass die erwartete Eigenschaft Abbildung in dem Level Graph Modell nicht benötigt wird, da den Entitäten im Level Graphen keine erwarteten Eigenschaften zugeordnet werden (vgl. Abb. 5.3) und die Typ Abbildung des Level Graphen, zusätzlich zu den Relationstypen und Komponententypen auch die Komponententypfragmente und Relationstypfragmente den jeweiligen Level Graph Knoten zuordnet (vgl. Abb. 5.3 i.V.m. Kap. 5.2.3).

Die einzelnen Elemente des Tupels, werden dabei wie folgt definiert:

1.) $AL_{\mathrm{lg_j}}$ ist die Menge aller Abstraktionslevel in dem Level Graphen lg_{j}, wobei jedes $al_{\mathrm{i}} \in AL_{\mathrm{lg_j}}$ ein Abstraktionslevel in dem Level Graphen lg_{j} repräsentiert (vgl. Def. 5.2.2).

2.) $LGK_{\mathrm{lg_j}}$ ist die Menge aller Level Graph Knoten, die in dem Level Graphen lg_{j} vorkommen, wobei $lgk_{\mathrm{i}} \in LGK_{\mathrm{lg_j}}$ genau einen Level Graph Knoten in dem Level Graphen lg_{j} repräsentiert (vgl. Def. 5.2.3).

3.) $LGR_{\mathrm{lg_j}}$ ist die Menge aller Level Graph Relationen in dem Level Graphen lg_{j}, wobei $lgr_{\mathrm{i}} \in LGR_{\mathrm{lg_j}}$ genau eine gerichtete Level Graph Relation zwischen zwei Level Graph Knoten in dem Level Graphen lg_{j} repräsentiert (vgl. Def. 5.2.7).

4.) $KT_{\mathrm{lg_j}}$ ist die Menge aller Komponententypen in dem Level Graphen lg_{j}, wobei jedes $kt_{\mathrm{i}} \in KT_{\mathrm{lg_j}}$ die Semantik für jeden $lgk_{\mathrm{i}} \in LGK_{\mathrm{lg_j}}$ beschreibt, die diesem Komponententyp durch die $typ_{\mathrm{lg_j}}$ Abbildung zugeordnet ist (analog Def. 4.3.1).

5.) $RT_{\mathrm{lg_j}}$ ist die Menge aller Relationstypen in dem Level Graphen lg_{j}, wobei jeder $rt_{\mathrm{i}} \in RT_{\mathrm{lg_j}}$ die Semantik für jeden $lgk_{\mathrm{i}} \in LGK_{\mathrm{lg_j}}$ beschreibt, die diesem Relationstypen durch die $typ_{\mathrm{lg_j}}$ Abbildung zugeordnet ist (analog Def. 4.5.1).

6.) $KTF_{\mathrm{lg_j}}$ ist die Menge aller Komponententypfragmente in dem Level Graphen lg_{j}, wobei jedes $ktf_{\mathrm{i}} \in KTF_{\mathrm{lg_j}}$ die Semantik für jeden $lgk_{\mathrm{i}} \in LGK_{\mathrm{lg_j}}$ beschreibt, die diesem Komponententypfragment durch die $typ_{\mathrm{lg_j}}$ Abbildung zugeordnet ist (vgl. Def. 5.2.8).

7.) $RTF_{\mathrm{lg_j}}$ ist die Menge aller Relationstypfragmente in dem Level Graphen lg_{j}, wobei jeder $rtf_{\mathrm{i}} \in RTF_{\mathrm{lg_j}}$ die Semantik für jeden $lgk_{\mathrm{i}} \in LGK_{\mathrm{lg_j}}$ beschreibt, die diesem Relationstypfragment durch die $typ_{\mathrm{lg_j}}$ Abbildung zugeordnet ist (vgl. Def. 5.2.9).

8.) $BE_{\mathrm{lg_j}}$ ist die Menge aller bereitgestellten Eigenschaften, die durch Modellierer den einzelnen Entitäten des Level Graphen lg_{j} zugeordnet wurden. Wobei jedes $be_{\mathrm{i}} \in BE_{\mathrm{lg_j}}$ eine bereitgestellte Eigenschaft beschreibt, die einem oder mehreren Level Graph Knoten lgk_{i}, Komponententypen kt_{i} oder Relationstypen rt_{i} über die $bereitgestellt_{\mathrm{lg_j}}$ Abbildung zugeordnet und dem Level Graphen lg_{j} zur Verfügung gestellt wird.

Und die einzelnen Abbildungen des Tupels werden wie nachfolgend definiert:

9.) $level_{\mathrm{lg_j}}$ ist die Abstraktionslevel Abbildung, die jedem $lgk_{\mathrm{i}} \in LGK_{\mathrm{lg_j}}$ genau einen Abstraktionslevel $al_{\mathrm{i}} \in AL_{\mathrm{lg_j}}$ zuordnet (vgl. Def. 5.2.4).

10.) typ_{lg_j} ist die Typ Abbildung die jedem $lgk_\text{i} \in LGK_{\text{lg}_\text{j}}$ einen Komponententypen kt_i, einen Relationstypen rt_i, einen Komponententypfragment ktf_i oder einen Relationstypfragment rtf_i zuordnet.

11.) $bereitgestellt_{\text{lg}_\text{j}}$ ist die bereitgestellt Eigenschaft Abbildung die jedem Level Graph Knoten lgk_i, Komponententypen kt_i und Relationstypen rt_i keine oder mehrere bereitgestellte Eigenschaften $BET \subseteq BE_{\text{lg}_\text{j}} \cup \emptyset$ zuordnet.

Die Definitionen und graphische Notationen der einzelnen Elemente eines Level Graphen werden in den nachfolgenden Kapiteln im Einzelnen definiert und erläutert.

5.2.2 Abstraktionslevel

Ein Abstraktionslevel im Level Graphen repräsentiert eine Teilmenge von Level Graph Knoten und Level Graph Relationen des Level Graphen. Die Zuordnung der Level Graph Relationen zu den Abstraktionslevels, ergibt sich durch die Zuordnung der Quell- und Ziel Level Graph Knoten der Level Graph Relationen. Alle Level Graph Knoten, die in demselben Abstraktionslevel in einem Level Graphen sind, weisen den gleichen Abstraktionsgrad auf. Der Abstraktionsgrad wird durch die Tiefe des jeweiligen Abstraktionslevels identifiziert, dabei stellte die Tiefe 0 den höchsten Abstraktionsgrad in einem Level Graphen dar. Formal werden die Abstraktionslevel wie folgt definiert:

Definition 5.2.2 (Abstraktionslevel)
Sei $AL_{\text{lg}_\text{j}} = \{al_1, al_2, ..., al_\text{i}, ..., al_\text{n} | n \in \mathbb{N} \wedge |AL_{\text{lg}_\text{j}}| \geq 2\}$ die nicht leere Menge aller Abstraktionslevels in einem Level Graphen $lg_\text{j} \in LG$, dann wird ein Abstraktionslevel $al_\text{i} \in AL_{\text{lg}_\text{j}}$ durch folgenden Tupel definiert:

$$\begin{gathered} AL_{\text{lg}_\text{j}} \subseteq \sum^{+} \times \sum^{+} \times \mathbb{N} \\ al_\text{i} = (id, name, tiefe) \in AL_{\text{lg}_\text{j}} \end{gathered} \tag{5.2}$$

1.) $\pi_1(al_\text{i}) = id$, wobei $id \in \sum^{+}$ ein eindeutiger globaler Identifikator von al_i ist.

2.) $\pi_2(al_\text{i}) = name$, wobei $name \in \sum^{+}$ eine Bezeichnung für al_i ist.

3.) $\pi_3(al_\text{i}) = tiefe$, wobei $tiefe \in \mathbb{N}$ die eindeutige Tiefe von al_i ist und dadurch eine Ordnung zwischen allen Abstraktionslevel aufgrund der Ordnungsrelation der Menge der natürlichen Zahlen und der Eindeutigkeit von $\pi_3(al_\text{i})$, impliziert wird.

Die Anzahl von Abstraktionslevel kann beliebig von Level Graph Modell zu Level Graph Modell variieren, aber es empfiehlt sich bei der Wahl der Anzahl von Abstraktionslevels sich an, die in Kapitel 2.1.3 genannte Anzahl von drei Abstraktionsebenen einer Systemarchitektur zu orientieren. In dieser Arbeit werden ausschließlich Level Graph Modelle mit maximal drei Abstraktionslevel betrachtet und untersucht. Dabei können die drei Abstraktionslevel auf, die

genannten Abstraktionsebenen im Kap. 2.1.3 gemappt werden. In der Abbildung 5.4 ist die graphische Notation, die für Abstraktionslevel in Level Graphen verwendet wird abgebildet, wobei exemplarisch die Konzeptebene (vgl. Kap. 2.1.2) als Swimlane dargestellt ist. Dabei kann diese Konzeptebene über den globalen Identifikator 1023 eindeutig identifiziert werden und weist eine Tiefe von 0 auf, womit dieses Abstraktionslevel den höchsten Abstraktionsgrad in einem Level Graphen darstellt. Je größer die Tiefe eines Abstraktionslevels ist, desto niedriger ist der Abstraktionsgrad von den Entitäten, die diesem Abstraktionslevel zugeordnet sind und desto höher ist der Detaillierungsgrad von diesen. Auf der rechten Hälfte der Abbildung ist die Kompaktform der graphischen Notation abgebildet, bei der lediglich die Tiefe des Abstraktionslevels im linken Rechteck der Swimlane angegeben wird.

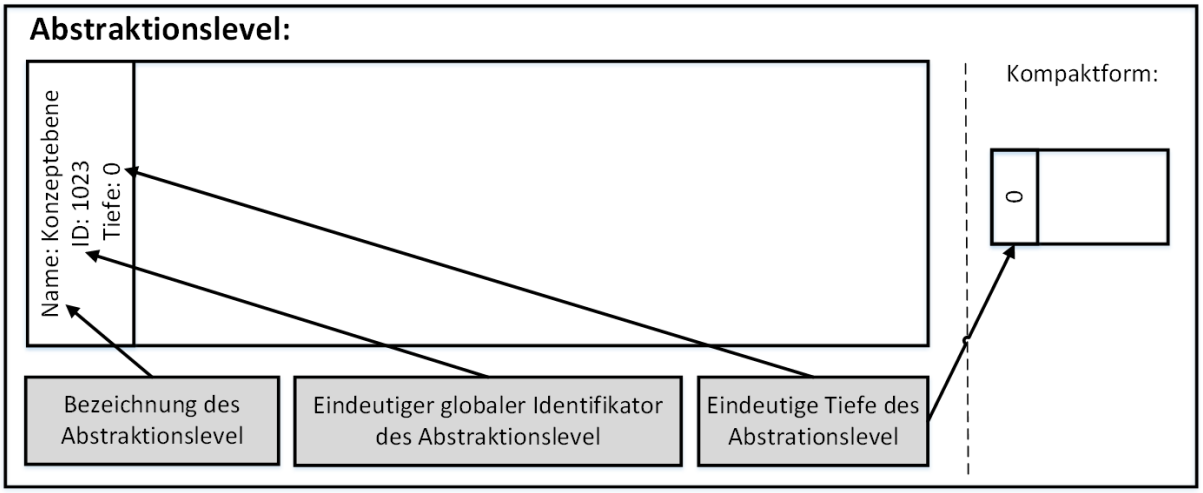

Abbildung 5.4: Notation von Abstraktionslevel

5.2.3 Level Graph Knoten

Ein Level Graph Knoten repräsentiert genau einen Knoten in einem Level Graphen. Jeder Level Graph Knoten in einem Level Graphen ist genau einem Komponententyp, einem Relationstyp, einem Komponententypfragment oder einem Relationstypfragment zugeordnet (vgl. Abb. 5.3)). Der Typ von dem Level Graph Knoten beschreibt die Semantik von einem Level Graph Knoten im Level Graphen. Wenn der Level Graph Knoten zum Beispiel vom Komponententyp ist, kann dieser Knoten in eine Komponente in der Topologie abgeleitet werden, die von dem selben Komponententyp wie der Level Graph Knoten ist (vgl. Kap. 5.5 und Abb. 5.15). Die Ableitungsregeln für die unterschiedlichen Typen werden im Detail in Kapitel 5.5 ausführlich erläutert. Des Weiteren ist jeder Level Graph Knoten einem Abstraktionslevel des Level Graphen über die Level Abbildung zugeordnet (vgl. Kap. 5.2.4). Jeder Level Graph Knoten kann zudem keine oder mehrere eingehende und ausgehende Level Graph Relationen haben, diese werden über die Menge der Level Graph Relationen abgebildet (vgl. Abb. 5.3). Formal wird ein Level Graph Knoten wie folgt definiert:

Definition 5.2.3 (Level Graph Knoten)
Sei $LGK_{\text{lg}_\text{j}} = \{lgk_1, lgk_2, ..., lgk_\text{i}, ..., lgk_\text{n} | n \in \mathbb{N}\}$ die Menge aller Level Graph Knoten in einem Level Graphen $lg_\text{j} \in LG$, dann wird ein Level Graph Knoten $lgk_\text{i} \in LGK_{\text{lg}_\text{j}}$ durch folgenden Tupel definiert:

$$\begin{aligned} LGK_{\text{lg}_\text{j}} &\subseteq \sum^+ \times \sum^+ \\ lgk_\text{i} &= (id, name) \in LGK_{\text{lg}_\text{j}} \end{aligned} \tag{5.3}$$

1.) $\pi_1(lgk_\text{i}) = id$, wobei $id \in \sum^+$ ein eindeutiger globaler Identifikator von lgk_i ist.

2.) $\pi_2(lgk_\text{i}) = name$, wobei $name \in \sum^+$ eine Bezeichnung für lgk_i ist.

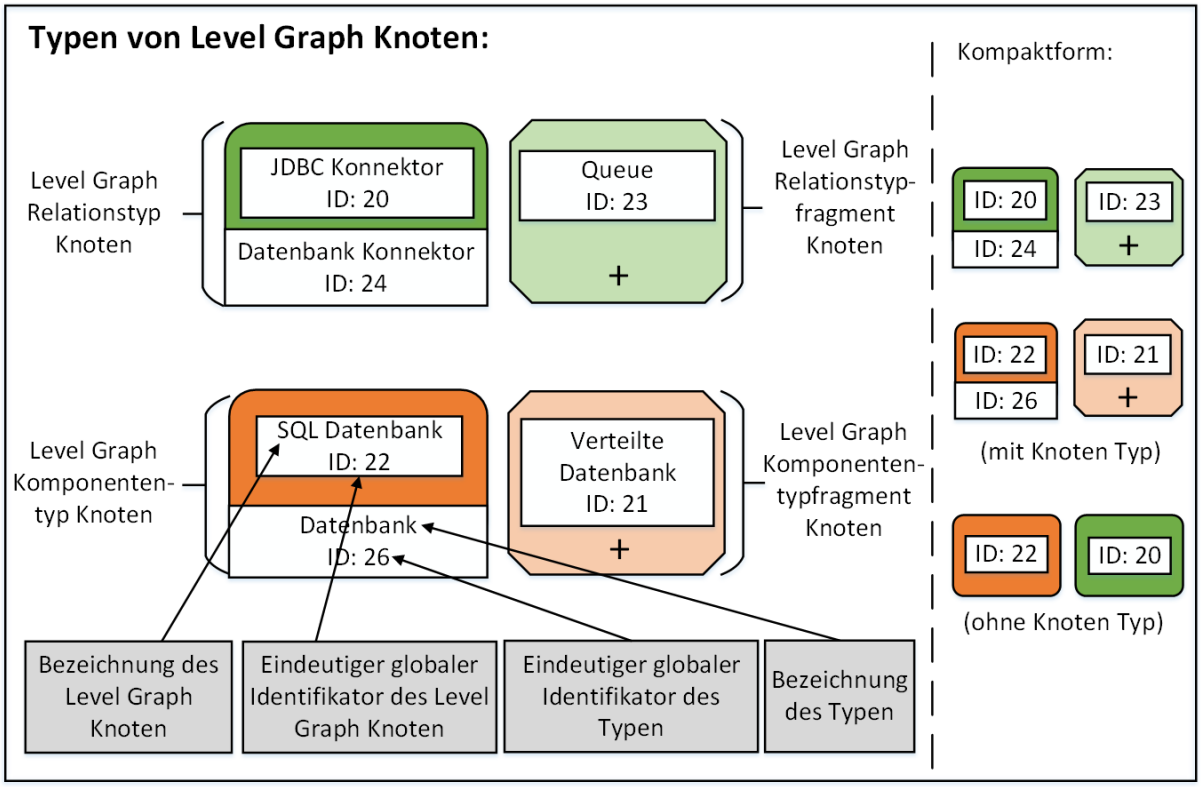

Abbildung 5.5: Notation von Level Graph Knoten

In der Abbildung 5.5 sind die graphischen Notationen, die für die unterschiedlichen Typen von Level Graph Knoten verwendet werden abgebildet. Dabei werden Level Graph Knoten vom Relationstyp (vgl. Kap. 4.5) und Komponententyp (vgl. Kap. 4.3) links in der Abbildung durch oben abgerundete Rechtecke dargestellt, wobei Level Graph Knoten vom Relationstyp dunkelgrün gefärbt werden und Level Graph Knoten vom Komponententyp dunkelorange gefärbt werden. Wohingegen die Level Graph Knoten daneben vom Relationstypfragment und vom Komponententypfragment durch Achtecke mit einem Plus Symbol dargestellt werden,

wobei Relationstypfragmente hellgrün gefärbt sind und Komponententypfragmente hellorange gefärbt sind. In dem mittleren weißen Rechteck wird der Name sowie die ID des jeweiligen Level Graph Knoten angegeben. In dem unteren angehängten weißen Rechteck bei den Level Graph Knoten vom Relationstyp und Komponententyp, wird der Name sowie die ID von dem jeweiligen zugeordneten Typen angegeben. Daher stellt zum Beispiel der linke obere Knoten einen JDBC Konnektor Level Graph Knoten mit der ID 20 dar, welcher von dem Datenbank Konnektor Relationstyp mit der ID 24 ist. Dieser Level Graph Knoten könnte in eine JDBC Konnektor Relation vom Relationstyp Datenbank Konnektor in einem Topologie Graphen abgeleitet werden (vgl. Kap. 5.5). Auf der rechten Seite der Abbildung 5.5 sind die Kompaktformen der Level Graph Knoten bei denen lediglich der globale eindeutige Identifikator angegeben wird dargestellt. Die Bedeutung der Level Graph Knoten vom Komponententypfragment und Relationstypfragment werden in den Kapiteln 5.2.8 und 5.2.9 im Detail definiert, wobei die formale Definition für den Komponententyp (vgl. Def.4.3.1) und Relationstyp (vgl. Def. 4.5.1) für Topologien bereits erfolgte und analog auf Level Graphen angepasst werden kann. Daher wird an dieser Stelle verzichtet diese nochmals separat für das Level Graph Metamodell zu definieren.

5.2.4 Level Abbildung

Die Level Abbildung ordnet jedem Level Graph Knoten genau einem Abstraktionslevel in dem Level Graphen zu. Dabei können keine oder mehrere Level Graph Knoten einem Abstraktionslevel in dem Level Graphen zugeordnet werden. Die Level Abbildung $level_{\text{lg}_\text{j}}$ wird formal wie folgt definiert:

Definition 5.2.4 (Level Abbildung)

$$level_{lg_j} : LGK_{lg_j} \rightarrow AL_{lg_j}, lgk_i \mapsto al_i \tag{5.4}$$

5.2.5 Verfeinerungsrelation

Eine Verfeinerungsrelation ist eine Level Graph Relation in einem Level Graphen und wird durch eine Kante in diesem dargestellt. Durch die Verfeinerungsrelationen wird bestimmt in welche andere Level Graph Knoten oder Level Graph Fragmente, ein Level Graph Knoten verfeinert werden kann (vgl. Kap. 5.7). Zudem werden über die Verfeinerungsrelationen auch die Eintritts, Austritts und inneren Knoten von Level Graph Fragmenten identifiziert (vgl. Kap. 5.2.8 und Kap. 5.2.9). Formal werden die Verfeinerungsrelationen wie folgt definiert:

Definition 5.2.5 (Verfeinerungsrelation)
Sei $VR_{\text{lg}} = \{vr_1, vr_2, ..., vr_{\text{i}}, ..., vr_{\text{n}} | n \in \mathbb{N}\}$ die Menge aller Verfeinerungsrelation in einem Level Graphen lg_{j}, dann wird eine Verfeinerungsrelation $vr_{\text{i}} \in VR_{\text{lg}_{\text{j}}}$ durch folgenden Tupel definiert:

$$\begin{aligned} VR_{\text{lg}_{\text{j}}} &\subseteq \sum^{+} \times LGK_{\text{lg}_{\text{j}}} \times LGK_{\text{lg}_{\text{j}}} \times \mathbb{B} \times \mathbb{B} \\ vr_{\text{i}} &= (id, lgk_{\text{q}}, lgk_{\text{z}}, eintritt, austritt) \in VR_{\text{lg}_{\text{j}}} \end{aligned} \tag{5.5}$$

1.) $\pi_1(vr_{\text{i}}) = id$, wobei $id \in \sum^{+}$ ein eindeutiger globaler Identifikator von vr_{i} ist.

2.) $\pi_2(vr_{\text{i}}) = lgk_{\text{q}}$, wobei $lgk_{\text{q}} \in LGK_{\text{lg}_{\text{j}}}$ der Quell Level Graph Knoten für vr_{i} ist.

3.) $\pi_3(vr_{\text{i}}) = lgk_{\text{z}}$, wobei $lgk_{\text{z}} \in LGK_{\text{lg}_{\text{j}}}$ der Ziel Level Graph Knoten für vr_{i} ist.

4.) $\pi_4(vr_{\text{i}}) = eintritt$, wobei $eintritt \in \mathbb{B}$ ein Wahrheitswert ist und angibt, ob der Ziel Level Graph Knoten der Verfeinerungsrelation ein Eintrittsknoten von einem Fragment ist oder nicht.

5.) $\pi_5(vr_{\text{i}}) = austritt$, wobei $austritt \in \mathbb{B}$ ein Wahrheitswert ist und angibt, ob der Ziel Level Graph Knoten der Verfeinerungsrelation ein Austrittsknoten von einem Fragment ist oder nicht.

Eine Level Graph Relation in einem Level Graphen repräsentiert genau dann eine Verfeinerungsrelation, wenn die Tiefe des Abstraktionslevel von dem Quell Level Graph Knoten kleiner als die Tiefe des Abstraktionslevel von dem Ziel Level Graph Knoten ist $\pi_3(level_{\text{lg}_{\text{j}}}(lgk_{\text{q}})) < \pi_3(level_{\text{lg}_{\text{j}}}(lgk_{\text{z}}))$ und eine der nachfolgenden Bedingungen erfüllt ist:

(i) $(typ_{\text{lg}_{\text{j}}}(lgk_{\text{q}}) \in KT_{\text{lg}_{\text{j}}}) \wedge (typ_{\text{lg}_{\text{j}}}(lgk_{\text{z}}) \in KT_{\text{lg}_{\text{j}}})$, die Level Graph Relation verbindet einen Quell Level Graph Knoten vom Komponententyp mit einem Ziel Level Graph Knoten vom Komponententyp.

(ii) $(typ_{\text{lg}_{\text{j}}}(lgk_{\text{q}}) \in RT_{\text{lg}_{\text{j}}}) \wedge (typ_{\text{lg}_{\text{j}}}(lgk_{\text{z}}) \in RT_{\text{lg}_{\text{j}}})$, die Level Graph Relation verbindet einen Quell Level Graph Knoten vom Relationstyp mit einem Ziel Level Graph Knoten vom Relationstyp.

(iii) $(typ_{\text{lg}_{\text{j}}}(lgk_{\text{q}}) \in RT_{\text{lg}_{\text{j}}}) \wedge (typ_{\text{lg}_{\text{j}}}(lgk_{\text{z}}) \in RTF_{\text{lg}_{\text{j}}})$, die Level Graph Relation verbindet einen Quell Level Graph Knoten vom Relationstyp mit einem Ziel Level Graph Knoten vom Relationstypfragment.

(iv) $(typ_{\text{lg}_{\text{j}}}(lgk_{\text{q}}) \in KT_{\text{lg}_{\text{j}}}) \wedge (typ_{\text{lg}_{\text{j}}}(lgk_{\text{z}}) \in KTF_{\text{lg}_{\text{j}}})$, die Level Graph Relation verbindet einen Quell Level Graph Knoten vom Komponententyp mit einem Ziel Level Graph Knoten vom Komponententypfragment.

Oder wenn die Tiefe des Abstraktionslevel von dem Quell Level Graph Knoten gleich der Tiefe des Abstraktionslevel von dem Ziel Level Graph Knoten ist $\pi_3(level_{\text{lg}_{\text{j}}}(lgk_{\text{q}})) = \pi_3(level_{\text{lg}_{\text{j}}}(lgk_{\text{z}}))$ und eine der nachfolgenden Bedingungen erfüllt ist:

(i) $(typ_{\mathrm{lg_j}}(lgk_{\mathrm{q}}) \in RTF_{\mathrm{lg_j}}) \wedge ((typ_{\mathrm{lg_j}}(lgk_{\mathrm{z}}) \in RT_{\mathrm{lg_j}}) \vee (typ_{\mathrm{lg_j}}(lgk_{\mathrm{z}}) \in KT_{\mathrm{lg_j}}))$, die Level Graph Relation verbindet einen Quell Level Graph Knoten vom Relationstypfragment mit einem Ziel Level Graph Knoten vom Relationstyp oder vom Komponententyp.

(ii) $((typ_{\mathrm{lg_j}}(lgk_{\mathrm{q}}) \in KTF_{\mathrm{lg_j}}) \wedge ((typ_{\mathrm{lg_j}}(lgk_{\mathrm{z}}) \in RT_{\mathrm{lg_j}}) \vee (typ_{\mathrm{lg_j}}(lgk_{\mathrm{z}}) \in KT_{\mathrm{lg_j}}))$, die Level Graph Relation verbindet einen Quell Level Graph Knoten vom Komponententypfragment mit einem Ziel Level Graph Knoten vom Relationstyp oder vom Komponententyp.

5.2.6 Kompatibilitätsrelation

Eine Kompatibilitätsrelation ist eine Level Graph Relation in einem Level Graphen und wird ebenfalls wie die Verfeinerungsrelation durch eine Kante im Level Graphen dargestellt. Durch die Kompatibilitätsrelation wird bestimmt, welche Level Graph Knoten vom Komponententyp zu welchen Level Graph Knoten vom Relationstyp kompatibel sind. Zudem kann über die Kompatibilitätsrelationen auch die Kompatibilität zwischen Komponententypfragmente und Relationstypfragmente im Level Graphen identifiziert werden. Die unterschiedlichen Kompatibilitätskategorien werden im Kapitel 5.4 im Detail erläutert. Im Gegensatz zu den Verfeinerungsrelationen benötigen die Kompatibilitätsrelationen keine zusätzlichen Wahrheitswerte im Tupel zum Identifizieren von Eintritts- oder Austrittsknoten von Fragmenten im Level Graphen. Formal werden die Kompatibilitätsrelationen wie folgt definiert:

Definition 5.2.6 (Kompatibilitätsrelation)
Sei $KR_{\mathrm{lg}} = \{kr_1, kr_2, ..., kr_{\mathrm{i}}, ..., kr_{\mathrm{n}} | n \in \mathbb{N}\}$ die Menge aller Kompatibilitätsrelationen in einem Level Graphen lg_{j}, dann wird eine Kompatibilitätsrelation $kr_{\mathrm{i}} \in KR_{\mathrm{lg_j}}$ durch folgenden Tupel definiert:

1.) $\pi_1(kr_{\mathrm{i}}) = id$, wobei $id \in \sum^+$ ein eindeutiger globaler Identifikator von kr_{i} ist.

2.) $\pi_2(kr_{\mathrm{i}}) = lgk_{\mathrm{q}}$, wobei $lgk_{\mathrm{q}} \in LGK_{\mathrm{lg_j}}$ der Quell Level Graph Knoten für kr_{i} ist.

3.) $\pi_3(kr_{\mathrm{i}}) = lgk_{\mathrm{z}}$, wobei $lgk_{\mathrm{z}} \in LGK_{\mathrm{lg_j}}$ der Ziel Level Graph Knoten für kr_{i} ist.

