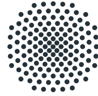

Eine Methode zum Verteilen, Adaptieren und Deployment partnerübergreifender Anwendungen

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Karoline Wild

aus Gießen

Hauptberichter:	Prof. Dr. Dr. h. c. Frank Leymann
Mitberichter:	Univ. Prof. Dr. Schahram Dustdar
Tag der mündlichen Prüfung:	14. Dezember 2022

Institut für Architektur von Anwendungssystemen

2022

Inhaltsverzeichnis

1 Einleitung **11**
1.1 Vision der Arbeit 14
1.2 Problemstellung und Forschungsbeiträge 16
1.3 Veröffentlichungen im Rahmen der Arbeit 22
1.4 Aufbau der Arbeit 26

2 Grundlagen des Deployments und verwandte Arbeiten **27**
2.1 Modellierung adaptiver Deployment-Modelle 28
2.2 Bereitstellung von Anwendungen 48
2.3 Anwendbarkeit und Anwendung von Mustern 56
2.4 Zusammenfassung und Forschungsfragen dieser Arbeit 73

3 DIVA-Methode für das Deployment partnerübergreifender Anwendungen **75**
3.1 Anforderungen an das partnerübergreifende Deployment 76
3.2 Die DIVA-Methode 78
3.3 Erweiterung der DIVA-Methode zur rekursiven Anwendung . . 87
3.4 Umsetzung der Anforderungen und Diskussion 89

4 Modellieren und Verteilen adaptiver Deployment-Modelle 93
4.1 Essential Deployment Meta Model (EDMM) 94
4.2 Erweiterungen von EDMM 99
4.3 Modellierung von globalen Deployment-Modellen 105
4.4 Verteilung von lokalen Deployment-Modellen 109
4.5 Zusammenfassung und Diskussion 124

5 Musterbasierte Problemerkennung und Modelladaption 127
5.1 Idee und Grundkonzept 128
5.2 Anwendung von Mustern in Deployment-Modellen 129
5.3 Muster zur Problemerkennung in Deployment-Modellen 131
5.4 Anwendung von Musterlösungen zur Modelladaption 137
5.5 Adapter zur Anpassung der Kommunikationsprotokolle 142
5.6 Zusammenfassung und Diskussion 148

6 Automatisiertes partnerübergreifendes dezentrales Deployment 155
6.1 Idee und Grundkonzept 156
6.2 Generierung lokaler Workflows zur Choreographie 157
6.3 Zusammenfassung und Diskussion 166

7 Architektur und Validierung des DIVA-Werkzeugs 169
7.1 Architektur des DIVA-Werkzeugs 170
7.2 Prototypische Implementierung des DIVA-Werkzeugs 177
7.3 Validierung und Evaluation 184
7.4 Integration mit anderen Systemen 193

8 Zusammenfassung und Ausblick 195
8.1 Zusammenfassung der Forschungsbeiträge 196
8.2 Ausblick 200

Literaturverzeichnis 201

Abbildungsverzeichnis 231

Tabellenverzeichnis 235

Zusammenfassung

Ein wesentlicher Aspekt einer effektiven Kollaboration innerhalb von Organisationen, aber vor allem organisationsübergreifend, ist die Integration und Automatisierung der Prozesse. Dazu zählt auch die Bereitstellung von Anwendungssystemen, deren Komponenten von unterschiedlichen Partnern, das heißt Abteilungen oder Unternehmen, bereitgestellt und verwaltet werden. Die dadurch entstehende verteilte, dezentral verwaltete Umgebung bedarf neuer Konzepte zur Bereitstellung. Die Autonomie der Partner und die Verteilung der Komponenten führen dabei zu neuen Herausforderungen. Zum einen müssen partnerübergreifende Kommunikationsbeziehungen realisiert und zum anderen muss das automatisierte dezentrale Deployment ermöglicht werden. Eine Vielzahl von Technologien wurde in den letzten Jahren entwickelt, die alle Schritte von der Modellierung bis zur Bereitstellung und dem Management zur Laufzeit einer Anwendung abdecken. Diese Technologien basieren jedoch auf einer zentralisierten Koordination des Deployments, wodurch die Autonomie der Partner eingeschränkt ist. Auch fehlen Konzepte zur Identifikation von Problemen, die aus der Verteilung von Anwendungskomponenten resultieren und die Funktionsfähigkeit der Anwendung einschränken. Dies betrifft speziell die partnerübergreifenden Kommunikationsbeziehungen.

Um diese Herausforderungen zu lösen, stellt diese Arbeit die DivA-Methode

zum Verteilen, Adaptieren und Deployment partnerübergreifender Anwendungen vor. Die Methode vereinigt die globalen und lokalen Partneraktivitäten, die zur Bereitstellung partnerübergreifender Anwendungen benötigt werden. Dabei setzt die Methode auf dem deklarativen Essential Deployment Meta Model (EDMM) auf und ermöglicht damit die Einführung deploymenttechnologieunabhängiger Modellierungskonzepte zur Verteilung von Anwendungskomponenten sowie zur Modellanalyse und -adaption. Das Split-and-Match-Verfahren wird für die Verteilung von Anwendungskomponenten basierend auf festgelegten Zielumgebungen und zur Selektion kompatibler Cloud-Dienste vorgestellt. Für die Ausführung des Deployments können EDMM-Modelle in unterschiedliche Technologien transformiert werden. Um die Bereitstellung komplett dezentral durchzuführen, werden deklarative und imperative Technologien kombiniert und basierend auf den deklarativen EDMM-Modellen Workflows generiert, die die Aktivitäten zur Bereitstellung und zum Datenaustausch mit anderen Partnern zur Realisierung partnerübergreifender Kommunikationsbeziehungen orchestrieren. Diese Workflows formen implizit eine Deployment-Choreographie.

Für die Modellanalyse und -adaption wird als Kern dieser Arbeit ein zweistufiges musterbasiertes Verfahren zur Problemerkennung und Modelladaption eingeführt. Dafür werden aus den textuellen Musterbeschreibungen die Problem- und Kontextdefinition analysiert und formalisiert, um die automatisierte Identifikation von Problemen in EDMM-Modellen zu ermöglichen. Besonderer Fokus liegt dabei auf Problemen, die durch die Verteilung der Komponenten entstehen und die Realisierung von Kommunikationsbeziehungen verhindern. Das gleiche Verfahren wird auch für die Selektion geeigneter konkreter Lösungsimplementierungen zur Behebung der Probleme angewendet. Zusätzlich wird ein Ansatz zur Selektion von Kommunikationstreibern abhängig von der verwendeten Integrations-Middleware vorgestellt, wodurch die Portabilität von Anwendungskomponenten verbessert werden kann.

Die in dieser Arbeit vorgestellten Konzepte werden durch das DivA-Werkzeug automatisiert. Zur Validierung wird das Werkzeug prototypisch implementiert und in bestehende Systeme zur Modellierung und Ausführung des Deployments von Anwendungssystemen integriert.

Abstract

An essential aspect of effective collaboration within organizations, but especially across organizations, is the integration and automation of processes. This also includes the deployment of application systems whose components are provided and managed by different partners, i.e., departments or companies. The resulting distributed, decentrally managed environment requires new concepts for deployment. The autonomy of the partners and the distribution of the components lead thereby to new challenges. On the one hand, cross-partner communication relations must be realized, and on the other hand, automated decentralized deployment must be enabled. A variety of technologies have been developed in recent years that cover all steps from modeling to deployment and management at runtime of applications. However, these are based on centralized coordination of the deployment, which restricts the autonomy of the partners. There is also a lack of concepts for identifying problems resulting from the distribution of application components that impede the functionality of the application. This specifically affects cross-partner communication relations.

To address these challenges, this thesis presents the DivA-method for distributing, adapting, and deploying cross-partner applications. The method combines the global and local activities of the involved partners required to deploy cross-partner applications. The method is based on the declarative

Essential Deployment Meta Model (EDMM) and, thus, enables the introduction of deployment technology-independent modeling concepts for the distribution of application components as well as for model analysis and adaptation. For redistributing an application, the SPLIT-AND-MATCH method is presented, which distributes application components based on specified target environments and selects compatible cloud services for deployment. To actually execute the deployment, EDMM-models can be transformed into artifacts of different technologies. To perform deployment in a completely decentralized manner, declarative and imperative technologies are combined and, based on the declarative EDMM-models, workflows are generated that orchestrate deployment activities and data exchange with other partners to realize cross-partner communication relations. These workflows implicitly form a deployment choreography.

For model analysis and adaptation, a two-step pattern-based problem detection and model adaptation procedure is introduced as the core of this work. For this purpose, problem descriptions and the context in which the problem occurs are analyzed and formalized from patterns to enable automated identification of problems in EDMM-models. Special focus is given to problems that arise due to the distribution of components, preventing the realization of communication relations. The same procedure is applied to the selection of appropriate concrete solution implementations to solve the problems. In addition, an approach for selecting communication drivers depending on the integration middleware used is presented, which can improve the portability of application components.

The concepts presented in this thesis are automated by the DIVA-tool. For validation, the tool is implemented prototypically and integrated into existing systems for modeling and executing the deployment of application systems.

Danksagungen

Ich möchte mich bei all jenen bedanken, die mich in den letzten Jahren während der Arbeit an dieser Dissertation begleitet haben. Mein ganz besonderer Dank gilt meinem Doktorvater Prof. Dr. Dr. h. c. Frank Leymann, der mich ermutigt hat den Pfad der Wissenschaft einzuschlagen und mich auf meinem Weg stets bestärkt und unterstützt hat. Darüber hinaus möchte ich mich bei Univ. Prof. Dr. Schahram Dustdar für die Übernahme des Mitberichts bedanken. Mein Dank gilt auch allen Kolleginnen und Kollegen für den wissenschaftlichen Austausch, die anregenden Diskussionen und die vielseitige Unterstützung. Ein besonderes Dankeschön möchte ich dabei sowohl an Dr. Uwe Breitenbücher und Dr. Oliver Kopp richten, die mir von Beginn an als Mentoren zur Seite standen, als auch an Michael Falkenthal, Lukas Harzenetter und Michael Wurster, die als Kollegen und Freunde immer ein offenes Ohr für mich hatten.

Meinen Eltern und meinem Bruder danke ich für den Zuspruch und die Ratschläge, mit denen sie mich schon mein ganzes Leben, aber vor allem in der Zeit der Arbeit an dieser Dissertation, mental unterstützt haben. Ganz besonders danke ich meinem Mann Benedikt für seine unermüdliche Geduld, sein Verständnis und seine Unterstützung. Zuletzt möchte ich meiner Tochter danken, die mir mit ihrer Ankunft den nötigen Anreiz zur zeitnahen Fertigstellung dieser Arbeit gegeben hat.

Kapitel 1

Einleitung

Ein wesentlicher Erfolgsfaktor heutiger Unternehmen ist die effektive Kollaboration - innerhalb der eigenen Organisation, aber vor allem auch über Organisationsgrenzen hinweg. Im Rahmen von Initiativen wie Industrie 4.0 oder Industrial Internet wurde in den vergangenen Jahren vermehrt der Fokus auf die vertikale und horizontale Integration von Prozessen und IT-Systemen gelegt. Während sich die vertikale Integration vor allem mit der Verknüpfung der digitalen und physischen Prozesse innerhalb einer Organisation befasst, ermöglicht die horizontale Integration die Kollaboration entlang der gesamten Wertschöpfungskette. Dabei spielen Interoperabilität, Flexibilität und das dezentrale Management eine entscheidende Rolle [CFA17]. Anwendungssysteme in solch verteilten, dezentral verwalteten Umgebungen integrieren typischerweise Komponenten, die von unterschiedlichen Partnern bereitgestellt werden. Die Partner sind autonome Organisationseinheiten und können, beispielsweise, verschiedene Unternehmen, aber auch unterschiedliche Bereiche innerhalb eines Unternehmens sein. Dies kann in verteilten komplexen Anwendungen resultieren, bei deren Management unterschiedliche Partner involviert sind und verschiedene Cloud-Dienste und Laufzeitumgebungen genutzt werden.

Die Verteilung der Komponenten einer Anwendung spielt vor allem beim Deployment eine wichtige Rolle. Sowohl die organisatorische Zuweisung zu Partnern als auch die Verteilung der Komponenten auf die verfügbaren (Cloud-)Ressourcen eines Partners müssen berücksichtigt werden. Dabei erhöht der Grad der Verteilung auch die Komplexität der Kommunikation zwischen den Komponenten. Unter anderem müssen Beschränkungen für den Zugriff von außen auf bestimmte Umgebungen oder Sicherheitsanforderungen bei der Realisierung der Kommunikationsbeziehungen berücksichtigt werden [HLB19]. Werden diese Probleme, welche durch die Verteilung der Komponenten entstehen, nicht bereits vor dem Deployment erkannt, ist das Terminieren, Adaptieren und erneute Deployment der Anwendung nötig, was einen hohen Aufwand bei allen beteiligten Partnern zur Folge hat. Jedoch ist nicht nur die Implementierung und Konfiguration der Anwendung eine Herausforderung, sondern auch die Ausführung des Deployments durch die Partner. Jeder Partner kontrolliert seine Ressourcen und das Deployment der dort bereitgestellten Komponenten. Mit gängigen zentralisierten Deployment-Technologien ist damit kein automatisiertes Deployment der gesamten Anwendung möglich, ohne die Autonomie der Partner einzuschränken. Das Deployment durch jeden Partner und der Informationsaustausch, um die Kommunikationsverbindungen zwischen den Komponenten, die von unterschiedlichen Partnern bereitgestellt werden, herstellen zu können, müssen manuell koordiniert werden. Dabei müssen sowohl die Provisionierungsreihenfolge der Komponenten berücksichtigt als auch benötigte Informationen wie Hostnamen oder Zugangsdaten ausgetauscht werden.

Die zentralen Herausforderungen für das Deployment partnerübergreifender Anwendungen sind daher (i) das Realisieren *partnerübergreifender Kommunikationsbeziehungen* sowie (ii) die *automatisierte dezentrale Ausführung des Deployments*. Für das Modellieren und das automatisierte Ausführen des Deployments wurde eine Vielzahl von Technologien entwickelt. Dazu gehören unter anderem Konfigurationsmanagementsysteme wie Chef [Che22], Systeme zum Infrastrukturmanagement wie Terraform [Has22], zur Verwaltung von Container-Anwendungen wie Kubernetes [The22b] oder cloudanbieterspezifische Tools wie CloudFormation [Ama22b] für Amazon Web

Services (AWS), die alle, trotz unterschiedlicher Schwerpunkte beim Funktionsumfang, auf den gleichen Modellierungsgrundsätzen beruhen [WBF+19]. Alle Technologien folgen einem zentralisierten Ansatz beim Deployment, das heißt, auch bei Technologien mit einer agentenbasierten Architektur, bei denen die Agenten die Deployment-Operationen ausführen und die entsprechenden Ressourcen verwalten, wird die Verteilung der Deployment-Aufgaben zentral koordiniert [WBB+17]. Für eine zentrale Koordination muss Zugriff auf interne APIs gewährt werden, und Details über das Deployment, die auszuführenden Operationen und die Struktur der Anwendung inklusive der verwendeten Middleware und Infrastruktur müssen offengelegt werden [KB17]. Solche Informationen werden jedoch typischerweise von Unternehmen nicht weitergegeben, und auch der Zugriff und damit die Kontrolle über interne Deployment-Prozesse werden nicht Dritten überlassen [HLB19]. Der gesamte Koordinationsprozess zum Deployment partnerübergreifender Anwendungen verläuft daher manuell: Die Provisionierungsreihenfolge muss festgelegt, Deployment-Informationen mit Partnern, welche diese zum Kommunikationsaufbau benötigen, müssen ausgetauscht und Integrationsprobleme bei partnerübergreifenden Kommunikationsbeziehungen erkannt und behoben werden.

Um Integrationsprobleme schon vor dem Deployment erkennen zu können bedarf es ein hohes Maß an technischem Know-how und Erfahrung, da die Implementierung der Anwendung, die Daten, welche ausgetauscht werden sollen, und die Umgebung in der die Komponenten bereitgestellt werden sollen, dabei entscheidende Rollen spielen. Es gibt eine Reihe von Arbeiten, die bewährte Lösungen von wiederkehrenden Problemen im Kontext der Realisierung und des Managements von Anwendungssystemen als *Muster* (engl. *„patterns“*) beschreiben [FLR+14; HW04; SFH+06]. Muster dokumentieren zwar bewährte Lösungen unabhängig von konkreten Technologien oder Programmiersprachen, jedoch gibt es Arbeiten, die auf Basis der abstrakten Lösungen die automatisierte Anwendung dieser Muster für eine bestimmte Domäne ermöglichen [Bre16; FCSK03; GL19]. Das Erkennen von Problemen bedarf jedoch immer noch Expertenwissens und kann zu einem hohen Aufwand bei der Realisierung von Anwendungssystemen führen.

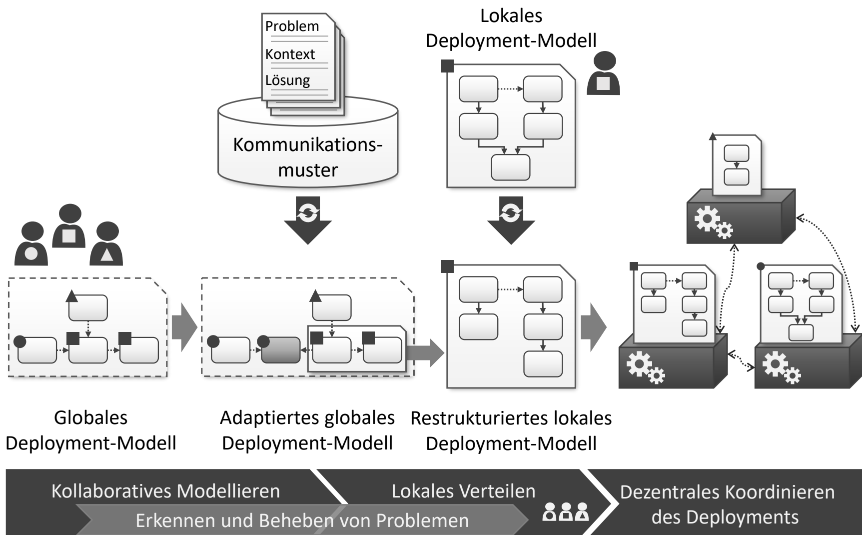

Abbildung 1.1: Vision des partnerübergreifenden Deployments.

1.1 Vision der Arbeit

Die Vision dieser Arbeit ist es, die zentralen Herausforderungen der Realisierung partnerübergreifender Kommunikationsbeziehungen und des automatisierten dezentralen Deployments von Anwendungen durch ein Konzept zum Modellieren, Verteilen, Adaptieren und Deployment partnerübergreifender Anwendungen zu lösen. Abbildung 1.1 veranschaulicht diese Vision: Die anwendungsspezifischen Komponenten und deren Beziehungen werden in einem kollaborativen Prozess modelliert und spezifizieren das *globale Deployment-Modell*, welches global verfügbare Informationen, wie die Zuordnung der Komponenten zu den jeweiligen Partnern sowie Parameter, die zur Erstellung der modellierten Kommunikationsbeziehungen benötigt werden, beinhaltet. Auf Basis bekannter Kommunikationsmuster können dann Probleme in dem globalen Deployment-Modell automatisiert erkannt und durch existierende Lösungsfragmente behoben werden. Das Modell

wird entsprechend adaptiert, wodurch auch strukturelle Änderungen durch zusätzlich benötigte Komponenten möglich sind. Im *lokalen Deployment-Modell* jedes Partners sind alle Konfigurations- und Implementierungsdetails enthalten, die für das Deployment des spezifizierten Teils der Anwendung benötigt werden. Die lokalen Deployment-Modelle werden aus dem globalen Deployment-Modell generiert und durch die entsprechenden partnerspezifischen Details ergänzt. Häufig stehen aus der Entwicklungs- und Testphase der Anwendung bereits Deployment-Modelle zur Verfügung, welche als Basis für die ressourcenspezifische Verteilung auf die lokal verfügbare Produktivumgebung zur Bereitstellung der Anwendungskomponenten genutzt werden können. Das *restrukturierte lokale Deployment-Modell* kann dann lokal verarbeitet und ausgeführt werden, wobei die Provisionierungsreihenfolge und der Informationsaustausch automatisiert global koordiniert werden, ohne Dritten Zugriff auf die Bereitstellungsumgebung zu geben. Durch diesen dezentralen Ansatz müssen nur notwendige Informationen mit anderen Partner geteilt werden, und die Kontrolle über Ausführung und Anpassung des Deployments bleibt im Verantwortungsbereich des jeweiligen Partners.

Die vorgestellte Vision ermöglicht es in einem kollaborativen Umfeld mit mehreren Partnern verteilte Anwendungssysteme vollautomatisiert bereitstellen zu können und dabei frühzeitig Probleme bei der Realisierung partnerübergreifender Anwendungen zu erkennen und zu beheben sowie die Kontrolle über die Durchführung des Deployments beim jeweiligen Partner zu belassen und trotzdem global die gesamte Bereitstellung automatisiert zu koordinieren. Dabei können auch neue Erkenntnisse im Bereich der Kommunikationsmuster erfasst und genutzt werden, um die Erkennung und Behebung von möglichen Problemen stetig zu verbessern. Da in diesem kollaborativen Prozess unterschiedliche Deployment-Technologien von den verschiedenen Partnern präferiert werden können, wird ein technologieunabhängiges Deployment-Metamodell als Fundament genutzt. Damit wird die Beschreibung des Deployments mit einem kanonischen Metamodell ermöglicht, welches dann für jeden Partner in die Artefakte der jeweils präferierten Deployment-Technologie transformiert werden kann. Dadurch können bereits von den Partnern genutzte Technologien verwendet werden.

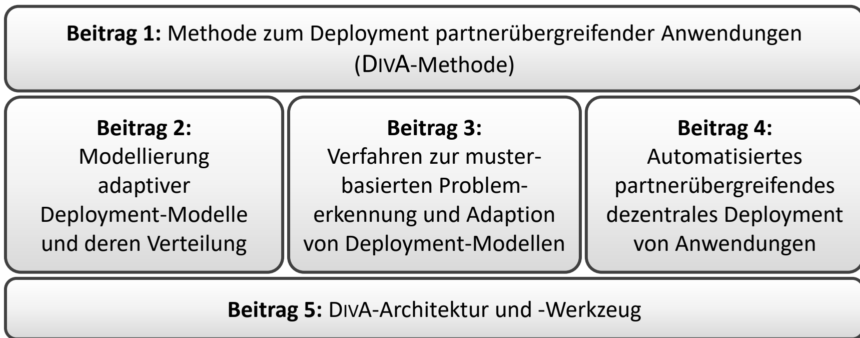

Abbildung 1.2: Überblick der Forschungsbeiträge dieser Arbeit.

1.2 Problemstellung und Forschungsbeiträge

Die in der Vision enthaltenen angestrebten Forschungsbeiträge und Problemstellungen werden im Folgenden erläutert. Abbildung 1.2 gibt einen Überblick über die Forschungsbeiträge dieser Arbeit und deren Zusammenhänge. Der erste Beitrag ist die DIVA-Methode („*Divide and Adapt*“-Methode), welche durch die Beiträge 2 bis 4 umgesetzt wird und durch das DIVA-Werkzeug in Beitrag 5 implementiert wird. Beitrag 2 präsentiert eine Metamodellerweiterung zur Modellierung globaler und lokaler Deployment-Modelle und zur automatisierten Verteilung dieser Modelle. Beitrag 3 stellt ein Verfahren zum Erkennen und Beheben von Problemen basierend auf Kommunikationsmustern vor. Das Verfahren kann dabei auch unabhängig von der vorgestellten Methode generell für das Erkennen von Problemen basierend auf Mustern in Deployment-Modellen genutzt werden und ist nicht auf Kommunikationsmuster beschränkt. Des Weiteren kann das Verfahren auch für die Anwendung auf andere Architektursichten außer der Deployment-Sicht angepasst werden. Beitrag 4 führt einen choreographiebasierten Prozess zum Deployment von partnerübergreifenden Anwendungen durch unabhängige dezentrale Deployment-Systeme der beteiligten Partner ein. Zuletzt ermöglicht das DIVA-Werkzeug in Beitrag 5 die IT-gestützte Durchführung der Methode. Im Folgenden werden die Beiträge im Detail vorgestellt.

1.2.1 Methode zum Deployment partnerübergreifender Anwendungen

Verfahren, die sich mit dem Verteilen von Anwendungskomponenten beschäftigen, fokussieren sich bisher auf unterschiedliche Ausführungsumgebungen, zum Beispiel im Kontext von Multi-Cloud-Anwendungen [AGLW14; NMPS17; WCX15]. Dabei werden verfügbare Cloud-Dienste basierend auf Dienstgüte-Kriterien (*engl. „Quality of Service“ (QoS)*) evaluiert und selektiert, und unter Nutzung modellgetriebener Ansätzen werden anbieterunabhängige in anbieterspezifische Deployment-Modelle transformiert. Dabei werden organisatorische Aspekte jedoch nicht berücksichtigt, da es sich ausschließlich um zentralisierte Ansätze handelt, die einen einzelnen Anwender betrachten. In Anwendungsfällen bei denen mehrere Partner involviert sind, spielen jedoch die Datenkapselung und Autonomie der Partner eine wichtige Rolle und müssen von der Modellierung bis zur Ausführung des Deployments berücksichtigt werden. Das erhöht nicht nur die Komplexität des Deployments, sondern muss auch bei der Modellierung partnerspezifischer Restriktionen berücksichtigt werden. Werden Problem erst beim Deployment entdeckt kann dies zu einem hohen Aufwand durch Restrukturierung, Adaption und Redeployment der betroffenen Komponenten führen.

Forschungsbeitrag 1 (Methode zum partnerübergreifender Anwendungen). Die vorliegende Arbeit präsentiert die DIVA-Methode („*Divide and Adapt*“-Methode), welche den gesamten Prozess von der Modellierung bis zur Ausführung des Deployments von partnerübergreifenden Anwendungen abdeckt. Die Methode (i) ist unabhängig von konkreten Deployment-Technologien, (ii) unterstützt die flexible Zuteilung von Komponenten zu Partnern und (iii) nutzt bewährtes Wissen in Form von Mustern, um ein funktionsfähiges Deployment zu erhalten. Durch die Automatisierung des Verteilens, der Problemerkennung und -behebung sowie des Deployments selbst werden Anwender bei der Anwendungsbereitstellung unterstützt und manuelle Fehler vermindert.

1.2.2 Modellierung adaptiver Deployment-Modelle und deren Verteilung

Zur Modellierung des Deployments von Anwendungssystemen werden deklarative und imperative Ansätze unterschieden [EBF+17], wobei sich in der Praxis Technologien mit einem deklarativen Ansatz durchgesetzt haben [WBF+19]. Deklarative Ansätze ermöglichen die strukturelle Spezifikation eines Anwendungssystems mit den Anwendungskomponenten, deren Beziehungen und deren Konfigurationsattributen. Für die technologieunabhängige Modellierung von Deployment-Modellen wurde bereits das *Essential Deployment Meta Model (EDMM)* eingeführt [WBF+19]. Das EDMM-Transformation-Framework unterstützt die Transformation von EDMM in 13 unterschiedliche deklarative Technologien, welche für die Ausführung des Deployments genutzt werden können [WBB+19; WBB+20b]. Jedoch fehlen bisher Modellierungskonstrukte, die als Grundlage für automatisierte Adaptionsverfahren zur Entwurfszeit des Deployments genutzt werden können. Um Adaptionsverfahren automatisiert auf Deployment-Modelle anwenden zu können, bedarf es neben der erweiterten Syntax auch einer definierten Semantik für die Komponenten und Beziehungen zwischen den Komponenten.

Forschungsbeitrag 2 (Modellierung adaptiver Deployment-Modelle und deren Verteilung). Zur Modellierung der globalen und lokalen Deployment-Modelle, zur Spezifizierung der Verteilung der Komponenten sowie zur Modellanalyse zur Problembehebung und Modelladaption wird im zweiten Forschungsbeitrag eine Erweiterung des EDMM vorgestellt. Dabei werden erweiterte Modellierungselemente nur für die Adaption zur Entwurfszeit eingeführt. Die Ausführung des Deployments basiert weiterhin auf den Basiselementen von EDMM und ermöglicht damit die Ausführung mit einer beliebigen EDMM-kompatiblen Deployment-Technologie. Für die organisatorische Verteilung von Komponenten auf Partner und die ressourcenspezifische Verteilung in der Produktivumgebung werden außerdem verschiedene Algorithmen vorgestellt.

1.2.3 Verfahren zur musterbasierten Problemerkennung und Adaption von Deployment-Modellen

Muster sind ein bewährtes Instrument zur Dokumentation von Best Practices für wiederkehrende Probleme unabhängig von einer spezifischen Technologie. Sie bieten eine menschenlesbare Beschreibung und folgen dabei einem spezifizierten Musterformat. In unterschiedlichen Domänen werden unterschiedliche Musterformate verwendet, jedoch umfasst die Beschreibung immer das *Problem*, den *Kontext* in dem das Problem auftritt, sowie die *Lösung* [MD97; WF12]. In vielen Verfahren dienen Muster als Basis für die Umsetzung von durch das jeweilige Muster beschriebene Lösungen [Bre16; FBB+14a; HBF+18]. Sie werden auch genutzt um zu erkennen, ob bestimmte Muster in gegebenen Modellen umgesetzt sind [DL06; FG13; HZ15; ME16]. Diese Verfahren stützen sich jedoch darauf, dass bereits bekannt ist, welche Muster angewendet werden sollten. Vor allem bei adaptiven Modellen, welche sich dynamisch an die jeweilige Situation anpassen können, können Probleme erst durch eine Adaption überhaupt entstehen. Bisherige Ansätze ermöglichen keine automatische Identifikation und Behebung von Problemen. Dies kann zu einem hohen Aufwand führen wenn Probleme erst während des Deployments oder danach entdeckt werden.

Forschungsbeitrag 3 (Verfahren zur musterbasierten Problemerkennung und Adaption von Deployment-Modellen). Im dritten Forschungsbeitrag wird ein Verfahren vorgestellt, welches auf Basis des dokumentierten Wissens in Mustern automatisiert (i) Probleme identifiziert und (ii) Deployment-Modelle entsprechend der verfügbaren Lösungen adaptiert. Basierend auf dem erweiterten EDMM werden die Beschreibung des Problems und des Kontexts als Regeln formalisiert, welche auf Deployment-Modelle angewendet werden können, um mögliche Probleme zu identifizieren. Zusätzlich werden Anwendbarkeitsregeln für konkrete technische Lösungen definiert, um Lösungen für eine spezifische Anwendung automatisiert zu identifizieren und das Deployment-Modell zu adaptieren.

1.2.4 Automatisiertes partnerübergreifendes dezentrales Deployment von Anwendungen

Unter den existierenden Technologien und Konzepten gibt es Verfahren, die eine dezentrale Ausführung des Deployments ermöglichen [WBB+17]. Neben Technologien für das Konfigurationsmanagement wie Chef, werden vor allem im Edge Computing und Internet der Dinge (engl. *„Internet of Things“ (IoT)*) dezentrale Verfahren eingesetzt, um die Komplexität des Deployments bei einer hohen Zahl eingebundener Geräte zu reduzieren [CV19]. Dabei werden vor allem hierarchische Koordinationsverfahren eingesetzt, um globale Anforderungen an lokale Managementkomponenten weiterzugeben, welche diese dezentral umsetzen. Damit ermöglichen diese Verfahren zwar eine dezentrale Ausführung der Deployment-Operationen, die Koordination erfolgt jedoch zentral. Datenkapselung und die Autonomie der Partner spielen dabei keine Rolle, sind aber bei partnerübergreifenden Anwendungen, welche über Organisationsgrenzen hinweg bereitgestellt und verwaltet werden müssen, essentiell.

Forschungsbeitrag 4 (Automatisiertes partnerübergreifendes dezentrales Deployment von Anwendungen). Der vierte Forschungsbeitrag stellt ein Konzept für das automatisierte Deployment partnerübergreifender Anwendungen vor, welches (i) jedem Partner die Kontrolle über das partnerspezifische Deployment ermöglicht, (ii) das globale Deployment automatisiert koordiniert und (iii) den Datenaustausch zwischen den Partnern auf die Menge der notwendigen Konfigurationsinformationen beschränkt. Damit wird die Autonomie der Partner bewahrt und nur Informationen, die für die Konfiguration von Kommunikationsbeziehungen benötigt werden, müssen ausgetauscht und zwischen den beteiligten Partnern geteilt werden.

1.2.5 DIVA-Architektur und -Werkzeug

Im Fokus der vorliegenden Arbeit stehen (i) die Verteilung von Anwendungskomponenten, (ii) das frühzeitige Erkennen und Beheben von Problemen bei Kommunikationsbeziehungen sowie (iii) die Automatisierung des Deployments partnerübergreifender Anwendungen. Um die Funktionsfähigkeit der eingeführten Konzepte in den vorherigen Forschungsbeiträgen zu demonstrieren und zu evaluieren werden sie prototypisch umgesetzt und in bestehende Deployment-Prozesse integriert. Ziel ist die Integration der Konzepte in existierende Werkzeuge zur Sicherstellung eines integrativen Prototypen. Dazu wurden unter anderem das OpenTOSCA Ökosystem [BEK+16], welches das Modellierungswerkzeug Winery [KBBL13], das Deploymentsystem OpenTOSCA Container [BBH+13] und ein Self-Service-Portal umfasst, sowie das EDMM-Transformation-Framework [WBB+19] erweitert. Diese Werkzeuge stehen alle als Open-Source Software zur Verfügung.

Forschungsbeitrag 5 (DIVA-Werkzeugunterstützung). In diesem Forschungsbeitrag werden die Systemarchitektur sowie die prototypische Implementierung der gesamten DIVA-Methode vorgestellt. Das Werkzeug umfasst folgende Komponenten: (i) ein Modellierungswerkzeug für globale und lokale Deployment-Modelle, (ii) ein Analyseframework zur automatisierte Problem- und Adaptionserkennung und (iii) ein dezentrales Deployment-System. Das DIVA-Werkzeug ist in das bestehende OpenTOSCA Ökosystem und EDMM-Transformation-Framework integriert. Die Integration ermöglicht Funktionalitäten des EDMM-Transformation-Frameworks in die DIVA-Methode einzubinden und so unter anderem die Ausführung des Deployments mit unterschiedlichen Technologien zu ermöglichen.

1.3 Veröffentlichungen im Rahmen der Arbeit

Im Rahmen dieser Dissertation von Karoline Wild, geb. Saatkamp, sind die folgenden Publikationen entstanden, welche entsprechend ihrer Relevanz für die vorliegende Arbeit sortiert gelistet sind:

1. [SBKL19a] K. Saatkamp, U. Breitenbücher, O. Kopp und F. Leymann. „An approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns“. In: *SIC Software-Intensive Cyber-Physical Systems*, 2019, S. 85-97
2. [WBK+20] K. Wild, U. Breitenbücher, K. Képes, F. Leymann und B. Weder. „Decentralized Cross-Organizational Application Deployment Automation: An Approach for Generating Deployment Choreographies Based on Declarative Deployment Models“. In: *Advanced Information Systems Engineering (CAiSE 2020)*, Springer, 2020, S. 20-35
3. [SBKL19b] K. Saatkamp, U. Breitenbücher, O. Kopp und F. Leymann. „Method, formalization, and algorithms to split topology models for distributed cloud application deployments“. In: *Computing*, 2019, S. 343-363
4. [SBF+19] K. Saatkamp, U. Breitenbücher, M. Falkenthal, L. Harzenetter und F. Leymann. „An Approach to Determine & Apply Solutions to Solve Detected Problems in Restructured Deployment Models using First-order Logic“. In: *Proceedings of the 9th International Conference on Cloud Computing and Service Science (CLOSER 2019)*, 2019, S. 495-506
5. [SBKL17] K. Saatkamp, U. Breitenbücher, O. Kopp und F. Leymann. „Topology Splitting and Matching for Multi-Cloud Deployments“. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, 2017, S. 247-258
6. [SBLW17] K. Saatkamp, U. Breitenbücher, F. Leymann, M. Wurster. „Generic Driver Injection for Automated IoT Application Deployment“. In: *Proceedings of the 19th International Conference on Information*

Integration and Web-based Applications & Services (iiWAS 2017), 2017, S. 320-329

7. [SBK+18] K. Saatkamp, U. Breitenbücher, K. Képes, F. Leymann, M. Zimmermann. „OpenTOSCA Injector: Vertical and Horizontal Topology Model Injection“. In: *Service-Oriented Computing - ICSOC 2017 Workshops*, 2018, S. 379-383
8. [SBKL18] K. Saatkamp, U. Breitenbücher, O. Kopp , F. Leymann. „Application Scenarios for Automated Problem Detection in TOSCA Topologies by Formalized Patterns“. In: *Papers from the 12th Advanced Summer School on Service-Oriented Computing (SummerSOC 2018)*, 2018, S. 43-53

Zusätzlich sind in Kooperation weitere Publikationen entstanden, die Grundlagen und ergänzende Konzepte für die vorliegende Arbeit umfassen:

1. [WBF+19] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. „The essential deployment metamodel: a systematic review of deployment automation technologies“. In: *SICS Software-Intensive Cyber-Physical Systems*, 2019, S. 63-75
2. [WBB+21] M. Wurster, U. Breitenbücher, A. Brogi, F. Diez, F. Leymann, J. Soldani, K. Wild. „Automating the Deployment of Distributed Applications by Combining Multiple Deployment Technologies“. In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*, 2021, S. 178-189
3. [KLWW21] K. Képes, F. Leymann, B. Weder, K. Wild. „SiDD: The Situation-Aware Distributed Deployment System“. In: Service-Oriented Computing – ICSOC 2020 Workshops, 2021, S. 72-76
4. [KBL+19] K. Képes, U. Breitenbücher, F. Leymann, K. Saatkamp, B. Weder. „Deployment of Distributed Applications Across Public and Private Networks“. In: *Proceedings of the 23rd IEEE International Enterprise Distributed Object Computing Conference (EDOC 2019)*, 2019, S. 236-242

Des Weiteren sind folgende Arbeiten entstanden, die über den Kontext der Arbeit hinausgehen:

1. [WBH+20a] K. Wild, U. Breitenbücher, L. Harzenetter, F. Leymann, D. Vietz, M. Zimmermann. „TOSCA4QC: Two Modeling Styles for TOSCA to Automate the Deployment and Orchestration of Quantum Applications“. In: *Proceedings of the* 24th *International Enterprise Distributed Object Computing Conference (EDOC 2020)*, 2020, S. 125-134
2. [SKL+19] K. Saatkamp, C. Krieger, F. Leymann, J. Sudendorf, M. Wurster. „Application Threat Modeling and Automated VNF Selection for Mitigation using TOSCA“. In: *2019 International Conference on Networked Systems (NetSys)*, 2019, S. 1-6
3. [BLF+21] J. Barzen, F. Leymann, M. Falkenthal, D. Vietz, B. Weder, K. Wild. „Relevance of Near-Term Quantum Computing in the Cloud: A Humanities Perspective“. In: *Cloud Computing and Services Science*, 2021, S. 25-58
4. [MFK+21] D. Martyniuk, M. Falkenthal, N. Karam, A. Paschke, K. Wild. „An Analysis of Ontological Entities to Represent Knowledge on Quantum Computing Algorithms and Implementations“. In: *Proceedings of the Conference on Digital Curation Technologies (Qurator 2021)*, 2021, S. 1-9
5. [SBL+21] M. Salm, J. Barzen, F. Leymann, B. Weder, K. Wild. „Automating the Comparison of Quantum Compilers for Quantum Circuits“. In: *Proceedings of the* 15th *Symposium and Summer School on Service-Oriented Computing (SummerSOC 2021)*, 2021, S. 64-80
6. [VBLW21] D. Vietz, J. Barzen, F. Leymann, K. Wild. „On Decision Support for Quantum Application Developers: Categorization, Comparision, and Analysis of Existing Technologies“. In: *Computational Science - ICCS 2021*, 2021, S. 127-141

7. [VKW21] D. Voigt, O. Kopp, K. Wild. „Systematic Literature Review Tools: Are we there yet?“. In: *Proceedings of the 13th Central European Workshop on Services and their Composition (ZEUS 2021)*, 2021, S. 83-88
8. [WBL+21] B. Weder, J. Barzen, F. Leymann, M. Salm, K. Wild. „QProv: A provenance system for quantum computing“. In: *IET QuantumCommunication*, 2021, S. 171-181
9. [LBF+20] F. Leymann, J. Barzen, M. Falkenthal, D. Vietz, B. Weder, K. Wild. „Quantum in the Cloud: Application Potentials and Research Opportunities“. In: *Proceedings of the 10th International Conference on Cloud Computing and Service Science (CLOSER 2020)*, 2020, S. 9-24
10. [SBB+20] M. Salm, J. Barzen, U. Breitenbücher, B. Weder, K. Wild. „The NISQ Analyzer: Automating the Selection of Quantum Computers for Quantum Algorithms“. In: *Proceedings of the 14th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2020)*, 2020, S. 66-85
11. [WBLW20] B. Weder, U. Breitenbücher, F. Leymann, K. Wild. „Integrating Quantum Computing into Workflow Modeling and Execution“. In: *Proceedings of the 13th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2020)*, 2020, S. 279-291
12. [WBB+20a] M. Weigold, J. Barzen, U. Breitenbücher, M. Falkenthal, F. Leymann, K. Wild. „Pattern Views: Concepts and Tooling of Interconnected Pattern Languages“. In: *Proceedings of the 14th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2020)*, 2020, S. 86-103
13. [HBL+19] L. Harzenetter, U. Breitenbücher, F. Leymann, K. Saatkamp, B. Weder, M. Wurster. „Automated Generation of Management Workflows for Applications Based on Deployment Models“. In: *Proceedings of the 23rd IEEE International Enterprise Distributed Object Computing Conference (EDOC 2019)*, 2019, S. 216-225

14. [ZBG+18] M. Zimmermann, U. Breitenbücher, J. Guth, S. Hermann, F. Leymann, K. Saatkamp. „Towards Deployable Research Object Archives Based On TOSCA“. In: *Papers from the 12th Advanced Summer School on Service-Oriented Computing (SummerSOC 2018)*, 201, S. 31-42
15. [ZBF+17] M. Zimmermann, U. Breitenbücher, M. Falkenthal, F. Leymann, K. Saatkamp. „Standard-based Function Shipping - How to use TOSCA for Shipping and Execution Data Analytics Software in Remote Manufacturing Environments“. In: *Proceedings of the 2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC 2017)*, 2017, S. 50-60

1.4 Aufbau der Arbeit

Bevor im Detail die Forschungsbeiträge dieser Arbeit erläutert werden, werden in Kapitel 2 die Grundlagen vorgestellt und verwandte Arbeiten diskutiert. Des Weiteren werden die abgeleiteten Forschungsfragen, die dieser Arbeit zu Grunde liegen, präsentiert. In Kapitel 3 wird die gesamte DivA-Methode vorgestellt und darauf aufbauend die einzelnen Bausteine in den darauffolgenden Kapiteln detailliert eingeführt. Die Erweiterung des EDMMs wird in Kapitel 4 spezifiziert und Algorithmen zur ressourcenspezifischen Verteilung der Komponenten eines Deployment-Modells basierend auf den Erweiterungen werden präsentiert. Das Verfahren zur musterbasierten Problemerkennung und -behebung wird anschließend in Kapitel 5 erläutert und eine konkrete Lösung zur Anpassung der Kommunikation zwischen Anwendungskomponenten wird vorgestellt. Der letzte Baustein, das dezentrale Deployment, wird in Kapitel 6 eingeführt. Für die IT-gestützte Durchführung der DivA-Methode werden das DivA-Werkzeug und die prototypische Implementierung in Kapitel 7 vorgestellt. Zuletzt werden die Erkenntnisse und Beiträge dieser Arbeit in Kapitel 8 zusammengefasst und diskutiert, gefolgt von einem Ausblick für weitere Arbeiten.

Grundlagen des Deployments und verwandte Arbeiten

In diesem Kapitel werden die Grundlagen und der aktuelle Stand der Forschung zum Deployment von Anwendungen und zur Anwendung von Mustern erläutert und diskutiert. Basierend darauf werden offene Forschungsfragen identifiziert, die die Motivation für diese Arbeit darstellen. Dazu werden in Abschnitt 2.1 verschiedene Modellierungsansätze und Modellierungssprachen zur Spezifikation des Deployments von Anwendungen sowie modellbasierte Ansätze zur Adaption von Deployment-Modellen vorgestellt. In Abschnitt 2.2 werden anschließend Verfahren und Technologien zur Bereitstellung von Anwendungen eingeführt, die auf der Verarbeitung von Deployment-Modellen basieren. Dabei liegt der besondere Fokus auf der Kombination verschiedener Technologien und der Koordination des Deployments. Abschließend werden in Abschnitt 2.3 die Anwendbarkeit und Anwendung von Mustern und Musterlösungen beleuchtet.

2.1 Modellierung adaptiver Deployment-Modelle

In der Softwareentwicklung ist die modellgetriebene Entwicklung (engl. *„Model-Driven Development“ (MDD)*) ein zentrales Konzept, welches das Modell in den Mittelpunkt der Entwicklung stellt [Sel03]. Dieses Konzept wurde im Bereich der Softwarearchitekturentwicklung als modellgetriebene Architektur (engl. *„Model-Driven Architecture“ (MDA)*) von der OMG standardisiert [OMG14a]. Das Modell erfüllt dabei verschiedene Eigenschaften [Sel03]: Ein Modell ist eine *Abstraktion* des repräsentierten Systems, muss aber *verständlich* sein. Die Verständlichkeit steht in direkter Verbindung mit der Ausdrucksstärke der gewählten Modellierungssprache. Trotz Abstraktion ist auch die *Genauigkeit* der Abbildung der betrachteten Merkmale des Systems wichtig. Diese und die Modellierungssprache sind für die *Vorhersagbarkeit*, zum Beispiel durch Modellanalysen, entscheidend. Neben dem Modell selbst steht bei MDA die Modelltransformation im Fokus. Das *Computation Independent Model (CIM)* spezifiziert die Anforderung der Anwendung unabhängig von der Realisierung. Basierend darauf wird das *Platform Independent Model (PIM)* modelliert, welches anschließend basierend auf einem *Platform Model (PM)*, welches die plattformspezifischen Elemente definiert, in ein *Platform Specific Model (PSM)* transformiert werden kann.

Das Konzept der modellgetriebenen Entwicklung hat sich auch beim Deployment von Anwendungen etabliert. Dazu werden im Folgenden zuerst die zwei grundlegenden Modellierungsansätze erläutert und anschließend verschiedene Modellierungssprachen und Methoden zur modellgetriebenen Selektion von Cloud-Diensten und zur Modelladaption vorgestellt.

2.1.1 Imperative und deklarative Deployment-Modelle

Grundsätzlich lassen sich zwei Ansätze zur Modellierung des Deployments von Anwendungssystemen unterscheiden [EBF+17]: imperative und deklarative Ansätze. In Abbildung 2.1 sind ein imperatives (links) und ein deklaratives (rechts) Deployment-Modell skizziert. Beide Modelle beschreiben das Deployment einer Anwendung (App), die Zugriff auf eine Datenbank

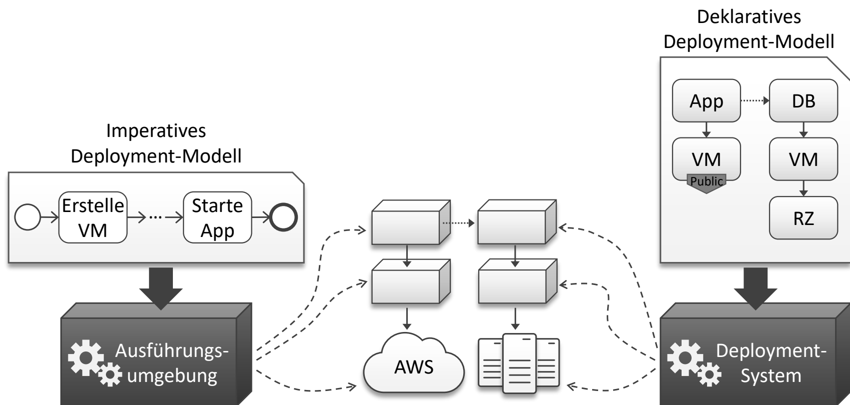

Abbildung 2.1: Imperatives und deklaratives Deployment-Modell.

(DB) benötigt und beide Komponenten sollen jeweils auf einer virtuellen Maschine (VM) bereitgestellt werden, die Datenbank im Rechenzentrum (RZ) und die Anwendung in einer Public Cloud, die basierend auf der spezifizierten Anforderung selektiert wird, in diesem Fall AWS. Während *deklarative* Deployment-Modelle einen gewünschten Zielzustand der Anwendung spezifizieren, definieren *imperative* Deployment-Modelle die auszuführenden Operationen und deren Ausführungsreihenfolge, um einen bestimmten Zielzustand zu erreichen. Zur Spezifikation des Zielzustands in deklarative Modellen wird die Struktur der Anwendung durch ihre Komponenten, deren Beziehungen und Konfigurationseigenschaften definiert [WBB+17]. Die benötigten Operationen und deren Reihenfolge zur Erreichung des Zielzustands werden durch das jeweilige Deployment-System vom Modell abgeleitet.

In der Praxis haben sich vor allem deklarative Ansätze für das Deployment von Anwendungen durchgesetzt [HAW11; WBF+19]. In Wurster et al. [WBF+19] wurden die 13 meist genutzten Deployment-Technologien analysiert, welche alle einen deklarativen Ansatz unterstützen. Dabei wurden unter anderem Technologien wie Puppet, Chef, Kubernetes, Terraform und AWS CloudFormation berücksichtigt. Auch fast alle Cloud-Modellierungssprachen, die von Bergmayr et al. [BBF+18] untersucht wurden und die das automa-

tisierte Deployment der modellierten Anwendung ermöglichen, basieren auf einem deklarativen Ansatz. Dazu zählen unter anderem CloudML [FRC+13], CAMEL [AKR+19] und der OASIS Standard TOSCA, das für *Topology and Orchestration Specification for Cloud Applications (TOSCA)* [OAS13; OAS20] steht. Der TOSCA Standard unterstützt jedoch nicht nur den deklarativen Ansatz, sondern ermöglicht auch die Kombination mit imperativen Modellen.

Zu den imperativen Deployment-Sprachen zählen unter anderem universelle Modellierungssprachen wie die Workflow-Sprachen *Business Process Execution Language (BPEL)* [OAS07b] und *Business Process Model and Notation (BPMN)* [OMG11], sowie Skriptsprachen. Darüber hinaus gibt es Erweiterungen der Workflow-Sprachen, wie zum Beispiel eine Erweiterung von BPMN zu BPMN4TOSCA [KBBL12], die spezielle *Task*-Typen für das Deployment umfasst. Im Allgemeinen ist die Spezifikation von deklarativen Deployment-Modellen weniger komplex als von imperativen Modellen, jedoch ist auch die Ausführung weniger anpassbar, während imperative Deployment-Modelle komplexer zu definieren sind, aber die volle Kontrolle über die Deployment-Schritte ermöglichen. Daher werden oft hybride Ansätze verwendet, welche deklarativer Modelle in imperative Modelle transformieren, um die Vorteile beider Ansätze zu nutzen [BBK+14; CCDT18].

Aufgrund der weiten Verbreitung deklarativer Modellierungssprachen in Wissenschaft und Praxis basiert die in dieser Arbeit vorgestellte modellgestützte Methode auf einem deklarativen Modellierungsansatz. Im Folgenden werden existierende deklarative Modellierungssprachen sowie modellgestützte Methoden zur Selektion von Cloud-Diensten zur Bereitstellung von Anwendungskomponenten sowie zur Adaption der Modelle betrachtet. In Abschnitt 2.2 wird noch einmal detaillierter auf die hybriden Ansätze im Kontext der Bereitstellung von Anwendungen eingegangen.

2.1.2 Sprachen zur Modellierung deklarativer Deployment-Modelle

Zur Modellierung deklarativer Deployment-Modelle wurde eine Vielzahl domänenspezifischen Sprachen (DSLs) entwickelt [BBF+18; WBF+19]. Im Folgenden werden DSLs vorgestellt und deren Eignung für die Modellierung

partnerübergreifender Anwendungen diskutiert. Dabei werden Sprachen aus dem Cloud- und Anwendungskonfigurationsbereich berücksichtigt.

Beginnend mit DSLs aus der Cloud-Domäne, sind in Tabelle 2.1 wichtige deklarative Cloud-Modellierungssprachen gelistet. Die Sprachen sind entsprechend ihrer Beschreibung im Text geordnet. Die Selektion der hier diskutierten Modellierungssprachen beruht auf der Analyse von Bergmayr et al. [BBF+18] sowie zusätzlichen Literaturstudien. Von der Vielzahl verfügbarer Modellierungssprachen werden ausschließlich deklarative Sprachen betrachtet, deren Modelle von Deployment-Systemen interpretiert und verarbeitet werden können. Des Weiteren sind nur Sprachen gelistet, die beliebige Cloud-Service-Modelle, das heißt unter anderem *Infrastructure-as-a-Service (IaaS)* und *Platform-as-a-Service (PaaS)*, unterstützen und eine Modellierung der Anwendung unabhängig von einem spezifischen Cloud-Anbieter ermöglichen. Technologiespezifische DSLs werden später im Kapitel im Kontext der Transformation von technologieagnostischen zu technologiespezifischen Modellen und Artefakten diskutiert. Für jede Sprache ist in Tabelle 2.1 aufgeführt, ob sie zum Deployment von Multi-Cloud-Anwendungen geeignet ist, welche Analyse- sowie Adaptionsmechanismen unterstützt werden, und es wird eine Auswahl an Tools, welche die jeweilige Sprache unterstützen, aufgeführt. Bei den Analyse- und Adaptionsmechanismen wird unterschieden, ob diese zur (i) Entwurfszeit, (ii) Deploymentzeit oder (iii) Laufzeit anwendbar sind. Dies entspricht der Deployment-Entwurfszeit, der Deployment-Laufzeit und der Anwendungslaufzeit von Arcangeli, Boujbel und Leriche [ABL15]. Diese Eigenschaften wurden aus der Analyse von Bergmayr et al. [BBF+18] sowie ergänzenden Literaturstudien zu den jeweiligen Sprachen entnommen.

Das Komponentenmodell Aeolus [DMZZ14] ermöglicht die Beschreibung der Komponenten einer Anwendung und deren Abhängigkeiten. Jede Komponente spezifiziert (i) angebotene Funktionalitäten (*engl. „Provide Ports“*), (ii) benötigte Funktionalitäten (*engl. „Require Ports“*) und (iii) einen Zustandsautomaten (*engl. „State Machine“*). Aeolus verknüpft die angebotenen und benötigten Funktionalitäten mit dem Zustand der Komponenten. Damit kann unter Berücksichtigung der Abhängigkeiten zwischen den Komponenten ein sogenannter *Deployment-Plan*, der die Reihenfolge des Deploy-

Sprache	Multi-Cloud	Analyse	Adaption	Deployment Tools (Auswahl)
Aeolus [DMZZ14]	✓	Entwurfszeit: Modellvalidierung & Deployment Optimierung+	Entwurfszeit: Verfeinerung & Verteilung+	Blender [DEZ+15], Zephyrus2 [ÁCJ+16]
CloudML* [FRC+13]	✓	✗	Entwurfszeit: Verfeinerung & Verteilung, Laufzeit: Rekonfiguration	CloudMF [FCS+18; FSR+14], MODAClouds [FAS17a]
CAMEL [AKR+19; RKN+16]	✓	Entwurfszeit+ & Laufzeit: Deployment Optimierung	Entwurfszeit: Verfeinerung & Verteilung, Laufzeit: Rekonfiguration	PaaSage [AKR+19], MELODIC [HS18]
MOCCA* [LFM+11]	✓	Entwurfszeit: Deployment Optimierung	✗	Cafe-Framework [MUL09]
TOSCA* [OAS13; OAS20]	✓	Entwurfszeit: Modellvalidierung+	Entwurfszeit & Deploymentzeit: Verfeinerung & Verteilung	OpenTOSCA [BEK+16], Tosker [BRS18], Cloudify [Clo22]
CAML* [BTN+14]	–	✗	Entwurfszeit: Verfeinerung	(nur durch Modelltransformation: CAML-TOSCA [BBK+16])

Tabelle 2.1: Überblick relevanter Cloud-Modellierungssprachen: Die gekennzeichneten Sprachen(*) wurden detailliert von Bergmayr et al. [BBF+18] analysiert. Die gekennzeichneten Mechanismen(+) sind Erweiterungen zu den nativ unterstützten Mechanismen.

ments der verschiedenen Komponenten definiert, berechnet werden. Über die Berechenbarkeit eines Deployment-Plans ist auch eine Validierung der Korrektheit der Abhängigkeiten zwischen den Komponenten möglich. Durch eine Erweiterung [ÁCJ+16] können außerdem VMs als Standort (*engl. „Location“*) modelliert und basierend auf den Anforderungen der Komponenten und den Eigenschaften der VMs das Deployment-Optimierungsproblem zur Minimierung der Kosten des Deployments durch optimales Verteilen der Komponenten auf die verfügbaren VMs gelöst werden. Dies wird jedoch nur zur Entwurfszeit betrachtet.

Die *Cloud Modelling Language (CloudML)* [FCS+18; FRC+13; FSR+14] fokussiert vor allem das Management von Multi-Cloud-Anwendungen und umfasst das Management nicht nur zur Entwurfszeit, sondern auch zur Laufzeit und folgt damit dem *models@runtime* [BBF09] Paradigma. CloudML ermöglicht ähnlich wie Aeolus die Spezifikation von Komponenten mit angebotenen und benötigten Funktionalitäten, wobei zwischen Kommunikations- und Hosting-Beziehungen zwischen Komponenten unterschieden wird. Des Weiteren werden Anbieter (*engl. „Provider“*) spezifiziert, die externe Komponenten, zum Beispiel VMs, in einer bestimmten Konfiguration bereitstellen. Mit diesen Informationen können cloudanbieterunabhängige Modelle (*engl. „Cloud Provider-Independent Models“ (CPIMs)*) in cloudanbieterspezifische Modelle (*engl. „Cloud Provider-Specific Models“ (CPSMs)*), angelehnt an die PIMs und PSMs von MDA, transformiert werden. Das *Cloud Modeling Framework (CloudMF)* für CloudML [FCS+18; FSR+14] basiert auch auf einem hybriden Ansatz und generiert aus dem deklarativen Modell einen Deployment-Plan bzw. bei Rekonfigurationen zur Laufzeit einen Adaptionsplan.

Eine Erweiterung von CloudML ist die *Cloud Application Modelling and Execution Language (CAMEL)* [AKR+19; RKN+16]. Konkret besteht CAMEL aus mehreren DSLs, die verschiedene Bereiche abdecken. Das zentrale Deployment-Metamodell basiert auf dem Metamodell von CloudML. Weitere DSLs ermöglichen unter anderem die Spezifikation von Anforderungen, Cloud-Anbietern oder auch Skalierungsregeln. Diese Informationen werden zur Selektion passender Cloud-Anbieter und -Dienste für bestimmte Komponenten und darauf basierender Verfeinerung und Verteilung der Komponenten sowie

zur Rekonfiguration zur Laufzeit, die im Gegensatz zu CloudML dynamisch über Regeln, zum Beispiel zur Skalierung, erfolgt, genutzt.

Für die *MOCCA (MOve to Clouds for Composite Applications)*-Methode [LFM+11] wurde ein Metamodell entwickelt, welches als Vorstufe des TOSCA-Standards gesehen werden kann. Ebenso wie die bereits vorgestellten Metamodelle ermöglicht MOCCA die Spezifikation von Komponenten und deren Beziehungen. Die Spezifikation der Beziehungen ist jedoch nur statisch und nicht dynamisch über definierte angebotene und benötigte Funktionalitäten der Komponenten möglich. Die Implementierungen der Komponenten werden jedoch von den logischen Komponentenbeschreibungen separiert und damit austauschbar gemacht. Komponenten und Beziehungen können mit nicht-funktionalen Eigenschaften, wie Sicherheitslevel oder Datendurchsatz, sowie präferierten Cloud-Anbietern annotiert werden. Basierend darauf kann die optimale Verteilung der Komponenten, die bisher nicht mit einem Cloud-Anbieter annotiert sind, berechnet werden. Konkrete Cloud-Dienste der Anbieter sowie benötigte Middleware- und Infrastrukturkomponenten werden dabei jedoch nicht berücksichtigt.

Der TOSCA-Standard [OAS13; OAS20] ist eine erweiterbare Modellierungssprache zur Spezifikation von portablen Deployment-Modellen für Cloud-Anwendungen, die von der OASIS standardisiert ist. TOSCA bietet ein umfangreiches Metamodell und Sprache mit einem erweiterbaren Typsystem für Komponenten, Beziehungen, aber auch Artefakte. In wissenschaftlichen Arbeiten wird TOSCA in vielen Bereichen genutzt [BM20]. Dabei wird TOSCA über Cloud Computing hinaus auch in anderen Domänen wie Internet der Dinge (*engl. „Internet of Things“ (IoT)*) oder Netzwerkfunktionsvirtualisierung (*engl. „Network Function Virtualization“ (NFV)*) eingesetzt. Nativ bietet TOSCA Möglichkeiten zur Substitution von abstrakten Komponenten durch Deployment-Modelle und die Spezifikation von Anforderungen (*engl. „Requirements“*) und Fähigkeiten (*engl. „Capabilities“*) zur Verfeinerung und Verteilung von Komponenten auf verschiedene Ressourcen oder Cloud-Anbieter. Darüber hinaus gibt es einige Arbeiten, die sich mit der Verifikation von TOSCA-Modellen beschäftigen [BM20]. TOSCA wird auch vermehrt in der Praxis unterstützt. Cloudify [Clo22] ist ein bereits in der Praxis einge-

setztes Deployment-System, das TOSCA-Modelle konsumieren kann. Auch ein Anbieter von Private-Cloud-Lösungen, OpenStack, ermöglicht die Transformation von TOSCA-Modellen in *Heat Orchestration Templates (HOT)*, die eigene DSL von OpenStack [Ope22].

Anders als die vorherigen Modellierungssprachen basiert die *Cloud Application Modelling Language (CAML)* [BTN+14] auf der universellen Modellierungssprache *Unified Modeling Language (UML)* [OMG07]. UML bietet verschiedene Diagrammtypen, darunter auch spezifisch für die Deployment-Sicht auf eine Architektur. Jedoch werden hier cloudspezifische Elemente, wie Cloud-Anbieter und deren Dienste, nicht berücksichtigt. CAML erweitert die bestehenden Modellierungselemente um dedizierte UML-Profile für die verschiedenen Cloud-Angebote. Im Fokus steht vor allem die Verfeinerung von CPIMs zu CPSMs. Ob sich CAML auch zur Modellierung von Multi-Cloud-Anwendungen eignet, ist nicht bekannt. Für CAML-Modelle gibt es kein Deployment-System, das diese Modelle prozessieren kann. Jedoch können die Modelle in TOSCA transformiert und damit durch TOSCA-Deployment-Systeme verarbeitet werden [BBK+16].

