Modellbasiertes Management von 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

Lukas Harzenetter

aus Memmingen

Hauptberichter: Prof. Dr. Dr. h. c. Frank Leymann

Mitberichter: Univ. Prof. Dr. Uwe Zdun

Tag der mündlichen Prüfung: 14. November 2023

Institut für Architektur von Anwendungssystemen
der Universität Stuttgart

2023

Inhaltsverzeichnis

1 Einleitung **13**
1.1 Ziel der Arbeit 16
1.2 Problemstellung und Forschungsbeiträge 17
1.3 Veröffentlichungen im Rahmen der Arbeit 24
1.4 Aufbau der Arbeit 28

2 Grundlagen und verwandte Arbeiten **29**
2.1 Modellgetriebene Entwicklung und Architektur 30
2.2 Bereitstellung und Management von Anwendungen 40
2.3 Muster, Mustersprachen und deren Verwendung 48
2.4 Generierung von Laufzeitmodellen 54
2.5 Zusammenfassung 57

3 Eine Methode für das Management von Anwendungen **59**
3.1 Die Lauffeuer-Methode 60
3.2 Varianten der Lauffeuer-Methode 67
3.3 Abgrenzung zu verwandten Arbeiten 69

4 Ein Metamodell für die musterbasierte Modellierung von Anwendungen 71
4.1 Das Essential Deployment Metamodel (EDMM) 72
4.2 EDMM-Erweiterung für die musterbasierte Modellierung 78
4.3 Modellierung von musterbasierten Deployment-Modellen 83

5 Verfeinern und Erkennen von Mustern 91
5.1 Konzeptidee und -übersicht 92
5.2 Metamodell für Verfeinerungsmodelle 94
5.3 Herausforderungen bei der Verfeinerung und Erkennung 97
5.4 Verfeinerungs- und Erkennungsmodelle 107
5.5 Verfeinerung von musterbasierten Deployment-Modellen 115
5.6 Erkennung von Mustern in Deployment-Modellen 123
5.7 Beispiel einer Verfeinerung von Mustern 127
5.8 Diskussion 130

6 Erweiterung von Anwendungen um Managementfunktionalitäten 135
6.1 Konzeptidee und -übersicht 136
6.2 Anreicherung von Managementfunktionalitäten 139
6.3 Generierung von Managementplänen 144
6.4 Management von zustandsbehafteten Anwendungen 146
6.5 Diskussion 150

7 Dynamisierung des Managements laufender Anwendungen 153
7.1 Konzeptidee und -übersicht 154
7.2 Ableiten von Instanzmodellen 157
7.3 Diskussion 164

8 Architektur und prototypische Umsetzung 167
8.1 Architektur des LAUFFEUER-Frameworks 168
8.2 Prototypische Implementierung 174
8.3 Validierung und Evaluation 178
8.4 Verwendung des Frameworks in anderen Arbeiten 183

9 Zusammenfassung und Ausblick 185

Literaturverzeichnis 189

Abbildungsverzeichnis 213

Mengenverzeichnis 215

Kurzzusammenfassung

Cloud-Anwendungen bestehen meist aus zahlreichen verschiedenen Komponenten, die zusammen die Funktionalität einer Anwendung umsetzen. Der Betrieb einer solchen Cloud-Anwendung umfasst dabei nicht nur die Bereitstellung der Anwendung, sondern auch ihr Management zur Laufzeit sowie ihre Außerbetriebnahme. Durch Deployment-Technologien lassen sich die Komponenten einer Cloud-Anwendung automatisiert und reproduzierbar bereitstellen sowie außer Betrieb nehmen. Dabei wird meist ein Deployment-Modell verwendet, welches den Zielzustand der Anwendung beschreibt und das von der entsprechenden Deployment-Technologie verarbeitet wird. Allerdings ist für das Erstellen eines Deployment-Modells Expertenwissen über die in der Anwendung verwendeten Komponenten und Technologien erforderlich, zum Beispiel welche Komponenten zusammenarbeiten können und welche Technologien sich gegenseitig ausschließen. Des Weiteren ist das Management ein zentraler Bestandteil einer sich im Betrieb befindlichen Anwendung. Wie auch für die Bereitstellung wird für das Management einer Anwendung Expertenwissen über die eingesetzten Komponenten und Technologien benötigt. Dabei können die Komponenten der Anwendung über mehrere Cloud-Anbieter und in-Haus-Lösungen verteilt sein und steigern somit die Komplexität dieser Aufgabe. Aktuelle Arbeiten und Deployment-Technologien unterstützten das automatisierte

Management von verteilten Anwendungskomponenten jedoch kaum. Da die Komponenten einer Anwendung darüber hinaus meist von mehreren interdisziplinären Teams entwickelt werden, werden zudem unterschiedliche Deployment-Technologien für einzelne Komponenten eingesetzt. Dies erschwert das ganzheitliche Management einer Anwendung zusätzlich.

In der vorliegenden Dissertation werden diese Herausforderungen angegangen, indem eine Methode vorgestellt wird, mit der (i) Anwendungen modelliert werden können, ohne dass Expertenwissen über Komponenten, Technologien und deren Konfigurationen benötigt wird. Anstelle der konkreten Komponenten und Technologien können Deployment-Modelle mit Muster modelliert werden, die abstrakte Lösungen zu wiederkehrenden Problemen beschreiben. Anschließend werden die Muster in einem solchen musterbasierten Deployment-Modell automatisiert zu passenden Komponenten und Technologien verfeinert. Bevor eine Anwendung im nächsten Schritt automatisiert bereitgestellt wird, kann das nun ausführbare Deployment-Modell (ii) automatisiert um zusätzliche Managementfunktionalitäten erweitert werden. Diese können zur Laufzeit der Anwendung ausgeführt werden und unterstützen sowohl verteilte Multi-Cloud-Anwendungen als auch Anwendungen, die mit mehreren Deployment-Technologien bereitgestellt werden. Zum Beispiel wird ein Managementkonzept eingeführt, das (iii) Anwendungen im aktuellen Zustand außer Betrieb nehmen und im selben Zustand wieder bereitstellen kann. Neben neu modellierten Anwendungen können die in dieser Arbeit vorgestellten Konzepte auch (iv) auf bereits laufende Anwendungen übertragen und angewendet werden. Die Umsetzbarkeit der Konzepte wird durch einen Open-Source Prototyp gezeigt.

Abstract

Cloud applications usually consist of numerous components that together form the functionality of an application. The lifecycle of such application includes not only the application's provisioning and decommissioning, but also its management at runtime. Today's deployment technologies enable the automated provisioning and decommissioning of such applications. Therefore, deployment models describing the target state of an application are usually created by developers and interpreted by corresponding deployment technologies. However, creating a deployment model requires expert knowledge about an application's components and technologies. For example, it must be known which components can work together and which technologies are mutually exclusive. Furthermore, management is a key element during the operation of an application. Similar to the deployment of an application, its management requires expert knowledge about the used components and technologies. Thereby, the components of an application may be distributed across multiple cloud providers and on-premise solutions, further increasing the complexity of the application's management. However, deployment technologies hardly supported the automated management of distributed applications. Moreover, since the components of an application are usually developed by several interdisciplinary teams, different deployment technologies are used, further complicating a holistic application management.

This dissertation addresses these challenges by presenting a method to (i) model applications without requiring expert knowledge about components, technologies, and their configurations. Instead of concrete components and technologies, deployment models can be modeled using patterns that describe abstract solutions to recurring problems. In the next step, the patterns in such a pattern-based deployment model are automatically refined to suitable components and technologies. The now executable deployment model can be (ii) automatically enriched with additional management functionalities, before the modeled application is automatically deployed. These can be executed during the application's runtime and support both distributed multi-cloud applications and applications deployed using multiple deployment technologies. Moreover, a management concept is introduced that can (iii) freeze applications in their current state and restore them in the same state as they were terminated. In addition to newly modeled applications, the concepts presented in this dissertation can also be (iv) applied to applications that are already running. The feasibility of the concepts is shown by an open-source prototype.

Danksagungen

Ich möchte mich bei allen herzlich für die große Unterstützung, Fürsprache und auch Abwechslung danken, die mich in den letzten Jahren während der Erstellung dieser Dissertation begleitet haben. Mein ganz besonderer Dank gilt meinem Betreuer und Doktorvater Prof. Dr. Dr. h. c. Frank Leymann. Vielen Dank für Deine stetige Ermutigung und die Möglichkeit, am IAAS forschen und arbeiten zu dürfen. Bei meinem Zweitgutachter Univ. Prof. Dr. Uwe Zdun möchte ich mich für Übernahme des Mitberichts bedanken.

Ein großes Dankeschön geht an meine Kolleginnen und Kollegen am Institut. Vor allem möchte ich meinem Mentor Prof. Dr. Uwe Breitenbücher für seine zahlreichen Denkanstöße und unermüdliche Unterstützung danken. Vielen Dank auch an Dr. Karoline Wild, Michael Wurster, Kálmán Képes und Benjamin Weder für Eure Freundschaft und Euren Rat in jeglicher Situation.

Mein besonderer Dank gilt meinen Eltern Ulrike und Erich. Danke, dass Ihr mich immer unterstützt habt und durch zahlreiche Ermutigungen mich bis zum Abschluss dieser Arbeit begleitet habt. Schließlich möchte ich meinen Brüdern, meiner Familie und meinen Freunden für den Zuspruch und die nötige Abwechslung danken, die mich immer wieder mit neuen Augen und neuem Engagement auf meine Arbeit haben sehen lassen.

Kapitel 1

Einleitung

In einer modernen Gesellschaft sind komplexe Anwendungen aus dem Alltag nicht mehr wegzudenken: Täglich werden mehrere Milliarden E-Mails versendet, Video-Konferenzen abgehalten oder Musik aus dem Internet gestreamt. Auch nach Feierabend wird die Nutzung digitaler Inhalte und Anwendungen nicht weniger, denn selbst Kühlschränke sind mittlerweile in der Lage, automatisiert Einkaufslisten zu generieren und entsprechende Benachrichtigungen an die Smartphones ihrer Benutzenden zu senden. All diese Anwendungen, deren Komplexität für die meisten Benutzenden nicht wahrnehmbar ist, müssen jedoch zunächst entwickelt und bereitgestellt werden. Neben der Bereitstellung (engl. „*Deployment*") selbst fallen während des Betriebs einer Anwendung jedoch zusätzliche Managementaufgaben an, um einen reibungslosen Betrieb der Anwendung zu gewährleisten. So müssen beispielsweise die Komponenten einer Anwendung skaliert werden, damit die Verfügbarkeit und eine hohe Verarbeitungsgeschwindigkeit der gesamten Anwendung sichergestellt werden kann. Auch regelmäßige Aufgaben, wie das Erstellen von Datenbank-Backups und das Installieren von Sicherheitsupdates, gehören zum Anwendungsmanagement und helfen dabei die Ausfallsicherheit der gesamten Anwendung zu erhöhen.

Im Allgemeinen ist die Bereitstellung und das Management von Anwendungen eine komplexe, zeitintensive und fehleranfällige Aufgabe [BBF+18; Opp03]. Daher wurden in den letzten Jahren Technologien entwickelt, die die Bereitstellung und Außerbetriebnahme von Anwendungen automatisieren. Dazu gehören unter anderem Puppet [Pup23], Terraform [Has23] oder Kubernetes [The23]. Spätestens durch das Aufkommen des Cloud-Computing-Paradigmen und der damit verbundenen Möglichkeit, IT-Ressourcen auf Abruf anzumieten und wieder freizugeben [Ley09], bekam die automatisierte Bereitstellung von Anwendungen auch einen finanziellen Aspekt: Im Gegensatz zu privaten IT-Ressourcen eines Unternehmens, welche hohe Kosten durch Anschaffung, Betrieb und Wartung verursachen, werden bei Cloud-Ressourcen ausschließlich die tatsächlich genutzten Einheiten in Rechnung gestellt. Somit können die Kosten einer Cloud-Anwendung durch deren automatisierte Bereitstellung und Außerbetriebnahme minimiert werden.

Um Anwendungen automatisiert bereitzustellen, verwenden Deployment-Technologien üblicherweise *Deployment-Modelle* [BBF+18]. Diese beschreiben meist die Struktur einer Anwendung [WBF+19] und werden von den Deployment-Technologien interpretiert, um die notwendigen Schritte für die Bereitstellung abzuleiten [EBF+17]. Die Erstellung solcher Deployment-Modelle erfordert jedoch Wissen über zahlreiche technische Details der verwendeten Komponenten sowie deren mögliche Konfigurationen. Insbesondere wenn die zu modellierende Anwendung aus mehreren Hundert Komponenten besteht und zahlreiche Technologien und Dienste diverser Cloud-Anbieter integriert werden sollen, stellt dies eine große Herausforderung dar. Beispielsweise muss die Kompatibilität der unterschiedlichen Komponenten untereinander bekannt sein: Kann eine .NET-Anwendung von Googles App Engine betrieben werden? Welche Versionen zweier Komponenten sind miteinander kompatibel? Welche Datenbanken werden unterstützt?

Generell müssen in verschiedenen Anwendungen meist ähnliche oder sogar die gleichen Probleme gelöst werden, zum Beispiel um die Kommunikation mehrerer verteilter Komponenten zu ermöglichen. Dafür verweisen Architekturbeschreibungen meist auf bewährte Lösungen, die in *Mustern* dokumentiert sind [BCK03]. Muster (engl. „*Patterns*“) beschreiben abstrakte

Lösungen zu wiederkehrenden Problemen in einer spezifischen Domäne auf technologieunabhängige Art und Weise [AIS77]. Im Cloud-Umfeld definieren beispielsweise die „Cloud Computing Patterns“ [FLR+14] Lösungsansätze, wie Anwendungen die Cloud-Prinzipien und Cloud-Ressourcen nutzen können. Zur Umsetzung solcher Muster während der Erstellung von Deployment-Modellen müssen Modellierende die Muster auf konkrete Technologien und Dienste von Cloud-Anbietern sowie entsprechender Konfigurationen abbilden. Dieser Schritt in der Erstellung eines Deployment-Modells erfordert zusätzlich zum tiefen technischen Wissen über unterschiedliche Technologien und deren Kompatibilitäten untereinander auch das Verständnis, wie die Muster in und mit den jeweiligen Technologien umgesetzt werden können.

Der Betrieb einer Anwendung umfasst allerdings nicht nur ihre Bereitstellung und Außerbetriebnahme, sondern auch ihr Management zur Laufzeit [Cha08]. Dabei ist das Management von Anwendungen und insbesondere von Multi-Cloud-Anwendungen, d. h. Anwendungen, die mehrere Cloud-Services von unterschiedlichen Anbietern verwenden, meist eine manuelle Aufgabe. Aktuelle Deployment-Technologien unterstützen lediglich die Ausführung einfacher Managementaufgaben wie die Skalierung einzelner Komponenten. Komplexe Aufgaben wie das Erstellen von Backups aller zustandsbehafteten Komponenten einer Anwendung, die über verschiedene Umgebungen verteilt sind, werden aktuell kaum unterstützt und müssen von Modellierenden manuell umgesetzt und ausgeführt werden.

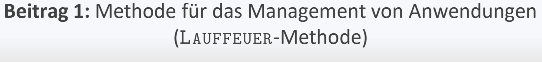
Auch die Erweiterbarkeit von Anwendungen um zusätzliche und ganzheitliche Managementfunktionalitäten, d. h. Managementaufgaben, die für die gesamte Anwendung ausgeführt werden können, ist aktuell nur manuell möglich. Das gilt sowohl für Anwendungen, deren Deployment-Modelle bekannt sind, als auch für bereits laufende Anwendungen. Zwar existieren Ansätze, die laufende Anwendungen von einer Umgebung in eine andere portieren können [AGH+15; CFH+05; HKF18; SPP+22], jedoch werden generische Managementaufgaben kaum unterstützt. Insbesondere für Anwendungen, deren Komponenten über verschiedene Umgebungen verteilt sind und möglicherweise mehrere Deployment-Technologien nutzen, fehlen aktuell Konzepte, die die Anwendungen automatisiert verwalten können.

1.1 Ziel der Arbeit

Die Modellierung und das Management von Multi-Cloud-Anwendungen sind aufgrund zahlreicher Technologien und Dienste von Cloud-Anbietern sehr komplex und fehleranfällig. Dabei reichen die aktuellen Herausforderungen von der Auswahl, Modellierung und Konfiguration benötigter Komponenten und Technologien im Deployment-Modell einer Anwendung bis hin zum automatisierten Management von Anwendungen zur Laufzeit. Das Ziel dieser Arbeit ist es daher, (i) die Erstellung von Deployment-Modellen ohne tiefgreifendes Expertenwissen über verfügbare Technologien und Cloud-Dienste zu ermöglichen, (ii) das Management von Multi-Cloud-Anwendungen wiederverwendbar zu automatisieren sowie (iii) bereits laufende Anwendungen um zusätzliche Managementaufgaben ergänzen und ausführen zu können.

Um das Ziel dieser Arbeit umzusetzen, wird eine Methode eingeführt, die es ermöglicht, (i) das Deployment-Modell einer Anwendung abstrakt mithilfe von Mustern zu beschreiben und diese (ii) automatisiert zu konkreten Technologien, Cloud-Diensten und deren Konfigurationen zu verfeinern. Anschließend können sowohl (iii) Deployment-Modelle als auch (iv) bereits laufende Anwendungen mit zusätzlichen Managementfunktionalitäten erweitert werden, die zur Laufzeit automatisiert ausgeführt werden können.

Durch das neue Modellierungskonzept können Deployment-Modelle anstatt mit konkreten Technologien und Cloud-Diensten mit Muster modelliert werden. Ein solches *musterbasiertes Deployment-Modell* kann anschließend automatisiert zu konkreten Komponenten und Diensten von Cloud-Anbieter verfeinert werden. Dadurch müssen Modellierende die konkreten Umsetzungen der Muster mit bestimmten Technologien nicht selbst kennen. Anschließend können Deployment-Modelle automatisiert um Managementfunktionalitäten erweitert werden, die die Anwendung zur Laufzeit verwalten. Dabei steht die Wiederverwendbarkeit der Managementfunktionalitäten im Vordergrund, um den Entwicklungsaufwand gering zu halten und die Auswahl an verfügbaren Managementfunktionalitäten kontinuierlich zu steigern. Auch können Anwendungen zur Laufzeit, d. h. nachdem sie bereitgestellt wurden, mit weiteren Managementfunktionalitäten ergänzt und verwaltet werden.



Beitrag 2: Musterbasierte Modellierung von Deployment-Modellen

Beitrag 3: Erkennen und Verfeinern von Mustern in Deployment-Modellen

Beitrag 4: Erweiterung von Deployment-Modellen um Management-funktionalitäten

Beitrag 5: Automatisiertes Management für laufende Anwen-dungen nachträglich ermöglichen

Beitrag 6: Architektur und prototypische Implementierung

Abbildung 1.1: Übersicht der Forschungsbeiträge

1.2 Problemstellung und Forschungsbeiträge

Um das zuvor skizzierte Ziel der Arbeit zu realisieren, werden im Folgenden die Problemstellungen beschrieben, die im Rahmen der Arbeit gelöst wurden. Abbildung 1.1 zeigt die einzelnen Forschungsbeiträge und wie diese zusammenhängen: Forschungsbeitrag 1 beschreibt die LAUFFEUER-Methode. Diese wird von den Forschungsbeiträgen 2 bis 5 umgesetzt und wird von Forschungsbeitrag 6 prototypisch umgesetzt. Forschungsbeitrag 2 definiert ein Metamodell, um Deployment-Modelle auf Basis der Muster, die für die zu modellierende Anwendung umgesetzt werden sollen, beschreiben zu können. Forschungsbeitrag 3 ermöglicht die automatisierte Verfeinerung von Mustern zu konkreten Komponenten in Deployment-Modellen und somit die automatisierte Umsetzung der Muster in konkrete Technologien und deren Konfigurationen. Darüber hinaus ermöglich Forschungsbeitrag 3 die Erkennung umgesetzer Muster in Deployment-Modellen. Forschungsbeitrag 4 führt eine Methode ein, um automatisiert (i) wiederverwendbare Managementfunktionalitäten zu identifizieren, (ii) ausgewählte Mangementfuunktionalitäten zu einer Anwendung hinzuzufügen und (iii) diese auszuführen. Forschungsbeitrag 5 zeigt, wie die Konzepte auf laufende Anwendungen angewendet werden können, während Forschungsbeitrag 6 alle Konzepte implementiert.

1.2.1 Methode für das Management von Anwendungen

Das Management von Anwendungen wurde von der Modellierung über die Bereitstellung bis hin zur Ausführung von Managementaufgaben in zahlreichen Arbeiten behandelt [BBF+18; BBK+14; KBL+19; KBL17]. Dafür werden Deployment-Modelle eingesetzt, die die Struktur einer Anwendung anhand ihrer Komponenten und deren Abhängigkeiten zueinander beschreiben. Die Erstellung solcher Deployment-Modelle erfordert jedoch tiefes technisches Wissen über verwendete Technologien sowie deren Konfigurationsmöglichkeiten. Darüber hinaus ist das Management von Anwendungen meist auf einfache Managementaufgaben wie das Skalieren einzelner Komponenten in einer bestimmten Umgebung beschränkt. Komplexe Managementaufgaben, insbesondere für Multi-Cloud-Anwendungen, werden kaum unterstützt. Auch die Erweiterbarkeit von sich bereits im Betrieb befindlichen Anwendungen um Managementfunktionalitäten ist bisher nicht möglich.

> **Forschungsbeitrag 1** (Methode für das Management von Anwendungen). In der vorliegenden Arbeit wird die LAUFFEUER-Methode (Laufende Anwendungen einfrieren, auftauen und verwalten) vorgestellt. Sie beschreibt, wie komplexe Anwendungen ohne tiefere technische Kenntnisse auf Basis von Mustern modelliert und im Anschluss automatisiert bereitgestellt und verwaltet werden können. Dabei kombiniert sie die in den einzelnen Forschungsbeiträgen 2 bis 5 eingeführten Konzepte und ermöglicht die Nutzung der einzelnen Beiträge in einem ganzheitlichen Zusammenhang. Die Methode ist (i) unabhängig von konkreten Technologien, (ii) ermöglicht neben der Konkretisierung von abstrakten Deployment-Modellen auch deren Abstraktion und (iii) kann sowohl mit Deployment-Modellen als auch mit Instanzmodellen unterschiedlicher Anwendung verwendet werden. Damit können die Konzepte der Forschungsbeiträge 2 bis 5 auch auf bereits laufende Anwendungen angewendet werden.

1.2.2 Musterbasierte Modellierung von Deployment-Modellen

Um die Bereitstellung von Anwendungen zu beschreiben, werden im Allgemeinen zwei Varianten von Deployment-Modellen unterschieden: *imperative* und *deklarative* Modelle [EBF+17]. In der Praxis haben sich deklarative Deployment-Modelle gegenüber den imperativen durchgesetzt: Bei einer Analyse der 13 meistgenutzten Deployment-Technologien wurde festgestellt, dass (i) alle 13 Technologien deklarative Modelle für die Beschreibung der Anwendung verwenden und (ii) jedes dieser Modelle auf ein gemeinsames Metamodell abgebildet werden kann – das technologieunabhängige *Essential Deployment Meta Model (EDMM)* [WBF+19]. Deployment-Modelle, die auf Basis von EDMM modelliert wurden, können anschließend automatisiert in die Modelle dieser 13 Deployment-Technologien transformiert und ausgeführt werden [WBB+19; WBB+20b]. Damit jedoch auch Muster während der Modellierung von Anwendungen unterstützt werden, bedarf es (i) eines Verfahrens, wie Muster in einem Deployment-Modell verwendet werden können sowie (ii) einer Erweiterung von EDMM zur Entwurfszeit.

> **Forschungsbeitrag 2** (Musterbasierte Modellierung von Deployment-Modellen). In diesem Forschungsbeitrag wird eine Erweiterung von EDMM vorgestellt, die es ermöglicht, neben konkreten Komponenten und deren Beziehungen auch Muster in einem Deployment-Modell zu verwenden. Dies erleichtert die Modellierung und das Lesen eines Deployment-Modells, da die Semantik der Komponenten und der gesamten Anwendung, d. h. welche konzeptionellen Ansätze und Lösungen sie umsetzen, direkt aus den verwendeten Mustern ersichtlich ist, anstatt diese aus den technischen Details der verwendeten Technologien und deren Konfigurationen interpretieren zu müssen. Durch die Verwendung von EDMM als Grundlage des neuen Modellierungskonzepts, können alle Forschungsbeiträge mit den 13 meistgenutzten Deployment-Technologien realisiert werden und sind daher deployment-technologieunabhängig.

1.2.3 Verfeinern und Erkennen von Mustern in Deployment-Modellen

Deployment-Modelle, die Muster beinhalten, um Komponenten und Konfiguration zu beschreiben, können von Deployment-Technologien nicht ausgeführt werden. Die Muster müssen zuvor zu konkreten Technologien und passenden Konfigurationen verfeinert werden. Um diesen Schritt zu automatisieren, bedarf es eines Verfahrens, das Muster in einem Deployment-Modell identifiziert und diese automatisiert zu konkreten Technologien verfeinert. Dabei müssen nicht nur Technologien ausgewählt, sondern insbesondere auch die Kompatibilitäten zwischen den einzelnen Technologien berücksichtigt werden. Die Umsetzungen der Muster mit konkreten Technologien und entsprechender Konfigurationen müssen dabei in maschinenlesbaren Dokumenten definiert werden. Daher wird ein zusätzliches Metamodell benötigt, das Muster auf konkrete Technologien abbildet.

> **Forschungsbeitrag 3** (Verfeinern und Erkennen von Mustern in Deployment-Modellen). Im dritten Forschungsbeitrag wird ein Verfahren eingeführt, das es ermöglicht, die mithilfe von Mustern definierten Deployment-Modelle zu ausführbaren Deployment-Modellen zu verfeinern. Dafür wird ein Metamodell definiert, welches eine Kombination von unterschiedlichen Mustern und Komponenten auf konkrete Technologien und entsprechende Konfigurationen abbildet. Mithilfe dieser *Verfeinerungsmodelle* ist es nicht nur möglich, Muster mit Technologien umzusetzen, sondern auch Muster in bereits ausführbaren Deployment-Modellen zu erkennen. Dieser Schritt ermöglicht es daher, auch in existierenden Deployment-Modellen festzustellen, ob alle in den Anforderungen einer Anwendung geforderten Konzepte und Verhaltensweisen umgesetzt wurden. Neben der automatisierten Umsetzung und Erkennung von Mustern bieten Verfeinerungsmodelle darüber hinaus auch die Möglichkeit, Wissen über die Realisierung der Muster mit konkreten Technologien zu dokumentieren und können dadurch als Nachschlagewerk und Lernmittel verwendet werden.

1.2.4 Erweiterung von Deployment-Modellen um Managementfunktionalitäten

Deployment-Technologien wie Terraform, Chef oder Kubernetes bieten meist nur einfache Managementfunktionalitäten – zum Beispiel das Skalieren einzelner Komponenten einer Anwendung. Ganzheitliche Managementfunktionalitäten, die mehrere Komponenten betreffen, werden kaum unterstützt. Aufgaben, wie das Erstellen eines Backups aller zustandsbehafteten Komponenten, müssen daher aktuell komplett manuell implementiert und ausgeführt werden. Ändern sich jedoch einzelne Komponenten in einer Anwendung, so müssen auch die manuell erstellten Managementfunktionalitäten angepasst werden. Je größer die Anwendungen und je umfangreicher die Änderungen sind, desto komplexer, zeitintensiver und fehleranfälliger werden die manuellen Anpassungen an den Managementautomatisierungen. Das Ziel dieses Forschungsbeitrags ist es daher, Managementfunktionalitäten (i) für einzelne Komponenten wiederverwendbar zu definieren, diese (ii) selektiv und automatisiert zu einem existierenden Deployment-Modell hinzufügen zu können und (iii) ausführbare Managementworkflows für jede hinzugefügte Managementfunktionalität zu generieren.

> **Forschungsbeitrag 4** (Erweiterung von Deployment-Modellen um Managementfunktionalitäten). Dieser Forschungsbeitrag präsentiert ein Verfahren, um existierende Deployment-Modelle automatisiert mit verfügbaren Managementfunktionalitäten zu erweitern. Anschließend können auf Basis des erweiterten Deployment-Modells ausführbare Managementworkflows für jede Funktionalität generiert werden. Dadurch kann jede Managementfunktionalität individuell für eine Anwendung ausgeführt werden. Wird eine Komponente einer Anwendung geändert, so können die verfügbaren Managementfunktionalitäten wieder automatisiert angereichert und die entsprechenden Managementworkflows neu generiert werden. Daher sind keine zusätzliche manuellen Managementschritte mehr nötig.

1.2.5 Automatisiertes Management für laufende Anwendungen nachträglich ermöglichen

Um die Forschungsbeiträge 3 und 4 auch für bereits laufende Anwendungen zu ermöglichen, müssen einige Voraussetzungen erfüllt sein: (i) Erstens muss ein *Instanzmodell* der Anwendung existieren. Dabei wird die Eigenschaft von Instanz- und Deployment-Modellen ausgenutzt, dass sie auf denselben Metamodellen basieren können [Bin15]. Generell können Instanzmodelle durch zahlreiche Ansätze automatisiert erstellt werden [BBKL13; Bin15; HBLE14; MDW+00]. Während Deployment-Technologien wie Chef, Puppet oder Kubernetes die von ihnen bereitgestellte Anwendung überwachen und entsprechende Laufzeitinformationen der Anwendung vorhalten, nutzt keiner der existierenden Ansätze diese Informationen, um ein Instanzmodell der Anwendung zu erstellen. Wird durch das Ausführen einer Managementfunktionalität der Zustand der Anwendung verändert, können diese Deployment-Technologien die Änderung erkennen und den ursprünglichen Zustand wiederherstellen. Daher müssen die Deployment-Technologien (ii) im Instanzmodell berücksichtigt und (iii) in die Ausführung der Managementfunktionalitäten eingebunden werden.

> **Forschungsbeitrag 5** (Automatisiertes Management für laufende Anwendungen nachträglich ermöglichen). Dieser Forschungsbeitrag beschreibt ein Verfahren, wie laufende Anwendungen, welche durch Deployment-Technologien bereitgestellt wurden, zur Laufzeit mit zusätzlichen Managementfunktionalitäten erweitert werden können. Dafür wird in ein Verfahren präsentiert, das die Besonderheiten von Instanzmodellen während der Ausführung von Managementfunktionalitäten berücksichtigt. Dabei wird gezeigt, wie ein EDMM-basiertes Instanzmodell einer Anwendung zunächst erzeugt und anschließend vervollständigt werden kann. Auf Basis dieses Instanzmodells können im nächsten Schritt die Konzepte zur Mustererkennung und Erweiterung von Managementfunktionalitäten angewendet werden.

1.2.6 Architektur und prototypische Implementierung

Im Fokus dieser Arbeit steht die Automatisierung der manuellen Aufgaben während der Modellierung, Bereitstellung und dem Management von Multi-Cloud-Anwendungen. Um die praktische Umsetzbarkeit der vorgestellten Konzepte zu zeigen, beschreibt der letzte Forschungsbeitrag dieser Arbeit die Architektur und prototypische Implementierung der einzelnen Forschungsbeiträge. Dabei wurden das OpenTOSCA Ökosystem [BEK+16], bestehend aus dem Modellierungswerkzeug *Winery* [KBBL13], dem Deployment-System *OpenTOSCA Container* [BBH+13] und der Verwaltungsoberfläche *Vinothek* [BBKL14], das *EDMM-Transformation Framework* [WBB+19] sowie der *Pattern Atlas* [LB21] erweitert. Alle Werkzeuge sind Open-Source-Projekte und können über GitHub eingesehen und frei verwendet werden.

> **Forschungsbeitrag 6** (Architektur und prototypische Implementierung). Zur Validierung der vorgestellten Konzepte beschreibt dieser Forschungsbeitrag die Architektur und Implementierung des LAUFFEUER-Frameworks, das die Forschungsbeiträge 1 bis 5 umsetzt. Dabei wird das Zusammenspiel des Pattern Atlas, des OpenTOSCA Ökosystems und des EDMM-Transformation Frameworks gezeigt. Insbesondere werden die Erweiterungen zur (i) musterbasierten Modellierung von Deployment-Modellen, (ii) Verfeinerung und Erkennung von Mustern in Deployment-Modellen, (iii) die automatisierte Erweiterung von Deployment- und Instanzmodellen sowie (iv) die Generierung von Managementworkflows vorgestellt. Darüber hinaus ermöglicht die Realisierung der Forschungsbeiträge im OpenTOSCA Ökosystem sowie des EDMM-Transformation Framework die Nutzung anderer Forschungsarbeiten. Dadurch erweitern sich die Möglichkeiten für das Bereitstellen und Verwalten von Anwendungen, da die modellierten Anwendungen mit jeder der 13 Deployment-Technologien automatisiert bereitgestellt und durch das vorgestellte LAUFFEUER-Framework automatisiert verwaltet werden können.

1.3 Veröffentlichungen im Rahmen der Arbeit

Im Rahmen der vorliegenden Arbeit wurden die folgenden begutachteten Publikationen in Fachzeitschriften und auf Konferenzen veröffentlicht, die Teile der in dieser Arbeit eingeführten Konzepte beschreiben:

1. [HBF+18] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, C. Krieger und F. Leymann. „Pattern-based Deployment Models and Their Automatic Execution". In: *Proceedings of the 11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2018)*. IEEE Computer Society, Dez. 2018, S. 41–52
2. [HBKL19] L. Harzenetter, U. Breitenbücher, K. Képes und F. Leymann. „Freezing and Defrosting Cloud Applications: Automated Saving and Restoring of Running Applications". In: *Software-Intensive Cyber-Physical Systems (SICS)* 35 (Aug. 2019), S. 101–114
3. [HBL+19] L. Harzenetter, U. Breitenbücher, F. Leymann, K. Saatkamp, B. Weder und M. Wurster. „Automated Generation of Management Workflows for Applications Based on Deployment Models". In: *Proceedings of the 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC 2019)*. IEEE, Okt. 2019, S. 216–225
4. [HBF+20] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth und 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
5. [HBB+21] L. Harzenetter, T. Binz, U. Breitenbücher, F. Leymann und M. Wurster. „Automated Generation of Management Workflows for Running Applications by Deriving and Enriching Instance Models". In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*. SciTePress, Mai 2021, S. 99–110
6. [HBF+21] L. Harzenetter, U. Breitenbücher, G. Falazi, F. Leymann und A. Wersching. „Automated Detection of Design Patterns in Declarative

Deployment Models". In: *Proceedings of the 14th IEEE/ACM International Conference on Utility Cloud Computing (UCC 2021)*. ACM, Dez. 2021, S. 36–45

7. [HBBL23] L. Harzenetter, U. Breitenbücher, T. Binz und F. Leymann. „An Integrated Management System for Composed Applications Deployed by Different Deployment Automation Technologies". In: *SN Computer Science* 4.370 (Apr. 2023), S. 1–16

Die im Folgenden aufgeführten Publikationen zeigen Zusammenarbeiten, die die in dieser Arbeit vorgestellten Konzepte beeinflusst haben. Einige der folgenden Publikationen bauen auf den hier vorgestellten Konzepten auf:

1. [FBF+18] G. Falazi, U. Breitenbücher, M. Falkenthal, L. Harzenetter, F. Leymann und V. Yussupov. „Blockchain-based Collaborative Development of Application Deployment Models". In: *On the Move to Meaningful Internet Systems: OTM 2018 Conferences (CoopIS 2018)*. Springer, Okt. 2018, S. 40–60
2. [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 Services Science (CLOSER 2019)*. SciTePress, Mai 2019, S. 495–506
3. [WBB+19] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani und V. Yussupov. „The EDMM Modeling and Transformation System". In: *Service-Oriented Computing – ICSOC 2019 Workshops*. Springer, Dez. 2019
4. [WBH+20a] K. Wild, U. Breitenbücher, L. Harzenetter, F. Leymann, D. Vietz und 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

5. [WBH+20b] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann und 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, Aug. 2020, S. 138–146
6. [WBH+20c] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani und 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
7. [ZBH+20] M. Zimmermann, U. Breitenbücher, L. Harzenetter, F. Leymann und V. Yussupov. „Self-Contained Service Deployment Packages“. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*. SciTePress, Mai 2020, S. 371–381
8. [YBB+22] V. Yussupov, U. Breitenbücher, A. Brogi, L. Harzenetter, F. Leymann und J. Soldani. „Serverless or Serverful? A Pattern-based Approach for Exploring Hosting Alternatives“. In: *Proceedings of the 16th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2022)*. Springer, Okt. 2022

Darüber hinaus wurden weitere Publikationen in Rahmen der Forschungsarbeit am Institut und in Zusammenarbeit mit der Universität zu Köln im Laufe des DFG Forschungsprojekts *SustainLife*[1] veröffentlicht:

1. [NHS+18] C. Neuefeind, L. Harzenetter, P. Schildkamp, U. Breitenbücher, B. Mathiak, J. Barzen und F. Leymann. „The SustainLife Project – Living Systems in Digital Humanities“. In: *Papers From the 12th Advanced Summer School of Service-Oriented Computing (SummerSOC 2018)*. IBM Research Division, Okt. 2018, S. 101–112

[1] https://www.iaas.uni-stuttgart.de/forschung/projekte/sustainlife

2. [NSM+18] C. Neuefeind, P. Schildkamp, B. Mathiak, J. Barzen, U. Breitenbücher, L. Harzenetter und F. Leymann. „Lebende Systeme in den Digital Humanities – das Projekt SustainLife“. In: *20. Workshop Software-Reengineering und -Evolution (WSRE 2018) der GI-Fachgruppe Software-Reengineering, Bad-Honnef, 02.-04. Mai 2018, Proceedings*. GI Gesellschaft für Informatik e.V. (GI), Mai 2018, S. 37–38
3. [NSM+19a] C. Neuefeind, P. Schildkamp, B. Mathiak, L. Harzenetter, J. Barzen, U. Breitenbücher und F. Leymann. „Technologienutzung im Kontext Digitaler Editionen. Eine Landschaftsvermessung“. In: *DHd 2019 Digital Humanities: multimedial & multimodal. Konferenzabstracts*. Zenodo, März 2019, S. 219–222
4. [NSM+19b] C. Neuefeind, P. Schildkamp, B. Mathiak, A. Marčić, F. Hentschel, L. Harzenetter, J. Barzen, U. Breitenbücher und F. Leymann. „Sustaining the Musical Competitions Database: A TOSCA-based Approach to Application Preservation in the Digital Humanities“. In: *Book of Abstracts of the Digital Humanities Conference 2019 (DH2019), Utrecht, 9.7. – 12.7.2019*. ADHO, Juli 2019
5. [SHB+20] P. Schildkamp, L. Harzenetter, U. Breitenbücher, F. Leymann, B. Mathiak und C. Neuefeind. „Modellierung und Verwaltung von DH-Anwendungen in TOSCA“. In: *DHd 2020 Spielräume: Digital Humanities zwischen Modellierung und Interpretation. Konferenzabstracts*. Zenodo, Feb. 2020, S. 36–38
6. [SHM+20] P. Schildkamp, L. Harzenetter, B. Mathiak, C. Neuefeind, J. Barzen, U. Breitenbücher und F. Leymann. „Workshop on Modelling and Maintaining Research Applications in TOSCA“. In: *DH2020*. ADHO, Juli 2020
7. [FHS+21] A. Fischer, L. Harzenetter, P. Schildkamp, U. Breitenbücher, C. Neuefeind, F. Leymann und B. Mathiak. „Modeling as a Sustainability Strategy for DH Software Applications“. In: *Book of Abstracts of the 2nd International Conference of the European Association for Digital Humanities (EDAH 2021)*. ADHO, Sep. 2021

8. [BBH+22] F. Bühler, J. Barzen, L. Harzenetter, F. Leymann und P. Wundrack. „Combining the Best of Two Worlds: Microservices and Micro Frontends as Basis for a New Plugin Architecture“. In: *Proceedings of the 16th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2022)*. Springer, Okt. 2022, S. 3–23

1.4 Aufbau der Arbeit

Im Folgenden werden zunächst in Kapitel 2 die Grundlagen vorgestellt und verwandte Arbeiten diskutiert. Danach wird in Kapitel 3 die LAUFFEUER-Methode vorgestellt. Anschließend wird das auf EDMM basierende Metamodell in Kapitel 4 eingeführt. Um in musterbasierten Deployment-Modellen beschriebene Anwendungen automatisiert bereitstellen zu können, präsentiert Kapitel 5 die automatisierte Verfeinerung der Muster zu konkreten Technologien. Die dabei entstehenden ausführbaren Deployment-Modelle können anschließend durch das Verfahren aus Kapitel 6 um Managementfunktionalitäten erweitert werden. Kapitel 7 beschreibt die Besonderheiten und nötigen Anpassungen, um die im Verlauf der Arbeit präsentierten Konzepte für bereits laufende Anwendungen ausführen zu können. Schließlich zeigt Kapitel 8, wie die Forschungsbeiträge im OpenTOSCA Ökosystem und dem EDMM-Transformation Framework umgesetzt wurden, während Kapitel 9 die Arbeit mit einer Zusammenfassung und einem Ausblick abschließt.

Kapitel 2

Grundlagen und verwandte Arbeiten

In diesem Kapitel werden die grundlegenden Konzepte im Bereich der Bereitstellung und des Managements von Anwendungen vorgestellt sowie der aktuelle Stand der Forschung diskutiert. Darauf aufbauend werden für jeden Bereich die Herausforderungen herausgearbeitet, die in der vorliegenden Arbeit gelöst werden. Abschnitt 2.1 beschäftigt sich mit dem Ansatz der modellgetriebenen Entwicklung und beschreibt dessen grundlegende Konzepte. Abschnitt 2.2 führt Methoden zur automatisierten Bereitstellung von Anwendungen sowie aktuelle Managementansätze ein. In Abschnitt 2.3 werden Muster und Mustersprachen eingeführt. Des Weiteren werden unterschiedliche Ansätze gezeigt, wie die für Menschen definierten Muster auch automatisiert verarbeitet werden können. Die automatisierte Erstellung von Anwendungs- bzw. Instanzmodellen von bereits laufenden Anwendungen wird schließlich in Abschnitt 2.4 präsentiert, während Abschnitt 2.5 die offenen Herausforderungen der einzelnen Forschungsbereiche zusammenfasst.

2.1 Modellgetriebene Entwicklung und Architektur

In der Softwareentwicklung werden Modelle verwendet, um das Verhalten und den Aufbau von Anwendungen zu beschreiben. Durch maschinenlesbare Dokumente können Modelle nicht nur zur Kommunikation verwendet, sondern auch automatisiert verarbeitet werden. Daraus entstand das Konzept der modellgetriebenen Entwicklung (engl. „*Model-Driven Development*“, MDD) [Sel03]. Im Gegensatz zur traditionellen Softwareentwicklung steht bei MDD nicht die Entwicklung von Code-Artefakten im Vordergrund, sondern die Erstellung von Modellen und deren automatisierte Transformation in andere Modelle oder sogar ausführbare (Code-) Artefakte [Béz05; Sch06].

Die Object Management Group (OMG) definierte mit der modellgetriebenen Architektur (engl. „*Model-Driven Architecture*“, MDA) [OMG10; OMG14; St00] eine allgemeine Methode, wie MDD bei der Entwicklung von komplexen Anwendungen umgesetzt werden kann. Dabei werden Modelle verwendet, um Anwendungen zu beschreiben, während *Metamodelle* die Modellierungssprache der jeweiligen Modelle definieren. Diese Modelle werden bei MDA in drei Abstraktionsebenen eingeteilt [OMG10; OMG14]: Auf der obersten, d. h. abstraktesten, Ebene werden sogenannte Domänenmodelle, oder auch „*Computation Independent Models*“ (CIMs), der Anwendung definiert. Ein CIM beschreibt abstrakt das zu erwartende Verhalten der Anwendung, jedoch ohne technische Details zu nennen. Eine Detailstufe tiefer definieren die „*Platform Independent Models*“ (PIMs) den Aufbau und das Verhalten einer Anwendung unabhängig von konkreten Technologien oder Anbietern. Alle technisch notwendigen Details werden in MDA schließlich in den sogenannten „*Platform Specific Models*“ (PSMs) definiert, wobei die Elemente eines PIMs durch ein „*Platform Model*“ vorgegeben sind [MSUW02].

Die in dieser Arbeit präsentierte Methode führt sowohl eine Sprache zur Modellierung abstrakter Deployment-Modellen ein, als auch die automatisierte Transformation von abstrakten in ausführbare Deployment-Modelle. Damit gehört die vorgestellte Methode in den Bereich von MDD bzw. MDA und nutzt plattformunabhängige Beschreibungen von Anwendungen (PIMs) sowie die Transformation dieser in existierende, ausführbare und plattformspezifische

Modelle (PSMs). In Abschnitt 2.1.1 wird daher zunächst auf Grundlagen und verwandte Arbeiten im Bereich der Deployment-Modellierung eingegangen, bevor Abschnitt 2.1.2 Ansätze zur Transformation von Modellen vorgestellt und die aktuellen Defizite in Abschnitt 2.1.3 zusammengefasst werden.

2.1.1 Modellierung von Deployment-Modellen

Die Architektur einer Anwendung beschreibt, neben allgemeinen Gestaltungsgrundsätzen, typischerweise die grundlegenden Konzepte, Prinzipien und Eigenschaften eines Systems anhand seiner Komponenten und deren Beziehungen untereinander [ISO11]. Daher können Softwarearchitekturen im Allgemeinen als Graphen beschrieben werden, wobei die Komponenten einer Anwendung die Knoten und die Kanten deren Beziehungen untereinander beschreiben [Le 98]. Eine Methodik, wie graph-basierte Anwendungen in der Cloud und deren Bereitstellung modelliert werden können, präsentieren Inzinger et al. [INS+14]. Dabei wird in einem manuellen iterativen Prozess zunächst ein rudimentäres Architekturmodell durch mehrere manuelle Verfeinerungsschritte und dem Umsetzen von Geschäftsanforderungen zu einem Modell der Anwendung weiterentwickelt, das konkrete Komponenten und Technologien für das Deployment der Anwendung beschreibt. Zur Auswahl der konkreten Komponenten werden dabei Entscheidungsbäume eingesetzt. Somit entsteht durch iteratives Verfeinern ein graphbasiertes Deployment-Modell der Anwendung, wobei die Komponenten der Anwendung die Konten und deren Abhängigkeiten und Relation untereinander die Kanten des Graphen darstellen [INS+14]. Graph-basierte Deployment-Modelle werden auch als *(Anwendungs-) Topologien* bezeichnet [BBF+18].

Generell können zwei Arten von Deployment-Modellen unterschieden werden: *imperative* Modelle und *deklarative* Modelle [EBF+17]. Während deklarative Deployment-Modelle definieren, *was* bereitgestellt werden soll, spezifizieren imperative Deployment-Modelle, *wie* eine Anwendung instanziiert wird. Das deklarative Modell auf der linken Seite in Abbildung 2.1 beschreibt beispielhaft die Komponenten einer Webanwendung sowie deren Beziehungen. In diesem Beispiel soll eine Webapp, die bestimmte Daten

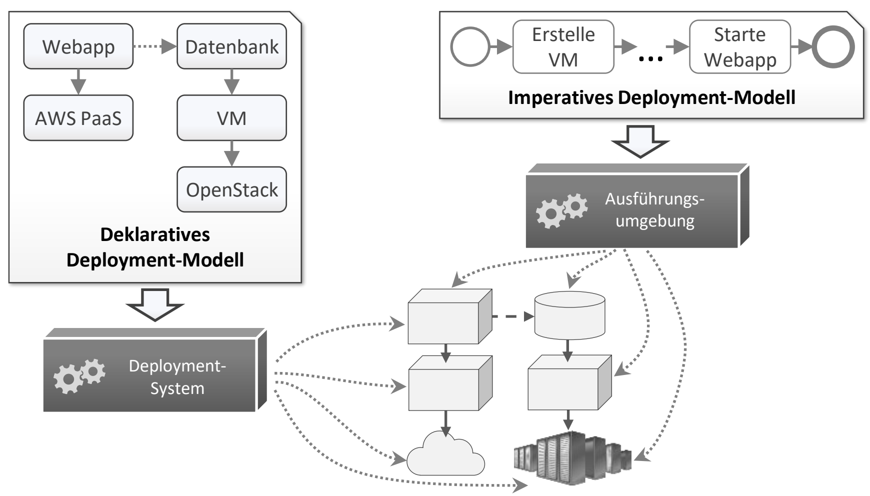


Abbildung 2.1: Deklarative und imperative Deployment-Modelle

aus einer Datenbank abruft, auf einem „*Platform as a Service (PaaS)*"-Dienst des Cloud-Anbieters *Amazon Web Services (AWS)* bereitgestellt werden. Die Datenbank soll dabei auf einer *virtuellen Maschine (VM)* in einer *Open-Stack*-Umgebung bereitgestellt werden. Ein deklaratives Deployment-Modell beschreibt somit den gewünschten Zielzustand einer Anwendung. Um die automatisierte Bereitstellung der Anwendung zu ermöglichen, muss ein entsprechendes Deployment-System das deklarative Deployment-Modell interpretieren und alle Deployment-Schritte daraus ableiten und ausführen.

Im Gegensatz dazu definiert ein imperatives Deployment-Modell alle notwendigen Schritte und deren Reihenfolge für die automatisierte Bereitstellung einer Anwendung. Um zum Beispiel die Webapp mit ihrer Datenbank bereitzustellen, muss zunächst eine VM erstellt werden, auf der dann die Datenbank zuerst installiert und anschließend konfiguriert werden kann. Schließlich kann die Webapp mit der Datenbank verbunden und gestartet werden. Daher sind imperative Deployment-Modelle meist als *Prozesse*, beispielsweise in Form von BPMN- [OMG11] oder BPEL- [OAS07] Workflows

oder als Skripte realisiert. Eine passende Ausführungsumgebung, z. B. eine BPMN-Engine wie Camunda [Cam23], kann die imperativen Deployment-Modelle somit direkt ausführen anstatt zuerst die notwendigen Schritte und deren Reihenfolge aus dem Modell ableiten zu müssen, wie es bei deklarativen Modellen der Fall ist. Somit können imperative Deployment-Modelle an individuelle Anforderungen angepasst werden und können sogar manuelle Schritte enthalten, während deklarative Deployment-Modelle auf bekannte und häufig verwendete Komponenten limitiert sind [BKLW17]. Die Erstellung und Wartung von deklarativen Modelle ist daher mit deutlich weniger Aufwand verbunden als die Erstellung von imperativen Modellen [BKLW17].

In der Praxis hat sich die deklarative Deployment-Modellierung von Anwendungen gegenüber der imperativen durchgesetzt [WBF+19]. In einer Studie, in der die 13 meistgenutzten Deployment-Technologien untersucht wurden, stellte sich heraus, dass alle 13 Technologien deklarative Modelle verwenden [WBF+19]. Dabei können die unterschiedlichen Modellierungssprachen aller 13 Technologien auf ein gemeinsames Metamodell abgebildet werden: das „*Essential Deployment Metamodel*“ (EDMM) [WBF+19]. Auf Basis von EDMM kann daher eine Anwendung technologieunabhängig modelliert und anschließend in technologiespezifische und ausführbare Deployment-Modelle der jeweils gewünschten Deployment-Technologie transformiert werden. So kann eine Anwendung beispielsweise in EDMM modelliert und schließlich von Deployment-Technologien wie Ansible [Red23], Chef [Pro23] oder Kubernetes [The23] automatisiert bereitgestellt werden. Dazu interpretieren diese Technologien jeweils ein deklaratives Deployment-Modell einer Anwendung und leiten die notwendigen Schritte daraus ab. Im Allgemein können deklarative Deployment-Modelle automatisiert in imperative transformiert und beide Varianten miteinander kombiniert werden [BBK+14].

Die Kombination von deklarativen und imperativen Deployment-Modellen wird zum Beispiel von der „*Topology and Orchestration Specification for Cloud Applications*“ (TOSCA) [OAS13b; OAS20], einer standardisierten Modellierungssprache für Cloud-Anwendungen, explizit beschrieben und genutzt. In TOSCA werden Anwendungen in sogenannten „*Service Templates*“ beschrieben. Diese enthalten (i) ein „*Topology Template*“, welches die Kompo-

nenten und Beziehungen der Anwendungen beschreibt, sowie (ii) mehrere *Pläne*. Pläne definieren bestimmte Managementaspekte, vor allem die Erstellung und Terminierung einer Anwendungsinstanz in Form von BPMN- oder BPEL-Workflows. Dabei ist ein Topology Template ein deklaratives Deployment-Modell der modellierten Anwendung, während ein Plan für die Erstellung einer Anwendungsinstanz, allgemein auch als „*Build-Plan*" bezeichnet [OAS13b], die imperative Entsprechung darstellt und die Reihenfolge der Aktivitäten zum Installieren, Konfigurieren und Starten der im Topology Template definierten Komponenten und Relationen enthält.

Generell wurde in Wissenschaft und Praxis eine Vielfalt an Deployment-Systemen und -Technologien entwickelt, welche meistens ihre eigene *domänenspezifische Sprache* (engl. „*Domain-specific Language*", DSL) definieren [BBF+18; WBF+19]. Zum Beispiel nutzt jede der bereits genannten Deployment-Technologien wie Ansible, Chef und Kubernetes, ihre eigene Sprache, um Anwendungen zu beschreiben. Auch in der Wissenschaft wurden zahlreiche Modellierungssprachen eingeführt [BBF+18] wie beispielsweise die „*Move to Clouds for Composite Applications*"-Methode (MOCCA) [LFM+11]. MOCCA führt ein Metamodell ein, das als Vorstufe des TOSCA-Standards gilt. Es beschreibt ein Metamodell, mit welchem die Artefakte und Beziehungen einer Anwendung beschrieben werden können sowie einen Formalismus, um die optimale Verteilung der einzelnen Komponenten auf unterschiedliche Cloud-Ressourcen zu berechnen [LFM+11]. Ähnlich dazu ermöglicht es beispielsweise das Komponentenmodell *Aeolus* [DMZZ14; DZZ12] Komponenten einer Anwendung und deren Beziehungen zu definieren. Für die Bereitstellung eines Aeolus-Modells können ebenfalls Deployment-Pläne berechnet und ausgeführt verwendet [DMZZ14].

Mit dem *SODALITE*-Framework [BDV22; DV22] können komplexe Cloud-Anwendungen in einer eigenen DSL definiert werden. Dabei können Aspekte wie verfügbare Ressourcen, Monitoring oder Optimierungen in separaten Modellen spezifiziert werden [GRV+22]. Um eine Anwendung bereitzustellen, generiert SODALITE ein ausführbares Service Template.

Die „*Cloud Modelling Language*" (CloudML) [FCS+18; FRC+13; FSR+14] beschreibt Anwendungen ebenfalls auf Basis ihrer Komponenten. Dabei

können Komponenten zwischen „*Infrastructure as a Service (IaaS)*“ und PaaS-Angeboten migriert werden. Darüber hinaus nutzt CloudML einen „*models@run.time*“ [BBF09] Ansatz, um Anwendungen dynamisch zur Laufzeit über automatisiert generierte Pläne anpassen zu können. Des Weiteren wurde CloudML von der „*Cloud Application Modelling and Execution Language*“ (CAMEL) [AKR+19] erweitert: Neben allen Möglichkeiten der Ausgangssprache, d. h., automatisierte Bereitstellung und Management von Cloud-Anwendungen, kombiniert CAMEL mehrere unterschiedliche Ansätze in einer „*megaDSL*“ [AKR+19], um beispielsweise zusätzlich Regeln für die automatische Skalierung von Komponenten definieren zu können. Auch die *ARGON* DSL [SIA17] unterstützt die Definition solcher Regeln in der Form von „*Policies*“. ARGON ist eine Modellierungssprache, um die Infrastruktur einer Anwendung deployment-technologieunabhängig zu beschreiben.

Der Konfigurationsdesigner „*Weaver*“ [KKM+19; OBT+22] ermöglicht die graphbasierte Modellierung von Anforderungen verschiedener Services, die anschließend in TOSCA Service Templates transformiert werden. Um die Anforderungen zu erfüllen, werden von Weaver „*Refinement Rules*“ eingesetzt, die die Anforderungen durch konkrete Komponenten umsetzen. Ähnliche Ansätze präsentieren Elhabbash et al. [EJBE19] mit der *SLO-ML*, wobei Anforderungen als „*Service Level Objectives*“ (SLOs) je Komponente definiert werden können, während Stötzner et al. [SBB+22] verschiedene Anforderungen als Varianten direkt in einem Service Template beschreiben.