Eine Level Graph Relation in einem Level Graphen repräsentiert genau dann eine Kompatibilitätsrelation, wenn die Tiefe des Abstraktionslevel von dem Quell Level Graph Knoten gleich der Tiefe des Abstraktionslevel von dem Ziel Level Graph Knoten ist $\pi_3(level_{\mathrm{lg_j}}(lgk_{\mathrm{q}})) = \pi_3(level_{\mathrm{lg_j}}(lgk_{\mathrm{z}}))$ und eine der nachfolgenden Bedingung erfüllt ist:

(i) $(typ_{\mathrm{lg_j}}(lgk_{\mathrm{q}}) \in KT_{\mathrm{lg_j}}) \wedge (typ_{\mathrm{lg_j}}(lgk_{\mathrm{z}}) \in RT_{\mathrm{lg_j}})$, die Level Graph Relation verbindet einen Quell Level Graph Knoten vom Komponententypen mit einem Ziel Level Graph Knoten vom Relationstyp

(ii) $(typ_{\mathrm{lg_j}}(lgk_{\mathrm{q}}) \in RT_{\mathrm{lg_j}}) \wedge (typ_{\mathrm{lg_j}}(lgk_{\mathrm{z}}) \in KT_{\mathrm{lg_j}})$, die Level Graph Relation verbindet einen Quell Level Graph Knoten vom Relationstyp mit einem Ziel Level Graph Knoten vom Komponententyp

5.2.7 Level Graph Relation

Eine Level Graph Relation repräsentiert genau eine gerichtete Kante in einem Level Graphen. Jede Relation in einem Level Graphen stellt entweder eine Kompatibilitätsrelation oder eine Verfeinerungsrelation in dem Level Graphen dar und hat genau einen Quell Level Graph Knoten und einen Ziel Level Graph Knoten (vgl. Abb. 5.3). Wobei die Tiefe des zugeordneten Abstraktionslevels des Ziel Level Graph Knoten immer größer oder gleich der Tiefe des zugeordneten Abstraktionslevel des Quell Level Graph Knoten ist. Formal werden die Level Graph Relationen als vereinigte Menge der Verfeinerungs- und Kompatibilitätsrelationen wie folgt definiert:

Definition 5.2.7 (Level Graph Relation)
Sei $LGR_{\mathrm{lg_j}} = \{lgr_1, lgr_2, ..., lgr_{\mathrm{i}}, ..., lgr_{\mathrm{n}} | n \in \mathbb{N}\} = (VR_{\mathrm{lg_j}} \cup KR_{\mathrm{lg_j}})$ die vereinigte Menge zweier disjunkter Mengen aller Verfeinerungsrelationen und Kompatibilitätsrelationen in einem Level Graphen lg_{j}, dann wird eine Level Graph Relation $lgr_{\mathrm{i}} \in LGR_{\mathrm{lg_j}}$ wie folgt definiert:

$$\begin{gathered} LGR_{\mathrm{lg_j}} = VR_{\mathrm{lg_j}} \cup KR_{\mathrm{lg_j}}, \text{wobei } VR_{\mathrm{lg_j}} \cap KR_{\mathrm{lg_j}} = \emptyset \\ lgr_{\mathrm{i}} \in LGR_{\mathrm{lg_j}}, \text{wobei } (lgr_{\mathrm{i}} \in KR_{\mathrm{lg_j}} \wedge lgr_{\mathrm{i}} \notin VR_{\mathrm{lg_j}}) \vee (lgr_{\mathrm{i}} \notin KR_{\mathrm{lg_j}} \wedge lgr_{\mathrm{i}} \in VR_{\mathrm{lg_j}}) \end{gathered} \tag{5.6}$$

Zudem gelten folgende zwei Bedingungen für Level Graph Relationen im Allgemeinen, die erfüllt sein müssen, damit eine Level Graph Relation eine valide Level Graph Relation darstellt:

(i) Für alle lgr_{i}, gilt das $\pi_3(level_{\mathrm{lg_j}}(lgk_{\mathrm{q}})) \leq \pi_3(level_{\mathrm{lg_j}}(lgk_{\mathrm{z}}))$ die Tiefe des Abstraktionslevels des Quell Level Graph Knoten kleiner gleich der Tiefe des Abstraktionslevels des Ziel Level Graph Knoten sein muss.

(ii) Für alle lgr_{i}, gilt $lgk_{\mathrm{q}} \neq lgk_{\mathrm{z}}$, das bedeutet es gibt keinen Level Graph Knoten mit einer Level Graph Relation, die eine Selbstreferenz darstellt.

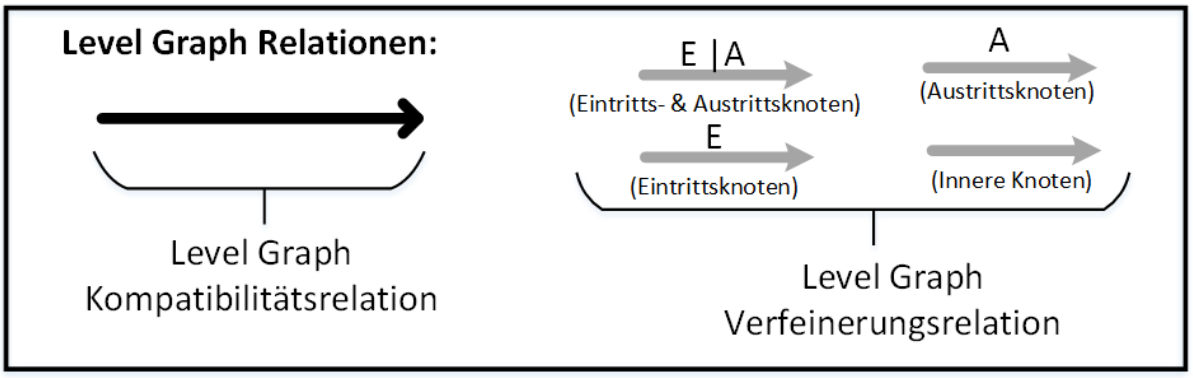

Abbildung 5.6: Notation Level Graph Relation

In der Abbildung 5.6 ist die graphische Notation, die für die unterschiedlichen Typen von Level Graph Relationen verwendet wird abgebildet. Dabei werden Kompatibilitätsrelationen durch schwarze gerichtete Pfeile dargestellt und Verfeinerungsrelationen durch graue gerichtete, gewichtete Pfeile dargestellt. Dabei wird durch ein E gekennzeichnet, dass der Ziel Level Graph Knoten der Verfeinerungsrelation einen Eintrittsknoten von einem Fragment darstellt und durch ein A wird gekennzeichnet, dass der Ziel Level Graph Knoten einen Austrittsknoten von einem Fragment darstellt.

Aus den oberen Definitionen (vgl. Def. 5.2.5, Def. 5.2.6 und Def. 5.2.7) und den jeweiligen Bedingungen der unterschiedlichen Arten von Level Graph Relationen, lassen sich die in Abbildung 5.7 gültigen Level Graph Relationen ableiten. Alle weiteren Level Graph Relationen, die nicht in der Abbildung 5.7 enthalten sind stellen aktuell keine gültigen Level Graph Relationen dar. Ausgenommen Verfeinerungsrelationen, die auch über mehrere Abstraktionslevel gehen (siehe Hinweis Abb. 5.7). Diese werden jedoch aktuell nicht von dem Verfeinerungsalgorithmus berücksichtig und sind daher für diese Arbeit nicht relevant.

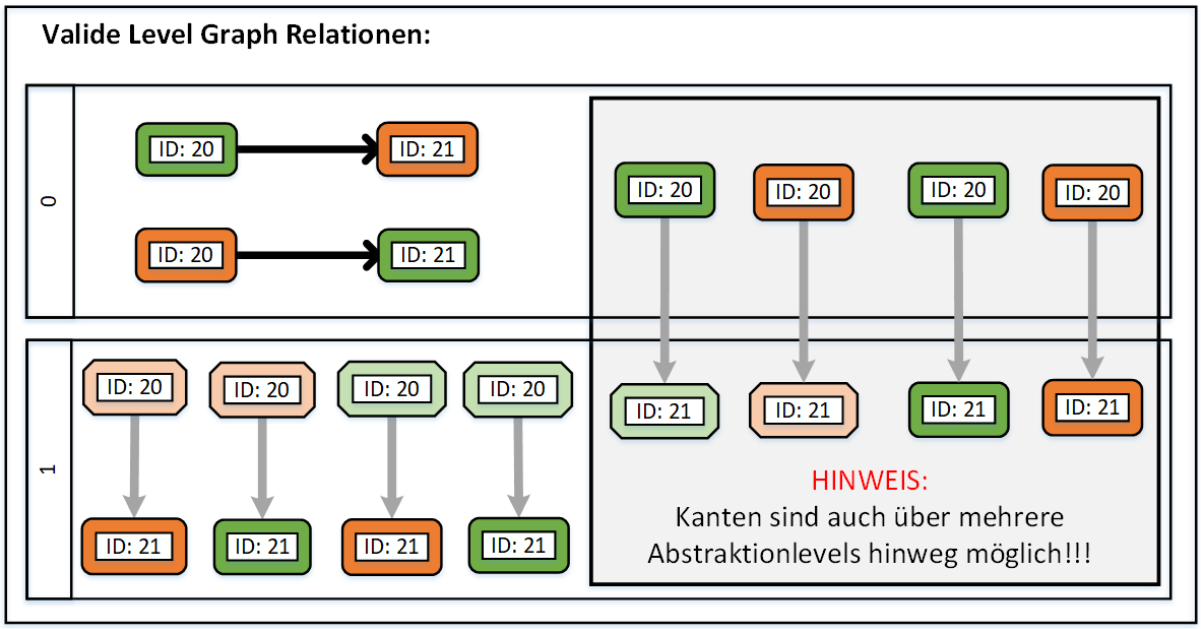

Abbildung 5.7: Valide Level Graph Relationen

5.2.8 Komponententypfragment

Ein Komponententypfragment beschreibt die Semantik von einem Knoten in einem Level Graphen. Jeder Level Graph Knoten, der vom Komponententypfragment ist, definiert einen Teilgraphen in einem Abstraktionslevel von dem Level Graphen durch seine ausgehenden Verfeinerungsrelationen. Die Komponententypfragmente werden für die Verfeinerung von geschlossenen Teiltopologiegraphen benötigt (vgl. Kap. 5.1 und Kap. 5.7). Die Komponententypfragmente definieren eine Menge von Eintritts-, Austritts- und inneren Knoten, durch die Gewichte der Verfeinerungsrelationen, wie zum Beispiel in der Abbildung 5.8 unten rechts dargestellt. Anhand den Eintritts- und Austrittsknoten wird die spätere Kompatibilität zwischen Fragmenten definiert (vgl. Kap. 5.4). Dabei können nur Level Graph Knoten vom Komponententyp (vgl. Kap. 4.3) als Eintritts- und Austrittsknoten für ein Komponententypfragment definiert werden. Die inneren Knoten des Komponententypfragments können vom Komponententyp sowie vom Relationstyp (vgl. Kap. 4.5) sein, jedoch nicht von einem Komponententypfragment oder von einem Relationstypfragment. Dies hat zur Folge, dass keine Verschachtelungen von Fragmenten und auch keine Verfeinerungen von einem Fragment in ein anderes Fragment zugelassen werden. Formal werden die Komponententypfragmente wie folgt definiert:

Definition 5.2.8 (Komponententypfragment)
Sei $KTF_{\mathrm{lg_j}} = \{ktf_1, ktf_2, ..., ktf_{\mathrm{i}}, ..., ktf_{\mathrm{n}} | n \in \mathbb{N}\} \subseteq KTF_{\mathrm{u}}$ die Menge aller Komponententypfragmente in einem Level Graphen lg_{j}, dann wird ein Komponententypfragment $ktf_{\mathrm{i}} \in KTF_{\mathrm{lg_j}}$ durch folgenden Tupel definiert:

$$\begin{aligned} KTF_{\mathrm{lg_j}} &\subseteq \sum^{+} \times \sum^{+} \\ ktf_{\mathrm{i}} &= (id, name) \in KTF_{\mathrm{lg_j}} \end{aligned} \tag{5.7}$$

1.) $\pi_1(ktf_{\mathrm{i}}) = id$, wobei $id \in \sum^{+}$ ein eindeutiger globaler Identifikator von ktf_{i} ist.

2.) $\pi_2(ktf_{\mathrm{i}}) = name$, wobei $name \in \sum^{+}$ eine Bezeichnung für ktf_{i} ist.

In der Abbildung 5.8 ist im oberen Teil die detaillierte graphische Notation von einem Komponententypfragment, welcher durch einen Level Graph Knoten Verteilte_Datenbank mit der ID 11 repräsentiert wird zu sehen. In diesem werden drei SQL_Datenbank Level Graph Knoten als Eintritts- und/oder Austrittsknoten durch ein E bzw. A oben im Knoten gekennzeichnet und am Rand des Fragments Knoten abgelegt. Diese drei SQL_Datenbank Knoten können über einen SQL_DB_Link Level Graph Knoten vom Datenbank Konnektor Relationstyp entsprechend der Kompatibilitätsrelationen miteinander verbunden werden (vgl. Kap. 5.5). Somit könnte bei einer Verfeinerung z.B. eine Datenbank Komponente in einer Topologie über einen Level Graph Knoten, der mit einer Verfeinerungsrelation mit diesem Level Graph Knoten Fragment verbunden, ist in eine verteilte Datenbank mit drei SQL_Datenbank Komponenten, die über die jeweiligen kompatiblen SQL_DB_Link Relationen miteinander verbunden sind verfeinert werden. Im unteren Teil der Abbildung 5.8 ist auf der linken Seite die Kompaktform

der detaillierten Darstellung und auf der rechten Seite die Kompaktform wie sie im Graphen dargestellt wird abgebildet.

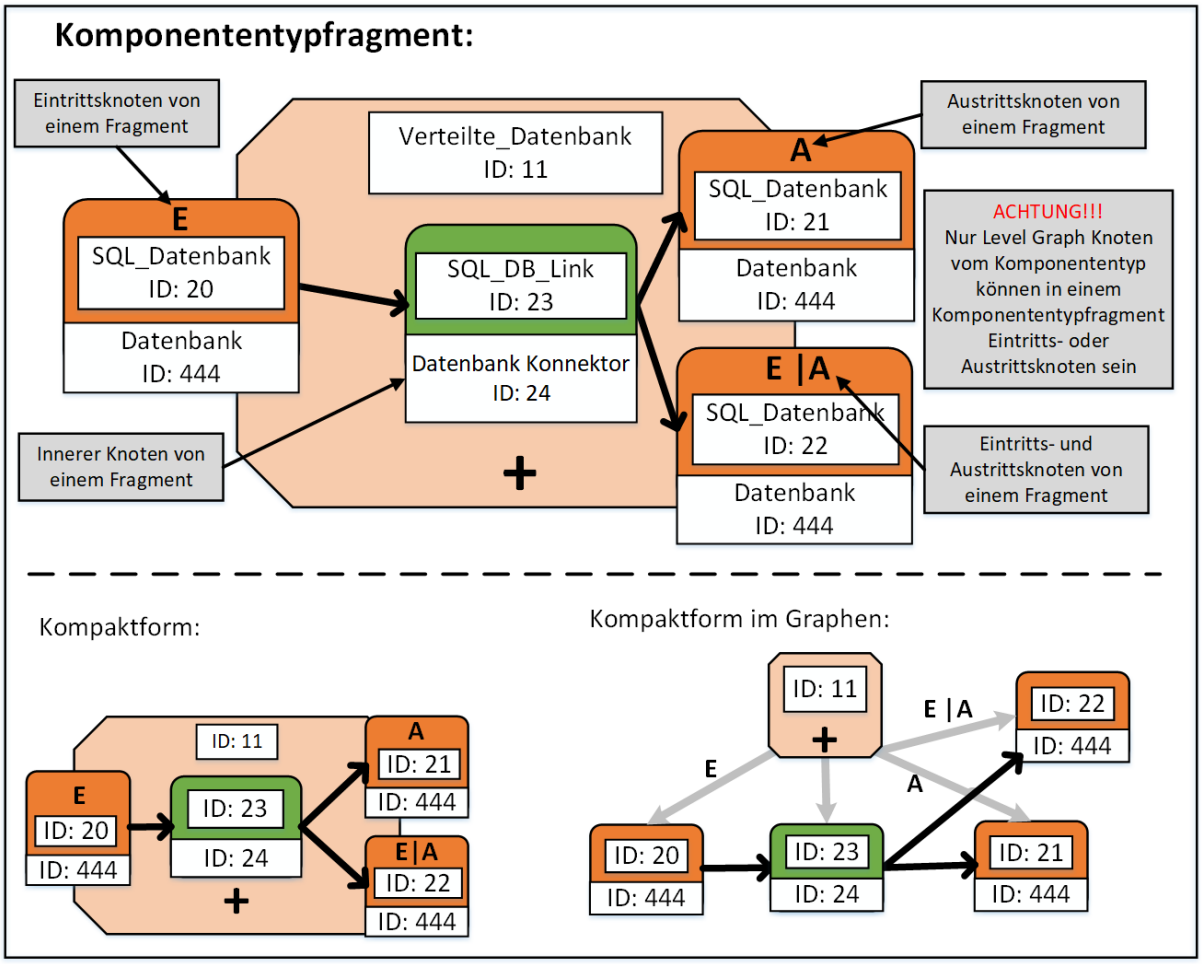

Abbildung 5.8: Notation Komponententypfragment

5.2.9 Relationstypfragment

Ein Relationstypfragment beschreibt die Semantik von einem Knoten in einem Level Graphen. Dabei gelten für Relationstypfragmente im Wesentlichen die gleichen Definitionen wie für Komponententypfragmente. Daher werden an dieser Stelle nur die Unterschiede zu einem Komponententypfragment aufgezeigt. Die Relationstypfragmente werden für die Verfeinerung von offenen Teiltopologiegraphen benötigt (vgl. Kap. 5.1 und 5.7). Dabei können nur Level Graph Knoten vom Relationstyp als Eintritts- und Austrittsknoten für ein Relationstypfragment definiert werden. Dadurch kann ein Komponententypfragment mit einem Relationstypfragment über Kompatibilitätsrelationen miteinander verbunden werden und die Kompatibilität zwischen Fragmenten definiert werden (vgl. Kap. 5.4). Für die inneren Knoten gilt dasselbe wie bei den

Komponententypfragmenten (vgl. 5.2.8). Formal werden die Relationstypfragmente wie folgt definiert:

Definition 5.2.9 (Relationstypfragment)
Sei $RTF_{lg_j} = \{rtf_1, rtf_2, ..., rtf_i, ..., rtf_n | n \in \mathbb{N}\}$ die Menge aller Relationstypfragmente in einem Level Graphen lg_j, dann wird ein Relationstypfragment $rtf_i \in RTF_{lg_j}$ durch folgenden Tupel definiert:

$$RTF_{lg_j} \subseteq \sum^+ \times \sum^+$$
$$rtf_i = (id, name) \in RTF_{lg_j} \tag{5.8}$$

1.) $\pi_1(rtf_i) = id$, wobei $id \in \sum^+$ ein eindeutiger globaler Identifikator von rtf_i ist.

2.) $\pi_2(rtf_i) = name$, wobei $name \in \sum^+$ eine Bezeichnung für rtf_i ist.

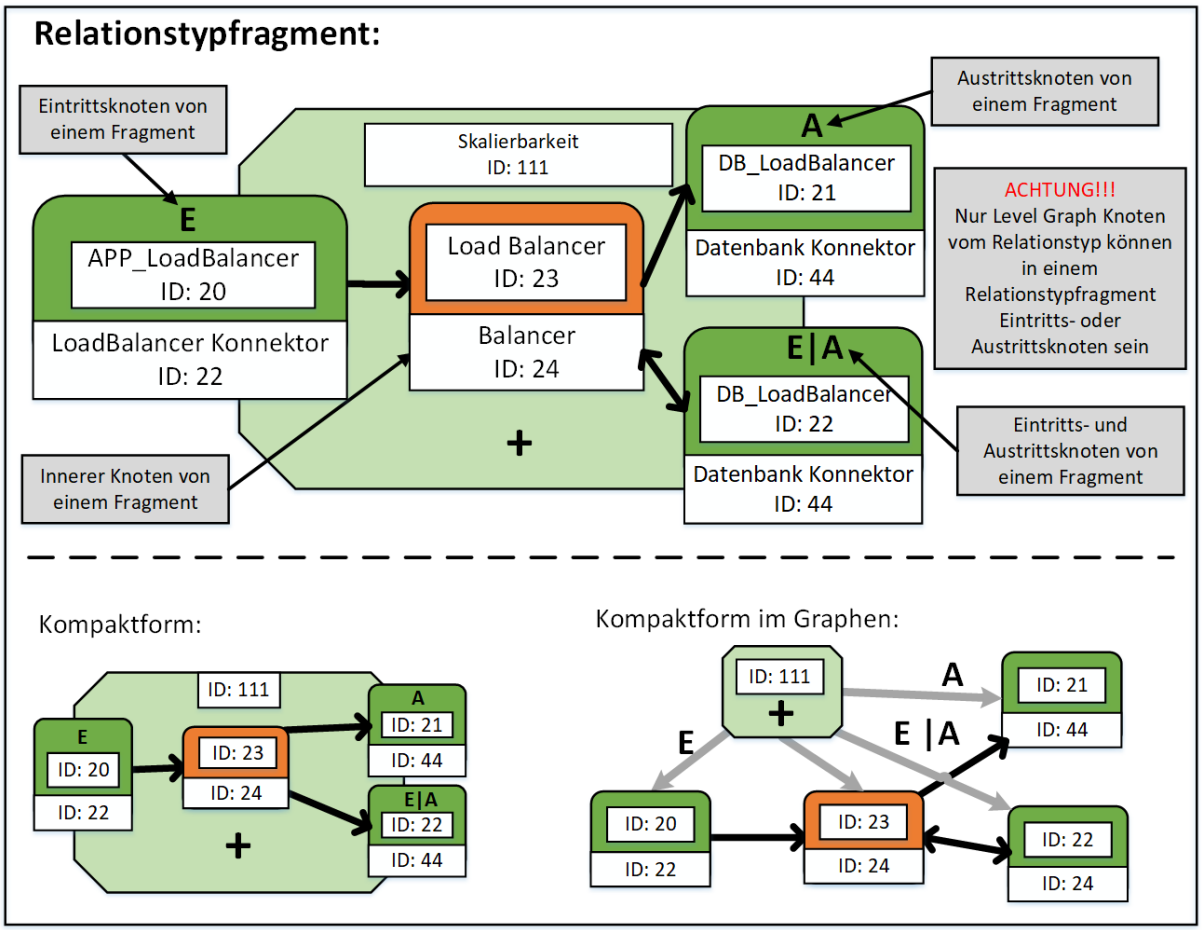

Abbildung 5.9: Notation Relationstypfragment

In der Abbildung 5.8 ist im oberen Teil die detaillierte graphische Notation von einem Relationstypfragment, welcher durch einen Level Graph Knoten Skalierbarkeit mit der ID 111 definiert wird zu sehen. Dieses Fragment stellt einen inneren Load_Balancer Level Graph Knoten mit der ID 23 vom Komponententyp Balancer mit der ID 24 dar. Dieser ist mit den jeweiligen Eintritts- und/oder Austrittsknoten über Kompatibilitätsrelationen verbunden. Bei den Eintritts- oder Austrittsknoten handelt es sich um einen App_LoadBalancer Level Graph Knoten mit ID 20 vom Relationstyp LoadBalancer Konnektor mit ID 22 und um zwei DB_LoadBalancer Level Graph Knoten mit ID 21 und ID 22 vom Relationstyp Datenbank Konnektor. Dieses Fragment könnte bei einer Verfeinerung z.B. genutzt werden um eine APP_DB Relation, die eine APP Komponente mit einer DB Komponente in einem Topologiemodell verbindet, in eine zwischen gelagerte Load Balancer Komponente zu verfeinern. Im unteren Teil der Abbildung 5.8 ist auf der linken Seite die Kompaktform der detaillierten Darstellung und auf der rechten Seite die Kompaktform von Relationstypfragmente wie sie im Graphen dargestellt werden abgebildet.

5.3 Level Graph Modell Beispiel

Anhand der oberen Definitionen von den einzelnen Elementen eines Level Graphen lassen sich die Strukturen des Level Graphen ableiten. Im Folgenden sollen, die Besonderheiten dieser Strukturen aufgezeigt und erläutert werden.

In der Abbildung 5.10 ist ein Level Graph Modell in der Kompaktform mit drei unterschiedlichen Abstraktionslevel dargestellt, dieser dient als Referenz Graph für die in den nachfolgenden Kapiteln vorgestellten Teilgraphen von einem Level Graphen (vgl. Kap. 5.4 und Kap. 5.7). Durch diesen Level Graph könnte zum Beispiel eine Komponente (K1) in einer abstrakten Topologie, die vom Komponententyp (KT1) ist, durch den linken oberen Level Graph Knoten (LGK1) der ebenfalls vom Komponententyp (KT1) ist, in eine neue Komponente (K2) verfeinert werden, die vom Komponententyp (KT3, KT4 oder KT5) ist. Des Weiteren wird durch das Abstraktionslevel der Tiefe 0 des Level Graphen, auch die Verfeinerung einer komplexeren abstrakten Topologie, die aus mehreren Elementen besteht ermöglicht. Bei dieser komplexeren abstrakten Topologie könnte es sich zum Beispiel um eine Quell Komponente (K1) vom Komponententyp (KT1), die mit einer weiteren Zielkomponente (K2) vom Komponententyp (KT2) über eine Relation (R1) vom Relationstyp (RT1) verbunden ist, handeln. In einer Verfeinerung dieser Topologie könnte zum Beispiel zuerst die Komponente (K1) über den Level Graph Knoten (LGK1) in eine Komponente von dem Komponententyp (KT4) oder (KT5) verfeinert werden. Allerdings könnte die Komponente (K1) vom Komponententyp (KT2) nicht in eine Komponente von dem Komponententyp (KT3) verfeinert werden, da diese Verfeinerung nicht kompatible zu dem verfeinerten Fragment der Relation (R1) vom Relationstyp (RT1) wäre. Die Verfeinerung der Relation (R1) vom Relationstyp (RT1) in ein Fragment, wird dabei über den Level Graph Knoten (LGK2) im Abstraktionslevel der Tiefe 0 und den Level Graph Knoten (LGK6) im Abstraktionslevel der Tiefe 1 durchgeführt. Als letztes würde noch die Komponente (K2) vom Komponententyp (KT2) über den Level Graph Knoten (LGK3) in ein Komponententypfragment,

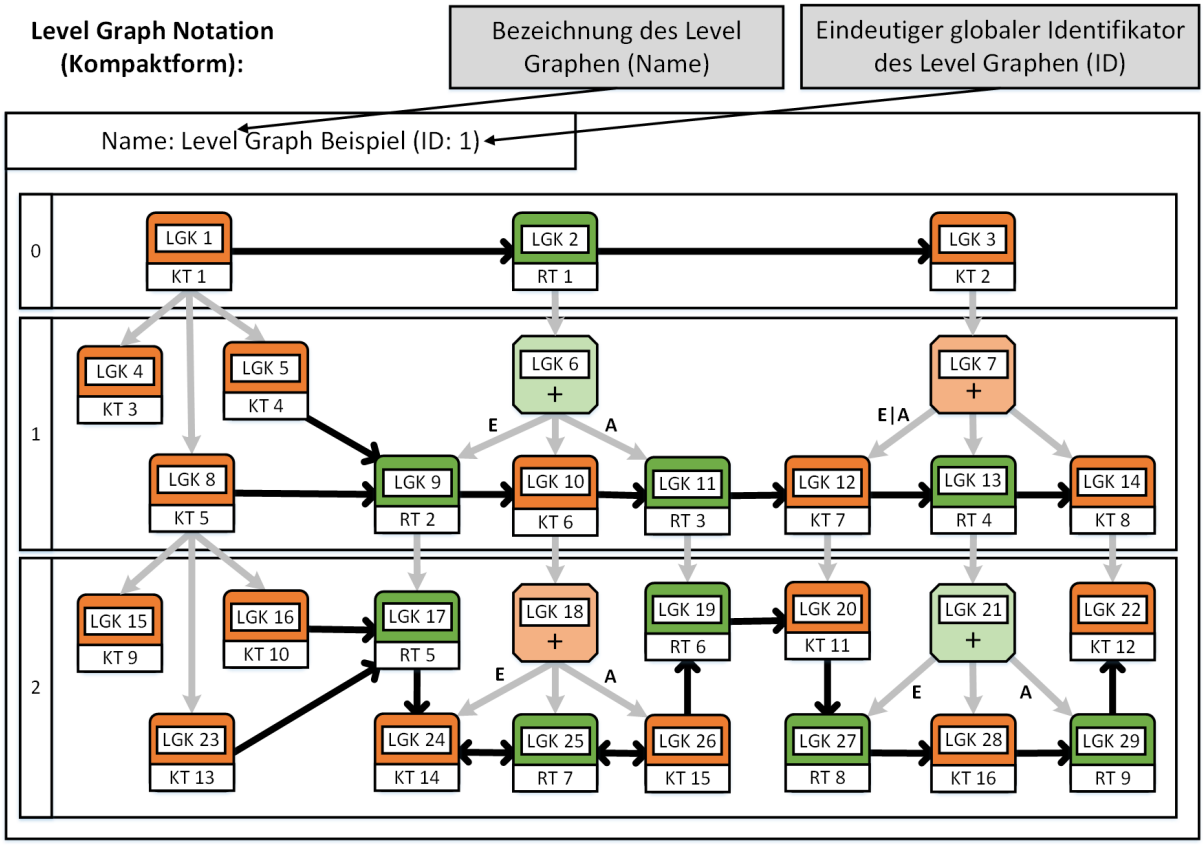

Abbildung 5.10: Beispiel und Notation von einem Level Graph Modell

welches durch den Level Graph Knoten (LGK7) repräsentiert wird verfeinert. Dabei müssen die einzelnen Verfeinerungen der Bestandteile einer Topologie jeweils auf Kompatibilität zu den vorher durchgeführten Verfeinerungen überprüft werden. Dabei wird in dieser Arbeit ein „Alles oder Nichts“ Prinzip bei der Verfeinerung zur Vereinfachung angenommen. Das hat zur Folge, dass alle Komponenten und Relationen in einem Topologiemodell von einem Abstraktionslevel in ein tieferes Abstraktionslevel verfeinert werden oder keine Verfeinerung des Topologiemodell durchgeführt wird. Die einzelnen Verfeinerungs-, Kompatibilitäts- und Ableitungsmöglichkeiten werden in den folgenden Kapiteln an abstrakten Beispielen im Detail erläutert und genauer definiert.