Es gibt noch weitere UML-basierende Frameworks, so wie *MULTICLAPP (MULTICLoud migratable and interoperable APPlications)* [GMMC13], URANO [RVT+18] oder DICER [ABD+18]. MULTICLAPP führt zur Modellierung eines PIM ein UML-Profil ein, welches jedoch keine Cloud-Anbieter und deren Dienste umfasst. Um die Merkmale von Cloud-Angeboten abzubilden, werden in MULICLAPP Feature-Modelle verwendet. Basierend auf den Merkmalen der Cloud-Angebote und der spezifizierten Anforderungen im PIM können so passende Angebote gefunden werden. Durch Modelltransformation wird basierend auf der Auswahl ein PSM generiert. Anschließend werden durch Modell-Text-Transformation ausführbare Artefakte generiert. URANO geht nach dem gleichen Prinzip vor, jedoch werden die Cloud-Anbieter und deren Dienste im UML-Modell abgebildet. Das DICER UML-Profil ist an CloudML angelehnt und für die Spezifizierung von Anforderungen für das Deployment der Komponenten wird die *Object Constraint Language (OCL)* [OMG14b] verwendet. Neben dem modellgetriebenen Ansatz, der auch von DICER unterstützt wird, ist die Transformation in verschiedene Deployment-Tech-

nologien eine Schlüsselfunktionalität. Konkret wurde die Transformation in TOSCA und Chef gezeigt [ABD+18].

Für die Modellierung, Verteilung, Adaption und das Deployment partnerübergreifender Anwendungen sind vor allem die Unterstützung von Multi-Cloud-Anwendungen und Adaptionsmöglichkeiten der Deployment-Modelle wichtige Grundeigenschaften, um die nötige Flexibilität bei der Adaption globaler Deployment-Modelle sowie der Restrukturierung lokaler Deployment-Modelle zu ermöglichen. Sprachen wie MOCCA und CAML sind daher grundsätzlich nicht als Basis geeignet (siehe Tabelle 2.1).

Die von DICER ermöglichte Transformation von einer generischen Modellierungssprache in verschiedene technologiespezifische DSLs ist essenziell, um die Integration in bestehende Technologielandschaften bei unterschiedlichen Partnern beim Deployment partnerübergreifender Anwendungen zu ermöglichen. Dies ist bei den bisher vorgestellten Sprachen nur bei TOSCA und den UML-basierten Modellierungssprachen möglich. Bei der Einführung von neuen Modellierungssprachen und Konzepten müssen damit auch die etablierten bereits in den Unternehmen eingesetzten Deployment-Technologien ersetzt werden, was eine große Hürde für die Nutzung in der Praxis darstellt. Die Kombinationsfähigkeit von ausdrucksstarken Modellierungssprachen und etablierten Deployment-Technologien wurde deshalb bereits in verschiedenen Arbeiten untersucht [BMG+19; QK18; SIA17; SIA19; WBBC16; WBF+19; WBH+20c].

Quint und Kratzke [QK18] betrachten ausschließlich verschiedene containerbasierte Plattformen wie Kubernetes, Docker Swarm und Mesos und führen eine plattformunabhängige DSL ein, welche durch Modelltransformation in die entsprechenden DSLs der Plattformen transformiert werden kann. Weerasiri et al. [WBBC16] führen für verschiedene Technologien, wie Docker Compose, Ansible oder Chef, domänenspezifische Modelle ein, die eine abstrakte Modellierung und anschließende Transformation in die Artefakte der jeweiligen Technologie ermöglichen. Zwar wird auch ermöglicht, dass Modelle aus Komponenten bestehen, die von verschiedenen Technologien verwaltet werden, jedoch gibt es kein technologieagnostisches Metamodell. Ein solches Metamodell wird von ARGON [SIA17; SIA19] eingeführt. Die ARGON DSL

ermöglicht sowohl die Transformation von CPIMs in CPSMs als auch die Transformation von CPSMs in verarbeitbare Artefakte von Technologien wie Terraform oder Ansible. Damit ermöglicht ARGON die Modellierung unabhängig sowohl vom Cloud-Anbieter als auch der eingesetzten Technologie. Brabra et al. [BMG+19] nutzen den TOSCA-Standard als technologieagnostisches Metamodell zur Transformation in Artefakte für Docker, Kubernetes, Terraform und Juju.

Keiner dieser Ansätze basiert jedoch auf einer systematischen Analyse der in der Praxis meist verbreiteten Deployment-Technologien. Damit ist nicht sichergestellt, dass diese Ansätze auch für weitere Technologien einsetzbar sind. Auch wird die unterschiedliche Ausdrucksmächtigkeit der Sprachen nicht berücksichtigt. In Zusammenarbeit mit Wurster et al. [WBF+19] wurden deshalb die 13 meist genutzten Deployment-Technologien analysiert und ein gemeinsames Metamodell extrahiert. Dieses *Essential Deployment Meta Model (EDMM)* dient als kanonisches Metamodell für die Transformation in die 13 untersuchten Technologien [WBB+19; WBB+20b]. Insbesondere wurden dabei auch Besonderheiten bei der Transformation von IaaS-basierten Modellen in PaaS-basierte Modelle berücksichtigt, zum Beispiel um das Deployment mit Kubernetes zu ermöglichen.

Um die Lücke zu wissenschaftlichen Arbeiten zu schließen, in denen TOSCA weite Verbreitung findet [BM20], wurde die Abbildung des TOSCA-Metamodells auf EDMM untersucht. Das TOSCA-Metamodell ist ausdrucksstärker als EDMM, das heißt, dass nicht alle Entitäten, die in TOSCA modelliert werden können, in EDMM abgebildet werden können, um eine Transformation in alle 13 Technologien zu ermöglichen. Die identifizierte Teilmenge der abbildbaren TOSCA-Entitäten wird *TOSCA Light* genannt und die Modellierung mit TOSCA Light in den Transformationsprozess integriert [WBH+20b; WBH+20c]. Zu den nicht unterstützten Entitäten zählen unter anderem Anforderungen und Fähigkeiten von Komponenten. Dabei muss jedoch zwischen Entwurfszeit und Deploymentzeit unterschieden werden. Zur Entwurfszeit können auch TOSCA-Elemente außerhalb von TOSCA Light genutzt werden, um zum Beispiel Modelladaptionen zu ermöglichen. Erst zur Deploymentzeit können diese nicht mehr berücksichtigt werden.

Die Kombination von TOSCA und EDMM bietet damit wichtige Funktionalitäten für das Deployment partnerübergreifender Anwendungen: TOSCA bietet Modellierungselemente, die zur Modellanalyse und -adaption genutzt werden können, und die Abbildung zu EDMM ermöglicht das Deployment mit unterschiedlichen Deployment-Technologien. Für die Erweiterung des EDMMs zur Realisierung der in dieser Arbeit vorgestellten Konzepte werden deshalb Modellierungselemente aus TOSCA übernommen (siehe Kapitel 4).

2.1.3 Adaption des Deployments

Bei den bisher aufgeführten Modellierungssprachen in Abschnitt 2.1.2 wurden bereits deren generellen Adaptionsfähigkeiten erläutert. In diesem Abschnitt werden konkrete Verfahren zur Anpassung des Deployments von Anwendungen eingeführt, welche teilweise die bereits diskutierten Modellierungssprachen nutzen. Dabei können grundsätzlich zwei Kategorien der Deployment-Adaptionen unterschieden werden: (i) Portabilität von Anwendungskomponenten und Selektion der Bereitstellungsumgebung [PV14] sowie (ii) Adaption zwischen Anwendungskomponenten [MMGC12]. Im Folgenden werden verschiedene Ansätze aus diesen Kategorien erläutert.

2.1.3.1 Portabilität und Selektion der Bereitstellungsumgebung

Die Bedeutung der bedarfsgerechten Selektion von Cloud-Anbietern und deren Dienste bzw. generell der Bereitstellungsumgebung von Anwendungskomponenten hat durch die Möglichkeit, Dienste von unterschiedlichen Cloud-Anbietern zu nutzen, zugenommen. Darüber hinaus gibt es unterschiedliche Anforderungen und Konfigurationen in der Testumgebung während der Entwicklungsphase und der Produktivumgebung, die zu Anpassungen des Deployments führen [Hei20; WAL15; WK18]. Für die Identifikation geeigneter Cloud-Dienste für spezifische Anwendungskomponenten gibt es verschiedene Verfahren, die angewendet werden [SDH+14]. Verbreitet sind unter anderem multiattributive Auswahlverfahren wie der analytische Hierarchieprozess (AHP) oder die Nutzwertanalyse. Ein Beispiel dafür ist die

Arbeit von Menzel und Ranjan [MR12], die AHP nutzen und den Preis des Dienstes, die Dienstgüte in Bezug auf die Performanz (CPU, RAM, Speicher), dessen Beliebtheit und Verfügbarkeit sowie die Latenzzeiten des Dienstes berücksichtigen. Ein weiteres Beispiel ist die Arbeit von Andrikopoulos et al. [AGLW14], die eine Nutzenfunktion aufstellen, um die operationalen Kosten für Speicherkapazität, Datenrate und Nutzungsdauer zu minimieren. Auch die Formulierung als Bedingungserfüllungsproblem (*engl. „Constraint Satisfaction Problem“ (CSP)*) wird angewendet. Baur et al. [BSG+18], zum Beispiel, führen eine Sprache ein um anwenderspezifische Bedingungen, etwa in Bezug auf den benötigten Arbeitsspeicher oder spezifische Ländereinschränkungen, in denen das Datencenter sein darf, zu formulieren. Auch Optimierungsverfahren wie lineare Programmierung [SS16] oder Greedy-Algorithmen [DGRB15] werden zur Minimierung der Deploymentkosten angewendet. Neben den Deploymentkosten spielt vor allem in industriellen Anwendungsszenarien der Datenfluss eine wichtige Rolle bei der Bestimmung der optimalen Bereitstellungsumgebung [ZBK+20]. So sollte, zum Beispiel, die Aggregation von Sensordaten in einer Fabrik möglichst nah an den Datenquellen geschehen, während Langzeitanalysen auch im zentralen Datencenter durchgeführt werden können. Die aufgeführten Arbeiten sind nur beispielhaft genannt, es gibt eine Vielzahl an Arbeiten, die sich mit der optimalen Selektion von Cloud-Anbietern und -Diensten beschäftigen. Dies wird im Rahmen dieser Arbeit jedoch nicht weiter vertieft, sondern es werden verschiedene Konzepte betrachtet, die die Portabilität von Anwendungen zwischen verschiedenen Cloud-Anbietern und -Diensten ermöglichen. Die Selektionsverfahren können mit den im Folgenden erläuterten Konzepten zur Portabilität kombiniert werden, wie von Baryannis et al. [BGK+13] gezeigt, die TOSCA zur Modellierung und CSP zur Selektion verwenden.

Die Voraussetzung, um die Bindung an einen Cloud-Anbieter (*engl. „Vendor-Lock-In“*) zu vermeiden, ist die Portabilität der Anwendungen. Die Portabilität zwischen Clouds stellt sicher, dass eine Anwendung oder ein Service unabhängig von dem zugrundeliegenden Cloud-Dienst auf die gleiche Weise funktioniert [PV14]. Petcu und Vasilakos [PV14] unterscheiden drei Kategorien für Lösungen des Portabilitätsproblems: (i) Verwendung einer Abstrakti-

onsschicht und Adaptern, (ii) Anwendung von Semantik und modellbasierten Lösungen und (iii) Anwendung von Standards. Die Lösungen der verschiedenen Kategorien sind meist komplementär. Da verschiedene Standards in den Kategorien (i) und (ii) Anwendung finden, wird Kategorie (iii) im Folgenden nicht gesondert betrachtet.

(i) Abstraktionsschicht und Adapter: Für die Entwicklung von portablen Anwendungen gibt es verschiedene Bibliotheken für verschiedene Programmiersprachen, wie jclouds für Java [The14] oder Libcloud [The22a] und CloudBridge [GLTA16] für Python, die eine Abstraktion für die Cloud-Dienste verschiedener Cloud-Anbieter bieten. Diese Bibliotheken bieten codebasierte Modelle für die Selektion eines spezifisches Cloud-Dienstes. Mechanismen für das flexible Deployment und Management fehlen jedoch.

Einen plattformbasierten Ansatz bietet zum Beispiel die mOSAIC-Plattform [MŞP14; PMPC13]. Sie ist eine Middleware, die die Entwicklung und Bereitstellung von portablen Anwendungen unterstützt. Die Plattform bietet unter anderem Anwendungsentwicklerinnen und -entwicklern Bibliotheken in unterschiedlichen Programmiersprachen, um Anwendungen unabhängig des jeweiligen Cloud-Anbieters zu implementieren. Die nativen APIs der verschiedenen Cloud-Anbieter sind durch einen entsprechenden Wrapper (*Driver API*) vereinheitlicht. Sie werden zur Deploymentzeit benötigt, um die Anwendung bereitzustellen. Eine *Service-Discoverer*-Komponente ermöglicht die Selektion eines geeigneten Cloud-Anbieters zur Deploymentzeit. Die Plattform unterstützt jedoch nur eine asynchrone ereignisgesteuerte Kommunikation zwischen den Komponenten.

Zusätzlich zu konkreten Lösungen gibt es Standardisierungsbemühungen, um die Heterogenität der APIs der Cloud-Anbieter zu reduzieren. Dazu zählen unter anderem der *Cloud Infrastructure Management Interface (CIMI)*-Standard [DMT12], der eine REST-basierte Schnittstelle für IaaS-Dienste spezifiziert. Ebenfalls eine REST-Schnittstelle definiert der *Open Cloud Computing Interface (OCCI)*-Standard [OGF09] der OGF, welcher jedoch für verschiedenen Cloud-Service-Modelle anwendbar ist. Speziell für

Managementaktivitäten für PaaS-Dienste definiert der *Cloud Application Management for Platforms (CAMP)*-Standard [OAS12] der OASIS eine einheitliche API und ein Paketformat, um Interoperabilität zwischen den verschiedenen Diensten zu ermöglichen. Eine Realisierung des CAMP-Standards ist Apache Brooklyn [The20], eine Deployment-Engine, die das Paketformat von CAMP unterstützt und die Bereitstellung auf unterschiedlichen PaaS-Diensten ermöglicht. Der CAMP-Standard bietet mit dem definierten Paketformat auch eine modellbasierte Lösung, und in verschiedenen Arbeiten wurde gezeigt, wie TOSCA-Modelle in CAMP-Modelle transformiert werden können [ALKH17; CCP15; CCPD16]. Alexander et al. [ALKH17] nutzen TOSCA dabei für das PIM und CAMP für das PSM. Die verschiedenen Konzepte sind also häufig komplementär und können teilweise auch beiden Kategorien zugeordnet werden.

(ii) Semantik und modellbasierte Lösungen: Im Fokus dieser Arbeit stehen vor allem die modellbasierten Lösungen, die eine flexible und adaptive Modellierung des Deployments von Anwendungen ermöglichen. Die grundsätzlichen Adaptionsmechanismen, die verschiedene Modellierungssprachen bieten, wurden bereits in Abschnitt 2.1.2 erläutert. Im Folgenden werden konkrete modellbasierte Verfahren vorgestellt. Viele der modellbasierten Lösungen nutzen TOSCA als Modellierungssprache und präsentieren unterschiedliche Konzepte für das Verteilen der Komponenten. Brogi et al. [BNRS18] und Hirmer et al. [HBBL14] führen Konzepte zur *Vervollständigung* unvollständiger TOSCA-Modelle basierend auf dem Abgleich von Anforderungen (*Requirements*) und Fähigkeiten (*Capabilities*) der Komponenten ein, wobei Brogi et al. [BNRS18] ausschließlich containerbasierte Anwendungskomponenten betrachten. Es werden nur die sogenannten *anwendungsspezifischen* Komponenten, deren Abhängigkeiten und deren Anforderungen spezifiziert und über ein Selektionsverfahren die Bereitstellungsumgebungen selektiert. Das TOSCA-Modell wird durch die selektierten Komponenten ergänzt und die Anforderungen werden durch entsprechende Hosting-Beziehungen aufgelöst.

Képes, Breitenbücher und Leymann [KBL17] führen TOSCA-*Templates* ein, welche die Struktur eines Deployment-Modells für einen bestimmten Anwendungstyp in einer bestimmten Umgebung mit bestimmten Anforderungen in Bezug auf die bereitgestellten Managementfunktionalitäten vorgeben. Entsprechend den Anforderungen des Nutzers kann ein Template selektiert und die paketierte Anwendung zum Deployment-Modell hinzugefügt und bereitgestellt werden. Ähnliche templatebasierte Verfahren, die jedoch nicht TOSCA als Modellierungssprache verwenden, werden auch von Antequera et al. [ACCM17], Arnold et al. [AEK+08] und Pfitzmann und Joukov [PJ11] eingeführt. Pfitzmann und Joukov [PJ11] bieten Templates mit verschiedenen VM-Images, welche basierend auf den Typen der Anwendungskomponenten und deren Abhängigkeiten selektiert werden können, um Anwendungen in die Cloud zu migrieren. Ein ähnliches Verfahren wird von Arnold et al. [AEK+08] eingesetzt. Sie stellen abstrakte Modellfragmente als Muster zur Verfügung, die zur Vervollständigung von Modellen verwendet werden können, indem virtuelle Komponenten auf bestehende Komponenten abgebildet werden. Diese templatebasierten Verfahren bieten keine Flexibilität bei der Verteilung der Komponenten, da sie auf vorbereiteten Templates basieren. Antequera et al. [ACCM17] kombinieren einen templatebasierten Ansatz mit einer flexiblen bedarfsgerechten Vervollständigung von Deployment-Modellen. Basierend auf definierten Anforderungen können passende existierende Templates, die Cloud-Ressourcen und deren Konfiguration definieren, identifiziert werden. Existiert kein passendes Template, wird ein neues konstruiert, welches den Anforderungen entspricht. In einer Erweiterung wurde dieser Ansatz auch mit TOSCA umgesetzt [ACCM19].

Die Verteilung der Komponenten auf verschiedene Cloud-Anbieter und konkrete geographische Regionen kann auch explizit mit TOSCA modelliert werden [BCS17; CDP17; WWB+13]. Für bestimmte Komponenten werden *Location Policies* definiert, welche geographische Einschränkungen für das Deployment definieren. Diese Policies werden auch genutzt, um die nicht-funktionalen Eigenschaften der Komponenten, die bestimmte Cloud-Dienste repräsentieren, zu spezifizieren. Basierend auf diesen Annotationen können damit passende Cloud-Dienste selektiert werden, auch für das dynamische

Redeployment zur Laufzeit [CDP17].

Ein musterbasiertes Verfahren, welches auch zur Portabilität von Anwendungen eingesetzt werden kann, wurde von Harzenetter et al. [HBF+18] eingeführt. Im Deployment-Modell, umgesetzt mit TOSCA, können Cloud-Muster als abstrakte Komponenten modelliert werden, um zum Beispiel das Hosting auf einer Public-Cloud auszudrücken. Durch ein Transformationsverfahren kann diese abstrakte Muster-Komponente durch ein passendes Fragment substituiert werden, welches die konkreten Komponenten für das Deployment, zum Beispiel bei AWS, enthält. Ein ähnliches Transformationsverfahren wird auch von Eilam et al. [EKK+06] angewendet, um ein logisches Deployment-Modell mit abstrakten Komponenten in ein konkretes Modell zu transformieren.

Ein MDA-inspiriertes Verfahren nutzt das MODAClouds-Projekt [ADM+12; FAS17b], welches, wie bereits in Abschnitt 2.1.2 genannt, CloudML für die Modellierung von Deployment-Modellen verwendet. Der Ansatz von MODAClouds ermöglicht die teilautomatisierte Transformation von CIM zu CPIM und CPSM. CloudML wird dabei für das CPIM und CPSM eingesetzt und mit QoS-Anforderungen angereichert, welche die Selektion von passenden Cloud-Anbietern und -Diensten leiten. Die konkreten Transformationsschritte und resultierenden Modelle werden jedoch in den Arbeiten nicht erläutert.

In verschiedenen Arbeiten werden auch semantische Modelle eingesetzt, um die Portabilität von Anwendungen zu ermöglichen [CD15; QHRD13; QRD16]. Die semantische Engine der mOSAIC-Plattform und die dazugehörige Ontologie können von Entwicklern genutzt werden, um passende Cloud-Anbieter und Cloud-Dienste zu selektieren [CD15]. Verfügbare Ressourcen und Komponenten können so semantisch annotiert und dadurch identifiziert werden. In einer anderen Arbeit werden Feature-Modelle, die die Funktionalitäten einer spezifischen Cloud spezifizieren, mit einer Ontologie verknüpft [QHRD13; QRD16]. Die Ontologie definiert dabei die gemeinsamen Konzepte aller Feature-Modelle und ermöglicht den Vergleich zwischen den möglicherweise semantisch unterschiedlichen Modellen, welche zur Selektion eines passendes Cloud-Anbeiters genutzt werden.

Die Verfahren beschränken sich dabei nicht nur auf die Entwurfs- und

Deploymentzeit, wie bereits bei dem Ansatz von Carrasco, Durán und Pimentel [CDP17] erläutert. Darüber hinaus gibt es weitere modellbasierte Ansätze speziell für die Migration von Anwendungen zur Laufzeit [BBKL13b; NB19]. Breitenbücher et al. [BBKL13b] und Nabavi und Bushehrian [NB19] generieren Migrationspläne basierend auf einem deklarativen Deployment-Modell unter Berücksichtigung der Deployment-Reihenfolge der Komponenten. Während Nabavi und Bushehrian [NB19] ausschließlich die Migration ermöglichen, führen Breitenbücher et al. [BBKL13b] einen generischen Ansatz ein, der die Anwendung von beliebigen Management-Mustern ermöglicht. Nabavi und Bushehrian [NB19] berücksichtigen auch verschiedene Kommunikationsmuster [Men14], die bei der Migration angewendet werden müssen, um die Kommunikation zwischen Komponenten wiederherzustellen, wenn diese auf unterschiedlichen Clouds verteilt bereitgestellt werden.

2.1.3.2 Funktionale und nicht-funktionale Adaptionen

Die Bereitstellung von Komponenten in Multi-Cloud-Umgebungen bedarf häufig der Anpassung der Kommunikation zwischen den Anwendungskomponenten oder der Komponenten selbst [JPCL14; Men14; MMGC12]. Sollen die ursprünglichen Komponenten nicht angepasst werden, werden häufig Adapter eingesetzt, um technologische Unterschiede zu kapseln. Für die Integration von Komponenten in Multi-Cloud-Umgebungen führen Fehling et al. [FLR+14] Muster ein, wobei auch speziell hybride Clouds berücksichtigt sind. Häufig wird zur Integration von verteilten Komponenten in hybriden Clouds eine nachrichtenbasierte Kommunikation verwendet. Die Muster von Fehling et al. [FLR+14] sind dabei eng verbunden mit den Mustern zur generellen Integration von Anwendungssystemen [HW04].

Basis zur Erkennung des Adaptionsbedarfs ist die Validierung der Anwendung. Das Werkzeug Sommelier [BDS18] ermöglicht die Validierung von TOSCA-Modellen in Bezug auf die Relationen zwischen Komponenten. Dabei wird geprüft, ob alle Bedingungen, die durch die Anforderungen und Fähigkeiten der verbundenen Komponenten definiert sind, erfüllt sind. Globale Anforderungen oder Einschränkungen werden hier jedoch nicht berücksich-

tigt und auch wird keine automatisierte Adaption angestoßen. Cuadrado, Duenas und Garcia-Carmona [CDG12] kombinieren die Validierung und automatisierte Anpassung eines Systems. Zur Prüfung der Korrektheit eines Systems wird zwischen Stabilität *(Stability)* und Erwünschtheit *(Desirability)* unterschieden, wobei Stabilität die Erfüllung von Abhängigkeiten und technischen Bedingungen der Komponenten gegenüber den Ressourcen darstellt und Erwünschtheit die Erfüllung von Zielen der verfügbaren Ressourcen, welche wichtig für die Erbringung der erwünschten Funktionalität sind. Im Deployment-Modell werden Anforderungen und Bedingungen der Komponenten, hier *Deployment Units* genannt, in Bezug auf passende Ressourcen zum Hosting definiert. Diese Bedingungen können dabei sowohl die Existenz als auch Nicht-Existenz bestimmter Ressourcen oder Eigenschaften spezifizieren. Die Bedingungen werden aus dem Modell extrahiert und als prädikatenlogische Formeln definiert. Basierend darauf kann die Korrektheit zur Laufzeit evaluiert werden und mittels SAT-Solver eine korrekte Konfiguration ermittelt werden. Anschließend werden Aktivitäten abgeleitet, die durchgeführt werden müssen, um die Anwendung in die korrekte Konfiguration zu überführen. Der Fokus liegt vor allem auf der Validierung der Ressourcen und nicht auf den Kommunikationsbeziehungen zwischen den Komponenten.

Für die Erweiterung eines bestehenden Deployment-Modells um Services, wie Monitoring- oder Sicherheits-Services, führen Palesandro et al. [PLB+17] die *TOSCA Manipulation Language (TML)* ein. Ein TML-Skript spezifiziert *Filter*, die Regeln definieren, wann und auf welche Komponenten die definierten *Aktionen* angewendet werden dürfen. Zuletzt validieren die *Checks*, ob das resultierende Modell valide ist zum erwarteten Ergebnis. Des Weiteren wird die Transformation in OpenStack Heat Templates ermöglicht, die generelle Transformation von TOSCA-Modellen in andere Technologien wird jedoch nicht wie von Wurster et al. [WBH+20c] systematisch untersucht. Einen ähnlichen Ansatz zur Adaption von Deployment-Modellen verfolgen Holmes und Zdun [HZ18], die die Transformationssprache *Epsilon Wizard Language (EWL)* nutzen, um Regeln zur Anwendung einer Transformation und entsprechende Aktionen zu definieren. Damit können globale

Anforderungen des Nutzers, wie zum Beispiel die Absicherung aller öffentlich erreichbaren Server durch einen Paketfilter, in einem Deployment-Modell umgesetzt werden. Bei Sun, Gray und White [SGW15] werden die Transformationregeln nicht spezifiziert, sondern die Aktivitäten eines Nutzers zur Adaption des Deployments werden aufgezeichnet und generalisiert bereitgestellt. Für die generelle Anwendbarkeit werden zu den Aktivitäten noch Vorbedingungen definiert, die wie bereits bei den vorherigen Modellen die Anwendbarkeit der Transformation spezifizieren. Dabei werden *strukturelle* und *attributsspezifische* Vorbedingungen unterschieden.

Die eingeführten Konzepte können zum Teil auch zur Realisierung der Portabilität eingesetzt werden, fokussieren aber die generelle Adaption von Deployment-Modellen und nicht primär die Portabilität. Für die in dieser Arbeit eingeführte Modelladaption zur Behebung von Problemen (Abschnitt 5.4) wird auch ein Verfahren zur Definition der Anwendbarkeit einer Lösung und der Spezifikation der eigentlichen Adaption eingesetzt. Der Fokus liegt dabei vor allem auf der Verknüpfung mit einer musterbasierten Modellanalyse zur Problemerkennung.

2.1.4 Zusammenfassung und Forschungsfragen

Deklarative Sprachen zur Modellierung des Deployments von Anwendungen haben sich in der Praxis gegen imperative Sprachen durchgesetzt und verschiedene Technologien und Standards mit jeweils eigenen DSLs wurden entwickelt. Zur Verhinderung eines Technologie-Lock-In wurden generische Modellierungssprachen eingeführt, um eine Transformation in verschiedene Technologien zu ermöglichen. Dabei hat EDMM gezeigt, dass es nicht nur kompatibel mit 13 verschiedenen Technologien ist, sondern auch auf eine Teilmenge der Modellelemente des TOSCA-Standards abgebildet werden kann. TOSCA wird vor allem in vielen wissenschaftlichen Arbeiten eingesetzt, um Modellierungskonzepte zum Portieren von Anwendungen und zur Adaption von Deployment-Modellen zu demonstrieren.

Mit der Portabilität von Anwendungen und der Selektion von geeigneten Cloud-Anbietern in Multi-Clouds haben sich bereits viele Arbeiten befasst.

Dabei werden jedoch Multi-Nutzer und organisatorische Aspekte nicht berücksichtigt, sondern es werden ausschließlich zentralisierte Ansätze unterstützt. Der zentralisierte Ansatz bezieht sich dabei nicht auf die Verteilung der Komponenten selbst, sondern auf die Verarbeitung des Deployment-Modells. Vor allem jedoch bei komplexen Anwendungen mit verschiedenen involvierten Organisationseinheiten müssen verschiedene Stufen der Modellierung unter Einbeziehung der involvierten Partner ermöglicht werden, um (i) globale *partnerübergreifende* und (ii) lokale *partnerspezifische* Informationen und Anforderungen im Deployment-Modell abzubilden.

Bei den partnerspezifischen Deployment-Modellen muss die konkrete Bereitstellung der Anwendungskomponenten des Partners berücksichtigt werden. Vorherige Arbeiten haben bereits gezeigt, dass Adaptionen des Deployments nötig sind, wenn die Anwendung von der Testumgebung in die Produktivumgebung portiert wird. Die Verfahren zur Portabilität beruhen auf definierten Templates oder der Vervollständigung beziehungsweise Transformation von partiell modellierten Deployment-Modellen, welche meist nur die anwendungsspezifischen Komponenten abbilden. Dabei wird Wissen aus der Entwicklungsphase, in der bereits Middleware- und Infrastrukturkomponenten eingesetzt wurden, verworfen.

Im Rahmen dieser Arbeit wird, um die genannten Probleme anzugehen, (i) ein Modellierungskonzept für globale partnerübergreifende Deployment-Modelle und lokale partnerspezifische Deployment-Modelle unter Einbeziehung verschiedener Partner basierend auf EDMM eingeführt und (ii) ein Restrukturierungsalgorithmus für lokale Deployment-Modelle zum Erhalt des Wissens aus vorherigen Deployments, zum Beispiel in der Testumgebung, entwickelt (Forschungsbeitrag 2). Das Modellierungskonzept bildet die Grundlage für die DivA-Methode und ermöglicht das Modellieren adaptiver Deployment-Modelle und deren organisatorische Verteilung auf verschiedene Partner sowie die ressourcenspezifische Verteilung auf spezifische Bereitstellungsumgebungen.

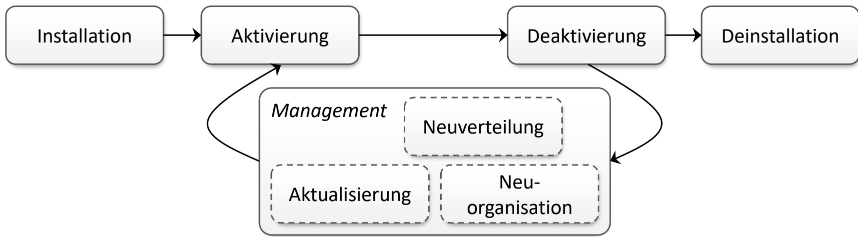

Abbildung 2.2: Deployment-Prozess und dessen Basisaktivitäten (angelehnt an [ABL15]).

2.2 Bereitstellung von Anwendungen

Die Bereitstellung und das Management von Anwendungen lassen sich als Prozess von der Installation bis zur Deinstallation definieren. Im Folgenden werden der grundlegende Prozess und die benötigten Operationen für das Deployment und Management vorgestellt. Des Weiteren wird die Kombination von deklarativen und imperativen Modellen für die Bereitstellung erläutert und es werden existierende deklarative Deployment-Technologien und ihre Eignung für ein partnerübergreifendes dezentrales Deployment analysiert und erste Ansätze für die dezentrale Koordination des Deployments diskutiert.

2.2.1 Deployment-Prozess und -Operationen

Arcangeli, Boujbel und Leriche [ABL15] definieren das Deployment einer Anwendung als Prozess von der Installation bis zur Deinstallation der Anwendung mit zusätzlichen Managementaktivitäten, die zur Laufzeit der Anwendung ausgeführt werden können. In Abbildung 2.2 ist der Deployment-Prozess mit den grundlegenden Aktivitäten dargestellt. Während der *Installation* werden die Komponenten bereitgestellt und konfiguriert. Die *Aktivierung* umfasst Operationen zum Starten der Komponenten, wobei die Aktivierung auch durch bestimmte Ereignisse getriggert werden kann. Umgekehrt werden bei der *Deaktivierung* die Komponenten gestoppt. Für be-

stimmte Managementaktivitäten ist die Deaktivierung nötig, um zum Beispiel eine Komponente zu aktualisieren oder eine Neuverteilung zu ermöglichen. Die *Deinstallation* entfernt die Komponenten und gibt die Ressourcen wieder frei. Zur Realisierung dieses Prozesses definiert der TOSCA-Standard Schnittstellen für den Lebenszyklus einer Komponente (*engl. „Lifecycle Operations“*) [OAS20]. Diese Schnittstellen müssen von allen Komponententypen implementiert sein und umfassen *create*, *configure*, *start*, *stop* und *delete*. Dieser Prozess ist Teil des *MAPE (Monitor, Analyze, Plan, Execute)*-Regelkreises und repräsentiert Aspekte der Ausführung (*Execute*). Zur Laufzeit können die Ressourcen und Anwendungskomponenten überwacht (*Monitor*) und analysiert (*Analyze*) werden. Werden die Anforderungen durch die Ressourcen nicht mehr erfüllt, kann ein neuer Selektionsprozess (*Plan*) angestoßen und ein neues Deployment oder eine Neuverteilung durchgeführt werden [WBB+17].

Die Ausführbarkeit einer Operation im Deployment-Prozess hängt dabei vom Zustand der Komponente ab [ABL15]. Auch die Abhängigkeiten zwischen den Komponenten müssen dabei berücksichtigt werden [ABL15; BBK+14; DMZZ14]. So müssen Infrastrukturkomponenten gestartet sein, bevor eine Komponente dort installiert werden kann. Ebenso kann eine Kommunikationsverbindung zu einer Komponente nur aufgebaut werden, wenn die Zielkomponente auch aktiv ist.

Die Deployment-Aktivitäten können deklarativ modelliert werden, da sie *zustandsmodifizierende* Aktivitäten sind [Bre16]. Der gewünschte Zielzustand kann modelliert, durch ein Deployment-System interpretiert und das Anwendungssystem in den gewünschten Zielzustand überführt werden. Auch unter den Managementaktivitäten gibt es zustandsmodifizierende Aktivitäten, jedoch sind einige auch *zustandserhaltende* Aktivitäten, wie zum Beispiel das Backup eines Systems oder die Durchführung von Tests [Bre16; HBL+19]. Für zustandserhaltende Aktivitäten werden deshalb häufig imperative Ansätze verwendet. Im Rahmen dieser Arbeit werden vor allem die Deployment-Aktivitäten bei partnerübergreifenden Anwendungen berücksichtigt und dabei die Installation und Aktivierung fokussiert.

2.2.2 Kombination deklarativer und imperativer Technologien

Die in Abschnitt 2.1.1 vorgestellten Modellierungsansätze für das Deployment und Management von Anwendungen haben verschiedene Vor- und Nachteile: Imperative Modelle sind hochgradig anpassbar, jedoch ist die Modellierung komplex, da der Kontroll- und Datenfluss zwischen allen Operationen berücksichtigt werden muss. Deklarative Modelle reduzieren die Komplexität der Modellierung, sind aber beschränkt in der Expressivität und Anpassbarkeit, da die auszuführenden Operationen vom Modell abgeleitet werden. Um die Vorteile beider Ansätze nutzen zu können, kombinieren verschiedene Arbeiten beide Ansätze, sowohl zum Deployment selbst als auch für das Management von Anwendungen [BBK+14; Bre16; CCDT18; EEKS11]. Dabei dient ein deklaratives Deployment-Modell als Eingabe, worauf basierend ein Workflow-Modell generiert wird. Bei der Generierung des Workflows werden (i) die auszuführenden Operationen und (ii) die Reihenfolge der Operationen berücksichtigt. Für die Workflow-Modelle werden unter anderem BPEL [BBK+14] und BPMN [CCDT18] verwendet. Ein imperatives Modell, welches alle grundlegenden Operationen umfasst, kann damit direkt aus dem deklarativen Modell abgeleitet und anschließend bei Bedarf angepasst werden. Für spezielle Managementaufgaben werden von Breitenbücher et al. [BBKL13b; BBKL14] und Breitenbücher [Bre16] Workflow-Fragmente bereitgestellt, deren Anwendbarkeit durch einen Detektor, der als deklaratives Modellfragment modelliert ist, bestimmt wird. Durch Subgraphisomorphie wird geprüft, ob das Workflow-Fragment auf ein gegebenes deklaratives Modell angewendet werden kann. Das deklarative Modell bestimmt auch die Reihenfolge, in der die Workflow-Fragmente ausgeführt werden müssen.

Alle Verfahren beruhen auf dem Orchestrierungsprinzip der Deployment- bzw. Managementoperationen, das bedeutet: ein zentraler Workflow kontrolliert die Ausführung und den Datenaustausch. Neben der Orchestrierung können auch Choreographien genutzt werden, bei denen die Kontrolle dezentral organisiert ist und nur die Schnittstellen zwischen den beteiligten Workflows global definiert sind. Für Choreographien können auch die be-

reits genannten Standards BPMN und BPEL [DKLW07] genutzt werden. Auch in der Domäne des Deployments und Managements gibt es choreographiebasierte Ansätze. Herry, Anderson und Rovatsos [HAR13] stellen einen planungsbasierten Ansatz zur Erzeugung einer Choreografie vor. Dabei wird ein Choreographie-Modell generiert, welches den Nachrichtenaustausch zwischen den beteiligten Agenten spezifiziert, die jeweils dafür zuständig sind, bestimmte Komponenten in einen definierten Zustand zu überführen. Auch Kopp und Breitenbücher [KB17] schlagen einen solchen Ansatz vor, um Deployment-Details, wie APIs oder Zugangsdaten, beim Deployment über mehrere private Clouds zu verbergen. Genau diese Anforderungen bestehen auch beim Deployment partnerübergreifender Anwendungen. Choreographien ermöglichen generell eine dezentrale Koordination. Im Folgenden werden die Koordination des Deployments bei verschiedenen Technologien beleuchtet und weitere Ansätze für eine dezentrale Koordination diskutiert.

2.2.3 Deployment-Koordination

Die Deployment-Systeme von den in Abschnitt 2.1.2 vorgestellten deklarativen Deployment-Technologien und Konzepten, darunter standardbasierte Lösungen wie OpenTOSCA oder Apache Brooklyn oder in Forschungsprojekten entwickelte Tools wie MODAClouds oder PaaSage, sowie die in der Praxis eingesetzten Technologien wie Terraform, Chef, Puppet, Ansible oder Kubernetes, unterstützen typischerweise eines der zwei Ausführungsmodelle: *zentrales* oder *agentenbasiertes* Deployment [WBB+17]. Beim *zentralen* Ausführungsmodell gibt es einen zentralen Manager, der die Deployment--Operationen ausführt, zum Beispiel via Secure Shell (SSH) Protokoll auf verteilten VMs. Dazu zählen unter anderem Ansible, Terraform oder die TOSCA-Implementierung OpenTOSCA. Beim *agentenbasierten* Deployment müssen vorab Agenten installiert werden, die lokal die Operationen ausführen. Weerasiri et al. [WBB+17] nennen dieses Ausführungsmodell *dezentrales* Deployment. Um eine Verwechslung mit dem dezentralen Deployment bei partnerübergreifenden Anwendungen zu vermeiden, wird diese Form der Ausführung im Folgenden *agentenbasiert* genannt. Ein zentraler Manager

delegiert dabei die Operationen an den jeweiligen Agenten, wobei jeder Agent nur die für ihn relevanten Operationen kennt. Unter anderem Puppet und Chef folgen diesem Ausführungsmodell. Beim agentenbasierten Modell wird zwar die Ausführung selbst dezentral durchgeführt, die Kontrolle liegt jedoch beim zentralen Manager.

Vor allem im IoT-Bereich findet das agentenbasierte Modell Anwendung, da die Zahl der zu verwaltenden Geräte häufig sehr hoch ist [VSI+15] oder auch in verteilten Anwendungsszenarien kein Zugriff auf alle Geräte von einem zentralen Manager möglich ist [KBL+19]. In Zusammenarbeit mit Képes et al. [KBL+19] wurde ein Konzept entwickelt, um das Deployment und Management von Anwendungskomponenten zu ermöglichen, die verteilt über verschiedene Umgebungen mit beschränkten Zugriffsrechten von außen bereitgestellt werden müssen, wie es zum Beispiel in Smart-Home-Szenarien der Fall ist, wenn mehrere Häuser involviert sind und neue Komponenten in den Häusern von einem Anbieter ausgerollt werden sollen. Basierend auf der TOSCA-Implementierung OpenTOSCA wurde ein Konzept entwickelt, welches die Bereitstellung einer Deployment-System-Instanz in jeder Umgebung vorsieht. In einer Peer-to-Peer-Architektur kommunizieren die Instanzen via nachrichtenbasierter Kommunikationstechnologie. Dabei implementieren sie das Publish-Subscribe-Muster [HW04]. Das Deployment wird hier jedoch auch von einer einzelnen Instanz initiiert und die Operationen, die von den jeweiligen anderen Instanzen ausgeführt werden müssen, werden delegiert. Bei jedem Deployment agiert somit eine Instanz als zentraler Manager und die übrigen Instanzen als Agenten. Jede beteiligte Instanz kennt außerdem alle verfügbaren Infrastrukturkomponenten in allen beteiligten Umgebungen. Die Rolle der jeweiligen Instanz ist dynamisch und kann dabei von Deployment zu Deployment variieren. Die Kontrolle über das Deployment liegt auch hier bei einem zentralen Manager. Ein ähnliches Verfahren wird auch von Sotiriadis, Bessis und Antonpoulos [SBA12] zur Selektion passender Infrastrukturkomponenten in verschiedenen Rechenzentren basierend auf benutzerspezifischen QoS-Anforderungen eingesetzt. Dabei wird ein Meta-Broker für jede Anfrage eines Nutzers generiert, der die Koordination mit den Brokern der teilnehmenden Rechenzentren übernimmt.

Speziell mit der Bereitstellung von organisationsübergreifenden Ad-hoc-Anwendungen beschäftigen sich Van Hoye et al. [VWDV22]. Sie beschränken sich dabei auf containerisierte Anwendungen, welche in einem gemeinsamen Kubernetes Cluster durch einen zentralen Manager bereitgestellt werden. Die Autoren argumentieren, dass für zeitkritische Bereitstellungen, wie es bei den von ihnen betrachteten Ad-hoc-Anwendungen der Fall ist, ausschließlich ein zentral gesteuertes Deployment in Frage kommt. Nicht mit organisationsübergreifenden sondern mit teamübergreifenden Deployments beschäftigt sich Sokolowski [Sok21]. Im Kontext funktionsübergreifender DevOps-Teams wird das unabhängige Deployment der einzelnen Komponenten durch die verschiedenen Teams ermöglicht, wobei jedoch Abhängigkeiten zwischen den Komponenten berücksichtigt werden. Jedes Team kann für ihre Komponenten ein eigenes Deployment-Modell spezifizieren, welches Abhängigkeiten zu Komponenten anderer Teams definiert. Wenn die Bereitstellung einer Komponente gestartet wird, die von anderen, noch nicht bereitgestellten Komponenten abhängt, wird die Bereitstellung pausiert, bis diese Komponente verfügbar ist. Das vorgestellte μS-Deployment-System zielt darauf ab, die Abhängigkeiten zwischen den Teams zu reduzieren, gleichzeitig aber die Funktionsfähigkeit der gesamten Anwendung nicht zu gefährden. Die Bereitstellung kann damit dezentral initiiert werden, technisch basiert die Lösung jedoch auf einem zentralen Deployment-System, welches die verschiedenen Deployment-Modell verarbeitet. Für gleichberechtigte Partner, die (i) ihre Infrastrukturkomponenten nicht offenlegen und (ii) die Kontrolle nicht Dritten überlassen wollen, sind diese Ansätze jedoch nicht ausreichend.

Einen kombinierten zentralen und dezentralen Ansatz führen Etchevers et al. [ESB+17] für das Deployment von VMs und dort laufenden Anwendungskomponenten ein. Basierend auf einem deklarativen Deployment-Modell werden von einem zentralen Manager erst alle benötigten VMs bereitgestellt, wobei jede VM mit einem lokalen Konfigurator ausgestattet ist. Die lokalen Konfiguratoren (i) installieren die lokalen Komponenten auf ihrer VM und (ii) koordinieren die Anbindung der Komponenten untereinander. Jeder Konfigurator kennt die Abhängigkeiten zu anderen lokalen sowie entfernten

Komponenten und schickt an alle entfernten Konfiguratoren, die Komponenten verwalten, die eine Verbindung zu einer Komponente des lokalen Konfigurators aufbauen, Informationen wie IP-Adresse und Port und eine Nachricht, wenn die entsprechende Komponente gestartet ist. Erst dann kann der entfernte Konfigurator die abhängige Komponente ebenfalls starten. Die Koordination zwischen den Konfiguratoren ist komplett dezentral und basiert auf nachrichtenbasierter Kommunikation. Jedoch ist der Ansatz auf VM-basierte Deployments, welche darüber hinaus weiterhin zentral verwaltet werden, beschränkt.

Ein ähnliches Vorgehen nutzen auch Sebrechts et al. [SVV+16; SVW+18]. Für jeden Service wird ein *Service-Agent* bereitgestellt, der das Management des entsprechenden Service übernimmt. Für Anwendungen, die mehrere Services umfassen, übernimmt ein *Controller* die Aufgabe eines zentralen Managers, stellt für jeden Service einen Service-Agent bereit und teilt das Deployment-Modell in Fragmente für jeden Service-Agenten auf. Die Konfiguration der verschiedenen Services erfolgt dann dezentral durch die Service-Agenten, die Informationen zur Anbindung der Services untereinander austauschen. Ebenso wie bei dem Konzept von Etchevers et al. [ESB+17] erfolgt nur die Konfiguration der Anwendungskomponenten dezentral. In beiden Fällen wird ein zentraler Manager benötigt, der die Konfiguration der einzelnen Komponenten bzw. Services vorgibt.

2.2.4 Zusammenfassung und Forschungsfragen

Das Deployment von Anwendungssystemen folgt einem definierten Prozess, der den Lebenszyklus einer Komponente während des Betriebs, von der Installation bis zur Deinstallation, abbildet. Bei deklarativen Deployment-Technologien sind für jeden Komponententyp definierte Operationen verfügbar, um eine Komponente des entsprechenden Typs in den vorgesehenen Zustand zu überführen, wobei die Abhängigkeiten zwischen den Komponenten berücksichtigt werden müssen. Die Expressivität und Anpassbarkeit von deklarativen Deployment-Modellen ist beschränkt, da nur solche Aktivitäten ausgeführt werden können, die deklarativ modelliert und

interpretiert werden können. Dies sind häufig zustandsmodifizierende Aktivitäten. Zustandserhaltende Aktivitäten können meist nur durch imperative Modelle adäquat modelliert werden.

Um die Vorteile von deklarativen und imperativen Deployment-Modellen nutzen zu können, gibt es Verfahren die beide Ansätze kombinieren und aus deklarativen Modellen imperative Modelle generieren, die bei Bedarf noch angepasst werden. Damit ist die Komplexität der Modellierung reduziert, indem deklarative Modelle genutzt werden, und gleichzeitig die Flexibilität für spezifische Managementaktivitäten durch die Anpassbarkeit der generierten imperativen Modelle gegeben. Häufig kommen dabei Modellierungssprachen wie BPMN und BPEL zum Einsatz. Dabei werden zwei Modellarten für die Komposition der Operationen und Services zur Durchführung des Deployments eingesetzt: Orchestrierung und Choreographien. Bei der Orchestrierung koordiniert ein zentraler Workflow den Kontroll- und Datenfluss, während bei der Choreographie verschiedene Workflows die lokale Durchführung übernehmen und nur die Schnittstellen global definiert sind und die Komposition dezentral organisiert ist.

Generell kann man bei der Deployment-Koordination zwischen dem zentralen und dem agentenbasierten Ausführungsmodell unterschieden, wobei auch beim agentenbasierten Modell ein zentraler Manager die Initialisierung übernimmt. Bisher gibt es kein dezentrales Verfahren, welches (i) die Trennung von globalen und lokalen Informationen in deklarativen Deployment-Modellen und (ii) eine vollständig dezentrale Deployment-Koordination ohne zusätzliche zentrale Kontrolleinheit ermöglicht. Beide Anforderungen sind besonders in industriellen Anwendungsfällen relevant, um Datensicherheit zu gewährleisten und Zugriffsbeschränkungen zu berücksichtigen und trotzdem das Deployment verteilter partnerübergreifender Anwendungen zu ermöglichen. In dieser Arbeit wird ein Konzept eingeführt, welches basierend auf der Kombination von deklarativen und imperativen Modellen die dezentrale Koordination des Deployments ermöglicht und dabei nur global relevante Informationen zwischen den beteiligten Partnern teilt (Forschungsbeitrag 4).

2.3 Anwendbarkeit und Anwendung von Mustern

Das Konzept, architektonisches Wissen und Best Practices für wiederkehrende Probleme als *Muster* (*engl. „Patterns“*) zu sammeln und strukturiert zu dokumentieren, wurde von Alexander, Ishikawa und Silverstein [AIS77] im Bereich des Hausbaus und der Städteplanung eingeführt. Dieses Konzept wurde in unterschiedlichen Bereichen der Informatik, aber auch in anderen Domänen, wie dem Maschinenbau, der Pädagogik oder Film- und Theaterkostümen übernommen, um Lösungswissen zu dokumentieren und Anwenderinnen und Anwender zu unterstützen. In der Informatik beschreiben *Architekturmuster* Systemstrukturen und spezifizieren die Interaktion zwischen Subsystemen, wie zum Beispiel die Muster zur Integration von Anwendungskomponenten (*engl. „Enterprise Integration Patterns“ (EIP)*) [HW04], zu Sicherheitsarchitekturen [SFH+06] oder zur Realisierung und Management von Cloud-Anwendungen [FLR+14]. *Entwurfsmuster* hingegen adressieren konkrete Entwurfsprobleme und die technische Realisierung zum Beispiel im Bereich der objektorientierten Programmierung [GHJV94]. Diese Architektur- und Entwurfsmuster beschreiben bewährte Lösungen für wiederkehrende Probleme unabhängig von einer bestimmten Technologie oder Programmiersprache. Sie bieten generische Lösungen, die auf eine Vielzahl von Anwendungsfällen auf unterschiedliche Weise angewendet werden können. Neben diesen *agnostischen* Mustern gibt es auch *proprietäre* Muster [ME16], wie sie unter anderem von Cloud-Anbietern wie AWS [Ama22a] angeboten werden, die Lösungen mit konkreten Technologien oder Cloud--Diensten dokumentieren.

Ein Muster beschreibt eine bewährte Lösung für ein wiederkehrendes Problem, zum Beispiel die Integration von verteilten Komponenten durch eine vertrauenswürdige Drittpartei. Muster existieren jedoch nicht unabhängig voneinander, sondern sind miteinander verknüpft: Ein Muster kann zum Beispiel eine Verfeinerung [FBB+15] oder eine Alternativlösung zu einem anderen Muster sein oder häufig mit einem anderen Muster kombiniert werden [FLR+14]. Die Muster und deren Relationen, die die semantischen Zusammenhänge dokumentieren, bilden eine *Mustersprache* (*engl. „Pattern*

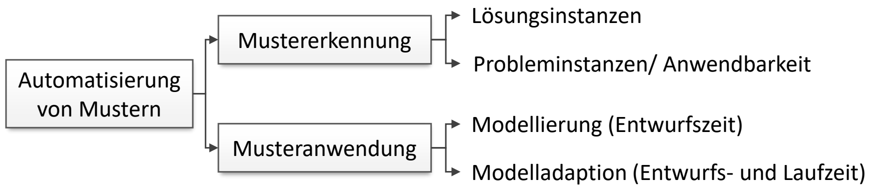

Abbildung 2.3: Verschiedene Bereiche in denen Automatisierungskonzepte von Mustern zum Einsatz kommen.

Language“), mit deren Hilfe Kombinationen von Mustern zur Lösung komplexer Probleme identifiziert werden können.

Muster sind primär für Menschen dokumentiert und in den genannten Büchern, auf Webseiten oder in Repositorien wie dem PatternAtlas [LB21] veröffentlicht. Die Dokumentation des Domänenwissens basiert auf einem Musterformat, das die Struktur für die Musterbeschreibung vorgibt. Auch wenn diese Formate von Mustersprache zu Mustersprache leicht variieren, sind die wesentlichen Teile eines Musterformats ähnlich: Jede der genannten Mustersprachen aus dem Bereich der Softwareentwicklung und -architektur verwendet ein Musterformat, das unter anderem das *Problem*, den *Kontext*, in dem das Problem auftritt, und die *Lösung* beschreibt. Diese wesentlichen Bestandteile eines Musterformats sind auch in den Richtlinien zum Dokumentieren von Mustern von Meszaros und Doble [MD97] und Wellhausen und Fiesser [WF12] enthalten.

Die Vielzahl an Mustern und die nicht-maschinenlesbare Dokumentation des Wissens erschwert die Erkennung und Anwendung von Mustern. In verschiedenen Bereichen wurden deshalb Konzepte zur Automatisierung von Mustern entwickelt. In dieser Arbeit werden diese Konzepte in verschiedene Kategorien eingeteilt, wie in Abbildung 2.3 dargestellt. Im Bereich der *Mustererkennung* werden Muster zum einen zur Erkennung von Lösungsinstanzen eines Musters in Architektur- oder Entwurfsmodellen und zum anderen zur Erkennung von Probleminstanzen, das heißt der Anwendbarkeit der Lösung eines Musters, eingesetzt. Eine Instanz ist dabei eine konkrete Umsetzung des Musters in einem Architektur- oder Entwurfsmodell. Im

Bereich der *Musteranwendung* werden Muster sowohl zur Modellierung von Architektur- und Entwurfsmodellen verwendet als auch zur Adaption bestehender Modelle. Adaptionen können dabei sowohl zur Entwurfs- als auch zur Laufzeit musterbasiert vorgenommen werden. Im Folgenden werden verschiedene Konzepte aus den zwei Bereichen näher betrachtet und die existierenden Ansätze und deren Anwendbarkeit für das Erkennen und Beheben von Problemen in Deployment-Modellen diskutiert.

2.3.1 Mustererkennung

Die automatisierte Erkennung von Mustern in Architektur- oder Entwurfsmodellen kann verschiedene Bereiche der Softwareentwicklung unterstützen [BP02]: Zum einen, um Realisierungen von Mustern, also Lösungsinstanzen, in einem Modell zu identifizieren oder auch Verbesserungsmöglichkeiten oder alternative Realisierungen des Musters vorzuschlagen oder die Einhaltung von Designregeln durchzusetzen. Zum anderen können Muster zur Lösung bestimmter Entwurfsprobleme vorgeschlagen und angewendet werden, indem die Anwendbarkeit geprüft wird, das heißt das Vorhandensein einer Probleminstanz des Musters in einem Modell. Die Voraussetzung für die automatisierte Mustererkennung in allen genannten Bereichen ist die Formalisierung der Muster, die, wie bereits erwähnt, vor allem in textueller Form vorliegen, in manchen Fällen ergänzt durch eine Skizze der Lösung. Die Formalisierung von Mustern sollte dabei als Ergänzung zu den existierenden textuellen Formen gesehen werde [TN03].

In Tabelle 2.2 ist ein Überblick über verwandte Arbeiten zur Mustererkennung in Modellen gegeben. Dabei werden die zwei Bereiche, Erkennung von *Lösungsinstanzen* und Erkennung von *Probleminstanzen* gesondert betrachtet. Die *Methode* gibt an, welches Verfahren zur Formalisierung der Muster angewendet wird, und die *Modellart*, auf welche Modellarten das Verfahren angewendet wird. In den verschiedenen Arbeiten werden auch unterschiedliche *Mustersprachen* verwendet, um die Verfahren zu demonstrieren. Dabei hängen Modellart und Mustersprache zusammen, da sie sich meist auf eine bestimmte Domäne beziehen. Generell wird von bisherigen Arbeiten vor

	Methode	Modellart	Mustersprache	Arbeiten
Lösungsinstanz	Graph-Matching	UML	Entwurfsmuster*	[KS07]
Lösungsinstanz	Graph-Matching	ARG	Entwurfsmuster*	[FZ11]
Lösungsinstanz	Graph-Matching	ARG	Architekturmuster	[Sar03]
Lösungsinstanz	Graph-Matching	TOSCA	Arbeitslastmuster	[TK16]
Lösungsinstanz	DSL-basiert	UML	Architekturmuster	[HZ15]
Lösungsinstanz	DSL-basiert	Graph	Entwurfsmuster*	[BF03]
Lösungsinstanz	Petrinetz-analysen	gefärbtes Petrinetz	Architekturmuster (EIP)	[FG13]
Lösungsinstanz	Logik erster Ordnung (Prolog)	OMT	Entwurfsmuster*	[KP96]
Lösungsinstanz	Logik erster Ordnung (Prolog)	UML	Entwurfsmuster*	[BP02; DE13; DL06; ME16; TN03]
Lösungsinstanz	Logik erster Ordnung (Prolog)	UML	Entwurfsmuster* & Antimuster	[PKG+00]
Lösungsinstanz	Logik erster Ordnung (Prolog)	UML	Performance-Antimuster	[CMT14]
Probleminstanz (Anwendbarkeit)	Graph-Matching	UML	Entwurfsmuster*	[FCSK03; KE07]
Probleminstanz (Anwendbarkeit)	Graph-Matching	Architektur-modell	Architekturmuster	[GL19]
Probleminstanz (Anwendbarkeit)	Graph-Matching	TOSCA	Cloud-Management-Muster (Laufzeit)	[BBKL13b; Bre16]
Probleminstanz (Anwendbarkeit)	DSL-basiert	Architektur-modell	Adaptionsmuster (Laufzeit)	[AB14a; AB14b]

Tabelle 2.2: Überblick verwandter Arbeiten und deren Verfahren zur Mustererkennung in Modellen. Die mit * gekennzeichneten Entwurfsmuster beziehen sich auf die Muster von Gamma et al. [GHJV94].

allem die Erkennung von Lösungsinstanzen und dabei besonders die Erkennung der Entwurfsmuster von Gamma et al. [GHJV94] und proprietären Architekturmustern fokussiert [HZ15].

2.3.1.1 Lösungsinstanzen von Mustern in Modellen erkennen

Es gibt mehrere Ansätze, die *Graph-Matching*-Verfahren verwenden, um Lösungsinstanzen in Modellen, die als gerichteter Graph spezifiziert sind, zu erkennen [FZ11; KS07; Sar03; TK16]. Kim und Shen [KS07] nutzen dafür eine UML-basierte Spezifikationssprache, die *Role-Based Metamodeling Language (RBML)* zur Spezifikation der statischen Elemente eines Entwurfsmusters. Damit können strukturelle Gleichheit mit UML-Klassendiagrammen überprüft und Instanzen des Musters identifiziert werden. Bei Fontana und Zanoni [FZ11] und Sartipi [Sar03] werden der Quellcode der Anwendung und die Muster als *Attributed Relational Graph (ARG)* dargestellt. Beide Verfahren verlangen kein exaktes Matching der Graphen, sondern ermöglichen durch einen zusätzlichen Klassifizierer [FZ11] bzw. ein heuristisches Suchverfahren [Sar03] die Identifikation von alternativen Realisierungen der Muster. In der Cloud-Domäne nutzen Tsigkanos und Kehrer [TK16] Graph-Matching um Arbeitslastmuster (*engl. „Workload Patterns“*) in TOSCA-Modellen zu erkennen, die als Bigraph formalisiert sind.

Auch die *DSL-basierten* Konzepte basieren auf einem Matching-Verfahren. Für die Repräsentation der Muster werden jedoch spezifische DSLs eingeführt, die die Basis für die Identifikation in einem Modell bilden. Haitzer und Zdun [HZ15] spezifizieren die Muster in einer eigenen DSL und transformieren UML-Klassendiagramme in eine eigene abstrakte Architektursprache, um die Existenz der Muster in dem jeweiligen Modell zu prüfen. Die Basis des Verfahrens sind sogenannte *Muster-Primitive* (*engl. „Pattern Primitives“*). Muster-Primitive sind immer wiederkehrende Bestandteile der Beschreibung von Musterlösungen und spezifizieren eine klare Semantik für die Elemente der Musterlösung [ZA05]. Sie sind die kleinste Einheit zur Beschreibung der Muster und dienen als fundamentale Modellelemente in den eingeführten DSLs. Kamal und Avgeriou [KA08] identifizieren ebenfalls Muster-Primitive

zur Modellierung von Lösungen von Architekturmustern als UML-Diagramme. Balanyi und Ferenc [BF03] führen die *Design Pattern Markup Language (DPML)* zur Modellierung von Entwurfsmustern ein. Der Quellcode einer Anwendung wird geparst und als abstrakter Graph repräsentiert. Die in DPML beschriebenen Muster werden als Standard XML-DOM-Baum geladen und ähnlich einem Graph-Matching wird das Matching zwischen den Klassen und deren Relationen geprüft. Sowohl die Graph-Matching- als auch die DSL-basierten Verfahren stützen sich auf die Prüfung von struktureller Gleichheit zweier Modellfragmente. Die Expressivität ist dabei auf die Erkennung der Existenz bestimmter Strukturmuster beschränkt.

Zur Analyse von Architekturmodellen von Middleware-Systemen zur Integration von Anwendungskomponenten, die die EIP-Mustersprache [HW04] anwenden, nutzen Fahland und Gierds [FG13] verschiedene Analyseverfahren für gefärbte Petrinetze (*engl. „Coloured Petri Nets“ (CPN)*). Der Fokus der Arbeit liegt auf der Formalisierung der EIP-Muster als CPN-Modelle. Damit können die Architekturmodelle als CPN dargestellt werden und etablierte Verfahren zur Analyse von Petrinetzen angewendet werden. Die Arbeit beschränkt sich dabei ausschließlich auf die Formalisierung von EIP-Mustern.

Eine weitere Kategorie bilden die Verfahren, die zur Formalisierung von Mustern Prädikate der *Logik erster Ordnung* verwenden. Eine der ersten Arbeiten in diesem Bereich zur Formalisierung der Entwurfsmuster stammt von Kramer und Prechelt [KP96], die die Formalisierung zur Erkennung von Lösungsinstanzen struktureller Entwurfsmuster in OMT Objekt-Diagrammen, welche Vorläufer der heute verbreiteten UML-Klassendiagramme sind, nutzen. Dabei werden auch die Muster als OMT Objekt-Diagramme repräsentiert und mithilfe von Konvertern sowohl die Muster als auch die Quellcode-Diagramme in Fakten und Regeln transformiert. Dabei wird Prolog zur Repräsentation der Fakten und Regeln verwendet. Prolog ist eine weit verbreitete logische Programmiersprache, die auf Hornklauseln der Logik erster Ordnung basiert und die Darstellung von vorhandenem Wissen als Fakten und Regeln ermöglicht [CM12]. Fakten beschreiben bestimmte Sachverhalte, d. h. Objekte und deren Beziehungen. Basierend auf Fakten können Abfragen formuliert werden, um entweder zu beweisen, dass ein Faktum

wahr ist, oder um neues Wissen basierend auf vorhandenen Fakten abzuleiten. Regeln sind erweiterte Fakten mit Bedingungen, die erfüllt sein müssen, damit die Regel zutrifft. Sie vereinfachen komplexe Abfragen. Die Muster werden dabei als Regeln formalisiert, welche auf Basis der Fakten geprüft werden. Da die Regeln aus OMT-Diagrammen generiert werden, umfassen die Regeln hier auch ausschließlich die Prüfung struktureller Gleichheit.