Binz [Bin15] definiert mit dem „*Enterprise Topology Graph*“ eine formale Beschreibungssprache für Anwendungen, wobei Komponenten und deren Beziehungen beschrieben werden können. Ähnlich dazu führt Breitenbücher [Bre16] eine formale Sprache zur Modellierung von Anwendungsbereitstellungen ein. Darauf aufbauend präsentiert Wild [Wil22] eine Formalisierung für EDMM. Alle drei nutzen dabei eine Teilmenge des TOSCA Topology Templates, wobei Breitenbücher und Wild zusätzliche Elemente definieren.

Im Gegensatz zu den Sprachen, die jeweils eigene Metamodelle definieren, nutzen einige andere Ansätze die „*Universal Modeling Language*“ (UML) [OMG07] und erweitern diese. Zum Beispiel nutzt die „*Cloud Application Modeling Language*“ (CAML) [BTN+14] UML, um Cloud-spezifische

Eigenschaften einer Anwendung zu beschreiben. Dabei definiert CAML auch einen Diagrammtyp, der die Bereitstellung der modellierten Anwendung definiert. Weitere UML-Profile wie „*DICER*“ [ABD+18], „*URANO*“ [RVT+18] oder „*MULTICLAPP*“ [GMMC13a; GMMC13b] bieten ebenfalls die Möglichkeit, die Bereitstellung von Anwendungen auf Basis von UML zu beschreiben.

Bei allen vorgestellten Sprachen und Technologien müssen die Komponenten einer Anwendung einzeln definiert und konfiguriert werden. Daher müssen Modellierende ein tiefes technisches Verständnis und Detailwissen über jede Komponente einer Anwendung besitzen, um diese korrekt zu beschreiben und den Anforderungen entsprechend zu konfigurieren. Besonders bei komplexen Anwendungen mit zahlreichen Komponenten unterschiedlichen Typs, z. B. Logikkomponenten, Messaging-Komponenten, Infrastrukturkomponenten und Datenbanken, stellt die manuelle Modellierung eine besondere Herausforderung dar. Das in dieser Arbeit präsentierte Modellierungskonzept ermöglicht es, Anwendungen auf Basis von Muster zu beschreiben und automatisiert zu konkreten Komponenten und entsprechenden Konfigurationen zu verfeinern. Damit wird die Modellierung von Anwendungen erleichtert, da lediglich die abstrakten Konzepte (Muster, s. Abschnitt 2.3) verstanden werden müssen, anstatt deren konkrete Umsetzungen mit diversen Komponenten, Technologien und Service-Angeboten.

Einen ähnlichen Ansatz zur musterbasierten Modellierung von Anwendungen nutzt die „*Cloud Application Modeling Solution*“ (Clams) [BDR21a]. Allerdings liegt der Fokus von Clams auf der Beschreibung von Szenarien, die in einer Anwendung ablaufen. Somit kann Clams dazu verwendet werden, Sequenzdiagramme und die Kommunikation einer Anwendung auf Basis von Mustern anstelle von konkreten Komponenten darzustellen.

Auch Fehling et al. [FLR+11] beschreiben ein Framework für die musterbasierte Modellierung von Anwendungen. Für eine automatisierte Bereitstellung einer Anwendung können dabei anbieterspezifischen Informationen sowie ausführbare „*Variability Models*“ [MUL09] an die Muster annotiert werden. Jedoch benutzen Fehling et al. weder Muster als separate Modellelemente in Deployment-Modellen, noch präsentieren sie einen Prototyp.

2.1.2 Transformation von Modellen

Bei der modellgetriebenen Entwicklung liegt das Hauptaugenmerk, neben dem Modell selbst, auf der Transformation der Modelle. Im Allgemeinen können dabei drei Transformationsarten unterschieden werden: (i) Modell-zu-Modell, (ii) Modell-zu-Artefakt sowie (iii) Artefakt-zu-Modell [OMG10]. Unabhängig von der Transformationsart werden Transformationen, ganz dem „*Everything is a model*"-Prinzip [Béz05] folgend, ebenfalls als Modelle beschrieben. Diese in MDA „*Transformation Specifications*" [OMG10] genannten Modelle beschreiben, wie die Elemente eines Ausgangsmodells auf ein Zielmodell abgebildet werden kann. Da Modelle in der Softwareentwicklung meist als Graphen dargestellt werden können [Le 98], kann für die Beschreibung der Transformationen im Allgemeinen die von Schürr [Sch95] definierte „*Triple Graph Grammar*" (TGG) verwendet werden.

Eine TGG beschreibt, wie ein Ausgangsgraph in einen Zielgraph transformiert werden kann und besteht aus drei Teilgraphen wie in Abbildung 2.2 dargestellt: Der Ausgangsgraph befindet sich in der Regel auf der linken Seite und wird als *linker Graph* bezeichnet. Dementsprechend stellt der *rechte Graph* den Zielgraphen dar. Zwischen den beiden Graphen beschreibt der *Entsprechungsgraph*, wie die Elemente des Ausgangsgraphen auf die Elemente des Zielgraphen abgebildet werden können. Übertragen auf MDA heißt das, dass der Entsprechungsgraph die Abbildung der Modellelemente des Ausgangsmodells auf die Elemente des Zielmodells beschreibt.

Die in Abbildung 2.2 dargestellte TGG zeigt, dass das Modellelement „A" aus dem linken Graphen auf die beiden Elemente „1" und „2" des rechten Graphen abgebildet werden. Ebenso werden die Elemente „C" und „D" gemeinsam auf „4" abgebildet. Es ist somit möglich, mehrere Modellelemente des Ausgangsmodells auf ein Modellelement des Zielmodells zusammenzuführen sowie ein Ausgangselement auf mehrere Zielelemente aufzuteilen. Dies bietet große Flexibilität in der Beschreibung von Transformationsmodellen und ermöglicht die Verwendung von TGGs in beide Richtungen, d. h. zur Transformation des linken in den rechten Graphen sowie zur Abbildung des rechten auf den linken Graphen [Sch95].

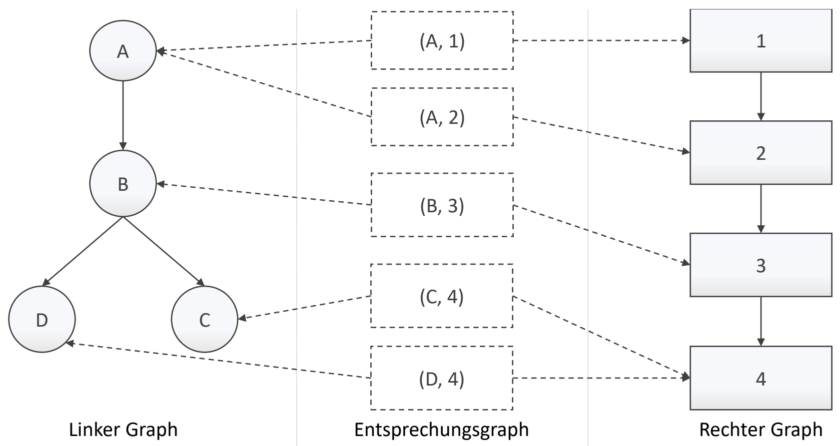


Abbildung 2.2: Illustration einer Triple Graph Grammar

In MDA können Transformationen zwischen zwei Modellen auf Basis ihrer Metamodelle definiert werden. Mellor et al. [MSUW02] definieren dafür einen MDA-Softwareentwicklungsprozess, wobei zwischen (i) der Abbildung der Metamodelle untereinander sowie (ii) deren Ausführung unterschieden wird. Eine Abbildung definiert dabei konkrete Regeln, die für eine Transformation angewendet werden müssen, die sogenannten „*Mapping Techniques*“ [MSUW02]. Eine Ausführungsumgebung, die diese Regeln verarbeiten kann, bildet somit das eigentliche „*Mapping*“ [MSUW02] und ist in der Lage, ein oder mehrere gegebene Modelle in das gewünschte Zielmodell zu transformieren. Ein Beispiel ist das *CAML2TOSCA* Framework [BBK+16], welches die Transformation eines abstrakten, UML-basierten Architekturmodells einer Anwendung in ein TOSCA Service Template definiert. Zur Automatisierung der Transformation wird ein bestehendes Transformationsframework genutzt. Ähnliche Funktionalitäten bieten das Modellierungswerkzeug der ARGON DSL [SIA19] sowie die „*DevOps-Base*“ [BMG+19]. Beide nutzten bestehende Transformationsframeworks für die automatisierte Umsetzung einer in ARGON [SIA19] bzw. in TOSCA [BMG+19] modellierten Anwendung, um *Infrastructure as Code (IaC)* Artefakte zu generieren.

Im Gegensatz dazu kombinieren andere Arbeiten die Abbildung und Ausführung. Anstelle einer generischen Ausführungsumgebung, die Abbildungsregeln zur Transformation eines Ausgangsmodells in ein Zielmodell verwenden kann, definieren kombinierte Methoden die Abbildungsregeln eines Ausgangsmodells auf ein Zielmodell im Code der Ausführungsumgebung. Somit wurden diese Werkzeuge speziell für die Transformation einer Modellart in eine andere entwickelt und können nicht für andere Transformationen genutzt werden. Zum Beispiel definieren Wurster et al. [WBB+19; WBB+21; WBH+20c] und Soldani et al. [SBB+23] Methoden, um Anwendungen in TOSCA zu beschreiben und transformieren Service Templates in Modelle von Deployment-Technologien wie Docker Compose, Kubernetes oder Terraform. Durch eine plug-in-basierte Architektur der Ausführungsumgebung kann das Framework jedoch um zusätzliche Modelle erweitert werden [WBH+20c].

Zalila et al. [ZKE+22] führen mit dem *MoDMaCAO*-Framework eine Erweiterung für das „*Open Cloud Computing Interface*" (OCCI) [NEP+16] ein. Während OCCI eine standardisierte Schnittstelle für das Management von verschiedenen Cloud-Ressourcen bietet, ermöglicht MoDMaCAO die grafische Modellierung einer Anwendung und präsentiert einen modellgetriebenen Ansatz, um eine Anwendung automatisiert bereitzustellen. Dafür wird ein OCCI-Modell in Ansible-Artefakte transformiert, welche anschließend ausgeführt werden können. Ein ähnliches Verfahren verwenden Weerasiri et al. [WBBC16]. Dabei wird kein generisches Metamodell wie TOSCA oder OCCI verwendet, um eine Anwendung abstrakt zu beschreiben, sondern werkzeugspezifische Modelle, die anschließend mithilfe von „Konnektoren" in Modelle unterschiedlicher Deployment-Technologien übersetzt werden.

Die in dieser Arbeit vorgestellte Methode zur Verfeinerung von musterbasierten Deployment-Modellen verwendet Konzepte aus beiden beschriebenen Ansätzen. Dabei werden Muster in einem Deployment-Modell zu konkreten Technologien und Anbietern verfeinert. Während *Verfeinerungsmodelle* Abbildungsregeln definieren, um eine bestimmte Konstellation aus Mustern und Komponenten zu verfeinern, ermöglicht es die Ausführungsumgebung, diese Verfeinerungsmodelle für eine Transformation bzw. *Verfeinerung* zu verwenden. Verfeinerungsmodelle definieren allerdings keine Abbildungen

zwischen zwei Metamodellen, sondern geben an, wie Fragmente eines Modells zu anderen Fragmenten im selben Modell transformiert werden können. Somit können zwar beliebige Modellfragmente eines Modells transformiert werden, jedoch nur innerhalb eines Metamodells.

2.1.3 Zusammenfassung

Aktuell existieren zahlreiche Sprachen zur Beschreibung von Anwendungsbereitstellungen. Allerdings wird für die Erstellung solcher Deployment-Modelle viel Detailwissen und technische Expertise benötigt, um das Zusammenspiel und die Konfiguration der einzelnen Komponenten einer Anwendung entsprechend der Anforderungen umzusetzen. Um dem entgegenzuwirken, wird in dieser Arbeit eine Sprache vorgestellt, die auf Basis von EDMM, Muster (s. Abschnitt 2.3) als eigne Modellelemente einführt (Forschungsbeitrag 2). Dadurch können Anwendungen abstrakt mit Muster modelliert werden, wobei Detailwissen über konkrete Technologien und deren Konfigurationen nicht benötigt wird. Dabei wird EDMM verwendet, da es (i) eine echte Teilmenge der vom TOSCA-Standard definierten Modellelemente verwendet [WBH+20c] sowie (ii) automatisiert auf alle gängigen Deployment-Technologien abgebildet werden kann [WBB+20b; WBH+20b].

Da die eingeführten musterbasierten Deployment-Modelle die Komponenten einer Anwendung abstrakt beschreiben, müssen diese zu konkreten Technologien und Angeboten verfeinert werden. Daher wird eine modellgetriebene Methode (Forschungsbeitrag 1) eingeführt, welche musterbasierte Deployment-Modelle durch eine automatisierte Transformation in ausführbare Deployment-Modelle transformieren kann (Forschungsbeitrag 3). Die Regeln der Transformation werden dabei in wiederverwendbaren Modellen definiert, die der Struktur der Triple Graph Grammars [Sch95] folgen.

2.2 Bereitstellung und Management von Anwendungen

Der Lebenszyklus einer Anwendung besteht aus (i) ihrem Entwicklungszyklus (engl. „*development*“), ihrer (ii) Verwaltung (engl. „*governance*“) sowie

(iii) den Betrieb der Anwendung (engl. „*operations*") [Cha08]. Dabei lässt sich der Betrieb in (a) die Bereitstellung der Anwendung, (b) ihr Management zur Laufzeit und (c) ihre Außerbetriebnahme gliedern [Bre16; Cha08]. Im Folgenden wird zunächst auf die automatisierte Bereitstellung und Außerbetriebnahme von Anwendungen eingegangen, gefolgt von Ansätzen für das Anwendungsmanagement sowie der Paketierung von Anwendungen.

2.2.1 Anwendungsbereitstellung und Außerbetriebnahme

Um eine Anwendung automatisiert bereitzustellen, wird üblicherweise ein Deployment-Modell der Anwendung von einer Laufzeitumgebung, dem *Deployment-System*, verarbeitet und ausgeführt [BBF+18]. Dabei werden die *Lebenszyklusoperationen* (engl. „*lifecycle operations*") ausgeführt, die eine Komponente (i) *installieren*, (ii) *konfigurieren* und (iii) *starten* können. Zu den Lebenszyklusoperationen gehören darüber hinaus auch Operationen für die Außerbetriebnahme einer Komponente, d. h. Operationen, die eine Komponente (iv) *stoppen* und (v) *deinstallieren* [OAS13a; OAS20]. Für jede Anwendungskomponente müssen daher diese Operationen zur Verfügung stehen, damit eine Instanz der Komponente automatisiert verwaltet werden kann. Die Implementierungen der Operationen können dabei entweder Teil des Deployment-Modells sein oder das Deployment-System kennt die nötigen Schritte, einen bestimmten Zustand einer Komponente herzustellen.

Die Reihenfolge der Lebenszyklusoperationen und welche Operationen überhaupt bei der Bereitstellung ausgeführt werden müssen, hängt dabei vom Zustand der Komponente sowie ihren Abhängigkeiten von anderen Komponenten ab [BBK+14; DMZZ14]. Zum Beispiel können Infrastrukturkomponenten wie virtuelle Maschinen bereits gestartet sein und müssen nicht mehr installiert, konfiguriert und gestartet werden. Des Weiteren muss beispielsweise eine Infrastrukturkomponente gestartet sein, bevor weitere Komponenten, wie Webserver oder Laufzeitumgebungen, darauf installiert und gestartet werden können. Somit kann die Reihenfolge der Lebenszyklusoperationen anhand der Anwendungstopologie automatisiert abgeleitet und ausgeführt werden [BBK+14; BKLW17]. Der dabei generierte Build-Plan

stellt ein imperatives Deployment-Modell dar und kann direkt ausgeführt oder abgelegt werden, wobei er nach Bedarf manuell angepasst und erweitert werden kann [BBK+14]. Dementsprechend kann eine Anwendung auch automatisiert außer Betrieb genommen werden, indem ihre Komponenten zuerst gestoppt und anschließend deinstalliert werden. Ein solcher Plan, der die Operationen zur Außerbetriebnahme einer Anwendung und deren Reihenfolge definiert, wird als „*Termination-Plan*" bezeichnet [OAS13b].

Einige Arbeiten behandeln bereits die Generierung von Build-Plänen. Zum Beispiel leiten Breitenbücher et al. [BBK+14] und Mietzner, Unger und Leymann [MUL09] zunächst aus einer Anwendungstopologie einen sogenannten „*Provisioning Order Graph (POG)*" ab, welcher die Reihenfolge definiert, in der die einzelnen Komponenten einer Anwendung bereitgestellt werden müssen. Im Anschluss wird daraus zuerst ein abstrakter, dann ein ausführbarer Provisionierungsplan generiert, welcher die jeweiligen Lebenszyklusoperationen nacheinander ausführt. Ein ähnliches Konzept präsentieren Lu et al. [LSS+13], welche jedoch keine Pläne erstellen, sondern die Bereitstellungsaktivitäten direkt ausführen, während Wild et al. [WBK+20] Choreographien für Bereitstellungen verteilter Anwendungen ableiten. Darüber hinaus gibt es Konzepte, die „*Planning*"*-Methoden* [EME+06; KHW+04; LR09] oder an Komponenten annotierte *Bedingungen* [SSMJ04] für die Bereitstellung und Konfiguration von Anwendungen verwenden.

All diese Konzepte können Anwendungen automatisiert bereitstellen, konfigurieren und außer Betrieb nehmen, jedoch berücksichtigt keines die Konservierung und Wiederherstellung des aktuellen Zustands der Anwendung. Daher gehen alle anwendungsspezifischen Daten verloren, wenn eine Anwendung heruntergefahren wird. Um dies zu verhindern, wurde ein Konzept entwickelt, mit dem es möglich ist, Pläne zu generieren, die den Zustand einer Anwendung persistent speichern und wiederherstellen können. Dafür wird das Deployment-Modell einer Anwendung automatisiert erweitert und zusätzliche Managementpläne werden generiert, die (i) die Außerbetriebnahme inklusive einer Persisitierung des Anwendungsszustands sowie (ii) die Bereitstellung inklusive einer Wiederherstellung der Daten automatisieren.

2.2.2 Anwendungsmanagement und Managementworkflows

Nachdem eine Anwendung erfolgreich bereitgestellt wurde, muss sie verwaltet und gepflegt werden. Es ist beispielsweise notwendig, Komponenten zu aktualisieren, automatisiert zu testen oder regelmäßig eine Kopie der Anwendungsdaten vorzunehmen. Diese Managementaufgaben können dabei in *zustandsändernde* und *zustandserhaltende* Operationen unterteilt werden [Bre16]. Während zustandsändernde Operationen die Struktur, die Konfiguration oder den Zustand einer Komponente verändern, interagieren zustandserhaltende Operationen lediglich mit der entsprechenden Komponente. Die Lebenszyklusoperationen, die Komponenten installieren und konfigurieren, sind damit zustandsändernde Operationen. Im Gegensatz dazu sind beispielsweise das Ausführen eines Backups oder Verfügbarkeitstests zustandserhaltende Operationen, da sie ausschließlich mit der Komponente interagieren, ihren Zustand oder ihre Konfiguration jedoch nicht verändern.

Um das Management von Anwendungen zu automatisieren, wurden bereits einige Ansätze präsentiert: Zum Beispiel nutzt Breitenbücher [Bre16] *Managementmuster*, um Managementaufgaben zu definieren. Diese Managementmuster spezifizieren dabei die notwendigen Schritte, z. B. zum Löschen oder Hinzufügen einer Komponente, welche anschließend als Managementannotationen an die betreffenden Komponenten geheftet werden. Um beispielsweise eine Datenbank von einem Cloud-Anbieter zu einem anderen zu migrieren, müssen die bestehenden Komponenten gestoppt und heruntergefahren werden, während die neuen Komponenten installiert, konfiguriert und gestartet werden müssen. Daher werden die entsprechenden Komponenten mit *Löschen*- und *Hinzufügen*-Annotationen im Deployment-Modell markiert. Anschließend können „*Planlets*“ auf Basis der Annotationen identifiziert werden, die bestimmte Teile der Managementaufgaben, z. B. das Herunterfahren der bestehenden Komponenten, bereits als Workflow-Fragmente umsetzen. Durch das Kombinieren verschiedener Planlets kann somit ein Workflow generiert werden, der die gesamte Managementaufgabe realisiert [Bre16]. Ähnlich zur Workflow-Generierung mit Planlets funktioniert die Workflow-Generierung mit „*Automation Signatures*“ [EEKS11].

Wagner et al. [WBK+17; WBL16] präsentieren eine Methode, um in TOSCA modellierte Anwendungen wiederzuverwenden. Dabei werden modellierte Anwendungen zu einer neuen Anwendung kombiniert, wobei auch die bereits modellierten Managementpläne zusammengeführt werden.

In den Bereich des Anwendungsmanagements fällt auch die Skalierung einzelner Komponenten, sodass beispielsweise eine höhere Auslastung der Anwendung abgefangen werden kann. Bei Deployment-Technologien wie Kubernetes oder Cloud-Anbietern wie AWS und Azure können dafür beispielsweise Metriken angegeben werden, um zu definieren, wann eine bestimmte Komponente skaliert werden muss. Darüber hinaus gibt es ähnliche Ansätze in der Wissenschaft [VRB11]. Zum Beispiel präsentieren Képes et al. [KBL17; KLZ20], wie „*Scaleout*"- und situationsabhängige „*Update*"-*Pläne* für einzelne Komponenten automatisiert generiert und ausgeführt werden können.

Calcaterra und Tomarchio [CT22] und Wurster et al. [WBKL18] präsentieren Möglichkeiten, um Anwendungen automatisiert testen zu können. Dabei können Tests an Komponenten annotiert werden, welche im Anschluss an eine Bereitstellung ausgeführt werden, um zum Beispiel die Erreichbarkeit und Konfiguration bestimmter Komponenten sicherzustellen. Des Weiteren definieren Calcaterra und Tomarchio ein Konzept, um Management-Workflows auf Basis von an Komponenten annotierten „*Policies*" abzuleiten, welche je eine Managementaufgabe einer Komponente realisieren.

Dabei ist es bisher nicht möglich existierende Deployment-Modelle automatisiert um zusätzliche Managementfunktionalitäten zu erweitern, sodass diese zur Laufzeit der Anwendung ausgeführt werden können. Deshalb wurde ein Konzept entwickelt, das es ermöglicht, für jede Bereitstellung unterschiedliche Managementfunktionalitäten für eine Anwendung auszuwählen und ganzheitlich für die gesamte Anwendung wiederholbar auszuführen (Forschungsbeitrag 4). Anwendungen, die über mehrere Cloud-Anbieter verteilt sind, werden dabei explizit unterstützt. Darüber hinaus kann das Konzept auf sich bereits im Betrieb befindliche Anwendungen übertragen werden, wodurch fehlende Managementfunktionalitäten nachträglich zu einer Anwendung automatisiert hinzugefügt und anschließend ausgeführt werden können (Forschungsbeitrag 5).

2.2.3 Speichern und Wiederherstellen von Anwendungszuständen

Während der Nutzung einer Anwendung entstehen im Allgemeinen geschäftsrelevante Daten, welche innerhalb der Anwendung gespeichert werden und somit Teil des Anwendungszustands sind. Dabei können zwei Arten von Komponenten unterschieden werden: die *zustandsbehafteten* (engl. „*stateful*“) sowie die *zustandslosen* (engl. „*stateless*“) Komponenten [FLR+14]. Zustandslose Komponenten speichern keine Daten zwischen Anfragen, sondern verarbeiten lediglich die Daten einer Anfrage. Daher können diese einfach migriert und skaliert werden, ohne dass Daten verloren gehen [FLR+14]. Im Gegensatz dazu speichern zustandsbehaftete Komponenten Daten. Ihre Skalierung erfordert somit die Synchronisation der Daten, um sicherzustellen, dass jede Instanz der Komponente auf dieselben Daten zugreift [FLR+14].

Die Sicherung und Übertragung der Anwendungszustände sind wichtige Bestandteile im Anwendungsmanagement, um im Falle eines Ausfalls den zuletzt gesicherten Zustand wiederherstellen zu können. Arbeiten in diesem Feld reichen dabei von der Migration einzelner Prozesse [HKF18] über die Erstellung von Kontrollpunkten (engl. „*checkpoints*“) [CL85; CSCM15; ELSC13; KR00] bis zum Transfer gesamter VMs [AGH+15; CFH+05; SPP+22].

Auch Technologien wie Docker [Tur14] und Kubernetes [The23] bieten die Möglichkeit, Zustände einer Komponente zu speichern und wiederherzustellen. Docker bietet dafür ein experimentelles „*Checkpoint*“-Feature an, welches den aktuellen Zustand eines Containers einfrieren und wiederherstellen kann [Doc23]. In Kubernetes können die sogenannten „*Stateful Sets*“ verwendet werden, um Zustände in einem Container mit einem anderen zu synchronisieren oder Zustände persistent zu speichern [The23].

Zwar existieren zahlreiche Methoden, um Zustände aus laufenden Anwendungen persistent zu speichern, jedoch gibt es aktuell kein ganzheitliches Konzept, das eine Anwendung zusammen mit ihren Daten terminieren und zu einem späteren Zeitpunkt wieder starten kann. Im Zuge des Forschungsbeitrag 4 wurde ein Konzept entwickelt, das es ermöglicht, beliebige Operationen, die die Zustände zustandsbehafteter Komponenten speichern, für die gesamte Anwendung in einem Workflow zu orchestrieren und auszuführen.

2.2.4 Paketierung von Deployment-Modellen

Damit eine Anwendung wiederholbar bereitgestellt und verwaltet werden kann, muss diese mit all ihren Abhängigkeiten paketiert werden. TOSCA definiert dafür das sogenannte „*Cloud Service Archive (CSAR)*“ [OAS13b; OAS20]. Ein CSAR enthält sowohl das Modell der Anwendung als auch die benötigten ausführbaren Artefakte. Dabei wird zwischen „*Implementierungsartefakten*“ (IAs) und „*Deployment-Artefakten*“ (DAs) unterschieden [OAS13b]: Ein DA ist ein Artefakt, welches die Geschäftslogik einer Komponente umsetzt, zum Beispiel ein Archiv mit allen PHP-Dateien einer PHP-Anwendung. Im Gegensatz dazu realisiert ein IA eine bestimmte Operation einer Komponente. Um zum Beispiel die bereits erwähnte PHP-Anwendung auf einem Webserver zu installieren, müssen die PHP-Dateien aus dem DA auf einem Webserver entpackt und in den von der Komponente definierten Kontextpfad verschoben werden. Dies kann durch ein IA realisiert werden, welches die *install* Operation der PHP-Anwendung implementiert.

Implementierungsartefakte können darüber hinaus auch Dateien aus dem Internet laden, um zum Beispiel einen Webserver zu installieren. Ein Nachteil dabei ist, dass der Webserver nicht installiert werden kann, wenn externe Ressourcen, zum Beispiel über das Internet, nicht (mehr) verfügbar sind. Um dem entgegenzuwirken, wurde in Zusammenarbeit mit Zimmermann et al. [ZBH+20] ein Konzept eingeführt, welches alle notwendigen Artefakte für die Bereitstellung und das Management der Anwendung in einem Deployment-Paket bündelt, dem sogenannten „*Self-contained Deployment Package*“. Damit können Anwendungen bereitgestellt werden, ohne dass externe Ressourcen während der Bereitstellung benötigt werden.

Zur automatisierten Erstellung dieser Self-contained Deployment Packages wurde das „*Self-Containment Packager Framework*“ [ZBH+20] eingeführt. Das Framework bekommt ein Deployment-Paket mit externen Abhängigkeiten übergeben und transformiert es in ein unabhängiges Deployment-Paket, indem es die aus dem Internet benötigten Dateien herunterlädt und dem Deployment-Paket als DAs hinzugefügt. Dabei werden auch die IAs entsprechend angepasst, sodass die heruntergeladenen Dateien anstelle der externen

Ressourcen bei der Ausführung verwendet werden. Durch die Zusammenführung aller Abhängigkeiten einer Anwendung in ihr Deployment-Paket wird sichergestellt, dass die Anwendung auch zu späteren Zeitpunkten bereitgestellt werden kann, ohne dass externe Abhängigkeiten fehlen.

Im Rahmen dieser Arbeit wird dieses Konzept wiederverwendet und erweitert, wobei der Fokus in dieser Arbeit auf den Geschäftsdaten der Anwendung liegt. Damit eine Anwendung in ihrem ursprünglichen Zustand wiederhergestellt werden kann, müssen auch die Daten Teil des Self-contained Deployment Packages sein. Daher werden die Daten, die in einer Anwendung gespeichert sind (s. Abschnitt 2.2.3), abgerufen und für die Wiederherstellung in einem neuen Deployment-Paket abgelegt (s. Abschnitt 6.4).

2.2.5 Zusammenfassung

Um Anwendungen automatisiert bereitzustellen und außer Betrieb zu nehmen, existieren diverse Ansätze, welche jedoch die Speicherung und Wiederherstellung des aktuellen Anwendungszustands nicht berücksichtigen. Zwar gibt es zahlreiche Methoden, um den Zustand einer Anwendung zu sichern und wiederherzustellen, allerdings werden diese nicht während der Bereitstellung oder Außerbetriebnahme einer Anwendung angewendet. Darüber hinaus müssen unterschiedliche Verfahren kombiniert werden, um die Zustände aller zustandsbehafteten Komponenten einer Anwendung automatisiert speichern und wiederherstellen zu können.

Außerdem ist es bisher nicht möglich, Managementfunktionalitäten zu einer Anwendung dynamisch hinzuzufügen. Bei jeder Bereitstellung einer Anwendung können sich die Anforderungen an benötigte Managementfunktionalitäten unterscheiden und sollten jeweils im Vorfeld ausgewählt werden. Darüber hinaus ist es aktuell nicht möglich, laufende Anwendungen nachträglich mit Managementfunktionalitäten zu erweitern und auszuführen.

In der vorliegenden Arbeit wird eine Möglichkeit vorgestellt, um automatisiert zusätzliche Managementfunktionalitäten zum Modell einer Anwendung hinzuzufügen: sowohl während der Modellierung (Forschungsbeitrag 4), als auch nachträglich zur Laufzeit (Forschungsbeitrag 5).

2.3 Muster, Mustersprachen und deren Verwendung

Muster (engl. „*Patterns*“) sind ein verbreitetes Mittel zur Dokumentation bewährter Lösungen für häufig wiederkehrende Probleme in einem bestimmten Bereich. Die Lösungsansätze der Muster werden dabei abstrakt definiert, sodass sie allgemeingültig und unabhängig von konkreten Umsetzungen gelten. Das Konzept der Muster wurde zuerst im Bereich der Architektur, d. h. im Hausbau und der Städteplanung, eingeführt [AIS77] und in den unterschiedlichsten Domänen übernommen: zum Beispiel in den Geisteswissenschaften, um Film- und Theaterkostüme zu beschreiben [Bar18], oder der Informatik [BMR+96; GHJV94; SB03]. Muster folgen dabei in der Regel einer klar definierten Struktur, die hauptsächlich aus (i) einem eindeutigen *Namen*, (ii) einer *Problembeschreibung*, (iii) der Beschreibung des *Kontextes*, in dem das Muster angewendet werden kann, (iv) einer Skizze einer bewährten *Lösung* sowie (v) einem *Icon* besteht [AIS77; Rei13].

Typischerweise sind Muster keine Einzellösungen, sondern sind mit anderen Mustern, die im selben Kontext relevant sein können, verknüpft. Durch diese *Relationen* kann ein Netzwerk an Mustern aufgespannt werden, eine sogenannte *Mustersprache* (engl. „*Pattern Language*“). Die Relationen zwischen Muster können dabei unterschiedliche Semantik haben [FL17]. Beispielsweise beschreiben Relationen mit *UND*-Semantik, dass zwei verwandte Muster häufig in Kombination verwendet werden, während *ODER*-Relationen alternative Möglichkeiten aufzeigen und *XOR*-Relationen exklusive Varianten eines Musters beschreiben [Rei13]. Daher können Relationen zwischen Muster Einblicke in äquivalente Probleme, Zusammenhänge oder andere Lösungen geben [FBB+15]. Relationen können auch dafür verwendet werden, um unterschiedliche Abstraktionsebenen der Probleme und Lösungen zu beschreiben, zum Beispiel hinsichtlich implementierungs- oder technologiespezifischer Details, und ermöglichen sogar die Navigation zwischen Mustern, die in unterschiedlichen Mustersprachen definiert sind [FBBL17; WBB+20a].

Das Prinzip der Muster und Mustersprachen hat sich allgemein in großen Teilen der Informatik etabliert. Im Bereich der Softwareentwicklung und -architektur gibt es mehrere verschiedene Mustersprachen. In der Cloud

Computing Domäne beschreiben zum Beispiel die *Cloud Computing Patterns (CCP)* [FLR+14] oder die *Internet of Things (IoT) Patterns* [RBF+16; RBF+17a; RBF+17b] bewährte Lösungen für wiederkehrende Probleme bei der Entwicklung von Software- und Hardwarekomponenten auf abstrakte und wiederverwendbare Art und Weise. Eine Abstraktionsebene tiefer definieren beispielsweise die proprietären Muster der Cloud-Anbieter AWS [Ama23] und Azure [Mic23], wie dieselben Probleme mit den jeweiligen Cloud-Angeboten von Amazon und Microsoft realisiert werden können. Während die Cloud Computing Patterns von Fehling et al. [FLR+14] technologie- und anbieterunabhängige Lösungen definieren, konkretisieren Amazon und Microsoft diese für ihre jeweiligen Angebote.

Neben unterschiedlichen Abstraktionsebenen können Muster auch unterschiedliche Aspekte einer Anwendung beschreiben. Dafür teilen Gamma et al. [GHJV94] ihre Muster zum Beispiel in drei Kategorien ein: (i) *Erstellungsmuster* (engl. „*creational patterns*“), (ii) *Strukturmuster* (engl. „*structural patterns*“) sowie (iii) *Verhaltensmuster* (engl. „*behavioral patterns*“). Diese Kategorien können dabei auch in der Mustersprache verwendet werden, um beispielsweise zu beschreiben, dass ein bestimmtes strukturelles Muster meist in Kombination mit einem Verhaltensmuster verwendet wird.

Im Allgemeinen sind Muster für Menschen geschrieben und sind meist in Büchern, Webseiten oder Repositorien wie dem *Pattern Atlas* [LB21] veröffentlicht. Durch ihre rein für menschliche Nutzer optimierte Form ist die automatisierte Verarbeitung von Mustern jedoch erschwert, da sie meist nicht maschinenlesbar aufbereitet sind. Es wurden bereits verschiedene Konzepte entwickelt, die die Anwendung von Mustern automatisieren. Im Folgenden werden diese vorgestellt, bevor verwandte Arbeiten im Bereich der automatisierten Musterverfeinerung und -erkennung diskutiert werden.

2.3.1 Musteranwendung

Generell dokumentieren Muster, wie wiederkehrende Probleme in einem speziellen Kontext gelöst werden können. Bevor ein Muster jedoch angewendet werden kann, muss es zunächst gemäß den Geschäftsanforderungen der

zu modellierenden Anwendung ausgewählt werden [Zdu07; ZZGL08]. Wird beispielsweise eine Komponente benötigt, welche hohe Auslastungen bewältigen kann, kann das Stateless Component-Muster [FLR+14] verwendet werden. Es definiert, wie eine Komponente aufgebaut sein muss, um sie ohne Zusatzaufwand skalieren zu können. Als Lösung gibt das Muster an, dass jeglicher Zustand der Komponente in einer externen Speicherkomponente abgelegt werden soll. Allerdings muss die konkrete Umsetzung von Entwickelnden zunächst aus der Lösungsbeschreibung abgeleitet und implementiert werden [FBB+14b]. Das heißt, dass Entwickelnde einer zustandslosen Komponente eine geeignete Speicherkomponente auswählen, anbinden und alle Datenzugriffe auf diese implementieren müssen. Um die Wahl einer konkreten Musterimplementierung zu unterstützen, führen Falkenthal et al. [FBB+14a; FBB+14b] sogenannte „*Solution Implementations*" ein. Diese verbinden Muster mit den Implementierungen und liefern somit Beispiele für die Umsetzung der Muster. Die Implementierungen können zusätzlich in *Lösungssprachen* (engl. „*Solution Languages*") [FL17] aggregiert werden, indem, ähnlich zu Mustersprachen, unterschiedliche Relationen zwischen den Lösungen definiert werden. Die Anwendbarkeit von Lösungssprachen wurde bereits für unterschiedlichen Domänen demonstriert [FBBL17].

Muster- und Lösungssprachen können, wie auch Deployment-Modelle, als Graphen dargestellt werden [FBL18]. Da Muster meist auch Relationen zu Mustern und Lösungen in anderen Muster- bzw. Lösungssprachen haben, können diese Relationen im Allgemeinen in speziellen *Sichten* (engl. „*Views*") dargestellt werden [AZ05; WBB+20a]. Somit können Problemstellungen, welche die Verwendung unterschiedlicher Muster aus mehreren Mustersprachen benötigen, gezielt zusammengefasst und angezeigt werden. Um beispielsweise eine elastische Cloud-Anwendung zu beschreiben, können Muster aus den Cloud Computing Patterns, den „*Enterprise Integration Patterns*" [HW04] sowie den „*Security Patterns*" [SFH+06] kombiniert werden. Eine Sicht, die das Zusammenspiel der dafür benötigten Muster beschreibt, kann dabei neue Relationen zwischen den Mustern definieren [WBB+20a].

Ähnlich zu den Solution Implementations zeigen Nowak et al. [NBF+12], wie Muster mit konkreten Implementierungen verknüpft werden können.

Dabei werden in TOSCA modellierte Anwendungsfragmente dazu verwendet, um die Cloud Computing Muster sowie die „*Green Business Process Muster*“ [NL13; NLS+11] mit konkreten Lösungen zu annotieren. Bei der Neumodellierung oder Adaption von Anwendungen können die an den Mustern annotierten Anwendungsfragmente somit als Bausteine in der neuen Anwendung wiederverwendet werden. Des Weiteren beschreiben Di Martino, Cretella und Esposito [DCE15; DCE17], wie die Cloud Computing Patterns verwendet werden können, um Modellierende im Entwurf und der Bereitstellung von Cloud-Anwendungen zu unterstützen. Sie führen ein semantisches Modell [DENM17] ein, um Prozesse als wiederverwendbare Muster zu beschreiben. Diese Prozessmuster können im Anschluss mit Cloud-Services, mit denen der Prozess in einer Cloud-Umgebung umgesetzt werden kann, verknüpft werden. Die Modelle können nicht für die Bereitstellung einer Anwendung verwendet werden, sondern dienen der Entscheidungsfindung, welche Cloud-Services genutzt werden können.

Um die Kommunikation zwischen verschiedenen Microservices abstrakt zu modellieren, führen Yussupov et al. [YBK+20] die „*Microservices Composition*“ (MICO) ein. Dabei werden die Enterprise Integration Patterns verwendet, um eine lose Kopplung von Microservices zu erreichen und die Kommunikation zwischen den einzelnen Microservices zu beschreiben. Während der Bereitstellung werden aus den verwendeten Muster konkrete Kommunikationskanäle und entsprechende Konfiguration generiert. Allerdings ist dieser Ansatz auf die Enterprise Integration Patterns beschränkt und jedes Muster muss im MICO-Framework implementiert werden.

Keiner der Ansätze verwendet Muster, um Deployment-Modelle anhand ihrer architektonischen, d. h. strukturellen und verhaltensbeschreibenden, Konzepte abstrakt darzustellen. Dies wird durch den Forschungsbeitrag 2 ermöglicht, indem Muster als „*first-class*“-Modellelemente eingeführt werden.

2.3.2 Musterverfeinerung

Bei der Verfeinerung von Mustern gibt es im Wesentlichen zwei Richtungen, die unterschieden werden können: Zum einen kann die Architektur

einer Anwendung *durch das Anwenden unterschiedlicher Muster* erweitert, bzw. konkretisiert, werden [GL18; GL19; Leh18; LHB18]. Zum anderen können *Muster selbst verfeinert* werden, beispielsweise durch (i) konkretere Muster, z. B. von technologieagnostisch zu technologiespezifisch oder durch (ii) konkrete Lösungen, z. B. ausführbare Implementierungen oder Code-Fragmente [EYG97; FBB+15; HS09; YBB+22].

Die bereits erwähnte Modellierung von Sequenzdiagrammen auf Basis von Mustern von Bibartiu, Dürr und Rothermel [BDR21a] kann auch automatisiert zu konkreten Komponenten und Technologien verfeinert werden [BDR21b]. Dabei werden Muster, abstrakte Komponenten sowie konkrete Komponenten in einem „*Refinement Tree*“ organisiert. An der Wurzel steht dabei eine abstrakte, allgemeingültige Lösung in der Form eines Musters. Je tiefer die inneren Knoten liegen, desto konkreter werden die Lösungen, bis hin zu konkreten Komponenten auf der Blattebene. Diese Refinement Trees werden dann in einer Kostenfunktion und Tiefensuche verwendet, um die Komponenten auszuwählen, die die geringsten Kosten aufweisen und allen Geschäftsanforderungen entsprechen [BDR21b]. Ähnlich dazu führen Hallstrom und Soundarajan [HS09] eine *Musterhierarchie* ein. Dabei beschreiben die sogenannten „*sub-patterns*“ unterschiedliche Varianten von abstrakteren Mustern und bilden somit eine Musterhierarchie.

Im Bereich der „*Service-oriented Architectures*“ (SOA) präsentieren Arnold et al. [AEK+07] ein Konzept, um verteilte Anwendungen abstrakt zu beschreiben. Dabei können die abstrakten Komponenten durch „*Patterns*“ mittels konkreten Lösungen *realisiert* werden [AEK+07]. Zum Beispiel kann eine abstrakte virtuelle Maschine durch ein konkretes Betriebssystem realisiert werden. Diese Realisierung von abstrakten Komponenten zu konkreten Lösungen kann automatisiert erfolgen [AEK+08]. Dabei müssen die abstrakten Komponenten jedoch genau einer konkreten Komponente entsprechen.

Zdun, Hentrich und Dustdar [ZHD07] präsentieren eine Methode, um SOAs mithilfe von Mustern und den sogenannten Muster-Primitiven (engl. „*Pattern Primitives*“) [ZA05] zu beschreiben. Dabei können Prozesse in einer SOA auf Basis von Muster als UML-Aktivitätsdiagramme abstrakt modelliert und mithilfe eines modellgetriebenen Ansatzes konkretisiert werden. Dabei

kann immer nur eine einzelne Aktivität in einem Aktivitätsdiagramm mit einem konkreteren bzw. detaillierten Prozess ausgetauscht werden.

Um die in dieser Arbeit vorgestellten musterbasierten Deployment-Modelle in ausführbare Deployment-Modelle transformieren zu können, wird eine Methode vorgestellt, die die Muster zu konkreten Komponenten verfeinert (Forschungsbeitrag 3). Dafür werden Verfeinerungsmodelle eingeführt, die eine Kombination aus mehreren Mustern zu konkreten Komponenten und deren Konfiguration verfeinern kann. Bei der Verfeinerung können sowohl Musterhierarchien berücksichtigt und abgebildet werden, als auch die Umsetzung eines Musters mit unterschiedlichen konkreten Lösungen.

2.3.3 Mustererkennung

Neben der Verwendung und Verfeinerung von Mustern zu konkreten Lösungen spielt auch die Mustererkennung eine wichtige Rolle in der Anwendungsmodellierung. So können beispielsweise weitere Lösungen für Muster gefunden, alternative Umsetzungen vorgeschlagen oder die Umsetzung bestimmter Muster verifiziert werden [BP02]. Des Weiteren erleichtert die Mustererkennung das Verständnis der in den Modellen verwendeten Konzepte. Anstatt das Verhalten und die Rollen der Komponenten und Relationen aus den technischen Details und Konfigurationen ableiten zu müssen, können diese Eigenschaften explizit durch die umgesetzten Muster beschrieben werden. In zahlreichen Arbeiten wurde die Erkennung von Mustern in unterschiedlichen Arten von Modellen bereits behandelt. Zum Beispiel von Arcelli Fontana und Zanoni [AZ11], Bergenti und Poggi [BP02], Haitzer und Zdun [HZ15], Kamal und Avgeriou [KA08] und Tsigkanos und Kehrer [TK16]. Zwar werden Muster in den meisten Arbeiten aus bekannten Strukturen, z. B. bestimmten Kombination aus Klassen oder Komponenten sowie deren Konfigurationen abgeleitet, allerdings werden diese nicht in einem Graphen dargestellt, welcher die abstrakte Architektur der Anwendung beschreibt.

Die Konzepte, die in der bereits erwähnten Methode zur Verfeinerung von musterbasierten Deployment-Modellen vorgestellt wird, können auch für die Erkennung eingesetzt werden. Somit lassen sich aus konkreten und ausführ-

baren Deployment-Modellen wieder musterbasierte Deployment-Modelle generieren, welche die realisierten Konzepte der Anwendung widerspiegeln.

In einigen Arbeiten im Bereich der Mustererkennung werden anstatt Muster sogenannte „*Antipatterns*", d. h. Muster, die regelmäßig gemachte Fehler beschreiben, sowie „*Code-Smells*", welche auf fehlerhaften Code hinweisen, identifiziert [AAG21]. Diese sollen Entwickelnden eine Hilfe bieten, um Implementierungsfehler zu vermeiden. Des Weiteren präsentieren Zimmermann et al. [ZBKL18] einen Ansatz, um sicherzustellen, dass Regeln, welche beispielsweise von Mustern vorgegeben werden, in Deployment-Modellen eingehalten werden, während Saatkamp et al. [SBF+19; SBKL19] Probleme, wie z. B. fehlende Verschlüsslung in der Kommunikation zwischen zwei Komponenten, in Deployment-Modellen identifizieren. Diese Art der Mustererkennung wird in der vorliegenden Arbeit jedoch nicht betrachtet.

2.3.4 Zusammenfassung

Zwar definieren die vorgestellten Konzepte Methoden, um Muster bei der Anwendungsmodellierung einzusetzen, jedoch gibt es bisher keine Modellierungssprache, mit der Deployment-Modelle auf Basis von Mustern beschrieben werden können. Fehling et al. [FLR+11] präsentieren zwar die Vision eines Frameworks für die Modellierung von Cloud-Anwendungen mithilfe von Mustern, jedoch fehlt eine entsprechende Modellierungssprache. Daher präsentiert diese Arbeit eine Modellierungssprache, in der Muster als Hauptmodellelemente verwendet werden können (Forschungsbeitrag 2) und zeigt, wie diese automatisiert zu konkreten Komponenten verfeinert werden können (Forschungsbeitrag 3). Darüber hinaus ist es zudem möglich, Muster in bereits modellierten Deployment-Modellen zu erkennen und erleichtert somit das Verständnis der im Modell umgesetzten Konzepte und Lösungen.

2.4 Generierung von Laufzeitmodellen

Für das Management von Anwendungen werden Laufzeitinformationen der einzelnen Komponenten benötigt. Zum Beispiel muss bekannt sein, unter wel-

cher IP-Adresse eine virtuelle Maschine erreichbar ist und welche Komponenten auf dieser betrieben werden. Anwendungen, für die keine detaillierten Laufzeitinformationen bekannt sind, können somit nicht verwaltet werden. Auch in der „models@run.time“ Community [BBF09; BGS19] werden Laufzeitinformationen der Anwendung benötigt, um diese über Instanzmodelle verwalten zu können. Die meisten dieser Methoden erfordern jedoch ein „a-priori“-*Modell* der Anwendung [BGS19]. Somit muss entweder manuell oder durch andere Methoden ein initiales Modell der Anwendung übergeben werden. Bencomo, Götz und Song [BGS19] zufolge ist die Generierung von Laufzeitmodellen daher noch ein offenes Forschungsgebiet.

Im Allgemeinen wird das Erkennen von sich im Betrieb befindlichen Komponenten auf zahlreiche verschiedene Arten durchgeführt. Zum Beispiel scannen Holm et al. [HBLE14] den Netzwerkverkehr, um daraus aktive Komponenten einer Anwendung abzuleiten, während Brogi, Cifariello und Soldani [BCS17] verschiedene Cloud-Dienste identifizieren, um sie bei der Modellierung von Anwendungen verwenden zu können. Andere Konzepte versuchen mehr Details herauszufinden und identifizieren für jede Komponente ihre Konfiguration, wie zum Beispiel Binz et al. [BBKL13], Farwick et al. [FAB+11], Fittkau, Roth und Hasselbring [FRH15], van Hoorn et al. [HHW+09], Machiraju et al. [MDW+00] und Menzel et al. [MKLT13]. All diese Konzepte konzentrieren sich dabei auf das Abrufen und Ableiten von Komponenten und deren Konfigurationen mithilfe spezieller Software wie Netzwerkscannern, Crawlern oder dynamischer Analyse, ohne dass sie auf bestehende Modelle oder Werkzeuge zurückgreifen. Im Gegensatz dazu wird in der vorliegenden Arbeit ein erstes Modell der Anwendung von den zum Deployment der einzelnen Komponenten verwendeten Deployment-Technologien abgerufen, da diese meist ein Instanzmodell der Anwendungen vorhalten, die sie verwalten. Da diese Instanzmodelle oft jedoch nur grundlegende Informationen über die Komponenten der verwalteten Anwendung bereithalten, werden zusätzlich verschiedene Ansätze, z. B. der Crawler von Binz et al. [BBKL13], in die präsentierte Methode integriert. Damit können die unvollständigen Instanzmodelle der Deployment-Technologien automatisiert um weitere Informationen über die Anwendung erweitern werden.

Generell werden unter dem Sammelbegriff der „*Software Architecture Reconstruction*“ (SAR) [DP09] Methoden entwickelt, welche die Architektur einer Anwendung rekonstruieren und in Modellen repräsentieren. Dazu gehören Ansätze, die die Architektur einer Anwendung aus ihrem Quell-Code [HZ14; NZP+21] oder ihren Deployment-Modellen [NP22; WBSB22] ableiten. Da der Fokus dieser Arbeit jedoch auf dem Management von laufenden Anwendungen liegt, wurden lediglich Ansätze betrachtet, die die Architektur einer laufenden Anwendung rekonstruieren und damit ebenso aktuelle Laufzeitinformationen einer Anwendung herausfinden können.

Bereits laufende Anwendungen nachträglich mit zusätzlichen Managementfunktionalitäten zu erweitern, wird von derzeit existierenden Konzepte für das Management von Anwendungen nicht unterstützt. Da aktuelle Deployment-Technologien im Allgemeinen nur sehr limitierte Managementfunktionalitäten zur Verfügung stellen (s. Abschnitt 2.2), ist das Hinzufügen zusätzlicher Managementfunktionalitäten notwendig. Eine Herausforderung stellt dabei die Einflussnahme der jeweiligen Deployment-Technologien dar, da diese die Anwendung meist überwachen und versuchen die Komponenten in einem bestimmten Zustand zu halten. Wird der Zustand einer Komponente von Dritten verändert, z. B. durch das Ausführen von zustandsändernden Managementfunktionalitäten, können diese Technologien die Änderung bemerken und den vorherigen Zustand wiederherstellen. Dies wird noch weiter erschwert, wenn mehrere Technologien zusammen für die Bereitstellung einer Anwendung verwendet werden. Zum Beispiel wird Terraform oft in Kombination mit anderen Deployment-Technologien wie Chef, Puppet oder Ansible eingesetzt, da es sich eher auf das Infrastrukturmanagement als auf das Konfigurationsmanagement konzentriert [WBB+21]. Daher wird in Forschungsbeitrag 5 gezeigt, wie der Managementansatz aus Forschungsbeitrag 4 verwendet werden kann, um bereits laufende Anwendungen, welche mit unterschiedlichen Deployment-Technologien instanziiert wurden, verwaltet werden können. Darüber hinaus können durch den Forschungsbeitrag 3 sogar umgesetzte Muster der laufenden Anwendung erkannt und in der Form eines musterbasierten Modells dargestellt werden.

2.5 Zusammenfassung

In den vorangegangenen Abschnitten wurde gezeigt, dass (i) die Modellierung von Deployment-Modellen eine große Herausforderung für Modellierende darstellt und (ii) das ganzheitliche Management von verteilten Anwendungen eine komplexe Aufgabe ist. Darüber hinaus wurde (iii) die Übertragung der Managementkonzepte auf bereits laufende Anwendungen, welche möglicherweise mit unterschiedlichen Deployment-Technologien bereitgestellt wurden, bisher kaum betrachtet. Dies erschwert die Modellierung und das Management von Anwendungen erheblich, da viel technisches Wissen über zahlreiche Komponenten und Technologien benötigt wird. Des Weiteren ist die Wartung einer Anwendung durch manuelle, bzw. manuell erstellter, Managementprozesse fehleranfällig und zeitaufwändig.

Es fehlt somit eine Modellierungsmethode, mit der Anwendungen auf deklarative Art und Weise beschrieben werden können, ohne dass Modellierende tiefgreifendes Expertenwissen über entsprechende Technologien und deren exakte Konfiguration haben müssen. Des Weiteren fehlt ein ganzheitliches Managementkonzept, das es erlaubt Anwendungen mit unterschiedlichen Managementfunktionalitäten auszustatten, ohne dass diese von für jede Anwendung neu implementiert werden müssen. Ein nachträgliches Hinzufügen solcher Managementfunktionalitäten zu bereits laufenden Anwendungen reduziert zusätzlich das benötigte technische Expertenwissen.

Durch die Einführung der LAUFFEUER-Methode im folgenden Kapitel werden diese Defizite behoben, indem (i) Deployment-Modelle auf Basis von Muster abstrakt und ohne technische Details beschrieben, (ii) Managementfunktionalitäten ganzheitlich für verteilte Anwendung wiederholbar ausgeführt und (iii) auf laufende Anwendungen übertragen werden können. Somit ermöglicht die vorgestellte Methode die Modellierung von Anwendungen durch die Verwendung von Mustern und erweitert deren Ausführung und Managementmöglichkeiten durch modellgetriebene Ansätze.

Kapitel 3

Eine Methode für das Management von Anwendungen

Dieses Kapitel stellt die LAUFFEUER-Methode[1] vor. Sie ermöglicht es, Anwendungen abstrakt auf Basis von Mustern zu beschreiben, das musterbasierte Deployment-Modell automatisiert in ein ausführbares Deployment-Modell zu transformieren sowie die Anwendung zur Laufzeit automatisiert zu verwalten. Dabei werden alle Phasen von der Modellierung über die Bereitstellung und den Betrieb bis hin zur Außerbetriebnahme einer Anwendung betrachtet. Die LAUFFEUER-Methode kombiniert alle im Rahmen dieser Arbeit veröffentlichten Arbeiten und bildet den Forschungsbeitrag 1.

In Abschnitt 3.1 wird zunächst ein Überblick über die Methode gegeben, bevor die einzelnen Schritte im Detail beschrieben werden. In Abschnitt 3.2 werden weitere Varianten der Methode vorgestellt, während verwandte Methoden des Anwendungsmanagements in Abschnitt 3.3 abgegrenzt werden.

[1] Laufende Anwendungen einfrieren, auftauen und verwalten

3.1 Die LAUFFEUER-Methode

Die LAUFFEUER-Methode ermöglicht es, die Bereitstellung von Anwendungen auf Basis ihrer Muster zu beschreiben und anschließend bereitzustellen, zu verwalten, außer Betrieb zu nehmen sowie im selben Zustand wieder zu instanziieren. Alle Schritte der Methode und deren jeweiligen Ein- bzw. Ausgabemodelle sind in Abbildung 3.1 dargestellt. Insgesamt besteht die LAUFFEUER-Methode aus sechs Schritten sowie zwei Einstiegsschritten, welche durch gestrichelte Linien in Abbildung 3.1 dargestellt sind. Bis auf den Einstiegsschritt A über die *Modellierung eines musterbasierten Deployment-Modells* können alle Schritte automatisiert werden. Die prototypische Umsetzung wird im Kapitel 8 gezeigt. Im Folgenden werden die Schritte erläutert.