5.4 Kompatibilitäts-Level Graph

Durch den Kompatibilitäts-Level Graphen wird definiert welche Level Graph Knoten und Level Graph Fragmente zueinander kompatibel sind. Dadurch soll bei einer Verfeinerung gewährleistet werden, dass nicht willkürliche spezifische Topologiemodelle entstehen, sondern nur Topologiemodelle erstellt werden, die auch letztendlich ausgeführt werden können. Der Kompatibilitäts-Level Graph von einem Level Graphen ergibt sich durch Löschen aller Verfeinerungsrelationen, dessen Quell Level Graph Knoten und Ziel Level Graph Knoten in unterschiedlichen Abstraktionslevel liegen und wird wie folgt definiert:

Definition 5.4.1 (Kompatibilitäts-Level Graph)
Der Kompatibilitäts-Level Graph ist ein möglicherweise nicht zusammenhängender, gerichteter, k-partiter Graph der alle Mengen des Level Graphen beinhaltet mit Ausnahme der Teilmenge der Verfeinerungsrelationen $VRT = \{vr_1, vr_2, ..., vr_\mathrm{i}, ...vr_\mathrm{n} | \pi_3(level_{\mathrm{lg_j}}(\pi_2(vr_\mathrm{i}))) \neq \pi_3(level_{\mathrm{lg_j}}(\pi_3(vr_\mathrm{i})))\} \subseteq VR_{\mathrm{lg_j}}$, deren Quell- und Ziel Level Graph Knoten in unterschiedlichen Abstraktionslevel liegen. Die k-Partitionen des Kompatibilitäts-Level Graphen werden durch die Komponententyp, Relationstyp, Komponententypfragment und Relationstypfragment Menge des Level Graph Metamodells identifiziert (vgl. Kap. 5.2.3 und Def. 5.2.1). Damit gilt für die Anzahl der k-Partitionen im Kompatibilitäts-Level Graphen k=4, da aktuell im Level Graph Metamodell vier unterschiedliche Typen von Level Graph Knoten definiert sind (vgl. Def. 5.2.1 und Def. 5.2.3). Daraus folgt, dass in jeder Partition sich ausschließlich Level Graph Knoten von einem Typen befinden. Der Kompatibilitäs-Level Graph $klg_{\mathrm{lg_j}}$ von einem Level Graphen lg_j wird formal wie folgt definiert:

$$\begin{aligned} klg_{\mathrm{lg_j}} = (&AL_{\mathrm{lg_j}}, LGK_{\mathrm{lg_j}}, LGR_{\mathrm{lg_j}} \backslash VRT, KT_{\mathrm{lg_j}}, RT_{\mathrm{lg_j}}, \\ &KTF_{\mathrm{lg_j}}, RTF_{\mathrm{lg_j}}, BE_{\mathrm{lg_j}}, level_{\mathrm{lg_j}}, typ_{\mathrm{lg_j}}, bereitgestellt_{\mathrm{lg_j}}) \end{aligned} \tag{5.9}$$

In der Abbildung 5.11 ist der Kompatibilitäts-Level Graph, des zuvor definierten Level Graphen (vgl. Kap. 5.3 i.V.m. Abb. 5.10) als ein Beispiel abgebildet. Dabei kann der Kompatibilitäts-Level Graph anhand den Abstraktionslevels vom Level Graphen in Kompatibilitäts-Abstraktionslevel Graphen und anhand der Level Graph Fragment Knoten (vgl. Kap. 5.2.8 und Kap. 5.2.9) in Kompatibilitäts-Fragment Graphen unterteilt werden. Dabei werden die einzelnen Kompatibilitäts-Abstraktionslevel Graphen durch die Swimlanes der Abstraktionslevels voneinander abgegrenzt. Wohingegen die Kompatibilitäts-Fragment Graphen von einem Level Graph Knoten vom Komponententypfragment und Relationstypfragment durch dessen ausgehende Verfeinerungsrelationen identifiziert werden. Als ein Beispiel für einen Kompatibilitäts-Fragment Graphen ist in der Abbildung 5.11, der Kompatibilitäts-Fragment Graph von dem LGK 18 durch einen gestricheltes graues Rechteck hervorgehoben.

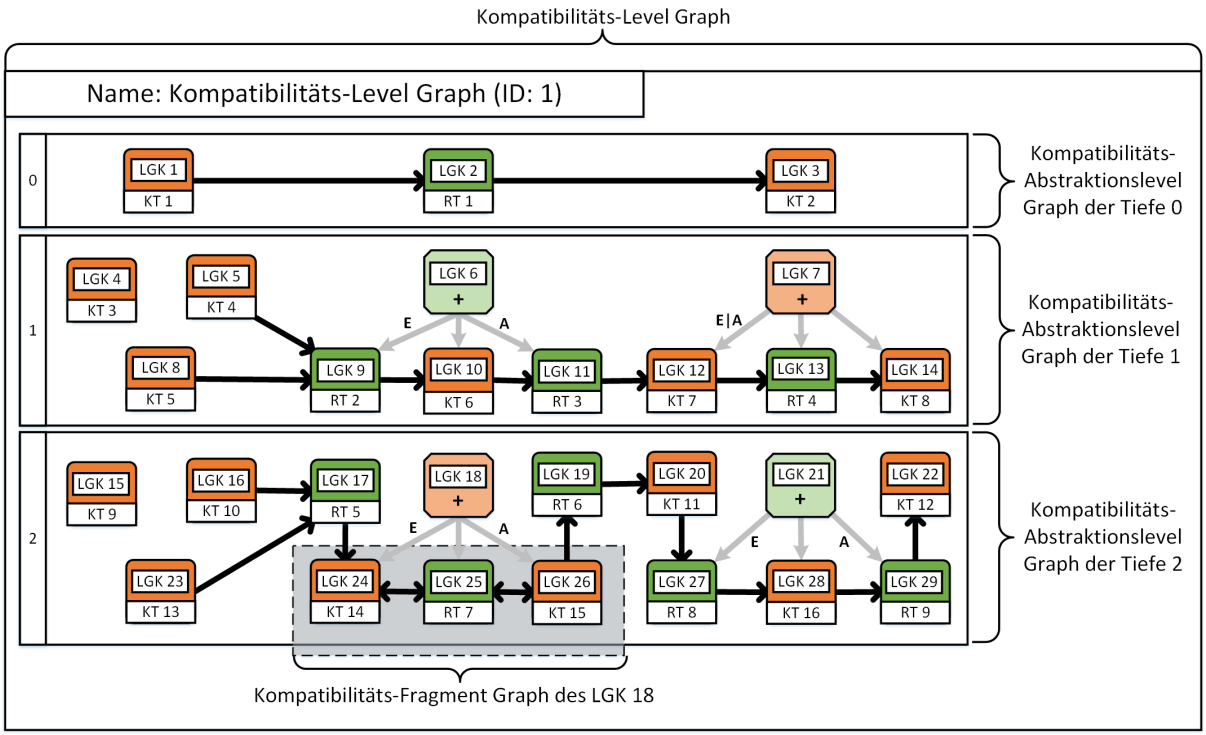

Abbildung 5.11: Kompatibilitäts-Level Graph

Der Kompatibilitäts-Level Graph sagt aus, dass ein Quell Level Graph Knoten $lgk_{\mathrm{q}} \in klg_{\mathrm{lg_j}}$ genau dann kompatibel zu einen anderen Ziel Level Graph Knoten $lgk_{\mathrm{z}} \in klg_{\mathrm{lg_j}}$ ist, wenn es eine Kompatibilitätsrelation $kr_{\mathrm{i}} \in KR_{\mathrm{klg_{lg_j}}}$ vom Quell Level Graph Knoten zum Ziel Level Graph Knoten gibt. Anhand von den Kompatibilitätsrelationen können unterschiedliche Kompatibilitätsbeziehung zwischen zwei Level Graph Knoten identifiziert werden. Dabei wird die Kompatibilitätsbeziehung zwischen zwei Level Graph Knoten in dieser Arbeit formal kurz geschrieben als $lgk_{\mathrm{q}} \rightarrow lgk_{\mathrm{z}}$ was bedeutet, dass der Level Graph Knoten lgk_{q} kompatibel zu dem Level Graph Knoten lgk_{z} ist.

Dabei können die Kompatibilitätsbeziehungen, des Kompatibilitäts-Level Graphen in vier wesentliche Kategorien eingeteilt werden. Dies wären eine 1-zu-1, eine 1-zu-N, eine N-zu-1 und eine N-zu-N Kompatibilität. Diese werden in den nachfolgenden Abschnitten im Detail erläutert und mit einem Beispiel verdeutlicht.

1-zu-1 Kompatibilität

Die 1-zu-1 Kompatibilität ist die einfachste Form der Kompatibilität und wird durch eine Kompatibilitätsrelation in dem Kompatibilitäts-Level Graphen festgelegt. Diese Kompatibilität drückt aus ob ein Level Graph Knoten vom Komponententyp zu einem Level Graph Knoten vom Relationstyp und umgekehrt kompatibel ist oder nicht. Die Kompatibilitätsrelation in

der Abbildung 5.12 auf der linken Seite zwischen dem Quell Level Graph Knoten LGK1 vom Relationstyp RT1 und dem Ziel Level Graph Knoten LGK2 vom Komponententyp KT1 hat die Bedeutung, dass eine beliebige Komponente vom Komponententyp KT1 eine kompatible Zielkomponente für eine beliebige Relation vom Relationstyp RT1 ist. Wohingegen die Kompatibilitätsrelation auf der rechten Seite in der Abbildung 5.12 zwischen den zwei Level Graph Knoten ausdrückt, dass eine beliebige Komponente vom Komponententyp KT2 eine kompatible Quellkomponente von einer beliebigen Relation vom Relationstyp RT2 ist.

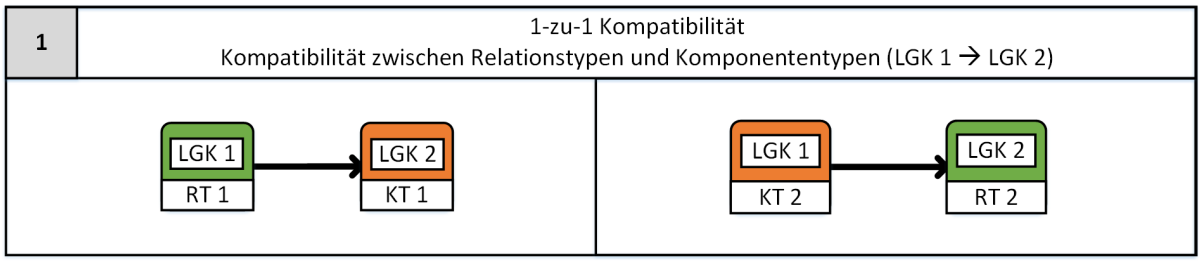

Abbildung 5.12: 1-zu-1 Kompatibilität

1-zu-N und N-zu-1 Kompatibilität

Die 1-zu-N oder N-zu-1 Kompatibilität drückt aus ob ein Level Graph Knoten vom Komponententyp, kompatibel zu einem Level Graph Knoten vom Relationstypfragment ist und umgekehrt (vgl. Abb. 5.13 (2.2)). Oder ob ein Level Graph Knoten vom Relationstyp kompatibel zu einem Level Graph Knoten vom Komponententypfragment ist und umgekehrt (vgl. Abb. 5.13 (2.1)).

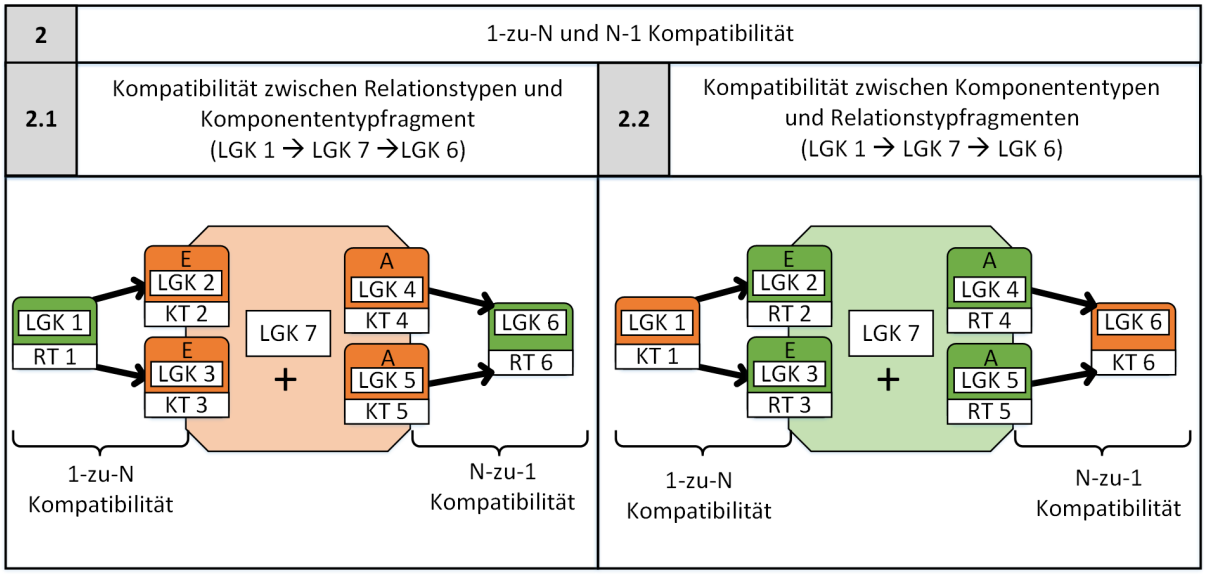

Abbildung 5.13: 1-zu-N und N-zu-1 Kompatibilität

Damit die 1-zu-N Kompatibilität gilt muss der Relationstyp oder Komponententyp Knoten zu allen Eintrittsknoten des Fragment Knoten eine 1-zu-1 Kompatibilität aufweisen und damit die N-zu-1 Kompatibilität gilt müssen alle Austrittsknoten des Fragment Knoten eine 1-zu-1 Kompatibilität zu den Level Graph Knoten vom Relationstyp oder vom Komponententyp aufweisen (vgl. Abb. 5.13).

N-zu-N Kompatibilität

Die N-zu-N Kompatibilität drückt aus ob ein Level Graph Knoten vom Komponententypfragment kompatibel zu einem Level Graph Knoten vom Relationstypfragment ist oder umgekehrt. Damit die N-zu-N Kompatibilität zwischen zwei Fragment Knoten LGK1 und LGK2 gilt müssen alle Austrittsknoten von LGK1 mit mindestens einer Kompatibilitätsrelation mit einem der Eintrittsknoten von LGK2 verbunden sein und alle Eintrittsknoten von LGK2 müssen mit mindestens einem Austrittsknoten von LGK1 über eine Kompatibilitätsrelation miteinander verbunden sein (vgl. Abb. 5.14). Wenn in der Abbildung 5.14 auf der linken Seite zum Beispiel die Kompatibilitätsrelation zwischen LGK3 und LGK7 nicht existieren, würde wäre der Level Graph Knoten LGK1 vom Komponententypfragment nicht kompatibel zu dem Level Graph Knoten LGK2 vom Relationstypfragment. Da sowohl ein Austrittsknoten von LGK1 keine Kompatibilitätsrelation zum nachfolgenden Fragment LGK2 aufweist, wie auch keine Kompatibilitätsrelation zu einem Eintrittsknoten vom LGK2 vom vorherigen Fragment LGK1 vorhanden ist.

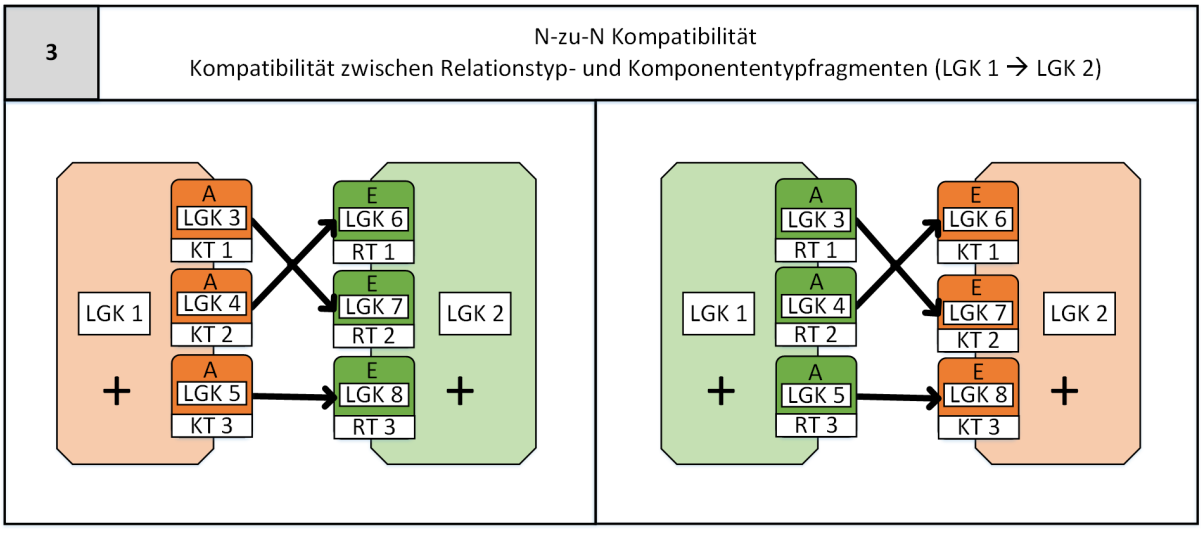

Abbildung 5.14: N-zu-N Kompatibilität

5.5 Ableitungen von Topologien

Auf der Grundlage des Topologie Metamodell aus Kapitel 4 und dem oben definierten Level Graph Metamodell (vgl. Kap. 5.2), die auf einer gemeinsamen Menge an Basiselementen bestehen, können im Folgenden die Ableitungsregeln zwischen Level Graphen und Topologien definiert werden. Durch die Ableitungen wird festgelegt welche Level Graph Strukturen, welchen Topologie Strukturen in einer Topologie entsprechen und umgekehrt. Dabei spiegelt die Struktur eines gesamten Level Graphen möglicherweise nicht nur eine Topologie Struktur wieder, sondern eine Vielzahl an unterschiedlichen Topologie Strukturen. Jedoch sind die Basisstrukturen, die in einer Topologie und in einem Level Graphen vorkommen immer die gleichen. Zu den zwei Basisableitungen gehören die Komponentenableitung und Relationsableitung, diese werden wie nachfolgend nach eigener Definition definiert:

1. **Komponentenableitung (vgl. Abb. 5.15 (1)):**
   Ein Level Graph Knoten vom Typ Komponententyp $(lgk_{\mathrm{i}} \in LGK_{\mathrm{lg_j}}) \wedge (typ_{\mathrm{lg_j}}(lgk_{\mathrm{i}}) \in KT_{\mathrm{lg_j}})$ kann in einer Topologie in eine Komponente $k_{\mathrm{i}} \in K_{\mathrm{t_j}}$ von demselben Komponententyp abgeleitet werden, dies wird kurz als $lgk_{\mathrm{i}} \Leftrightarrow k_{\mathrm{i}}$ geschrieben. Wobei folgendes für die Komponentenableitung gilt:

$$\begin{aligned} &lgk_{\mathrm{i}} \Leftrightarrow k_{\mathrm{i}}, \\ &\text{gilt genau dann wenn} \\ &typ_{\mathrm{lg_j}}(lgk_{\mathrm{i}}) = typ_{\mathrm{t_j}}(k_{\mathrm{i}}) \end{aligned} \tag{5.10}$$

2. **Relationsableitung (vgl. Abb. 5.15 (2)):**
   Ein Level Graph Knoten vom Typ Relationstyp $(lgk_{\mathrm{i}} \in LGK_{\mathrm{lg_j}}) \wedge (typ_{\mathrm{lg_j}}(lgk_{\mathrm{i}}) \in RT_{\mathrm{lg_j}})$ kann in einer Topologie in eine Relation $r_{\mathrm{i}} \in R_{\mathrm{t_j}}$ von demselben Relationstyp abgeleitet werden, dies wird kurz als $lgk_{\mathrm{i}} \Leftrightarrow r_{\mathrm{i}}$ geschrieben. Wobei folgendes für die Relationsableitung gilt:

$$\begin{aligned} &lgk_{\mathrm{i}} \Leftrightarrow r_{\mathrm{i}}, \\ &\text{gilt genau dann wenn} \\ &typ_{\mathrm{lg_j}}(lgk_{\mathrm{i}}) = typ_{\mathrm{t_j}}(r_{\mathrm{i}}) \end{aligned} \tag{5.11}$$

In der Abbildung 5.15 ist auf der linken Seite die Komponentenableitung von einem Level Graph Knoten LGK1 der vom Komponententyp KT1 ist dargestellt. Dabei wird der Level Graph Knoten LGK1 in eine Komponente K1 mit dem gleichen Komponententyp KT1 abgeleitet. Auf der rechten Seite dagegen ist die Relationsableitung abgebildet bei der ein Level Graph Knoten LGK2 vom Relationstyp RT1 in eine Relation vom Relationstyp RT1 in einer Topologie abgeleitet wird.

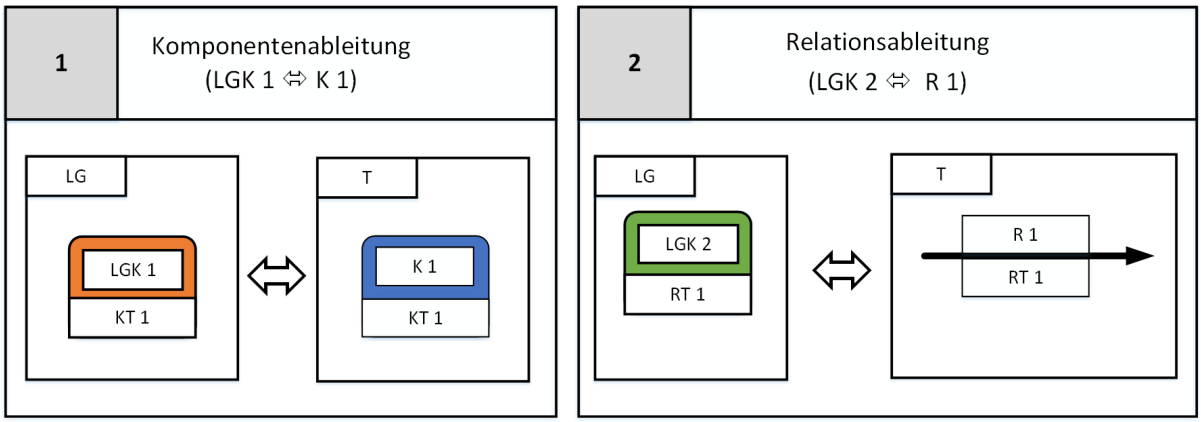

Abbildung 5.15: Komponentenableitung und Relationsableitung

Auf Grundlage der Basisableitungen lassen sich vier Strukturableitungen bilden, die von zentraler Bedeutung für die Ableitung von zusammenhängenden Strukturen im Level Graphen sind. Diese vier Strukturableitungen werden dabei wie folgt definiert:

1. **Folgeableitung:**
   Eine Folge ist eine abwechselnde Folge von drei Level Graph Knoten von Komponententypen und Relationstypen, die durch Kompatibilitätsrelationen miteinander im Level Graphen verbunden sind. Dabei beginnt und endet jede Folge mit einem Level Graph Knoten vom gleichen Knoten Typen. Formal wird eine Folge im Level Graphen wie folgt definiert:

$$(lgk_{\mathrm{q}} \rightarrow lgk_{\mathrm{m}} \rightarrow lgk_{\mathrm{z}}),$$
$$\text{wobei gilt}$$
$$((typ_{\mathrm{lg_j}}(lgk_{\mathrm{q}}) \in KT_{\mathrm{lg_j}}) \wedge (typ_{\mathrm{lg_j}}(lgk_{\mathrm{m}}) \in RT_{\mathrm{lg_j}}) \wedge (typ_{\mathrm{lg_j}}(lgk_{\mathrm{z}}) \in KT_{\mathrm{lg_j}})) \quad (5.12)$$
$$\vee$$
$$((typ_{\mathrm{lg_j}}(lgk_{\mathrm{q}}) \in RT_{\mathrm{lg_j}}) \wedge (typ_{\mathrm{lg_j}}(lgk_{\mathrm{m}}) \in KT_{\mathrm{lg_j}}) \wedge (typ_{\mathrm{lg_j}}(lgk_{\mathrm{z}}) \in RT_{\mathrm{lg_j}}))$$

   Eine abwechselnde Folge im Level Graphen kann in eine Abfolge von Komponenten und Relationen in einer Topologie abgeleitet werden, wobei die abgeleitete Folge in eine offene und geschlossene Folge unterschieden werden kann. Dabei könnte eine Folge in einem Level Graphen durch die Verwendung von Kontrollstrukturen, wie z.B. eine Schleife, mehrfach nacheinander in unendlich viele Folgen in einer Topologie abgeleitet werden. Allerdings wird in dieser Arbeit zur Vereinfachung angenommen, dass eine Folge in einem Level Graphen nur einmal in eine Folge in einer Topologie abgeleitet wird. Durch diese Vereinfachung soll gewährleistet werden, dass keine Ableitungen von unendlichen Topologien ermöglicht werden. Die Folgeableitungen werden formal wie folgt definiert:

(i) **Geschlossene Folgeableitung (vgl. Abb. 5.16 (1)):**
Eine geschlossene Folge liegt vor wenn der Quell Level Graph Knoten und der Ziel Level Graph Knoten einer Folge vom Komponententyp sind.

$$(lgk_{\mathrm{q}} \rightarrow lgk_{\mathrm{m}} \rightarrow lgk_{\mathrm{z}}) \Leftrightarrow (k_{\mathrm{q}} \rightarrow r_{\mathrm{m}} \rightarrow k_{\mathrm{z}}),$$
$$\text{gilt genau dann wenn}$$
$$((typ_{\mathrm{lg_j}}(lgk_{\mathrm{q}}) \in KT_{\mathrm{lg_j}}) \wedge (typ_{\mathrm{lg_j}}(lgk_{\mathrm{m}}) \in RT_{\mathrm{lg_j}}) \wedge (typ_{\mathrm{lg_j}}(lgk_{\mathrm{z}}) \in KT_{\mathrm{lg_j}})) \quad (5.13)$$
$$\wedge$$
$$((lgk_{\mathrm{q}} \Leftrightarrow k_{\mathrm{q}}) \wedge (lgk_{\mathrm{m}} \Leftrightarrow r_{\mathrm{m}}) \wedge (lgk_{\mathrm{z}} \Leftrightarrow k_{\mathrm{z}}))$$

(ii) **Offene Folgeableitung (vgl. Abb. 5.16 (2)):**
Eine offene Folge liegt vor wenn der Quell Level Graph Knoten und der Ziel Level Graph Knoten einer Folge vom Relationstyp sind.

$$(lgk_{\mathrm{q}} \rightarrow lgk_{\mathrm{m}} \rightarrow lgk_{\mathrm{z}}) \Leftrightarrow (r_{\mathrm{q}} \rightarrow k_{\mathrm{m}} \rightarrow r_{\mathrm{z}}),$$
$$\text{gilt genau dann wenn}$$
$$((typ_{\mathrm{lg_j}}(lgk_{\mathrm{q}}) \in RT_{\mathrm{lg_j}}) \wedge (typ_{\mathrm{lg_j}}(lgk_{\mathrm{m}}) \in KT_{\mathrm{lg_j}}) \wedge (typ_{\mathrm{lg_j}}(lgk_{\mathrm{z}}) \in RT_{\mathrm{lg_j}})) \quad (5.14)$$
$$\wedge$$
$$((lgk_{\mathrm{q}} \Leftrightarrow r_{\mathrm{q}}) \wedge (lgk_{\mathrm{m}} \Leftrightarrow k_{\mathrm{m}}) \wedge (lgk_{\mathrm{z}} \Leftrightarrow r_{\mathrm{z}}))$$

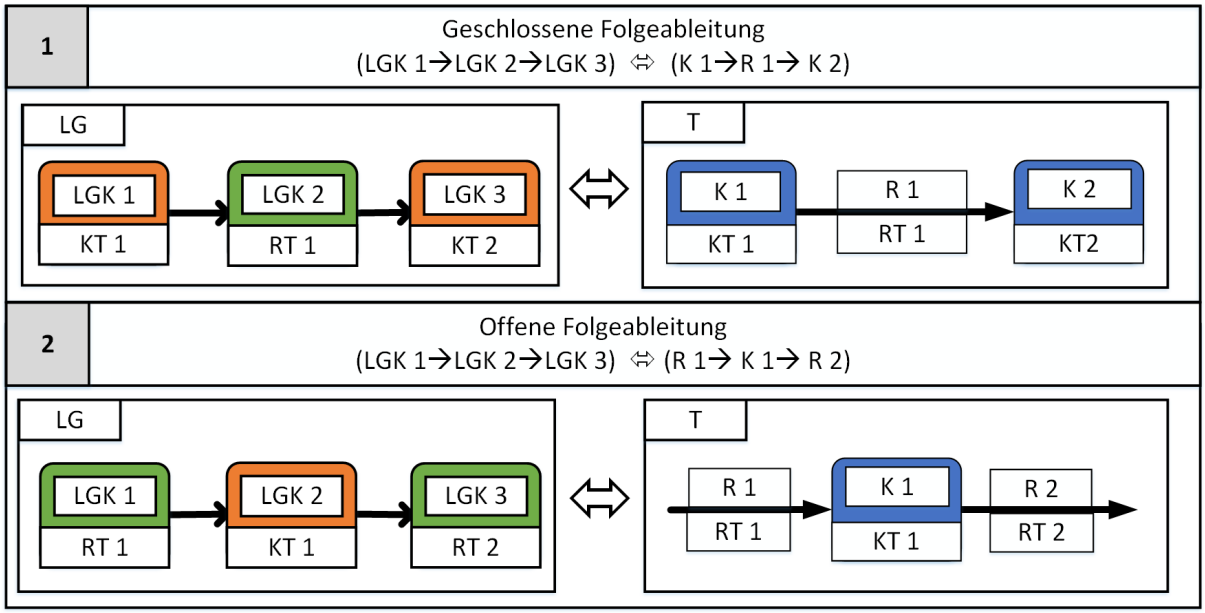

Abbildung 5.16: Geschlossene und offene Folgeableitung

Im oberen Teil der Abbildung 5.16 (1) ist ein Beispiel von einer Folge in einem Level Graphen abgebildet, die in eine geschlossene Folge in einer Topologie abgeleitet werden kann. Dabei wird umgekehrt die geschlossene Folge in der Topologie durch, die oben linke dargestellte Folge im Level Graphen repräsentiert. In einer geschlossenen Folge in einer Topologie existieren keine Relationen ohne Ziel oder Quell Komponente. In dem unteren Teil der Abbildung 5.16 (2) hingegen ist ein abstraktes Beispiel von einer Folge in einem Level Graphen abgebildet, die in eine offene Folge in der Topologie abgeleitet werden kann. In einer offenen Folge in einer Topologie gibt es genau eine Relation ohne Ziel Komponente und genau eine Relation ohne Quell Komponente. In den nachfolgenden Strukturableitungen werden lediglich Beispiele von geschlossenen Topologie Strukturen betrachtet, jedoch lassen sich diese Beispiele analog auf offene Topologie Strukturen anpassen, indem alle Level Graph Knoten vom Komponententyp in Level Graph Knoten vom Relationstyp und alle Level Graph Knoten vom Relationstyp in Level Graph Knoten vom Komponententyp umgewandelt werden.