Die übrigen Arbeiten, die Logik erster Ordnung verwenden, beschränken sich fast ausschließlich auf die Entwurfsmuster von Gamma et al. [GHJV94] und die Anwendung auf UML-Diagramme [BP02; DE13; DL06; ME16; PKG+00; TN03]. Bergenti und Poggi [BP02] formalisieren die Entwurfsmuster als Prolog-Regeln. Dabei wird jedes Muster durch zwei Regeln spezifiziert: Zum einen für die Definition struktureller Bedingungen in UML-Klassendiagrammen und zum anderen für die Definition von Bedingungen in UML-Kollaborationsdiagrammen. Ziel ist, basierend auf den erkannten Mustern, der Softwareentwicklerin oder dem Softwareentwickler Verbesserungsvorschläge zum Beispiel zu Namenskonventionen zu geben. Ein ähnliches Verfahren nutzen auch Dae-Kyoo Kim und Lunjin Lu [DL06], die strukturelle Bedingungen in einem UML-Klassendiagramm als Prolog-Fakten und die Muster als Abfragen definieren.

Anstatt die Muster als Regeln zu formulieren, können sowohl die Informationen über den Quellcode als auch die Entwurfsmuster als Fakten definiert werden und generische Erkennungsregeln zum Vergleich der strukturellen Definition des Quellcodes und der Muster genutzt werden [DE13; ME16; PKG+00]. Die Fakten zum Quellcode werden dabei aus UML-Diagrammen und die Fakten zu den Mustern aus einer semantischen Repräsentation in OWL [DE13; ME16] oder auch aus UML-Diagrammen [PKG+00] generiert. Dies vereinfacht die Formalisierung der Muster, schränkt aber auch die musterspezifische Spezifikation von Gleichheit ein. Aufgrund der Transformation der Muster aus einem strukturellen Modell kann auch hier die volle Mächtigkeit von logischer Programmierung nicht ausgenutzt werden. Besonders die Nicht-Existenz von Elementen ist bei allen Ansätzen, die aus der modellbasierten strukturellen Beschreibung resultieren, nicht ausdrückbar.

Im Vergleich zu den bisherigen Ansätzen formalisieren Paakki et al.

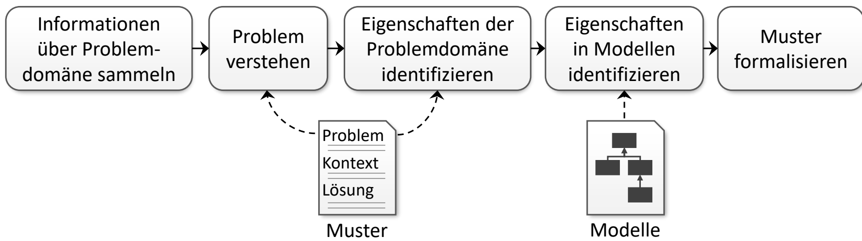

Abbildung 2.4: Prozess zur Formalisierung der Problemdomäne (adaptiert von Kim und El Khawand [KE07]).

[PKG+00] und Cortellessa, Marco und Trubiani [CMT14] *Antimuster*. Antimuster sind konzeptionell den Mustern sehr ähnlich, jedoch dokumentieren sie nicht bewährte Lösungen, sondern häufig auftretende schlechte Lösungen, die negative Auswirkungen auf, zum Beispiel, die entwickelte Software haben können [Cop96]. Paakki et al. [PKG+00] beschreiben Entwurfs-Antimuster als UML-Diagramme, welche in Prolog-Fakten transformiert werden, um diese in UML-Diagrammen zu erkennen. Cortellessa, Marco und Trubiani [CMT14] beschreiben Arbeitslast-Antimuster als logische Prädikate um in UML-Komponenten-, -Sequenz-, und -Deploymentdiagrammen diese zu identifizieren. Dafür müssen die Modelle mit spezifischen Informationen zur Arbeitslast annotiert sein. Aufgrund der Natur von Antimustern werden damit (Entwurfs-)Probleme in Modellen erkannt. Anders als die Problembeschreibung von Mustern, stellen Antimuster konkrete schlechte Lösungen dar und nicht nur den Kontext, in dem ein spezifisches Muster angewendet werden kann. Die Anwendbarkeit eines Musters impliziert grundsätzlich nicht, dass eine schlechte Lösung umgesetzt wurde.

2.3.1.2 Probleminstanzen von Mustern in Modellen erkennen

Ziel der Formalisierung der Problembeschreibung von Mustern ist es, die Anwendbarkeit eines Musters in einem bestimmten Kontext automatisiert bestimmen zu können. Im Vergleich zur Erkennung von Lösungsinstanzen gibt es in diesem Bereich nur wenige Forschungsarbeiten. Fokussiert wur-

de vor allem die Erkennung von Lösungsinstanzen, da diese häufig bereits durch Architekturdiagramme in den Musterbeschreibungen beschrieben sind, wie zum Beispiel bei den Entwurfsmustern von Gamma et al. [GHJV94], die UML-Diagramme zur Veranschaulichung nutzen. Die Problembeschreibungen sind jedoch häufig rein textuell und damit in ihrer Formalisierung komplexer [KE07; TN03]. In Tabelle 2.2 sind verschiedene Verfahren, die zur Formalisierung der Problembeschreibung genutzt werden, gelistet.

Wie im Bereich der Erkennung von Lösungsinstanzen, werden auch hier *Graph-Matching*-Verfahren eingesetzt, um die Anwendbarkeit von Entwurfsmustern [FCSK03; KE07], generellen Architekturmustern [GL19] und Cloud-Management-Mustern [BBKL13b; Bre16] zu bestimmen. Kim und El Khawand [KE07] nutzen das gleiche Verfahren wie bereits in ihren Arbeiten zur Erkennung von Lösungsinstanzen [DL06; KS07]: Die Problemdomäne wird in RBML modelliert und mit UML-Diagrammen abgeglichen. Wird die Struktur im UML-Diagramm erkannt, ist das entsprechende Muster anwendbar. Darüber hinaus führen die Autoren einen Prozess zur Formalisierung der Problemdomäne ein, der in Abbildung 2.4 dargestellt ist. Die Basis für die Formalisierung bilden die Musterbeschreibungen sowie reale Beispiele, d.h. Modelle, die im ersten Schritt gesammelt werden. Im zweiten Schritt muss das Problem generell verstanden werden. Dazu werden verschiedene Abschnitte des Musterformats berücksichtigt, nicht nur die Beschreibung des Problems, sondern auch der Kontext, in dem das Muster angewendet werden kann, sowie Situationen, die die Anwendung forcieren, die bekannten Umsetzungen des Musters und die Beziehungen zu anderen Mustern der Mustersprache. Aus den textuellen Beschreibungen werden im dritten Schritt Eigenschaften abgeleitet, die in einem Modell vorhanden sind, wenn das entsprechende Problem vorliegt. Um ein vollständiges Bild der Problemdomäne zu erhalten werden zusätzlich im vierten Schritt reale Beispiele verwendet, um die Erkenntnisse aus dem vorherigen Schritt zu verifizieren und zu ergänzen. Anschließend kann die Problemdomäne mit den identifizierten Eigenschaften zur Bestimmung der Anwendbarkeit eines Musters formalisiert werden.

France et al. [FCSK03] gehen einen Schritt weiter und führen eine muster-

basierte Methode zur Modelltransformation von UML-Klassendiagrammen basierend auf Entwurfsmustern ein. Dazu werden drei generelle Bausteine für ein Muster definiert: Problem-, Transformations- und Lösungsspezifikation. Die Problem- und Lösungsspezifikationen werden als UML-Diagramme definiert. Die konkrete Umsetzung der Transformationsspezifikation wird nicht erläutert. Auf diesen drei Bausteinen basiert auch das Konzept von Guth und Leymann [GL19] um generelle Architekturmuster in generischen graphbasierten Architekturmodellen zu erkennen. Es werden Graphfragmente zur Erkennung der Anwendbarkeit und zur Spezifikation zusätzlich benötigter Komponenten zur Realisierung des Musters spezifiziert, sowie ein Operator zur Adaption des Architekturmodells definiert. Der Operator ist als musterspezifischer Algorithmus umgesetzt. Bei beiden Verfahren ist unklar, auf welcher Basis die Modelle zur Erkennung der Anwendbarkeit definiert wurden. Indem auch nur Strukturelemente verglichen werden, kann das Fehlen von Komponenten nicht zur Problemspezifikation genutzt werden, wodurch die Expressivität stark eingeschränkt ist.

Die bisherigen Verfahren fokussierten sich vor allem auf die Entwurfszeit einer Anwendung. Aber auch zur Laufzeit können Muster angewendet und die Anwendbarkeit automatisiert bestimmt werden. Zur Anwendung von Cloud-Management-Mustern führen Breitenbücher et al. [BBKL13b] und Breitenbücher [Bre16] ein Graph-Matching-Verfahren zur Erkennung von Subgraphen in TOSCA-Modellen ein. Die Erkennung der Anwendbarkeit erfolgt damit analog zu den bereits vorgestellten Verfahren. Die Anwendung der Musterlösung wird jedoch in Form von Workflow-Fragmenten bereitgestellt, die Operationen zur Ausführung der Managementaktivität definieren, um die Anwendung in den gewünschten Zielzustand zu überführen.

Ein *DSL-basiertes* Verfahren für die Erkennung der Anwendbarkeit zur Laufzeit nutzen Ahmad und Babar [AB14a; AB14b]. Fokus der Arbeiten ist die Adaption von Architekturmodellen basierend auf definierten Adaptionsmustern. Jedes Muster wird in einer Mustervorlage durch Bedingungen, Operatoren und die Auswirkungen auf das Modell spezifiziert. Die Bedingungen umfassen sowohl Vorbedingungen, die erfüllt sein müssen bevor das Muster angewendet werden kann, als auch Nachbedingungen und In-

varianten. Im Vergleich zu den vorherigen Verfahren werden hier Muster direkt in Logs über Architekturveränderungen identifiziert [AJP12]. Das vereinfacht die Formalisierung der Muster, da bereits technisches Wissen zur Anwendung in den Logs enthalten ist.

In den meisten Fällen ist die Erkennung einer Probleminstanz bzw. der Anwendbarkeit eines Musters in einem Modell nur Teil einer Methode zur Modelladaption, und der Fokus liegt vor allem auf der Anwendung von Mustern in einem Modell [AB14a; AB14b; BBKL13b; Bre16; FCSK03; GL19].

2.3.2 Anwendung von Musterlösungen

Die Anwendung von Musterlösungen ist abhängig von (i) der Domäne, (ii) der Mustersprache und (iii) der Sicht auf das System. Die Domäne bestimmt, welche Ausprägungen von Musterlösungen es gibt, und die Musterlösungen sind in der Mustersprache als generische Konzepte beschrieben. Abhängig davon, auf welcher Abstraktionsschicht des Systems die Musterlösungen angewendet werden sollen, müssen diese generischen Konzepte konkretisiert werden. Betrachtet man die bereits mehrfach erwähnten Entwurfsmuster von Gamma et al. [GHJV94], so werden die Musterlösungen in Form von UML-Diagrammen in der Mustersprache dokumentiert, wodurch sie unabhängig von konkreten Technologien und Programmiersprachen sind. Soll ein Entwurfsmuster auf ein UML-Diagramm, welches in der Entwurfsphase zur Spezifikation eines Systems verwendet wird, angewendet werden, können die dokumentierten Musterlösungen in Form von UML-Diagrammen als Basis für eine systemspezifische Verfeinerung verwendet werden. Soll ein Entwurfsmuster hingegen zur Erstellung oder Adaption von Quellcode angewendet werden, werden konkrete Codefragmente benötigt, welche nicht im Muster selbst dokumentiert sind. Solche konkreten Lösungen müssen dann von den Entwicklerinnen und Entwicklern erst von den generischen Konzepten des Musters abgeleitet und umgesetzt werden [FBB+14b].

Um die Dokumentation solcher konkreten Lösungen eines Musters in einer wiederverwendbaren Weise zu ermöglichen, haben Falkenthal et al. [FBB+14a; FBB+14b] ein Konzept eingeführt, um sie als *Lösungsimplemen-*

tierungen (*engl. „Solution Implementations“*) zu beschreiben, die mit einem Muster verknüpft sind. Diese Lösungsimplementierungen sind wiederverwendbare Artefakte, z. B. ausführbare Software-Artefakte, Codeschnipsel oder Konfigurationsdateien, wobei die Anwendung des Konzepts nicht auf die IT-Domäne beschränkt ist. Falkenthal et al. [FBB+14a] wenden das Konzept auch auf die Domäne der Kostümmuster [Bar18] an. Zusätzlich können Auswahlkriterien definiert werden, wann eine bestimmte Lösungsimplementierung verwendet werden soll, und Aggregatoren spezifizieren, wie Lösungsimplementierungen zu einer Gesamtlösung integriert werden können.

Bei der Verknüpfung von agnostischen Mustern mit konkreten Lösungen werden auch die proprietären Mustersprachen berücksichtigt [DCE17; FBB+15]. Falkenthal et al. [FBB+15] führen dazu das Konzept der *Verfeinerung* (*engl. „Refinement“*) für Muster unterschiedlicher Sprachen ein, um agnostische Muster mit proprietären Mustern zu verknüpfen. So kann beispielsweise das Elastic-Load-Balancer-Muster (ELB) von Fehling et al. [FLR+14] durch das Scale-Out-Muster (SO) von AWS [Ama22a] verfeinert werden, welches die Umsetzung des ELB-Musters mit Diensten von AWS dokumentiert. Die Konzepte, die im ELB-Muster zur Realisierung des Musters beschrieben sind, können auf die AWS-Dienste des SO-Musters abgebildet werden. Die Verknüpfung von agnostischen und proprietären Mustern sowie der dabei verwendeten Dienste nutzen auch Di Martino, Cretella und Esposito [DCE17] zur Selektion geeigneter Cloud-Dienste basierend auf Anwendungsanforderungen und zur vereinfachten Portabilität zwischen Cloud-Anbietern.

Im Folgenden werden verschiedene Konzepte aus der Musteranwendung diskutiert. Dabei werden ausschließlich Konzepte betrachtet, die sich mit der Anwendung von Mustern auf Modelle von Anwendungssystemen beschränken. Zuerst werden Konzepte zur Modellierung mit Mustern eingeführt und anschließend Verfahren zur automatisierten Adaption von Modellen basierend auf dem Wissen aus Mustersprachen erläutert.

2.3.2.1 Modellierungskonzepte mit Mustern

In der Cloud-Domäne gibt es verschiedene Arbeiten, die die Cloud-Muster von Fehling et al. [FLR+14] bei der Modellierung von Deployment-Modellen einsetzen [FLR+11; FLRS12; HBF+18; HBF+20; NBF+12]. Die Vision der Integration von Mustern in Tools zum Deployment von Cloud-Anwendungen wurde von Fehling et al. [FLR+11; FLRS12] vorgestellt. Die Cloud-Muster werden dabei mit Modellfragmenten annotiert, die konkrete Realisierungen, zum Beispiel das Deployment einer Ubuntu-VM auf Amazon EC2, dem IaaS-Dienst von AWS, repräsentieren. Die Komponenten und Beziehungen zwischen den Komponenten eines Architekturmodells, welches ausschließlich anwendungsspezifische Komponenten umfasst, werden anschließend mit Mustern annotiert. Basierend auf den Annotationen im Modell können dann durch die Entwicklerin oder den Entwickler passende Modellfragmente für die Realisierung und das Deployment der Komponenten und Kommunikationsbeziehungen identifiziert werden. Nowak et al. [NBF+12] kombinieren die Cloud-Muster mit den *Green-Geschäftsprozessmustern* [NLS+11] und ermöglichen die Annotation von Komponenten in TOSCA-Modellen, um passende Modellfragmente, die bestimmte Cloud-Dienste realisieren, zu selektieren. In beiden Verfahren sind Modellfragmente die Lösungsimplementierungen der Cloud-Muster, welche als Annotationen in Architektur- bzw. Deployment-Modellen verwendet werden. Es werden jedoch keine Mechanismen vorgestellt, die eine automatisierte Selektion von Lösungsimplementierungen ermöglichen.

Harzenetter et al. [HBF+18; HBF+20] führen einen MDA-basierten Ansatz ein, der die Modellierung mit agnostischen Mustern in Deployment-Modellen und die automatisierte Verfeinerung der Muster durch Modellfragmente ermöglicht. Strukturelle Muster werden als eigenständige Modellierungselemente betrachtet. So kann zum Beispiel ausgedrückt werden, dass eine *Relationale Datenbank* auf einer *Private Cloud* bereitgestellt werden soll, beides als Musterelemente modelliert. Ein Deployment-Modell kann so aus Komponenten und Mustern sowie deren Beziehungen bestehen. Zusätzlich können sowohl Komponenten als auch Muster mit Verhaltensmustern

annotiert werden, um zum Beispiel auszudrücken, dass eine bestimmte Komponente als *zustandslose Komponente* implementiert werden muss oder eine Kommunikationsbeziehung durch das SECURE-CHANEL-Muster verschlüsselt werden muss. Zur Demonstration des Modellierungskonzepts werden die Cloud-Muster [FLR+14], Muster zur Integration von Anwendungssystem [HW04] und Muster für Sicherheitsarchitekturen [SFH+06] verwendet. *Verfeinerungmodelle* spezifizieren welche Teile eines Modells durch welche Verfeinerungsstruktur ersetzt werden können und wie sich das neue Modellfragment in das bestehende Modell einfügt. Die Anwendbarkeit einer Lösungsimplementierung wird durch einen *Detektor* erkannt, der Mechanismus ist ähnlich zu den Graph-Matching-Verfahren zur Anwendbarkeit eines Musters [BBKL13b; Bre16; GL19], wobei hier nicht die Anwendbarkeit eines Musters, sondern einer konkreten Lösungsimplementierung eines bereits selektierten Musters geprüft wird.

Beschränkt auf die EIP-Muster von Hohpe und Woolf [HW04] ermöglichen Fahland und Gierds [FG13] die Transformation von Architekturmodellen, modelliert mit den EIP-Mustern, in CPN-Modelle, welche zur Analyse und zur weiteren Verfeinerung in ausführbare Systeme genutzt werden können.

Auch Arnold et al. [AEK+07] haben ein Konzept zur Modellierung mit Mustern in Deployment-Modellen veröffentlicht. Anstatt Muster als eigenständige Modellelemente zur Modellierung einzusetzen, werden hier Muster als Modellfragmente bestehend aus abstrakten Komponenten bereitgestellt, welche mit anderen Fragmenten oder bestehenden Modellen integriert werden. Die abstrakten Komponenten werden über definierte *Realisiserungslinks* durch konkrete Komponenten ersetzt. Dazu wird auch hier ein Graph-Matching-Verfahren angewendet. Ein ähnliches Verfahren wird von Park et al. [PKYY20] zur Selektion und Integration von Cloud-Diensten genutzt. Basierend auf den proprietären Mustern verschiedener Cloud-Anbieter können hybride Cloud-Anwendungen modelliert werden. Ein Muster wird dabei durch Knoten und Kanten abgebildet, die die verschiedenen Cloud-Dienste und deren Beziehungen repräsentieren, welche benötigt werden, um das Muster zu realisieren. Anstatt konkreter Cloud-Dienste der verschiedenen Anbieter werden generalisierte Dienstkategorien definiert, die auf die kon-

kreten Dienste der Anbieter abgebildet werden können. Durch die Selektion mehrerer Muster kann so das ursprüngliche Modell durch weitere Dienste ergänzt werden. Anschließend kann ein bestimmter Anbieter ausgewählt werden und die generischen Dienste können durch die spezifischen Dienste des Anbieters verfeinert werden. Die Mechanismen zur Kombination verschiedener Muster wie auch zur Verfeinerung der generischen Dienste wird dabei nicht erläutert.

Lehrig, Hilbrich und Becker [LHB18] dokumentieren Architekturmuster als Templates zur Analyse von Architekturmodellen bezüglich ihrer QoS. Dabei werden die Architekturmodelle mit dem Wissen aus den dokumentierten Mustern, hier *Architektur-Templates (ATs)* genannt, annotiert und basierend auf der Definition des ATs analysiert, ob die Bedingungen, die im AT spezifiziert sind, erfüllt sind. Zusätzlich können die ATs genutzt werden, um das Architekturmodell zu vervollständigen, zum Beispiel um einen Loadbalancer hinzuzufügen. Dazu werden zusätzliche Transformationsregeln in einem AT definiert. Die Muster werden hier sowohl zur Annotation der Architekturmodelle als auch zur Modelladaption verwendet. Im Folgenden werden weitere Verfahren zur Modelladaption mit Mustern vorgestellt. Grundsätzlich ermöglicht die Modellierung mit Mustern sowohl die Modellanalyse [LHB18], als auch die schrittweise Verfeinerung von abstrakten Modellen zu konkreten Realisierungen [AEK+07; FLR+11; FLRS12; HBF+18; HBF+20; NBF+12].

2.3.2.2 Modelladaption mit Mustern

Die generelle Modelladaption wird durch eine Vielzahl an Konzepten unterstützt. Unter anderem ermöglichen die bereits in Abschnitt 2.1.3.2 erörterten und die generell diskutierten MDA-basierten Verfahren durch die schrittweise Verfeinerung der Architektur die Adaption von Modellen. Im Folgenden werden ausschließlich Ansätze diskutiert, die Wissen aus Mustern zur Modelladaption anwenden. Bereits in Abschnitt 2.3.1.2 wurden verschiedene Konzepte vorgestellt, die Muster zur Adaption von Modellen nutzen und die die Anwendbarkeit eines Musters automatisiert bestimmen [AB14a; AB14b; BBKL13b; Bre16; FCSK03; GL19]. Diese Konzepte basieren auf der Annahme,

dass ein Muster nur eine dokumentierte Lösung hat, welche auf Modellebene angewendet werden kann. Damit sind sie auf eine spezifische Abstraktion im Modell beschränkt. Für die Modelladaption werden dabei verschiedene Verfahren angewendet. Ahmad und Babar [AB14a; AB14b] und France et al. [FCSK03] definieren dafür ein Template mit Operationen, die zur Adaption ausgeführt werden müssen. Ahmad und Babar [AB14a; AB14b] führen dazu zusätzlich drei Operationen ein für das Hinzufügen *(ADD)*, das Entfernen *(REM)* und das Modifizieren *(MOD)* von Elementen im Architekturmodell. Guth und Leymann [GL19] definieren einen Operator für jedes Muster, der einen Algorithmus implementiert, der die Modelladaption entsprechend der vorgegebenen Struktur des Musters durchführt. Ein dedizierter Algorithmus für jedes Muster wird auch für die Anwendung von Cloud-Management-Mustern in Deployment-Modellen verwendet [BBKL13b; Bre16].

Auch Capelli und Scandurra [CS16] definieren nur die generellen Lösungen von serviceorientierten Architekturmustern (SOA-Muster) [Arc], um diese zur Adaption von Architekturmodellen zu nutzen. Zur Modellierung der Architektur wird der OASIS Standard *Service Component Architecture (SCA)* [OAS07a] verwendet. Damit können die Komponenten einer Anwendung mit den nach außen angebotenen *Services* sowie den benötigten Services, genannt *Referenzen*, modelliert werden. Zusätzlich können die Komponenten über *Eigenschaften* konfiguriert und die *Anbindung* an andere Services definiert werden. Die Muster werden als Templates bereitgestellt, die sowohl die strukturellen Eigenschaften als auch das Verhalten des Musters beschreiben. Dazu wird die *SCA-ASM (Abstract State Machine)* Spezifikation zur Modellierung verwendet, welche die Definition von zusätzlichen Regeln ermöglicht, um das Verhalten von Services zu beschreiben. Die Muster können dann zum einen zur Modellanalyse verwendet werden, um die korrekte Instanziierung bestimmter Muster zu überprüfen, und zum anderen zur Adaption des Modells. Die Anwendung der Muster ist dabei jedoch nur teilautomatisiert. Die Entwicklerin oder der Entwickler muss die Komponenten und Services selektieren, die eine bestimmte Rolle in dem anzuwendenden Muster spielen.

Spezielle Cloud-Migrationsmuster [JPCL14] wenden Jamshidi, Pahl und

Mendonça [JPM17] zur Migration von Anwendungen in die Cloud an. Dabei wird auch spezifiziert wie die Systemarchitektur vor und nach der Migration aussieht. Die Selektion der Muster ist jedoch nicht automatisiert und auch die Transformation zur Zielarchitektur ist nicht definiert.

2.3.3 Zusammenfassung und Forschungsfragen

Muster werden in verschiedenen Domänen zur Dokumentation von Best Practices genutzt. Vor allem in unterschiedlichen Bereichen der Informatik und Informationstechnologie findet dieses Konzept Anwendung. Die Best Practices werden textuell mit grafischen Ergänzungen erfasst und sind vor allem für die Verarbeitung durch den Menschen gedacht. Dies macht es schwierig, Muster in, zum Beispiel, Softwareentwicklungstools zu integrieren. Verschiedene Verfahren wurden entwickelt, um Musterbeschreibungen zu formalisieren, um sie zur (i) automatisierten Erkennung und (ii) automatisierten Anwendung von Mustern nutzen zu können. Die Repräsentation der Muster ist dabei immer abhängig von der Domäne sowie vom Anwendungsbereich des Musters.

Im Fokus bisheriger Arbeiten zur Erkennung von Mustern lag vor allem die Formalisierung von Entwurfsmustern zur Erkennung in UML-Klassendiagrammen. Verfahren die dabei auf Graph-Matching-Ansätzen aufbauen haben nur eine eingeschränkte Expressivität, da unter anderem die Nicht-Existenz dadurch nicht abgebildet werden kann. Darüber hinaus wurde vor allem die Überprüfung von Lösungsinstanzen bei der Erkennung von Mustern adressiert. Im Kontext von partnerübergreifenden Anwendungen, welche entsprechend den beteiligten Partnern adaptiert werden müssen, steht vor allem die Erkennung von Problemen im Fokus. Bisher werden auch bei der Erkennung der Anwendbarkeit von Mustern vor allem Graph-Matching-Verfahren angewendet, die mit den bereits diskutierten Einschränkungen einhergehen. Die Erkennung der Nicht-Existenz ist jedoch vor allem bei der Problemerkennung essentiell. Diese Möglichkeit bieten nur die Verfahren, die Logik erster Ordnung, häufig in Form von in Prolog implementierten Fakten und Regeln, zur Formalisierung nutzen. Dies wird jedoch bisher nur zur

Erkennung von Lösungsinstanzen eingesetzt und häufig werden die Prolog-Regeln aus Graphrepräsentationen der Muster transformiert, wodurch die gleiche Einschränkung wie bei den Graph-Matching-Verfahren gegeben ist.

Die Anwendung eines Musters und was eine Musterlösung im Speziellen ist, hängt von der Sicht auf das System ab. So können konkrete Lösungen, Lösungsimplementierungen genannt, Codeschnipsel, Konfigurationsdateien, aber auch Modellfragmente sein. In MDA-basierten Verfahren können Muster als Modellelemente oder zur Annotation in Modellen verwendet werden, um diese dann schrittweise durch konkretere Lösungsimplementierungen zu ersetzen. Bei der Adaption von Modellen durch das Wissen der Musterbeschreibungen wird derzeit die Realisierung von Mustern im konkreten technologischen Kontext nicht berücksichtigt. Es wird eine Musterlösung bereitgestellt, die Variabilität, zum Beispiel in Bezug auf verwendete Technologien, wird dabei jedoch nicht berücksichtigt.

Zur Adressierung der Forschungslücken ist ein zentrales Konzept dieser Arbeit die Erkennung von Problemen in Deployment-Modellen durch das dokumentierte Wissen in Mustern, wobei Logik erster Ordnung genutzt wird, um eine ausdrucksstarke regelbasierte Formalisierung der Muster zu ermöglichen. Des Weiteren wird zum Beheben erkannter Probleme ein Konzept zur Anwendung von Musterlösungen in Deployment-Modellen eingeführt, wobei der technologische Kontext und die Variabilität der Realisierung des Musters abhängig des Kontexts berücksichtigt wird (Forschungsbeitrag 3).

2.4 Zusammenfassung und Forschungsfragen dieser Arbeit

Die vorherigen Abschnitte haben gezeigt, (i) welche Modellierungssprachen und Konzepte zur Definition und Adaption von Deployment-Modellen genutzt werden, (ii) wie deklarative und imperative Deployment-Technologien zur Bereitstellung von Anwendungen kombiniert und das Deployment koordiniert werden können und (iii) welche Verfahren zur automatisierten Erkennung und Anwendung von Mustern in Architektur- und Entwurfsmodellen bereits angewendet werden. In allen drei Bereichen wurden offene

Forschungsfragen identifiziert, die essentiell für das Verteilen, Adaptieren und Deployment partnerübergreifender Anwendungen sind.

Derzeit fehlt ein dezentrales Deployment-Konzept, das sowohl bei der Modellierung als auch bei der Bereitstellung organisatorische Aspekte berücksichtigt. Die Trennung globaler partnerübergreifender und lokaler partnerspezifischer Informationen und Anforderungen im Deployment-Modell sowie die dezentrale Koordination des Deployments stehen dabei im Fokus. Darüber hinaus wird bei der Verteilung von Anwendungskomponenten durch Modelladaptionsverfahren Wissen aus vorherigen Deployments, zum Beispiel aus der Testumgebung, nicht berücksichtigt, die jedoch bereits erprobte Technologien für spezifische Anwendungen beinhalten können.

Durch die Einbindung der organisatorischen Sicht in die Deployment-Modelle entsteht ein erhöhter Adaptionsbedarf sowohl durch die Verteilung auf die verschiedenen Partner als auch durch die Verteilung auf verschiedene Umgebungen. Das frühzeitige Erkennen und Beheben von potenziellen Problemen ist dabei entscheidend. Die Vielzahl an dokumentierten Mustern bietet dafür die Grundlage. Jedoch bieten die bisherigen Verfahren zum automatisierten Erkennen der Anwendbarkeit von Mustern nicht die nötige Ausdrucksstärke, um verschiedene Problemklassen abbilden zu können. Auch Konzepte zur Anwendung der Muster sind bisher auf abstrakte Architektur- und Entwurfsmodelle beschränkt. Technologiespezifische Rahmenbedingungen, die eine Selektion bestimmter Lösungsimplementierungen nötig machen, werden dabei außer Acht gelassen. Zur Adressierung der offenen Fragestellungen wird im nächsten Kapitel die DivA-Methode vorgestellt.

Kapitel 3

DivA-Methode für das Deployment partnerübergreifender Anwendungen

Dieses Kapitel stellt die DivA-Methode („*Divide and Adapt*“-Methode) zum Verteilen, Adaptieren und Deployment partnerübergreifender Anwendungen vor. Die Methode umfasst alle Schritte zur Bereitstellung, von der Modellierung bis zur Ausführung. Das Management zur Laufzeit wird dabei nicht betrachtet. Dazu werden zuerst die Anforderungen an die Methode in Abschnitt 3.1 spezifiziert und anschließend die einzelnen Schritte der Methode in Abschnitt 3.2 eingeführt. Die Methode unterstützt nicht nur die Verteilung auf Partner derselben organisatorischen Ebene, sondern auch die rekursive Verteilung auf weitere kleinere Organisationseinheiten eines Partners. So kann die Verantwortlichkeit nicht nur zwischen Unternehmen, sondern auch

weiter innerhalb eines Unternehmens auf unterschiedliche Abteilungen verteilt werden. Diese Erweiterung der Methode wird in Abschnitt 3.3 erläutert. Abschließend wird in Abschnitt 3.4 die Umsetzung der Anforderungen aus Abschnitt 3.1 in der DIVA-Methode diskutiert.

3.1 Anforderungen an das partnerübergreifende Deployment

Auf Basis der in Kapitel 2 diskutierten verwandten Arbeiten und identifizierten offenen Fragestellungen werden in diesem Abschnitt Anforderungen an eine Methode zum Deployment partnerübergreifender Anwendungen abgeleitet. Diese Anforderungen dienen zur Verifizierung der DIVA-Methode, die im darauffolgenden Abschnitt eingeführt wird.

Anforderung 1 (Modellierung von Organisationseinheiten). Die verschiedenen beteiligten Partner beim Deployment einer Anwendung müssen im Deployment-Modell abgebildet werden können. Dazu gehört unter anderem die Zuweisung von Anwendungskomponenten sowie benötigte Informationen zum automatisierten Deployment (abgeleitet aus Abschnitt 2.1.2 und Abschnitt 2.1.3).

Anforderung 2 (Datenkapselung) *(engl. „Information Hiding“)*. Bei einer Kooperation verschiedener Partner sollen nicht alle internen Informationen des Deployments offen gelegt werden müssen. Ausschließlich Informationen, die partnerübergreifend für das automatisierte Deployment der Anwendungskomponenten relevant sind, müssen mit anderen Partner geteilt werden (abgeleitet aus Abschnitt 2.1.3 und Abschnitt 2.2.3).

Anforderung 3 (Technologieunabhängigkeit). Eine Vielzahl an Deployment-Technologien wird von verschiedenen Unternehmen in unterschiedlichen Bereichen eingesetzt. Aufgrund der Heterogenität der verfügbaren Technologien muss die Methode technologieunabhängig sein und die Nutzung unterschiedlicher Technologien durch verschiedene Partner ermöglichen (abgeleitet aus Abschnitt 2.2.2 und Abschnitt 2.1.2).

Anforderung 4 (Wiederverwendbarkeit von Deployment-Informationen). Bereits vor dem Betrieb der Anwendung werden Deployment-Modelle zum Beispiel zur Bereitstellung in der Testumgebung genutzt. Die bereits erprobten Technologien sollen in den Deployment-Modellen für die Produktivumgebung automatisiert übernommen werden können und nur bei Bedarf ausgetauscht werden, auch bei einer Neuverteilung der Anwendungskomponenten (abgeleitet aus Abschnitt 2.1.3).

Anforderung 5 (Ermittlung des Adaptionsbedarfs bei Verteilung). Probleme und potenzielle Adaptionsbedarfe in verteilten Deployment-Szenarien, die durch die Zuordnung von Anwendungskomponenten zu Organisationseinheiten sowie durch die ressourcenspezifische Verteilung der Anwendungskomponenten entstehen, müssen frühzeitig zur Entwurfszeit identifiziert werden können (abgeleitet aus Abschnitt 2.3.1 und Abschnitt 2.1.3).

Anforderung 6 (Kontextspezifische Modelladaption). Die Adaption eines Deployment-Modells zur Behebung erkannter Probleme muss die kontextspezifischen Informationen des Deployment-Modells und der Anwendung berücksichtigen. Vor allem technologiespezifische Vorgaben müssen dabei berücksichtigt werden, um die Funktionsfähigkeit des Deployments zu gewährleisten (abgeleitet aus Abschnitt 2.3.2 und Abschnitt 2.1.3).

Anforderung 7 (Automatisierte dezentrale Deployment-Koordination). Bei der Beteiligung mehrerer unabhängiger Partner muss eine dezentrale Koordination des Deployments durch die Partner möglich sein. Die Kontrolle über das Deployment liegt dabei beim jeweiligen Partner, der den betreffenden Teil der Anwendung verwaltet (abgeleitet aus Abschnitt 2.2.3).

Anforderung 8 (Robuste Ausführung des Deployments). Das Deployment muss robust und verlässlich durchgeführt werden können. Fehler beim Deployment eines Partners dürfen nicht zu einem inkonsistenten Zustand der gesamten verteilten Anwendung führen (abgeleitet aus Abschnitt 2.2.1).

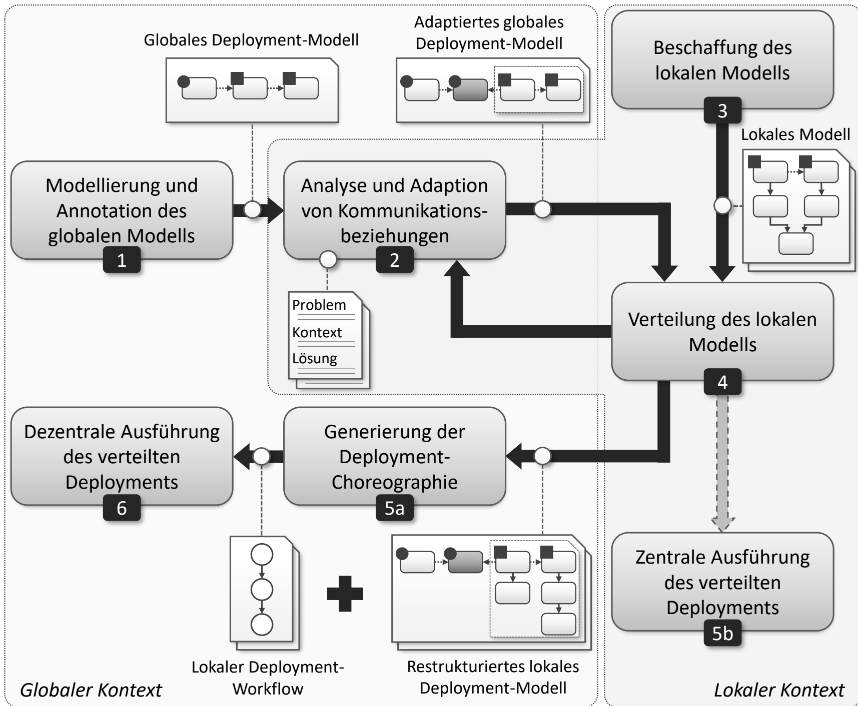

Abbildung 3.1: Überblick über die DIVA-Methode: Methodenschritte und resultierende Artefakte.

3.2 Die DIVA-Methode

Die DIVA-Methode führt einen Prozess für die Modellierung, Analyse, Adaption und das Deployment partnerübergreifender Anwendungen ein. Die Methode umfasst sechs Schritte, die in den *globalen* bzw. *lokalen* Kontext eingeordnet sind. Der Kontext der Schritte spezifiziert dabei, ob diese Schritte in Abhängigkeit zu anderen Partnern (global) oder unabhängig (lokal) durchgeführt werden. In jedem Schritt werden verschiedene Konzepte und Methoden angewendet, die im Rahmen dieser Arbeit entwickelt wurden. In Abbildung 3.1 ist die gesamte Methode mit ihren Schritten, deren benötigten

Informationen und den jeweils resultierenden Artefakten abgebildet.

Im initialen Schritt der Methode (Schritt 1) werden das *globale Deployment-Modell* modelliert und die Komponenten mit den jeweils verantwortlichen Partnern annotiert. Ausschließlich partnerübergreifende Deployment-Informationen sind im globalen Modell enthalten. Anschließend wird das Modell basierend auf dem Wissen aus Mustern auf potenzielle Probleme, die eine ordnungsgemäße Bereitstellung und den Betrieb der Anwendung verhindern, analysiert und bei Bedarf adaptiert, zum Beispiel durch Hinzufügen zusätzlicher Komponenten (Schritt 2). Beide Schritte finden im globalen Kontext mit dem Wissen aller Partner statt. Das resultierende *adaptierte* globale Deployment-Modell dient jedem Partner als Input für das lokale Deployment. Zusätzlich zu dem globalen Modell wird, wenn vorhanden, das *lokale Deployment-Modell* aus früheren Phasen des Entwicklungsprozesses beschafft (Schritt 3), um das vorhandene Wissen bei der Verteilung der Komponenten zu berücksichtigen (Schritt 4). Durch die ressourcenspezifische Verteilung der Komponenten kann es zu weiteren Problem kommen, die zu einer Adaption des lokalen Modells führen (Schritt 2). Die Analyse und Adaption finden dabei im lokalen Kontext des jeweiligen Partners statt und haben keine Auswirkungen auf die Deployment-Modelle der anderen Partner. Das resultierende *restrukturierte* lokale Deployment-Modell dient als Basis für die Generierung der *lokalen Deployment-Workflows* (Schritt 5a). Die Generierung der Workflows basiert auf definierten Schnittstellen zwischen den Partnern. Die lokalen Workflows formen eine Choreographie, welche die dezentrale Ausführung des Deployments ermöglicht (Schritt 6).

Die Methode kann auch ausschließlich im lokalen Kontext eines einzelnen Partners eingesetzt werden. Dabei würden die Schritte im globalen Kontext entfallen und ein zentrales Deployment, wie es bereits von verschiedensten Technologien unterstützt wird, ausgeführt werden (Schritt 5b). In der *Single-Partner*-Variante startet die Methode mit der Beschaffung des lokalen Deployment-Modells (Schritt 3) und bietet damit eine Ergänzung zu existierenden Methoden im Zusammenspiel von Softwareentwicklung und Betrieb (*engl. „DevOps“*). Im Folgenden werden alle Schritte und resultierenden Artefakte im Detail beschrieben.

3.2.1 Schritt 1: Modellierung und Annotation des globalen Modells

Im ersten Schritt werden unter Beteiligung aller involvierter Partner das *globale Deployment-Modell* modelliert und die Komponenten mit den Partnern, die jeweils für die Bereitstellung verantwortlich sind, annotiert.

Definition 3.1 (Globales Deployment-Modell – informell) Ein globales Deployment-Modell (GDM) mit mehreren Partnern ist ein annotiertes deklaratives Deployment-Modell, das alle global sichtbaren Informationen enthält, einschließlich (i) der anwendungsspezifischen Komponenten, (ii) der notwendigen Informationen über die Hosting-Umgebung und (iii) der Partner, denen Teile des GDMs zugewiesen sind.

Die anwendungsspezifischen Komponenten des GDMs repräsentieren den zu realisierenden Anwendungsfall. Dazu gehören typischerweise Komponenten, die die Geschäftslogik der Anwendung umfassen, sowie Komponenten zur Datenspeicherung und Kommunikation. Zusätzlich müssen für jede dieser Komponenten und deren Relationen die Lebenszyklusoperationen sowie deren Anforderungen an die Hosting-Umgebung definiert werden. Die Eingabeparameter der Operationen müssen entweder (a) von der Komponente selbst, (b) von einer anderen anwendungsspezifischen Komponente oder (c) von der Hosting-Umgebung während des Deployments bereitgestellt werden. Um zum Beispiel eine Verbindung zu einer anderen Komponente aufzubauen werden deren Hostname und möglicherweise Zugangsdaten benötigt. Basierend auf den Anforderungen und Eingabeparametern der Operationen können *Platzhalter* für die Hosting-Umgebung der jeweiligen Komponente mit den nötigen Fähigkeiten und Parametern generiert werden. Die Platzhalter werden bei der Verteilung der Komponenten durch konkrete Komponenten ersetzt (Schritt 3). Die Komponenten und Relationen des GDMs können mit zusätzlichen Informationen annotiert werden, die das Verhalten beschreiben. Dazu zählen zum Beispiel Informationen über die auszutauschenden Daten zwischen den Komponenten oder die Zugriffsrestriktionen bestimmter Komponenten. Diese Eigenschaften bestimmten das

globale Verhalten der Anwendung nach dem Deployment und sind deshalb Teil des GDMs.

Die involvierten Partner werden als zusätzliche Entitäten im Deployment-Modell spezifiziert. Dazu gehört neben der Identifikation der Partner auch Informationen zum Datenaustausch während des Deployments, um eine dezentrale Koordination zwischen den Partnern zu ermöglichen (Anforderung 1). Diese Kommunikationsschnittstelle muss für jeden Partner spezifiziert werden und global zur Verfügung stehen. Das globale Deployment-Modell enthält damit die minimale Menge an Informationen, die global sichtbar sein müssen, d.h. die partnerübergreifend zur Verfügung stehen müssen (Anforderung 2).

3.2.2 Schritt 2: Analyse und Adaption von Kommunikationsbeziehungen

Durch die Zuteilung der Komponenten zu Partnern im vorherigen Schritt wurde die organisatorische Verteilung der Komponenten vorgenommen. Durch diese Verteilung können Probleme entstehen, die bei einer zentralisierten Bereitstellung in einer einzigen Umgebung nicht auftreten. Dies betrifft vor allem die partnerübergreifenden Kommunikationsbeziehungen, die, zum Beispiel durch Zugriffsrestriktionen, nicht wie vorher geplant umgesetzt werden können.

Wie bereits in Abschnitt 2.3.1 diskutiert, dokumentieren Muster bewährte Lösungen für häufig auftretende Probleme. Die manuelle Anwendung der Muster ist jedoch aufwendig, beginnend mit der Identifikation eines geeigneten Musters in einer konkreten Situation. Ziel des zweiten Schritts ist deshalb die automatisierte Erkennung von potenziellen Problemen durch Modellanalysen sowie die automatisierte Identifikation und Anwendung von Adapationsmöglichkeiten zur Behebung der Probleme. Dafür wird das Wissen aus Mustern für diesen Schritt formalisiert und (i) zur Analyse sowie (ii) zur Adaption von Deployment-Modellen angewendet.

Für die Analyse werden die Beschreibungen des Problems und des Problemkontexts der Muster entsprechend dem Domänenmodell der Deployment-Modelle als logische Formeln definiert. Die Basis bilden die struktu-

rellen Eigenschaften des Modells sowie die Konfigurationsparameter und zusätzlichen Annotationen zur Beschreibung des Verhaltens. Die formalisierten Muster werden dann zur regelbasierten Analyse eines GDMs genutzt, um potenzielle Probleme zu identifizieren (Anforderung 5).

Während für die Identifikation der Probleme die abstrakte technologieunabhängige Beschreibung agnostischer Muster ausreicht, sind die generischen Lösungsbeschreibungen für die Adaption eines Deployment-Modells ungeeignet, da die Anwendbarkeit einer Lösung von den bereits verwendeten Technologien in einem Deployment-Modell abhängig ist. So können, zum Beispiel, für die verschlüsselte Übertragung der Daten sowohl Proxies für die TLS-Verschlüsselung für das HTTP-Protokoll zwischen den kommunizierenden Komponenten als auch direkt auf der Vermittlungsschicht das IPsec-Protokoll verwendet werden. Die Anwendbarkeit der Lösungsimplementierungen hängt jedoch vom *Deployment-Kontext* ab. TLS für HTTP kann nur verwendet werden, wenn HTTP für die Kommunikation zwischen den Komponenten genutzt wird. Und auch IPsec kann nur verwendet werden, wenn das verwendete Protokoll auf der Vermittlungsschicht konfiguriert werden kann. Beide Lösungsimplementierungen bieten dieselbe generische Lösung, die verschlüsselte Datenübertragung, die Umsetzung ist jedoch technologiespezifisch und abhängig vom Deployment-Kontext (Anforderung 6). Der Deployment-Kontext wird ebenfalls, wie die Muster, als logische Formel für jede Lösungsimplementierung formalisiert. Zusätzlich wird ein *Adaptionsalgorithmus* bereitgestellt, der die Adaption des Deployment-Modells vornimmt. Im Sinne der Definition von Falkenthal et al. [FBB+14b] entspricht der Adaptionsalgorithmus der Lösungsimplementierung für das Problem. Den nach der Adaption zusätzlichen Komponenten im resultierenden *adaptierten globalen Deployment-Modell* müssen dann noch Partnern zugeteilt werden. Das Modell wird anschließend an alle beteiligten Partner verteilt.

Schritt 2 ist jedoch nicht auf den globalen Kontext beschränkt, sondern kann vor allem auch nach der ressourcenspezifischen Verteilung der Komponenten (Schritt 4) angewendet werden. Speziell ist dabei mehr Wissen über die konkret verwendeten Technologien vorhanden, wodurch die automatisierte Adaption ermöglicht wird.

3.2.3 Schritt 3: Beschaffung des lokalen Deployment-Modells

Unter Ausnutzung des gleichen Metamodells in der Entwicklungsphase und der operationalen Phase einer Anwendung wird die Bereitstellung der Anwendungen sowohl in Test- als auch in Produktivumgebungen vereinfacht. Deployment-Modelle der Entwicklungsphase können so wiederverwendet und für die Anforderungen in der Produktivumgebung angepasst werden [FCSS15]. Im dritten Schritt wird dafür, falls vorhanden, ein bereits existierendes Deployment-Modell aus der Entwicklungsphase beschafft. Dieses Modell wird mit dem GDM aus dem vorherigen Schritt zusammengeführt und spezifiziert damit für die Komponenten des jeweiligen Partners konkrete Middleware- und Infrastrukturkomponenten für das Deployment. Da dieses Modell ausschließlich für die Spezifikation partnerspezifischer Deployment-Informationen genutzt wird, spezifiziert es das *lokales Deployment-Modell* eines jeden Partners.

Definition 3.2 (Lokales Deployment-Modell – informell) Ein lokales Deployment-Modell (LDM) ist ein teilweise substituiertes GDM mit detaillierten Middleware- und Infrastrukturkomponenten für die vom jeweiligen Partner verwalteten anwendungsspezifischen Komponenten.

Durch die Verwendung der Deployment-Modelle aus der Entwicklungsphase kann bereits erlangtes Wissen in Bezug auf verwendete Technologien weiterverwendet werden und als Basis für die ressourcenspezifische Verteilung der Komponenten in der Produktivumgebung dienen (Anforderung 4).

3.2.4 Schritt 4: Verteilung der lokalen Komponenten

Das LDM aus dem vorherigen Schritt dient als Grundlage für die ressourcenspezifische Verteilung der Komponenten. Die Verteilung der Komponenten kann unterschiedliche Gründe haben: Die Testumgebung unterscheidet sich von der Produktivumgebung, es wurden Teile der IT ausgelagert oder neue Cloud-Anbieter oder -Dienste sind verfügbar. Infolgedessen müssen Anwendungen, die zuvor, zum Beispiel, im eigenen Datencenter bereitgestellt

wurden, möglicherweise auf Public-Cloud-Dienste umverteilt werden. Das manuelle Anpassen von Deployment-Modellen ist jedoch aufwendig und fehleranfällig, besonders wenn es sich um umfangreiche Anwendungen handelt. Es muss sichergestellt werden, dass die selektierten Cloud-Dienste den Anforderungen der Anwendungskomponenten entsprechen und alle technischen Abhängigkeiten erfüllt werden.

Das Ziel der Verteilung ist es, die bereits erprobten Technologien aus den vorherigen Deployments wiederzuverwenden (Anforderung 4). Jede anwendungsspezifische Komponente wird dafür zuerst mit der gewünschten *Zielumgebung* annotiert. Die Granularität der spezifizierten Zielumgebung kann dabei variieren: Es kann ein konkreter Cloud-Anbieter, wie zum Beispiel „AWS", ein bestimmtes Servicemodell, beispielsweise „IaaS", oder auch ein konkreter Service, wie „AWS EC2", als Zielumgebung definiert werden. Die Selektion der Zielumgebung erfolgt manuell oder durch eines der Verfahren, die in Abschnitt 2.1.3.1 vorgestellt wurden. Für jede Zielumgebung steht eine Menge an potenziellen Cloud-Diensten zur Verfügung. Basierend auf den Anforderungen im Deployment-Modell und den bereitgestellten Fähigkeiten der Cloud-Dienste wird für jede anwendungsspezifische Komponente ein passender Dienst selektiert und das Modell entsprechend adaptiert. Dabei werden ausschließlich Middleware- und Infrastrukturkomponenten entfernt bzw. ausgetauscht, die für die selektierte Umgebung nicht mehr verwendet werden können, alle anderen Komponenten bleiben erhalten. So wird, zum Beispiel, wenn die Komponente auf einem IaaS-Dienst bereitgestellt werden soll und bereits eine konfigurierte virtuelle Maschine dafür im Modell vorhanden ist, diese, wenn möglich, wiederverwendet, und es werden ausschließlich die Informationen zum Cloud-Anbieter ausgetauscht. Damit ist für jede Verwendung der Anwendung eine individuelle Verteilung möglich und im Falle einer Änderung der Verteilungspräferenzen kann die Anwendung neu verteilt und entsprechend bereitgestellt werden.

Durch die Verteilung der Komponenten können auch hier neue Probleme auftreten. Zur Analyse potenzieller Probleme und zur Adaption in Bezug auf die Kommunikationsbeziehungen zwischen den Komponenten des LDMs kann erneut die Analyse- und Adaptionsmethode aus Schritt 2 im lokalen

Kontext des Partners angewendet werden. Das Ergebnis dieses Schritts ist ein restrukturiertes LDM, welches alle Deployment-Informationen umfasst, die für die Bereitstellung der Komponenten des Partners benötigt werden. Jeder Partner kennt das globale Modell sowie die spezifische Verteilung seiner eigenen Komponenten im partnerspezifischen LDM. Abhängig davon welche Deployment-Technologien vom jeweiligen Partner eingesetzt werden, müssen die Teile des LDMs noch in Artefakte, die von der jeweiligen Technologie verarbeitet werden können, transformiert werden (Anforderung 3) [WBB+19; WBH+20b; WBH+20c].

3.2.5 Schritt 5a: Generierung der Deployment-Choreographie

Für das Deployment wird ein hybrides Verfahren, welches deklarative und imperative Deployment-Technologien kombiniert, angewendet, wodurch ein vollständig dezentrales Deployment der gesamten Anwendung ermöglicht wird (Anforderung 7). Im fünften Schritt generiert dafür jeder Partner einen *lokalen Deployment-Workflow* basierend auf dem restrukturierten LDM des vorherigen Schritts.

Definition 3.3 (Lokaler Deployment-Workflow – informell) Ein lokaler Deployment-Workflow (LDW) wird aus einem LDM abgeleitet und (i) orchestriert alle lokalen Deployment-Aktivitäten und (ii) koordiniert das gesamte Deployment und den Datenaustausch, um die partnerübergreifenden Kommunikationsbeziehungen zu realisieren. Alle LDWs der Partner bilden implizit die Deployment-Choreografie, welche die globale Koordination des Deployments über Organisationsgrenzen hinweg ermöglicht.

Die Generierung von zentralisierten Deployment-Workflows basierend auf deklarativen Deployment-Modellen, die alle Deployment-Aktivitäten orchestrieren, wurde bereits in verschiedenen Arbeiten behandelt [BBK+14; CCDT18], jedoch werden die globale Koordination und der Informationsaustausch dabei noch nicht abgedeckt. Zur Realisierung jeder partnerübergreifenden Kommunikationsbeziehung im GDM ist eine Sende- und Empfangsaktivität erforderlich, um die Informationen nach dem Deployment

der Zielkomponente und vor dem Verbindungsaufbau auszutauschen. Darüber hinaus muss das Deployment der gesamten Anwendung sichergestellt werden. Unabhängig davon, von welchem Partner die Initialisierung des Deployments ausgeht, muss das Deployment bei allen Partnern angestoßen werden. Die Generierung der LDWs basiert auf definierten Schnittstellen, die das öffentlich sichtbare Verhalten abbilden. Die korrekte Reihenfolge sowohl der lokalen Deployment-Aktivitäten als auch der Aktivitäten zum Datenaustausch zwischen den Partnern wird aus den LDMs abgeleitet. Das Ergebnis dieses Schritts sind LDWs, die jeder Partner generiert hat und die zusammen die Deployment-Choreographie bilden.

3.2.6 Schritt 6: Dezentrales Deployment

Im letzten Schritt wird das Deployment der Anwendung durchgeführt. Dazu müssen alle Partner die generierten LDWs in einer Ausführungsumgebung bereitstellen. Jeder der Partner kann das Deployment der gesamten Anwendung initialisieren. Eine Initialisierungsnachricht wird anschließend an alle übrigen Partner geschickt, um das Deployment der jeweiligen Komponenten anzustoßen. Über die im GDM definierten Kommunikationsschnittstellen der Partner können Nachrichten (i) zur Initialisierung und (ii) zur Realisierung partnerübergreifender Kommunikationsbeziehungen ausgetauscht werden.

Beim dezentralen Deployment kommt es vor allem auf eine zuverlässige und robuste Durchführung an. Entweder alle oder keine der Komponenten ist am Ende der Durchführung installiert und betriebsbereit. Workflow-Technologien sind besonders gut dafür geeignet, um im Fehlerfall Kompensationsaktivitäten auszuführen, um einen konsistenten Zustand der Anwendung zu gewährleisten (Anforderung 8). Die Kompensation wird jedoch im Rahmen dieser Arbeit nicht detailliert betrachtet.

3.2.7 Schritt 5b (Single-Partner-Variante): Zentrales Deployment

Die DivA-Methode ist nicht nur beschränkt auf Multi-Partner-Szenarien, sondern kann auch von einem einzelnen Partner zur Verteilung, Analyse, Ad-

aption und Deployment einer Anwendung angewendet werden. Die Methode kommt dabei als ergänzendes Werkzeug beim Zusammenspiel von Softwareentwicklung und Betrieb einer Anwendung zum Einsatz, aber auch bei der individuellen Verteilung für unterschiedliche Anwendungsfälle, zum Beispiel bei der Bereitstellung der Anwendung in unterschiedlichen Umgebungen für verschiedene Nutzer.

In dieser Single-Partner-Variante werden ausschließlich die Schritte des lokalen Kontexts der Methode, wie in Abbildung 3.1 abgebildet, durchgeführt. Die Methode in der Single-Partner-Variante startet mit Schritt 3, der Beschaffung des lokalen Deployment-Modells, welches gleichzeitig das globale Modell ist, da keine weiteren Partner involviert sind. Die Verteilung, Analyse und Adaption des Deployment-Modells werden wie bereits in den Schritten 2, 4 und 5 erläutert durchgeführt. Für die Ausführung des Deployments wird jedoch kein dezentraler Ansatz benötigt. Damit fallen die Schritte 5a und 6 weg und werden durch Schritt 5b ersetzt.

Beim zentralen Deployment können bereits existierende Technologien genutzt werden. Aber auch hier kommen häufig hybride Verfahren zum Einsatz, zum einen, um nutzerspezifische Anpassungen beim Deployment vornehmen zu können [BBK+14; CCDT18], zum anderen auch um verschiedene Deployment-Technologien zu kombinieren [WBB+21]. Die in Schritt 5b erläuterten Aktivitäten zum Informationsaustausch zwischen den Partnern sind eine Ergänzung zu diesen bereits existierenden hybriden Verfahren. Damit kann die DivA-Methode sowohl in Multi-Partner- als auch in Single-Partner-Szenarien angewendet werden.

3.3 Erweiterung der DivA-Methode zur rekursiven Anwendung

Die DivA-Methode ermöglicht die dezentrale Koordination des Deployments partnerübergreifender Anwendungen. Die Methode ist dabei nicht auf die Verteilung auf einer organisatorischen Ebene beschränkt. Die Verantwortung für die Bereitstellung einzelner Anwendungskomponenten kann innerhalb einer Organisationseinheit weiter auf kleinere Organisationseinheiten aufge-

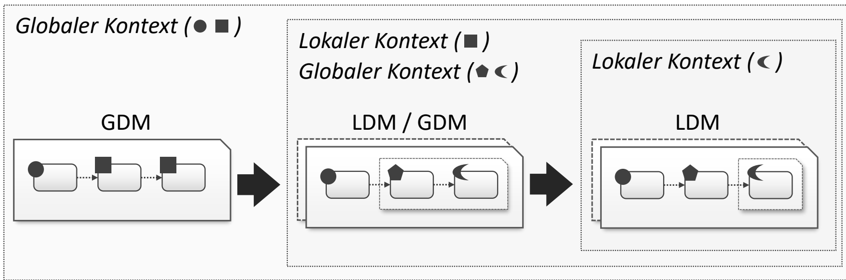

Abbildung 3.2: Rekursive Anwendung der DIVA-Methode.

teilt werden, zum Beispiel innerhalb eines Unternehmens auf verschiedene unabhängige Abteilungen.

Die Methode kann dazu rekursiv angewendet werden, wie in Abbildung 3.2 dargestellt. Beginnend mit der Zuordnung der übergeordneten Organisationseinheiten wird im globalen Kontext das GDM modelliert (Schritt 1 der DIVA-Methode) und bei Bedarf adaptiert (Schritt 2 der DIVA-Methode). Das GDM wird jedem Partner zur weiteren Verarbeitung zur Verfügung gestellt. Das LDM eines Partners kann wiederum als GDM für untergeordnete Organisationseinheiten dienen. Die Anwendungskomponenten im LDM können weiteren Organisationseinheiten zugeteilt werden (Schritt 1 der DIVA-Methode) und erneut adaptiert werden (Schritt 2 der DIVA-Methode). Die Schritte können beliebig oft rekursiv angewendet werden für jede übergeordnete Organisationseinheit.

Die Rekursivität bezieht sich dabei nur auf die ersten beiden Schritte der Methode. Die Ausführung des Deployments übernehmen jeweils die untergeordneten Organisationseinheiten. Jedoch muss dabei berücksichtigt werden, dass die im ersten GDM definierten Kommunikationsschnittstellen der Partner auch für alle untergeordneten Organisationseinheiten valide sind, um den Nachrichtenaustausch während des Deployments zu gewährleisten.

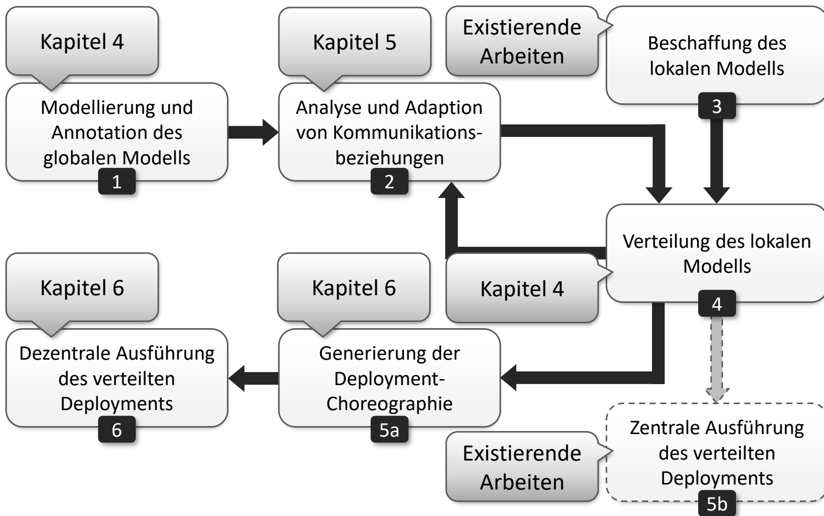

Abbildung 3.3: Umsetzung der DIVA-Methode.