3.1.1 Schritt A: Modellierung eines musterbasierten Deployment-Modells

Den Haupteinstiegspunkt in die LAUFFEUER-Methode bildet die manuelle *Modellierung eines musterbasierten Deployment-Modells*. Dafür wird in Kapitel 4 ein Metamodell definiert, das es erlaubt sowohl konkrete Komponenten als auch Muster in einer Anwendungstopologie darzustellen. Die Muster können dabei Konzepte abstrakter Komponenten repräsentieren sowie das Verhalten eines anderen Musters oder einer konkreten Komponente definieren. Um beispielsweise zu beschreiben, dass eine in Java implementierte Logikkomponente auf einem PaaS-Angebot eines beliebigen Cloud-Anbieters bereitgestellt werden soll, kann dies mit den beiden Cloud Computing Patterns PLATFORM AS A SERVICE [FLR+14] und PUBLIC CLOUD [FLR+14] beschrieben werden. Ein entsprechendes musterbasierten Deployment-Modell definiert eine Java-Anwendung, welche eine Relation zu einem PLATFORM AS A SERVICE-Muster hat und anzeigt, dass diese darauf bereitgestellt werden soll. Ähnlich dazu hat das PaaS-Muster eine Relation zu einem PUBLIC CLOUD-Muster. Soll die Logikkomponente darüber hinaus auch unvorhersagbare Arbeitslasten bewältigen können, so kann das UNPREDICTABLE WORKLOAD-Muster [FLR+14] an die entsprechende Komponente im musterbasierten Deployment-Modell annotiert werden.

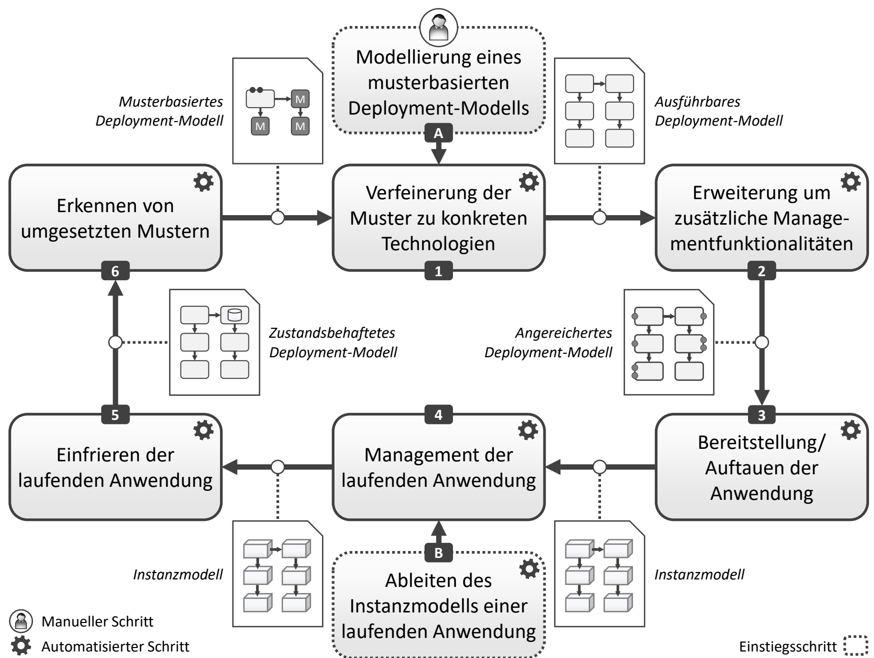


Abbildung 3.1: Übersicht über die Schritte der LAUFFEUER-Methode

3.1.2 Schritt 1: Verfeinerung der Muster zu konkreten Technologien

Ist eine Anwendung in einem musterbasierten Deployment-Modell beschrieben, so beinhaltet das Modell abstrakte Elemente, welche vor der Bereitstellung zu konkreten Komponenten verfeinert werden müssen. Diese Verfeinerung kann automatisiert erfolgen und verwendet dabei sogenannte *Verfeinerungsmodelle*. Verfeinerungsmodelle sind Triple Graph Grammars [Sch95], wobei der Ausgangsgraph ein musterbasierten Deployment-Modell ist, während der Zielgraph konkrete Komponenten und deren Konfigurationen enthält. Somit definieren Verfeinerungsmodelle, wie bestimmte Kombinationen aus Mustern und Komponenten mit konkreten Technologien umgesetzt werden können und realisieren die von Falkenthal et al. [FBBL17] eingeführte Abbildung zwischen „*Pattern Languages*“ und „*Solution Languages*“.

In Schritt 1 der LAUFFEUER-Methode, *Verfeinerung der Muster zu konkreten Technologien*, werden daher Verfeinerungsmodelle dazu verwendet, um die Muster in musterbasierten Deployment-Modellen zu konkreten Komponenten und Technologien zu verfeinern. Der Verfeinerungsprozess wird in Kapitel 5 beschrieben und bekommt, wie in Abbildung 3.1 dargestellt, ein musterbasiertes Deployment-Modell als Eingabe und generiert ein *ausführbares Deployment-Modell*. Optional kann das resultierende Modell von Modellierenden manuell erweitert und konfiguriert werden, um beispielsweise den Benutzernamen oder benötigten Arbeitsspeicher einer VM anzupassen.

3.1.3 Schritt 2: Erweiterung um zusätzliche Managementfunktionalitäten

Nachdem ein musterbasiertes Deployment-Modell zu einem ausführbaren Deployment-Modell transformiert wurde, kann im zweiten Schritt die *Erweiterung um zusätzliche Managementfunktionalitäten* erfolgen. Dabei werden die konkreten Technologien in dem ausführbaren Deployment-Modell analysiert und verfügbare Managementfunktionalitäten gesucht. Welche Funktionalitäten für die aktuelle Bereitstellung genutzt werden sollen, kann entweder manuell oder komplett automatisiert ausgewählt werden. Dadurch können Ressourcen geschont werden, wenn nicht für jede Bereitstellung alle Funktionalitäten benötigt werden, beispielsweise für Testbereitstellungen.

Durch die Erweiterung des Deployment-Modells um zusätzliche Managementfunktionalitäten wird ein *angereichertes Deployment-Modell* generiert. Dieses beinhaltet, neben dem Modell und den IAs und DAs der Anwendung, auch die Implementierungen für die erweiterten Managementfunktionalitäten. Für die Ausführung der Managementfunktionalitäten werden anschließen Workflows generiert, die jeweils die konkrete Abfolge der notwendigen Operationen definieren und die Managementfunktionalität wiederholbar ausführen können. Um beispielsweise zu testen, ob alle Komponenten nach der Bereitstellung erreichbar sind, werden zuerst Infrastrukturkomponenten wie VMs und anschließend Logikkomponenten getestet. Weitere Funktionalitäten sind dabei z.B. das *Einfrieren* und *Auftauen* einer Anwendung. Das Konzept der Mangementerwiterung wird in Kapitel 6 erläutert.

3.1.4 Schritt 3: Bereitstellung/Auftauen der Anwendung

Im nächsten Schritt wird die modellierte Anwendung bereitgestellt. Dafür werden bereits existierende Arbeiten verwendet, wie die Kombinationen von deklarativen und imperativen Deployment-Modellen [BBK+14] oder die parallele Verwendung mehrerer Deployment-Technologien [WBB+21].

Soll allerdings eine zuvor eingefrorene Anwendung wieder bereitgestellt, d. h. *aufgetaut* werden, müssen weitere Schritte während der Bereitstellung erfolgen. Dabei muss der Zustand aller zustandsbehafteten Komponenten wiederhergestellt werden. Je nach Komponente können allerdings unterschiedliche Reihenfolgen der Lebenszyklusoperationen oder lediglich einzelnen Operationen benötigt werden. Daher werden für das Auftauen von Anwendungen separate Workflows generiert. Die Generierung dieser ist Teil des Forschungsbeitrags 4 und wird in Kapitel 6 beschrieben.

Unabhängig davon, ob eine Anwendung bereitgestellt oder aufgetaut wird, wird in diesem Schritt ein ausführbares und angereichertes Deployment-Modell verarbeitet und instanziiert. Durch die Instanziierung werden die Laufzeitinformationen der einzelnen Komponenten in einem *Instanzmodell* gespeichert. Dieses entspricht dabei dem Deployment-Modell, wobei alle Eigenschaften der Komponenten und Relationen konkrete Werte zugewiesen bekommen und Komponenten sowie deren Relationen aufgrund mehrerer Instanzen in der entsprechenden Anzahl enthalten sein können.

3.1.5 Schritt 4: Management der laufenden Anwendung

Mit der Bereitstellung einer Anwendung wird ein Instanzmodell erzeugt, welches die Laufzeitinformationen der einzelnen Komponenten und Relationen beschreibt. Es beinhaltet beispielsweise die aktuellen IP-Adressen von virtuellen Maschinen, dokumentiert die Endpunkte von Topics und Queues oder die aktuelle Zahl an Instanzen aller Komponente. Ein solches Instanzmodell wird in diesem Schritt für das *Management der laufenden Anwendung* verwendet, um die im zweiten Schritt (s. 3.1.3) generierten Workflows mit den aktuellen Laufzeitinformationen der Anwendung ausführen zu können.

Durch die Verwendung von Workflows für die Ausführung von Managementfunktionalitäten können diese beispielsweise unterbrochen und wieder fortgesetzt werden [LR00]. Zusätzlich können inkonsistente Zustände der zu verwaltenden Anwendung vermieden werden, indem Transaktionen und die Möglichkeit zur Fehlerbehandlung in den Workflows verwendet werden [LR00]. Wenn beispielsweise beim Installieren von Updates an einer Komponente ein Fehler auftritt, kann die Lauffähigkeit der gesamten Anwendung beeinträchtigt sein. Werden alle Update-Operationen in einer Transaktion mit Fehlerbehandlung ausgeführt, kann der vorherige, lauffähige Zustand wiederhergestellt werden [KLZ20]. Dadurch können Ausfälle und die damit verbunden Kosten reduziert oder sogar vermieden werden.

3.1.6 Schritt 5: Einfrieren der laufenden Anwendung

Die Außerbetriebnahme einer Anwendung ist Bestandteil jeder Deployment-Technologie und kann durch diese automatisiert erfolgen. Dabei werden die gebundenen Ressourcen wieder frei und verursachen keine Kosten mehr. Allerdings geht der aktuelle Zustand der Anwendung verloren, falls keine Datensicherung der zustandsbehafteten Komponenten vorgenommen wurde.

Im Schritt fünf der LAUFFEUER-Methode wird dies verhindert, indem das *Einfrieren der laufenden Anwendung* ermöglicht wird. Dabei wird vor der Terminierung aller zustandsbehafteten Komponenten eine Sicherung erstellt. Diese Sicherungen werden in einem *zustandsbehafteten Deployment-Modell* dauerhaft abgelegt, sodass die Anwendung bei einer erneuten Bereitstellung im zuvor eingefrorenen Zustand wiederhergestellt werden kann.

Das Einfrieren einer laufenden Anwendung benötigt als Eingabe das aktuelle Instanzmodell der Anwendung, um dem Workflow die entsprechenden Laufzeitinformationen der Komponenten zu übergeben. Dabei wird ein zustandsbehaftetes Deployment-Modell generiert, das, ähnlich zum Self-contained Deployment-Modell [ZBH+20], alles beinhaltet, um eine Anwendung in dem aktuellen Zustand wieder auftauen zu können. Dadurch wird es möglich, Anwendungen, die zeitweise nicht genutzt werden, herunterzufahren und damit Ressourcen zu schonen und Kosten zu sparen.

Ob sich das Einfrieren einer Anwendung lohnt, ist allerdings von zahlreichen Faktoren abhängig. Dabei sind beispielsweise die Datenmenge, die Dauer der Abschaltung oder die Geschwindigkeit der Netzwerkanbindung relevante Faktoren. Müssen zum Sichern einer Datenbank beispielsweise mehrere Terabyte übertragen werden, können zusätzliche Kosten für die Nutzung des Netzwerks anfallen oder die Sicherung dauert länger als die geplante Abschaltung. Daher kann es sich lohnen eine Anwendung ungenutzt weiterlaufen zu lassen, anstatt sie herunterzufahren und einzufrieren.

3.1.7 Schritt 6: Erkennen von umgesetzten Mustern

Beim Einfrieren einer Anwendung wird ein zustandsbehaftetes Deployment-Modell generiert. Dieses enthält möglicherweise andere Komponenten als das Modell, mit dem die Anwendung ursprünglich bereitgestellt wurde. Zum Beispiel kann sich durch das Installieren von Updates oder Ausführen anderer Managementfunktionalität die Zusammensetzung der Komponenten ändern. Dies spiegelt sich im Instanzmodell der Anwendung wider und schließlich auch in dem davon abgeleiteten zustandsbehafteten Deployment-Modell.

Um festzustellen, welche architektonischen Konzepte in der eingefrorenen Anwendung verwendet wurden, kann in Schritt sechs, durch das *Erkennen von umgesetzten Mustern*, wieder ein musterbasiertes Deployment-Modell der Anwendung generiert werden. Wie auch bei der Musterverfeinerung werden bei der Mustererkennung die Verfeinerungsmodelle verwendet, um das Modell zu transformieren. Dabei wird die Bidirektionalität der Verfeinerungsmodelle genutzt: In diesem Fall ist das musterbasierte Deployment-Modellfragment der Zielgraph, während das Fragment mit den konkreten Komponenten und passenden Konfigurationen den Ausgangsgraphen darstellt. Dadurch werden in diesem Schritt aus zustandsbehafteten, bzw. im Allgemeinen ausführbaren, Deployment-Modellen musterbasierte Deployment-Modelle generiert. Diese können im Anschluss wieder durch Schritt eins zu ausführbaren Deployment-Modellen konkretisiert werden (s. Abschnitt 3.1.2). Dabei muss das zustandsbehaftete Deployment-Modell nicht zwingend mit dem neu verfeinerten Modell übereinstimmen, da die

Muster zu anderen Technologien und Service-Angeboten verfeinert werden können. Die Zustände der Komponenten sind dabei sowohl im generierten musterbasierten als auch im ausführbaren Deployment-Modell erhalten.

3.1.8 Schritt B: Ableiten des Instanzmodells einer laufenden Anwendung

Die LAUFFEUER-Methode hat neben dem Einstieg über die Modellierung eines musterbasierten Deployment-Models (s. Abschnitt 3.1.1) einen zweiten Einstiegsschritt wie in Abbildung 3.1 dargestellt. Dabei ist es möglich, bereits laufende Anwendungen, welche mit anderen Deployment-Technologien, oder sogar manuell, bereitgestellt wurden, um weitere Managementfunktionalitäten zu erweitern und diese anschließend auszuführen.

Um eine bereits laufende Anwendung nachträglich verwalten zu können, wird ein Instanzmodell der Anwendung benötigt. Dieses kann von den eingesetzten Deployment-Technologien, welche dazu verwendet wurden, um die Anwendung bereitzustellen, automatisiert abgerufen und vervollständigt werden (s. Kapitel 7). Im Anschluss können zusätzliche Managementfunktionalitäten ausgewählt und die dafür entsprechenden Workflows generiert werden (analog zu Abschnitt 3.1.3). Dabei müssen jedoch weitere Bedingungen beachtet werden, da die Deployment-Technologien, welche die Anwendung bereitgestellt haben, möglicherweise die Anwendung überwachen. Werden Zustandsänderungen an der Anwendung von extern vorgenommen, so können die Deployment-Technologien dies bemerken und die Änderungen verwerfen, indem sie den ihnen bekannten Zustand wiederherstellen. Entsprechende Managementfunktionalitäten müssen die Deployment-Technologien daher über die Zustandsänderung informieren, was bei der Auswahl durch entsprechende Annotationen beachtet wird.

Auch das Einfrieren einer laufenden Anwendung durch diese Methode ist möglich. Allerdings fehlen im abgeleiteten Instanzmodell meist die für die Bereitstellung benötigten Artefakte der Logikkomponenten. Werden die Artefakte jedoch später manuell von Modellierenden zu dem Modell hinzugefügt, kann die Anwendung wieder bereitgestellt bzw. aufgetaut werden.

3.2 Varianten der Lauffeuer-Methode

Im vorherigen Abschnitt 3.1 wurden die einzelnen Schritte der Lauffeuer-Methode sowie die beiden Haupteinstiegspunkte vorgestellt. Im Folgenden werden weitere Einstiegspunkte und Varianten präsentiert und diskutiert.

Variante 1 (Einstieg mit einem existierenden, ausführbaren Deployment-Modell). Mit einem bestehenden und ausführbaren Deployment-Modell kann auf zwei Arten in die Lauffeuer-Methode eingestiegen werden:

Variante 1.1 (Einstieg mit Schritt 2: Erweiterung um zusätzliche Managementfunktionalitäten). Die Erweiterung von Deployment-Modellen um weitere Managementfunktionalitäten basiert auf der Analyse der in einem Modell verwendeten Komponenten (s. Abschnitt 3.1.3). Die Managementerweiterung benötigt somit ein ausführbares Deployment-Modell als Eingabe, da die verfügbaren Managementfunktionalitäten von den konkret verwendeten Technologien abhängig sind. Es macht folglich keinen Unterschied, ob das ausführbare Deployment-Modell durch eine Verfeinerung eines musterbasierten Deployment-Modells oder manuell durch Modellierende erstellt wurde. Folglich kann die Lauffeuer-Methode auch auf bereits existierende Deployment-Modelle angewendet werden.

Variante 1.2 (Einstieg mit Schritt 6: Erkennen von umgesetzten Mustern). Abbildung 3.1 zeigt als Eingabe des sechsten Schritts ein zustandsbehaftetes Deployment-Modell. Im Allgemeinen sind zustandsbehaftete Deployment-Modelle eine Sonderform der ausführbaren Deployment-Modelle, da diese lediglich weitere Artefakte für die Wiederherstellung des letzten Anwendungszustands beinhalten. Das Erkennen von umgesetzten Mustern kann daher auch auf bereits existierende Deployment-Modelle angewendet werden. Dabei entsteht wiederum ein musterbasiertes Deployment-Modell, das im Anschluss zu konkreten Technologien verfeinert werden kann. Wie bereits in Abschnitt 3.1.7 beschrieben, kann sich die Technologie- und Anbieterauswahl der beiden ausführbaren Modelle jedoch unterscheiden.

Variante 2 (Nichtausführen einzelner Schritte). Das Ausführen einzelner Schritte der Methode ist optional. Allerdings bedingen sich einzelne Schritte gegenseitig und können daher nur gemeinsam ausgeführt werden.

Variante 2.1 (Nichtausführen des Anwendungsmanagements, Schritt 4). Die Laufzeit einzelner Anwendung variiert stark. So gibt es Anwendungen, welche über mehrere Wochen, Monate oder Jahre laufen, während andere lediglich für ein paar Stunden, Minuten oder sogar nur Sekunden genutzt werden. Im Falle der sich lange im Betrieb befindlichen Anwendungen ist das Management ein wichtiger Aspekt, um ihre Sicherheit und Funktionalität sicherzustellen. Wird eine Anwendung jedoch nur kurz benötigt, so kann die Zeit, die für das Management benötigt wird, größer sein als ihre eigentliche Laufzeit. Trotzdem kann eine solche Anwendung mithilfe der LAUFFEUER-Methode verwaltet werden, indem das Management der laufenden Anwendung übersprungen wird. Dadurch ergeben sich keine weiteren Auswirkungen auf andere Schritte der Methode.

Variante 2.2 (Nichtausführen des Einfrierens, Schritt 5). Ähnlich wie bei der Laufzeit einer Anwendung gibt es auch im Umgang mit Anwendungsdaten unterschiedliche Ansätze. Während einige Anwendungen ihre Daten stets persistent in Datenbanken oder Speichern ablegen, verarbeiten andere Anwendungen lediglich die ihnen in einer Anfrage zur Verfügung gestellten Daten. Zweitere können entsprechend als *zustandslose* Anwendungen bezeichnet werden. Folglich braucht eine solche Anwendung nicht eingefroren und wieder aufgetaut werden. Darüber hinaus ist es möglich, dass der aktuelle Zustand einer Anwendung nicht (mehr) benötigt wird und dementsprechend ein Einfrieren der Anwendung nicht notwendig ist. Somit wird beim Herunterfahren der Anwendung kein zustandsbehaftetes Deployment-Modell generiert. Alle weiteren Schritte der LAUFFEUER-Methode können anschließend mit dem aktuellen Modell der Anwendung ausgeführt werden.

Variante 2.3 (Nichtausführen des Erkennens von Mustern, Schritt 6). Um das Erkennen der in einer Anwendung umgesetzten Muster zu erleichtern, wird in Schritt sechs der LAUFFEUER-Methode aus einem ausführbaren bzw.

zustandsbehafteten Deployment-Modell ein musterbasiertes Deployment-Modell generiert. Durch Schritt eins kann dieses wiederum konkretisiert und zu einem ausführbaren Modell verfeinert werden. Wird nach dem Einfrieren Schritt sechs nicht ausgeführt, so entfällt auch Schritt eins. Dafür kann entweder direkt zu Schritt zwei, dem Erweitern um zusätzliche Managementfunktionalitäten, oder Schritt 3, dem Auftauen bzw. der Bereitstellung der Anwendung, übergegangen werden.

3.3 Abgrenzung zu verwandten Arbeiten

Ähnlich zur LAUFFEUER-Methode präsentiert Binz [Bin15] mit der *automatisierten Migration von Anwendungen* (AROMA-Methode) ein Konzept, um Anwendungen zu verwalten. Im Gegensatz zur AROMA-Methode liegt der Fokus der LAUFFEUER-Methode nicht auf der Migration von Anwendungen, sondern ermöglicht, neben der musterbasierten Modellierung und Erkennung von Mustern in ausführbaren Deployment-Modellen, auch das nachträgliche Hinzufügen und Ausführen diverser Managementfunktionalitäten zur Laufzeit einer Anwendung. Der Einstiegspunkt in die LAUFFEUER-Methode über die Ableitung eines Instanzmodells einer Anwendung basiert auf den Grundkonzepten der AROMA-Methode und erweitert diese.

In der Arbeit von Breitenbücher [Bre16] wird die „*Pattern-based Application Management*“ PALMA-Methode vorgestellt. Dabei können Anwendungen deklarativ durch das Anwenden von Managementmustern verwaltet werden. Zum Beispiel kann das Installieren eines Updates einer Komponente deklarativ definiert werden, wobei die betroffenen Komponenten entsprechende Annotationen bekommen: Die alte Version der Komponente muss deinstalliert, die neue installiert und gestartet werden, während von der aktualisierten Komponente abhängige Komponenten möglicherweise neu konfiguriert und gestartet werden müssen. Auf Basis dieser Annotationen werden im Anschluss entsprechende Managementworkflows generiert. Im Gegensatz dazu verwendet die LAUFFEUER-Methode Muster als Modellelemente in Deployment-Modellen und kann diese zu konkrete Technologien

und Komponenten verfeinern. Darüber hinaus ist das Management einer Anwendung dadurch möglich, dass verfügbare Managementfunktionalitäten gesucht und automatisiert zu einem Deployment-Modell hinzugefügt werden können. Anschließend wird für jede Managementfunktionalität ein entsprechender Managementworkflow generiert.

Kapitel 4

Ein Metamodell für die musterbasierte Modellierung von Anwendungen

Im Folgenden wird ein Metamodell für musterbasierte Deployment-Modelle vorgestellt. Es basiert auf dem technologieunabhängigen Essential Deployment Metamodel (EDMM), welches zunächst in Abschnitt 4.1 vorgestellt wird. In Abschnitt 4.2 wird eine Erweiterung von EDMM definiert, um die Modellierung von Topologien mit Mustern zu ermöglichen. Schließlich wird in Abschnitt 4.3 eine Beispielanwendung gezeigt und diskutiert, die auch in den nachfolgenden Kapiteln als durchgängiges Beispiel verwendet wird.

4.1 Das Essential Deployment Metamodel (EDMM)

Das Essential Deployment Metamodel (EDMM) [WBF+19] ermöglicht es, Deployment-Modelle unabhängig von konkreten Deployment-Technologien zu definieren. Es basiert auf einer Untersuchung der 13 meistgenutzten Deployment-Technologien [WBF+19] und beschreibt die Modellelemente, die von allen 13 Deployment-Technologien unterstützt werden. Zu den Deployment-Technologien gehören unter anderem Kubernetes [The23], Ansible [Red23], Terraform [Has23] und Chef [Pro23] sowie anbieterspezifische Technologie wie AWS CloudFormation [Ama22] und OpenStack Heat [Rac23]. Auch Teile des TOSCA-Standards [OAS13b; OAS20] können auf EDMM abgebildet werden – das sogenannte „*TOSCA Light*“ [WBH+20c].

In Zusammenarbeit mit Wurster et al. [WBB+19; WBB+20b; WBH+20b; WBH+20c] wurde darüber hinaus gezeigt, dass auf Deployment-Modelle, die auf EDMM-basieren, automatisiert mit allen 13 Technologien bereitgestellt werden können. Auch die Kombination mehrerer Deployment-Technologien für die Bereitstellung einer Anwendung ist möglich [WBB+21].

Daher wird in der vorliegenden Arbeit EDMM als Grundlage verwendet und dahingehend erweitert, dass musterbasierte Deployment-Modelle, im Folgenden auch als *MDM* abgekürzt, erstellt werden können. Die MDM-Erweiterung wird lediglich zur abstrakten Modellierung genutzt. Vor der Bereitstellung müssen alle Erweiterungselemente zu EDMM-nativen Elementen übersetzt werden, d. h., Muster werden zu konkreten Komponenten und Konfigurationen verfeinert. Somit können die vorgestellten Konzepte mit jeder auf EDMM abbildbaren Deployment-Technologie umgesetzt werden.

In Abbildung 4.1 ist das erweiterte EDMM graphisch dargestellt. Während die hellgrauen Elemente von EDMM definiert werden, stellen die dunkelgrauen Elemente die MDM-Erweiterung dar. Gestrichelte Elemente sind abstrakt. Im Allgemeinen definiert EDMM ein Deployment-Modell als gerichteten, gewichteten und gefärbten Graphen, wobei die Knoten die Komponenten einer Anwendung darstellen und die Kanten ihre Relationen zueinander. Dabei sind die Komponenten und Relationen typisiert, wodurch die Eigenschaften und das Verhalten der jeweiligen Elemente wiederverwendbar definiert wird.

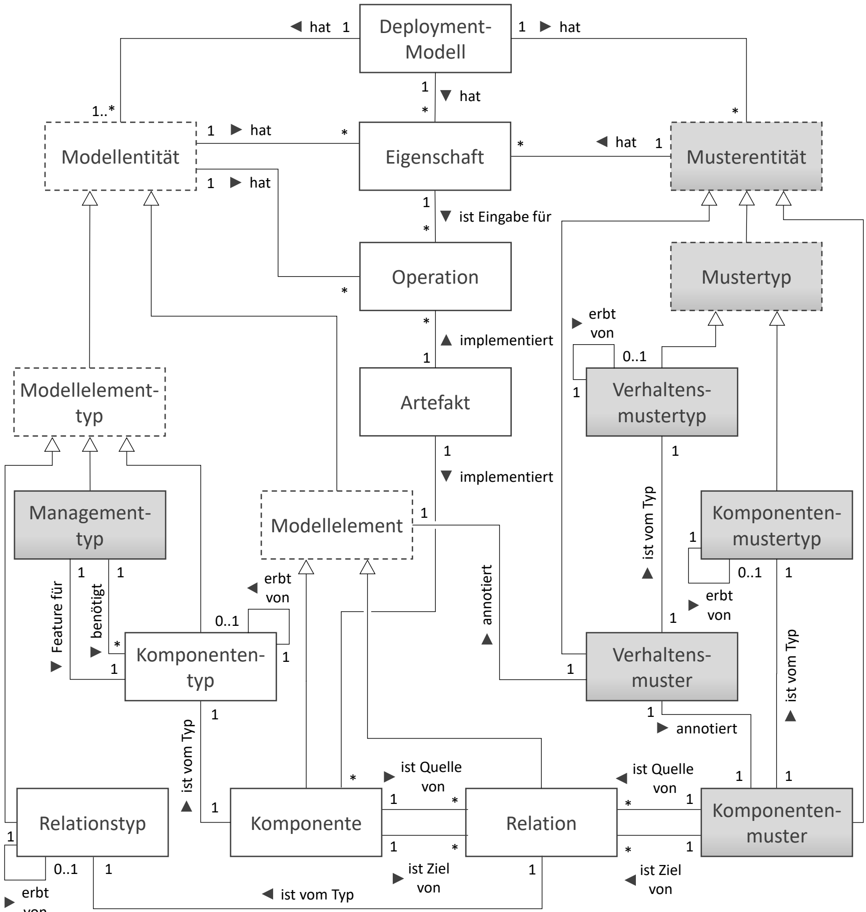


Abbildung 4.1: Darstellung des Metamodells für die musterbasierte Modellierung von Anwendungen. Alle weißen Elemente stammen aus dem zugrundeliegenden EDMM [WBF+19], während alle dunkelgrauen Elemente die in dieser Arbeit eingeführte Erweiterung für die musterbasierte Modellierung darstellen. Aufgrund der Lesbarkeit wurde eine Kante zwischen den Artefakten und den Komponentenmustern weggelassen.

Für die formale Definition von EDMM-Modellen sei Σ die Menge aller Unicode Zeichen. Dann ist die Menge aller Wörter über Σ durch die kleenesche Hülle definiert als Σ^*. Die positive Hülle $\Sigma^+ := \Sigma^* \setminus \{\varepsilon\}$ schließt dabei das leere Wort ε aus. Sei darüber hinaus $\mathcal{B} = \{wahr, falsch\}$ die Menge der booleschen Werte wahr und falsch. Die folgende Definition fasst die Metamodelle DMMN [BEK+16], ETG [Bin15] und DivA [Wil22] zusammen.

Definition 4.1 (Deployment-Modell). Ein EDMM-basiertes deklaratives Deployment-Modell ist ein gerichteter, gewichteter und gefärbter Graph. Die Knoten des Graphen stellen die Komponenten einer Anwendung dar, während die Kanten ihre Relationen beschreiben. Sei $\mathcal{T}$ die Menge aller Deployment-Modelle, dann ist ein Deployment-Modell $t \in \mathcal{T}$ definiert als:

$$t = (K_t, R_t, KT_t, RT_t, E_t, O_t, A_t,$$
$$typ, supertyp, eigenschaften, operationen, artefakte)$$

Dabei sind die einzelnen Elemente des zwölf-Tupels definiert als:

- $K_t \subseteq \Sigma^+$ ist die Menge aller *Komponenten* in t. Eine *Komponente* $k_i \in K_t$ repräsentiert eine physische, funktionale oder logische Einheit einer Anwendung. Dabei identifiziert jedes $k_i \in K_t$ eine Komponente.
- $R_t \subseteq \Sigma^+ \times K_t \times K_t$ ist die Menge aller *Relationen* zwischen zwei Komponenten in t. Eine *Relation* $r_i = (Id, Quelle, Ziel) \in R_t$ ist eine gerichtete physische, funktionale oder logische Abhängigkeit zwischen zwei Komponenten $Quelle, Ziel \in K_t$, wobei *Quelle* der Ausgangsknoten und *Ziel* der Zielknoten der gerichteten Relation darstellt.
- $KT_t \subseteq \Sigma^+ \times \mathcal{B}$ ist die Menge der *Komponententypen* in t. Ein *Komponententyp* $kt_i \in KT_t$ ist eine wiederverwendbare Entität, welche die Semantik einer *Komponente* $k_j \in K_t$ definiert, der dieser Komponententyp zugeordnet ist. Jeder Komponententyp $kt_i = (Name, Abstrakt) \in KT_t$ ist dabei ein Tupel aus dem eindeutig identifizierbaren Namen des Komponententyps sowie der Angabe, ob der Typ eine abstrakte oder instanziierbare Komponente repräsentiert.

- $RT_t \subseteq \Sigma^+ \times \mathcal{B}$ ist die Menge der *Relationstypen* in t. Ein *Relationstyp* $rt_i \in RT_t$ ist eine wiederverwendbare Entität, welche die Semantik einer *Relation* $r_j \in R_t$ definiert, der dieser Relationstyp zugeordnet ist. Analog zu den Komponententypen definiert jeder Relationstyp $rt_i = (Name, Abstrakt) \in RT_t$ einen eindeutig identifizierbaren Namen des Relationstyps sowie die Angabe, ob der Typ eine abstrakte oder instanziierbare Relation beschreibt.

Für die Definition von entitätsübergreifenden Eigenschaften in EDMM werden die nachfolgenden Vereinigungsmengen definiert:

- Die Menge der *Modellelemente* $ME_t = K_t \cup R_t$ ist die Vereinigung aller Komponenten und Relationen eines Deployment-Modells. Zusammen bilden sie den Topologiegraphen einer Anwendung.
- Die Menge der *Modellelementtypen* $MET_t = KT_t \cup RT_t$ ist die Vereinigung aller Komponenten- und Relationstypen eines Deployment-Modells.
- Die Menge der *Modellentitäten* $M_t = ME_t \cup MET_t$ ist die Vereinigung aller Modellelemente sowie -typen in einem Deployment-Modell.

Im Folgenden werden entitätsübergreifende EDMM-Elemente definiert:

- $E_t \subseteq \Sigma^+ \times \Sigma^*$ ist die Menge der *Eigenschaften* in t. Eine *Eigenschaft* $e_i = (Schlüssel, Wert) \in E_t$ beschreibt den angestrebten Zustand eines Modellelements oder den aktuellen Zustand einer Modellentität. Der initiale Wert einer Eigenschaft ist immer das leere Wort ε.
- $O_t \subseteq \Sigma^+ \times \wp(Parameter_t) \times \wp(Parameter_t)$ definiert die Menge der *Operationen* in t. Eine Operation $o_i = (Name, Eingabe, Ausgabe) \in O_t$ ist eine ausführbare Prozedur, um eine Modellentität zu verwalten. Sie besteht aus einem eindeutigen Namen sowie Parametern zur Ein- und Ausgabe. Die Menge der *Parameter* $Parameter_t \subseteq \Sigma^+ \times \Sigma^* \times \mathcal{B}$ beschreibt die Eingabe- bzw. Ausgabewerte der Operation. Ein $param_i = (Schlüssel, Wert, benötigt) \in Parameter_t$ definiert einen Eingabe- bzw. Ausgabewert sowie, ob dieser für die Ausführung der Operation zwingend benötigt wird oder optional ist.

- $A_t \subseteq \Sigma^+ \times \Sigma^+$ ist die Menge der *Artefakte* in t. Ein Artefakt $a_i = (Name, Referenz) \in A_t$ implementiert eine Komponente oder Operation und wird für deren Ausführung benötigt. Es besteht aus einem eindeutigen Namen sowie einer Referenz zu der Datei, welche das Artefakt repräsentiert – zum Beispiel ein Skript oder ZIP-Archiv.

Um die Zuweisung der Typen, Eigenschaften, Operationen und Artefakte zu den jeweiligen Modellentitäten abrufen zu können, werden im Folgenden die entsprechenden Abbildungen definiert. Abbildungen können unabhängig von einem bestimmten Deployment-Modell t definiert werden, da die Abbildungen einer Menge auf eine andere immer auch die Abbildungen aller ihrer Teilmengen, d. h. beispielsweise $ME_t \subseteq ME$ und bezeichnet alle Modellelemente der Topologie t, beinhaltet.

- *typ* ist die Abbildung, die jedem *Modellelement* $me_i \in ME$ seinen entsprechenden *Modellelementtypen* $met_j \in MET$, der seine Semantik definiert, zuweist. Die Abbildung ist definiert als:

  $$typ : ME \rightarrow MET$$

  Dabei gilt, dass eine Komponente immer einem Komponententyp und eine Relation immer einem Relationstyp zugeordnet werden muss. Daher gilt die folgende Einschränkung:

  $$\forall me_i \in ME : \big(me_i \in K : typ(me_i) \in KT\big) \vee \big(me_i \in R : typ(me_i) \in RT\big)$$

- *eigenschaften* ist die Abbildung, die jeder *Modellentität* $m_i \in M$ sowie dem Deployment-Modell selbst ihre Eigenschaften zuweist:

  $$eigenschaften : M \cup \mathcal{T} \rightarrow \wp(E)$$

- *operationen* ist die Abbildung, die jeder *Modellentität* $m_i \in M$ ihre für das Deployment benötigten Operationen zuordnet.

  $$operationen : M \rightarrow \wp(O)$$

- *artefakte* ist die Abbildung, die jeder *Komponente* $k_i \in K$ und jeder *Operation* $o_j \in O$ ihre Artefakte zuweist.

$$artefakte : K \cup O \to \wp(A)$$

Um Vererbung zwischen Modellelementtypen zu erlauben wird die nachfolgende Abbildung definiert. Damit ist es möglich, die Semantik eines Typs, d. h. seine Eigenschaften, Operationen und Artefakte zu vererben.

- *supertyp* ist die Abbildung, die einem *Modellelementtyp* seinen *Supertyp* zuordnet. Dabei wird ein Modellelementtyp $met_i \in MET$ entweder (i) genau einem anderen Modellelementtyp $met_j \in MET$, wobei $i \neq j$ gilt, oder (ii) keinem Modellelementtyp, d. h. „∅“ zugeordnet. Zusätzlich darf ein Komponententyp nur von einem anderen Komponententyp bzw. ein Relationstyp nur von einem andren Relationstyp erben.

  $$supertyp : MET \to MET \cup \emptyset$$

  Dabei gelten die folgenden Einschränkungen:

  $$\forall met_i \in MET : supertyp(met_i) = \emptyset \vee \Big(met_i \neq supertyp(met_i) \\ \wedge \Big(\big(met_i \in KT \Rightarrow supertyp(met_i) \in KT\big) \\ \vee \big(met_i \in RT \Rightarrow supertyp(met_i) \in RT\big)\Big)\Big)$$

- *typen* ist die Abbildung, die den Typen und alle transitiv auflösbaren Supertypen eines *Modellelements* $me_i \in ME$ sowie eines *Modellelementtyps* $met_j \in MET$ aufschlüsselt. Die Abbildung ist definiert als:

  $$typen : ME \cup MET \to \wp(MET)$$

  Dabei darf kein Typ, sei es direkt oder transitiv, von sich selbst erben:

  $$\forall met_i \in MET : met_i \notin typen(met_i)$$

Definition 4.2 (Projektion). Für den Zugriff auf eine Komponente eines Tupels wird die Projektion π definiert. Sei X eine Menge von Tupeln $x_i = (Start, \ldots, \sigma, \ldots, Ende) \in X$, wobei jede Komponente in x_i eindeutig durch einen Namen benannt ist, dann ist die Projektion $\pi_\sigma(x_i)$ eine Abbildung auf die Komponente σ des Tupels x_i.

Um zum Beispiel auf das Ziel einer Relation $r_i \in R_t$ zuzugreifen, kann die Projektion $\pi_{Ziel}(r_i) = Ziel \in r_i \in R_t$ verwendet werden. Die Referenz eines Artefakts $a_j \in A_t$ kann beispielsweise über $\pi_{Referenz}(a_j)$ abgefragt werden.

4.2 EDMM-Erweiterung für die musterbasierte Modellierung

Auf Basis des im vorherigen Abschnitt definierten formalen Metamodells für EDMM wird im Folgenden die Erweiterung für die Modellierung von musterbasierten Deployment-Modelle definiert. In Abbildung 4.1 ist die MDM-Erweiterung durch dunkelgraue Elemente dargestellt.

Definition 4.3 (Musterbasiertes Deployment-Modell, MDM). Ein musterbasiertes Deployment-Modell ist ein Deployment-Modell einer Anwendung, indem zusätzlich zu konkreten Komponenten und deren Konfiguration Muster verwendet werden können. *Komponentenmuster* können als Knoten eingesetzt werden, während *Verhaltensmuster* das Verhalten von Komponenten, Relationen und Komponentenmuster definieren. Sei $\mathcal{T}$ die Menge aller musterbasierten Deployment-Modelle, dann ist ein MDM $t \in \mathcal{T}$ definiert als:

$$t = (K_t, R_t, KT_t, RT_t, E_t, O_t, A_t, \mathbf{KM_t}, \mathbf{KMT_t}, \mathbf{MGT_t}, \mathbf{AT_t}, \mathbf{VM_t}, \mathbf{VMT_t}, \\ typ, supertyp, eigenschaften, operationen, artefakte, \\ \mathbf{annotationen}, \mathbf{features})$$

Dabei sind die zusätzlichen Elemente des zwanzig-Tupels definiert als:

- $KM_t \subseteq \Sigma^+$ ist die Menge aller *Komponentenmuster* in t. Ein Komponentenmuster $km_i \in KM_t$ ist ein Muster, das eine oder mehrere konkrete Komponenten beschreibt und vor dem Deployment zu einer konkreten

Komponente verfeinert werden muss. Jedes $km_i \in KM_t$ identifiziert dabei ein Komponentenmuster.

- $KMT_t \subseteq \Sigma^+$ ist die Menge der *Komponentenmustertypen* in t. Ein Komponentenmustertyp $kmt_i \in KMT_t$ ist eine wiederverwendbare Entität, welche die Semantik eines Komponentenmusters $km_j \in KM_t$ definiert, das dem Komponentenmustertyp kmt_i zugeordnet ist. Jeder Komponentenmustertyp $kmt_i \in KMT_t$ beschreibt den eindeutig identifizierbaren Namen des Komponentenmustertyps.
- $MGT_t \subseteq \Sigma^+ \times \Sigma^+ \times KT_t \times \wp(KT_t) \times \wp(\Sigma^+)$ ist die Menge der *Managementtypen* in t. Ein $mgt_i \in MGT_t$ ist eine wiederverwendbare Entität, welche Managementoperationen für einen Komponententyp definiert. Ein $mgt_i = (Id, Feature, FeatureFür, Benötigt, Technologien) \in MGT_t$ besteht aus (i) einer ID, (ii) dem Namen des Features, (iii) dem Komponententyp, für den das Feature bestimmt ist, (iv) eine Menge an benötigten Komponententypen, welche für die Ausführung der Operationen im Stack einer Anwendung vorhanden sein muss, sowie (v) die Menge der unterstützten Deployment-Technologien.
- $AT_t \subseteq \Sigma^+$ ist die Menge aller Artefakttypen in t. Ein Artefakttyp $at_i \in AT_t$ ist eine wiederverwendbare Entität, welche die Semantik eines Artefakts $a_j \in A_t$ beschreibt. Jeder Artefakttyp $at_i \in AT_t$ beschreibt den eindeutig identifizierbaren Namen des Artefakttyps.

Um die *Verhaltensmuster* und deren *Verhaltensmustertypen* zu definieren, wird zunächst die Hilfsmenge der *Strukturelemente* SE_t definiert. In Anschluss werden weitere Vereinigungsmengen für Muster definiert.

- $SE_t \subseteq ME_t \cup KM_t$ ist die Menge aller *Strukturelemente* in t. Sie vereint alle Komponenten, Relationen und Komponentenmuster eines musterbasierten Deployment-Modells t. Gemeinsam bilden die Strukturelemente den Topologiegraphen eines musterbasierten Modells.
- $VM_t \subseteq \Sigma^+$ ist die Menge aller *Verhaltensmuster* in t. Ein Verhaltensmuster $vm_i \in VM_t$ ist ein Muster, das das Verhalten einer Komponente,

Relation oder eines Komponentenmusters beschreibt, an dem es annotiert ist. Jedes $vm_i \in VM_t$ identifiziert dabei ein Verhaltensmuster.

- $VMT_t \subseteq \Sigma^+$ ist die Menge der *Verhaltensmustertypen* in t. Ein Verhaltensmustertyp $vmt_i \in VMT_t$ ist eine wiederverwendbare Entität, welche die Semantik eines Verhaltensmusters $vm_j \in VM_t$ beschreibt, das dem Verhaltensmustertyp vmt_i zugeordnet ist. Jeder Verhaltensmustertyp $vmt_i \in VMT_t$ enthält den eindeutigen Namen des Typs.
- $MT_t \subseteq KMT_t \cup VMT_t$ ist die Menge aller *Mustertypen* in t. Sie vereinigt dabei die Komponentenmustertypen mit den Verhaltensmustertypen.
- $P_t \subseteq KM_t \cup VM_t \cup MT_t$ ist die Menge aller *Musterentitäten*, wobei P für das englische Wort „*Patterns*" steht. Sie vereinigt sowohl die Mustertypen, als auch die Komponentenmuster und Verhaltensmuster.

Zusätzlich werden bereits existierende Elemente und Abbildungen angepasst:

- Die Menge der Relationen in t wird neu definiert als $R_t \subseteq \Sigma^+ \times MK_t \times MK_t$. Dabei ist $MK_t = K_t \cup KM_t$ die Menge der *Modellknoten*, wobei ein $mk_i \in MK_t$ eine Komponente oder ein Komponentenmuster ist, das die Quelle bzw. das Ziel einer Relation $r_i = (Id, Quelle, Ziel) \in R_t$ darstellt.
- Auch die Menge der Komponententypen wird neu definiert, um angeben zu können, dass ein Komponententyp durch Managementfunktionalitäten erweitert wurde. Daher wird die Menge neu definiert als $KT_t \subseteq \Sigma^+ \times \mathcal{B} \times \mathcal{B}$. Ein $kt_i = (Name, Abstrakt, Angereichert) \in KT_t$ definiert dabei nicht nur den Namen des Komponententyps sowie der Angabe, ob der Typ eine abstrakte oder instanziierbare Komponente repräsentiert, sondern ebenso, ob der Komponententyp durch ein oder mehrere Managementtypen $mgt \in MGT_t$ um zusätzliche Managementfunktionalitäten (s. Kapitel 6) erweitert wurde.
- Die Menge der Modellelementtypen in t beinhaltet nun auch die Managementtypen sowie die Artefakttypen. Somit sind die Modellelementtypen definiert als: $MET_t = KT_t \cup RT_t \cup MGT_t \cup AT_t$.

- Die *typ*-Abbildung wird erweitert, sodass diese auch Komponentenmuster, Verhaltensmuster und Artefakte ihren jeweiligen Typ zuweist:

  $$typ : ME \cup KM \cup VM \cup A \rightarrow MET \cup MT$$

  Managementtypen definieren zusätzliche Managementoperationen für Komponententypen, können jedoch nicht als Typ eines anderen Modellelements verwendet werden. Somit gilt die erweiterte Einschränkung:

  $$\begin{aligned} &\forall e_i \in \{ME \cup KM \cup VM \cup A\} : \\ &\quad \big(e_i \in K \Rightarrow typ(e_i) \in KT\big) \vee \big(e_i \in R \Rightarrow typ(e_i) \in RT\big) \vee \\ &\quad \big(e_i \in KM \Rightarrow typ(e_i) \in KMT\big) \vee \big(e_i \in VM \Rightarrow typ(e_i) \in VMT\big) \vee \\ &\quad \big(e_i \in A \Rightarrow typ(e_i) \in AT\big) \end{aligned}$$

- Da auch Musterentitäten Eigenschaften haben, wird die *eigenschaften*-Abbildung um diese entsprechend erweitert:

  $$eigenschaften : M \cup P \cup \mathcal{T} \rightarrow \wp(E)$$

- Vererbung zwischen Mustertypen ist ebenfalls möglich:

  $$supertyp : MET \cup MT \rightarrow MET \cup MT \cup \emptyset$$

  Wobei gilt, dass ein Typ immer nur vom gleichen Typ erben darf:

  $$\begin{aligned} &\forall et_i \in \{MET \cup MT\} : supertyp(et_i) = \emptyset \vee \Big(et_i \neq supertyp(et_i) \wedge \Big(\\ &\quad \big(et_i \in KT \Rightarrow supertyp(et_i) \in KT\big) \vee \big(et_i \in RT \Rightarrow supertyp(et_i) \in RT\big) \\ &\quad \vee \big(et_i \in KMT \Rightarrow supertyp(et_i) \in KMT\big) \\ &\quad \vee \big(et_i \in VMT \Rightarrow supertyp(et_i) \in VMT\big) \\ &\quad \vee \big(et_i \in AT \Rightarrow supertyp(et_i) \in AT\big)\Big)\Big) \end{aligned}$$

- Entsprechend wird auch die *typen*-Abbildung erweitert:

$$typen : SE \cup MET \cup MT \rightarrow \wp(MET) \cup \wp(MT)$$

  Dabei darf kein Typ, sei es direkt oder transitiv, von sich selbst erben:

$$\forall m_i \in \left\{MET \cup MT\right\} : m_i \notin typen(m_i)$$

- Um Zustandsartefakte bei der Mustererkennung nicht zu verlieren, können Artefakte auch an Komponentenmuster annotiert sein. Aufgrund der Lesbarkeit von Abbildung 4.1 wurde eine Kante zwischen den Artefakten und den Komponentenmustern jedoch weggelassen.

$$artefakte : K \cup O \cup KM \rightarrow \wp(A)$$

Darüber hinaus enthält die MDM-Erweiterung die folgenden Abbildungen:

- *annotationen* ist die Abbildung, die jedem Strukturelement $se_i \in SE_t$ all ihre annotierten Verhaltensmuster zuweist:

$$annotationen : SE \rightarrow \wp(VM)$$

- *features* ist die Abbildung, die allen Komponententypen ihre Managementfunktionalitäten, d. h. Managementtypen, zuordnet:

$$features : KT \rightarrow \wp(MGT)$$

  Ebenso gilt, dass der Komponententyp in den entsprechenden Managementtypen genannt sein muss:

$$\forall kt_i \in KT \ \forall feature \in features(kt_i) : \pi_{FeatrueFür}(feature) = kt_i$$

4.3 Modellierung von musterbasierten Deployment-Modellen

Durch die im Abschnitt 4.2 präsentierte EDMM Erweiterung ist es möglich, Anwendungen abstrakt auf Basis der Muster zu beschreiben, welche für den Betrieb der Anwendung umgesetzt werden sollen. Dies erleichtert die Modellierung von Anwendungen erheblich, da anstelle von konkreten Komponenten und deren Konfigurationen lediglich die abstrakten Konzepte verstanden werden müssen und nicht, wie diese durch bestimmte Technologie konkret umgesetzt werden. Um eine Cloud-Anwendung zu beschreiben, können zum Beispiel die *Cloud Computing Patterns* [FLR+14], die *Enterprise Integration Patterns* [HW04], die *Security Patterns* [SFH+06] sowie die *Hosting Patterns* [YBB+22; YSB+21] verwendet werden.

Die Verwendung von Mustern zur Beschreibung von Anwendungen und deren Bereitstellung wird im Folgenden erläutert. Zuvor wird jedoch eine Einordnung der Muster in die Kategorien *Komponentenmuster* und *Verhaltensmuster* diskutiert sowie eine Liste aller in dieser Arbeit verwendeten Muster und deren Einordnung in Komponenten- und Verhaltensmuster präsentiert.

4.3.1 Einteilung in Komponenten- und Verhaltensmuster

Muster können allgemein in unterschiedliche Kategorien eingeteilt werden. Gamma et al. [GHJV94] teilen ihre Muster beispielsweise in die drei Kategorien *Erstellungsmuster*, *Strukturmuster* und *Verhaltensmuster* (s. Abschnitt 2.3) ein, während Fehling et al. [FLR+14] insgesamt 13 Kategorien für die Gruppierung der Cloud Computing Patterns definieren.

Um Anwendungen auf Basis von Mustern modellieren zu können, werden Muster im Rahmen dieser Arbeit in *Komponentenmuster* und *Verhaltensmuster* eingeteilt. Dabei ist ein Komponentenmuster ein Muster, das eine oder mehrere Komponenten in einer Topologie repräsentieren kann und somit die Struktur der Anwendung beschreibt und beeinflusst. Ein Beispiel für ein Komponentenmuster ist das bereits erwähnte PLATFORM AS A SERVICE-Muster [FLR+14]. Es definiert, wie eine Logikkomponente ohne zusätzlichen Managementaufwand bereitgestellt werden kann. Unterschiedliche Anbieter

setzten das PaaS-Muster um: zum Beispiel AWS mit dem *Beanstalk*-Service, der *App Service* von Microsoft oder die *App Engine* von Google. Somit repräsentiert das PaaS-Muster in einem musterbasierten Deployment-Modell eine dieser konkreten Lösungen, die in einem ausführbaren Deployment-Modell als konkrete Komponenten dargestellt werden. Im Gegensatz dazu ist ein Verhaltensmuster ein Muster, das (i) das Verhalten oder (ii) weitere Informationen, wie die Art der Komponente, einer Komponente oder eines Komponentenmusters beschreibt. So kann beispielsweise das UNPREDICTABLE WORKLOAD-Muster [FLR+14] dazu verwendet werden, um zu beschreiben, dass für eine Komponente oder ein Komponentenmuster eine nicht vorhersagbare Auslastung erwartet wird. Während der Verfeinerung können daher bereits die Technologien herausgefiltert werden, die mit einem solchen Verhalten umgehen können.

Definition 4.4 (Komponentenmuster – informell). Ein Komponentenmuster ist ein Muster, das eine oder mehrere Komponenten einer Anwendung abstrakt repräsentiert. Dabei beeinflusst es die Struktur der in einem musterbasierten Deployment-Modell beschriebenen Anwendung.

Definition 4.5 (Verhaltensmuster – informell). Ein Verhaltensmuster ist ein Muster, das das Verhalten eines Komponentenmusters, einer konkreten Komponente oder einer Relation abstrakt definiert. Darüber hinaus können Verhaltensmuster dazu verwendet werden, um zusätzliche Informationen an ein Komponentenmuster, eine Komponente oder eine Relation zu annotieren.

Die Einteilung von Muster in Komponenten- und Verhaltensmuster ist, im Gegensatz zu der Kategorisierung innerhalb der jeweiligen Mustersprache, nicht fest, sondern kann je nach Sichtweise variieren. Zum Beispiel kann das USER INTERFACE COMPONENT-Muster [FLR+14] in einem Modell mit hohem Abstraktionsgrad als Komponentenmuster verwendet werden, während es als Verhaltensmuster in einem konkreteren Modell verwendet werden kann, wenn der Typ der Komponente bereits definiert ist. Andererseits gibt es Muster, die immer ein Verhalten oder eine Komponente beschreiben. Zum Beispiel beschreibt das AT-LEAST-ONCE DELIVERY-Muster [FLR+14], dass

eine Nachricht mindestens einmal zugestellt werden muss. Dabei beschreibt es eindeutig das Verhalten einer Messaging-Komponente. Ein Beispiel für ein Komponentenmuster ist das RELATIONAL DATABASE-Muster [FLR+14]. Im Allgemeinen können die meisten Komponentenmuster jedoch auch dazu verwendet werden, zusätzliche Information an Komponenten zu annotieren.

In der nachfolgenden Tabelle 4.1 werden alle in dieser Arbeit verwendeten Muster kurz mit ihrem Icon, Namen sowie der Problemstellung, die sie lösen, vorgestellt. In der letzten Spalte „Typ“ ist zudem angegeben, in welche Kategorie das jeweilige Muster eingeordnet wurde. Dabei steht „K“ für ein Komponentenmuster, während „V“ auf ein Verhaltensmuster hindeutet. Die Muster sind in der Tabelle alphabetisch nach ihrem Namen sortiert.