2. **Sequenzableitung:**
Eine Sequenz ist eine Aneinanderreihung von Folgen in einem Level Graphen, wobei der letzte Level Graph Knoten in einer Folge, der erste Level Graph Knoten in der nächsten Folge ist. Eine Sequenz in einem Level Graphen kann in eine Sequenz in einer Topologie abgeleitet werden. Dabei kann von einer Sequenz auch jede Teilsequenz in eine Sequenz in der Topologie und jede Folge der Sequenz in eine Folge in der Topologie umgewandelt werden. Sequenzen können zudem in endliche und unendliche Sequenzen unterteilt werden. Diese werden wie folgt definiert:

   (i) Eine Sequenz in einem Level Graphen ist genau dann eine endliche Sequenz, wenn kein Weg $W_{\mathrm{h}} = \{lgk_{\mathrm{s}}, ..., lgk_{\mathrm{z}}\}$ mit Zyklen von dem Start Level Graph Knoten lgk_{s} zum Ziel Level Graph Knoten lgk_{z} und kein rückwärts Weg $W_{\mathrm{r}} = \{lgk_{\mathrm{z}}, ..., lgk_{\mathrm{s}}\}$ vom Ziel Level Graph Knoten lgk_{z} zum Start Level Graph Knoten lgk_{q} in der Sequenz im Level Graphen existiert. Eine endliche Sequenz im Level Graph stellt immer eine nicht zyklische Struktur in einer Topologie dar. In der Abbildung 5.17 (1) ist ein Beispiel einer endlichen Sequenz in einem Level Graphen abgebildet.

   (ii) Eine Sequenz in einem Level Graphen ist genau dann eine unendliche Sequenz, wenn ein Weg $W_{\mathrm{h}} = \{lgk_{\mathrm{s}}, ..., lgk_{\mathrm{z}}\}$ von dem Start Level Graph Knoten lgk_{s} zum Ziel Level Graph Knoten lgk_{z} mit einen Zyklus oder ein rückwärts Weg $W_{\mathrm{r}} = \{lgk_{\mathrm{z}}, ..., lgk_{\mathrm{s}}\}$ vom Ziel Level Graph Knoten lgk_{z} zum Start Level Graph Knoten lgk_{s} in der Sequenz im Level Graphen existiert. Eine unendliche Sequenz im Level Graph kann in zyklische und nicht zyklische Strukturen in einer Topologie abgeleitet werden. In der Abbildung 5.17 (2) ist ein Beispiel einer unendlichen Sequenz in einem Level Graphen abgebildet. Die dargestellte zyklische Struktur in dem Level Graphen könnte in unendlich viele unterschiedliche Topologie Strukturen abgeleitet werden. Um zu gewährleisten, dass nur endlich viele Topologien aus einem Level Graphen in dem Verfeinerungsalgorithmus (vgl. Kap. 6.4) bei der Ableitung einer Topologie generiert werden, wird jeder Level Graph Knoten vom

Komponententyp in genau eine Komponente von dem gleichen Komponententypen abgeleitet. Danach wird jeder Level Graph Knoten vom Relationstyp in jede mögliche Relation, die kompatibel zu den zuvor abgeleiteten Komponenten ist, abgeleitet und generiert. Durch diesen Mechanismus bei der Ableitung wird sichergestellt, dass eine Level Graph Struktur nur in ein Topologiemodell und nicht in unendliche viele Topologiemodelle abgeleitet werden kann. Zudem führt dies dazu, dass der entwickelte Algorithmus in Kapitel 6.4 letztendlich auch terminiert, da nicht beliebig viele Topologiemodelle abgeleitet werden, sondern nur ein Topologiemodell.

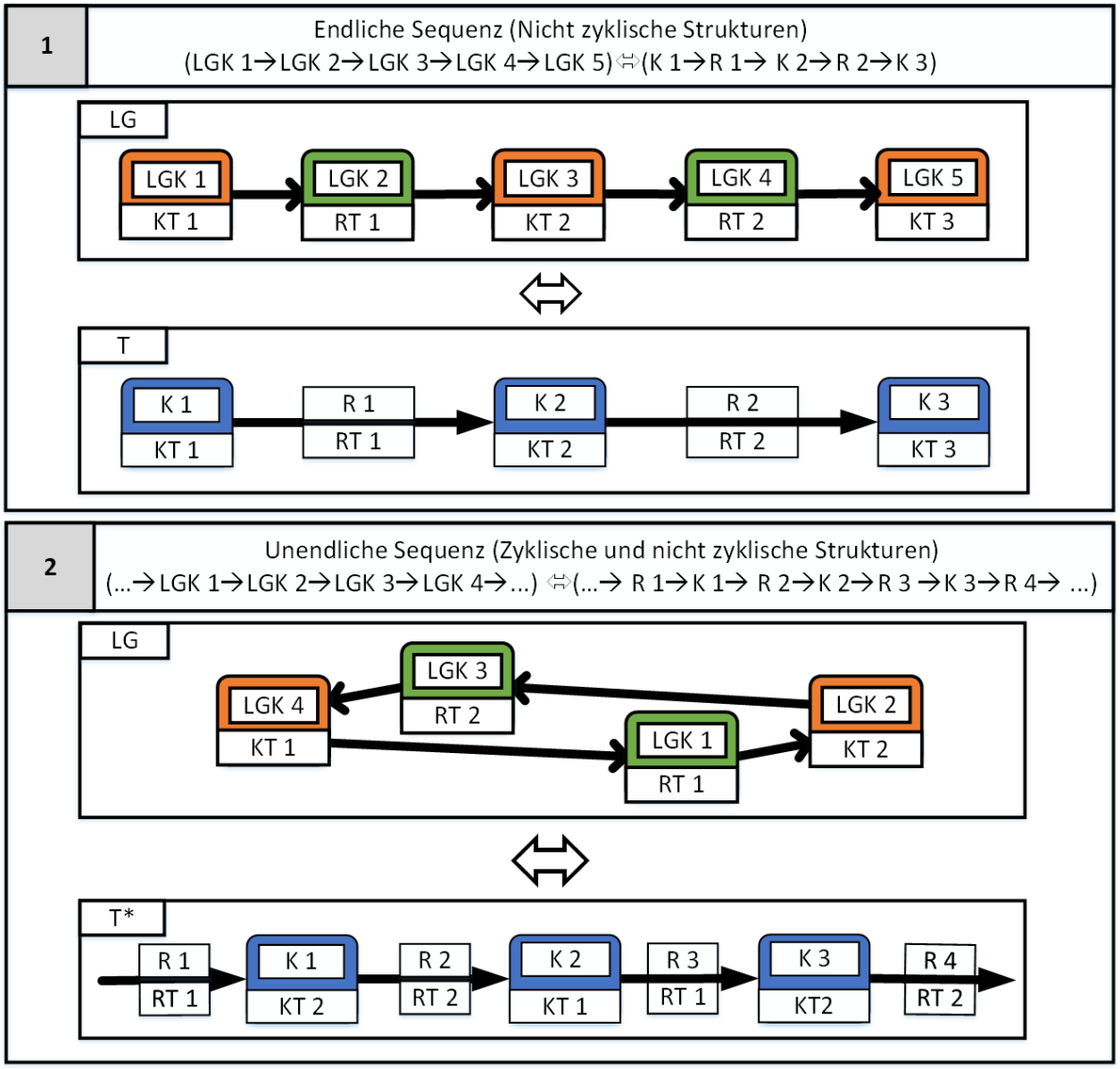

Abbildung 5.17: Sequenzableitungen

3. **Vereinigungsableitung:**
   Eine Vereinigung wird durch mindestens zwei Folgen im Level Graphen dargestellt, wobei der Ziel Level Graph Knoten aller Folgen, der gleiche Level Graph Knoten sein muss. Eine Vereinigung in einem Level Graphen kann in eine Vereinigung in einer Topologie umgewandelt werden, wobei die Vereinigung in der Topologie immer eine einzige Zielkomponente hat. Dabei können drei Arten von Vereinigungen unterschieden werde, diese Vereinigungsableitungen werden dabei wie folgt definiert:

   (i) Eine Vereinigung mit gleichen Komponententypen und gleichen Relationstypen ist eine Vereinigung, die durch mindestens zwei Folgen im Level Graphen abgebildet wird. Dabei besitzen die einzelnen Folgen in der Vereinigung einen gleichen mittleren und Ziel Level Graph Knoten. Auf der linken Seite in der Abbildung 5.18 (1) ist ein Beispiel für eine Vereinigung mit gleichen Komponententypen und Relationstypen abgebildet. Dabei müssen alle Quell Level Graph Knoten von den unterschiedlichen einzelnen Folgen den gleichen Komponententypen besitzen. In dem dargestellten Beispiel haben daher sowohl der Level Graph Knoten LGK1, als auch der Level Graph Knoten LGK4 in der Vereinigung im Level Graphen den gleichen Komponententypen KT1.

   (ii) Eine Vereinigung mit unterschiedlichen Komponententypen und gleichen Relationstypen ist eine Vereinigung, die durch mindestens zwei Folgen in dem Level Graphen abgebildet wird. Dabei besitzen die einzelnen Folgen in der Vereinigung einen gleichen mittleren und Ziel Level Graph Knoten. In der Mitte der Abbildung 5.18 (2) ist ein Beispiel für eine Vereinigung mit unterschiedlichen Komponententypen und gleichen Relationstypen abgebildet. Dabei haben alle Quell Level Graph Knoten der unterschiedlichen einzelnen Folgen in der Vereinigung im Level Graphen einen unterschiedlichen Komponententypen. In dem dargestellten Beispiel besitzt daher der Level Graph Knoten LGK1 einen Komponententypen KT1 und der Level Graph Knoten LGK4 einen anderen Komponententypen KT2 in der Vereinigung im Level Graphen.

   (iii) Eine Vereinigung mit unterschiedlichen Komponententypen und unterschiedlichen Relationstypen ist eine Vereinigung, die durch mindestens zwei Folgen in dem Level Graphen abgebildet wird. Dabei besitzen die einzelnen Folgen einen gleichen Ziel Level Graph Knoten. Auf der rechten Seite in der Abbildung 5.18 (3) ist ein Beispiel für eine Vereinigung mit unterschiedlichen Komponententypen und unterschiedlichen Relationstypen abgebildet. Dabei haben alle Quell Level Graph Knoten der unterschiedlichen einzelnen Folgen in der Vereinigung im Level Graphen einen unterschiedlichen Komponententypen und alle mittleren Level Graph Knoten einen unterschiedlichen Relationstypen. In dem dargestellten Beispiel besitzt daher der Level Graph Knoten LGK1 einen Komponententypen KT1 und der Level Graph Knoten LGK4 einen anderen Komponententypen KT2. Zudem besitzt in der Vereinigung im Level Graphen in diesem Fall der Level Graph Knoten

LGK2 einen Relationstypen RT1 und der Level Graph Knoten LGK5 einen anderen Relationstypen RT2.

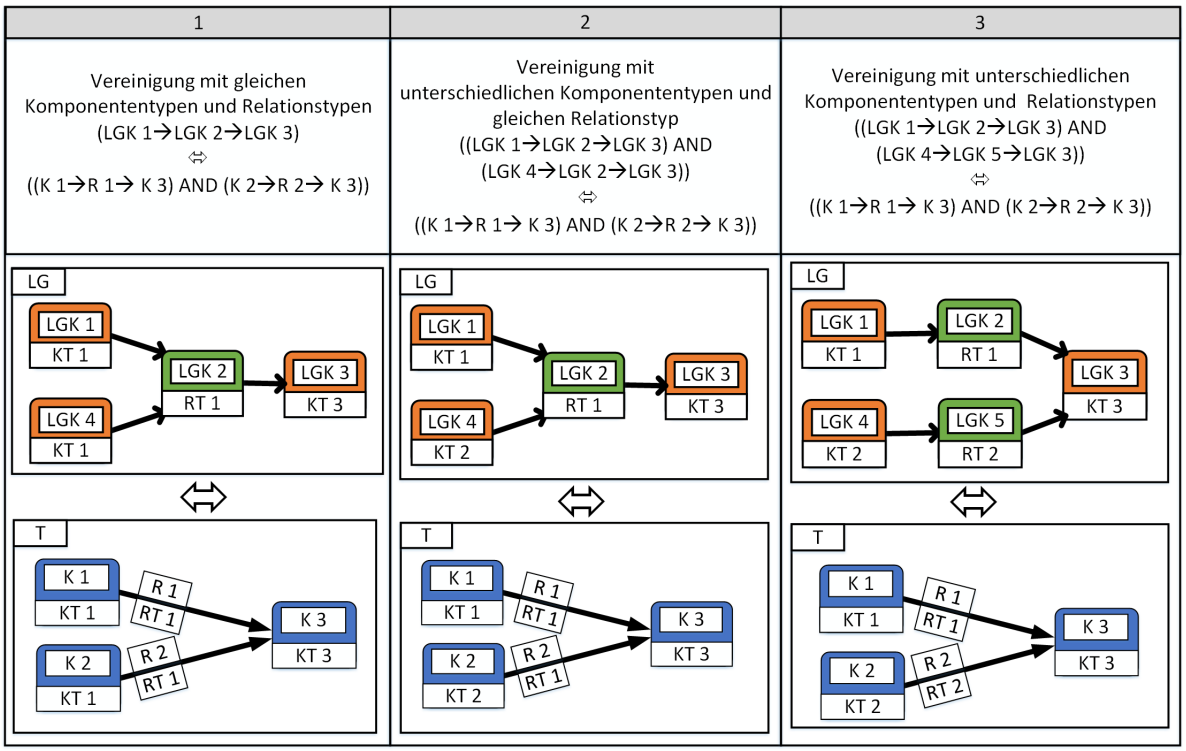

Abbildung 5.18: Vereinigungsableitungen

4. **Spaltungsableitung:**
Eine Spaltung wird durch mindestens zwei Folgen im Level Graphen dargestellt, wobei der Quell Level Graph Knoten aller einzelnen Folgen der gleiche Level Graph Knoten sein muss. Eine Spaltung in einem Level Graphen kann in eine Spaltung in einer Topologie umgewandelt werden, wobei die Spaltung in der Topologie eine einzige Quell Komponente besitzt. Dabei können drei Arten von Spaltungsableitungen unterschieden werden, diese Spaltungsableitungen werden wie folgt definiert:

   (i) Eine Spaltung mit gleichen Komponententypen und gleichen Relationstypen ist eine Spaltung, die durch mindestens zwei Folgen in dem Level Graphen abgebildet wird. Dabei besitzen die einzelnen Folgen in der Spaltung einen gleichen mittleren und Quell Level Graph Knoten. Auf der linken Seite in der Abbildung 5.19 (1) ist ein Beispiel für eine Spaltung mit gleichen Komponententypen und Relationstypen abgebildet. Dabei müssen alle Ziel Level Graph Knoten von den unterschiedlichen einzelnen Folgen den gleichen Komponententypen besitzen. In dem dargestellten Beispiel haben daher sowohl der Level Graph Knoten LGK3 als auch der Level Graph Knoten LGK4 in der Spaltung im Level Graphen den gleichen Komponententypen KT2.

(ii) Eine Spaltung mit unterschiedlichen Komponententypen und gleichen Relationstypen ist eine Spaltung, die durch mindestens zwei Folgen in dem Level Graphen abgebildet wird. Dabei besitzen die einzelnen Folgen in der Spaltung einen gleichen mittleren und einen gleichen Quell Level Graph Knoten. In der Mitte der Abbildung 5.19 (2) ist ein Beispiel für eine Spaltung mit unterschiedlichen Komponententypen und gleichen Relationstypen abgebildet. Dabei haben alle Ziel Level Graph Knoten der unterschiedlichen einzelnen Folgen in der Spaltung im Level Graphen einen unterschiedlichen Komponententypen. In dem dargestellten Beispiel besitzt daher in der Spaltung im Level Graphen, der Level Graph Knoten LGK3 einen Komponententypen KT2 und der Level Graph Knoten LGK4 einen anderen Komponententypen KT3.

(iii) Eine Spaltung mit unterschiedlichen Komponententypen und unterschiedlichen Relationstypen ist eine Spaltung, die durch mindestens zwei Folgen in dem Level Graphen abgebildet wird. Dabei besitzen die einzelnen Folgen einen gleichen Quell Level Graph Knoten. Auf der rechten Seite in der Abbildung 5.19 (3) ist ein Beispiel für eine Spaltung mit unterschiedlichen Komponententypen und unterschiedlichen Relationstypen abgebildet. Dabei haben alle Ziel Level Graph Knoten, der unterschiedlichen einzelnen Folgen in der Vereinigung im Level Graphen, einen

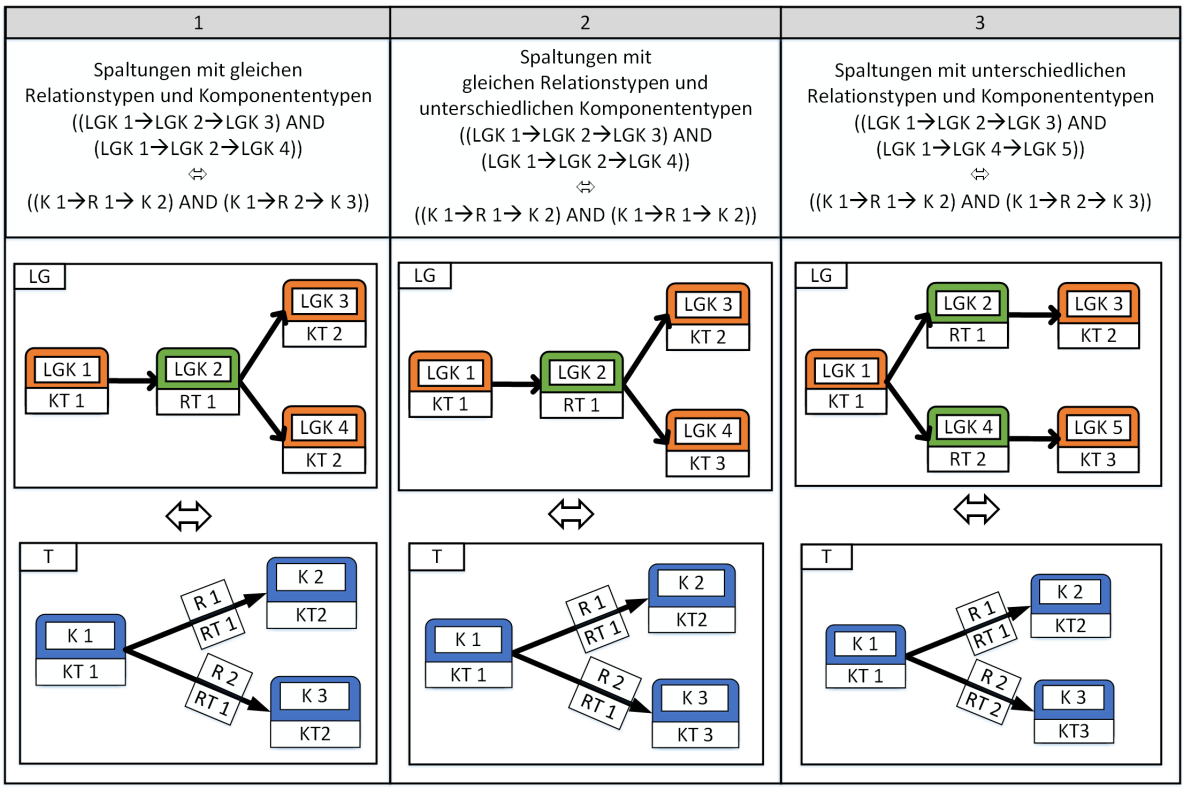

Abbildung 5.19: Spaltungsableitungen

unterschiedlichen Komponententypen und alle mittleren Level Graph Knoten einen unterschiedlichen Relationstypen. In dem dargestellten Beispiel besitzt daher der Level Graph Knoten LGK3 einen Komponententypen KT2 und der Level Graph Knoten LGK5 einen anderen Komponententypen KT3. Zudem besitzt in der Spaltung im Level Graphen in diesem Fall der Level Graph Knoten LGK2 einen Relationstypen RT1 und der Level Graph Knoten LGK4 einen anderen Relationstypen RT2.

Aus den zwei Basisableitungen und den vier Strukturableitungen lassen sich beliebige Strukturen aus einem Level Graphen in eine Topologie ableiten und umgekehrt. Die Ableitungen bilden die Grundlage für eine Generierung einer Topologie aus einem Level Graphen.

5.6 Konformität von Topologien

Anhand der oberen Ableitungsdefinitionen (vgl. Kap.5.5) kann nun festgelegt werden, wann eine Topologie zu einem Level Graphen konform ist. Diese Eigenschaft wird im Weiteren als Konformität von Topologien bezeichnet und drückt aus, ob sich eine Topologie aus einem Level Graphen ableiten lässt oder nicht. Dabei ist zu beachten, dass eine Ableitung einer Topologie aus einem Level Graph Modell nicht gleichzusetzen ist mit einer Verfeinerung einer Topologie anhand eines Level Graph Modells. Da bei einer Ableitung einer Topologie aus einem Level Graphen lediglich die Kompatibilitätsrelationen beachtet werden und bei einer Verfeinerung zusätzlich die Verfeinerungsrelationen berücksichtigt werden. Dabei kann die Konformität zum einen anhand der Anzahl an Level Graphen und zum anderen anhand der Abstraktionslevel von einem Level Graphen in unterschiedliche Kategorien unterteilt werden. Für diese Arbeit jedoch ist nur die Single Level Single Graph Konformität von Relevanz, da der Algorithmus zur Verfeinerungen von Topologien, der in Kap. 6.4 dieser Arbeit vorgestellt wird, auf der Annahme basiert, dass als Eingabe ein Single Level Single Graph konformes Topologiemodell übergeben wird.

Single Level Single Graph Konformität

Durch die Single Level Single Graph Konformität wird angegeben, ob eine Topologie konform zu genau einem Abstraktionslevel in genau einem Level Graphen ist, und somit aus diesem Abstraktionslevel des Level Graphen abgeleitet werden kann. Die Single Level Single Graph Konformität wird wie folgt definiert:

Definition 5.6.1 (Single Level Single Graph Konformität)
Eine Topologie t_{j} ist Single Level Single Graph konform $\kappa_{\text{SLSG}}(al_{\text{lg}_{\text{j}}})$ zu einem Abstraktionslevel $al_{\text{lg}_{\text{j}}}$ von einem Level Graphen lg_{j} genau dann, wenn es für jede Folge in der Topologie eine Folgeableitung in dem Kompatibilitäts-Abstraktionslevel Graphen $klg_{\text{al}_{\text{lg}_{\text{j}}}}$, des Abstraktionslevel $al_{\text{lg}_{\text{j}}}$ im Level Graphen lg_{j} gibt, aus dem die Folge in der Topologie abgeleitet werden kann:

$$\begin{gathered} t_j = \kappa_{SLSG}(al_{lg_j}), \\ \text{gilt genau dann wenn} \\ \forall((k_q \rightarrow r_m \rightarrow k_z) \in t_j)\exists(((lgk_q \rightarrow lgk_m \rightarrow lgk_z) \Leftrightarrow (k_q \rightarrow r_m \rightarrow k_z)) \in klg_{al_{lg_j}}) \end{gathered} \tag{5.15}$$

5.7 Verfeinerungs-Level Graph

Durch den Verfeinerungs-Level Graph wird festgelegt, welche Level Graph Knoten in andere Level Graph Knoten oder Level Graph Fragmente verfeinert werden können. Somit ist der Verfeinerungs-Level Graph, die Grundlage für eine automatisierte Verfeinerung von einem abstrakten Topologiemodell in ein spezifisches Topologiemodell. Die Fließrichtung des Verfeinerungs-Level Graphen geht dabei immer vom höchsten Abstraktionslevel in das tiefste Abstraktionslevel. Diese Art von Graph wird daher auch als hierarchischer Graph bezeichnet. Der Verfeinerungs-Level Graph ergibt sich aus dem Level Graphen durch Löschen aller Kompatibilitätsrelationen und wird wie folgt definiert:

Definition 5.7.1 (Verfeinerungs-Level Graph)
Der Verfeinerungs-Level Graph $\text{vlg}_{\text{lg}_{\text{j}}}$ von einem Level Grapen lg_{j} ist ein gerichteter, gefärbter, gewichteter und Zyklen freier, möglicherweise nicht zusammenhängender Graph. Durch die Kanten in dem Verfeinerungs-Level Graphen wird genau eine Fließrichtung, vom höchsten Abstraktionslevel in das tiefste Abstraktionslevel festgelegt. Der Verfeinerungs-Level Graph $\text{vlg}_{\text{lg}_{\text{j}}}$ beinhaltet alle Mengen des Level Graphen lg_{j} (vgl. Kap. 5.2.1) mit Ausnahme der Teilmenge der Kompatibilitätsrelationen $\text{KR}_{\text{lg}_{\text{j}}}$ (vgl. Kap. 5.2.6) und wird formal wie folgt definiert:

$$\begin{gathered} vlg_{\text{lg}_{\text{j}}} = (AL_{\text{lg}_{\text{j}}}, LGK_{\text{lg}_{\text{j}}}, VR_{\text{lg}_{\text{j}}}, KT_{\text{lg}_{\text{j}}}, RT_{\text{lg}_{\text{j}}}, \\ KTF_{\text{lg}_{\text{j}}}, RTF_{\text{lg}_{\text{j}}}, BE_{\text{lg}_{\text{j}}}, level_{\text{lg}_{\text{j}}}, typ_{\text{lg}_{\text{j}}}, bereitgestellt_{\text{lg}_{\text{j}}}) \end{gathered} \tag{5.16}$$

In der Abbildung 5.20 ist der Verfeinerungs-Level Graph des zuvor definierten Level Graphen (vgl. Kap. 5.3 i.V.m. Abb. 5.10) als ein Beispiel abgebildet. Zudem sind die Verfeinerungs-Level Graphen von den einzelnen Level Graph Knoten LGK 3 und LGK 7 gekennzeichnet. Diese können aus dem Verfeinerungs-Level Graphen mittels Breitensuche (engl. Breadth-First-Search (BFS)) oder Tiefensuche (engl. Depth-First-Search (DFS)) ermittelt werden.