3.4 Umsetzung der Anforderungen und Diskussion

Die DIVA-Methode ist Forschungsbeitrag 1 dieser Arbeit. Die einzelnen Schritte der DIVA-Methode werden durch die weiteren Forschungsbeiträge dieser Arbeit umgesetzt oder sind bereits durch existierende Arbeiten abgedeckt. Abbildung 3.3 zeigt, in welchen folgenden Kapiteln die Realisierung der Methodenschritte erläutert wird. In Kapitel 4 wird als Forschungsbeitrag 2 ein Modellierungskonzept für adaptive Deployment-Modelle eingeführt, welches die Modellierung und Transformation von GDMs und LDMs ermöglicht (Anforderung 1) und damit die Datenkapselung sicherstellt (Anforderung 2). Dieses beruht auf dem EDMM zur Ausführung des Deployments [WBF+19; WBH+20c]. Durch die Konformität mit EDMM können die restrukturierten LDMs in beliebige Technologien transformiert werden. Damit kann sichergestellt werden, dass die gesamte Methode unabhängig von einer konkreten Deployment-Technologie ist (Anforderung 3).

In Kapitel 4 werden darüber hinaus Algorithmen für die automatisierte

ressourcenspezifische Verteilung von LDMs in Schritt 4 der DIVA-Methode eingeführt. Der Fokus liegt dabei auf der Wiederverwendung von Technologiewissen und Kompatibilitäten aus vorherigen Deployments der Anwendung (Anforderung 4). Lokale Deployment-Modelle aus früheren Deployments, zum Beispiel aus der Entwicklungsphase, stehen entweder bereits in entsprechenden Repositorien als EDMM-Modelle zur Verfügung oder es können Crawling-Konzepte zur Anwendung kommen, die das Deployment-Modell durch Analyse laufender Instanzen der Anwendung, zum Beispiel in der Testumgebung, ableiten [BBKL13a; EBLW17]. Für das Zusammenführen des GDMs aus Schritt 2 und des beschafften LDMs aus Schritt 3 zur weiteren anwendungsfallspezifischen Verteilung können ebenfalls existierende Mechanismen zur Substitution von Anwendungskomponenten, wie von Harzenetter et al. [HBF+18] zur Verfeinerung von musterbasierten Deployment-Modellen eingeführt, verwendet werden.

Für die Analyse und Adaption von Kommunikationsbeziehungen in Schritt 2 wird der Forschungsbeitrag 3 in Kapitel 5 vorgestellt, eine musterbasierte Methode zur Problemerkennung und Modelladaption. Ziel ist die frühzeitige automatisierte Erkennung von potentiellen Problemen, die durch die Verteilung der Komponenten im Deployment-Modell auftreten können (Anforderung 5). Für die automatisierte Behebung der erkannten Probleme wird außerdem ein Verfahren zur Erkennung geeigneter Lösungsimplementierungen und zur automatisieren Modelladaption eingeführt (Anforderung 6).

Für die Ausführung des Deployments werden die restrukturierten LDMs aus Schritt 4 zuerst in die verwendeten Deployment-Technologien transformiert. Dabei können für ein LDM auch mehrere Technologien für unterschiedliche Teile verwendet werden. Dafür und auch für die zentrale Ausführung in Schritt 5b können die existierenden Arbeiten von Wurster et al. [WBB+19; WBB+21; WBH+20b; WBH+20c] verwendet werden.

Für die dezentrale Ausführung des Deployments, Schritt 5a und 6 der DIVA-Methode, wird in Kapitel 6 der Forschungsbeitrag 4 vorgestellt. Workflows werden basierend auf den deklarativen Deployment-Modellen der Partner generiert. Die Workflows können dabei das Deployment der Anwendungskomponenten mit unterschiedlichen Technologien orchestrieren.

Durch die Verwendung von Workflows wird die vollständig dezentrale Koordination des Deployments der gesamten Anwendung ermöglicht (Anforderung 7). Darüber hinaus sind Workflow-Technologien ein Werkzeug für die zuverlässige und robuste Ausführung von Anwendungen (Anforderung 8). Das gesamte DivA-Werkzeug, das die Automatisierung der DivA-Methode realisiert, wird in Kapitel 7 vorgestellt und ist der letzte Forschungsbeitrag, Forschungsbeitrag 5, dieser Arbeit.

Die bereits diskutierte Technologieunabhängigkeit der Methode bezieht sich nicht nur auf die verwendeten Deployment-Technologien, sondern auch auf die Workflow-Technologien. Jeder Partner kann eine beliebige Worflow-Technologie einsetzen. Eine prototypische Implementierung wurde im Rahmen der Arbeit zur Validierung mit BPEL und BPMN umgesetzt. Nur für die Modellierung, Adaption und Restrukturierung werden neue Modellierungskonstrukte benötigt, die als Erweiterung von EDMM umgesetzt sind. Diese Erweiterung betrifft jedoch nur die Entwurfsphase und nicht die Deployment-Phase der Anwendung und hat damit keinen Einfluss auf die Transformationsmöglichkeiten von EDMM-Modellen in unterschiedliche Deployment-Technologien.

Kapitel 4

Modellieren und Verteilen adaptiver Deployment-Modelle

Dieses Kapitel stellt Forschungsbeitrag 2 dieser Arbeit vor. Als Grundlage wird zuerst das EDMM zur technologieagnostischen Modellierung von Deployment-Modellen in Abschnitt 4.1 vorgestellt. Für die Berücksichtigung von Organisationseinheiten sowie zur Verteilung von Anwendungskomponenten und zur Modelladaption werden in Abschnitt 4.2 Erweiterungen des Metamodells eingeführt. Die Konzepte zur organisatorischen und ressourcenspezifischen Verteilung sowie zur Restrukturierung von Anwendungskomponenten werden anschließend in Abschnitt 4.3 und Abschnitt 4.4 präsentiert. Die in diesem Kapitel eingeführten Konzepte basieren teilweise auf bereits im Rahmen dieser Arbeit veröffentlichten wissenschaftlichen Publikationen [SBK+18; SBKL17; SBKL19b].

4.1 Essential Deployment Meta Model (EDMM)

Das *Essential Deployment Meta Model (EDMM)* wurde in Zusammenarbeit mit Wurster et al. [WBF+19] als Ergebnis der Analyse von 13 deklarativen Deployment-Technologien spezifiziert. Das EDMM umfasst alle Modellierungselemente, die von den untersuchten Technologien unterstützt werden. EDMM ermöglicht (i) die technologieagnostische Modellierung von deklarativen Deployment-Modellen und (ii) die Transformation in die technologiespezifischen Modelle der 13 untersuchten Deployment-Technologien zur Ausführung des Deployments [WBB+19; WBB+20b].

Die Transformation definiert dabei nicht ausschließlich die syntaktische Abbildung in die DSL der Zieltechnologie, sondern abhängig von den generellen Eigenschaften der Technologie auch eine semantische Transformation, die abhängig von der Kategorie der Technologie ist. Die untersuchten Technologien lassen sich in drei Kategorien einteilen [WBF+19]: Allzweck (*engl. „general-purpose“*), Anbieterspezifisch (*engl. „provider-specific“*) und Plattformspezifisch (*engl. „platform-specific“*). Die Allzweck-Technologien unterstützen verschiedene Anbieter und Servicemodelle, wohingegen die Anbieterspezifischen auf die Serviceangebote eines spezifischen Anbieters und die Plattformspezifischen auf ein Servicemodell oder eine konkrete Plattform und teilweise auch auf bestimmte Artefakttypen, zum Beispiel Docker Container, beschränkt sind. Zu den Allzweck-Technologien gehören unter anderem Terraform, Chef oder Ansible, zu den Anbieterspezifischen Angebote von Cloud-Anbietern wie AWS CloudFormation und zu den Plattformspezifischen Technologien wie Kubernetes oder Docker Compose. Die semantische Transformation ermöglicht unter anderem den Austausch von äquivalenten Services des jeweiligen Anbieters bei einer Transformation in eine anbieterspezifische Technologie oder auch die Containerisierung von Anwendungskomponenten, wenn in EDMM ein IaaS-basiertes Deployment spezifiziert ist, jedoch Technologien wie Kubernetes selektiert werden.

Das dieser Arbeit zugrundeliegende, auf EDMM basierende Metamodell mit seinen Klassen, deren Relationen und speziell den Erweiterungen dieser Arbeit, die in Abschnitt 4.2 vorgestellt werden, ist in Abbildung 4.1 graphisch

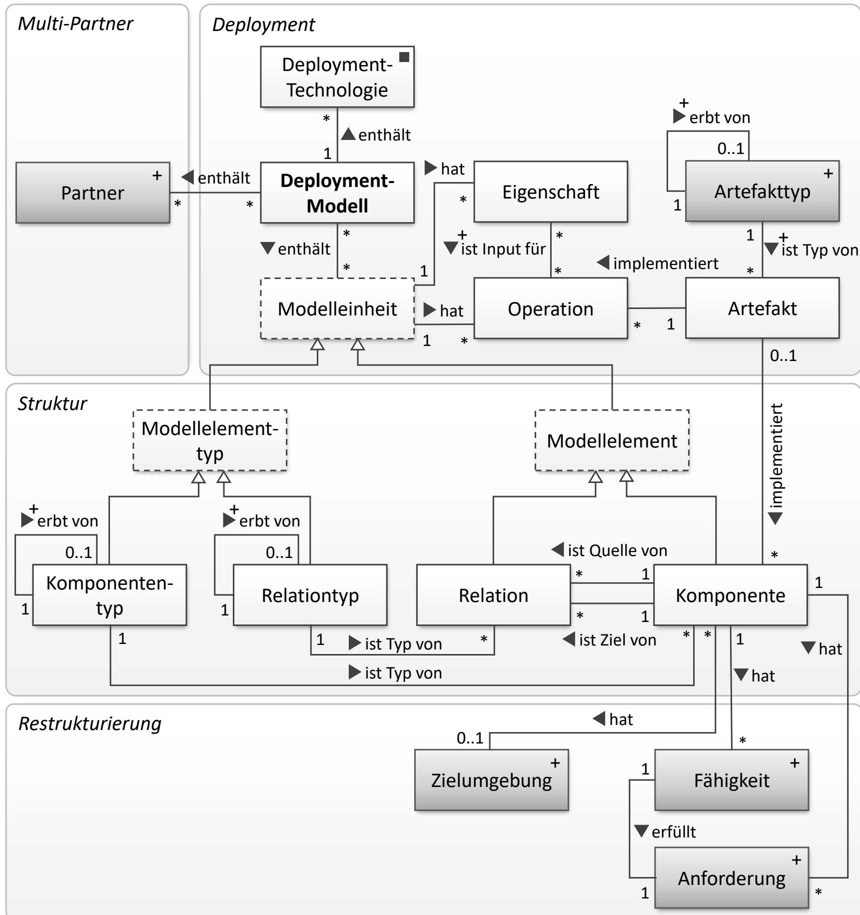

Abbildung 4.1: Erweitertes Metamodell basierend auf EDMM [WBF+19]. Alle mit (+) markierten Elemente wurden im Rahmen der Arbeit erweitert. Das mit (■) markierte Elemente ist eine Erweiterung aus einer anderen Arbeit [WBB+21].

dargestellt. Abstrakte Klassen sind dabei durch eine gestrichelte Umrandung dargestellt. Für die formale Definition eines Deployment-Modells sei Σ die Menge der Zeichen in der ASCII-Tabelle. Damit ist die Menge aller Wörter über Σ, außer dem leeren Wort, mit Σ^+ definiert. Für die übersichtliche Darstellung von Tupeln wird die Projektion π_j verwendet, die wie folgt definiert ist: Seien $X_1, ..., X_n$ beliebige Mengen, dann ist das kartesische Produkt $X_I = X_i \times ... \times X_n$ die Menge aller n-Tupel, deren i-te Komponente, $1 \leq i \leq n$, ein Element $x_i \in X_i$ ist. Die Projektion π_j, $1 \leq j \leq n$, ist dann die Abbildung $\pi_j : X_1 \times ... \times X_n \rightarrow X_j, (x_1, ..., x_n) \mapsto x_j$, die ein Tupel auf seine j-te Komponente abbildet.

Im Folgenden werden die Klassen des EDMM (ohne Erweiterungen) formal definiert. Die Erweiterungen dieser Arbeit werden anschließend in Abschnitt 4.2 erläutert und definiert. Die formale Definition ist dabei inspiriert von der *Declarative Application Management Modeling and Notation (DMMN)* [Bre16]. Für EDMM gibt es noch keine veröffentlichte formale Spezifikation. Im Folgenden wird ausschließlich auf die Attribute eingegangen, die für diese Arbeit wichtig sind. Die zentrale Klasse ist das *Deployment-Modell* und ist wie folgt definiert:

Definition 4.1 (Deployment-Modell) Ein deklaratives Deployment-Modell ist ein gerichteter azyklischer Graph mit gefärbten Knoten und gewichteten Kanten. Sei D die Menge aller Deployment-Modelle, dann wird $d \in D$ definiert als Tupel:

$$d = (K_d, R_d, KT_d, RT_d, E_d, O_d, A_d, DT_d, typ_d,$$
$$operationen_d, eigenschaften_d, artefakt_d, techgruppe_d)$$

Die Mengen im Tupel repräsentieren die Klassen des EDMMs und sind dabei wie folgt definiert:

- $K_d \subseteq \Sigma^+$ ist die Menge aller *Komponenten* in d, wobei jedes Element $k_i \in K_d$ die Identität einer Komponente darstellt.

- $R_d \subseteq \Sigma^+ \times K_d \times K_d$ ist die Menge aller *Relationen* in d, wobei jedes Element $r_i = (Id, k_s, k_t) \in R_d$ eine eindeutig identifizierbare Relation zwischen dem Quellknoten k_s und dem Zielknoten k_t beschreibt.
- $KT_d \subseteq \Sigma^+ \times \{wahr, falsch\}$ definiert die Menge der *Komponententypen*. Jedes Element $kt_d = (Id, Abstrakt) \in KT_d$ spezifiziert einen eindeutig identifizierbaren Typ, der die Semantik von Komponenten der Anwendung beschreibt. $Abstrakt \in \{wahr, falsch\}$ ist ein Wahrheitswert, der ausdrückt, ob es sich um einen instanziierbaren oder abstrakten Typ handelt.
- $RT_d \subseteq \Sigma^+ \times \{wahr, falsch\}$ definiert die Menge der *Relationtypen*. Jedes Element $rt_d = (Id, Abstrakt) \in RT_d$ spezifiziert einen eindeutig identifizierbaren Typ, der die Semantik von Relationen der Anwendung beschreibt. $Abstrakt \in \{wahr, falsch\}$ ist analog zu KT_d definiert.
- $E_d \subseteq \Sigma^+ \times \Sigma^+ \times (\Sigma^+ \cup \{\perp\})$ ist eine Menge von *Eigenschaften*. Ein Eigenschaftselement $e_d = (Id, Datentyp, Wert) \in E_d$ spezifiziert die Konfiguration einer Komponente oder Relation der Anwendung. Dabei ist $E_d \subseteq DE$, wobei DE die Menge aller Datenelemente ist. Der *Datentyp* spezifiziert, welche Eingabe für den *Wert* erlaubt ist, zum Beispiel nur Integer oder String Werte. Falls $Wert = \perp$, ist der Wert der Eigenschaft unspezifiziert. Der Wert kann jedoch als Eingabe zur Deploymentzeit benötigt werden, wenn die Eigenschaft von einer Operation als Eingabeparameter referenziert wird.
- $O_d \subseteq \Sigma^+ \times \wp(Parameter_d) \times \wp(Parameter_d)$ ist die Menge der *Operationen*, wobei jede Operation $o_i = (Id, Eingabeparameter, Ausgabeparameter) \in O_d$ auf eine Komponente oder Relation angewendet werden kann und die benötigten Eingabeparameter und Ausgabeparameter spezifiziert. Ein Parameter aus der Menge $Parameter \subseteq DE \subseteq \Sigma^+ \times \Sigma^+ \times (\Sigma^+ \cup \{\perp\})$ wird dabei definiert als $param_i = (Id, Datentyp, Wert)$. Ein Parameter und eine Eigenschaft können das selbe Datenelement referenzieren. Damit können Eigenschaftswerte als Eingabeparameter für Operationen dienen.

- $A_d \subseteq \Sigma^+ \times \Sigma^+$ definiert die Menge aller *Artefakte* in d. Jedes Artefakt $a_i = (Id, Artefaktreferenz) \in A_d$ spezifiziert die Implementierung für eine Komponente oder Operation. Die *Artefaktreferenz* referenzieren dabei die eigentliche Implementierung, zum Beispiel ein Skript oder eine ZIP-Datei.
- $DT_d \subseteq \Sigma^+$ bezeichnet die *Deployment-Technologien*, die für das Deployment der Komponenten in d verwendet werden, wobei jede Technologie $dt_i \in DT_d$ durch einen eindeutigen Namen identifiziert wird.

Zusätzlich zu den konkreten Klassen werden drei abstrakte Klassen in EDMM eingeführt, die für die Definition der Abbildungen verwendet werden: Die Menge der *Modellelemente* ist die Vereinigung aller Komponenten und Relationen in d, $ME_d := K_d \cup R_d$, und die Menge der *Modellelementtypen* die Vereinigung der Typen, $MET_d := KT_d \cup RT_d$. Sie bilden zusammen die Menge der *Modelleinheiten* definiert als $M_d := ME_d \cup MET_d$. Die folgenden Abbildungen werden für ein Deployment-Modell d definiert:

- typ_d bildet jedes Modellelement $me_i \in ME_d$ auf genau einen Modellelementtyp $met_i \in MET_d$ ab und bestimmt damit die Färbung der Knoten bzw. die Gewichtung der Kanten des Graphen: $typ_d : ME_d \rightarrow MET_d$ mit $typ\,|_k : ME_d \rightarrow KT_d$ und $typ\,|_r : ME_d \rightarrow RT_d$.
- $operationen_d$ bildet jede Modelleinheit $m_i \in M_d$ auf die Menge an Operationen ab, die zum Deployment und Management der Modelleinheit benötigt werden: $operationen_d : M_d \rightarrow \wp(O_d)$.
- $eigenschaften_d$ bildet jede Modelleinheit $m_i \in M_d$ auf die Menge an Eigenschaften ab, die zur Konfiguration und Beschreibung benötigt werden: $eigenschaften_d : M_d \rightarrow \wp(E_d)$.
- $artefakt_d$ bildet jede Komponente $k_i \in K_d$ und jede Operation $o_i \in O_d$ auf das Artefakt ab, welches die Komponente bzw. Operation implementiert. Eine Komponente kann dabei auch von keinem Artefakt implementiert werden, wenn es sich zum Beispiel um einen Service handelt, und somit auch auf $\perp$ abgebildet werden: $artefakt_d : K_d \cup O_d \rightarrow A_d \cup \{\perp\}$.

- *techgruppe*$_d$ bildet jede Komponente $k_i \in K_d$ auf die Deployment-Technologie $dt_i \in DT_d$ ab, die zum Deployment verwendet werden soll. Diese Abbildung ermöglicht das Deployment von Teilen der Anwendung mit unterschiedlichen Technologien: $techgruppe_d : K_d \rightarrow DT_d$.

Die Klasse der Deployment-Technologien DT_d und die Abbildung $techgruppe_d$ sind Erweiterungen des EDMMs, die in Zusammenarbeit mit Wurster et al. [WBB+21] eingeführt wurden, um die Nutzung und Orchestrierung mehrere Technologien zum Deployment einer Anwendung zu ermöglichen. Damit können, zum Beispiel für die Bereitstellung der Infrastruktur, andere Technologien verwendet werden als für die Bereitstellung und Konfiguration der Anwendungskomponenten. Diese Erweiterung wird zur Entwurfszeit benötigt, um die Transformation in technologiespezifische Deployment-Modelle zu ermöglichen. Diese Elemente spielen im Kontext der einzelnen Deployment-Technologien keine Rolle für die Ausführung des Deployments selbst und müssen daher nicht auf die DSL der verschiedenen Technologien abgebildet werden können.

4.2 Erweiterungen von EDMM

Das EDMM wurde im Rahmen dieser Arbeit erweitert, um (i) Organisationseinheiten abzubilden sowie (ii) die Verteilung und Adaption zu ermöglichen. In Abbildung 4.1 sind die erweiterten Klassen dunkelgrau hinterlegt und die neuen Klassen sowie Relationen mit einem (+) markiert. Die Definition 4.1 wird erweitert und ein Deployment-Modell im Rahmen dieser Arbeit wie folgt definiert:

Definition 4.2 (Erweitertes Deployment-Modell) Ein deklaratives Deployment-Modell für partnerübergreifende Anwendungen ist ein nach Definition 4.1 spezifiziertes Deployment-Modell mit zusätzlichen Klassen zur Spezifikation von Partnern, Zielumgebungen, Artefakttypen, Fähigkeiten und Anforderungen. Es ermöglicht darüberhinaus die Definition von Vererbungshierarchien. Sei D die Menge aller Deployment-Modelle, dann wird $d \in D$

definiert als erweitertes Tupel:

$$d = (K_d, R_d, KT_d, RT_d, E_d, O_d, A_d, DT_d, \boldsymbol{P_d}, \boldsymbol{F_d}, \boldsymbol{AF_d}, \boldsymbol{Z_d}, \boldsymbol{AT_d}, \boldsymbol{typ_d},$$
$$operationen_d, eigenschaften_d, artefakt_d, techgruppe_d, \boldsymbol{supertyp_d},$$
$$\boldsymbol{partner_d}, \boldsymbol{ziel_d}, \boldsymbol{f\ddot{a}higkeiten_d}, \boldsymbol{anforderungen_d})$$

Dabei ist:

- P_d die Menge aller *Partner* in d (Definition 4.7),
- F_d die Menge aller *Fähigkeiten* in d (Definition 4.9),
- AF_d die Menge aller *Anforderungen* in d (Definition 4.11),
- Z_d die Menge aller *Zielumgebungen* in d (Definition 4.14),
- AT_d die Menge aller *Artefakttypen* in d (Definition 4.3),
- typ_d eine erweiterte Abbildung, die jedem Modellelement und jedem Artefakt dessen Typ zuordnet (Definition 4.4),
- $supertyp_d$ eine Abbildung, die jedem Modellelementtyp und Artefakttyp dessen Supertyp zuordnet (Definition 4.5),
- $partner_d$ eine Abbildung, die jeder Komponente einen Partner zuordnet (Definition 4.8),
- $ziel_d$ eine Abbildung, die jeder Komponente eine Zielumgebung zuordnet (Definition 4.15),
- $f\ddot{a}higkeiten_d$ eine Abbildung, die jeder Komponente dessen Fähigkeiten zuordnet (Definition 4.10),
- $anforderungen_d$ eine Abbildung, die jeder Komponente dessen Anforderungen zuordnet (Definition 4.12).

Diese Erweiterung hat ebenfalls keinen Einfluss auf die Transformation in die verschiedenen technologiespezifischen Modelle und wird ausschließlich zur Verteilung und Modelladaption zur Entwurfszeit und zur Generierung der Workflows benötigt. Das Deployment mit einer konkreten Technologie bleibt unberührt, weshalb diese Erweiterungen nicht auf die DSL der jeweiligen Technologien abgebildet werden müssen.

4.2.1 Modellierung von Artefakten

Für die Modelladaption und Selektion passender Kommunikationstreiber (Abschnitt 5.5) muss die Semantik von Artefakten in einem Deployment-Modell definiert werden können. Angelehnt an den TOSCA-Standard [OAS13] werden dazu *Artefakttypen* eingeführt:

Definition 4.3 (Artefakttyp) Zur Spezifikation der Typen der Artefakte in einem Deployment-Modell d wird die Menge $AT_d \subseteq \Sigma^+ \times \{wahr, falsch\}$ definiert. Jedes Element $at_i = (Id, Abstrakt) \in AT_d$ wird durch einen eindeutigen Namen identifiziert und $Abstrakt \in \{wahr, falsch\}$ ist analog zu KT_d definiert.

Definition 4.4 (Erweiterung Abbildung typ_d) Die Abbildung typ_d bildet bisher jedes Modellelement $me_i \in ME_d$ auf genau einen Modellelementtyp $met_i \in MET_d$ ab und bestimmt damit die Färbung der Knoten bzw. die Gewichtung der Kanten des Graphen. Die Abbildung wird erweitert, um jedes Artefakt $a_i \in A_d$ auf genau einen Artefakttyp $at_i \in AT_d$ abzubilden: $typ_d : ME_d \cup A_d \rightarrow MET_d \cup AT_d$.

4.2.2 Modellierung von Vererbung

Für die musterbasierte Erkennung und Behebung von Problemen in Deployment-Modellen reicht eine einfache Typisierung der Modellelemente nicht aus. Die Probleme bzw. der Deployment-Kontext der Lösungsimplementierungen können meist auf abstrakten Elementtypen definiert werden. Dadurch wird eine höhere Generalisierung erreicht und die Regeln zur Problemerkennung und -behebung können breiter angewendet werden. Dafür wird die Abbildung $supertyp_d$ auf Basis der Definition von Breitenbücher [Bre16] wie folgt definiert:

Definition 4.5 (Abbildung $supertyp_d$) Die Abbildung bildet jeden Modellelementtyp $met_i \in MET_d$ auf einen oder keinen anderen Modellelementtyp $met_j \in MET_d$ ab, sowie jeden Artefakttyp $at_i \in AT_d$ auf einen oder keinen anderen Artefakttyp $at_j \in AT_d$, wobei $i \neq j$ gilt. Der Modellelementtyp met_j bzw. der Artefakttyp at_j heißt *Supertyp* vom Modellelementtyp met_i bzw.

Artefakttyp at_i und dieser erbt definierte Eigenschaften und Operationen des Supertyps. Hat ein Modellelementtyp bzw. Artefakttyp keinen Supertyp wird dies durch $\perp$ ausgedrückt: $supertyp_d : MET_d \cup AT_d \rightarrow MET_d \cup AT_d \cup \{\perp\}$.

Um darüber hinaus die Menge aller direkter und transitiver Supertypen eines Modellelementtyps bzw. Artefakttyps abbilden zu können, wird nach Breitenbücher [Bre16] eine weitere Abbildung definiert:

Definition 4.6 (Abbildung *supertypen$_d$*) Die Abbildung bildet jeden Modellelementtyp $met_i \in MET_d$ auf alle Modellelementtypen $met_j \in MET_d$ ab von denen er direkt oder transitiv erbt, wobei $i \neq j$ gilt. Das gleiche gilt für alle Artefakttypen: $supertypen_d : MET_d \cup AT_d \rightarrow \wp(MET_d) \cup \wp(AT_d)$.

4.2.3 Modellierung von Partnern

Zur Modellierung von Organisationseinheiten in Deployment-Modellen wird die Klasse der *Partner* eingeführt, die wie folgt definiert ist:

Definition 4.7 (Partner) Die Menge der in einem Deployment-Modell $d \in D$ involvierten Partner wird als $P_d \subseteq \Sigma^+ \times \Sigma^+$ definiert. Jedes Element $p_i = (Id, Endpunkt) \in P_d$ wird durch einen eindeutigen Namen identifiziert und spezifiziert den Kommunikationsendpunkt, der für den Austausch von Nachrichten während des Deployments benötigt wird.

Jede Komponente muss einem Partner zugeordnet werden, um die Verteilung der Zuständigkeit für das Deployment der einzelnen Komponenten zu spezifizieren. Dafür wird die folgende Abbildung definiert:

Definition 4.8 (Abbildung *partner$_d$*) Die Abbildung bildet jede Komponente $k_i \in K_d$ auf genau einen Partner $p_i \in P_d$ ab: $partner_d : K_d \rightarrow P_d$.

4.2.4 Modellierung von Fähigkeiten und Anforderungen

Für die Komposition von Services und zur Selektion geeigneter Cloud-Anbieter und Cloud-Dienste ermöglichen Modellierungssprachen wie TOSCA und Aeolus die Spezifikation von *Fähigkeiten* und *Anforderungen* für Komponenten in

Deployment-Modellen, durch deren Abgleich geeignete Dienste identifiziert werden können (siehe Abschnitt 2.1). Dieser Mechanismus wird in dieser Arbeit auch verwendet und wie folgt definiert:

Definition 4.9 (Fähigkeit) Zur Spezifikation der Fähigkeiten von Komponenten in einem Deployment-Modell $d \in D$ wird die Menge $F_d \subseteq \Sigma^+ \times \{Container, Endpunkt\}$ definiert. Jedes Element $f_i = (Id, Basistyp) \in F_d$ wird durch einen eindeutigen Namen identifiziert. Der Basistyp spezifiziert welcher Basisrelationtyp zur Erfüllung einer Anforderung durch diese Fähigkeit verwendet werden muss. Beim Basistyp „Container" handelt es sich um eine Bereitstellungsfähigkeit, welche mit einer Relation vom Typ „HostedOn" oder einem davon erbenden Relationtyp verwendet werden kann. Beim Basistyp „Endpunkt" handelt es sich um eine Kommunikatiosschnittstelle, welche mit einer Relation vom Typ „ConnectsTo" oder einem davon erbenden Relationtyp verknüpft werden kann.

Zur Zuweisung von Fähigkeiten wird folgende Abbildung definiert:

Definition 4.10 (Abbildung $fähigkeiten_d$) Jeder Komponente $k_i \in K_d$ kann eine Menge an Fähigkeiten zugeordnet werden, die sie anderen Komponenten anbietet: $fähigkeiten_d : K_d \rightarrow \wp(F_d)$.

Analog zu den Fähigkeiten werden Anforderungen definiert, die Komponenten zugeordnet werden können:

Definition 4.11 (Anforderung) Zur Spezifikation der Anforderungen von Komponenten in einem Deployment-Modell $d \in D$ wird die Menge $AF_d \subseteq \Sigma^+ \times F_d$ definiert. Jedes Element $af_i = (Id, benötigteFähigkeit) \in AF_d$ wird durch einen eindeutigen Namen identifiziert, und die benötigte Fähigkeit zur Erfüllung der Anforderung wird definiert.

Zur Zuweisung von Anforderungen wird folgende Abbildung definiert:

Definition 4.12 (Abbildung $anforderungen_d$) Jeder Komponente $k_i \in K_d$ kann eine Menge an Anforderungen zugeordnet werden, die durch eine andere Komponente erfüllt werden muss: $anforderungen_d : K_d \rightarrow \wp(AF_d)$.

Alle Anforderungen von Komponenten in $d \in D$ müssen erfüllt sein bevor eine Komponente bereitgestellt werden kann. Eine Anforderung kann, zum Beispiel, das Vorhandensein einer bestimmten Laufzeitumgebung oder auch eine Verbindung zu einer SQL-Datenbank sein.

Definition 4.13 (Erfüllbarkeit von Anforderungen) Für die Erfüllung einer Anforderung gilt: Sei $k_i \in K_d$ und $af_i \in anforderungen_d(k_i)$ mit $\pi_2(af_i) = f_i$ und $\exists k_j \in K_d$ mit $f_i \in f\ddot{a}higkeiten_d(k_j)$, dann ist die Erfüllung von af_i durch f_i wie folgt definiert:

$$\begin{aligned} af_i \sim_e f_i \Leftrightarrow \exists r_i \in R_d : \pi_2(r_i) = k_i \wedge \pi_3(r_i) = k_j \wedge \\ \Big(\big(\pi_2(f_i) = Container \wedge (\pi_1(typ_d(r_i)) = hostedOn \vee \\ (\exists rt_i \in supertypen_d(typ_d(r_i)) : \pi_1(rt_i) = hostedOn))\big) \vee \\ \big(\pi_2(f_i) = Endpunkt \wedge (\pi_1(typ_d(r_i)) = connectsTo \vee \\ (\exists rt_i \in supertypen_d(typ_d(r_i)) : \pi_1(rt_i) = connectsTo))\big)\Big). \end{aligned}$$

Die Definition beruht auf den Ergebnissen von Wurster et al. [WBB+20c], wonach grundsätzlich zwei Typen von Relationen, *„HostedOn“* und *„ConnectsTo“*, ausreichen um das Deployment einer komponentenbasierten Anwendung zu beschreiben.

4.2.5 Modellierung von Zielumgebungen

Für die ressourcenspezifische Verteilung von Komponenten werden Zielumgebungen definiert, die eine automatisierte Verteilung und Adaption ermöglichen.

Definition 4.14 (Zielumgebung) Zur Spezifikation von Zielumgebungen, die zur Verteilung von Komponenten in einem Deployment-Modell $d \in D$ verwendet werden können, wird die Menge $Z_d \subseteq \Sigma^+$ definiert. Jedes Element $z_i = (Id) \in Z_d$ wird durch einen eindeutigen Namen identifiziert.

Zur Zuweisung einer Zielumgebung zu einer Komponente wird folgende Abbildung definiert:

Definition 4.15 (Abbildung $ziel_d$) Die Abbildung bildet jede Komponente $k_i \in K_d$ auf genau eine oder keine Zielumgebung $z_i \in Z_d$ ab: $ziel_d : K_d \rightarrow Z_d \cup \{\perp\}$.

Die ressourcenspezifische Verteilung basierend auf definierten Zielumgebungen wird in Abschnitt 4.4 vorgestellt.

4.3 Modellierung von globalen Deployment-Modellen

Das erweiterte EDMM ist die Grundlage für die in dieser Arbeit vorgestellten Modellierungskonzepte. In diesem Abschnitt wird die Modellierung von GDMs und die Annotation der Partner und damit Schritt 1 der DIVA-Methode detailliert vorgestellt. Die Erstellung eines GDMs umfasst zwei Schritte: (1) Die Modellierung der anwendungsspezifischen Komponenten und (2) die Generierung der Platzhalter für die Hosting-Umgebungen der anwendungs-

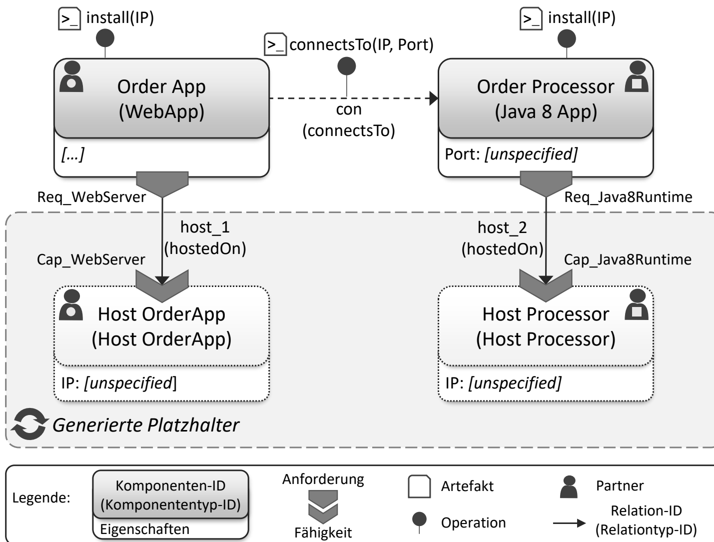

Abbildung 4.2: Beispiel eines GDM mit generierten Platzhaltern.

spezifischen Komponenten. Die generierten Platzhalter werden später für die Substitution durch konkrete Komponenten des jeweiligen Partners benötigt und definieren Eigenschaften, die zur Bereitstellung der anwendungsspezifischen Komponenten benötigt werden.

In Abbildung 4.2 ist ein beispielhaftes GDM abgebildet. Die verwendete graphische Notation ist angelehnt an die VINO4TOSCA-Notation [BBK+12]. Das GDM repräsentiert eine Bestellanwendung, die aus einer Webkomponente (*„Order App“*) und einer Verarbeitungskomponente (*„Order Processor“*) besteht. Dies sind die anwendungsspezifischen Komponenten der Anwendung. Anwendungsspezifische Komponenten repräsentieren die Geschäftslogik und können nicht zur Bereitstellung anderer Komponenten genutzt werden. Für die Menge der anwendungsspezifische Komponenten $AK \subseteq K_d$ gilt:

$$\forall k_i \in AK \nexists r_i \in R_d : \Big(\pi_3(r_i) = k_i \wedge \big(\pi_1(typ_d(r_i)) = hostedOn \vee (\exists rt_i \in supertypen_d(typ_d(r_i)) : \pi_1(rt_i) = hostedOn)\big)\Big).$$

Für die Identifikation geeigneter Middleware- und Infrastrukturkomponenten zur Bereitstellung müssen außerdem die Anforderungen der anwendungsspezifischen Komponenten definiert werden. Dies ist vor allem für die spätere Substitution und Verteilung wichtig. Die dunkelgrau dargestellten anwendungsspezifischen Komponenten in Abbildung 4.2 spezifizieren Anforderungen, zum Beispiel benötigt die „Order App“ einen Webserver. Diese Anforderungen müssen später von der jeweiligen Hosting-Umgebung erfüllt werden. Weiterhin sind für diese Komponenten sowie deren „Connects-To“-Beziehungen Operationen mit Eingabeparametern definiert. Um eine Verbindung zur „Order Processor“-Komponente aufzubauen, werden die IP-Adresse und die Portnummer benötigt. Entweder stellt die Zielkomponente, in diesem Fall die „Order Processor“-Komponente, die entsprechenden Informationen als Eigenschaften zur Verfügung, wie es bei der Portnummer der Fall ist, oder diese müssen von Middleware- oder Infrastrukturkomponenten bereitgestellt werden, wie in diesem Beispiel die IP-Adresse. Dazu werden *Platzhalter*-Komponenten generiert, die alle Fähigkeiten und Eigenschaften

enthalten, die von den später selektierten Middleware- und Infrastrukturkomponenten bereitgestellt werden müssen, um ein valides Deployment-Modell zu erhalten.

Die Generierung der Platzhalter-Komponenten wird von Algorithmus 4.1 beschrieben. Dazu werden alle anwendungsspezifischen Komponenten mit unerfüllten Anforderungen überprüft (Zeile 3-34). Für jede dieser Komponenten wird (i) ein abstrakter Komponententyp, (ii) eine Platzhalter-Komponente dieses Typs und (iii) eine HostedOn-Relation zwischen den Komponenten zum Deployment-Modell hinzugefügt (Zeile 5-8). Anschließend werden die benötigten Fähigkeiten zur Erfüllung der Anforderungen ergänzt (Zeile 9-13). Des Weiteren muss sicher gestellt werden, dass zur Deploymentzeit alle Eingabeparameter der Lebenszyklusoperationen der anwendungsspezifischen Komponente (Zeile 14-20) sowie der Kommunikationsbeziehungen (Zeile 21-32) im Deployment-Modell repräsentiert sind. Diese werden als Eigenschaften der Platzhalter-Komponente hinzugefügt.

Das resultierende GDM mit allen generierten Platzhaltern samt deren Fähigkeiten und Eigenschaften ist beispielhaft in Abbildung 4.2 gezeigt. Das abgebildete GDM umfasst auch zusätzlich bereits die Artefakte, die die Operationen implementieren. Diese sind jedoch nicht zwingend erforderlich und können auch später durch den jeweiligen Partner hinzugefügt werden. Die Artefakte gehören nicht zu der minimalen Menge an Informationen, die global verfügbar sein müssen, um ein dezentrales partnerübergreifendes Deployment zu ermöglichen.

Für die Spezifikation der Verteilung der Komponenten auf die beteiligten Partner müssen abschließend die Komponenten mit den entsprechenden Partnern annotiert werden. Das GDM wird an alle Partner verteilt, um die lokale partnerspezifische Verarbeitung zu ermöglichen.

Algorithmus 4.1 GenerierePlatzhalter(d)

1: *// Füge Platzhalter-Komponenten für alle anwendungsspezifischen Komponenten*
2: *// mit unerfüllten Anforderungen ein*
3: **for all** $(k_i \in AK_d \mid \exists af_i \in anforderungen_d(k_i) : (\nexists k_j \in K_d :$
4: $\exists f_i \in fähigkeiten_d(k_j) : af_i \sim_e f_i))$ **do**
5: **let** kt_h, k_h, r_h
6: $KT_d := KT_d \cup \{kt_h\}, \pi_2(kt_h) := wahr$
7: $K_d := K_d \cup \{k_h\}, typ_d(k_h) := kt_h$
8: $R_d := R_d \cup \{r_h\}, \pi_2(r_h) := k_i, \pi_3(r_h) := k_h$
9: $typ_d(r_h) := rt_h, \pi_1(rt_h) = hostedOn$
10: *// Füge die benötigten Fähigkeiten zur Platzhalter-Komponente*
11: **for all** $(af_i \in anforderungen_d(k_i) \mid \nexists k_j \in K_d :$
12: $\exists f_i \in fähigkeiten_d(k_j) : af_i \sim_e f_i)$ **do**
13: $fähigkeiten_d(k_h) := fähigkeiten_d(k_h) \cup \{\pi_2(af_i)\}$
14: **end for**
15: *// Füge die Eigenschaften zur Platzhalter-Komponente, die von den Operationen*
16: *// der anwendungsspezifischen Komponente als Eingabe benötigt werden*
17: **for all** $(o_i \in operationen_d(k_i))$ **do**
18: **for all** $(param_k \in \pi_2(o_i) : param_k \notin eigenschaften_d(k_i))$ **do**
19: $eigenschaften_d(k_i) := eigenschaften_d(k_i) \cup \{param_k\}$
20: **end for**
21: **end for**
22: *// Prüfe alle eingehenden ConnectsTo-Relationen*
23: **for all** $(r_i \in R_d \mid \pi_3(r_i) = k_i \wedge (\pi_1(typ_d) = connectsTo \vee$
24: $(\exists rt_i \in supertypen_d(typ_d(r_i)) : \pi_1(rt_i) = connectsTo)))$ **do**
25: *// Füge die Eigenschaften zur Platzhalter-Komponente, die von den*
26: *// Operationen der Relation als EIngabe benötigt werden*
27: **for all** $(o_r \in operationen_d(r_i))$ **do**
28: **for all** $(param_o \in \pi_2(o_r) : param_o \notin eigenschaften_d(k_i) \wedge$
29: $param_o \notin eigenschaften_d(\pi_2(r_i)))$ **do**
30: $eigenschaften_d(k_h) := eigenschaften_d(k_h) \cup \{param_o\}$
31: **end for**
32: **end for**
33: **end for**
34: **end for**

4.4 Verteilung von lokalen Deployment-Modellen

Das annotierte GDM spezifiziert, welcher Partner für die Bereitstellung welcher Komponenten verantwortlich ist. Das GDM umfasst nur die Informationen, die global zur Koordination des gesamten Deployments benötigt werden. Wie die einzelnen Komponenten bereitgestellt werden, wird lokal durch den jeweiligen Partner spezifiziert.

4.4.1 Wiederverwendung existierender Deployment-Modelle

Eine Anforderung bei der lokalen partnerspezifischen Modellierung des Deployments ist die Wiederverwendung existierender Deployment-Modelle aus, zum Beispiel, der Entwicklungsphase oder vorherigen Deployments (Anforderung 4). Im Schritt 3 der DIVA-Methode wird dafür, falls vorhanden, ein bereits existierendes konkretes Deployment-Modell beschafft, welches als Grundlage für die ressourcenspezifische Verteilung des Modells für die gewünschte Verteilung und die gewählten Cloud-Anbieter dient. Dieses Deployment-Modell ersetzt im *lokalen Teil* des GDMs, das heißt, in dem Teil, der mit dem jeweiligen Partner annotiert ist, die Platzhalter-Komponenten.

In Abbildung 4.3 ist exemplarisch die Substitution der Platzhalter vom Typ „Host Processor“ und „Host DB“ abgebildet. Die Abbildung zeigt eine Erweiterung des Beispiels aus Abbildung 4.2, bei der zusätzlich zur „Java 8 App“-Komponente noch eine „MySQL Datenbank“-Komponente vom zweiten Partner bereitgestellt wird. Zur besseren Lesbarkeit wurden einige Details abstrahiert. So sind nur die Typen der Komponenten und nicht die Komponenten-IDs abgebildet.

Zur Substitution der Platzhalter muss die Übereinstimmung verschiedener Faktoren geprüft werden. Die Überprüfung der anwendungsspezifischen Komponenten erfolgt über ein Graph-Matching-Verfahren. Dafür wird auf dem Verfahren zur Verfeinerung von Deployment-Modellen, eingeführt von Harzenetter et al. [HBF+18], zur Detektion der Übereinstimmung der Komponenten aufgebaut. Mit Hilfe des *Subgraphisomorphismus* wird geprüft, ob der Teilgraph des lokalen Deployment-Modells, der die anwendungsspezifi-

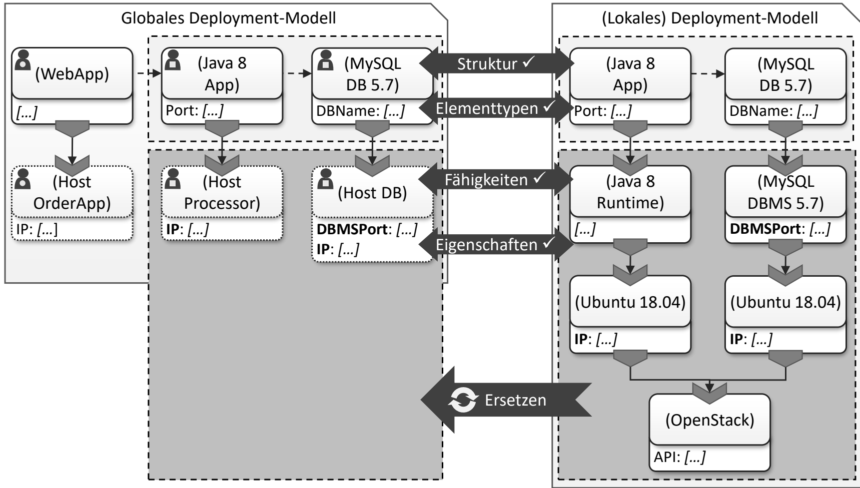

Abbildung 4.3: Beispiel für die Verwendung existierender lokaler Deployment-Modelle als Basis für die Verteilung.

schen Komponenten umfasst, ein Teilgraph des lokalen Teils des GDM ist, wobei speziell neben der Struktur des Graphen (i) die Färbung (Komponententypen) und (ii) die Gewichtung (Relationtypen) übereinstimmen müssen. Damit wird überprüft, ob das Deployment-Modell dieselbe Anwendungsstruktur widerspiegelt.

Im zweiten Schritt muss die Substitutionsfähigkeit der Platzhalter-Komponenten durch die Middleware- und Infrastrukturkomponenten des LDMs überprüft werden. Die Platzhalter-Komponente kann ersetzt werden, wenn (i) die Fähigkeiten der Komponente, die eine direkte HostedOn-Relation zur anwendungsspezifischen Komponente hat, alle Fähigkeiten der Platzhalter-Komponente bereitstellt und (ii) alle Eigenschaften der Platzhalter-Komponente durch Komponenten im Hosting-Stack der anwendungsspezifischen Komponente definiert sind. Im Beispiel in Abbildung 4.3 werden die Eigenschaften des Platzhalters „Host DB“ durch die Eigenschaften der Komponenten „MYSQL DBMS 5.7“ und „Ubuntu 18.04“ abgedeckt. Zur Prüfung der Übereinstimmung der Eigenschaften aller Komponenten, die direkt

oder transitiv mit einer HostedOn-Relation mit der anwendungsspezifischen Komponenente verbunden sind, wird die transitive Hülle berechnet. Sei die transitive Hülle der HostedOn-Relationen $R^+_{d'}$ von einem Deployment-Modell d wie folgt definiert:

Definition 4.16 (Transitive Hülle) Sei $d \in D$ ein Deployment-Modell mit der Menge an Komponenten K_d und der Menge an berücksichtigten Relationen $R_{d'} = \{r_t \in R_d \mid typ_d(r_t) = hostedOn \vee (\exists rt_t \in supertypen_l(typ_l(r_t)) : \pi_1(rt_t) = hostedOn)\}$, dann umfasst die transitive Hülle $R^+_{d'}$ für jeden Weg in d, unter der Berücksichtigung von $R_{d'}$, von einer Komponente $k_a \in K_d$ zu $k_b \in K_d$ eine Relation $r_t \in R^+_{d'}$ mit $\pi_2(r_t) = k_a$ und $\pi_3(r_t) = k_b$.

Sei weiterhin $d \in D$ das GDM und $l \in D$ das LDM mit den äquivalenten anwendungsspezifischen Komponenten $k_i \in K_d$ und $k_z \in K_l$ mit $typ_d(k_i) \sim typ_l(k_x)$. Für die Überprüfung der Substitutionsfähigkeit werden die folgenden Regeln angewendet:

- Fähigkeiten: Sei $k_p \in K_d$ die Platzhalter-Komponente von k_i für die gilt $\exists r_p \in R_d : (\pi_2(r_p) = k_i \wedge \pi_3(r_p) = k_p \wedge \pi_1(typ_d(r_p)) = hostedOn)$ und $k_h \in K_l$ die Platzhalter-Komponente von k_x für die gilt $\exists r_h \in R_l : (\pi_2(r_h) = k_z \wedge \pi_3(r_h) = k_h \wedge (\pi_1(typ_l(r_h)) = hostedOn)$, dann muss gelten: $f\ddot{a}higkeiten_d(k_p) \subseteq f\ddot{a}higkeiten_l(k_h)$.
- Eigenschaften: Sei $R^+_{l'}$ die transitive Hülle von $R_{l'}$, dann muss gelten: $\forall e_p \in eigenschaften_d(k_p) \exists k_y \in K_l \exists r_t \in R^+_{l'} : (\pi_2(r_t) = k_x \wedge \pi_3(r_t) = k_y \wedge e_p \in eigenschaften_l(k_y))$.

Im resultierenden LDM sind die Platzhalter-Komponenten durch existierende Deployment-Modell-Fragmente ersetzt und stellen damit ein ausführbares Modell dar. Die Verteilung der Komponente aus früheren Deployments stimmt jedoch nicht unbedingt mit den Anforderungen des konkreten Anwendungsfalls überein, weshalb das LDM restrukturiert werden kann, um die gewünschte ressourcenspezifische Verteilung zu erzielen.

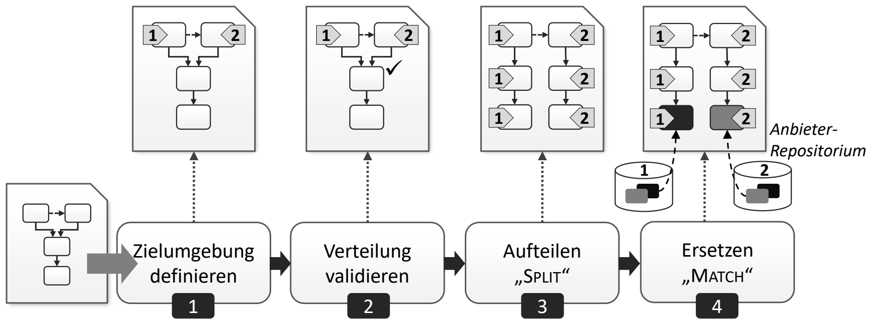

Abbildung 4.4: Übersicht der Split-and-Match-Methode eines Deployment-Modells (adaptiert von Saatkamp et al. [SBKL17]).

4.4.2 Split-and-Match-Methode

Deployment-Modelle müssen entsprechend strategischer Entscheidungen im Unternehmen, zum Beispiel zum Out- oder Insourcing, spezifiziert werden. Bei jeder Bereitstellung einer Anwendung müssen jedoch andere Anforderungen berücksichtigt und das Deployment-Modell entsprechend angepasst werden. Wird eine Anwendung zum Beispiel von unterschiedlichen Unternehmen verwendet, kann jedes über eine andere Infrastruktur verfügen, die bei der Bereitstellung und Verteilung der Komponenten berücksichtigt werden muss. Auch die Infrastruktur in der Test- und Produktivumgebung kann variieren. Die Anpassung von Deployment-Modellen führt typischerweise dazu, dass eine Vielzahl an Technologien ersetzt werden muss. Um diese Probleme anzugehen, wird die Split-and-Match-Methode eingeführt, um ein Deployment-Modell automatisch entsprechend der gewünschten Verteilung über mehrere Cloud-Anbieter und -Dienste anzupassen.

In Abbildung 4.4 sind die Adaptionsschritte der Split-and-Match-Methode abgebildet. Die Modellierung des Modells vor und die Bereitstellung nach der Adaption sind nicht abgebildet. Das Deployment-Modell kann zum Beispiel ein wiederverwendetes Modell aus der Entwicklungsphase sein (siehe Abschnitt 4.4.1). Die Ausführung des Deployments wird in Kapitel 6 vorgestellt. Die Split-and-Match-Methode im Kontext der DivA-Methode wird

zur Verteilung des LDMs angewendet und damit nur im lokalen Kontext eines Partners. Die Komponenten im GDM, die von anderen Partnern bereitgestellt werden, werden aus diesem Grund dabei nicht weiter berücksichtigt. Die Methode findet daher auch in der Single-Partner-Variante Anwendung und ist nicht auf den Einsatz in partnerübergreifenden Anwendungen beschränkt.

Im ersten Schritt werden mindestens alle anwendungsspezifischen Komponenten mit Zielumgebungen annotiert. Jede Zielumgebung spezifiziert die gewünschte Umgebung, in der die jeweilige Komponente bereitgestellt werden soll. Je nach Verteilungsziel kann die Zielumgebung auf verschiedenen Granularitätsebenen gewählt werden: Die Methode ist nicht auf IaaS beschränkt, sondern es können zum Beispiel auch PaaS-Angebote gezielt adressiert werden. Die Verteilung und Auswahl der Anbieter kann manuell oder automatisch auf Basis von Ansätzen zur optimalen Verteilung, wie sie in Abschnitt 2.1.3.1 vorgestellt wurden, festgelegt werden. Die Verteilungsentscheidung selbst ist nicht Teil dieser Methode.

Im zweiten Schritt muss die Verteilungsentscheidung validiert werden, um zu prüfen, ob eine Verteilung entsprechend der Annotationen mit Zielumgebungen grundsätzlich möglich ist. Dabei wird ausschließlich geprüft, ob die Annotationen widersprüchlich sind. Ob ein passender Dienst zur Bereitstellung in der Zielumgebung vorhanden ist, wird in diesem Schritt nicht geprüft. Für eine valide Annotation müssen (i) alle anwendungsspezifischen Komponenten mit einer Zielumgebung annotiert sein und (ii) alle direkten oder transitiven Nachfolger einer Komponente, die durch eine Relation vom Typ „HostedOn“ oder einem davon erbenden Relationtyp verbunden sind, entweder mit der gleichen Zielumgebung annotiert sein oder keine Zielumgebung zugewiesen bekommen. Die Gründe für diese Anforderungen sind wie folgt: Ist eine anwendungsspezifische Komponente nicht annotiert, kann sie später nicht zugeordnet werden. Außerdem kann eine Komponente nicht auf eine Komponente bereitgestellt werden, die in einer anderen Umgebung eingesetzt wird.

Falls die vorgegebene Verteilung gültig ist, kann das Modell im dritten Schritt entsprechend den Annotationen aufgeteilt werden. Dabei werden vorerst die bereits verwendeten Middleware- und Infrastrukturkomponenten

dupliziert, wenn zwei mit unterschiedlichen Zielumgebungen annotierte Komponenten auf derselben Middleware- bzw. Infrastruktur bereitgestellt wurden. Dies ist schematisch in Abbildung 4.4 bei Schritt 3 dargestellt. Zusätzlich werden die Annotationen bis zu den darunter liegenden Komponenten propagiert, so dass diese Verteilungsinformation an jeder Komponente zur Verfügung steht. Im letzten Schritt werden die existierenden Middleware- und Infrastrukturkomponenten durch verfügbare Dienste in der Zielumgebung ersetzt. *Anbieter-Repositorien* speichern Informationen über die Typen von Komponenten, die ein Anbieter bereitstellt. Die unterstützten Dienste sind als Komponentenvorlagen gespeichert. Die Repositorien werden durch die ID der Zielumgebung identifiziert, so dass ein Durchsuchen der Komponentenvorlagen möglich ist. Falls kein passender Dienst gefunden wird, kann eine andere Zielumgebung ausgewählt werden. Für die einzelnen Schritte der Methode werden im Folgenden Algorithmen vorgestellt, die zur Validierung, zum Aufteilen („SPLIT“) und zum Ersetzen („MATCH“) verwendet werden.

4.4.2.1 Validierung der Aufteilung

In diesem Abschnitt wird ein Algorithmus zur Validierung der Annotationen mit Zielumgebungen vorgestellt. Es wird geprüft, ob eine Aufteilung der Komponenten basierend auf den Annotationen möglich ist. Zusätzlich zu dem erweiterten EDMM in Abschnitt 4.2 wird für den Algorithmus noch der direkte *Vorgänger* und *Nachfolger* einer Komponente definiert.

Definition 4.17 (Vorgängerkomponente) Sei $d \in D$ ein Deployment-Modell mit Komponenten und Relationen, dann ist die Vorgängerkomponente $k_v \in K_d$ einer Komponente $k_i \in K_d$ der Quellknoten einer Relation, deren Zielknoten die betrachtete Komponente k_i ist. Dies ist definiert durch die Abbildung: $vorgänger_d : K_d \rightarrow \wp(K_d)$ mit $vorgänger_d(k_i) = \{k_v \in K_d \mid \exists r_i \in R_d : \pi_2(r_i) = k_v \wedge \pi_3(r_i) = k_i\}$.

Für die spezifische Identifikation der Komponente, die von der betrachteten Komponente gehostet wird, wird die Definition weiter eingeschränkt.

Definition 4.18 (Hosting-Vorgängerkomponente) Sei $d \in D$ ein Deployment-Modell mit Komponenten und Relationen, dann kann jede Komponente $k_i \in K_d$ keine oder mehrere Vorgängerkomponenten $k_v \in K_d$ haben, die mit einer Relation vom Typ „HostedOn“ oder einem davon erbenden Relationtyp verbunden sind. Dies ist definiert durch die Abbildung: $hostvorgänger_d : K_d \rightarrow \wp(K_d)$ mit $hostvorgänger_d(k_i) = \{k_v \in K_d \mid (\exists r_i \in R_d : \pi_2(r_i) = k_v \wedge \pi_3(r_i) = k_i \wedge (\pi_1(typ_d(r_i)) = hostedOn \vee (\exists rt_i \in supertypen_d(typ_d(r_i)) : \pi_1(rt_i) = hostedOn)))\}$.

Analog dazu wird der Nachfolger einer Komponente wie folgt definiert:

Definition 4.19 (Nachfolgerkomponente) Sei $d \in D$ ein Deployment-Modell mit Komponenten und Relationen, dann ist die Nachfolgerkomponente $k_n \in K_d$ einer Komponente $k_i \in K_d$ der Zielknoten einer Relation, deren Quellknoten die betrachtete Komponente k_i ist. Dies ist definiert durch die Abbildung: $nachfolger_d : K_d \rightarrow \wp(K_d)$ mit $nachfolger_d(k_i) = \{k_n \in K_d \mid \exists r_i \in R_d : \pi_2(r_i) = k_i \wedge \pi_3(r_i) = k_n\}$.

Im Gegensatz zu den Hosting-Vorgängerkomponenten kann eine Komponente nur genau eine oder keine Hosting-Nachfolgerkomponente haben.

Definition 4.20 (Hosting-Nachfolgerkomponente) Sei $d \in D$ ein Deployment-Modell mit Komponenten und Relationen, dann hat jede Komponente $k_i \in K_d$ höchstens eine Nachfolgerkomponente $k_n \in K_d$, die mit einer Relation vom Typ „HostedOn“ oder einem davon erbenden Relationtyp verbunden ist. Dies ist definiert durch die Abbildung: $hostnachfolger_d : K_d \rightarrow K_d \cup \{\perp\}$ mit $hostnachfolger_d(k_i) = \{k_n \in K_d \mid \exists r_i \in R_d : \pi_2(r_i) = k_i \wedge \pi_3(r_i) = k_n \wedge (\pi_1(typ_d(r_i)) = hostedOn \vee (\exists rt_i \in supertypen_d(typ_d(r_i)) : \pi_1(rt_i) = hostedOn))\}$.

Für die Validierung der Annotationen mit Zielumgebungen müssen zwei Bedingungen erfüllt sein, die im Algorithmus 4.2 geprüft werden:

(1) Alle anwendungsspezifischen Komponenten sind einer Zielumgebung zugeordnet (Zeile 6-8) und

(2) es gibt keine widersprüchlichen Annotationen von Komponenten die direkt oder transitiv mit einer Relation vom Typ „HostedOn“ oder einem davon erbenden Typen mit einer anwendungsspezifischen Komponente verbunden sind (Zeile 10-14).

Für Bedingung (2) wird die transitive Hülle, über die auch die transitiven Abhängigkeiten der Komponenten geprüft werden können, für die HostedOn-Relationen, wie in Definition 4.16 erläutert, berechnet (Zeile 2). Sind beide Bedingungen erfüllt, ist die Aufteilung *valide* im Sinne der Widerspruchsfreiheit der Annotationen (Zeile 16).

Algorithmus 4.2 ValidiereSplit(d)

1: *// Berechne transitive Hülle*
2: $R^+_{d'}$:= **BerechneTransitiveHülle**(d)
3: *// Prüfe alle anwendungsspezifischen Komponenten*
4: **for all** $(k_i \in K_d : hostvorgänger_d(k_i) = \emptyset)$ **do**
5:     *// Überprüfe Bedingung (1)*
6:     **if** $(ziel_d(k_i) = \bot)$ **then**
7:         **return** false
8:     **end if**
9:     *// Überprüfe Bedingung (2)*
10:     **if** $(\exists r_t \in R^+_{d'} : \pi_2(r_t) = k_i \wedge \pi_3(r_t) = k_j \wedge$
11:     $ziel_d(k_j) \neq \bot \wedge ziel_d(k_i) \neq ziel_d(k_j))$ **then**
12:         **return** false
13:     **end if**
14: **end for**
15: **return** true

4.4.2.2 Split-Phase

Die Aufteilung („Split“) der Komponenten entsprechend den annotierten Zielumgebungen hat zum Ziel, die Aufteilung auch bei den Middleware- und Infrastrukturkomponenten widerzuspiegeln. Da Informationen über bereits verwendete Technologien erhalten bleiben sollen, werden die existierenden Middleware- und Infrastrukturkomponenten nicht entfernt, sondern für jede Komponente, die später in einer anderen Zielumgebung bereitgestellt werden soll, dupliziert. Das Ergebnis ist ein funktionsfähiges Deployment-Modell, welches jedoch die Middleware- und Infrastrukturkomponenten der vorherigen Verteilung enthält. Erst im nächsten Schritt werden diese ersetzt.