Tabelle 4.1: Übersicht der in dieser Arbeit verwendeten Muster

Icon	Name	Problemstellung	Typ
	CONSUMER-MANAGED SCALING CONFIGURATION [YBB+22]	Wie kann eine Softwarekomponente bereitgestellt werden, die horizontal skaliert werden muss und gleichzeitig eine maßgeschneiderte Skalierungskonfiguration erfordert?	V
	EVENT-DRIVEN CONSUMER [HW04]	Wie kann eine Anwendung automatisch Nachrichten abrufen, sobald sie verfügbar sind?	V
	EXACTLY-ONCE DELIVERY [FLR+14]	Wie kann sichergestellt werden, dass eine Nachricht nur genau einmal an einen Empfänger zugestellt wird?	V
	EXECUTION ENVIRONMENT [FLR+14]	Wie können sich mehrere Anwendungskomponenten eine Hosting-Umgebung effizient teilen?	K

	INFORMATION OBSCURITY [SFH+06]	Wie kann sichergestellt werden, dass die von einem System erfassten und sensiblen Daten vor unberechtigtem Zugriff geschützt sind?	V
	MESSAGE-ORIENTED MIDDLEWARE [FLR+14]	Wie können Kommunikationspartner Informationen asynchron mit einem anderen Partner austauschen?	K
	PLATFORM AS A SERVICE [FLR+14]	Wie können Anwendugen mehrerer Kunden dieselbe Ausführungsumgebung nutzen, sodass diese auf Nachfrage und mit einem pay-per-use Preismodell genutzt werden kann?	V
	PRIVATE CLOUD [FLR+14]	Wie können die Eigenschaften der Cloud in Umgebungen mit hohen Anforderungen an Datenschutz, Sicherheit und Vertrauen realisiert werden?	K
	PROCESSING COMPONENT [FLR+14]	Wie kann die Verarbeitung elastisch auf verteilte Ressourcen skaliert werden, während unterschiedliche Konfigurationen für verschiedene Kunden umgesetzt werden können?	V
	PROVIDER-MANAGED SCALING CONFIGURATION [YBB+22]	Wie kann eine Komponente bereitgestellt werden, die horizontal skaliert werden muss, jedoch keine spezielle Skalierungskonfiguration erfordert?	V
	PUBLIC CLOUD [FLR+14]	Wie können die Eigenschaften der Cloud für eine große Kundegruppe angeboten werden?	K
	MESSAGE CHANNEL [HW04]	Wie kommuniziert eine Anwendung mit einer anderen über Messaging?	K

	Relational Database [FLR+14]	Wie können Daten so gespeichert werden, dass Beziehungen zwischen ihnen ausgedrückt werden können und Abfragen möglich sind?	K
	Secure Channel [SFH+06]	Wie kann die Übertragung von Daten im öffentlichen oder halböffentlichen Raum abgesichert werden?	V
	Static Workload [FLR+14]	Wie kann eine gleichmäßige Auslastung einer Anwendung charakterisiert werden und wie können diese von Cloud Computing profitieren?	V
	Serverless Hosting [YSB+21]	Wie kann eine Softwarekomponente bereitgestellt werden, ohne die Verwaltung der eingesetzten Technologien oder der Skalierungskonfiguration selbst übernehmen zu müssen?	K
	Stateless Component [FLR+14]	Wie kann die Elastizität einer Anwendungskomponente erhöht werden?	V
	Unpredictable Workload [FLR+14]	Wie können zufällige und unvorhersehbare Auslastungen charakterisiert werden und wie können Anwendungen, die diese Auslastungen erfahren, von Cloud Computing profitieren?	V
	User Interface Component [FLR+14]	Wie können Benutzeroberflächen interaktiv von Menschen angesprochen werden, während sie gleichzeitig konfigurierbar und von der restlichen Anwendung entkoppelt sind?	V

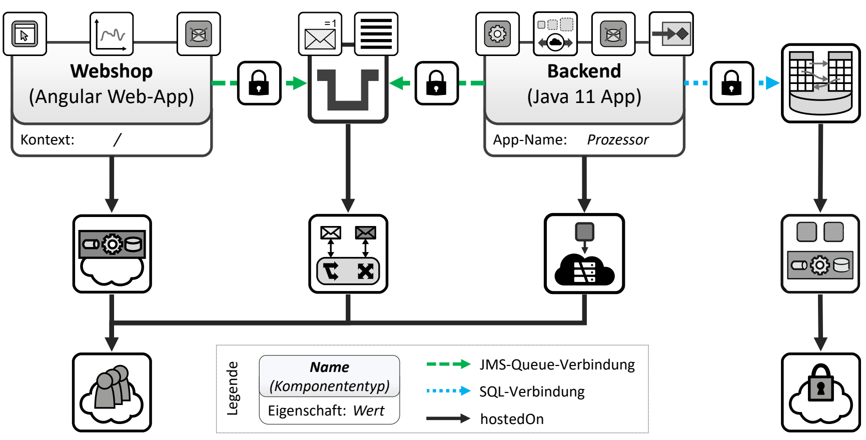


Abbildung 4.2: Ein Beispiel für ein musterbasiertes Deployment-Modell

4.3.2 Musterbasiertes Deployment-Modell einer Beispielanwendung

Ein Beispiel, wie eine Anwendung auf Basis von Mustern beschrieben werden kann, ist in Abbildung 4.2 dargestellt. Alle verwendeten Muster sind in Tabelle 4.1 mit ihrem Icon und Namen gelistet. Die gezeigte Anwendung besteht aus einer Benutzeroberfläche mit dem Namen *Webshop* sowie einer *Backend*-Komponente, die sich zu einer Relational Database verbindet. Zur Kommunikation zwischen den beiden Logikkomponenten soll ein Message Channel verwendet werden, welcher von einer Message-oriented Middleware auf einer Public Cloud bereitgestellt wird. Während der Webshop eine Instanz des Komponententyps *Angular Web-App* ist, die direkt auf dem Hauptkontext, d. h. „/“, erreichbar sein soll, ist das Backend eine *Java 11 App*, welche den App-Namen „Prozessor“ haben soll. Der Webshop soll von einem PaaS-Angebot derselben Public Cloud bereitgestellt werden, auf der auch der Message Channel laufen soll. Ähnlich dazu soll das Backend auf einem Serverless Hosting-Angebot derselben Public Cloud bereitgestellt werden. Die Relationen, die die Bereitstellung einer Komponente auf einer anderen beschreiben, sind dabei jeweils vom Relationstyp *hostedOn* und als schwarze Pfeile in Abbildung 4.2 dargestellt. Im Gegensatz zu

den Logikkomponenten soll die relationale Datenbank jedoch aufgrund von Datenschutzverordnungen auf einem EXECUTION ENVIRONMENT in einer PRIVATE CLOUD bereitgestellt werden. Dementsprechend sollen auch die Kommunikationswege zwischen dem Webshop, des MESSAGE CHANNELS, des Backends sowie der Datenbank abgesichert sein. Daher sind alle horizontalen Relationen mit einem SECURE CHANNEL-Muster annotiert. Dabei sind die Relationen zu dem MESSAGE CHANNEL-Muster jeweils vom Typ *JMS-Queue-Verbindung*, während die Relation zwischen Backend und Datenbank eine Instanz des Relationstyps *SQL-Verbindung* ist.

Um das Verhalten der Komponenten sowie des MESSAGE CHANNELS anzugeben, sind diese mit Verhaltensmuster annotiert. Da der Webshop eine Web-basierte Benutzeroberfläche ist, wurde er mit dem USER INTERFACE COMPONENT-Muster annotiert. Darüber hinaus hält der Webshop selbst keinen Zustand und wird eine nicht vorhersagbare Anzahl an parallelen Zugriffen erfahren. Daher wurde er mit dem UNPREDICTABLE WORKLOAD-Muster sowie dem STATELESS COMPONENT-Muster annotiert. Auch die Backend-Komponente ist zustandslos und entsprechend gekennzeichnet. Allerdings ist diese direkt mit dem PROVIDER-MANAGED SCALING CONFIGURATION-Muster, anstatt dem UNPREDICTABLE WORKLOAD annotiert. Dementsprechend soll der Cloud-Anbieter die automatisierte Skalierung der Komponente übernehmen. Außerdem sind das PROCESSING COMPONENT-Muster und das EVENT-DRIVEN CONSUMER-Muster annotiert, da das Backend aufgerufen werden soll, sobald Anfragen auf dem MESSAGE CHANNEL eingehen.

Während das USER INTERFACE COMPONENT- und PROCESSING COMPONENT-Muster keine Auswirkungen auf die Konfigurationen der Logikkomponenten haben, können aufgrund der anderen Verhaltensmuster konkrete Konfigurationen abgeleitet werden. Zum Beispiel kann von der nicht vorhersagbaren Auslastung des Frontends abgeleitet werden, dass es skaliert werden muss. In Kombination mit dem STATELESS COMPONENT-Muster kann zudem eine Konfiguration für eine horizontale Skalierung abgeleitet werden. Sind Komponenten nicht zustandslos, kann eine horizontale Skalierung nicht automatisch abgeleitet werden, da sonst die Zustände verloren gehen oder sie zwischen den Instanzen synchronisiert werden müssen.

Der MESSAGE CHANNEL ist zusätzlich mit den beiden Mustern EXACTLY-ONCE DELIVERY und INFORMATION OBSCURITY annotiert. Durch das EXACTLY-ONCE DELIVERY-Muster soll sichergestellt werden, dass jede Nachricht genau einmal zu einer Instanz des Backends zugestellt wird und die Nachrichten nicht mehrfach verarbeitet werden. Das INFORMATION OBSCURITY-Muster beschreibt, dass, wie auch der SECURE CHANNEL bei der Übertragung der Daten zwischen den einzelnen Anwendungskomponenten, die in dem MESSAGE CHANNEL gespeicherten Nachrichten vor unbefugten Zugriffen geschützt werden müssen. Dies kann beispielsweise durch eine verschlüsselte Speicherung der Nachrichten realisiert werden.

Die Anwendung aus Abbildung 4.2 wird in den folgenden Kapiteln als laufendes Beispiel verwendet. Die vorgestellten Konzepte werden damit an einer durchgängigen Beispielanwendung erläutert, wodurch die Evolution der Anwendung im Verlauf der LAUFFEUER-Methode verdeutlicht wird.

Kapitel 5

Verfeinern und Erkennen von Mustern in Deployment-Modellen

In diesem Kapitel wird eine Methode präsentiert, die es ermöglicht, musterbasierte Deployment-Modelle automatisiert zu ausführbaren Deployment-Modellen zu verfeinern sowie Muster in ausführbaren Deployment-Modellen zu erkennen. Sie realisiert damit die Schritte eins und sechs der Lauffeuer-Methode. In Abschnitt 5.1 wird die Konzeptidee präsentiert, Abschnitt 5.2 führt ein Metamodell für *Verfeinerungsmodelle* ein, während die Abschnitte 5.4 und 5.3 Herausforderungen und Beispiele präsentieren. Anschließend werden in den Abschnitten 5.5 und 5.6 Algorithmen vorgestellt. Abschnitt 5.7 zeigt eine Beispielverfeinerung und Abschnitt 5.8 diskutiert die Methode.

Die automatisierte Verfeinerung von Mustern wurde bereits in [HBF+18; HBF+20] veröffentlicht, während die Mustererkennung in [HBF+21] präsentiert wurde. Das Konzept wurde darüber hinaus in anderen Ansätzen wiederverwendet, um z. B. Quanten-Anwendungen zu modellieren [WBH+20a] oder unterschiedliche Hosting-Möglichkeiten zu untersuchen [YBB+22].

5.1 Konzeptidee und -übersicht

Musterbasierte Deployment-Modelle können nicht zur automatisierten Bereitstellung der modellierten Anwendung verwendet werden, da sie die Zusammensetzung der Komponenten und Konfigurationen einer Anwendung abstrakt beschreiben. Daher müssen musterbasierten Deployment-Modelle zunächst zu einem konkreten und ausführbaren Deployment-Modell verfeinert werden. Dafür werden die sogenannten *Verfeinerungsmodelle* (engl. „*Pattern Refinement Models*", PRMs) eingeführt, die die Abbildung der Muster auf konkrete Anbieter, Komponenten und Konfigurationen ermöglichen. Mithilfe dieser PRMs können musterbasierte Deployment-Modelle automatisiert in ausführbare Deployment-Modelle transformiert werden. Ein PRM definiert dabei jeweils die Abbildung eines musterbasierten Deployment-Modellfragments auf ein Fragment eines ausführbaren Deployment-Modells.

Wie in Abbildung 5.1 dargestellt, ist die Verfeinerung bzw. Erkennung von Mustern ein iterativer Prozess mit je zwei Schritten. In jeder Iteration wird dabei in Schritt (A) ein *PRM-Repository* automatisiert nach PRMs durchsucht, die die Muster in *einem musterbasierten Deployment-Modell* (MDM) zu konkreten Technologien verfeinern können. Eines der gefunden PRMs kann dann entweder (i) automatisiert oder (ii) manuell ausgewählt werden. Mithilfe des ausgewählten PRMs wird in Schritt (B) das gefundene Fragment im musterbasierten Deployment-Modell ersetzt. Dieser Prozess wiederholt sich so lange, bis das MDM entweder keine Muster mehr enthält, oder kein PRM mehr gefunden werden kann, das das MDM weiter konkretisieren kann.

Um Muster in ausführbaren Deployment-Modellen zu erkennen, wird der Prozess umgekehrt. Anstatt nach Mustern zu suchen, die von PRMs zu konkreten Technologien verfeinert werden können, werden die PRMs dazu genutzt, um Technologien wieder auf die von ihnen umgesetzten Mustern zu abstrahieren. Der Prozess selbst ändert sich dabei nicht. Allerdings können nicht alle PRMs auch für die Erkennung von Mustern verwendet werden. Zum Beispiel kann anhand von annotierten UNPREDICTABLE WORKLOAD- und STATELESS COMPONENT-Muster abgeleitet werden, dass eine Komponente skaliert werden muss, jedoch kann aufgrund einer automatischen Skalierung

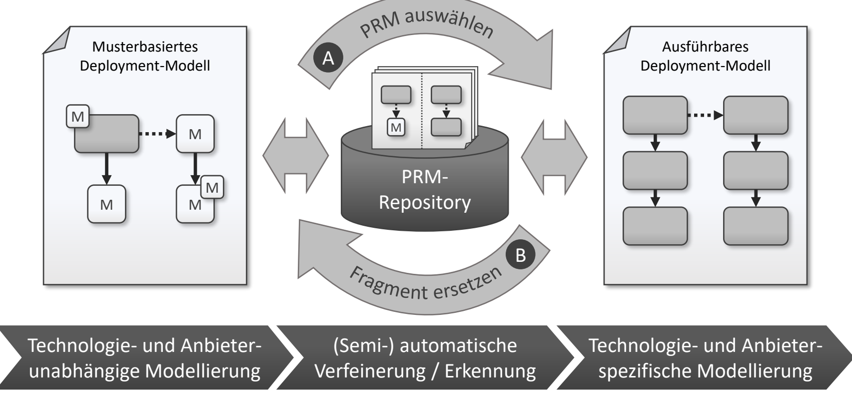


Abbildung 5.1: Prozess zur Verfeinerung und Erkennung von Mustern

nicht auf einen Unpredictable Workload geschlossen werden. Daher wird im Folgenden zwischen *Verfeinerungsmodellen* (PRMs) und *Erkennungsmodellen* (engl. „*Pattern Detection Models*“, PDMs) unterschieden. Da Muster, die in einem ausführbaren Deployment-Modell erkannt werden können, auch im musterbasierten Deployment-Modell einer Anwendung verwendet werden können, können diese auch für die Verfeinerung verwendet werden. Daher wird die Annahme getroffen, dass PDMs immer auch PRMs sind.

Im Allgemeinen bestehen Verfeinerungs- und Erkennungsmodelle aus einer *Musterstruktur*, einer *Lösungsstruktur*[1] sowie mehreren unterschiedlichen *Zuordnungen*. Die Musterstruktur definiert dabei ein Fragment eines musterbasierten Deployment-Modells, während die Lösungsstruktur eine entsprechende konkrete Umsetzung dieser Muster als Fragment eines ausführbaren Deployment-Modells spezifiziert. Um herauszufinden, ob ein PRM auf ein Deployment-Modell angewendet werden kann, müssen Subgraphen, die der Musterstruktur bei der Verfeinerung bzw. der Lösungsstruktur bei der Erkennung entsprechen, in dem aktuell untersuchten Deployment-Modell enthalten sein. Mithilfe der Zuordnungen können die gefundenen Strukturen anschließend in die jeweils andere transformiert werden (s. Abschnitt 5.5).

[1]Die Musterstruktur wurde ursprünglich als „Detector“ bzw. „Abstraction Structure“ und die Lösungsstruktur als „Refinement Structure“ bzw. „Technical Detector“ eingeführt.

5.2 Metamodell für Verfeinerungsmodelle

Um musterbasierte Deployment-Modelle in ausführbare Deployment-Modelle transformieren zu können, wird im Folgenden Abschnitt ein Metamodell für Verfeinerungs- und Erkennungsmodelle definiert.

Definition 5.1 (Verfeinerungsmodell – infromell). Ein Verfeinerungsmodell definiert Regeln zur Verfeinerung eines musterbasierten Deployment-Modellfragments in ein Fragment eines ausführbaren Deployment-Modells.

Definition 5.2 (Erkennungsmodell (PDM) – informell). Ein Erkennungsmodell ist ein Verfeinerungsmodell, das sowohl zur Verfeinerung von musterbasierten Deployment-Modellen als auch zur Erkennung von Mustern in ausführbaren Deployment-Modellen verwendet werden kann.

Definition 5.3 (Verfeinerungs- und Erkennungsmodell (PRM)). Für Verfeinerungsmodelle wird ein Metamodell definiert: Sei $\mathcal{V}$ die Menge aller Verfeinerungsmodelle, dann ist ein Verfeinerungsmodell $v \in \mathcal{V}$ definiert als:

$$v = (MusterS_v, LösungsS_v, ErkennungsM_v,$$
$$VZ_v, AZ_v, EZ_v, RZ_v, BZ_v)$$

Die einzelnen Elemente des Tupels sind dabei wie folgt definiert:

- $MusterS_v \in \mathcal{T}$ ist die *Musterstruktur* des PRMs v und ein musterbasiertes Deployment-Modell, welches eine Topologie aus Mustern und Komponenten beschreibt. Die in der Musterstruktur enthaltenen Muster können mithilfe dieses PRMs v zu den konkreten Komponenten der *Lösungsstruktur* verfeinert bzw. erkannt werden.
- $LösungsS_v \in \mathcal{T}$ stellt die *Lösungsstruktur* des PRMs v dar und ein Deployment-Modell, das die Muster in der Musterstruktur von v mit konkreten Komponenten und deren entsprechenden Konfigurationen sowie Abhängigkeiten umsetzt.

- $ErkennungsM_v \in \mathcal{B}$ definiert einen Wahrheitswert der angibt, ob das Verfeinerungsmodell v als *Erkennungsmodell* eingesetzt werden kann.
- $VZ_v \subseteq VM_{MusterS_v} \times ME_{LösungsS_v} \times \Sigma^+$ ist die Menge der *Verhaltenszuordnungen* in v. Eine Verhaltenszuordnung $vz \in VZ_v$ beschriebt, wie sich ein Verhaltensmuster $MusterE \in VM_{MusterS_v}$ auf eine bestimmte Eigenschaft $Eigenschaft \in \Sigma^+$ eines Modellelements aus der Lösungsstruktur $LösungsE \in ME_{LösungsS_v}$ auswirkt.

  $$vz = (MusterE, LösungsE, Eigenschaft)$$

  Jede Eigenschaft einer Verhaltenszuordnung muss dabei auch eine Eigenschaft des Modellelements aus der Lösungsstruktur sein:

  $$\forall (vm, le, e) \in VZ\ \exists e_{le} \in eingeschaften(le) : \pi_{Schlüssel}(e_{le}) = e$$

- $AZ_v \subseteq MK_{MusterS_v} \times MK_{LösungsS_v} \times AT_{MusterS_v}$ ist die Menge aller in v definierten *Artefaktzuordnungen*. Eine Artefaktzuordnung $az \in AZ_v$ gibt an, dass Artefakte des Typs $Artefakttyp \in AT_{MusterS_v}$ von dem Modellknoten der Musterstruktur $MusterE \in MK_{MusterS_v}$ zu dem Modellknoten in der Lösungsstruktur $LösungsE \in MK_{LösungsS}$ verschoben werden:

  $$az = (MusterE, LösungsE, Artefakttyp)$$

- $EZ_v \subseteq ME_{MusterS_v} \times \Sigma^+ \times ME_{LösungsS_v} \times \Sigma^+$ ist die Menge an *Eigenschaftszuordnungen* in v. Eine Eigenschaftszuordnung $ez \in EZ_v$ beschreibt, dass die Eigenschaft $MusterProp \in eingenschaften(MusterE)$ des Modellelements $MusterE \in ME_{MusterS_v}$ bei der Anwendung des PRMs in die Eigenschaft $MusterProp \in eingenschaften(LösungsE)$ des Modellelements $LösungsE \in ME_{LösungsS}$ kopiert wird.

  $$ez = (MusterE, MusterProp, LösungsE, LösungsProp)$$

- $RZ_v \subseteq MK_{MusterS_v} \times RT_{MusterS_v} \times \mathcal{D} \times MK_{LösungsS_v} \times \{KT_{MusterS_v} \cup KMT_{MusterS_v} \cup \perp\}$ ist die Menge aller *Relationszuordnungen* in v. Eine Relationszuordnung $rz \in RZ_v$ definiert, wohin eine Relation, die nicht Teil des in einem Deployment-Modell gefunden Subgraphen ist, bei der Anwendung des PRMs v umgehängt werden muss:

  $$rz = (MusterE, Relationstyp, Richtung,$$
  $$LösungsE, Endpunkttyp)$$

  Die Elemente einer Relationszuordnung rz sind dabei der Modellknoten der Musterstruktur $MusterE \in MK_{MusterS_v}$, an dem eine Relation von Typ $Relationstyp \in RT_{MusterS_v}$ in der angegebenen Richtung $Richtung \in \mathcal{D} = \{eingehend, ausgehend\}$ zu dem Modellknoten der Lösungsstruktur $LösungsE \in MK_{LösungsS_v}$ umgeleitet werden kann. Dabei kann optional eine Einschränkung definiert werden, sodass die Relation nur umgeleitet werden kann, wenn der Ausgangs- bzw. Zielknoten der Relation ein Knoten des in $Endpunkttyp \in \{KT_{MusterS_v} \cup MT_{MusterS_v} \cup \perp\}$ definierten Knotentyps ist.

- $BZ_v \subseteq MK_{MusterS_v} \times MK_{LösungsS_v}$ ist die Menge aller *Bleibzuordnungen* in v. Eine Bleibzuordnung $bz \in BZ_v$ definiert, dass ein Modellknoten $MusterE \in MK_{MusterS_v}$ aus der Musterstruktur unverändert in der Lösungsstruktur bestehen bleibt. Dabei wird $MusterE$ an der Stelle eingesetzt, an der der Modellknoten $LösungsE \in MK_{LösungsS_v}$ in der Lösungsstruktur positioniert ist. Bleibzuordnungen sind definiert als:

  $$bz = (MusterE, LösungsE)$$

  Dabei gilt die Einschränkung, dass für jeden Modellknoten maximal eine Bleibzuordnung definiert werden kann:

  $$\nexists bz_i, bz_j \in BZ_v : \pi_{MusterE}(bz_i) = \pi_{MusterE}(bz_j) \vee$$
  $$\pi_{LösungsE}(bz_i) = \pi_{LösungsE}(bz_j)$$

5.3 Herausforderungen bei der Verfeinerung und Erkennung

Bei der Verfeinerung von musterbasierten Deployment-Modellen zu ausführbaren Deployment-Modellen sowie bei der Erkennung von Mustern in ausführbaren Deployment-Modellen, werden einzelne Fragment der graphbasierten Modelle nacheinander transformiert (s. Abschnitt 5.1). Durch diesen Ansatz entstehen einige Herausforderungen, die gelöst werden müssen. Zum Beispiel kann ein Modellknoten nicht immer mit genau einem anderen Modellknoten ausgetauscht werden, sondern muss zum Teil durch mehrere andere Modellknoten ersetzt werden. Um zu verhindern, dass Relationen, die von außen in das zu transformierenden Fragment ein- bzw. ausgehen, verloren gehen, müssen diese Relationen zu den neu hinzugefügten Modellknoten umgehängt werden. Solche Relationen werden als *externe Relationen* bezeichnet. Zusätzlich kann es erforderlich sein, dass externe Relationen bei einer Transformation unterschiedlich behandelt werden müssen. Zum Beispiel können unterschiedliche Transformationsregeln für externe Relationen gelten. In Abbildung 5.2 sind externe Relationen durch Kreise markiert und werden im Verlauf dieses Abschnitts im Detail erläutert.

Definition 5.4 (Externe Relation – informell). Eine externe Relation ist eine Relation, die (i) Teil eines Deployment-Modells ist, worin Muster verfeinert oder erkannt werden sollen. Eine externe Relation geht (ii) an einem Modellknoten des Fragments des Deployment-Modells ein oder aus, das verfeinert bzw. erkannt werden soll. Die Relation ist (iii) nicht Teil dieses Fragments.

Die folgende Liste fasst die Herausforderungen zusammen, die für die Transformation von musterbasierten Deployment-Modellen zu ausführbaren Deployment Modellen sowie deren Rückweg gelöst werden müssen:

1. Das Fragment des Ausgangsgraphen, das transformiert werden soll, muss zunächst ausgewählt und identifiziert werden.
2. Der Graph, der das Fragment ersetzen soll, muss dem Ausgangsgraphen hinzugefügt werden, während das identifizierte Fragment ent-

fernt werden muss. Dabei dürfen keine Abhängigkeiten sowie weitere Informationen während der Transformation verloren gehen.

3. Alle externen Relationen müssen zu den neu hinzugefügten Modellknoten umgehängt werden. Für verschiedene Relationen können unterschiedliche Transformationsregeln gelten.
4. Artefakte, die an Modellknoten des zu transformierenden Fragments angehängt sind, dürfen nicht verloren gehen. Sie müssen ebenfalls an die neu hinzugefügten Modellknoten verschoben werden.
5. Um das Verhalten einer konkreten Komponente, wie einer Web-App, beschreiben zu können, werden Verhaltensmuster an der Komponente annotiert. Bei der Verfeinerung müssen diese Verhaltensmuster von einem PRM identifiziert werden, wobei die annotierte Komponente bei einer Transformation nicht ersetzt werden darf.
6. Verhaltensmuster können sich auf bestimmte Eigenschaften einer oder mehrere Komponenten auswirken. Es gibt jedoch auch Verhaltensmuster, die durch die Anwesenheit eines bestimmten Komponententyps umgesetzt werden. Beide Varianten müssen unterstützt werden.
7. Eigenschaften dürfen bei einer Transformation nicht verloren gehen.

Im Folgenden werden die Herausforderungen näher betrachtet und anhand von den in Abbildungen 5.2 und 5.3 dargestellten Beispielen erläutert. Im linken oberen Teil der Abbildung 5.2 ist ein musterbasiertes Deployment-Modell dargestellt, das verfeinert werden soll. Es besteht aus insgesamt sieben Modellknoten unterschiedlichen Typs, welche verschiedene Abhängigkeiten untereinander beschreiben. Die Abhängigkeiten sind durch die abgebildeten Relationen dargestellt. Während der Modellknoten D darüber hinaus der Eigenschaft P1 den Wert „iaas“ zuweist, ist an Modellknoten B ein Artefakt vom Artefakttyp x annotiert. Des Weiteren ist auf der rechten Seite in Abbildung 5.2 ein PRM dargestellt. Es beschreibt, wie eine Musterstruktur bestehend aus Modellknoten der Typen B und D zu der dargestellten Lösungsstruktur verfeinert werden kann. Der Modellknoten des Typs B, MS1, hat dabei eine Abhängigkeit von Modellknoten MS2 des Typs D.

Damit das musterbasierte Deployment-Modell mit dem gezeigten PRM verfeinert werden kann, muss im ersten Schritt ein Subgraph in dem MDM identifiziert werden, der der Musterstruktur des PRMs entspricht. Dabei müssen die Typen der Modellelemente übereinstimmen sowie deren Eigenschaften[1]. Beinhaltet ein MDM einen solchen Subgraphen, wird für jedes Modellelement des Subgraphen eine Zuordnung auf das entsprechende Modellelement der Musterstruktur definiert: die sogenannten *Strukturelementzuordnungen*. Die Strukturelementzuordnungen sind in Abbildung 5.2 durch blau gestrichelte Pfeile zwischen der Musterstruktur und dem musterbasierten Deployment-Modell dargestellt. Dabei entsprechen die Knoten B und D des musterbasierten Deployment-Modells den Knoten MS1 und MS2 der Musterstruktur. Dementsprechend wird auch eine Strukturelementzuordnung für die Relation definiert. Somit ist sichergestellt, dass bei der Anwendung des PRMs, d. h. bei der Transformation des Subgraphen, immer auf die betreffenden Modellelemente zugegriffen werden kann.

Definition 5.5 (Strukturelementzuordnung – informell). Eine Strukturelementzuordnung beschreibt die Zuordnung eines Modellelements eines (musterbasierten) Deployment-Modells auf ein Modellelement einer Muster- bzw. Lösungsstruktur eines PRMs. Dabei entsprechen die Modellelemente des Deployment-Modells jeweils einem Modellelement (i) der Musterstruktur bei der Verfeinerung bzw. (ii) der Lösungsstruktur bei der Mustererkennung.

Um den identifizieren Subgraphen in dem musterbasierten Deployment-Modell zu der im PRM definierten Lösungsstruktur zu verfeinern, wird die Lösungsstruktur zunächst zu dem Deployment-Modell hinzugefügt. Anschließend müssen die externen Relationen zu den neu hinzugefügten Modellknoten umgehängt werden. Der zu verfeinernde Subgraph hat in diesem Fall fünf externe Relationen. Diese sind in Abbildung 5.2 durch blaue Kreise gekennzeichnet und mit den Namen a bis e gekennzeichnet. Wie diese externen Relationen umgehängt werden müssen, kann jedoch nicht ohne weitere

[1] Die Details zur Anwendbarkeitsprüfung eines PRMs auf ein Deployment-Modell und zur Anwendung eines Verfeinerungsmodells sind in den Abschnitten 5.4.2 und 5.5 beschrieben.

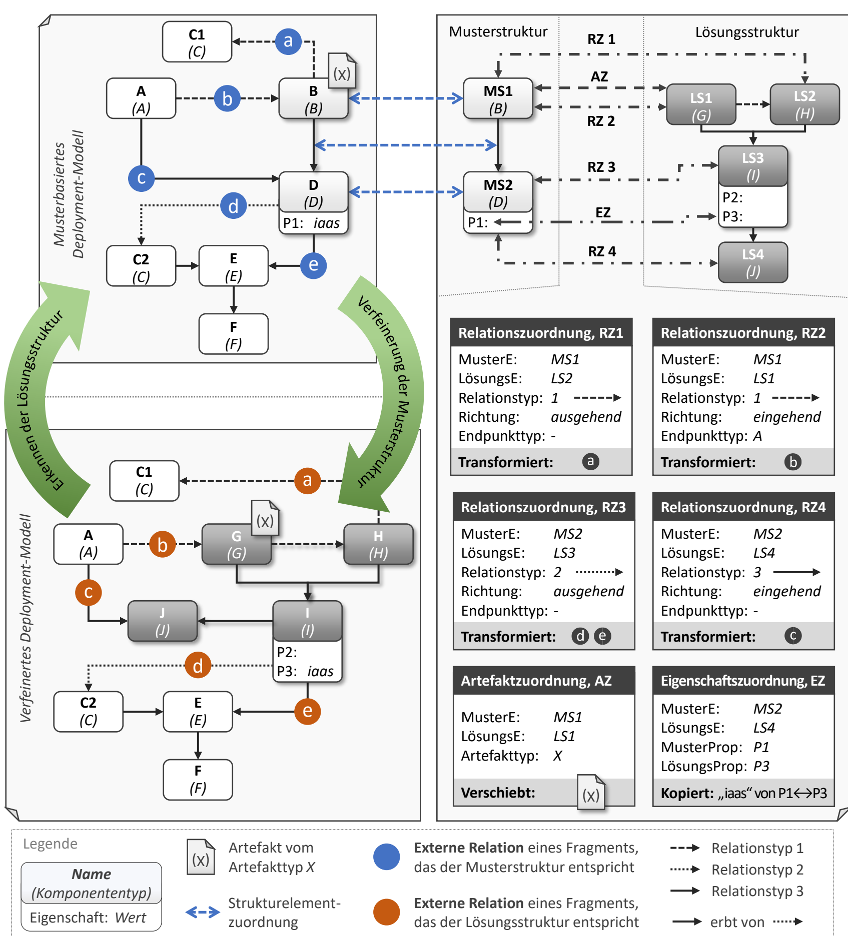


Abbildung 5.2: Transformation eines musterbasierten Deployment-Modells ohne Verhaltensmuster. Durch das Anwenden des dargestellten PRMs kann der obere Graph in den unteren transformiert werden und umgekehrt.

Informationen entschieden werden. Es kann beispielsweise nicht automatisiert entschieden werden, ob die externe Relation (a), die eine Abhängigkeit von B zu C1 beschreibt, an den Knoten LS1, LS2, LS3 oder LS4 aus der Lösungsstruktur umgehängt werden soll. Daher werden *Relationszuordnungen* eingeführt. Relationszuordnungen definieren Transformationsregeln, um externe Relation bei der Anwendung eines PRMs umhängen zu können.

Definition 5.6 (Relationszuordnung – informell). Eine Relationszuordnung definiert, wie eine externe Relation während der Anwendung eines PRMs auf ein Deployment-Modell von einem durch die Transformation entfernten Modellknoten zu einem neu hinzugefügten Modellknoten umzuhängen ist.

Insgesamt sind in dem PRM in Abbildung 5.2 vier Relationszuordnungen definiert. Jede Relationszuordnung spezifiziert, wie eine externe Relation, die an dem entsprechenden Modellknoten ein- oder ausgeht und dem angegebenen Relationstyp entspricht, umgehängt werden kann. Die Relationszuordnung RZ1 gibt beispielsweise an, dass Relationen die an einem MS1 entsprechenden Modellknoten ausgehen und den Relationstyp 1 haben zu LS2 aus der Lösungsstruktur umgehängt werden müssen. Somit kann durch das Anwenden der Relationszuordnung RZ1 die externe Relation (a) umgehängt werden. Wie im verfeinerten Deployment-Modell dargestellt, geht die Relation (a) anschließend von dem eingefügten Knoten H aus. Analog dazu definieren die beiden Relationszuordnungen RZ2 und RZ4, wie die externen Relationen (b) und (c) umgehängt werden können. Während die Relation (b) nach der Transformation am Knoten G eingeht, wird (c) zu J umgeleitet.

Die Relationszuordnung RZ 3 definiert, wie externe Relationen des Relationstyps 2, die am Modellknoten MS2 ausgehen, umgehängt werden können. Sie werden zu dem Modellknoten LS4 in der Lösungsstruktur umgehängt. Eine Besonderheit ist in diesem Fall, dass der Relationstyp 3 von Relationstyp 2 erbt. Somit kann während der Transformation des Deployment-Modells sowohl die Relation (d) als auch die Relation (e) umgehängt werden.

Im nächsten Schritt muss sichergestellt werden, dass keine Artefakte, die an den Knoten B und D annotiert sind, verloren gehen. Dafür werden *Arte-*

faktzuordnungen eingeführt. Eine Artefaktzuordnung definiert, wie Artefakte eines bestimmten Artefakttyps von einem Modellknoten der Musterstruktur zu einem Modellknoten der Lösungsstruktur verschoben werden können.

Definition 5.7 (Artefaktzuordnung – informell). Eine Artefaktzuordnung definiert, wie ein Artefakt eines bestimmten Typs während einer Transformation von einem durch die Transformation entfernten Modellknoten zu einem hinzugefügten Modellknoten umzuhängen ist.

In dem PRM in Abbildung 5.2 ist eine Artefaktzuordnung für Artefakte des Artefakttyps x definiert. Die Artefakte müssen dabei an dem Modellknoten, der MS1 entspricht, annotiert sein und werden während der Transformation zu dem Modellknoten LS1 der Lösungsstruktur verschoben. Im verfeinerten Deployment-Modell in Abbildung 5.2 ist dies durch den Knoten G dargestellt.

Schließlich können *Eigenschaftszuordnungen* dafür verwendet werden, um bereits gesetzte Eigenschaftswerte während der Verfeinerung nicht zu verlieren. In Abbildung 5.2 definiert der Modellknoten D im Deployment-Modell beispielsweise, dass die Eigenschaft P1 den Wert „iaas“ hat. Durch die Eigenschaftszuordnung EZ wird definiert, dass Werte, die für die Eigenschaft P1 im Knoten MS2 angegeben wurden, während der Verfeinerung in den Wert der Eigenschaft P3 des Knotens LS3 kopiert werden. Daher wird der Wert „iaas“ in die Eigenschaft P3 des Knotens I kopiert.

Definition 5.8 (Eigenschaftszuordnung – informell). Eine Eigenschaftszuordnung beschreibt, wie eine Eigenschaft eines Strukturelements der Musterstruktur in eine Eigenschaft der Lösungsstruktur umgesetzt werden kann.

Durch das Einführen der Relations-, Artefakt- und Eigenschaftszuordnungen ist es möglich, die Verfeinerung von Komponentenmuster zu realisieren. Sie ermöglichen es, die Transformationsregeln für externe Relationen, annotierte Artefakte und Eigenschaftswerte eindeutig zu definieren.

Wie bereits beschrieben, kann das in Abbildung 5.2 dargestellte musterbasierte Deployment-Modell somit durch das abgebildete PRM verfeinert werden. Dabei entsteht das verfeinerte Deployment-Modell, das links unten

dargestellt ist. Anstelle der Modellknoten B und D enthält das Deployment-Modell nun die Modellknoten G, H, I und J. Entsprechend der Relationszuordnungen wurden alle fünf externen Relationen umgehängt. Die Relation ⓐ beschreibt nun eine Verbindung zwischen den Modellknoten H und C1, während ⓑ eine Abhängigkeit von A nach G definiert und ⓒ A mit J verbindet. Schließlich gehen sowohl ⓓ als auch ⓔ von dem Modellknoten I aus und spezifizieren Abhängigkeiten von I zu C und E. Auch das Artefakt und der Wert der Eigenschaft P1 wurden entsprechend der im PRM definierten Zuordnungen an die angegebenen Modellelemente verschoben.

Ebenso wie die Verfeinerung der Musterstruktur, kann das in Abbildung 5.2 dargestellte PRM auch für die Erkennung der Lösungsstruktur verwendet werden. Anstatt einen Subgraphen, der der Musterstruktur des PRMs entspricht, in dem Deployment-Modell zu suchen, wird bei der Erkennung ein Subgraph gesucht, der der Lösungsstruktur eines PRMs entspricht. Somit kann das PRM sowohl Muster im musterbasierten Deployment-Modell verfeinern, als auch die Muster im verfeinerten Deployment-Modell erkennen. Dies ist durch die grünen Pfeile dargestellt.

In Abbildung 5.3 sind weitere Beispiele abgebildet. Sie beschreiben Herausforderungen, die bei der Verfeinerung bzw. Erkennung von Verhaltensmuster auftreten. Das musterbasierte Deployment-Modell 1, das oben in der Mitte in Abbildung 5.3 dargestellt ist, besteht aus drei Modellknoten, K1, L1 und M1. K1 ist mit zwei unterschiedlichen Verhaltensmustern der Typen u und v annotiert und soll während der Verfeinerung nicht ausgetauscht werden. Dies ist beispielsweise nötig, um das Verhalten einer konkreten Komponente, die mit Verhaltensmuster annotiert ist, wie die Java App in Abbildung 4.2, umsetzen zu können. Der Modellknoten L1 soll hingegen verfeinert werden.

Auf der rechten Seite von Abbildung 5.3 ist ein PRM dargestellt. Dieses kann einen Modellknoten des Typs K, der mit zwei Verhaltensmuster der Typen u und v annotiert ist und von einem Modellknoten des Typs L abhängig ist, verfeinern. Dabei wirkt sich das Verhaltensmuster des Typs v auf den Wert der Eigenschaften P4 und P5 des Modellknotens LS6 aus. Dies ist durch die dargestellten *Verhaltenszuordnungen* definiert. Verhaltenszuordnungen beschreiben, wie sich ein Verhaltensmuster in der Musterstruktur durch seine

Verfeinerung auf eine Konfiguration, d. h. Eigenschaft, eines Modellelements in der Lösungsstruktur auswirkt. Wie in Abbildung 5.3 dargestellt, können mehrere Verhaltenszuordnungen für ein Verhaltensmuster definiert sein. In diesem Fall wirkt sich das Verhaltensmuster immer auf alle Eigenschaften aus. Das bedeutet, wird ein Subgraph in einem Deployment-Modell identifiziert, der der im PRM dargestellten Musterstruktur entspricht, werden die Werte der Eigenschaften P4 und P5 auf „v1" und „v2" gesetzt. Dieser Fall ist im verfeinerten Deployment-Modell 1 dargestellt, welches das Ergebnis der Verfeinerung des musterbasierten Deployment-Modells 1 zeigt.

Definition 5.9 (Verhaltenszuordnung – informell). Eine Verhaltenszuordnung definiert, wie sich ein Verhaltensmuster an einem Komponentenmuster, einer Komponente oder einer Relation der Musterstruktur auf eine Eigenschaft einer Komponente oder Relation in der Lösungsstruktur auswirkt.

Zusätzlich zu Verhaltensmuster, die sich direkt auf Eigenschaften auswirken, kann es Verhaltensmuster geben, die durch die Anwesenheit einer oder mehrerer Komponenten umgesetzt werden. Für solche Verhaltensmuster werden keine Verhaltenszuordnungen definiert. Ein solches Verhaltensmuster stellt das Verhaltensmuster des Typs u in Abbildung 5.3 dar.

Existieren Verhaltenszuordnungen für ein Verhaltensmuster in einem PRM, kann das PRM auch für die Verfeinerung eines musterbasierten Deployment-Modells verwendet werden, wenn das Verhaltensmuster nicht an dem entsprechenden Modellknoten annotiert ist. Das Verhaltensmuster ist somit optional. Diese Variante ist in dem musterbasierten Deployment-Modell 2 in Abbildung 5.3 dargestellt. Der einzige Unterschied zu dem musterbasierten Deployment-Model 1 ist, dass kein Verhaltensmuster des Typs u an dem Modellknoten K2 annotiert ist. Durch die Verhaltenszuordnungen, die für das Verhaltensmuster des Typs u im PRM definiert sind, kann das PRM dennoch für die Verfeinerung des musterbasierten Deployment-Modells 2 verwendet werden. Das Verhaltensmuster des Typs v muss jedoch, wie dargestellt, vorhanden sein. Ansonsten würde möglicherweise ein Verhalten umgesetzt werden, das nicht gewollt ist oder sogar vermieden werden muss.

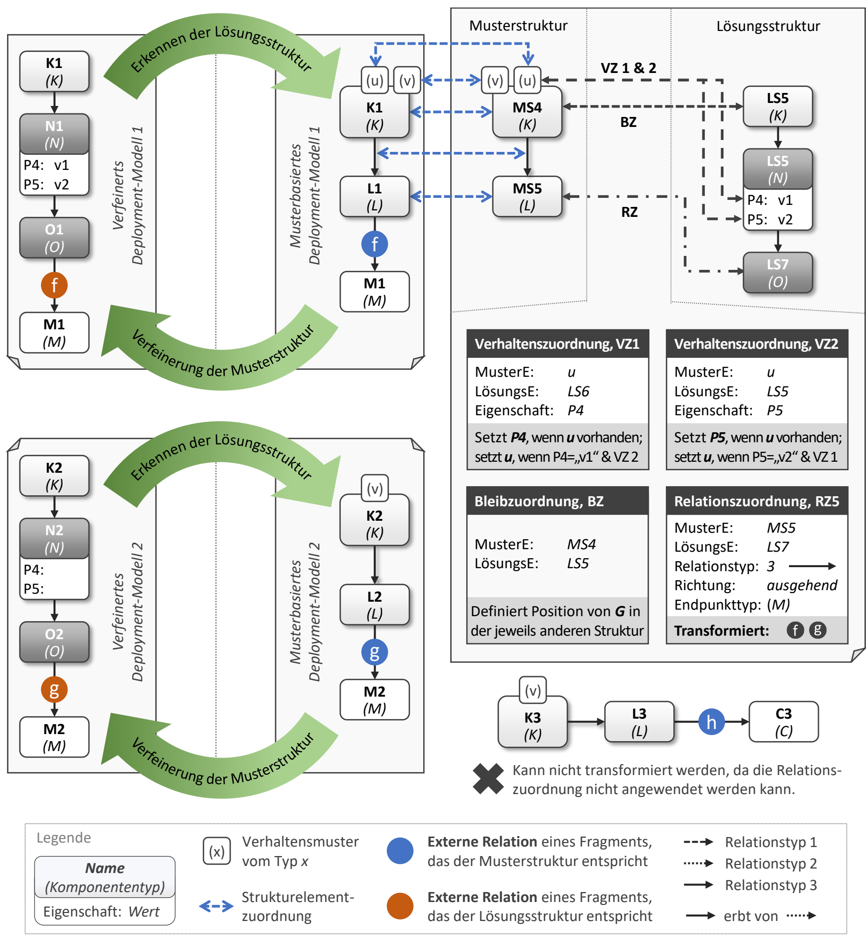


Abbildung 5.3: Transformation eines musterbasierten Deployment-Modells mit Verhaltensmuster. Durch das Anwenden des dargestellten PRMs können die dargestellten Graphen in den jeweils ihnen gegenüberliegenden Graphen transformiert werden.

Wie in Abbildung 4.2 dargestellt, können Verhaltensmuster an konkrete Komponenten annotiert sein. Diese dürfen jedoch während der Verfeinerung nicht durch andere Komponenten ersetzt werden, sondern müssen im Modell verbleiben. Um jedoch Verhaltensmuster an solchen konkreten Komponenten verfeinern zu können, müssen diese von den Modellknoten unterschieden werden, die ersetzt werden sollen. Dafür werden die *Bleibzuordnungen* eingeführt. Diese definieren, welche Modellknoten bei der Verfeinerung nicht ersetzt werden. Darüber hinaus kann über die Bleibzuordnungen abgeleitet werden, welche Relationen ein verbleibender Modellknoten zu den neu hinzugefügten Modellknoten hat. Wie in dem PRM in Abbildung 5.3 dargestellt, hat der Modellknoten MS4 eine Bleibzuordnung zu dem Modellknoten LS5 der Lösungsstruktur. Dadurch ist definiert, dass ein Modellknoten, der MS4 entspricht, bei der Verfeinerung an der Stelle des Knotens von LS5 in dem Deployment-Modell verbleibt. Dies ist auch in den verfeinerten Deployment-Modellen 1 und 2 dargestellt: Die Modellknoten K1 und K2 verbleiben im Deployment-Modell nach der Verfeinerung mit einer Abhängigkeit zu den Modellknoten N1 bzw. N2. Dies gilt bei der Mustererkennung entsprechend.

Definition 5.10 (Bleibzuordnung – informell). Eine Bleibzuordnung definiert, dass ein Modellknoten bei der Verfeinerung oder Erkennung eines Deployment-Modells im Modell verbleibt und nicht ersetzt wird.

Schließlich definiert das PRM in Abbildung 5.3 eine Relationszuordnung. Durch diese können die externen Relationen (f) und (g) der Deployment-Modelle 1 und 2 umgehängt werden. Das dritte Deployment-Modell, das unter dem PRM in Abbildung 5.3 abgebildet ist, enthält zwar einen der Musterstruktur entsprechenden Subgraphen, allerdings kann die Relationszuordnung nicht angewendet werden. Die Relationszuordnung definiert durch die Angabe eines zulässigen *Endpunkttyps*, dass lediglich externe Relationen umgehängt werden können, die einen Modellknoten des Typs M zum Ziel haben. Weil die externe Relation (h) jedoch an einem Modellknoten des Typs C eingeht, kann die Relationszuordnung und somit das gesamte PRM nicht auf das dritte Deployment-Modell angewendet werden.

5.4 Verfeinerungs- und Erkennungsmodelle

In den folgenden Abschnitten werden zunächst konkrete Verfeinerungsmodelle beschrieben, bevor Algorithmen für die automatisierte Anwendung von PRMs auf (musterbasierte) Deployment-Modelle erläutert werden.

5.4.1 Beispiele für Verfeinerungs- und Erkennungsmodelle

Verfeinerungsmodelle können, wie in Abbildungen 5.4 und 5.5 gezeigt, grafisch dargestellt werden. Abbildung 5.4 beschreibt, wie ein Message Channel, der von einer Message-oriented Middleware in einer Public Cloud bereitgestellt wird, mit AWS-Angeboten umgesetzt werden kann: Eine *SQS Queue*, die auf dem *Simple Queue Service* Angebot von *AWS* erstellt wird. Abbildung 5.5 hingegen zeigt, wie eine *Java 11 App*, welche mit einem Static Workload annotiert ist, auf einem Execution Environment in einer Private Cloud bereitgestellt werden kann – zum Beispiel auf einer *Ubuntu 22.04* VM, welche in einer *OpenStack*-Umgebung bereitgestellt wird.

Zwischen den Muster- und Lösungsstrukturen beider PRMs sind die für die Transformation benötigten Zuordnungen definiert. In Abbildung 5.4 ist beispielsweise dargestellt, wie sich Verhaltensmuster, welche an einer Komponente oder einem Muster annotiert sind, auf konkrete Eigenschaften auswirken können. Das Exactly-once Delivery-Muster, das an dem Message Channel annotiert ist und sicherstellen soll, dass jede Nachricht genau einmal zugestellt wird, kann von einer SQS Queue vom Typ *FIFO* umgesetzt werden. Ähnlich dazu kann das Information Obscurity-Muster durch eine serverseitige Verschlüsselung der Nachrichteninhalte umgesetzt werden. Daher sind beide Verhaltensmuster durch eine Verhaltenszuordnung auf die jeweilige Eigenschaft der SQS Queue in der Lösungsstruktur dargestellt.

Des Weiteren sind zwei Relationszuordnungen in Abbildung 5.4 dargestellt. Gehen beispielsweise bei dem Message Channel externe Relationen vom Typ *connectsTo* ein, so müssen diese nach dem Umbau der Topologie an der SQS Queue eingehen. Externe Relationen vom Typ *hostedOn*, die an der Public Cloud eingehen, werden zu dem Knoten vom Typ *AWS* umgeleitet.

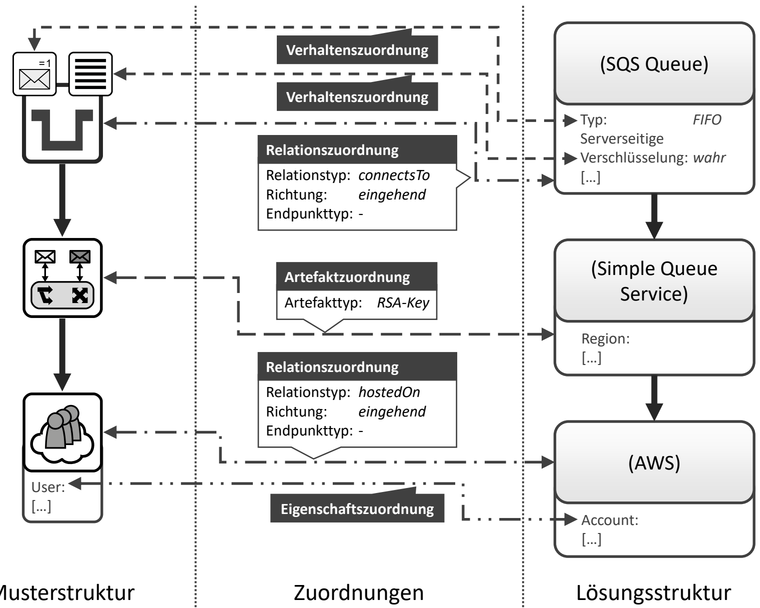


Abbildung 5.4: PRM zur Verfeinerung und Erkennung eines MESSAGE CHANNELS, welcher auf einer PUBLIC CLOUD bereitstellt wird

Zur Nachrichtenverschlüsselung durch die MESSAGE-ORIENTED MIDDLEWARE, kann ein *RSA-Key* als Artefakt an das Komponentenmuster in einem Deployment-Modell angehängt werden. Um dieses Artefakt während der Verfeinerung nicht zu verlieren, ist in Abbildung 5.4 eine Artefaktzuordnung zwischen der MESSAGE-ORIENTED MIDDLEWARE und dem *Simple Queue Service* definiert. Es gibt an, dass Artefakte des Typs *RSA-Key*, die an einer MESSAGE-ORIENTED MIDDLEWARE angehängt wurden, während der Verfeinerung an die Simple Queue Service-Komponente umgehängt werden.

Schließlich ist in Abbildung 5.4 eine Eigenschaftszuordnung dargestellt. Diese beschreibt, dass der Wert, der für den *User* der PUBLIC CLOUD definiert ist, auf den *Account* der AWS Komponente kopiert werden soll.

In Abbildung 5.5 ist ein weiteres PRM dargestellt. Analog zu den Relationszuordnungen in Abbildung 5.4 sind Relationszuordnungen in Abbil-

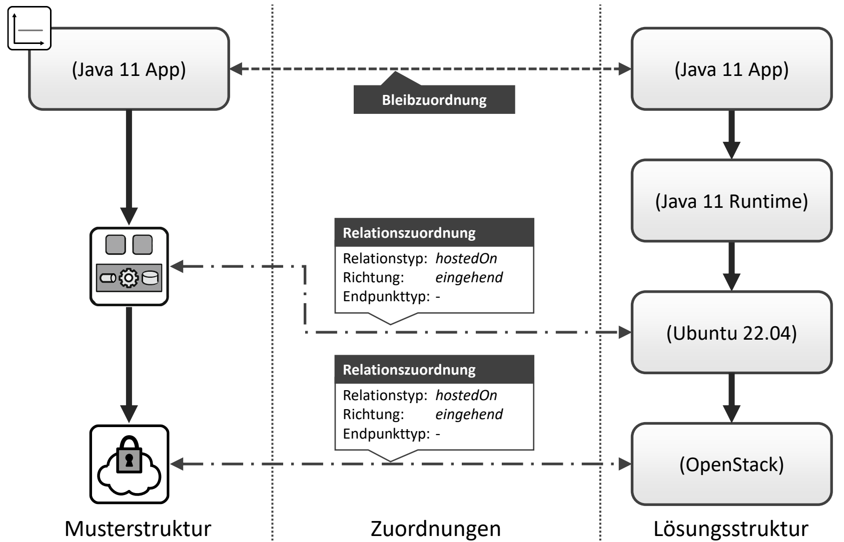


Abbildung 5.5: PRM zur Verfeinerung einer Java 11 App auf einem EXECUTION ENVIRONMENT in einer PRIVATE CLOUD

dung 5.5 definiert: Eingehende Relationen vom Typ *hostedOn* am EXECUTION ENVIRONMENT werden bei einer Transformation zu der Ubuntu VM umgehängt, während eingehende hostedOn-Relationen an der Private Cloud zu OpenStack umgeleitet werden. Zusätzlich ist in Abbildung 5.5 eine Bleibzuordnung dargestellt. Sie definiert, dass ein Strukturelement, das Teil des identifizierten Subgraphen ist, während der Transformation nicht aus der Topologie entfernt werden darf. Somit wird die verbleibende Komponente an den durch die Bleibzuordnung definierten Platz in der Lösungsstruktur verschoben. In diesem Fall bleibt die Java 11 App bestehen und soll bei der Verfeinerung auf der *Java 11 Runtime* bereitgestellt werden. Um jedoch das annotierte STATIC WORKLOAD-Muster verfeinern zu können, wird die Java 11 App-Komponente benötigt. Somit werden Bleibzuordnungen immer benötigt, um Verhaltensmuster umzusetzen, die an Logikkomponenten, bzw. generell Komponenten, die im Modell verbleiben sollen, annotiert sind.

5.4.2 Anwendbarkeit von Verfeinerungs- und Erkennungsmodellen

Damit ein PRM für die Verfeinerung bzw. Erkennung auf ein Deployment-Modell angewendet werden kann, muss die jeweilige Struktur des PRMs als Subgraph in dem Deployment-Modell enthalten sein. Zum Beispiel kann die Musterstruktur des in Abbildung 5.4 dargestellten PRMs in dem in Abbildung 4.2 gezeigten musterbasierten Deployment-Modell gefunden werden, das PRM aus Abbildung 5.5 jedoch nicht: Während das MDM in Abbildung 4.2 einen Subgraphen enthält, der exakt der Musterstruktur des PRMs aus Abbildung 5.4 entspricht, beschreibt die Musterstruktur des PRMs aus Abbildung 5.5 eine Java 11 App, die auf einem `Execution Environment` bereitgestellt wird. Da das MDM aus Abbildung 4.2 keinen solchen Subgraphen enthält, kann das PRM aus Abbildung 5.5 nicht angewendet werden.