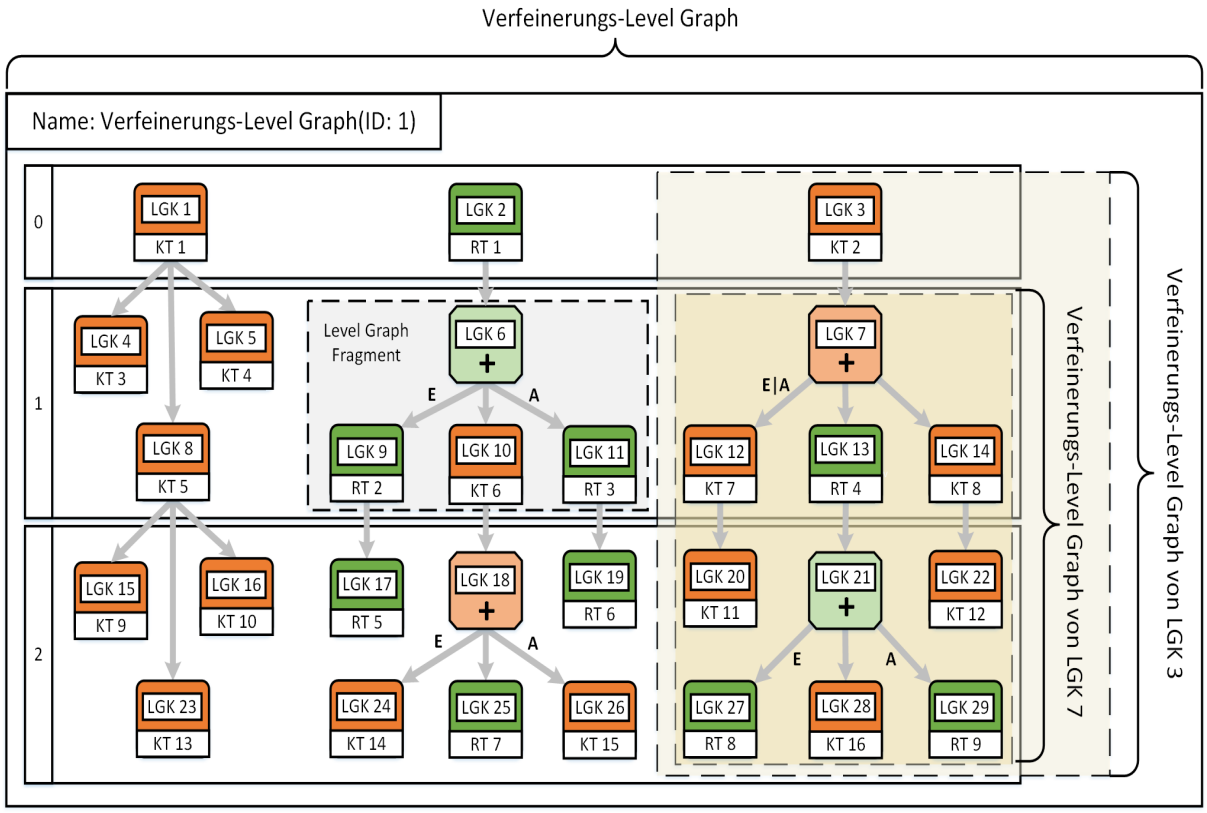

Abbildung 5.20: Verfeinerungs-Level Graph

Der Verfeinerungs-Level Graph legt zum einen fest, dass ein Quell Level Graph Knoten $lgk_{\mathrm{q}} \in LGK_{\mathrm{lg_j}}$ genau dann in einen anderen Ziel Level Graph Knoten $lgk_{\mathrm{z}} \in LGK_{\mathrm{lg_j}}$ verfeinert werden kann, wenn die Level Graph Knoten in unterschiedlichen Abstraktionslevels liegen $\pi_3(level(lgk_{\mathrm{q}})) \neq \pi_3(level(lgk_{\mathrm{z}}))$ und eine Verfeinerungsrelation $vr_{\mathrm{i}} \in VR_{\mathrm{lg_j}}$ zwischen diesen Knoten existiert. Diese Beziehung zwischen zwei Knoten ist die Verfeinerungsbeziehung und wird formal kurz geschrieben als $lgk_{\mathrm{q}} \downarrow lgk_{\mathrm{z}}$, was bedeutet das der Level Graph Knoten lgk_{q} in den Level Graph Knoten lgk_{z} verfeinert werden kann. Aus dieser Grundbeziehung lassen sich weitere Beziehungen ableiten, die in einem Verfeinerungs-Level Graphen gelten:

1. **Verfeinerungsbeziehug:**
   Wenn es eine Verfeinerungsrelation zwischen zwei Level Graph Knoten im Verfeinerungs-Level Graphen gibt, dann kann der Quell Level Graph Knoten in den Ziel Level Graph Knoten verfeinert werden:

$$lgk_{\mathrm{q}} \downarrow lgk_{\mathrm{z}} \tag{5.17}$$

2. **Transitivität:**
   Wenn es einen Weg zwischen zwei Level Graph Knote im Verfeinerungs-Level Graphen gibt, dann kann der Quell Level Graph Knoten in alle Level Graph Knoten verfeinert werden, die auf dem Weg zum Ziel Level Graph Knoten liegen:

$$(lgk_{\mathrm{q}} \downarrow lgk_{\mathrm{m}}) \wedge (lgk_{\mathrm{m}} \downarrow lgk_{\mathrm{z}}) \Rightarrow (lgk_{\mathrm{q}} \downarrow lgk_{\mathrm{z}}) \tag{5.18}$$

3. **Ausschließlichkeit:**
   Wenn ein Quell Level Graph Knoten in einen Ziel Level Graph Knoten im Verfeinerungs-Level Graphen verfeinert werden kann, dann kann der Ziel Level Graph Knoten nicht in den Quell Level Graph Knoten verfeinert werden:

$$(lgk_{\mathrm{q}} \downarrow lgk_{\mathrm{z}}) \Rightarrow \neg(lgk_{\mathrm{z}} \downarrow lgk_{\mathrm{q}}) \tag{5.19}$$

Zum anderen können anhand des Verfeinerungs-Level Graphen, die einzelnen Level Graph Fragmente identifiziert werden. In der Mitte der Abbildung 5.20 ist als Beispiel ein Level Graph Fragment vom Relationstypfragment (vgl. Kap. 5.2.9) durch ein hellgraues Rechteckt hervorgehoben. Zum Identifizieren der Fragmente im Level Graphen müssen lediglich alle ausgehenden Verfeinerungsrelationen von einem Level Graph Knoten, der vom Komponententypfragment oder Relationstypfragment ist (vgl. Kap. 5.2.8 und Kap. 5.2.9), betrachtet werden.

Des Weiteren können anhand des Level Graphen drei Arten von Verfeinerungen durchgeführt werden. Diese Kategorien werden anhand der Anzahl an ausgehenden Elementen und resultierenden Elementen differenziert. Dabei kann wie bei den Transformationsarten im Allgemeinen in eine 1-zu-1, eine 1-zu-N und eine N-zu-N Verfeinerung unterschieden werden (vgl. Kap. 2.4). Die N-zu-N Verfeinerung stellt die Verfeinerung von einer Topologie, in eine andere Topologie dar und wird in Kapitel 6.4 durch den entworfenen Verfeinerungsalgorithmus durchgeführt und an dieser Stelle im Detail erläutert. Die N-zu-N Verfeinerung besteht aus mehreren 1-zu-1 und 1-zu-N Verfeinerungen, die im Algorithmus nacheinander ausgeführt werden. In den nachfolgenden Abschnitten werden diese zwei Grundverfeinerungen im Einzelnen beschrieben und Beispiele für die jeweilige Verfeinerung aufgezeigt.

Es existiert noch eine weitere Verfeinerungskategorie, die sogenannte N-zu-1 Verfeinerung. Diese wird jedoch aktuell nicht vom Level Graph Modell unterstützt. Da diese die einzelnen Komponenten einer Topologie nicht verfeinert, sondern eher wieder verschmelzt und daher eine Form der Generalisierung darstellt. Dies ist aktuell noch eine Schwäche des Level Graphen und muss daher durch Erweiterungen des Level Graph Metamodell gelöst werden. Ein möglicher Lösungsansatz wäre den in Kapitel 6.4 vorgestellten Verfeinerungsalgorithmus um einen zusätzlichen Verschmelzungsschritt zu erweitern, der auf den entstandenen spezifischen Topologiemodellen im Anschluss ausgeführt wird.

1-zu-1 Verfeinerung

Bei der 1-zu-1 Verfeinerung handelt es sich um die leichteste Form der Verfeinerung bei der ein Level Graph Knoten von einem Komponententyp oder Relationstyp in einen anderen Level Graph Knoten vom Komponententyp oder Relationstyp verfeinert wird. Dies entspricht der Verfeinerung von einer Komponente, in eine andere Komponente (vgl. Abb. 5.21 (1)) oder von einer Relation in eine andere Relation in einer Topologie (vgl. Abb. 5.21 (2)).

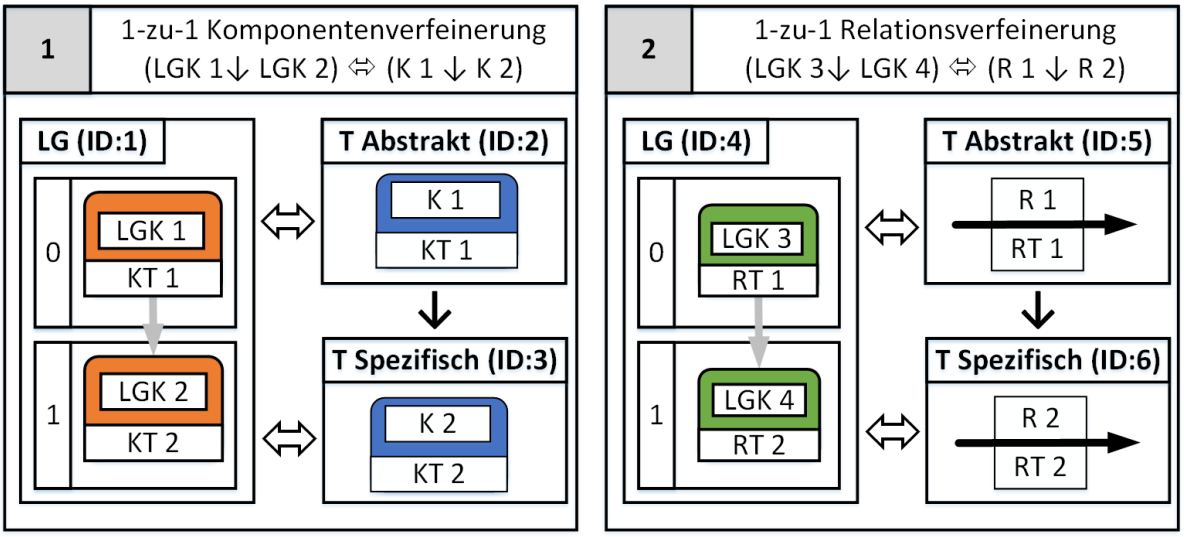

Abbildung 5.21: 1-zu-1 Verfeinerung

In der Abbildung 5.21 auf der linken Seite ist ein Beispiel für eine Komponentenverfeinerung aufgezeigt. Der Level Graph LG mit der ID 1 dient dabei als Verfeinerungsmodell und besteht aus zwei angrenzenden Abstraktionslevel mit jeweils einem Level Graph Knoten vom Komponententyp, die über eine Verfeinerungsrelation miteinander verbunden sind. Anhand von diesem Level Graphen kann zum Beispiel eine Komponente K1, die vom Komponententyp KT1 ist, in einer abstrakten Topologie (ID: 2), in eine andere spezifischere Komponente K2 vom Komponententyp KT2 verfeinert werden. Auf der rechten Seite in der Abbildung 5.21 (2) ist ein Beispiel für eine Relationsverfeinerung dargestellt. Dabei dient in diesem Fall der Level Graph LG mit der ID 4 als Verfeinerungsmodell. Dieser besteht wie in dem Beispiel zuvor aus zwei angrenzenden Abstraktionslevel. Allerdings sind in diesem Fall in den jeweiligen Abstraktionslevel Level Graph Knoten vom Relationstyp enthalten, die durch eine Verfeinerungsrelation miteinander verbunden sind. Durch diesen Level Graphen (ID: 4) könnte eine Relation R1 vom Relationstyp RT1 in einer abstrakten Topologie (ID:5) in eine spezifische Relation R2 vom Relationstyp RT2 verfeinert werden. Wenn bei einer Verfeinerung von einer komplexeren Topologie jede Komponente und Relation in dieser Topologie durch eine 1-zu-1 Verfeinerung verfeinert werden würde, dann würde sich die Struktur der abstrakten Topologie und der spezifischen Topologie nicht voneinander unterscheiden. Aus diesem Grund wurden

Level Graph Fragmente eingeführt (vgl. Kap. 5.2.8 und Kap. 5.2.9), damit über diese eine 1-zu-N Verfeinerung durchgeführt werden kann und somit die Struktur von abstrakten Topologien verändert werden kann.

1-zu-N Verfeinerung

Bei der 1-zu-N Verfeinerung handelt es sich um eine komplexere und vielfältigere Form der Verfeinerung, bei der ein Level Graph Knoten, von einem Komponententyp oder Relationstyp, über einen Level Graph Knoten, vom Komponententypfragment oder Relationstypfragment, in mehrere andere Level Graph Knoten, vom Komponententyp und Relationstyp, verfeinert wird. Dies entspricht entweder der Verfeinerung von einer Komponente in eine geschlossene Topologie oder von einer Relation in eine offene Topologie. Diese Form der Verfeinerung kann in der Praxis dann angewandt werden, wenn z.B. Komponenten modularisiert oder auf mehrere unterschiedliche Anbieter verteilt werden sollen (vgl. Abb. 5.25). Dabei wird von einer geschlossenen Verfeinerung gesprochen, wenn ein Level Graph Knoten von einem Komponententyp über einen Level Graph Knoten von einem Komponententypfragment verfeinert wird. Da durch diese Verfeinerung immer Level Graph Strukturen entstehen, die in eine gültige Topologie abgeleitet werden (vgl. Abb. 5.22). Im Gegensatz zu der geschlossenen Verfeinerung, gibt es noch die offene Verfeinerung, bei der ein Level Graph Knoten von einem Relationstyp

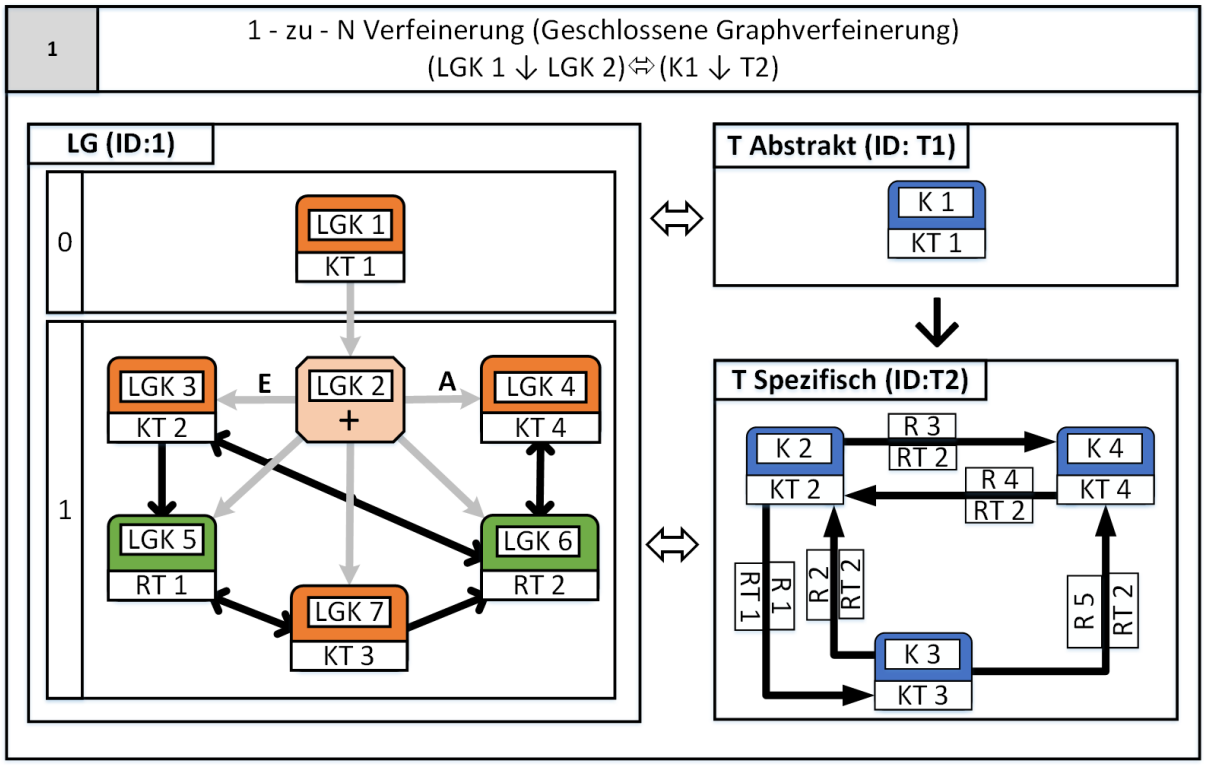

Abbildung 5.22: 1-zu-N geschlossene Topologieverfeinerung

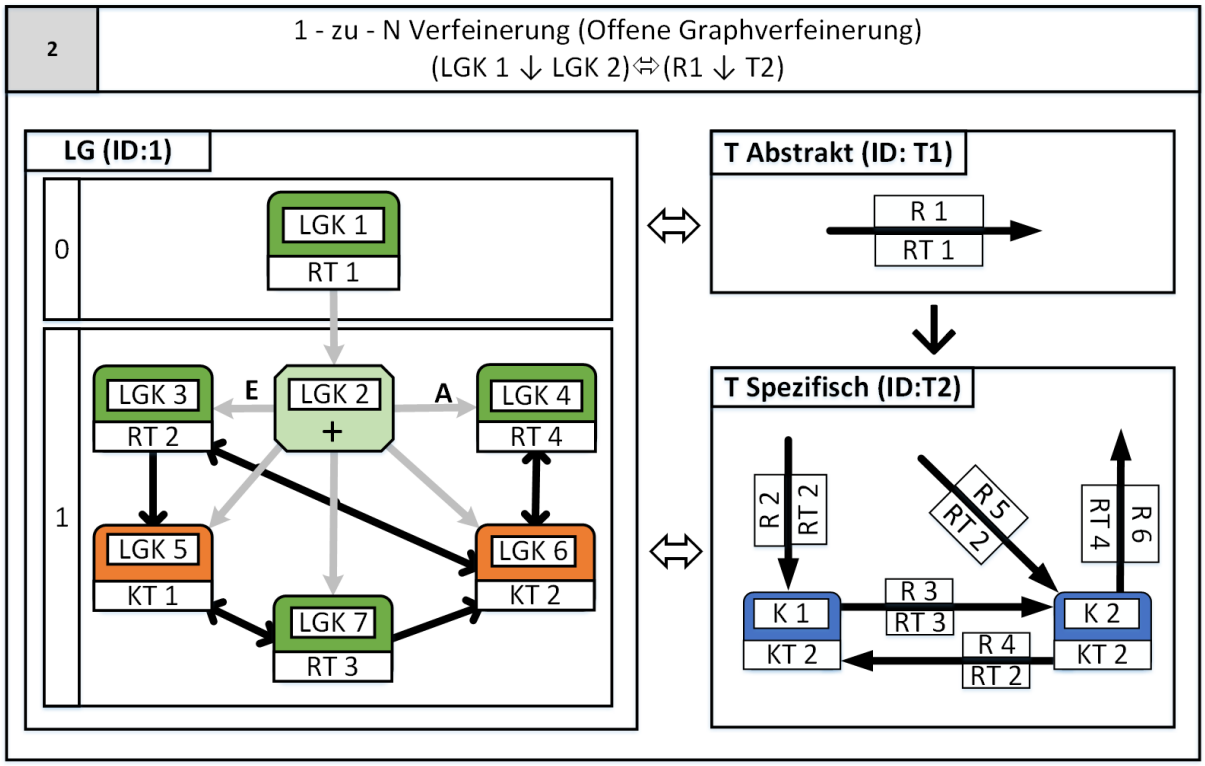

Abbildung 5.23: 1-zu-N offene Topologieverfeinerung

über einen Level Graph Knoten von einem Relationstypfragment verfeinert wird. Diese Verfeinerung wird als offene Verfeinerung bezeichnet, da durch diese offene Topologien abgeleitet werden (vgl. Abb. 5.23). Bei einer offenen Topologie existiert mindestens eine Relation ohne eine Quellkomponente und mindestens eine Relation ohne Zielkomponente. Daher stellen die offenen Topologien einzeln betrachtet keine gültigen Topologien dar. Die offenen Topologien werden dazu verwendet um zwei geschlossene Topologien miteinander zu verbinden. Damit zum Beispiel zwei geschlossene Topologien A und B mit einer offenen Topologie C miteinander verbunden werden können, müssen zunächst alle Relationen ohne Quellkomponente in der offenen Topologie C mit einer Komponente der geschlossenen Topologie A verknüpft werden. Danach müssen alle Relationen ohne Zielkomponente in der offenen Topologie C mit einer Komponente der geschlossenen Topologie B verknüpft werden.

In der Abbildung 5.22 ist ein Beispiel einer geschlossenen Topologieverfeinerung dargestellt. Der Level Graph Knoten LGK1 vom Komponententyp KT1 auf der linken Seite im Level Graphen LG (ID:1) kann in diesem Beispiel über einen weiteren Level Graph Knoten LGK2 vom Komponententypfragment, in eine komplexere Level Graph Struktur aus mehreren Level Graph Knoten vom Relationstyp und Komponententyp verfeinert werden. Auf der rechten Seite in der Abbildung 5.22 ist eine abstrakte Topologie (ID:T1) abgebildet, die anhand des Level Grapen LG (ID:1) in eine geschlossene spezifische Topologie (ID:T2) verfeinert werden

kann. In der Abbildung 5.23 hingegen ist ein Beispiel einer offenen Topologieverfeinerung dargestellt.

Die allgemeine 1-zu-N Verfeinerung kann in spezielle Verfeinerungsarten unterteilt werden, wobei diese verschiedene Bedeutungen in der Verfeinerung von Topologien haben können. Zwei dieser Spezialformen werden im Folgenden anhand von einem Beispiel vorgestellt und verdeutlicht.

1. **Kettenverfeinerung:**
   Eine Kettenverfeinerung liegt vor, wenn ein Level Graph Knoten von einem Fragment Typ genau einen Eintritts- und Austrittsknoten hat und eine Sequenz repräsentiert. Diese Art der Verfeinerung stellt eine horizontale oder vertikale Verfeinerung in einer Topologie dar. In der Praxis wird diese Verfeinerung angewandt, wenn eine Komponente modularisiert oder eine Komponente als intermediäre Komponente eingefügt wird (vgl. Abb. 5.24).

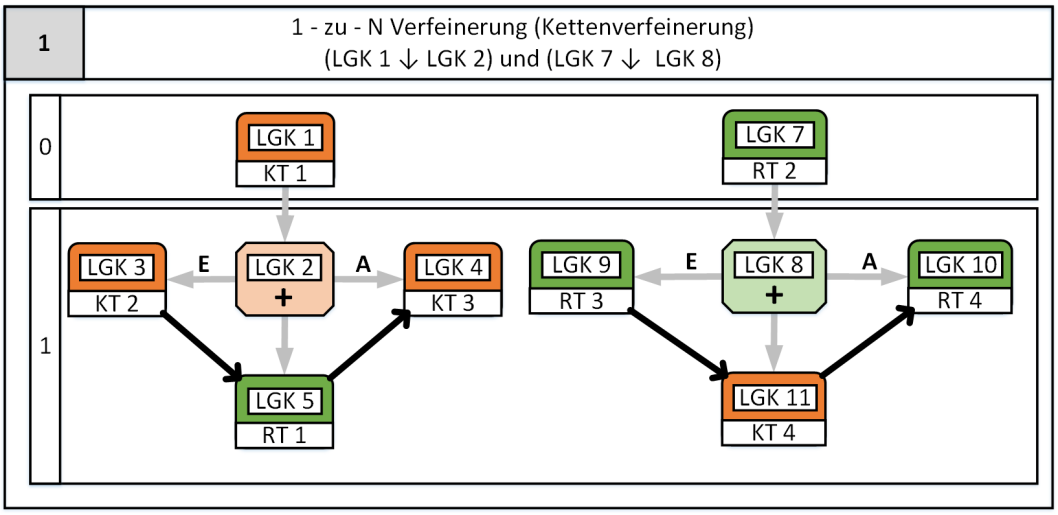

Abbildung 5.24: 1-zu-N Kettenverfeinerung

2. **Duplizierungsverfeinerung:**
   Eine Duplizierungsverfeinerung liegt vor wenn ein Level Graph Knoten vom Komponententyp eine Verfeinerungsrelation zu einem Level Graph Knoten vom Komponententypfragment hat und dieser zwei getrennte beliebige Teilgraphen repräsentiert, wobei jeder dieser Teilgraphen jeweils einen Eintritts- und Austrittsknoten beinhalten muss. Dies gilt analog für eine Verfeinerung von Relationstypen über Relationstypfragmente. Diese Art der Verfeinerung kann dazu genutzt werden Bestandteile einer Topologie zu duplizieren (vgl. Abb. 5.25). Ein Anwendungsfall in der Praxis für diese Art wäre, das Verfeinern von einer Cloud-Anwendung die auf mehreren unterschiedlichen Anbietern bereitgestellt und ausgeführt werden soll.

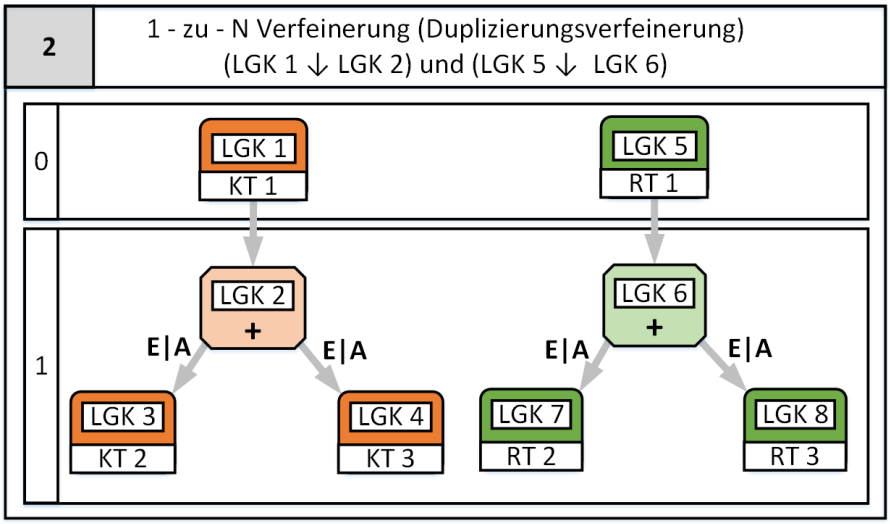

Abbildung 5.25: 1-zu-N Duplizierungsverfeinerung

Dies waren nur zwei Sonderformen, die durch einen Level Graphen modelliert und für die Verfeinerung genutzt werden können. Dabei könnte durch die Kombination von diesen Sonderformen wiederum weitere Verfeinerungsstrukturen erstellt werden, wie z.B. eine Ketten-Duplizierungs-Verfeinerung, bei der eine Komponente in zwei Sequenzen dupliziert wird, dadurch kann man Parallelität in den verfeinerten Topologien erzeugen (vgl. Abb. 5.26).