Algorithmus 4.3 berechnet die Aufteilung und gibt ein funktionsfähiges Deployment-Modell zurück, welches die gewünschte Aufteilung strukturell widerspiegelt. Dafür wird zuerst eine Kopie des Modells erstellt, um Komponenten zur iterativen Verarbeitung schrittweise zu entfernen (Zeile 1). Um sicherzustellen, dass jede Annotation korrekt von oben nach unten im Deployment-Modell propagiert wird, werden immer, ausgehend von den Komponenten, deren Hosting-Vorgänger keine weiteren Hosting-Vorgänger haben, die Annotationen überprüft (Zeilen 4-39). Damit werden im ersten Schritt die Hosting-Nachfolger der anwendungsspezifischen Komponenten betrachtet. Iterativ werden die Komponenten, die keine weiteren Vorgänger haben, nach jedem Durchlauf entfernt (Zeile 36-38). Nimmt man das Deployment-Modell aus Abbildung 4.3 als Beispiel, würden zuerst die Komponente vom Typ „Java 8 Runtime“ und vom Typ „MySQL DBMS 5.7“ betrachtet werden. Für jede Komponente werden die Zielumgebungen der Hosting-Vorgänger geprüft. Wenn alle Vorgänger mit der gleichen Zielumgebung annotiert sind, wird diese Zielumgebung der Komponente zugewiesen (Zeilen 8-9). Andernfalls werden die Komponente und ihre Beziehungen für jede zugewiesene Zielumgebung dupliziert (Zeilen 10-35). Zuerst wird die Komponente (Zeilen 13-15) und dann werden die Relationen dupliziert, um alle Hosting-Vorgänger mit dem Duplikat mit derselben Zielumgebung zu verbinden (Zeilen 17-24 mit $b_1(r)$ und $b_2(r)$). Das Gleiche gilt für die übrigen eingehenden Relationen (Zeilen 17-24 mit $b_1(r)$ und $b_3(r)$), die

Algorithmus 4.3 TeileDeploymentModell(d)

```
1:  d' := d
2:  // Prüfe Komponenten, deren Hosting-Vorgänger keine Hosting-Vorgänger haben
3:  while (∃k_i ∈ K_d' : hostvorgänger_d'(k_i) ≠ ∅ ∧
4:      (∀k_v ∈ hostvorgänger_d'(k_i) : hostvorgänger_d'(k_v) = ∅)) do
5:      // Wenn alle Vorgänger dieselbe Zielumgebung annotiert haben, füge diese
6:      // zur betrachteten Komponente hinzu
7:      if (∃z_i ∈ Z_d' ∀k_v ∈ hostvorgänger_d'(k_i) : ziel_d'(k_v) = z_i) then
8:          ziel_d(k_i) := z_i; ziel_d'(k_i) := z_i
9:      else
10:         // Sonst dupliziere die Komponente für jede Zielumgebung
11:         for all (z_i ∈ Z_d' : ∃k_j ∈ hostvorgänger_d'(k_i) : ziel_d'(k_j) = z_i) do
12:             k_neu := k_i
13:             K_d := K_d ∪ {k_neu}; K_d' := K_d' ∪ {k_neu}
14:             typ_d(k_neu) := typ_d(k_i); typ_d'(k_neu) := typ_d'(k_i)
15:             ziel_d(k_neu) := z_i; ziel_d'(k_neu) := z_i
16:             // Dupliziere die Relationen und verknüpfe sie
17:             b_1(r) := (π_3(r) = k_i)
18:             b_2(r) := (π_2(r) = k_v ∧ k_v ∈ hostvorgänger_d'(k_i) ∧ ziel_d'(k_v) = z_i)
19:             b_3(r) := (π_2(r) = k_j ∧ k_j ∉ hostvorgänger_d'(k_i))
20:             for all (r_i ∈ R_d' : (b_1(r_i) ∧ b_2(r_i)) ∨ (b_1(r_i) ∧ b_3(r_i))) do
21:                 r_neu := r_i
22:                 R_d := R_d ∪ {r_neu}; R_d' := R_d' ∪ {r_neu}
23:                 π_2(r_neu) := π_2(r_i); π_3(r_neu) := k_neu
24:                 typ_d(r_neu) := typ_d(r_i); typ_d'(r_neu) := typ_d(r_i)
25:             end for
26:             for all (r_j ∈ R_d' : π_2(r_j) = k_i) do
27:                 R_d := R_d ∪ {r_neu}; R_d' := R_d' ∪ {r_neu}
28:                 π_2(r_neu) := k_neu; π_3(r_neu) := π_3(r_j)
29:                 typ_d(r_neu) := typ_d(r_j); typ_d'(r_neu) := typ_d(r_j)
30:             end for
31:         end for
32:         // Entferne original Komponente und deren Relationen
33:         K_d := K_d \ {k_i}; K_d' := K_d' \ {k_i}
34:         R_d := R_d \ {r_i | π_2(r_i) = k_i ∨ b_1(r_i)}
35:         R_d' := R_d' \ {r_i | π_2(r_i) = k_i ∨ b_1(r_i)}
36:     end if
37:     // Entferne Hosting-Vorgänger und deren Relationen in d'
38:     K_d' := K_d' \ {k_j | k_j ∈ hostvorgänger_d'(k_i)}
39:     R_d' := R_d' \ {r_j | ∀c_j ∈ hostvorgänger_d'(k_i) : (π_2(r_j) = k_j ∨ π_3(r_j) = k_j)}
40: end while
41: return d
```

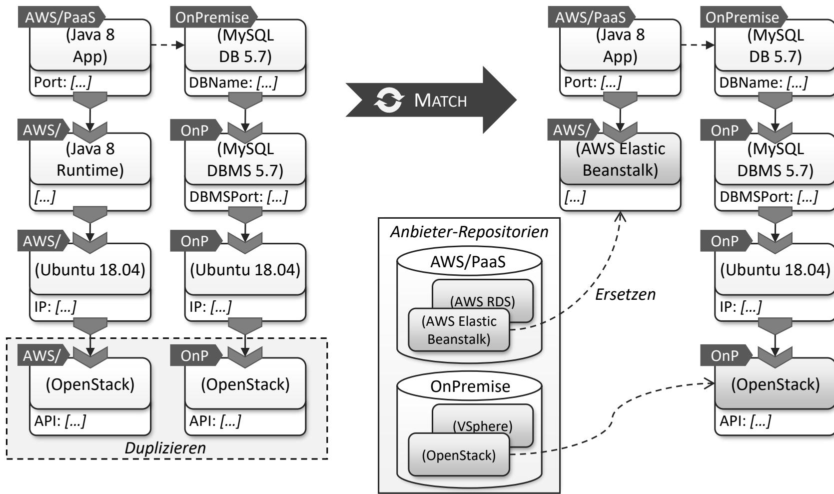

Abbildung 4.5: Beispiel für das Ergebnis der SPLIT-Phase (links) und MATCH-Phase (rechts) mit zwei Zielumgebungen.

nicht vom Typ „HostedOn“ oder einem davon erbenden Typ sind, und alle ausgehenden Relationen (Zeilen 25-28) der ursprünglichen Komponente. Schließlich werden die Originalkomponente und ihre Relationen aus dem Deployment-Modell entfernt (Zeilen 32-34). Zusätzlich werden die Hosting-Vorgänger der betrachteten Komponente und deren Relationen aus der Arbeitskopie d' entfernt (Zeilen 37-38). Die gesamte Iteration (Zeilen 4-39) wird so lange wiederholt, bis keine Komponenten mit Hosting-Vorgängern mehr in d' vorhanden sind.

Das Ergebnis der SPLIT-Phase ist beispielhaft in Abbildung 4.5 im linken Deployment-Modell abgebildet, welches auf dem Beispiel aus Abbildung 4.3 basiert. Für die Aufteilung wurden zwei verschiedene Zielumgebungen gewählt, „AWS/PaaS“ und „OnPremise“, und das Modell entsprechend aufgeteilt, das heißt gemeinsame Komponenten dupliziert. Das resultierende Deployment-Modell ist funktionsfähig und kann zur Bereitstellung verwendet werden. Es spiegelt jedoch noch nicht die durch die Annotation mit Zielumgebungen gewünschte Verteilung auf konkrete Cloud-Dienste wider.

4.4.2.3 MATCH-Phase

Im letzten Schritt der SPLIT-AND-MATCH-Methode werden geeignete Dienste für die Bereitstellung der Komponenten in den gewünschten Zielumgebungen selektiert. Dafür stehen Anbieter-Repositorien zur Verfügung, die Komponentenvorlagen für die in der Zielumgebung verfügbaren Dienste enthalten. Die Spezifikation der Zielumgebungen ist dabei nicht durch die Methode vorgegeben. Die Zielumgebungen können hierarchisch, ähnlich einer Datei-Verzeichnisstruktur, modelliert werden: Durch die Zielumgebung „AWS“ würden alle Repositorien nach geeigneten Diensten durchsucht werden, die mit „AWS“ beginnen. Durch den Schrägstrich „/“ können weitere untergeordnete Repositorien adressiert werden, zum Beispiel nur die PaaS-Angebote von AWS durch „AWS/PaaS“. Die konkrete Struktur sowie die IDs können durch die jeweilige Administratorin oder den jeweiligen Administrator festgelegt und den Anwenderinnen und Anwendern zur Verfügung gestellt werden.

Definition 4.21 (Anbieter-Repositorien) Sei $\mathcal{Z}$ die Menge aller global verfügbaren Zielumgebungen. Dann gilt für ein Deployment-Modell $d \in D$, dass $Z_d \subseteq \mathcal{Z}$. Die Menge $\mathcal{AR} \subseteq \mathcal{Z} \times \wp(D)$ spezifiziert die Menge an global verfügbaren Anbieter-Repositorien. Für jedes $z_i \in \mathcal{Z}$ gibt es damit ein Anbieter-Repositorium $ar_i \in \mathcal{AR}$ mit $ar_i = (z_i, D_i), D_i \subseteq D$. Die Komponentenvorlagen sind als (unvollständige) Deployment-Modelle spezifiziert.

Algorithmus 4.4 berechnet das Matching mit den verfügbaren Diensten. Dafür werden das Deployment-Modell d, die Menge der Anbieter-Repositorien $\mathcal{AR}$ sowie die Menge M übergeben. Die Menge M umfasst alle erfolgreich eingefügten Komponenten aus den Repositorien in d. Dabei wird von unten im Deployment-Modell begonnen und iterativ versucht, die Blattknoten durch passende Dienste zu ersetzen (Zeile 3-30). Für das iterative Vorgehen wird eine Arbeitskopie d' des Deployment-Modells erzeugt (Zeile 1). Ziel ist Middleware- und Infrastrukturkomponenten, die bereits erprobt wurden, wiederzuverwenden, wenn dies möglich ist. Für jeden Hosting-Vorgänger eines Blattknoten wird versucht, eine passende Komponente, welche die

Algorithmus 4.4 ErsetzeKomponenten($d, \mathcal{AR}, M$)

1: $d' := d$
2: *// Überprüfe Ersetzungsoption für Komponenten ohne Hosting-Nachfolger in d'*
3: **while** $(\exists k_i \in K_{d'} : k_i \notin M \wedge hostnachfolger_{d'}(k_i) = \varnothing \wedge$
4: $\quad hostvorgänger_{d'}(k_i) \neq \varnothing)$ **do**
5: $\quad$ *// Finde für jeden Vorgänger eine Hosting-Komponente*
6: $\quad$ **for all** $(k_v \in vorgänger_{d'}(k_i))$ **do**
7: $\quad\quad k_{neu} := \perp$
8: $\quad\quad$ **if** $(\exists k_m \in M : ziel_{d'}(k_m) = ziel_{d'}(k_v) \wedge (\exists af_v \in anforderungen_{d'}(k_v)$
9: $\quad\quad\quad \exists f_m \in fähigkeiten_{d'}(k_m) : \pi_2(af_v) = f_m))$ **then**
10: $\quad\quad\quad k_{neu} := k_m$
11: $\quad\quad$ **else if** $(\exists k_j \in d_j \in \pi_2(ar_i) : \pi_1(ar_i) = ziel_{d'}(k_v) \wedge$
12: $\quad\quad\quad (\exists af_v \in anforderungen_{d'}(k_v) \exists f_j \in fähigkeiten_{d'}(k_j) :$
13: $\quad\quad\quad \pi_2(af_v) = f_j))$ **then**
14: $\quad\quad\quad k_{neu} := k_j$
15: $\quad\quad\quad K_d := K_d \cup \{k_{neu}\}; K_{d'} := K_{d'} \cup \{k_{neu}\}$
16: $\quad\quad\quad ziel_{d'}(k_{neu}) := ziel_{d'}(k_v)$
17: $\quad\quad\quad M := M \cup \{k_{neu}\}$
18: $\quad\quad$ **end if**
19: $\quad\quad$ **if** $(k_{neu} \neq \perp)$ **then**
20: $\quad\quad\quad$ *// Verknüpfe Vorgänger mit neuer Hosting-Komponente und allen*
21: $\quad\quad\quad$ *// Komponenten mit einer Relation zur ersetzten Komponente*
22: $\quad\quad\quad$ **VerknüpfeKomponente(**k_i, k_v, k_{neu}, d, d'**)**
23: $\quad\quad\quad$ *// Entferne überflüssige Hosting-Komponenten des Vorgängers in d*
24: $\quad\quad\quad$ **LöscheHostingKomponenten(**k_i, k_v, d**)**
25: $\quad\quad$ **end if**
26: $\quad$ **end for**
27: $\quad$ *// Entferne betrachtete Komponente und dessen Relationen in d'*
28: $\quad K_{d'} := K_{d'} \setminus \{k_i\}$
29: $\quad R_{d'} := R_{d'} \setminus \{r_j \mid \pi_2(r_j) = k_i \vee \pi_3(r_j) = k_i\}$
30: **end while**
31: *// Menge der Komponenten ohne passende Hosting-Komponente*
32: $K_f := \{k_z \in K_d \mid hostnachfolger_{d'}(k_z) = \perp \wedge k_z \notin M\}$
33: **return** d, K_f, M

Anforderungen erfüllt, zu finden (Zeile 6-26). Dazu wird zuerst die Menge M der bereits eingefügten Komponenten in d geprüft (Zeile 8-10) und falls keine passende Komponente gefunden wurde, das gesamte Anbieter-Repositorium (Zeile 11-18). Wird im Repositorium eine passende Komponente gefunden, wird diese in das Deployment-Modell eingefügt und in die Menge M aufgenommen.

Algorithmus 4.5 VerknüpfeKomponente$(k_i, k_v, k_{neu}, d, d')$

1: *// Verknüpfe Vorgänger-Komponente*
2: $r_h := (Id, k_v, k_{neu})$
3: $R_d := R_d \cup r_h; R_{d'} := R_{d'} \cup r_h$
4: $typ_d(r_h) := rt_h, rt_h \in RT_d : \pi_1(rt_h) = hostedOn; typ_{d'}(r_h) := typ_d$
5: *// Dupliziere eingehende Relationen der ersetzten Komponente*
6: **for all** $(r_e \in R_{d'} \mid \pi_3(r_e) = k_i \wedge \pi_2(r_e) = k_j \wedge k_j \notin hostvorgänger_{d'}(k_i))$ **do**
7: $\quad r_{neu} := r_e; \pi_3(r_{neu}) = k_{neu}$
8: $\quad R_d := R_d \cup \{r_{neu}\}; R_{d'} := R_{d'} \cup \{r_{neu}\}$
9: **end for**
10: *// Dupliziere ausgehende Relationen der ersetzten Komponente*
11: **for all** $(r_a \in R_{d'} \mid \pi_2(r_a) = k_i)$ **do**
12: $\quad r_{neu} := r_a; \pi_2(r_{neu}) = k_{neu}$
13: $\quad R_d := R_d \cup \{r_{neu}\}; R_{d'} := R_{d'} \cup \{r_{neu}\}$
14: **end for**

Wurde in M oder im Repositorium eine Komponente gefunden, wird der Vorgänger mit der Hosting-Komponente verbunden und alle weiteren eingehenden und ausgehenden Relationen der zu ersetzenden Komponente werden dupliziert (Zeile 19-22). Dies führt Algorithmus 4.5 durch, wofür die zu ersetzende Komponente k_i, die Vorgänger-Komponente k_v, die neue Hosting-Komponente k_{neu} sowie das Deployment-Modell d und die Arbeitskopie d' übergeben werden. Zuerst wird die Vorgänger-Komponente mit der neuen Hosting-Komponente verbunden (Zeile 1-4). Dann werden sowohl alle eingehenden Relationen (Zeile 5-9) als auch ausgehende Relationen (Zeile 10-14) für die neue Hosting-Komponente dupliziert.

Anschließend müssen in Algorithmus 4.4 alle noch vorhandenen direkten oder transitiven Hosting-Komponenten des Vorgängers gelöscht werden, solange sie nicht noch von anderen Komponenten benötigt werden, da sie

noch nicht ersetzt werden konnten (Zeile 24). Damit ist sichergestellt, dass Komponenten erst dann aus d gelöscht werden, wenn sie für alle direkten oder transitiven Vorgänger ersetzt wurden. Dieses rekursive Vorgehen zum Entfernen der Komponenten wird von Algorithmus 4.6 beschrieben. Die zu ersetzende Komponente k_i, der betrachtete Vorgänger k_v, für den eine Hosting-Komponente in der Zielumgebung gefunden wurde, und das Deployment-Modell d, werden übergeben. Basierend darauf werden drei Fälle unterschieden:

- Wenn die zu ersetzende Komponente noch weitere Vorgänger hat, dann wird die Komponente nicht gelöscht, sondern nur die Relation zwischen k_i und k_v (Zeile 1-2).
- Wenn die zu ersetzende Komponente noch einen Hosting-Nachfolger hat, dann wird der Algorithmus rekursiv aufgerufen mit dem Hosting-Nachfolger, um die direkten sowie alle transitiven Hosting-Komponenten zu entfernen (Zeile 3-5).
- Wenn es keine Nachfolger gibt und nur einen Vorgänger, nämlich k_v, dann werden die Komponente k_i sowie alle Relationen zu ihr entfernt.

Algorithmus 4.6 LöscheHostingKomponenten(k_i, k_v, d)

1: **if** ($|hostvorgänger_d(k_i)| \geq 2$) **then**
2: $\quad R_d := R_d \setminus \{r_i | \pi_3(r_i) = k_i \wedge \pi_2(r_i) = k_v\}$
3: **else if** ($hostnachfolger_d(k_i) \neq \emptyset$) **then**
4: $\quad$ **LöscheHostingKomponenten(**$hostnachfolger_d(k_i), k_i, d$**)**
5: **end if**
6: **if** ($hostnachfolger_d(k_i) = \emptyset \wedge hostvorgänger_d(k_i) = \{k_v\}$) **then**
7: $\quad K_d := K_d \setminus \{k_i\}$
8: $\quad R_d := R_d \setminus \{r_i | \pi_3(r_i) = k_i \vee \pi_2(r_i) = k_i\}$
9: **end if**

In Algorithmus 4.4 wird, nachdem für alle Vorgänger nach passenden Hosting-Komponenten gesucht wurde, die betrachtete Komponenten k_i aus der Arbeitskopie d' entfernt (Zeile 28-29). Anschließend wird iterativ versucht, die Komponenten, welche zuvor noch die Rolle der Vorgänger hatten, zu ersetzen, falls für sie noch keine passenden Hosting-Komponenten gefunden

wurden. Dieser Prozess wird bis zu den anwendungsspezifischen Komponenten fortgeführt. Werden auch für diese keine passenden Komponenten gefunden, die die Anforderungen erfüllen, dann konnte in der gewählten Zielumgebung keine passende Hosting-Komponente gefunden werden. Diese Komponenten werden dann in der Menge K_f zurückgegeben, da für diese Menge neue Zielumgebungen selektiert werden müssen (Zeile 32).

Das Ergebnis von Algorithmus 4.4 wird beispielhaft in Abbildung 4.5 auf der rechten Seite gezeigt. Die Datenbank wird im eigenen Datencenter auf einer virtuellen Maschine bereitgestellt, während die Java-Anwendung jetzt einen AWS PaaS-Dienst nutzt, „AWS Elastic Beanstalk". Alle Komponenten, die auch in der selektierten Zielumgebung verwendet werden können, wurden wiederverwendet. Für jeden Nutzer oder Anwendungsfall kann damit die Anwendung entsprechend der Nutzeranforderungen verteilt werden.

4.5 Zusammenfassung und Diskussion

Die Nutzung und Erweiterung von EDMM ermöglichen die deploymenttechnologieagnostische Modellierung der Anwendung und Kombination verschiedener Deployment-Technologien zur Deploymentzeit. Die Erweiterungen von EDMM in dieser Arbeit beeinflussen dabei nicht die Transformation in die verschiedenen Deployment-Technologien, sondern sie werden ausschließlich (i) zur Entwurfszeit für die Substitution, Verteilung und Adaption des Modells verwendet, sowie (ii) zur Spezifikation der Organisationseinheiten, welche die globale Teilung des Deployment-Modells definieren. Das Deployment im globalen Kontext muss dabei durch eine zusätzliche Technologie unterstützt werden, dafür werden in dieser Arbeit Workflow-Technologien eingesetzt (Kapitel 6), da die bekannten Deployment-Technologien die Anforderungen für die dezentrale Bereitstellung nicht erfüllen und damit ausschließlich im lokalen Kontext für die Bereitstellung von Fragmenten der Anwendung genutzt werden können.

Die Erweiterungen in Bezug auf die Spezifikation von Anforderungen und Fähigkeiten ist inspiriert vom TOSCA-Standard [OAS13; OAS20]. Mit TOSCA

Light haben Wurster et al. [WBH+20c] gezeigt, dass auch der TOSCA-Standard auf EDMM abbildbar ist. Die formale Definition der einzelnen Elemente basiert auf der Definition von DMMN [Bre16], das eine formale deklarative Modellierungssprache für Deployment-Modelle einführt.

Die Trennung von globalem und lokalem Deployment-Modell in dieser Arbeit ist besonders in Bezug auf die Offenlegung von internen Informationen und Schnittstellen relevant. Die Trennung von anwendungsspezifischen Komponenten und den benötigten Infrastruktur- und Middlewarekomponenten ist inspiriert von Andrikopoulos et al. [AGLW14], die diesen Teil eines Deployment-Modells als *α-Topologie* spezifizieren. Die α-Topologie wird von Andrikopoulos et al. [AGLW14] als Basis für eine optimale Verteilung auf Cloud-Anbieter und -Dienste basierend auf einer Nutzenfunktion genutzt.

Das Konzept zur Wiederverwendung existierender Deployment-Modelle nutzt ein Graph-Matching-Verfahren, das von Harzenetter et al. [HBF+18] eingeführt wurde und für die Überprüfung der Substitutionsfähigkeit der Platzhalter-Komponenten modifiziert wurde. Im Rahmen dieser Arbeit wird das Verfahren angewendet, um die Substitution der Platzhalter mit Fragmenten aus einem anderen Deployment-Modell zu prüfen. Durch diesen Substitutionsschritt können (i) interne Informationen zur Bereitstellung erst im lokalen Kontext zum Deployment-Modell automatisiert hinzugefügt werden und (ii) existierende Deployment-Informationen aus anderen Modellen, zum Beispiel aus der Entwicklungsphase, einfach in einem neuen Kontext wiederverwendet werden.

Die Verteilung der Komponenten ist jedoch meist anwendungsfallspezifisch und muss für jede Bereitstellung der Anwendung angepasst werden. Dies gilt insbesondere auch zwischen der Testumgebung und der Produktivumgebung. Die Split-and-Match-Methode ermöglicht die Partitionierung entsprechend definierter Annotationen von gewählten Zielumgebungen und das Ersetzen existierender Hosting-Komponenten mit Diensten, die in den entsprechenden Zielumgebungen verfügbar sind. Ein Verfahren zur strukturellen Vervollständigung von deklarativen Deployment-Modellen wird auch von Hirmer et al. [HBBL14] eingeführt, das ebenfalls auf dem Matching von Anforderungen und Fähigkeiten basiert. Dabei wird jedoch von

anwendungsspezifischen Komponenten ausgegangen und ausschließlich die Vervollständigung betrachtet. Die templatebasierten Verfahren von Antequera et al. [ACCM17], Arnold et al. [AEK+08] und Pfitzmann und Joukov [PJ11] ermöglichen auch die Vervollständigung bzw. Konkretisierung von abstrakten Komponenten. Das Matching beruht dabei auf vordefinierten Vorlagen, wodurch die Flexibilität eingeschränkt ist. Auch die Spezifikation der Verteilung ist dabei nicht berücksichtigt.

Konzepte zur Verteilung einer Anwendung in einer hybriden Cloud werden von Kaviani, Wohlstadter und Lea [KWL14] und Smit et al. [SSSL12] eingeführt. Sie stellen Verfahren zur Partitionierung der Anwendungs- und Datenschicht vor. Basierend auf dem Abhängigkeitsgraph zwischen den Einheiten können gültige Partitionen berechnet werden. Die Partitionierung der Anwendungs- und Datenschicht ermöglicht die Aufteilung monolithischer Anwendungen und berücksichtigt weniger die spezifischen Anforderungen bei der Bereitstellung der Partitionen. Ein ähnliches Verfahren basierend auf der Analyse des Abhängigkeitsgraphen wird auch in der Split-Phase verwendet. Das Verfahren ist jedoch nicht auf lediglich zwei Partitionen beschränkt. Jedoch ist das Vorgehen auf einen vollständig mit präferierten Zielumgebungen annotierten Graphen beschränkt. Indifferenzen können nicht ausgedrückt werden. Die in Abschnitt 2.1.3.1 vorgestellten Verfahren zur Selektion von Cloud-Anbietern und -Diensten können die hier vorgestellte Split-and-Match-Methode ergänzen und zur Bestimmung der Zielumgebungen zur Annotation des Deployment-Modells genutzt werden. Zimmermann et al. [ZBK+20] nutzen die Split-and-Match-Methode zum Beispiel für die datenflussabhängige Platzierung von Anwendungskomponenten in Industrie-4.0-Anwendungsszenarien.

Bisher ist die Methode außerdem auf die Entwurfszeit beschränkt und unterstützt keine Änderungen der Präferenzen zur Laufzeit basierend auf der Zustandsüberwachung der Anwendung. Die Berechnung der Verteilung und das Ersetzen der Hosting-Komponenten kann jedoch mit dynamischen Migrationsansätzen zur Laufzeit, wie beispielsweise von Carrasco, Durán und Pimentel [CDP18], kombiniert werden.

Musterbasierte Problemerkennung und Modelladaption

Das im vorherigen Kapitel vorgestellte Konzept zur Verteilung von Anwendungen kann zu neuen vorher nicht existierenden Problem führen, verursacht durch Kommunikationsrestriktionen, Sicherheitslücken oder Inkompatibilitäten. Dieses Kapitel führt dafür ein musterbasiertes Verfahren ein, welches automatisiert (i) Probleme identifiziert (Abschnitt 5.3) und (ii) Deployment-Modelle entsprechend verfügbarer Lösungen adaptiert (Abschnitt 5.4 und Abschnitt 5.5). Das Grundkonzept wird in Abschnitt 5.1 vorgestellt und die Anwendbarkeit von Mustern auf Deployment-Modelle in Abschnitt 5.2 diskutiert. Das Verfahren wird zur Realisierung des zweiten Schritts der DivA-Methode (Kapitel 3) eingesetzt und stellt Forschungsbeitrag 3 der vorliegenden Arbeit dar. Die in diesem Kapitel präsentierten Resultate basieren auf bereits im Rahmen dieser Arbeit veröffentlichten wissenschaftlichen Publikationen [SBF+19; SBKL18; SBKL19a; SBLW17].

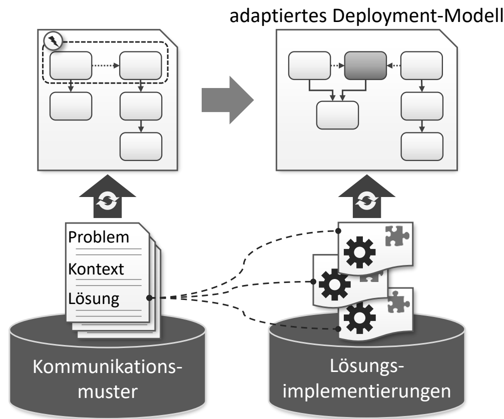

Abbildung 5.1: Grundkonzept der musterbasierten Problemerkennung und Modelladaption.

5.1 Idee und Grundkonzept

Die zentrale Idee des musterbasierten Verfahrens ist die automatisierte Erkennung von Problemen in Deployment-Modellen sowie die kontextspezifische Empfehlung von konkreten Lösungen zum Beheben der identifizierten Probleme, welche automatisiert auf das Deployment-Modell angewendet werden können. In Abbildung 5.1 ist das Grundkonzept des Verfahrens skizziert. Das Wissen über häufig wiederkehrende Probleme ist in Mustern dokumentiert, welche die Grundlage für die Problemerkennung bilden. Die Beschreibung des Problems ist dabei nicht auf einen Abschnitt im Musterformat beschränkt. Wie bereits in Abschnitt 2.3.1.2 erläutert, werden dazu zusätzlich das Wissen aus dem Kontext, in dem das Problem auftreten bzw. das Muster angewendet werden kann, sowie Situationen, die die Anwendung forcieren, berücksichtigt. Die typischerweise menschenlesbaren Muster werden formalisiert, um die Problemerkennung zu automatisieren. Die formale Spezifikation muss dabei als Ergänzung zu den existierenden textuellen Musterbeschreibungen gesehen werden. Sie ermöglichen die maschinelle Verarbeitung des Wissens und somit die Werkzeugunterstützung des Verfahrens. In dieser Arbeit wer-

den ausschließlich Muster aus verschiedenen Mustersprachen betrachtet, die die Kommunikation zwischen verteilten Komponenten betreffen.

Mit den Mustern sind zusätzlich Lösungsimplementierungen verknüpft, die die automatisierte Adaption eines Deployment-Modells in einem speziellen Kontext ermöglichen. Die Anwendbarkeit einer Lösung hängt dabei von der Struktur und den Eigenschaften des Deployment-Modells sowie den dort bereits verwendeten Technologien und Diensten ab. Die Adaptionslogik für ein Deployment-Modell ist in dieser Arbeit die Lösungsimplementierung. In anderen Domänen können auch andere Artefakte als Lösungsimplementierungen dienen [FBB+14a].

Die Anwendung des Verfahrens erfolgt sequenziell, wobei durch die Adaption weitere Probleme auftreten können. Ein durch das Verfahren identifiziertes Problem ist jedoch nicht zwangsweise ein Problem, welches behoben werden muss. Es zeigt ausschließlich, dass die Voraussetzungen für die Empfehlung der Anwendung des Musters erfüllt sind. Die individuellen Entwurfsentscheidungen einer Architektin oder eines Architekten können dieser Empfehlung jedoch entgegenstehen. In den folgenden Abschnitten wird das Verfahren zur Problemerkennung (Abschnitt 5.3) sowie zur Modelladaption (Abschnitt 5.4 und Abschnitt 5.5) detailliert erläutert. Zuvor wird die allgemeine Anwendbarkeit von Architektur- und Entwurfsmustern in Deployment-Modellen diskutiert.

5.2 Anwendung von Mustern in Deployment-Modellen

Vorhandenes Architektur- und Designwissen wird in Mustern in verschiedenen Domänen bereitgestellt. Die Muster sind dabei für verschiedene Stakeholder und deren Anliegen dokumentiert und damit für einen bestimmten Standpunkt (*engl. „Viewpoint“*). Selbst innerhalb einer Mustersprache gibt es unterschiedliche Standpunkte, für die Muster erfasst sind. So enthält die Sprache der Sicherheitsmuster beispielsweise Muster wie das SECURE-CHANNEL-Muster für den Entwurf sicherer Internetanwendungen sowie Muster auf strategischer Ebene für die Integration von Sicherheitsdiensten

innerhalb einer Organisation [SFH+06]. Auch die Muster im Bereich Cloud Computing decken sowohl die Charakteristika von Arbeitslastmustern als auch Integrationskonzepte für verteilte Anwendungen ab [FLR+14]. Die unterschiedlichen Standpunkte, die von Mustersprachen abgedeckt werden, eröffnen ein breites Spektrum an Anwendungsmöglichkeiten von Mustern. Aufgrund der Vielzahl an Mustersprachen und Mustern kann eine einzelne Anwenderin oder ein einzelner Anwender (i) nicht alle relevanten Muster in einer Domäne kennen und daher (ii) auch nicht alle möglichen Probleme, die in einem restrukturierten Deployment-Modell einer verteilten Anwendung auftreten können, im Blick haben.

Zur Unterstützung von Architektinnen und Architekten sowie Anwendungsentwicklerinnen und -entwicklern, um relevante Muster aus verschiedenen Mustersprachen für einen bestimmten Standpunkt zu identifizieren, wurde in Weigold et al. [WBB+20a] das Konzept der *Mustersichten* (*engl. „Pattern Views“*) entwickelt. Mustersichten ermöglichen (i) den Kontext, in dem eine Menge von Mustern relevant ist, explizit zu definieren und (ii) neue zusätzliche Relationen zwischen Mustern zu spezifizieren, die in diesem konkreten Kontext relevant sind. Dies erleichtert vor allem die manuelle Anwendung von Mustern, zum Beispiel für die Architektur einer neuen Cloud-Anwendung.

Der Begriff der Architektur- und Entwurfsmuster im Bereich der Informatik macht bereits deutlich, dass die Intention dieser Muster darin besteht, die Entwicklungsphase neuer Softwaresysteme zu unterstützen. Das vom SECURE-CHANNEL-Muster [SFH+06] beschriebene Problem, dass sensible Daten über einen unsicheren Kanal über ein öffentliches Netzwerk gesendet werden oder das Problem, dass die eingehende Kommunikation durch eine Firewall eingeschränkt wird, wie vom APPLICATION-COMPONENT-PROXY-Muster [FLR+14] beschrieben, kann jedoch nicht nur bei der Entwicklung eines Systems, sondern auch bei dessen Umstrukturierung auftreten. Dabei werden Architektur- und Entwurfsmuster auf eine neue Domäne angewendet: Deployment-Modelle. Dies erleichtert sowohl (i) das Erkennen des Anpassungsbedarfs, der durch eine Verteilung der Anwendungskomponenten entstehen kann, als auch (ii) die Lösung von dadurch entdeckten Problemen.

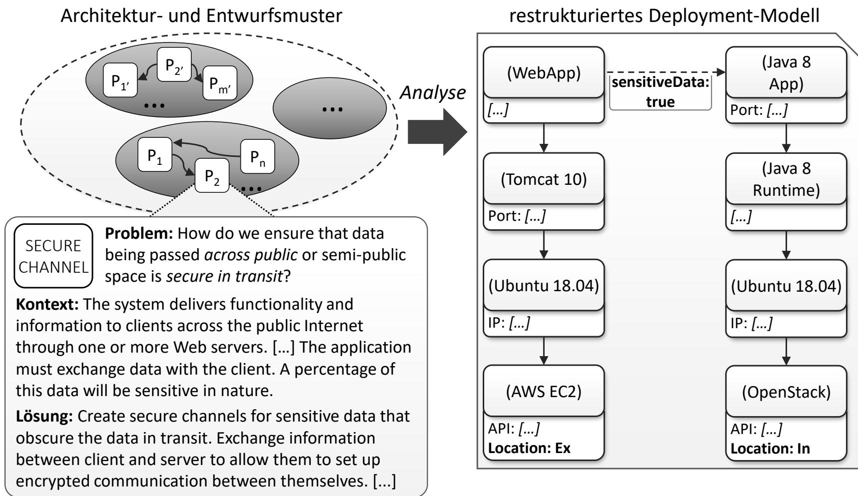

Abbildung 5.2: Beispiel für die musterbasierte Problemerkennung mit Auszug aus der Musterbeschreibung des SECURE-CHANNEL-Musters aus Schumacher et al. [SFH+06].

5.3 Muster zur Problemerkennung in Deployment-Modellen

Das dokumentierte Wissen in Mustern wird zur Erkennung von Problemen in Deployment-Modellen angewendet. Auch wenn Muster, anders als Antimuster, Best Practices beschreiben, wird das von dem Muster adressierte Problem sowie der Kontext, in dem das Muster angewendet werden kann, mit erfasst. Dieses Wissen soll Architektinnen und Architekten sowie Entwicklerinnen und Entwickler bei der Bestimmung der Anwendbarkeit eines bestimmtes Musters unterstützen. Das Beispiel in Abbildung 5.2 rechts zeigt ein restrukturiertes Deployment-Modell für eine Anwendung bestehend aus zwei Komponenten, einer Web-Anwendung und einer Java-App. Als Resultat entweder der Verteilung auf verschiedene Partner oder der Restrukturierung mit Hilfe der SPLIT-AND-MATCH-Methode kann die Verteilung der einzelnen Komponenten auf unterschiedliche Umgebungen definiert werden.

In diesem Beispiel resultiert die Verteilung aus der Restrukturierung, da

hier bereits detailliertes Wissen über die verwendeten Middleware- und Infrastrukturkomponenten vorhanden ist. Anstatt auf einer einzelnen VM sollen die Komponenten in einer Private-Cloud und einer Public-Cloud bereitgestellt werden. Wie bereits in Abschnitt 3.2 erläutert, kann das Deployment-Modell mit zusätzlichen Informationen zu den Komponenten und Relationen annotiert werden, die das Verhalten beschreiben. Die Komponenten vom Typ „AWS EC2“ und „OpenStack“, zum Beispiel, haben zusätzliche Informationen über ihre logische Lokation. Des Weiteren ist die „ConnectsTo“-Relation mit der Information über die Art der Daten, die zwischen den Komponenten ausgetauscht werden, annotiert. Durch die Änderung der Verteilung findet die Kommunikation zwischen den beiden Komponenten nun über ein öffentliches Netzwerk statt. Das bedeutet, dass durch die Restrukturierung die unverschlüsselten Daten offen für Man-in-the-Middle-Angriffe sind. Dieses Problem wird vom SECURE CHANNEL-Muster adressiert. In Abbildung 5.2 ist ein Ausschnitt aus der Musterbeschreibung abgebildet. Daraus geht hervor, wann das Muster angewendet werden sollte, das heißt wann eine unsichere Kommunikation vorliegt. Auf Basis dieser Informationen kann ein Deployment-Modell analysiert werden, um mögliche Probleme zu identifizieren.

Zur Automatisierung des Verfahrens muss das Wissen maschinenlesbar zur Verfügung gestellt werden. In Abschnitt 2.3.1 wurden verschiedene Methoden zur Erkennung von Problem- und Lösungsinstanzen von Mustern diskutiert. Die Konzepte zur Erkennung von Lösungsinstanzen, welche Logik erster Ordnung und im Speziellen Prolog zur Implementierung verwenden, sind dabei ausdrucksstärker als Graph-Matching-Verfahren, da auch die Nicht-Existenz von Elementen ausgedrückt werden kann sowie Bedingungen durch das logische Oder verknüpft werden können. Bisher wird diese Methode ausschließlich zur Erkennung von Lösungsinstanzen und nicht zur Erkennung von Probleminstanzen angewendet, obwohl hier der größte Nutzen der Methode liegt. Vor allem die Definition der Nicht-Existenz bestimmter Elemente ist für die Formalisierung von Problemen wichtig. Im Folgenden wird die Formalisierung der Deployment-Modelle sowie der Muster detailliert erläutert.

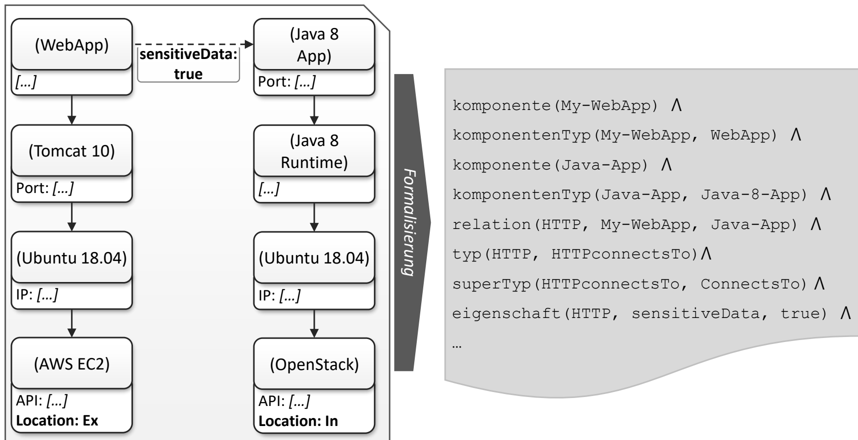

Abbildung 5.3: Formalisierung eines Deployment-Modells.

5.3.1 Formalisierung von Deployment-Modellen

Logik erster Ordnung ermöglicht Aussagen über Objekte, deren Beziehungen und Eigenschaften. So kann vorhandenes Wissen dargestellt und neues Wissen durch logische Implikationen abgeleitet werden. Zur Formalisierung eines Deployment-Modells wird das erweiterte EDMM aus Abschnitt 4.2 für die Syntax und die Semantik der Entitäten genutzt, wobei ausschließlich strukturelle Eigenschaften betrachtet werden. Operationen beispielsweise sind hier nicht von Bedeutung, da sie ausschließlich zur Ausführung der Bereitstellung dienen und keinen Einfluss auf die Struktur und das Verhalten der Anwendung zur Laufzeit haben. Basierend auf diesem Metamodell können Deployment-Modelle durch eine Modell-zu-Modell-Transformation als logische Formeln dargestellt werden. Diese bilden die Wissensbasis aus der neues Wissen abgeleitet werden kann. Dazu werden folgende Prädikatensymbole eingeführt:

- *komponente (komponenten-id)*: Komponente,
- *relation (relation-id, komponenten-id, komponenten-id)*: Relation zwischen einer Quell- und Zielkomponente,

- *typ (element-id, typ-id)*: Typ eines Modellelements,
- *superTyp (typ-id, typ-id)*: Supertyp eines Typs,
- *eigenschaft (element-id, eigenschaft-id, wert)*: Eigenschaft eines Modellelements,

Ein Auszug aus der logischen Formel, die die Fakten über das Beispiel aus Abbildung 5.2 definiert, ist in Abbildung 5.3 dargestellt. Die Modell-zu-Modell-Transformation von EDMM in die logische Formel ermöglicht die regelbasierte Modellanalyse. Die eingeführten Prädikatensymbole sind die Grundlage für die Erkennung von Problemen. Sie repräsentieren die *Muster-Primitive*, die zur formalen Definition der Muster in dieser Domäne verwendet werden. Zusätzlich basiert der gesamte Modellierungsansatz auf definierten Modellelementtypen, die zusätzliche Semantik für die Definition von Modellen sowie Mustern bringen. Haitzer und Zdun [HZ15] nutzen ebenfalls Muster-Primitive zur Annotation von Architekturmodellen zur Erkennung von Musterlösungen.

5.3.2 Formalisierung von Mustern

Die Formalisierung der Muster zum Zweck der Problemerkennung basiert auf dem Vorgehensmodell von Kim und El Khawand [KE07], welches in Abbildung 2.4 dargestellt und in Abschnitt 2.3.1.2 erläutert wurde. Dabei ist die resultierende Formalisierung abhängig (i) von der Domäne und (ii) der Interpretation durch die Autorin oder den Autor. Diese Verzerrung lässt sich nicht vermeiden und nur durch ein dediziertes systematisches Vorgehen unter Einbeziehung verschiedener Modelle als Referenzszenarien verringern. Für die Problemerkennung in Deployment-Modellen werden Muster spezifisch für EDMM formalisiert.

Alle wesentlichen Aussagen der Problem- und Kontextbeschreibung eines Musters müssen auf Basis des in Abbildung 4.1 vorgestellten EDMMs formalisiert werden. Dazu wurden bereits im vorherigen Abschnitt die Basis-Prädikatensymbole zur Beschreibung der strukturellen Merkmale eines Deployment-Modells eingeführt. Diese Fakten sind die Bedingungen für die

logischen Formeln, die die Musterregeln repräsentieren. Für jede in natürlicher Sprache beschriebene Aussage in der Musterbeschreibung müssen entsprechende Bedingungen extrahiert werden, die für diese Regel erfüllt sein müssen. Im Folgenden wird beispielhaft die Musterbeschreibung des SECURE-CHANNEL-Musters analysiert, und für jede Aussage wird die entsprechende Bedingung definiert. In Abbildung 5.2 ist der relevante Ausschnitt der Musterbeschreibung abgebildet.

Die Kontextbeschreibung des SECURE-CHANNEL-Musters besagt, dass sensible Daten ausgetauscht werden. In einem Deployment-Modell kann dies als Eigenschaft ausgedrückt werden. Somit dient folgende Tatsache als Bedingung für die Musterregel mit *r* als Variable für eine Relation: *eigenschaft(r, Sensitivedata, True)*. Daten können nur ausgetauscht werden, wenn eine Verbindung zwischen zwei Komponenten hergestellt wird. Eine Relation vom Typ „ConnectsTo“ muss also in einer Topologie enthalten sein. Hierfür werden zwei Fakten verwendet: zum einen für die Existenz der Relation, $relation(r, k_1, k_2)$, und zum anderen für den Typ der Relation, *typ(r, t)*. Zusätzlich wird das Funktionssymbol *superTypen(t)* eingeführt, welches alle Typen von denen der Typ *t* direkt oder transitiv erbt zurückgibt.

Eine weitere Bedingung ist, dass die beiden Komponenten, für die diese Relation existiert, über ein öffentliches Netz kommunizieren, das heißt, sie sind in unterschiedlichen Umgebungen bereitgestellt, was durch das Prädikat $unterschiedlicheLokationen(k_1, k_2)$ ausgedrückt wird. Diese Tatsache wiederum ist abgeleitet aus der Wissensbasis. Dafür wird zusätzlich das Funktionssymbol *hostingStack(k)* eingeführt, welches alle Komponenten, die mit einer Relation vom Typ „HostedOn“ direkt oder transitiv mit einer Komponente *k* verbunden sind, zurückgibt. Mit dem Prädikat *teilVon(a, b)* kann geprüft werden, ob ein Element *a* Teil einer Menge *b* ist. Zusätzlich wird das Prädikat *ungleich(a,b)* eingeführt, welches angibt, ob zwei Werte ungleich sind. Damit kann folgendes Wissen abgeleitet werden:

$$\forall k_1 \forall k_2\text{: } (\exists h_1 \exists h_2 \exists l_1 \exists l_2 \\ \quad \mathit{teilVon}(h_1, \mathit{hostingStack}(k_1)) \wedge \mathit{teilVon}(h_2, \mathit{hostingStack}(k_2)) \wedge \\ \quad \mathit{eigenschaft}(h_1, \mathit{Location}, l_1) \wedge \mathit{eigenschaft}(h_2, \mathit{Location}, l_2) \wedge \\ \quad \mathit{ungleich}(l_1, l_2)) \Leftrightarrow \mathit{unterschiedlicheLokationen}(k_1, k_2)$$

Die Formel besagt, dass zwei Komponenten k_1 und k_2 in unterschiedlichen Umgebungen sind, wenn es jeweils eine Komponente gibt, die ein direkter oder transitiver Host der jeweiligen Komponente ist, der als Eigenschaft die Lokation der Komponente definiert und dieser Wert sich bei den beiden Hosts unterscheidet. Die Lokation ist dabei bei restrukturierten Deployment-Modellen die annotierte Zielumgebung.

Die letzte Bedingung des Musters ist, dass zwei Komponenten über einen unsicheren Kanal kommunizieren. Dies ist der Fall, wenn ein Sicherheitsmechanismus fehlt. Die Eigenschaft, dass die Kommunikation verschlüsselt stattfindet, kann durch entsprechende Eigenschaften der Relation ausgedrückt werden: *eigenschaft(r, Encrypted, True)*. Das durch das SECURE CHANNEL-Muster beschriebene Problem kann somit wie folgt ausgedrückt werden:

$$\forall k_1 \forall k_2\text{: } (\exists r\ \mathit{relation}(r, k_1, k_2) \wedge \mathit{unterschiedlicheLokationen}(k_1, k_2) \wedge \\ \quad (\mathit{typ}(r, \mathit{ConnectsTo}) \vee \mathit{teilVon}(\mathit{ConnectsTo}, \mathit{superTypen}(\mathit{typ}(r)))) \wedge \\ \quad \mathit{eigenschaft}(r, \mathit{SensitiveData}, \mathit{True}) \wedge \neg \mathit{eigenschaft}(r, \mathit{Encrypted}, \mathit{True})) \\ \quad \Leftrightarrow \mathit{unsichereKommunikation}(k_1, k_2)$$

Die Definition beruht auf der Annahme einer geschlossenen Welt (*engl. „Closed-World-Assumption“*). Das bedeutet, dass alle Informationen über die betrachtete Welt vorliegen. Da die Betrachtung sich ausschließlich auf ein Deployment-Modell einer Anwendung bezieht, kann diese Annahme getroffen werden. Dadurch kann auch die Nicht-Existenz bestimmter Fakten aus der Wissensbasis ausgewertet werden. Nichtsdestotrotz ist die Aussagekraft durch die Modellierung des Deployment-Modells beschränkt. Werden wichtige Informationen im Deployment-Modell nicht abgebildet, steht dieses Wissen nicht zur Auswertung der Muster zur Verfügung.

Um den vorgestellten Ansatz zu automatisieren, müssen die logischen Formeln als Logikprogramm ausgedrückt werden. Für die prototypische

Implementierung und Validierung wird in dieser Arbeit die Logik-Programmiersprache Prolog verwendet. Die prototypische Umsetzung des Werkzeugs wird in Kapitel 7 erläutert. Zur Validierung der Methode werden in Abschnitt 7.3 außerdem weitere Muster eingeführt.

5.4 Anwendung von Musterlösungen zur Modelladaption

Im vorherigen Abschnitt wurde ein Verfahren zur automatisierten Problemerkennung in Deployment-Modellen basierend auf Musterbeschreibungen eingeführt. Diese generischen und technologieunabhängigen Beschreibungen sind als Indikatoren für Probleme ausreichend. Die in Musterbeschreibungen angegebenen konzeptionellen Lösungen und deren textuell beschriebenen beispielhaften Implementierungen hingegen sind nicht ausreichend für eine automatisierte Implementierung einer Lösung oder, im vorliegenden Anwendungsfall, zur Modelladaption. Eine Vielzahl von möglichen konkreten Implementierungen können für ein einzelnes Muster existieren. Welche dieser Lösungsimplementierungen geeignet ist, hängt dabei unter anderem von der Domäne, verwendeten Technologien und Programmiersprachen ab.

Zum automatisierten Beheben identifizierter, zu lösender Probleme in Deployment-Modellen wird im Folgenden ein Verfahren zur (i) Identifikation und (ii) Anwendung geeigneter Lösungen vorgestellt. Das Verfahren ist von Falkenthal et al. [FBB+14a; FBB+14b] inspiriert, deren Konzepte bereits in Abschnitt 2.3.2 kurz vorgestellt wurden: Um die Dokumentation konkreter Lösungen eines Musters in einer wiederverwendbaren Weise zu ermöglichen, werden diese als *Lösungsimplementierungen* beschrieben, die mit einem Muster verknüpft sind. Diese Lösungsimplementierungen sind wiederverwendbare Artefakte, zum Beispiel in Form von ausführbaren Softwareartefakten, Codeschnipseln oder Konfigurationsdateien. Auswahlkriterien bestimmen, wann eine bestimmte Lösungsimplementierung verwendet werden soll. Dieses Konzept erleichtert die Anwendung des Musters auf ähnliche Anwendungsfälle.

Dieses generelle domänenunabhängige Konzept wurde im Rahmen dieser

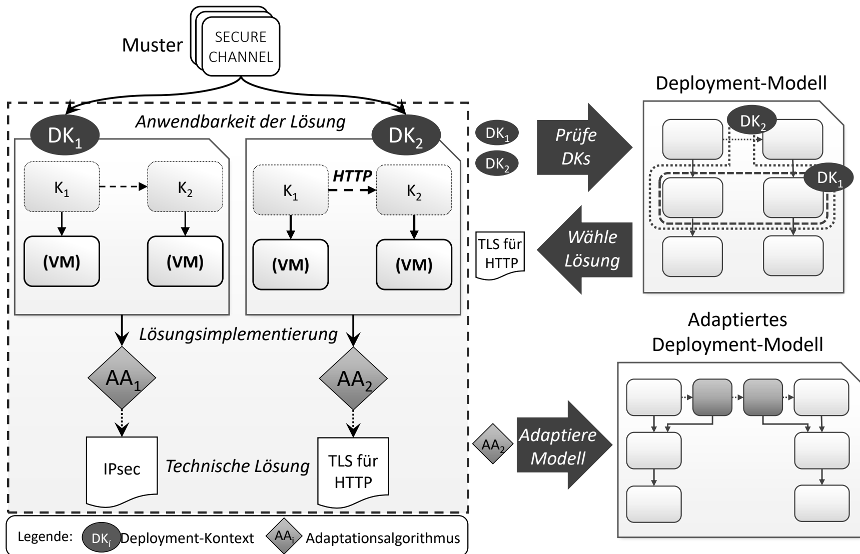

Abbildung 5.4: Erkennen, Selektieren und Anwenden von Lösungen in Deployment-Modellen zum Beheben von Problemen (adaptiert von Saatkamp et al. [SBF+19]).

Arbeit für die Adaption von Deployment-Modellen angewendet und für die speziellen Anforderungen erweitert. In Abbildung 5.4 ist das Grundkonzept für die Erkennung, Selektion und Anwendung von Musterlösungen in Deployment-Modellen abgebildet. Für jede technische Lösung wird der *Deployment-Kontext (DK)* spezifiziert, in dem die entsprechende Lösung anwendbar ist. Der Deployment-Kontext definiert die technischen Voraussetzungen für die Anwendbarkeit einer Lösungsimplementierung. Zur Veranschaulichung sind in Abbildung 5.4 beispielhaft zwei DKs grafisch dargestellt. Die graphische Darstellung dient nur der Anschaulichkeit und Verständlichkeit des Gesamtkonzepts anhand des Beispiels. Für die Erkennung der Anwendbarkeit wird jedoch kein Graph-Matching-Verfahren, sondern ebenfalls Logik erster Ordnung angewendet. Zusätzlich wird mit dem Deployment-Kontext die eigentliche Lösungsimplementierung in Form eines *Adaptionsalgorithmus*

(AA) verknüpft, der die Umsetzung der Lösung in einem Deployment-Modell ermöglicht. Links in Abbildung 5.4 ist beispielhaft wieder das Secure Channel-Muster mit zwei technischen Lösungen, *IPsec* und *TLS für HTTP* dargestellt. Zur Absicherung der Kommunikation auf der Netzwerkschicht kann IPsec verwendet werden. Dazu wird eine VPN-Verbindung zwischen den beiden Hosts aufgebaut. Zur Absicherung der Kommunikation auf der Anwendungsschicht können TLS-Zertifikate verwendet werden, um zum Beispiel eine HTTP-Verbindung abzusichern. Um diese Lösungen für eine konkrete Anwendung einsetzen zu können, müssen verschiedene Bedingungen erfüllt sein: IPsec kann nur verwendet werden, wenn die VMs zur Bereitstellung der Anwendungskomponenten konfigurierbar sind. Falls HTTP als Kommunikationsprotokoll verwendet wird, können, je nach Anwendungsfall, die Server entsprechend konfiguriert oder TLS-Proxys eingefügt werden.

Der TLS-Verschlüsselungsmechanismus ist zwar unabhängig von den Hosting-Komponenten, aber die erforderlichen Anpassungsschritte variieren je nachdem wie die Anwendungskomponenten bereitgestellt werden: Die Anpassungsschritte zum Einfügen eines TLS-Proxys für eine Komponente, die auf einer VM bereitgestellt wird, unterscheiden sich von den Anpassungen, die erforderlich sind, um einen Sidecar-Container an einen Docker-Container anzuhängen, der auf einer Docker-Engine bereitgestellt wird. Die technische Lösung ist in beiden Fällen dieselbe, jedoch muss ein zusätzlicher DK spezifiziert und AA implementiert werden. Der DK ist somit wesentlich, um die Anwendbarkeit einer generellen technischen Lösung, aber auch die Selektion eines spezifischen AAs zu ermöglichen.

Der generelle Ablauf zur Erkennung, Selektion und Anwendung von Lösungen ist rechts in Abbildung 5.4 dargestellt. Zuerst wird geprüft, welche DKs im Deployment-Modell enthalten sind. Dabei können auch mehrere übereinstimmende DKs identifiziert werden. Welche der anwendbaren Lösungen präferiert wird, kann von der Entwicklerin oder dem Entwickler manuell selektiert werden. Anschließend wird im letzten Schritt die gewählte Lösungsimplementierung, das heißt der AA, auf das Deployment-Modell angewendet und das Modell entsprechend adaptiert. Im Folgenden werden die Spezifikation des DKs und die Modelladaption im Detail erläutert.

5.4.1 Formalisierung des Deployment-Kontexts

In Abbildung 5.4 sind zwei DKs für unterschiedliche technische Lösungen graphisch veranschaulicht. Jedoch reichen Graph-Matching-Verfahren nicht zur Prüfung der Übereinstimmung von DKs mit einem gegebenen Deployment-Modell aus. Ebenso wie bei der Problemerkennung ist auch hier die Definition der Nicht-Existenz in bestimmten Fällen notwendig. Jeder DK wird daher als logische Formel ausgedrückt, die die Anwendbarkeit eines AA ausdrückt. Eine solche logische Formel kann auf Basis des formalisierten Deployment-Modells, wie in Abschnitt 5.3.1 vorgestellt, ausgewertet werden. Im Folgenden wird der DK_2 für die „TLS für HTTP"-Lösung, wie in Abbildung 5.4 dargestellt, exemplarisch beschrieben. Bevor die logische Formel für DK_2 vorgestellt wird, muss ein weiteres Prädikatensymbol eingeführt werden: Um auszudrücken, dass eine Komponente *h* von einem bestimmten Typ *t* im hostingStack von einer anderen Komponente *k* enthalten ist, wird das Prädikatensymbol *hostVon(h, k, t)* definiert. Seien *h*, *k*, *t* Variablen, dann kann das Wissen über den Host wie folgt ausgedrückt werden:

$$\forall h \forall k \forall t : ((\mathit{typ}(h, t) \vee (\exists s\ \mathit{teilVon}(t, \mathit{superTypen}(s)) \wedge \mathit{typ}(h,s))) \wedge \mathit{teilVon}(h, \mathit{hostingStack}(k))) \Leftrightarrow \mathit{hostVon}(h, k, t)$$

Die Formel besagt, wenn eine Komponente *h* existiert, die von Typ *t* und Teil des hostingStack der Komponente *k* ist, dann ist *h* ein Host von *k*. Dieses Wissen kann genutzt werden, um die Anwendbarkeit, das heißt den DK für die „TLS für HTTP"-Lösung zu spezifizieren. Diese Lösung wird immer auf zwei Komponenten angewendet, weshalb das Prädikatensymbol $\mathit{tlsOnVM}(k_1, k_2)$ für die Anwendbarkeit der Lösung eingeführt wird, die wie folgt abgeleitet wird:

$$\forall k_1 \forall k_2 : (\exists h_1 \exists h_2 \exists r\ \mathit{hostVon}(h_1, k_1, VM) \wedge \mathit{hostVon}(h_2, k_2, VM) \wedge \mathit{relation}(r, k_1, k_2) \wedge (\mathit{typ}(r, \mathit{HTTPConnectsTo}) \vee \mathit{teilVon}(\mathit{HTTPConnectsTo}, \mathit{superTypen}(\mathit{typ}(r))))) \Leftrightarrow \mathit{tlsOnVM}(k_1, k_2)$$

Dieser Deployment-Kontext stimmt mit einem Deployment-Modell überein, wenn die zwei betrachteten Komponenten k_1 und k_1 auf einer Komponente

vom Typ „VM“ oder einem davon erbenden Typ gehostet werden und sie mit einer Relation vom Typ „HTTPConnectsTo“ oder einem davon erbenden Typ verbunden sind. Dieses Beispiel zeigt, wie die Anwendbarkeit eines AAs spezifiziert werden kann. Weitere Anwendungsbeispiele werden zur Validierung des Konzepts in Abschnitt 7.3 eingeführt.

5.4.2 Modelladaption

Die im vorherigen Schritt erkannten übereinstimmenden DKs bestimmen die Anwendbarkeit der verknüpften AAs, wobei mehrere DKs mit dem Deployment-Modell übereinstimmen können und damit verschiedene Lösungen realisiert werden können. Die bevorzugte Lösung muss manuell ausgewählt werden, da benutzerspezifische Präferenzen berücksichtigt werden müssen. In dem in Abbildung 5.4 dargestellten Beispiel wird die „TLS für HTTP“-Lösung ausgewählt. Basierend auf dieser Auswahl muss das Deployment-Modell entsprechend angepasst werden.

In Abschnitt 2.1.3 wurden verschiedene Konzepte zur Modelladaption diskutiert. In dieser Arbeit werden Adaptionsalgorithmen, ähnlich wie von Breitenbücher [Bre16] und Guth und Leymann [GL19] verwendet, eingesetzt. Die AAs kapseln die Anpassungsschritte, die zur Realisierung einer definierten Lösung ausgeführt werden müssen. AAs können über ein Lösungsimplementierungs-Repositorium verfügbar gemacht werden [FBFL15; LB21]. Ein DK und sein verknüpfter AA sind in Bezug auf die betrachteten Modellentitäten eng gekoppelt. Ein AA kann nur auf den Modellentitäten operieren, die in dem entsprechenden DK definiert sind. Nur dadurch kann die Funktionsfähigkeit des AAs für das Deployment-Modell sichergestellt werden. Nur diese Modellentitäten und deren spezifischen Eigenschaften, die im DK geprüft werden, können als Annahme für den AA dienen.

Für die Realisierung der selektierten Lösung im Beispiel in Abbildung 5.4, das heißt die Verschlüsselung der Kommunikation mit TLS-Zertifikaten, müssen zwei Proxies zwischen die kommunizierenden Komponenten geschaltet werden, um die Anfragen und Antworten zu verschlüsseln. Diese Proxies werden auf der jeweiligen VM jeder Komponente bereitgestellt. Da-

mit ist die „TLS für HTTP“-Lösung realisiert und das Problem der unsicheren Kommunikation im Deployment-Modell durch die in AA_2 implementierten Anpassungen gelöst. Das Beispiel zeigt bereits, dass die Adaption auf die Elemente des Deployment-Modells beschränkt ist. Der Quellcode der einzelnen Komponenten kann nicht angepasst werden.

5.5 Adapter zur Anpassung der Kommunikationsprotokolle

Eine spezielle Herausforderung sind die unterschiedlichen Kommunikationsprotokolle, die eine Hürde bei der Portabilität und Anpassung von Anwendungen für unterschiedliche Umgebungen und die Verteilung der Komponenten darstellen. Vor allem für IoT-Anwendungen mit einer nachrichtenbasierten Kommunikation gib es eine Vielzahl an Integrationssystemen, die unterschiedliche Transportprotokolle wie MQTT, XMPP oder AMQP unterstützen. Deshalb wird in diesem Abschnitt ein Konzept zur flexiblen Selektion von Kommunikationstreibern abhängig von der in der entsprechenden Umgebung verwendeten Integrations-Middleware eingeführt. Dadurch kann die Zahl zusätzlicher Komponenten reduziert werden, ohne den Quellcode der zu integrierenden Komponenten anpassen zu müssen, und ein Technologie-Lock-In in Bezug auf spezifische Integrations-Middleware wird verhindert. Um die genannten Ziele zu erreichen, wird (i) ein programmiersprachenunabhängiges Programmiermodell eingeführt, um die Anwendung unabhängig von der verwendeten Integrations-Middleware zu implementieren und die Selektion von Kommunikationstreibern entsprechend der Middleware zu ermöglichen. Darüber hinaus wird (ii) ein Konzept zur middlewareunabhängigen Modellierung und zur automatisierten Selektion in Abhängigkeit der gewählten Integrations-Middleware vorgestellt. Das Modellierungs- und Selektionskonzept beruht dabei auf Deployment-Modellen entsprechend dem erweiterten EDMM.

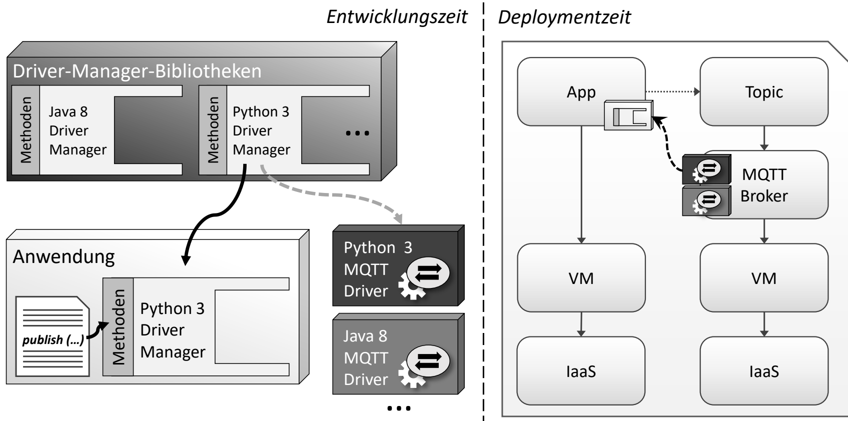

Abbildung 5.5: Programmiermodell und Artefakte zur Deploymentzeit (adaptiert von Saatkamp et al. [SBLW17]).

5.5.1 Programmiermodell

Das Ziel des Programmiermodells ist es, die Kommunikation zwischen einer Anwendung und der Integrations-Middleware vollständig zu kapseln. Durch die Verwendung der vordefinierten *Driver-Manager*-Bibliothek für die Implementierung einer Anwendung kann zu jeder Middleware, die einen passenden Treiber bereitstellt, eine Verbindung hergestellt werden. Ein Beispiel für die Verwendung von Integrations-Middleware ist das Veröffentlichen von Daten nach dem PUBLISH-SUBSCRIBE-Muster [HW04]. In Abbildung 5.5 ist das Konzept anhand einer Anwendung, die Daten auf ein *Topic* veröffentlicht, dargestellt. Die Anwendung ist in diesem Anwendungsfall ein sogenannter *Publisher*, der Daten auf das Topic schreibt. Subskribenten können Subskriptionen auf bestimmte Topics erstellen und über neu veröffentlichte Daten informiert werden und diese nach dem *Push*- oder *Poll*-Prinzip abrufen. Dies ist eine nachrichtenbasierte Kommunikation und wird auch häufig zur Integration bei Anwendungen in hybriden Clouds angewendet [FLR+14].

In Abbildung 5.5 werden das Programmiermodell zur Entwicklungszeit (links) und die Artefakte zur Deploymentzeit (rechts) gezeigt. Zur Entwick-

lungszeit werden verschiedene *Driver-Manager*-Bibliotheken zur Verfügung gestellt. Diese Bibliotheken sind für verschiedene Programmiersprachen und Versionen bereitgestellt, beispielsweise für Java 8 oder Python 3. Für das Publish-Subscribe-Muster stellen sie die Methoden *publish()* zum Veröffentlichen und *subscribe()* zur Subskription eines Topics zur Verfügung. Die konkrete Implementierung dieser Methoden ist jedoch von der jeweiligen Integrations-Middleware abhängig, da das unterstützte Transportprotokoll und andere middlewarespezifische Eigenschaften variieren können. Zwei Rollen werden zur Entwicklungszeit unterschieden: Die *Anwendungsentwicklerin* bzw. der *Anwendungsentwickler*, die die Driver-Manager-Bibliothek zur Implementierung der Anwendung nutzen, und der *Middleware-Anbieter*, der passende Treiber in verschiedenen Programmiersprachen, welche die Schnittstellen des Driver-Managers implementieren, bereitstellt. Die Treiber werden vom Driver-Manager zur Herstellung einer Kommunikationsverbindung mit der Integrations-Middleware verwendet. Für jede Middleware können Treiber in verschiedenen Programmiersprachen verfügbar sein. Ein MQTT-Broker stellt beispielsweise Treiber in Python 3 und Java 8 zur Verfügung, welche die Methoden zur Herstellung einer MQTT-Verbindung zum Broker umsetzen. Die Treiber müssen für jede Middleware nur einmal implementiert werden und können für jede Anwendung wiederverwendet werden.

Die implementierten Komponenten werden als Artefakte an die Komponenten des Deployment-Modells hinzugefügt. Der Komponententyp der Integrations-Middleware, im rechts dargestellten Beispiel in Abbildung 5.5 ein MQTT-Broker wie beispielsweise Eclipse Mosquitto [Ecl], definiert dafür für jeden verfügbaren Treiber in unterschiedlichen Programmiersprachen ein Artefakt. Auch der Quellcode der Anwendung selbst wird als Artefakt an die App-Komponenten angehängt. Zur Deploymentzeit muss der passende Treiber für die Anwendung selektiert und zusammen mit der Anwendung bereitgestellt werden, um die Kommunikationsbeziehung zwischen Anwendung und Integrations-Middleware zu realisieren.

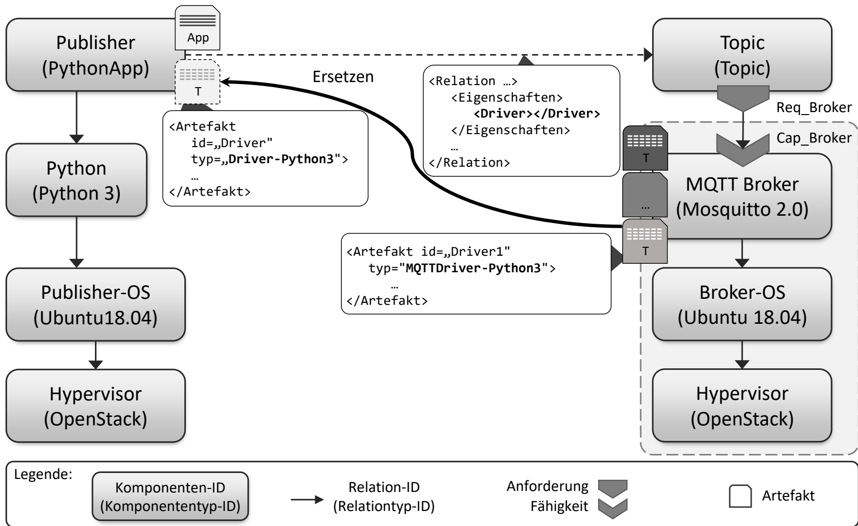

Abbildung 5.6: Modellierung und automatisierte Treiberselektion (adaptiert von Saatkamp et al. [SBLW17]).

5.5.2 Modellierung und automatisierte Treiberselektion

Die Selektion des passenden Treibers geschieht auf Basis des Deployment-Modells der Anwendung. In Abbildung 5.6 ist das bereits in Abbildung 5.5 eingeführte Beispiel detailliert dargestellt. Das Deployment-Modell der Anwendung kann unabhängig von einer konkreten Integrations-Middleware modelliert werden. Dafür wird die „Topic“-Komponente mit einer Anforderung modelliert, wie rechts in Abbildung 5.6 gezeigt. Durch das Match-Verfahren, welches in Abschnitt 4.4.2.3 zur Selektion geeigneter Hosting-Komponenten vorgestellt wurde, kann basierend auf der Anforderung eine passende verfügbare Integrations-Middleware in der jeweiligen Umgebung identifiziert und hinzugefügt werden. In diesem Beispiel ist dies Mosquitto 2.0, die eine MQTT-basierte Kommunikation ermöglicht. Die „MQTT Broker“-Komponente hat alle verfügbaren Treiber in unterschiedlichen Programmiersprachen, zum Beispiel in Python, als Artefakte angehängt.

Um einen geeigneten Treiber für eine konkrete Anwendungskomponente zu identifizieren wird die Vererbung zwischen Artefakttypen genutzt: Jeder Treiber erbt vom Supertyp „Driver“, einem abstrakten Artefakttyp. Zusätzlich gibt es für die verschiedenen Programmiersprachen und Versionen abstrakte Artefakttypen, die von „Driver“ erben, wie beispielsweise „Driver-Python3“. Ein Artefakt von diesem abstrakten Artefakttyp ist in Abbildung 5.6 an die „Publisher“-Komponente geheftet. Von diesen programmiersprachenspezifischen Artefakttypen erben wiederum die konkreten Artefakttypen der eigentliche Treiberimplementierungen, wie der „MQTTTreiber-Python3“ Artefakttyp. Damit ergibt sich für das genannte Beispiel folgende Vererbungshierarchie: „Driver“ → „Driver-Python3“ → „MQTTDriver-Python3“. Diese Vererbungskonvention ermöglicht die automatisierte Selektion passender Treiber, wie in Algorithmus 5.1 beschrieben.

Algorithmus 5.1 SelektiereTreiber(d)

1: *// Prüfe alle Komponenten mit einem abstrakten Artefakt vom Typ „Treiber“*
2: **for all** $(k_i \in K_d : \exists a_i \in artefakt_d(k_i) : (\pi_2(typ_d(a_i)) = wahr \wedge$
3: $\exists at_i \in AT_d : (\pi_1(at_i) = Driver \wedge at_i \in supertypen_d(typ_d(a_i)))))$ **do**
4: *// Prüfe alle ausgehenden Relationen vom Typ „connectsTo“*
5: **for all** $(r_i \in R_d : \pi_2(r_i) = k_i \wedge (\pi_1(typ_d(r_i)) = connectsTo \vee (\exists rt_c \in RT_d :$
6: $(\pi_1(rt_c) = connectsTo \wedge rt_c \in supertypen_d(typ_d(r_i)))))$ **do**
7: $k_m :=$ **FindeMiddleware**$(d, \pi_3(r_i))$
8: *// Wenn ein Treiber gefunden wurde, füge diesen zur Komponente hinzu*
9: **if** $(k_m \neq \perp \wedge \exists a_m \in artefakt_d(k_m) : supertyp_d(a_m) = typ_d(a_i))$ **then**
10: $artefakt_d(k_i) := artefakt_d(k_i) \cup \{a_m\}$
11: **let** $e_r \in eigenschaften_d(r_i) : \pi_1(e_r) = Driver$
12: $\pi_2(e_r) := (String, \pi_1(a_m))$
13: **else**
14: *// Wird kein Treiber gefunden, ist die Anwendung nicht deploybar*
15: **return** false
16: **end if**
17: **end for**
18: $artefakt_d(k_i) := artefakt_d(k_i) \setminus \{a_i\}$
19: **end for**
20: **return** true

Alle Komponenten, die ein abstraktes Artefakt definieren, welches vom Supertyp „Driver“ erbt, müssen für die Selektion berücksichtigt werden (Zeile 2-19). Da eine Komponente Kommunikationsbeziehungen zu verschiedenen Integrationssystemen aufbauen kann, müssen alle ausgehenden Kommunikationsbeziehungen bei der Selektion passender Treiber betrachtet werden (Zeile 5-17). Für jede Zielkomponente muss die Middleware-Komponente, welche die verschiedenen Treiber definiert, identifiziert werden. Dafür wird Algorithmus 5.2 verwendet. Der Algorithmus prüft rekursiv alle Komponenten in einem Hosting-Stack bis die Komponente mit definierten Treibern gefunden wurde (Zeile 2-4) oder es keine weiteren Hosting-Nachfolgerkomponenten mehr gibt (Zeile 8-9). Dafür wird der Algorithmus immer wieder mit der nächsten Hosting-Nachfolgerkomponente aufgerufen (Zeile 5-7) bis eine der zwei genannten Abbruchbedingungen eintritt.

Algorithmus 5.2 FindeMiddleware(d, k_j)

1: *// Prüfe ob Komponente Treiber-Artefakte angehängt hat*
2: **if** $(\exists a_j \in artefakt_d(k_j) : \exists at_j \in AT_d :$
3: $\quad (\pi_1(at_j) = Driver \land at_j \in supertypen_d(typ_d(a_j))))$ **then**
4: $\quad$ **return** k_j
5: **else if** $(hostnachfolger_d(k_j) \neq \emptyset)$ **then**
6: $\quad$ *// Sonst rekursiver Aufruf mit Hosting-Nachfolgerkomponente*
7: $\quad$ **return FindeMiddleware**$(d, hostnachfolger_d(k_j))$
8: **else**
9: $\quad$ **return** false
10: **end if**

Wurde eine Komponente mit Treiber-Artefakten identifiziert, wird in Algorithmus 5.1 geprüft, ob einer der Artefakttypen der definierten Treiber von dem abstrakten Artefakttyp erbt, der als Platzhalter an der zu betrachtenden Komponente definiert ist (Zeile 9). Existiert ein solches Artefakt mit dem entsprechenden Typ, wird dieses Artefakt zur Komponente hinzugefügt und die Id des Treibers als Eigenschaft der Relation definiert, um bei der Herstellung der Kommunikationsbeziehung identifizieren zu können, mit welchem Treiber kommuniziert werden muss (Zeile 10-12). Wird kein passender Treiber gefunden, ist die Anwendung nicht deploybar, da die

Kommunikationsbeziehung nicht aufgebaut werden kann (Zeile 13-15). Erst wenn alle ausgehenden Relationen geprüft sind, kann das abstrakte Artefakt entfernt werden. Das abstrakte Artefakt, wie in Abbildung 5.6 links zu sehen, wird benötigt, um den Selektionsbedarf zu ermitteln und die passenden Treiber zu selektieren. Nur wenn alle abstrakten Artefakte durch konkrete Arefakte ersetzt wurden, ist das Deployment-Modell umsetzbar.

Dieses Verfahren ist für alle Programmiersprachen, für die Mechanismen existieren, um eine Anwendung über externe Bibliotheken oder Skripte auszuführen, geeignet. Programmiersprachen wie Python, Java oder C++ ermöglichen Reflexion und dynamisches Linken zur Laufzeit. Für Sprachen, die nur statisches Verknüpfen beherrschen ist dieses Konzept nicht anwendbar. Für diese Sprachen können die Treiber als eigene Komponenten bereitgestellt werden, um das gleiche Ergebnis zu erzielen: Die Treiber werden als eigenständige Softwarekomponenten bereitgestellt und eine Kommunikation wird zwischen der Anwendung und dem Treiber hergestellt, der die empfangenen Daten an die angeschlossene Middleware weiterleitet. Dazu kann das bereits in Abschnitt 5.4 beschriebene Verfahren zur Modelladaption verwendet werden. Das in diesem Abschnitt eingeführte treiberbasierte Verfahren stellt eine Ergänzung zum Modelladaptionsverfahren dar und kann die Komplexität der Anwendung reduzieren.

5.6 Zusammenfassung und Diskussion

Das musterbasierte Verfahren zur Problemerkennung und Modelladaption ermöglicht, durch Verteilung und Restrukturierung auftretende Probleme frühzeitig automatisiert zu erkennen und zu beheben. Die Anwendbarkeit von Mustern, die von vielen Arbeiten betrachtet wird [Bre16; FCSK03; GL19; KE07], wird dabei anders als in vorherigen Arbeiten in einem zweistufigen Ansatz geprüft: Zuerst wird das Vorliegen eines Problems basierend auf der im Muster definierten Problem- und Kontextbeschreibung geprüft. Dadurch wird ausschließlich die Relevanz des Musters geprüft. Die Prüfung basiert auf einer generischen Definition des Problems. Die Anwendbarkeit der beschrie-

benen Lösung des Musters auf einen konkreten Anwendungsfall ist jedoch noch nicht gegeben, da die Realisierung unter anderem von den verwendeten Programmiersprachen, Technologien und Services abhängig ist. Zur Anwendung einer konkreten Lösungsimplementierung müssen daher auch diese Aspekte überprüft werden. Bei bisherigen Verfahren wurde sowohl die Problemerkennung als auch die Umsetzung der Lösung entweder auf einem hohen Abstraktionsgrad definiert [FCSK03; GL19] oder sehr spezifisch für konkrete Anwendungsfälle [AB14a; AB14b; Bre16]. Das in dieser Arbeit verwendete zweistufige Verfahren ermöglicht sowohl die Verwendung der generischen Beschreibungen der Muster als auch die spezifische Definition zur Anwendbarkeit konkreter Lösungsimplementierungen.

Das vorgestellte Verfahren ist jedoch nicht beschränkt auf die Erkennung von Problemen, welche in Mustern dokumentiert sind. Die Grundlage für die Formalisierung können unterschiedlich sein, zum Beispiel Compliance-Regeln oder Designprinzipien einer Organisation oder eines Unternehmens sowie Kompatibilitätseinschränkungen von Komponenten. Ein auf Graph-Matching basierendes Verfahren zur Prüfung von Compliance-Regeln in Deployment-Modellen wurde von Krieger et al. [KBKL18] eingeführt. Ebenso wie bei den musterbasierten Konzepten, die Graph-Matching-Verfahren zur Mustererkennung verwenden und ausführlich in Abschnitt 2.3.1 diskutiert wurden, kann die Nicht-Existenz von Elementen nicht geprüft werden. Auch Holmes und Zdun [HZ18] ermöglichen das Refactoring von Deployment-Modellen basierend auf Kundenanforderungen. Aus den Kundenanforderungen wird Architekturwissen abgeleitet und auf dieser Grundlage werden Transformationsregeln definiert, die Vorbedingungen spezifizieren, die erfüllt sein müssen, um die Transformation anzuwenden. Für die Migration von Cloud-Anwendungen führen Frey, Hasselbring und Schnoor [FHS13] *CloudMIG* ein, um die Umstrukturierung von existierenden Softwaresystemen zu cloudbasierten Anwendungen zu unterstützen. Die Zielarchitektur wird auf Basis von *Cloud Environment Constraints (CEC)* geprüft, um die Kompatibilität der Anwendung mit den selektierten Cloud-Umgebungen zu analysieren. Werden die CECs verletzt muss die Zielarchitektur angepasst werden. Für die CECs ist zusätzlich definiert wie schwerwiegend eine Verletzung für

die Anwendung ist. Die CECs werden für jede Cloudumgebung definiert, wie zum Beispiel Einschränkungen bezüglich der Größe einer Anwendung oder der unterstützten Programmiersprachen. Dabei werden ausschließlich vertikale Relationen berücksichtigt.

Das Verfahren zur Problemerkennung wurde im Rahmen dieser Arbeit bereits in einer wissenschaftlichen Publikation veröffentlicht [SBKL19a]. Das gleiche Prinzip wurde später auch zur Erkennung von schlechten Designs in Architekturen (*engl. „Architectural Smells“*) genutzt [BNS20; KVM+20; SMNB21]. Dabei wird geprüft ob wichtige Designprinzipien verletzt werden. Brogi, Neri und Soldani [BNS20] und Soldani et al. [SMNB21] erkennen Architectural Smells in Micorservice-Architekturen modelliert als TOSCA-basierte Deployment-Modelle. Dafür werden die Bedingungen, wann ein Designprinzip verletzt ist, als logische Formeln formalisiert. Auch die Adaption des Modells wird ermöglicht, jedoch können keine konkreten Implementierungen selektiert werden, sondern ausschließlich abstrakte Modellierungskonstrukte hinzugefügt werden. Auch Kumara et al. [KVM+20] nutzen das Verfahren zur Erkennung von Architectural Smells in TOSCA-basierten Deployment-Modellen. Dazu wird eine Ontologie definiert, welche wichtige Konzepte von TOSCA umfasst und auch konkrete Komponenten- und Relationtypen definiert. Dies ermöglicht eine standardisierte Modellierung der Deployment-Modelle. Erkennungsregeln für Architectural Smells werden in Form von SPARQL Anfragen definiert. Ein TOSCA-basiertes Deployment-Modell wird dafür zu einer Instanz der Ontologie transformiert und anschließend durch die definierten Regeln analysiert. Beide Arbeiten wurden nach dem musterbasierten Verfahren zur Problemerkennung veröffentlicht und bauen auf demselben Konzept auf.

5.6.1 Allgemeingültigkeit der Formalisierung von Mustern

In Abschnitt 2.3.1 wurden bereits ausführlich verschiedene Verfahren zur Erkennung von Lösungs- sowie Probleminstanzen von Mustern diskutiert. Zusammengefasst sind Graph-Matching-Verfahren verbreitet, jedoch ermöglichen sie nicht die Prüfung der Nicht-Existenz von Elementen. Logik erster

Ordnung wurde bisher nur für die Erkennung von Lösungsstrukturen verwendet und in dieser Arbeit auch für die Probleminstanzen angewendet. Grundsätzlich gilt für diese Arbeit wie auch für die übrigen Arbeiten, dass die Formalisierung der Muster auf der Interpretation der informellen textlichen Musterbeschreibungen durch die Autorinnen und Autoren beruht [CMT14; HZ18]. Darüber hinaus ist die Formalisierung auf eine bestimmte Domäne bzw. Sprache beschränkt. Die Formalisierung wird auf Basis einer bestimmten Modellierungs- oder Programmiersprache definiert, da die Syntax und auch Semantik entscheidende Rollen bei der automatisierten Erkennung spielen.

Die meisten Arbeiten in diesem Bereich beschränken sich auf die Analyse struktureller Eigenschaften von Modellen oder Codefragmenten. Verhaltensaspekte werden nur von wenigen Arbeiten berücksichtigt [HBF+20; TN03]. Dabei müssen zwei verschiedene Arten von Verhaltensmustern unterschieden werden: Harzenetter et al. [HBF+20] berücksichtigen die Konfiguration von Komponenten in Deployment-Modellen, wohingegen Taibi und Ngo [TN03] die Zustandsveränderung eines Objekts zur Laufzeit betrachten. Verhaltensmuster im Sinne der Konfiguration von Komponenten lassen sich auch mit dem in dieser Arbeit vorgestellten Konzept analysieren.

Die Variabilität von Instanzen sowohl der Lösungen als auch der Probleme ist eine weitere Schwierigkeit bei der Definition allgemeingültiger formalisierter Muster. Die Regeln zur Erkennung von Problemen können entweder auf abstrakten Typen definieren werden, wodurch die Genauigkeit einschränkt wird, oder es müssen verschiedene Regeln für ein Muster definieren werden, wodurch sich der Aufwand zur Definition und die Komplexität der Formalisierung der Muster erhöht. Mit dem in dieser Arbeit vorgestellten Konzept können sowohl generische Regeln als auch mehrere sehr spezifische Regeln für ein Muster definiert werden. Damit kann für jedes Muster individuell entschieden werden, ob eine generische oder mehrere spezielle Regeln besser geeignet sind.

5.6.2 Terminierung des musterbasierten Verfahrens

In einem Deployment-Modell können mehrere Probleme erkannt werden, und die Anpassung des Modells kann zu neuen Problemen führen. Die Reihenfolge, in der die Probleme gelöst werden, ist jedoch nicht vorgeschrieben und wird durch die Selektion der Entwicklerin oder des Entwicklers festgelegt. Das gesamte Verfahren zur Problemerkennung und Lösungsanwendung ist ein iterativer Prozess bei dem die zu lösenden Probleme sequenziell selektiert werden. Anders als in anderen Arbeiten wird nicht davon ausgegangen, dass die Menge an anzuwendenden Mustern und die Sequenz, auch Lösungspfad (*engl. „Solution Path"*) genannt, bestehend aus Mustern einer Mustersprache und deren Anwendungsreihenfolge, vorab bekannt sind [FBB+14a; FBB+14b; GL19].

Es gibt jedoch keine Garantie dafür, dass das musterbasierte Verfahren, welches in dieser Arbeit vorgestellt wird, schließlich terminiert. Jede Lösungsanwendung ändert den Zustand des Systems. Im Falle von Deployment-Modellen bedeutet dies, dass Anwendungskomponenten und Relationen zwischen Komponenten entfernt, hinzugefügt oder neu konfiguriert werden können. Wenn eine Anwendungssequenz zum gleichen Zustand wie zuvor führt, bestehen auch die gleichen Probleme wie zuvor und somit wird die Anwendungssequenz nicht beendet, solange keine alternative Sequenz existiert. Bei ausschließlicher Verwendung von Mustern einer alexandrinischen Mustersprache, die einer hierarchischen Struktur von Mustern folgen, kann es solche Schleifen nicht geben [FBL18] und auch in etablierten Mustersprachen mit wohldefinierten Verknüpfungen zwischen den Mustern [Zdu07] sind solche destruktiven Sequenzen von Mustern nicht enthalten. Oft werden jedoch Muster aus verschiedenen Mustersprachen benötigt, um die erkannten Probleme in einem System zu lösen. Auch wenn formal definiert werden kann, dass einzelne Mustersprachen zu einer Sprache aggregiert werden können [FBL18; WBB+20a], werden diese sprachübergreifenden Verknüpfungen von den Autorinnen und Autoren der Mustersprachen oft nicht spezifiziert. Damit ist keine wohldefinierte Semantik der Verknüpfungen zwischen den Mustern vorhanden. Darüber hinaus können nach der

Anwendung einer Lösung neue Probleme auftreten, sodass die Menge der Probleme und Lösungen nicht im Voraus bekannt ist.

Damit lässt sich zu Beginn des musterbasierten Verfahrens zur Problemerkennung und Modelladaption nicht bestimmen, ob das Verfahren terminiert. Wird jedoch das System in einen bereits zuvor erreichten Zustand überführt, das hießt, ist das Deployment-Modell zum Zeitpunkt $t+x$ identisch mit dem Deployment-Modell zum Zeitpunkt t, dann liegt ein Zyklus vor. Durch die Selektion alternativer Lösungen auf der Anwendungssequenz könnte dieser Zyklus beendet werden, eine Terminierung kann jedoch trotzdem nicht garantiert werden.

5.6.3 Anwendbarkeit von Modelladaption

Durch die Erkennung des durch ein Muster beschriebenen Problems lässt sich nicht direkt die Anwendbarkeit einer konkreten Lösung ableiten. Durch die Trennung dieser Aspekte in dieser Arbeit und das resultierende zweistufige Verfahren kann die Anwendbarkeit unterschiedlicher Lösungen im Kontext eines bestimmten Musters separat von der Problemerkennung erfolgen. Dabei wird jedoch das gleiche Verfahren angewendet, um den Deployment-Kontext zu definieren, in dem die Lösung anwendbar ist.

Verschiedene Konzepte zur generellen Modelladaption und speziell zur Anwendung von Mustern wurden bereits in Abschnitt 2.1.3.2 und Abschnitt 2.3.2.2 diskutiert. Ebenso wie bei der Erkennung von Problemen schränken die häufig verwendeten Graph-Matching-Verfahren zur Erkennung der Anwendbarkeit die Expressivität ein. Grundsätzlich basieren alle Konzepte auf der Erkennung von Vorbedingungen, die erfüllt sein müssen, um die Adaption oder Transformation anwenden zu können. Bei den musterbasierten Verfahren wurde jedoch bisher nicht zwischen der generellen Anwendbarkeit des Musters, das heißt der Problemerkennung, und der Anwendbarkeit einer konkreten Lösung, die eine Implementierung der im Muster beschriebenen generischen Lösung repräsentiert, unterschieden. Das zweistufige Verfahren, welches in dieser Arbeit eingeführt wurde, ermöglicht diese Unterscheidung. Dabei muss sichergestellt sein, dass durch die

Anwendung einer Lösungsimplementierung nach einem erkannten Problem alle Merkmale des Problems im Modell eliminiert werden. Dazu zählt unter anderem auch die Annotation mit zusätzlichem Wissen.

Mit dem Konzept zur flexiblen Treiberselektion zur Anpassung der Kommunikationsprotokolle wird ein zusätzlicher Ansatz für die Adaption von Kommunikationsbeziehungen in dieser Arbeit vorgestellt. Das Konzept verschiedene Treiber zu verwenden ist auch in anderen Bereichen verbreitet, zum Beispiel bei der Selektion von Cloud-Diensten, die verschiedene Treiber für die unterschiedlichen Cloud-Anbieter nutzen [GMMC13; MŞP14].

Automatisiertes partnerübergreifendes dezentrales Deployment

Für das Deployment partnerübergreifender Anwendungen im letzten Schritt der DivA-Methode (Kapitel 3) wird in diesem Kapitel ein dezentrales Verfahren vorgestellt, welches die Autonomie der Partner sowie die Datenkapselung berücksichtigt. Das Konzept beruht auf der Kombination von deklarativen und imperativen Technologien zur Ausführung des Deployments. Dies wurde bereits in zentralisierten Deploymentverfahren erprobt und wird in diesem Kapitel auf das dezentrale Deployment erweitert. In Abschnitt 6.1 wird erst die grundlegende Idee präsentiert, und anschließend wird in Abschnitt 6.2 die Generierung imperativer Modelle speziell für partnerübergreifende Anwendungen vorgestellt. Dieser Ansatz automatisiert das dezentrale Deployment und stellt Forschungsbeitrag 4 dieser Arbeit dar. Das Verfahren wurde im Rahmen dieser Arbeit bereits in einer wissenschaftlichen Publikation veröffentlicht [WBK+20].

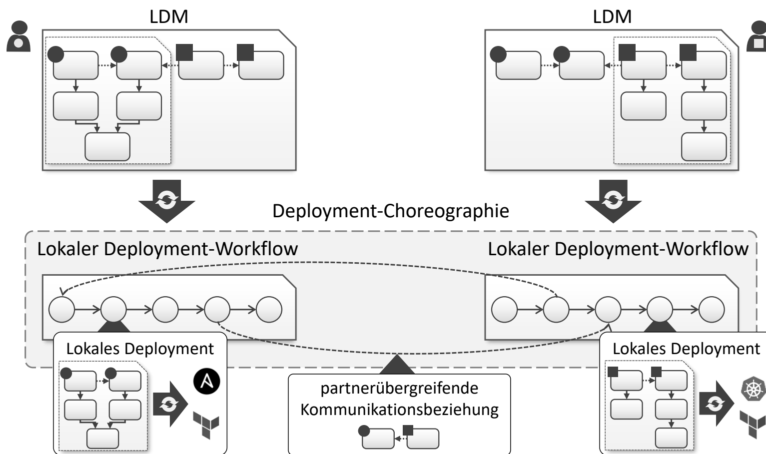

Abbildung 6.1: Grundkonzept des automatisierten partnerübergreifenden dezentralen Deployments.

6.1 Idee und Grundkonzept

Die grundlegende Idee des dezentralen Deployments ist es, die gesamte Anwendungsbereitstellung dezentral global zu koordinieren, während die beteiligten Partner die lokale Ausführung kontrollieren und steuern können. In Abbildung 6.1 ist das Grundkonzept skizziert. Als Resultat der in den vorherigen Kapiteln beschriebenen Verfahren zur Verteilung (Kapitel 4) und musterbasierten Modellanalyse und -adaption (Kapitel 5) stehen restrukturierte adaptierte LDMs der beteiligten Partner zur Verfügung, welche mit partnerspezifischen Deployment-Informationen angereichert sind. Für jedes LDM wird ein *lokaler Deployment-Workflow (LDW)* generiert, welcher die Aktivitäten zur Bereitstellung der lokalen Komponenten sowie zum Nachrichtenaustausch mit den anderen Partnern orchestriert.

Für die lokalen Komponenten können von jedem Partner die jeweils präferierten Deployment-Technologien, wie beispielsweise Terraform, Ansible oder Chef, genutzt werden. Dabei können auch mehrere Technologien kombiniert werden, die unterschiedliche Teile der Anwendung verwal-

ten [WBB+21]. Zusätzlich werden Informationen ausgetauscht, die zur Realisierung partnerübergreifender Kommunikationsbeziehungen benötigt werden. Die anwendungsspezifischen Komponenten sowie die Zuordnung der Partner sind Teil des GDMs und damit global allen beteiligten Partnern bekannt. Dadurch kann (i) die korrekte Reihenfolge der Aktivitäten abgeleitet werden und können (ii) die partnerübergreifenden Kommunikationsbeziehungen sowie der zugehörige Partner identifiziert werden.

Die Nutzung von Workflow-Technologien zur Ausführung des Deployments ermöglicht zusätzlich die bei Bedarf manuelle Anpassung des Deployments, wenn zusätzliche Aktivitäten über die aus dem Deployment-Modell abgeleiteten Aktivitäten hinaus benötigt werden. Durch eine rein programmatische Lösung wäre die Anpassung an spezifische Anforderungen nicht ohne Änderungen des Quellcodes möglich. Zusätzliche Aktivitäten können beispielsweise menschliche Tätigkeiten, wie die Freigabe von Ressourcen, oder auch zusätzliche Konfigurationsaktivitäten sein.

Die verschiedenen LDWs bilden die *Deployment-Choreographie*. In diesem Fall ist dies eine teilnehmerorientierte (*engl. „participant-driven“*) Realisierung der Choreographie, da die Choreographie sich aus Fragmenten zusammensetzt, die von autonomen Teilnehmern kontrolliert werden [TMKO17]. Jedoch sind die Fragmente nicht das Resultat eines Fragmentierungsprozesses, sondern die Fragmente werden basierend auf global definierten Schnittstellen und Kommunikationsendpunkten verteilt durch die jeweiligen Partner generiert.

6.2 Generierung lokaler Workflows zur Choreographie

Die Generierung von Deployment-Workflows für die zentralisierte Orchestrierung der gesamten Anwendung ist bereits durch existierende Konzepte möglich [BBK+14; CCDT18]. Generell gilt: Für vertikale Relationen, das heißt Relationen vom Typ „HostedOn“ oder eines spezialisierten Typs, der davon erbt, muss die Zielkomponente verfügbar sein, damit die Quellkomponente bereitgestellt werden kann. Für horizontale Relationen, das heißt

Relationen vom Typ „ConnectsTo“ oder eines spezialisierten Typs, der davon erbt, müssen beide Komponenten verfügbar sein, bevor die Relation realisiert werden kann [BBK+14]. Eine horizontale Relation wird meist durch die Konfiguration der Quellkomponente realisiert. Ob eine horizontale Relation unabhängig von der Bereitstellung der Quellkomponente, das heißt, der Komponente, die eine Verbindung aufbaut, hergestellt werden kann, ist dabei auch abhängig von der Implementierung der Komponente und der verwendeten Deployment-Technologie. Die Bereitstellungsreihenfolge der Komponenten ergibt sich daher durch die Umkehrung der Relationen im Deployment-Modell. Diese Ansätze zur Generierung zentralisierter Deployment-Workflows wurden erweitert, um anstatt die Bereitstellung zentral zu orchestrieren, diese dezentral zu choreographieren. Dabei muss (i) die globale Koordination des gesamten Deployments und (ii) der Austausch zwischen den Partnern zur Realisierung partnerübergreifender Kommunikationsbeziehungen ermöglicht werden. Diese speziellen Relationen im Deployment-Modell sind wie folgt definiert:

Definition 6.1 (Partnerübergreifende Kommunikationsbeziehung)
Partnerübergreifende Kommunikationsbeziehungen (PKs) sind vom Typ „ConnectsTo“ oder eines von diesem Typ erbenden Relationtyps und zwischen Komponenten, die von verschiedenen Partnern verwaltet werden. Für die Menge der partnerübergreifenden Kommunikationsbeziehungen $PK \subseteq R_d$ gilt:

$$\begin{aligned}\forall r_i \in PK : \Big(&\pi_1(typ_d(r_i)) = connectsTo \vee \\ &\big(\exists rt_i \in supertypen_d(typ_d(r_i)) : \pi_1(rt_i) = connectsTo\big)\Big) \wedge \\ &\Big(\exists k_s, k_t \in K_d \exists p_s, p_t \in P_d, p_s \neq p_t : \pi_2(r_i) = k_s \wedge \pi_3(r_i) = k_t \wedge \\ &\qquad partner_d(k_s) = p_s \wedge partner_d(k_t) = p_t\Big).\end{aligned}$$

Zur Realisierung einer PK müssen die Partner die Eingabeparameter der jeweiligen Operation, die die Relation implementiert, austauschen. In Abbildung 4.2 wurde bereits ein einfaches Beispiel eines GDMs eingeführt. Zur Herstellung einer Verbindung zu der Java-Anwendung werden die IP-

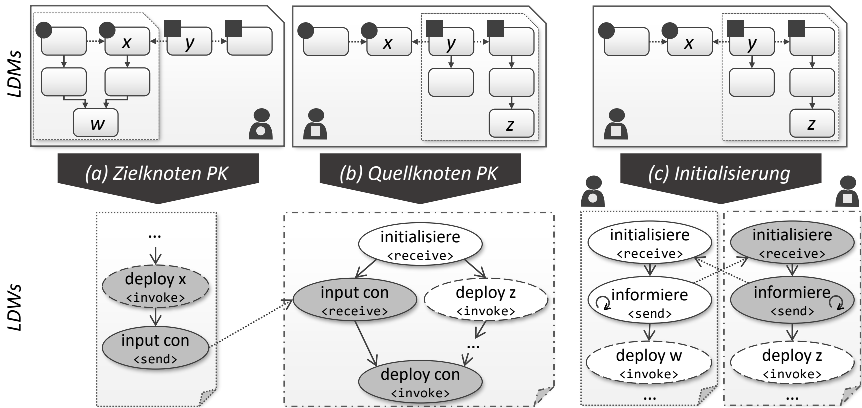

Abbildung 6.2: Aktivitäten (a) für Zielknoten von PKs, (b) für Quellknoten von PKs und (c) zur Initialisierung des gesamten Deployments (adaptiert von Wild et al. [WBK+20]).

Adresse und die Port-Nummer benötigt. Beide Eingabeparameter müssen von dem Partner bereitgestellt werden, der die Zielkomponente verwaltet. Da diese Informationen erst zum Zeitpunkt des Deployments verfügbar sind, muss dieser Datenaustausch während des Deployments ermöglicht werden. Darüber hinaus muss das Deployment der gesamten Anwendung sichergestellt werden. Unabhängig davon, welcher Partner das Deployment initiiert, müssen alle anderen Partner die Bereitstellung ebenfalls anstoßen. Vor allem die korrekte Reihenfolge der Aktivitäten, auch im globalen Kontext, muss berücksichtigt werden, um einen Deadlock zur Laufzeit zu vermeiden.

6.2.1 Fallunterscheidung zur Generierung der Aktivitäten für LDWs

Für die Generierung der LDWs müssen drei Fälle unterschieden werden, um die globale Bereitstellung zu ermöglichen. Diese Fallunterscheidung ist in Abbildung 6.2 dargestellt. Im oberen Teil der Abbildung sind abstrahierte LDMs von zwei Partnern dargestellt und im unteren Teil die generierten Aktivitäten in den LDWs für den jeweiligen Fall. Fall (a) zeigt die Aktivitäten aus der Perspektive des Partners, der die Zielkomponente einer PK verwaltet,

Fall (b) die Aktivitäten aus der Perspektive des Partners, der die Quellkomponente einer PK verwaltet und Fall (c) die Aktivitäten die zur globalen Initialisierung des gesamten Deployments benötigt werden. Zunächst wird eine Definition für lokale Deployment-Workflow-Modelle in Anlehnung an die Produktionsprozessdefinition gegeben [Kha08; LR00]:

Definition 6.2 (Lokales Deployment-Workflow-Modell) Sei d ein GDM mit einer Menge an Partnern P, dann existiert für jeden Partner $p_i \in P$ ein LDM d_i. Für jeden Partner p_i ist ein lokales Deployment-Workflow-Modell w_i basierend auf d_i wie folgt definiert:

$$w_i = (A_{w_i}, E_{w_i}, V_{w_i}, i_{w_i}, o_{w_i}, typ_{w_i}, modellref_{w_i})$$

Die Elemente des Tuples w_i sind dabei wie folgt definiert:

- $A_{w_i} \subseteq \wp(\Sigma)$ ist die Menge aller *Aktivitäten* in w_i, wobei jedes Element $a_i \in A_{w_i}$ eine eindeutig identifizierbare Aktivität beschreibt.
- $E_{w_i} \subseteq A_{w_i} \times A_{w_i}$ ist die Menge aller *Kontrollkonnektoren* in w_i, wobei jedes Element $e_i = (a_s, a_t) \in E_{w_i}$ ein eindeutig identifizierbarer Konnektor zwischen der Aktivität a_s und der Aktivität a_t ist. Die Aktivität a_s muss abgeschlossen sein bevor die Aktivität a_t starten kann.
- $V_{w_i} \subseteq \wp(\Sigma) \times \wp(\Sigma)$ ist die Menge der Datenelemente, wobei ein Datenelement als $v_i = (Datentyp, Wert) \in V_{w_i}$ definiert ist.
- i_{w_i} bildet jede Aktivität $a_i \in A_{w_i}$ auf die Menge an Datenelementen ab, die als Eingabeparameter für die Aktivität benötigt werden. Dies wird *Eingabecontainer* genannt: $i_{w_i} : A_{w_i} \rightarrow \wp(V_{w_i})$.
- o_{w_i} bildet jede Aktivität $a_i \in A_{w_i}$ auf die Menge an Datenelementen ab, die als Ausgabeparameter von der Aktivität zurückgegeben werden. Dies wird *Ausgabecontainer* genannt: $o_{w_i} : A_{w_i} \rightarrow \wp(V_{w_i})$.
- typ_{w_i} bildet jede Aktivität $a_i \in A_{w_i}$ auf genau einen Aktivitätstypen ab, wobei die Menge der Aktivitätstypen anders als bei Deployment-Modellen beschränkt ist: $typ_{w_i} : A_{w_i} \rightarrow \{\texttt{receive}, \texttt{send}, \texttt{invoke}\}$.

- $\mathit{modellref}_{w_i}$ bildet jede Aktivität $a_i \in A_{w_i}$ auf kein, ein oder mehrere Modellelemente aus dem Deployment-Modell d ab, auf die sich die Aktivität bezieht: $\mathit{modellref}_{w_i} : A_{w_i} \rightarrow \wp(M_d) \cup \{\perp\}$

Die Menge der Aktivitätstypen ist auf die für diesen Anwendungsfall benötigten Typen beschränkt und wird von gängigen Workflow-Technologien, wie BPEL und BPMN, unterstützt. Die Bezeichnungen können jedoch abhängig von der jeweiligen Technologie abweichen. `Receive`-Aktivitäten ermöglichen das Empfangen von Nachrichten, `send`-Aktivitäten ermöglichen das Senden von Nachrichten und `invoke`-Aktivitäten die Ausführung von Services oder Skripten, zum Beispiel für das Deployment bestimmter Komponenten. Basierend auf dieser Definition können LDWs für die in Abbildung 6.2 gezeigten Fälle wie folgt definiert werden. In Abbildung 6.2 sind die resultierenden Fragmente der drei Fälle, (a) Zielknoten PK, (b) Quellknoten PK und (c) Initialisierung zur Veranschaulichung skizziert. Dabei sei d ein GDM und d_i und d_j die zugehörigen LDMs der Partner p_i und p_j.

(a) Zielknoten PK: Für alle Komponenten $k_t \in K_{d_i}$ mit $\mathit{partner}_{d_i}(k_t) = p_i$, für die eine Relation $r_p \in R_{d_i}$ mit $\pi_3(r_p) = k_t$ existiert, wird nach der Deploy-Aktivität $a_t \in A_{w_i}$ der Zielkomponente k_t mit $\mathit{typ}_{w_i}(a_t) = \texttt{invoke}$ und $k_t \in \mathit{modellref}_{w_i}(a_t)$, eine Aktivität a_c mit $\mathit{typ}_{w_i}(a_c) := \texttt{send}$ und $\mathit{modellref}_{w_i}(a_c) := r_p$ zum Senden der Verbindungsinformationen mit einem Kontrollkonnektor $e_c = (a_t, a_c)$ zu w_i hinzugefügt. Sobald Aktivität a_t abgeschlossen ist, können die Informationen zur Herstellung einer Verbindung versendet werden. Sei außerdem $op_c \in \mathit{operationen}_d(r_p)$ die Operation zur Realisierung der Relation, dann ist $i_{w_i}(a_c) = \pi_2(op_c)$, das heißt die Eingabeparameter der Operation sind die Eingabeparameter der Aktivität.

(b) Quellknoten PK: Für alle Komponenten $k_s \in K_{d_j}$ mit $\mathit{partner}_{d_j}(k_s) = p_j$, für die eine Relation $r_p \in R_{d_j}$ mit $\pi_2(r_p) = k_s$ existiert, wird vor der Deploy-Aktivität $a_s \in A_{w_j}$ der Quellkomponente k_s mit $\mathit{typ}_{w_j}(a_s) = \texttt{invoke}$ und $k_s \in \mathit{modellref}_{w_j}(a_s)$, eine Aktivität a_r mit $\mathit{typ}_{w_j}(a_r) := \texttt{receive}$ und $\mathit{modellref}_{w_j}(a_r) := r_p$ zum Empfangen der Verbindungsinformationen mit einem Kontrollkonnektor $e_r = (a_r, a_s)$ zu w_j hinzugefügt.

Bevor die Quellkomponente der PK bereitgestellt wird, müssen die Verbindungsinformationen verfügbar sein.

(c) Initialisierung: Jeder LDW w_i, w_j startet mit der Initialisierungsaktivität $a_{init} \in A_{w_i}$ mit $typ_{w_i}(a_{init}) = \texttt{receive}$ und $modellref_{w_i} = \perp$. Zur Sicherstellung, dass nicht nur der LDW ausgeführt wird, sondern global die Instanziierung initialisiert wird, wird nach a_{init} eine Benachrichtigung an alle übrigen Partner zur Initialisierung geschickt. Damit wird in w_i für jeden $p_z \in P \setminus \{p_i\}$ eine Aktivität a_n mit $typ_{w_i}(a_n) = \texttt{send}$ mit einem Kontrollkonnektor $e(a_{init}, a_n)$ hinzugefügt.

Mit diesen Regeln können alle zusätzlich benötigten Aktivitäten, die über die Aktivitäten zum Deployment der lokalen Komponenten hinausgehen und die für die dezentrale Ausführung benötigt werden, für jeden LDW generiert werden. Abhängig von der Deployment-Technologie und Umsetzung der Workflows werden im schlechtesten Fall bei n Partnern $n*(n-1)$ Nachrichten verschickt, wobei jeder Partner $n-1$ Nachrichten verwerfen muss. Dieser Fall tritt ein, wenn jeder Partner jeden anderen Partner informiert. Über die global gültige Korrelations-Id für jede Bereitstellung der Anwendung wird jeder LDW für eine Instanz der Anwendung nur einmal instanziiert. Durch Modellierungsmechanismen, wie sie beispielsweise von BPMN in Form von Gateways zur Definition von Bedingungen bereitgestellt werden, kann die Anzahl an Initialisierungsnachrichten auf $n-1$ Nachrichten reduziert werden, indem nur unter der Bedingung, dass der Partner selbst das Deployment initialisiert, die übrigen Partner informiert werden.

Für die Generierung der übrigen Aktivitäten sowie zur Selektion der konkreten Workflow-Fragmente wird der von Breitenbücher et al. [BBK+14] vorgestellte Plan-Generator verwendet und erweitert: Basierend auf dem Deployment-Modell wird die Provisionierungsreihenfolge der Komponenten bestimmt und die PKs identifiziert. Dies dient als Input zur Generierung eines Gerüsts des Deployment-Workflows, welches anschließend mit einer konkreten Workflow-Technologie umgesetzt werden. Für diese Arbeit wurde BPMN verwendet, wie in Kapitel 7 näher erläutert.

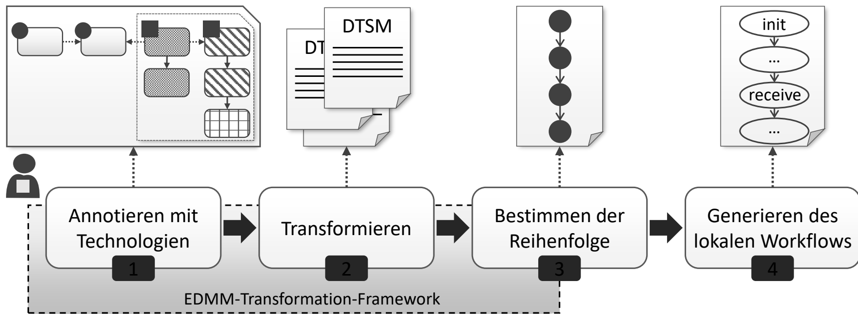

Abbildung 6.3: Prozess zur Workflow-Generierung mit Integration des EDMM-Transformation-Frameworks zur Nutzung verschiedener Deployment-Technologien. Die dunkelgrau hinterlegten Schritte sind durch existierende Arbeiten abgedeckt [WBB+19; WBB+21].

6.2.2 Integration mit EDMM-Framework

Um die Nutzung verschiedener Deployment-Technologien durch die jeweiligen Partner zu ermöglichen, wurde das dezentrale Deployment-Konzept mit dem EDMM-Transformation-Framework integriert und mit dem in Zusammenarbeit mit Wurster et al. [WBB+21] entwickelten Verfahren zur Kombination unterschiedlicher Deployment-Technologien verknüpft. In Abbildung 6.3 ist das Vorgehen skizziert. Die ersten zwei Schritte sind bereits durch existierende Arbeiten abgedeckt [WBB+19; WBB+21], der dritte Schritt wurde entsprechend den Anforderungen an das dezentrale Deployment erweitert und ein Schritt zur Generierung von LDWs hinzugefügt.

Im ersten Schritt werden sogenannte *Technologieregionen* im lokalen Teil des LDMs annotiert. Für jede Komponente $k_i \in K_d$ wird durch die $techgruppe_d(k_i)$ festgelegt, durch welche Deployment-Technologie sie bereitgestellt und verwaltet werden soll. Basierend auf den Annotationen werden die Deployment-Gruppen bestimmt, die alle Komponenten umfassen, die von einer Technologie in einem „*One-Shot*"-Deployment bereitgestellt werden können. Dabei darf es keine zyklischen Abhängigkeiten zwischen den

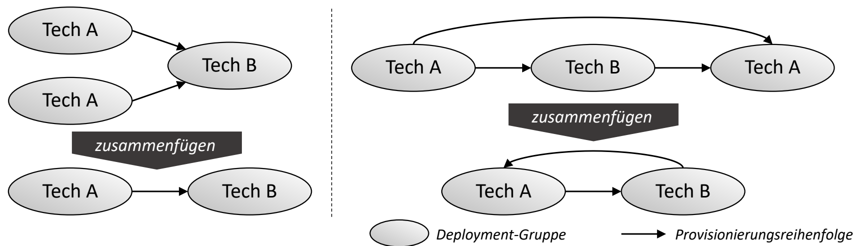

Abbildung 6.4: Deployment-Gruppen, die zusammengefügt werden können (links) und die nicht zusammengefügt werden können (rechts) (adaptiert von Saatkamp et al. [SBKL19b]).

Deployment-Gruppen geben [SBKL19b]. Dazu wird der *Provisionierungsreihenfolgegraph (PRG)* berechnet, dessen Knoten die Deployment-Gruppen und dessen Kanten deren Provisionierungsreihenfolge repräsentieren. Das Vorgehen basiert auf dem im Rahmen dieser Arbeit entwickelten Verfahren zur Fragmentierung von Deployment-Modellen für unterschiedliche Partner [SBKL19b] und wurde für das Deployment mit unterschiedlichen Technologien adaptiert. In Abbildung 6.4 werden zwei Fälle bei der Bestimmung der Deployment-Gruppen unterschieden: Die Gruppen links, die *Technologie A* zugeordnet sind, können zusammengefügt werden, da keine zyklischen Abhängigkeiten durch die Zusammenführung entstehen. Die Gruppen rechts, die *Technologie A* zugeordnet sind, können hingegen nicht zusammengefügt werden, da dadurch der vormals azyklische PRG zyklisch werden würde. So kann iterativ die Anzahl an Deployment-Gruppen durch Kantenkontraktion des PRG reduziert werden, beginnend mit der Zuordnung von einer Deployment-Gruppe pro Komponente. Bei der Provisionierungsreihenfolge der Deployment-Gruppen ist zu berücksichtigen, dass die Kanten zwischen den Deployment-Gruppen durch Umdrehen der Relationen im Deployment-Modell erzielt werden. Bei der Bestimmung der Deployment-Gruppen müssen auch die Abhängigkeiten zu den Komponenten der anderen Partner berücksichtigt werden. Die Komponenten werden dabei ohne Zuordnung einer spezifischen Technologie, sondern ausschließlich in Bezug

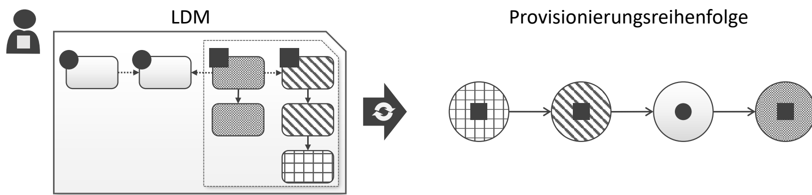

Abbildung 6.5: Abgeleitete Provisionierungsreihenfolge aus LDM.

auf die Zuordnung zu einem anderen Partner betrachtet.

Im zweiten Schritt werden alle Deploymentgruppen des Partners in *deploymenttechnologiespezifische Modelle (DTSMs)* transformiert. In diesem Schritt werden alle Files und Modelle generiert, die von der jeweiligen Technologie, wie Kubernetes, Terraform oder Chef benötigt werden. Dabei müssen die Abhängigkeiten zwischen den Gruppen berücksichtigt werden und Konfigurationsparameter sowie Laufzeitparameter, wie beispielsweise die IP-Adresse, als Ausgabe- und Eingabeparameter berücksichtigt werden können. Welche Parameter von anderen DTSMs benötigt werden, kann aus den Eigenschaften der Komponenten im Deployment-Modell abgeleitet werden.

Im dritten Schritt wird die Provisionierungsreihenfolge der DTSMs bestimmt, die als Basis für die Generierung des lokalen Workflows dient. Dazu wird der PRG aus dem ersten Schritt topologisch sortiert. Dabei müssen auch die Abhängigkeiten zu den Komponenten der übrigen Partner berücksichtigt werden. Aus dem Beispiel in Abbildung 6.3 ergibt sich damit die in Abbildung 6.5 dargestellte Reihenfolge der Bereitstellung: Zuerst muss die karierte Komponente mit der entsprechenden Technologie bereitgestellt werden, anschließend können die gestreiften Komponenten mit einer anderen Technologie dort installiert werden. Die Komponente des anderen Partners (•) muss dabei vor dem grau gepunkteten Stack bereitgestellt sein, damit eine Verbindung zu der Komponente des anderen Partners hergestellt werden kann. Die Bereitstellung durch den anderen Partner kann dabei auch parallel zu den karierten und gestreiften Komponenten erfolgen. Dieser Graph dient als Basis für den vierten Schritt, die Generierung des Workflows.

Im vierten Schritt werden basierend auf dem PRG die entsprechenden Aktivitäten, wie bereits in Abschnitt 6.2.1 erläutert, generiert. Bei der Verwendung verschiedener Deployment-Technologien werden nicht die einzelnen Lebenszyklusoperationen der Komponenten im Workflow aufgerufen, wie es bei dem Konzept von Breitenbücher et al. [BBK+14] der Fall ist, sondern das Deployment der DTSM durch die jeweilige Technologie initiiert. Der LDW orchestriert damit (i) das Deployment durch die verschiedenen Technologien und (ii) den Nachrichtenaustausch mit den Partnern. Das Konzept ist dabei unabhängig von einer konkreten Workflow-Technologie.

Zur Generierung und Ausführung der LDWs benötigt jeder Partner eine Instanz des DivA-Werkzeugs und eine Workflow-Engine für die gewählte Workflow-Technologie. In Kapitel 7 wird das Werkzeug sowie die konkrete Ausführung anhand von BPMN erläutert.

6.3 Zusammenfassung und Diskussion

Das dezentrale Deployment durch Kombination deklarativer und imperativer Technologien ermöglicht die Erhaltung der Souveränität und Autonomie der beteiligten Partner und die Bereitstellung verteilter partnerübergreifender Anwendungen ohne zentralisierte Kontrollinstanz. Aus dem deklarativen Deployment-Modell können die lokalen Deployment-Workflows abgeleitet und automatisiert generiert werden. Die lokalen Workflows orchestrieren das Deployment mit, bei Bedarf, unterschiedlichen Deployment-Technologien, den Nachrichtenaustausch mit anderen Partnern sowie die globale Initialisierung des gesamten Deployments. Die Verwendung von Workflow-Technologien ermöglicht die zuverlässige, robuste und transaktionale Ausführung des Deployments und die Kompensation im Fehlerfall.

Workflows und im speziellen Choreographien werden in verschiedenen Bereichen eingesetzt und es gibt verschiedene Ansätze zur Modellierung von Choreografien, zum Beispiel mit BPEL [DKLW07] oder zur Partitionierung von Orchestrierungs-Workflows in mehrere Workflows, die eine Choreographie bilden [JDB16; Kha08; KL06]. Allerdings basieren die meisten Deploy-

ment-Technologien auf einem deklarativen Deployment-Modell [WBF+19], da die manuelle Definition der einzelnen Aktivitäten, die in der richtigen Reihenfolge ausgeführt werden müssen, um einen gewünschten Zustand einer Anwendung zu erreichen, fehleranfällig ist. Daher wurde der Fokus beim dezentralen Deployment nicht auf die Modellierung von Deployment-Choreographien gelegt, sondern auf die Generierung unabhängiger Workflows, die implizit eine Deployment-Choreographie bilden. Breitenbücher et al. [BBK+14] demonstrierten wie man Workflows aus deklarativen Deployment-Modellen ableiten kann. Ihr Ansatz ermöglicht jedoch nur die Generierung von Workflows, die für ein zentralisiert koordiniertes Deployment verwendet werden können. Die Generierung von zentralisierten Workflows, die später durch die genannten Partitionierungsverfahren fragmentiert werden, führt zu zusätzlichem Adaptionsaufwand. Deshalb wurde der Generierungsansatz von Breitenbücher et al. [BBK+14] so erweitert, dass LDWs unabhängig voneinander basierend auf definierten Schnittstellendefinitionen generiert werden. Nichtsdestotrotz können die genannten Partitionierungsverfahren für bereits existierende Management-Pläne, welche auf die beteiligten Partner verteilt werden müssen, zum Einsatz kommen. Das Management während der Laufzeit der Anwendung wird im Rahmen dieser Arbeit jedoch nicht betrachtet und bringt neue Herausforderungen bei der Koordination zwischen den beteiligten Partnern, beispielsweise beim Update von Komponenten, mit sich.

Angelehnt an die Definition von Aryal, Marshall und Altmann [AMA19] folgt der dezentrale Deployment-Ansatz einer *Multipoint Full-Duplex* Konfiguration, bei der alle Beteiligten, in diesem Fall Deployment-Partner, miteinander kommunizieren können und es keine übergeordnete zentrale Koordinationseinheit gibt. Die Instanzen des DivA-Werkzeugs bilden eine Föderation, die auf dezentraler Interaktion beruht, so wie auch viele Ansätze zur Realisierung von Cloud-Föderationen [AB16]. Eine Cloud-Föderation oder auch föderierte Cloud (*engl. „Federated Clouds“*) ist eine Form der *Inter-Cloud*, einer Cloud von Clouds [AB16; GB14]: Während die Multi-Cloud ein nutzerzentrischer Ansatz ist, bei dem mehrere unabhängige Cloud-Anbieter, die keine Kenntnis voneinander haben, zur Bereitstellung einer Anwendung

von der Nutzerin oder dem Nutzer herangezogen werden, ist eine föderierte Cloud ein anbieterzentrischer Ansatz, die einen Zusammenschluss von verschiedenen Clouds darstellt, bei dem die Nutzerin oder der Nutzer nicht zwangsweise Kenntnis über die Nutzung unterschiedlicher Clouds hat. Peer-to-Peer-Architekturen ermöglichen die dezentrale Koordination zwischen den beteiligten Anbietern, wobei häufig Broker für die Kommunikation genutzt werden [GB14; SBA12]. In föderierten Clouds liegt der Fokus auf der Verhandlung zwischen den Beteiligten über die Nutzung von Ressourcen und die Durchführung von Migrationen oder das Skalieren von Anwendungskomponenten. Die Souveränität bezieht sich damit ausschließlich auf die Infrastruktur und nicht auf die Anwendungsebene. Durch die dezentrale Bereitstellung, wie sie in dieser Arbeit eingeführt wird, können auch zusätzliche Management-Dienste, beispielsweise für Backups, von den einzelnen Partnern genutzt werden, ohne dass dies den übrigen Partnern bekannt gemacht werden muss. Dies gilt solange die Änderungen nicht die Funktionalität der übrigen Komponenten beeinflussen.

Kapitel 7

Architektur und Validierung des DivA-Werkzeugs

In diesem Kapitel wird zuerst die Architektur des DivA-Werkzeugs sowie dessen prototypische Implementierung vorgestellt. Das DivA-Werkzeug setzt die in den vorherigen Kapiteln eingeführten Konzepte um und ermöglicht damit die Ausführung und Automatisierung der DivA-Methode (Kapitel 3). Das Werkzeug ist in bestehende Systeme zur Modellierung und Bereitstellung von Anwendungen integriert. In Abschnitt 7.1 wird zuerst die Architektur des Werkzeugs präsentiert. Die prototypische Implementierung wird anschließend in Abschnitt 7.2 vorgestellt. Basierend auf der prototypischen Implementierung werden in Abschnitt 7.3 die musterbasierte Problemerkennung und -behebung sowie die Workflow-Generierung validiert. Die Integration mit anderen Werkzeugen, welche nicht im Fokus dieser Arbeit stehen, wird in Abschnitt 7.4 kurz erläutert. Das DivA-Werkzeug stellt damit den letzten Forschungsbeitrag, Forschungsbeitrag 5, dieser Arbeit dar.

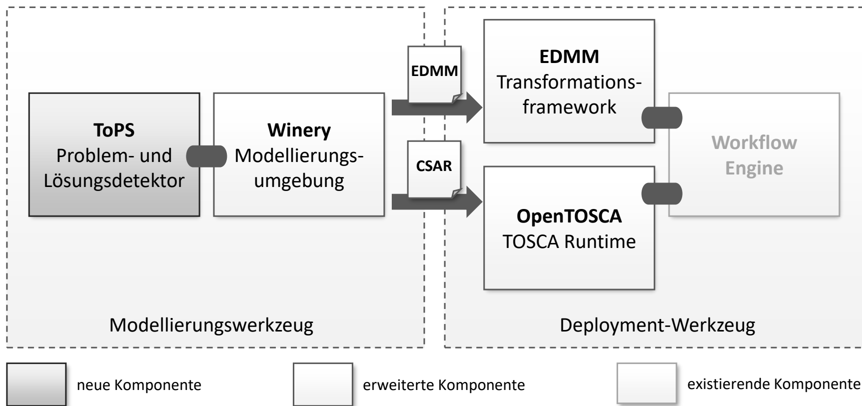

Abbildung 7.1: Grobarchitektur DIVA-Werkzeug.

7.1 Architektur des DIVA-Werkzeugs

Abbildung 7.1 zeigt die Grobarchitektur des DIVA-Werkzeugs. Das Werkzeug besteht aus verschiedenen Komponenten, die zur Realisierung der DIVA-Methode genutzt, erweitert, neu entwickelt und integriert wurden. Grundsätzlich besteht das Werkzeug aus zwei Teilen: (i) dem *Modellierungswerkzeug*, welches zur Entwurfszeit verwendet wird, und (ii) dem *Deployment-Werkzeug*, welches zur Deploymentzeit eingesetzt wird.

Das Modellierungswerkzeug besteht aus einer Modellierungsumgebung zur Spezifikation und Adaption von Deployment-Modellen sowie einer Komponente zur musterbasierten Problem- und Lösungserkennung. Für diese Arbeit wurde dafür die Modellierungsumgebung *Winery* [BEK+16; KBBL13] erweitert und die Komponente *Topology Problem and Solution Detector (ToPS)* entwickelt. Das Deployment-Werkzeug besteht aus zwei alternativen Komponenten abhängig davon, ob der TOSCA-Standard und dementsprechend eine TOSCA-Runtime, wie *OpenTOSCA* [BBH+13; BEK+16] oder andere deklarative Deployment-Technologien, bei Bedarf auch in Kombination, für die Bereitstellung verwendet werden sollen. Eine standardkonforme TOSCA-Runtime kann ein sogenanntes *Cloud Service Archive (CSAR)* verarbeiten [OAS13;

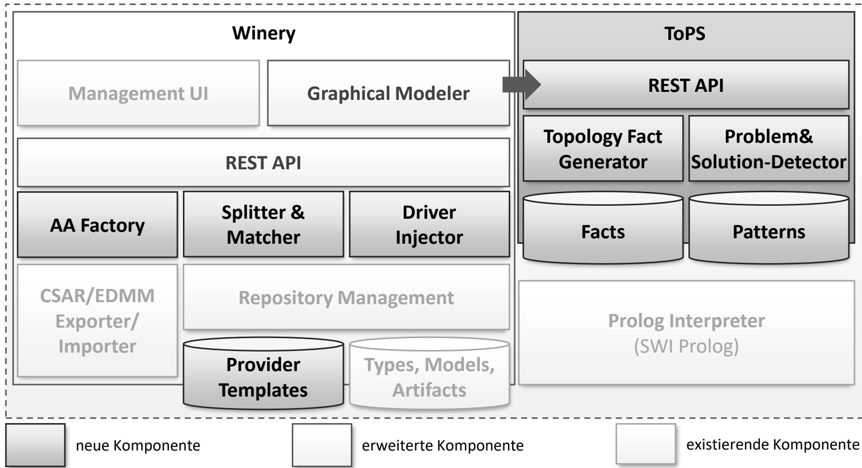

Abbildung 7.2: Architektur des Modellierungswerkzeugs.

OAS20]. Für die Verwendung unterschiedlicher deklarativer Deployment-Technologien wurde das *EDMM-Transformation-Framework* [WBB+19], wie bereits in Abschnitt 6.2.2 erläutert, erweitert, welches ein EDMM-konformes Deployment-Modell als Eingabe erwartet. Für die Ausführung generierter LDWs wird von beiden Komponenten eine Workflow-Engine verwendet, wie beispielsweise *Camunda* [Cam22] für BPMN.

7.1.1 Modellierungswerkzeug

Den größten Teil des DivA-Werkzeugs umfasst das Modellierungswerkzeug. Abbildung 7.2 zeigt die Architektur des Modellierungswerkzeugs. Die Modellierungsumgebung Winery wurde um Bausteine (i) zum Modellieren von Partnern und Zielumgebungen, (ii) zum Verteilen basierend auf der Split-and-Match-Methode und (iii) zur Treiberselektion erweitert. Zusätzlich wurde der musterbasierte Detektor ToPS entwickelt, welcher für die Problemerkennung und Lösungsselektion verwendet wird.

Die *Management UI* und der *Graphical Modeler* der Winery sind graphische Benutzerschnittstellen, die die Verwaltung der Modellelementtypen

und Deployment-Modelle sowie die graphische Modellierung der Deployment-Modelle als gerichtete Graphen ermöglichen. Neben der visuellen Modellierung bietet der Graphical Modeler auch Funktionen zur Verteilung, Analyse und Adaption eines Deployment-Modells. Dazu wird mit dem *REST API* der Winery sowie dem *REST API* des ToPS kommuniziert. Die *Splitter&Matcher*-Komponente implementiert die in Abschnitt 4.4.2 vorgestellte SPLIT-AND-MATCH-Methode. Basierend auf den im Deployment-Modell definierten Zielumgebungen und den verfügbaren Modellfragmenten im *Provider-Templates*-Repositorium wird das Deployment-Modell restrukturiert und passende Dienste werden selektiert und hinzugefügt. Die verfügbaren Dienste in einer Zielumgebung sind als unvollständige Deployment-Modelle im Repositorium verfügbar. Dazu werden erst die definierte Verteilung validiert (Algorithmus 4.2), anschließend ein Deployment-Modell erzeugt, welches die Verteilung strukturell widerspiegelt (Algorithmus 4.3) und anschließend die bestehenden Infrastruktur- und Middleware-Komponenten im Deployment-Modell durch kompatible Dienste der Zielumgebung ersetzt (Algorithmus 4.4). Die *Driver-Injection*-Komponente implementiert den Algorithmus aus Abschnitt 5.5 zur Selektion von passenden Kommunikationstreibern, welche von einer nachrichtenorientierten Middleware zur Verfügung gestellt werden (Algorithmus 5.1). Die abstrakten Artefakte im Deployment-Modell werden durch ein konkretes Artefakt, das von der Middleware bereitgestellt wird, ersetzt. Den Zugriff auf die verschiedenen Modellentitäten, Deployment-Modelle und Fragmente der Zielumgebungen ermöglicht die *Repository-Management*-Komponente.

Der Graphical Modeler nutzt außerdem ToPS, der die in Abschnitt 5.3 vorgestellte musterbasierte Problemerkennung sowie die Identifikation geeigneter Lösungen, wie in Abschnitt 5.4 vorgestellt, implementiert. ToPS wurde im Rahmen dieser Arbeit entwickelt und besteht hauptsächlich aus zwei Komponenten: *Topology Fact Generator* und *Problem&Solution-Detector*. Zur Analyse eines Deployment-Modells transformiert der Topology Fact Generator das Deployment-Modell in entsprechende Prolog-Fakten, die im *Facts-Repositorium* abgelegt werden. Sie bilden die Wissensbasis, auf der die Gültigkeit der als Prolog-Regeln formalisierten Muster und DKs, die im *Patterns-Repositorium*

abgelegt werden, geprüft wird. Die Problem&Solution-Detector Komponente nutzt den *Prolog Interpreter* zur Prüfung der Regeln. Die Menge der erkannten Probleme bzw. der passenden DKs mit einer Referenz auf den AA wird dann zurückgegeben. Die AAs sind nicht Teil von ToPS, sondern können von einem externen Dienst oder der Winery bereitgestellt werden. Wird der Algorithmus von der Winery bereitgestellt, löst die *AA Factory* den angeforderten Algorithmus auf und wendet ihn auf das Deployment-Modell an.

Für das spätere Deployment durch eine TOSCA-Runtime bzw. die weitere Verarbeitung durch das EDMM-Transformation-Framework ermöglicht die Winery den Export eines Deployment-Modells als CSAR oder EDMM durch den entsprechenden *Exporter*. Ein CSAR kann von jeder TOSCA-konformen Runtime verarbeitet werden und die modellierte Anwendung kann entsprechend bereitgestellt werden. Für die Bereitstellung mit anderen Technologien muss das EDMM erst in die Artefakte der jeweiligen Technologie transformiert werden.

7.1.2 Deployment-Werkzeug

Für die Ausführung des Deployments wurde das EDMM-Transformation-Framework erweitert. Abbildung 7.3 zeigt die erweiterte Architektur. Die Nutzung einer TOSCA-Runtime wird hier nicht detailliert diskutiert, da der Fokus dieser Arbeit auf der Integration des DIVA-Werkzeugs mit unterschiedlichen Deployment-Technologien bei der Ausführung des Deployments liegt, um den Einsatz der Konzepte unabhängig von den bereits genutzten Deployment-Technologien in einer Organisation zu ermöglichen. Die Integration mit einer TOSCA-Runtime wird in Abschnitt 7.4 kurz erläutert.

Das EDMM-Transformation-Framework bietet den Nutzern ein *Command Line Interface (CLI)* sowie eine einfache graphische Benutzeroberfläche zum Transformieren und Deployment. Über das *REST API* werden die Funktionalitäten bereitgestellt und können damit auch unabhängig von den Benutzerschnittstellen mit anderen Systemen integriert werden. Der *Model Parser* und der *Model Divider* sind die Kernkomponenten zur Verarbeitung eines EDMM-Deployment-Modells. Der Model Parser übersetzt das Modell in eine

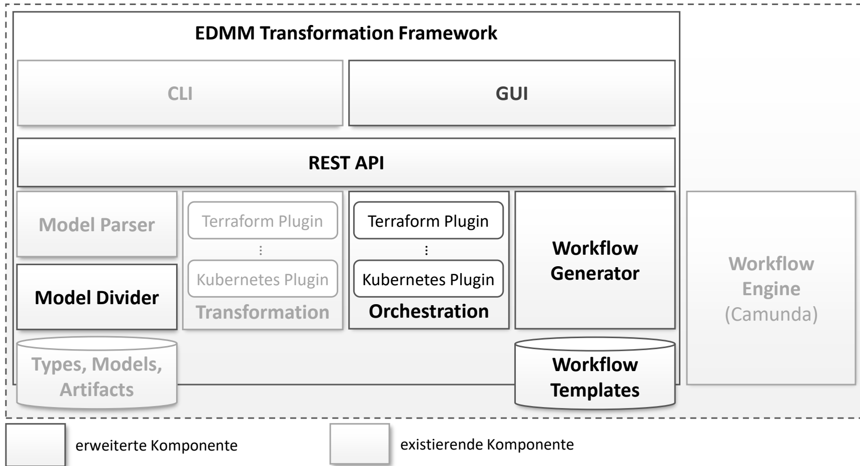

Abbildung 7.3: Architektur des Deployment-Werkzeugs (erweitert von Wurster et al. [WBB+21]).

interne graphbasierte Datenstruktur. Auf dieser Datenstruktur berechnet der Model Divider den PRG mit den verschiedenen Deployment-Gruppen. Dabei werden bei partnerübergreifenden Anwendungen ausschließlich für die partnerspezifischen Komponenten Deployment-Gruppen spezifiziert und die Komponenten anderer Partner als externe Abhängigkeiten markiert. Sie müssen jedoch trotzdem im PRG berücksichtigt werden. Für jede Deployment-Gruppe wird abhängig von der annotierten Technologiegruppe das entsprechende *Transformation-Plugin* ausgeführt, welches aus dem EDMM-Modellfragment das DTSM generiert. Der Plugin-Mechanismus ermöglicht die Erweiterung um weitere Technologien, soweit sie mit EDMM kompatibel sind. Jedes Plugin umfasst die Logik mit spezifischen Transformationsregeln, um die technologiespezifischen Dateien und Artefakte zu generieren.

Die *Orchestration*-Komponente berechnet die Provisionierungsreihenfolge und stellt die verschiedenen *Orchestration-Plugins* bereit. Ein Orchestration-Plugin kapselt die Interaktion mit der entsprechenden Deployment-Technologie und nutzt dafür, zum Beispiel, verfügbare SDKs oder CLIs. Jedes

Plugin ermittelt die benötigten Eingabeparameter und kann ein DTSM mit gegebenen Eingabeparametern ausführen und nach erfolgreicher Ausführung Laufzeitinformationen, wie beispielsweise IP-Adressen, abrufen. Alle Instanzinformationen werden zur Ausführungszeit gespeichert und können abgerufen werden, um andere DTSMs zu verarbeiten oder diese mit anderen Partnern zu teilen. Um die Orchestrierung der verschiedenen Deployment-Aktivitäten über die LDWs zu ermöglichen, werden die Schnittstellen der Orchestration-Plugins über das REST API bereitgestellt.

Der *Workflow Generator* wurde erweitert, um nicht nur Workflows für die zentralisierte Ausführung des Deployments zu ermöglichen. Der Workflow Generator erzeugt basierend auf der berechneten Provisionierungsreihenfolge mithilfe von Vorlagen aus dem *Workflow Templates*-Repositorium den LDW. Für die Generierung stehen Templates für vier generische Aktivitätsarten zur Verfügung: die Initiierung, das Deployment mit einer Technologie und das Senden und Empfangen von Nachrichten. Die Templates zur Initiierung sowie zum Senden und Empfangen wurden speziell für partnerübergreifende Anwendungen hinzugefügt. Für die Ausführung der generierten Workflows wird eine Workflow-Engine genutzt. Bei der Verwendung standardisierter Modellierungssprachen wie BPEL oder BPMN können beliebige standardkonforme Workflow-Engines genutzt werden. Bei der prototypischen Realisierung des Werkzeugs wurde BPMN verwendet und Camunda als Workflow-Engine eingesetzt.

7.1.3 Architektur und Verarbeitungsschritte mit zwei Partnern

Für die Bereitstellung partnerübergreifender Anwendungen und zur Anwendung der in Kapitel 3 vorgestellten DIVA-Methode benötigt jeder Partner eine Instanz des in den vorherigen Abschnitten erläuterten DIVA-Werkzeugs. In Abbildung 7.4 sind die Peer-to-Peer-Architektur und die Verarbeitungsschritte mit zwei Partnern skizziert. Das GDM wird zentral modelliert und allen Partnern bereitgestellt. Das GDM kann das Ergebnis gemeinsamer Workshops der Partner sein und aus der Architektur der partnerübergreifenden Anwendung abgeleitet werden. Jedem Partner steht das Modellierungswerk-

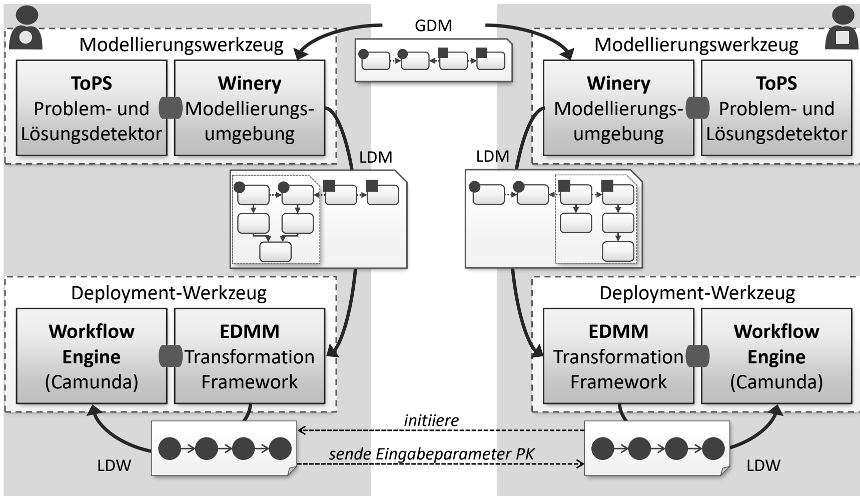

Abbildung 7.4: Architektur und Verarbeitungsschritte mit zwei Partnern.

zeug zur weiteren Verteilung und Adaption des LDMs zur Verfügung. Das LDM dient anschließend als Eingabe für das Deployment-Werkzeug, welches auf Basis des LDMs den LDW generiert.

Über die GUI des EDMM-Transformation-Framework kann einer der Partner die Instanziierung der Anwendung initiieren. Durch die Initiierung wird eine Instanz des LDWs in der Workflow-Engine erstellt und ausgeführt. Die Initiierung der übrigen Partner erfolgt über eine dedizierte Initiierungsnachricht. Die einzelnen Aktivitäten eines Deployment-Workflows sind nur dem jeweiligen Partner bekannt, ausschließlich die Kommunikationsendpunkte zum Nachrichtenaustausch sowie das Nachrichtenformat für die Initiierung sowie zur Übermittlung von Eingabeparametern für PKs müssen global bekannt sein. Um sicherzustellen, dass jeder LDW nur einmal für eine Instanz der Anwendung ausgeführt wird, wird eine Korrelation-ID generiert, die eindeutig für die gesamte Choreographie ist.

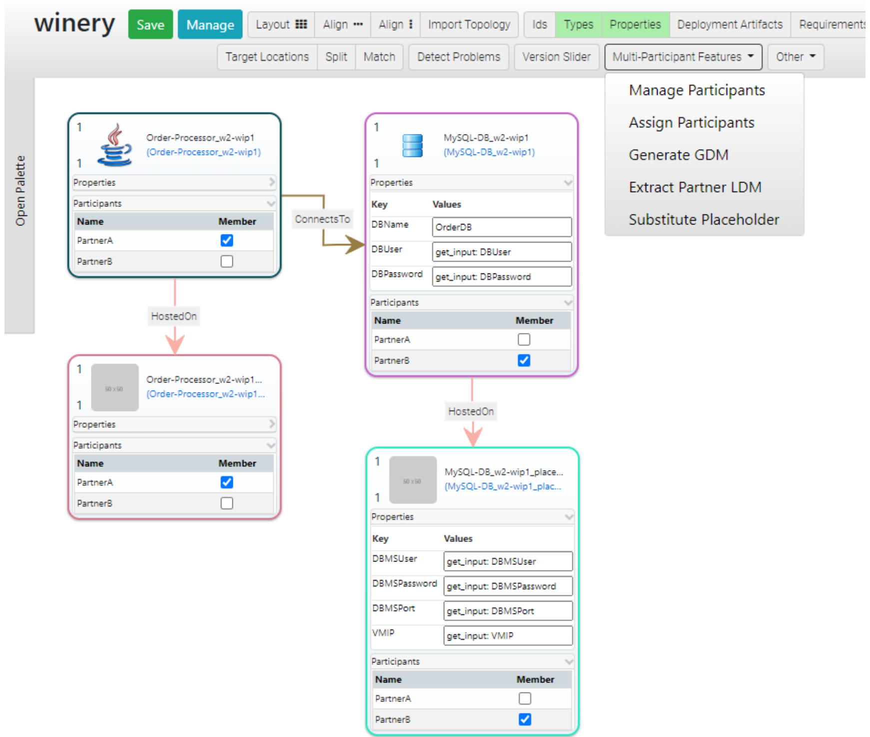

Abbildung 7.5: *Graphical Modeler* der Winery mit GDM.

7.2 Prototypische Implementierung des DIVA-Werkzeugs

Wie bereits erläutert basiert das DIVA-Werkzeug auf der Erweiterung und Integration existierender Modellierungs- und Deployment-Werkzeuge.

7.2.1 Modellierungswerkzeug

Winery[1] ist ein Open-Source Modellierungstool unter der Apache 2.0 oder EPL-2.0 Lizenz. Für die Benutzeroberflächen wird das TypeScript-basierte

[1] https://github.com/eclipse/winery

Framework für Webanwendungen Angular[1] verwendet. Über HTTP wird mit dem REST-basierten API der Winery kommuniziert. Serverseitig ist die Winery ein Java-basierter Web-Service und die API wurden mit JAX-RS realisiert. Für die Speicherung der Dateien in den Repositorien wird das Dateisystem des Servers genutzt. Die Deployment-Modelle werden standardmäßig in XML serialisiert und im Dateisystem des Servers gespeichert. Dabei wird das TOSCA-Metamodell verwendet. Die Deployment-Modelle können jedoch auch als EDMM-Modelle in YAML exportiert werden. Dafür wird das Modell in die Syntax von EDMM transformiert. Zum Datenaustausch zwischen GUI und Server wird JSON verwendet.

Auch mit dem REST-basierten API von ToPS wird über HTTP kommuniziert. ToPS ist ebenfalls ein Java-basierter Web-Service und nutzt das Spring Framework für die Realisierung der API. ToPS steht unter der Apache 2.0 Lizenz zur Verfügung. Zum Parsen eines Deployment-Modells nutzt ToPS den WineryClient. Für die Ausführung der Prolog-Regeln der Muster und DKs wird SWI Prolog[2] genutzt. Zur Anbindung an die Java-Anwendung wird die Schnittstelle JPL[3] genutzt. Dadurch wird die Ausführung und Manipulation von Prolog-Abfragen aus der Java-Anwendung ermöglicht. Die Fakten und die Muster sind innerhalb der Anwendung auf dem Dateisystem des Servers abgelegt.

7.2.1.1 Modellierung und Verteilung von Deployment-Modellen

Das graphische Modellierungswerkzeug der Winery ist in Abbildung 7.5 abgebildet. Es ermöglicht die graphbasierte Modellierung von Deployment-Modellen. Über die *Palette* (links) können Komponenten via Drag-and-Drop hinzugefügt werden und Relationen zwischen den Komponenten gezogen werden. Diese Grundfunktionalitäten der Winery werden genutzt, um GDMs zu modellieren. In dem abgebildeten simplifizierten Beispiel ist ein GDM für eine Anwendung abgebildet, die aus zwei anwendungsspezifischen Kompo-

[1] `https://angular.io/`
[2] `https://www.swi-prolog.org/`
[3] `https://jpl7.org/index`

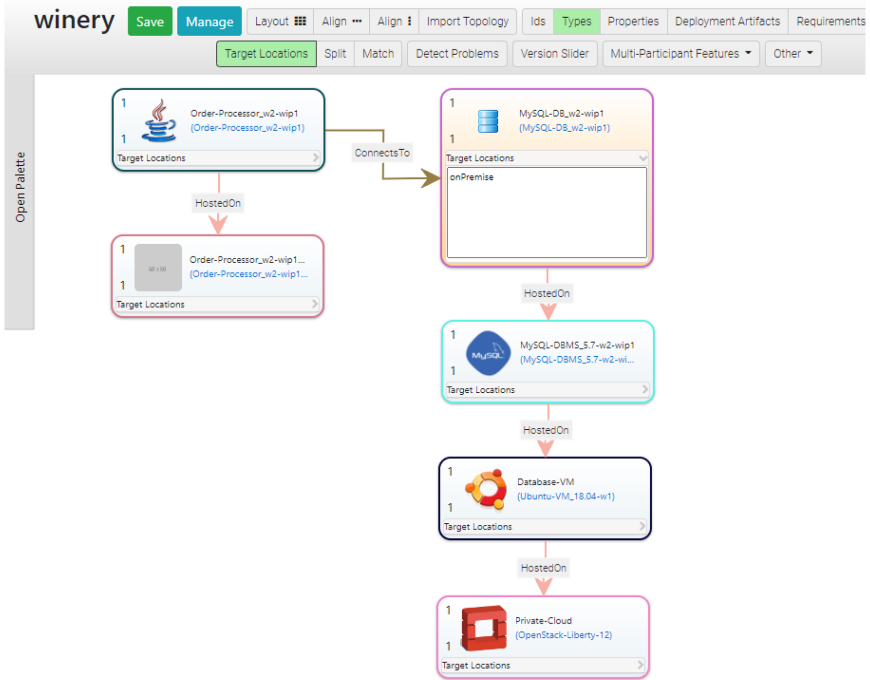

Abbildung 7.6: LDM mit definierter Zielumgebung.

nenten besteht. Für die Datenbank-Komponente sind Eigenschaften definiert, die zur Konfiguration dienen.

Für die Implementierung der DIVA-Methode wurden zusätzliche *Multi-Participant-Features* (rechts oben) zur Verwaltung der Partner, Generierung der Platzhalter-Komponenten, Extraktion der LDMs für die beteiligten Partner und zum Ersetzen der Platzhalter durch existierende Modellfragmente integriert. Für die Komponenten des in Abbildung 7.5 dargestellten GDMs wurden Partner hinzugefügt und die Platzhalter-Komponenten generiert. Der Platzhalter für die Hosting-Komponente der Datenbank-Komponente definiert weitere Eigenschaften, die die Operation, welche die „Connects-To“-Relation implementiert, als Eingabeparameter benötigt. Diese müssen

durch die konkreten Middleware- und Infrastrukturkomponenten, die durch das Ersetzen und Restrukturieren hinzugefügt werden, definiert werden.

Zur ressourcenspezifischen Verteilung eines LDMs können Zielumgebungen (*Target Locations*) für die Komponenten definiert werden, wie in Abbildung 7.6 gezeigt. Durch die definierten Zielumgebungen werden die Modellfragmente in einem bestimmten Namespace adressiert. Für das abgebildete Beispiel werden alle Modellfragmente in einem spezifischen Namespace, hier `http://www.opentosca.org/providers/onpremise`, auf ihre Kompatibilität geprüft. Besteht der lokale Teil des LDMs aus mehr als einem Stack kann vorher noch die *Split*-Funktion zum Aufteilen entsprechend den definierten Zielumgebungen genutzt werden, bevor mit der *Match*-Funktion passende Hosting-Komponenten ermittelt werden.

7.2.1.2 Musterrepositorium und musterbasierte Analyse

Für die Realisierung der Problemerkennung und der Identifikation von passenden Lösungen werden die in Abschnitt 5.3.2 und Abschnitt 5.4.1 eingeführten logischen Formeln als Prolog-Regeln definiert. Diese Prolog-Regeln, vor allem für die Problemerkennung, sind eine Ergänzung zur eigentlich textuellen Musterbeschreibung. Um eine Integration der Regeln in die textuellen Beschreibungen zu ermöglichen und den Zugang für Musterautoren zu erleichtern, werden die Musterbeschreibungen in Markdown-Dateien gespeichert. Damit ist später auch eine Integration mit anderen Musterrepositorien wie dem *PatternAtlas* [LB21] möglich.

In Abbildung 7.7 ist ein Ausschnitt aus der Musterbeschreibung des SECURE-CHANNEL-Musters mit der entsprechenden Prolog-Regel zur Erkennung des Vorhandenseins eines Problems, definiert nach diesem Muster, abgebildet. Die abgebildete Regel entspricht der in Abschnitt 5.3.2 eingeführten logischen Formel. In Abbildung 7.8 ist analog dazu die Beschreibung der Lösungsimplementierungen für „TLS für HTTP“, wenn die VMs, auf denen die kommunizierenden Komponenten bereitgestellt sind, konfigurierbar sind, abgebildet. Für Lösungsimplementierungen gibt es anders als für die Musterbeschreibungen keine definierten Musterformate. Das hier verwendete

26 lines (21 sloc) | 832 Bytes   Raw Blame

Secure Channel

Problem

Insecure Communication

Problem Rule

```
insecure_public_communication(Component_1, Component_2) :-
        property(Relation_ID, sensitivedata, true),
        relation(Relation_ID, Component_1, Component_2),
        relation_types(RT),
        member(Relation_ID, RT),
        member(connectsto, RT),
        components_in_different_locations(Component_1, Component_2),
        not(property(Relation_ID, encrypted, true)).
```

Solution Description

Create secure channels for sensitive data that obscure the data in transit. Exchange information between client and server to allow them to set up encrypted communication between themselves. [...]

Reference

M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, Inc., 2006.

Abbildung 7.7: Ausschnitt aus Musterbeschreibung mit Prolog-Regel.

Format ist angelehnt an die Definition von [FBB+14b]. Die Beschreibung einer Lösungsimplementierung umfasst eine textuelle Beschreibung der generellen Lösung, eine Referenz auf die Muster, für die sie eine Lösung repräsentiert, den als Prolog-Regel definierten DK, der das Selektionskriterium darstellt, und eine Referenz auf die eigentliche Lösungsimplementierung, in diesem Fall einen AA. Die Prolog-Regel zur automatisierten Erkennung des DK entspricht der logischen Formel aus Abschnitt 5.4.1. Da die Menge an

25 lines (18 sloc) | 640 Bytes

Raw | Blame

TLS for HTTP on VM

Description

The TLS for HTTP on VM solution inserts proxies for the communication between the two communicating components. The proxies enable a secure communication.

Related Patterns

- Secure Channel

Selection Criteria

```
tlsonvm(Component_1, Component_2) :-
    host(Component_1, Host_1, vm),
    host(Component_2, Host_2, vm),
    relation(Relation_ID, Component_1, Component_2),
    relation_types(RT),
    member(Relation_ID, RT),
    member(httpconnectsto, RT).
```

Concrete Solution Implementation

secureVmProxy

Abbildung 7.8: Ausschnitt aus Lösungsbeschreibung mit Prolog-Regel.

Lösungsbeschreibungen dynamisch wachsen kann, verweist die Lösungsbeschreibung auf das entsprechende Muster, jedoch nicht das Muster auf die Menge an Lösungsbeschreibungen.

Im Graphical Modeler der Winery kann die Erkennung der Probleme angestoßen werden. Alle erkannten Probleme und die betroffenen Komponenten werden angezeigt und zur Lösung eines der Probleme kann aus allen anwendbaren Lösungsimplementierungen gewählt werden. Entsprechend der beschriebenen Lösung wird das Deployment-Modell adaptiert, wobei

ausschließlich strukturelle Adaptionen sowie Änderungen der Konfiguration vorgenommen werden können. Die Erkennung von Problemen und die Anwendung von Lösungen erfolgen sequenziell.

7.2.2 Deployment-Werkzeug

Das EDMM-Transformation-Framework[1] steht unter der Apache 2.0 Lizenz als Open-Source Software zur Verfügung. Sowohl die CLI als auch die simplifizierte GUI können YAML-Dateien entsprechend der EDMM YAML Spezifikation[2] mit den Erweiterungen für partnerübergreifende Anwendungen als Eingabe verarbeiten. Für die GUI wurde das JavaScript-Framework React[3] mit Material-UI-Komponenten[4] verwendet. Serverseitig ist das EDMM-Transformation-Framework eine Java-basierte Anwendung und nutzt das Spring-Framework. Das interne Datenformat ist eine graphbasierte Darstellung eines EDMM-Modells. Dafür wird die Java-Bibliothek JGraphT[5] verwendet.

Der Workflow-Generator nutzt zur Generierung der Workflows die Apache FreeMarker Template Engine[6]. BPMN-Workflow-Fragmente stehen als Template zur Verfügung und werden mit den Parametern des jeweiligen EDMM-Deployment-Modells gefüllt und die Aktivitäten entsprechend der Provisonierungsreihenfolge zu einem vollständigen Workflow zusammengefügt. Zur Ausführung wird die Camunda Workflow-Engine verwendet. Über die GUI kann die Instanziierung initiiert werden.

[1] `https://github.com/UST-EDMM`
[2] `https://github.com/UST-EDMM/spec-yaml`
[3] `https://reactjs.org/`
[4] `https://material-ui.com/`
[5] `https://jgrapht.org/`
[6] `https://freemarker.apache.org/`

7.3 Validierung und Evaluation

Für die Validierung und Evaluation der DIVA-Methode wurde der im vorherigen Abschnitt vorgestellte Prototyp genutzt. Es wurden weitere Muster zur Problemerkennung und Modelladaption formalisiert (Abschnitt 7.3.1) sowie die Generierung von Workflows mit BPMN umgesetzt (Abschnitt 7.3.2).

7.3.1 Anwendungsszenarien zur Problemerkennung und Modelladaption

Zusätzlich zu den bereits in Abschnitt 5.3.2 eingeführten SECURE-CHANNEL-Muster und der „TLS für HTTP“-Lösungsimplementierung in Abschnitt 5.4.1 werden noch weitere Problembeschreibungen von Mustern und DKs zur Detektion von passenden Lösungsimplementierungen formalisiert, um die Anwendbarkeit der Methode zu validieren.

7.3.1.1 Anwendungsszenarien zur Musterformalisierung

Um die Anwendbarkeit des Formalisierungskonzepts von den Problem- und Kontextbeschreibungen von Mustern zu demonstrieren, wurden weitere drei Muster aus der Cloud-Mustersprache [FLR+14] formalisiert. Alle drei Muster sind Teil der Gruppe der Cloud-Integrations-Muster. Dazu wird im Folgenden ein Ausschnitt der jeweiligen Musterbeschreibung und die dazugehörige abgeleitete Prolog-Regel dargestellt.

Für die Implementierung wird die logische Programmiersprache Prolog verwendet. Wie bereits diskutiert wird dabei die Annahme einer geschlossenen Welt zugrunde gelegt. Zusätzlich wird die schwache Negation (*engl. „Negation as Failure“*) angenommen, das heißt: kann ein Ausdruck durch Resolution nicht bewiesen werden, dann wird davon ausgegangen, dass er nicht wahr ist. Basierend auf diesen Annahmen können alle logischen Formeln aus den vorherigen Abschnitten mittels Prolog ausgedrückt werden. In Prolog steht „:-“ für die Implikation, das logische UND „$\wedge$“ wird als „,“ ausgedrückt, das logische ODER „$\vee$“ als „;“ und jede Regel endet mit „.“. Prädikate, Funktionen und Konstanten beginnen mit Kleinbuchstaben, während Variablen mit einem Großbuchstaben beginnen.

(i) Application-Component-Proxy-Muster: Die folgende Textpassage ist ein Ausschnitt aus der Musterbeschreibung in Fehling et al. [FLR+14]:

Problem:

> „How can an application component be accessed if direct access to its hosting environment is restricted?"

Context:

> „Application components of a distributed application are deployed in different cloud environments that form a hybrid cloud. These environments often have different privacy, security, and trust properties. [...] However, application components hosted in unrestricted environments, for example, a public cloud, may have to access application components hosted in a restricted environment, for example a private cloud or corporate data center, but direct access may be unavailable. [...]"

Aus der Musterbeschreibung lassen sich die Kriterien für das Vorliegen eines Problems wie folgt ableiten: Die Komponenten kommunizieren direkt miteinander, zum Beispiel über HTTP, die Komponenten werden in unterschiedlichen Umgebungen bereitgestellt und die Komponente, zu der eine Verbindung hergestellt werden soll, ist in einer Umgebung bereitgestellt, die keinen Zugriff von außen ermöglicht. Daraus kann die folgende Prolog-Regel definiert werden:

```
directAccessToRestrictedEnvironment(C1, C2) :-
  direct_communication(C1, C2),
  components_in_different_locations(C1, C2),
  restricted_environment(C2).
```

Das Prädikat `components_in_different_locations(C1, C2)` wurde bereits in Abschnitt 5.3.2 erläutert. Zusätzlich wurden zwei zweitere Regeln eingeführt, welche die Definition von Regeln zur Erkennung von Problemen vereinfachen. Eine direkte Kommunikation zwischen zwei Komponenten findet statt, wenn die Komponenten mit einer entsprechenden Relation verbunden sind und die Zielkomponente nicht vom Typ „Channel" bzw. einem davon erbenden Typ ist. Zur Vereinfachung der Formulierung, ob eine Relation

oder eine Komponente direkt oder transitiv von einem bestimmten Typ erbt, wurden die Prädikate `relation_types(RT)` und `component_types(RT)` eingeführt, die jeweils eine Liste umfassen, die die Relation-ID bzw. Komponenten-ID sowie alle Typ-IDs enthält. `member` ist ein vordefiniertes Prädikat in Prolog zur Prüfung, ob ein Objekt in einer Liste enthalten ist. Die entsprechende Regel in Prolog sieht wie folgt aus:

```
direct_communication(C1, C2) :-
  relation(Relation_ID, C1, C2),
  relation_types(RT),
  member(Relation_ID, RT),
  member(connectsto, RT),
  component_types(CT),
  member(C2, CT),
  not(member(channel, CT)).
```

Eine weitere Regel besagt, ob eine Komponente in einer Umgebung bereitgestellt ist, die keine eingehende Kommunikation ermöglicht. Dieses Wissen ist als Eigenschaft an eine Komponente im Stack annotiert. Damit sieht die Regel in Prolog wie folgt aus:

```
restricted_environment(C1) :-
  property(P, inboundcommunication, false),
  hosting_stack(S),
  member(C1, S),
  member(P, S).
```

(ii) MESSAGE MOVER-Muster: Die folgende Textpassage ist ein Ausschnitt aus der Musterbeschreibung in Fehling et al. [FLR+14]:

Problem:

> „How can message queues of different providers be integrated without an impact on the application component using them?“

Context:

> „The application components comprising a distributed application (160)

> often exchange data using messaging. These messages are stored in message queues. [...] If these queues reside in different cloud environments that form a hybrid cloud (75) accessibility to queues of one environment may be restricted for application components that are deployed in another environment. [...] Therefore, each of the application components shall access a message queue hosted in the cloud environment where the application component itself is hosted. [...]“

Dieses Muster adressiert die nachrichtenbasierte Kommunikation von zwei Komponenten, von denen eine in einer Umgebung bereitgestellt ist, auf die von außen nicht zugriffen werden kann. In diesem Muster geht es konkret um den Zugriff auf eine *Queue*. Basierend auf dem Wissen aus den Integrationsmustern [HW04] stellt dies jedoch eine Einschränkung der nachrichtenbasierten Kommunikation dar. Um alle nachrichtenbasierten Kommunikationstypen einzubeziehen wird von einem generischen „Channel“ ausgegangen. Damit kann mit folgender Regel das Problem erkannt werden:

```
distributed_messaging(C1, C2) :-
  messaging_communication(Channel, C1, C2),
  components_in_different_locations(C1, C2),
  restricted_environment(C1),
  not(restricted_environment(C2)).
```

Für diese Regel wurde eine weitere Regel eingeführt, die auch in anderen Formalisierungen zur Problemerkennung genutzt wird. Mit der Regel `messaging_communication(ChannelComp, C1, C2)` wird geprüft, ob eine nachrichtenbasierte Kommunikation zwischen zwei Komponenten stattfindet. Dafür muss für beide Komponenten jeweils eine Relation zu einer Komponenten vom Typ „Channel“ bzw. einem davon erbenden Typ, hier als Variable „ChannelKomp“ definiert, bestehen. Diese Relationen müssen vom Typ „ConnectsTo“ oder einem spezialisierten Typ sein. Damit kann die Hilfsregel wie folgt definiert sein:

```
messaging_communication(ChannelComp, C1, C2) :-
  component_types(CT),
  member(ChannelComp, CT),
  member(channel, CT),
  relation(R1, C1, ChannelComp),
  relation(R2, C2, ChannelComp),
  relation_types(RT1),
  member(R1, RT1),
  member(connectsto, RT1),
  relation_types(RT2),
  member(R2, RT2),
  member(connectsto, RT2).
```

(iii) INTEGRATION PROVIDER-Muster: Die folgende Textpassage ist ein Ausschnitt aus der Musterbeschreibung in Fehling et al. [FLR+14]:

Problem:

> „How can components in different environments be integrated through a 3rd-party provider?“

Context:

> „When companies collaborate or one company has to integrate applications of different regional offices, different applications or the components of a distributed application are distributed among different hosting environments. Communication between these environments may be restricted. Especially, hosting environments may restrict any incoming communication initiated from the outside. Communication leaving the restricted environments is, however, often allowed. Therefore, additional integration components are required that have to be accessible from restricted environments. [...]“

Im Gegensatz zum vorherigen Muster löst dieses Muster das Problem, dass beide Komponenten in Umgebungen bereitgestellt sind, die keine eingehende Kommunikation ermöglichen. Dabei ist es unabhängig davon, ob die Komponenten nachrichtenbasiert oder direkt kommunizieren. Die entsprechende

Prolog-Regel ist wie folgt definiert:

```
integration_of_restricted_environments(C1, C2) :-
  components_in_different_locations(C1, C2),
  component_in_restricted_environment(C1),
  component_in_restricted_environment(C2),
  ((messaging_communication(Channel, C1, C2),
  component_in_restricted_environment(Channel));
  direct_communication(C1, C2)).
```

Da der Schwerpunkt der DivA-Methode auf den Kommunikationsbeziehungen zwischen den Komponenten liegt, wurden zur Validierung Muster, die die Kommunikation bzw. Integration von Anwendungskomponenten adressieren, betrachtet. Die in dieser Arbeit vorgestellten Muster stehen zur Problemerkennung in ToPS zur Verfügung. Weitere Muster können als Markdown-Dateien hinzugefügt werden und damit die Wissensbasis erweitert werden. Die Wissensbasis beschränkt sich ausschließlich auf Regeln zur Identifikation von Problemen in Deployment-Modellen, die entsprechend dem erweiterten EDMM spezifiziert sind.

7.3.1.2 Anwendungsszenarien zur Lösungsimplementierungen

Für das Secure-Channel-Muster werden im Folgenden zwei weitere Lösungen, zu der bereits in Abschnitt 5.4.1 vorgestellten und in Abbildung 7.8 abgebildeten Lösung, eingeführt und es wird dabei auf die Formalisierung des Deployment-Kontexts eingegangen. Analog zu den bereits vorgestellten Mustern können weitere Lösungen als Markdown-Dateien zu ToPS hinzugefügt werden und damit die Wissensbasis bezüglich bekannter Lösungen für erkannte Probleme erweitert werden.

(i) TLS-For-HTTP-on-Docker-Deployment-Kontext: Dies ist eine weitere Lösungsimplementierung, die dieselbe technische Lösung wie die bereits vorgestellte Lösungimplementierung realisiert, nämlich Proxies zur Verschlüsselung der HTTP-basierten Kommunikation. Jedoch besteht hier ein anderer

Deployment-Kontext. Die Anwendungskomponenten werden in Docker Containern bereitgestellt und die virtuellen Maschinen sind im Deployment-Modell nicht verwaltbar. Damit müssen die Proxies als Sidecar-Container bereitgestellt werden und dies bedarf einer anderen Modelladaption als das Hinzufügen von Proxies, die auf den VMs der Anwendungskomponenten bereitgestellt werden. Der formalisierte DK ist wie folgt definiert:

```
tlsondocker(C1, C2) :-
  host(C1, Host_1, dockerengine),
  host(C2, Host_2, dockerengine),
  not(host(C1, Host_1, vm)),
  not(host(C2, Host_2, vm)),
  relation(Relation_ID, C1, C2)
  relation_types(RT),
  member(Relation_ID, RT),
  member(httpconnectsto, RT).
```

Das Prädikat `host/3` wurde bereits in Abschnitt 5.4.1 eingeführt und ist wahr, wenn im Hosting-Stack der Komponente eine Komponente von dem entsprechenden Komponententyp vorhanden ist. Damit bestehen zwei unterschiedliche DKs mit unterschiedlichen AAs, die dieselbe technische Lösung in einem Deployment-Modell realisieren, dies jedoch auf unterschiedliche Art und Weise. Dies zeigt die Notwendigkeit, den genauen technischen Deployment-Kontext bei der Selektion von Lösungen zu berücksichtigen.

(ii) IPSec-Deployment-Kontext: Mit dem IPSec-Protokoll lässt sich ein virtuelles privates Netzwerk (VPN) zwischen zwei Maschinen aufbauen. Dazu müssen die virtuellen Maschinen, auf denen die Anwendungskomponenten bereitgestellt werden, entsprechend konfiguriert sein. Diese Lösung ist damit anwendbar, wenn die VMs im Deployment-Modell konfigurierbar sind. Das Selektionskriterium, das heißt der DK, kann damit wie folgt definiert werden:

```
ipsec(C1, C2) :-
  host(C1, Host_1, vm),
  host(C2, Host_2, vm).
```

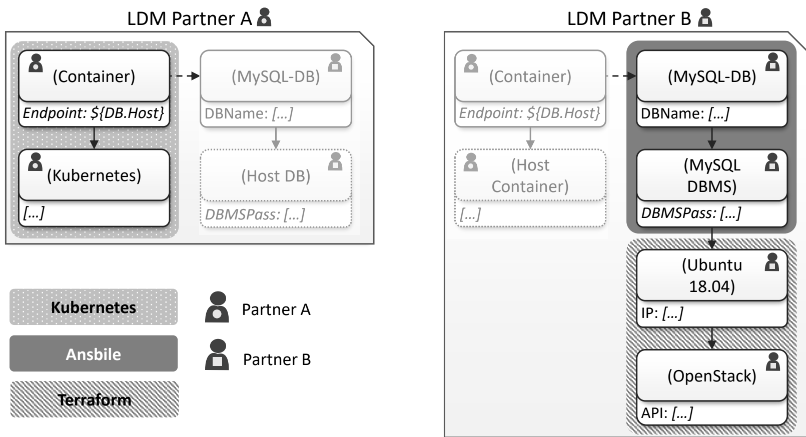

Abbildung 7.9: Anwendungsszenario zur Workflow-Generierung mit zwei Partnern und drei Technologien.

Basierend auf dieser einfachen Regel kann die Anwendbarkeit der Lösung ermittelt werden und die identifizierten VMs können im positiven Fall durch entsprechend konfigurierte VMs ausgetauscht werden. Insgesamt sind alle Modelladaptionen auf strukturelle Änderungen und Konfigurationen der Komponenten im Deployment-Modell beschränkt. Dabei bestimmt der DK, auf welchen Komponenten der Adaptionsalogrithmus operieren kann.

7.3.2 Workflow-Generierung mit BPMN

Das bereits in Abbildung 7.5 und Abbildung 7.6 dargestellte Beispiel wird für das Deployment mit zwei Partnern und drei unterschiedlichen Technologien verwendet. In Abbildung 7.9 sind die zwei resultierenden LDMs der zwei Partner, Partner A und Partner B, abstrahiert dargestellt. Die Order-Processor-Komponente wird als Container bereitgestellt. Jeder Partner kann für seine Komponenten die gewünschte Deployment-Technologie selektieren. Die Transformation in verschiedene Technologien wird bereits durch das EDMM-Transformation-Framework unterstützt [WBB+21]. Partner A nutzt Kubernetes und Partner B für die VM Terraform und für die Konfi-

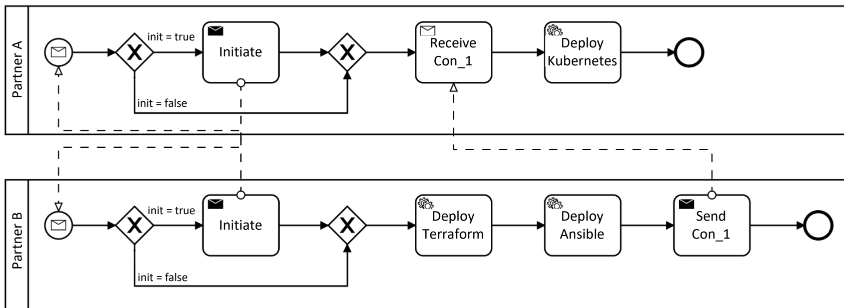

Abbildung 7.10: BPMN Kollaborationsdiagramm mit den generierten Workflows basierend auf den LDMs in Abbildung 7.9.

guration der VM und Installation der Datenbank Ansible. Mit der Notation `${<Komponente>.<Eigenschaft>}` können Eigenschaften oder Instanzparameter, wie die IP-Adresse und Port-Nummer, die als „Host"-Parameter vom EDMM-Transformation-Framework bereitgestellt werden, von anderen Komponenten referenziert werden.

Für die Komponenten des jeweiligen Partners und die annotierten Technologiegruppen werden die Deployment-Gruppen bestimmt und die Modellfragmente in die Artefakte und Dateien der jeweiligen Technologie transformiert. Auf Basis der Provionsierungsreihenfolge wird anschließend jeweils ein LDW generiert. In Abbildung 7.10 sind die zwei resultierenden Workflows in einem BPMN-Kollaborationsdiagramm dargestellt. Jeder Workflow wird durch eine Nachricht instanziiert. Kommt die Nachricht vom Partner selbst (init = true), dann werden an alle übrigen Partner Initiierungsnachrichten geschickt, um das gesamte Deployment anzustoßen. Kommt die Initiierungsnachricht von einem anderen Partner (init = false), dann werden keine weiteren Initiierungsnachrichten verschickt. Dies wird basierend auf einem Wahrheitswert in der Initiierungsnachricht, welcher angibt ob der Partner selbst der Initiator ist oder nicht, ausgedrückt. Anschließend wird das Deployment der Deployment-Gruppen entsprechend ihrer Reihenfolge im PRG durchgeführt. Partner B muss erst die VM mittels Terraform bereitstel-

len bevor mit Ansible die übrigen Komponenten installiert werden können. Partner A muss erst auf die Nachricht mit den Endpoint-Informationen von Partner B warten, damit das Deployment mit Kubernetes durchgeführt werden kann.

Auch andere Workflow-Technologien können für die Generierung und Ausführung der LDWs genutzt werden. In der prototypischen Implementierung des DIVA-Werkzeugs wurde BPMN als Modellierungssprache und die Camunda Workflow-Engine zur Ausführung verwendet.

7.4 Integration mit anderen Systemen

Teile des DIVA-Werkzeugs werden auch in anderen Anwendungsfällen eingesetzt, zum einen zum TOSCA-basierten Deployment und zum anderen zur situationsabhängigen Partnerselektion.

7.4.1 TOSCA-basiertes Deployment

Die Modellierungsumgebung Winery unterstützt nativ den TOSCA-Standard. Dadurch ist auch die Integration mit einer TOSCA-Runtime zur Ausführung des Deployments möglich. Dazu sind jedoch auch spezifische Erweiterungen erforderlich, um partnerübergreifende Anwendungen bereitstellen zu können: Das Modell muss entsprechend den annotierten Partnern interpretiert werden und Workflows zum Orchestrieren der lokalen Deployment-Aktivitäten sowie zum Nachrichtenaustausch müssen generiert werden können.

Die TOSCA-Runtime *OpenTOSCA Container*[1] wurde im Rahmen der Arbeit zur Generierung lokaler Deployment-Workflows erweitert, um partnerübergreifende Anwendungen bereitstellen zu können [WBK+20]. OpenTOSCA Container wird anstelle des EDMM-Transformation-Frameworks zur Ausführung des Deployments verwendet, wobei hier keine weiteren Deployment-Technologien zum Einsatz kommen. Im Gegensatz zum EDMM-Transformation-Framework wird BPEL als Modellierungssprache zur Generierung der Workflows verwendet.

[1] `https://github.com/OpenTOSCA/container`

7.4.2 Situationsabhängige Partnerselektion

In Zusammenarbeit mit Képes et al. [KLWW21] wurden die Konzepte zur Modellierung und Bereitstellung partnerübergreifender Anwendungen mit dem situationsbezogenen Management von Anwendungssystemen verknüpft. Anwendungsspezifische Komponenten können dabei mit mehreren Partnern annotiert werden und zusätzlich werden sogenannte *Policies* definiert, die spezifizieren wann welcher Partner selektiert werden soll. Dies kann, zum Beispiel, von der Verfügbarkeit oder den Kosten abhängig sein. Zur Deploymentzeit werden basierend auf der aktuellen Situation die Policies ausgewertet und der entsprechende Partner, der in der herrschenden Situation laut Policy zu wählen ist, selektiert und eine entsprechende Initiierungsnachricht geschickt. Die Partner, die nicht selektiert wurden, erhalten keine Initiierungsnachricht und sind damit nicht Teil der Instanz.

Zusammenfassung und Ausblick

Die automatisierte Bereitstellung von Anwendungen ist ein integraler Bestandteil des Managements von IT-Systemen heutiger Unternehmen, um eine effektive Kollaboration zwischen der Entwicklung und dem Betrieb von Anwendungssystemen zu ermöglichen. Um den Anforderungen der verteilten, stetig komplexer werdenden Anwendungssysteme gerecht zu werden, wurde in den letzten Jahren eine Vielzahl an Technologien entwickelt. Dabei wurden organisatorische Aspekte jedoch nicht berücksichtigt. In immer mehr Bereichen spielt jedoch die horizontale Integration über Organisationseinheiten, seien es unterschiedliche Teile eines Unternehmens oder unterschiedliche Unternehmen, eine entscheidende Rolle. Diese Arbeit hat ein dezentrales Deployment-Konzept eingeführt, welches die Autonomie der verschiedenen involvierten Partner sicherstellt und gleichzeitig die automatisierte partnerübergreifende Bereitstellung ermöglicht. Im Folgenden werden in Abschnitt 8.1 die Forschungsbeiträge der Arbeit zusammengefasst und in Abschnitt 8.2 ein Ausblick auf zukünftige Arbeiten gegeben.

8.1 Zusammenfassung der Forschungsbeiträge

Mit der DIVA-Methode in Kapitel 3 wurde ein ganzheitliches Konzept für die Modellierung, Analyse, Adaption und das Deployment partnerübergreifender Anwendungen eingeführt. Sie ist Forschungsbeitrag 1 dieser Arbeit und definiert einen Prozess, der alle Schritte zur Bereitstellung, von der Modellierung bis zur Ausführung, abdeckt, wobei das Management zur Laufzeit nicht berücksichtigt wird. Auf Basis der in Kapitel 2 diskutierten Grundlagen und verwandten Arbeiten wurden Anforderungen definiert (Abschnitt 3.1), die von der DIVA-Methode erfüllt werden. Die Methodenschritte sind in einen globalen und einen lokalen Kontext eingeordnet. Der Kontext spezifiziert, ob die jeweiligen Schritte in Abhängigkeit zu anderen Partnern (global) oder unabhängig (lokal) durchgeführt werden. So müssen die anwendungsspezifischen Komponenten im globalen Kontext modelliert werden, die partnerspezifischen Details zur Bereitstellung werden jedoch im lokalen Kontext durch Verfahren zur Verteilung und Modelladaption spezifiziert. Die DIVA-Methode ist jedoch nicht auf partnerübergreifende Anwendungen beschränkt, die Konzepte zur Analyse und Adaption von Deployment-Modellen können auch in einer Single-Partner-Variante genutzt werden. Dabei entfallen die Schritte des globalen Kontexts, und etablierte Verfahren zur zentralen Ausführung des Deployments können eingesetzt werden. Modellierung, Modellanalyse und -adaption basieren auf einem deklarativen Deployment-Modell und die Ausführung auf einem imperativen Modell, konkret einem Workflow. Die Methode ist dabei jedoch technologieunabhängig.

In Kapitel 4 wurden spezifische Modellierungskonzepte für partnerübergreifende Anwendungen eingeführt, die Forschungsbeitrag 2 der Arbeit repräsentieren. Zum einen wurde das EDMM erweitert und zum anderen eine Methode für die Verteilung der Komponenten sowohl auf verschiedene Partner als auch auf verschiedene Cloud-Anbieter präsentiert. Grundsätzlich wird dabei zwischen dem globalen Deployment-Modell (GDM) mit allen global relevanten Informationen und den lokalen Deployment-Modellen (LDMs) mit zusätzlichen partnerspezifischen Details unterschieden. Durch die Metamodellerweiterung können Komponenten mit Partnern annotiert

werden sowie Anforderungen und Fähigkeiten spezifiziert werden, wie es unter anderem auch der TOSCA-Standard ermöglicht. Durch den strukturellen Abgleich sowie die definierten Anforderungen und Fähigkeiten lassen sich existierende Deployment-Modelle wiederverwenden, besonders um die effektive Kooperation zwischen Softwareentwicklung und Betrieb zu ermöglichen und Deployment-Modelle aus der Entwicklungsphase als Basis für die Deployment-Modelle in der operativen Phase zu verwenden.

Zusätzlich können Komponenten mit Zielumgebungen annotiert werden, um die Verteilung auf unterschiedliche Cloud-Anbieter zu ermöglichen. Für die Automatisierung der Verteilung der Komponenten wurde die Split-and-Match-Methode eingeführt (Abschnitt 4.4.2). Der Fokus liegt dabei auf der schrittweisen Adaption eines Deployment-Modells zur Realisierung der vom Anwender vorgegebenen Verteilung auf unterschiedliche Cloud-Anbieter und zur Erhaltung eines validen Deployment-Modells. Die Verteilungsentscheidung selbst ist nicht Teil der Methode, eine Vielzahl an Arbeiten existieren jedoch, die sich mit der optimalen Verteilung von Anwendungskomponenten beschäftigen (Abschnitt 2.1.3.1).

Zur Erkennung von durch die Verteilung der Anwendungskomponenten auf verschiedene Partner oder Cloud-Anbieter entstandenen Problemen, die die Funktionalität der Anwendung einschränken, wurde in Kapitel 5 ein musterbasiertes Verfahren zur Problemerkennung und Modelladaption eingeführt. Die zentrale Idee ist, das Wissen über häufig wiederkehrende Probleme, welches in Mustern dokumentiert ist, zu nutzen, um Deployment-Modelle zu analysieren und eine kontextspezifische Empfehlung von konkreten Lösungen zur Behebung der identifizierten Problemen zu geben, welche automatisiert auf die Deployment-Modelle angewendet werden können. Die Anwendbarkeit konkreter Lösungen ist dabei abhängig von der Struktur und den Eigenschaften des Deployment-Modells sowie den verwendeten Technologien und Diensten.

Zur Automatisierung des Verfahrens wird das Wissen aus den Mustern formalisiert. Dazu wird Logik erster Ordnung verwendet und zur Implementierung die Logikprogrammiersprache Prolog. Im Gegensatz zu den häufig verwendeten Graph-Matching-Verfahren kann damit auch die Nicht-Existenz

von Elementen geprüft werden. Dabei liegt die Annahme einer geschlossenen Welt zugrunde, die für die Domäne der Deployment-Modelle gegeben ist. Es wird davon ausgegangen, dass alles notwendige Wissen über die Bereitstellung im Deployment-Modell spezifiziert ist. Auch für die Spezifikation der Anwendbarkeit konkreter Lösungen wird Logik erster Ordnung und Prolog verwendet, um den Deployment-Kontext (DK) zu formalisieren, in dem die Lösungsimplementierungen, in Form von Adaptionsalgorithmen (AAs), angewendet werden können. Das Verfahren ermöglicht durch diesen zweistufigen Ansatz sowohl die Verwendung der generischen Beschreibungen der Muster für die Problemerkennung als auch die spezifische technologieabhängige Definition zur Anwendbarkeit konkreter Lösungsimplementierungen. Dabei ist das Verfahren nicht auf die Formalisierung von Mustern beschränkt, auch Entwurfsprinzipien oder Compliance-Regeln können die Basis sein. Die Gültigkeit der Formalisierung der Muster ist dabei jedoch auf die Domäne der Deployment-Modelle und das EDMM beschränkt. Bei der Definition allgemeingültiger Regeln stellt außerdem die Variabilität von sowohl Problem- als auch Lösungsinstanzen in Deployment-Modellen eine Schwierigkeit dar. Entweder werden die Regeln auf hoher Abstraktionsebene definiert, wodurch die Genauigkeit eingeschränkt ist, oder es müssen mehrere spezifische Regeln definiert werden, wodurch sich die Komplexität bei der Definition der Regeln erhöht. Das vorgestellte Verfahren macht diesbezüglich keine Einschränkungen und kann zur Definition sowohl abstrakter Regeln als auch mehrerer spezifischer Regeln verwendet werden.

Für die Modelladaption wurde neben den AAs auch ein Konzept zur Selektion von Kommunikationstreibern eingeführt (Abschnitt 5.5). Im Gegensatz zu den auf die reine Anpassung des Deployment-Modells ausgerichteten AAs wird dabei ein Programmiermodell eingeführt, welches die Implementierung von Anwendungskomponenten unabhängig von dem später verwendeten Kommunikationsprotokoll ermöglicht. Einzig das Kommunikationsmuster wird dabei festgelegt. Abhängig von der jeweiligen Integrations-Middleware können anschließend die benötigten Kommunikationstreiber selektiert werden. Dadurch wird die Portabilität der Anwendungen erleichtert und die Adaption für unterschiedliche Middleware kann ohne zusätzliche Kompo-

nenten erfolgen. Das Selektionsverfahren wird dabei automatisiert auf der Struktur und den Eigenschaften des Deployment-Modells durchgeführt.

Für die Ausführung der Bereitstellung ist die Autonomie der Partner, das heißt der beteiligten Organisationseinheiten, eine wichtige Anforderung. Dafür wurde als Forschungsbeitrag 4 in Kapitel 6 ein Konzept zum automatisierten partnerübergreifenden dezentralen Deployment eingeführt. Zur dezentralen Koordination des Deployments ohne zentrale Koordinationseinheit generiert jeder Partner aus seinem LDM einen lokalen Deployment-Workflow (LDW), welcher die Aktivitäten zur Bereitstellung der lokalen Komponenten sowie zum Nachrichtenaustausch mit den übrigen Partnern orchestriert. Die generierten Workflows formen eine Deployment-Choreographie, wofür die Schnittstellen und Kommunikationsendpunkte global abgestimmt und bekannt sein müssen. Für das Deployment der lokalen Komponenten, die auch verteilt bei unterschiedlichen Cloud-Anbietern bereitgestellt werden können, während deren Deployment zentral durchgeführt wird, können unterschiedliche deklarative Deployment-Technologien eingesetzt werden. Alle Erweiterungen des EDMMs wurden für die Entwurfsphase eingeführt, für die Deployment-Phase sind diese jedoch nicht relevant und es können beliebige Deployment-Technologien, auch in Kombination, eingesetzt werden, wenn diese EDMM-konform [WBF+19] sind. Auch für die Workflows können beliebige Technologien verwendet werden, wenn diese die Ausführung von Choreographien ermöglichen. Damit werden deklarative und imperative Technologien kombiniert.

Der letzte Forschungsbeitrag, Forschungsbeitrag 5, wird in Kapitel 7 beschrieben und umfasst die Architektur, prototypische Implementierung und Validierung des DivA-Werkzeugs, das die Automatisierung der DivA-Methode ermöglicht. Das Werkzeug ist in bestehende Systeme zur Modellierung und Ausführung des Deployments von Anwendungssystemen integriert und erweitert diese um Funktionalitäten zur Umsetzung der eingeführten Konzepte. Mit Fokus auf die musterbasierte Problemerkennung und Modelladaption wurden noch weitere Muster und DKs formalisiert. Auch wurde das Werkzeug bereits mit Konzepten zum situationsabhängigen Management verknüpft, um so eine situationsabhängige Selektion von Partnern zu ermöglichen.

8.2 Ausblick

Die DIVA-Methode beschäftigt sich ausschließlich mit dem Prozess zur Modellierung und initialen Bereitstellung von partnerübergreifenden Anwendungen. Das Management zur Laufzeit wird dabei bisher nicht berücksichtigt. Konzepte zum Anwendungsmanagement wie sie von Breitenbücher [Bre16] oder Harzenetter et al. [HBL+19] vorgestellt werden, bauen auf der Generierung von Management-Plänen basierend auf deklarativen Deployment-Modellen auf und könnten die bisherigen Arbeiten zum dezentralen Deployment ergänzen. Dabei müssen die Abhängigkeiten zwischen den Komponenten der Partnern berücksichtigt werden und insbesondere die Auswirkungen von zustandsmodifizierenden Management-Aktivitäten analysiert werden. Darüber hinaus sollten auch bereits bestehende Management-Aktivitäten betrachtet und analysiert werden, die Komponenten betreffen, welche auf verschiedene Partner verteilt sind. Diese Aktivitäten müssen verteilt und global koordiniert ausgeführt werden können. Damit kann dann der gesamte Lebenszyklus von partnerübergreifenden Anwendungen berücksichtigt werden.

Im Rahmen dieser Arbeit lag der Fokus auf Mustern, die die Kommunikation zwischen Anwendungskomponenten betreffen. In Zukunft können auch weitere Mustersprachen einbezogen werden und auch sind die Konzepte zur musterbasierten Problemerkennung und Modelladaption nicht auf Muster zur Kommunikation beschränkt. Bisher wurden die Integrationsmuster aus der Cloud-Mustersprache [FLR+14] und Muster zur sicheren Kommunikation aus den Sicherheitsmustern [SFH+06] berücksichtigt. Muster aus anderen Bereichen dieser Sprachen, beispielsweise zu verschiedenen Cloud-Diensten, können ebenso aufgenommen werden wie Muster, zum Beispiel zur Migration von Anwendungen [JPCL14; Men14]. Das grundsätzliche Konzept zur musterbasierten Problemerkennung ist auch nicht auf Deployment-Modelle beschränkt, auch die Anwendung auf andere Domänen kann in Zukunft untersucht und hier die Anwendbarkeit des in dieser Arbeit vorgestellten Verfahrens validiert werden.

Literaturverzeichnis

[AB14a] A. Ahmad, M. A. Babar. „A Framework for Architecture-Driven Migration of Legacy Systems to Cloud-Enabled Software“. In: *Proceedings of the WICSA 2014 Companion Volume*. WICSA ’14 Companion. Association for Computing Machinery, 2014 (Zitiert auf S. 59, 65, 66, 70, 71, 149).

[AB14b] A. Ahmad, M. A. Babar. „Towards a Pattern Language for Self-Adaptation of Cloud-Based Architectures“. In: *Proceedings of the WICSA 2014 Companion Volume*. WICSA ’14 Companion. Association for Computing Machinery, 2014 (Zitiert auf S. 59, 65, 66, 70, 71, 149).

[AB16] M. R. M. Assis, L. F. Bittencourt. „A survey on cloud federation architectures: Identifying functional and non-functional properties“. In: *Journal of Network and Computer Applications* 72 (2016), S. 51–71 (Zitiert auf S. 167).

[ABD+18] M. Artac, T. Borovšak, E. Di Nitto, M. Guerriero, D. Perez-Palacin, D. A. Tamburri. „Infrastructure-as-Code for Data-Intensive Architectures: A Model-Driven Development Approach“. In: *2018 IEEE International Conference on Software Architecture (ICSA)*. 2018, S. 156–165 (Zitiert auf S. 35, 36).

[ABL15] J. P. Arcangeli, R. Boujbel, S. Leriche. „Automatic deployment of distributed software systems: Definitions and state of the art“. In: *Journal of Systems and Software* 103 (2015), S. 198–218 (Zitiert auf S. 31, 48, 49).

[ACCM17] R. B. Antequera, P. Calyam, A. A. Chandrashekara, S. Malhotra. „Recommending Resources to Cloud Applications Based on Custom Templates Composition“. In: *Proceedings of the Computing Frontiers Conference*. CF'17. Association for Computing Machinery, 2017, S. 136–145 (Zitiert auf S. 42, 126).

[ACCM19] R. B. Antequera, P. Calyam, A. A. Chandrashekara, R. Mitra. „Recommending heterogeneous resources for science gateway applications based on custom templates composition“. In: *Future Generation Computer Systems* 100 (2019), S. 281–297 (Zitiert auf S. 42).

[ÁCJ+16] E. Ábrahám, F. Corzilius, E. B. Johnsen, G. Kremer, J. Mauro. „Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies“. In: *Dependable Software Engineering: Theories, Tools, and Applications*. Springer International Publishing, 2016, S. 229–245 (Zitiert auf S. 32, 33).

[ADM+12] D. Ardagna, E. Di Nitto, P. Mohagheghi, S. Mosser, C. Ballagny, F. D'Andria, G. Casale, P. Matthews, C.-S. Nechifor, D. Petcu, A. Gericke, C. Sheridan. „MODAClouds: A model-driven approach for the design and execution of applications on multiple Clouds“. In: *2012 4th International Workshop on Modeling in Software Engineering (MISE)*. 2012, S. 50–56 (Zitiert auf S. 43).

[AEK+07] W. Arnold, T. Eilam, M. Kalantar, A. V. Konstantinou, A. A. Totok. „Pattern Based SOA Deployment“. In: *Proceedings of the Fifth International Conference on Service-Oriented Computing (ICSOC 2007)*. Springer, Sep. 2007, S. 1–12 (Zitiert auf S. 69, 70).

[AEK+08] W. Arnold, T. Eilam, M. Kalantar, A. V. Konstantinou, A. A. Totok. „Automatic Realization of SOA Deployment Patterns in Distributed Environments“. In: *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC 2008)*. Springer, Dez. 2008, S. 162–179 (Zitiert auf S. 42, 126).

[AGLW14] V. Andrikopoulos, S. Gómez Sáez, F. Leymann, J. Wettinger. „Optimal Distribution of Applications in the Cloud“. In: *Proceedings of the 26th International Conference on Advanced Information Systems Engineering (CAiSE 2014)*. Springer, Juni 2014, S. 75–90 (Zitiert auf S. 17, 39, 125).

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Aug. 1977 (Zitiert auf S. 56).

[AJP12] A. Ahmad, P. Jamshidi, C. Pahl. „Graph-Based Pattern Identification from Architecture Change Logs“. In: *Advanced Information Systems Engineering Workshops*. Springer Berlin Heidelberg, 2012, S. 200–213 (Zitiert auf S. 66).

[AKR+19] A. P. Achilleos, K. Kritikos, A. Rossini, G. M. Kapitsaki, J. Domaschka, M. Orzechowski, D. Seybold, F. Griesinger, N. Nikolov, D. Romero, G. A. Papadopoulos. „The cloud application modelling and execution language“. In: *Journal of Cloud Computing* 8.1 (2019) (Zitiert auf S. 30, 32, 33).

[ALKH17] K. Alexander, C. Lee, E. Kim, S. Helal. „Enabling End-to-End Orchestration of Multi-Cloud Applications“. In: *IEEE Access* 5 (2017), S. 18862–18875 (Zitiert auf S. 41).

[AMA19] R. G. Aryal, J. Marshall, J. Altmann. „Architecture and Business Logic Specification for Dynamic Cloud Federations“. In: *Economics of Grids, Clouds, Systems, and Services*. Bd. 11819. Springer International Publishing, 2019, S. 83–96 (Zitiert auf S. 167).

[Ama22a] Amazon Web Services. *AWS Solution Constructs Patterns*. 2022. URL: `https://aws.amazon.com/solutions/constructs/patterns/` (Zitiert auf S. 56, 67).

[Ama22b] Amazon Web Services. *CloudFormation Official Site*. 2022. URL: `https://aws.amazon.com/cloudformation` (Zitiert auf S. 12).

[Arc] Arcitura Education. *Service-Oriented Architecture Patterns*. URL: `https://patterns.arcitura.com/soa-patterns` (Zitiert auf S. 71).

[Bar18] J. Barzen. „Wenn Kostüme sprechen - Musterforschung in den Digital Humanities am Beispiel vestimentärer Kommunikation im Film“. Diss. Universität zu Köln, Philosophische Fakultät, 2018 (Zitiert auf S. 67).

[BBF+18] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann. „A Systematic Review of Cloud Modeling Languages“. In: *ACM Computing Surveys (CSUR)* 51.1 (Feb. 2018), S. 1–38 (Zitiert auf S. 29–32).

[BBF09] G. Blair, N. Bencomo, R. B. France. „Models@ run.time“. In: *Computer* 42.10 (2009), S. 22–27 (Zitiert auf S. 33).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. „OpenTOSCA – A Runtime for TOSCA-based Cloud Applications“. In: *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*. Bd. 8274. LNCS. Springer, Dez. 2013, S. 692–695 (Zitiert auf S. 21, 170).

[BBK+12] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, D. Schumm. „Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA“. In: *On the Move to Meaningful Internet Systems: OTM 2012 (CoopIS 2012)*. Springer, Sep. 2012, S. 416–424 (Zitiert auf S. 106).

[BBK+14] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. „Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA“. In: *International Conference on Cloud Engineering (IC2E 2014)*. IEEE, März 2014, S. 87–96 (Zitiert auf S. 30, 49, 50, 85, 87, 157, 158, 162, 166, 167).

[BBK+16] A. Bergmayr, U. Breitenbücher, O. Kopp, M. Wimmer, G. Kappel, F. Leymann. „From Architecture Modeling to Application Provisioning for the Cloud by Combining UML and TOSCA“. In: *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016)*. SciTePress, Apr. 2016, S. 97–108 (Zitiert auf S. 32, 35).

[BBKL13a] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. „Automated Discovery and Maintenance of Enterprise Topology Graphs“. In: *Proceedings of the 6th IEEE International Conference on Service Oriented Computing and Applications (SOCA 2013)*. IEEE, Dez. 2013, S. 126–134 (Zitiert auf S. 90).

[BBKL13b] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. „Pattern-based Runtime Management of Composite Cloud Applications“. In: *Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER 2013)*. SciTePress, Mai 2013, S. 475–482 (Zitiert auf S. 44, 50, 59, 64–66, 69–71).

[BBKL14] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. „Automating Cloud Application Management Using Management Idioms“. In: *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS 2014)*. Xpert Publishing Services, Mai 2014, S. 60–69 (Zitiert auf S. 50).

[BCS17] A. Brogi, P. Cifariello, J. Soldani. „DrACO: Discovering available cloud offerings“. In: *Computer Science - Research and Development* 32.3 (2017), S. 269–279 (Zitiert auf S. 42).

[BDS18] A. Brogi, A. Di Tommaso, J. Soldani. „Sommelier: A Tool for Validating TOSCA Application Topologies“. In: *Model-Driven Engineering and Software Development*. Springer International Publishing, 2018, S. 1–22 (Zitiert auf S. 44).

[BEK+16] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. „The OpenTOSCA Ecosystem - Concepts & Tools“. In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016* (Dez. 2016), S. 112–130 (Zitiert auf S. 21, 32, 170).

[BF03] Z. Balanyi, R. Ferenc. „Mining design patterns from C++ source code“. In: *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* 2003, S. 305–314 (Zitiert auf S. 59, 61).

[BGK+13] G. Baryannis, P. Garefalakis, K. Kritikos, K. Magoutis, A. Papaioannou, D. Plexousakis, C. Zeginis. „Lifecycle Management of Service-Based Applications on Multi-Clouds: A Research Roadmap“. In: MultiCloud ’13. Association for Computing Machinery, 2013, S. 13–20 (Zitiert auf S. 39).

[BLF+21] J. Barzen, F. Leymann, M. Falkenthal, D. Vietz, B. Weder, K. Wild. „Relevance of Near-Term Quantum Computing in the Cloud: A Humanities Perspective“. In: *Cloud Computing and Services Science* 1399 (März 2021), S. 25–58 (Zitiert auf S. 24).

[BM20] J. Bellendorf, Z. Á. Mann. „Specification of cloud topologies and orchestration using TOSCA: a survey“. In: *Computing* 102.8 (2020), S. 1793–1815 (Zitiert auf S. 34, 37).

[BMG+19] H. Brabra, A. Mtibaa, W. Gaaloul, B. Benatallah, F. Gargouri. „Model-Driven Orchestration for Cloud Resources“. In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, S. 422–429 (Zitiert auf S. 36, 37).

[BNRS18] A. Brogi, D. Neri, L. Rinaldi, J. Soldani. „Orchestrating incomplete TOSCA applications with Docker“. In: *Science of Computer Programming* 166 (2018), S. 194–213 (Zitiert auf S. 41).

[BNS20] A. Brogi, D. Neri, J. Soldani. „Freshening the Air in Microservices: Resolving Architectural Smells via Refactoring“. In: *Service-Oriented Computing – ICSOC 2019 Workshops*. Springer International Publishing, 2020, S. 17–29 (Zitiert auf S. 150).

[BP02] F. Bergenti, A. Poggi. „Improving UML Designs Using Automatic Design Pattern Detection“. In: *Handbook of Software Engineering and Knowledge Engineering*. World Scientific, 2002, S. 771–784 (Zitiert auf S. 58, 59, 62).

[Bre16] U. Breitenbücher. „Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements“. Dissertation. University of Stuttgart, Faculty 5, 2016 (Zitiert auf S. 13, 19, 49, 50, 59, 64–66, 69–71, 96, 101, 102, 125, 141, 148, 149, 200).

[BRS18] A. Brogi, L. Rinaldi, J. Soldani. „TosKer: A synergy between TOSCA and Docker for orchestrating multicomponent applications“. In: *Software: Practice and Experience* 48.11 (2018), S. 2061–2079 (Zitiert auf S. 32).

[BSG+18] D. Baur, D. Seybold, F. Griesinger, H. Masata, J. Domaschka. „A Provider-Agnostic Approach to Multi-cloud Orchestration Using a Constraint Language“. In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2018, S. 173–182 (Zitiert auf S. 39).

[BTN+14] A. Bergmayr, J. Troya, P. Neubauer, M. Wimmer, G. Kappel. „UML-based Cloud Application Modeling with Libraries, Profiles, and Templates“. In: *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE 2014)*. CEUR-WS.org, Sep. 2014, S. 56–65 (Zitiert auf S. 32, 35).

[Cam22] Camunda Services GmbH. *Camunda Official Site*. 2022. URL: `https://camunda.com` (Zitiert auf S. 171).

[CCDT18] D. Calcaterra, V. Cartelli, G. Di Modica, O. Tomarchio. „A Framework for the Orchestration and Provision of Cloud Services Based on TOSCA and BPMN“. In: *Cloud Computing and Service Science*. Springer International Publishing, 2018, S. 262–285 (Zitiert auf S. 30, 50, 85, 87, 157).

[CCP15] J. Carrasco, J. Cubo, E. Pimentel. „Towards a Flexible Deployment of Multi-cloud Applications Based on TOSCA and CAMP“. In: *Advances in Service-Oriented and Cloud Computing*. Springer International Publishing, 2015, S. 278–286 (Zitiert auf S. 41).

[CCPD16] J. Carrasco, J. Cubo, E. Pimentel, F. Durán. „Deployment over Heterogeneous Clouds with TOSCA and CAMP“. In: *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016)*. ScitePress, 2016, S. 170–177 (Zitiert auf S. 41).

[CD15] G. Cretella, B. Di Martino. „A semantic engine for porting applications to the cloud and among clouds“. In: *Software: Practice and Experience* 45.12 (2015), S. 1619–1637 (Zitiert auf S. 43).

[CDG12] F. Cuadrado, J. C. Duenas, R. Garcia-Carmona. „An Autonomous Engine for Services Configuration and Deployment“. In: *IEEE Transactions on Software Engineering* 38.3 (2012), S. 520–536 (Zitiert auf S. 45).

[CDP17] J. Carrasco, F. Durán, E. Pimentel. „Component-wise Application Migration in Bidimensional Cross-cloud Environments“. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,* SciTePress, 2017, S. 287–297 (Zitiert auf S. 42–44).

[CDP18] J. Carrasco, F. Durán, E. Pimentel. „Runtime Migration of Applications in a Trans-Cloud Environment“. In: *Service-Oriented Computing – ICSOC 2017 Workshops*. Springer International Publishing, 2018, S. 55–66 (Zitiert auf S. 126).

[CFA17] L. M. Camarinha-Matos, R. Fornasiero, H. Afsarmanesh. „Collaborative Networks as a Core Enabler of Industry 4.0“. In: *Collaboration in a Data-Rich World*. Springer International Publishing, 2017, S. 3–17 (Zitiert auf S. 11).

[Che22] Chef Software Inc. *Chef Official Site*. 2022. URL: `https://www.chef.io/` (Zitiert auf S. 12).

[Clo22] Cloudify Platform Ltd. *Cloudify Official Site*. 2022. URL: `https://cloudify.co/` (Zitiert auf S. 32, 34).

[CM12] W. F. Clocksin, C. S. Mellish. *Programming in Prolog: Using the ISO standard*. Springer Science & Business Media, 2012 (Zitiert auf S. 61).

[CMT14] V. Cortellessa, A. D. Marco, C. Trubiani. „An approach for modeling and detecting software performance antipatterns based on first-order logics“. In: *Software & Systems Modeling* 13.1 (2014), S. 391–432 (Zitiert auf S. 59, 63, 151).

[Cop96] J. O. Coplien. *Software Patterns*. SIGS Books & Multimedia, 1996 (Zitiert auf S. 63).

[CS16] S. Capelli, P. Scandurra. „A framework for early design and prototyping of service-oriented applications with design patterns“. In: *Computer Languages, Systems & Structures* 46 (2016), S. 140–166 (Zitiert auf S. 71).

[CV19] R. Casadei, M. Viroli. „Coordinating Computation at the Edge: a Decentralized, Self-Organizing, Spatial Approach“. In: *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*. 2019, S. 60–67 (Zitiert auf S. 20).

[DCE17] B. Di Martino, G. Cretella, A. Esposito. „Cloud services composition through cloud patterns: a semantic-based approach“. In: *Soft Computing* 21.16 (2017), S. 4557–4570 (Zitiert auf S. 67).

[DE13] B. Di Martino, A. Esposito. „Automatic Recognition of Design Patterns from UML-Based Software Documentation“. In: *Proceedings of International Conference on Information Integration and Web-Based Applications & Services*. IIWAS ’13. Association for Computing Machinery, 2013, S. 280–289 (Zitiert auf S. 59, 62).

[DEZ+15] R. Di Cosmo, J. Eiche Antoineand Mauro, S. Zacchiroli, G. Zavattaro, J. Zwolakowski. „Automatic Deployment of Services in the Cloud with Aeolus Blender“. In: *Service-Oriented Computing*. Springer Berlin Heidelberg, 2015, S. 397–411 (Zitiert auf S. 32).

[DGRB15] A. V. Dastjerdi, S. K. Garg, O. F. Rana, R. Buyya. „CloudPick: a framework for QoS-aware and ontology-based service deployment across clouds“. In: *Software: Practice and Experience* 45.2 (2015), S. 197–231 (Zitiert auf S. 39).

[DKLW07] G. Decker, O. Kopp, F. Leymann, M. Weske. „BPEL4Chor: Extending BPEL for Modeling Choreographies“. In: *Proceedings of the IEEE 2007 International Conference on Web Services (ICWS 2007), Salt Lake City, Utah, USA, July 2007*. Hrsg. von I. C. Society. Salt Lake City: IEEE Computer Society, Juli 2007, S. 296–303 (Zitiert auf S. 51, 166).

[DL06] Dae-Kyoo Kim, Lunjin Lu. „Inference of design pattern instances in UML models via logic programming“. In: *11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)*. 2006 (Zitiert auf S. 19, 59, 62, 64).

[DMT12] DMTF. *Cloud Infrastructure Management Interface (CIMI) Version 1.0.0*. Distributed Management Task Force (DMTF). 2012 (Zitiert auf S. 40).

[DMZZ14] R. Di Cosmo, J. Mauro, S. Zacchiroli, G. Zavattaro. „Aeolus: a Component Model for the Cloud“. In: *Information and Computation* (Jan. 2014), S. 100–121 (Zitiert auf S. 31, 32, 49).

[EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. „Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications“. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications (PATTERNS 2017)*. Xpert Publishing Services, Feb. 2017, S. 22–27 (Zitiert auf S. 18, 28).

[EBLW17] C. Endres, U. Breitenbücher, F. Leymann, J. Wettinger. „Anything to Topology - A Method and System Architecture to Topologize Technology-Specific Application Deployment Artifacts“. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017), Porto, Portugal*. SciTePress, Apr. 2017, S. 180–190 (Zitiert auf S. 90).

[Ecl] Eclipse Foundation. *Eclipse Mosquitto Official Site*. URL: `https://mosquitto.org/` (Zitiert auf S. 144).

[EEKS11] T. Eilam, M. Elder, A. V. Konstantinou, E. Snible. „Pattern-based Composite Application Deployment“. In: *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011)*. IEEE, Mai 2011, S. 217–224 (Zitiert auf S. 50).

[EKK+06] T. Eilam, M. Kalantar, A. Konstantinou, G. Pacifici, J. Pershing, A. Agrawal. „Managing the configuration complexity of distributed applications in Internet data centers“. In: *Communications Magazine* 44.3 (März 2006), S. 166–177 (Zitiert auf S. 43).

[ESB+17] X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, N. De Palma. „Reliable self-deployment of distributed cloud applications“. In: *Software: Practice and Experience* 47.1 (2017), S. 3–20 (Zitiert auf S. 53, 54).

[FAS17a] N. Ferry, M. Almeida, A. Solberg. „The MODAClouds model-driven development“. In: *Model-Driven Development and Operation of Multi-Cloud Applications*. Springer, 2017, S. 23–33 (Zitiert auf S. 32).

[FAS17b] N. Ferry, M. Almeida, A. Solberg. „The MODAClouds model-driven development“. In: *Model-Driven Development and Operation of Multi-Cloud Applications*. Springer, 2017, S. 23–33 (Zitiert auf S. 43).

[FBB+14a] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. „Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains“. In: *International Journal On Advances in Software* 7.3&4 (Dez. 2014), S. 710–726 (Zitiert auf S. 19, 66, 67, 129, 137, 152).

[FBB+14b] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. „From Pattern Languages to Solution Implementations“. In: *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS 2014)*. Xpert Publishing Services, Mai 2014, S. 12–21 (Zitiert auf S. 66, 82, 137, 152, 181).

[FBB+15] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann, A. Hadjakos, F. Hentschel, H. Schulze. „Leveraging Pattern Application via Pattern Refinement“. In: *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC 2015)*. epubli, Juni 2015, S. 38–61 (Zitiert auf S. 56, 67).

[FBFL15] C. Fehling, J. Barzen, M. Falkenthal, F. Leymann. „PatternPedia – Collaborative Pattern Identification and Authoring“. In: *Proceedings of PURPLSOC (Pursuit of Pattern Languages for Societal Change). The Workshop 2014.* Aug. 2015, S. 252–284 (Zitiert auf S. 141).

[FBL18] M. Falkenthal, U. Breitenbücher, F. Leymann. „The Nature of Pattern Languages“. In: *Pursuit of Pattern Languages for Societal Change*. tredition, Okt. 2018, S. 130–150 (Zitiert auf S. 152).

[FCS+18] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, A. Solberg. „CloudMF: Model-Driven Management of Multi-Cloud Applications“. In: *ACM Trans. Internet Technol.* 18.2 (2018) (Zitiert auf S. 32, 33).

[FCSK03] R. France, S. Chosh, E. Song, D.-K. Kim. „A metamodeling approach to pattern-based model refactoring“. In: *IEEE Software* 20.5 (2003), S. 52–58 (Zitiert auf S. 13, 59, 64, 66, 70, 71, 148, 149).

[FCSS15] N. Ferry, F. Chauvel, H. Song, A. Solberg. „Continous Deployment of Multi-Cloud Systems“. In: *Proceedings of the 1st International Workshop on Quality-Aware DevOps*. QUDOS 2015. Association for Computing Machinery, 2015, S. 27–28 (Zitiert auf S. 83).

[FG13] D. Fahland, C. Gierds. „Analyzing and Completing Middleware Designs for Enterprise Integration Using Coloured Petri Nets“. In: *Advanced Information Systems Engineering*. Springer Berlin Heidelberg, 2013, S. 400–416 (Zitiert auf S. 19, 59, 61, 69).

[FHS13] S. Frey, W. Hasselbring, B. Schnoor. „Automatic conformance checking for migrating software systems to cloud infrastructures and platforms“. In: *Journal of Software: Evolution and Process* 25.10 (2013), S. 1089–1115 (Zitiert auf S. 149).

[FLR+11] C. Fehling, F. Leymann, R. Retter, D. Schumm, W. Schupeck. „An Architectural Pattern Language of Cloud-based Applications“. In: *Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP 2011)*. ACM, Okt. 2011 (Zitiert auf S. 68, 70).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, Jan. 2014, S. 367 (Zitiert auf S. 13, 44, 56, 67–69, 130, 143, 184–186, 188, 200).

[FLRS12] C. Fehling, F. Leymann, J. Rütschlin, D. Schumm. „Pattern-Based Development and Management of Cloud Applications“. In: *Future Internet* 4.1 (März 2012), S. 110–141 (Zitiert auf S. 68, 70).

[FRC+13] N. Ferry, A. Rossini, F. Chauvel, B. Morin, A. Solberg. „Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems“. In: *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing (CLOUD 2013)*. IEEE, Juli 2013, S. 887–894 (Zitiert auf S. 30, 32, 33).

[FSR+14] N. Ferry, H. Song, A. Rossini, F. Chauvel, A. Solberg. „CloudMF: Applying MDE to Tame the Complexity of Managing Multi-Cloud Applications“. In: *Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2014)*. IEEE, Dez. 2014, S. 269–277 (Zitiert auf S. 32, 33).

[FZ11] F. A. Fontana, M. Zanoni. „A tool for design pattern detection and software architecture reconstruction“. In: *Information sciences* 181.7 (2011), S. 1306–1324 (Zitiert auf S. 59, 60).

[GB14] N. Grozev, R. Buyya. „Inter-Cloud architectures and application brokering: taxonomy and survey“. In: *Software: Practice and Experience* 44.3 (März 2014), S. 369–390 (Zitiert auf S. 167, 168).

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Okt. 1994 (Zitiert auf S. 56, 59, 60, 62, 64, 66).

[GL19] J. Guth, F. Leymann. „Pattern-based rewrite and refinement of architectures using graph theory“. In: *Software-Intensive Cyber-Physical Systems (SICS)* (Aug. 2019), S. 1–12 (Zitiert auf S. 13, 59, 64–66, 69–71, 141, 148, 149, 152).

[GLTA16] N. Goonasekera, A. Lonie, J. Taylor, E. Afgan. „CloudBridge: A Simple Cross-Cloud Python Library“. In: *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*. XSEDE16. Association for Computing Machinery, 2016 (Zitiert auf S. 40).

[GMMC13] J. Guillén, J. Miranda, J. M. Murillo, C. Canal. „Developing Migratable Multicloud Applications Based on MDE and Adaptation Techniques“. In: *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*. NordiCloud ’13. Association for Computing Machinery, 2013, S. 30–37 (Zitiert auf S. 35, 154).

[HAR13] H. Herry, P. Anderson, M. Rovatsos. „Choreographing configuration changes“. In: *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*. 2013, S. 156–160 (Zitiert auf S. 51).

[Has22] HashiCorp, Inc. *Terraform Official Site*. 2022. URL: `https://www.terraform.io/` (Zitiert auf S. 12).

[HAW11] H. Herry, P. Anderson, G. Wickler. „Automated Planning for Configuration Changes“. In: *Proceedings of the 25th International Conference on Large Installation System Administration (LISA 2011)*. USENIX, Dez. 2011, S. 57–68 (Zitiert auf S. 29).

[HBBL14] P. Hirmer, U. Breitenbücher, T. Binz, F. Leymann. „Automatic Topology Completion of TOSCA-based Cloud Applications“. In: *Proceedings des CloudCycle14 Workshops auf der 44. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*. Bd. 232. LNI. Bonn: Gesellschaft für Informatik e.V. (GI), Sep. 2014, S. 247–258 (Zitiert auf S. 41, 125).

[HBF+18] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, C. Krieger, F. Leymann. „Pattern-based Deployment Models and Their Automatic Execution“. In: *11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2018)*. IEEE Computer Society, Dez. 2018, S. 41–52 (Zitiert auf S. 19, 43, 68, 70, 90, 109, 125).

[HBF+20] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, F. Leymann. „Pattern-based Deployment Models Revisited: Automated Pattern-driven Deployment Configuration“. In: *Proceedings of the Twelfth International Conference on Pervasive Patterns and Applications (PATTERNS 2020)*. Xpert Publishing Services, Okt. 2020, S. 40–49 (Zitiert auf S. 68, 70, 151).

[HBL+19] L. Harzenetter, U. Breitenbücher, F. Leymann, K. Saatkamp, B. Weder, M. Wurster. „Automated Generation of Management Workflows for Applications Based on Deployment Models“. In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC 2019)*. IEEE, Okt. 2019, S. 216–225 (Zitiert auf S. 25, 49, 200).

[Hei20] R. Heinrich. „Architectural runtime models for integrating runtime observations and component-based models“. In: *Journal of Systems and Software* 169 (2020) (Zitiert auf S. 38).

[HLB19] R. Hentschel, C. Leyh, T. Baumhauer. „Critical success factors for the implementation and adoption of cloud services in SMEs“. In: *Proceedings of the 52nd Hawaii International Conference on System Sciences*. 2019, S. 7342–7351 (Zitiert auf S. 12, 13).

[HS18] G. Horn, P. Skrzypek. „MELODIC: Utility Based Cross Cloud Deployment Optimisation“. In: *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. 2018, S. 360–367 (Zitiert auf S. 32).

[HW04] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004 (Zitiert auf S. 13, 44, 52, 56, 61, 69, 143, 187).

[HZ15] T. Haitzer, U. Zdun. „Semi-automatic architectural pattern identification and documentation using architectural primitives“. In: *Journal of Systems and Software* 102 (2015), S. 35–57 (Zitiert auf S. 19, 59, 60, 134).

[HZ18] T. Holmes, U. Zdun. „Refactoring Architecture Models for Compliance with Custom Requirements“. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS ’18. Association for Computing Machinery, 2018, S. 267–277 (Zitiert auf S. 45, 149, 151).

[JDB16] W. Jaradat, A. Dearle, A. Barker. „Towards an autonomous decentralized orchestration system“. In: *Concurrency and Computation: Practice and Experience* 28.11 (2016), S. 3164–3179 (Zitiert auf S. 166).

[JPCL14] P. Jamshidi, C. Pahl, S. Chinenyeze, X. Liu. „Cloud Migration Patterns: A Multi-Cloud Service Architecture Perspective“. In: *Service-Oriented Computing - ICSOC 2014 Workshops*. Springer, Nov. 2014, S. 6–19 (Zitiert auf S. 44, 71, 200).

[JPM17] P. Jamshidi, C. Pahl, N. C. Mendonça. „Pattern-based multi-cloud architecture migration“. In: *Software: Practice and Experience* 47.9 (2017), S. 1159–1184 (Zitiert auf S. 71).

[KA08] A. W. Kamal, P. Avgeriou. „Modeling Architectural Patterns' Behavior Using Architectural Primitives“. In: *Proceedings of the Second European Conference on Software Architecture (ECSA 2008)*. Springer, Okt. 2008, S. 164–179 (Zitiert auf S. 60).

[KB17] O. Kopp, U. Breitenbücher. „Choreographies are Key for Distributed Cloud Application Provisioning“. In: *Proceedings of the 9th Central-European Workshop on Services and their Composition*. CEUR-WS.org, 2017 (Zitiert auf S. 13, 51).

[KBBL12] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications“. In: *Proceedings of the 4th International Workshop on the Business Process Model and Notation (BPMN 2012)*. Springer, Sep. 2012, S. 38–52 (Zitiert auf S. 30).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „Winery – A Modeling Tool for TOSCA-based Cloud Applications“. In: *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*. Springer, Dez. 2013, S. 700–704 (Zitiert auf S. 21, 170).

[KBKL18] C. Krieger, U. Breitenbücher, K. Képes, F. Leymann. „An Approach to Automatically Check the Compliance of Declarative Deployment Models“. In: *Papers from the 12th Advanced Summer School on Service-Oriented Computing (SummerSoC 2018)*. IBM Research Division, Okt. 2018, S. 76–89 (Zitiert auf S. 149).

[KBL+19] K. Képes, U. Breitenbücher, F. Leymann, K. Saatkamp, B. Weder. „Deployment of Distributed Applications Across Public and Private Networks“. In: *Proceedings of the 23rd International Enterprise Distributed Object Computing Conference (EDOC 2019)*. IEEE, Okt. 2019, S. 236–242 (Zitiert auf S. 23, 52).

[KBL17] K. Képes, U. Breitenbücher, F. Leymann. „The SePaDe System: Packaging Entire XaaS Layers for Automatically Deploying and Managing Applications“. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SciTePress, Apr. 2017, S. 626–635 (Zitiert auf S. 42).

[KE07] D.-K. Kim, C. El Khawand. „An approach to precisely specifying the problem domain of design patterns“. In: *Journal of Visual Languages & Computing* 18.6 (2007), S. 560–591 (Zitiert auf S. 59, 63, 64, 134, 148).

[Kha08] R. Khalaf. „Supporting business process fragmentation while maintaining operational semantics: a BPEL perspective“. dissertation. Universität Stuttgart, 2008 (Zitiert auf S. 160, 166).

[KL06] R. Khalaf, F. Leymann. „Role-based Decomposition of Business Processes using BPEL“. In: *International Conference on Web Services (ICWS 2006)*. IEEE Computer Society, Sep. 2006, S. 770–780 (Zitiert auf S. 166).

[KLWW21] K. Képes, F. Leymann, B. Weder, K. Wild. „SiDD: The Situation-Aware Distributed Deployment System“. In: *Service-Oriented Computing – ICSOC 2020 Workshops*. Springer International Publishing, Mai 2021, S. 72–76 (Zitiert auf S. 23, 194).

[KP96] C. Kramer, L. Prechelt. „Design recovery by automated search for structural design patterns in object-oriented software“. In: *Proceedings of WCRE ’96: 4rd Working Conference on Reverse Engineering*. 1996, S. 208–215 (Zitiert auf S. 59, 61).

[KS07] D.-K. Kim, W. Shen. „An Approach to Evaluating Structural Pattern Conformance of UML Models“. In: *Proceedings of the 2007 ACM Symposium on Applied Computing*. SAC ’07. Association for Computing Machinery, 2007, S. 1404–1408 (Zitiert auf S. 59, 60, 64).

[KVM+20] I. Kumara, Z. Vasileiou, G. Meditskos, D. A. Tamburri, W.-J. Van Den Heuvel, A. Karakostas, S. Vrochidis, I. Kompatsiaris. „Towards Semantic Detection of Smells in Cloud Infrastructure Code“. In: WIMS 2020. Association for Computing Machinery, 2020, S. 63–67 (Zitiert auf S. 150).

[KWL14] N. Kaviani, E. Wohlstadter, R. Lea. „Partitioning of web applications for hybrid cloud deployment“. In: *Journal of Internet Services and Applications* 5.14 (2014) (Zitiert auf S. 126).

[LB21] F. Leymann, J. Barzen. „Pattern Atlas“. In: Springer International Publishing, Apr. 2021, S. 67–76 (Zitiert auf S. 57, 141, 180).

[LBF+20] F. Leymann, J. Barzen, M. Falkenthal, D. Vietz, B. Weder, K. Wild. „Quantum in the Cloud: Application Potentials and Research Opportunities“. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*. SciTePress, Mai 2020, S. 9–24 (Zitiert auf S. 25).

[LFM+11] F. Leymann, C. Fehling, R. Mietzner, A. Nowak, S. Dustdar. „Moving Applications to the Cloud: An Approach based on Application Model Enrichment“. In: *International Journal of Cooperative Information Systems* 20.3 (Sep. 2011), S. 307–356 (Zitiert auf S. 32, 34).

[LHB18] S. Lehrig, M. Hilbrich, S. Becker. „The architectural template method: templating architectural knowledge to efficiently conduct quality-of-service analyses“. In: *Software: Practice and Experience* 48.2 (2018), S. 268–299 (Zitiert auf S. 70).

[LR00] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000 (Zitiert auf S. 160).

[MD97] G. Meszaros, J. Doble. „Pattern Languages of Program Design 3“. In: Addison-Wesley, 1997. Kap. A Pattern Language for Pattern Writing, S. 529–574 (Zitiert auf S. 19, 57).

[ME16] B. D. Martino, A. Esposito. „A rule-based procedure for automatic recognition of design patterns in UML diagrams“. In: *Software: Practice and Experience* 46.7 (2016), S. 983–1007 (Zitiert auf S. 19, 56, 59, 62).

[Men14] N. C. Mendonça. „Architectural Options for Cloud Migration“. In: *Computer* 47.8 (2014), S. 62–66 (Zitiert auf S. 44, 200).

[MFK+21] D. Martyniuk, M. Falkenthal, N. Karam, A. Paschke, K. Wild. „An Analysis of Ontological Entities to Represent Knowledge on Quantum Computing Algorithms and Implementations“. In: *Proceedings of the Conference on Digital Curation Technologies (Qurator 2021)*. Bd. 2836. CEUR Workshop Proceedings. CEUR-WS.org, Feb. 2021, S. 1–9 (Zitiert auf S. 24).

[MMGC12] J. Miranda, J. M. Murillo, J. Guillén, C. Canal. „Identifying Adaptation Needs to Avoid the Vendor Lock-in Effect in the Deployment of Cloud SBAs“. In: *Proceedings of the 2nd International Workshop on Adaptive Services for the Future Internet and 6th International Workshop on Web APIs and Service Mashups*. WAS4FI-Mashups ’12. Association for Computing Machinery, 2012, S. 12–19 (Zitiert auf S. 38, 44).

[MR12] M. Menzel, R. Ranjan. „CloudGenius: Decision Support for Web Server Cloud Migration“. In: *Proceedings of the 21st International Conference on World Wide Web*. WWW ’12. Association for Computing Machinery, 2012, S. 979–988 (Zitiert auf S. 39).

[MŞP14] V. I. Munteanu, C. Şandru, D. Petcu. „Multi-cloud resource management: cloud service interfacing“. In: *Journal of Cloud Computing* 3.1 (2014) (Zitiert auf S. 40, 154).

[MUL09] R. Mietzner, T. Unger, F. Leymann. „Cafe: A Generic Configurable Customizable Composite Cloud Application Framework“. In: *On the Move to Meaningful Internet Systems: OTM 2009 (CoopIS 2009)*. Springer, Nov. 2009, S. 357–364 (Zitiert auf S. 32).

[NB19] S. Y. Nabavi, O. Bushehrian. „An adaptive plan-oriented and continuous software migration to cloud in dynamic enterprises“. In: *Software: Practice and Experience* 49.9 (2019), S. 1365–1378 (Zitiert auf S. 44).

[NBF+12] A. Nowak, T. Binz, C. Fehling, O. Kopp, F. Leymann, S. Wagner. „Pattern-driven Green Adaptation of Process-based Applications and their Runtime Infrastructure“. In: *Computing* (Feb. 2012), S. 463–487 (Zitiert auf S. 68, 70).

[NLS+11] A. Nowak, F. Leymann, D. Schleicher, D. Schumm, S. Wagner. „Green Business Process Patterns“. In: *Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP 2011)*. ACM, Okt. 2011 (Zitiert auf S. 68).

[NMPS17] E. D. Nitto, P. Matthews, D. Petcu, A. Solberg, Hrsg. *Model-Driven Development and Operation of Multi-Cloud Applications: The MODAClouds Approach*. Springer Nature, 2017 (Zitiert auf S. 17).

[OAS07a] OASIS. *SCA Assembly Model Specification Version 1.00*. Organization for the Advancement of Structured Information Standards (OASIS). März 2007 (Zitiert auf S. 71).

[OAS07b] OASIS. *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*. Organization for the Advancement of Structured Information Standards (OASIS). 2007 (Zitiert auf S. 30).

[OAS12] OASIS. *Cloud Application Management for Platforms (CAMP) Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS). Aug. 2012 (Zitiert auf S. 41).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS). 2013 (Zitiert auf S. 30, 32, 34, 101, 124, 170).

[OAS20] OASIS. *TOSCA Simple Profile in YAML Version 1.3*. Organization for the Advancement of Structured Information Standards (OASIS). 2020 (Zitiert auf S. 30, 32, 34, 49, 124, 171).

[OGF09] OGF. *Open Cloud Computing Interface (OCCI) Specification Version 1.1*. Open Grid Forum (OGF). 2009 (Zitiert auf S. 40).

[OMG07] OMG. *OMG Unified Modeling Language (UML)*. Object Management Group (OMG). 2007 (Zitiert auf S. 35).

[OMG11] OMG. *Business Process Model and Notation (BPMN) Version 2.0*. Object Management Group (OMG). 2011 (Zitiert auf S. 30).

[OMG14a] OMG. *Model Driven Architecture (MDA) MDA Guide rev. 2.0*. Object Management Group (OMG). 2014 (Zitiert auf S. 28).

[OMG14b] OMG. *Object Constraint Language (OCL)*. Object Management Group (OMG). 2014 (Zitiert auf S. 35).

[Ope22] OpenStack Community. *TOSCA Parser*. 2022. URL: https://opendev.org/openstack/tosca-parser/ (Zitiert auf S. 35).

[PJ11] B. Pfitzmann, N. Joukov. „Migration to Multi-image Cloud Templates". In: *2011 IEEE International Conference on Services Computing*. 2011, S. 80–87 (Zitiert auf S. 42, 126).

[PKG+00] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, A. I. Verkamo. „Software metrics by architectural pattern mining". In: *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*. Citeseer, 2000, S. 325–332 (Zitiert auf S. 59, 62, 63).

[PKYY20] J. Park, U. Kim, D. Yun, K. Yeom. „Approach for Selecting and Integrating Cloud Services to Construct Hybrid Cloud". In: *Journal of Grid Computing* 18.3 (2020), S. 441–469 (Zitiert auf S. 69).

[PLB+17] A. Palesandro, M. Lacoste, N. Bennani, C. Ghedira-Guegan, D. Bourge. „Mantus: Putting Aspects to Work for Flexible Multi-Cloud Deployment". In: *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. 2017, S. 656–663 (Zitiert auf S. 45).

[PMPC13] D. Petcu, G. Macariu, S. Panica, C. Crăciun. „Portable Cloud applications—From theory to practice". In: *Future Generation Computer Systems* 29.6 (2013), S. 1417–1430 (Zitiert auf S. 40).

[PV14] D. Petcu, A. V. Vasilakos. „Portability in clouds: approaches and research opportunities". In: *Scalable Computing: Practice and Experience* 15.3 (2014), S. 251–270 (Zitiert auf S. 38, 39).

[QHRD13] C. Quinton, N. Haderer, R. Rouvoy, L. Duchien. „Towards Multi-Cloud Configurations Using Feature Models and Ontologies". In: *Proceedings of the 2013 International Workshop on Multi-Cloud Applications and Federated Clouds*. MultiCloud '13. Association for Computing Machinery, 2013, S. 21–26 (Zitiert auf S. 43).

[QK18] P.-C. Quint, N. Kratzke. „Towards a Lightweight Multi-Cloud DSL for Elastic and Transferable Cloud-native Applications". In: *Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,* SciTePress, 2018, S. 400–408 (Zitiert auf S. 36).

[QRD16] C. Quinton, D. Romero, L. Duchien. „SALOON: a platform for selecting and configuring cloud environments“. In: *Software: Practice and Experience* 46.1 (2016), S. 55–78 (Zitiert auf S. 43).

[RKN+16] A. Rossini, K. Kritikos, N. Nikolov, J. Domaschka, F. Griesinger, D. Seybold, D. Romero, M. Orzechowski. *CAMEL Documentation v2015.9*. 2016 (Zitiert auf S. 32, 33).

[RVT+18] L. F. Rivera, N. M. Villegas, G. Tamura, M. Jiménez, H. A. Müller. „UML-Driven Automated Software Deployment“. In: *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*. CASCON ’18. IBM Corp., 2018, S. 257–268 (Zitiert auf S. 35).

[Sar03] K. Sartipi. „Software architecture recovery based on pattern matching“. In: *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* 2003, S. 293–296 (Zitiert auf S. 59, 60).

[SBA12] S. Sotiriadis, N. Bessis, N. Antonpoulos. „Decentralized meta-brokers for inter-cloud: Modeling brokering coordinators for interoperable resource management“. In: *2012 9th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2012)*. 2012, S. 2462–2468 (Zitiert auf S. 52, 168).

[SBB+20] M. Salm, J. Barzen, U. Breitenbücher, F. Leymann, B. Weder, K. Wild. „The NISQ Analyzer: Automating the Selection of Quantum Computers for Quantum Algorithms“. In: *Proceedings of the 14th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2020)*. Springer International Publishing, Dez. 2020, S. 66–85 (Zitiert auf S. 25).

[SBF+19] K. Saatkamp, U. Breitenbücher, M. Falkenthal, L. Harzenetter, F. Leymann. „An Approach to Determine & Apply Solutions to Solve Detected Problems in Restructured Deployment Models Using First-Order Logic“. In: *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*. SciTePress, Mai 2019, S. 495–506 (Zitiert auf S. 22, 127, 138).

[SBK+18] K. Saatkamp, U. Breitenbücher, K. Képes, F. Leymann, M. Zimmermann. „OpenTOSCA Injector: Vertical and Horizontal Topology Model Injection“. In: *Service-Oriented Computing – ICSOC 2017 Workshops*. Bd. 10797. Lecture Notes in Computer Science (LNCS). Springer International Publishing, Jan. 2018, S. 379–383 (Zitiert auf S. 23, 93).

[SBKL17] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. „Topology Splitting and Matching for Multi-Cloud Deployments“. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SciTePress, Apr. 2017, S. 247–258 (Zitiert auf S. 22, 93, 112).

[SBKL18] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. „Application Scenarios for Automated Problem Detection in TOSCA Topologies by Formalized Patterns“. In: *Papers From the 12th Advanced Summer School of Service-Oriented Computing (SummerSOC 2018)*. IBM Research Division, Okt. 2018, S. 43–53 (Zitiert auf S. 23, 127).

[SBKL19a] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. „An approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns“. In: *SICS Software-Intensive Cyber-Physical Systems* (Feb. 2019), S. 1–13 (Zitiert auf S. 22, 127, 150).

[SBKL19b] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. „Method, formalization, and algorithms to split topology models for distributed cloud application deployments“. In: *Computing* 102 (Apr. 2019), S. 343–363 (Zitiert auf S. 22, 93, 164).

[SBL+21] M. Salm, J. Barzen, F. Leymann, B. Weder, K. Wild. „Automating the Comparison of Quantum Compilers for Quantum Circuits“. In: *Proceedings of the 15th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2021)*. Springer International Publishing, Sep. 2021, S. 64–80 (Zitiert auf S. 24).

[SBLW17] K. Saatkamp, U. Breitenbücher, F. Leymann, M. Wurster. „Generic Driver Injection for Automated IoT Application Deployments“. In: *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services; Salzburg, Austria,*

December 4-6, 2017. ACM, Dez. 2017, S. 320–329 (Zitiert auf S. 22, 127, 143, 145).

[SDH+14] L. Sun, H. Dong, F. K. Hussain, O. K. Hussain, E. Chang. „Cloud service selection: State-of-the-art and future research directions“. In: *Journal of Network and Computer Applications* 45 (2014), S. 134–150 (Zitiert auf S. 38).

[Sel03] B. Selic. „The pragmatics of model-driven development“. In: *IEEE Software* 20.5 (2003), S. 19–25 (Zitiert auf S. 28).

[SFH+06] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, Inc., Jan. 2006, S. 565 (Zitiert auf S. 13, 56, 69, 130, 131, 200).

[SGW15] Y. Sun, J. Gray, J. White. „A demonstration-based model transformation approach to automate model scalability“. In: *Software & Systems Modeling* 14.3 (2015), S. 1245–1271 (Zitiert auf S. 46).

[SIA17] J. Sandobalin, E. Insfran, S. Abrahão. „An Infrastructure Modelling Tool for Cloud Provisioning“. In: *2017 IEEE International Conference on Services Computing (SCC)*. 2017, S. 354–361 (Zitiert auf S. 36).

[SIA19] J. Sandobalin, E. Insfran, S. Abrahão. „ARGON: A Model-Driven Infrastructure Provisioning Tool“. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019, S. 738–742 (Zitiert auf S. 36).

[SKL+19] K. Saatkamp, C. Krieger, F. Leymann, J. Sudendorf, M. Wurster. „Application Threat Modeling and Automated VNF Selection for Mitigation using TOSCA“. In: *2019 International Conference on Networked Systems (NetSys)*. IEEE, Okt. 2019, S. 1–6 (Zitiert auf S. 24).

[SMNB21] J. Soldani, G. Muntoni, D. Neri, A. Brogi. „The μTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures“. In: *Software: Practice and Experience* 51.7 (2021), S. 1591–1621 (Zitiert auf S. 150).

[Sok21] D. Sokolowski. „Deployment Coordination for Cross-Functional DevOps Teams“. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, S. 1630–1634 (Zitiert auf S. 53).

[SS16] T. Subramanian, N. Savarimuthu. „Application based brokering algorithm for optimal resource provisioning in multiple heterogeneous clouds“. In: *Vietnam Journal of Computer Science* 3.1 (2016), S. 57–70 (Zitiert auf S. 39).

[SSSL12] M. Smit, M. Shtern, B. Simmons, M. Litoiu. „Partitioning Applications for Hybrid and Federated Clouds“. In: *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*. CASCON ’12. IBM Corp., 2012, S. 27–41 (Zitiert auf S. 126).

[SVV+16] M. Sebrechts, T. Vanhove, G. Van Seghbroeck, T. Wauters, B. Volckaert, F. De Turck. „Distributed Service Orchestration: Eventually Consistent Cloud Operation and Integration“. In: *2016 IEEE International Conference on Mobile Services (MS)*. 2016, S. 156–159 (Zitiert auf S. 54).

[SVW+18] M. Sebrechts, G. Van Seghbroeck, T. Wauters, B. Volckaert, F. De Turck. „Orchestrator conversation: Distributed management of cloud applications“. In: *International Journal of Network Management* 28.6 (2018), e2036 (Zitiert auf S. 54).

[The14] The Apache Software Foundation. *jclouds*. 2014. URL: `https://jclouds.apache.org/` (Zitiert auf S. 40).

[The20] The Apache Software Foundation. *Apache Brooklyn*. 2020. URL: `https://brooklyn.apache.org/` (Zitiert auf S. 41).

[The22a] The Apache Software Foundation. *Libcloud*. 2022. URL: `https://libcloud.apache.org/` (Zitiert auf S. 40).

[The22b] The Linux Foundation. *Kubernetes Official Site*. 2022. URL: `https://kubernetes.io/` (Zitiert auf S. 12).

[TK16] C. Tsigkanos, T. Kehrer. „On Formalizing and Identifying Patterns in Cloud Workload Specifications“. In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 2016, S. 262–267 (Zitiert auf S. 59, 60).

[TMKO17] N. Taušan, J. Markkula, P. Kuvaja, M. Oivo. „Choreography in the embedded systems domain: A systematic literature review“. In: *Information and Software Technology* 91 (2017), S. 82–101 (Zitiert auf S. 157).

[TN03] T. Taibi, D. C. L. Ngo. „Formal Specification of Design Patterns - A Balanced Approach“. In: *Journal of Object Technology* 2.4 (2003), S. 127–140 (Zitiert auf S. 58, 59, 62, 64, 151).

[VBLW21] D. Vietz, J. Barzen, F. Leymann, K. Wild. „On Decision Support for Quantum Application Developers: Categorization, Comparison, and Analysis of Existing Technologies“. In: *Computational Science – ICCS 2021*. Springer International Publishing, Juni 2021, S. 127–141 (Zitiert auf S. 24).

[VKW21] D. Voigt, O. Kopp, K. Wild. „Systematic Literature Review Tools: Are we there yet?“ In: *Proceedings of the 13th Central European Workshop on Services and their Composition (ZEUS 2021)*. Bd. 2839. CEUR Workshop Proceedings. CEUR-WS.org, Feb. 2021, S. 83–88 (Zitiert auf S. 25).

[VSI+15] M. Vögler, J. Schleicher, C. Inzinger, S. Nastic, S. Sehic, S. Dustdar. „LEONORE – Large-Scale Provisioning of Resource-Constrained IoT Deployments“. In: *2015 IEEE Symposium on Service-Oriented System Engineering*. 2015, S. 78–87 (Zitiert auf S. 52).

[VWDV22] L. Van Hoye, T. Wauters, F. De Turck, B. Volckaert. „A secure cross-organizational container deployment approach to enable ad hoc collaborations“. In: *International Journal of Network Management* 32.4 (2022), e2194 (Zitiert auf S. 53).

[WAL15] J. Wettinger, V. Andrikopoulos, F. Leymann. „Enabling DevOps Collaboration and Continuous Delivery Using Diverse Application Environments“. In: *Proceedings of the 23rd International Conference on Cooperative Information Systems (CoopIS 2015)*. Springer, Okt. 2015 (Zitiert auf S. 38).

[WBB+17] D. Weerasiri, M. C. Barukh, B. Benatallah, Q. Z. Sheng, R. Ranjan. „A Taxonomy and Survey of Cloud Resource Orchestration Techniques“. In: *ACM Computer Surveys* 50.2 (Mai 2017) (Zitiert auf S. 13, 20, 29, 49, 51).

[WBB+19] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, V. Yussupov. „The EDMM Modeling and Transformation System". In: *Service-Oriented Computing – ICSOC 2019 Workshops*. Springer, Dez. 2019 (Zitiert auf S. 18, 21, 37, 85, 90, 94, 163, 171).

[WBB+20a] M. Weigold, J. Barzen, U. Breitenbücher, M. Falkenthal, F. Leymann, K. Wild. „Pattern Views: Concept and Tooling of Interconnected Pattern Languages". In: *Proceedings of the 14th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2020)*. Springer International Publishing, Dez. 2020, S. 86–103 (Zitiert auf S. 25, 130, 152).

[WBB+20b] M. Wurster, U. Breitenbücher, A. Brogi, L. Harzenetter, F. Leymann, J. Soldani. „Technology-Agnostic Declarative Deployment Automation of Cloud Applications". In: *Proceedings of the 8th European Conference on Service-Oriented and Cloud Computing (ESOCC 2020)*. Springer International Publishing, März 2020, S. 97–112 (Zitiert auf S. 18, 37, 94).

[WBB+20c] M. Wurster, U. Breitenbücher, A. Brogi, F. Leymann, J. Soldani. „Cloud-native Deploy-ability: An Analysis of Required Features of Deployment Technologies to Deploy Arbitrary Cloud-native Applications". In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*. SciTePress, Mai 2020, S. 171–180 (Zitiert auf S. 104).

[WBB+21] M. Wurster, U. Breitenbücher, A. Brogi, F. Diez, F. Leymann, J. Soldani, K. Wild. „Automating the Deployment of Distributed Applications by Combining Multiple Deployment Technologies". In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*. SciTePress, Mai 2021, S. 178–189 (Zitiert auf S. 23, 87, 90, 95, 99, 157, 163, 174, 191).

[WBBC16] D. Weerasiri, M. C. Barukh, B. Benatallah, J. Cao. „A Model-Driven Framework for Interoperable Cloud Resources Management". In: *Service-Oriented Computing*. Springer International Publishing, 2016, S. 186–201 (Zitiert auf S. 36).

[WBF+19] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. „The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies“. In: *SICS Software-Intensive Cyber-Physical Systems* 35 (Aug. 2019), S. 63–75 (Zitiert auf S. 13, 18, 23, 29, 30, 36, 37, 89, 94, 95, 167, 199).

[WBH+20a] K. Wild, U. Breitenbücher, L. Harzenetter, F. Leymann, D. Vietz, M. Zimmermann. „TOSCA4QC: Two Modeling Styles for TOSCA to Automate the Deployment and Orchestration of Quantum Applications“. In: *Proceedings of the 24th International Enterprise Distributed Object Computing Conference (EDOC 2020)*. IEEE, Okt. 2020, S. 125–134 (Zitiert auf S. 24).

[WBH+20b] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani. „TOSCA Lightning: An Integrated Toolchain for Transforming TOSCA Light into Production-Ready Deployment Technologies“. In: *Advanced Information Systems Engineering (CAiSE Forum 2020)*. Bd. 386. Lecture Notes in Business Information Processing. Springer International Publishing, Aug. 2020, S. 138–146 (Zitiert auf S. 37, 85, 90).

[WBH+20c] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani, V. Yussupov. „TOSCA Light: Bridging the Gap between the TOSCA Specification and Production-ready Deployment Technologies“. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*. SciTePress, Mai 2020, S. 216–226 (Zitiert auf S. 36, 37, 45, 85, 89, 90, 125).

[WBK+20] K. Wild, U. Breitenbücher, K. Képes, F. Leymann, B. Weder. „Decentralized Cross-Organizational Application Deployment Automation: An Approach for Generating Deployment Choreographies Based on Declarative Deployment Models“. In: *Proceedings of the 32nd Conference on Advanced Information Systems Engineering (CAiSE 2020)*. Bd. 12127. Lecture Notes in Computer Science. Springer International Publishing, Juni 2020, S. 20–35 (Zitiert auf S. 22, 155, 159, 193).

[WBL+21] B. Weder, J. Barzen, F. Leymann, M. Salm, K. Wild. „QProv: A provenance system for quantum computing“. In: *IET QuantumCommunication* 2.4 (Dez. 2021), S. 171–181 (Zitiert auf S. 25).

[WBLW20] B. Weder, U. Breitenbücher, F. Leymann, K. Wild. „Integrating Quantum Computing into Workflow Modeling and Execution“. In: *Proceedings of the 13th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2020)*. IEEE Computer Society, Dez. 2020, S. 279–291 (Zitiert auf S. 25).

[WCX15] X. Wang, J. Cao, Y. Xiang. „Dynamic cloud service selection using an adaptive learning mechanism in multi-cloud computing“. In: *Journal of Systems and Software* 100 (2015), S. 195–210 (Zitiert auf S. 17).

[WF12] T. Wellhausen, A. Fiesser. „How to Write a Pattern?: A Rough Guide for First-time Pattern Authors“. In: *Proceedings of the 16th European Conference on Pattern Languages of Programs (EuroPLoP 2011)*. ACM, Juli 2012 (Zitiert auf S. 19, 57).

[WK18] F. Willnecker, H. Krcmar. „Multi-Objective Optimization of Deployment Topologies for Distributed Applications“. In: *ACM Transactions on Internet Technology* 18.2 (2018) (Zitiert auf S. 38).

[WWB+13] T. Waizenegger, M. Wieland, T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, B. Mitschang, A. Nowak, S. Wagner. „Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing“. In: *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*. Springer, Sep. 2013, S. 360–376 (Zitiert auf S. 42).

[ZA05] U. Zdun, P. Avgeriou. „Modeling Architectural Patterns Using Architectural Primitives“. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*. ACM, Okt. 2005, S. 133–146 (Zitiert auf S. 60).

[ZBF+17] M. Zimmermann, U. Breitenbücher, M. Falkenthal, F. Leymann, K. Saatkamp. „Standards-based Function Shipping - How to use TOSCA for Shipping and Executing Data Analytics Software in Remote Manufacturing Environments“. In: *Proceedings of the 21st IEEE International Enterprise Distributed Object Computing Conference (EDOC 2017)*. IEEE, Okt. 2017, S. 50–60 (Zitiert auf S. 26).

[ZBG+18] M. Zimmermann, U. Breitenbücher, J. Guth, S. Hermann, F. Leymann, K. Saatkamp. „Towards Deployable Research Object Archives Based on TOSCA“. In: *Papers from the 12th Advanced Summer School on Service-Oriented Computing (SummerSoC 2018)*. IBM Research Division, Okt. 2018, S. 31–42 (Zitiert auf S. 26).

[ZBK+20] M. Zimmermann, U. Breitenbücher, K. Képes, F. Leymann, B. Weder. „Data Flow Dependent Component Placement of Data Processing Cloud Applications“. In: *Proceedings of the 2020 IEEE International Conference on Cloud Engineering (IC2E 2020)*. IEEE Computer Society, Apr. 2020, S. 83–94 (Zitiert auf S. 39, 126).

[Zdu07] U. Zdun. „Systematic Pattern Selection Using Pattern Language Grammars and Design Space Analysis“. In: *Software: Practice & Experience* 9 (Juli 2007), S. 983–1016 (Zitiert auf S. 152).

Alle URLs wurden zuletzt am 13.09.2022 geprüft.

Abbildungsverzeichnis

1.1 Vision des partnerübergreifenden Deployments. 14
1.2 Überblick der Forschungsbeiträge dieser Arbeit. 16

2.1 Imperatives und deklaratives Deployment-Modell. 29
2.2 Deployment-Prozess und dessen Basisaktivitäten (angelehnt an [ABL15]). 48
2.3 Verschiedene Bereiche in denen Automatisierungskonzepte von Mustern zum Einsatz kommen. 57
2.4 Prozess zur Formalisierung der Problemdomäne (adaptiert von Kim und El Khawand [KE07]). 63

3.1 Überblick über die DivA-Methode: Methodenschritte und resultierende Artefakte. 78
3.2 Rekursive Anwendung der DivA-Methode. 88
3.3 Umsetzung der DivA-Methode. 89

4.1 Erweitertes Metamodell basierend auf EDMM [WBF+19]. Alle mit (+) markierten Elemente wurden im Rahmen der Arbeit erweitert. Das mit (■) markierte Elemente ist eine Erweiterung aus einer anderen Arbeit [WBB+21]. 95

4.2 Beispiel eines GDM mit generierten Platzhaltern. 105
4.3 Beispiel für die Verwendung existierender lokaler Deployment-Modelle als Basis für die Verteilung. 110
4.4 Übersicht der SPLIT-AND-MATCH-Methode eines Deployment-Modells (adaptiert von Saatkamp et al. [SBKL17]). 112
4.5 Beispiel für das Ergebnis der SPLIT-Phase (links) und MATCH-Phase (rechts) mit zwei Zielumgebungen. 119

5.1 Grundkonzept der musterbasierten Problemerkennung und Modelladaption. 128
5.2 Beispiel für die musterbasierte Problemerkennung mit Auszug aus der Musterbeschreibung des SECURE-CHANNEL-Musters aus Schumacher et al. [SFH+06]. 131
5.3 Formalisierung eines Deployment-Modells. 133
5.4 Erkennen, Selektieren und Anwenden von Lösungen in Deployment-Modellen zum Beheben von Problemen (adaptiert von Saatkamp et al. [SBF+19]). 138
5.5 Programmiermodell und Artefakte zur Deploymentzeit (adaptiert von Saatkamp et al. [SBLW17]). 143
5.6 Modellierung und automatisierte Treiberselektion (adaptiert von Saatkamp et al. [SBLW17]). 145

6.1 Grundkonzept des automatisierten partnerübergreifenden dezentralen Deployments. 156
6.2 Aktivitäten (a) für Zielknoten von PKs, (b) für Quellknoten von PKs und (c) zur Initialisierung des gesamten Deployments (adaptiert von Wild et al. [WBK+20]). 159
6.3 Prozess zur Workflow-Generierung mit Integration des EDMM-Transformation-Frameworks zur Nutzung verschiedener Deployment-Technologien. Die dunkelgrau hinterlegten Schritte sind durch existierende Arbeiten abgedeckt [WBB+19; WBB+21]. 163

6.4 Deployment-Gruppen, die zusammengefügt werden können (links) und die nicht zusammengefügt werden können (rechts) (adaptiert von Saatkamp et al. [SBKL19b]). 164
6.5 Abgeleitete Provisionierungsreihenfolge aus LDM. 165

7.1 Grobarchitektur DIVA-Werkzeug. 170
7.2 Architektur des Modellierungswerkzeugs. 171
7.3 Architektur des Deployment-Werkzeugs (erweitert von Wurster et al. [WBB+21]). 174
7.4 Architektur und Verarbeitungsschritte mit zwei Partnern. . . 176
7.5 *Graphical Modeler* der Winery mit GDM. 177
7.6 LDM mit definierter Zielumgebung. 179
7.7 Ausschnitt aus Musterbeschreibung mit Prolog-Regel. 181
7.8 Ausschnitt aus Lösungsbeschreibung mit Prolog-Regel. 182
7.9 Anwendungsszenario zur Workflow-Generierung mit zwei Partnern und drei Technologien. 191
7.10 BPMN Kollaborationsdiagramm mit den generierten Workflows basierend auf den LDMs in Abbildung 7.9. 192

Tabellenverzeichnis

2.1 Überblick relevanter Cloud-Modellierungssprachen: Die gekennzeichneten Sprachen(*) wurden detailliert von Bergmayr et al. [BBF+18] analysiert. Die gekennzeichneten Mechanismen(+) sind Erweiterungen zu den nativ unterstützten Mechanismen. 32
2.2 Überblick verwandter Arbeiten und deren Verfahren zur Mustererkennung in Modellen. Die mit * gekennzeichneten Entwurfsmuster beziehen sich auf die Muster von Gamma et al. [GHJV94]. 59