Um einen Subgraphen, der der Muster- oder Lösungsstruktur eines PRMs entspricht, in einem Deployment-Modell zu identifizieren, müssen nicht nur die Muster- und Komponententypen übereinstimmen, sondern auch die im PRM definierten Eigenschaften und annotierten Verhaltensmuster. Dafür wird ein *Entsprechungsoperator* für zwei Strukturelemente wie folgt definiert:

Definition 5.11 (Entsprechungsoperator). Der Entsprechungsoperator „$\vec{\approx}$“ dient dazu, um festzustellen, ob zwei Strukturelemente $se_v \in SE_{MusterS} \cup SE_{LösungsS}$ und $se_t \in SE_t$ eines PRMs v und eines MDMs t dasselbe Strukturelement beschreiben. Der Entsprechungsoperator ist definiert als:

$$\begin{aligned}
se_v \vec{\approx} se_t :\Leftrightarrow\ & typ(se_v) \in typen(se_t) \wedge \\
& \big(\forall a_x \in annotationen(se_v) : \big(\exists a_y \in annotationen(se_t) : typ(a_x) \in typen(a_y)\big) \\
& \quad \vee \big(\exists vz_z \in VZ_v\ \nexists a_y \in annotationen(se_t) : \\
& \qquad typ(a_x) \in typen(a_y)\ \wedge\ \pi_{MusterE}(vz_z) = a_x\big)\big) \wedge \\
& \big(\forall prop_i \in eigenschaften(se_v)\ \exists prop_j \in eigenschaften(se_t) : \\
& \quad \pi_{Schlüssel}(prop_i) = \pi_{Schlüssel}(prop_j)\ \wedge\ \big(\pi_{Wert}(prop_i) = \pi_{Wert}(prop_j) \\
& \qquad \vee\ \pi_{Wert}(prop_i) = \varnothing\ \vee\ (\exists vz_y \in VZ_v : \\
& \qquad\quad \pi_{MusterE}(vz_y) = se_v\ \wedge\ \pi_{Eigenschaft}(vz_y) = \pi_{Schlüssel}(prop_i))\big)\big)
\end{aligned}$$

Ein Strukturelement, d. h., eine Komponente, Muster oder Relation einer Muster- oder Lösungsstruktur se_v eines PRMs v, stimmt mit einem Strukturelement se_t eines Deployment-Modells t überein, wenn: (i) Der Typ von se_t oder einer seiner Supertypen gleich dem Typ von se_v ist. (ii) Alle Annotationen, die bei se_t definiert sind, entweder auch an se_v annotiert sind oder Verhaltenszuordnungen für Annotationen existieren, die nicht an se_t annotiert sind. Außerdem müssen (iii) alle Eigenschaften, die nicht Teil einer Verhaltenszuordnung sind und in se_v gesetzt sind, mit dem gleichen Wert in se_t gesetzt sein. Dabei gilt die Ausnahme, dass wenn eine Eigenschaft von se_v keinen Wert definiert, jeder beliebige Wert für die entsprechende Eigenschaft in se_t zulässig ist. Wirkt sich ein Verhaltensmuster direkt auf eine Eigenschaft aus, wie z. B. in Abbildung 5.4 dargestellt, kann das PRM auch dazu verwendet werden, Fragmente zu verfeinern, die nicht alle Verhaltensmuster definieren. Dabei werden die Werte aus den Verhaltenszuordnungen nicht übernommen und die Eigenschaft bleibt leer. Analog dazu gilt, dass wenn bei der Mustererkennung eine Eigenschaft in der Lösungsstruktur einen konkreten Wert definiert und dieser Teil einer Verhaltenszuordnung ist, das entsprechende Verhaltensmuster nur dann erkannt wird, wenn für diese Eigenschaft der in der Lösungsstruktur definierte Wert gesetzt ist. Existieren mehrere Verhaltenszuordnungen für ein Verhaltensmuster, so müssen alle Eigenschaften die entsprechenden Werte aufweisen. Für den Fall, dass keine Zuordnung für ein Verhaltensmuster definiert ist, realisiert die Lösungsstruktur dieses Muster direkt, ohne dass dies durch eine besondere Eigenschaft konfiguriert werden kann. In diesem Fall wird bei der Mustererkennung das Verhaltensmuster nur dann erkannt, sofern alle anderen Kriterien des Entsprechungsoperators erfüllt sind.

Mit dem Entsprechungsoperator können Subgraphalgorithmen, wie der *VF2* Algorithmus [CFSV04], erweitert werden, sodass die Muster- oder Lösungsstruktur eines PRMs als Subgraph in einem Deployment-Modell erkannt werden können. Wird in einem Deployment-Modell die Muster- bzw. Lösungsstruktur als Subgraph erkannt, wird für eine mögliche Transformation durch das PRM eine Zuordnung zwischen den sich entsprechenden Komponenten und Relationen benötigt. Diese Zuordnungen heißen im Fol-

genden als *Strukturelementzuordnungen*. Mithilfe dieser Strukturelementzuordnungen wird es möglich, die Muster, Komponenten und Relationen eines Deployment-Modells, die der Musterstruktur eines PRMs entsprechen, durch die Komponenten und Relationen der Lösungsstruktur des PRMs zu ersetzen. Auch die Rückrichtung, d. h., das Ersetzen einer erkannten Lösungsstruktur eines PRMs durch dessen Musterstruktur, ist damit möglich.

Definition 5.12 (Strukturelementzuordnung). Eine Strukturelementzuordnung $sez = (Entsprechung, Suchelement) \in SEZ_{a,b} \subseteq SE_a \times SE_b$ ist ein Tupel zweier Strukturelemente aus unterschiedlichen musterbasierten Deployment-Modellen $a, b \in \mathcal{T}$. Dabei entspricht das $Suchelement \in SE_b$ einem Strukturelement $Entsprechung \in SE_a$ aus dem Deployment-Modell a, d. h. es gilt der Entsprechungsoperator $Suchelement \overrightarrow{\approx} Entsprechung$.

Da ein musterbasiertes Deployment-Modell mehrere Subgraphen enthalten kann, die der Musterstruktur eines PRMs entsprechen, müssen die einzelnen Subgraphen identifizierbar sein. Wurde ein Subgraph in einem musterbasierten Deployment-Modells identifiziert, der der Musterstruktur eines PRMs entspricht, werden, wie oben beschrieben, Strukturelementzuordnungen zwischen dem Subgraphen und der Musterstruktur definiert. Die Menge aller Strukturelementzuordnungen eines Subgraphen auf eine Musterstruktur eines PRMs wird *Graphzuordnung* genannt. Die in den Abbildungen 5.2 und 5.3 dargestellten Strukturelementzuordnungen bilden beispielsweise jeweils eine Graphzuordnung des identifizierten Subgraphen auf die Musterstruktur des jeweiligen PRMs. Durch die Einführung von Graphzuordnungen können einzelne Subgraphen eindeutig identifiziert werden. Graphzuordnungen gelten für die Verfeinerung als auch entsprechend für die Erkennung von Mustern durch die Lösungsstruktur eines PRMs.

Definition 5.13 (Graphzuordnung). Seien $a, b \in \mathcal{T}$ unterschiedliche musterbasierte Deployment-Modelle und sei b ein Subgraph von a. Dann ist eine Graphzuordnung $gz \in GZ_{a,b} \subseteq \wp(SE_a \times SE_b)$ die Menge aller Strukturelementzuordnungen, die ein Vorkommen des Subgraphen b in dem Deployment-Modell a darstellt. Dabei gilt $\forall se \in SE_b\ \exists!(Entsprechung, se) \in gz$.

Wird ein Subgraph in einem Deployment-Modell gefunden, der einer Muster- bzw. Lösungsstruktur entspricht, kann dieser nur dann automatisiert in die jeweils andere Struktur transformiert werden, wenn auch alle externen Relationen des Subgraphen sowie alle Artefakte der Modellknoten umgehängt werden können. Dafür müssen entsprechende Relations- und Artefaktzuordnungen in dem PRM definiert sein. Existieren für alle externen Relationen und Artefakte passende Zuordnungen, d. h., es können alle externen Relationen und Artefakte bei der Transformation umgehängt werden, dann gilt ein PRM als *anwendbar*. Ist ein PRM nicht anwendbar, kann keine automatisierte Transformation erfolgen, da in diesem Fall nicht definiert ist, wie die Artefakte und Relationen bei der Transformation umzuhängen sind.

Um zu überprüfen, ob ein PRM v auf ein musterbasiertes Deployment-Modell t angewendet werden kann, wird es zusammen mit einer konkreten Graphzuordnung $gz \in GZ_{t,\pi_{MusterS}(v)}$, die eine Abbildung der Musterstruktur aus v in t beschreibt, an den Algorithmus 5.1 übergeben. In Zeile 1 wird zunächst über alle Modellknoten des Subgraphen iteriert. Für jeden dieser Modellknoten muss anschließend geprüft werden (Zeilen 2-12), ob Relationen existieren, die in einem Knoten ein- oder ausgehen (Zeile 3). Ist dies der Fall, können Relationen, die Teil des Subgraphen sind, ausgeschlossen werden (Zeile 4), da diese bereits von der Transformation berücksichtigt werden. Somit ist sichergestellt, dass die aktuell untersuchte Relation eine *externe Relation* ist, die eine Abhängigkeit zu dem aktuellen Modellknoten des Subgraphen beschreibt und während der Transformation umgeleitet werden muss. Existiert keine Relationszuordnung, die dies ermöglicht (Zeilen 5-9), wird der Wahrheitswert *falsch* zurückgegeben (Zeile 11). Andernfalls gibt es eine Relationszuordnung, die (i) den entsprechenden Modellknoten aus der Musterstruktur als Musterelement definiert (Zeile 5) sowie (ii) einen Relationstypen definiert, der in den Typen der aktuell untersuchten Relation enthalten ist (Zeile 6). Des Weiteren muss (iii) die Richtung der Relation mit der in der Relationszuordnung angegebenen Richtung übereinstimmen (Zeile 7 und Algorithmus 5.2) und, sofern angegebenen (Zeile 8), (iv) der Typ des Ziel- bzw. Quellknotens der Relation entsprechend in der Relationszuordnung definiert sein (Zeile 9 und Algorithmus 5.2). Existiert eine

Algorithmus 5.1 istAnwendbar $\left(v \in \mathcal{V},\ t \in \mathcal{T},\ gz \in GZ_{t,\pi_{MusterS}(v)}\right)$

1: **for all** $(sez_i \in gz\ :\ \pi_{Entsprechung}(sez_i) \in MK_t)$ **do**
2: **if** $(\exists r_j \in R_t\ :$
3: $(\pi_{Quelle}(r_j) = \pi_{Entsprechung}(sez_i)\ \vee \pi_{Ziel}(r_j) = \pi_{Entsprechung}(sez_i)) \wedge$
4: $(\nexists sez_y \in gz : \pi_{Entsprechung}(sez_y) = r_j) \wedge$
5: $(\nexists rz_x \in RZ_v : \pi_{MusterE}(rz_x) = \pi_{Suchelement}(sez_i) \wedge$
6: $\pi_{Relationstyp}(rz_x) \in typen(r_j) \wedge$
7: $\pi_{Richtung}(rz_x) = \text{RICHTUNG}(r_j, \pi_{Entsprechung}(sez_i)) \wedge$
8: $(\pi_{Endpunkttyp}(rz_x) = \perp \vee$
9: $\pi_{Endpunkttyp}(rz_x) \in typen(\text{ENDPUNKT}(r_j, \pi_{Entsprechung}(sez_i))))$
10:)) **then**
11: **return** falsch
12: **end if**
13: **if** $\left(\exists a_j \in artefakte(\pi_{Entsprechung}(sez_i))\ :\right.$
14: $\left.\nexists az_x \in AZ_v : \pi_{Artefakttyp}(az_x) \in typen(a_j)\right)$ **then**
15: **return** falsch
16: **end if**
17: **end for**
18: **return** wahr

solche Relationszuordnung für alle externen Relationen, werden im nächsten Schritt die Artefakte der Modellknoten betrachtet (Zeile 13-16): Existieren Artefakte (Zeile 13), die nicht von einer Artefaktzuordnung zu einem Modellknoten des einzufügenden Fragments umgehängt werden können (Zeile 14), ist das PRM nicht anwendbar (Zeile 15). Können alle externen Relationen und Artefakte umgehängt werden, ist das PRM anwendbar (Zeile 18).

In den Zeilen 7 und 9 von Algorithmus 5.1 werden Hilfsmethoden verwendet, um die Richtung und den Endpunkt der Relation in Abhängigkeit von dem aktuell untersucht Modellknoten zu bestimmen. Diese Hilfsmethoden werden in Algorithmus 5.2 definiert. Beide bekommen jeweils eine Relation sowie einen Modellknoten übergeben. Die RICHTUNGS-Methode leitet anhand der Relation r und des Modellknotens mk ab, ob mk die Quelle oder das Ziel von r ist und bildet dies entsprechend auf die Richtungswerte $\mathcal{D}$, *ausgehend* und *eingehend*, ab. Ähnlich dazu gibt die ENDPUNKT-Methode den jeweils anderen Knoten der Relation in Bezug auf den Knoten mk zurück.

Algorithmus 5.2 Hilfsmethoden für Relationen in einem MDM $t \in \mathcal{T}$

Die Eingabe für beide Prozeduren ist: $(r \in R_t,\ mk \in MK_t)$

1: **procedure** RICHTUNG
2:     **if** $(mk = \pi_{Quelle}(r))$ **then**
3:         **return** ausgehend
4:     **end if**
5:     **if** $(mk = \pi_{Ziel}(r))$ **then**
6:         **return** eingehend
7:     **end if**
8:     **return** $\perp$
9: **end procedure**

1: **procedure** ENDPUNKT
2:     **if** $(mk = \pi_{Quelle}(r))$ **then**
3:         **return** $\pi_{Ziel}(r)$
4:     **end if**
5:     **if** $(mk = \pi_{Ziel}(r))$ **then**
6:         **return** $\pi_{Quelle}(r)$
7:     **end if**
8:     **return** $\perp$
9: **end procedure**

Algorithmus 5.1 beschreibt, wie geprüft werden kann, ob ein PRM zur Verfeinerung einer in einem musterbasierten Deployment-Modell identifizierten Musterstruktur verwendet werden kann. Er gilt jedoch ebenso für die Erkennung. Dabei wird anstatt der Musterstruktur die Lösungsstruktur des PRMs übergeben sowie ein Subgraph, der die Zuordnung zwischen der Lösungsstruktur und dem aktuell untersuchten Deployment-Modell beschreibt.

5.5 Verfeinerung von musterbasierten Deployment-Modellen

Im vorherigen Abschnitt 5.4 wurden Beispiele von PRMs vorgestellt und erklärt, wie festgestellt werden kann, ob ein PRM zur Verfeinerung eines musterbasierten Deployment-Modells verwendet werden kann. Kann ein PRM für die Verfeinerung eines Fragments in einem musterbasierten Deployment-Modell verwendet werden, so gilt es als *anwendbar*. Im Folgenden wird die *Anwendung* eines PRMs, d.h. die automatisierte Verfeinerung eines ausgewählten Subgraphen, auf das aktuell zu verfeinernde MDM erläutert.

Die Anwendung eines PRMs auf ein musterbasiertes Deployment-Modell ist in Algorithmus 5.3 definiert. Der Algorithmus beginnt mit einer Schleife, die prüft, ob das MDM t noch Muster enthält. Ansonsten wurden bereits alle Muster umgesetzt und das damit *ausführbare* Deployment-Modell t

wird zurückgegeben (Zeile 33). Sind jedoch noch Komponenten- oder Verhaltensmuster in t enthalten, werden alle PRMs in einem Repository auf ihre Anwendbarkeit hin überprüft. Dazu wird im ersten Schritt nach PRMs gesucht, deren Musterstruktur als Subgraphen in t identifiziert werden können. Dabei gelten zwei Strukturelemente als gleich, wenn sie den in Definition 5.11 definierten Entsprechungsoperator erfüllen. Die Hilfsmethode `berechneMusterSubgraphen` findet diese Subgraphen, indem sie einen Graphisomorphismusalgorithmus wie den *VF2* [CFSV04] verwendet, welcher mit dem Entsprechungsoperator erweitert wurde, um übereinstimmende Knoten und Kanten zu identifizieren. Die Hilfsmethode gibt alle Graphzuordnungen zwischen t und den Musterstrukturen aller PRMs in $\mathcal{V}$ zurück, wobei die Musterstrukturen als Subgraphen in t gefunden wurden (Zeile 2).

Da ein PRM kann allerdings nur dann auf t angewendet werden, wenn es alle externen Relationen und verlinkten Artefakte transformieren kann, müssen die gefundenen PRMs zunächst auf ihre Anwendbarkeit auf t überprüft werden. Ist ein PRM nicht anwendbar, könnten möglicherweise Information bei der Transformation verloren gehen. Daher werden alle PRMs, deren Musterstruktur in t zwar gefunden wurde jedoch nach Algorithmus 5.1 nicht als *anwendbar* gelten, ignoriert. Alle anderen PRMs werden in die Liste der *Kandidaten* aufgenommen, die in dieser Iteration auf t anwendbar sind (Zeilen 5-6). Wurden keine Kandidaten gefunden (Zeile 7), kann keine Verfeinerung erfolgen und t wird im aktuellen Zustand zurückgegeben (Zeile 29).

Im nächsten Schritt muss ein Kandidat gewählt werden, der angewendet werden soll (Zeile 8). Die Auswahl eines Kandidaten kann sowohl automatisiert als auch manuell geschehen und wird durch den Aufruf der `wählePRM`-Methode ausgeführt. Wird kein Kandidat gewählt, wird die Verfeinerung beendet (Zeilen 9-11). Ein Kandidat ist dabei ein Tupel aus einem PRM und einem konkreten Subgraphen, der verfeinert werden soll. Im Anschluss wird der Subgraph transformiert (Zeilen 12-27 von Algorithmus 5.3): Zuerst werden alle Strukturelemente der Lösungsstruktur des PRMs v_i in t eingefügt, die nicht Teil einer Bleibzuordnung sind (Zeile 12). Diese Elemente existieren bereits in t und müssen separat behandelt werden (s. Zeile 17 und Algorithmus 5.4). In den Zeilen 15 bis 23 wird daraufhin über alle Elemente

Algorithmus 5.3 verfeinere $\left(t \in \mathcal{T}\right)$

```
1:  while (∃muster ∈ KM_t ∪ VM_t) do
2:      G_{t,V} := berechneMusterSubgraphen(t, V)
3:      // Nur wenn ein PRM alle Relationen und Artefakte transformieren
4:      // kann, gilt es als Kandidat, der auf t angewendet werden kann.
5:      kandidaten := {(v_i, gz) : v_i ∈ V ∧ gz ∈ GZ_{t,π_{MusterS}(v_i)} ∈ G_{t,V}
6:                                  ∧ istAnwendbar(v_i, t, gz) }
7:      if (kandidaten ≠ ∅) then
8:          kandidat = (v_i, gz) := wählePRM(kandidaten)
9:          if (kandidat = ⊥) then
10:             return t
11:         end if
12:         SE_t := SE_t ∪ {se_j ∈ SE_{LösungsS_{v_i}} : ∄s_x ∈ BZ_{v_i} : π_{LösungsE}(s_x) = se_j}
13:         // Wende die Transformationen für jedes
14:         // Strukturelementzuordnung des Subgraphen an.
15:         for all ((se_t, se_{MusterE}) ∈ gz) do
16:             if (se_t ∈ MK_t) then
17:                 wendeBleibZuordnungenAn(v_i, t, (se_t, se_{MusterE}))
18:                 wendeArtefaktZuordnungenAn(v_i, t, (se_t, se_{MusterE}))
19:                 wendeRelationsZuordnungenAn(v_i, t, gz, (se_t, se_{MusterE}))
20:             end if
21:             wendeVerhaltensZuordnungenAn(v_i, t, (se_t, se_{MusterE}))
22:             wendeEigenschaftsZuordnungenAn(v_i t, (se_t, se_{MusterE}))
23:         end for
24:         // Finde alle Modellelemente, die aus t gelöscht werden müssen.
25:         SE_{Löschen} := {se_j ∈ SE_t | ∃sez_x ∈ gz ∄s_y ∈ BZ_{v_i} :
26:                 π_{Entsprechung}(sez_x) = se_j ∧ π_{Suchelement}(sez_x) = π_{MusterE}(s_y)}
27:         SE_t := SE_t \ SE_{Löschen}
28:     else
29:         return t
30:     end if
31: end while
32: return t
```

des Subgraphen, d. h. Strukturelementzuordnungen zwischen t und der Musterstruktur des PRMs v_i, iteriert, um die Verfeinerung durchzuführen.

Bei der Verfeinerung werden nacheinander alle in dem PRM definierten Zuordnungen angewendet. Dabei wird zunächst geprüft, ob es sich bei den aktuell untersuchten Strukturelementen um eine Zuordnung zwischen zwei Modellknoten handelt (Zeile 16). Ist dies der Fall, werden alle Bleibzuordnungen (Zeile 17 und Algorithmus 5.4), Artefaktzuordnungen (Zeile 18 und Algorithmus 5.5) und Relationszuordnungen (Zeile 19 und Algorithmus 5.6) angewendet, da diese nur für Modellknoten definiert werden können. Anschließend werden die Verhaltenszuordnungen (Zeile 21 und Algorithmus 5.7) und die Eigenschaftszuordnungen (Zeile 22 und Algorithmus 5.8) verarbeitet. Diese werden nicht nur für die Transformation von Modellknoten, sondern auch für Relationen angewendet.

Als letzten Schritt der Verfeinerung werden alle verfeinerten Strukturelemente aus dem musterbasierten Deployment-Modell entfernt (Algorithmus 5.3, Zeilen 25-27). Ausgenommen davon sind die Modellknoten, die durch eine Bleibzuordnung nur zur Identifizierung von Verhaltensmuster an Logikkomponenten in der Musterstruktur des PRMs benötigt wurden. Durch die Löschung aller Komponenten- und Verhaltensmuster aus t, die von dem aktuell angewendeten PRM verfeinert wurden, wird aus dem musterbasierten Deployment-Modell nach und nach ein ausführbares Deployment-Modell. Im weiteren Verlauf der Verfeinerung wird wieder nach anwendbaren Verfeinerungsmodellen gesucht, bis entweder keine Muster mehr enthalten sind (Zeile 1) oder keine weiteren PRMs angewendet werden können (Zeile 7).

In den Abschnitten 5.5.1 bis 5.5.3 wird zunächst die Anwendung der Modellknoten-spezifischen Zuordnungen erläutert, während Abschnitte 5.5.4 und 5.5.5 die Anwendung der Verhaltens- und Eigenschaftszuordnungen für alle von der Verfeinerung betroffenen Strukturelemente beschreibt. Die Algorithmen 5.4 bis 5.6 und 5.8 sind dabei für die Verfeinerung und Erkennung anwendbar, während Algorithmus 5.7 nur für die Verfeinerung gilt.

Algorithmus 5.4 Anwendbar für die Verfeinerung & Erkennung
wendeBleibZuordnungenAn$\big(v \in \mathcal{V},\ t \in \mathcal{T},\ (mk_t, mk_{MusterE}) \in SEZ_{t,\pi_{MusterS}(v)}\big)$

1: **if** $\big(\exists bz_i \in BZ_v : \pi_{MusterE}(bz_i) = mk_{MusterE}\big)$ **then**
2:     **for all** $\big(r_j \in R_t\ :\ r_y \in R_{LösungsS_v} : r_j = r_y\big)$ **do**
3:         **if** $\big(\pi_{Quelle}(r_y) = \pi_{LösungsE}(bz_i)\big)$ **then**
4:             $\pi_{Quelle}(r_j) := mk_t$
5:         **else if** $\big(\pi_{Ziel}(r_y) = \pi_{Lösungs}(bz_i)\big)$ **then**
6:             $\pi_{Ziel}(r_j) := mk_t$
7:         **end if**
8:     **end for**
9: **end if**

5.5.1 Anwendung von Bleibzuordnungen

Um eine Bleibzuordnung anzuwenden, wird das Tupel der beiden Modellknoten sowie das PRM v und das MDM t an Algorithmus 5.4 übergeben. Daher wird zunächst geprüft, ob eine Bleibzuordnung existiert, das angibt, ob der Modellknoten mk_t während der Transformation ignoriert werden soll (Zeile 1). Ist dies der Fall, müssen alle Relationen, die aus der Lösungsstruktur zu t hinzugefügt wurden und von mk_t ein- oder ausgehen sollen, umgehängt werden. Dafür wird für alle Relationen, die sowohl in t als auch in der Lösungsstruktur enthalten sind (Zeile 2), geprüft, ob für die Relationen in der Lösungsstruktur eine Abhängigkeit zu dem Modellknoten definiert ist, der mk_t entspricht (Zeilen 3-7). Ist der entsprechende Modellknoten der Quellknoten einer hinzugefügten Relation wird diese aktualisiert, sodass mk_t der Quellknoten der Relation wird (Zeile 3-4). Analog dazu wird der Zielknoten der hinzugefügten Relation aktualisiert, wenn der entsprechende Modellknoten das Ziel ist (Zeile 5-6).

5.5.2 Anwendung von Artefaktzuordnungen

Im zweiten Schritt der Modellknotentransformation werden die Artefaktzuordnungen angewendet (Zeile 16 in Algorithmus 5.3). Diese definieren, wie Artefakte, beispielsweise Zustandsartefakte, von einem Muster an eine

Algorithmus 5.5 Anwendbar für die Verfeinerung & Erkennung
wendeArtefaktZuordnungenAn$\big(v \in \mathcal{V},\ t \in \mathcal{T},\ (mk_t, mk_{MusterE}) \in SEZ_{t,\pi_{MusterS}(v)}\big)$

1: *// Anwenden der Artefaktzuordnungen: kopieren der Artefakte*
2: **for all** $\big(a_i \in A_t\ :\ \big(\exists az_j \in AZ_v : a_i \in artefakte(mk_t)$
3: $\quad \wedge\ mk_{MusterE} = \pi_{MusterE}(az_j)\ \wedge\ \pi_{Artefakttyp}(az_j) \in typen(a_i)\big)\big)$ **do**
4: $\quad zielElement := mk_x \in MK_t : mk_x = \pi_{LösungsE}(az_x)$
5: $\quad artefakte(zielElement) := artefakte(zielElement) \cup \{a_i\}$
6: **end for**

konkrete Komponente umgehängt werden können. Daher wird in Algorithmus 5.5 über alle Artefakte aus t iteriert, die dem aktuellen Modellknoten mk_t zugeordnet sind und eine Artefaktzuordnung existiert (Zeile 2), dessen Musterelement der Musterknoten $mk_{MusterE}$ der Strukturelementzuordnung ist und die Typen der Artefakte in den Artefaktzuordnungen definiert sind (Zeile 3). Durch die Anwendbarkeitsprüfung der PRMs ist hierbei sichergestellt, dass keine Artefakte existieren, die einem Modellknoten aus dem Subgraphen zugeordnet sind und nicht von dem aktuell angewendeten PRM umgehängt werden können (s. Algorithmus 5.1). Um ein Artefakt umzuhängen, wird der Modellknoten *zielElement* gesucht, der aus der Lösungsstruktur zu t hinzugefügt wurde und das Ziel der Artefaktzuordnungen darstellt (Zeile 4). Anschließend wird das Artefakt *zielElement* zugewiesen (Zeile 5).

5.5.3 Anwendung von Relationszuordnungen

Als letzten Schritt der Modellknoten-spezifischen Transformation müssen alle externen Relationen des Modellknotens mk_t umgehängt werden. Dafür werden die Relationszuordnungen, wie in Algorithmus 5.6 definiert, angewendet. Externe Relationen des Modellknotens mk_t sind die Relationen, die an mk_t ein- oder ausgehen und nicht Teil des Subgraphen sind. Somit wird eine Schleife über alle externen Relationen ausgeführt (Zeile 2-3). Wie auch bei den Artefaktzuordnungen ist durch die Anwendbarkeitsprüfung sichergestellt, dass alle externen Relationen umgehängt werden können. Daher wird

Algorithmus 5.6 Anwendbar für die Verfeinerung & Erkennung
wendeRelationsZuordnungenAn$\big(v \in \mathcal{V},\ t \in \mathcal{T},$
$gz \in GZ_{t,\pi_{MusterS}(v)},\ (mk_t, mk_{MusterE}) \in gz\big)$

1: *// Anwenden der Relationszuordnungen: umhängen externer Relationen*
2: **for all** $\big(r_i \in R_t\ :\ (\pi_{Quelle}(r_i) = mk_t \vee \pi_{Ziel}(r_i) = mk_t)$
3: $\quad \wedge \nexists sez_j \in gz : \pi_{Entsprechung}(sez_j) = r_i\big)$ **do**
4: $\quad rm := rz_x \in RZ_v : \pi_{MusterE}(rz_x) = mk_{MusterE} \wedge$
5: $\quad\quad \pi_{Relationstyp}(rz_x) \in typen(r_i) \wedge$
6: $\quad\quad \pi_{Richtung}(rz_x) = \textsc{Richtung}(r_j, mk_t) \wedge$
7: $\quad\quad (\pi_{Endpunkttyp}(rz_x) = \bot \vee$
8: $\quad\quad\ \pi_{Endpunkttyp}(rz_x) \in typen(\textsc{Endpunkt}(r_j, mk_t)))$
9: $\quad$ **if** $\big(\textsc{Richtung}(r_j, mk_t) = ausgehend\big)$ **then**
10: $\quad\quad \pi_{Quelle}(r_j) := \pi_{LösungsE}(rm)$
11: $\quad$ **else if** $\big(\textsc{Richtung}(r_j, mk_t) = eingehend\big)$ **then**
12: $\quad\quad \pi_{Ziel}(r_j) := \pi_{LösungsE}(rm)$
13: $\quad$ **end if**
14: **end for**

in den Zeilen 4-8 die entsprechende Relationszuordnung *rz* herausgesucht, welches die aktuelle externe Relation zu einem Modellknoten der Lösungsstruktur umhängen kann. Es muss dabei den Typen der Relation sowie dessen Richtung an mk_t entsprechend definieren. Darüber hinaus kann der jeweilige Endpunkt der Relation eingeschränkt sein, welcher ebenfalls entsprechend in der Relationszuordnung angeben sein muss. Mithilfe des Relationszuordnungen *rz* kann der neue Zielknoten (Zeilen 9-10) bzw. Quellknoten (Zeilen 11-12) der Relation identifiziert und zugewiesen werden.

5.5.4 Anwendung von Verhaltenszuordnungen

Für die Verfeinerung von Verhaltensmuster gilt generell, dass die Lösungsstruktur eines PRMs bereits alle in der Musterstruktur enthaltenen Verhaltensmuster realisiert. Daher werden in Algorithmus 5.7 zunächst alle Verhaltensmuster des aktuellen Strukturelements se_t gelöscht, welche ebenso in der Musterstruktur des PRMs definiert sind (Zeilen 2-6). Existieren zusätzliche

Algorithmus 5.7 Anwendbar für die Verfeinerung
wendeVerhaltensZuordnungenAn$\big(v \in \mathcal{V}, t \in \mathcal{T}, (se_t, se_{MusterE}) \in SEZ_{t,\pi_{MusterS}(v)}\big)$

1: *// Anwenden der Verhaltenszuordnungen: Umsetzen annotierter Muster*
2: **for all** $\big(anno_i \in annotationen(se_t)\ :\ \big(\exists vm_j \in VM_{MusterS_v}$:
3: $\quad typ(vm_j) \in typen(anno_i)\ \wedge\ vm_j \in annotationen(se_{MusterE})\big)\big)$ **do**
4: $\quad$ *// Das Verhaltensmuster wurde umgesetzt und wird gelöscht*
5: $\quad VM_t := VM_t \setminus \{vm_i\}$
6: **end for**
7: *// Existieren nicht erkannte Verhaltensmuster, müssen die Werte*
8: *// der Eigenschaften der Modellelemente, die in den entsprechenden*
9: *// Verhaltenszuordnungen definiert sind, gelöscht werden.*
10: **for all** $\big(vm_i \in VM_{MusterS}$:
11: $\quad \big(\nexists vm_j \in annotations(mk_t) : typ(vm_i) \in typen(vm_j)\big)\big)$ **do**
12: $\quad$ **for all** $\big(vz_y \in VZ_v\ :\ vm_i = \pi_{MusterE}(vz_y)\big)$ **do**
13: $\quad\quad se_{in} := se_z \in SE_t : se_z = \pi_{LösungsE}(vz_y)$
14: $\quad\quad e_{Ziel} := e_l \in eigenschaften(se_{in}) : \pi_{Eigenschaft}(vz_y) = \pi_{Schlüssel}(e_l)$
15: $\quad\quad \pi_{Wert}(e_{Ziel}) := \varepsilon$
16: $\quad$ **end for**
17: **end for**

Verhaltensmuster an dem Modellelement der Musterstruktur $se_{MusterE}$, die jedoch nicht an dem Modellelement se_t annotiert sind (Zeile 10-11), sind dem Entsprechungsoperator zufolge (Definition 5.11) Verhaltenszuordnungen für diese Verhaltensmuster angegeben. Dadurch ist definiert, dass sich diese Verhaltensmuster auf bestimmte Eigenschaften von Komponenten und Relationen auswirken. Daher müssen alle Eigenschaften, auf die sich ein Verhaltensmuster auswirkt, auf das leere Wort gesetzt werden. Andernfalls würden möglicherweise ungewollte Konfigurationen umgesetzt. Durch eine Schleife über alle Verhaltenszuordnungen eines Verhaltensmusters wird dies erzielt (Zeile 12): In Zeile 13 wird das der aktuellen Verhaltenszuordnung entsprechende Strukturelement se_{in} gesucht, dessen Eigenschaft betroffen ist. Anschließend wird die in der Verhaltenszuordnung definierte Eigenschaft e_{Ziel} identifiziert (Zeile 14) und ihr Wert auf das leere Wort gesetzt (Zeile 15). Somit ist sichergestellt, dass nur die Verhaltensmuster umgesetzt werden,

Algorithmus 5.8 Anwendbar für die Verfeinerung & Erkennung wende-EigenschaftsZuordnungenAn$\big(v \in \mathcal{V},\ t \in \mathcal{T},\ (se_t, se_{MusterE}) \in SEZ_{t,\pi_{MusterS}(v)}\big)$

1: **for all** $\big(ez_i \in EZ_v\ :\ \pi_{MusterE}(ez_i) = se_{MusterE}\big)$ **do**
2: $\quad se_{eingefügt} := se_j \in SE_t : se_j = \pi_{LösungsE}(ez_i)$
3: $\quad e_{Ziel} := e_y \in eigenschaften(se_{eingefügt}) : \pi_{LösungsProp}(ez_i) = \pi_{Schlüssel}(e_y)$
4: $\quad e_{AusgangsProp} := e_l \in eigenschaften(se_t) : \pi_{Schlüssel}(e_l) = \pi_{MusterProp}(ez_i)$
5: $\quad \pi_{Wert}(e_{Ziel}) := \pi_{Wert}(e_{AusgangsProp})$
6: **end for**

die sowohl in t als auch in der Musterstruktur des PRMs definiert sind. Alle anderen Verhaltensmuster werden dabei durch die in der Lösungsstruktur enthaltenen Komponenten und Technologien umgesetzt.

5.5.5 Anwendung von Eigenschaftszuordnungen

Sind alle Relationen sowie Artefakte umgehängt und Verhaltensmuster umgesetzt, werden schließlich die Eigenschaftszuordnungen angewendet. Diese definieren, wie der Eigenschaftswert eines Strukturelements in die hinzugefügte Lösungsstruktur übertragen werden kann. Algorithmus 5.8 beschreibt deren Anwendung auf eine Strukturelementzuordnung: Für jede Eigenschaftszuordnung, die dem aktuellen Strukturelement der Musterstruktur entspricht, (Zeile 1), werden zuerst das Zielelement $se_{eingefügt}$ und (Zeile 2) die im Eigenschaftszuordnung betreffende Zieleigenschaft gesucht (Zeile 3). Anschließend wird die Eigenschaft des Strukturelements se_t gesucht (Zeile 4) und ihr Wert dem Wert der Zieleigenschaft zugewiesen (Zeile 5).

5.6 Erkennung von Mustern in Deployment-Modellen

Die Erkennung von Mustern in Deployment-Modellen folgt im Allgemeinen dem gleichen Prozess wie die Verfeinerung. Um Muster in ausführbaren Deployment-Modellen zu identifizieren und diese in musterbasierte Deployment-Modelle zu transformieren, wird im Folgenden auf die Gemeinsamkeiten und Unterschiede beider Verfahren eingegangen.

Anstatt Subgraphen in einem musterbasierten Deployment-Modell, die den Musterstrukturen von PRMs entsprechen, zu identifizieren und zu konkreten Komponenten zu verfeinern, wird bei der Mustererkennung in ausführbaren Deployment-Modellen nach Subgraphen gesucht, die der Lösungsstruktur von Erkennungsmodellen (PDMs) entsprechen, um diese durch die entsprechenden Musterstrukturen zu ersetzen. Erkennungsmodelle sind im Allgemeinen Verfeinerungsmodelle, die zusätzlich für die Erkennung von Mustern verwendet werden können. Wie bereits in Abschnitt 5.1 erwähnt, können nicht alle Muster auf der Grundlage des Deployment-Modells erkannt werden. Um zum Beispiel festzustellen, ob die Arbeitslast einer Anwendung statisch ist, sich ständig ändert oder sogar unvorhersehbar ist, kann nur während des Betriebs gemessen und nicht aus dem Deployment-Modell abgeleitet werden. Daher kann zwar von einem Unpredictable Workload-Muster auf eine automatisierte Skalierung geschlossen werden, jedoch kann dieses Verhalten nicht aus einem ausführbaren Deployment-Modell abgeleitet werden. Darum wird zwischen Verfeinerungs- und Erkennungsmodellen unterschieden, wobei Erkennungsmodelle immer auch als Verfeinerungsmodelle verwendet werden können, da Muster, die erkannt werden können, immer auch zu den erkennbaren Strukturen verfeinert werden können.

In Algorithmus 5.9 ist der Algorithmus der Mustererkennung beschreiben. Im Gegensatz zur Verfeinerung beginnt der Algorithmus mit einer Endlosschleife (Zeile 1), da keine allgemeine Abbruchbedingung definiert werden kann. Die Abbruchbedingung in der Verfeinerung ergibt sich aus den verfeinerten Mustern: Sind weder Verhaltens- noch Komponentenmuster in einem Deployment-Modell vorhanden, war die Verfeinerung erfolgreich und kann beendet werden. Da in einem musterbasierten Deployment-Modell jedoch auch konkrete Komponenten wie beispielsweise Logikkomponenten enthalten sein können, gilt die Umkehrung der Bedingung, d. h., ein Modell enthält ausschließlich Muster und keine konkreten Komponenten, nicht als Abbruchbedingung. Daher wird für die Erkennung eine Endlosschleife eingesetzt, welche jedoch während jederzeit unterbrochen werden kann, indem kein anwendbarer PDM-Kandidat gefunden wird (Zeile 8) oder die Erkennung durch eine leere Auswahl explizit beendet wird (Zeile 10).

Algorithmus 5.9 erkenneMuster $\left(t \in \mathcal{T}\right)$

```
1:  while (wahr) do
2:      G_{t,V} := berechneLösungsSubgraphen(t, V)
3:      // Nur wenn ein PDM alle Relationen und Artefakte transformieren
4:      // kann, gilt es als Kandidat, der auf t angewendet werden kann.
5:      kandidaten := {(v_i, gz) | v_i ∈ V ∧ π_ErkennungsM(v_i) = wahr
6:                                 ∧ gz ∈ GZ_{t,π_LösungsS(v_i)} ∈ G_{t,V}
7:                                 ∧ istAnwendbar(v_i, t, gz) }
8:      if (kandidaten ≠ ∅) then
9:          kandidat = (v_i, gz) := wählePDM(kandidaten)
10:         if (kandidat = ⊥) then
11:             return t
12:         end if
13:         SE_t := SE_t ∪ {se_j ∈ SE_{MusterS_{v_i}} | ∄s_x ∈ BZ_{v_i} : π_MusterE(s_x) = se_j}
14:         VM_t := VM_t ∪ VM_{MusterS_{v_i}}
15:         // Wende die Transformationen für jedes
16:         // Strukturelementzuordnung des Subgraphen an.
17:         for all ((se_t, se_LösungsE) ∈ gz) do
18:             if (se_t ∈ MK_t) then
19:                 wendeBleibZuordnungenAn(v_i, t, (se_t, se_LösungsE))
20:                 wendeArtefaktZuordnungenAn(v_i, t, (se_t, se_LösungsE))
21:                 wendeRelationsZuordnungenAn(v_i, t, gz, (se_t, se_LösungsE))
22:             end if
23:             erkenneVerhaltensMuster(v_i, t, (se_t, se_LösungsE))
24:             wendeEigenschaftsZuordnungenAn(v_i, t, (se_t, se_LösungsE))
25:         end for
26:         // Finde alle Modellelemente, die aus t gelöscht werden müssen.
27:         SE_Löschen := {se_j ∈ SE_t | ∃sez_x ∈ gz ∄s_y ∈ BZ_{v_i} :
28:                 π_Entsprechung(sez_x) = se_j ∧ π_Suchelement(sez_x) = π_LösungsE(s_y)}
29:         SE_t := SE_t \ SE_Löschen
30:     else
31:         return t
32:     end if
33: end while
```

Algorithmus 5.10 Anwendbar für die Erkennung
erkenneVerhaltensMuster$\big(v \in \mathcal{V},\ t \in \mathcal{T},\ (se_t, se_{LösungE}) \in SEZ_{t,\pi_{LösungS}(v)}\big)$

1: **for all** $\big(vm_i \in VM_{MusterS_v}\ :\ \big(\exists vz_j \in VZ_v$
2: $\quad\quad \exists prop_x \in eigenschaften(se_t)\ \exists prop_y \in eigenschaften(se_{LösungsE}) :$
3: $\quad\quad\quad vm_i = \pi_{MusterE}(vz_j)\ \wedge\ \pi_{LösungsE}(vz_j) = se_{LösungsE}$
4: $\quad\quad\quad \wedge\ \pi_{Schlüssel}(prop_x) = \pi_{Schlüssel}(prop_y) = \pi_{Eigenschaft}(vz_j)$
5: $\quad\quad\quad \wedge\ \pi_{Wert}(prop_x) \neq \pi_{Wert}(prop_y)\big)\big)$ **do**
6: $\quad VM_t := VM_t \setminus \{vm_i\}$
7: **end for**

Ob ein PDM auf ein Deployment-Modell t anwendbar ist, folgt denselben Regeln wie die Anwendung von PRMs bei der Verfeinerung. Zunächst muss die Lösungsstruktur als Subgraph in t identifiziert werden (Zeile 2). Wie bei der Verfeinerung findet die Hilfsmethode `berechneLösungsSubgraphen` alle Graphzuordnungen zwischen t und den Lösungsstrukturen aller PRMs in $\mathcal{V}$ mithilfe eines Graphisomorphismusalgrotihmus, welcher mit dem Entsprechungsoperator erweitert wurde, um übereinstimmende Knoten und Kanten zu finden. Der Entsprechungsoperator gilt ohne weitere Änderungen. Des Weiteren gilt auch die Anwendbarkeitsprüfung aus Algorithmus 5.1. Anstelle eines Subgraphen zwischen t und der Musterstruktur eines PRMs wird jedoch der Subgraph zwischen t und der Lösungsstruktur eines PDMs übergeben (Zeile 7). Alle auf t anwendbaren PDMs werden wiederum als Kandidaten gesammelt (Zeilen 5-7) und, falls die Liste der Kandidaten nicht leer ist (Zeile 8), wird durch die `wählePDM`-Methode ein Kandidat automatisiert oder manuell ausgewählt (Zeile 9). Wurde kein Kandidat gewählt, wird die gesamte Erkennung unterbrochen (Zeilen 10-12).

Im nächsten Schritt werden alle Strukturelemente der Musterstruktur des ausgewählten PDMs zu den Strukturelementen aus t hinzugefügt, die nicht Teil einer Bleibzuordnung sind (Zeile 13). Außerdem werden alle Verhaltensmuster aus der Musterstruktur zu t hinzugefügt (Zeile 14). Im Anschluss wird die Transformation analog zur Transformation der Verfeinerung durchgeführt (Zeilen 17-25). Eine Ausnahme bildet dabei die Erkennung der Verhaltensmuster (Zeile 23), welche in Algorithmus 5.10 definiert ist.

Im Allgemeinen gelten Verhaltensmuster als umgesetzt, wenn diese entweder (i) nicht Teil einer Verhaltenszuordnung sind oder (ii) alle in Verhaltenszuordnungen definierten Eigenschaften in t wiedergefunden werden können. Da in Zeile 14 des Algorithmus 5.9 bereits alle Verhaltensmuster der Musterstruktur zu t hinzugefügt wurden, müssen im nächsten Schritt all diejenigen Verhaltensmuster entfernt werden, die diese Anforderungen nicht erfüllen. Dafür wird in Algorithmus 5.10 über alle Verhaltensmuster iteriert, die Teil einer Verhaltenszuordnung sind (Zeile 1 und 3). Existiert für ein Verhaltensmuster eine Verhaltenszuordnung, deren Eigenschaftswert nicht in t wiedergefunden werden kann (Zeilen 2-5), wird das Verhaltensmuster wieder aus t entfernt (Zeile 6), da nicht alle notwendigen Eigenschaften, die das Verhaltensmuster beeinflussen, wie erwartet gesetzt sind.

Als letzten Schritt der Mustererkennung werden, wie auch bei der Verfeinerung, alle Strukturelemente, die als Subgraph der Lösungsstruktur des PDMs in t identifiziert wurden, gelöscht (Algorithmus 5.9, Zeilen 27-29). Modellknoten, die durch eine Bleibzuordnung ignoriert werden, sind hiervon jedoch ausgenommen. Die Mustererkennung wird im Anschluss so lange fortgeführt, bis entweder keine anwendbaren PDM-Kandidaten mehr gefunden werden können (Zeilen 8 und 31) oder die Erkennung explizit durch eine leere PDM-Auswahl beendet wird (Zeilen 10 und 11).

5.7 Beispiel einer Verfeinerung von Mustern

Die Verfeinerung einer Anwendung ist ein iterativer Prozess und besteht möglicherweise aus zahlreichen Iterationen. Die Anzahl der Iterationen hängt von der Größe der Anwendung und der Menge der Muster ab, die in einer Iteration verfeinert werden. Das Deployment-Modell ist während der Verfeinerung nicht ausführbar, da jeweils nur einzelne Fragmente des Modells verfeinert werden. Abbildung 5.6 beschreibt eine ausführbare Variante des in Abbildung 4.2 beschriebenen musterbasierten Deployment-Modells.

Auf der linken Seite des ausführbaren Deployment-Modells in Abbildung 5.6 ist der Webshop als *Angular-Web-App* dargestellt. Wie auch in dem

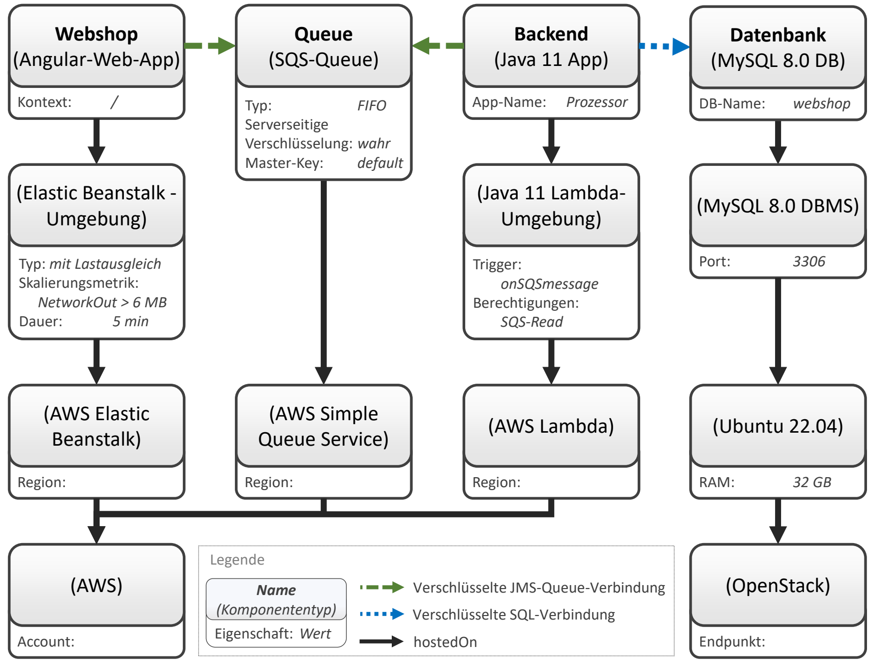


Abbildung 5.6: Eine mögliche Verfeinerung des in Abbildung 4.2 dargestellten musterbasierten Deployment-Modells

musterbasierten Deployment-Modell in Abbildung 4.2 soll der Webshop auf dem Root-Pfad erreichbar sein. Anstatt jedoch auf einem PaaS-Muster bereitgestellt zu sein, wird der Webshop nun von einer *Elastic Beanstalk-Umgebung* bereitgestellt. Durch die Annotationen des Webshops im MDM, welche sie als eine zustandslose UI-Komponente auszeichnen, für die eine unvorhersehbare Arbeitslast erwartet wird, wurde die *Elastic Beanstalk*-Umgebung für ein automatisches Skalieren konfiguriert. Daher ist der Umgebungstyp „mit Lastausgleich" und die Skalierungsmetrik als „NetworkOut > 6 MB" für eine Dauer von fünf Minuten definiert. Daher wird automatisch eine neue Instanz des Webshops hochgefahren, sobald die aktuelle Instanz über fünf Minuten lang konstant mit mindestens sechs Megabyte Daten sendet. Dabei wird die *Elastic Beanstalk*-Umgebung von dem *AWS Elastic Beanstalk*-

Angebot bereitgestellt. Allerdings sind die Region, in welcher die *Elastic Beanstalk*-Umgebung betrieben werden soll, sowie der Account, der für *AWS* verwendet werden soll, nicht angegeben. Diese Werte können nicht aus dem MDM abgeleitet werden und müssen von Modellierenden im Anschluss oder während der Bereitstellung angegeben werden.

Da in dem musterbasierten Deployment-Modell der Webshop, die Queue sowie das Backend auf derselben Public Cloud bereitgestellt werden sollen, werden auch in der ausführbaren Variante alle drei Komponenten von einem Anbieter bereitgestellt. Daher wurde der MESSAGE CHANNEL zu einer Queue des Komponententyps *SQS Queue* verfeinert, welche von dem *AWS Simple Queue Service*-Angebot verwaltet wird. Aufgrund des annotierten INFORMATION OBSCURITY-Musters wurde die Queue so konfiguriert, dass die Inhalte der Nachrichten durch eine serverseitige Verschlüsselung „verschleiert" werden. Des Weiteren wurde das EXACTLY-ONCE DELIVERY-Muster umgesetzt, indem der Typ der Queue auf „*FIFO*" gesetzt wurde.

Ähnlich zum Webshop ist auch das Backend als zustandslose Komponente, welche eine nicht vorhersagbare Arbeitslast erwartet, in Abbildung 4.2 dargestellt. Zusätzlich ist das Backend eine verarbeitende Komponente, die nur durch bestimmte Events ausgeführt wird, d. h. sie wurde im MDM mit dem EVENT-DRIVEN CONSUMER-Muster annotiert. Des Weiteren sollte das Backend von einer STATELESS HOSTING-Umgebung bereitgestellt werden, welches zu einer *Java 11 Lambda*-Umgebung des *AWS Lambda*-Angebots verfeinert wurde. Um das im musterbasierten Deployment-Modell modellierte Verhalten der Backend-Komponente umzusetzen, ist die *Java 11 Lambda*-Umgebung so konfiguriert, dass neue Nachrichten in der Queue die Ausführung des Backends auslösen. Dafür ist die *Trigger*-Eigenschaft auf den Wert „*onSQSmessage*" gesetzt. Alle weiteren Verhaltensmuster wurden dabei direkt durch die Eigenschaften von *AWS Lambda* realisiert, da diese als *Function as a Service*-Angebot für einen automatischen Lastausgleich des Backends sorgt und somit die unvorhersehbare Arbeitslast abfangen kann.

Auch die relationale Datenbank, welche in Abbildung 4.2 auf einem EXECUTION ENVIRONMENT in einer PRIVATE CLOUD laufen sollte, wurde in Abbildung 5.6 mit konkreten Technologien ersetzt. Eine *MySQL 8.0 DB* in

einem entsprechenden *MySQL 8.0 DBMS* (Datenbankmanagementsystem) wird auf einer *Ubuntu 22.04* VM bereitgestellt. Als Private Cloud soll eine *OpenStack*-Instanz dienen und die VM mit der Datenbank verwalten.

Neben konkreten Komponenten wurden auch die mit dem Secure Channel annotierten Relationen verfeinert. Anstelle der Relationen vom Typ *JMS-Queue-Verbindung* sind diese nun vom Relationstyp *verschlüsselte JMS-Queue-Verbindung*. Somit ist sichergestellt, dass die Kommunikation zwischen den beiden Komponenten durch eine Verschlüsselung von Zugriffen von Außen abgesichert sind. Ähnlich dazu wurde auch die *SQL-Verbindung* zu einer *verschlüsselten SQL-Verbindung* verfeinert, um einen sicheren Datenaustausch zwischen Backend und Datenbank zu gewährleisten.

5.8 Diskussion

Die Idee von musterbasierten Deployment-Modellen besteht darin, die Komponenten und Konfigurationen einer Anwendung abstrakt beschreiben zu können. Wie in Abbildung 4.2 gezeigt, kann eine Anwendung auf Basis von Muster modelliert werden. Die in Abbildung 5.6 dargestellte Beispielanwendung zeigt, wie ein Webshop, der sich über einen Message Channel zu einem Backend verbindet, umgesetzt werden kann. Allerdings erwarten die Implementierungen der Logikkomponenten in der Regel eine bestimmte Messaging-Technologie, zu der sie sich verbinden können. Daher dürfen bei der Verfeinerung nur PRMs ausgewählt werden, die die modellierten Muster zu geeigneten konkreten Technologien verfeinern. Die präsentierte Methode stellt diese korrekte Verfeinerung sicher, indem Modellierende die Beziehungen zwischen Mustern und konkreten Komponenten detailliert definieren müssen. Das Frontend und das Backend verbinden sich mit dem Message Channel über eine *JMS-Queue-Verbindung*, was den Verfeinerungsalgorithmus darauf beschränkt, nur PRMs als anwendbar zu identifizieren, die das Muster zu einer kompatiblen Messaging-Technologie verfeinern. Dies erfordert jedoch eine detaillierte Modellierung aller Beziehungen zwischen den Komponenten und Mustern eines musterbasierten Deployment-Modells.

Neben einer detaillierten Modellierung der musterbasierten Anwendung werden im Allgemeinen auch detaillierte und widerspruchsfrei modellierte Verfeinerungs- und Erkennungsmodelle benötigt. Werden beispielsweise in einem Verfeinerungsmodell Verhaltensmuster modelliert, die durch Verhaltenszuordnungen bestimmte Eigenschaften beeinflussen, dürfen keine Eigenschaftszuordnungen existieren, die diese Werte bei der Verfeinerung überschreiben. Auch Verhaltenszuordnungen, die Eigenschaftszuordnungen überschreiben, führen zu inkonsistenten Transformationen. Dabei müssen Modellierende sicherstellen, dass PRMs und PDMs keine Zuordnungen definieren, die sich gegenseitig beeinflussen können. Ansonsten kann es bei der Verfeinerung und Erkennung zu falschen und inkonsistenten Transformationen kommen. Dies gilt sowohl für Modellknoten, die in der Topologie verbleiben sollen, d. h. Bleibzuordnungen, als auch für Strukturelemente, die verfeinert werden. Zwar können Eigenschafts-, Artefakt- oder Relationszuordnungen für verbleibende Modellknoten definiert werden, diese sind jedoch nicht nötig, da diese Modellknoten während der Transformation ignoriert werden. Sie werden ausschließlich dazu benötigt, um annotierte Verhaltensmuster an Logikkomponenten zu erkennen und zu verfeinern.

Generell ist die Methode durch die Anwendbarkeit der verfügbaren PRMs bzw. PDMs sowie den zuvor genannten Bedingungen begrenzt. Zur Verfeinerung oder Erkennung von Mustern werden möglicherweise große Mengen an PRMs und PDMs benötigt. In jeder Iteration der Algorithmen werden Teile eines Deployment-Modells durch eine Muster- oder Lösungsstruktur ausgetauscht. Dadurch können PRMs gegebenenfalls nicht mehr als anwendbar erkannt werden, die in einer früheren Iteration anwendbar waren, da ein Muster oder eine Komponente bereits ausgetauscht wurde. Die Anwendung von PRMs und PDMs ist somit nicht kommutativ. Wenn beispielsweise das in Abbildung 5.4 dargestellte PRM auf das MDM in Abbildung 4.2 angewendet wird, würde das Public Cloud Muster verfeinert werden. Soll in einer zweiten Iteration ein ähnliches PRM, das eine Angular-Web-App auf einem PaaS Muster auf einer Public Cloud verfeinern kann, angewendet werden, wäre dies nicht mehr möglich, da das Public Cloud Muster nicht mehr in dem MDM gefunden und somit kein übereinstimmender Subgraph identifiziert

werden kann. Dies führt dazu, dass eine große Anzahl unterschiedlicher Erkennungs- und Verfeinerungsmodelle benötigt wird, die dieselben Technologiekonfigurationen beschreiben – jedoch mit verschiedenen Muster- und Lösungsstrukturen. Durch die Einführung der Verhaltenszuordnungen zur bedingten Erkennung und Verfeinerung von Verhaltensmustern wird die Anzahl der benötigten Verfeinerungsmodelle bereits reduziert, da PRMs auch angewendet werden können, wenn die Verhaltensmuster nicht in dem zu verfeinernde MDM enthalten ist (s. Definition 5.11). Allerdings werden nach wie vor zahlreiche Permutationen der PRMs und PDMs für eine vollständige Erkennung und Verfeinerung von großen Deployment-Modellen benötigt.

Des Weiteren ist es unter Umständen nicht möglich, alle Muster in einem Deployment-Modell zu verfeinern bzw. zu erkennen. Dies kann verschiedene Gründe haben: Zum Beispiel kann (i) das Repository, das PRMs und PDMs enthält, nicht vollständig sein oder lediglich wenige Verfeinerungsmodelle enthalten. (ii) Muster können architektonische Aspekte der Anwendung beschreiben, die sich nicht auf konkrete Komponenten, Beziehungen und Konfigurationen abbilden lassen. Außerdem gibt es (iii) Muster, die das Fehlen anderer Muster oder Komponenten erfordern; dies kann mit dem subgraphenbasierten Ansatz allerdings nicht erkannt werden. Zum Beispiel kann das Customizable Deployment Stack-Muster [YBB+22] nicht in Kombination mit dem Fixed Deployment Stack-Muster [YBB+22] verwendet werden, da sie exklusive Alternativen zueinander beschreiben. Schließlich (iv) gibt es Muster, die nicht anhand des Deployment-Modells einer Anwendung erkannt werden können, sondern nur anhand von Laufzeitmessungen. Um zum Beispiel festzustellen, ob die Arbeitslast einer Anwendung statisch ist, sich ständig ändert oder sogar unvorhersehbar ist, kann die Arbeitslast nur gemessen und nicht aus Konfigurationen abgeleitet werden.

Zur Erkennung und Verfeinerung von Mustern in Deployment-Modellen wird zwar eine große Menge an PRMs und PDMs benötigt, sind diese jedoch einmal definiert, können sie in jeder neuen Erkennung bzw. Verfeinerung verwendet werden. Daher ist zwar die initiale Erstellung mit viel Aufwand verbunden, jedoch ist dieser nur einmal nötig. Darüber hinaus können Verfeinerungs- und Erkennungsmodelle nicht nur zur automatisier-

ten Transformation genutzt werden, sondern dienen gleichermaßen der Dokumentation. Da PRMs und PDMs definieren, wie bestimmte Muster und Musterkonstellationen zu konkreten Technologien und deren Konfigurationen verfeinert werden können, ist in ihnen das Wissen dokumentiert, wie die Muster mit verschiedenen Technologien und Service-Anbietern umgesetzt werden können. Somit eignen sich PRM- und PDM-Repositorys auch für die Aus- und Weiterbildung sowie als Nachschlagewerk für Modellierende, die Deployment-Modelle entwickeln. Die grafische Darstellung der Verfeinerungsmodelle hilft dabei zusätzlich beim Verständnis.

Der Ansatz der iterativen Verfeinerung und Erkennung von Mustern in Deployment-Modellen bzw. Architekturmodellen im Allgemeinen realisiert darüber hinaus die von Falkenthal et al. [FBB+14a; FBB+14b; FBB+15] eingeführten Konzepte zur Navigation zwischen verschiedenen Ebenen von Mustern, Mustersprachen und deren Lösungen. In einer Zusammenarbeit mit Yussupov et al. [YBB+22] konnte dies bereits gezeigt werden. Dabei werden Verfeinerungsmodelle nicht nur zwischen Mustern und konkreten Technologien definiert, sondern auch zwischen Mustern unterschiedlicher Ebenen. Zum Beispiel kann die Architektur einer Anwendung mit diesem Verfahren durch das Aufzeigen verschiedener Lösungsoptionen, d. h. PRM-Kandidaten, von einer sehr abstrakten Beschreibung über spezialisierte sowie anbieterspezifische Mustersprachen immer konkreterer werden, bis schließlich konkrete Technologien und Anbieter gewählt werden können [YBB+22]. Damit eignet sich die Methode nicht nur für die musterbasierte Modellierung von Deployment-Modellen, sondern auch im Allgemeinen für die Modellierung, Verfeinerung und Erkennung von Mustern in einer Anwendung.

Kapitel 6

Erweiterung von Anwendungen um Managementfunktionalitäten

Nachdem im vorherigen Kapitel eine Methode präsentiert wurde, um musterbasierte Deployment-Modelle in ausführbare zu transformieren, wird in diesem Kapitel ein Verfahren beschrieben, mit dem ausführbare Deployment-Modelle automatisiert um zusätzliche Managementfunktionalitäten erweitert werden können. Dafür werden in Abschnitt 6.1 die Konzeptidee und -übersicht präsentiert, während in den Abschnitten 6.2 und 6.3 die Einzelheiten des Verfahrens vorgestellt werden. Anschließend wird die Workflowgenerierung für das Einfrieren und Auftauen von Anwendung in Abschnitt 6.4 vorgestellt. Zum Abschluss wird die Methodik in Abschnitt 6.5 diskutiert.

Die in diesem Kapitel präsentierten Konzepte und Verfahren wurden im Rahmen dieser Arbeit bereits in Publikationen auf zwei Konferenzen, der EDOC 2019 [HBL+19] und SummerSoC 2019 [HBKL19], vorgestellt. Darüber hinaus wurden die Konzepte in den Folgearbeiten [HBB+21; HBBL23], welche auch in Kapitel 7 vorgestellt werden, wiederverwendet.

6.1 Konzeptidee und -übersicht

Die Bereitstellung von Anwendungen kann mit Deployment-Technologien wie Terraform, Chef oder Kubernetes vollständig automatisiert werden. Die automatisierte Verwaltung bereitgestellter Anwendungen ist jedoch nur in begrenztem Umfang möglich: Während Funktionalitäten, wie die Skalierung oder Neukonfiguration von Anwendungskomponenten, in der Regel von Cloud-Anbietern oder deklarativen Konfigurationsmanagementtechnologien nativ unterstützt werden, werden ganzheitliche Managementfunktionen, die mehrere Komponenten betreffen und über verschiedene Umgebungen verteilt sind, nur in begrenztem Umfang unterstützt. Um beispielsweise Backups aller zustandsbehafteten Komponenten durchzuführen oder die Verfügbarkeit aller bereitgestellten Komponenten zu testen, ist eine manuelle Implementierung solcher Funktionen erforderlich, da Bereitstellungen in mehreren Umgebungen weder von einem einzigen Cloud-Anbieter verwaltet werden können, noch Aufgaben wie das Sichern aller Datenbanken oder das Testen der Komponenten von diesen Technologien unterstützt wird. Deshalb wird in diesem Kapitel ein Konzept präsentiert, das es ermöglicht, eine Anwendung mit zusätzlichen Managementfunktionalitäten zu erweitern und diese automatisiert über mehrere verteilte Umgebungen hinweg auszuführen. Ein Überblick über das Konzept ist in Abbildung 6.1 dargestellt.

Die Grundidee des Konzepts ist es, auf Basis der Komponententypen in einem ausführbaren Deployment-Modell verfügbare Managementfunktionalitäten in einem Repository zu identifizieren und die Komponententypen des Deployment-Modells mit den identifizierten Funktionalitäten zu erweitern. Die zusätzlichen Managementfunktionalitäten werden auch als *Managementfeatures* bezeichnet. Um die Erweiterung des Deployment-Modells um zusätzliche Managementfeatures zu erreichen, werden dabei *angereicherte Deployment-Modelle* generiert. Zusätzlich zu den Lebenszyklusoperationen beinhalten Komponententypen in angereicherten Deployment-Modelle auch Operationen, die die angereicherten Managementfunktionalitäten realisieren. Im Anschluss werden für die einzelnen Managementfeatures Workflows generiert, die zur Laufzeit der Anwendung ausgeführt werden können.

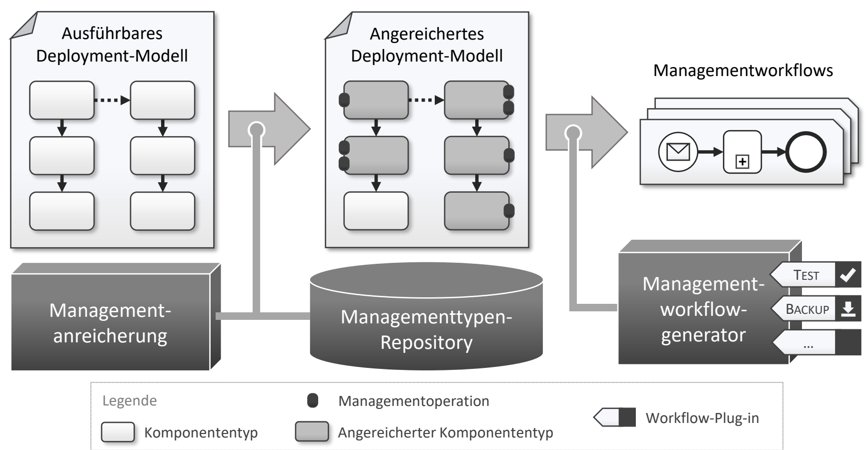


Abbildung 6.1: Erweiterung von ausführbaren Deployment-Modellen um zusätzliche Managementfunktionalitäten

Wie in Abbildung 6.1 dargestellt, wird ein ausführbares Deployment-Modell zunächst von der *Managementanreicherung* analysiert. Dabei werden Komponententypen und *Managementtypen* unterschieden. Komponententypen beschreiben und definieren dabei die Semantik von Komponenten, die diesem Typ zugewiesen sind, während Managementtypen die Managementoperationen eines Managementfeatures für einen Komponententyp definieren. Managementoperation für eine MySQL-Datenbank sind z. B. Backup-Operationen oder Operationen, die die Erreichbarkeit der Datenbank testen. Damit stehen für diesen Datenbanktyp die beiden Managementfeatures `Backup` und `Test` zur Verfügung. Sind für eine Web-App beispielsweise nur verschiedene Verfügbarkeits- oder Erreichbarkeitstests definiert, steht nur das `Test`-Feature für diesen Komponententyp zur Auswahl.

Findet die Managementanreicherung verfügbare Managementfeatures, können diese durch eine automatische oder manuelle Auswahl automatisiert zu dem ausführbaren Deployment-Modell hinzugefügt werden. Dabei werden neue Komponententypen generiert, die neben den Lebenszyklusoperationen auch die Operationen beinhalten, die die ausgewählten Ma-

nagementfeatures realisieren. Diese neuen Typen werden als *angereicherte Komponententypen* bezeichnet und gekennzeichnet. Alle Komponententypen, für die angereicherte Komponententypen generiert wurden, werden schließlich in dem Deployment-Modell ausgetauscht. Dadurch stehen die Managementfeatures der Anwendung zur Laufzeit zur Verfügung. In Abbildung 6.1 sind die angereicherten Komponententypen in dunkelgrau dargestellt, während zusätzliche Operationen als schwarze Ovale an den jeweiligen Komponenten annotiert sind. Im Gegensatz zu Komponententypen können Managementtypen nicht für die Modellierung verwendet werden, da diese nur Management-, jedoch keine Lebenszyklusoperationen definieren.

Im zweiten Schritt wird das angereicherte Deployment-Modell an einen *Managementworkflowgenerator* übergeben, welcher für jedes Managementfeature einen separaten Workflow generiert. Der Managementworkflowgenerator ist dabei eine Plug-in-basierte Anwendung, welche für jedes unterstützte Managementfeature ein Plug-in benötigt. Ein Plug-in muss dabei für das unterstützte Managementfeature einen Workflow generieren können. Durch die Plug-in-Architektur ist es möglich, dass die Anforderungen eines jeden Managementfeatures abgebildet und umgesetzt werden können. Je nach Managementfeature müssen beispielsweise mehrere Operationen in einer bestimmter Reihenfolge ausgeführt werden. Um beispielsweise einen Test-Workflow zu generieren, welcher alle Operationen des `Test`-Features ausführt, ist es sinnvoll, zuerst die Tests der Komponenten auszuführen, welche sich in einem Stack ganz unten befinden. Schlagen diese bereits fehl, kann davon ausgegangen werden, dass Komponenten, die auf diesen Komponenten bereitgestellt wurden, ebenfalls nicht erreichbar sind bzw. die Bereitstellung nicht erfolgreich war. Wird jedoch ein Workflow generiert, der Backups von allen zustandsbehafteten Komponenten erstellt, kann beispielsweise zwischen den Komponenten unterschieden werden, die Backups im laufenden Betrieb erstellen können und denen, die zuerst in einen Wartungsmodus versetzt werden müssen, um Inkonsistenzen zu vermeiden.

Die generierten Managementworkflows können im Anschluss zur Laufzeit der Anwendung ausgeführt werden. Durch die Generierung der Workflows ist zudem sichergestellt, dass diese mehrfach ausgeführt werden können.

6.2 Anreicherung von Managementfunktionalitäten

Im Folgenden werden Managementtypen beschrieben sowie ein Algorithmus zur automatisierten Anreicherung von Managementfunktionalitäten.

6.2.1 Managementtypen

Um die Anreicherung von Anwendungen mit zusätzlichen Managementfunktionalitäten zu ermöglichen, müssen diese zunächst in einem Repository definiert werden. Dazu können Deployment-Modellierende *Managementtypen* definieren, die für einen bestimmten *Komponententyp* Managementoperationen für ein Managementfeature zur Verfügung stellen. Daher muss ein Managementtyp (i) den Namen des Features, für das es Managementoperationen anbietet, (ii) den Komponententyp, für den die Operationen implementiert sind, sowie (iii) mögliche Anforderungen der Operationen definieren. Ist eine Testoperation zum Beispiel in einem Skript definiert, das nur auf bestimmten Betriebssystemen ausgeführt werden kann, kann eine Liste der Anforderungen an dem Managementtyp angegeben werden. Werden keine besonderen Anforderungen innerhalb der Anwendung für die Ausführung der Managementoperation benötigt, beispielsweise wenn die Managementoperation als externer Service implementiert ist, sind die Anforderungen des entsprechenden Managementtyps undefiniert.

Definition 6.1 (Managementoperation – informell). Eine Managementoperation implementiert eine bestimmte Managementfunktionalität, zum Beispiel das Erstellen eines Backups oder das Ausführen eines Verfügbarkeitstests.

Definition 6.2 (Managementfeature – informell). Ein Managementfeature ist der Name einer Managementfunktionalität. Für ein Managementfeature können unterschiedliche Managementoperationen definiert sein.

Definition 6.3 (Managementtyp – informell). Ein Managementtyp beschreibt die Operationen und zusätzlichen Eigenschaften, um ein bestimmtes Managementfeature für einen Komponententyp zur Verfügung zu stellen. Er realisiert somit ein Managementfeature für einen Komponententyp.

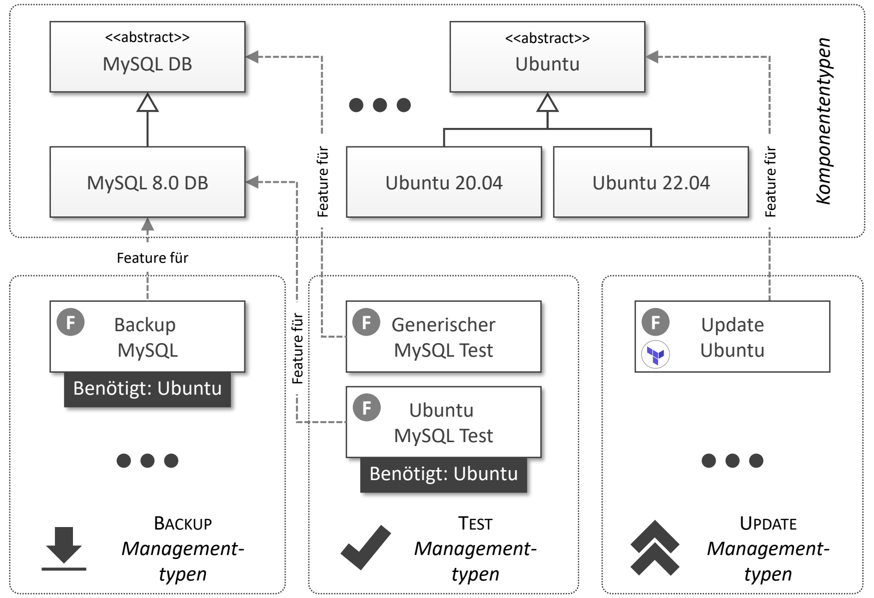


Abbildung 6.2: Beispiele für Managementtypen

In Abbildung 6.2 sind vier Managementtypen für die drei Managementfeatures `Backup`, `Test` und `Update` dargestellt. Auf der linken Seite befinden sich Managementtypen für das `Backup`-Feature. Der *Backup MySQL* Managementtyp stellt dabei eine Backup-Operation für den *MySQL 8.0 DB* Komponententyp zur Verfügung. Dabei wird jedoch ein Ubuntu-Betriebssystem für die Ausführung der Operationen benötigt. Ähnlich dazu sind zwei unterschiedliche Managementtypen für das `Test`-Feature dargestellt, wobei nur eines eine Ubuntu-Komponente in dem Anwendungsstack der Datenbank benötigt. Der zweite Managementtyp ist auch in anderen Konstellationen verwendbar. Darüber hinaus definiert der Managementtyp *Generischer MySQL Test* Testoperationen für alle Komponententypen, die von dem abstrakten *MySQL DB* Typ erben. Im Gegensatz dazu implementieren die Managementtypen *Backup MySQL* und *Ubuntu MySQL Test* das jeweilige Managementfeature spezifisch für eine MySQL Datenbank der Version 8.0.

Auf der rechten Seite in Abbildung 6.2 sind Managementtypen für das Update-Feature dargestellt. Der *Ubuntu Update* Managementtyp stellt, ähnlich wie der *Generische MySQL Test*, Managementoperationen für alle Komponententypen zur Verfügung, die von dem *Ubuntu* Komponententyp erben. Darüber hinaus ist am *Ubuntu Update* Managementtyp Terraform annotiert, dargestellt durch das Terraform-Logo. Dies bedeutet, dass die enthaltenen Managementoperationen mit Terraform umgesetzt wurden. Daher können diese Operationen nur dann zu einer Ubuntu Komponente hinzugefügt werden, wenn diese von Terraform verwaltet wird. Die Deployment-Technologie, die eine Komponente bereitstellen soll, kann entsprechend annotiert werden.

6.2.2 Automatisierte Managementanreicherung

Die Anreicherung von Deployment-Modellen mit zusätzlichen Managementfeatures ist in Algorithmus 6.1 definiert. Als Eingabe wird dem Algorithmus ein *ausführbares* Deployment-Modell t übergeben. Auf Basis der verwendeten Komponententypen wird in einem gegebenen Managementfeature-Repository nach verfügbaren Managementfeatures gesucht. Dafür wird für alle Komponenten des Deployment-Modells (Zeile 2) überprüft, ob Managementtypen existieren, die zusätzliche Operationen für den Typ der Komponente definieren (Zeile 3). Unterstützt ein Managementtyp das Managementfeature nur für bestimmte Deployment-Technologien, so muss die Komponente zusätzlich mit einer dieser Deployment-Technologien bereitgestellt und verwaltet werden (Zeilen 4-5). Dies wird benötigt, um zu verhindern, dass Änderungen, welche von zustandsändernden Managementoperationen verursacht werden, nicht von der Deployment-Technologie, die die Komponente möglicherweise überwacht, rückgängig gemacht werden. Wurden Managementtypen für eine Komponente identifiziert, müssen die Anforderungen des Managementtyps erfüllt werden, indem die nötigen Komponententypen im Stack identifiziert werden (Zeilen 7-18).

Um den Stack einer Komponente zu identifizieren, werden transitiv alle *Host-Komponenten* der aktuell untersuchten Komponente k_i identifiziert. Da der Host einer Komponente diese bereitstellt bzw. „*hosted*“, muss eine Relati-

Algorithmus 6.1 reichereAn $\left(t \in \mathcal{T}\right)$

1: $\mathit{features} := \emptyset$
2: **for all** $\left(k_i \in K_t\right)$ **do**
3: **for all** $\left(mgt_j \in MGT_t : \pi_{FeatureFür}(mgt_j) \in \mathit{typen}(k_i)\right.$
4: $\wedge\ (\pi_{Technologien}(mgt_j) = \perp\ \vee\ \exists dt_y \in \pi_{Technologien}(mgt_j) :$
5: $\left.\exists(\text{verwaltetVon}, dt_y) \in \mathit{eigenschaften}(k_i))\right)$ **do**
6: *// Identifiziere alle transitiven Host-Komponenten von* k_i
7: $\mathit{stack}_{k_i} := \emptyset$
8: $k_q = k_i$
9: **while** $\left(\exists k_h \in K_t\ :\left(\ \exists r_x \in R_t : \text{hostedOn} \in \mathit{typen}(r_x)\right.\right.$
10: $\left.\left.\wedge\ \pi_{Quelle}(r_x) = k_q \wedge \pi_{Ziel}(r_x) = k_h\right)\right)$ **do**
11: $\mathit{stack}_{k_i} := \mathit{stack}_{k_i}\ \cup\ \left\{k_h\right\}$
12: $k_q = k_h$
13: **end while**
14: *// Überprüfe, ob alle Anforderungen eines Features erfüllt sind*
15: **if** $\left(\pi_{Benötigt}(mgt_j) = \perp \vee\right.$
16: $\left.\forall req \in \pi_{Benötigt}(mgt_j)\ \exists k_x \in \mathit{stack}_{k_i} : req \in \mathit{typen}(k_x)\right)$ **then**
17: $\mathit{features} := \mathit{features} \cup \left\{(k_i, mgt_j)\right\}$
18: **end if**
19: **end for**
20: **end for**
21: $\mathit{gewählteFeatures} := \textsc{wähleFeatures}(\mathit{features})$
22: **for all** $\left(k_i \in K_t\ :\ \left(\exists(k_x, mgt_y) \in \mathit{gewählteFeatures} : k_i = k_x\right)\right)$ **do**
23: $\mathit{erweiteterTyp} := \mathit{typ}(k_i)$
24: $\pi_{Angereichert}(\mathit{erweiteterTyp}) := \mathit{wahr}$
25: **for all** $\left((k_x, mgt_y) \in \mathit{gewählteFeatures} : k_i = k_X\right)$ **do**
26: $\mathit{operationen}(\mathit{erweitertTyp}) :=$
27: $\mathit{operationen}(\mathit{erweitertTyp})\ \cup\ \mathit{operationen}(mgt_y)$
28: $\mathit{eigenschaften}(\mathit{erweiterterTyp}) :=$
29: $\mathit{eigenschaften}(\mathit{erweiterterTyp})\ \cup\ \mathit{eigenschaften}(mgt_y)$
30: **end for**
31: $\mathit{typ}(k_i) := \mathit{erweiteterTyp}$
32: $\mathit{eigenschaften}(k_i) := \mathit{eigenschaften}(k_i) \cup$
33: $\left\{(s, v_1) \in \mathit{eigenschaften}(\mathit{erweiterterTyp}) \mid \nexists(s, v_2) \in \mathit{eigenschaften}(k_i)\right\}$
34: **end for**

on des Typs *hostedOn* zwischen der Komponente und ihrem Host existieren. Dabei ist die Komponente die Quelle und der Host das Ziel der Relation (Zeilen 9-10). Existiert eine solche Host-Komponente k_h, wird diese zum Stack hinzugefügt (Zeile 11). Anschließend wird transitiv nach dem Host von k_h gesucht (Zeile 12), bis alle transitiven Host-Komponenten der Komponente k_i identifiziert wurden (Zeilen 9-13). Mithilfe des Stacks können im Anschluss die verfügbaren Managementtypen für die Komponente k_i identifiziert werden (Zeilen 15-17). Definiert ein Managementtyp keine besonderen Anforderungen (Zeile 15) oder sind die benötigten Komponententypen im Stack enthalten (Zeile 16), können die im Managementtyp enthalten Operationen zu der Komponente k_i hinzugefügt werden (Zeile 17). Anschließend können alle identifizierten Managementfeatures in der `wähleFeatures`-Methode entweder durch eine automatisierte oder manuelle Auswahl ausgewählt werden (Zeile 21). Dabei ist es möglich, die verfügbaren Managementfeatures für jede Komponente separat auszuwählen.

Nachdem die gewünschten Managementfeatures für die einzelnen Komponenten ausgewählt wurden, wird für jede Komponente ein entsprechender *angereicherte Komponententyp* generiert (Zeilen 22-33). Diese angereicherten Komponententypen beinhalten zusätzlich zu den Lebenszyklusoperationen auch die gewählten Managementfeatures in Form der benötigten Managementoperation. Um einen angereicherten Komponententypen zu erstellen, wird zunächst eine Kopie des aktuellen Komponententyps erstellt (Zeile 23). Diese wird anschließend als angereicherter Komponententyp durch eine entsprechende Eigenschaft markiert (Zeile 24). Da die Managementoperationen der gewählten Managementfeatures möglicherweise zusätzliche Eigenschaften benötigen, werden im nächsten Schritt alle Operationen und Eigenschaften der gewählten Managementtypen in den neuen angereicherten Komponententypen kopiert (Zeile 25-30). Dann wird der Komponente der neue Typ zugewiesen, welcher nun alle gewählten Managementfeatures beinhaltet. Um den möglicherweise neu hinzugefügten Eigenschaften des angereicherten Komponententyps Werte zuweisen zu können, müssen darüber hinaus die Eigenschaften der Komponente aktualisiert werden. Die Eigenschaften an einem Komponententyp definieren dabei, welche Eigenschaften

mit welchem Datentyp existieren, während eine Komponente diesen Eigenschaften konkrete Werte zuweist. Neue Eigenschaften des angereicherten Komponententyps müssen daher auf die Komponenten übertragen werden (Zeilen 28-29 & 32-33). Ist für eine Eigenschaft ein Standardwert definiert, wird dieser der Komponente zugewiesen. In einem manuellen Schritt können weitere Werte eingegeben oder Standardwerte überschrieben werden.

6.3 Generierung von Managementplänen

Mit der präsentierten Methode zur Managementanreicherung lassen sich beliebige Managementworkflows vollautomatisch generieren. Im Allgemeinen werden Workflows auch als *Pläne* bezeichnet, da sie alle Aktivitäten beschreiben, die in einer bestimmten Reihenfolge ausgeführt werden müssen. Im Folgenden wird dafür eine ähnliche Methode verwendet, wie sie von Breitenbücher et al. [BBK+14] vorgestellt wurde, um Provisionierungspläne anhand deklarativer Deployment-Modelle zu generieren. Anstatt jedoch die Bereitstellungslogik für eine Anwendung abzuleiten, werden Managementworkflows für bestimmte Managementfeatures generiert. Um beispielsweise die Bereitstellung zu testen, kann ein `Test`-Plan, der nur die Operationen des `Test`-Features ausführt, abgeleitet werden. Daher werden bestehende Ansätze kombiniert und adaptiert, wie das Testen der Bereitstellung [WBKL18] und das Einfrieren laufender Cloud-Anwendungen [HBKL19], um sie als unabhängig ausführbare Managementfunktionalitäten nutzen zu können.

Wie in Abbildung 6.1 dargestellt, wird für die Managementanreicherung der *Managementworkflowgenerator* eingeführt, der feature-spezifische Plug-ins benötigt, um imperative Workflows auf der Grundlage eines angereicherten Deployment-Modells zu generieren. Diese Plug-ins kapseln die feature-spezifische Logik, wie der resultierende imperative Workflow generiert wird. Ein Plug-in zur Generierung eines Testplans umfasst beispielsweise die Logik, in welcher Reihenfolge die Testoperationen ausgeführt werden. Zum Beispiel muss zunächst die Erreichbarkeit der Infrastruktur- und Plattformkomponenten sichergestellt sein, bevor die darauf bereitgestellten Komponenten

getestet werden können. Je nach Managementfeature können die Workflows die Managementoperationen dabei parallel oder sequenziell ausführen.

Um einen Managementworkflow zu generieren, analysiert ein Feature-Plug-in zunächst das angereicherte Deployment-Modell und prüft, ob die in dem Modell enthaltenen Komponententypen Managementoperationen definieren, die es verarbeiten kann. Werden unterstützte Managementoperationen gefunden, beginnt das Plug-in mit der Generierung entsprechender Workflowfragmente, die für jede Komponente die benötigten Operationen des angereicherten Managementfeatures ausführen. Dabei beinhaltet ein Plug-in jeweils die Logik, um für ein Managementfeature die Reihenfolge der entsprechenden Operationen zu generieren. Das `Test`-Plug-in prüft beispielsweise zuerst, ob ein gegebenes Deployment-Modell Testoperationen enthält. Anschließend ist es in der Lage, die Reihenfolge, in der die Testoperationen der einzelnen Komponenten ausgeführt werden müssen, zu generieren.

Da Managementoperationen meist Eingabeparameter benötigen, müssen die Eingabeparameter einer Operation durch passende Eigenschaftswerte der jeweiligen Komponenten, für die die Operation ausgeführt werden soll, im Deployment-Modell identifiziert werden. Dabei können die Eingabeparameter auch durch die Eigenschaften der zugrunde liegenden Infrastruktur, Plattform oder anderen Komponenten, wovon die aktuelle Komponente abhängt, erfüllt werden. Um beispielsweise eine Verbindung von einer Komponente zu einer Datenbank herzustellen, wird die IP-Adresse und der Port der Datenbank benötigt. Diese Eigenschaften müssen zur Herstellung der Verbindung an die entsprechende Operation zur Laufzeit übergeben werden. Daher enthalten die generierten Workflows nicht nur, in welcher Reihenfolge einzelne Managementoperationen ausgeführt werden müssen, sondern auch, wie die notwendigen Ein- und Ausgabeparameter aus dem Laufzeitmodell an die Managementoperationen übergeben werden.

Darüber hinaus können die Plug-ins auch bereits etablierte Managementansätze implementieren und kombinieren. Um beispielsweise Backups von verteilten Datenbanken und Systemen zu realisieren, wurden zahlreiche Konzepte entwickelt und publiziert [AGH+15; CFH+05; CL85; CSCM15; ELSC13; KR00]. In einem großen System mit diversen Komponenten und

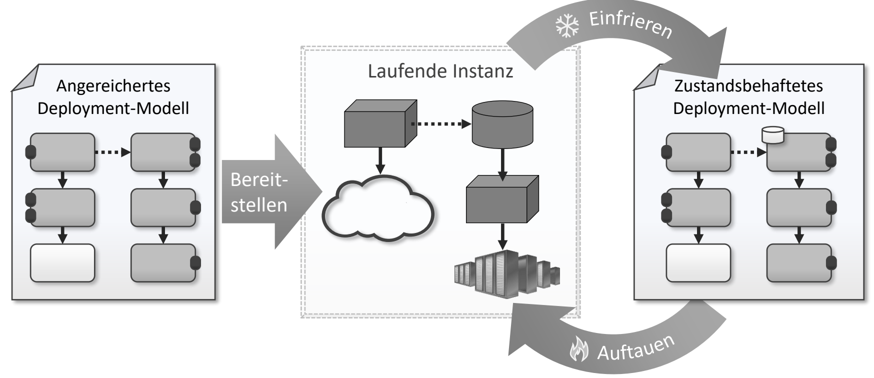


Abbildung 6.3: Einfrieren und Auftauen einer Anwendung

Technologien müssen jedoch meist mehrere solcher Ansätze kombiniert werden, um alle zustandsbehafteten Komponenten, zum Beispiel Datenbanken und virtuelle Maschinen, sichern zu können. Diese Kombination kann durch unterschiedliche Implementierungen der einzelnen Managementoperationen erreicht werden. Ein entsprechendes Backup-Feature-Plug-in ist dann in der Lage, einen Workflow zu generieren, der alle Backup-Operationen in der benötigten Reihenfolge und den notwendigen Parametern ausführen kann.

6.4 Management von zustandsbehafteten Anwendungen

Neben der musterbasierten Modellierung und Managementanreicherung sind zwei Schritte der LAUFFEUER-Methode das Einfrieren und Auftauen einer Anwendung (Schritte 3 und 5 in Abbildung 3.1). Im Folgenden wird daher die Umsetzung dieser zusätzlichen Managementfeatures vorgestellt. Die Idee des „Freeze and Defrost“-Konzepts ist in Abbildung 6.3 dargestellt. Für das Einfrieren und Auftauen werden, ebenso wie für die `Test`- und `Backup`-Features, Managementworkflows generiert. Dafür müssen in dem Managementworkflowgenerator Plug-ins integriert werden, die einen „*Freeze*“-Plan sowie einen „*Defrost*“-Plan generieren können.

6.4.1 Generierung von Freeze-Plänen

Im Gegensatz zum Backup-Feature werden beim Einfrieren einer Anwendung nicht nur die Daten gesichert, sondern die Komponenten werden danach zusätzlich heruntergefahren. Somit können im Allgemeinen für die Generierung des Freeze-Plans die Managementoperationen des Backup-Feature wiederverwendet werden. Da jedoch möglicherweise weitere oder andere Schritte notwendig sind, werden diese Operationen im Folgenden *Freeze-Operationen* genannt. Zusammen mit den Lebenszyklusoperationen, die für das Terminieren der Komponenten verwendet werden, kann das entsprechende Plug-in einen Workflow generieren, der den Zustand der Anwendung sichert und sie danach terminiert. Dabei wird im ersten Schritt der aktuelle Zustand jeder zustandsbehafteten Komponente der Anwendung gesichert und persistent abgelegt. Im Anschluss werden alle Instanzen aller Anwendungskomponenten gestoppt und heruntergefahren. Der letzte Schritt in einem Freeze-Plan ist schließlich die Erstellung eines *zustandsbehafteten Deployment-Modells*, das zusätzlich zu allen benötigten Artefakten für die Bereitstellung und dem Management der Anwendung auch die heruntergeladenen Zustände beinhaltet, wie in Abbildung 6.3 dargestellt.

Wie in Abbildung 6.4 dargestellt, läuft die Auswahl der Operationen, welche in einem Freeze-Plan ausgeführt werden sollen, in einem mehrstufigen Prozess ab. Dadurch wird sichergestellt, dass auch zustandsbehaftete Komponenten, für die keine Operationen zur Sicherung und Wiederherstellung ihres aktuellen Zustands bei der Managementanreicherung gefunden wurden, entsprechend gesichert und wiederhergestellt werden können. Dabei gilt die Annahme, dass Komponenten, die an einer tieferen Position eines Stacks vorkommen, die auf ihnen bereitgestellten Komponenten sichern und wiederherstellen können, wenn diese Freeze-Operationen definieren. Daher werden zunächst alle zustandsbehafteten Komponenten einer Anwendung identifiziert – z. B. über ihren Komponententyp oder Eigenschaftswerte, die sie als zustandsbehaftete Komponente ausweisen. In Abbildung 6.4 sind zustandsbehaftete Komponenten in dunkelgrau markiert. Im nächsten Schritt werden alle Komponenten herausgesucht, für die Operationen für das Einfrie-

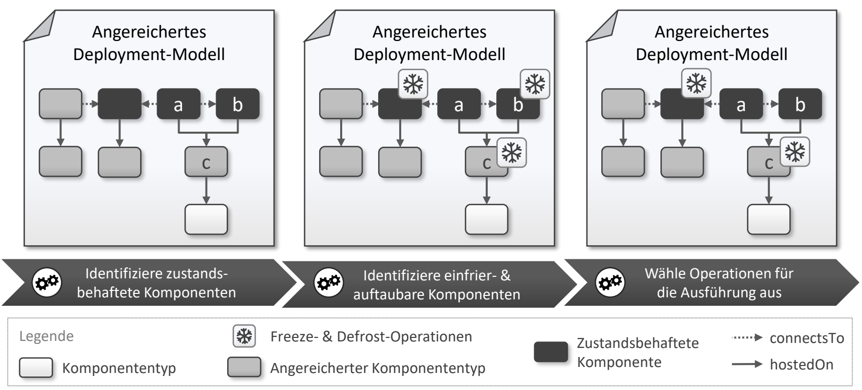


Abbildung 6.4: Auswahl der Freeze- und Defrost-Plan Operationen

ren im angereicherten Deployment-Modell vorhanden sind. Wie in der Mitte von Abbildung 6.4 dargestellt, können Komponenten, die selbst zustandslos, jedoch indirekt zustandsbehaftet sind, solche Operationen definieren, da sie zustandsbehaftete Komponenten bereitstellen können. Indirekt zustandsbehaftete Komponenten sind beispielsweise VMs oder PaaS-Angebote. Um zu vermeiden, dass der Zustand von zustandsbehafteten Komponenten verloren geht und dass Zustände nicht mehrfach gesichert werden, werden möglicherweise nicht alle Freeze-Operationen Teil des Freeze-Plans: In dem in Abbildung 6.4 dargestellten Fall wird die Freeze-Operation der zustandsbehafteten Komponente b nicht Teil des Freeze-Plans, da für die Komponente a keine Freeze-Operation existiert und die Komponenten a und b von c bereitgestellt werden. Im Plan wird daher lediglich die Freeze-Operation der Host-Komponente c ausgeführt, da c die auf ihr bereitgestellten Komponenten wiederherstellen kann. Dadurch ist sichergestellt, dass auch die zustandsbehaftete Komponente a, die keine Freeze-Operation definiert, wiederhergestellt werden kann. Existiert für a eine Freeze-Operation, wird die Freeze-Operation der unterliegenden Komponente c nicht ausgeführt.

Nachdem alle Daten aus zustandsbehafteten Komponenten einer Anwendung gesichert wurden, werden die Zustandsartefakte den jeweiligen Komponenten im Deployment-Modell der Anwendung hinzugefügt. Damit wird

das Deployment-Modell zu einem *zustandsbehafteten Deployment-Modell*, das für die Wiederherstellung der Anwendung verwendet werden kann. Anschließend werden alle anderen Komponenten gestoppt und heruntergefahren.

6.4.2 Generierung von Defrost-Plänen

Um eine Anwendung im eingefrorenen Zustand wiederherstellen zu können, muss ein Defrost-Plan generiert werden, der sowohl die Komponenten instanziiert und konfiguriert als auch die gespeicherten Zustände wiederherstellen kann. Ebenso wie bei der Bereitstellung einer Anwendung werden dabei die Lebenszyklusoperationen einer Komponente verwendet, um diese zu installieren, zu konfigurieren und zu starten. Hat eine Komponente jedoch einen Zustand, so muss anstelle der Lebenszyklusoperationen eine *Defrost-Operation* verwendet und im Plan ausgeführt werden. Diese Operationen werden durch `Defrost`-Managementtypen zur Verfügung gestellt und realisieren, wie der Zustand der entsprechenden Komponente wiederhergestellt werden kann. Für die Wiederherstellung müssen je nach Komponententyp unterschiedliche Schritte ausgeführt werden, die von den Defrost-Operationen umgesetzt werden müssen. Um zum Beispiel das Abbild einer virtuellen Maschine wiederherzustellen, muss lediglich das Abbild in den Hypervisor geladen und gestartet werden. Im Gegensatz dazu muss eine Datenbank zuerst angelegt werden, bevor die Daten wiederhergestellt werden können. Wie auch für den Freeze-Plan gilt dabei die Annahme, dass unterliegende Komponenten immer auch die auf ihnen bereitgestellten Komponenten beinhalten. Daher werden für Komponenten, die auf einer wiederhergestellten Komponente bereitgestellt werden, lediglich die Konfigurations- und Startoperationen ausgeführt. Dadurch wird sichergestellt, dass die Komponenten nach der Wiederherstellung auch gestartet und neue Laufzeitinformationen wie IP-Adressen entsprechend aktualisiert und konfiguriert werden. Welche dieser Operationen ausgeführt werden müssen, wird analog zu der Auswahl der Freeze-Operationen ausgewählt. Die in Abbildung 6.4 dargestellten und im vorherigen Abschnitt beschrieben Schritte werden für die Generierung von Defrost-Plänen in ähnlicher Weise durchgeführt.

Wie in Abbildung 6.3 dargestellt, können lediglich zustandsbehaftete Deployment-Modelle mit einem Defrost-Plan wiederhergestellt werden, da ansonsten keine Zustandsartefakte in dem Deployment-Modell enthalten sind, die wiederhergestellt werden können. Im Gegensatz dazu kann jedoch eine zustandsbehaftetes Deployment-Modell auch ohne das Wiederherstellen der Zustände bereitgestellt werden, indem der Provisionierungsplan bzw. Build-Plan anstatt des Defrost-Plans ausgeführt wird.

6.5 Diskussion

Das vorgestellte Konzept zur Anreicherung von Managementfunktionalitäten und zur Generierung von Managementworkflows ermöglicht die wiederverwendbare Definition von Managementfeatures, die für eine gesamte, über mehrere Umgebungen verteilte, Anwendung erweitert und ausgeführt werden können. Anstatt jede Funktionalität manuell implementieren zu müssen, können Modellierende auswählen, welche Managementfeatures sie für eine Anwendung benötigen, woraufhin diese automatisiert zu der Anwendung hinzugefügt werden. Die Managementworkflows werden vollautomatisch generiert, sodass manuelle Anpassungen der Managementfunktionalitäten, die bei Änderungen einer Anwendung für manuell erstellte Managementaufgaben notwendig wären, nicht erforderlich sind.

Um neue Managementfunktionalitäten bereitzustellen, müssen jedoch entsprechende Managementtypen bereitgestellt und die Workflowgenerierung jeweils um das entsprechendes Plug-in erweitert werden. Dabei muss das Workflow-Plug-in in der Lage sein, ausführbare Workflows anhand neuer oder bestehender Operationen zu generieren. Für die Implementierung neuer Managementtypen und deren -operationen werden Entwickelnde mit Erfahrung im Umgang mit den Komponententypen, für die die Operationen umgesetzt werden sollen, benötigt. Sind die Managementtypen und das Feature-Plug-in jedoch einmal umgesetzt und öffentlich verfügbar, können neue und bereits existierende Deployment-Modelle automatisch mit dem neuen Managementfeature angereichert werden.

Damit eine Anwendung im Rahmen der LAUFFEUER-Methode eingefroren und wieder aufgetaut werden kann, müssen die folgenden Voraussetzungen erfüllt sein: (i) Eine zustandsbehaftete Komponente muss Freeze- und Defrost-Operationen definieren, die in der Lage sind, den Anwendungszustand der Komponente abzurufen und wiederherzustellen. Wenn für eine zustandsbehaftete Komponente keine Freeze- und Defrost-Operationen definiert sind, muss eine darunter liegende Komponente diese Komponenten einfrieren können. Ansonsten würde der Zustand dieser Komponente beim Herunterfahren der Anwendung verloren gehen. Wenn im gesamten Stack einer zustandsbehafteten Komponente keine Freeze- und Defrost-Operationen gefunden werden, ist die gesamte Anwendung nicht zum Einfrieren und Auftauen geeignet. (ii) Zum Zeitpunkt, an dem das Einfrieren einer Anwendung ausgelöst wird, wird angenommen, dass sich die Anwendung im Leerlauf befindet. Andernfalls können externe Anfragen den Zustand der Anwendung ändern und zu einem inkonsistenten Zustand der Daten führen. Daher ist die Methode nicht geeignet, um nur eine Momentaufnahme des aktuellen Anwendungsstatus zu erstellen, sondern ermöglicht es seinen Nutzern, eine Anwendung einzufrieren und aufzutauen, um Ressourcen und Geld zu sparen, wenn die Anwendung nicht genutzt wird. Werden jedoch bekannte Backup-Verfahren, wie zum Beispiel Checkpointing [CSCM15; KR00] verwendet, kann dieses Problem umgangen werden.

Ähnlich zu der in diesem Kapitel präsentierten Methode führen Calcaterra und Tomarchio [CT22] eine Methode ein, um Anwendungen automatisiert verwalten zu können. Dabei werden, wie auch in dieser Methode, für die jeweiligen Managementfunktionalitäten Workflows generiert. Allerdings ist die von Calcaterra und Tomarchio vorgeschlagene Methode auf Anwendungen limitiert, die in TOSCA modelliert wurden und basiert auf der Annotation der gewünschten Managementfeatures. Im Gegensatz dazu kann die hier vorgestellte Methode auch für andere Deployment-Technologien angewendet werden, sofern ihre Modelle auf EDMM abbildbar sind. Darüber hinaus kann die Methode nachträglich auf laufenden Anwendungen angewendet werden, die mit mehreren Deployment-Technologien bereitgestellten wurden. Die entsprechende Erweiterung wird im folgenden Kapitel 7 gezeigt.

Kapitel 7

Dynamisierung des Managements laufender Anwendungen

Die Lauffeuer-Methode ermöglicht es nicht nur, neu modellierte Anwendungen mit zusätzlichen Managementfunktionalitäten zu erweitern, sondern kann auch nachträglich auf laufende Anwendungen angewendet werden. Dafür wird in diesem Kapitel eine Methode vorgestellt, die von einer laufenden Anwendung ein Instanzmodell ableitet und mithilfe der Managementanreicherung Workflows generieren kann, die auf der sich bereits im Betrieb befindlichen Anwendung ausgeführt werden können. Dafür wird im nachfolgenden Abschnitt 7.1 zunächst die Konzeptidee und -übersicht präsentiert, während der Abschnitt 7.2 auf die Details der Methodenschritte eingeht. Schließlich wird in Abschnitt 7.3 die vorgestellte Methode zum Dynamisieren des Managements laufender Anwendungen diskutiert.

Die Methode wurde bereits auf der CLOSER Konferenz [HBB+21] und im Springer Nature Computer Science Journal [HBBL23] veröffentlicht.

7.1 Konzeptidee und -übersicht

Durch die im vorherigen Kapitel 6 vorgestellte Methode zur Managementanreicherung können verteilte Anwendungen automatisiert mit Managementfunktionalitäten erweitert und verwaltet werden. Auch Hybrid- und Multi-Cloud-Anwendungen werden unterstützt. Solche Anwendungen werden jedoch meist aufgrund ihrer Komplexität und Größe von unterschiedlichen Teams implementiert und betrieben. Dabei werden meist auch verschiedene Deployment-Technologien für die einzelnen Teile einer Anwendung eingesetzt [DMPS17; GGTP19]. Daher fehlt meist eine Gesamtübersicht über die verteilte Anwendung. Dies erschwert das Management und Ausführen von Managementfunktionalitäten für die gesamte Anwendung erheblich. Deshalb wird in diesem Kapitel ein Konzept vorgestellt, mit dem laufende Anwendungen, die mit mehreren unterschiedlichen Deployment-Technologien bereitgestellt wurden, ganzheitlich verwaltet werden können. Eine Übersicht des daraus resultierenden *Integrierten Managementsystems für zusammengesetzte Anwendungen* ist in Abbildung 7.1 dargestellt.

Um eine bereits bereitgestellte und sich im Betrieb befindliche Anwendung nachträglich verwalten zu können, muss zunächst ein ganzheitliches *Instanzmodell* der Anwendung erstellt werden. Mithilfe eines solchen Instanzmodells können anschließend verfügbare Managementfunktionalitäten gesucht und entsprechende Managementworkflows generiert werden. Anschließend kann die Anwendung durch das Ausführen der Workflows verwaltet werden. Dabei werden *zustandserhaltende* und *zustandsändernde* Managementfunktionalitäten unterschieden. Da einige Deployment-Technologien die von ihnen bereitgestellten Anwendungen überwachen, müssen die Deployment-Technologien über Zustandsänderungen der Anwendung informiert werden. Damit wird verhindert, dass die Deployment-Technologie Änderung an einer Komponente wieder rückgängig macht. Daher müssen die zugrundeliegenden Deployment-Technologien jeder Komponente bei der Managementanreicherung berücksichtigt und von zustandsändernden Managementoperationen unterstützt werden. Zustandserhaltende Operationen können hingegen ohne Einschränkungen erweitert und von Workflows ausgeführt werden.

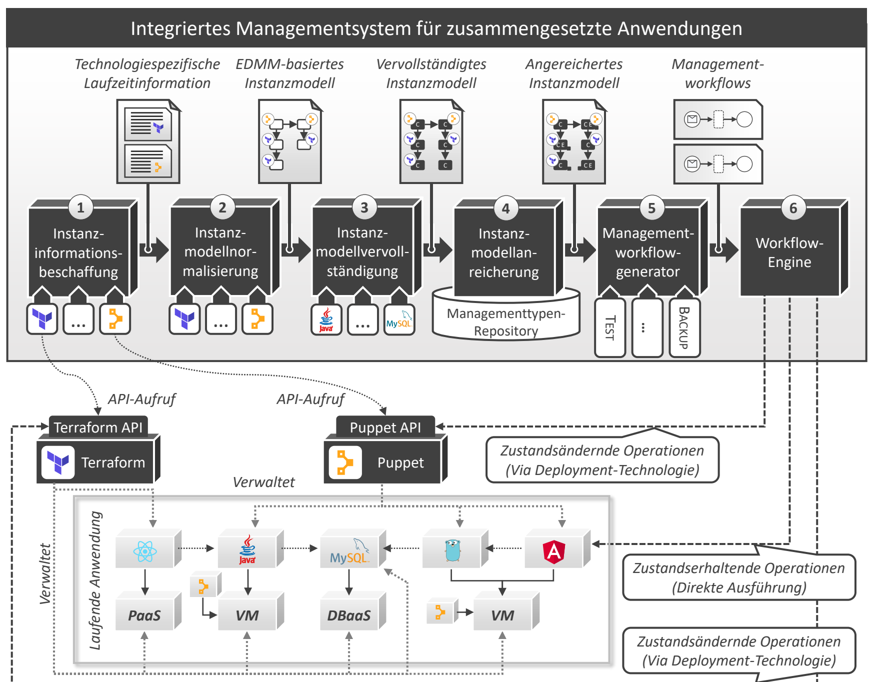


Abbildung 7.1: Übersicht über das Integrierte Managementsystem für zusammengesetzte Anwendungen, adaptiert von [HBBL23]

Der erste Schritt in der Ableitung eines Instanzmodells einer laufenden Anwendung ist in Abbildung 7.1 durch ①, der *Instanzinformationsbeschaffung*, dargestellt. Dabei werden durch deployment-technologiespezifische Plug-ins Informationen über eine Anwendung direkt von den Deployment-Technologien bezogen. Wie in Abbildung 7.1 gezeigt, kann beispielsweise ein Terraform-Plug-in direkt mit der API einer Terraform-Instanz interagieren, während ein Puppet-Plug-in mit einem Puppet „*Primary Server*" kommuniziert. Dadurch können aktuelle, jedoch deployment-technologiespezifische Laufzeitinformationen einer Anwendung gewonnen werden.

In Schritt ② werden die deployment-technologiespezifischen Laufzeitinformationen durch die *Instanzmodellnormalisierung* in ein EDMM-basiertes

Instanzmodell umgewandelt. Wie auch bei der Instanzinformationsbeschaffung interpretieren deployment-technologiespezifische Plug-ins die Informationen und generieren ein EDMM-kompatibles Instanzmodell. Jedes Normalisierungs-Plug-in muss dabei in der Lage sein, die abgerufenen Instanzinformationen zu verarbeiten, d. h. (i) bereitgestellte Komponenten zu identifizieren, (ii) ihre Typen zu erkennen und sie auf entsprechende Komponententypen abzubilden sowie (iii) die, durch den Komponententyp definierten, Eigenschaften einer Komponente mit den aktuellen Werten aus der deployment-technologiespezifischen Laufzeitinformationen zu füllen. Als Resultat wird ein *EDMM-basiertes Instanzmodell* der Anwendung generiert.

Die Menge und die Qualität der deployment-technologiespezifischen Laufzeitinformationen unterscheidet sich von Technologie zu Technologie erheblich. Daher sind die daraus resultierenden Instanzmodelle meist nicht vollständig. Aus diesem Grund wird das EDMM-basierte Instanzmodell im Schritt ③ durch die *Instanzmodellvervollständigung* analysiert. Dabei können durch komponentenspezifische Plug-ins (i) fehlende Komponenten identifiziert, (ii) Komponententypen konkretisiert, (iii) fehlende Eigenschaftswerte gefunden oder (iv) horizontale Relationen zwischen Komponenten abgeleitet werden. Als Ergebnis dieser Vervollständigung wird aus dem EDMM-basierten Instanzmodell der Anwendung ein *vervollständigtes Instanzmodell*.

Ab dem nächsten Schritt ④, der *Instanzmodellanreicherung*, wird die Methode aus Kapitel 6 wiederverwendet und erweitert. Dabei ist eine Unterscheidung der Managementfunktionalitäten in zustandserhaltende und zustandsändernde unerlässlich. Wie bereits erwähnt, müssen zustandsändernde Managementoperationen die Deployment-Technologie berücksichtigen, die die betreffende Komponente bereitgestellt hat. Andernfalls könnten Änderungen an einer Komponente, die durch das Ausführen einer Managementfunktionalität hervorgerufen wurde, wieder rückgängig gemacht werden. Daher müssen sowohl die Komponenten als auch Managementtypen, die Operationen definieren, die den Zustand einer Komponente ändern, mit den Deployment-Technologien annotiert sein, die sie bereitgestellt haben bzw. unterstützen. Während der Auswahl der Managementfeatures können dadurch nicht kompatible Implementierungen automatisiert gefiltert werden.

Wurde das Instanzmodell durch die Managementanreicherung erweitert, wird das *vervollständigte und angereicherte Instanzmodell* an den Managementworkflowgenerator in Schritt ⑤ übergeben. Dabei werden durch die Feature-Plug-ins Managementworkflows für jedes Managementfeature generiert. Zur automatisierten Ausführung werden die Workflows anschließend in Schritt ⑥ in einer *Workflow-Engine* bereitgestellt. Wird ein Managementworkflow ausgeführt, müssen zustandsändernde Operationen die Deployment-Technologien über die Änderungen der betreffenden Komponente über deren API informieren, während zustandserhaltende Operationen ohne Einschränkungen direkt mit allen Komponenten interagieren können.

7.2 Ableiten von Instanzmodellen

Für das nachträglich Erweitern einer Anwendung um Managementfunktionalitäten, wird zunächst ein Instanzmodell der Anwendung benötigt. Einerseits kann ein bereits existierendes Instanzmodell einer Anwendung wiederverwendet werden, andererseits kann das Modell auch automatisiert erstellt werden. Zur Erstellung von Instanzmodellen existieren zahlreiche Methoden und Prozesse [BBKL13; Bin15; HBLE14; MDW+00]. Allerdings berücksichtigt keine der existierenden Methoden die zum Deployment der Anwendung verwendeten Technologien. Deshalb werden, wie in Abbildung 7.1 gezeigt, die Deployment-Technologien herangezogen, um Laufzeitinformationen über eine laufende Anwendungsinstanz abzufragen und ein Instanzmodell abzuleiten. Anschließend werden in Schritt ③ existierende Methoden eingesetzt, um das von den Deployment-Technologien abgeleitete Instanzmodell mit fehlenden und manuell bereitgestellten Komponenten und Details zu vervollständigen. Die Schritte werden im Folgenden beschrieben.

7.2.1 Schritt 1: Instanzinformationsbeschaffung

Anwendungen, die aus zahlreichen Komponenten bestehen, sind typischerweise mit mehreren Deployment-Technologien bereitgestellt [WBB+21]. Um die Laufzeitinformationen, die die Deployment-Technologien über eine An-

wendung haben, zu bekommen, müssen daher alle involvierten Deployment-Technologien abgefragt werden. Die in Schritt ① genutzte *Instanzinformationsbeschaffung* verwendet darum eine Plug-in-Architektur, in der jede unterstützte Deployment-Technologie durch ein Plug-in realisiert wird. Um die Plug-ins aufrufen zu können, wurde eine einheitliche API definiert, die jedes Plug-in umsetzen muss. Dabei realisiert ein solches Deployment-Technologie-Plug-in die Funktionalität, um Informationen über die von der entsprechenden Deployment-Technologie verwalteten Komponenten abzufragen. Die Umsetzungen der einzelnen Plug-ins können dabei unterschiedlich aussehen. Zum Beispiel verwendet das Terraform-Plug-in die sogenannte *State*-Datei, um die Laufzeitinformationen einer mit Terraform verwalteten Anwendung, abzurufen, während das Puppet-Plug-in diese Informationen über die Puppet-API abruft[1]. Die einheitliche Plug-in-API definiert darüber hinaus auch Methoden, um Informationen über die Deployment-Technologie selbst, wie Zugangsdaten für den Zugriff auf die Deployment-Technologie, abrufen zu können. Die in diesem Schritt von den Plug-ins abgerufenen *technologiespezifischen Laufzeitinformationen* werden anschließend in Schritt ② in einem *EDMM-basierten Instanzmodell* zusammengefasst.

7.2.2 Schritt 2: Instanzmodellnormalisierung

Für die Erstellung eines ganzheitlichen Instanzmodells einer Anwendung werden im Schritt ② alle deployment-technologiespezifischen Instanzmodelle in ein technologieagnostisches Instanzmodell zusammengeführt. Als technologieagnostisches Instanzmodell wird ebenfalls EDMM verwendet. Daher werden in der *Instanzmodellnormalisierung* aus allen deployment-technologiespezifischen Instanzmodellen zunächst ein EDMM-Modell je Deployment-Technologie abgeleitet. Diese EDMM-Modelle je Deployment-Technologie werden dann in ein ganzheitliches Modell zusammengeführt.

Bei der Zusammenführung der einzelnen Instanzmodelle müssen Komponenten, die in zwei oder mehr Instanzmodellen vorkommen, identifiziert und vereinigt werden. Dies ist beispielsweise der Fall, wenn eine Kompo-

[1] Insgesamt wurden vier Deployment-Technologie-Plug-ins umgesetzt (s. Abschnitt 8.3).

nente a von einer Deployment-Technologie verwaltet wird und eine andere Komponente b, die von einer zweiten Deployment-Technologie verwaltet wird, eine Abhängigkeit zu a hat. Um die Abhängigkeit der Komponente b verwalten zu können, kann das deployment-technologiespezifische Instanzmodell der zweiten Deployment-Technologie die Komponente a ebenfalls beinhalten, ohne diese jedoch zu verwalten. Damit die Komponente a nicht doppelt in dem ganzheitlichen Instanzmodell der Anwendung auftaucht, ist eine Ähnlichkeitsprüfung der Komponenten erforderlich. Diese verwendet die Eigenschaften und Komponententypen der entsprechenden Komponenten. Dabei können sich die Komponententypen jedoch unterscheiden, da eine Deployment-Technologie möglicherweise mehr Informationen über eine Komponente bereithält als eine andere. Um zum Beispiel zu erkennen, dass zwei Komponenten in zwei verschiedenen Modellen dieselbe VM-Instanz darstellen, kann ihre öffentliche IP-Adresse als eindeutige Eigenschaft verwendet werden, auch wenn sie beispielsweise einmal konkret als Ubuntu 22.04 Instanz und einmal als generische VM identifiziert wurde. Ebenso kann herausgefunden werden, dass zwei von unterschiedlichen Deployment-Technologien bereitgestellte Komponenten eine Verbindung zu derselben Datenbankinstanz herstellen: Stimmt der Datenbankname und die Adresse überein, verwenden beide dieselbe Instanz. Dabei gilt die Annahme, dass eine Komponente von genau einer Deployment-Technologie verwaltet wird und jede andere Deployment-Technologie lediglich Verbindungen zu der laufenden Komponenteninstanz herstellt. Verwaltet eine Deployment-Technologie eine Komponente, wird sie mit einer Annotation versehen, der sie als *verwaltet von* dieser Deployment-Technologie kennzeichnet. Andernfalls wird sie als *identifiziert von* der entsprechenden Deployment-Technologie annotiert, um anzugeben, dass sie von der Technologie zwar erkannt, jedoch nicht von ihr verwaltet wird. Die *identifiziert von* Anmerkung dient dabei lediglich der Nachvollziehbarkeit, wie welche Komponente erkannt wurde. Als Ergebnis wird ein *EDMM-basiertes Instanzmodell* der laufenden Anwendung generiert.

Zusätzlich zu dem abgeleiteten Instanzmodell werden auch die Zugangsinformationen zu den einzelnen Deployment-Technologien dem Deployment-Modell hinzugefügt. Diese werden benötigt, um zustandsändernde Manage-

mentoperationen ausführen zu können. Dafür werden für jede Deployment-Technologie die technologiespezifischen Zugangsdaten, beispielsweise Endpunkte, Passwörter, etc., an das EDMM-basierte Instanzmodell angehängt.

7.2.3 Schritt 3: Instanzmodellvervollständigung

Der Detailgrad der Instanzmodelle, die aus den Deployment-Technologien abgerufen werden können, unterscheidet sich erheblich. Während in Kubernetes und in Terraform hauptsächlich Informationen über die Infrastruktur, zum Beispiel VMs und Container, abgeleitet werden können, liefern Chef und Puppet auch Informationen über einzelne Softwarekomponenten. Allerdings ist es nicht immer möglich alle Informationen über eine Komponente abzuleiten, da beispielsweise Konfigurationsparameter oder Versionen nicht immer in den Instanzmodellen der Deployment-Technologien enthalten sind.

Um diese Lücken zu füllen, wird in Schritt ③ das Instanzmodell vervollständigt. Die *Instanzmodellvervollständigung* verarbeitet dabei das EDMM-basierte Instanzmodell und identifiziert iterativ zusätzliche Eigenschaften der Komponenten. So können unter anderem (i) bisher nicht identifizierte Komponenten gefunden, (ii) die Typen bereits identifizierter Komponenten verfeinert, (iii) fehlende Eigenschaftswerte gefüllt oder (iii) horizontale Beziehungen zwischen Komponenten erkannt werden. Um dies zu erreichen, wird die von Binz et al. [BBKL13; Bin15] eingeführte Crawler-Methode für das Erstellen von Instanzmodellen eingesetzt und erweitert: Dabei werden komponentenspezifische *Plug-ins*[1] nacheinander ausgeführt, bis kein Plug-in mehr neue Informationen über eine Anwendung zu einem Instanzmodell hinzufügen kann. Zum Beispiel kann ein MySQL-Plug-in die Version einer auf einer VM installierten MySQL Datenbank identifizieren, während ein zweites Plug-in erkennen kann, dass eine Java-Anwendung eine Verbindung zu dieser Datenbank herstellt und wo sich diese Datenbank befindet. Das zweite Plug-in ist somit in der Lage, Relationen vom Typ *connectsTo* zwischen Java-Komponenten und Datenbankkomponenten zu identifizieren und dem Instanzmodell hinzuzufügen.

[1] Zur Validierung wurden 12 Technologie-Plug-ins umgesetzt (s. Abschnitt 8.3).

Um die Plug-ins zu identifizieren, die in der Lage sind, mehr Details über die Anwendung zu erkennen, wurden die von Binz et al. [BBKL13; Bin15] vorgestellten Konzepte erweitert: Während die Plug-ins in der ursprünglichen Variante lediglich eine Liste an Komponententypen definieren, die diese möglicherweise verfeinern können, können Plug-ins in der hier vorgestellten Methode komplexere *Detektoren* definieren. Jedes Plug-in kann dabei mehrere Detektoren angeben, d. h. Topologiegfragmente, die eine Konstellation von Komponenten definieren, die das Plug-in konkretisieren kann. Ähnlich zu den Verfeinerungsmodellen (s. Abschnitt 5.4) können ausführbare Plug-ins identifiziert werden, indem ein Subgraph eines Instanzmodells dem Detektor eines Plug-ins entspricht. Wie bei den PRMs, entspricht ein Subgraph des Instanzmodells einem Detektor eines Plug-ins, wenn die Modellknoten und Relationen nach dem Entsprechungsoperator (s.Definition 5.11) übereinstimmen. Dadurch können Plug-ins gezielter eingesetzt werden. Zum Beispiel ist ein Tomcat-Plug-in in der Lage (i) einen Tomcat Webserver auf einer VM zu identifizieren, (ii) eine Komponente des abstrakten Komponententyps *Webserver* zu einem konkreten Tomcat Komponententyp zu verfeinern, (iii) die konkrete Version einer Tomcat Komponente zu identifizieren und (vi) den Kontextpfad und den Port einer auf einem Tomcat bereitgestellten Webanwendung zu ermitteln. Daher definiert das Plug-in vier Detektoren, wobei (i) der erste eine Komponente eines abstrakten VM Komponententyps enthält, (ii) der zweite Detektor eine Komponente des Komponentettyps *Webserver*, (iii) der dritte eine Komponente des Typs *Tomcat* ohne Version, während (vi) der letzte Detektor eine Komponente des Typs *Web-App* definiert, die auf einem Tomcat-Webserver gehostet wird. Wurde eine Komponente konkretisiert, wird sie von dem Plug-in entsprechend annotiert, um anzugeben, woher Details über die Komponente kommen.

Ein Beispiel eines vervollständigen Instanzmodells ist in Abbildung 7.2 dargestellt. Es beschreibt eine Anwendung, die mit Kubernetes, Terraform und Puppet bereitgestellt wurde. Die Anwendung ist eine Webanwendung, die über ein Java-Backend Daten aus einer MySQL-Datenbank abruft. Auf der linken Seite in Abbildung 7.2 befindet sich eine Webanwendung vom Typ *NodeJS Web-App*, welche auf einem *NodeJS 10*-Webserver in einem *Alpine*

Container installiert ist. Das Zahnrad mit dem Kubernetes-Logo an der linken oberen Ecke der *Alpine Container*-Komponente gibt an, dass Kubernetes diesen Container verwaltet. Im Gegensatz dazu ist die darunterliegende *Docker-Engine* zwar mit einem Kubernetes-Logo annotiert, jedoch nicht mit einem Zahnrad, da Kubernetes selbst die *Docker-Engine* nicht verwaltet, sie aber im Instanzmodell enthalten ist. Dementsprechend ist die *Docker-Engine*-Komponente lediglich mit einem runden *identifiziert von Kubernetes*-Icon annotiert. Darüber hinaus sind die Softwarekomponenten, die in einem Container installiert sind, Kubernetes nicht bekannt. Deshalb sind der *NodeJS 10*-Webserver und die *NodeJS Web-App* nicht Teil des Instanzmodells, das von Kubernetes abgerufen wurde und bis Schritt ③ nicht Teil des Instanzmodells. Erst durch Plug-ins der Instanzmodellvervollständigung wurden der Webserver und die Web-App identifiziert. Komponenten, die nur durch ein Plug-in identifiziert wurden, sind ausschließlich mit einer Lupe annotiert.

Das Backend der laufenden Anwendung sowie die dazugehörige Datenbank wurde von Puppet auf einer *Ubuntu 22.04* VM bereitgestellt, während die VM selbst von Terraform verwaltet wird. In Abbildung 7.2 ist dies durch die annotierten Zahnräder mit den entsprechenden Logos an den Komponenten dargestellt. Ähnlich wie die *Docker-Engine* wurde die *OpenStack*-Umgebung zwar durch Terraform identifiziert, jedoch verwaltet Terraform diese nicht. Da die Ubuntu VM von Terraform bereitgestellt, von Puppet als laufende Instanz verwendet und von einem Plug-in mit mehr Informationen angereichert wurde, ist die Komponente mit drei Annotationen gekennzeichnet. Im Gegensatz dazu wurden alle Eigenschaften der *MySQL 8.0 DBMS* und *Java 11* Komponenten direkt durch Puppet identifiziert. Somit tragen die Komponenten lediglich die *veraltet von Puppet* Annotation.

Die Datenbank, wie auch das Backend, sind von Puppet verwaltet und wurden durch weitere Plug-ins konkretisiert. Der Port, auf dem das Backend auf Anfragen des Frontends wartet, wurde beispielsweise durch ein Java-Plug-in identifiziert, während der Datenbankname der Datenbank durch ein MySQL-Plug-in erkannt wurde. Die Relation zwischen dem Backend und der Datenbank kann automatisiert von einem Plug-in anhand von Konfigurationsdateien herausgefunden werden. Allerdings können möglicherweise nicht

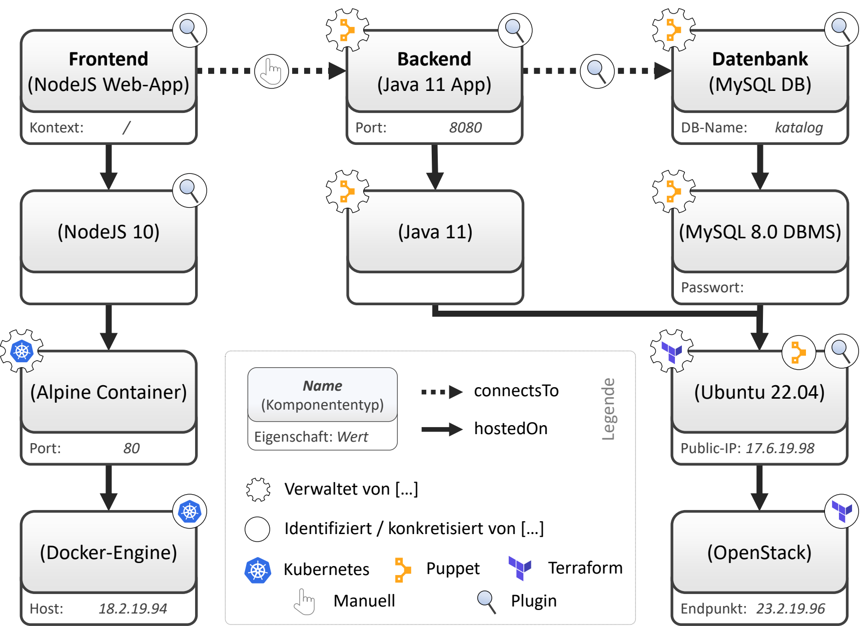


Abbildung 7.2: Ein Beispiel für ein vervollständigtes Instanzmodell einer laufenden Anwendung

alle Eigenschaften und horizontalen Relationen automatisiert herausgefunden werden, da diese meist komponentenspezifisch sind und nicht generisch identifiziert werden können. Betreibende und Entwickelnde können jedoch das Instanzmodell im Nachgang durch einen manuellen Schritt um eventuell fehlende Relationen, Eigenschaften oder sogar Komponenten vervollständigen. In dem hier gezeigten Beispiel konnte die Relation zwischen dem Frontend und dem Backend nicht automatisiert gefunden werden und wurde manuell hinzugefügt, wie die annotierte Hand beschreibt.

7.2.4 Schritte 4-6: Managementanreicherung und -ausführung

Die Anreicherung der vervollständigten Instanzmodelle mit verfügbaren Managementfunktionalitäten folgt demselben Prozess wie für Deployment-

Modelle (s. Kapitel 6). Zunächst werden, wie in Schritt ④ in Abbildung 7.1 dargestellt, die verfügbaren Managementtypen identifiziert und entsprechend der verwendeten Deployment-Technologie sowie weiteren Anforderungen gefiltert. Anschließend können die gewünschten Managementfeatures je Komponente gewählt und automatisiert angereichert werden. Das *angereicherte Instanzmodell* wird daraufhin in Schritt ⑤ vom Managementworkflowgenerator verarbeitet, wobei *Managementworkflows* für jedes Managementfeature generiert werden. Schließlich werden die Workflows in einer Workflow-Engine, wie in Schritt ⑥ dargestellt, zur Ausführung bereitgestellt.

Zur Ausführung von zustandsändernden Managementoperationen einer Komponente werden die Zugangsdaten für die Deployment-Technologie benötigt, die die Komponente verwaltet. Dies ist nötig, um die Technologien entsprechend über die Änderungen zu informieren oder die Änderungen direkt darüber auszuführen. Da zustandsändernde Managementoperationen direkt mit den Deployment-Technologien interagieren muss, müssen die notwendigen Zugangsdaten als Eingabeparameter der Operation definiert sein. Dadurch kann der Managementworkflowgenerator die benötigten Informationen aus dem Instanzmodell auslesen und an die jeweilige Managementoperation übergeben. Zur Ausführung zustandserhaltender Operationen werden diese nicht benötigt, wodurch sie direkt mit den Komponenten der Anwendung interagieren können.

7.3 Diskussion

Die vorgestellte Methode ermöglicht es, laufende Anwendungen nachträglich zu verwalten. Dies wird erreicht, indem Instanzmodelle der Anwendungen erstellt und mit zusätzlichen Managementfunktionalitäten angereichert werden. Dabei wird davon ausgegangen, dass eine Komponente von genau einer Deployment-Technologie verwaltet wird. Um zustandsändernden Managementoperationen auszuführen, muss die jeweilige Deployment-Technologie über die Änderungen informiert werden. Andernfalls können die Änderungen von den Technologien rückgängig gemacht werden. Zustandserhaltende

Funktionalitäten sind davon nicht betroffen und können ohne Berücksichtigung der Deployment-Technologien angereichert und ausgeführt werden.

Die Instanzinformationen, die von den Deployment-Technologien abgerufen werden können, unterscheiden sich erheblich in ihrer Aussagekraft und ihrem Detailgrad. Daher wurde die Plug-in-basierte Instanzmodellvervollständigung hinzugefügt, um zusätzliche Komponenten, Eigenschaften oder Relationen zwischen Komponenten zu identifizieren oder zu verfeinern. Dafür wird jedoch eine große Anzahl an Plug-ins benötigt, die komponentenspezifische Informationen ableiten können. Sobald jedoch ein Plug-in einmal implementiert ist, kann es in jeder Ausführung wiederverwendet werden. Aktuell stehen 12 technologiespezifische Plug-ins zur Verfügung (s. Abschnitt 8.3). Je mehr Plug-ins zur Verfügung stehen, desto mehr Details können für laufende Anwendungen automatisiert identifiziert werden.

Die Identifizierung von *connectsTo*-Relationen zwischen zwei Komponenten ist schwierig, selbst wenn zahlreiche Plug-ins zur Vervollständigung der Instanzmodelle verfügbar sind. Der Grund dafür ist, dass die Konfiguration einer Komponente auf zahlreiche Arten realisiert werden kann. Eine gängige Konfigurationsmöglichkeit ist zum Beispiel die Verwendung von Umgebungsvariablen, die von außen gesetzt werden können. Zu welcher Komponente eine Umgebungsvariable gehört, lässt sich jedoch weder generisch bestimmen, noch sind die Namen und Werte der Variablen standardisiert. Die Suche nach gemeinsamen Bestandteilen in den Variablennamen oder Konfigurationen kann zwar Informationen liefern, die zur Ableitung *connectsTo*-Relationen verwendet werden können, jedoch gibt es keine Garantie, dass diese automatisch identifiziert werden können. Daher kann es erforderlich sein, *connectsTo*-Relationen manuell hinzuzufügen, damit das abgeleitete Instanzmodell alle Beziehungen zwischen den einzelnen Komponenten vollständig abbildet.

Um zustandsändernde Funktionalitäten zu implementieren, muss die Deployment-Technologie, die die betreffende Komponente verwaltet, in den Vorgang eingebunden werden. Andernfalls können die Änderungen von der Technologie wieder rückgängig gemacht werden. Daher können für jede zustandsändernde Funktionalität unterschiedliche Implementierungen erfor-

derlich sein. Beispielsweise können für das Management von Containern, die mit Kubernetes bereitgestellt werden, die sogenannten *Kubernetes Operations* verwendet werden, während Komponenten, die von Puppet verwaltet werden, die Implementierung in der Puppet DSL erfordern kann. Dabei existieren zwei Möglichkeiten, um Implementierung einer Managementfunktionalität für mehrere Deployment-Technologien zu realisieren: Entweder wird für jede Deployment-Technologie ein Managementtyp definiert, der die Managementfunktionalität für die entsprechende Deployment-Technologie umsetzt, oder es wird ein Managementtyp verwendet, wobei die Implementierung der Managementfunktionalität mit mehreren Deployment-Technologien umgehen kann. Daher können die Implementierungen komplex werden und Entwickelnde müssen die API der entsprechenden Deployment-Technologien kennen und verwenden. Ähnlich wie bei den Plug-ins können die einmal implementierten Operationen jedoch in jeder Anwendung wiederverwendet werden, die eine ähnliche Kombination aus Deployment-Technologien und Komponenten verwendet.

Im Gegensatz zu den Deployment-Modellen stellen die abgeleiteten Instanzmodelle nur eine laufende Anwendung dar und enthalten keine Artefakte. Da außerdem Relationen oder sogar Eigenschaften fehlen können, können die Instanzmodelle nicht für die Bereitstellung der Anwendung genutzt werden. Durch die Abbildung der technologiespezifischen Informationen auf ein EDMM-basiertes Instanzmodell ist es jedoch möglich Managementfunktionalitäten auszuführen. Die existierenden Komponententypen erlauben die Überprüfung, ob alle notwendigen Informationen für die Ausführung einer Managementfunktionalität vorhanden sind. Fehlen einzelne Eingabeparameter, wie beispielsweise Passwörter, können diese manuell nachgetragen werden. Um eine Anwendung anhand ihres Instanzmodells wieder bereitzustellen, können existierende Arbeiten verwendet werden [Bin15; HBLE14]. Darüber hinaus ist es immer möglich, das Modell manuell anzupassen und zu erweitern, sodass die Anwendung wieder bereitgestellt werden kann.

In diesem Kapitel wurde auf algorithmische Beschreibungen verzichtet, da diese jeweils technologiespezifische Lösungen (s. Abschnitt 8.3) erfordert.

Kapitel 8

Architektur und prototypische Umsetzung

Im folgenden Kapitel wird die Architektur und die prototypische Umsetzung des Lauffeuer-Frameworks vorgestellt. Die Umsetzung aller beschriebenen Konzepte aus den Kapiteln 4 bis 7 in einem zusammenhängenden Prototyp bildet dabei den abschließenden Forschungsbeitrag 6 dieser Dissertation.

Zu Beginn des Kapitels wird in Abschnitt 8.1 eine Übersicht der Framework-Architektur vorgestellt und die einzelnen Komponenten sowie deren Interaktionen werden im Detail beschrieben. Anschließend werden in Abschnitt 8.2 einzelne Details der Umsetzung beleuchtet, wie die Abbildung des Metamodells auf den TOSCA-Standard, der für die Modellierung und Ausführung der beschriebenen Konzepte verwendet wurde. Abschnitt 8.3 beschreibt die Validierung und Evaluation der Lauffeuer-Methode. Schließlich wird die Verwendung des Lauffeuer-Frameworks in anderen Arbeiten vorgestellt.

Die einzelnen Komponenten des Prototyps wurden jeweils zu den einzelnen Publikationen erstellt und entsprechend erweitert. Daher wurden Teile des Prototyps bereits in den Erstautor-Veröffentlichungen [HBB+21; HBBL23; HBF+18; HBF+20; HBF+21; HBKL19; HBL+19] beschrieben.

8.1 Architektur des LAUFFEUER-Frameworks

Für die Umsetzung der in den vorherigen Kapiteln vorgestellten Methoden und Konzepte wurde das LAUFFEUER-Framework[1] entwickelt. Dabei erweitert es bereits existierende Frameworks und integriert diese zu einem zusammenhängenden System, wie in Abbildung 8.1 dargestellt.

Die zentrale Komponente des Frameworks bildet das Modellierungswerkzeug *Winery*[2] [KBBL13]. Winery ermöglicht die grafische Modellierung von Deployment-Modellen und wurde um die Möglichkeit erweitert, musterbasierte Deployment-Modell zu erstellen, zu verfeinern sowie Muster in ausführbaren Modellen zu erkennen. Dafür wurde der *Pattern Atlas*[3] [LB21] als zentrales Repository für Muster und Mustersprachen als Datenbank in Winery integriert. In diesem Zuge wurde der Pattern Atlas entsprechend um die Möglichkeit erweitert, Muster im Kontext der Deployment-Modellierung als Verhaltens- und Komponentenmuster zu kategorisieren. Auf Basis dieser Einordnung können Modellierende die Muster aus dem Pattern Atlas in Winery zur Beschreibung von Anwendungen verwenden und haben gleichzeitig Zugriff auf die Beschreibungen und Erklärungen der einzelnen Muster.

Um das Instanzmodell einer Anwendung zu generieren, wurde das *TOSCin*[4]-Werkzeug entwickelt. Es ist Teil des EDMM-Frameworks und realisiert die Ableitung von Instanzmodellen, indem es Laufzeitinformationen einer Anwendung von den verwendeten Deployment-Technologien interpretiert und auf EDMM abbildet. Das resultierende Instanzmodell kann im Anschluss durch Winery automatisiert vervollständigt und manuell angepasst werden, bevor die Anwendung durch das *OpenTOSCA* Ökosystem[5] verwaltet werden kann. Dabei realisiert der OpenTOSCA Container[6] [BBH+13] die Workflowgenerierung und übernimmt die Koordination und Ausführung der Deployment- und Managementworkflows über eine *Workflow-Engine*.

[1] https://github.com/lharzenetter/lauffeuer-framework
[2] https://github.com/eclipse/winery
[3] https://github.com/PatternAtlas
[4] https://github.com/UST-EDMM/edmm/tree/master/TOSCin
[5] https://github.com/OpenTOSCA
[6] https://github.com/OpenTOSCA/container

Abbildung 8.1: Grobarchitektur des LAUFFEUER-Frameworks

8.1.1 Modellierungswerkzeug Winery

Das in Abbildung 8.2 dargestellte Modellierungswerkzeug besteht aus zwei User Interface Komponenten, der *Management UI* und der *Grafischen Topologiemodellierung*, sowie einem Backend, das mehrere Komponenten bündelt. Der Zugriff auf die Backend-Komponenten wird durch eine REST-API ermöglicht. Diese wurde, wie auch die beiden Frontend-Komponenten, um entsprechende Endpunkte erweitert, sodass die vorgestellten Konzepte abgebildet werden können. Dabei wurde Winery um die folgenden Funktionalitäten erweitert: (i) die Unterstützung für die Modellierung *musterbasierter Deployment-Modelle*, (ii) die Konkretisierung der Muster durch die *Musterverfeinerung*, (iii) die Abstraktion ausführbarer Deployment-Modelle durch die *Mustererkennung*, (iv) die Erweiterung von Anwendung mit zusätzlichen Features durch die *Managementanreicherung* sowie (v) die *Instanzmodellvervollständigung*, um fehlende Eigenschaften in Instanzmodellen zu erkennen. Zusätzlich wurde (vi) der *Pattern Atlas* als Muster-Repository eingebunden und es wurden neue Repositorys für (vii) Managementtypen und (viii) Verfeinerungs- und Erkennungsmodelle geschaffen.

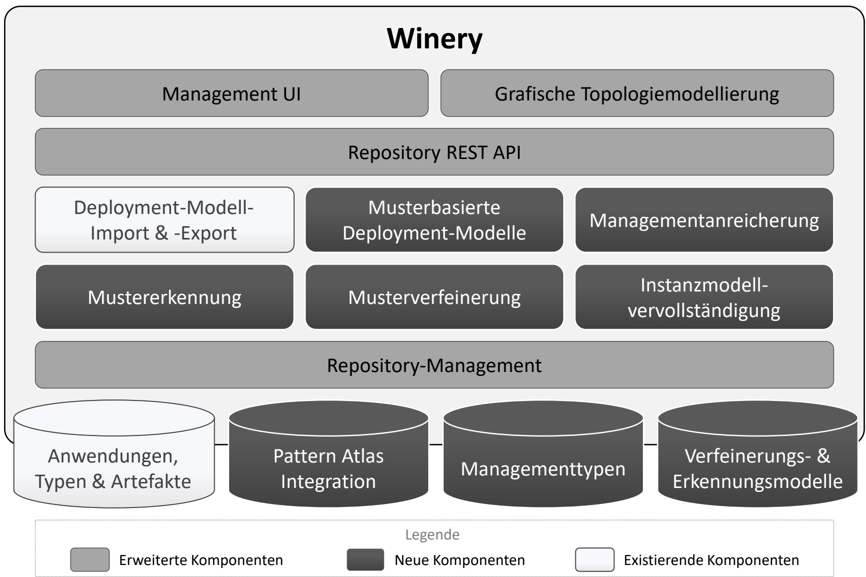


Abbildung 8.2: Die Komponenten des erweiterten Modellierungswerkzeugs

Während die Komponente der *Musterbasierten Deployment-Modelle* Grundfunktionalitäten für die musterbasierte Modellierung von Anwendungen bereitstellt, realisieren die *Mustererkennung* und *Musterverfeinerung* die in Kapitel 5 vorgestellten Konzepte. Funktionalitäten, die sowohl für die Erkennung als auch für die Verfeinerung verwendet werden, d. h. Algorithmen 5.1, 5.2 und 5.4 bis 5.8, sind daher in der *Musterbasierten Deployment-Modelle*-Komponente implementiert. Algorithmus 5.3 beschreibt die Verfeinerung von Mustern und ist in der entsprechenden Komponente implementiert. Die *Mustererkennung* realisiert schließlich die Algorithmen 5.9 und 5.10.

Die Implementierung der in Kapitel 6 vorgestellten Managementanreicherung besteht grundlegend aus der gleichnamigen Komponente sowie einem Repository, welches die zusätzlichen Managementfunktionalitäten in Form der Managementtypen beinhaltet. Um verfügbare Managementfeatures zu einer Anwendung hinzuzufügen, implementiert die Komponente

den Algorithmus 6.1 und verwendet das neue Repository. Im Gegensatz zu den existierenden Komponenten- und Relationstypen können die Managementtypen nicht direkt für die Modellierung von Komponenten verwendet werden, sondern dienen lediglich der Definition für zusätzliche Managementfunktionalitäten eines bestimmten Komponententyps. Daher wurde das *Repository-Management* entsprechend angepasst und greift auf Managementtypen Repositorys nur während der Managementanreicherung zu.

Für die automatisierte Instanzmodellvervollständigung ist es für Modellierende hilfreich, die Topologie der Anwendung zu sehen. Dadurch können fehlende Komponenten, Relationen und Eigenschaften leichter identifiziert werden. Auch Plug-ins, die weitere Details erkennen können, können den Modellierenden grafisch anzeigen, welche Komponenten sie konkretisieren können. Daher wurde die *Instanzmodellvervollständigung* in das Modellierungswerkzeug integriert und erweitert die grafische Topologiemodellierung um entsprechende Funktionalitäten, wie das Hervorheben betroffener Komponenten oder die Auswahl der Plug-ins, die ausgeführt werden sollen.

Insgesamt wurde Winery daher um zahlreiche Funktionalitäten erweitert, die die Modellierung und das Modellmanagement von Anwendungen automatisieren und vereinfachen. Weitere Einzelheiten und Details zur technischen Umsetzung werden im folgenden Abschnitt 8.2 beleuchtet.

8.1.2 Workflowgenerierung und -ausführung mit OpenTOSCA

Zur Umsetzung der *Workflowgenerierung* wurde die Ausführungsumgebung des OpenTOSCA Ökosystems, der sogenannte *OpenTOSCA Container* erweitert. Dieser beinhaltet bereits einen Plug-in-basierten *Workflowgenerator*, wobei existierende Plug-ins, wie beispielsweise das *Deploy*-Plug-in, wiederverwendet werden, um eine Anwendung automatisiert bereitstellen zu können. Entsprechend der in Kapitel 6 vorgestellten Konzepte wurden Plug-ins umgesetzt, die unter anderem Workflows für das Einfrieren und Auftauen einer Anwendung generieren. Die Zusammensetzung der relevanten Komponenten des OpenTOSCA Containers sind in Abbildung 8.3 dargestellt.

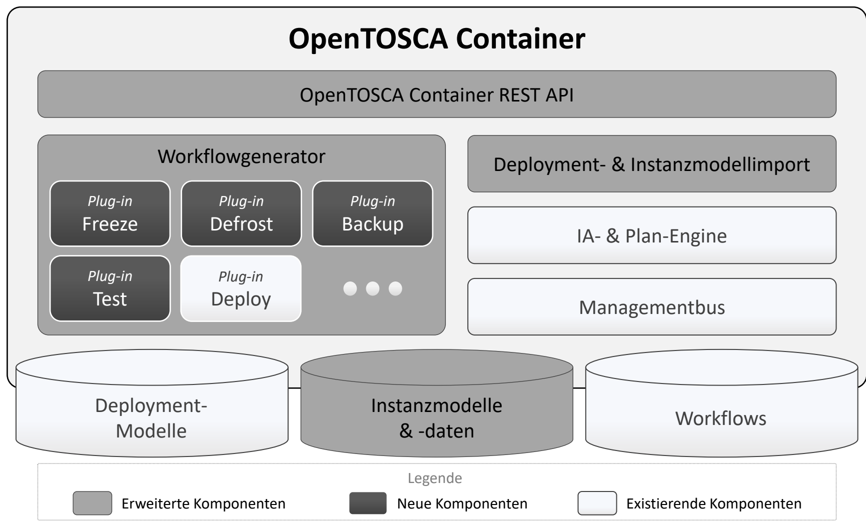


Abbildung 8.3: Die Komponenten des OpenTOSCA Containers, der Workflowgenerierung und Ausführungsumgebung

Um zusätzlich zur Workflowgenerierung für ausführbare Deployment-Modelle auch die Möglichkeit zu schaffen, Managementworkflows für Instanzmodelle zu generieren, wurde der Workflowgenerator entsprechend erweitert. Darüber hinaus wurden die Importfunktionalität sowie die Instanzmodelldatenbank angepasst, sodass diese auch die Instanzmodelle und -daten einer bereits laufenden Anwendung importieren und verwalten können. Für den entsprechenden Zugriff von Außen wurde die REST-API des OpenTOSCA Containers zusätzlich überarbeitet und um die benötigten Endpunkte erweitert. Der OpenTOSCA Container ermöglicht somit die automatisierte Generierung und Ausführung von Workflows zur Bereitstellung, Verwaltung und Außerbetriebnahme von Anwendungen.

8.1.3 Instanzmodellgenerierung mit TOSCin

Die Ableitung eines Instanzmodells einer laufenden Anwendung anhand ihrer Deployment-Technologien wurde in der neuen Komponente *TOSCin*

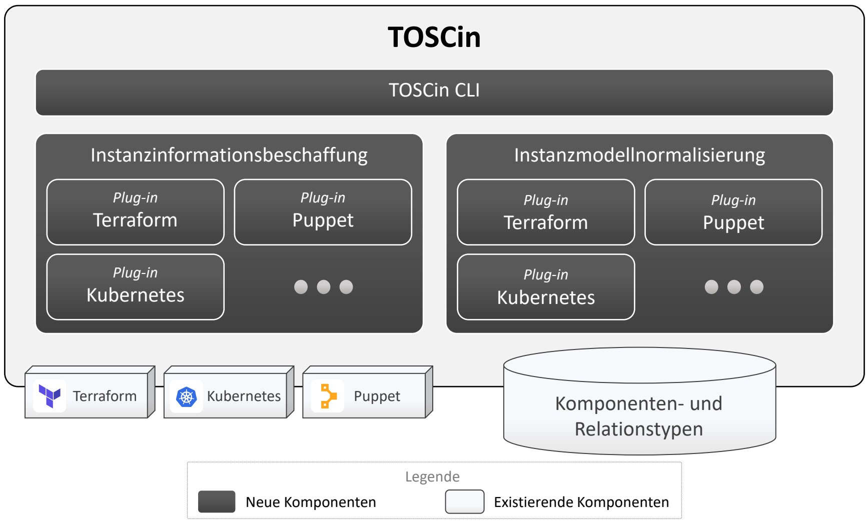


Abbildung 8.4: Die Komponenten der Instanzmodellgenerierung

implementiert. Im Gegensatz zu den anderen Komponenten verfügt TOSCin nicht über ein grafisches User Interface, sondern kann über die Kommandozeile (engl. „Command Line Interface", CLI) verwendet werden.

Wie in Kapitel 7 beschrieben, implementiert TOSCin die beiden Schritte der *Instanzinformationsbeschaffung* sowie die *Instanzmodellnormalisierung*. Da die Komponenten dabei jeweils deployment-technologiespezifische Informationen verarbeiten müssen, sind beide plug-in-basiert. In Abbildung 8.4 sind dabei beispielhaft die Plug-ins für die Deployment-Technologien Terraform, Puppet und Kubernetes dargestellt. Durch die Instanzmodellnormalisierung wird dabei ein EDMM-basiertes Instanzmodell generiert, indem ein Repository von existierenden Komponenten- und Relationstypen verwendet wird. Das resultierende Instanzmodell wird schließlich an Winery gesendet und gespeichert. Dort kann es im Anschluss durch die grafische Topologiemodellierungskomponente angezeigt und mithilfe der Instanzmodellvervollständigung um fehlende Informationen ergänzt werden, bevor es zur Workflowgenerierung an den OpenTOSCA Container gegeben wird.

8.2 Prototypische Implementierung

Die prototypische Umsetzung der LAUFFEUER-Methode wurde mit dem OpenTOSCA Ökosystem realisiert. Da TOSCA alle Konzepte von EDMM beinhaltet [WBH+20c], wird im Folgenden zunächst die Umsetzung des Metamodells mit TOSCA beschrieben. Im Anschluss werden Details und Erweiterungen einzelner, relevanter Komponenten des Ökosystems vorgestellt.

8.2.1 Abbildung des Metamodells auf TOSCA

Die Topology Orchestration Specification for Cloud Applications (TOSCA) [OAS13b; OAS20] ist eine standardisierte Modellierungssprache zur Beschreibung, Bereitstellung und Verwaltung von Anwendungen. Ähnlich wie bei EDMM, werden Anwendungen dabei als gerichtete Graphen beschrieben, welche typisierte Knoten und Kanten enthalten. Auch das in Kapitel 4 vorgestellte EDMM-basierte Metamodell kann mit TOSCA umgesetzt werden.

Um eine Anwendung in TOSCA zu beschreiben, definiert der TOSCA Standard *Service Templates*. Diese beschrieben alle Informationen einer Anwendung wie Ein- und Ausgabeparameter, Managementworkflows, genannt *Plans*, sowie die Topologie der Anwendung – das *Topology Template*. Innerhalb eines *Topology Templates* werden Knoten als *Node Templates* und Kanten als *Relationship Template* bezeichnet. Daher entspricht ein Deployment-Modell aus EDMM einem Service Template, während Komponenten und Relationen durch Node bzw. Relationship Templates abgebildet werden. Ähnlich wie Komponenten und Relationen sind Node Templates und Relationship Templates durch ein Typsystem definiert. Dabei entsprechen *Node Types* Komponententypen, während Relationstypen durch *Relationship Types* abgebildet werden. Die Node Types und Relationship Types können durch sogenannte *Property Definitions* Eigenschaften definieren, denen die Node Templates, bzw. Relationship Templates in einer Topologie konkrete Werte zuweisen können. Darüber hinaus können an einem Typ in TOSCA *Interfaces* und *Operations* mit *Input Parameter* und *Output Parameter* definiert werden. Operations können durch *Artifact Templates* mit ausführbaren Arte-

fakten, die die entsprechende Operation implementieren, verknüpft werden. Solche Artifact Templates werden in TOSCA *Implementation Artifacts (IA)* genannt. Im Gegensatz dazu stellen *Deployment Artifacts (DA)* die Artefakte einer Komponente dar. Deployment Artifacts sind dabei *Artifact Templates*, welche Implementierungen einer Komponente darstellen und einem *Node Template* zugeordnet sind. Des Weiteren sind auch *Artifact Templates* einem Artefakttyp zugeordnet, die sogenannten *Artifact Types*.

Um die musterbasierte Modellierung von Anwendungen in TOSCA zu ermöglichen, wird das Konzept der Namespaces in TOSCA ausgenutzt. Ein Namespace gruppiert eine Menge an *TOSCA Definitions*, d. h. alle in TOSCA definierbaren Elemente wie zum Beispiel Service Templates, Node Types oder Relationship Types. Eine TOSCA Definition muss dabei immer genau einem Namespace angehören. Daher wurden *Muster-Namespaces* eingeführt, um Mustertypen in TOSCA beschreiben zu können. Ein Muster-Namespace ist ein Namespace, der durch eine Eigenschaft angibt, dass in diesem Namespace nur Muster liegen. Daher wird ein Node Type in einem Muster-Namespace erstellt, um ein Komponentenmustertyp in TOSCA zu modellieren. Da ein Komponentenmuster ein Komponentenmustertyp instanziiert, müssen Node Templates, die Komponentenmuster darstellen, von Komponenten unterscheidbar sein. In Winery werden Komponentenmuster von Komponenten unterschieden, indem Komponentenmuster als quadratische Knoten und Komponenten als Rechtecke dargestellt werden. Die Beispielanwendung aus Abbildung 4.2 ist als Screenshot von Winery in Abbildung 8.5 abgebildet.

Für die Annotation von *Node Templates* definiert TOSCA bereits *Policies*, die einem *Policy Type* zugeordnet sind. Daher werden Policies in TOSCA verwendet, um Verhaltensmuster an Komponenten und Relationen annotieren zu können. Wie auch Komponentenmuster müssen Verhaltensmustertypen, welche in TOSCA durch Policy Types dargestellt werden, einem Muster-Namespace angehören. Die Möglichkeit Policies an Relationship Templates zu annotieren ist zwar in TOSCA nicht vorgesehen, allerdings handelt es sich hierbei um eine Erweiterung zur Modellierungszeit. Durch die Verfeinerung des musterbasierten Deployment-Modells wird ein standardkonformes Modell generiert, das von jeder TOSCA-Runtime verarbeitet werden kann.

8.2.2 Verfeinerungs- und Erkennungsmodelle in TOSCA

Zur Verfeinerung und Erkennung von Mustern in Deployment-Modellen werden die in Kapitel 5 vorgestellten Verfeinerungs- und Erkennungsmodelle benötigt. Dafür wurde in Winery die Möglichkeit geschaffen diese als eigene Modellelemente erstellen zu können. Erkennungsmodelle unterscheiden sich hierbei von den Verfeinerungsmodellen lediglich in einer Annotation, die sie als Erkennungsmodelle ausgibt. Wie in Kapitel 5 diskutiert, können Erkennungsmodelle auch zur Verfeinerung genutzt werden, Verfeinerungsmodelle jedoch nicht zur Erkennung, da sie möglicherweise Muster verfeinern, die nicht erkannt werden können. Der Aufbau der Modelle in TOSCA ist jedoch ansonsten identisch: Die Muster- und Lösungsstrukturen eines Verfeinerungs- und Erkennungsmodells sind jeweils als Topology Template umgesetzt. Eine Zuordnung zwischen zwei Modellelementen der beiden Topology Templates beinhaltet dabei die IDs der jeweiligen Node bzw. Relationship Templates. Zusammen mit den anderen Elementen der Zuordnungen sowie der Art der Zuordnungen werden sie in dem jeweiligen Modell gespeichert.

8.2.3 Erkennung, Verfeinerung und Anreicherung in Winery

In den Algorithmen und Beschreibungen für die Verfeinerung (Algorithmus 5.3), Erkennung (Algorithmus 5.9), Managementanreicherung (Algorithmus 6.1) und Instanzmodellvervollständigung (Abschnitt 7.2) sind jeweils Schritte für die Auswahl eines bestimmten Elements definiert. Dabei kann die Auswahl sowohl automatisiert als auch manuell erfolgen. Damit die einzelnen Schritte nachvollziehbar bleiben, wurden in der Implementierung der Algorithmen jeweils eine manuelle Auswahl umgesetzt. Die Implementierung wartet dabei, bis Modellierende eine Auswahl getroffen haben und wendet anschließend zum Beispiel ein Verfeinerungsmodell automatisch an.

Um diesen manuellen Schritt zu ermöglichen, wurden Websockets implementiert, die die Algorithmen ausführen und die Auswahl dem Modellierenden überlassen. Dabei können die zur Auswahl stehenden Optionen im Detail betrachtet werden und die betroffenen Komponenten und Relationen

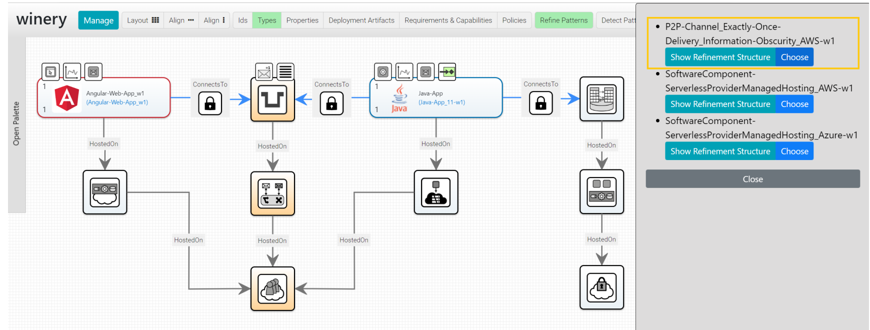


Abbildung 8.5: Verfeinerung eines musterbasierten Deployment-Modells

werden entsprechend hervorgehoben. Wurde eine Auswahl getroffen, wird der Algorithmus fortgesetzt, bis die nächste Auswahl benötigt wird oder der Algorithmus terminiert. Wird keine Auswahl getroffen oder eine leere Auswahl zurückgesendet, wird die Ausführung der Algorithmen beendet.

In Abbildung 8.5 ist eine musterbasierte Anwendung dargestellt, die gerade verfeinert wird. Die modellierte Anwendung entspricht dabei der in Abbildung 4.2 gezeigten Anwendung und besteht aus einem Angular-Frontend, einem Java-Backend und einer Datenbank. Zur Kommunikation zwischen dem Frontend und dem Backend soll eine sichere Verbindung über ein Message Channel-Muster hergestellt werden. Die hervorgehobenen Komponentenmuster, d. h. der Message Channel auf der Message-oriented Middleware in der Public Cloud in Abbildung 8.5, können durch das erste auf der rechten Seite dargestellten Verfeinerungsmodell konkretisiert werden. Die betroffenen Komponenten werden hervorgehobenen, sobald sich die Maus über dem Namen des Verfeinerungsmodells befindet. Wird nun dieses Verfeinerungsmodell ausgewählt, wird es automatisiert angewendet und das neue Modell wird, wie in Abbildung 8.6 dargestellt, angezeigt. Aus den Eigenschaften der eingefügten *SQS-Queue*-Komponente kann zudem entnommen werden, dass die Verhaltensmuster entsprechend umgesetzt wurden. Im nächsten Schritt kann nun aus zwei PRMs gewählt werden. Die Erkennung, Anreicherung und Vervollständigung funktionieren analog dazu.

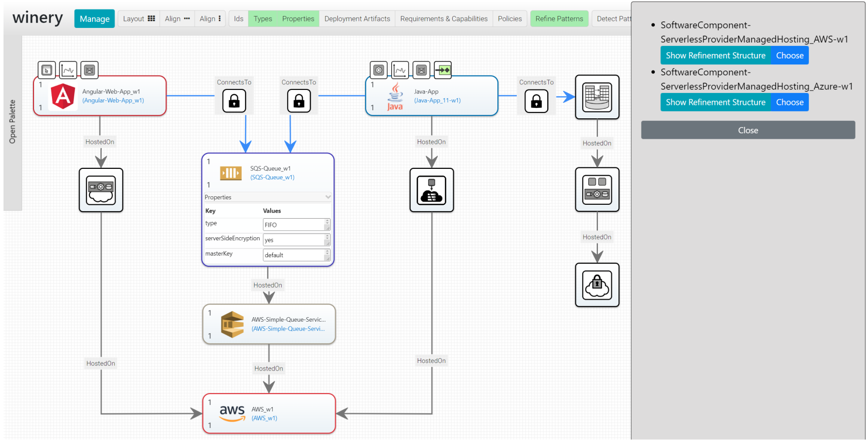


Abbildung 8.6: Beispiel eines musterbasiertes Deployment-Modells nach der Anwendung des ersten Verfeinerungsmodells

8.3 Validierung und Evaluation

Um die Lauffeuer-Methode zu validieren und evaluieren, wurde der in den vorherigen Abschnitten vorgestellte Prototyp verwendet. Im Folgenden wird ein weiteres Beispiel eines musterbasierten Deployment-Modells vorgestellt, bevor die Ableitung des Instanzmodells einer komplexen Microservice-Anwendung vorgestellt wird. Schließlich wird das Management beider Anwendungen durch die Generierung von Managementworkflows beschrieben.

8.3.1 Verfeinerung und Erkennung von Mustern

In Abbildung 8.7 ist ein weiteres musterbasiertes Deployment-Modell dargestellt. Es besteht aus zwei konkreten Komponenten, einer Web-App auf der linken Seite und einer Backend-Komponente, die sich zu einem Relational Database-Muster verbindet. Darüber hinaus kommunizieren die konkreten Komponenten nicht direkt über einen Message Channel miteinander, sondern verwenden das Application Component Proxy [FLR+14], um HTTP-Anfragen auf eine Messaging-Technologie zu übersetzen. Anstelle

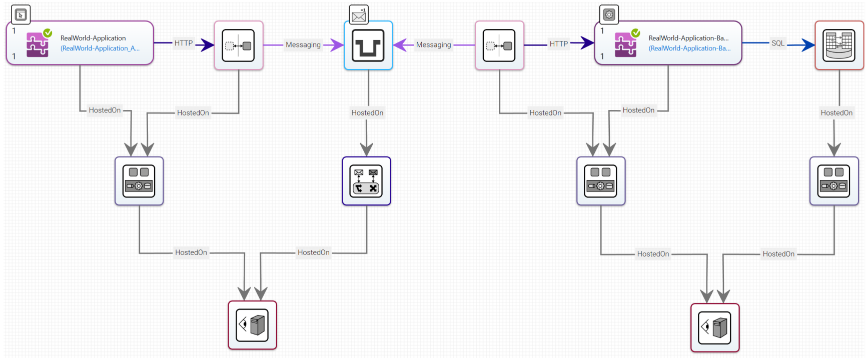


Abbildung 8.7: Beispiel eines weiteren musterbasierten Deployment-Modells

eines Public Cloud- oder Private Cloud-Musters sollen darüber hinaus die Execution Environments sowie die Message-oriented Middleware direkt auf einem Hypervisor [FLR+14] bereitgestellt werden.

Eine mögliche Verfeinerung des musterbasierten Deployment-Modells ist in Abbildung 8.8 dargestellt. Dabei wurden alle Komponentenmuster zu konkreten Komponenten umgesetzt. Auch die Umsetzung der Application Component Proxy-Muster konnte generisch umgesetzt werden: In dem verwendeten PRM-Repository existieren PRMs, die HTTP-Verbindungen so übersetzen können, dass sie über ein MQTT-Topic ausgetauscht und wieder in HTTP übersetzt werden können. Das musterbasierte Deployment-Modell, das ausführbare Deployment-Modell und die für die Verfeinerung verwendeten PRMs sind im Lauffeuer-Methode-Modellierungs-Repository[1] verfügbar.

Insgesamt wurden für die Verfeinerung und Erkennung von Anwendungen 46 Verfeinerungs- und Erkennungsmodelle definiert. Damit lassen sich die beiden vorgestellten musterbasierten Deployment-Modelle, siehe Abbildung 4.2 bzw. Abbildung 8.5 und Abbildung 8.7, automatisiert verfeinern und wieder erkennen. Alle PRMs sind dabei wiederverwendbar und zur freien Verfügung im Lauffeuer-Methode-Modellierungs-Repository definiert. Somit existieren bereits zahlreiche Verfeinerungsmodelle, um musterbasierte

[1] `https://github.com/lharzenetter/tosca-lauffeuer-modeling`

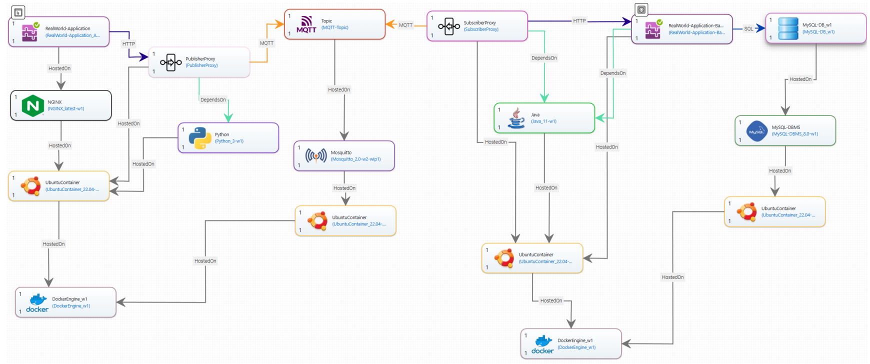


Abbildung 8.8: Eine mögliche Verfeinerung des MDMs aus Abbildung 8.7

Deployment-Modelle in ausführbare zu transformieren. Durch die Verfeinerung bzw. Erkennung einzelner Komponentenmuster in einer Iteration kann ein in der vorherigen Iteration anwendbares PRM nicht mehr anwendbar sein (s. Abschnitt 5.8). Deshalb wurden im Schnitt je PRM drei verschiedene Versionen erstellt, wobei in jeder Version unterschiedliche Modellknoten bereits transformiert wurden. Die Erstellung unterschiedlicher PRM Versionen ist aktuell eine manuelle Aufgabe und kann möglicherweise in zukünftigen Arbeiten automatisiert werden.

8.3.2 Generierung von Instanzmodellen

Zur Validierung des integrierten Managementsystems für zusammengesetzte Anwendungen (s. Abbildung 7.1) wurde die Microservice-Demoanwendung Sock Shop[2] verwendet. Der Sock Shop besteht aus insgesamt 15 Microservices, die den Aufbau eines Webshops mit unterschiedlichen Technologien demonstrieren. Um ein verteiltes Deployment mit unterschiedlichen Deployment-Technologien umzusetzen, wurde der Sock Shop in drei Teile aufgeteilt. Anschließend wurde jeder Teil mit einer anderen Deployment-Technologie bereitgestellt: Während der Katalogservice zusammen mit der

[2] https://microservices-demo.github.io/

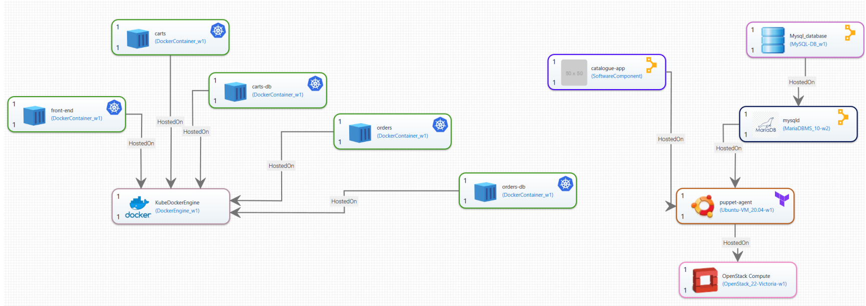


Abbildung 8.9: Instanzmodell des angepassten Sock Shops Demoanwendung nach dem Instanzmodellnormalisierungschritt

Katalogdatenbank mit Puppet auf einer VM bereitgestellt wurde, wurde die VM selbst von Terraform hochgefahren und konfiguriert. Alle weiteren Services wurden durch Kubernetes in einem Cluster verwaltet.

Um im nächsten Schritt ein Instanzmodell der Anwendung mithilfe des in Kapitel 7 vorgestellten Konzepts automatisiert verwalten zu können, wurde TOSCin eingesetzt. Als Eingabe wurden TOSCin die Endpunkte des Puppet-Servers und Kubernetes sowie der Pfad zur Terraform State-Datei übergeben. Alle Laufzeitinformationen, die aus den einzelnen Deployment-Technologien abgerufen werden konnten, wurden durch TOSCin in der Instanzmodellnormalisierung in ein EDMM-basiertes Instanzmodell zusammengeführt. Dieses ist in Abbildung 8.9 dargestellt. Ähnlich zu dem in Abbildung 7.2 beschriebenen Instanzmodell sind die eine Komponente durch die Deployment-Technologie annotiert, die die Komponente verwalten. Entsprechend sind die Docker-Container mit Kubernetes annotiert, dargestellt durch das Kubernetes-Icon, während die VM-Komponente das Terraform-Logo beinhaltet. Die Komponenten des Katalogservices sind analog dazu mit dem Puppet-Logo gekennzeichnet. Im Gegensatz dazu wurden die Docker-Engine und die OpenStack-Instanz lediglich von den Deployment-Technologien erkannt, verwalten diese jedoch nicht. Um die Zuordnungen zischen den „verwaltet von"- und „identifiziert von"-Annotationen nicht zu vertauschen (s. Abschnitt 7.2.3), werden diese im Modellierungswerkzeug nicht angezeigt.

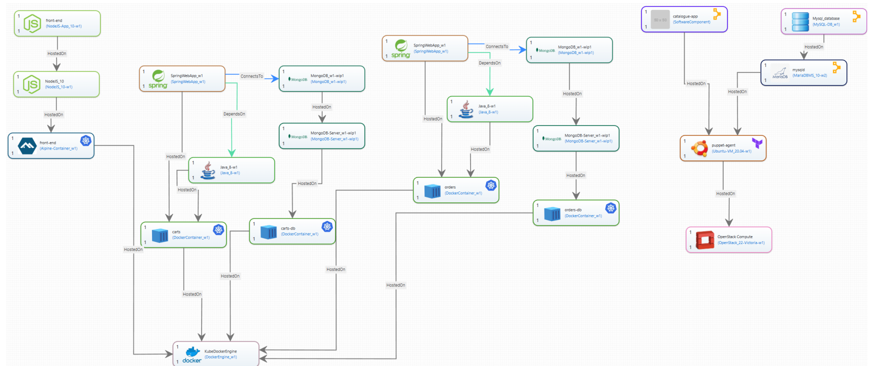


Abbildung 8.10: Instanzmodell des angepassten Sock Shops Demoanwendung nach dem Instanzmodellvervollständigungsschritt

Zur Übersichtlichkeit enthalten die Abbildungen 8.9 und 8.10 nicht alle 15 Microservices, sondern lediglich den Frontend-Service sowie die Katalog-, Einkaufswagen- und Bestellservices inklusive ihrer Datenbanken.

Im nächsten Schritt kann das normalisierte Instanzmodell durch die Instanzmodellvervollständigung erweitert werden. Dafür wurden insgesamt 12 technologiespezifische Plug-ins implementiert und nacheinander auf das in Abbildung 8.9 gezeigt Instanzmodell angewendet. Dabei können weitere Komponenten identifiziert und bereits im Instanzmodell enthalte Komponenten konkretisiert werden. Das Resultat ist in Abbildung 8.10 dargestellt. Neben den Komponenten konnten auch einige Relationen identifiziert werden, wie die Verbindungen der Einkaufswagen- und Bestellservices zu ihrer jeweiligen Datenbank. Durch eine manuelle Erweiterung des Instanzmodells können darüber hinaus fehlende Relationen hinzugefügt werden.

8.3.3 Verwaltung von Anwendungen

Zur automatisierten Verwaltung von Anwendungen, wurde der Managementworkflowgenerator des OpenTOSCA Containers um fünf zusätzliche Plug-ins erweitert. Mit diesen ist es nun möglich, Managementworkflows für eine Anwendung zu generieren, die die Anwendung einfrieren, auftauen,

aktualisieren, testen und Backups ihrer zustandsbehafteten Komponenten erstellen können. Entsprechend wurden verschiedene Managementoperationen implementiert, verfügbar im Managementfeature-Repository[3], die durch Managementtypen automatisch an die Komponenten von Deployment- und Instanzmodellen angereichert werden können (s. Abschnitt 6.2). Die anschließend durch den Managementworkflowgenerator generierten Workflows können sowohl durch OpenTOSCA verwaltete als auch durch Instanzmodelle beschriebene Anwendungen automatisiert verwalten.

8.4 Verwendung des Frameworks in anderen Arbeiten

Das LAUFFEUER-Framework wird, wie auch die einzelnen Konzepte der gesamten Methode, zum Teil in anderen Arbeiten wiederverwendet.

8.4.1 Modellieren und Ausführen von Quantenanwendungen

Das Quantencomputing führt neue Rechenparadigmen ein, die versprechen, Probleme, die mit klassischen Computern praktisch nicht gelöst werden können, zu lösen. Dabei müssen Quantenanwendungen jedoch in klassische Anwendungen integriert werden. Allerdings unterscheidet sich der Einsatz von Quantenanwendungen derzeit erheblich von klassischen Anwendungen. Um diese Probleme zu überwinden wurde das TOSCA4QC-Modellierungskonzept vorgestellt [WBH+20a]. Dabei wurden in Zusammenarbeit mit Wild et al. zwei Deployment-Modellierungsstile entwickelt: (i) einen SDK-spezifischen Modellierungsstil, der alle technischen Bereitstellungsdetails abdeckt, und (ii) einen SDK-unabhängigen Modellierungsstil, der allgemeine Modellierungsprinzipien unterstützt. Mithilfe von *Topologieverfeinerungsmodellen* (engl. „*Topology Refinement Models*“), welche anstelle der Musterstruktur eine zweite Lösungsstruktur enthalten, kann das SDK-agnostische Modell zu einem ausführbaren SDK-spezifischen Modell verfeinert werden. Dazu wurde das LAUFFEUER-Framework entsprechend erweitert, um die Modellierung und Anwendung von Topologieverfeinerungsmodellen zu ermöglichen.

[3] `https://github.com/OpenTOSCA/tosca-definitions-management-features`

8.4.2 Identifizieren von Hosting-Alternativen

Für die Bereitstellung einer Anwendungskomponente können verschiedene Cloud-Service-Modelle mit unterschiedlichen Verwaltungsanforderungen verwendet werden. Zum Beispiel existieren Cloud-Service-Modelle, die mehr Kontrolle über die einzelnen Komponenten erlauben, während andere einfacher zu benutzen sind, jedoch keine weiteren Anpassungen ermöglichen. Die Suche nach einer geeigneten Lösung ohne klare Richtlinien ist jedoch mühsam. Daher existieren Muster, die verschiedene Managementkompromisse darstellen und bei der Suche nach geeigneten Hosting-Optionen für eine bereitzustellende Komponente helfen können [YBB+22; YSB+21]. Die Auswahl eines geeigneten Musters und die Untersuchung, welche konkreten Bereitstellungsalternativen verwendet werden können, ist jedoch den Modellierenden überlassen. In einer Zusammenarbeit mit Yussupov et al. [YBB+22] wurde daher eine Methode vorgestellt, die Modellierenden bei der Verwendung der Muster sowie bei der Verfeinerung ausgewählter Muster unterstützt. Dafür wurde das LAUFFEUER-Framework genutzt, um Verfeinerungsmodelle zu definieren, die unterschiedliche Abstraktionsebenen der Hosting-Muster [YBB+22] darstellen. Dabei können Mustern zunächst zu konkreteren Mustern verfeinert werden, bis hin zur Auswahl konkreter Technologien und Cloud-Service-Modelle. Somit ist es möglich, automatisiert zwischen den unterschiedlichen Abstraktionsebenen der Muster zu wechseln und eine konkrete Lösung zu finden.

Kapitel 9

Zusammenfassung und Ausblick

Die Modellierung und das automatisierte Management von verteilten Anwendungen stellt nach wie vor große Herausforderungen für Modellierende dar. Dafür präsentiert die vorliegende Dissertation als Forschungsbeitrag 1 die Lauffeuer-Methode in Kapitel 3. Sie ermöglicht eine abstrakte Modellierung von Anwendungen mithilfe von Mustern anstelle konkreter Komponenten und Technologien. Dadurch wird verhindert, dass das Modellieren von Deployment-Modellen tiefgreifendes Expertenwissen über verfügbare Komponenten, Technologien und Cloud-Dienste erfordert. Anstatt konkreter Komponenten und Technologien müssen lediglich die abstrakten Konzepte der benötigten Muster verstanden und im Modell verwendet werden. Ein solches Deployment-Modell, das auf Basis der Muster einer Anwendung erstellt wurde, wird als *musterbasiertes Deployment-Modell* bezeichnet und bildet den Forschungsbeitrag 2 dieser Arbeit aus Kapitel 4.

Im nächsten Schritt der Lauffeuer-Methode kann das abstrakte, musterbasierte Deployment-Modell automatisiert zu konkreten Komponenten

und Technologien verfeinert werden. Dafür wurden *Verfeinerungsmodelle* eingeführt, die definieren, wie eine bestimmte Konstellation aus Mustern und Komponenten zu konkreten Technologien verfeinert werden kann. In einem iterativen Prozess werden Verfeinerungsmodelle auf das musterbasierte Deployment-Modell angewendet, bis alle Muster verfeinert wurden und das Deployment-Modell *ausführbar* ist. Während Komponentenmuster dabei meist zu konkreten strukturellen Komponenten verfeinert werden, werden Verhaltensmuster in der Regel durch bestimmte Konfiguration der betreffenden Komponenten umgesetzt. Darüber hinaus ist es nicht nur möglich Muster zu Technologien zu verfeinern, sondern auch Muster in ausführbaren Deployment-Modellen zu erkennen. Zusammen bilden diese in Kapitel 5 präsentierten Konzepte den Forschungsbeitrag 3.

Ein *ausführbares Deployment-Modell* beschreibt am Ende des Verfeinerungsprozesses die Bereitstellung einer Anwendung, sodass diese von einer entsprechenden Deployment-Technologie bereitgestellt werden kann. Um die Anwendung allerdings zur Laufzeit auch verwalten zu können, können verfügbare Managementfunktionalitäten zu dem Modell hinzugefügt werden. Für jede Managementfunktionalität wird anschließend ein Managementworkflow generiert, der die entsprechende Funktionalität, zum Beispiel das Erstellen eines Backups aller zustandsbehafteten Komponenten, automatisiert ausführt. Darüber hinaus wurde ein Konzept präsentiert, das es ermöglicht, eine Anwendung in ihrem aktuellen Zustand einzufrieren, d. h. herunterzufahren und den aktuellen Zustand persistent abzulegen, und im selben Zustand wieder bereitzustellen. Diese Managementfunktionalitäten können sowohl für neue (Forschungsbeitrag 4, Kapitel 6) als auch für laufende Anwendungen (Forschungsbeitrag 5, Kapitel 7) automatisiert angereichert und für die gesamte Anwendung ausgeführt werden.

Um die Umsetzbarkeit der Lauffeuer-Methode zu demonstrieren, wurde schließlich in Kapitel 8 das Lauffeuer-Framework vorgestellt und beschreiben. Es realisiert alle Methoden und Konzepte der Kapitel 3 bis 7 auf Basis des TOSCA-Standards. Durch die Beschreibung der Konzepte anhand des technologieunabhängigen Metamodells EDMM sind alle Konzepte jedoch auf die 13 meist verwendeten Deployment-Technologien anwendbar.

In den Schritten der LAUFFEUER-Methode sind zum Teil Auswahlmöglichkeiten gegeben, um zum Beispiel ein anwendbares Verfeinerungsmodell auszuwählen. Diese wurden im vorgestellten Prototyp als manuelle Schritte implementiert, sodass Modellierende wählen können. Für eine komplett automatisierte Lösung der Verfeinerung oder der Managementanreicherung können in zukünftigen Arbeiten beispielsweise Planning-Methoden verwendet werden. Diese sind in der Lage verschiedene Varianten zu testen und verschiedene Lösungen zu präsentieren. Dadurch können große musterbasierte Deployment-Modelle schneller zu ausführbaren verfeinert werden.

Für die Verfeinerung und Erkennung von musterbasierten Deployment-Modellen wird eine große Zahl an Verfeinerungs- bzw. Erkennungsmodellen benötigt. Wird beispielsweise ein Komponentenmuster, wovon mehrere Komponenten abhängen, von einem Verfeinerungsmodell zu einer konkreten Komponente verfeinert, können möglicherweise vorher anwendbare Verfeinerungsmodelle nicht mehr angewendet werden. Verfeinerungsmodelle werden auf ihre Anwendbarkeit geprüft, indem ein Subgraph-Verfahren eingesetzt wird. Sind die benötigten Elemente daher nicht mehr in der Topologie einer Anwendung enthalten, können die Verfeinerungsmodelle möglicherweise nicht mehr angewendet werden. Wird das bereits verfeinerte Komponentenmuster jedoch von einem Verfeinerungsmodell zu einer Komponente gleichen Typs verfeinert, so könnte das Verfeinerungsmodell dennoch angewendet werden. Zukünftige Arbeiten können dies umsetzten, indem sie beispielsweise Permutationen der Verfeinerungsmodelle generieren, wobei einzelne Komponentenmuster bereits mit den konkreten Komponenten verfeinert werden und erweitern somit die Anwendbarkeit der Verfeinerungsmodelle.

Im Allgemeinen können darüber hinaus zukünftig weitere Deployment-Technologien für die Instanzmodellgenerierung umgesetzt werden. Aktuell sind lediglich Plug-ins für Puppet, Kubernetes und Terraform implementiert. Des Weiteren wurden für die Instanzmodellvervollständigung bisher nur Plug-ins implementiert, die einige oft verwendete Technologien unterstützen. Um die Zahl der unterstützten Technologien zu steigern, können zukünftig weitere Plug-ins implementiert werden. Dadurch steigt auch die Zahl der Anwendungen, die von der LAUFFEUER-Methode verwaltet werden können.

Literaturverzeichnis

[AAG21] A. AbuHassan, M. Alshayeb, L. Ghouti. „Software smell detection techniques: A systematic literature review". In: *Journal of Software: Evolution and Process* 33.3 (2021), e2320 (Zitiert auf S. 54).

[ABD+18] M. Artač, 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)*. IEEE, 2018, S. 156–15609 (Zitiert auf S. 36).

[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. 52).

[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. 52).

[AGH+15] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, A. Yousafzai, F. Xia. „A survey on virtual machine migration and server consolidation frameworks for cloud data centers". In: *Journal of Network and Computer Applications* 52 (2015), S. 11–25 (Zitiert auf S. 15, 45, 145).

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Aug. 1977 (Zitiert auf S. 15, 48).

[AKR+19] A. P. Achilleos, K. Kritikos, A. Rosssini, G. M. Kapitsaki, J. Domschka, M. Orzechowski, D. Seybold, N. Nokolov, D. Romero, G. A. Papadopoulos. „The cloud application modelling and execution language“. In: *Journal of Cloud Computing* 8.20 (Dez. 2019) (Zitiert auf S. 35).

[Ama22] Amazon. *AWS CloudFormation*. Feb. 2022. URL: `https://aws.amazon.com/de/cloudformation/` (Zitiert auf S. 72).

[Ama23] Amazon. *AWS Solutions Constructs-Muster*. 2023. URL: `https://aws.amazon.com/de/solutions/constructs/patterns` (Zitiert auf S. 49).

[AZ05] P. Avgeriou, U. Zdun. „Architectural Patterns Revisited – A Pattern Language“. In: *In 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*. UVK - Universitaetsverlag Konstanz, Juli 2005 (Zitiert auf S. 50).

[AZ11] F. Arcelli 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. 53).

[Bar18] J. Barzen. „Wenn Kostüme sprechen – Musterforschung in den Digital Humanities am Beispiel vestimentärer Kommunikation im Film“. Dissertation. Universität zu Köln, Philosophische Fakultät, 2018 (Zitiert auf S. 48).

[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. 14, 18, 31, 34, 41).

[BBF09] G. Blair, N. Bencomo, R. B. France. „Models@run.time“. In: *Computer* 42.10 (2009), S. 22–27 (Zitiert auf S. 35, 55).

[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. 23, 168).

[BBH+22] F. Bühler, J. Barzen, L. Harzenetter, F. Leymann, P. Wundrack. „Combining the Best of Two Worlds: Microservices and Micro Frontends as Basis for a New Plugin Architecture“. In: *Proceedings of the 16th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2022)*. Springer, Okt. 2022, S. 3–23 (Zitiert auf S. 28).

[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. 18, 33, 41, 42, 63, 144).

[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. 38).

[BBKL13] 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. 22, 55, 157, 160, 161).

[BBKL14] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. „Vinothek - A Self-Service Portal for TOSCA“. In: *Proceedings of the 6th Central-European Workshop on Services and their Composition (ZEUS 2014)*. CEUR-WS.org, Feb. 2014, S. 69–72 (Zitiert auf S. 23).

[BCK03] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Apr. 2003 (Zitiert auf S. 14).

[BCS17] A. Brogi, P. Cifariello, J. Soldani. „DrACO: Discovering available cloud offerings“. In: *Computer Science - Research and Development* 32.3-4 (2017), S. 269–279 (Zitiert auf S. 55).

[BDR21a] O. Bibartiu, F. Dürr, K. Rothermel. „Clams: A Cloud Application Modeling Solution“. In: *Proceedings of the 2021 IEEE International Conference on Services Computing (SCC 2021)*. IEEE, Sep. 2021, S. 1–10 (Zitiert auf S. 36, 52).

[BDR21b] O. Bibartiu, F. Dürr, K. Rothermel. „Optimal Refinement for Component-based Architectures“. In: *Proceedings of the 2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC 2021)*. IEEE, Okt. 2021, S. 142–151 (Zitiert auf S. 52).

[BDV22] L. Baresi, E. Di Nitto, D. Vladušič. „The SODALITE Approach: An Overview“. In: *Deployment and Operation of Complex Software in Heterogeneous Execution Environments: The SODALITE Approach*. Springer, 2022, S. 9–21 (Zitiert auf S. 34).

[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. 23, 74).

[Béz05] J. Bézivin. „On the Unification Power of Models“. In: *Software and Systems Modeling (SoSyM)* 4.2 (Mai 2005), S. 171–188 (Zitiert auf S. 30, 37).

[BGS19] N. Bencomo, S. Götz, H. Song. „Models@run.time: a guided tour of the state of the art and research challenges“. In: *Software and Systems Modeling* 18.5 (2019), S. 3049–3082 (Zitiert auf S. 55).

[Bin15] T. Binz. „Crawling von Enterprise Topologien zur automatisierten Migration von Anwendungen: eine Cloud-Perspektive“. Dissertation. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, 2015 (Zitiert auf S. 22, 35, 69, 74, 157, 160, 161, 166).

[BKLW17] U. Breitenbücher, K. Képes, F. Leymann, M. Wurster. „Declarative vs. Imperative: How to Model the Automated Deployment of IoT Applications?“ In: *Proceedings of the 11th Advanced Summer School on Service Oriented Computing*. IBM Research Division, Nov. 2017, S. 18–27 (Zitiert auf S. 33, 41).

[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)*. IEEE, 2019, S. 422–429 (Zitiert auf S. 38).

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Okt. 1996 (Zitiert auf S. 48).

[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. 53).

[Bre16] U. Breitenbücher. „Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements“. Dissertation. Universität Stuttgart, Fakultät für Informatik, Elektrotechnik und Informationstechnik, 2016 (Zitiert auf S. 35, 41, 43, 69).

[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. 35).

[Cam23] Camunda Services GmbH. *Camunda*. 2023. URL: `https://camunda.com/` (Zitiert auf S. 33).

[CFH+05] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield. „Live migration of virtual machines“. In: *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*. USENIX Association, 2005, S. 273–286 (Zitiert auf S. 15, 45, 145).

[CFSV04] L. P. Cordella, P. Foggia, C. Sansone, M. Vento. „A (sub)graph Isomorphism Algorithm for Matching Large Graphs“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.10 (2004), S. 1367–1372 (Zitiert auf S. 111, 116).

[Cha08] D. Chappell. *What is Application Lifecycle Management*. White Paper. Dez. 2008 (Zitiert auf S. 15, 41).

[CL85] K. M. Chandy, L. Lamport. „Distributed Snapshots: Determining Global States of Distributed Systems“. In: *ACM Trans. Comput. Syst.* 3.1 (1985), S. 63–75 (Zitiert auf S. 45, 145).

[CSCM15] J. Cao, M. Simonin, G. Cooperman, C. Morin. „Checkpointing as a Service in Heterogeneous Cloud Environments“. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, Mai 2015, S. 61–70 (Zitiert auf S. 45, 145, 151).

[CT22] D. Calcaterra, O. Tomarchio. „Automated Generation of Application Management Workflows using TOSCA Policies“. In: *Proceedings of the 12th International Conference on Cloud Computing and Services Science - CLOSER*. SciTePress, 2022, S. 97–108 (Zitiert auf S. 44, 151).

[DCE15] B. Di Martino, G. Cretella, A. Esposito. „Cloud Services Composition Through Cloud Patterns“. In: *Adaptive Resource Management and Scheduling for Cloud Computing*. Springer, 2015, S. 128–140 (Zitiert auf S. 51).

[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. 51).

[DENM17] B. Di Martino, A. Esposito, S. Nacchia, S. A. Maisto. „A semantic model for business process patterns to support cloud deployment“. In: *Computer Science - Research and Development* 32.3 (2017), S. 257–267 (Zitiert auf S. 51).

[DMPS17] E. Di Nitto, P. Matthews, D. Petcu, A. Solberg, Hrsg. *Model-Driven Development and Operation of Multi-Cloud Applications: The MODAClouds Approach*. Springer, 2017 (Zitiert auf S. 154).

[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. 34, 41).

[Doc23] Docker Inc. *docker checkpoint*. 2023. URL: `https://docs.docker.com/engine/reference/commandline/checkpoint` (Zitiert auf S. 45).

[DP09] S. Ducasse, D. Pollet. „Software Architecture Reconstruction: A Process-Oriented Taxonomy“. In: *IEEE Transactions on Software Engineering* 35.4 (2009), S. 573–591 (Zitiert auf S. 56).

[DV22] E. Di Nitto, D. Vladušič. „Orchestrating Heterogeneous Applications: Motivation and State of the Art“. In: *Deployment and Operation of Complex Software in Heterogeneous Execution Environments: The SODALITE Approach*. Springer, 2022, S. 1–8 (Zitiert auf S. 34).

[DZZ12] R. Di Cosmo, S. Zacchiroli, G. Zavattaro. „Towards a Formal Component Model for the Cloud“. In: *Software Engineering and Formal Methods*. Springer, 2012, S. 156–171 (Zitiert auf S. 34).

[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. 14, 19, 31).

[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. 43).

[EJBE19] A. Elhabbash, A. Jumagaliyev, G. S. Blair, Y. Elkhatib. „SLO-ML: A Language for Service Level Objective Modelling in Multi-Cloud Applications“. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. UCC'19. Auckland, New Zealand: Association for Computing Machinery, 2019, S. 241–250 (Zitiert auf S. 35).

[ELSC13] I. P. Egwutuoha, D. Levy, B. Selic, S. Chen. „A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems“. In: *The Journal of Supercomputing* 65.3 (Sep. 2013), S. 1302–1326 (Zitiert auf S. 45, 145).

[EME+06] K. El Maghraoui, A. Meghranjani, T. Eilam, M. Kalantar, A. Konstantinou. „Model Driven Provisioning: Bridging the Gap Between Declarative Object Models and Procedural Provisioning Tools“. In: *Proceedings of the 7th International Middleware Conference (Middleware 2006)*. Springer, Nov. 2006, S. 404–423 (Zitiert auf S. 42).

[EYG97] A. Eden, A. Yehudai, J. Gil. „Precise Specification and Automatic Application of Design Patterns“. In: *Proceedings of the 12th IEEE International Conference Automated Software Engineering (ASE 1997)*. IEEE, Nov. 1997, S. 143–152 (Zitiert auf S. 52).

[FAB+11] M. Farwick, B. Agreiter, R. Breu, S. Ryll, K. Voges, I. Hanschke. „Automation Processes for Enterprise Architecture Management“. In: *EDOC 2011*. IEEE, 2011, S. 340–349 (Zitiert auf S. 55).

[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. 50, 133).

[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. 50, 133).

[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. 48, 52, 133).

[FBBL17] M. Falkenthal, J. Barzen, U. Breitenbücher, F. Leymann. „Solution Languages: Easing Pattern Composition in Different Domains“. In: *International Journal on Advances in Software* 10.3&4 (Dez. 2017), S. 263–274 (Zitiert auf S. 48, 50, 61).

[FBF+18] G. Falazi, U. Breitenbücher, M. Falkenthal, L. Harzenetter, F. Leymann, V. Yussupov. „Blockchain-based Collaborative Development of Application Deployment Models“. In: *On the Move to Meaningful Internet Systems: OTM 2018 Conferences (CoopIS 2018)*. Springer, Okt. 2018, S. 40–60 (Zitiert auf S. 25).

[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. 50).

[FCS+18] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, A. Solberg. „CloudMF: Model-Driven Management of Multi-Cloud Applications“. In: *ACM Transactions on Internet Technology* 18.2 (Mai 2018), S. 1–24 (Zitiert auf S. 34).

[FHS+21] A. Fischer, L. Harzenetter, P. Schildkamp, U. Breitenbücher, C. Neuefeind, F. Leymann, B. Mathiak. „Modeling as a Sustainability Strategy for DH Software Applications“. In: *Book of Abstracts of the 2nd International Conference of the European Association for Digital Humanities (EDAH 2021)*. ADHO, Sep. 2021 (Zitiert auf S. 27).

[FL17] M. Falkenthal, F. Leymann. „Easing Pattern Application by Means of Solution Languages“. In: *Proceedings of the Ninth International Conference on Pervasive Patterns and Applications (PATTERNS)*. Xpert Publishing Services, Feb. 2017, S. 58–64 (Zitiert auf S. 48, 50).

[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. 36, 54).

[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. 15, 45, 49, 50, 60, 83–87, 178, 179).

[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. 34).

[FRH15] F. Fittkau, S. Roth, W. Hasselbring. „ExplorViz: Visual runtime behavior analysis of enterprise application landscapes“. In: *Proceedings of the 23rd European Conference on Information Systems (ECIS 2015)*. AIS, 2015 (Zitiert auf S. 55).

[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. 34).

[GGTP19] M. Guerriero, M. Garriga, D. A. Tamburri, F. Palomba. „Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry“. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, S. 580–589 (Zitiert auf S. 154).

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Okt. 1994 (Zitiert auf S. 48, 49, 83).

[GL18] J. Guth, F. Leymann. „Towards Pattern-based Rewrite and Refinement of Application Architectures“. In: *Papers From the 12th Advanced Summer School on Service-Oriented Computing (SummerSOC’18)*. IBM Research Division, Okt. 2018, S. 90–100 (Zitiert auf S. 52).

[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. 52).

[GMMC13a] J. Guillén, J. Miranda, J. M. Murillo, C. Canal. „A service-oriented framework for developing cross cloud migratable software“. In: *The Journal of Systems and Software* 86.9 (Sep. 2013), S. 2294–2308 (Zitiert auf S. 36).

[GMMC13b] J. Guillén, J. Miranda, J. Murillo, C. Canal. „A UML Profile for Modeling Multicloud Applications“. In: *Proceedings of the Second European Conference on Service-Oriented and Cloud Computing (ESOCC 2013)*. Springer, Sep. 2013, S. 180–187 (Zitiert auf S. 36).

[GRV+22] J. Gorroñogoitia, D. Radolović, Z. Vasileiou, G. Meditskos, A. Karakostas, S. Vrochidis, M. Bachras. „The SODALITE Model-Driven Approach“. In: *Deployment and Operation of Complex Software in Heterogeneous Execution Environments: The SODALITE Approach*. Springer, 2022, S. 23–52 (Zitiert auf S. 34).

[Has23] HashiCorp. *Terraform*. 2023. URL: `https://www.terraform.io/` (Zitiert auf S. 14, 72).

[HBB+21] L. Harzenetter, T. Binz, U. Breitenbücher, F. Leymann, M. Wurster. „Automated Generation of Management Workflows for Running Applications by Deriving and Enriching Instance Models“. In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*. SciTePress, Mai 2021, S. 99–110 (Zitiert auf S. 24, 135, 153, 167).

[HBBL23] L. Harzenetter, U. Breitenbücher, T. Binz, F. Leymann. „An Integrated Management System for Composed Applications Deployed by Different Deployment Automation Technologies“. In: *SN Computer Science* 4.370 (Apr. 2023), S. 1–16 (Zitiert auf S. 25, 135, 153, 155, 167).

[HBF+18] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, C. Krieger, F. Leymann. „Pattern-based Deployment Models and Their Automatic Execution“. In: *Proceedings of the 11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2018)*. IEEE Computer Society, Dez. 2018, S. 41–52 (Zitiert auf S. 24, 91, 167).

[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. 24, 91, 167).

[HBF+21] L. Harzenetter, U. Breitenbücher, G. Falazi, F. Leymann, A. Wersching. „Automated Detection of Design Patterns in Declarative Deployment Models“. In: *Proceedings of the 14th IEEE/ACM International Conference on Utility Cloud Computing (UCC 2021)*. ACM, Dez. 2021, S. 36–45 (Zitiert auf S. 24, 91, 167).

[HBKL19] L. Harzenetter, U. Breitenbücher, K. Képes, F. Leymann. „Freezing and Defrosting Cloud Applications: Automated Saving and Restoring of Running Applications“. In: *Software-Intensive Cyber-Physical Systems (SICS)* 35 (Aug. 2019), S. 101–114 (Zitiert auf S. 24, 135, 144, 167).

[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 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC 2019)*. IEEE, Okt. 2019, S. 216–225 (Zitiert auf S. 24, 135, 167).

[HBLE14] H. Holm, M. Buschle, R. Lagerström, M. Ekstedt. „Automatic data collection for enterprise architecture models“. In: *Software and Systems Modeling* 13.2 (2014), S. 825–841 (Zitiert auf S. 22, 55, 157, 166).

[HHW+09] A. van Hoorn, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, D. Kieselhorst. *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework*. Kiel, 2009 (Zitiert auf S. 55).

[HKF18] M. Horii, Y. Kojima, K. Fukuda. „Stateful process migration for edge computing applications“. In: *2018 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2018, S. 1–6 (Zitiert auf S. 15, 45).

[HS09] J. Hallstrom, N. Soundarajan. „Reusing patterns through design refinement“. In: *International Conference on Software Reuse*. Springer, 2009, S. 225–235 (Zitiert auf S. 52).

[HW04] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004 (Zitiert auf S. 50, 83, 85, 86).

[HZ14] T. Haitzer, U. Zdun. „Semi-automated architectural abstraction specifications for supporting software evolution“. In: *Science of Computer Programming* 90 (2014). Special Issue on Component-Based Software Engineering and Software Architecture, S. 135–160 (Zitiert auf S. 56).

[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. 53).

[INS+14] C. Inzinger, S. Nastic, S. Sehic, M. Vögler, F. Li, S. Dustdar. „MADCAT: A Methodology for Architecture and Deployment of Cloud Application Topologies“. In: *2014 IEEE 8th International Symposium on Service Oriented System Engineering*. IEEE, 2014, S. 13–22 (Zitiert auf S. 31).

[ISO11] ISO/IEC/IEEE. *ISO/IEC/IEEE 42010:2011(E): Systems and software engineering — Architecture description*. Standard. Geneva, CH, 2011 (Zitiert auf S. 31).

[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. 53).

[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. 23, 168).

[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. 18).

[KBL17] K. Képes, U. Breitenbücher, F. Leymann. „Integrating IoT Devices Based on Automatically Generated Scale-Out Plans“. In: *2017 IEEE 10th Conference on Service-Oriented Computing and Applications, SOCA 2017, 22-25 November 2017, Kanazawa, Japan*. IEEE Computer Society, 2017, S. 155–163 (Zitiert auf S. 18, 44).

[KHW+04] A. Keller, J. L. Hellerstein, J. L. Wolf, K.-L. Wu, V. Krishnan. „The CHAMPS System: Change Management with Planning and Scheduling“. In: *Proceedings of the 10th Network Operations and Management Symposium (NOMS 2004)*. IEEE, Apr. 2004, S. 395–408 (Zitiert auf S. 42).

[KKM+19] T. Kuroda, T. Kuwahara, T. Maruyama, K. Satoda, H. Shimonishi, T. Osaki, K. Matsuda. „Weaver: A Novel Configuration Designer for IT/NW Services in Heterogeneous Environments“. In: *2019 IEEE*

Global Communications Conference (GLOBECOM). 2019, S. 1–6 (Zitiert auf S. 35).

[KLZ20] K. Képes, F. Leymann, M. Zimmermann. „Situation-Aware Updates for Cyber-Physical Systems“. In: *Proceedings of the 14th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2020)*. Springer, Dez. 2020, S. 12–32 (Zitiert auf S. 44, 64).

[KR00] S. Kalaiselvi, V. Rajaraman. „A survey of checkpointing algorithms for parallel and distributed computers“. In: *Sadhana* 25.5 (Okt. 2000), S. 489–510 (Zitiert auf S. 45, 145, 151).

[LB21] F. Leymann, J. Barzen. „Pattern Atlas“. In: Springer, Apr. 2021, S. 67–76 (Zitiert auf S. 23, 49, 168).

[Le 98] D. Le Métayer. „Describing software architecture styles using graph grammars“. In: *IEEE Transactions on Software Engineering* 24.7 (1998), S. 521–533 (Zitiert auf S. 31, 37).

[Leh18] S. M. Lehrig. „Efficiently Conducting Quality-of-Service Analyses by Templating Architectural Knowledge“. Dissertation. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, und Information Technology, 2018 (Zitiert auf S. 52).

[Ley09] F. Leymann. „Cloud Computing: The Next Revolution in IT“. In: *Proceedings of the 52th Photogrammetric Week*. Wichmann Verlag, Sep. 2009, S. 3–12 (Zitiert auf S. 14).

[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. 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. 52).

[LR00] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000 (Zitiert auf S. 64).

[LR09] K. Levanti, A. Ranganathan. „Planning-based Configuration and Management of Distributed Systems“. In: *Proceedings of the 11th IFIP/IEEE International Symposium on Integrated Network Management (IM 2009)*. IEEE, Juni 2009, S. 65–72 (Zitiert auf S. 42).

[LSS+13] H. Lu, M. Shtern, B. Simmons, M. Smit, M. Litoiu. „Pattern-Based Deployment Service for Next Generation Clouds“. In: *Proceedings of the IEEE Ninth World Congress on Services (SERVICES 2013)*. IEEE, Juni 2013, S. 464–471 (Zitiert auf S. 42).

[MDW+00] V. Machiraju, M. Dekhil, K. Wurster, P. K. Garg, M. L. Griss, J. Holland. „Towards Generic Application Auto-Discovery“. In: *Proceedings of the 7th IEEE/IFIP Network Operations and Management Symposium (NOMS 2000)*. IEEE, Apr. 2000, S. 75–87 (Zitiert auf S. 22, 55, 157).

[Mic23] Microsoft Corporation. *Cloud Design Patterns*. 2023. URL: `https://docs.microsoft.com/en-us/azure/architecture/patterns` (Zitiert auf S. 49).

[MKLT13] M. Menzel, M. Klems, H. A. Le, S. Tai. „A configuration crawler for virtual appliances in compute clouds“. In: *2013 IEEE International Conference on Cloud Engineering (IC2E)*. 2013, S. 201–209 (Zitiert auf S. 55).

[MSUW02] S. J. Mellor, K. Scott, A. Uhl, D. Weise. „Model-Driven Architecture“. In: *Advances in Object-Oriented Information Systems*. Springer, Sep. 2002, S. 290–297 (Zitiert auf S. 30, 38).

[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. 36, 42).

[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. 50).

[NEP+16] R. Nyrén, A. Edmonds, A. Papaspyrou, T. Metsch, B. Parák. *Open Cloud Computing Interface – Core*. Techn. Ber. Open Grid Froum, 2016 (Zitiert auf S. 39).

[NHS+18] C. Neuefeind, L. Harzenetter, P. Schildkamp, U. Breitenbücher, B. Mathiak, J. Barzen, F. Leymann. „The SustainLife Project – Living Systems in Digital Humanities“. In: *Papers From the 12th Advanced Summer School of Service-Oriented Computing (SummerSOC 2018)*. IBM Research Division, Okt. 2018, S. 101–112 (Zitiert auf S. 26).

[NL13] A. Nowak, F. Leymann. „Green Business Process Patterns - Part II“. In: *Proceedings of the 6th IEEE International Conference on Service Oriented Computing & Applications (SOCA 2013)*. IEEE, Dez. 2013, S. 168–173 (Zitiert auf S. 51).

[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. 51).

[NP22] J. Nicacio, F. Petrillo. „An Approach to Build Consistent Software Architecture Diagrams Using Devops System Descriptors“. In: MODELS ’22. Montreal, Quebec, Canada: ACM, Okt. 2022, S. 312–321 (Zitiert auf S. 56).

[NSM+18] C. Neuefeind, P. Schildkamp, B. Mathiak, J. Barzen, U. Breitenbücher, L. Harzenetter, F. Leymann. „Lebende Systeme in den Digital Humanities – das Projekt SustainLife“. In: *20. Workshop Software-Reengineering und -Evolution (WSRE 2018) der GI-Fachgruppe Software-Reengineering, Bad-Honnef, 02.-04. Mai 2018, Proceedings*. GI Gesellschaft für Informatik e.V. (GI), Mai 2018, S. 37–38 (Zitiert auf S. 27).

[NSM+19a] C. Neuefeind, P. Schildkamp, B. Mathiak, L. Harzenetter, J. Barzen, U. Breitenbücher, F. Leymann. „Technologienutzung im Kontext Digitaler Editionen. Eine Landschaftsvermessung“. In: *DHd 2019 Digital Humanities: multimedial & multimodal. Konferenzabstracts.* Zenodo, März 2019, S. 219–222 (Zitiert auf S. 27).

[NSM+19b] C. Neuefeind, P. Schildkamp, B. Mathiak, A. Marčić, F. Hentschel, L. Harzenetter, J. Barzen, U. Breitenbücher, F. Leymann. „Sustaining the Musical Competitions Database: A TOSCA-based Approach to Application Preservation in the Digital Humanities“. In: *Book of Abstracts of the Digital Humanities Conference 2019 (DH2019), Utrecht, 9.7. – 12.7.2019*. ADHO, Juli 2019 (Zitiert auf S. 27).

[NZP+21] E. Ntentos, U. Zdun, K. Plakidas, P. Genfer, S. Geiger, S. Meixner, W. Hasselbring. „Detector-based component model abstraction for microservice-based systems“. In: *Computing* 103.11 (2021), S. 2521–2551 (Zitiert auf S. 56).

[OAS07] 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. 32).

[OAS13a] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS). 2013 (Zitiert auf S. 41).

[OAS13b] 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. 33, 34, 42, 46, 72, 174).

[OAS20] OASIS. *TOSCA Simple Profile in YAML Version 1.3*. Organization for the Advancement of Structured Information Standards (OASIS). 2020 (Zitiert auf S. 33, 41, 46, 72, 174).

[OBT+22] S. E. Ooi, R. Beuran, Y. Tan, T. Kuroda, T. Kuwahara, N. Fujita. „SecureWeaver: Intent-Driven Secure System Designer“. In: Sat-CPS ’22. New York, NY, USA: ACM, 2022, S. 107–116 (Zitiert auf S. 35).

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

[OMG10] OMG. *The MDA Foundation Model*. 2010 (Zitiert auf S. 30, 37).

[OMG11] OMG. *Business Process Model and Notation (BPMN) Version 2.0*. Object Management Group (OMG). 2011 (Zitiert auf S. 32).

[OMG14] OMG. *Model Driven Architecture (MDA) MDA Guide rev. 2.0*. 2014 (Zitiert auf S. 30).

[Opp03] D. Oppenheimer. „The importance of understanding distributed system configuration“. In: *Proceedings of the 2003 Conference on Human Factors in Computer Systems Workshop (CHI 2003)*. ACM, Apr. 2003 (Zitiert auf S. 14).

[Pro23] Progress Software Corporation. *Chef*. 2023. URL: `https://chef.io` (Zitiert auf S. 33, 72).

[Pup23] Puppet. *Puppet*. 2023. URL: https://puppet.com/ (Zitiert auf S. 14).

[Rac23] Rackspace Cloud Computing. *OpenStack Heat*. 2023. URL: https://docs.openstack.org/heat/ (Zitiert auf S. 72).

[RBF+16] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. „Internet of Things Patterns“. In: *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, Juli 2016, S. 1–21 (Zitiert auf S. 49).

[RBF+17a] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. „Internet of Things Patterns for Devices“. In: *Proceedings of the Ninth international Conferences on Pervasive Patterns and Applications (PATTERNS 2017)*. Xpert Publishing Services, Feb. 2017, S. 117–126 (Zitiert auf S. 49).

[RBF+17b] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. „Internet of Things Patterns for Devices: Powering, Operating, and Sensing“. In: *International Journal on Advances in Internet Technology* 10.3&4 (Dez. 2017), S. 106–123 (Zitiert auf S. 49).

[Red23] Red Hat, Inc. *Ansible*. 2023. URL: https://ansible.com (Zitiert auf S. 33, 72).

[Rei13] R. Reiners. „An Evolving Pattern Library for Collaborative Project Documentation“. Diss. RWTH Aachen University, 2013 (Zitiert auf S. 48).

[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)*. ACM, 2018, S. 257–268 (Zitiert auf S. 36).

[SB03] D. Schmidt, F. Buschmann. „Patterns, frameworks, and middleware: their synergistic relationships“. In: *25th International Conference on Software Engineering, 2003. Proceedings.* 2003, S. 694–704 (Zitiert auf S. 48).

[SBB+22] M. Stötzner, S. Becker, U. Breitenbücher, K. Képes, F. Leymann. „Modeling Different Deployment Variants of a Composite Application in a Single Declarative Deployment Model“. In: *Algorithms* 15.10 (2022) (Zitiert auf S. 35).

[SBB+23] J. Soldani, U. Breitenbücher, A. Brogi, L. Frioli, F. Leymann, M. Wurster. „Tailoring Technology-Agnostic Deployment Models to Production-Ready Deployment Technologies“. In: *Cloud Computing and Services Science*. Springer, 2023, S. 1–24 (Zitiert auf S. 39).

[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. 25, 54).

[SBKL19] 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. 54).

[Sch06] D. C. Schmidt. „Model-driven engineering“. In: *Computer* 39.2 (Feb. 2006), S. 25–31 (Zitiert auf S. 30).

[Sch95] A. Schürr. „Specification of graph translators with triple graph grammars“. In: *Graph-Theoretic Concepts in Computer Science*. Springer, 1995, S. 151–163 (Zitiert auf S. 37, 40, 61).

[Sel03] B. Selic. „The pragmatics of model-driven development“. In: *IEEE Software* 20.5 (2003), S. 19–25 (Zitiert auf S. 30).

[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. 50, 83, 86, 87).

[SHB+20] P. Schildkamp, L. Harzenetter, U. Breitenbücher, F. Leymann, B. Mathiak, C. Neuefeind. „Modellierung und Verwaltung von DH-Anwendungen in TOSCA“. In: *DHd 2020 Spielräume: Digital Humanities zwischen Modellierung und Interpretation. Konferenzabstracts*. Zenodo, Feb. 2020, S. 36–38 (Zitiert auf S. 27).

[SHM+20] P. Schildkamp, L. Harzenetter, B. Mathiak, C. Neuefeind, J. Barzen, U. Breitenbücher, F. Leymann. „Workshop on Modelling and Maintaining Research Applications in TOSCA“. In: *DH2020*. ADHO, Juli 2020 (Zitiert auf S. 27).

[SIA17] J. Sandobalin, E. Insfran, S. Abrahao. „An Infrastructure Modelling Tool for Cloud Provisioning“. In: *2017 IEEE International Conference on Services Computing (SCC)*. 2017, S. 354–361 (Zitiert auf S. 35).

[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. 38).

[SPP+22] C. Saravanakumar, R. Priscilla, B. Prabha, A. Kavitha, M. Prakash, C. Arun. „An Efficient On-Demand Virtual Machine Migration in Cloud Using Common Deployment Model“. In: *Computer Systems Science and Engineering* 42.1 (2022), S. 245–256 (Zitiert auf S. 15, 45).

[SSMJ04] A. Sahai, S. Singhal, V. Machiraju, R. Joshi. „Automated generation of resource configurations through policies“. In: *Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004.* Juni 2004, S. 107–110 (Zitiert auf S. 42).

[St00] R. Soley, the OMG Staff Strategy Group. „Model driven architecture“. In: *OMG white paper* 308 (Nov. 2000) (Zitiert auf S. 30).

[The23] The Linux Foundation. *Kubernetes*. 2023. URL: `https://kubernetes.io` (Zitiert auf S. 14, 33, 45, 72).

[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)*. IEEE, 2016, S. 262–267 (Zitiert auf S. 53).

[Tur14] J. Turnbull. *The Docker Book*. James Turnbull, Juli 2014 (Zitiert auf S. 45).

[VRB11] L. M. Vaquero, L. Rodero-Merino, R. Buyya. „Dynamically Scaling Applications in the Cloud“. In: *SIGCOMM Comput. Commun. Rev.* 41.1 (Jan. 2011), S. 45–52 (Zitiert auf S. 44).

[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. 19, 23, 25, 39, 72).

[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, Dez. 2020, S. 86–103 (Zitiert auf S. 48, 50).

[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, März 2020, S. 97–112 (Zitiert auf S. 19, 40, 72).

[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. 39, 56, 63, 72, 157).

[WBBC16] D. Weerasiri, M. C. Barukh, B. Benatallah, J. Cao. „A Model-Driven Framework for Interoperable Cloud Resources Management“. In: *ICSOC 2016: Service-Oriented Computing*. Springer, 2016, S. 186–201 (Zitiert auf S. 39).

[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. 14, 19, 33, 34, 72, 73).

[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. 25, 91, 183).

[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, Aug. 2020, S. 138–146 (Zitiert auf S. 26, 40, 72).

[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. 26, 39, 40, 72, 174).

[WBK+17] S. Wagner, U. Breitenbücher, O. Kopp, A. Weiß, F. Leymann. „Fostering the Reuse of TOSCA-based Applications by Merging BPEL Management Plans". In: *Cloud Computing and Services Science: 6th International Conference (CLOSER 2016) - Revised Selected Papers*. Bd. 740. Communications in Computer and Information Science. Springer, Juli 2017, S. 232–254 (Zitiert auf S. 44).

[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, Juni 2020, S. 20–35 (Zitiert auf S. 42).

[WBKL18] M. Wurster, U. Breitenbücher, O. Kopp, F. Leymann. „Modeling and Automated Execution of Application Deployment Tests“. In: *Proceedings of the IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC 2018)*. IEEE Computer Society, Okt. 2018, S. 171–180 (Zitiert auf S. 44, 144).

[WBL16] S. Wagner, U. Breitenbücher, F. Leymann. „A Method For Reusing TOSCA-based Applications and Management Plans“. In: *Proceedings of the 6th International Conference on Cloud Computing and Service Science (CLOSER 2016)*. Rome: SciTePress, Apr. 2016, S. 181–191 (Zitiert auf S. 44).

[WBSB22] M. Weller, U. Breitenbücher, S. Speth, S. Becker. „The Deployment Model Abstraction Framework“. In: *Lecture Notes in Computer Science (LNCS)*. Springer, Okt. 2022 (Zitiert auf S. 56).

[Wil22] K. Wild. „Eine Methode zum Verteilen, Adaptieren und Deployment partnerübergreifender Anwendungen“. Dissertation. Universität Stuttgart, Fakultät für Informatik, Elektrotechnik und Informationstechnik, 2022 (Zitiert auf S. 35, 74).

[YBB+22] V. Yussupov, U. Breitenbücher, A. Brogi, L. Harzenetter, F. Leymann, J. Soldani. „Serverless or Serverful? A Pattern-based Approach for Exploring Hosting Alternatives“. In: *Proceedings of the 16th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2022)*. Springer, Okt. 2022 (Zitiert auf S. 26, 52, 83, 85, 86, 91, 132, 133, 184).

[YBK+20] V. Yussupov, U. Breitenbücher, C. Krieger, F. Leymann, J. Soldani, M. Wurster. „Pattern-based Modelling, Integration, and Deployment of Microservice Architectures“. In: *Proceedings of the 24th International Enterprise Distributed Object Computing Conference (EDOC 2020)*. IEEE, Okt. 2020, S. 40–50 (Zitiert auf S. 51).

[YSB+21] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, F. Leymann. „From Serverful to Serverless: A Spectrum of Patterns for Hosting Application Components“. In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*. SciTePress, Mai 2021, S. 268–279 (Zitiert auf S. 83, 87, 184).

[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. 52).

[ZBH+20] M. Zimmermann, U. Breitenbücher, L. Harzenetter, F. Leymann, V. Yussupov. „Self-Contained Service Deployment Packages“. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*. SciTePress, Mai 2020, S. 371–381 (Zitiert auf S. 26, 46, 64).

[ZBKL18] M. Zimmermann, U. Breitenbücher, C. Krieger, F. Leymann. „Deployment Enforcement Rules for TOSCA-based Applications“. In: *Proceedings of The Twelfth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2018)*. Xpert Publishing Services, Sep. 2018, S. 114–121 (Zitiert auf S. 54).

[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. 50).

[ZHD07] U. Zdun, C. Hentrich, S. Dustdar. „Modeling Process-Driven and Service-Oriented Architectures Using Patterns and Pattern Primitives“. In: *ACM Trans. Web* 1.3 (Sep. 2007), 14–es (Zitiert auf S. 52).

[ZKE+22] F. Zalila, F. Korte, J. Erbel, S. Challita, J. Grabowski, P. Merle. „MoDMaCAO: a model-driven framework for the design, validation and configuration management of cloud applications based on OCCI“. In: *Software and Systems Modeling* (Sep. 2022) (Zitiert auf S. 39).

[ZZGL08] O. Zimmermann, U. Zdun, T. Gschwind, F. Leymann. „Combining Pattern Languages and Architectural Decision Models in a Comprehensive and Comprehensible Design Method“. In: *Seventh Working IEEE/IFIP Conference on Software Architecture - WICSA 2008*. IEEE, Feb. 2008, S. 156–166 (Zitiert auf S. 50).

Alle Links wurden zuletzt am 18. Februar 2023 geprüft.

Abbildungsverzeichnis

1.1 Übersicht der Forschungsbeiträge 17

2.1 Deklarative und imperative Deployment-Modelle 32
2.2 Illustration einer Triple Graph Grammar 38

3.1 Übersicht über die Schritte der Lauffeuer-Methode 61

4.1 Darstellung des Metamodells für die musterbasierte Modellierung von Anwendungen 73
4.2 Ein Beispiel für ein musterbasiertes Deployment-Modell . . . 88

5.1 Prozess zur Verfeinerung und Erkennung von Mustern 93
5.2 Transformation eines musterbasierten Deployment-Modells ohne Verhaltensmuster. 100
5.3 Transformation eines musterbasierten Deployment-Modells mit Verhaltensmuster. 105
5.4 PRM zur Verfeinerung und Erkennung eines Message Channels, welcher auf einer Public Cloud bereitstellt wird . . . 108
5.5 PRM zur Verfeinerung einer Java 11 App auf einem Execution Environment in einer Private Cloud 109

5.6 Eine mögliche Verfeinerung des in Abbildung 4.2 dargestellten musterbasierten Deployment-Modells 128

6.1 Erweiterung von ausführbaren Deployment-Modellen um zusätzliche Managementfunktionalitäten 137
6.2 Beispiele für Managementtypen 140
6.3 Einfrieren und Auftauen einer Anwendung 146
6.4 Auswahl der Freeze- und Defrost-Plan Operationen 148

7.1 Übersicht über das Integrierte Managementsystem für zusammengesetzte Anwendungen, adaptiert von [HBBL23] . . 155
7.2 Ein Beispiel für ein vervollständigtes Instanzmodell einer laufenden Anwendung 163

8.1 Grobarchitektur des Lauffeuer-Frameworks 169
8.2 Die Komponenten des erweiterten Modellierungswerkzeugs . 170
8.3 Die Komponenten des OpenTOSCA Containers, der Workflowgenerierung und Ausführungsumgebung 172
8.4 Die Komponenten der Instanzmodellgenerierung 173
8.5 Verfeinerung eines musterbasierten Deployment-Modells . . 177
8.6 Beispiel eines musterbasiertes Deployment-Modells nach der Anwendung des ersten Verfeinerungsmodells 178
8.7 Beispiel eines weiteren musterbasierten Deployment-Modells 179
8.8 Eine mögliche Verfeinerung des MDMs aus Abbildung 8.7 . . 180
8.9 Instanzmodell des angepassten Sock Shops Demoanwendung nach dem Instanzmodellnormalisierungschritt 181
8.10 Instanzmodell des angepassten Sock Shops Demoanwendung nach dem Instanzmodellvervollständigungsschritt 182

Mengenverzeichnis

Symbol	Kurzbeschreibung
A_t	Menge aller Artefakte eines MDMs t $A_t \subseteq \Sigma^+ \times \Sigma^+$ $a_i = (Name, Referenz) \in A_t$
AZ_v	Menge aller Artefaktzuordnungen eines PRMs v $AZ_v \subseteq MK_{MusterS_v} \times MK_{LösungsS_v} \times AT_{MusterS_v}$ $az_i = (MusterE, LösungsE, Artefakttyp) \in AZ_v$
AT_t	Menge aller Artefakttypen eines MDMs t $AT_t \subseteq \Sigma^+$ $at_i = (Name) \in AT_t$
$\mathcal{B}$	Menge der booleschen Wahrheitswerte $\mathcal{B} = \{wahr, falsch\}$
$\mathcal{D}$	Menge der Richtungen einer Relation $\mathcal{D} = \{eingehend, ausgehend\}$
E_t	Menge aller Eigenschaften eines MDMs t $E_t \subseteq \Sigma^+ \times \Sigma^*$ $e_i = (Schlüssel, Wert) \in E_t$

EZ_v Menge aller Eigenschaftszuordnungen eines PRMs v
$EZ_v \subseteq ME_{MusterS_v} \times \Sigma^+ \times ME_{LösungsS_v} \times \Sigma^+$
$ez_i = (MusterE, MusterProp, LösungsE, LösungsProp)$

$GZ_{a,b}$ Menge aller Graphzuordnungen, wobei b in a enthalten ist

K_t Menge aller Komponenten eines MDMs t
$K_t \subseteq \Sigma^+$
$k_i = (Name) \in K_t$

KM_t Menge aller Komponentenmuster eines MDMs t
$KM_t \subseteq \Sigma^+$
$km_i = (Name) \in KM_t$

KMT_t Menge aller Komponentenmustertypen eines MDMs t
$KMT_t \subseteq \Sigma^+$
$kmt_i = (Name) \in KMT_t$

KT_t Menge aller Komponententypen eines MDMs t
$KT_t \subseteq \Sigma^+ \times \mathcal{B} \times \mathcal{B}$
$kt_i = (Name, Abstrakt, Angereichert) \in KT_t$

M_t Menge aller Modellentitäten eines MDMs t
$M_t = ME_t \cup MET_t$

ME_t Menge aller Modellelemente eines MDMs t
$ME_t = K_t \cup R_t$

MET_t Menge aller Modellelementtypen eines MDMs t
$MET_t = KT_t \cup RT_t \cup MGT_t \cup AT_t$

MGT_t Menge aller Managementtypen eines MDMs t
$MGT_t \subseteq \Sigma^+ \times \Sigma^+ \times KT_t \times \wp(KT_t) \times \wp(\Sigma^+)$
$mgt_i = (Id, Feature, FeatureFür, Benötigt, Technologien) \in MGT_t$

MK_t Menge aller Modellknoten eines MDMs t
$MK_t = K_t \cup KM_t$

MT_t Menge aller Mustertypen eines MDMs t
$MT_t \subseteq KMT_t \cup VMT_t$

O_t	Menge aller Operationen eines MDMs t $O_t \subseteq \Sigma^+ \times \wp(Parameter_t) \times \wp(Parameter_t)$ $o_i = (Name, Eingabe, Ausgabe) \in O_t$
P_t	Menge aller Musterentitäten eines MDMs t $P_t \subseteq KM_t \cup VM_t \cup MT_t$
$Parameter_o$	Parameter einer Operation $o \in O_t$ $Parameter_t \subseteq \Sigma^+ \times \Sigma^* \times \mathcal{B}$ $param_i = (Schlüssel, Wert, benötigt) \in Parameter_t$
R_t	Menge aller Relationen eines MDMs t $R_t \subseteq \Sigma^+ \times MK_t \times MK_t$ $r_i = (Id, Quelle, Ziel) \in R_t$
RZ_v	Menge aller Relationszuordnungen eines PRMs v $RZ_v \subseteq MK_{MusterS_v} \times RT_{MusterS_v} \times \mathcal{D} \times MK_{LösungsS_v} \times \{KT_{MusterS_v} \cup MT_{MusterS_v} \cup \perp\}$ $rz_i = (MusterE, Relationstyp, Richtung, LösungsE, Endpunkttyp) \in RZ_v$
RT_t	Menge aller Relationstypen eines MDMs t $RT_t \subseteq \Sigma^+ \times \mathcal{B}$ $rt_i = (Name, Abstrakt) \in RT_t$
SE_t	Menge aller Strukturelemente eines MDMs t $SE_t \subseteq ME_t \cup KM_t$
$SG^{a,b}$	Subgraph – Menge aller Strukturelementzuordnungen sem_i eines Subgraphen, der dem MDM b entspricht und in dem MDM a enthalten ist. $sem_i = (Entsprechung, Suchelement) \in SG_k^{a,b} \in GZ_{a,b}$
BZ_V	Menge aller Bleibzuordnungen eines PRMs v $BZ_v \subseteq MK_{MusterS_v} \times MK_{LösungsS_v}$ $bz_i = (MusterE, LösungsE) \in BZ_v$
$\mathcal{T}$	Menge aller musterbasierten Deployment-Modelle (MDMs)

$\mathcal{V}$	Menge aller Verfeinerungsmodelle (PRMs)
VM_t	Menge aller Verhaltensmuster eines MDMs t $VM_t \subseteq \Sigma^+$ $vm_i = (Id) \in VM_t$
VZ_v	Menge aller Verhaltenszuordnungen eines PRMs v $VZ_v \subseteq VM_{MusterS_v} \times ME_{LösungsS_v} \times \Sigma^+$ $vz_i = (MusterE, LösungsE, Eigenschaft) \in VZ_v$
VMT_t	Menge aller Verhaltensmustertypen eines MDMs t $VMT_t \subseteq \Sigma^+$ $vmt_i = (Name) \in VMT_t$
Σ^*	Menge aller Wörter im Unicode Alphabet Σ
Σ^+	Menge aller Wörter, ohne das leere Wort ε $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$