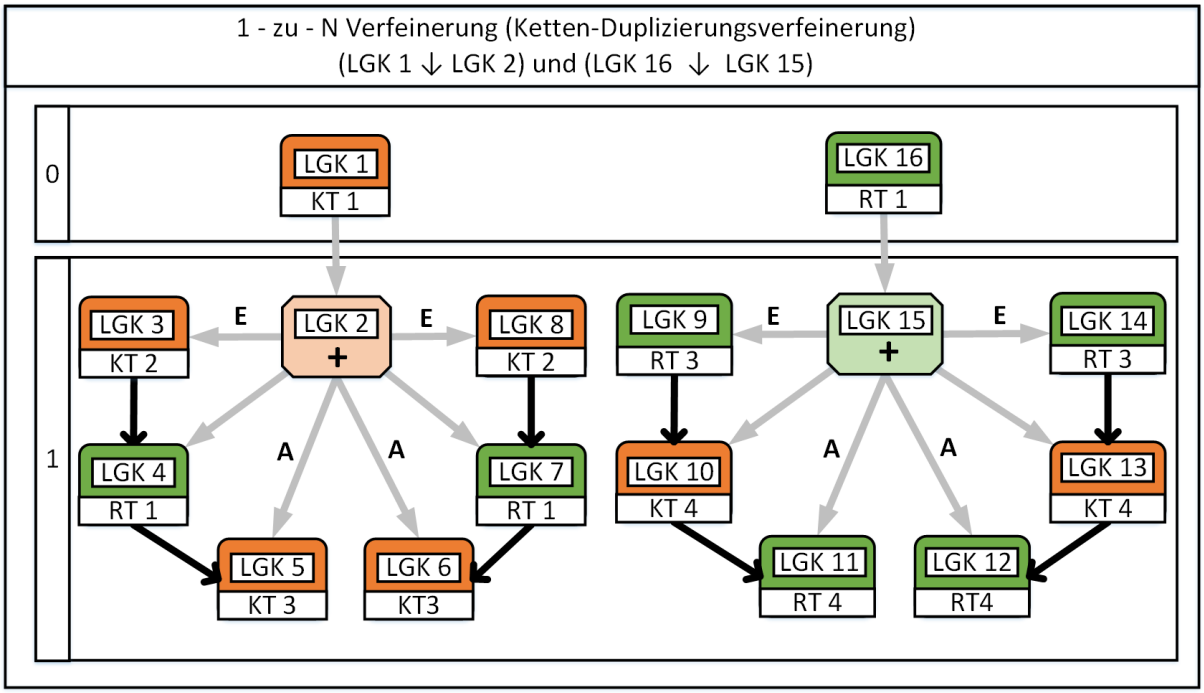

Abbildung 5.26: 1-zu-N Ketten-Duplizierungsverfeinerung

6 Methode zur Verfeinerung von Topologien

Im Folgenden Kapitel wird auf Basis des definierten Topologie Metamodell und des Level Graph Metamodell, die Methode zur semi-automatischen Verfeinerung von Topologiemodellen erläutert. Die Abbildung 6.1 stellt den entwickelten semi-automatisierten Verfeinerungsprozess mit seinen wesentlichen Hauptbestandteilen gesamtheitlich dar. Dieser kann in fünf Prozessschritte unterteilt werden, die bei einer Verfeinerung einmalig oder mehrfach durchlaufen werden (vgl. Abb. 6.1). Der Prozess beginnt dabei mit der Modellierung eines Level Graphen, welcher als ein Verfeinerungsmodell dient (1). Im zweiten Schritt wird eine abstrakte Topologie entweder abhängig oder unabhängig von dem modellierten Level Graph Modell erstellt (2). Im nächsten Schritt werden die modellierten Modelle mit erwarteten bzw. bereitgestellten Eigenschaften angereichert (3). Der vierte Schritt stellt die automatisierte Verfeinerung von der abstrakten Topologie anhand eines Level Graphen durch einen Verfeinerungsalgorithmus dar

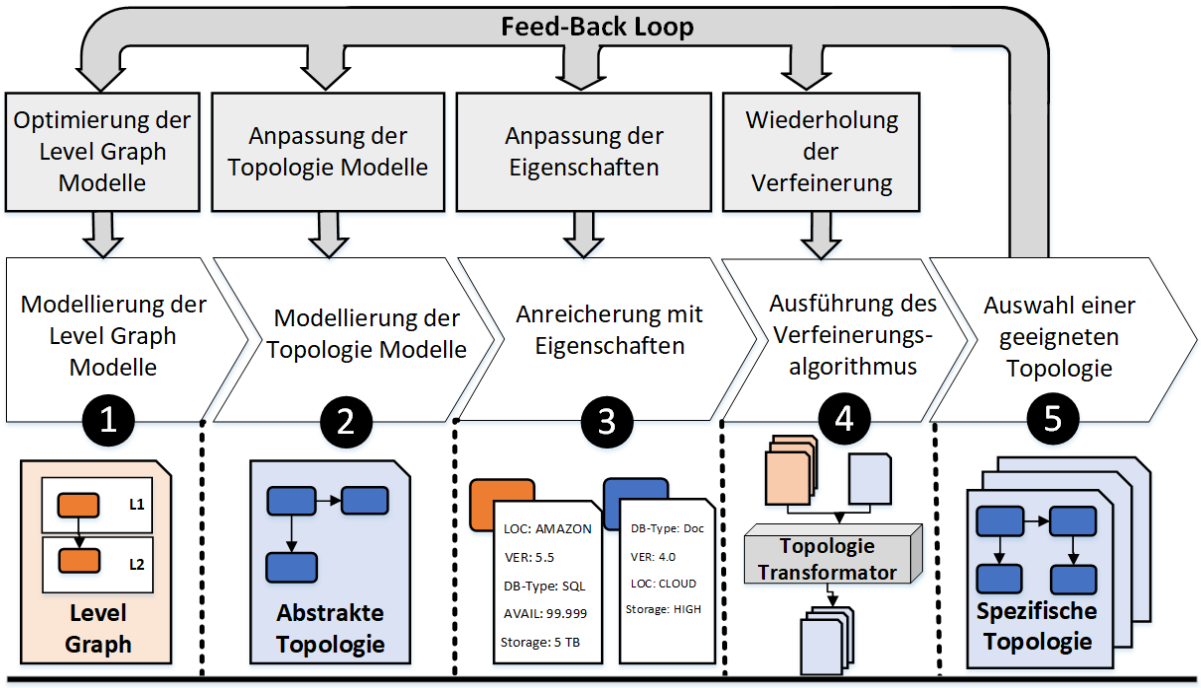

Abbildung 6.1: Verfeinerungsverfahren für Topologien

(4). Im letzten Schritt können die erstellten spezifischen Topologien manuell von erfahrenen Systemarchitekten ausgewählt und angepasst werden. Alternativ könnten die Schritte (1) bis (4) durch eine Feed-Back Loop wiederholt werden und die jeweiligen Topologiemodelle oder Level Graph Modelle angepasst und optimiert werden. Die einzelnen Schritte werden in den nachfolgenden Abschnitten im Detail konkretisiert, dabei wird herausgestellt was bei den jeweiligen Schritten zu beachten ist und was als Ergebnis in jedem Schritt produziert wird.

6.1 Schritt 1: Modellierung der Level Graph Modelle

Im ersten Schritt des semi-automatischen Verfeinerungsprozesses muss zunächst ein Level Graph Modell erstellt werden, anhand dem die Verfeinerung durchgeführt werden kann. Dies erzeugt zu Beginn der Einführung eines Verfeinerungsprozesses zunächst einen Mehraufwand, weil ein zusätzliches Modell zu den Topologiemodellen für die Verfeinerung erstellt werden muss. Jedoch muss dieses Modell nur einmal erstellt werden und dann im Laufe der Zeit innerhalb eines Level Graph Managementprozesses angepasst werden. Dazu werden Operationen wie Löschen, Updaten oder Hinzufügen von Level Graph Knoten und Level Graph Relationen auf einem Level Graph Modell im Laufe der Zeit durchgeführt. Diese Operationen müssen aktuell noch manuell im ArchRef Prototyp (vgl. Kap. 7) ausgeführt werden. Allerdings wäre das Ziel auch diese Schritte zukünftig durch ein Registrierungsservice für neue Level Graph Knoten und Level Graph Relationen zu automatisieren, damit der Aufwand für die Modellierung von einem Level Graphen reduziert werden kann (vgl. Kap. 8.2). Ein weiterer Punkt der bei der Erstellung von Level Graph Modellen zu beachten ist, sind die Eigenschaften welche ein Level Graph Modell aufweisen kann. Diese Eigenschaften können aus den Eigenschaften, welche Modelle im Allgemeinen aufweisen, abgeleitet und speziell auf Level Graph Modelle angepasst werden (vgl. Kap. 2.2.1). Dabei wurden folgende vier wesentliche Eigenschaften identifiziert, die ein Level Graph Modell aufweisen kann:

- ***Minimalität:***
  Da die Laufzeit des Verfeinerungsalgorithmus von der Anzahl der Knoten sowie der Anzahl von Verfeinerungsrelationen im Level Graphen abhängig ist (vgl. Kap. 6.4), sollte die Menge der Knoten und der Kanten in einem Level Graphen so minimal wie möglich sein. Daher sollte ein Level Graph Modell nur jeden Komponententyp (vgl. Kap. 4.3) und Relationstyp (vgl. Kap. 4.5) durch einen Level Graph Knoten repräsentieren, damit die Anzahl der Knoten so minimal wie möglich ist. Zudem sollten nicht relevante Verfeinerungsrelationen sowie Kompatibilitätsrelationen (vgl. Kap. 4.5) vermieden werden.
- ***Genauigkeit:***
  Durch ein Level Graph Modell sollten nur Topologien erstellt werden können, die auch in der Realität vorkommen können. Daraus folgt, dass keine fiktiven Modelle durch ein Level Graph Modell entstehen sollten.

- ***Allgemeingültigkeit:***
  Durch ein Level Graph Modell sollten auch Topologiemodelle erstellt und verfeinert werden können, die möglicherweise noch nicht in der realen Welt bekannt sind. Dadurch sollen bei der Verfeinerung von Topologien neue Systemarchitekturstrukturen entdeckt werden können. Der vollständige Level Graph, der in jedem seiner Abstraktionslevel jeden möglichen Typ als einen Level Graph Knoten sowie jede valide Verfeinerungsrelation und Kompatibilitätsrelation zwischen seinen Level Graph Knoten beinhaltet, stellt einen perfekten allgemeingültigen Level Graphen dar. Da durch diesen alle Möglichkeiten einer Verfeinerung abgedeckt sind. Jedoch ist die Anzahl der Kanten und Knoten in diesem Graphen sehr groß und die Genauigkeit sehr gering.
- ***Einfachheit:***
  Die Struktur des Level Graphen sollte möglichst leicht verständlich bleiben, daher sollten unnötige Level Graph Knoten sowie Kompatibilitäts- und Verfeinerungsrelationen und die Anzahl der Abstraktionslevel so gering wie möglich gehalten werden.

Je nachdem was der Zweck eines Level Graph Modell ist, können die Eigenschaften von Level Graph Modellen unterschiedlich priorisiert werden (vgl. Kap. 2.2.1). In dieser Arbeit liegt der Fokus auf Level Graph Modelle, welche die Topologiemodelle möglichst genau verfeinern können. Daher spielt die Genauigkeit eine übergeordnete Rolle, gegenüber den anderen Eigenschaften von Level Graph Modellen. Der letzte Punkt der bei der Erstellung von Level Graph Modellen als Architekt zu beachtet ist, ist das nur valide Level Graph Modelle erstellt werden. Ein Level Graph Modell gilt als valide, wenn es keine Beschränkungen, die im Level Graph Metamodell existieren, verletzt (vgl. Kap 5.2). Um dies zu gewährleisten gibt es zwei Ansätze die verfolgt werden können. Der erste Ansatz wäre ein beliebiges Level Graph Modell zu entwerfen und anschließend durch einen Algorithmus auf Verletzungen zu prüfen und die Verletzungen dem Architekten anzuzeigen, damit dieser anschließend Anpassungen vornehmen kann. Der zweite Ansatz wäre erst gar keine ungültigen Level Graph Modelle als Architekt erstellen zu können. Dieser Ansatz wird auch in der ArchRef Prototyp Implementierung verfolgt (vgl. Kap. 7). Somit ist das Ergebnis am Ende des ersten Schrittes in dem Verfeinerungsprozess, ein valides Level Graph Modell, welches für die Verfeinerung eingesetzt und dem Verfeinerungsalgorithmus als Eingabe übergeben werden kann.

6.2 Schritt 2: Modellierung der Topologiemodelle

In dem zweiten Schritt des Verfeinerungsprozesses, muss ein Topologiemodell erstellt werden, welches als Ausgangsmodell für die Verfeinerung dient. Dabei existieren folgende zwei Möglichkeiten um das Topologiemodell zu erstellen:

- ***Unabhängige Modellierung***:
  Das Topologiemodell wird durch die direkte Verwendung von Komponententypen und Relationstypen aus unterschiedlichen Repositories modelliert. Wobei ein Repository

eine Ansammlung von existierenden Komponententypen und Relationstypen darstellt (vgl. Kap. 7). Dabei kann es sein, dass Topologien entstehen die Komponententypen und Relationstypen verwenden, die in den Level Graph Modellen möglicherweise nicht durch Level Graph Knoten repräsentiert werden oder in unterschiedlichen Abstraktionslevel des Level Graphen zugeordnet sind. Dies hat zur Folge, dass Topologien mit einem gemischten Abstraktionsgrad oder Topologien, die nicht konform zu Level Graph Modelle sind, entstehen können. In dieser Arbeit wurde jedoch angenommen, dass ausschließlich Modelle erstellt werden, die einen gleichen Abstraktionsgrad aufweisen und konform zu genau einem Abstraktionslevel des Level Graphen sind. Daher ist in dem aktuellen ArchRef Prototyp noch kein Algorithmus implementiert, welcher gemischte Topologien verfeinern kann. Jedoch unterstützt der ArchRef Prototyp, die Modellierung von unabhängigen Topologiemodellen (vgl. Kap. 7.2 u. Abb. 7.4) und kann daher um eine Verfeinerung von Topologiemodelle mit gemischten Abstraktionsgrad erweitert werden (vgl. Kap. 8.2).

- ***Abhängige Modellierung***:
  Das Topologiemodell wird durch die Verwendung von Komponententypen und Relationstypen erstellt, die in Level Graph Modellen enthalten sind. Durch diese Art der Modellierung können Level Graph konforme Topologiemodelle erstellt werden. Dabei ist die Konformität von Topologien abhängig von den Abstraktionsleveln und der Menge von Level Graphen aus denen die Komponententypen und Relationstypen für die Erstellung einer Topologie entnommen wurden (vgl. Kap. 5.6). Der ArchRef Prototyp unterstützt den Architekten dabei Topologiemodelle, die Single Level Single Graph Konform sind zu erstellen (vgl. Kap. 7.2 und Abb. 7.3).

In dieser Masterarbeit wurde für die Implementierung des Verfeinerungsalgorithmus angenommen, dass am Ende des zweiten Verfeinerungsschrittes abstrakte Topologiemodelle heraus kommen, die eine Single Level Single Graph Konformität zu dem Abstraktionslevel der Tiefe 0 von genau einem Level Graphen aufweisen (vgl. Def. 5.6.1). Dadurch ist gewährleistet, dass die Topologiemodelle einen durchgängigen Abstraktionsgrad besitzen, welches dem ersten Abstraktionslevel in den Level Graph Modellen entsprechen. Der zweite Schritt im Verfeinerungsprozess wird immer dann wiederholt, wenn neue abstrakte Topologiemodelle benötigt werden.

6.3 Schritt 3: Anreicherung mit Eigenschaften

Im dritten Schritt werden die Level Graph Modelle und die Topologiemodelle um bereitgestellte Eigenschaften bzw. erwartete Eigenschaften angereichert (vgl. Kap. 4.6), falls diese nicht schon während der Modellierung in Schritt eins und zwei angereichert wurden. Die Anreicherung mit Eigenschaften ist notwendig um die Verfeinerung von Topologiemodellen zu optimieren. Da bisher die Verfeinerung nur auf Basis der Komponententypen und Relationstypen durchgeführt wird (vgl. Kap. 5.7), stellt jede Verfeinerungsrelation eine gültige Verfeinerung dar. Dadurch

müsste jede Kombinationsmöglichkeit mit jeder Verfeinerungsrelation ausprobiert werden. Um die Anzahl von gültigen Verfeinerungsrelationen zu beschränken, wurde das Konzept von erwarteten und bereitgestellten Eigenschaften eingeführt (vgl. Kap. 4.6). Durch diese Konstrukte soll die Anzahl an möglichen Verfeinerungen eingeschränkt und das Auffinden von erwünschten Topologiemodellen erhöht werden.

In der Abbildung 6.2 ist im oberen Teil eine abstrakte Topologie mit der ID 123 mit einer Datenbank Komponente (ID:1) vom Komponententyp Datenbank_Typ (ID:2) abgebildet. Ein Systemarchitekt könnte nun zu der Entscheidung kommen, die Daten in Form von Dokumente, wie z.B. als JSON Dateien in einer Datenbank abzuspeichern. Daher definiert er, die erwartete Eigenschaft Speicherart mit den Wert Dokumente und hängt diese an die Datenbank Komponente (ID:1) an. Diese Komponente könnte nun anhand von dem in der Abbildung 6.2 unten dargestellten Level Graph Modell (ID:321) verfeinert werden. Dabei würde die Verfeinerung im ersten Abstraktionslevel mit der Tiefe 0 beginnen und nach einem gleichen Komponententyp suchen und bei dem Level Graph Knoten Datenbank (ID:3) fündig werden. Anschließend würde der Verfeinerungsalgorithmus prüfen, ob die Ziel Level Graph Knoten der ausgehenden Verfeinerungsrelationen, in diesem Fall der MongoDB (ID:4) und MS_SQL_Server (ID:5) Level Graph Knoten, die erwarteten Eigenschaften der Datenbank Komponente mit ID 1 erfüllen (vgl. Kap. 6.4). In diesem Fall würde nur der MongoDB (ID:4) Level Graph Knoten vom MongoDB_Typ (ID:6) die erwartete Eigenschaft erfüllen und damit die einzige gültige Verfeinerung für die Datenbank mit der ID 1 darstellen.

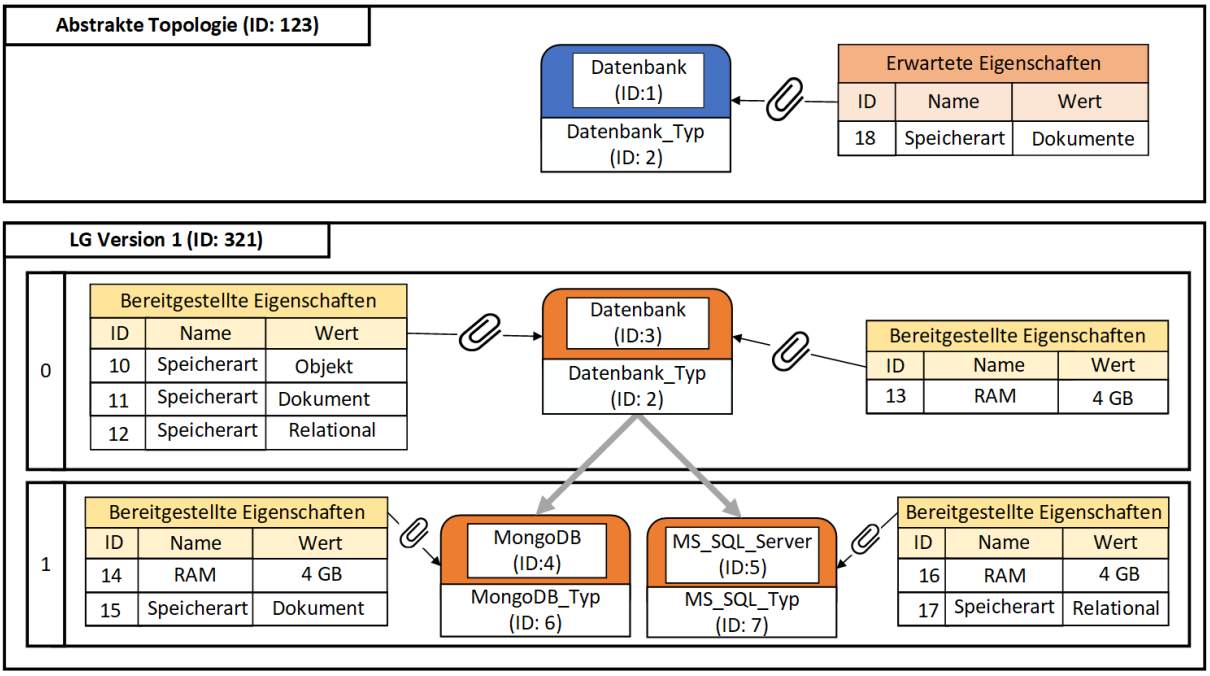

Abbildung 6.2: Erwartete und bereitgestellte Eigenschaften

Des Weiteren können Level Graph Knoten unabhängig von den bereitgestellten Eigenschaften ihres Typen, eigene zusätzliche bereitgestellte Eigenschaften aufweisen. Durch diese zusätzlichen bereitgestellten Eigenschaften wird eine Individualisierung von Level Graph Modellen ermöglicht, da nun zwei Level Graph Knoten, die zwar vom gleichen Typen sind, sich in ihren bereitgestellten Eigenschaften unterscheiden können. Mit diesem Mechanismus können unterschiedliche Konfigurationsmöglichkeiten von dem gleichen Typen in einem Level Graphen dargestellt werden. Zum Beispiel könnte in der Abbildung 6.2 ein zweiter MongoDB Level Graph Knoten vom MongoDB_Typ mit einer anderen ID dargestellt werden, der jedoch 8 GB Ram und eine zusätzliche Eigenschaft Storage mit 30 GB bereitstellt. Ein weiterer Grund für die individuellen bereitgestellten Eigenschaften von Level Graph Knoten, sind die Level Graph Fragmente. Da durch die Kombination von mehreren Level Graph Knoten neue Eigenschaften in der Summe entstehen können, die durch die einzelnen Level Graph Knoten nicht bereitgestellt werden konnten. Zudem gilt, dass ein Level Graph Knoten alle bereitgestellten Eigenschaften von allen möglichen Verfeinerungen beinhalten muss, damit eine durchgängige Verfeinerung ermöglicht werden kann. Daher besitzt in der Abbildung 6.2 der Datenbank Level Graph Knoten mit der ID 1 sowohl die bereitgestellten Eigenschaften des MongoDB Level Graph Knoten mit der ID 4, als auch die bereitgestellten Eigenschaften des MS_SQL_Server mit der ID 5. Um sicher zu stellen, dass ein Level Graph Knoten alle bereitgestellten Eigenschaften seiner spezifischen Level Graph Knoten beinhaltet muss immer, wenn eine bereitgestellte Eigenschaft hinzugefügt wird, diese im Verfeinerungs-Level Graphen nach oben durch propagiert werden. Dieser Mechanismus wird jedoch aktuell noch nicht von dem ArchRef Tool unterstützt.

Der Name wie auch der Wert von Eigenschaften werden durch String Datentypen repräsentiert (vgl. Kap. 4.6). Daher wird nur auf Gleichheit der Werte im Verfeinerungsalgorithmus geprüft. Das Matching von Eigenschaften wird dabei über den Namen der Eigenschaften durchgeführt, daher werden zwei Eigenschaften als gleiche Eigenschaften betrachtet, wenn ihre Namen gleich sind. Da keine anderen Datentypen außer Strings für die Werte von Eigenschaften zugelassen werden, sind keine Restriktionen auf den Wertebereichen der jeweiligen Eigenschaften, wie z.B. die Speicherkapazität der Komponente sollte größer als 1 TB sein oder die Kosten für eine Komponente sollten zwischen 1000 und 5000 Euro liegen, möglich. Daher muss das Metamodell um weitere Bestandteile und Metriken erweitert werden (vgl. Kap. 8). Die Endergebnisse des dritten Schrittes im Verfeinerungsprozesses sind somit angereicherte Level Graph Modelle und Topologiemodelle mit bereitgestellten bzw. erwarteten Eigenschaften.

6.4 Schritt 4: Ausführung des Verfeinerungsalgorithmus

Im vierten Schritt des Verfeinerungsprozesses wird die automatisierte Verfeinerung von Topologien anhand von einem Algorithmus ausgeführt. Dabei werden nachfolgende Annahmen in dieser Arbeit zu Vereinfachung und Eingrenzung getroffen:

- **1. Annahme:** Der Algorithmus bekommt ein abstraktes Topologiemodell sowie Level Graph Modell als Eingabe übergeben. Diese Modelle sind zu den Definitionen im Kapitel 4 bzw. Kapitel 5 valide.
- **2. Annahme:** Die eingegebene abstrakte Topologie ist Single Level Graph Konform zu dem Abstraktionslevel mit der Tiefe 0 in dem eingegebenen Level Graph Modell, welches für die Verfeinerung verwendet wird (vgl. Def. 5.6.1). Dadurch besitzt jedes Element in der Topologie den gleichen Abstraktionsgrad und die Topologie gibt diesen Abstraktionsgrad dem Verfeinerungsalgorithmus als Tiefe des Startabstraktionslevels mit.
- **3. Annahme:** In dem Verfeinerungsalgorithmus wird eine Topologie von dem Startabstraktionslevel immer in das nächste tiefer liegende Abstraktionslevel des Level Graphen verfeinert. Danach werden die Ergebnisse zurück gegeben und es wird zunächst eine Auswahl getroffen bevor eine erneute Verfeinerung mit dem Algorithmus ausgeführt wird. Dadurch soll die Anzahl an vorgeschlagenen spezifischen Topologien reduziert werden und dem Systemarchitekten eine bessere Kontrolle über die Verfeinerung ermöglicht werden.

Algorithmus 6.1 InitializationTopologieRefinement($t_{abstract} \in T$, $lg_{refine} \in LG$)

```
1:
2:  // Initialisierung aller benötigten Variablen für die Verfeinerung von Topologien
3:
4:  T_solutions := { };                          // Lösungsmenge von generierten Topologiemodellen
5:  lg_solultion := new lg();                                  // Level Graph Lösungsmodell
6:  al_start := absractionLevelOf(t_abstract);                 // Startlevel der Verfeinerung
7:  queue_TE := K_t_abstract ∪ R_t_abstract;        // Queue mit allen abstrakten Topologieelementen
8:
9:  // Wähle ein zufälliges Topologieelement als Startelement aus und starte die Verfeinerung
10: if (K_t_abstract ≠ ∅ ) then
11:     te_start := queue_TE.next();
12:     REFINEMENTSTEP(te_start, lg_solution, T_solutions, al_start, queue_TE, lg_refine);
13: end if
14:
15: return T_solutions;
16:
```

In dem Listing 6.1 ist die Initialisierung für die automatische Verfeinerung von Topologiemodellen in Pseudo-Code dargestellt. Dabei sind alle wesentlichen benötigten Variablen, die für eine automatische Verfeinerung verwendet werden abgebildet. Der Verfeinerungsalgorithmus bekommt zunächst eine valide abstrakte Topologie t_{abstract} sowie ein Level Graph Modell lg_{refine}, welches als Verfeinerungsmodell dient, als Eingabe übergeben. Des Weiteren wird noch eine leere Menge von Topologiemodellen $T_{\text{solutions}}$ definiert. In dieser Menge werden die gefundenen und generierten spezifischen Topologiemodelle hinzugefügt und am Ende als Lösungsmenge an den Systemarchitekten zurückgegeben. Zusätzlich zu der Lösungsmenge von Topologiemodellen, wird noch ein leeres Level Graph Modell lg_{solution} initialisiert. Dieses Level Graph Modell repräsentiert die aktuelle verfeinerte Level Graph Lösungsstruktur, welche letztendlich in ein spezifisches Topologiemodell abgeleitet wird. Danach wird noch das Startabstraktionslevel al_{start} anhand von dem eingegebenen abstrakten Topologiemodell t_{abstract} bestimmt und alle Komponenten und Relationen der abstrakten Topologie werden in eine Warteschlange $queue_{\text{TE}}$ vereinigt. Abschließend wird geprüft ob, die Menge der Komponenten in der abstrakten Topologie nicht leer sind. Falls Komponenten in der abstrakten Topologie vorhanden sind, danm wird ein beliebiges Topologieelement aus der Warteschlange $queue_{\text{TE}}$ als Startelement te_{start} entnommen und der erste Verfeinerungsschritt wird mit diesem Topologieelement aufgerufen.

Der Pseudo-Algorithmus für den Verfeinerungsschritt ist im Listing 6.2 im Detail aufgeführt. Dabei wird die *RefinementStep* Methode als Rekursion definiert. In dieser Methode wird zuerst nach einem Level Graph Knoten lgk_{q} im Startabstraktionslevel al_{start} gesucht, welcher den gleichen Typen aufweist wie das aktuell zu verfeinernde Topologieelement te_{start} (vgl. 6.2 (1)). Wenn ein Level Graph Knoten mit gleichen Typen wie das aktuelle Topologieelement gefunden wurde, müssen alle ausgehenden Verfeinerungsrelationen auf Gültigkeit überprüft werden. Damit eine Verfeinerungsrelation eine gültige Verfeinerung darstellt, muss zunächst der Ziel Level Graph Knoten der Verfeinerungsrelation in dem angrenzenden Abstraktionslevel des Startabstraktionslevels al_{start} liegen. Das bedeutet, dass aktuell nur Verfeinerungsrelationen betrachtet werden, die nicht über mehrere Abstraktionslevel hinweggehen. Zudem müssen alle erwarteten Eigenschaften des Topologieelementes te_{start} durch bereitgestellte Eigenschaften des Ziel Level Graph Knoten der Verfeinerungsrelation $\pi_3(vr_i)$ erfüllt sein (vgl. 6.2 (2)). Damit eine erwartete Eigenschaft als erfüllt gilt, muss sowohl der Name als auch der Wert der Eigenschaft mit dem Namen und dem Wert von einer der bereitgestellten Eigenschaften übereinstimmen.

Das Matching zwischen den erwarteten Eigenschaften und den bereitgestellten Eigenschaften wird in der *PropertiesProvided* Methode ausgeführt. Nachdem eine gültige Verfeinerungsrelation gefunden wurde, muss geprüft werden, ob die Verfeinerung $lgk_q \downarrow \pi_3(vr_i)$ kompatibel zu den vorigen bereits durchgeführten Verfeinerungen lg_{solution} ist (vgl. 6.2 (3)). Dabei muss bei einer Verfeinerung einer Relation der abstrakten Topologie geprüft werden, ob diese kompatibel zu den vorher durchgeführten Verfeinerungen von ihrer Quell- und Zielkomponente ist. Wohingegen bei einer Verfeinerung einer Komponente der abstrakten Topologie geprüft werden muss, ob diese kompatibel zu den vorher durchgeführten Verfeinerungen aller ausgehenden oder eingehenden Relationen ist.

Algorithmus 6.2 RefinementStep(te_{start}, $lg_{solution}$, $T_{solutions}$, al_{start}, $queue_{TE}$, lg_{refine})

```
1:
2:  // (1) Suche nach gleichen Typen im Verfeinerungsmodell
3:  for (lgk_q ∈ LGK_{lg_refine}) do
4:      if ((π3(level(lgk_q)) = π3(al_start)) ∧ (π1(typ(lgk_q)) = π1(typ(te_start)))) then
5:
6:          // (2) Suche nach gültigen Verfeinerungsrelationen
7:          for (vr_i ∈ VR_{lg_refine}) do
8:              if ((π1(π2(vr_i)) = π1(lgk_q)) ∧ (π3(level(π3(vr_i))) = (π3(al_start) + 1))) then
9:                  if (propertiesProvided(erwartet(te_start), bereitgestellt(π3(vr_i)))) then
10:
11:                     // (3) Verfeinerung kompatibel zur bisherigen Verfeinerung
12:                     if (compatibleToPrevRefinement(lg_solution, (lgk_q ↓ π3(vr_i)))) then
13:
14:                         // (4) Füge die Verfeinerung zu der aktuellen Level Graph Lösung hinzu
15:                         lg_solution.add(lgk_q ↓ π3(vr_i));
16:
17:                         // (5) Entferne das aktuelle Topologieelement aus der Warteschlange
18:                         queue_TE.remove(te_start);
19:
20:                         // (6) Verfeinere nächstes Topologieelement in der Warteschlange
21:                         if (!queue_TE ≠ ∅) then
22:                             te_next := queue_TE.next();
23:                             refinementStep(te_next, ... , lg_refine);
24:                         else
25:                             // (7) Generiere spezifisches Topologiemodell
26:                             T_solutions.add(lg_solution ⇔ t_specific);
27:                         end if
28:
29:                         // (8) Entferne die letzte Verfeinerung
30:                         lg_solution.remove(lgk_q ↓ π3(vr_i));
31:
32:                         // (9) Füge das Topologieelement wieder der Warteschlange hinzu
33:                         // und suche nach einer weiteren Lösung
34:                         queue_TE.add(te_start);
35:
36:                     end if
37:                 end if
38:             end if
39:         end for
40:     end if
41: end for
42:
```

Um die Kompatibilität zwischen der bisher verfeinerten Lösung lg_{solution} und der aktuellen Verfeinerung $lgk_q \downarrow \pi_3(vr_i)$ zu überprüfen werden die einzelnen Kompatibilitätskategorien, die im Kapitel 5.4 beschrieben und erläutert wurden in der *CompatibleToPrevRefinement* Methode, für die zuvor genannten Fälle, überprüft. Im Fall, dass die Verfeinerung kompatibel zu der bisherigen Lösung ist, werden die verfeinerten Level Graph Knoten zu der aktuellen Level Graph Lösung hinzugefügt (vgl. 6.2 (4)) und das aktuelle Topologieelement wird vorübergehend aus der Warteschlange entfernt (vgl. 6.2 (5)). Die unterschiedlichen Verfeinerungsarten, welche in einem Level Graphen vorkommen können, wurden bereits in Kapitel 5.7 im Detail aufgeführt. Anschließend wird geprüft, ob noch weitere Topologieelemente in der Warteschlange $queue_{TE}$ vorhanden sind, die bisher noch nicht verfeinert wurden. Falls noch Topologieelemente vorhanden sind, wird ein zufälliges Topologieelement aus der Warteschlange $queue_{TE}$ als nächstes Topologieelement te_{next} ausgewählt und die *RefinementStep* Methode wird rekursiv aufgerufen (vgl. 6.2 (6)).

Wenn alle Topologieelemente verfeinert wurden und die Warteschlange $queue_{TE}$ leer ist, dann wird ein spezifisches Topologiemodell t_{specific} aus der gefundenen Level Graph Lösung lg_{solution} abgeleitet und zu der Lösungsmenge $T_{\text{solutions}}$ hinzugefügt. Die Ableitungsregeln für die Ableitung von Topologie Strukturen aus einer Level Graph Struktur wurden bereits in dem Kapitel 5.5 im Detail beschrieben. Dabei wird bei der Ableitung zunächst jeder Level Graph Knoten vom Komponententyp im Lösungs-Level Graph in genau eine Komponente vom gleichen Typen in der spezifischen Topologie abgeleitet. Danach wird jeder Level Graph Knoten vom Relationstyp in jede mögliche Relation, die kompatibel zu den bereits abgeleiteten Komponenten ist, zu dem spezifischen Topologiemodell hinzugefügt. Durch diesen Vorgang wird genau ein spezifisches Topologiemodell aus einer Level Graph Struktur abgeleitet und es wird vermieden, dass unendlich viele Topologiemodelle aus einer Level Graph Struktur abgeleitet werden (vgl. Kap. 5.5). Nachdem die rekursiv aufgerufene *RefinementStep* Methode beendet wurde oder nachdem ein spezifisches Topologiemodell generiert wurde, wird die jeweilige durchgeführte Verfeinerung des aktuellen Topologieelementes te_{start} rückgängig gemacht (vgl. 6.2 (8)) und das Topologieelement wird der Warteschlange $queue_{TE}$ wieder hinzugefügt (vgl. 6.2 (9)). Dadurch kann nach weiteren Verfeinerungsmöglichkeiten gesucht werden. Der Verfeinerungsalgorithmus endet, wenn für das erste ausgewählte Topologieelement keine mögliche Verfeinerung mehr in dem Startabstraktionslevel al_{start} des Verfeinerungsmodell lg_{refine} gefunden wird.

Am Ende des Verfeinerungsalgorithmus wird die gesamte Lösungsmenge an gefundenen spezifischen Topologiemodelle $T_{\text{solutions}}$ an den Systemarchitekten zurückgegeben (vgl. List. 6.1). Die Systemarchitekten können daraufhin eine geeignete spezifische Topologie auswählen oder den Verfeinerungsalgorithmus erneut auf einem tieferen Abstraktionslevel des Verfeinerungsmodell ausführen lassen (vgl. Kap. 6.5).

6.5 Schritt 5: Auswahl einer geeigneten Topologie

Im fünften und letzten Schritt des Verfeinerungsprozesses wird eine der erstellten spezifischen Topologien durch den Systemarchitekten ausgewählt. Dieser kann letztendlich entscheiden, ob eines der generierten spezifischen Topologiemodelle für die automatische Bereitstellung und Ausführung in einer Laufzeitumgebung geeignet ist oder nicht. Wenn eine spezifische Topologie jedoch noch nicht geeignet ist, gibt es verschiedene Möglichkeiten die von einem Systemarchitekten innerhalb einer Feed-Back Loop durchgeführt werden können (vgl. Abb. 6.1):

- ***Manuelle Anpassung der spezifischen Topologie***:
  Der Systemarchitekt könnte das ausgewählte spezifische Topologiemodell selbständig manuell für den Bereitstellungsprozess fertigstellen und die fehlenden Informationen zu dem spezifischen Topologiemodell hinzufügen.
- ***Erneute Ausführung des Verfeinerungsalgorithmus***:
  Der Systemarchitekt könnte den Verfeinerungsalgorithmus erneut mit dem spezifischen Topologiemodell ausführen lassen und alternative Level Graph Modelle für die Verfeinerung verwenden. Dabei wird das spezifische Topologiemodell zu einem abstrakten Topologiemodell und es werden neue spezifische Topologiemodelle generiert.
- ***Anpassen der erwarteten Eigenschaften***:
  Der Systemarchitekt könnte die erwarteten Eigenschaften der abstrakten oder spezifischen Topologiemodelle sowie die bereitgestellten Eigenschaften von einem Level Graph Modell anpassen und optimieren. Anschließend könnte der Verfeinerungsalgorithmus erneut mit den angepassten Modellen ausgeführt werden.
- ***Anpassen der abstrakten Topologie***:
  Der Systemarchitekt könnte das zugehörige abstrakte Topologiemodell, des ausgewählten spezifischen Topologiemodell anpassen und mit den angepassten abstrakten Topologiemodell den Verfeinerungsalgorithmus erneut ausführen lassen.
- ***Optmieren des Level Graph Modell***:
  Der Systemarchitekt könnte Strukturen in den spezifischen Topologiemodellen entdecken, die es in der Realität nicht gibt. Daher müssten die verwendeten Level Graph Modelle optimiert werden, damit diese Strukturen nicht mehr durch den Level Graphen in den nächsten Verfeinerungen generiert werden können und die Genauigkeit der Level Graph Modelle somit erhöht wird.

Am Ende des letzten Schrittes des Verfeinerungsprozesses und somit auch als Endergebnis von dem Verfeinerungsprozess entsteht ein Topologiemodell, welches von dem Systemarchitekten für eine automatisierte Bereitstellung und Ausführung von Topologiemodellen in einer Laufzeitumgebung als Grundlage verwendet werden könnte.

7 Implementierung

In diesem Kapitel wird der implementierte Architecural Refinement (ArchRef) Tool Prototyp vorgestellt. In diesem Prototyp wird das zuvor erarbeitete Konzept und die Methode für eine automatisierte Verfeinerung als eine Webanwendung implementiert. Der gesamte Quellcode kann aus einem GitHub [Git] Repository, unter der folgenden Referenz [Kau17] heruntergeladen und verwendet werden. Dabei diente WINERY [Kop+13] aus dem OpenTosca Ecosystem [Bin+13] als Referenz und Vorlage für die Implementierung des ArchRef Tool Prototyps und es wurde versucht sich so nahe wie möglich an den Begrifflichkeiten der WINERY Implementierung zu orientieren. Die ArchRef Implementierung steht unter einer Apache 2.0 Lizenz [Apa] zur weiteren freien Verfügung bereit.

7.1 ArchRef Systemarchitektur

Die ArchRef Implementierung besteht aus zwei wesentlichen Komponenten, einer ArchRef Client Komponente und einer ArchRef Server Komponente, die mittels eines REST-Interface miteinander kommunizieren. Die Abbildung 7.1 gibt den Aufbau der gesamtem Systemarchitektur wieder. Die folgenden beiden Hauptbestandteile der Implementierung werden im Anschluss im Detail erläutert.

ArchRef Client Komponente:

Die ArchRef Client Komponente, die im oberen Teil der Abbildung 7.1 dargestellt ist, stellt eine Angular4 [Ang] Webanwendung kombiniert mit Bootstrap4 [MOc] Komponenten dar. Diese Webanwendung wurde mit der Verwendung von HTML5 [Htm], Typescript [Doc], SVG [W3cb] und CSS [Css] implementiert und dient als Web-Frontend Komponente. Die Client Komponente ist im Wesentlichen zuständig für die Darstellungen des ArchRef Tool in einem Browser und stellt somit eine getrennte View Komponente von den restlichen Komponenten dar. Es wird empfohlen den Chrom Browser von Google [Chr] zu verwenden, da ausschließlich dieser während der Entwicklung zum Testen verwendet wurde. Des Weiteren kann die Angular4 Anwendung in drei Kern Module unterteilt werden. Dem Topologie Modellierungstool zum Erstellen von Topologien (1), dem Level Graph Modellierungstool zum Erstellen von Level Graph Modellen (2) und dem Verwaltungstool zum Erstellen und Verwalten aller Daten in der ArchRef Anwendung (3). Diese drei Module werden in Kapitel 7.2 anhand ihrer Oberfläche

genauer vorgestellt. Die ArchRef Client Komponenten kommuniziert mittels eines REST-Interface mit der ArchRef Server Komponente. Dabei werden im Wesentlichen, die Daten mittels HTTP-Requests als JSON Objekte im Body übertragen. Ausnahme dabei ist der Import und Export von XML-Dateien, dort werden die Daten Objekte im XML-Format im Body übertragen.

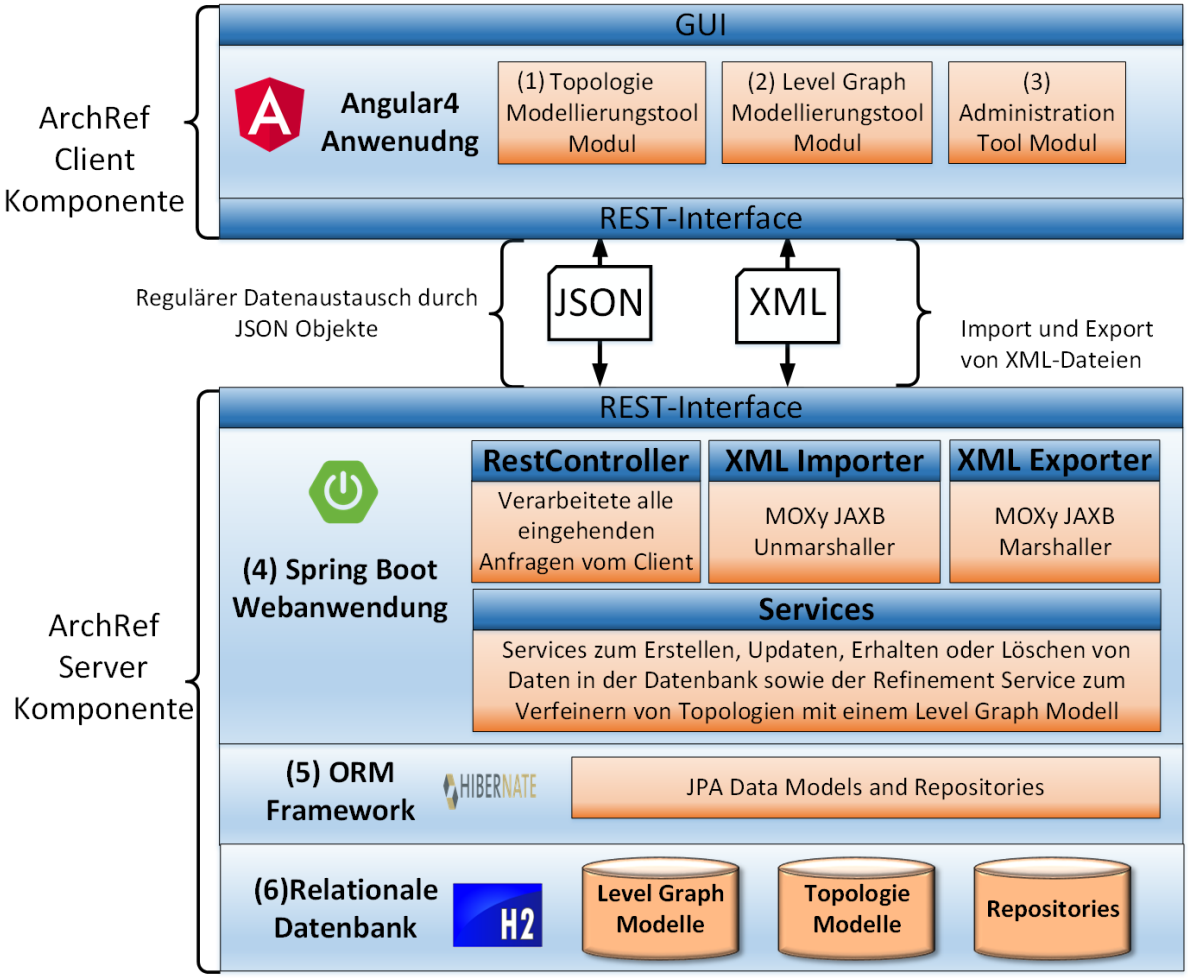

Abbildung 7.1: Systemarchitektur von ArchRef

ArchRef Server Komponente:

Die ArchRef Server Komponente, die im unteren Teil der Abbildung 7.1 abgebildet ist, hat im Wesentlichen die Aufgabe den Verfeinerungsalgorithmus auszuführen sowie die Daten der jeweiligen Modelle zu verwalten und bei Bedarf als XML-Dateien zu importieren oder zu exportieren. Dabei wird als XML Parser MOXy JAXB [Mox] verwendet, da dieser einige zusätzliche Features unterstützt die JAXB standardmäßig nicht bereitstellt, wie z.B. die Verwendung von dem Long Datentyp als ID in einer XML-Schema Definition. Die ArchRef

Server Komponente kann in drei wesentliche Bestandteile aufgeteilt werden: Eine Spring Boot Webanwendung (4), welche die RestController implementiert, die für die Verarbeitung aller eingehenden HTTP Request zuständig sind. Diese RestController rufen die jeweiligen Services auf, die im Wesentlichen für das Erstellen, Updaten, Löschen und Laden von Daten aus der Datenbank verwendet werden. Dabei wurde der Verfeinerungsalgorithmus ebenfalls als ein Service der Spring Boot Webanwendung implementiert und in der Implementierung als RefinementService bezeichnet. Zudem werden ein XML Importer und Exporter von der Spring Boot Anwendung bereitgestellt mit dem XML-Dateien importiert und exportiert werden können. Der zweite Bestandteil stellt ein Objekt Relationales Mapping Framework (5) dar, welches dazu dient die JPA Daten Modell Strukturen auf relationale Datenbanktabellen zu mappen. Dafür wurde das ORM Hibernate Framework [Hib] verwendet. Dieses Framework realisiert das Bindeglied zwischen Spring Boot Webanwendung und der H2 Datenbank [H2]. Die H2 Datenbank stellt den dritten Bestandteil der Server Komponente dar (6) und dient dazu die verschiedenen Datenobjekte, wie Level Graph Modelle, Topologie Modelle und Repositories in relationale Datenbanktabellen abzuspeichern. Repositories stellen dabei eine Sammlung von Komponententypen (vgl. 4.3) und Relationstypen (vgl. 4.5) dar.

7.2 ArchRef Web-Frontend

Das Web-Frontend von ArchRef besteht aus drei wesentlichen unterschiedlichen Modulen mit unterschiedlichen Weboberflächen, die über ein Navigationsmenü in einem Dashboard aufgerufen werden können und im Folgenden anhand von Screenshots vorgestellt werden. Dabei handelt es sich um das Administrationstool, das Level Graph Modellierungstool und das Topologie Modellierungstool.

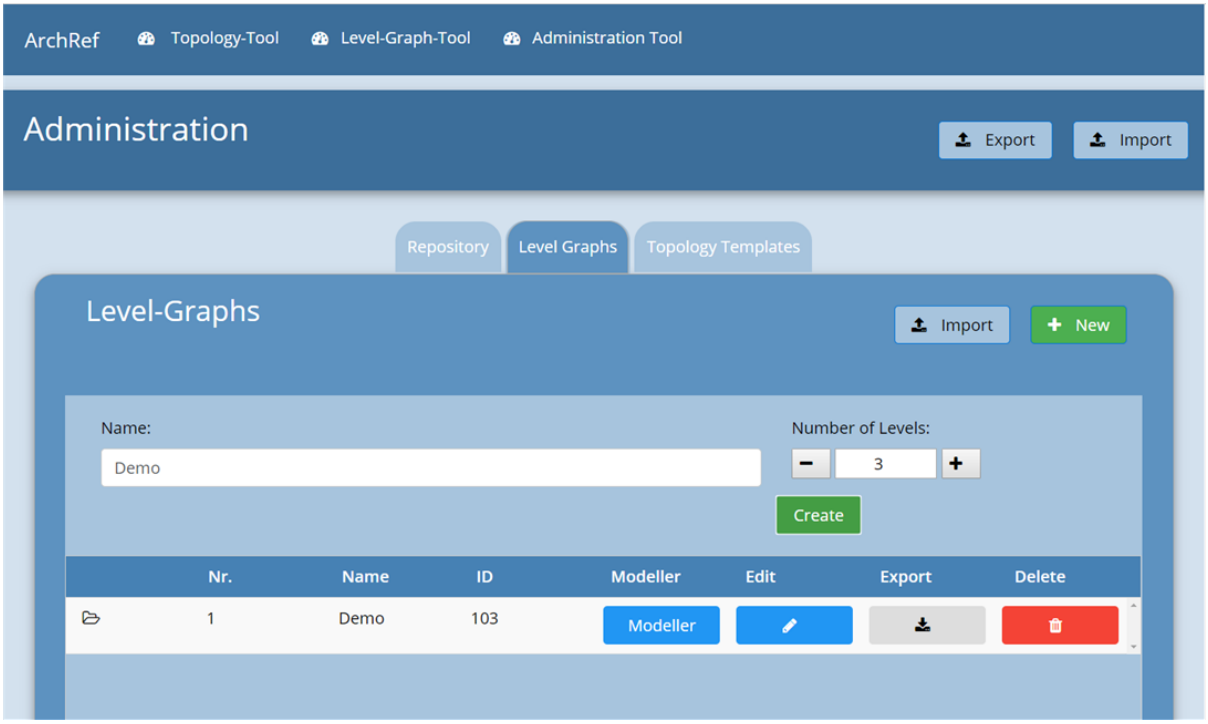

Abbildung 7.2: Web-Frontend des Administrationstool in ArchRef

In der Abbildung 7.2 ist die Administrationstool Web-Oberfläche abgebildet. In dieser ist oben die Navigationsleiste zu sehen, in der durch die verschiedenen ArchRef Tools navigiert werden kann. Zudem kann in dem Hauptbereich in der Mitte durch verschiedene Tab-Container navigiert werden und die Daten von Repositories, Level Graph Modellen und Topologie Modellen innerhalb von Datentabellen hinzugefügt, gelöscht, geändert, importiert oder exportiert werden. Zudem können auch Detailansichten aufgerufen werden indem auf die jeweilige Zeile in der Datentabelle geklickt wird. Des Weiteren können auch die entsprechenden Modellierungstools mit den jeweiligen Modellen aufgerufen werden.

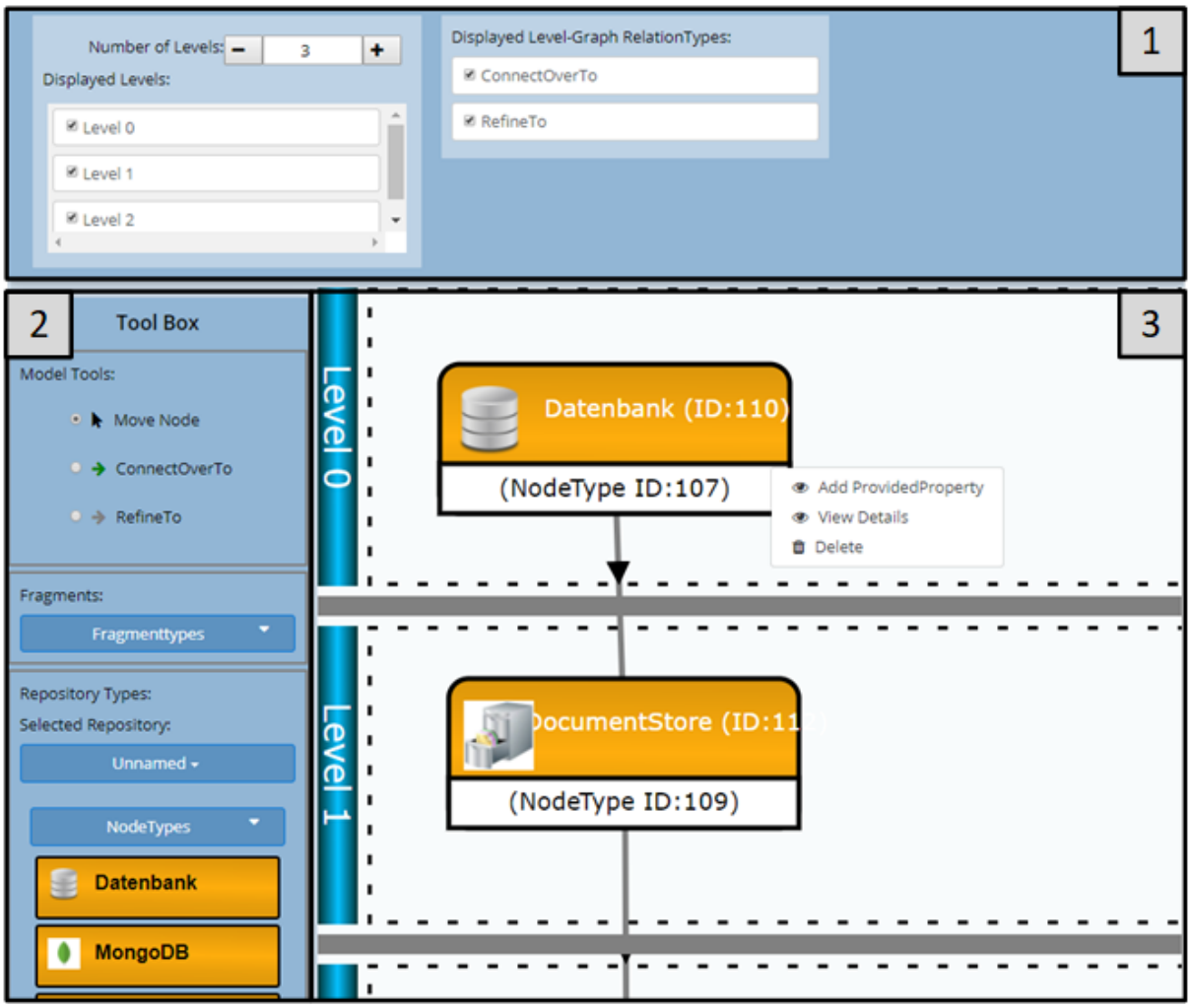

Abbildung 7.3: Web-Frontend des Level Graph Modellierungstool in ArchRef

In der Abbildung 7.3 ist die Level Graph Modellierungstool Oberfläche abgebildet. Diese lässt sich in drei wesentliche Bestandteile aufteilen. (1) Einen Filterungsbereich im oberen Teil der Oberfläche zum Ein- und Ausblenden von Abstraktionslevels sowie Verfeinerungs- und Kompatibilitätsrelationen, wobei die ConnectOverTo Relationen die Kompatibilitätsrelationen und die RefineTo Relationen die Verfeinerungsrelationen des Level Graph Modells darstellen. Zudem können zusätzliche Abstraktionslevel erstellt werden oder bestehende Abstraktionslevel gelöscht werden. (2) Der zweite Bestandteil ist eine Toolbox auf der linken Seite, in der verschiedene Komponententypen und Relationstypen aus unterschiedlichen Repositories sowie Tools zum Erstellen von Kompatibilitäts- und Verfeinerungsrelationen ausgewählt werden können. (3) Im letzten Bereich werden die Level Graph Modelle modelliert und dargestellt. Zudem können über Kontextmenüs bereitgestellte Eigenschaften zu den Level Graph Knoten hinzugefügt werden und in die Detailansicht von einzelnen Knoten und Kanten im Level Graphen gewechselt werden.

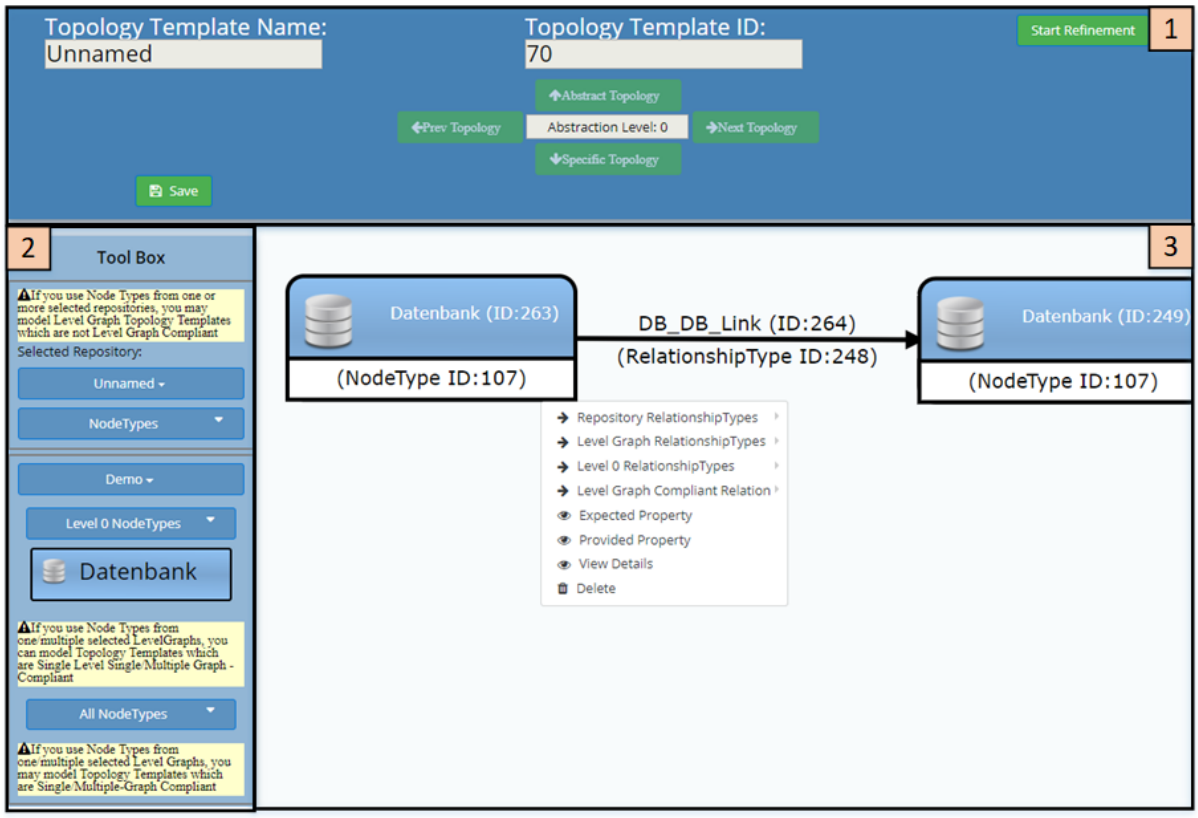

Abbildung 7.4: Web-Frontend des Topologie Modellierungstool in ArchRef

Die Topologie Modellierungstool Oberfläche lässt sich ebenfalls in drei grundsätzliche Bestandteile aufteilen. Im oberen Bereich der Abbildung 7.4 befindet sich der Navigator mit dem zwischen den abstrakten und spezifischen Topologien navigiert werden kann. Zudem kann in der oberen rechten Ecke der Verfeinerungsservice in dem Backend-Server aufgerufen werden (1). Auf der linken Seite der Abbildung befindet sich die Toolbox, in der verschiedene Komponententypen und Relationstypen aus unterschiedlichen Repositories sowie Level Graphen ausgewählt werden können (2). In dem rechten zentralen großen Hauptbereich werden die Topologie Modelle modelliert und dargestellt. Zudem können über Kontextmenüs Relationen zwischen den Komponenten von den Topologien erstellt werden (3).

7.3 ArchRef Datenstrukturen

Die Datenstrukturen von der ArchRef-Implementierung werden intern in der ArchRef Anwendung als JSON-Objekte dargestellt und verarbeitete. Diese JSON-Objekte können bei Bedarf durch einen XML-Export in separate XML-Dateien außerhalb der Anwendung exportiert werden. In dem Listing 7.1 kann, die aktuelle XML-Struktur von einem Level Graph Knoten entnommen werden. Dieser stellt einen Datenbank Knoten in einem Level Graphen mit ID 2 dar und hat eine Referenz zu dem Komponententyp mit der ID 41. Zudem stellt dieser Level Graph Knoten vier Eigenschaften bereit, wie z.B. eine Eigenschaft mit dem Namen Provider und dem Wert AmazonAws. Das Type Element ist die Bezeichnung für den Typen des Level Graph Knoten, in diesem Fall wäre der Level Graph Knoten vom NODETYPE und würde dem Komponententyp entsprechen (vgl. Kap. 5.2.3). Bei der Implementierung wurde versucht sich so nahe wie möglich an den Terminologien, der TOSCA Spezifikation [Tos] zu halten, damit eine möglicherweise spätere Integration von dem ArchRef Ansatz in TOSCA erleichtert wird [Tos].

Listing 7.1 XML Struktur von Level Graph Knoten in ArchRef

```
1
2  <LevelGraphNode name="Datenbank" id="44" levelGraphId="2" levelGraphNodeTypeId="41"
       abstractionLevelId="1">
3        <ProvidedProperties>
4              <ProvidedProperty id="3" name="Type" value="DocumentStore" id="3"
                   entityProvidedId="44"/>
5              <ProvidedProperty id="4" name="Type" value="SQL_Datenbank"
                   entityProvidedId="44"/>
6              <ProvidedProperty id="6" name="Provider" value="AmazonAws"
                   entityProvidedId="44"/>
7              <ProvidedProperty id="7" name="Provider" value="AzureMicrosoft"
                   entityProvidedId="44"/>
8        </ProvidedProperties>
9        <Type>NODETYPE</Type>
10 </LevelGraphNode>
```

8 Fazit und Ausblick

In diesem abschließenden Kapitel werden nochmals die erarbeiteten Ergebnisse nacheinander zusammengefasst und kritisch betrachtet. Anschließend wird noch ein Ausblick auf weitere mögliche Forschungen und Erweiterungen des semi-automatisierten Verfeinerungsprozess sowie Level Graph Modell angeführt.

8.1 Fazit

Aufbauend auf den erarbeiteten Grundlagen, den verwandten Ansätzen und dem angepassten Topologie Metamodell wurde ein graphbasiertes Level Graph Konzept erarbeitet, welches sich grundlegend als geeigneter Ansatz für eine automatisierte Verfeinerung von Topologiemodellen erwiesen hat. Die Level Graph Modelle weisen jedoch einige identifizierte Schwächen auf, wie z.B. in den Bereichen der N-zu-1 Verfeinerung (vlg. Kap. 5.7) und in der Ableitung von unendlichen Topologie Strukturen (vgl. Kap. 5.5 und Kap. 5.7). Diese Schwächen und Problematiken müssten genauer untersucht werden und durch geeignete Lösungsansätze bewältigt werden, damit eine vollständige automatisierte Verfeinerung ermöglicht werden kann. Des Weiteren wurde ein erster Entwurf einer Methodik vorgeschlagen, die von Systemarchitekten als ein semi-automatisierter Verfeinerungsprozess genutzt werden kann, um Topologiemodelle zu verfeinern. Dieser Verfeinerungsprozess könnte in Unternehmen in die Entwurfsphase, bei der Entwicklung von Systemarchitekturen sowie in einem Enterprise Applikation Management Prozess integriert und verwendet werden. Allerdings weist auch noch die Methodik einige Schwächen auf die beseitigt werden müssten, um eine praxisreife Verfeinerungsmethode implementieren zu können. Zum einen können in der vorgeschlagenen Methode aktuell nur Topologiemodelle mit gleichen Abstraktionsgrad erstellt und verfeinert werden (vgl. Kap. 6.2). Diese Beschränkungen müsste gelockert werden und eine Verfeinerung von gemischten Topologiemodellen sollte ermöglicht werden damit auch Verfeinerungsrelationen, die über mehrere Abstraktionslevel im Level Graphen gehen im automatisierten Verfeinerungsalgorithmus berücksichtigt und verarbeitet werden. Zum anderen werden in der Methodik aktuell nur erwartete und bereitgestellte Eigenschaften mit Name-Wert Paare, die als Strings interpretiert werden verwendet, um die Verfeinerungsmöglichkeiten zu beschränken und zu optimieren. Daher müssten Verfeinerungsmetriken anhand derer gültige Verfeinerungsbeziehungen identifiziert werden können in einer separaten Arbeit ausgearbeitet und untersucht werden sowie in das Level Graph Metamodell und dem Topologie Metamodell integriert werden. Zudem sollten auch bereits existierende Ansätze und Standards wie z.B. WS-Policy, die zur Beschreibung

von Quality of Service (QoS) von Web-Services verwendet werden in die Topologie sowie Level Graph Modelle integriert werden [Wsp]. Im letzten Abschnitt dieser Arbeit wurde das frei verfügbare ArchRef Verfeinerungstool, welches zur Erstellung von Topologie und Level Graph Modellen sowie zur Durchführung des Verfeinerungsalgorithmus genutzt werden kann, vorgestellt. ArchRef stellt einen Prototyp eines Verfeinerungstools dar mit dem das erarbeitet Konzept und die Methodik in weiteren Arbeiten im Detail getestet, analysiert und erweitert werden kann. Dabei eignet sich ArchRef aktuell um grundlegende Topologiemodelle erstellen zu können, jedoch werden Multi-Graph Topologiemodelle aktuell noch nicht optimal dargestellt. Deshalb müssten die Modellierungstools von ArchRef erweitert und verbessert werden. Des Weiteren müsste der Verfeinerungsalgorithmus für gemischte Topologiemodelle integriert und umgesetzt werden, damit auch gemischte Topologien von ArchRef verfeinert werden können. Aktuell wird eine Relationale Datenbank zum dauerhaften Speichern von Datenmodellen verwendet. Diese speichert die Daten in tabellarischen Strukturen ab. Daher wäre denkbar diese durch eine Graphdatenbank wie z.B. Neo4j [Neo] zu ersetzen, da die Topologie sowie Level Graph Modelle durch Graphstrukturen dargestellt werden. Abschließend ist das Gesamtergebnis positiv, dass mit einer graphbasierten Verfeinerungsmethodik Topologiemodelle automatisiert verfeinert werden können. Allerdings muss, damit ArchRef als produktionsreifes Tool eingesetzt werden kann, die vorherigen aufgezeigten Problematiken bewältigt und optimiert werden. Zudem müssen auch noch weitere Forschungen in den angrenzenden Bereichen des automatischen Verfeinerungsprozesses durchgeführt werden, diese werden kurz in dem nächsten Abschnitt aufgezeigt.

8.2 Ausblick

Damit der Verfeinerungsprozess anhand von Level Graph Modellen vollständig automatisiert werden kann, muss zunächst das Level Graph Modell optimiert und standardisiert werden, als Ausgangslage dafür kann das ArchRef XML-Schema sowie die ArchRef-Tool Implementierung [Kau17] verwendet werden. In einem nächsten Schritt müsste die Lücke zwischen den automatischen Verfeinerungsprozess und dem automatischen Bereitstellungs- und Ausführungsprozess von Topologiemodellen in verschiedenen Laufzeitumgebungen geschlossen werden. Dazu könnte in einem ersten Schritt die Integration von ArchRef in das OpenTOSCA Ecosystem [Bin+13] vorgenommen werden. Daher wurde versucht die Begrifflichkeiten in dem erstellten ArchRef XML-Schema so nahe wie möglich an der TOSCA Spezifikation zu halten. Ein weiterer Forschungsbereich in Bezug auf das Level Graph Modell ist, die Integration eines automatischen Managementprozesses sowie die automatisierte Optimierung von Level Graph Modellen. Dies könnte mittels einer automatisierten Feed-Back Loop und einen Registrierungsservice für Level Graph Bestandteile untersucht und integriert werden. Ein letztes Forschungsgebiet stellt die Identifikation von Mustern in Level Graph Modellen dar, um dadurch neue Entwurfsmuster für Systemarchitekturen zu entdecken. Damit stellt sich die Frage, wie kann ein Level Graph Modell nicht nur für die Verfeinerung von Modellen verwendet werden, sondern auch für den umgekehrten Prozess der Generalisierung und Abstraktion von

Modellen. Erst wenn all diese Forschungsfragen erfolgreich durch Lösungen, auf Basis des Level Graph Metamodell umgesetzt und beantwortet werden können, ist man in Richtung einer vollständigen Automatisierung von Systemarchitekturen unterwegs.

Literaturverzeichnis

[Abe13] D. Abel. *Petri-netze für Ingenieure: Modellbildung und Analyse diskret gesteuerter Systeme*. Springer-Verlag, 2013 (zitiert auf S. 28).

[Acm] *ACME Studio Tool Documentation*. URL: http://www.cs.cmu.edu/~./acme/AcmeStudio/index.html (zitiert auf S. 35).

[All07] T. Allweyer. „Erzeugung detaillierter und ausführbarer Geschäftsprozessmodelle durch Modell-zu-Modell-Transformationen." In: *EPK*. Bd. 303. 2007, S. 23–38 (zitiert auf S. 29).

[All97] R. J. Allen. *A Formal Approach to Software Architecture*. Techn. Ber. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1997 (zitiert auf S. 34).

[Amaa] (Zitiert auf S. 44).

[Amab] (Zitiert auf S. 53).

[Amac] Amazon. *WebApplicationHosting*. URL: http://media.amazonwebservices.com/architecturecenter/AWS_ac_ra_web_01.pdf (zitiert auf S. 21).

[Ang] *Angular4Dokumentation*. URL: http://devdocs.io/angular/ (zitiert auf S. 107).

[Apa] ApacheSoftwareFoundation. *Apache 2.0 License*. URL: https://www.apache.org/licenses/LICENSE-2.0 (zitiert auf S. 107).

[Ard+12] D. Ardagna, E. Di Nitto, G. Casale, D. Petcu, P. Mohagheghi, S. Mosser, P. Matthews, A. Gericke, C. Ballagny, F. D'Andria et al. „Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds". In: *Proceedings of the 4th international workshop on modeling in software engineering*. IEEE Press. 2012, S. 50–56 (zitiert auf S. 38).

[BMS13] P. Bernus, K. Mertins, G. J. Schmidt. *Handbook on architectures of information systems*. Springer Science & Business Media, 2013 (zitiert auf S. 21).

[Bal+07] D. Balasubramanian, A. Narayanan, C. van Buskirk, G. Karsai. „The graph rewriting and transformation language: GReAT". In: *Electronic Communications of the EASST* 1 (2007) (zitiert auf S. 39).

[Bal11a] H. Balzert. „Der Entwurfsprozess". In: *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. Springer, 2011, S. 481–485 (zitiert auf S. 18).

[Bal11b] H. Balzert. „Der Software-Lebenszyklus“. In: *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb* (2011), S. 1–4 (zitiert auf S. 17).

[Bau90] B. Baumgarten. „Petri-Netze“. In: *Grundlagen und Anwendungen. BI-Wissenschaftsverlag* (1990) (zitiert auf S. 35).

[Bin+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. „OpenTOSCA–a runtime for TOSCA-based cloud applications“. In: *International Conference on Service-Oriented Computing*. Springer. 2013, S. 692–695 (zitiert auf S. 107, 116).

[Bin+14] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. „TOSCA: Portable automated deployment and management of cloud applications“. In: *Advanced Web Services*. Springer, 2014, S. 527–549 (zitiert auf S. 26).

[Bre16] U. Breitenbücher. „Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements“. In: (2016) (zitiert auf S. 19, 41, 42, 44–52).

[Böh02] R. Böhm. *System-Entwicklung in der Wirtschaftsinformatik*. vdf Hochschulverlag AG, 2002 (zitiert auf S. 22, 23).

[CH03] K. Czarnecki, S. Helsen. „Classification of model transformation approaches“. In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. Bd. 45. 3. USA. 2003, S. 1–17 (zitiert auf S. 33).

[CH06] K. Czarnecki, S. Helsen. „Feature-based survey of model transformation approaches“. In: *IBM Systems Journal* 45.3 (2006), S. 621–645 (zitiert auf S. 33).

[Chr] *ChromBrowserDownloadSeite*. URL: https://www.google.de/chrome/browser/desktop/index.html (zitiert auf S. 107).

[Cle+02] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, R. Little. *Documenting software architectures: views and beyond*. Pearson Education, 2002 (zitiert auf S. 23).

[Css] *CSS*. URL: https://wiki.selfhtml.org/wiki/CSS (zitiert auf S. 107).

[DHT01] E. M. Dashofy, A. Van der Hoek, R. N. Taylor. „A highly-extensible, XML-based architecture description language“. In: *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*. IEEE. 2001, S. 103–112 (zitiert auf S. 35).

[DHT02] E. M. Dashofy, A. Van der Hoek, R. N. Taylor. „An infrastructure for the rapid development of XML-based architecture description languages“. In: *Proceedings of the 24th international conference on Software engineering*. ACM. 2002, S. 266–276 (zitiert auf S. 35).

[DN+17] E. Di Nitto, P. Matthews, D. Petcu, A. Solberg. *Model-Driven Development and Operation of Multi-Cloud Applications: The MODAClouds Approach*. Springer, 2017 (zitiert auf S. 38).

[DWC10] T. Dillon, C. Wu, E. Chang. „Cloud computing: issues and challenges“. In: *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. Ieee. 2010, S. 27–33 (zitiert auf S. 17).

[Doc] *TypeScriptDocumentation*. URL: https://www.typescriptlang.org/docs/home.html (zitiert auf S. 107).

[Dud06] W. R. Dudenredaktion. *Duden - Die deutsche Rechtschreibung. Bd. 1*. 24. völlig neu bearb. u. erw. Mannheim: Dudenverlag, Bibliographisches Institut & F.A. Brockhaus, 2006. ISBN: 978-3-411-04014-8 (zitiert auf S. 21).

[EM01] A. Egyed, N. Medvidovic. „Consistent architectural refinement and evolution using the unified modeling language“. In: *1st Workshop on Describing Software Architecture with UML, co-located with ICSE*. 2001, S. 83–87 (zitiert auf S. 17).

[ERR06] A. C. H. Ehrig, U. M. L. Ribeiro, G. Rozenberg. „Graph Transformations“. In: (2006) (zitiert auf S. 29, 39).

[FS13] O. K. Ferstl, E. J. Sinz. *Grundlagen der Wirtschaftsinformatik*. Walter de Gruyter GmbH & Co KG, 2013 (zitiert auf S. 21).

[Feh+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer Science & Business Media, 2014 (zitiert auf S. 44).

[Fow02] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002 (zitiert auf S. 22).

[GMW10] D. Garlan, R. Monroe, D. Wile. „Acme: An architecture description interchange language“. In: *CASCON First Decade High Impact Papers*. IBM Corp. 2010, S. 159–173 (zitiert auf S. 33, 35).

[GPR07] V. Gruhn, D. Pieper, C. Röttgers. *MDA®: Effektives Software-Engineering mit UML2® und EclipseTM*. Springer-Verlag, 2007 (zitiert auf S. 24, 36, 37).

[Git] *GitHubRepositories*. URL: https://github.com (zitiert auf S. 107).

[H2] *H2 Database Engine - Relational Database*. URL: http://www.h2database.com/html/main.html (zitiert auf S. 109).

[HM08] W. Hesse, H. C. Mayr. „Modellierung in der Softwaretechnik: eine Bestandsaufnahme“. In: *Informatik-Spektrum* 31.5 (2008), S. 377–393 (zitiert auf S. 33).

[HNS00] C. Hofmeister, R. Nord, D. Soni. *Applied software architecture*. Addison-Wesley Professional, 2000 (zitiert auf S. 22).

[HS04] C. Richter-von Hagen, W. Stucky. *Business-Process-und Workflow-Management*. Springer-Verlag, 2004 (zitiert auf S. 35).

[Hib] *Hibernate Object Relational Mapping Framework*. URL: http://hibernate.org/orm/ (zitiert auf S. 109).

[Hil00] R. Hilliard. „Ieee-std-1471-2000 recommended practice for architectural description of software-intensive systems“. In: *IEEE, http://standards. ieee. org* 12.16-20 (2000), S. 2000 (zitiert auf S. 22, 23).

[Htm] *HTML 5 Dokumentation*. URL: https://www.w3.org/TR/html5/ (zitiert auf S. 107).

[Háj98] P. Hájek. *Metamathematics of fuzzy logic*. Bd. 4. Springer Science & Business Media, 1998 (zitiert auf S. 27).

[JK05] F. Jouault, I. Kurtev. „Transforming models with ATL“. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2005, S. 128–138 (zitiert auf S. 39).

[Jou+08] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev. „ATL: A model transformation tool“. In: *Science of computer programming* 72.1 (2008), S. 31–39 (zitiert auf S. 30, 39).

[KBC05] A. Kalnins, J. Barzdins, E. Celms. „Model transformation language MOLA“. In: *Model Driven Architecture*. Springer, 2005, S. 62–76 (zitiert auf S. 38, 39).

[KWB03] A. G. Kleppe, J. B. Warmer, W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003 (zitiert auf S. 26).

[Kar+03] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle. „On the use of graph transformation in the formal specification of model interpreters“. In: *J. UCS* 9.11 (2003), S. 1296–1321 (zitiert auf S. 39).

[Kau17] A. Kaul. *kaular/ArchRef*. 2017. URL: https://github.com/kaular/ArchRef (zitiert auf S. 35, 107, 116).

[Kel02] W. Keller. „Enterprise Application Integration“. In: *Erfahrungen aus der Praxis. dpunkt* (2002) (zitiert auf S. 21).

[Kop+13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „Winery–a modeling tool for TOSCA-based cloud applications“. In: *International Conference on Service-Oriented Computing*. Springer. 2013, S. 700–704 (zitiert auf S. 26, 107).

[LNZ04] Y. Liu, A. H. Ngu, L. Z. Zeng. „QoS computation and policing in dynamic web service selection“. In: *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. ACM. 2004, S. 66–73 (zitiert auf S. 18).

[Lea09] N. Leavitt. „Is cloud computing really ready for prime time“. In: *Growth* 27.5 (2009), S. 15–20 (zitiert auf S. 17).

[Li+10] A. Li, X. Yang, S. Kandula, M. Zhang. „CloudCmp: comparing public cloud providers“. In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM. 2010, S. 1–14 (zitiert auf S. 17).

[MDA08] O. MDA. *Object Management Group Model Driven Architecture*. 2008 (zitiert auf S. 36, 37).

[MMB+03] J. Miller, J. Mukerji, M Belaunde et al. „MDA guide“. In: *Object Management Group* (2003) (zitiert auf S. 26, 27).

[MOF02] O. MOF. *Meta Object Facility (MOF) Specification v1. 4*. 2002 (zitiert auf S. 28, 34, 36).

[MOc] J. T. Mark Otto, B. contributors. *Bootstrap*. URL: https://v4-alpha.getbootstrap.com/ (zitiert auf S. 107).

[MT00] N. Medvidovic, R. N. Taylor. „A classification and comparison framework for software architecture description languages“. In: *IEEE Transactions on software engineering* 26.1 (2000), S. 70–93 (zitiert auf S. 34).

[MT10] N. Medvidovic, R. N. Taylor. „Software architecture: foundations, theory, and practice“. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM. 2010, S. 471–472 (zitiert auf S. 24).

[MVG06] T. Mens, P. Van Gorp. „A taxonomy of model transformation“. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), S. 125–142 (zitiert auf S. 29–31).

[MW83] I. Merriam-Webster. *Webster's ninth new collegiate dictionary*. Merriam-Webster, 1983 (zitiert auf S. 21).

[Mac+06] R. Machado, J. Fernandes, P. Monteiro, H. Rodrigues. „Refinement of software architectures by recursive model transformations“. In: *Product-Focused Software Process Improvement* (2006), S. 422–428 (zitiert auf S. 17).

[Men+06] T. Mens, P. Van Gorp, D. Varró, G. Karsai. „Applying a model transformation taxonomy to graph transformation technology“. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), S. 143–159 (zitiert auf S. 30, 31).

[Mod] *MODAClouds*. URL: http://www.modaclouds.eu/ (zitiert auf S. 38).

[Mox] *MOXy*. URL: https://wiki.eclipse.org/EclipseLink/Examples/MOXy (zitiert auf S. 108).

[Nel09] M. R. Nelson. „Building an open cloud“. In: *Science* 324.5935 (2009), S. 1656–1657 (zitiert auf S. 17).

[Neo] *Neo4j*. URL: https://neo4j.com/ (zitiert auf S. 116).

[Obj] *UML-Unified Modelling Language Specification*. URL: http://www.omg.org/spec/UML (zitiert auf S. 28, 34, 36).

[Oqu04] F. Oquendo. „π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures“. In: *ACM SIGSOFT Software Engineering Notes* 29.3 (2004), S. 1–14 (zitiert auf S. 33).

[Poo+02] J. Poole, D. Chang, D. Tolbert, D. Mellor. *Common warehouse metamodel*. Bd. 20. John Wiley & Sons, 2002 (zitiert auf S. 36).

[RH06] R. Reussner, W. Hasselbring. *Handbuch der Software-Architektur*. dpunkt Heidelberg, 2006 (zitiert auf S. 21).

[RJB04] J. Rumbaugh, I. Jacobson, G. Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004 (zitiert auf S. 23, 24, 34).

[SB94] A. Stephan, A. Beckermann. „Emergenz“. In: *Information Philosophie* 1994.3 (1994) (zitiert auf S. 17).

[SK03] S. Sendall, W. Kozaczynski. „Model transformation: the heart and soul of model-driven software development“. In: *IEEE Software* 20.5 (2003), S. 42–45. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231150 (zitiert auf S. 29).

[SNH95] D. Soni, R. L. Nord, C. Hofmeister. „Software architecture in industrial applications“. In: *Software Engineering, 1995. ICSE 1995. 17th International Conference on*. IEEE. 1995, S. 196–196 (zitiert auf S. 24, 25).

[San93] D. Sangiorgi. „Expressing mobility in process algebras: first-order and higher-order paradigms“. In: (1993) (zitiert auf S. 33).

[Sei03] E. Seidewitz. „What models mean“. In: *IEEE software* 20.5 (2003), S. 26–32 (zitiert auf S. 26).

[Sel03] B. Selic. „The pragmatics of model-driven development“. In: *IEEE software* 20.5 (2003), S. 19–25 (zitiert auf S. 26).

[Sou66] J. Soubiran. „Vitruv. Zehn Bücher über Architektur übersetzt und mit Anmerkungen versehen von Dr. Curt Fensterbusch“. In: (1966) (zitiert auf S. 21).

[Sta78] P. H. Starke. *Petri-Netze*. Akademie der Wissenschaften der DDR, Zentralinstitut für Kybernetik und Informationsprozesse, 1978 (zitiert auf S. 35).

[Tar35] A. Tarski. „Zur Grundlegung der Boole'schen Algebra. I“. In: *Fundamenta mathematicae* 24 (1935), S. 177–198 (zitiert auf S. 28).

[Tos] *Topology and Orchestration Specification for Cloud Applications Version 1.0. 25 November 2013*. 2013. URL: http://docs.oasis-open.org/tosca (zitiert auf S. 26, 113).

[VL03] A. Van Lamsweerde. „From system goals to software architecture“. In: *Formal Methods for Software Architectures* (2003), S. 25–43 (zitiert auf S. 17).

[Vos00] G. Vossen. *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme*. Bd. 1. Oldenbourg München, 2000 (zitiert auf S. 23, 28).

[W3ca] *Extensible Markup Language (XML)*. URL: https://www.w3.org/XML/ (zitiert auf S. 33).

[W3cb] *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. URL: http://www.w3.org/TR/SVG11/ (zitiert auf S. 107).

[Web] *Chapter 4 Design Guidelines for Secure Web Applications*. URL: https://msdn.microsoft.com/en-us/library/ff648647.aspx (zitiert auf S. 21).

[Wes+01] D. B. West et al. *Introduction to graph theory*. Bd. 2. Prentice hall Upper Saddle River, 2001 (zitiert auf S. 28).

[Wsp] *WS-Policy*. URL: https://www.w3.org/Submission/WS-Policy/ (zitiert auf S. 116).

[Xad] *xADL - A Highly Extensible Architecture Description Language for Software and Systems*. URL: http://isr.uci.edu/projects/xarchuci/ (zitiert auf S. 35).

[Xmi] *Documents Associated with XML Metadata Interchange™ (XMI®) Version 2.5.1*. URL: http://www.omg.org/spec/XMI/2.5.1/ (zitiert auf S. 36).

[Zim+15] O. Zimmermann, L. Wegmann, H. Koziolek, T. Goldschmidt. „Architectural decision guidance across projects-problem space modeling, decision backlog management and cloud computing knowledge“. In: *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*. IEEE. 2015, S. 85–94 (zitiert auf S. 40).

[ÁES01] J. Álvarez, A. Evans, P. Sammut. „Mapping between levels in the metamodel architecture“. In: *«UML» 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools* (2001), S. 34–46 (zitiert auf S. 27).

Alle URLs wurden zuletzt am 25. 09. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift