Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

Diplomarbeit Nr. 3493

Ein automatisches Verfahren zur Erzeugung von lauffähigen TOSCA Service-Templates, basierend auf DevOps-Artefakten

Shaojun Zhang

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Johannes Wettinger
begonnen am:	09.07.2013
beendet am:	08.01.2014
CR-Klassifikation:	K6; D2.11; D2.13

Kurzfassung

In letzter Zeit wurden eine Reihe von Ansätzen, Werkzeugen und Communities entwickelt, um das aktuell viel diskutierte DevOps-Paradigma realisierbar zu machen. Prominente Beispiele dafür sind Chef, Puppet oder Juju. All diese Ansätze verfolgen das Ziel, wiederverwendbare Artefakte (Skripte und Konfigurationsdefinitionen) zu erstellen, zu veröffentlichen und miteinander zu kombinieren (Orchestrierung), um damit durch einen hohen Grad an Automatisierung ein effizientes Deployment und Management von Services oder Applikationen in einer Cloud-Umgebung zu ermöglichen. Daher werden diese Werkzeuge häufig auch mit den Begriffen "Configuration Management Tooling" (Chef, Puppet, etc.) oder "Service Orchestration Tooling" (Juju, etc.) bezeichnet. Das Hauptproblem ist hierbei, dass die von den DevOps-Communities erstellten und veröffentlichten Artefakte nicht portabel sind, weil sie durch proprietäre Ansätze implementiert werden und damit von einer ebenfalls proprietären Laufzeitumgebung abhängig sind. Das heißt, dass diese Artefakte sich nur mit den Werkzeugen verarbeiten lassen, mit denen sie erstellt wurden. Dadurch weisen diese Artefakte einen sehr geringen Grad an Portabilität auf.

Um das oben genannte Problem zu vermeiden, sind Standardisierungsbemühungen im Bereich des Cloud Computing von wichtiger Bedeutung. Die "Topology and Orchestration Specification for Cloud Applications" (TOSCA) stellt einen Standardisierungsansatz dar, um solche Artefakte auf eine portable Art und Weise zu erstellen, sodass durch deren Orchestrierung portable Service-Templates entstehen können.

Ziel dieser Diplomarbeit ist die Entwicklung eines automatischen Verfahrens, um einsetzbare TOSCA Service-Templates zu erzeugen, basierend auf existierenden von den hinter Juju und Chef stehenden DevOps-Communities veröffentlichten Artefakten. Damit können diese Artefakte und die TOSCA Service-Templates, die diese Artefakte verwenden, von jeder TOSCA-konformen Laufzeitumgebung [65] verarbeitet werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Ausschnittsverzeichnis

Abkürzungsverzeichnis

BPEL: Business Process Execution Language

BPMN: Business Process Model and Notation

CSAR: Cloud Service Archive

DSL: Domain-specific Language

TOSCA: Topology and Orchestration Specification for Cloud Applications

URI: Uniform Resource Identifier

URL: Uniform Resoure Locator

UUID: Universal Unique Identifier

WSDL: Web Services Description Language

XML: Extensible Markup Language

XSD: XML Schema Definition

1 Einleitung

Dieses Kapitel beschreibt die Einleitung der Diplomarbeit. Zunächst wird der Hintergrund des Cloud-Service-Managements dargestellt und die Probleme von DevOps-Ansätzen sowie ihre Lösungen besprochen. Aus diesem Kontext heraus wird die Problemstellung und somit das Ziel der Diplomarbeit spezifiziert. Am Ende der Einleitung wird eine Übersicht über die Struktur der Arbeit und die Beschreibungen des Kapitels präsentiert.

1.1 Hintergrund

Die Reduzierung der Kosten für das Infrastruktur- und Service-Management ist einer der wichtigsten Aspekte von Cloud Computing, weil das traditionelle IT-Service-Management aufwendig ist [1]. Dieses Ziel wird durch die Automatisierung des gesamten Managements von Services in der Cloud erreicht. Das Management von Cloud-Services ist nicht auf das Deployment und Stilllegen von Service-Instanzen beschränkt. Dazu gehören weitere Managementaufgaben, die ausgeführt werden müssen, sobald eine bestimmte Service-Instanz deployed wurde. Beispielsweise muss sich die Service-Instanz je nach der aktuellen Arbeitsbelastung vergrößern und verkleinern. Zurzeit bieten viele Anbieter auf dem Gebiet von Cloud Computing [2] [3] proprietäre Angebote, mit denen die hochgradig skalierbaren Applikationen und Services erstellt werden können. Diese können Infrastruktur-Angebote wie Amazon Web Services [54] oder Plattform-Angebote wie Google App Engine [55] sein. Außerdem stellen Cloud-Anbieter auch proprietäre Werkzeuge wie z.B. "CloudFormation" und "Auto Scaling" von Amazon Web Services zur Verfügung, um das Management von Applikationen und Services in der Cloud zu automatisieren. Die Lernkurve ist vergleichsweise flach, weil diese Angebote und Werkzeuge einfach zu bedienen sind.

Wenn die Services immer komplexer werden, wird ihr Management jedoch zunehmend schwieriger [4]. Es könnte sogar unmöglich sein, den Service von einem Cloud-Anbieter zu einem anderen zu verschieben, weil noch Standards fehlen, die Portabilität zu gewährleisten. Das führt zu "Vendor Lock-in" [5] und schlechter Verwaltbarkeit ("Manageability"). Bei der Situation "Vendor Lock-in" ist Providerwechsel schwierig oder sogar unmöglich. Portabilität ist sehr wichtig für die Services in der Cloud, um "Vendor Lock-in" zu verhindern. Zurzeit sind viele Cloud-Services selbst portabel und können damit von einem Cloud-Anbieter zu einem anderen verschoben werden. Aber es könnte für das Service-Management nicht zutreffend sein. Das Management dieser Services wird oft auf das Provider-spezifische Managementwerkzeug beschränkt. Da das Managementwerkzeug unterschiedlich ist, könnte sich die Art des Managements eines bestimmten Cloud-Service komplett ändern, das heißt, der Service könnte auf eine ganz unterschiedliche Weise verwaltet werden, wenn dieser Service zu einem anderen Cloud-Anbieter verschoben wird. Darum ist Portabilität unbedingt erforderlich für die Services in der Cloud ist, vor allem, wenn es um das Service-Management geht. Deshalb muss es garantiert sein, dass die Verwaltbarkeit von Services in der Cloud verbessert wird, ohne auf die Portabilität zu verzichten.

Bis heute zeigen die sogenannten DevOps-Ansätze [6] [7] [8] das führende Paradigma für das effiziente Management von Services und Applikationen auf eine hochautomatisierte Weise. Das ursprüngliche Ziel dieser Methodologien ist es, die Entwickler und das Betriebspersonal

zusammenzubringen. Dieses Ziel wird hauptsächlich durch die Automatisierung aller Deployment- und Managementaufgaben erreicht.

In letzter Zeit haben sich eine Reihe von Ansätzen, Werkzeugen und Communities etabliert, um das aktuell viel diskutierte DevOps-Paradigma realisierbar zu machen. Prominente Beispiele dafür sind Chef [14], Puppet [15] oder Juju [16]. All diese Ansätze verfolgen das Ziel, wiederverwendbare Artefakte (Skripte und Konfigurationsdefinitionen) zu erstellen, zu veröffentlichen und miteinander zu kombinieren (Orchestrierung), um damit durch einen hohen Grad an Automatisierung ein effizientes Deployment und Management eines Service oder einer Anwendung in einer Cloud-Umgebung zu ermöglichen. Daher werden diese Werkzeuge häufig auch mit den Begriffen Konfigurationsmanagementwerkzeug (Chef, Puppet, etc.) oder Service-Orchestrierungswerkzeug (Juju, etc.) bezeichnet. Unter Konfigurationsmanagement versteht man die Disziplin im Bereich des System- und Infrastruktur-Managements, unterschiedliche Konfigurationen und deren Definitionen sauber zu verwalten. Der Begriff Service-Orchestrierung meint im Kontext dieser Arbeit die Möglichkeit, unterschiedliche Services (z.B. eine MySQL-Datenbank, ein Caching-System, etc.) miteinander zu verbinden.

Konfigurationsmanagementwerkzeug

Ein Konfigurationsmanagementwerkzeug [4] [9] [10] wie z.B. Chef oder Puppet realisiert die eigentliche Automatisierung. Demzufolge sind die entsprechenden Managementprozesse zuverlässiger und kostengünstiger im Gegensatz zur manuellen Durchführung dieser Prozesse. Folglich ist es viel einfacher, das initiale Deployment sowie das erneute Deployment von Services in verschiedenen Umgebungen wie z.B. Entwicklung, Test und Produktion durchzuführen. Die Philosophie hinter der DevOps-Bewegung ist, agile Methodologien in die Welt des IT-Infrastruktur- und IT-Service-Managements zu bringen [6]. Dies wird durch die Implementierung des Konzepts "Infrastructure as Code" unter Verwendung eines Konfigurationsmanagementwerkzeuges erreicht. Das Konzept basiert auf der Annahme, dass fast jede Aktion auf der Infrastruktur-Management-Ebene programmatisch automatisiert werden kann [11]. Folglich stellen die dieses Konzept implementierenden Produkte wie z.B. Chef oder Puppet eine Skriptsprache oder eine DSL ("Domain-specific Language") zur Verfügung, um plattformunabhängige Konfigurationsdefinitionen für das Deployment und Management eines IT-Service zu erstellen und zu verwalten [4]. Natürlich beschränkt sich das Konfigurationsmanagement nicht auf die Implementierung von DevOps-Ansätzen. Das Konzept von "Infrastructure as Code" zielt im Allgemeinen auf die Automatisierung und Kostenreduktion von Service-Management. Dies sind wesentliche Teile des Cloud-Computing-Paradigmas [12] [13].

Service-Orchestrierungswerkzeug

Orchestrierung beschreibt das Arrangieren, Koordinieren und Management von komplexen Computersystemen, Middlewares und Services [56]. Bei der Service-Orchestrierung handelt es sich um das automatische Deployment und Management eines Service auf eine bestimmte Art und Weise. Wir gehen davon aus, dass die Struktur und das Managementverhalten eines Service durch ein Service-Topologie-Modell, das aus mehreren Topologie-Modell-Komponenten besteht, spezifiziert wird [1]. Als Beispiel könnten zwei Topologie-Modell-Komponenten zu einem Topologie-Modell für das Deployment und Management einer Web-

Anwendung gehören: ein Webserver "Apache" [51] und ein Datenbankserver "MySQL" [26]. Eine Topologie-Modell-Komponente enthält Skripte, die normalerweise unter Verwendung einer Skriptsprache wie Python oder Perl implementiert werden. Diese Skripte realisieren die Managementaktionen wie z.B. das Deployment und die Aktualisierung seiner Komponenten, die bezüglich einer Service-Instanz des bestimmten Topologie-Modells durchgeführt werden können. Wir konzentrieren uns auf das Service-Deployment als eine der wichtigsten Managementaufgaben, die auf den Topologie-Modellen basieren. Zurzeit gibt es bereits vorhandene Topologie-Modell-Komponenten, die öffentlich zur Verfügung stehen. Diese Topologie-Modell-Komponenten können verwendet werden, um das Deployment und Management von Services in einer Cloud-Umgebung durchzuführen. Ein prominentes Beispiel dafür ist Juju als ein Service-Orchestrierungswerkzeug. Durch Juju können viele Topologie-Modell-Komponenten erstellt werden. Jede solche Komponente kann mit einer anderen Komponente kombiniert werden, um ein Service-Topologie-Modell zu erstellen. Dieses Service-Topologie-Modell kann in einer Cloud-Umgebung instanziiert und verwaltet werden. Der Kern einer Komponente ist eine Menge von Skripten, die das automatisierte Management einer bestimmten Service-Instanz ermöglichen.

Der Ansatz "Konfigurationsmanagement" ist nicht geeignet für das Management von komplexen Services. Es kann umständlich und zeitaufwendig werden, eine große und komplizierte Service-Topologie, die aus verschiedenen Maschinen besteht, unter Verwendung eines Konfigurationsmanagementwerkzeuges zu verwalten. Auch die Infrastruktur für die Versorgung einer üblichen Web-Anwendung kann sehr schnell komplex werden, weil es mehrere Technologien gibt, die erforderlich sind, um z.B. Lastverteilung ("Load Balancing"), Zwischenspeichern ("Caching") und Volltextindexierung ("Full-Text Index") zu realisieren. Um solche Infrastruktur zu spezifizieren, wird eine Menge "Infrastruktur-Code" erstellt. Demzufolge ist es schwierig, die Code-Struktur sauber zu halten, denn es gibt kein ganzheitliches Service-Modell im Hintergrund. Dieses Service-Modell enthält die Service-Topologie, die die Struktur eines Service, der instanziiert werden kann, bestimmt. Außerdem wird jede einzelne Änderung des "Infrastruktur-Codes" ein Risiko, weil es schwer ist, die Folgen jener bestimmten Änderung abzuschätzen [9].

Im Gegensatz zum Konfigurationsmanagementwerkzeug beschreibt ein Service-Orchestrierungswerkzeug ein Service-Topologie-Modell, mit dem das Deployment und Management konkreter Service-Instanzen in einer Cloud-Umgebung bewerkstelligt werden kann. Das Service-Orchestrierungswerkzeug ermöglicht den Modell-getriebenen Managementansatz einer Topologie von Cloud-Services und Cloud-Applikationen. Aber die Artefakte, die durch diese Werkzeuge erstellt wurden, lassen sich nur mit diesen Werkzeugen verarbeiten. Beispielsweise können Juju-Artefakte nur mithilfe der Juju-Laufzeitumgebung, in der sie erstellt wurden, ausgeführt und verarbeitet werden. Dadurch besitzen diese Artefakte einen sehr geringen Grad an Portabilität.

Um solche oben genannten Probleme zu lösen, gibt es bereits Standardisierungsbestrebungen im Bereich des Modell-getriebenen Cloud-Managements, die sich auf die Portabilität des Managements konzentrieren: Die "Topology and Orchestration Specification for Cloud Applications" (TOSCA) [17] ist ein aktueller Standard, der durch eine Reihe von prominenten Unternehmen in der Branche wie IBM, SAP und Hewlett-Packard unterstützt wird. TOSCA ermöglicht die Spezifikation von portablen Service-Topologie-Modellen und portablen

Topologie-Modell-Komponenten, die von jeder TOSCA-konformen Cloud-Umgebung deployed und verwaltet werden können. Allerdings spezifiziert TOSCA als ein Beispiel vom Modell-getriebenen Cloud-Management nicht direkt, wie die "Lower-Level" Aktionen auf einer virtuellen Maschine durchgeführt werden. Diese Aktionen sind z.B. die Ausführung von Skripten, um eine bestimmte Software-Komponente auf einer virtuellen Maschine zu installieren und zu konfigurieren. Deshalb ist es bedeutungsvoll, das Konfigurationsmanagement mit dem Modell-getriebenen Cloud-Management zu integrieren, um die Mängel der einzelnen Ansätze zu minimieren.

1.2 Problemstellung

Mit TOSCA könnten zwar die existierenden Probleme der DevOps-Werkzeuge gelöst werden, aber der Einsatz von TOSCA hat momentan starke Einschränkungen, weil noch ein entsprechendes Ökosystem dafür fehlt. Ein solches Ökosystem besteht aus einer aktiven Community, die die auf TOSCA basierenden Topologie-Modelle und Topologie-Modell-Komponenten zur Verfügung stellt. Im Gegensatz zu TOSCA gibt es bereits die entsprechenden Ökosysteme für die DevOps-Werkzeuge. Zum Beispiel stellt die Juju-Community mehr als hundert Topologie-Modell-Komponenten zur Verfügung. Ebenso bietet die Community für Chef auch viele Konfigurationsdefinitionen an. Solche von diesen Communities veröffentlichten Artefakte sind wiederverwendbar und können als Open-Source-Software verwendet werden.

1.3 Ziel der Arbeit

Das Ziel dieser Arbeit ist basierend auf existierenden von den hinter Juju und Chef stehenden DevOps-Communities veröffentlichten Artefakten diese Artefakte selbst auf eine portable Art und Weise zu verpacken, sodass durch deren Orchestrierung portable TOSCA-Artefakte ("Service-Templates") entstehen können. Basierend auf diesen TOSCA-Artefakten kann das Deployment und Management konkreter Service-Instanzen in einer Cloud-Umgebung bewerkstelligt werden, die eine dem TOSCA-Standard konforme Laufzeitumgebung [65] zur Verfügung stellt.

Zu diesem Zweck soll ein automatisches Verfahren entwickelt werden. Mit diesem Verfahren können aus den von den DevOps-Communities veröffentlichten Artefakten die entsprechenden TOSCA-Artefakte möglichst automatisch erstellt werden.

Zu den konkreten Aufgaben gehören: (1) die Erzeugung von TOSCA Node-Types aus bestehenden Chef-Artefakten, (2) die Erzeugung von TOSCA Relationship-Types aus bestehenden Juju-Artefakten und (3) die Erzeugung von TOSCA Service-Templates durch die Orchestrierung der Node-Types und Relationship-Types, die auf bestehenden Artefakten basieren.

1.4 Struktur der Arbeit

Die Arbeit ist in mehrere Kapitel gegliedert. Auf die Einleitung folgt ein Grundlagenkapitel. In diesem Grundlagenkapitel werden existierende Technologien wie Chef, Juju und TOSCA beschrieben, die als Grundlage dieser Arbeit dienen.

Im dritten Kapitel werden die Anforderungen an das automatische Verfahren dargestellt. Ein Beispiel ist die Abstraktion der Eigenheiten verschiedener DevOps-Ansätze (Chef, Juju, etc.). Als das Ergebnis der Abstraktion wird ein Modell für jeden verschiedenen DevOps-Ansatz generiert. Außerdem wird besprochen, welche Grenzen und Einschränkungen es dafür gibt. Am Ende werden zwei Beispiele für die Modell-Transformation (Chef nach TOSCA und Juju nach TOSCA) dargestellt.

Das vierte Kapitel beschreibt die Konzepte für das automatische Verfahren. Hier wird ausführlich besprochen, wie TOSCA Node-Types aus bestehenden Chef-Artefakten und TOSCA Relationship-Types aus bestehenden Juju-Artefakten erstellt werden. Darüber hinaus wird im letzten Unterkapitel beschrieben, wie die TOSCA Service-Templates durch die Orchestrierung der generierten Node-Types und Relationship-Types erzeugt werden.

Im fünften Kapitel werden der Entwurf und die Implementierung eines Prototyps zur Evaluierung des automatischen Verfahrens präsentiert. Damit soll gezeigt werden, dass die Konzepte für das automatische Verfahren tatsächlich realisiert werden können. Das heißt, dass durch diesen entwickelten Prototyp die entsprechenden TOSCA Node-Types, TOSCA Relationship-Types und TOSCA Service-Templates möglichst automatisch generiert werden können.

Das sechste Kapitel zeigt die Evaluierung des automatischen Verfahrens durch den entwickelten Prototyp. Die konkrete Evaluierungsarbeit besteht aus zwei Teilen: der Test der Funktionalität des Prototyps und die Überprüfung der Korrektheit und Gültigkeit von den mit dem entwickelten Prototyp generierten CSAR-Dateien für TOSCA Service-Templates.

Kapitel 7 beinhaltet eine Zusammenfassung der Arbeit und einen kurzen Ausblick auf mögliche weiterführende Arbeiten.

2 Grundlagen

Dieses Kapitel soll dem Leser Grundlagen vermitteln, die zum Verständnis der Diplomarbeit erforderlich sind. Drei Bereiche sind hierzu von besonderer Bedeutung. Im ersten Unterkapitel werden zunächst wichtige Informationen über Chef als ein Beispiel des Konfigurationsmanagementwerkzeuges gegeben. Unterkapitel 2.2 beschreibt wichtige Inhalte über Juju als ein Beispiel des Service-Orchestrierungswerkzeuges. Im letzten Unterkapitel wird TOSCA als ein aktueller Standard des Cloud-Service-Managements im Bereich des Modell-getriebenen Cloud-Managements erläutert.

2.1 Chef

Chef ist ein "Cloud-Infrastructure-Automation-Framework" und wird entwickelt, um die Vorteile des Konfigurationsmanagements zur Infrastruktur zu bringen. Chef erleichtert das Deployment von Servern und Applikationen auf jedem physischen, virtuellen oder Cloud-Standort, unabhängig von der Größe der Infrastruktur [18]. Chef stützt sich auf abstrakte Konfigurationsdefinitionen ("Cookbooks" und "Recipes"), die in Ruby [35] geschrieben und wie Source-Codes verwaltet werden. Jede Konfigurationsdefinition beschreibt, wie ein bestimmter Teil der Infrastruktur erstellt und verwaltet werden sollte. Chef benutzt dann diese Konfigurationsdefinitionen auf Servern und Applikationen, wodurch eine vollständig automatisierte Infrastruktur verwirklicht wird. Beim Deployment eines neuen Knotens ist das Einzige, was Chef wissen muss, welche Konfigurationsdefinitionen zur Anwendung kommen.

2.1.1 Architektur

In diesem Unterkapitel wird eine Einführung zur Architektur von Chef gegeben. Es werden die grundlegenden Funktionen von den wichtigen Komponenten einer Chef-Organisation (Client/Server-Umgebung) [18] erläutert. Außerdem wird besprochen, wie diese Komponenten bei der Verwendung von Chef zur Verwaltung der Infrastruktur miteinander kommunizieren.

Die Abbildung 2.1 zeigt die wichtigen Komponenten einer Chef-Organisation und die Beziehungen zwischen diesen Komponenten. Diese Komponenten arbeiten zusammen, um dem Chef die Informationen und Anweisungen anzubieten. Chef benötigt diese Informationen und Anweisungen, um seine Arbeit ausführen zu können. Chef besteht aus drei wesentlichen Komponenten: einem Server, einem (oder mehreren) Knoten ("Nodes") und mindestens einer Workstation.

Nodes

Eine Chef-Organisation besteht aus einer beliebigen Kombination von physischen, virtuellen und Cloud-basierten Knoten. Ein Cloud-basierter Knoten wird in einer externen Cloud-basierten Service gehostet, wie z.B. Amazon Virtual Private Cloud [57], OpenStack [58], Rackspace [59], Google Compute Engine [60] oder Windows Azure [61]. Wenn die Instanzen auf Cloud-basierten Services erstellt werden, wird Chef für das Deployment und Konfigurieren dieser Instanzen verwendet. Ein virtueller Knoten ist eine Maschine, die nur als eine Software-Implementation läuft. Aber ansonsten verhält sie sich ähnlich wie eine physische Maschine. Ein physischer Knoten ist normalerweise ein physischer Server, der

durch einen Chef-Client konfiguriert wird und mit einem Netzwerk verbunden ist. Das heißt, dass jeder Knoten einen Chef-Client enthalten und durch ein Netzwerk mit einem Chef-Server kommunizieren kann. Der Chef-Client führt die verschiedenen Infrastruktur-Automatisierungsaufgaben durch, die jeder Knoten benötigt. Die wichtigen Aufgaben eines Chef-Clients sind z.B. (1) Registrierung und Authentifizierung des Knoten mit dem Chef-Server, (2) Laden aller erforderlichen Cookbooks, welche die Recipes, die Attributes und alle anderen Abhängigkeiten ("Dependencies") enthalten, und (3) Unternehmen der geeigneten und erforderlichen Maßnahmen zum Konfigurieren des Knotens.

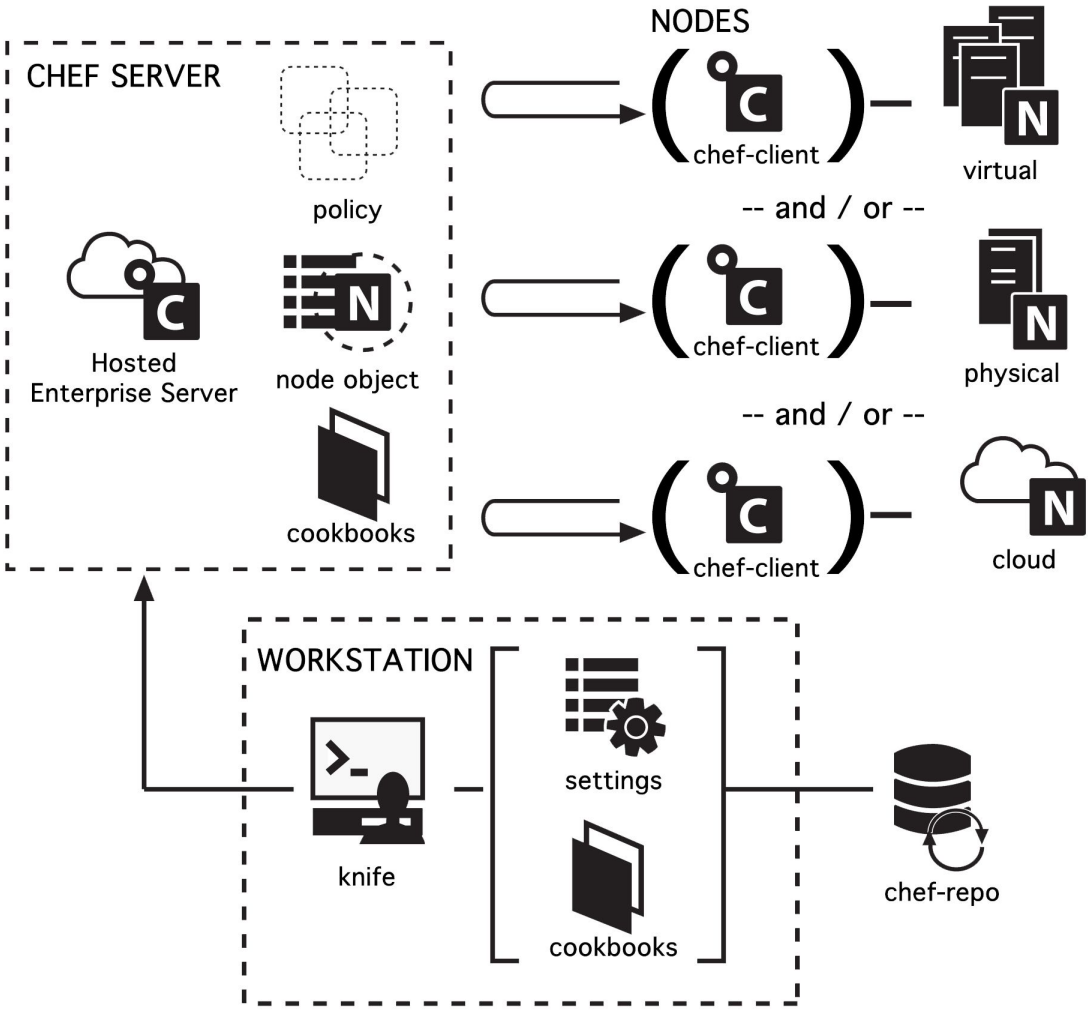

Abbildung 2.1: Die vereinfachte Architektur von Chef [18]

Chef-Workstations

Eine Chef-Workstation ist ein Computer, auf dem das Software "Knife" installiert ist. Knife wird zum Synchronisieren mit dem Chef-Repository und zur Interaktion mit einem einzelnen Chef-Server verwendet. Die wichtigen Aufgaben einer Chef-Workstation sind z.B. (1) Entwicklung von Cookbooks und Recipes unter Verwendung von Ruby, (2) Synchronisierung vom Chef-Repository mit einem Versionsverwaltungssystem ("Version Control System"), (3) Hochladen der Konfigurationsdateien vom Chef-Repository zum Chef-Server mithilfe von

Knife, (4) Definieren von Rollen ("Roles") und (5) Interaktion mit den Knoten, wenn es erforderlich ist, wie z.B. Durchführung einer Bootstrap-Operation. Eine Rolle versammelt verschiedene Recipes, um eine bestimmte Darstellung vom Knoten zu bilden. Beispielsweise können Rollen die Arten von den Knoten definieren, wie z.B. "Webserver" oder "Datenbankserver" [21]. Ein bestimmter Knoten kann keine oder mehrere Rollen haben. Recipes können mit einer oder mehreren Rollen verbunden sein. Dieser Mechanismus ermöglicht es, Recipes mit Knoten zu verbinden, ohne sie direkt zuzuweisen [20].

Zwei wichtigen Komponenten der Workstations sind Knife und Repository. Knife ist ein Kommandozeilen-Werkzeug, das eine Schnittstelle zwischen einem lokalen Chef-Repository und einem Chef-Server zur Verfügung stellt. Unter Verwendung von Knife verwalten die Benutzer z.B. Knoten, Cookbooks, Recipes, Rollen sowie Cloud-Ressourcen und Installation von Chef-Client etc. Das Chef-Repository ist der Speicherort, in dem einige Datenobjekte wie z.B. Cookbooks, Rollen und Konfigurationsdateien gespeichert werden. Das Chef-Repository befindet sich auf einer Workstation und sollte mit einem Versionsverwaltungssystem wie z.B. Git [19] synchronisiert werden. Alle Daten im Chef-Repository sollten wie Source-Code behandelt werden. Knife wird zum Hochladen von Daten aus dem Chef-Repository auf den Chef-Server verwendet. Nach dem Hochladen werden diese Daten von Chef verwendet, um alle Knoten zu verwalten. Diese Knoten sind auf dem Chef-Server registriert. Außerdem verwendet Chef diese Daten, um sicherzustellen, dass die richtigen Cookbooks, Rollen und andere Einstellungen korrekt auf den Knoten angewendet werden.

Chef-Server

Der Chef-Server fungiert als ein Zentrum für die Konfigurationsdaten der Infrastruktur. Er speichert die Daten wie z.B. die Cookbooks, die Rollen und die Metadaten, welche für das Konfigurieren der Knoten erforderlich sind. Die Metadaten beschreiben jeden registrierten Knoten, der durch den Chef-Client verwaltet wird. Die Knoten verwenden den Chef-Client, um den Server um die Konfigurationsdetails wie z.B. Recipes, Templates und File-Distributions zu bitten. Der Chef-Client führt dann möglichst viel Konfigurationsarbeit auf den Knoten selbst und nicht auf dem Server aus.

Verwendungsmodi von Chef

Es gibt zwei Verwendungsmodi von Chef: (1) Chef-Solo: Chef-Solo ist eine Open-Source-Version des Chef-Clients und ermöglicht die Verwendung von Cookbooks auf den Knoten, ohne auf einen Server zuzugreifen. Chef-Solo wird auf dem Knoten lokal ausgeführt und erfordert, dass sich ein Cookbook und alle seine Abhängigkeiten ("Dependencies") auf derselben physischen Festplatte des Knotens befinden. (2) Chef Client und Chef Server: Chef-Client ist ein Chef-Agent, der wie Chef-Solo auf dem Knoten lokal ausgeführt wird. Chef-Client verbindet sich mit einem Chef-Server und erfährt vom Server, welche Cookbooks und Ressourcen auf dem lokalen Knoten durchgeführt werden.

2.1.2 Cookbooks

Cookbooks sind die grundlegenden Einheiten der Verteilung ("Distribution") in Chef und die Art, wie Chef-Benutzer die Konfigurationsinformation verpacken, verteilen und gemeinsam benutzen. Sie kapselt alle Ressourcen ein, die zur Automatisierung der Infrastruktur benötigt werden. So ist es ganz einfach, den anderen Chef-Benutzern solche Ressourcen zur

Verfügung zu stellen. Sie enthalten Recipes, Attribut-Dateien, Templates und andere Konfigurationsartefakte. Wenn Chef-Client ausgeführt wird, werden die in der Run-Liste aufgelisteten Recipes zusammen mit den anderen Inhalten in dem Cookbook, das diese Recipes enthält, zum Knoten übertragen. Diese Recipes werden dann auf dem Knoten verwendet, damit der Knoten richtig konfiguriert werden kann. Normalerweise enthält ein Cookbook die notwendigen Informationen zum Konfigurieren eines einzelnen Service oder eines einzelnen Teils des Systems. Beispielsweise könnte ein Cookbook "Users" [62] für das Konfigurieren von Benutzern den Zugang zum System besitzen und ein Cookbook "Apache" [52] den Apache-Webserver konfigurieren. Cookbooks können von jedem Benutzer mit grundlegenden Fähigkeiten im Bereich der Programmierung erstellt werden. Darüber hinaus können die Cookbooks geschrieben werden, ohne die Details über eine Zielumgebung für das Deployment zu speichern. Dies bedeutet, dass diese Cookbooks zwischen verschiedenen Organisationen und Unternehmen wiederverwendet werden können. Benutzer haben bereits mehr als 300 Cookbooks auf der Chef-Community-Website veröffentlicht und zur Verfügung gestellt. Deshalb können Sie direkt mit den vorhandenen Cookbooks die Services und die Applikationen auf Ihrer Maschine installieren und konfigurieren, ohne ein neues Cookbook schreiben zu müssen. Abbildung 2.2 zeigt zwei Beispiele, welche Komponenten die Cookbooks "mysql" und "apache 2" besitzen und wie die Inhalte von diesen Cookbooks aussehen.

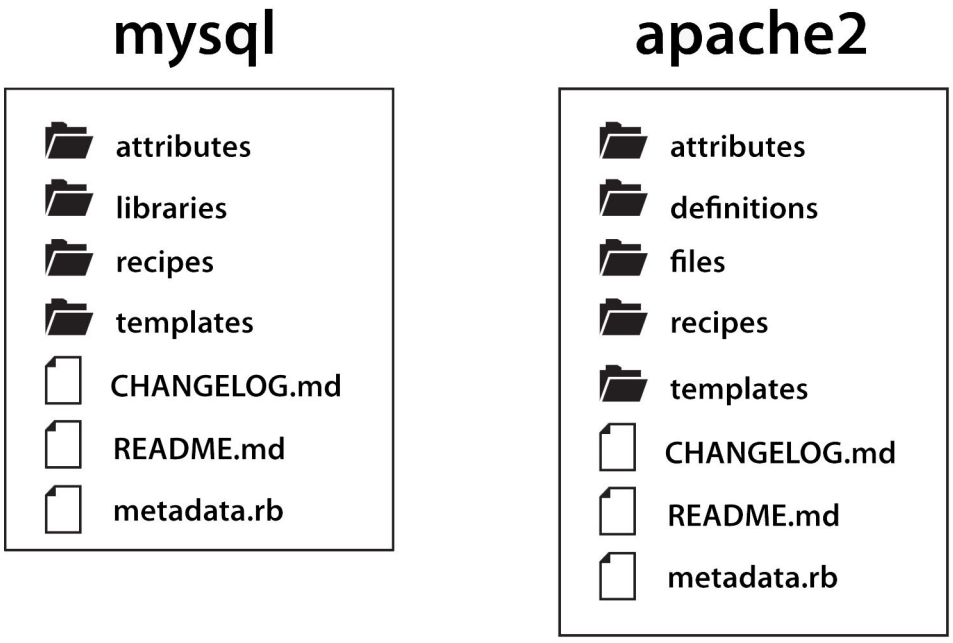

Abbildung 2.2: Die Komponenten von Cookbooks "mysql" und "apache 2"

Attributes - Ein Attribut ist ein spezifisches Detail zu einem Knoten, wie z.B. eine IP-Adresse, ein Hostname, eine Portnummer und so weiter. Attribute werden vom Chef-Client verwendet, um den Zustand des Knotens zu erfahren. Ein Attribut kann in einem Cookbook (oder einem Recipe) definiert und dann zum Überschreiben ("Override") der Standardeinstellungen ("Default Settings") auf einem Knoten verwendet werden.

Recipes - Ein Recipe ist das grundlegendste Konfigurationselement. Ein Recipe muss alle erforderlichen Ressourcen für das Konfigurieren eines Systems definieren und wird in einem

Cookbook gespeichert. Außerdem muss ein Recipe zu einer Run-Liste hinzugefügt werden, bevor sie durch den Chef-Client verwendet werden kann. Die Recipes werden immer in der gleichen Reihenfolge ausgeführt, wie sie in einer Run-Liste aufgelistet werden.

Definitions - Eine Definition wird verwendet, um neue Ressourcen zu erstellen, indem eine oder mehrere existierenden Ressourcen aneinandergefügt werden.

Files - Cookbooks werden häufig auf mehreren Plattformen ausgeführt und oft aufgefordert, eine spezielle Datei zu einer speziellen Plattform zu transportieren. Eine Datei-Distribution ist eine spezielle Art der Ressource, die einem Cookbook mitteilt, wie die Dateien verteilt werden.

Libraries - Eine Library ermöglicht die Verwendung von beliebigen Ruby-Codes in einem Cookbook dadurch, entweder indem die vom Chef-Client verwendeten Klassen erweitert werden oder eine neue Klasse direkt implementiert wird.

Templates - Ein Template ist eine mit der Markup-Sprache geschriebene Datei, die Ruby-Anweisungen verwendet, um komplexe Konfigurationsszenarien zu lösen.

Metadata - Eine Metadata-Datei wird verwendet, um sicherzustellen, dass jedes Cookbook zu jedem Knoten richtig und ordnungsgemäß deployed wird. Da die Metadata-Datei in einem Cookbook sehr wichtig für diese Arbeit ist, wird sie im nächsten Unterkapitel ausführlich besprochen.

Der Chef-Client verwendet nicht nur Ruby als Referenzsprache für die Erstellung von Cookbooks und die Definition von Recipes sondern auch eine erweiterbare DSL für spezielle Ressourcen. Eine angemessene Menge von Ressourcen steht dem Chef-Client zur Verfügung. Diese ist ausreichend, um die üblichsten Infrastruktur-Automatisierungsszenarien zu unterstützen. Diese DSL kann jedoch auch erweitert werden, wenn zusätzliche Ressourcen und Fähigkeiten benötigt werden.

2.1.3 Metadaten

Für jedes Cookbook muss eine kleine Menge von Metadaten spezifiziert werden. Diese Metadaten werden in einer Datei namens "metadata.rb" gespeichert. Diese Datei "metadata.rb" befindet sich im Wurzelverzeichnis jedes Cookbook. Die Inhalte der Datei "metadata.rb" bieten dem Server einige Hinweise an, damit die Cookbooks auf jedem Knoten richtig eingesetzt werden. Außerdem weisen sie den Server darauf hin, welche Cookbooks auf einem gegebenen Knoten eingesetzt werden sollten. Die Datei "metadata.rb" wird für ein automatisiertes System zur Entdeckung und Installation von Cookbooks wesentlich sein.

Eine Datei "metadata.rb" wird automatisch erstellt, wenn ein Cookbook unter Verwendung von Knife erstellt wird. Die Datei "metadata.rb" wird nie direkt interpretiert, sondern zuerst vom Server kompiliert und als JSON-Daten in der Datei namens "metadata.json" gespeichert. Die JavaScript Object Notation (JSON) [63] ist ein kompaktes Datenformat in für Mensch und Maschine einfach lesbarer Textform zum Zweck des Datenaustauschs zwischen Anwendungen. Diese JSON-Datei ist die echte Metadaten-Datei und wird beim Hochladen des Cookbook oder beim Ausführen des Befehls "knife cookbook metadata" generiert. Die Datei "metadata.json" kann direkt bearbeitet werden, wenn temporäre Änderungen

vorgenommen werden müssen. Jedes nachfolgenden Hochladen oder jede Aktion, die die Datei "metadata.rb" generiert, wird dazu führen, dass die vorhandene Datei "metadata.json" mit der neu generierten Datei "metadata.rb" überschrieben wird. Deshalb sollte jede erforderliche permanente Änderung an Metadaten nur in der Datei "metadata.rb" vorgenommen werden.

Im Folgenden werden einige Abschnitte, die für diese Arbeit sehr wichtig sind, in der Datei "metadata.rb" beschrieben.

Name - Der Name eines Cookbook.

Description - Eine kurze Beschreibung eines Cookbook und seiner Funktionalität.

Recipe - Eine kurze Beschreibung für ein Recipe.

Supports - zeigt, welche Plattformen das Cookbook unterstützt.

Depends - zeigt, dass ein Cookbook eine Abhängigkeit ("Dependency") von anderen Cookbook mit der Versionsnummer besitzt. Das Cookbook mit dem passenden Namen und der passenden Version muss auf dem Server existieren und bei der Ausführung von Chef-Client zum Knoten übertragen werden. Es ist sehr wichtig, dass der Abschnitt "depends" korrekte Daten enthält. Wenn diese Daten nicht korrekt sind, ist der Chef-Client nicht in der Lage, die Konfiguration des Systems erfolgreich durchzuführen.

Attribute - Die Liste der Attribute, die für das Konfigurieren eines Cookbook erforderlich sind. Der Name eines Attributs ist erforderlich und die anderen sind optional, wie z.B. "display_name" (der Name, der in der Benutzeroberfläche angezeigt wird), "description" (eine kurze Beschreibung), "type" (der Typ von Wert, entweder "String" oder "Array") und "default" (Default-Wert des Attributs) etc.

2.1.4 Bestehende Defizite und Zusammenfassung

Chef als ein Konfigurationsmanagementwerkzeug implementiert das Konzept "Infrastructure as Code". Durch Chef kann man die plattformunabhängigen Konfigurationsdefinitionen wie z.B. Cookbooks und Recipes erstellen und verwalten, um das automatisierte Deployment und Management eines Service oder einer Anwendung zu ermöglichen. Aber für das Management von komplexen Services ist Chef nicht praktisch. Als ein Beispiel zeigt Abbildung 2.3, wie das Deployment von Sugar [22] als ein Service in der Cloud unter Verwendung von Chef ermöglicht wird. Sugar ist ein Web-basiertes Customer-Relationship-Management-System und steht als Open-Source-Software öffentlich zur Verfügung.

Für das tatsächliche Deployment von Sugar werden zwei virtuellen Maschinen bereitgestellt. Auf einer Maschine werden durch die automatische Ausführung der Artefakte der Apache-Webserver, das PHP-Modul und die Sugar-Applikation installiert und konfiguriert. Der MySQL-Server und die Sugar-Datenbank werden auf der anderen Maschine installiert und konfiguriert. Um das Deploment zu beenden wird die Applikation mit der Datenbank verbunden. Alle diese Aktionen werden durch die Ausführung der entsprechenden Konfigurationsdefinitionen wie z.B. Cookbooks und Recipes implementiert. Das heißt, dass eine große Menge Chef-Codes erstellt werden müssen, um solche Infrastruktur zu

spezifizieren. Dadurch ist es schwierig, die Code-Struktur sauber zu halten. Außerdem können keine expliziten Beziehungen zwischen diesen Konfigurationsdefinitionen definiert werden, da es kein ganzheitliches Service-Modell dafür gibt. Solches Service-Modell enthält die Service-Topologie, die die ganze Struktur für das Deployment von Sugar bestimmt. Daraus folgt, dass es umständlich und zeitaufwendig werden kann, einen großen und komplizierten Service, der aus verschiedenen Maschinen besteht, unter Verwendung von Chef zu verwalten.

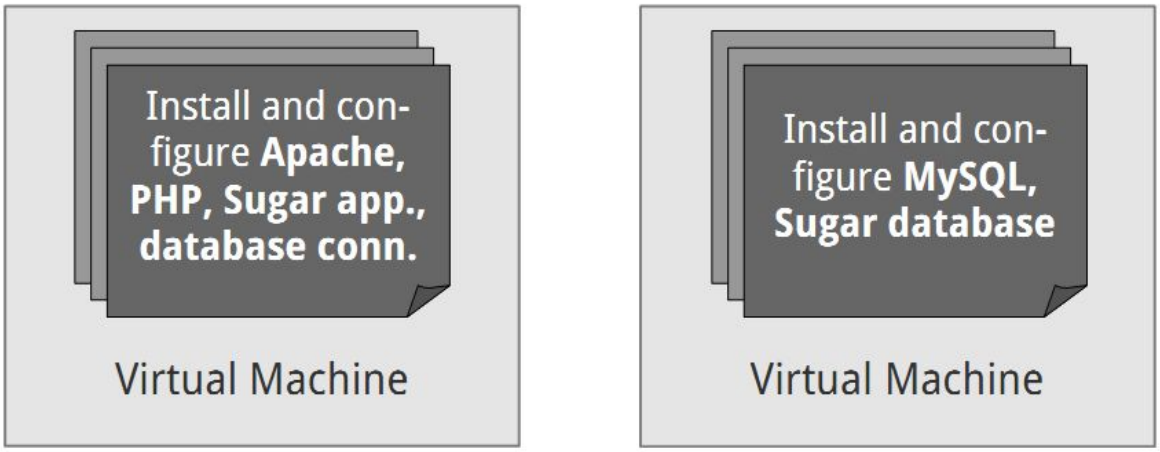

Abbildung 2.3: Deployment mithilfe des Konfigurationsmanagements [20]

Chef ist ein beliebtes Konfigurationsmanagementprodukt. Seine Open-Source-Version ist öffentlich verfügbar und kann kostenlos genutzt werden. Konfigurationsdefinitionen in Chef nennt man Recipes. Sie sind grundsätzlich die in einer DSL geschriebenen Skripte zur Darstellung des Ziel-Zustands eines Systems [4]. Ein oder mehrere Recipes sind in einem Cookbook gebündelt. Außer Cookbooks können Rollen definiert werden, um die Arten der Knoten zu spezifizieren. Durch eine Rolle kann beispielsweise ein Knoten zu einem Webserver konfiguriert werden. Der Chef-Server speichert alle Cookbooks und Rollen. Darüber hinaus verwaltet die Server-Komponente eine Run-Liste für jeden registrierten Knoten. Der Chef-Client wird auf jedem Knoten ausgeführt, um den Chef-Server zu verbinden. Eine bestimmte Run-Liste weist Recipes und Rollen einem Knoten zu. Zur Ausführung der Recipes ist der Chef-Server nicht erforderlich. Außer dem Client/Server-Modus steht auch ein Chef-Solo-Modus zur Verfügung. Unter Verwendung dieses Modus kann der Chef-Client die Recipes direkt auf dem Knoten ausführen, ohne sich mit einem Chef-Server zu kommunizieren [9].

2.2 Juju

Juju zielt darauf ab, ein Service-Deployment- und Orchestrierungswerkzeug zu sein, das die Zusammenarbeit zwischen den Services sowie die einfache Verwaltung dieser Services ermöglicht. Verschiedene Service-Entwickler können mit Juju Services selbstständig erstellen und die Kommunikation von diesen Services durch ein einfaches Konfigurationsprotokoll koordinieren. Dann können die Service-Benutzer die Services von verschiedenen Service-Entwicklern nehmen und sie sehr komfortabel in einer Umgebung bereitstellen. Das Ergebnis ist, dass mehrere Maschinen und Komponenten transparent zusammenarbeiten können, um die angeforderten Services zur Verfügung zu stellen.

2.2.1 Arbeitsweise

In diesem Unterkapitel wird beschrieben, wie man mit Juju durch einfache Befehlen Services deployen kann. Auf die Installation und die Konfiguration von Juju folgt es, eine Bootstrap-Umgebung einzurichten. Sie ist eine Instanz in der Cloud und wird von Juju verwendet, um Services zu deployen und verwalten. In der Praxis wird das Einrichten einer Bootstrap-Umgebung durch einen einfachen Juju-Befehl "juju bootstrap" automatisch implementiert. Falls eine solche Bootstrap-Umgebung erfolgreich aufgebaut wurde, können wir jetzt durch einige einfachen Juju-Befehle die Services deployen, verwalten und von außen zugreifbar zu machen. Ein kleines Beispiel für das Deployment des Service "WordPress" [25] zum Aufbau und zur Pflege eines Weblogs wird hier gegeben. Im Beispiel werden zuerst zwei Services "WordPress" und "MySQL" installiert und konfiguriert. Dann wird ein Beziehung "WordPress verbindet mit MySQL" zwischen diesen zwei Services aufgebaut. Schließlich wird der Service "WordPress" von außen zugreifbar gemacht. All diese Arbeiten werden in Juju nur mit vier einfachen Juju-Befehlen erledigt. Zu Beginn deployen wir die Services "WordPress" und "MySQL" durch die Juju-Befehle "juju deploy wordpress" und "juju deploy mysql". Dann wird der Service "WordPress" durch den Juju-Befehl "juju add-relation wordpress mysql" mit dem Service "MySQL" verbunden. Am Ende wird der Juju-Befehl "juju expose wordpress" verwendet, um den Service “WordPress” von außen zugreifbar zu machen. Mit diesem kleinen Beispiel kann man verstehen, dass man nur einige Juju-Befehle einzugeben braucht, um Services zu deployen und zu verwalten. Allerdings muss man beachten, dass ein Juju-Befehl nicht direkt die tatsächlichen Arbeiten für das Deployment und die Verwaltung von Services implementiert, sondern er informiert Juju über den Zustand, in dem sich die Umgebung befindet. Die tatsächlichen Arbeiten werden durch das System in der vorher eingerichteten Bootstrap-Umgebung erledigt.

2.2.2 Juju Charm

Juju verwendet Charms, um Softwares, sogenannte Services zu deployen. Charms definieren, wie sich Services integrieren (Relation von Services) und wie ihre Service-Einheiten auf Ereignisse in der verteilten Umgebung reagieren. Ein Service in Juju ist eine Anwendung oder eine Gruppe von Anwendungen und kann als eine einzelne Komponente verwendet werden. In der Regel können sich mehrere Services miteinander kombinieren, um einen komplexeren Service aufzubauen. Als ein Beispiel könnte "WordPress" als ein Service eingesetzt werden. Um seine Aufgaben korrekt zu verrichten, könnte "WordPress" mit einem Service "Datenbank" und einem Service "Load-Balancer" kommunizieren. Eine Service-Instanz in Juju besitzt zu Beginn genau eine Service-Einheit. Es können jedoch weitere Service-Einheiten zu dieser Instanz hinzugefügt werden, um z.B. Skalierbarkeit zu ermöglichen. Alle Service-Einheiten für einen bestimmten Service nutzen gemeinsam dasselbe Charm, dieselben Beziehungen und dieselbe Konfiguration, die vom Benutzer bereitgestellt werden. Beispielsweise kann eine MySQL-Datenbank-Instanz zu Beginn genau eine Service-Einheit (eine virtuelle Maschine) besitzen. Später können dann weitere Service-Einheiten (weitere virtuelle Maschinen) zu dieser Instanz hinzugefügt und mit der ursprünglichen Service-Einheit verknüpft werden. Ein Charm stellt die Definition des Service zur Verfügung. Zur Definition gehören auch seine Metadaten, die Abhängigkeiten von anderen Services, die notwendigen Pakete sowie die Logik für die Verwaltung der Anwendung. In Abbildung 2.4 wird ein Beispiel für die Struktur eines Charm dargestellt.

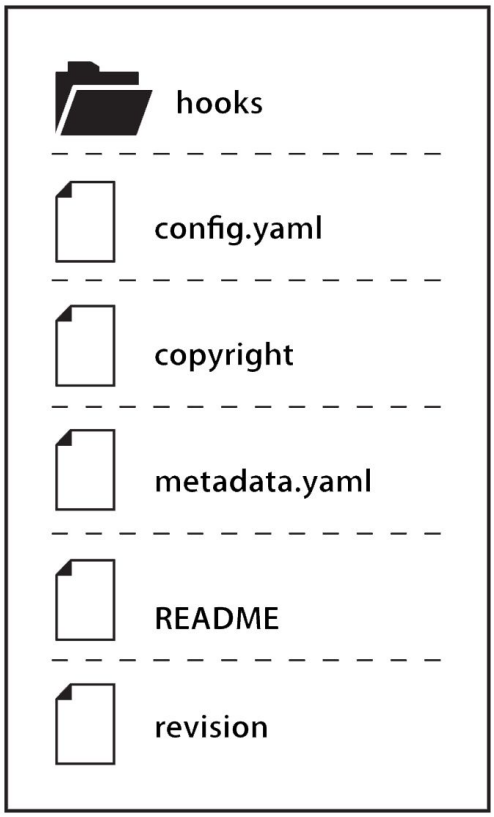

Abbildung 2.4: Ein Beispiel für die Struktur eines Charm

Normalerweise enthält jedes Charm ein Verzeichnis namens "hooks" und eine "metadata.yaml" Datei. Manche Charms enthalten noch eine "config.yaml" Datei. Details werden in den folgenden Unterkapiteln besprochen.

2.2.2.1 Das Verzeichnis "hooks"

In dem Verzeichnis "hooks" gibt es viele Dateien. Jede dieser Dateien wird als "Hook" bezeichnet. Die Hooks in einem Charm sind auf einem Ubuntu-Server ausführbare Dateien und können unter Verwendung von einer beliebigen Skriptsprache oder Programmiersprache geschrieben werden. Juju verwendet die Hooks, um eine Service-Einheit über die Veränderungen in ihrem Lebenszyklus oder in der verteilten Umgebung zu benachrichtigen. Ein für eine Service-Einheit laufender Hook kann diese Umgebung überprüfen. Außerdem kann es die gewünschten lokalen Änderungen auf der Maschine, wo sich dieser Hook befindet, vornehmen sowie die Einstellung der Relation ändern.

In der Regel gibt es in Bezug auf den Lebenszyklus einer Service-Einheit folgende Hooks: "install", "config-changed", "start" und "stop". Der Hook "install" wird zur Installation einer Service-Einheit verwendet und nur einmal ausgeführt, bevor alle anderen Hooks aufgerufen werden. Der Hook "config-changed" wird immer unmittelbar nach der Ausführung des Hooks "install" ausgeführt. Außerdem wird er auch verwendet, wenn sich die Service-Konfiguration ändert. Der Hook "start" wird ausgeführt, wenn die Service-Einheit gestartet wird. Dagegen wird der Hook "stop" ausgeführt, wenn die Service-Einheit gestoppt wird. Es können noch weitere Hooks verwendet werden, die als "Relation-Hook" bezeichnet und nach der Ausführung des Hooks "start" und vor der Ausführung des Hooks "stop" ausgeführt werden. Sie werden auf jeder Service-Einheit aufgerufen, wenn eine Relation hergestellt oder geändert

wird. Es gibt vier Arten Relation-Hooks. Der Hook "<relation name>-relation-joined" wird ausgeführt, wenn eine entfernte Service-Einheit an der Beziehung teilnimmt. Der Hook "<relation name>-relation-changed" wird ausgeführt, wenn eine entfernte Service-Einheit an der Beziehung teilnimmt oder seine Beziehungseinstellungen ändert. Der Hook "<relation name>-relation-departed" wird ausgeführt, wenn eine entfernte Service-Einheit eine Beziehung verlässt. Dies könnte passieren, wenn die Service-Einheit entfernt wurde, sein Service zerstört wurde oder die Beziehung zwischen diesem Service und dem entfernten Service entfernt wurde. Der Hook "<relation name>-relation-broken" bezieht sich auf die Beziehung selbst und wird ausgeführt, sobald die lokale Service-Einheit bereit ist, die Beziehung selbst zu verlassen. Diese Service-Einheit kann dann alle Konfigurationsinformationen über diese Beziehung beseitigen.

Ein Beispiel für das Verzeichnis "hooks" des Charm "MySQL" [26] wird in Abbildung 2.5 gezeigt. In dieser Abbildung werden nicht alle Hooks fürs Charm "MySQL" gezeigt, sondern die einigen typischen Hooks. Der Hook "install" wird zur Installation des Service "MySQL" verwendet. Der Hook "config-changed" wird nach der Ausführung des Hooks "install" zum Konfigurieren des Service "MySQL" ausgeführt. Der Hook "start" wird zum Starten einer Service-Einheit "MySQL" ausgeführt und der Hook "stop" wird verwendet, um eine Service-Einheit "MySQL" zu beenden.

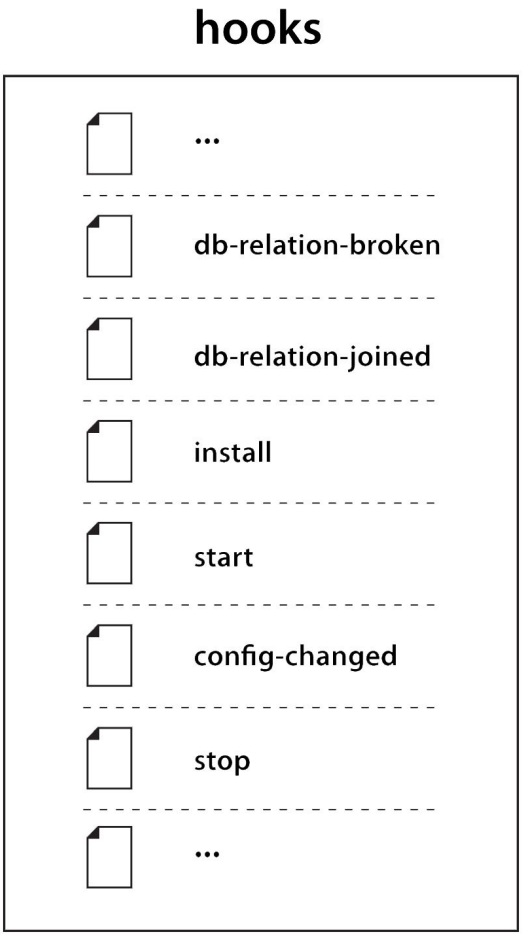

Abbildung 2.5: Ein Bespiel für das Verzeichnis "hooks"

In diesem Verzeichnis "hooks" gibt es noch zwei Relation-Hooks "db-relation-joined" und "db-relation-broken". Der Hook "db-relation-joined" wird aufgerufen, wenn eine Beziehung - z.B. eine Datenbankverbindung - zu einer Service-Einheit hinzugefügt wird. Der Hook "db-relation-broken" wird aufgerufen, wenn die Beziehung entfernt wird. Dabei wird die Service-Einheit die Konfigurationsinformationen zur Datenbankverbindung löschen.

2.2.2.2 Die Datei "metadata.yaml"

Die Datei "metadata.yaml" ist eine YAML-Datei. YAML [24] ist eine einfache Markup-Sprache zur Datenserialisierung, die sowohl gut von Menschen lesbar sein soll als auch vollautomatisch von Maschinen verarbeitbar ist. Diese Datei befindet sich im Wurzelverzeichnis eines Charm und beschreibt das Charm. Wir nehmen das Charm "WordPress" [25] als Beispiel. Seine "metadata.yaml" Datei wird teilweise in Abbildung 2.6 dargestellt.

```
name: wordpress
summary: "WordPress is a full featured web blogging tool, this charm deploys it."
maintainer: Marco Ceppi
description: |
  This will install and setup WordPress optimized to run in the cloud. This install,
  in particular, will place Ngnix and php-fpm configured to scale horizontally with
  Nginx's reverse proxy
requires:
  db:
    interface: mysql
provides:
  website:
    interface: http
```

Abbildung 2.6: Die Datei "metadata.yaml" des Charm "WordPress"

Diese Datei "metadata.yaml" deklariert ein Charm mit dem Namen "WordPress". Die ersten vier Abschnitte geben folgende Informationen über dieses Charm an: den Namen des Charm, die Information über den Ersteller des Charm, eine kurze und eine lange Beschreibung.

Der Abschnitt "provides" beschreibt, welche Services das Charm "WordPress" tatsächlich zur Verfügung stellt. Das Charm "WordPress" ist ein Web-basierter Service der Blogging-Plattform, der ein einfaches Interface "http" zur Verfügung stellt. Der hier angegebene Name "website" ist ein lokaler Beziehungsname ("Relation-Name") und identifiziert diese Beziehung eindeutig innerhalb des Charm "WordPress". Und das Interface "http" wird von den anderen Charms verwendet, wenn sie eine Beziehung mit diesem Charm herstellen wollen. Juju überprüft Interfaces, wenn Juju versucht, festzustellen, ob zwei Services miteinander verknüpft werden können.

Der Abschnitt "requires" beschreibt, welche Services das Charm "WordPress" braucht. Für das Charm "WordPress" wird eine Beziehung mit einer Datenbank definiert. Diese Beziehung wird lokal "db" genannt und besitzt das Interface "mysql". Durch das Überprüfen der Metadaten des Charm "MySQL" erfährt Juju, dass dieses Charm die Fähigkeit einer Datenbank mit dem Interface "mysql" zur Verfügung stellt. Das heißt, dass eine Beziehung zwischen den Charms "WordPress" und "MySQL" mit dem gleichen Interface "mysql"

implizit hergestellt werden kann. Ein Beispiel für die Datei "metadata.yaml" des Charm "MySQL" wird teilweise in Abbildung 2.7 gezeigt.

```
name: mysql
summary: MySQL is a fast, stable and true multi-user, multi-threaded SQL
  database
maintainer: Marco Ceppi
description: |
  MySQL is a fast, stable and true multi-user, multi-threaded SQL database
  server. SQL (Structured Query Language) is the most popular database query
  language in the world. The main goals of MySQL are speed, robustness and
  ease of use.
provides:
  db:
    interface: mysql
peers:
  cluster:
    interface: mysql-ha
requires:
  slave:
    interface: mysql-oneway-replication
```

Abbildung 2.7: Die Datei "metadata.yaml" des Charm "MySQL"

Der Abschnitt "provides" beschreibt, dass das Charm "MySQL" die Fähigkeit einer Datenbank zur Verfügung stellt. Der lokale Beziehungsname ist "db" und das Interface "mysql" wird beispielsweise vom Charm "WordPress" verwendet, wenn "WordPress" eine Beziehung mit dem Charm "MySQL" herstellen will. Außerdem wird der Abschnitt "peers" in der Datei "metadata.yaml" vom Charm "MySQL" definiert, der im Unterkapitel 2.2.3 besprochen wird.

2.2.2.3 Die Datei "config.yaml"

Die Datei "config.yaml" befindet sich auch im Wurzelverzeichnis eines Charm. In dieser Datei werden einige Konfigurationsoptionen definiert, auf die das Charm zugreift. Diese Konfigurationsdaten beschreiben, wie ein Service konfiguriert wird. Charms erlauben nur, die Konfigurationsoptionen zu bearbeiten, die von dem Ersteller des Charm explizit definiert werden. Diese Optionen werden nicht nur für eine bestimmte Service-Einheit oder Beziehung verwendet, sondern für den gesamten Service. Beispielsweise können wir mit dem Charm "WordPress einen Service namens "myblog" deployen. Dieser Service könnte eine Option "blog-title" definieren. Diese Option kontrolliert den Titel des zu veröffentlichenden Blogs. Die Änderungen an dieser Option gelten für alle Service-Einheiten, die zu einer bestimmten Service-Instanz des Service "myblog" gehören. Dabei wird ein entsprechender Hook auf jeder von diesen Service-Einheiten aufgerufen.

In Abbildung 2.8 wird gezeigt, wie eine "config.yaml" Datei aussieht. Jede Option enthält eine lesbare Beschreibung und einen optionalen Default-Wert "default". Zusätzlich kann möglicherweise ein Typ "type" spezifiziert werden. Alle Optionen haben einen Default-Typ von 'string'. Er bedeutet, dass sein Wert nur als eine Text-Zeichenfolge behandelt wird. Andere gültige Optionen sind 'int' und 'float'.

```
options:
    port:
      default: 80
       type: int
       description: Port to listen on
   admin-email:
     #type: str is implied
     default: null
     description: Email address for the site administrator
```

Abbildung 2.8: Ein Bespiel für die Datei "config.yaml"

2.2.3 Relation in Juju

Dieses Unterkapitel beschäftigt sich mit den Beziehungen zwischen Services in Juju. Eine Bezeihung wird normalerweise in Juju als "Relation" bezeichnet. Deshalb wird das Wort "Relation" in diesem Unterkapitel verwendet. In der Datei "metadata.yaml" könnte es drei Abschnitte "provides", "requires" und "peers" geben. Ein Beispiel dafür wurde in Abbildung 2.7 gegeben. Die drei Abschnitte definieren die verschiedenen Relationen, an den das Charm teilnehmen wird. Relationen in Juju haben drei Haupteigenschaften: ein Interface, eine Art und einen Namen. Das Relation-Interface ist ein eindeutiger Name, durch den die Service-Einheiten unter Verwendung von ihren jeweiligen Hooks die Informationen austauschen können. Solange der Name identisch ist, bedeutet das, dass die Charms auf eine kompatible Art und Weise geschrieben wurden. Deshalb darf die Relation durch das gleiche Interface hergestellt werden. Relationen mit verschiedenen Interfaces können nicht hergestellt werden. Die Relation-Art informiert darüber, ob eine Service-Einheit, die das gegebene Charm deployed, als ein "Provider", ein "Requirer" oder ein "Peer" in der Relation dienen wird. Providers und Requirers ergänzen sich gegenseitig. Folglich kann ein Service, der ein Interface zur Verfügung stellt, eine Relation besitzen. Diese Relation wird nur mit dem Service, der das gleiche Interface braucht, hergestellt und umgekehrt. Peer-Relationen werden zwischen den Service-Einheiten innerhalb eines Service, der diese Relation deklariert, automatisch hergestellt. Das dient dazu, diese Service-Einheiten zusammenzubinden, um Master und Slaves, Ringe oder andere strukturelle Organisation, die die zugrunde liegende Software unterstützt, aufzubauen. Der Relation-Name identifiziert die entsprechende Relation innerhalb des Charm eindeutig. Außerdem erlaubt er, dass ein einzelnes Charm (und Service und Service-Einheiten, die das Charm benutzen) mehrere Relationen mit dem gleichen Interface aber zu unterschiedlichen Zwecken hat. Dieser Identifizierer (Relation-Name) wird in Hook-Namen verwendet. In Abbildung 2.9 wird ein Beispiel für die Require/Provide-Relation in Juju gezeigt. Wenn dieses Service-Modell realisiert wird, wird Juju alle Service-Einheiten des Service "WordPress" darüber informieren, dass eine Relation mit den jeweiligen Service-Einheiten des Service "MySQL" hergestellt wurde. Dieses Ereignis wird dadurch mitgeteilt, dass die entsprechenden Hooks mithilfe der lokalen Relation-Namen auf beiden Service-Einheiten aufgerufen werden. Falls die Verbindung zwischen den Services "WordPress" und "MySQL" im Beispiel getriggert wird, werden die Hooks "db-relation-

joined, db-relation-changed" auf der WordPress-Seite aufgerufen. Entsprechende Hooks "server-relation-joined" und "server-relation-changed" werden auf der MySQL-Seite aufgerufen.

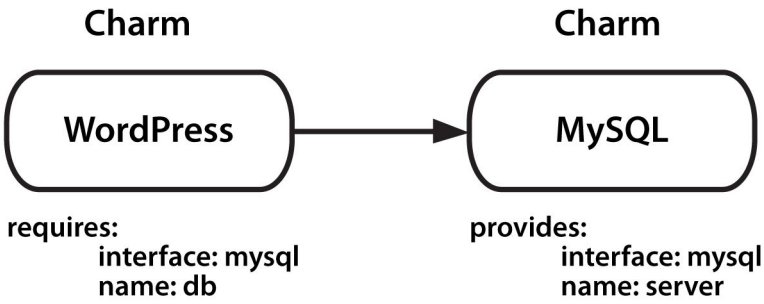

Abbildung 2.9: Beispiel für die Require/Provide-Relation

2.2.4 Bestehende Defizite und Zusammenfassung

Juju als ein Service-Orchestrierungswerkzeug ermöglicht das Modell-getriebene Cloud-Management. Die Juju-Community veröffentlicht mehr als einhundert Topologie-Modell-Komponenten als Open-Source-Software. Diese Komponenten sind gemeinsam nutzbar und wiederverwendbar und werden als "Charms" bezeichnet. Ein Charm enthält in der Regel Shell- oder Python-Skripte. Diese Skripte werden verwendet, um das automatische Management einer bestimmten Service-Instanz zu ermöglichen. Ein Charm kann mit einem anderen Charm kombiniert werden, um ein Service-Topologie-Modell zu erstellen. Dieses Service-Topologie-Modell kann in der Cloud-Umgebung instanziiert und verwaltet werden. Darüber hinaus können die Beziehungen zwischen den Service-Instanzen hergestellt werden.

Bezüglich der Portabilität gibt es jedoch starke Einschränkungen, weil die Juju-Artefakte nur mithilfe der Juju-Laufzeitumgebung ausgeführt und verarbeitet werden können. Das bedeutet, dass die von der Juju-Community zur Verfügung gestellten Charms nur durch die Juju-Engine verarbeitet werden können. Außerdem beschränken die Skripte in Charms die Portabilität auf zwei Arten: (1) Die Skripte verwenden eine Reihe von Befehlen und Umgebungsvariablen, die auf jeder von Juju verwalteten virtuellen Maschine verfügbar sind. (2) Die Skripte werden so designt, dass sie nur auf Ubuntu-Linux ausgeführt werden können. Infolgedessen misslingt ihre Ausführung auf den anderen Linux-Varianten und den anderen Plattformen.

2.3 Topology and Orchestration Specification for Cloud Applications

Cloud Computing [3] kann wertvoller werden, wenn die (semi-)automatische Erstellung und Verwaltung von Cloud-Services auf der Anwendungsschicht in den verschiedenen Cloud-Umgebungen eingesetzt werden kann. Somit können die Services interoperabel bleiben. Die TOSCA-Spezifikation [17] stellt eine Sprache zur Verfügung, die die Service-Komponenten und ihre Beziehungen mithilfe einer Service-Topologie ("Service-Topology") beschreibt. Außerdem bietet diese Sprache noch die Beschreibung der Verwaltungsprozeduren an, welche die Services mittels Orchestrierungsprozesse ("Orchestration-Processes") erstellen, ändern und terminieren. In TOSCA werden diese Prozesse als Pläne [29] [44] [45] bezeichnet. Die Kombination von Topologie und Orchestrierung in einem Service-Template beschreibt, was

unter Deployments in verschiedenen Umgebungen benötigt wird. Das Ziel ist das interoperable Deployment von Cloud-Services und ihrer Verwaltung während des gesamten Lebenszyklus zu ermöglichen, wenn die Applikationen in unterschiedlichen Cloud-Umgebungen deployed werden.

2.3.1 Einführung

Unter Verwendung von TOSCA kann eine Service-Topologie modelliert werden. So wird ein portables, ausführbares Service-Modell erstellt werden. Außerdem beinhaltet dieses Service-Modell alle seine Teile, aus denen es besteht und wird verwendet, um die Service-Instanzen in der Cloud zu deployen und zu verwalten [29]. Bevor TOSCA eingeführt wurde, hatte sich die Forschung auf die Migration der Services von einer Cloud-Umgebung zu einer anderen konzentriert, ohne die Portabilität von Managementaspekte zu berücksichtigen [30] [31].

Das Metamodell von TOSCA ist technisch durch eine XML-Schema-Definition spezifiziert. Es legt die Struktur eines Service-Template fest. Die wichtigen Teile zur Beschreibung einer Service-Topologie in einem Service-Template sind: Node-Types, Relationship-Types und das Topology-Template. Dies wird in Abbildung 2.10 dargestellt.

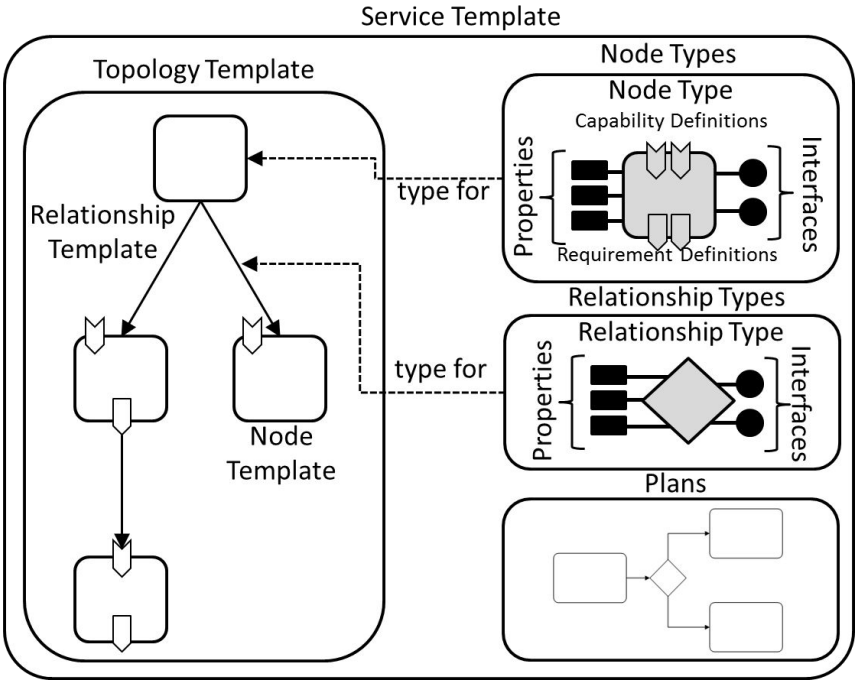

Abbildung 2.10: Strukturelle Elemente eines Service-Template und ihrer Beziehungen [17]

Node-Types repräsentieren die Komponenten, die im Topology-Template verwendet werden. Ein einzelner Node-Type, wie z.B. "Datenbankserver", kann als ein Node-Template im Topology-Template einmal oder mehrmals instanziiert werden. Relationship-Types, wie z.B. "Hosted-On", können als Relationship-Templates im Topology-Template instanziiert werden, um die Beziehung zwischen zwei bestimmten Node-Templates darzustellen. Das Topology-Template definiert die tatsächliche topologische Struktur eines IT-Service. Es besteht aus Node-Templates und Relationship-Templates.

Abbildung 2.11 zeigt ein Beispiel für einen Node-Type, der eine Datenbankserver-Komponente definiert. Ein Node-Type kann die Definitionen der beliebigen Eigenschaften

wie z.B. "Benutzername" und "Passwort" besitzen. Diese Eigenschaften werden explizit definiert und an einen bestimmten Node-Type angehängt. Node-Templates können konkrete Werte für diese Eigenschaften definieren. Darüber hinaus kann ein Node-Type die Interfaces besitzen. Ein bestimmtes Interface stellt die Operationen zur Verfügung, die die Möglichkeiten zur Interaktion eines Knotens des angegebenen Node-Type definieren. Wir gehen davon aus, dass ein beliebiger Node-Type ein Lebenszyklus-Interface besitzt, das mindestens zwei Operationen zur Verfügung stellt, wie z.B. den Knoten eines bestimmten Node-Type zu erstellen und zu terminieren. Bis jetzt ist die Definition des Node-Type abstrakt und zeigt noch nicht, wie eine Operation implementiert wird. So können ein oder mehrere konkrete Implementation-Artifacts mit einer Operation verknüpft werden. Ein solches Implementation-Artifact wird durch die Definition eines Artifact-Template innerhalb des Service-Template erstellt. Dann kann das Artifact-Template als ein Implementation-Artifact an mindestens eine Operation angehängt werden. Eine Implementierung für die Operation "create" kann beispielsweise ein Unix-Shell-Skript zur Installation des Datenbankservers sein. Darüber hinaus kann ein anderes Skript an dieselbe Operation angehängt werden. Das Skript könnte die entsprechenden Aktionen auf Windows-basierten Systemen ausführen. Das Anhängen mehrerer Implementation-Artifacts an einer bestimmten Operation verbessert die Portabilität des Service-Template, da die Operation auf verschiedenen Plattformen ausgeführt werden kann. Die Definition von Relationship-Types ist ähnlich wie die Definition von Node-Types.

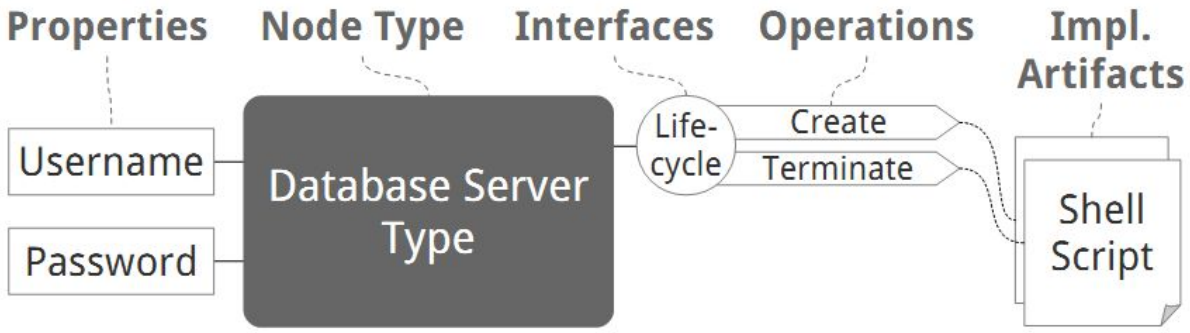

Abbildung 2.11: Beispiel für einen Node-Type [20]

TOSCA konzentriert sich nicht nur auf die Spezifizierung der Service-Topologie. Pläne können definiert werden, um den gesamten Lebenszyklus einer Anwendung wie z.B. Deployment, Wartung und Termination zu unterstützen. Diese Pläne können unter Verwendung der Sprachen wie z.B. BPMN [27] oder BPEL [28] definiert werden. Im Rahmen dieser Arbeit sind die Pläne nicht wichtig. Diese Arbeit konzentriert sich auf das Topologie-Modell.

Alle Dateien wie z.B. Skripte, ausführbare Dateien oder Programme und Pläne, die innerhalb des Service-Template referenziert werden, werden zusammen mit dem Service-Template in ein Cloud-Service-Archiv (CSAR) eingebaut. Das CSAR ist komplett “self-contained”. Das heißt, dass das CSAR alles zum Deployment und Management eines Cloud-Service beinhaltet. Dieser Cloud-Service wird durch das Service-Template spezifiziert und dieses Service-Template ist auch in der entsprechenden CSAR-Datei eingebaut. Die Software, die die CSARs verarbeiten kann, wird als TOSCA-Laufzeitumgebung (TOSCA-Container) [65] bezeichnet.

2.3.2 Service-Templates und Artifacts

Ein Artefakt repräsentiert den Inhalt, der zur Realisierung eines Deployment benötigt wird. Es kann eine ausführbare Datei (z.B. ein Skript, ein ausführbares Programm, ein Image), eine Konfigurationsdatei, eine Datendatei oder etwas (z.B. eine Library), das für die Ausführung von anderen ausführbaren Dateien benötigt wird, sein. Artefakte können verschiedene Arten, z.B. EJBs oder Python-Skripte sein. Der Inhalt eines Artefakts hängt von seiner Art ab. Normalerweise werden deskriptive Metadaten auch zusammen mit dem Artefakt zur Verfügung stehen. Diese Metadaten könnten erforderlich sein, um das Artefakt korrekt zu verarbeiten, z.B. durch die Beschreibung der entsprechenden Ausführungsumgebung. TOSCA unterscheidet zwei Arten von Artefakten: Implementation-Artifacts und Deployment-Artifacts.

Ein Implementation-Artifact repräsentiert die ausführbare Datei einer Operation eines Node-Type, und ein Deployment-Artifact repräsentiert die ausführbare Datei für die Materialisierung von Instanzen eines Knoten. Der grundlegende Unterschied zwischen Implementation-Artifacts und Deployment-Artifacts ist: (1) der Zeitpunkt, wann das Artefakt deployed wird, und (2) durch welche Entität und wohin das Artefakt deployed wird.

Die Operationen eines Node-Type führen die Verwaltungsaktionen auf dem Node-Type oder auf den Instanzen des Node-Type durch. Die Implementierungen dieser Operationen können als Implementation-Artifacts zur Verfügung gestellt werden. Folglich müssen die Implementation-Artifacts der entsprechenden Operationen in der Verwaltungsumgebung deployed werden, bevor jede Verwaltungsoperation gestartet werden kann. Mit anderen Worten muss eine TOSCA-konforme Umgebung in der Lage sein, die Typen der Implementation-Artifacts zu verarbeiten. Diese Artifact-Types werden zum Ausführen dieser Verwaltungsoperationen benötigt. Eine solche Verwaltungsoperation könnte beispielsweise die Instanziierung eines Node-Type sein.

Für die Instanziierung eines Node-Type wird benötigt, dass die Deployment-Artifacts in der verwalteten Ziel-Umgebung zur Verfügung stehen. Zu diesem Zweck unterstützt ein TOSCA-Container eine Reihe von Arten der Deployment-Artifacts, die er verarbeiten kann. Ein Service-Template, das Implementation- oder Deployment-Artifacts von nicht-unterstützten Typen enthält, kann durch den Container nicht verarbeitet werden.

2.3.3 Requirements and Capabilities

TOSCA kann die Anforderungen ("Requirements") und Fähigkeiten ("Capabilities") von Komponenten eines Service bestimmten. Beispielsweise hängt eine Komponente von einem Feature ab, das von einer anderen Komponente zur Verfügung gestellt wird. Oder eine Komponente besitzt eine bestimmte Anforderung an die Hosting-Umgebung wie z.B. für die Allokation von bestimmten Ressourcen.

Anforderungen und Fähigkeiten werden unter Verwendung von Requirement-Definitions und Capability-Definitions in Node-Types modelliert. Requirement-Types und Capability-Types werden als wiederverwendbare Einheiten definiert, sodass diese Definitionen im Zusammenhang mit verschiedenen Node-Types verwendet werden können. Beispielsweise könnte ein Requirement-Type "DatabaseConnectionRequirement" definiert werden, um die Anforderung eines Klienten für eine Datenbankverbindung zu beschreiben. Dieser Requirement-Type kann dann für alle Arten von Node-Types wiederverwendet werden.

Beispielsweise repräsentieren solche Node-Types, dass eine Applikation eine Verbindung zu einem Datenbankserver benötigt.

Node-Templates, die die entsprechende Node-Types mit Requirement-Definitions oder Capability-Definitions besitzen, enthalten die Darstellungen von den jeweiligen Anforderungen und Fähigkeiten mit spezifischem Inhalt zum jeweiligen Node-Template. Beispielsweise stellt die in einem Node-Template repräsentierte Anforderung konkrete Werte für die im Requirement-Type definierten Eigenschaften zur Verfügung, während die Requirement-Types nur die Metadaten der Anforderung darstellen. Darüber hinaus können Anforderungen und Fähigkeiten von Node-Templates in einem Topology-Template unter Verwendung von Relationship-Templates optional verbunden werden, um anzuzeigen, dass eine bestimmte Anforderung eines Knotens durch eine von einem anderen Knoten zur Verfügung gestellte Fähigkeit erfüllt wird.

2.3.4 TOSCA Cloud Service ARchive (CSAR)

Um in einer bestimmten Umgebung die Durchführung und die Verwaltung des Lebenszyklus einer Cloud-Anwendung zu unterstützen, müssen alle entsprechenden Artefakte in dieser Umgebung verfügbar sein. Das heißt, dass neben dem Service-Template der Cloud-Anwendung die Deployment-Artifacts und die Implementation-Artifacts in dieser Umgebung verfügbar sein müssen. Um die Verfügbarkeit von allen genannten Elementen zu garantieren, definiert diese Spezifikation ein entsprechendes Archiv-Format namens Cloud-Service-Archive (CSAR). Abbildung 2.12 zeigt die Struktur einer CSAR-Datei.

Abbildung 2.12: Die Struktur einer CSAR-Datei

Ein CSAR ist eine Zipdatei, die mindestens zwei Verzeichnisse enthält: "TOSCA-Metadata" und "Definitions". Darüber hinaus können andere Verzeichnisse in einer CSAR-Datei enthalten sein, d.h., der Ersteller einer CSAR-Datei hat die Freiheit, die Inhalte einer CSAR-Datei und die Strukturierung dieser Inhalte den Cloud-Anwendungen entsprechend zu definieren.

Das Verzeichnis "TOSCA-Metadata" enthält die Metadaten, welche die anderen Inhalte der CSAR-Datei beschreiben. Diese Metadaten werden als "TOSCA-Metadatei" bezeichnet. Diese Datei besitzt den Dateinamen "TOSCA.meta".

Das Verzeichnis "Definitions" enthält ein oder mehrere TOSCA-Definitions-Dokumente (Dateiendung ".tosca"). Diese Definitions-Dateien enthalten in der Regel Definitionen bezüglich der Cloud-Anwendung des CSAR. Darüber hinaus kann eine CSAR-Datei nur die Definition der Elemente für Wiederverwendung in anderen Kontexten enthalten. Beispielsweise könnte eine CSAR-Datei verwendet werden, um eine Reihe von Node-Types und Relationship-Types mit ihren jeweiligen Implementierungen zu verpacken, die dann von Service-Templates in anderen CSAR-Dateien verwendet werden können. In den Fällen, wo eine komplette Cloud-Anwendung in einer CSAR-Datei verpackt ist, muss eins der TOSCA-Definitions-Dokumente im Verzeichnis "Definitions" eine Definition für Service-Template enthalten, die die Struktur und das Verhalten der Cloud-Anwendung definiert.

2.3.5 TOSCA-Definitions-Dokument

Alle Elemente, die zum Definieren eines TOSCA Service-Template nötig sind, wie z.B. Node-Type-Definitionen, Relationship-Type-Definitionen sowie Service-Templates selbst, sind Teil eines TOSCA-Definitions-Dokuments. Dieses Unterkapitel beschreibt die allgemeine Struktur eines TOSCA-Definitions-Dokuments.

Der XML-Ausschnitt 2.1 beschreibt ein Pseudo-Schema, das die XML-Syntax eines TOSCA-Definitions-Dokuments definiert. Im Folgenden werden nur die wichtigen Elemente besprochen, die diese Arbeit betreffen.

"?" bedeutet ein optionales Element oder Attribut.

"*" bedeutet null oder mehrere Elemente bzw. Attribute.

"+" bedeutet ein oder mehrere Elemente bzw. Attribute.

"|" bedeutet Auswählen. Zum Beispiel zeigt "a|b" eine Wahl zwischen "a" und "b".

"(" und ")" werden verwendet, um den Rahmen der Operatoren "?", "*", "+" und "|" anzugeben.

Definitions: Das Element ist das Wurzelelement eines TOSCA-Definitions-Dokuments.

Import: Das Element deklariert eine Abhängigkeit von externen TOSCA-Definitionen, XML-Schema-Definitionen oder WSDL-Definitionen. Eine beliebige Anzahl von Elementen *Import* könnten als Kindelemente des Elements *Definitions* erscheinen.

ServiceTemplate: Das Element spezifiziert ein komplettes Service-Template für eine Cloud-Anwendung. Ein Service-Template enthält eine Definition des Topology-Template der Cloud-Anwendung sowie eine beliebige Anzahl von Plänen. Innerhalb des Service-Template können alle Typ-Definitionen wie z.B. Node-Types und Relationship-Types verwendet werden. Diese Typ-Definitionen werden in demselben oder im importierten Definitions-Dokument definiert.

```
01  <Definitions id="xs:ID"
02               name="xs:string"?
03               targetNamespace="xs:anyURI">
04
05      <Extensions>
06          <Extension namespace="xs:anyURI"
07                     mustUnderstand="yes|no"?/> +
08      </Extensions> ?
09
10      <Import namespace="xs:anyURI"?
11              location="xs:anyURI"?
12             importType="xs:anyURI"/> *
13
14      <Types>
15          <xs:schema .../> *
16      </Types> ?
17
18    (
19      <ServiceTemplate> ... </ServiceTemplate>
20    |
21      <NodeType> ... </NodeType>
22    |
23      <NodeTypeImplementation> ... </NodeTypeImplementation>
24    |
25      <RelationshipType> ... </RelationshipType>
26    |
27      <RelationshipTypeImplementation>...
            </RelationshipTypeImplementation>
28    |
29      <RequirementType> ... </RequirementType>
30    |
31      <CapabilityType> ... </CapabilityType>
32    |
33      <ArtifactType> ... </ArtifactType>
34    |
35      <ArtifactTemplate> ... </ArtifactTemplate>
36    |
37      <PolicyType> ... </PolicyType>
38    |
39      <PolicyTemplate> ... </PolicyTemplate>
40    ) +
41
42  </Definitions>
```

Ausschnitt 2.1: XML-Syntax eines TOSCA-Definitions-Dokuments

NodeType: Das Element spezifiziert einen Typ des Knotens, der als ein Typ für die Node-Templates eines Service-Template referenziert werden kann.

NodeTypeImplementation: Das Element spezifiziert die Implementierung des Verwaltbarkeit-Verhaltens ("Manageability Behavior") eines Node-Type, der als ein Typ der Node-Templates eines Service-Template referenziert werden kann.

RelationshipType: Das Element spezifiziert einen Typ der Beziehung ("Relationship"), der als ein Typ für die Relationship-Templates eines Service-Template referenziert werden kann.

RelationshipTypeImplementation: Das Element spezifiziert die Implementierung des Verwaltbarkeit-Verhaltens ("Manageability Behavior") eines Relationship-Type, der als ein Typ der Relationship-Templates eines Service-Template referenziert werden kann.

RequirementType: Das Element spezifiziert einen Typ der Anforderung ("Requirement"), der in den entsprechenden Node-Types deklariert wird.

CapabilityType: Das Element spezifiziert einen Typ der Fähigkeit ("Capability"), der in den entsprechenden Node-Types deklariert wird.

ArtifactType: Das Element spezifiziert einen Typ des Artefakts, das innerhalb eines Service-Template verwendet wird. Beispielsweise könnten Artifact-Types die Anwendungsmodule (z.B. die Dateien mit der Dateiendung ".war" oder ".ear"), die Betriebssystem-Pakete (z.B. RPMs) oder die Image-Dateien von virtuellen Maschinen (z.B. die Dateien mit der Dateiendung ".ova") sein.

ArtifactTemplate: Das Element spezifiziert ein Template, das ein Artefakt beschreibt. Dieses Artefakt wird durch Teile eines Service-Template referenziert. Beispielsweise könnte das installierbare Artefakt für einen Anwendungsserver als ein Artifact-Template definiert werden.

Ein TOSCA-Definitions-Dokument muss mindestens eines der Elemente *ServiceTemplate*, *NodeType*, *NodeTypeImplementation*, *RelationshipType*, *RelationshipTypeImplementation*, *RequirementType*, *CapabilityType*, *ArtifactType*, *ArtifactTemplate*, *PolicyType*, oder *PolicyTemplate* definieren. Es kann aber beliebig viele dieser Elemente in einer beliebigen Reihenfolge definieren.

Diese Technik unterstützt eine modulare Definition von Service-Templates. Beispielsweise kann ein Definitions-Dokument nur die Definitionen von Node-Type und Relationship-Type enthalten, die dann in ein anderes Definitions-Dokument importiert werden können. Das zweite Definitions-Dokument definiert dann nur ein Service-Template und verwendet die importierten Node-Types und Relationship-Types. Ebenso können Node-Type-Properties in separaten XML-Schema-Definitions-Dokumenten definiert werden, die bei dem Definieren eines Node-Type importiert und referenziert werden.

3 Anforderungen an ein automatisches Verfahren

Die Hauptaufgabe der Arbeit ist die Entwicklung eines Verfahrens, mit dem aus den von den DevOps-Communities veröffentlichten Artefakten die entsprechenden TOSCA Service-Templates möglichst automatisch erstellt werden können. Solche Artefakte können durch verschiedene DevOps-Ansätze erstellt werden. Im Grundlagenkapitel wurden zwei Beispiele für die DevOps-Ansätze dargestellt: Chef (Unterkapitel 2.1) und Juju (Unterkapitel 2.2). Da diese Werkzeuge wie Chef und Juju eigene Eigenheiten besitzen, werden die Artefakte auf verschiedene Art und Weise generiert. Das heißt, dass die inneren Strukturen dieser Artefakte unterschiedlich sind. Wegen der Unterschiede dieser DevOps-Ansätze ergeben sich jedoch viele Schwierigkeiten bei der Entwicklung des Verfahrens. Deshalb wird eine hauptsächliche Anforderung an das Verfahren gestellt, die Eigenheiten verschiedener Ansätze wie Chef und Juju sowie ihre Artefakte zu abstrahieren, um diese Unterschiede verbergen zu können. Außerdem folgen daraus zwei Anforderungen: (1) Das Verfahren zur Erzeugung von TOSCA-NodeTypes, Relationship-Types und Service-Templates soll möglichst vollautomatisch sein. (2) Verschiedene Artefakttypen sollen miteinander kombiniert werden. Beispielsweise können sich zwei TOSCA Node-Types in einem Topology-Template kombinieren lassen. Ein Node-Type wurde aus einem Juju-Charm "WordPress" generiert und beinhaltet das Charm. Der andere wurde aus einem Chef-Cookbook "mysql" generiert und beinhaltet das Cookbook. Im Folgenden wird die Haupanforderung ausführlich vorgestellt.

3.1 Abstraktion der Eigenheiten verschiedener DevOps-Ansätze

Um ein automatisches Verfahren zu ermöglichen, werden zwei Abstraktion-Arten besprochen: Prozessabstraktion und Datenabstraktion. Die Prozessabstraktion bedeutet im Rahmen dieser Arbeit die Abbildung der Elemente der Quelle auf die Elemente des Zieles. Die Voraussetzung für die Abbildung ist, die äquivalenten Elemente zwischen der Quelle und dem Ziel zu finden. Damit wird garantiert, dass ein entsprechendes Ziel-Element aus einem bestimmten Quelle-Element generiert werden kann. In unserem Fall sind die Quellen die Artefakte (z.B. Chef-Cookbooks und Juju-Charms), die von verschiedenen DevOps-Ansätzen erstellt werden. Das Ziel ist ein TOSCA Service-Template. Die Hauptaufgabe für die sogenannte Prozessabstraktion ist, die Elemente aus Chef- oder Juju-Artefakten zu den Elementen im TOSCA Service-Template zu transformieren. Die konkreten Konzepte dafür werden im Kaptiel 4 ausführlich dargestellt.

Die Datenabstraktion bedeutet im Rahmen dieser Arbeit, durch Datenkapselung komplexe Objekte abstrakt darzustellen. Die konkrete Aufgabe besteht darin, die komplexe Struktur der von den DevOps-Ansätzen erstellten Artefakte zu modellieren. Im Grundlagenkapitel werden zwei verschiedene DevOps-Ansätze (Chef als ein Konfigurationsmanagementwerkzeug und Juju als ein Service-Orchestrierungswerkzeug) vorgestellt. Durch diese Ansätze können die Artefakte von verschiedenen Typen generiert werden. Um die Unterschiede dieser Artefakte verbergen zu können, werden diese Artefakte abstrahiert. Das Ergebnis der Abstraktion ist, dass für jeden Artefakttyp ein entsprechendes abstraktes Modell generiert wird. Folglich findet die Transformation von diesen Artefakten in TOSCA Service-Templates auf einer abstrakten Modell-Ebene statt. Die Modell-Transformation von den DevOps-Anätzen (Chef und Juju) hin zu TOSCA wird im Unterkapitel 3.3 ausführlich erläutert.

3.2 Grenzen und Einschränkungen

Im letzten Unterkapitel wurde die Anforderung an das automatische Verfahren dargestellt. Das heißt, dass die Eigenheiten verschiedener DevOps-Ansätze abstrahiert werden, um ihre Unterschiede zu verbergen. So können die abstrakten Modelle für die verschiedenen DevOps-Ansätze erstellt werden. Dabei können einige Einschränkungen entstehen. Die Transformation zwischen zwei Modellen ist tatsächlich die Transformation zwischen den äquivalenten Komponenten in zwei Modellen. Falls zwei Modelle das ähnliche Metamodell besitzen, das heißt, dass sie die ähnliche Struktur besitzen und all ihre Komponenten äquivalent sind, dann ist es möglich, durch ein automatisches Verfahren die Modell-Transformation zu ermöglichen. Beispielsweise hat Juju als ein Service-Orchestrierungswerkzeug das ähnliche Metamodell wie TOSCA. Deshalb kann eine äquivalente Modell-Transformation zwischen Juju und TOSCA einfach ermöglicht werden. Falls zwei Modelle kein ähnliches Metamodell besitzen, das heißt, dass es vielleicht in einem Modell ein äquivalenter Teil fehlt, dann wird die Modell-Transformation mithilfe eines automatischen Verfahrens beschränkt. Beispielsweise werden keine Beziehungen zwischen den Chef-Cookbooks explizit definiert. Deshalb können Chef-Cookbooks durch ein automatisches Verfahren zu Relationship-Types im TOSCA Service-Template nicht transformiert werden. Im folgenden Unterkapitel werden zwei Beispiele für die Modell-Transformation beschrieben.

3.3 Modell-Transformation

Dieses Unterkapitel befasst sich mit zwei Beispielen für die Modell-Transformation. Zuerst wird die Modell-Transformation von Chef als ein Konfigurationsmanagementwerkzeug nach TOSCA dargestellt. Dann wird die Modell-Transformation von Juju als ein Service-Orchestrierungswerkzeug nach TOSCA besprochen.

3.3.1 Modell-Transformation von Chef nach TOSCA

Aus den vorigen Inhalten wissen wir, dass TOSCA zur Realisierung vom Modell-getriebenen Cloud-Management verwendet wird, indem das Service-Topologie-Modell ("Service-Template") auf einer höheren Ebene spezifiziert wird. Solche Service-Topologie-Modelle von TOSCA spezifizieren grundsätzlich die Graphen. Diese Graphen bestehen aus den Knoten und den Beziehungen zwischen den Knoten, um die ganze Struktur für das Deployment eines Service zu bestimmen. In TOSCA werden die Beziehungen und die Knoten als separate Topologie-Modell-Komponenten explizit modelliert. Um dies genauer zu formulieren, beschreibt eine solche Spezifikation das Topology-Template, die Node-Types und die Relationship-Types. Darüber hinaus müssen die Implementation-Artifacts auf der unteren Ebene an die Operationen von Node-Types und Relationship-Types angehängt werden, um die Funktionalität dieser Operationen wie die Installierung und die Konfiguration einer bestimmten Software-Komponente zu realisieren. Aber TOSCA konzentriert sich nicht auf die Aspekte der unteren Ebene, wie z.B. die Implementation-Artifacts auf der unteren Ebene. Der geradlinige Ansatz ist, Shell-Skripte zu implementieren, um die Software-Komponenten zu installieren und zu konfigurieren. Da Shell-Skripte für die Ausführung der einfachen Aufgaben auf einer spezifischen Plattform verwendet werden sollen, wäre eine plattformunabhängige Skriptsprache wie Python oder Ruby eine bessere Wahl, um die portablen Artefakte zu erstellen.

Chef als ein beliebtes Konfigurationsmanagementwerkzeug kann diesen Schwachpunkt ausgleichen. Für Chef gibt es schon viele Konfigurationsdefinitionen wie z.B. Cookbooks für den Apache-Webserver und den MySQL-Datenbankserver, die bereits verfügbar sind, um viele Software-Komponenten zu installieren und zu konfigurieren. Da diese Konfigurationsdefinitionen durch die plattformunabhängige Skriptsprache "Ruby" und eine interne DSL geschrieben werden, sind sie portabel und können als die Implementation-Artifacts auf der unteren Ebene von TOSCA dienen. Außerdem können keine expliziten Beziehungen zwischen diesen Konfigurationsdefinitionen definiert werden, da es kein ganzheitliches Service-Topologie-Modell in Chef gibt. Folglich muss nur ein Schritt durchgeführt werden, um die von der Chef-Community veröffentlichten Konfigurationsdefinitionen zu den TOSCA-konformen Topologie-Modell-Komponenten zu verwandeln. Aus jedem Chef-Cookbook muss ein Node-Type in TOSCA generiert werden. Die Relationship-Types können jedoch in TOSCA nicht erzeugt werden, weil die entsprechenden Beziehungen zwischen den Cookbooks in Chef fehlen. In Abbildung 3.1 wird ein Beispiel für die Modell-Transformation von Chef nach TOSCA gezeigt.

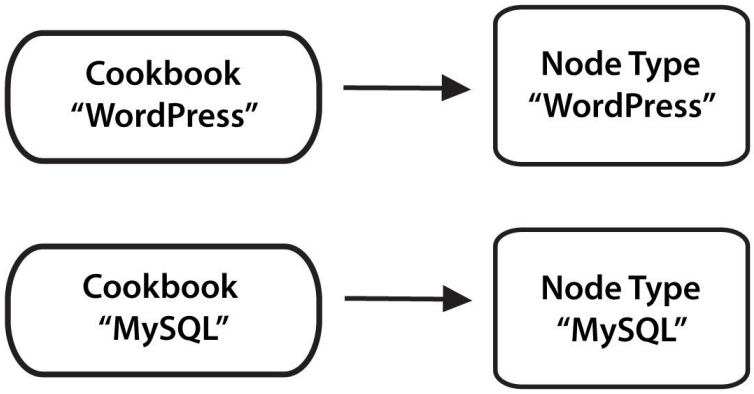

Abbildung 3.1: Beispiel für Modell-Transformation von Chef nach TOSCA

In diesem Besipiel wird dargestellt, dass aus den Chef-Cookbooks "Web-Applikation" und "Datenbankserver" zwei entsprechende Node-Types generiert werden, die die Knoten in einem TOSCA Topologie-Modell repräsentieren. Aus der Beziehung "Die Web-Applikation verbindet sich mit dem MySQL-Datenbankserver" wird kein Relationship-Type erzeugt, die die entsprechende Beziehung in einem TOSCA Topologie-Modell repräsentieren kann. Der Grund dafür ist, dass in Chef keine Beziehung zwischen diesen zwei Cookbooks explizit definiert wird. Auch wenn eine solche Beziehung zwischen ihnen implizit existieren könnte, ist es ziemlich schwierig, aus dieser Beziehung ein entsprechender TOSCA Relationship-Type zu generieren.

3.3.2 Modell-Transformation von Juju nach TOSCA

Aus dem Grundlagenkapitel wissen wir, dass die Juju-Community mehr als einhundert Topologie-Modell-Komponenten als Open-Source-Software zur Verfügung stellt. Eine solche Komponente wird ein "Charm" genannt und kann mit einer anderen Komponente kombiniert werden, um ein Service-Topologie-Modell zu erstellen. Dieses Service-Topologie-Modell kann in der Cloud-Umgebung instanziiert und verwaltet werden. Das heißt, dass Juju als ein beliebtes Service-Orchestrierungswerkzeug das ähnliche Metamodell wie TOSCA hat und das Modell-getriebene Cloud-Management ermöglicht. Im Grunde spezifizieren die Topologie-

Modelle von TOSCA und Juju die Graphen. Diese Graphen bestehen aus den Knoten und den Beziehungen ("Relations") zwischen den Knoten, um die Struktur eines Cloud-Service zu definieren. In TOSCA werden die Beziehungen und die Knoten als separate Topologie-Modell-Komponenten explizit modelliert, während Juju nur die Knoten als Topologie-Modell-Komponenten spezifiziert. Folglich müssen zwei wichtige Schritte durchgeführt werden, um die von der Juju-Community veröffentlichten Topologie-Modell-Komponenten zu den TOSCA-konformen Topologie-Modell-Komponenten zu verwandeln [1]. (1) Aus jedem Juju-Charm muss eine TOSCA-konforme Topologie-Modell-Komponente generiert werden. Demzufolge kann jeder Knoten, der mit Juju modelliert werden kann, auch mit TOSCA modelliert werden. Die Beziehungen zwischen diesen Knoten können jedoch in TOSCA nicht modelliert werden, weil die entsprechenden Topologie-Modell-Komponenten fehlen. (2) Deshalb müssen zusätzliche TOSCA-konformen Topologie-Modell-Komponenten aus jeder Beziehung, die mit Juju implizit modelliert werden kann, generiert werden. In Abbildung 3.2 wird ein Beispiel für die Modell-Transformation von Juju nach TOSCA gezeigt.

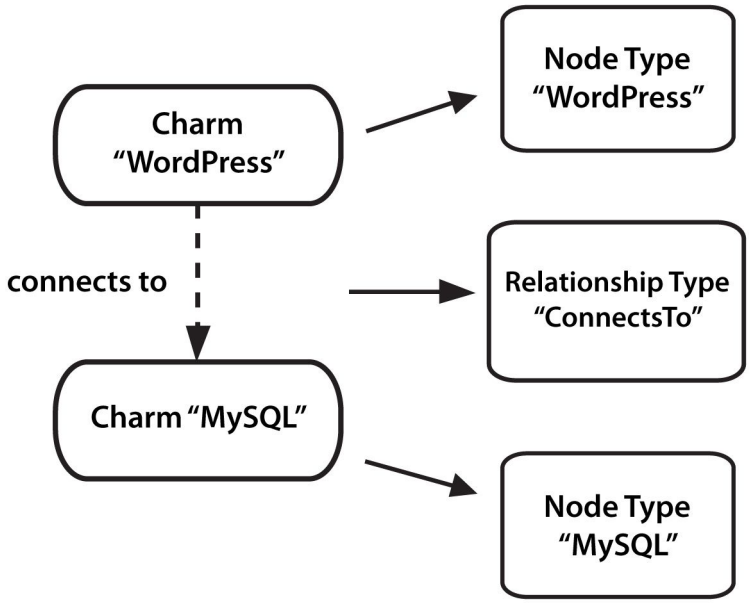

Abbildung 3.2: Beispiel für Modell-Transformation von Juju nach TOSCA

In diesem Beispiel wird dargestellt, dass aus den Juju-Charms "WordPress" und "MySQL" zwei entsprechende Topologie-Modell-Komponenten generiert werden. Diese Topologie-Modell-Komponenten repräsentieren die Knoten in einem TOSCA Topologie-Modell ("Service-Topology"). Aus der Relation "Die WordPress-Applikation verbindet mit dem MySQL-Datenbankserver", die in Juju implizit modelliert werden kann, wird eine separate Topologie-Modell-Komponente erzeugt, die die entsprechende Beziehung in einem TOSCA Topologie-Modell repräsentieren kann.

4 Konzepte für ein automatisches Verfahren

In diesem Kapitel werden die Konzepte für das automatische Verfahren vorgestellt, mit dem die TOSCA Service-Templates aus den von den DevOps-Communities veröffentlichten Artefakten automatisch generiert werden. Zu den konkreten Konzepten gehören das Konzept für die Erzeugung von TOSCA Node-Types aus bestehenden Chef-Artefakten, das Konzept für die Erzeugung von TOSCA Relationship-Types aus bestehenden Juju-Artefakten und das Konzept für die Erzeugung von TOSCA Service-Templates durch die Orchestrierung der generierten Node-Types und Relationship-Types.

4.1 Erzeugung von TOSCA Node-Types aus bestehenden Chef-Artefakten

Dieses Unterkapitel beschäftigt sich mit dem Konzept für die Generierung von TOSCA Node-Types aus Chef-Artefakten. Im Folgenden wird beschrieben, wie die Abbildung der Elemente in einem Chef-Cookbook zu den Elementen in einem TOSCA-Definitions-Dokument für einen Node-Type realisiert wird. Die Idee ist, die äquivalenten Elemente zwischen ihnen zu finden. Zuerst wird durch die Attribute, die in der Matadatei namens "metadata.rb" in einem Cookbook definiert werden, ein entsprechendes Dokument für die Node-Type-Properties erzeugt. Dann wird durch die anderen Informationen in dieser Matadatei ein entsprechendes TOSCA-Definitions-Dokument für einen Node-Type generiert. Schließlich wird die entsprechende CSAR-Datei erzeugt, die alle notwendigen Dokumente und Artefakte enthält.

4.1.1 Erzeugung des Node-Type-Properties-Dokuments

In einem Cookbook oder in einem Recipe können Attribute definiert werden. Ein Attribut ist eine bestimmte Information über den Knoten, wie z.B. ein Hostname, eine IP-Adresse, eine Netzwerk-Schnittstelle, ein Dateisystem, die Anzahl der Klienten, die ein auf einem Knoten laufender Service akzeptieren, und so weiter. Wenn ein Cookbook während des Chef-Runs geladen wird, werden diese Attribute mit den Attributen verglichen, die bereits auf dem Knoten vorhanden sind. Wenn die Cookbook-Attribute die höhere Priorität als die Default-Attribute haben, wird Chef die Werte der neuen Attribute auf dem Knoten verwenden.

Ein TOSCA Node-Type kann durch die Node-Type-Properties die Eigenschaften (die in Abbildung 2.10 dargestellten "Properties") einer Software-Komponente definieren. Diese Eigenschaften werden verwendet, um die Informationen über eine Service-Instanz zu speichern: die statische Information (z.B. die Hardware-Spezifikation einer virtuellen Maschine) und die Laufzeit-Information (z.B. die IP-Adresse). Die Node-Type-Properties können in separaten XML-Schema-Definitionen definiert werden, die beim Definieren eines Node-Type importiert werden.

Es gibt eine bestimmte Beziehung zwischen den Eigenschaften des TOSCA Node-Type und den Attributen des Chef-Cookbook. Für eine Interface-Operation des Node-Type wird ein entsprechendes Artifact-Template als ein Implementation-Artifact referenziert. In unserem Fall wird hier ein Artifact-Template des Chef-spezifischen Artifact-Type aufgerufen. Für die Ausführung eines Chef-spezifischen Artifact-Template werden die Werte der Attribute im Cookbook benötigt. Um das zu verwirklichen, können die Eigenschaften des Node-Type (die Node-Type-Properties) zu Cookbook-Attributen abgebildet werden. Deshalb kann aus den Attributen in der Metadatei eines Cookbook ein entsprechendes Node-Type-Properties-

Dokument generiert werden. Das Dokument besitzt die Dateiendung ".xsd" (XML Schema Definition) und wird zum Definieren der Struktur für ein XML-Element verwendet. Das XML-Element definiert die Struktur der Node-Type-Properties. Ein Beispiel für das Node-Type-Properties-Dokument, das aus den Attributen des Cookbook "mysql" generiert wurde, wird im Anhang 1 präsentiert.

4.1.2 Erzeugung von Requirement-Types und Capability-Types

Ein Requirement-Type ist eine wiederverwendbare Entität, die eine Art Anforderung ("Requirement") beschreibt. Ein Node-Type kann deklarieren, solche Anforderungen (die in Abbildung 2.10 dargestellten "Requirement Definitions") zu besitzen. Beispielsweise kann ein Requirement-Type für eine Datenbankverbindung definiert werden. Verschiedene Node-Types (z.B. ein Node-Type für eine Web-Anwendung) können deklarieren, eine Anforderung für eine Datenbankverbindung zu besitzen.

Ein Capability-Type ist eine wiederverwendbare Entität, die eine Art Fähigkeit ("Capability") beschreibt. Ein Node-Type kann deklarieren, solche Fähigkeiten (die in Abbildung 2.10 dargestellten "Capability Definitions") bereitstellen zu können. Beispielsweise kann ein Capability-Type für einen Datenbankserver definiert werden. Verschiedene Node-Types (z.B. ein Node-Type für eine Datenbank) können deklarieren, die Fähigkeit eines Datenbankservers zur Verfügung zu stellen. Im Grunde legen die Requirement-Types die Anforderungen an einen Node-Type fest. Im Gegensatz dazu definieren die Capability-Types die Fähigkeiten, die ein Node-Type zur Verfügung stellen kann.

Wir wissen, dass ein Chef-Cookbook alle erforderlichen Informationen und Ressourcen für die Konfiguration eines Service oder einer Applikation enthält. Das heißt, dass aus einem Cookbook genau eine konkrete Fähigkeit, wie z.B. einen MySQL-Datenbankserver zu konfigurieren, verwirklicht werden kann. Die Anforderungen können jedoch nicht genauer im TOSCA Topologie-Modell beschrieben werden. Der Grund dafür ist, dass sich ein Cookbook auf der unteren Ebene im TOSCA Topologie-Modell befindet und als ein Implementation-Artifact für eine Interface-Operation eines TOSCA Node-Type verwendet wird. Deshalb kann nur ein entsprechender Capability-Type aus einem Chef-Cookbook generiert wird.

Der XML-Ausschnitt 4.1 zeigt ein Beispiel für den Capability-Type bezüglich des Chef-Cookbook "mysql".

```
01  <CapabilityType name="mysql"
02      targetNamespace="http://community.opscode.com/cookbooks/
03                       mysql/capabilites"/>
```

Ausschnitt 4.1: XML-Syntax für den Capability-Type

Der aus dem Cookbook "mysql" generierten Node-Type stellt genau eine Fähigkeit "mysql" zur Verfügung. Folglich wird ein entsprechender Capability-Type names "mysql" generiert.

4.1.3 Erzeugung von Artifact-Types und Artifact-Templates

Die Erzeugung von Artifact-Types und Artifact-Templates ist eine sehr wichtige Aufgabe für diese Arbeit, um aus Chef-Cookbooks die entsprechenden TOSCA Node-Types zu generieren. Artifact-Types sind wiederverwendbare Entitäten und definieren Typen von Artifact-

Templates. Diese Artifact-Templates dienen als Deployment-Artifacts für Node-Templates oder als Implementation-Artifacts für die Interface-Operationen von Node-Types und Relationship-Types. In unserem Fall wird hier ein Artifact-Template als Implementation-Artifact für die Interface-Operationen eines Node-Type referenziert. Das Artifact-Template definiert in der Regel die Werte der Eigenschaften innerhalb des Elements *Properties*. Außerdem stellt normalerweise ein Artifact-Template eine Referenz oder mehrere Referenzen auf das tatsächliche Artefakt selbst zur Verfügung. Es kann eine Datei in der CSAR-Datei sein, welche das gesamte Service-Template enthält. Es kann auch an einem entfernten Ort wie einem FTP-Server verfügbar sein.

Im Folgenden wird dargestellt, wie die Artifact-Templates und die entsprechenden Artifact-Types aus Chef-Cookbooks generiert werden. Das heißt, wie die Chef-Cookbooks als die Implementation-Artifacts in ein TOSCA Service-Topologie-Modell integriert werden. Es gibt zwei verschiedene Ansätze [20] zur Einbettung von Chef-Cookbooks in ein Service-Topologie-Modell: die direkte Integration und die transparente Integration.

4.1.3.1 Die direkte Integration

Der direkte Ansatz zur Einbettung von Chef-Cookbooks in ein Service-Template ist, einen Chef-spezifischen Artifact-Type zu definieren, welche der Struktur der Artefakte "Chef-Cookbooks" entspricht. Außerdem können die Implementation-Artifacts des Chef-spezifischen Artifact-Type von einer beliebigen TOSCA-Laufzeitumgebung bearbeitet werden. Beliebige Typen von Implementation-Artifacts können durch das Erstellen einer entsprechenden XML-Schema-Definition definiert werden. Eine TOSCA-Laufzeitumgebung, die ein Service-Template mit diesen Implementation-Artifacts verarbeitet, muss den entsprechenden Artifact-Type verstehen. Das heißt, dass die TOSCA-Laufzeitumgebung den Chef-spezifischen Artifact-Type verstehen muss, um die entsprechenden Implementation-Artifacts verarbeiten zu können.

Ein Artifact-Template wird innerhalb eines TOSCA Service-Template definiert und kann dann als ein Implementation-Artifact für eine bestimmte Operation referenziert werden. Durch den direkten Integrationsansatz können hier Chef-spezifische Artifact-Templates erzeugt werden. Die TOSCA-Laufzeitumgebung muss den Inhalt des Chef-spezifischen Artifact-Template (insbesondere den Inhalt des Elements *ChefArtifactProperties*) verstehen, um die entsprechenden Aktionen durchführen zu können. Das Element *ChefArtifactProperties* wird in der XML-Schema-Definition (in der Datei "ChefArtifact.xsd") für den Chef-spezifischen Artifact-Type definiert und verwendet, um die Struktur der Chef-spezifischen Artifact-Type-Properties zu definieren. Die Datei "ChefArtifact.xsd" wird im Anhang 4 präsentiert. Alle Chef-bezogenen Informationen wie das Mapping von Node-Type-Properties zu Cookbook-Attributen und die Run-Liste, in der die erforderlichen Recipes für die Konfiguration eines Service oder einer Anwendung gelistet sind, werden im Element *ChefArtifactProperties* definiert.

4.1.3.2 Die transparente Integration

Chef-Cookbooks können auch auf eine transparente Weise unter Verwendung des Standard-Artifact-Type "Script Artifact" [64] in ein Service-Topologie-Modell eingebettet werden. Die Artefakte des Type "Script Artifact" können von einer beliebigen TOSCA-Laufzeitumgebung verarbeitet werden. Folglich muss die TOSCA-Laufzeitumgebung die Implementation-

Artifacts der spezifischen Artifact-Types wie die Chef-spezifische Artefakte nicht verstehen. Dieser transparente Integrationsansatz kann in der Praxis durch die Erstellung der Wrapper-Skripte realisiert werden. Wrapper-Skripte können das Konfigurationsmanagementwerkzeug mit entsprechenden Parametern aufrufen, um auf die Konfigurationsdefinitionen in Cookbooks zu verweisen.

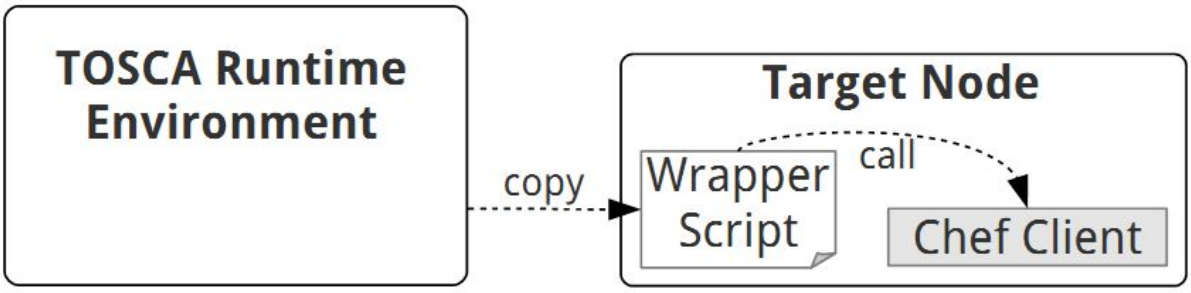

Abbildung 4.1: Transparente Integration mithilfe eines Wrapper-Skripts [20]

Abbildung 4.1 zeigt, wie die Wrapper-Skripte verwendet werden, um die Operationen auf einem Target-Knoten unter Verwendung von Chef durchzuführen. Erstens kopiert die TOSCA-Laufzeitumgebung das entsprechende Wrapper-Skript zu dem Target-Knoten und triggert dann die Ausführung des Wrapper-Skripts. Zusätzliche Dateien, die für die Ausführung des Wrapper-Skripts erforderlich sind, werden auch zu den Target-Knoten kopiert. Zweitens ruft das Wrapper-Skript den Chef-Client. Diese Wrapper-Skripte können als die Implementation-Artifacts des Typs "Script Artifact" in das TOSCA Topologie-Modell eingebettet werden.

Einige Einschränkungen sind im Wrapper-Skript fest kodiert, wie z.B. die Konfiguration des Chef-Client, der die Konfigurationsdefinitionen in Cookbooks verarbeitet, die Run-Liste sowie das Mapping von Eigenschaften. Alle Chef-bezogenen Informationen sind im Wrapper-Skript versteckt. Das führt zum Verlust von Flexibilität, die Konfigurationsdefinitionen zu verarbeiten, weil die TOSCA-Laufzeitumgebung nicht versteht, was in einem Wrapper-Skript geschieht. Sie kann nicht kontrollieren, wie das Wrapper-Skript seine Arbeit ausführt. Darüber hinaus ist es schwer, die Wrapper-Skripte portabel zu machen.

4.1.3.3 Der bevorzugte Ansatz

Für den zweiten Integrationsansatz sind die Chef-Spezifika des Artifact-Template für die TOSCA-Laufzeitumgebung vollständig transparent. Die Laufzeit-Implementierung muss nichts über Chef wissen. Alle notwendigen Aktionen zur Realisierung des Artefakts sind jedoch im Wrapper-Skript fest kodiert. Die TOSCA-Laufzeitumgebung hat wenig Flexibilität bei der Durchführung der entsprechenden Aktionen. Die Mappings von Eigenschaften und Parametern müssen im Wrapper-Skript versteckt sein. Im Gegensatz dazu sind für den direkten Integrationsansatz diese Mappings im Artifact-Template explizit definiert. Außerdem wird kein Chef-Client bei dem direkten Ansatz aufgerufen. Deshalb wird der direkte Integrationsansatz in Bezug auf Chef in dieser Arbeit bevorzugt.

Der XML-Ausschnitt 4.2 zeigt ein Beispiel für ein Chef-spezifisches Artifact-Template, das einen MySQL-Datenbankserver installiert und konfiguriert. Das Beispiel zeigt die Struktur eines Artifact-Template aus einem Chef-spezifischen Artifact-Type.

```
<ArtifactTemplate id="322de67d-721b-4ab5-ae4f-a280076496c2"
       xmlns:artifacts="http://docs.oasis-open.org/
                        tosca/ns/2011/12/Artifacts"
       type="artifacts:ChefArtifact">
  <Properties>
    <artifacts:ChefArtifactProperties
        xmlns:artifacts="http://docs.oasis-open.org/
                         tosca/ns/2011/12/Artifacts">
      <Cookbooks>
        <Cookbook name="build-essential"
             location="/files/chef/cookbooks/build-essential.zip"/>
        <Cookbook name="openssl"
             location="/files/chef/cookbooks/openssl.zip"/>
        <Cookbook name="mysql"
             location="/files/chef/cookbooks/mysql.zip"/>
      </Cookbooks>
      <Mappings>
        <PropertyMapping property="/mysql/server_root_password"
             cookbookAttribute="mysql/server_root_password"
             mode="input"/>
        ...
      </Mappings>
      <RunList>
        <Include>
          <RunListEntry cookbookName="mysql" recipeName="server"/>
        </Include>
      </RunList>
    </artifacts:ChefArtifactProperties>
  </Properties>
  <ArtifactReferences>
    <ArtifactReference reference="/files/chef/cookbooks/">
      <Include pattern="build-essential.zip"/>
      <Include pattern="openssl.zip"/>
      <Include pattern="mysql.zip"/>
    </ArtifactReference>
  </ArtifactReferences>
</ArtifactTemplate>
```

Ausschnitt 4.2: XML-Syntax für das Chef-spezifische Artefact-Template

Das Element *ArtifactTemplate* besitzt zwei Attribute: *id* und *type*. Das Attribut *id* spezifiziert einen eindeutigen Bezeichner für dieses bestimmte Artifact-Template innerhalb des Service-Template. Das Attribut *type* spezifiziert den Artifact-Type. Der Inhalt des Elements *Properties* ist Chef-spezifisch. Die Struktur des Elements *ChefArtifactProperties* kann unter Verwendung einer XML-Schema-Definition definiert werden. Sie enthält die folgenden Teile:

Cookbooks: Jedes Cookbook, das zur Realisierung eines bestimmten Artifact-Template erforderlich ist, wird durch ein Element *Cookbook* referenziert. Eingeschlossen werden hier die Cookbooks, die innerhalb der Run-Liste direkt referenziert werden, sowie die Cookbooks, von denen die in der Run-Liste referenzierten Cookbooks abhängig sind. Diese Cookbooks werden durch die Abschnitte "depends" in der Metadatei eines Cookbook gezeigt.

Mappings: Werte der Node-Type-Properties können zu Cookbook-Attributen durch das Element *PropertyMapping* abgebildet werden. Das Attribut *propertyPath* enthält einen XPath-Ausdruck, der auf eine bestimmte Eigenschaft verweist. Das Attribut *cookbookAttribute* wird verwendet, um auf das entsprechende Cookbook-Attribut zu referenzieren. Das Attribut *mode* spezifiziert die Richtung des Mapping und besitzt zwei Werte "input" und "output". Der Wert "input" bedeutet, dass der Wert der Eigenschaft zum Cookbook-Attribut zugeordnet wird, bevor die Chef-Recipes ausgeführt werden. Der Wert "output" ordnet das Cookbook-Attribut zur Eigenschaft zu, nachdem die Chef-Recipes ausgeführt werden.

RunList: Im Wesentlichen definiert ein Chef-spezifisches Artifact-Template, welche Recipes in der entsprechenden Run-Liste gezeigt werden. Das Element *RunList* enthält ein Kindelement *Include*. Das Element *Include* besteht aus mindestens einem Element *RunListEntry*, das auf Recipes verweist. Die Recipes in der Run-Liste werden in der angegebenen Reihenfolge ausgeführt. In der Metadatei eines Cookbook werden viele Recipes durch die Abschnitte "recipe" gezeigt. Ein oder mehrere Recipes können ausgewählt und zur Run-Liste hinzugefügt werden.

Das Element *ArtifactReferences* enthält einen Verweis oder mehrere Verweise auf die Dateien, die in die CSAR-Datei gelegt werden und für die Verarbeitung des Artefakts notwendig sind. Im Beispiel wird das Cookbook "mysql" referenziert. Außerdem werden noch zwei Cookbooks "build-essential" und "openssl" referenziert, von denen das Cookbook "mysql" abhängig ist.

4.1.4 Erzeugung von Node-Type und Node-Type-Implementation

Ein Node-Type ist eine wiederverwendbare Entität, die den Typ eines Node-Template oder von mehreren Node-Templates definiert. Das Element *NodeType* bezüglich eines Chef-Cookbook besteht aus den folgenden Elementen: *PropertiesDefinition*, *CapabilityDefinition* und *Interfaces*.

Der XML-Ausschnitt 4.3 zeigt ein Beispiel für einen Node-Type bezüglich des Chef-Cookbook "mysql".

In dem Unterkapitel 4.1.1 wird beschrieben, dass aus den Attributen in der Metadatei eines Cookbook ein entsprechendes Node-Type-Properties-Dokument generiert werden kann. Durch das Element *PropertiesDefinition* kann die Struktur der Eigenschaften des Node-Type wie z.B. seine Konfiguration und sein Zustand mittels XML-Schema spezifiziert werden. In unserem Fall besitzt das Element *PropertiesDefinition* das Attribut *element*, das den Namen eines XML-Elements angibt. Dieses XML-Element definiert die Struktur der Node-Type-Properties.

Unterkapitel 4.1.2 zeigt, dass die aus Chef-Cookbooks generierten Node-Types keine Anforderungen haben. Für jeden aus einem Chef-Cookbook generierten Node-Type sollte genau eine Fähigkeit generiert werden. Das Element *CapabilityDefinition* definiert eine bestimmte Fähigkeit, die der Node-Type zur Verfügung stellen kann. Durch das Attribut *capabilityType* kann der entsprechende Capability-Type identifiziert werden. Für das Cookbook "mysql" hat beispielsweise der entsprechende Node-Type "MySQL" genau eine Fähigkeit "mysql".

```
01    <NodeType name="mysql_nodetype"
02        targetNamespace="http://community.opscode.com/cookbooks">
03      <PropertiesDefinition
04          xmlns:properties="http://community.opscode.com/
05                            cookbooks/mysql/nodetype_properties"
06          element="properties:mysql-properties"/>
07      <CapabilityDefinitions>
08        <CapabilityDefinition name="mysql"
09            xmlns:capabilites="http://community.opscode.com/
10                               cookbooks/mysql/capabilites"
11            capabilityType="capabilites:mysql"
12            lowerBound="0" upperBound="unbounded"/>
13        </CapabilityDefinitions>
14      <Interfaces>
15        <Interface
16            name="http://docs.oasis-open.org/
17                  tosca/ns/2011/12/interfaces/server">
18          <Operation name="create"/>
19        </Interface>
20      </Interfaces>
21    </NodeType>
```

Ausschnitt 4.3: XML-Syntax für den Node-Type

Die Funktionen, die auf (einer Instanz von) einem entsprechenden Node-Template durchgeführt werden können, werden durch die Interfaces des Node-Type definiert. Das Element *Interfaces* enthält die Definitionen der Operationen, die auf (Instanzen von) dem Node-Type durchgeführt werden können. Beispielsweise werden die Operationen wie z.B. "install", "start", "stop" verwendet, um in der TOSCA-Umgebung den Lebenszyklus eines Service oder einer Anwendung durchzuführen und zu verwalten. Aber Chef befindet sich auf der Ebene von Implementation-Artifacts in TOSCA. Es gibt bei Chef keine solchen Lebenszyklus-Operationen. Um das Deployment eines aus einem Chef-Cookbook generierten CSAR zu ermöglichen, wird eine Lebenszyklus-Operation "create" für jeden Node-Type im Element *Interface* definiert. Implementation-Artifacts zur Installierung und Konfigurierung der entsprechenden Software-Komponenten werden an diese Operation "create" angehängt.

Eine Node-Type-Implementation beschreibt den ausführbaren Code, der einen spezifischen Node-Type implementiert. Die Node-Type-Implementation stellt eine Sammlung von ausführbaren Dateien oder Programmen zur Verfügung, welche die Interface-Operationen eines Node-Type (auch bekannt als Implementation-Artifacts) implementieren. Außerdem

stellt sie eine Sammlung von ausführbaren Dateien oder Programmen zur Verfügung, die nötig sind, um die Instanzen von Node-Templates (auch bekannt als Deployment-Artifacts) zu erstellen. Diese ausführbaren Dateien oder Programme werden als separate Artifact-Templates definiert und von den Implementation-Artifacts und den Deployment-Artifacts einer Node-Type-Implementation referenziert.

Der XML-Ausschnitt 4.4 zeigt ein Beispiel für eine Node-Type-Implementation, die das Chef-spezifische Artifact-Template (dargestellt in Unterkapitel 4.1.3.3) mit der Interface-Operation "create" im aus dem Chef-Cookbook "mysql" generierten Node-Type verknüpfen kann.

```
01  <NodeTypeImplementation
02      xmlns:cookbooks="http://community.opscode.com/cookbooks"
03      name="mysql_nodetypeimplementation"
04      targetNamespace="http://community.opscode.com/cookbooks"
05      nodeType="cookbooks:mysql_nodetype">
06    <RequiredContainerFeatures>
07      <RequiredContainerFeature
08          feature="redhat|amazon|centos|debian|ubuntu|freebsd|
09                   mac_os_x|scientific|suse|windows"/>
10    </RequiredContainerFeatures>
11    <ImplementationArtifacts>
12      <ImplementationArtifact name="create"
13          xmlns:artifacts="http://docs.oasis-open.org/
14                           tosca/ns/2011/12/Artifacts"
15          xmlns:mysql="http://community.opscode.com/
16                       cookbooks/mysql/nodetype"
17          interfaceName="http://docs.oasis-open.org/
18                         tosca/ns/2011/12/interfaces/server"
19          operationName="create"
20          artifactType="artifacts:ChefArtifact"
21          artifactRef="mysql:322de67d-721b-4ab5-ae4f-a280076496c2"/>
22    </ImplementationArtifacts>
23  </NodeTypeImplementation>
```

Ausschnitt 4.4: XML-Syntax für die Node-Type-Implementation

Im Fall des aus dem Chef-Cookbook "mysql" generierten Node-Type wird ein entsprechendes Implementation-Artifact für die Operation "create" durch das Element *ImplementationArtifact* definiert. Dieses Element hat die folgenden Attribute: *name*, *artifactType*, *artifactRef*, *interfaceName* und *operationName*. Das Attribut *name* spezifiziert den Namen des Artefakts. Das Attribut *artifactType* spezifiziert den Typ des Artefakts. Das optionale Attribut *artifactRef* enthält einen Namen, der ein Artifact-Template als ein Implementation-Artifact identifiziert. Der Wert des Attributs *artifactRef* entspricht dem Wert des Attributs *id* im Artifact-Template (dargestellt in Unterkapitel 4.1.3.3). Das optionale Attribut *interfaceName* spezifiziert den Namen des Interface und entspricht dem Interface-Namen im entsprechenden Node-Type. Das optionale Attribut *operationName* spezifiziert den Namen der Operation und referenziert die Interface-Operation "create" im entsprechenden Node-Type. Außerdem kann

die Node-Type-Implementation das Element *RequiredContainerFeatures* besitzen. Dieses Element spezifiziert die Hinweise für einen TOSCA-Container, dass er eine Implementierung, die einer bestimmten Umgebung entspricht, richtig auswählen kann.

4.1.5 Erzeugung der entsprechenden CSAR-Datei

Aus dem Grundlagenkapitel wissen wir, dass eine TOSCA CSAR-Datei erstellt werden muss, um die Verfügbarkeit aller Elemente für eine Cloud-Anwendung in einer TOSCA-konformen Umgebung zu garantieren. Diese CSAR-Datei enthält alle erforderlichen Elemente für die Cloud-Anwendung. Im Fall von Chef gehören zu solchen Elementen das Node-Type-Properties-Dokument, das TOSCA-Definitions-Dokument für den Node-Type, die TOSCA-Metadatei "TOSCA.meta" sowie alle erforderlichen Artefakte wie z.B. Cookbooks und Recipes. Außerdem gibt es die Möglichkeit, dass ein Cookbook von den anderen Cookbooks abhängig ist, welche auch in der Matadatei definiert werden. Solche Cookbooks müssen auch zu der aus diesem Cookbook generierten CSAR-Datei integriert werden. Beispielsweise wird in der Metadatei des Cookbook "mysql" definiert, dass dieses Cookbook von zwei anderen Cookbooks "openssl" und "build-essential" abhängig ist. Diese Cookbooks müssen auch zu der aus dem Cookbook "mysql" generierten CSAR-Datei integriert werden. Wenn die Ausführung von Cookbooks "openssl" und "build-essential" auch von den anderen Cookbooks abhängig ist, dann müssen diese entsprechenden Cookbooks in die CSAR-Datei integriert werden. Im Fall des Cookbook "mysql" werden nur die drei Cookbooks "mysql", "openssl" und "build-essential" in die CSAR-Datei integriert, weil die Cookbooks "openssl" und "build-essential" von keinem Cookbook abhängig sind.

4.2 Erzeugung von TOSCA Node-Types aus bestehenden Juju-Artefakten

In diesem Unterkapitel wird das Konzept für die Generierung von TOSCA Node-Types aus Juju-Artefakten kurz dargestellt. Die Erzeugung von TOSCA Node-Types aus Juju-Artefakten gehört nicht zum Rahmen dieser Arbeit. Es wird hier besprochen, damit die Leser das Konzept für die Generierung von TOSCA Relationship-Types aus Juju-Artefakten besser verstehen können. Im Folgenden wird durch eine Abbildung dargestellt, wie ein TOSCA Node-Type aus einem Juju-Charm generiert wird. In der Abbildung 4.2 gibt es einige gestrichelten Linien, die zeigen, dass durch eine Datei in einem Charm eine entsprechende Datei für eine TOSCA CSAR-Datei generiert werden kann.

Als erster Schritt wird durch die YAML-Datei namens "config" im Charm ein entsprechendes Node-Type-Properties-Dokument erzeugt. Einige Konfigurationsoptionen wie z.B. eine IP-Adresse oder eine Netzwerk-Schnittstelle werden in dieser YAML-Datei definiert und vom Charm benutzt, um die Informationen über eine Service-Instanz zu speichern. Solche Konfigurationsoptionen entsprechen den Eigenschaften ("Properties") des TOSCA Node-Type. Eine Hook-Datei wird hier in einem Artefakt-Template definiert und als ein konkretes Implementation-Artifact für eine Interface-Operation des Node-Type aufgerufen. Für die Ausführung eines Hooks werden die Werte der Konfigurationsoptionen benötigt. Deshalb brauchen wir ein Node-Type-Properties-Dokument, in dem die entsprechenden Eigenschaften für die Ausführung dieser "Hook" Artefakt-Templates definiert werden. Die Node-Type-Properties können in separaten XML-Schema-Definitionen definiert werden, die beim Definieren eines Node-Type importiert werden. Es gibt noch die Möglichkeit, dass es keine

YAML-Datei "config" im Charm gibt. Das bedeutet, dass kein entprechendes Node-Type-Properties-Dokument generiert werden muss.

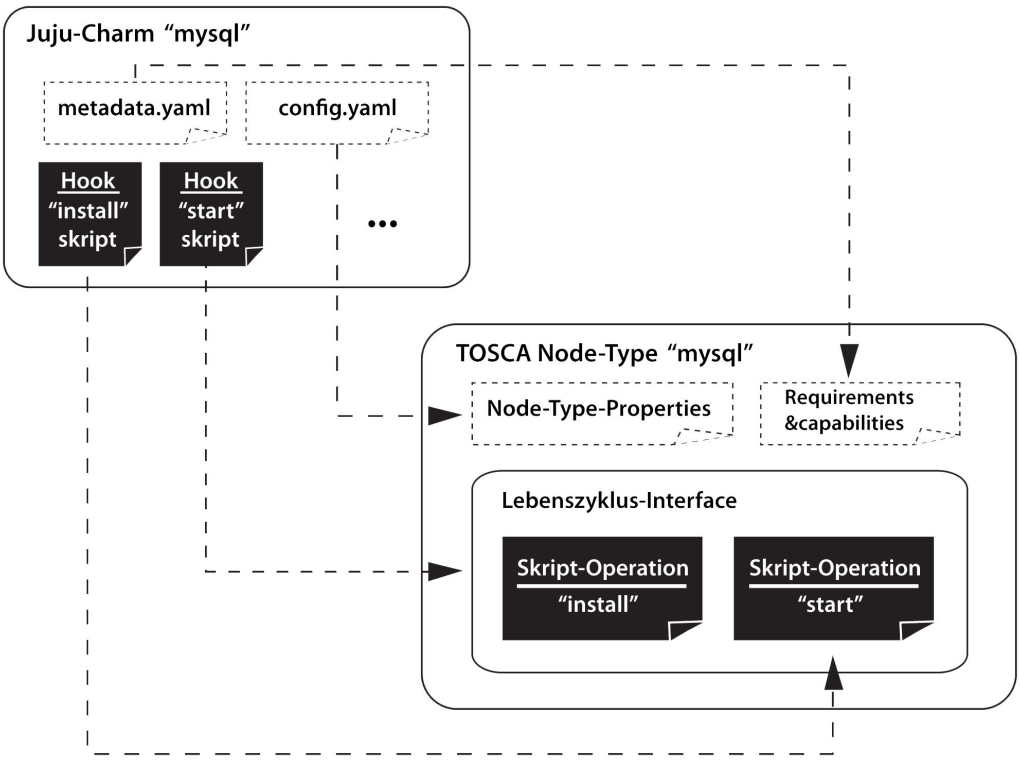

Abbildung 4.2: Beispiel für Erzeung vom TOSCA Node-Type aus Juju-Charm "MySQL"

Der zweite Schritt ist die Generierung eines TOSCA-Definitions-Dokuments für einen Node-Type. Zuerst können durch die Informationen über die Abschnitte "requires" und "provides" in der Datei "metadata.yaml" vom Charm die entsprechenden Elemente *RequirementType* und *CapabilityType* für das TOSCA-Definitions-Dokument generiert werden. Außerdem können die entsprechenden Elemente *ArtifactType* und *ArtifactTemplate* für die Interface-Operationen für den Lebenszyklus eines Cloud-Service erzeugt werden, wenn in dem Verzeichnis "hooks" die Dateien wie z.B. "install", "start" und "stop" vorhanden sind. Hier wird ein Standard-Artifact-Type "Script-Artifact" definiert. Die Artefakte des Typs "Script-Artifact" können von einer beliebigen TOSCA-Laufzeitumgebung verarbeitet werden. Um die tatsächlichen Hooks aufzurufen, werden die zusätzlichen Wrapper-Skripte als die tatsächlichen Artifact-Templates vom Typ "Script-Artifact" erstellt. Der Grund dafür ist, dass nicht bekannt ist, in welcher Skriptsprache diese Hook-Dateien implementiert wurden. Bei TOSCA muss man definieren, um welche Art von Skript (z.B. Shell oder Python) es sich handelt. Deswegen braucht man die zusätzliche Datei "install.sh" als ein Wrapper-Skript, um die Hook-Datei "install" aufzurufen. Dasselbe gilt auch für "start" und "stop" [20]. Schließlich werden durch die vorher erzeugten Elemente die entsprechenden Elemente *NodeType* und *NodeTypeImplementation* generiert. Das Element *NodeType* bezüglich Juju besteht aus den folgenden Elementen: *PropertiesDefinition*, *RequirementDefinition*, *CapabilityDefinition* und *Interfaces*. Durch das Element *PropertiesDefinition* kann ein XML-Element angegeben werden, welches die Struktur der Node-Type-Properties definiert. Die Elemente *RequirementDefinition* und *CapabilityDefinition* können durch die entsprechenden Elemente *RequirementType* und *CapabilityType* generiert werden. Es gibt bei Juju die Lebenszyklus-

Operationen wie z.B. "install", "start" und "stop". Um das Deployment eines aus Juju generierten CSAR zu ermöglichen, muss jede Lebenszyklus-Operation im Element *Interface* im Node-Type definiert werden. Solche Operationen werden verwendet, um in der TOSCA-Umgebung den Lebenszyklus eines Service oder einer Anwendung zu verwalten. Eine Node-Type-Implementation implementiert einen entsprechenden Node-Type. Das Element *NodeTypeImplementation* kann durch das Element *ImplementationArtifact* die Verweise auf die konkreten Artifact-Templates für die Lebenszyklus-Operationen im Node-Type definieren. Außerdem kann das Element *RequiredContainerFeatures* den Hinweis für den TOSCA-Container spezifizieren, dass die Hooks aus Juju als die konkreten Artifact-Templates nur unter Ubuntu-Linux ausgeführt werden können.

Ergänzung zum zweiten Schritt: Möglicherweise könnte es den Abschnitt "peers" in der Datei "metadata.yaml" gegeben. Dieser Abschnitt definiert in Juju eine Art Relation "Peer-Relation". Da die Peer-Relationen zwischen den Service-Einheiten innerhalb eines Service automatisch hergestellt werden, kann für die Peer-Relation ein Peers-Interface im entsprechenden Node-Type generiert werden. Das heißt, dass aus dem Abschnitt "peers" ein Interface namens "peers" generiert werden kann. Das Interface "peers" wird wie das Lebenszyklus-Interface als ein Element im Element *NodeType* definiert. Die Elemente, auf die sich das Inferface "peers" bezieht, sind die Elemente *ArtifactType*, *ArtifactTemplate*, *Interface* (im Element *NodeType*) und *ImplementationArtifact* (im Element *NodeTypeImplementation*). Das Konzept für die Generierung vom Peers-Interface ist genauso wie das Konzept für die Generierung vom Lebenszyklus-Interface, das oben besprochen wurde.

Als letzter Schritt muss eine entsprechende CSAR-Datei generiert werden. Diese Datei enthält die vorher generierten Node-Type-Properties-Dokument und TOSCA-Definitions-Dokument für den Node-Type sowie die TOSCA-Metadatei "TOSCA.meta". Außerdem muss sie alle originale Dateien (alle Hooks und die anderen Dateien) in der Charm-Datei enthalten. Diese Dateien werden noch als die entsprechenden Artefakte verwendet, um in der TOSCA-Umgebung den Lebenszyklus eines Service oder einer Anwendung zu verwalten. Wo sich diese Dateien in der CSAR-Datei befinden sollen, kann der Ersteller der CSAR-Datei selbst entscheiden. In unserem Fall werden sie alle in dem Verzeichnis "Files" gespeichert.

4.3 Erzeugung von TOSCA Relationship-Types aus bestehenden Juju-Artefakten

Dieses Unterkapitel beschäftigt sich mit dem Konzept für die Generierung von TOSCA Relationship-Types aus Juju-Artefakten. Aus dem Grundlagenkapitel wissen wir, dass es in Juju zwei Arten von Beziehungen gibt: Peer-Relation und Require/Provide-Relation. Da sich die Peer-Relation nur auf ein einzelnes Charm bezieht, kann für die Peer-Relation ein Interface namens "peers" im entsprechenden TOSCA Node-Type generiert werden. Das konkrete Konzept dafür wurde in Unterkapitel 4.2 besprochen. Im Folgenden wird beschrieben, wie ein TOSCA Relationship-Type aus der Beziehung zwischen zwei Juju-Charms generiert wird. Diese Beziehung wird in Juju implizit definiert und als "Require/Provide-Relation" bezeichnet. Das heißt, wie die Abbildung der Elemente in zwei Juju-Charms mit einer bestimmten Beziehung zu den Elementen in einem TOSCA-Definitions-Dokument für einen Relationship-Type realisiert wird. Die Idee ist, die äquivalenten Elemente zwischen ihnen zu finden. Zuerst wird ein entsprechendes TOSCA-

Definitions-Dokument für einen Relationship-Type generiert. Zu diesem Dokument gehören die Definitionen der Elemente *ArtifactType*, *ArtifactTemplate*, *RelationshipType* und *RelationshipTypeImplementation*. Dann wird die entsprechende CSAR-Datei erzeugt, die alle notwendigen Dokumente und Artefakte enthält.

4.3.1 Erzeugung von Artifact-Types und Artifact-Templates

Artifact-Types sind wiederverwendbare Entitäten und definieren Typen von Artifact-Templates. Im Fall von Juju wird hier das Artifact-Template als Implementation-Artifact für die Interface-Operationen des Relationship-Type referenziert. Die tatsächlichen Artefakte, auf die das Artifact-Template verweist, sind die sogenannten Relation-Hooks. Die in Hooks definierten Skripte verwenden eine Reihe von Befehlen und Umgebungsvariablen, die nur auf der von Juju verwalteten virtuellen Maschine verfügbar sind. Diese Einschränkung kann durch das Erzeugen von Wrapper-Skripten ausgeglichen werden. Diese Wrapper-Skripte bereiten die Ausführungsumgebung vor und rufen dann die tatsächlichen Skripte in Juju-Charms auf. Folglich muss die TOSCA-Laufzeitumgebung die Implementation-Artifacts der spezifischen Artifact-Types wie die Juju-Artefakte nicht verstehen. Das heißt, dass die Laufzeit-Implementierung von TOSCA nichts über Juju wissen muss: z.B. in welcher Skriptsprache diese Hook-Dateien implementiert wurden. Dieser Ansatz wird als transparente Integration bezeichnet und in Unterkapitel 4.1.4.2 ausführlich erläutert.

Zuerst wird ein Standard-Artifact-Type "Script Artifact" definiert. Die Artefakte des Typs "Script Artifact" können von einer beliebigen TOSCA-Laufzeitumgebung verarbeitet werden. Dann werden die zusätzlichen Wrapper-Skripte erstellt, um die tatsächlichen Relation-Hooks aufzurufen. Ein Beispiel ist, dass sich die Anwendung "WordPress" mit dem Datenbankserver "MySQL" verbindet. Aus dem Abschnitt "requires" in der Datei "metadata.yaml" im Charm "WordPress" erfährt man, dass die Anwendung "WordPress" eine Verbindung mit dem Datenbankserver "MySQL" benötigt. Der Beziehungsname ist "db" und der Interface-Name ist "mysql". Der Beziehungsname wird verwendet, um die entsprechenden Hooks zu finden: "db-relation-changed", "db-relation-departed" und "db-relation-broken". Ebenso erfährt man aus dem Abschnitt "provides" in der Datei "metadata.yaml" im Charm "MySQL", dass dieses Charm eine Verbindung mit dem Datenbankserver "MySQL" zur Verfügung stellt. Der Beziehungsname ist "db" und der Interface-Name ist "mysql". Der Beziehungsname wird verwendet, um die entsprechenden Hooks zu finden: "db-relation-joined" und "db-relation-broken". Der Interface "mysql" bezeugt, dass eine Beziehung zwischen des Charm "WordPress" und des Charm "MySQL" entstehen kann. Wenn "WordPress" mit "MySQL" verbunden ist, werden die Hooks "db-relation-changed", "db-relation-departed" und "db-relation-broken" auf der WordPress-Seite aufgerufen. Entsprechende Hooks "db-relation-joined" und "db-relation-broken" werden auf der MySQL-Seite aufgerufen. Als Ergebnis wird für jedes Relation-Hook eine entsprechende Wrapper-Datei erstellt. Diese Wrapper-Dateien können unter Verwendung der Implementation-Artifacts des Typs "Script Artifact" in das TOSCA Topologie-Modell eingebettet werden.

Der XML-Ausschnitt 4.5 zeigt ein Beispiel für ein Artifact-Template des Typs "Script Artifact" bezüglich eines Wrappers, der das tatsächliche Relation-Hook "db-relation-joined" im Charm "MySQL" aufruft.

```
01  <ArtifactTemplate id="cc449a07-8218-4b37-9e01-3e833d5bd09b"
02      xmlns:artifacts="http://docs.oasis-open.org/
03                       tosca/ns/2011/12/Artifacts"
02      type="artifacts:ScriptArtifact">
03    <Properties>
04      <artifacts:ScriptArtifactProperties
05          xmlns:artifacts="http://docs.oasis-open.org/
06                           tosca/ns/2011/12/Artifacts>
07        <ScriptLanguage>sh</ScriptLanguage>
08        <PrimaryScript>
09          Files/charm_mysql/tosca_scripts/db-relation-joined.sh
10        </PrimaryScript>
11      </artifacts:ScriptArtifactProperties>
12    </Properties>
13    <ArtifactReferences>
14     <ArtifactReference reference="Files/charm_mysql/tosca_scripts/">
15        <Include pattern="db-relation-joined.sh"/>
16      </ArtifactReference>
17    </ArtifactReferences>
18  </ArtifactTemplate>
```

Ausschnitt 4.5: XML-Syntax für das Artifact-Template vom Standard-Artifact-Type

Der Inhalt des Elements *ScriptArtifactProperties* beschränkt sich auf einige generische Metadaten in Bezug auf das entsprechende Skript. Das Element *ScriptArtifactProperties* wird in der XML-Schema-Definition [64] (in der Datei "ScriptArtifact.xsd") für den Standard-Artifact-Type definiert und verwendet, um die Struktur der Eigenschaften des Standard-Artifact-Type zu definieren. Die Datei "ScriptArtifact.xsd" wird im Anhang 3 präsentiert. Alle Juju-bezogenen Informationen werden im Wrapper-Skript "db-relation-joined.sh" versteckt. Folglich sind die Juju-Spezifika des Artifact-Template vollständig transparent für die TOSCA-Laufzeitumgebung. Das heißt, dass die Laufzeit-Implementierung nichts über Juju wissen muss.

4.3.2 Erzeugung von Relationship-Type und Relationship-Type-Implementation

Ein Relationship-Type ist eine wiederverwendbare Entität, die den Typ eines Relationship-Template oder von mehreren Relationship-Templates definiert. Das Element *RelationshipType* bezüglich zwei Juju-Charms besteht aus den folgenden Elementen: *SourceInterfaces*, *TargetInterfaces*, *ValidSource* und *ValidTarget*. Die Operationen, die auf (einer Instanz von) einem Relationship-Template durchgeführt werden können, werden durch die Interfaces des Relationship-Type definiert. Es gibt zwei Arten Interfaces für den Relationship-Type: Source-Interfaces und Target-Interfaces. Das Element *SourceInterfaces* enthält die Definitionen der Interfaces. Diese Interfaces können auf der Quelle einer Beziehung durchgeführt werden. Ebenso werden die Interfaces, die im Element *TargetInterfaces* definiert werden, auf dem Ziel der Beziehung durchgeführt werden. Die Definitionen dieser Interfaces werden in Form von verschachtelten Elementen *Interface* angegeben. Der Inhalt des Elements *Interface* ist ähnlich wie der Inhalt des Elements

Interface im Node-Type (Unterkapitel 4.1.4). Das Element *ValidSource* spezifiziert den Typ des Objektes, das als Quelle für Beziehungen zulässig ist. Ebenso spezifiziert das Element *ValidTarget* den Typ des Objektes, das als Ziel für Beziehungen zulässig ist. Der Typ der Quelle und der Typ des Zieles müssen miteinander übereinstimmen. Das heißt, dass der Typ des Zieles ein Node-Type sein muss, wenn der Typ der Quelle ein Node-Type ist. Anderenfalls muss der Typ des Zieles ein Capability-Type sein, wenn der Typ der Quelle ein Requirement-Type ist. Außerdem muss dieser Capability-Type mit dem Capability-Type, der im Attribut *requiredCapabilityType* des entsprechenden Elements *RequirementType* spezifiziert wird, übereinstimmen.

Der XML-Ausschnitt 4.6 zeigt ein Beispiel für den Relationship-Type "ConnectsTo" bezüglich zwei Services "WordPress " und "MySQL" und entspricht dem Beispiel, das im letzten Unterkapitel dargestellt wurde.

```
01  <RelationshipType name="ConnectsTo"
02      targetNamespace="http://jujucharms.com/charms/relations">
03    <SourceInterfaces>
04      <Interface name="http://jujucharms.com/charms/wordpress">
05        <Operation name="db-relation-changed"/>
06        <Operation name="db-relation-departed"/>
07        <Operation name="db-relation-broken"/>
08      </Interface>
09    </SourceInterfaces>
10    <TargetInterfaces>
11      <Interface name="http://jujucharms.com/charms/mysql">
12        <Operation name="db-relation-joined"/>
13        <Operation name="db-relation-broken"/>
14      </Interface>
15    </TargetInterfaces>
16    <ValidSource xmlns:charm="http://jujucharms.com/charms"
17        typeRef="charm:wordpress"/>
18    <ValidTarget xmlns:charm="http://jujucharms.com/charms"
19        typeRef="charm:mysql"/>
20  </RelationshipType>
```

Ausschnitt 4.6: XML-Syntax für den Relationship-Type

Der Relationship-Type definiert den Typ der Beziehung "ConnectsTo". Das Element *ValidSource* spezifiziert durch das Attribut *typeRef*, dass der Typ der Quelle dieser Beziehung der Node-Type "WordPress" ist. Ebenso spezifiziert das Element *ValidTarget*, dass der Typ des Zieles dieser Beziehung der Node-Type "MySQL" ist. Das Element *Interface* im Element *SourceInterfaces* definiert die Operationen "db-relation-changed", "db-relation-departed" und "db-relation-broken", die auf der Quelle "WordPress" dieser Beziehung aufgerufen werden, um diese Beziehung zu verwalten und zu beenden. Ebenso definiert das Element *Interface* im Element *TargetInterfaces* die Operationen "db-relation-joined" und "db-relation-broken", die auf dem Ziel "MySQL" dieser Beziehung aufgerufen werden, um diese Beziehung herzustellen und zu beenden.

Eine Relationship-Type-Implementation beschreibt den ausführbaren Code, der einen spezifischen Relationship-Type implementiert. Die Relationship-Type-Implementation stellt auch eine Sammlung von ausführbaren Dateien oder Programmen zur Verfügung, welche die Interface-Operationen eines Relationship-Type implementieren. Diese ausführbaren Dateien oder Programme werden als separate Artifact-Templates definiert und von den Implementation-Artifacts einer Relationship-Type-Implementation referenziert. Im Fall von aus Juju-Charms generierten Relationship-Types wird ein entsprechendes Implementation-Artifact für jede Interface-Operation durch das Element *ImplementationArtifact* definiert. Beispielsweise werden im Relationship-Type "MySQL" die Operation "db-relation-joined" (an einer Verbindung mit dem Datenbankserver teilzunehmen) und die Operation "db-relation-broken" (eine Verbindung mit dem Datenbankserver zu verlassen) definiert.

Der XML-Ausschnitt 4.7 zeigt ein Beispiel für eine Relationship-Type-Implementation, die das Artifact-Template (dargestellt in Ausschnitt 4.5) vom Typ "Script Artifact" mit der Operation "db-relation-joined" vom Target-Interface im aus den Juju-Charms "WordPress" und "MySQL" generierten Relationship-Type verknüpfen kann.

```
01  <RelationshipTypeImplementation
02      xmlns:relation="http://jujucharms.com/charms/relations"
03      name="ConnectsTo_relationshiptypeimplementation"
04      targetNamespace="http://jujucharms.com/charms/relations"
05      relationshipType="relation:ConnectsTo_relationshiptype">
06    <RequiredContainerFeatures>
07      <RequiredContainerFeature
08          feature="http://jujucharms.com/platform/ubuntu"/>
09    </RequiredContainerFeatures>
10    <ImplementationArtifacts>
11      ...
12      <ImplementationArtifact name="mysql-db-relation-joined"
13          xmlns:artifacts="http://docs.oasis-open.org/
14                           tosca/ns/2011/12/Artifacts"
15          xmlns:ConnectsTo="http://jujucharms.com/charms/
16                            precise/ConnectsTo/relationshiptype"
17          interfaceName="http://jujucharms.com/charms/mysql"
18          operationName="db-relation-joined"
19          artifactType="artifacts:ScriptArtifact"
20          artifactRef="ConnectsTo:cc449a07-...-9e01-3e833d5bd09b"/>
21      ...
22  </RelationshipTypeImplementation>
```

Ausschnitt 4.7: XML-Syntax für die Relationship-Type-Implementation

Das Element *ImplementationArtifact* hat die folgenden Attribute: *name*, *artifactType*, *artifactRef*, *interfaceName* und *operationName*. Das Attribut *name* spezifiziert den Namen des Artefakts. Das Attribut *artifactType* spezifiziert den Typ des Artefakts. Das optionale Attribut *artifactRef* enthält einen Namen, der ein Artifact-Template als ein Implementation-Artifact identifiziert. Der Wert des Attributs *artifactRef* entspricht dem Wert des Attributs *id* im Artifact-Template (dargestellt in Ausschnitt 4.5). Das optionale Attribut *interfaceName*

spezifiziert den Namen des Interface und entspricht dem Interface-Namen im entsprechenden Relationship-Type. Das optionale Attribut *operationName* spezifiziert den Namen der Operation und referenziert die Operation "db-relation-joined" für das Target-Interface im entsprechenden Relationship-Type. Außerdem kann das Element *RequiredContainerFeatures* den Hinweis für einen TOSCA-Container spezifizieren, dass die Hooks aus Juju als die konkreten Artefakte nur unter Ubuntu-Linux ausgeführt werden können.

4.3.3 Erzeugung der entsprechenden CSAR-Datei

Schließlich muss eine TOSCA CSAR-Datei generiert werden. In unserem Fall enthält diese CSAR-Datei alle erforderlichen Elemente für die Beziehung zwischen zwei Services oder Anwendungen. Zu solchen Elementen gehören das TOSCA-Definitions-Dokument für den Relationship-Type, die TOSCA-Metadatei "TOSCA.meta" sowie alle erforderlichen Artefakte. Diese Artefakte sind die originalen Dateien (alle Hooks und die anderen Dateien) in den Charm-Dateien. Sie werden noch als die entsprechenden Artefakte aufgerufen werden, um in der TOSCA-Umgebung den Lebenszyklus einer Beziehung zwischen zwei Services oder Anwendungen zu verwalten. Wo sich diese Dateien in der CSAR-Datei befinden sollen, kann der Ersteller der CSAR-Datei selbst entscheiden. In unserem Fall werden sie alle in dem Verzeichnis "Files" gespeichert.

4.4 Erzeugung von TOSCA Service-Templates

Dieses Unterkapitel beschäftigt sich mit dem Konzept für die Generierung von TOSCA Service-Templates durch Orchestrierung der Node-Types und Relationship-Types, die auf bestehenden Chef- oder Juju-Artefakten basieren. Um genauer auszudrücken, werden diese Node-Types und Relationship-Types aus Chef-Cookbooks und Juju-Charms mittels der in Unterkapitel 4.1, 4.2 und 4.3 dargestellten Konzepte generiert. Zuerst wird ein entsprechendes TOSCA-Definitions-Dokument für ein Service-Template generiert. Ein Service-Template besteht in der Regel aus zwei wichtigen Teilen: Topology-Template und Pläne. Da sich diese Arbeit nicht auf die Pläne konzentriert, wird nur das Konzept für die Erzeugung des Element *TopologyTemplate* besprochen. Dieses Dokument enthält eine Reihe von Definitionen für die Elemente *Import* und die Definition des Elements *TopologyTemplate*. Das Element *TopologyTemplate* besteht aus einer Reihe der Elemente *NodeTemplate* und *RelationshipTemplate*. Schließlich wird die entsprechende CSAR-Datei für das Service-Template erzeugt, die alle notwendigen Dokumente und Artefakte enthält.

4.4.1 Erzeugung der Elemente *Import*

Durch das Element *Import* können die externen Dokumente für die TOSCA-Definitionen, die XML-Schema-Definitionen oder die WSDL-Definitionen ins Service-Template importiert werden. Da die bestehenden Node-Types und Relationship-Types verwendet werden, um das entsprechende Service-Template zu erzeugen, werden die TOSCA-Definitions-Dokumente für die entsprechenden Node-Types und Relationship-Types importiert. Durch jeden importierten Node-Type kann ein entsprechendes Node-Template im Service-Template generiert werden. Dasselbe gilt auch für die Generierung von Relationship-Templates.

Im Folgenden wird beschrieben, wie ein TOSCA Topology-Template generiert wird. Das Element *TopologyTemplate* spezifiziert die gesamte Struktur der Cloud-Anwendung, die durch das Service-Template definiert wird. Es werden die Komponenten, aus denen diese Cloud-Anwendung besteht, und die Beziehungen zwischen diesen Komponenten definiert.

Die Komponenten eines Service werden als Node-Templates bezeichnet und die Beziehungen zwischen diesen Komponenten werden als Relationship-Templates bezeichnet. Als Nächstes wird ausführlich beschrieben, wie die Node-Templates und die Relationship-Templates erzeugt werden.

4.4.2 Erzeugung von Node-Templates

Ein entsprechendes Node-Template wird durch einen importierten Node-Type generiert. Das Element *NodeTemplate* spezifiziert eine bestimmte Komponente, die als ein Bestandteil in der Cloud-Anwendung verwendet wird. Sein Attribut *type* verweist auf den Node-Type, der den Typ des Node-Template spezifiziert. Das Element *NodeTemplate* besteht aus den folgenden Elementen: *Properties*, *Requirements* und *Capabilities*.

Der XML-Ausschnitt 4.8 zeigt ein Beispiel für ein Node-Template vom Node-Type "WordPress", der aus dem Charm "WordPress" generiert wurde.

```
01  <NodeTemplate name="wordpress nodeTemplate"
02      id="wordpress_nodeTemplate"
03      xmlns:nodeType="http://jujucharms.com/charms"
04      type="nodeType:wordpress_nodetype">
05    <Properties>
06      <wordpress-properties>
07        <tuning>single</tuning>
08        <debug>no</debug>
09        <engine>nginx</engine>
10      </wordpress-properties>
11    </Properties>
12    <Requirements>
13      <Requirement name="db" id="wordpress_db"
14          xmlns:requirementType="http://jujucharms.com/interfaces"
15          type="requirementType:mysql"/>
16      <Requirement name="nfs" id="wordpress_nfs"
17          xmlns:requirementType="http://jujucharms.com/interfaces"
18          type="requirementType:mount"/>
19      <Requirement name="cache" id="wordpress_cache"
20          xmlns:requirementType="http://jujucharms.com/interfaces"
21          type="requirementType:memcache"/>
22    </Requirements>
23    <Capabilities>
24      <Capability name="website" id="wordpress_website"
25          xmlns:capabilityType="http://jujucharms.com/interfaces"
26          type="capabilityType:http"/>
27    </Capabilities>
28  </NodeTemplate>
```

Ausschnitt 4.8: XML-Syntax für das Node-Template

Das Element *Properties* spezifiziert die Ausgangswerte ("Initial Values") für die Eigenschaften des Node-Type (die Node-Type-Properties). Die Ausgangswerte werden durch

ein Instanz-Dokument des XML-Schemas der entsprechenden Node-Type-Properties spezifiziert. Nicht allen Eigenschaften des Node-Type können die Ausgangswerte zugewiesen werden. Das heißt, dass einige Elemente oder Attribute in der Instanz, die das Element *Properties* zur Verfügung stellt, möglicherweise verloren gehen. Sobald das definierte Node-Template instanziiert wurde, muss sich die XML-Darstellung der Eigenschaften des Node-Type durch die assoziierte XML-Schema-Definition validieren lassen. Bei der Generierung von Node-Types aus Chef- oder Juju-Artefakten wird auch das entsprechende Node-Type-Properties-Dokument erzeugt. Dieses Dokument ist eine XML-Datei mit der Dateiendung ".xsd" und speichert die XML-Schema-Definition. Dieses XML-Schema definiert die Struktur von XML-Dokumenten (XML-Instanzen) und ermöglicht eine Validierung. Im Beispiel wird die XML-Instanz (das Element *wordpress-properties*) als Inhalt des Elements *Properties* verwendet. Diese XML-Instanz wurde durch das Node-Type-Properties-Dokument erzeugt, das durch die Datei "config.yaml" im Charm "WordPress" generiert wurde.

Das Element *Requirements* enthält eine Liste der Anforderungen für das Node-Template. Diese Liste entspricht der Liste der Anforderungsdefinitionen des Node-Type, der im Attribut *type* des Node-Template spezifiziert wird. Das heißt, dass eine Anforderung für das Node-Template durch eine entsprechende Anforderungsdefinition des Node-Type erzeugt werden kann. Jede Anforderung wird in einem separaten verschachtelten Element *Requirement* spezifiziert.

Das Element *Capabilities* enthält eine Liste der Fähigkeiten für das Node-Template. Diese Liste entspricht der Liste der Fähigkeitsdefinitionen des Node-Type, der im Attribut *type* des Node-Template spezifiziert wird. Das heißt, dass eine Fähigkeit für das Node-Template durch eine entsprechende Fähigkeitsdefinition des Node-Type erzeugt werden kann. Jede Fähigkeit wird in einem separaten verschachtelten Element *Capability* spezifiziert.

4.4.3 Erzeugung von Relationship-Templates

Ein entsprechendes Relationship-Template wird durch einen importierten Relationship-Type generiert. Das Element *RelationshipTemplate* spezifiziert eine bestimmte Beziehung zwischen den Komponenten der Cloud-Anwendung. Sein Attribut *type* verweist auf den Relationship-Type, der den Typ des Relationship-Template definiert. Das Element *RelationshipTemplate* besteht aus den folgenden Elementen: *SourceElement* und *TargetElement*.

Der XML-Ausschnitt 4.9 zeigt ein Beispiel für ein Relationship-Template vom Relationship-Type "ConnectsTo", der aus zwei Charms "WordPress" und "MySQL" generiert wurde.

```
01  <RelationshipTemplate name="ConnectsTo relationshipTemplate"
02      id="ConnectsTo_relationshipTemplate"
03      xmlns:relationshipType="http://jujucharms.com/charms/relations"
04      type="relationshipType:ConnectsTo_relationshiptype">
05    <SourceElement ref="wordpress_nodeTemplate"/>
06    <TargetElement ref="mysql_nodeTemplate"/>
07  </RelationshipTemplate>
```

Ausschnitt 4.9: XML-Syntax für das Relationship-Template

Das Element *SourceElement* spezifiziert die Quelle der Beziehung, die durch das aktuelle Relationship-Template dargestellt wird. Ebenso spezifiziert das Element *TargetElement* das Ziel dieser Beziehung. Im Fall, dass ein Node-Type als gültige Quelle im Relationship-Type definiert wird, muss ein Node-Template des entsprechenden Node-Type im Element *SourceElement* referenziert werden. Dasselbe gilt auch für das Element *TargetElement*. Im Beispiel referenziert das Element *SourceElement* das Node-Template "WordPress", während das Element *TargetElement* das Node-Template "MySQL" referenziert.

4.4.4 Erzeugung der entsprechenden CSAR-Datei

Als letzter Schritt wird eine TOSCA CSAR-Datei für das Service-Template generiert. Diese CSAR-Datei enthält nicht nur das generierte TOSCA-Definitions-Dokument für das Service-Template sondern auch alle importierten TOSCA-Definitions-Dokumente für Node-Types und Relationship-Types sowie die Node-Type-Properties-Dokumente für die entsprechenden Node-Types. Außerdem beinhaltet sie die TOSCA-Metadatei "TOSCA.meta" und alle erforderlichen Artefakte wie z.B. Chef-Cookboks, Juju-Charms und andere Ressourcen. Diese Dateien werden in unserem Fall in dem Verzeichnis "Files" gespeichert und als tatsächliche Artefakte verwendet, um die entsprechenden Services oder Anwendungen und die Beziehungen zwischen ihnen zu verwalten.

5 Entwurf und Implementierung

In Kapitel 4 wurden die Konzepte für das automatische Verfahren zur Erzeugung von TOSCA Service-Templates aus den Chef- und Juju-Artefakten dargestellt. Um diese Konzepte zu evaluieren, wird ein Prototyp entwickelt, mit dem sich alle öffentlich zugänglichen Chef- und Juju-Artefakte in CSARs verwenden lassen können. In diesem Kapitel werden zuerst die Analyse und der Entwurf des Prototyps besprochen. Darauf folgt die Beschreibung der Implementierung des Prototyps.

5.1 Anforderungsanalyse

Vor dem Entwurf des Prototyps müssen die Anforderungen an den Prototyp analysiert werden. In diesem Unterkapitel wird zunächst auf die funktionalen Anforderungen an den Prototyp eingegangen. Darauf folgt eine Beschreibung der nicht-funktionalen Anforderungen. Schließlich werden zusätzliche Funktionalitäten des Prototyps dargestellt.

5.1.1 Funktionale Anforderungen

Die funktionalen Anforderungen an den Prototyp werden in diesem Unterkapitel erläutert. Der Prototyp soll folgende Anforderungen erfüllen:

Eine TOSCA CSAR-Datei für den Node-Type soll durch eine URL als Eingabe des Prototyps erzeugt werden. Die URL lokalisiert eine Ressource wie z. B. ein Chef-oder ein Juju-Artefakt. Neben dem TOSCA-Definitions-Dokument für den Node-Type und dem entsprechenden Node-Type-Properties-Dokument soll diese generierte CSAR-Datei noch alle entsprechenden Chef- oder Juju-Artefakte enthalten. Diese Arbeit konzentriert sich nur auf die Erzeugung von TOSCA Node-Types aus Chef-Artefakten. Die Erzeugung von TOSCA Node-Types aus Juju-Artefakten wurde in der Studienarbeit "Vorlagen für das Deployment von Services und Applikationen in der Cloud" [33] dargestellt.

Eine TOSCA CSAR-Datei für den Relationship-Type soll durch zwei URLs als Eingabe des Prototyps erzeugt werden. Jede URL lokalisiert in diesem Fall ein Juju-Artefakt. Außerdem muss garantiert werden, dass eine implizite Beziehung zwischen diesen zwei Juju-Artefakten bestehen soll. Neben dem TOSCA-Definitions-Dokument für den Relationship-Type soll diese CSAR-Datei noch alle entsprechenden Juju-Artefakte enthalten.

Eine TOSCA CSAR-Datei für das Service-Template soll durch eine Reihe von Node-Types und Relationship-Types als Eingabe des Prototyps erzeugt werden. Diese Node-Types und Relationship-Types sollen durch den entwickelten Prototyp aus bestehenden Chef- oder Juju-Artefakten erzeugt werden. Neben den TOSCA-Definitions-Dokumenten für die Node-Types, die Relationship-Types und das Service-Template sowie den Node-Type-Properties-Dokumenten soll diese CSAR-Datei noch alle entsprechenden Chef- oder Juju-Artefakte enthalten.

5.1.2 Nicht-funktionale Anforderungen

Neben den funktionalen Aspekten sollen die folgenden nicht-funktionalen Anforderungen erfüllt werden. Der Prototyp soll in Java implementiert werden und sich als JAR-Datei verpacken lassen, sodass er als Library einfach in anderen Java-Projekten verwendet werden

kann. Die Komponenten des Prototyps sollen lose gekoppelt werden, um die Erweiterung des Prototyps und den Austausch der verschiedenen Teile für neue Implementierungen zu erleichtern.

5.1.3 Zusätzliche Funktionalitäten des Prototyps

Der Prototyp soll außerdem zusätzliche Funktionen erfüllen: Beispielsweise kann ein entsprechendes Artefakt durch eine URL zu einem Repository heruntergeladen werden, das ein Chef- oder Juju-Artefakt enthält. Außerdem sollen auch die Operationen zu einem Ordner zur Verfügung gestellt werden. Dazu gehören das Suchen einer Datei in einem Ordner durch einen gegebenen Namen, das Auflisten der Inhalte in einem Ordner, das Kopieren der Inhalte von einem Ordner zu einem anderen und das Packen eines Ordners zu einer Zipdatei.

5.2 Entwurf des Prototyps

Dieses Unterkapitel geht auf den Entwurf des Prototyps ein. Es wird zunächst die allgemeine Architektur des Prototyps dargestellt. Diese Architektur zeigt wichtige Komponenten, aus denen der Prototyp besteht. Darauf folgt die interne Struktur jeder Komponente. Diese interne Struktur zeigt die notwendigen Teilkomponenten innerhalb jeder Komponente. Schließlich wird beschrieben, in welcher Reihenfolge die Teilkomponenten innerhalb jeder Komponente ausgeführt werden.

5.2.1 Architektur des Prototyps

Dieses Unterkapitel beschreibt die funktionale Architektur des Prototyps. Der zu entwickelnde Prototyp besteht aus drei wichtigen Komponenten. Die Funktion, die von jeder Komponente implementiert wird, entspricht jeder der funktionalen Anforderungen (Unterkapitel 5.1.1). Eine grobe Übersicht über die Komponenten des Prototyps wird in Abbildung 5.1 gezeigt. Dies sind die Komponenten "Node-Type-Generator", "Relationship-Type-Generator" und "Service-Template-Generator". Diese drei Komponenten arbeiten unabhängig voneinander und realisieren jeweils eigene Funktionen.

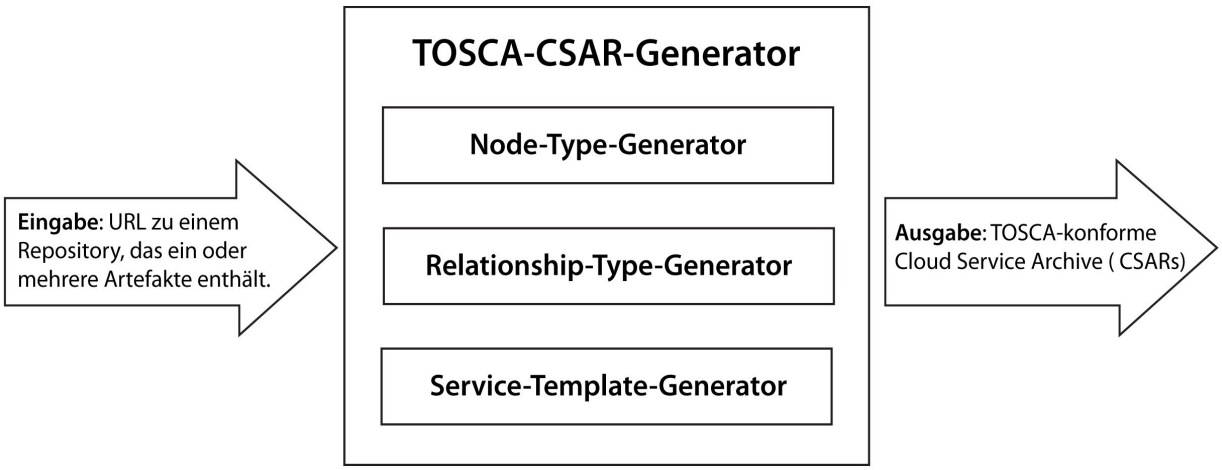

Abbildung 5.1: Eine grobe Übersicht über die Komponenten des Prototyps

Die Aufgabe des "Node-Type-Generator" ist aus einem Chef- oder Juju-Artefakt eine TOSCA CSAR-Datei für den Node-Type zu erzeugen. Zu den Aufgaben dieser Komponente gehören (1) das Herunterladen eines Chef- oder Juju-Artefakts durch eine entsprechende URL, (2) die

Erzeugung des Node-Type-Properties-Dokuments, (3) die Erzeugung des TOSCA-Definitions-Dokuments für den Node-Type und (4) das Packen aller notwendigen Dokumente und Ressourcen (z.B. Cookbooks und Recipes von Chef sowie Charms und Hooks von Juju) zu der generierten CSAR-Datei. Die Eingabe dieser Komponente ist eine URL, die sich mit einem Repository (z.B. GIT [19] und Bazaar [47]) verbindet, das ein Chef-Cookbook oder Juju-Charm enthält. Die Ausgabe dieser Komponente ist eine TOSCA CSAR-Datei für den entsprechenden Node-Type.

Die Aufgabe des "Relationship-Type-Generator" ist aus zwei Juju-Artefakten mit einer impliziten Beziehung eine TOSCA CSAR-Datei für den Relationship-Type zu erzeugen. Zu den Aufgaben dieser Komponente gehören (1) das Herunterladen von Juju-Artefakten durch zwei entsprechende URLs, (2) die Erzeugung des TOSCA-Definitions-Dokuments für den Relationship-Type und (3) das Packen aller notwendigen Dokumente und Ressourcen wie z.B. Charms und Hooks von Juju zu der generierten CSAR-Datei. Die Eingabe dieser Komponente ist zwei URLs. Jede URL verbindet sich mit einem Repository, das ein Juju-Charm enthält. Außerdem soll eine Beziehung zwischen diesen zwei Juju-Charms bestehen. Die Ausgabe dieser Komponente ist eine TOSCA CSAR-Datei für den entsprechenden Relationship-Type.

Die Aufgabe des "Service-Template-Generator" ist aus einer Reihe von TOSCA-Definitions-Dokumenten für Node-Types und Relationship-Types eine TOSCA CSAR-Datei für das Service-Template zu erzeugen. Diese TOSCA-Definitions-Dokumente für Node-Types und Relationship-Types werden von den obengenannten Komponenten generiert und hier als Eingabe verwendet. Die wichtige Aufgabe dieser Komponente ist die Erzeugung des TOSCA-Definitions-Dokuments für das Service-Template. Außerdem müssen alle notwendigen Ressourcen (z.B. Cookbooks und Recipes von Chef sowie Charms und Charms von Juju) und Dokumente in die generierte CSAR-Datei als Zipdatei gepackt werden. Zu den Dokumenten gehören die als Eingabe importierten TOSCA-Definitions-Dokumente für Node-Types und Relationship-Types, das generierte TOSCA-Definitions-Dokument für das Service-Template sowie die entsprechenden Node-Type-Properties-Dokumente. Die Eingabe dieser Komponente ist eine Reihe von TOSCA-Definitions-Dokumenten für Node-Types und Relationship-Types. Die Ausgabe dieser Komponente ist eine TOSCA CSAR-Datei für das entsprechende Service-Template.

5.2.2 Interne Struktur der Komponenten

In Unterkapitel 5.2.1 wurden die allgemeine Architektur des Prototyps und ihre wichtigen Komponenten vorgestellt. In diesem Unterkapitel wird die interne Struktur der einzelnen Komponente beschrieben. Diese Struktur zeigt, welche Teilkomponenten die einzelne Komponente besitzen muss, um ihre Funktionen erledigen zu können. Die Teilkomponenten unterscheiden sich in zwei Arten: Kernkomponenten und Hilfskomponenten.

Abbildung 5.2 zeigt eine grobe Übersicht über die interne Struktur jeder Komponente des Prototyps. Jede Komponente besteht aus zwei Kernkomponenten (Metamodell-Generator und Metamodell-Converter) und anderen Hilfskomponenten (Downloader, FileUtils, XSD-/XML-Generator und ZIP-File-Handler).

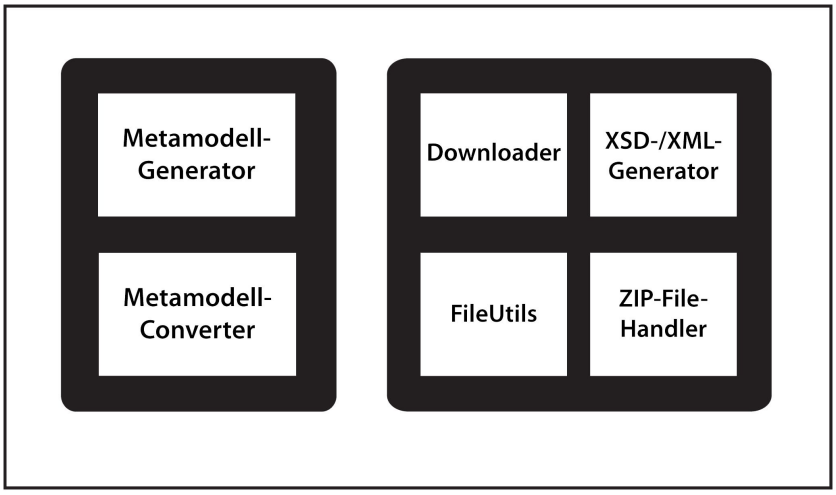

Abbildung 5.2: Die interne Struktur jeder Komponente

Die Kernkomponente "Metamodell-Generator" liest das Chef- oder Juju-Artefakt ein, analysiert es und generiert daraus das entsprechende Metamodell. Unter Metamodell versteht man im Rahmen dieser Arbeit ein abstraktes Objekt, in dem alle Elemente der Artefakte wie z.B. Chef-Cookbooks, Juju-Charms und TOSCA-Definitions-Dokumente auf eine abstrakte Art und Weise gespeichert werden können. Es werden drei Metamodell-Arten definiert: Cookbook-Metamodell für ein Chef-Cookbook, Charm-Metamodell für ein Juju-Charm und TOSCA-Metamodell für ein TOSCA-Definitions-Dokument. Das Cookbook-Metamodell dient zum Speichern von Elementen eines Chef-Cookbook. Dasselbe gilt auch für das Charm-Metamodell und das TOSCA-Metamodell. Diese Kernkomponente besitzt drei Funktionseinheiten: Cookbook-Ruby-Reader, Charm-YAML-Reader und TOSCA-XML-Reader. Der Cookbook-Ruby-Reader bekommt die Metadatei "metadata.rb" im Chef-Cookbook übergeben und liefert als Ausgabe ein entsprechendes Cookbook-Metamodell. Der Charm-YAML-Reader liest die Metadatei "metadata.yaml" im Juju-Charm ein und generiert ein entsprechendes Charm-Metamodell. Der TOSCA-XML-Reader bekommt ein TOSCA-Definitions-Dokument für den Node-Type oder den Relationship-Type übergeben und liefert als Ausgabe ein entsprechendes TOSCA-Metamodell.

Die Kernkomponente "Metamodell-Converter" konvertiert ein Metamodell oder mehrere Metamodelle zu einem anderen Metamodell. Diese Kernkomponente besitzt drei Funktionseinheiten: Cookbook-Metamodell-Converter, Charm-Metamodell-Converter und TOSCA-Metamodell-Converter. Der Cookbook-Metamodell-Converter für den Node-Type-Generator bekommt ein Cookbook-Metamodell übergeben und liefert als Ausgabe ein entsprechendes TOSCA-Metamodell für einen Node-Type. Der Charm-Metamodell-Converter für den Node-Type-Generator liest ein Charm-Metamodell ein und generiert ein entsprechendes TOSCA-Metamodell für einen Node-Type. Im anderen Fall bekommt der Charm-Metamodell-Converter für den Relationship-Type-Generator zwei Charm-Metamodelle übergeben und liefert als Ausgabe ein entsprechendes TOSCA-Metamodell für einen Relationship-Type. Der TOSCA-Metamodell-Converter bekommt eine Reihe von TOSCA-Metamodellen für Node-Types und Relationship-Types übergeben und liefert als Ausgabe ein entsprechendes TOSCA-Metamodell für ein Service-Template. Das TOSCA-Metamodell dient zum Speichern von XML-Elementen eines TOSCA-Definitions-Dokuments.

Neben diesen Kernkomponenten besitzt jede Komponente noch einige notwendigen Hilfskomponenten. Beispielsweise kann die Hilfskomponente "Downloader" ein entsprechendes Artefakt wie z.B. ein Chef-Cookbook oder ein Juju-Charm durch eine angegebene URL zu einem Repository herunterladen, das dieses Artefakt enthält. Die Komponente "Service-Template-Generator" enthält jedoch diese Hilfskomponente nicht, weil sie nicht die Chef- oder Juju-Artefakte sondern die TOSCA-Definitions-Dokumente für Node-Types und Relationship-Types verarbeitet. Die Funktion der Hilfskomponente "XSD-/XML-File-Generator" ist eine XSD- oder XML-Datei zu generieren. In unserem Fall kann der XML-File-Generator durch ein TOSCA-Metamodell ein entsprechendes TOSCA-Definitions-Dokument erzeugen. Wenn ein TOSCA-Metamodell die Elemente für die Node-Type-Properties enthält, kann der XSD-File-Generator durch dieses TOSCA-Metamodell ein entsprechendes XML-Schema-Definitions-Dokument generieren. Außerdem stellt die Hilfskomponente "FileUtils" die Operationen zu einem Ordner zur Verfügung. Dazu gehören das Suchen einer Datei in einem Ordner durch einen gegebenen Datei-Namen, das Auflisten der Inhalte in einem Ordner und das Kopieren der Inhalte von einem Ordner zu einem anderen. Und die Hilfskomponente "ZIP-File-Handler" kann einen Ordner zu einer Zipdatei packen.

5.2.3 Die Funktionsweise der Komponenten

In diesem Unterkapitel wird die Funktionsweise jeder Komponente des Prototyps beschrieben. Die Teilkomponenten jeder Komponente werden in sequenzieller Reihenfolge ausgeführt. Außerdem wird die Ausgabe einer Teilkomponente als Eingabe der nächsten Teilkomponente verwendet. Zur Übersichtlichkeit werden einige Schritte in den folgenden Abbildungen weggelassen. Beispielsweise muss eine TOSCA-Metadatei "TOSCA.meta" für jede CSAR-Datei generiert werden. Darüber hinaus müssen alle entsprechenden Dokumente und Ressourcen (z.B. Chef-Cookbooks und Juju-Charms) zur CSAR-Datei gezippt werden. In den folgenden Abbildungen repräsentiert der Kreis die Prozedur, die ausgeführt wird, die Rechtecke vor und nach dem Kreis die Ein- und Ausgabe.

5.2.3.1 Die Funktionsweise der Komponente "Node-Type-Generator"

In Abbildung 5.3 wird ein vereinfachtes Ablaufdiagramm gezeigt. Das Ablaufdiagramm stellt die Funktionsweise der Teilkomponenten innerhalb der Komponente "Node-Type-Generator" für Chef dar. Die Funktionsweise wird in 4 Schritten beschrieben.

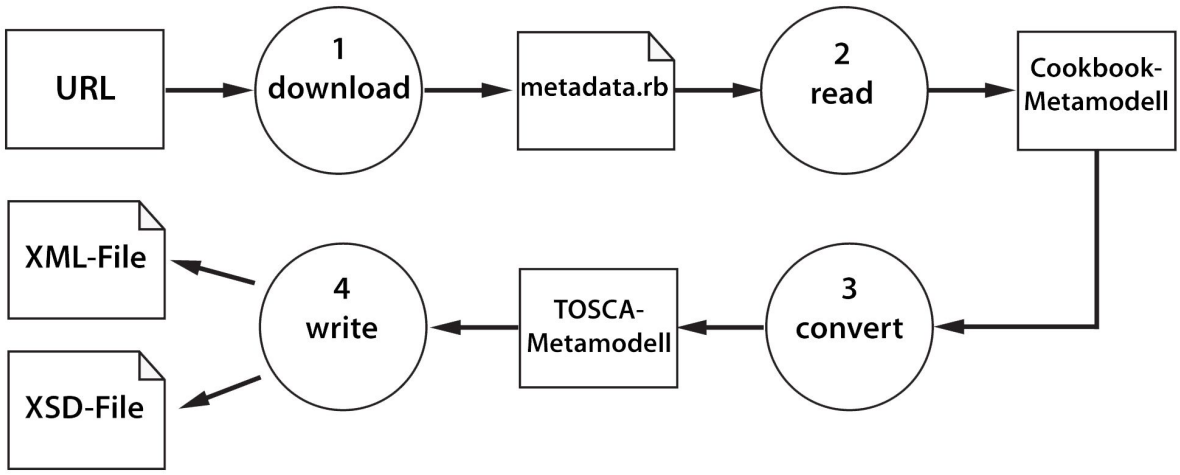

Abbildung 5.3: Ablaufdiagramm der Komponente "Node-Type-Generator"

Schritt 1: Der Downloader lädt ein Chef-Cookbook durch eine angegebene URL herunter. Die Metadatei "metadata.rb" im heruntergeladenen Chef-Cookbook kann als Eingabe des Cookbook-Ruby-Readers verwendet werden.

Schritt 2: Der Cookbook-Ruby-Reader liest die Metadatei "metadata.rb" im Chef-Cookbook ein und generiert ein entsprechendes Cookbook-Metamodell. Das daraus erzeugte Cookbook-Metamodell wird als Eingabe des Cookbook-Metamodell-Converters verwendet.

Schritt 3: Der Cookbook-Metamodell-Converter bekommt ein Cookbook-Metamodell übergeben. Das Cookbook-Metamodell wird durch eine Reihe von Transformationsregeln zu einem entsprechenden TOSCA-Metamodell für den Node-Type konvertiert. Diese Transformationsregeln sollen dem Konzept zur Erzeugung von Node-Types aus Chef-Artefakten (Unterkapitel 4.1) entsprechen. Das daraus erzeugte TOSCA-Metamodell wird als Eingabe des XSD-/XML-File-Generators verwendet.

Schritt 4: Der XML-File-Generator bekommt ein TOSCA-Metamodell für den Node-Type übergeben und kann dieses TOSCA-Metamodell in ein entsprechendes TOSCA-Definitions-Dokument schreiben. Wenn die Elemente für Node-Type-Properties im TOSCA-Metamodell vorhanden sind, wird durch den XSD-File-Generator ein entsprechendes Node-Type-Properties-Dokument generiert.

5.2.3.2 Die Funktionsweise der Komponente "Relationship-Type-Generator"

In Abbildung 5.4 wird ein vereinfachtes Ablaufdiagramm gezeigt. Das Ablaufdiagramm stellt die Funktionsweise der Teilkomponenten innerhalb der Komponente "Relationship-Type-Generator" für Juju dar. Die Funktionsweise wird in 4 Schritten beschrieben.

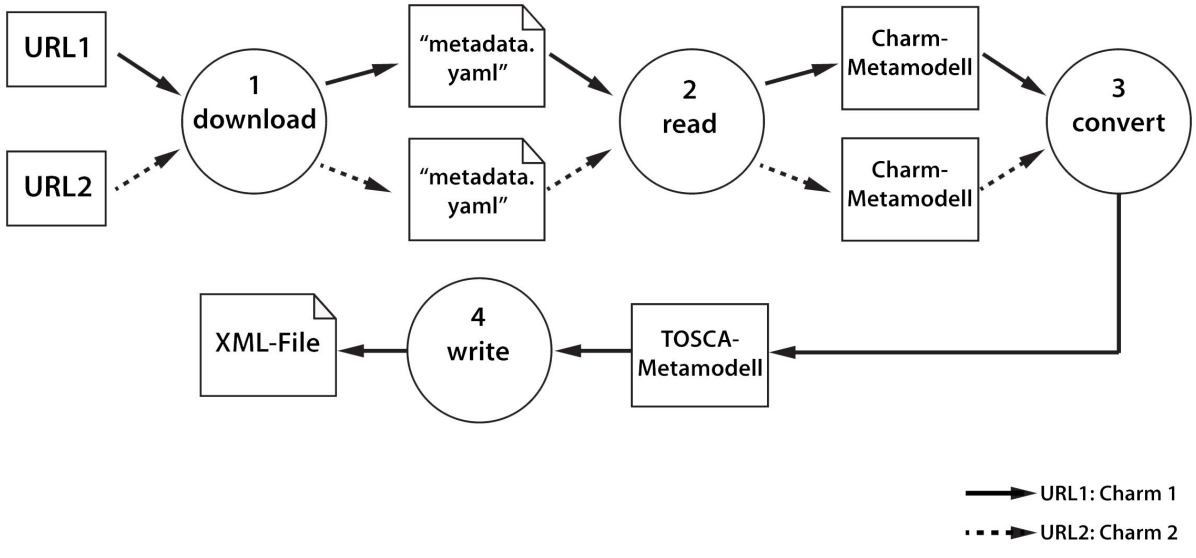

Abbildung 5.4: Ablaufdiagramm der Komponente "Relationship-Type-Generator"

Schritt 1: Durch den Downloader können zwei Juju-Charms durch zwei angegebene URLs heruntergeladen werden. Zwischen diesen zwei Juju-Charms soll eine implizite Beziehung bestehen, aus der ein entsprechender Relationship-Type generiert werden kann. Die Metadateien "metadata.yaml" in den zwei heruntergeladenen Juju-Charms können als Eingabe des Charm-YAML-Readers verwendet werden.

Schritt 2: Der Charm-YAML-Reader kann durch das Lesen von zwei Metadateien "metadata.yaml" in Juju-Charms zwei entsprechende Charm-Metamodelle generieren. Die daraus erzeugten Charm-Metamodelle werden als Eingabe des Charm-Metamodell-Converters verwendet.

Schritt 3: Der Charm-Metamodell-Converter bekommt zwei Charm-Metamodelle als Eingabe. Diese zwei Charm-Metamodelle werden durch eine Reihe von Transformationsregeln zu einem entsprechenden TOSCA-Metamodell für den Relationship-Type konvertiert. Diese Transformationsregeln sollen dem Konzept zur Erzeugung von Relationship-Types aus Juju-Artefakten (Unterkapitel 4.3) entsprechen. Das daraus erzeugte TOSCA-Metamodell wird als Eingabe des XML-File-Generators verwendet.

Schritt 4: Der XML-File-Generator bekommt ein TOSCA-Metamodell für den Relationship-Type übergeben und kann dieses TOSCA-Metamodell in ein entsprechendes TOSCA-Definitions-Dokument schreiben.

5.2.3.3 Die Funktionsweise der Komponente "Service-Template-Generator"

In Abbildung 5.5 wird ein vereinfachtes Ablaufdiagramm gezeigt. Das Ablaufdiagramm stellt die Funktionsweise der Teilkomponenten innerhalb der Komponente "Service-Template-Generator" dar. Die Funktionsweise wird in 3 Schritten beschrieben.

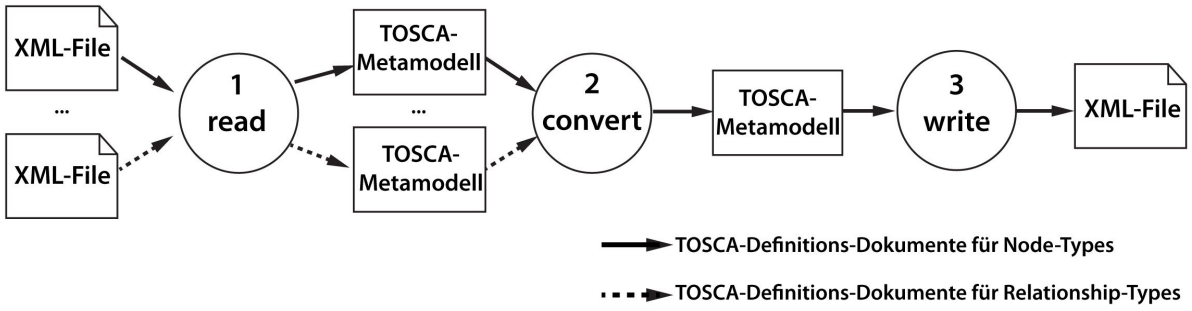

Abbildung 5.5: Ablaufdiagramm der Komponente "Service-Template-Generator"

Schritt 1: Der TOSCA-XML-Reader liest eine Reihe von TOSCA-Definitions-Dokumenten für Node-Types und Relationship-Types ein und generiert eine Reihe von TOSCA-Metamodellen für Node-Types und Relationship-Types. Die daraus erzeugten TOSCA-Metamodelle werden als Eingabe des TOSCA-Metamodell-Converters verwendet.

Schritt 2: Der TOSCA-Metamodell-Converter bekommt eine Reihe von TOSCA-Metamodellen für Node-Types und Relationship-Types übergeben. Diese TOSCA-Metamodelle werden durch eine Reihe von Transformationsregeln zu einem entsprechenden TOSCA-Metamodell für das Service-Template konvertiert. Diese Transformationsregeln sollen dem Konzept zur Erzeugung von Service-Templates (Unterkapitel 4.4) entsprechen. Das daraus erzeugte TOSCA-Metamodell wird als Eingabe des XML-File-Generators verwendet.

Schritt 3: Der XML-File-Generator bekommt ein TOSCA-Metamodell für das Service-Template übergeben und es kann in ein entsprechendes TOSCA-Definitions-Dokument geschrieben werden.

5.3 Implementierung des Prototyps

In den folgenden Abschnitten wird die Implementierung des Prototyps ausführlich beschrieben. Es werden drei Java-Pakete "org.tosca.csar", "org.tosca.meta" und "org.tosca.util" entwickelt. In diesen Paketen werden die verschiedenen Schnittstellen und ihre Implementierungsklassen sowie die Methoden in den Klassen implementiert. Im Folgenden werden die drei Pakete in Details besprochen. Zur Übersichtlichkeit werden die Parameter der Methoden weggelassen.

5.3.1 Das Paket "org.tosca.csar"

Im Java-Paket "org.tosca.csar" werden drei Schnittstellen zur Erzeugung von CSAR-Dateien definiert. Diese Schnittstellen sind "INodeTypeGenerator", "IRelationshipTypeGenerator" und "IServiceTemplateGenerator". Außerdem werden zur Übersichtlichkeit zwei letzen Schritte in den folgenden drei Abbildungen weggelassen. Nachdem das entsprechende TOSCA-Definitions-Dokument generiert wurde, wird die TOSCA-Metadatei "TOSCA.meta" durch Aufruf der Methode "generate" der Klasse "MetaFileGenerator" generiert. Schließlich wird die Methode "generate" der Klasse "ZIPFileGenerator" aufgerufen, um alle generierten Dokumente (z.B. das Node-Type-Properties-Dokument, das TOSCA-Definitions-Dokument für den Node-Type, den Relationship-Type und das Service-Template sowie die TOSCA-Metadatei) und alle entsprechenden Artefakte (z.B. Chef-Cookbooks und Juju-Charms) zu einer CSAR-Datei zu zippen.

5.3.1.1 Die Schnittstelle "INodeTypeGenerator"

In der Schnittstelle "INodeTypeGenerator" wird eine abstrakte Methode "generate" definiert, um eine CSAR-Datei für den Node-Type zu generieren. Die Implementierungsklassen dieser Schnittstelle könnten die Klassen "NodeTypeFromCookbook" und "NodeTypeFromCharm" sein. Die Implementierung der Klasse "NodeTypeFromCharm" wurde in der Studienarbeit "Vorlagen für das Deployment von Services und Applikationen in der Cloud" [33] dargestellt. In dieser Arbeit beschäftigen wir uns nur mit der Implementierung der Klasse "NodeTypeFromCookbook".

NodeTypeFromCookbook

Die Klasse "NodeTypeFromCookbook" implementiert die Schnittstelle "INodeTypeGenerator" und überschreibt die abstrakte Methode "generate". Diese Methode wird verwendet, um eine TOSCA CSAR-Datei für den Node-Type aus einem Chef-Cookbook zu generieren. Diese Methode benötigt einen Eingabeparameter "csar_location" vom Typ "java.lang.String". Dieser Eingabeparameter zeigt, wo die generierte CSAR-Datei ausgegeben werden soll. Als Ausgabe liefert diese Methode eine entsprechende CSAR-Datei für den Node-Type. In Abbildung 5.6 wird ein vereinfachtes Pseudo-Sequenzdiagramm für die Methode “generate” dargestellt. Im ersten Schritt ruft die Methode "generate" die Methode "read" der Klasse "MetamodelFromCookbook" auf. Die Methode "read" liest die Metadatei "metadata.rb" im Chef-Cookbook ein und liefert als Ausgabe ein Objekt der Klasse "CookbookMetamodel". Im zweiten Schritt wird die Methode "convert" der Klasse "CookbookToTosca" aufgerufen. Die Methode "convert" bekommt das generierte CookbookMetamodel-Objekt übergeben und liefert als Ausgabe ein Objekt der Klasse "TOSCAMetamodel".

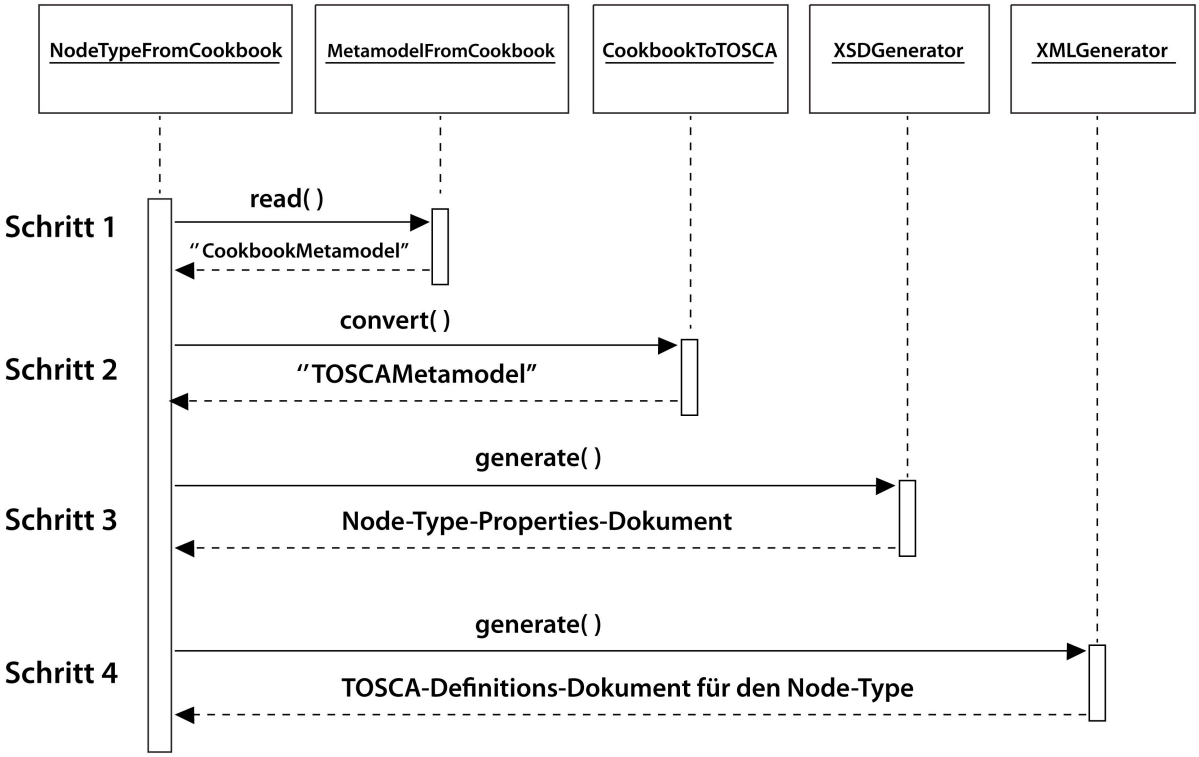

Abbildung 5.6: Sequenzdiagramm für die Klasse "NodeTypeFromCookbook"

Wenn die Elemente für die Node-Type-Properties im TOSCAMetamodel-Objekt vorhanden ist, wird als dritter Schritt die Methode "generate" der Klasse "XSDGenerator" aufgerufen, um ein entsprechendes Node-Type-Properties-Dokument zu generieren. Andernfalls kann der dritte Schritt ignoriert. Durch das generierte TOSCAMetamodel-Objekt generiert als letzter Schritt die Methode "generate" der Klasse "XMLGenerator" ein TOSCA-Definitions-Dokument für den Node-Type.

5.3.1.2 Die Schnittstelle "IRelationshipTypeGenerator"

In der Schnittstelle "IRelationshipTypeGenerator" wird eine abstrakte Methode "generate" definiert, um eine TOSCA CSAR-Datei für den Relationship-Type zu erzeugen. Die Klasse "RelationshipTypeFromCharms" ist die einzige Implementierungsklasse dieser Schnittstelle.

RelationshipTypeFromCharms

Die Klasse "RelationshipTypeFromCharms" implementiert die Schnittstelle "IRelationshipTypeGenerator" und überschreibt die abstrakte Methode "generate". Diese Methode wird verwendet, um eine TOSCA CSAR-Datei für den Relationship-Type aus zwei Juju-Charms mit einer bestimmten Beziehung zu generieren. Diese Methode benötigt einen Eingabeparameter "csar_location" vom Typ "java.lang.String". Dieser Eingabeparameter zeigt, wo die generierte CSAR-Datei ausgegeben werden soll. Als Ausgabe liefert diese Methode eine entsprechende CSAR-Datei für den Relationship-Type. In Abbildung 5.7 wird ein vereinfachtes Pseudo-Sequenzdiagramm für die Methode "generate" dargestellt. Im ersten Schritt ruft die Methode "generate" die Methode "read" der Klasse "MetamodelFromCharm" auf. Die Methode "read" liest die Metadatei "metadata.yaml" im Juju-Charm ein und liefert als Ausgabe ein Objekt der Klasse "CharmMetamodel". Da es zwei Juju-Charms gibt, wird

die Methode "read" zweimal aufgerufen. Deshalb werden zwei CharmMetamodel-Objekte generiert.

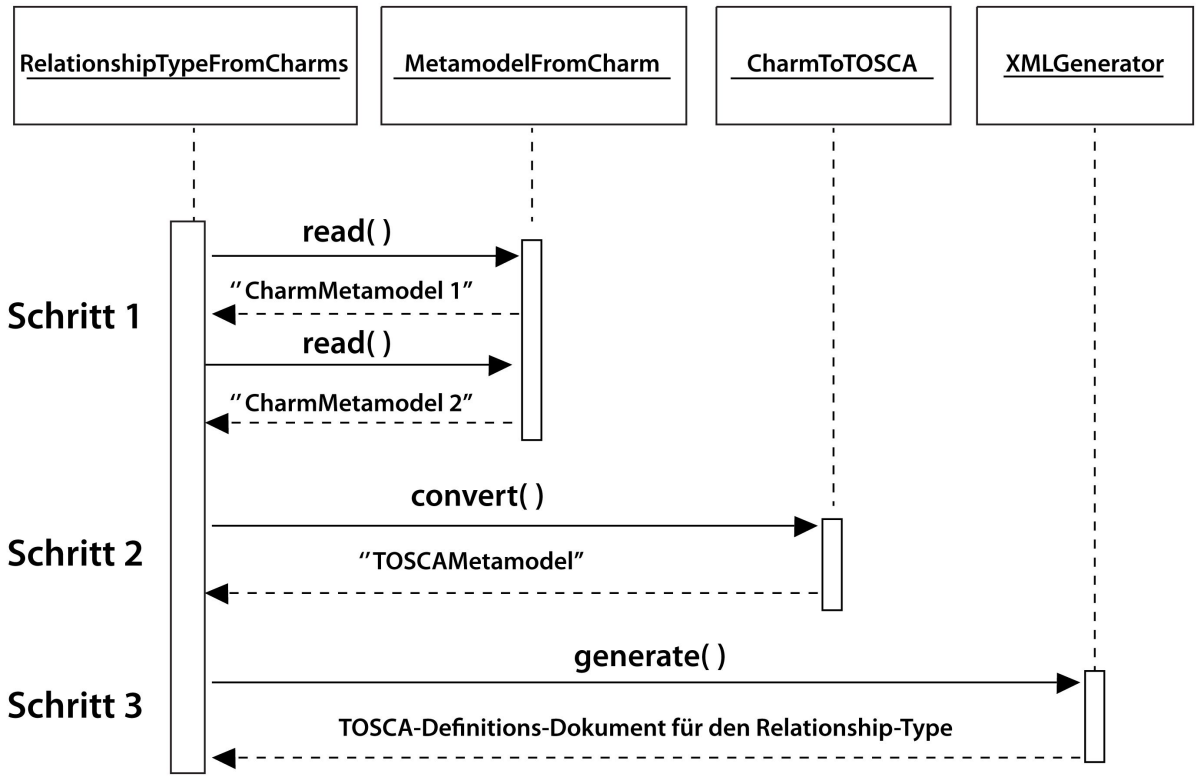

Abbildung 5.7: Sequenzdiagramm für die Klasse "RelationshipTypeFromCharms"

Im zweiten Schritt wird die Methode "convert" der Klasse "CharmToTOSCA" aufgerufen. Die Methode "convert" bekommt zwei generierten CharmMetamodel-Objekte übergeben und liefert als Ausgabe ein Objekt der Klasse "TOSCAMetamodel". Durch das generierte TOSCAMetamodel-Objekt generiert als letzter Schritt die Methode "generate" der Klasse "XMLGenerator" ein TOSCA-Definitions-Dokument für den Relationship-Type.

5.3.1.3 Die Schnittstelle "IServiceTemplateGenerator"

In der Schnittstelle "IServiceTemplateGenerator" wird eine abstrakte Methode "generate" definiert, um eine TOSCA CSAR-Datei für das Service-Template zu erzeugen. Die Klasse "ServiceTemplateFromTypes" ist die einzige Implementierungsklasse dieser Schnittstelle.

ServiceTemplateFromTypes

Die Klasse "ServiceTemplateFromTypes" implementiert die Schnittstelle "IServiceTemplateGenerator" und überschreibt die abstrakte Methode "generate". Diese Methode wird verwendet, um eine TOSCA CSAR-Datei für das Service-Template aus einer Reihe von TOSCA-Definitions-Dokumenten für Node-Types und Relationship-Types zu generieren. Diese Methode benötigt einen Eingabeparameter "csar_location" vom Typ "java.lang.String". Dieser Eingabeparameter zeigt, wo die generierte CSAR-Datei ausgegeben werden soll. Als Ausgabe liefert diese Methode eine entsprechende CSAR-Datei für das Service-Template. Die TOSCA-Definitions-Dokumente für Node-Types und Relationship-Types können wir durch die Konstruktormethode "ServiceTemplateFromTypes" bekommen.

In Abbildung 5.8 wird ein vereinfachtes Pseudo-Sequenzdiagramm für die Methode "generate" dargestellt.

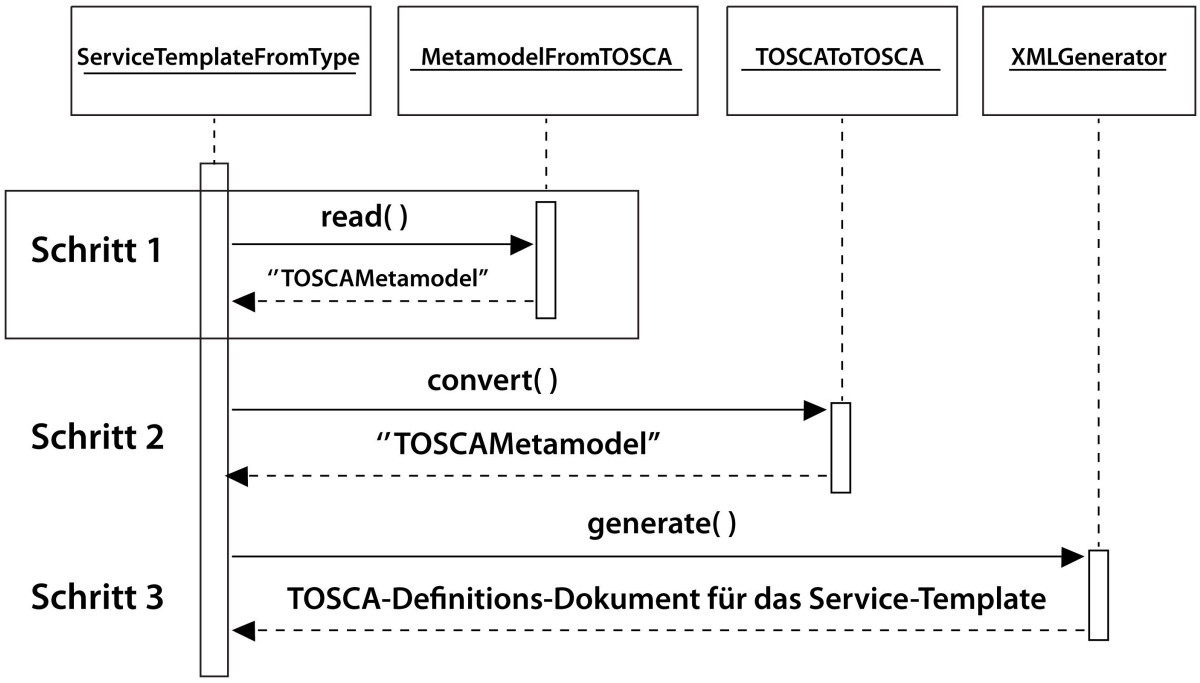

Abbildung 5.8: Sequenzdiagramm für die Klasse "ServiceTemplateFromType"

Im ersten Schritt ruft die Methode "generate" für jedes TOSCA-Definitions-Dokument einmal die Methode "read" der Klasse "MetamodelFromTOSCA" auf. Das heißt, dass der erste Schritt im Kästchen in der Abbildung mehrmals wiederholt wird. Diese Methode "read" liest ein TOSCA-Definitions-Dokument ein und liefert ein Objekt der Klasse "TOSCAMetamodel" als Ausgabe. Wir instanziieren zwei Objekte vom Typ "java.util.List". Ein Objekt dient zum Speichern der generierten TOSCAMetamodel-Objekte für Node-Types. Das andere dient zum Speichern der generierten TOSCAMetamodel-Objekte für Relationship-Types. Im zweiten Schritt wird die Methode "convert" der Klasse "TOSCAToTOSCA" aufgerufen. Die Methode "convert" bekommt einen Service-Namen vom Typ "java.lang.String" für das Service-Template und zwei generierten Objekte vom Typ "java.util.List" übergeben und liefert als Ausgabe ein Objekt der Klasse "TOSCAMetamodel". Durch das generierte TOSCAMetamodel-Objekt generiert als letzter Schritt die Methode "generate" der Klasse "XMLGenerator" ein TOSCA-Definitions-Dokument für das Service-Template.

5.3.2 Das Paket "org.tosca.meta"

In Abbildung 5.9 wird ein Klassendiagramm ohne Methoden und Attributen für das Java-Paket "org.tosca.meta" dargestellt. In dieserm Paket werden drei Schnittstellen und ihre Implementierungsklassen definiert. Die Implementierungsklassen der Schnittstelle "IMetamodel" könnten die Klasse "TOSCAMetamodel", "CookbookMetamodel" und "CharmMetamodel" sein. Die Implementierungsklassen der Schnittstelle "IMetamodelGenerator" könnten die Klassen "MetamodelFromCookbook", "MetamodelFromCharm" und "MetamodelFromTOSCA" sein. Die Implementierungsklassen der Schnittstelle "IMetamodelConverter" könnten die Klassen "CookbookToTOSCA", "CharmToTOSCA" und "TOSCAToTOSCA" sein.

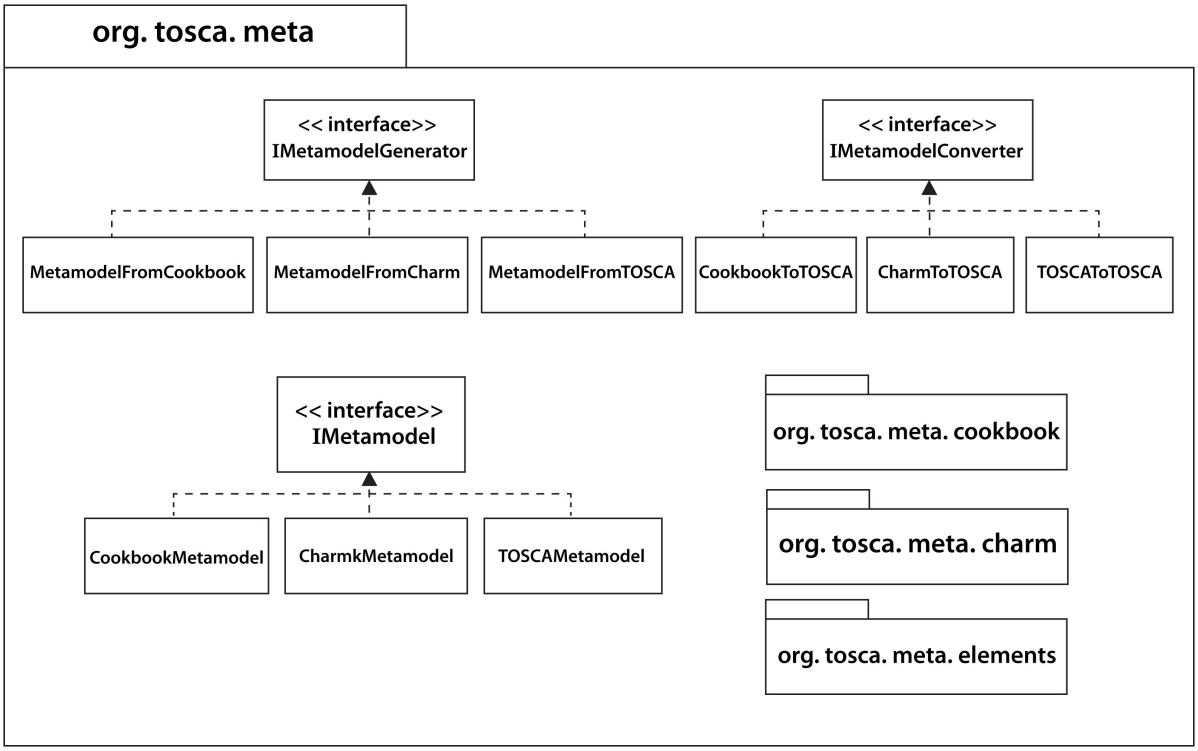

Abbildung 5.9: Klassendiagramm für das Java-Paket "org.tosca.meta"

Außerdem werden im Paket "org.tosca.meta" noch drei Pakete definiert. Diese Pakete sind "org.tosca.meta.charm", "org.tosca.meta.cookbook" und "org.tosca.meta.elements". Details werden in den folgenden Unterkapiteln besprochen.

5.3.2.1 Die Schnittstelle "IMetamodel"

In der Schnittstelle "IMetamodel" wird keine Methode definiert. Es wird nur gezeigt, dass eine Klasse als Metamodell verwendet wird, wenn diese Klasse diese Schnittstelle implementiert.

TOSCAMetamodel

Die Klasse "TOSCAMetamodel" implementiert die Schnittstelle "IMetamodel". Diese Klasse enthält als Attribute alle Elemente im TOSCA-Definitions-Dokument. Diese Elemente werden durch die entsprechenden Klassen definiert. Beispielsweise können die Elemente *Definitions*, *Import*, *RequirementType, CapabilityType*, *ArtifactType*, *ArtifactTemplate*, *NodeType NodeTypeImplementation, RelationshipType* und *RelationshipTypeImplementation* durch die Klassen "TDefinitions", "TImport", "TRequirementType", "TCapabilityType", "TArtifactType", "TArtifactTemplate", "TNodeType", "TNodeTypeImplementation", "TRelationshipType" und "TRelationshipTypeImplementation" definiert werden. Zur Generierung dieser Klassen wird die JAXB-Technologie [34] benutzt. JAXB ist die Abkürzung von "Java Architecture for XML Binding" und ist eine Programmschnittstelle in Java, die es ermöglicht, Daten aus einer XML-Schema-Instanz heraus automatisch an Java-Klassen zu binden, und diese Java-Klassen aus einem XML-Schema heraus zu generieren [42]. So können die entsprechenden Java-Klassen aus dem TOSCA-Schema [17] heraus generiert und im Java-Paket "org.tosca.meta.elements" gespeichert werden. Außerdem besitzt die Klasse "TOSCAMetamodel" noch ein Attribut "propertiesXSD" der Klasse

"TPropertiesXSD". Die Klasse "TPropertiesXSD" wird manuell geschrieben und im Java-Paket "org.tosca.meta.elements" gespeichert. Sie dient zur Generierung eines Node-Type-Properties-Dokuments. Durch ein Objekt der Klasse "TOSCAMetamodel" kann ein entsprechendes TOSCA-Definitions-Dokument generiert werden.

CookbookMetamodel

Die Klasse "CookbookMetamodel" implementiert die Schnittstelle "IMetamodel". In dieser Klasse werden alle Informationen z.B. "recipes", "dependencies" und "attribute" in der Datei "metadata.rb" in einem Chef-Cookbook gespeichert. Für jede Information wird eine entsprechende Klasse erstellt. Dies sind die Klassen "Recipe", "Dependency" und "Attribute" und werden im Java-Paket "org.tosca.meta.cookbook" gespeichert. Durch ein Objekt der Klasse "CookbookMetamodel" kann ein entsprechendes Objekt der Klasse "TOSCAMetamodel" generiert werden.

CharmMetamodel

Die Klasse "CharmMetamodel" implementiert die Schnittstelle "IMetamodel". In dieser Klasse werden alle Informationen z.B. "requires", "provides" und "peers" in der Datei "metadata.yaml" in einem Juju-Charm gespeichert. Für jede Information wird eine entsprechende Klasse erstellt. Dies sind die Klassen "Require", "Provide" und "Peer" und werden im Java-Paket "org.tosca.meta.charm" gespeichert. Durch ein Objekt der Klasse "CharmMetamodel" kann ein entsprechendes Objekt der Klasse "TOSCAMetamodel" generiert werden.

5.3.2.2 Die Schnittstelle "IMetamodelGenerator"

In der Schnittstelle "IMetamodelGenerator" wird eine abstrakte Methode "read" definiert, um ein entsprechendes Metamodell aus Metadaten eines Artefakts zu erzeugen. Beispielsweise kann diese Methode ein entsprechendes Cookbook- oder Charm-Metamodell aus der Datei "matadata.rb" in einem Chef-Cookbook oder aus der Datei "matadata.yaml" in einem Juju-Charm generieren. Außerdem kann durch diese Methode ein entsprechendes TOSCA-Metamodell aus einem TOSCA-Definitions-Dokument für den Node-Type oder den Relationship-Type erzeugt werden.

MetamodelFromCookbook

Die Klasse "MetamodelFromCookbook" implementiert die Schnittstelle "IMetamodelGenerator" und überschreibt die Methode "read". Diese Methode wird verwendet, um ein Objekt der Klasse "CookbookMetamodel" aus der Datei "metadata.rb" in einem Chef-Cookbook zu generieren. Die Datei "metadata.rb" können wir durch die Konstruktormethode "MetamodelFromCookbook" bekommen. Als Ausgabe liefert diese Methode ein CookbookMetamodel-Objekt. Da Chef von der Skriptsprache Ruby implementiert wurde, können wir die Ruby-Skripte benutzen, um die Datei "metadata.rb" einfacher zu lesen und zu analysieren. Um den Aufruf der Ruby-Skripte in der Java-Laufzeitumgebung zu realisieren, haben wir uns für JRuby [36] entschieden. Der Grund dafür ist, dass JRuby eine Implementierung eines Ruby-Interpreters in Java ist und die Interaktion von Java und Ruby in beiden Richtungen ermöglicht. Damit ermöglicht JRuby die Nutzung von Ruby als eine alternative Sprache für die Java-Laufzeitumgebung [37]. Wir erstellen eine JRuby-Datei

namens "metaparser.rb". Diese Datei enthält die Ruby-Skripte zum Lesen der Inhalte der Datei "metadata.rb" eines Chef-Cookbook und liefert als Ausgabe ein Objekt der Klasse "CookbookMetamodel". Im Anhang 2 wird die Datei "metaparser.rb" präsentiert. Zuerst erstellen wir eine Instanz der Klasse "java.script.ScriptEngine" für JRuby. Durch diese Instanz kann die Datei "metaparser.rb" aufgerufen werden. Um dies zu realisieren, müssen die JAR-Dateien "jsr223.jar" und "jruby-1.7.4" in die Java-Laufzeitumgebung importiert werden.

MetamodelFromCharm

Die Klasse "MetamodelFromCharm" implementiert die Schnittstelle "IMetamodelGenerator" und überschreibt die Methode "read". Diese Methode wird verwendet, um ein Objekt der Klasse "CharmMetamodel" aus der Datei "metadata.yaml" und der Datei "config.yaml" (wenn sie vorhanden ist) in einem Juju-Charm zu generieren. Die Dateien "metadata.rb" und "config.yaml können wir durch die Konstruktormethode "MetamodelFromCharm" bekommen. Als Ausgabe liefert diese Methode ein CharmMetamodel-Objekt. Zu den konkreten Arbeiten gehören, (1) die YAML-Datei zu lesen und zu analysieren, (2) die Inhalte der YAML-Datei in den abstrakten Objekten zu speichern und (3) die abstrakten Objekte zu einem CharmMetamodel-Objekt zu transformieren. Für die Implementierung dieser Arbeiten werden drei Klassen "YamlModel", "YamlModelList" und "YamlReader" verwendet. Die ersten zwei Klassen dienen zum Speichern der Inhalte in einer YAML-Datei. Die letzte Klasse dient zum Lesen und Analysieren einer YAML-Datei. Dabei wird eine YAML-Software von Drittanbietern verwendet: SnakeYAML [41] ist ein YAML-Parser für Programmiersprache "Java" und bekannt dafür, dass er ein kompletter YAML1.1-Parser ist. Durch SnakeYAML kann die YAML-Datei einfach gelesen und analysiert werden. Die konkrete Implementierung dieser drei Klassen wurde in der Studienarbeit "Vorlagen für das Deployment von Services und Applikationen in der Cloud" [33] dargestellt.

MetamodelFromTOSCA

Die Klasse "MetamodelFromTOSCA" implementiert die Schnittstelle "IMetamodelGenerator" und überschreibt die Methode "read". Diese Methode wird verwendet, um ein Objekt der Klasse "TOSCAMetamodel" aus einem TOSCA-Definitions-Dokument zu generieren. Das TOSCA-Definitions-Dokument können wir durch die Konstruktormethode "MetamodelFromTOSCA" bekommen. Als Ausgabe liefert diese Methode ein TOSCAMetamodel-Objekt. Die konkrete Arbeit ist, jedes XML-Element im TOSCA-Definitions-Dokument als ein entsprechendes Objekt im TOSCAMetamodel-Objekt zu speichern. Durch JAXB können die XML-Elemente in einem TOSCA-Definitions-Dokument zu den entsprechenden Objekten transformiert werden.

5.3.2.3 Die Schnittstelle "IMetamodelConverter"

In der Schnittstelle "IMetamodelConverter" werden drei abstrakten Methoden "convert" definiert. Die Methode mit einem Parameter dient dazu, ein Cookbook- oder Charm-Metamodell zu einem TOSCA-Metamodell für einen Node-Type zu konvertieren. Die Methode mit zwei Parametern dient dazu, zwei Charm-Metamodelle zu einem TOSCA-Metamodell für einen Relationship-Type zu konvertieren. Die Methode mit drei Parametern dient dazu, eine Reihe von TOSCA-Metamodellen für Node-Types und Relationship-Types zu einem TOSCA-Metamodell für ein Service-Template zu konvertieren.

CookbookToTOCSA

Die Klasse "CookbookToTOCSA" implementiert die Schnittstelle "IMetamodelConverter" und überschreibt drei abstrakten Methoden "convert". In den Methoden mit mehreren Parametern werden keine Codes geschrieben und NULL wird zurückgegeben. Die Methode mit einem Parameter bekommt ein CookbookMetamodel-Objekt übergeben und liefert als Ausgabe ein TOSCAMetamodel-Objekt für einen Node-Type. Der Kern dieser Methode ist, durch die entsprechenden Transformationsregeln ein Cookbook-Metamodell zu einem TOSCA-Metamodell zu konvertieren. Diese Regeln müssen dem Konzept (Unterkapitel 4.1) für die Erzeugung von TOSCA Node-Types aus Chef-Artefakten entsprechen.

Zuerst wird ein Objekt der Klasse "TOSCAMetamodel" zum Speichern der XML-Elemente in einem TOSCA-Definitions-Dokument instanziiert. Wenn das CookbookMetamodel-Objekt ein Objekt der Klasse "java.util.List" zum Speichern der Cookbook-Attribute enthält, kann ein Objekt der Klasse "TPropertiesXSD" erzeugt werden. Das generierte Objekt enthält die XSD-Elemente zur Erzeugung eines Node-Type-Properties-Dokuments und wird im TOSCAMetamodel-Objekt gespeichert. Diese XSD-Elemente werden durch Dom4j [38] erzeugt. Dom4j ist eine in der Programmiersprache Java geschriebene Open-Source-Programmierschnittstelle für den Zugriff und die Verarbeitung von XML-Dokumenten [39]. Dabei muss ein Objekt der Klasse "TImport" zum Importieren des generierten Node-Type-Properties-Dokuments in das entsprechende TOSCA-Definitions-Dokument für den Node-Type erzeugt. Dieses Objekt wird dann im TOSCAMetamodel-Objekt gespeichert.

Außerdem können durch das CookbookMetamodel-Objekt die folgenden Elemente erzeugt werden: *Definitions*, *CapabilityType*, *ArtifactType*, *ArtifactTemplate*, *NodeType* und *NodeTypeImplementation*. Jedes Element kann durch die Instanziierung eines Objektes der entsprechenden Klasse implementiert werden. Beispielsweise kann das Element *NodeType* durch das Erzeugen eines Objektes der Klasse "TNodeType" implementiert werden. Dasselbe gilt auch für die Generierung von anderen Elementen. Diese generierten Objekte werden auch im TOSCAMetamodel-Objekt gespeichert. Da die Erzeugung der Elemente auf gleiche Art und Weise implementiert wird, werden wir als Beispiel nur die Implementierung der Elemente *ArtifactType* und *ArtifactTemplate* beschreiben.

In Unterkapitel 4.1.3 wird besprochen, dass der direkte Ansatz verwendet wird, um die Chef-Cookbooks in ein Service-Template einzubetten. Deswegen erzeugen wir durch die Klasse "TArtifactType" das Element *ArtifactType* namens "ChefArtifact", welche der Struktur der Chef-Artefakte entspricht. Beim Erzeugen des Elements *ArtifactTemplate* benutzen wir einen unveränderlichen Universally Unique Identifier (UUID) [40] als ein Wert des Attributs *id* des Elements *ArtifactTemplate*. Ein UUID als ein Standard für Identifikatoren beschreibt einen 128-Bit-Wert und wird verwendet, um Informationen eindeutig kennzeichnen zu können. Hier dient ein UUID dazu, ein Artifact-Template eindeutig zu identifizieren. Der Inhalt des Kindelements *Properties* ist Chef-spezifisch. Folglich brauchen wir ein Element *ChefArtifactProperties* als Inhalt des Element *Properties*. Durch die JAXB-Technologie kann die Klasse "ChefArtifactProperties" aus der XML-Schema-Datei "ChefArtifact.xsd" heraus generiert und im Java-Paket "org.tosca.meta.elements" gespeichert werden. Die Klasse "ChefArtifactProperties" enthält als Attribute alle Elemente wie z.B. *Cookbook*, *Mappings* und *Runlist*, die in der Datei "ChefArtifact.xsd" definiert sind. Die Generierung des Elements

ChefArtifactProperties und seiner Kindelemente *Cookbooks*, *Mappings* und *RunList* wird durch die Klasse "TChefArtifactProperties" implementiert. Dann wird das Element *ChefArtifactProperties* als ein Objekt der Klasse "JAXBElement" zum Element *Properties* hinzugefügt.

CharmToTOCSA

Die Klasse "CharmToTOCSA" implementiert die Schnittstelle "IMetamodelConverter" und überschreibt drei Methoden "convert". In der Methode mit drei Parametern werden keine Codes geschrieben und NULL wird zurückgegeben. Die Methode mit einem Parameter bekommt ein CharmMetamodel-Objekt übergeben und liefert als Ausgabe ein Objekt von der Klasse "TOSCAMetamodel" für einen Node-Type. Der Kern dieser Methode ist, durch die entsprechenden Transformationsregeln ein Charm-Metamodell zu einem TOSCA-Metamodell zu konvertieren. Diese Regeln müssen dem Konzept (Unterkapitel 4.2) für die Erzeugung von TOSCA Node-Types aus Juju-Artefakten entsprechen. Die konkrete Implementierung dieser Methode wurde in der Studienarbeit "Vorlagen für das Deployment von Services und Applikationen in der Cloud" [33] dargestellt. In dieser Diplomarbeit beschäftigen wir uns nur mit der Implementierung der Methode mit zwei Parametern. Diese Methode bekommt zwei CharmMetamodel-Objekte übergeben und liefert als Ausgabe ein Objekt von der Klasse "TOSCAMetamodel" für einen Relationship-Type. Der Kern dieser Methode ist, durch die entsprechenden Transformationsregeln zwei Charm-Metamodelle zu einem TOSCA-Metamodell zu konvertieren. Diese Regeln müssen dem Konzept (Unterkapitel 4.3) für die Erzeugung von TOSCA Relationship-Types aus zwei Juju-Artefakten entsprechen.

Zuerst wird ein Objekt der Klasse "TOSCAMetamodel" zum Speichern der XML-Elemente in einem TOSCA-Definitions-Dokument instanziiert. Dann können durch diese zwei CharmMetamodel-Objekte die folgenden Elemente erzeugt werden: *Definitions*, *ArtifactType*, *ArtifactTemplate*, *RelationshipType* und *RelationshipTypeImplementation*. Jedes Element kann durch die Instanziierung eines Objektes der entsprechenden Klasse implementiert werden. Beispielsweise kann das Element *RelationshipType* durch das Erzeugen eines Objektes der Klasse "TRelationshipType" implementiert werden. Dasselbe gilt auch für die Generierung von anderen Elementen. Diese generierten Objekte werden auch im TOSCAMetamodel-Objekt gespeichert. Da die Erzeugung der Elemente auf gleiche Art und Weise implementiert wird, werden wir als Beispiel nur die Implementierung der Elemente *ArtifactType* und *ArtifactTemplate* beschreiben.

In Unterkapitel 4.3.1 wird besprochen, dass Juju-Charms auf eine transparente Art und Weise unter Verwendung des Standard-Artifact-Type "Script Artifact" in ein Service-Template eingebettet werden. Deswegen erzeugen wir zuerst durch die Klasse "TArtifactType" das Element *ArtifactType* namens "ScriptArtifact". Dann muss festgelegt werden, welche Relation-Hooks zur Herstellung der entsprechenden Beziehung aufgerufen werden. Für diese Relation-Hooks müssen die zusätzlichen Wrapper-Dateien erstellt werden, um diesen transparenten Ansatz zu realisieren. Folglich muss für jede Wrapper-Datei ein Artifact-Template vom Standard-Artifact-Type "Script Artifact" generiert werden. Beim Erzeugen des Elements *ArtifactTemplate* benutzen wir einen UUID, um das Artifact-Template eindeutig zu identifizieren. Der Inhalt seines Kindelements *Properties* ist vom Standard-Artifact-Type

"Script Artifact". Folglich brauchen wir ein Element *ScriptArtifactProperties* als Inhalt des Element *Properties*. Durch die JAXB-Technologie kann die entsprechende Klasse "ScriptArtifactProperties" aus der XML-Schema-Datei "ScriptArtifact.xsd" heraus generiert und im Java-Paket "org.tosca.meta.elements" gespeichert werden. Die Klasse "ScriptArtifactProperties" enthält als Attribute alle Elemente wie z.B. *ScriptLanguage* und *PrimaryScript*, die in der Datei "ScriptArtifact.xsd" definiert sind. Die Generierung des Elements *ScriptArtifactProperties* und seiner Kindelemente *ScriptLanguage* und *PrimaryScript* wird durch die Klasse "ScriptArtifactProperties" implementiert. Schließlich wird das Element *ScriptArtifactProperties* als ein Objekt der Klasse "JAXBElement" zum Element *Properties* hinzugefügt.

TOSCAToTOCSA

Die Klasse "TOSCAToTOCSA" implementiert die Schnittstelle "IMetamodelConverter" und überschreibt drei Methoden "convert". In den Methoden mit einem und zwei Parametern werden keine Codes geschrieben und NULL wird zurückgegeben. Die Methode mit drei Parametern bekommt einen Service-Namen vom Typ "java.lang.String" für das Service-Template und zwei Objekte vom Typ "java.util.List" übergeben und liefert als Ausgabe ein Objekt der Klasse "TOSCAMetamodel" für ein Service-Template. Ein Eingabe-Objekt dient zum Speichern der TOSCAMetamodel-Objekte für Node-Types. Das andere dient zum Speichern der TOSCAMetamodel-Objekte für Relationship-Types. Der Kern dieser Methode ist, durch die entsprechenden Transformationsregeln die TOACA-Metamodelle für Node-Types und Relationship-Types zu einem TOSCA-Metamodell für ein Service-Tempate zu konvertieren. Diese Regeln müssen dem Konzept (Unterkapitel 4.4) für die Erzeugung von TOSCA Service-Templates durch Orchestrierung der Node-Types und Relationship-Types entsprechen.

Zuerst wird ein Objekt der Klasse "TOSCAMetamodel" zum Speichern der XML-Elemente in einem TOSCA-Definitions-Dokument für ein Service-Template instanziiert. Dann werden die XML-Elemente *Definition, ServiceTemplate* und *TopologyTemplate* generiert. Außerdem können durch zwei Eingabe-Objekte eine Reihe der folgenden Elemente erzeugt werden: *Import*, *NodeTemplate* und *RelationshipTemplate*. Beispielsweise können die Elemente *Import* und *NodeTemplate* durch ein TOSCAMetamodel-Objekt für einen Node-Type erzeugt werden. Dasselbe gilt auch für die Generierung der Elemente *RelationshipTemplate*. Diese Elemente können durch die Instanziierung der Objekte der entsprechenden Klassen "Definitions", "TServiceTemplate", "TTopologyTemplate" "TImport", "TNodeTemplate" und "TRelationshipTemplate" implementiert werden. Diese generierten Objekte werden auch im TOSCAMetamodel-Objekt für das Service-Template gespeichert.

5.3.3 Das Paket "org.tosca.util"

Im Java-Paket "org.tosca.util" werden einige Klassen für die Hilfefunktionen des Prototyps implementiert. In der Klasse "Downloader" werden zwei Methoden "git" und "bazaar" definiert, die zum Herunterladen der verschiedenen Artefakte dienen. Da Chef-Artefakte in einem öffentlichen GIT Repository auf GitHub [46] hinterlegt werden, wird die Methode "git" zum Herunterladen der Chef-Artefakte durch eine entsprechende URL als Eingabe verwendet. Ebenso wird die Methode "bazaar" zum Herunterladen der Juju-Artefakte verwendet, da sie in einem öffentlichen Bazaar [47] Repository auf Lauchpad [48] aufbewahrt werden.

Die Klasse "XSDGenerator" dient zur Generierung einer XSD-Datei (in unserem Fall, eines Node-Type-Properties-Dokuments). Zuerst wird ein XML-Dokument durch die Methode "createDocument" der Klasse "org.dom4j.DocumentHelper" erzeugt. Dieses XML-Dokument dient zum Speichern aller entsprechenden XML-Elemente für das TOSCA Node-Type-Properties-Dokument. Dann müssen das Wurzelelement und seine Kindelemente zu diesem Dokument hinzugefügt werden. Schließlich wird dieses Dokument durch die Methode "write" der Klasse "org.dom4j.io.XMLWriter" in eine XSD-Datei geschrieben. Um diese Klassen zu verwenden, muss die JAR-Datei "dom4j-1.6.1" in die Java-Laufzeitumgebung importiert werden.

Die Klasse "XMLGenerator" dient zur Generierung einer XML-Datei (in unserem Fall, eines TOSCA-Definition-Dokuments). Zuerst wird eine JAXBContext-Instanz durch die Methode "newInstance" der Klasse "JAXBContext" erstellt, um die Funktion "marschall" in JAXBContext verwenden zu können. Diese Methode bekommt als Eingabe im Rahmen dieser Arbeit den Paket-Namen "org.tosca.meta.elements" vom Typ "java.lang.String". Dieses Paket "org.tosca.meta.elements" enthält alle Java-Klassen, die durch die JAXB-Technologie aus dem TOSCA-Schema [17] generiert wurden. Dann wird ein Objekt der Klasse "Definitions" durch die Methode "createDefinitions" der Klasse "org.tosca.meta.elements.ObjectFactory" erzeugt. Dieses Objekt ist das Wurzelelement eines TOSCA-Definitions-Dokuments und die anderen entsprechenden Elemente müssen zum Wurzelelement hinzugefügt werden. Schließlich wird das Wurzelelement durch die Methode "marshal" der Klasse "javax.xml.bind.Marshaller" in eine XML-Datei geschrieben.

Die Klasse "MetaFileGenerator" dient zur Generierung der Metadatei "TOSCA.meta". Die Methode "generate" in dieser Klasse bekommt zwei Pfadnamen vom Typ "java.lang.String" übergeben. Der erste Pfadname zeigt, wo sich der Ordner für ein TOSCA-Artefakt befindet. Der zweite zeigt, wo die zu generierende Metadatei "TOSCA.meta" ausgegeben werden sollte. Die konkrete Implementierung ist, dass ein Objekt der Klasse "java.lang.StringBuffer" durch die Klasse "java.io.FileOutputStream" in eine Datei vom Typ "java.io.File" geschrieben wird. Dasselbe gilt auch für die Klasse "WrapperFileGenerator" zur Generierung der Wrapper-Datei.

Die Klasse "FileUtils" stellt die Operationen zum Ordner zur Verfügung, z.B. nach einer Datei zu suchen, alle Dateinamen und Pfadnamen aufzulisten und einen Ordner sowie die Inhalte im Ordner zu einem anderen Ordner zu kopieren. Dazu wird das Java-Paket "java.io" verwendet. Die Klasse "ZIPFileGenerator" dient dazu, einen Ordner zu einer Zipdatei zu komprimieren. Dazu wird das Java-Paket "java.util.zip" verwendet. Die konkrete Implementierung dieser zwei Klassen wurde in der Studienarbeit "Vorlagen für das Deployment von Services und Applikationen in der Cloud" [33] dargestellt.

6 Evaluation

Um die in Kapitel 4 beschriebenen Konzepte zu evaluieren, wurde ein Prototyp entwickelt. Mit dem entwickelten Prototyp können die entsprechenden TOSCA Service-Templates aus den Chef- und Juju-Artefakten erzeugt werden. In Kapitel 5 werden der Entwurf und die Implementierung dieses Prototyps dargestellt und es wurde gezeigt, dass der Prototyp drei wichtigen Aufgaben erledigen kann. Zu den konkreten Aufgaben gehören die Erzeugung von TOSCA Node-Types aus Chef-Cookbooks oder Juju-Charms, die Erzeugung von TOSCA Relationship-Types aus zwei Juju-Charms und die Erzeugung von TOSCA Service-Templates aus diesen generierten Node-Types und Relationship-Types. Im Folgenden werden die Konzepte mittels des entwickelten Prototyps anhand eines Beispiels überprüft. Die Aufgabe der Evaluierung besteht aus zwei Teilen: (1) Zuerst werden die Funktionalität des Prototyps getestet. Das heißt, dass die CSARs durch den entwickelten Prototyp automatisch generiert werden können; (2) dann werden die generierten CSARs validiert.

6.1 Test der Funktionalität des Prototyps

Zum Testen der Funktionalität des entwickelten Prototyps wird in Abbildung 6.1 ein Beispiel für ein TOSCA Service-Template namens "WordPress-Service" gegeben. Dieses Service-Template besteht aus drei Knoten "WordPress", "MySQL" und "Apache2" und zwei Beziehungen "ConnectsTo" und "HostedOn".

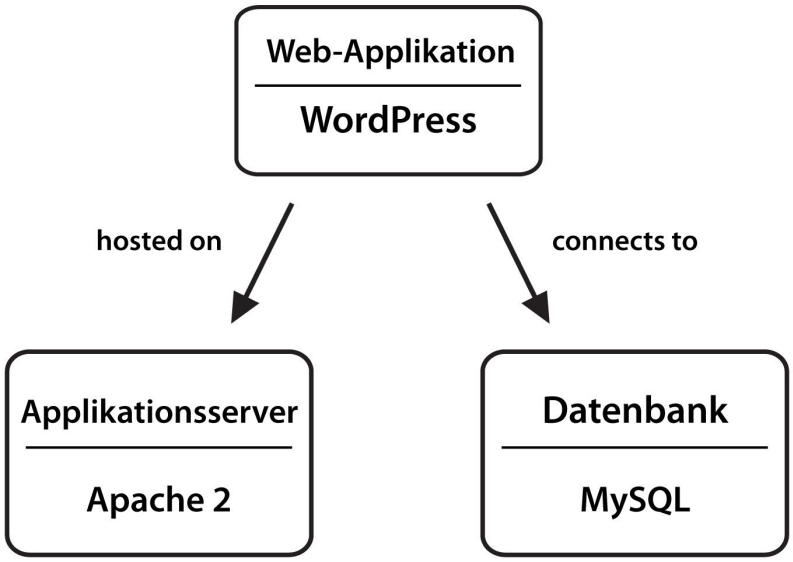

Abbildung 6.1: Beispiel für das TOSCA Service-Template "WordPress-Service"

Zuerst wird mittels der Komponente "Node-Type-Generator" des Prototyps ein Node-Type für die Applikationsserver "Apache 2" aus einem entsprechenden Chef-Cookbook automatisch generiert. In diesem Fall ist die Eingabe des Prototyps die URL, die sich mit dem Repository für das Chef-Cookbook "apache 2" verbindet. Auf gleiche automatische Art und Weise können mithilfe des Prototyps die Node-Types für die Applikation "WordPress" und die Datenbank "MySQL" aus den entsprechenden Juju-Charms automatisch erzeugt werden. Hierbei sind die Eingaben die URLs, die auf das Repository für Juju-Charms "WordPress" und "MySQL" verweisen. Das Ziel für die Verwendung von verschiedenen Typen der

Artefakte wie Chef-Cookbooks und Juju-Charms ist, zu ermöglichen, dass verschiedene Artefakttypen miteinander kombiniert werden können. Für die Beziehung "ConnectsTo" zwischen der Applikation "WordPress" und des Datenbankservers "MySQL" kann die Komponente "Relationship-Type-Generator" des Prototyps ein entsprechender Relationship-Type aus den Juju-Charms "WordPress" und "MySQL" automatisch generieren werden. Der einzige Teil für das Service-Template, den ich manuell implementiert habe, ist der Relationship-Type für die Beziehung "HostedOn". Das entsprechende TOSCA-Definitions-Dokument für diesen Relationship-Type enthält zwei Elemente *ValidSource* und *ValidTarget*. Sie spezifizieren die Typen der Quelle und des Zieles der Beziehung "HostedOn". In unserem Fall spezifiziert das Element *ValidSource* den Node-Type "WordPress", während das Element *ValidTarget* den Node-Type "Apache 2" spezifiziert. Schließlich kann die Komponente "Service-Template-Generator" des Prototyps durch diese generierten Node-Types und Relationship-Types die entsprechende TOSCA CSAR-Datei für das Service-Template "WordPress-Service" automatisch generieren. Neben den Node-Types und Relationship-Types sowie dem Service-Template selbst enthält diese CSAR-Datei auch alle notwendigen Artefakte (Cookbooks und Recipes von Chef sowie Charms und Hooks von Juju).

6.2 Validieren von CSARs

In Unterkapitel 6.1 wurde dargestellt, dass durch den entwickelten Prototyp die TOSCA CSAR-Datei für das Service-Template "WordPress-Service" automatisch generiert werden kann. In dieser CSAR-Datei lassen sich alle entsprechenden Chef- und Juju-Artefakte verwenden. In diesem Unterkapitel werden die Korrektheit und die Gültigkeit dieser CSAR-Datei geprüft, damit basierend auf diesem Service-Template das Deployment und Management der konkreten Service-Instanzen in einer Cloud-Umgebung verwirklicht werden kann. Die Voraussetzung dafür ist, dass diese Cloud-Umgebung eine dem TOSCA-Standard konforme Laufzeitumgebung zur Verfügung stellen muss. Zum Validieren dieser CSAR-Datei wurde das Werkzeug "Winery" [49] verwendet. Winery [50] ist eine webbasierte Umgebung und wird verwendet, um TOSCA-Topologien und Pläne grafisch zu modellieren. Diese Umgebung enthält eine Komponente zur Verwaltung von Types und Templates. Mit dieser Komponente können alle in der TOSCA-Spezifikation definierten Elemente erstellt und verarbeitet werden. Alle Informationen werden in einem Repository gespeichert. Dieses Repository ist dafür verantwortlich, CSARs zu importieren und zu exportieren.

Zuerst wurde diese CSAR-Datei zum Werkzeug "Winery" importiert. Nachdem der Import der CSAR-Datei erfolgreich durchgeführt wurde, haben wir als Ergebnis einen Graphen bekommen. Dieser Graph, der in Abbildung 6.2 gezeigt wird, entspricht dem Graphen, der in Abbildung 6.1 dargestellt wird. Daraus ergibt sich, dass diese CSAR-Datei korrekt ist.

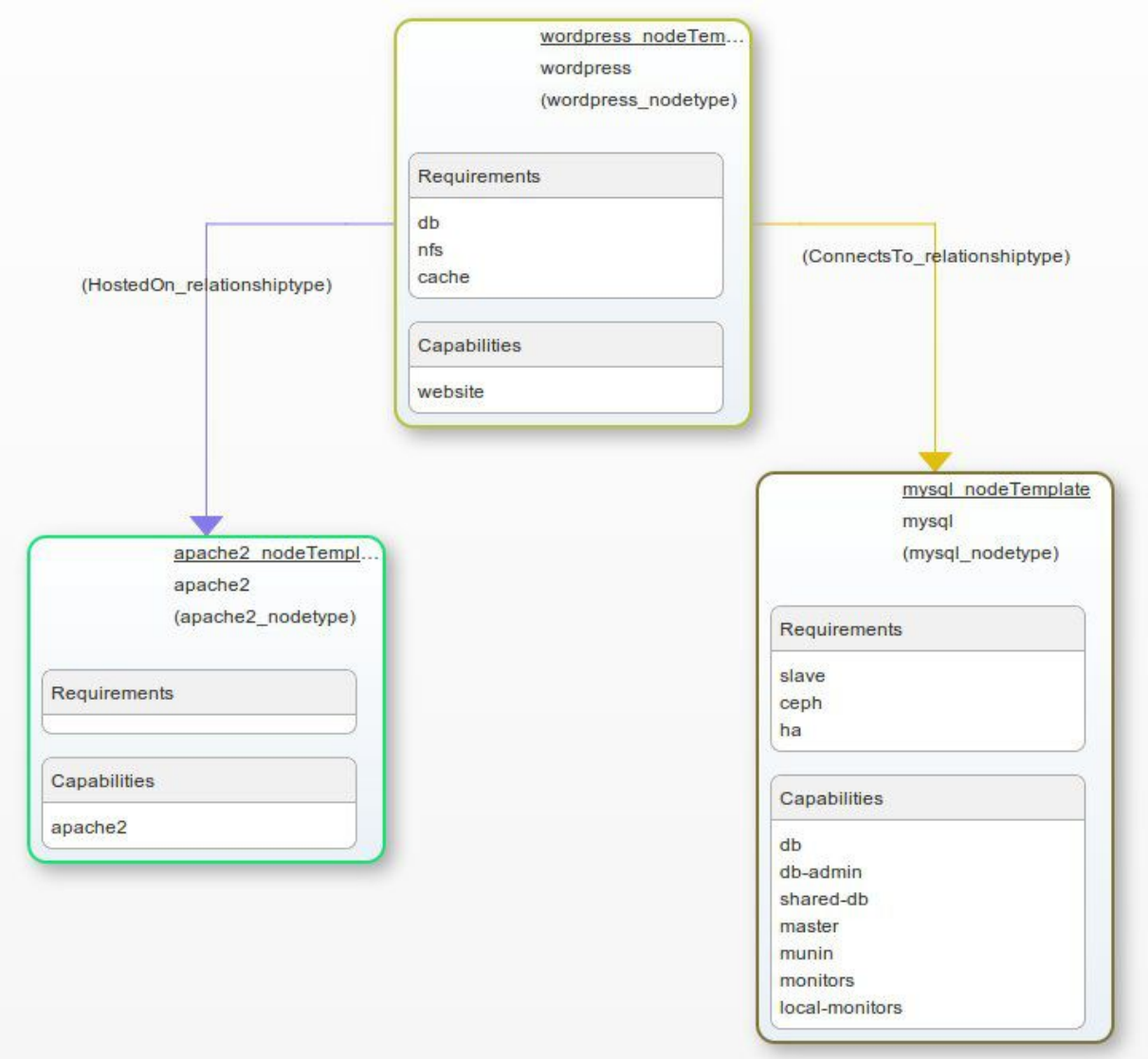

Abbildung 6.2: Ergebnis fürs Validieren der CSAR "WordPress-Service"

7 Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde ein automatisches Verfahren dargestellt, mit dem die TOSCA Service-Template basierend auf existierenden von den hinter Juju und Chef stehenden DevOps-Communities veröffentlichten Artefakten erzeugt werden können. Diese generierten TOSCA Service-Templates, die diese Artefakte verwenden, können von jeder TOSCA-konformen Laufzeitumgebung verarbeitet werden.

Im Grundlagenkapitel wurden die DevOps-Ansätze (Chef und Juju) und deren Verwendung zur Realisierung des Deployments und Managements dargestellt. Außerdem wurde in Abbildung 2.6 gezeigt, dass Topology-Template und Pläne die zentralen Elemente eines TOSCA Service-Template sind. Ein Topology-Template besteht aus Node-Templates und Relationship-Templates. Diese Arbeit beschäftigte sich ausschließlich mit der Generierung von Service-Templates ohne Pläne. Dazu gehören (1) die Generierung von Node-Types, die den Typ eines oder mehrerer Node-Templates definieren, (2) die Generierung von Relationship-Types, die den Typ eines oder mehrerer Relationship-Templates definieren, und (3) die Generierung des Topology-Template. In Kapitel 3 wurde besprochen, dass die verschiedenen DevOps-Ansätze (Juju und Chef) abstrahiert werden müssen, um ihre Topologie abstrakt zu modellieren. Das Ergebnis der Abstraktion ist, dass für jede Art dieser Ansätze ein entsprechendes Topologie-Modell generiert wurde. Außerdem wurde als Beispiel die Modell-Transformation von Juju nach TOSCA sowie von Chef nach TOSCA erläutert. In Kapitel 4 wurden die Konzepte für das automatische Verfahren erläutert und es wurde beschrieben, wie ein Node-Type aus einem Chef-Cookbook und ein Relationship-Type aus zwei Juju-Charms erzeugt werden. Außerdem wurde in Unterkapitel 4.4 dargestellt, wie ein Service-Template durch Orchestrierung dieser generierten Node-Types und Relationship-Types erstellt wird. Um dieses automatische Verfahren zu evaluieren, wurde in Kapitel 5 ein Prototyp entwickelt, mit dem sich alle öffentlich zugänglichen Chef- und Juju-Artefakte in CSARs verwenden lassen. Es wurde dargestellt, welche Funktionalitäten der Prototyp besitzt und mit welchen wichtigen Komponenten der Prototyp aufgebaut wird. Außerdem wurde in Unterkapitel 5.3 die konkrete Implementierung des Prototyps besprochen. In Kapitel 6 wurde gezeigt, dass die CSAR-Datei durch den entwickelten Prototyp automatisch generiert werden kann. Außerdem wurde diese CSAR-Datei durch das Werkzeug "Winery" [50] validiert.

Eine Möglichkeit für zukünftige Arbeiten ist die Erzeugung von entsprechenden Plänen, die verwendet werden, um den Lebenszyklus eines Cloud-Service oder einer Cloud-Anwendung zu verwalten. Diese Pläne sollten unter Verwendung von existierenden Workflow-Sprachen wie BPMN [27] oder BPEL [28] zwischen verschiedenen Cloud-Umgebungen und Cloud-Anbietern portabel sein. Dies ermöglicht es, dass das Management von Cloud-Services wiederverwendbar und portabel ist. Außerdem bezog sich diese Arbeit stark auf Chef und Juju. Allerdings sind die in dieser Arbeit beschriebenen Konzepte nicht Chef- oder Juju-spezifisch. Folglich kann das in dieser Arbeit beschriebene Verfahren auf weitere ähnliche Werkzeuge und Artefakte übertragen werden. Beispielsweise können auch die Konzepte bezüglich Chef für andere Konfigurationsmanagementwerkzeuge wie z.B. Puppet [15] und CFEngine [34] implementiert werden, da sie eine sehr ähnliche Architektur besitzen.

Literaturverzeichnis

Alle Weblinks wurden das letzte Mal am 01.12.2013 geprüft.

[1] Wettinger, Kopp und Leymann: Improving Portability of Cloud Service Topology Models Relying on Script-Based Deployment. In: CEUR Workshop Proceedings; Online Proceedings for Scientific Workshops. (2013)

[2] Leymann: Cloud Computing. it – Information Technology, 53(4). (2011)

[3] Mell und Grance: The NIST Definition of Cloud Computing. National Institute of Standards and Technology. (2011)

[4] Günther, Haupt und Splieth: Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures. Technical report, Very Large Business Applications Lab Magdeburg, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg. (2010)

[5] Benjamin, Waldemar, Christian, Philipp und Schahram: Winds of Change: From Vendor Lock-In to the Meta Cloud. Vienna University of Technology. (2013)

[6] Humble und Farley: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley Professional. (2010)

[7] Humble und Molesky: Why Enterprises Must Adopt Devops to Enable Continuous Delivery. Cutter IT Journal, 24(8):6. (2011)

[8] Shamow: Devops at Advance Internet: How We Got in the Door. IT Journal, page 14. (2011)

[9] Nelson-Smith: Test-Driven Infrastructure with Chef. O'Reilly Media, Inc. (2011)

[10] Delaet, Joosen und Vanbrabant: A Survey of System Configuration Tools. In Proceedings of the 24th Large Installations Systems Administration (LISA) conference. (2010)

[11] Smith: Hype Cycle for Cloud Computing. (2011)

[12] Leymann: Cloud Computing: The Next Revolution in IT. In Photogrammetric Week '09. Wichmann Verlag. (2009)

[13] Vaquero, Rodero-Merino, Caceres und Lindner: A Break in the Clouds: Towards a Cloud Definition. ACM SIGCOMM Computer Communication Review, 39(1):50–55. (2008)

[14] Chef Cookbooks. http://community.opscode.com/cookbooks

[15] Puppet Webseite. https://puppetlabs.com/

[16] Juju Charm Browser. http://jujucharms.com/

[17] TOSCA Specification, Version 1.0 Committee Specification 01. 18 March 2013. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html

[18] Chef Documentation: http://docs.opscode.com/

[19] Git - Verteiltes Versionsverwaltungssystem: http://git-scm.com/

[20] Wettinger, Behrendt, Binz, Breitenbücher, Breiter, Leymann, Moser, Schwertle, Spatzier: Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA. In: Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER). (2013)

[21] Delaet und Joosen: PoDIM: A language for high-level configuration management. In Proceedings of the Large Installations Systems Administration (LISA) Conference, Berkeley, CA. (2007)

[22] SugarCRM Webseite. http://www.sugarcrm.com/

[23] Juju Documentation. https://juju.ubuntu.com/docs/

[24] YAML Webseite. http://www.yaml.org/

[25] WordPress Webseite. http://wordpress.com/

[26] MySQL Webseite. http://www.mysql.de/

[27] Business Process Model and Notation (BPMN) Version 2.0, Object Management Group specification: http://www.bpmn.org/

[28] Web Services Business Process Execution Language (BPEL) Version 2.0, OASIS specification: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

[29] Binz, Breiter, Leymann und Spatzier: Portable Cloud Services Using TOSCA. Internet Computing, IEEE, 16(3):80–85. (2012)

[30] Leymann, Fehling, Mietzner, Nowak und Dustdar: Moving Applications to the Cloud: An Approach Based on Application Model Enrichment. International Journal of Cooperative Information Systems, 20(3):307. (2011)

[31] Binz, Leymann und Schumm: CMotion: A Framework for Migration of Applications into and between Clouds. In 2011 IEEE International Conference on Service-Oriented Computing and Applications. IEEE. (2011)

[32] Breitenbücher, Binz, Kopp und Leymann: Pattern-Based Runtime Management of Composite Cloud Applications. In Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER). (2013)

[33] Zhang: Studienarbeit: Vorlagen für das Deployment von Services und Applikationen in der Cloud. Betreuer: Wettinger. (2013)

[34] JAXB Reference Implementation. https://jaxb.java.net/

[35] Ruby Webseite. https://www.ruby-lang.org/en/

[36] JRuby Webseite. http://jruby.org/

[37] JRuby Wikipedia. http://de.wikipedia.org/wiki/JRuby

[38] Dom4j Webseite. http://dom4j.sourceforge.net/

[39] Dom4j Wikipedia. http://de.wikipedia.org/wiki/Dom4j

[40] UUID Wikipedia. http://de.wikipedia.org/wiki/Universally_Unique_Identifier

[41] Snakeyaml Webseite. http://code.google.com/p/snakeyaml/

[42] JAXB Wikipedia. http://de.wikipedia.org/wiki/Java_Architecture_for_XML_Binding

[43] CFEngine Webseite. http://cfengine.com/

[44] Baun, Kunze und Tai: Cloud Computing - Web-basierte dynamische IT-Services, 2. Aufl. ed., Heidelberg, Dordrecht, London, New York: Springer-Verlag. (2011)

[45] Kopp, Binz, Breitenbücher, Leymann: BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications. In: Mendling, Jan (Hrsg); Weidlich, Matthias (Hrsg): 4th International Workshop on the Business Process Model and Notation. (2012)

[46] GitHub Webseite. https://github.com/

[47] Bazaar Webseite. http://bazaar.canonical.com/

[48] Launchpad Webseite. https://launchpad.net/

[49] Winery Webseite. http://dev.winery.opentosca.org:8080/winery/servicetemplates/

[50] Kopp, Binz, Breitenbücher, und Leymann: Winey - A Modeling Tool for TOSCA-based Cloud Applications. Springer-Verlag. (2013)

[51] Apache Webseite. http://www.apache.org/

[52] Chef Cookbook Apache2. http://community.opscode.com/cookbooks/apache2

[53] Chef Cookbook WordPress. http://community.opscode.com/cookbooks/wordpress

[54] Amazon Web Services Webseite. http://aws.amazon.com/de/

[55] Google App Engine Webseite. https://developers.google.com/appengine/

[56] Definition der Orchestration. http://en.wikipedia.org/wiki/Service_orchestration

[57] Amazon Virtual Private Cloud Webseite. http://aws.amazon.com/de/vpc/

[58] OpenStack Webseite. http://www.openstack.org/

[59] Rackspace Webseite. http://www.rackspace.com/

[60] Google Compute Engine Webseite. https://developers.google.com/compute/

[61] Windows Azure Webseite. http://www.windowsazure.com/de-de/

[62] Chef Cookbook User. http://community.opscode.com/cookbooks/user

[63] JavaScript Object Notation. http://de.wikipedia.org/wiki/JavaScript_Object_Notation

[64] TOSCA Implementer's Recommendations for Interoperable TOSCA Implementations, Version 1.0 Interoperability Subcommittee, Working Draft 01, Rev. 05, 20 May 2013. https://www.oasis-open.org/committees/document.php?document_id=49302&wg_abbrev=tosca

[65] TOSCA Primer, Version 1.0. Committee Note Draft (CND) 01, Working Draft 07, Revision 01, 08 February 2013. https://www.oasis-open.org/committees/document.php?document_id=48201&wg_abbrev=tosca

Anhang 1

Node-Type-Properties-Dokument für den Node-Type "MySQL" aus dem Cookbook "mysql"

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns="http://community.opscode.com/cookbooks/mysql/nodetype
      _properties" targetNamespace="http://community.opscode.com/
      cookbooks/mysql/nodetype_properties">

  <xs:complexType name="t-mysql-properties">
    <xs:sequence>
      <xs:element name="mysql/server_root_password" type="string"
        default="randomly generated">
        <xs:annotation>
          <xs:documentation xml:lang="en">Randomly generated password
            for the mysqld root user</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="mysql/bind_address" type="string"
        default="ipaddress">
        <xs:annotation>
          <xs:documentation xml:lang="en">Address that mysqld should
            listen on</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="mysql/data_dir" type="string"
        default="/var/lib/mysql">
        <xs:annotation>
          <xs:documentation xml:lang="en">Location of mysql
            databases</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="mysql/conf_dir" type="string"
        default="/etc/mysql">
        <xs:annotation>
          <xs:documentation xml:lang="en">Location of mysql conf
            files</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="mysql/ec2_path" type="string"
        default="/mnt/mysql">
        <xs:annotation>
          <xs:documentation xml:lang="en">Location of mysql directory
```

```
      on EC2 instance EBS volumes</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:element name="mysql/reload_action" type="string"
  default="reload">
  <xs:annotation>
    <xs:documentation xml:lang="en">Action to take when mysql
      conf files are modified</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:element name="mysql/tunable" type="hash">
  <xs:annotation>
    <xs:documentation xml:lang="en">Hash of MySQL tunable
      attributes</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:element name="mysql/tunable/key_buffer" type="string"
  default="250M"/>
<xs:element name="mysql/tunable/max_connections" type="string"
  default="800"/>
<xs:element name="mysql/tunable/wait_timeout" type="string"
  default="180"/>
<xs:element name="mysql/tunable/net_read_timeout" type="string"
  default="30"/>
<xs:element name="mysql/tunable/net_write_timeout" type="string"
  default="30"/>
<xs:element name="mysql/tunable/back_log" type="string"
  default="128"/>
<xs:element name="mysql/tunable/table_cache" type="string"
  default="128"/>
<xs:element name="mysql/tunable/table_open_cache" type="string"
  default="128"/>
<xs:element name="mysql/tunable/max_heap_table_size"
  type="string" default="32M"/>
<xs:element name="mysql/tunable/expire_logs_days" type="string"
  default="10"/>
<xs:element name="mysql/tunable/max_binlog_size" type="string"
  default="100M"/>
<xs:element name="mysql/client" type="hash">
  <xs:annotation>
    <xs:documentation xml:lang="en">Hash of MySQL client
      attributes</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:element name="mysql/client/version" type="string"
  default="6.0.2"/>
<xs:element name="mysql/client/arch" type="string"
```

```
        default="win32"/>
      <xs:element name="mysql/client/package_file" type="string"
        default="mysql-connector-c-6.0.2-win32.msi"/>
      <xs:element name="mysql/client/url" type="string"
        default="http://www.mysql.com/get/Downloads/Connector-C/
        mysql-connector-c-6.0.2-win32.msi/from/http://
        mysql.mirrors.pair.com/"/>
      <xs:element name="mysql/client/package_name" type="string"
        default="MySQL Connector C 6.0.2"/>
      <xs:element name="mysql/client/basedir" type="string"
        default="C:\Program Files (x86)\MySQL\Connector C 6.0.2"/>
      <xs:element name="mysql/client/lib_dir" type="string"
        default="C:\Program Files (x86)\MySQL\
        Connector C 6.0.2\lib\opt"/>
      <xs:element name="mysql/client/bin_dir" type="string"
        default="C:\Program Files (x86)\MySQL\Connector C 6.0.2\bin"/>
      <xs:element name="mysql/client/ruby_dir" type="string"
        default="system ruby"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="mysql-properties" type="t-mysql-properties"/>

</xs:schema>
```

Anhang 2

Die Datei "metaparser.rb"

```
require './lib/chef/metadata'
require 'java'
java_import 'org.tosca.meta.CookbookMetamodel'
java_import 'org.tosca.meta.cookbook.Recipe'
java_import 'org.tosca.meta.cookbook.Dependency'
java_import 'org.tosca.meta.cookbook.Attribute'

metadata_file=$pathname
puts metadata_file
cmm=CookbookMetamodel.new
metadata = Chef::Cookbook::Metadata.new
metadata.from_file(metadata_file)

name="#{metadata.name}"
cmm.setName(name.to_java)

description="#{metadata.description}"
cmm.setDescription(description.to_java)

metadata.recipes.each { |name, description|
      name="#{name}"
      description=" #{description}"
      recipe=Recipe.new
      recipe.setName(name)
      recipe.setDescription(description)
      cmm.getRecipes().add(recipe)
}

metadata.platforms.each { |platform, version|
      platform="#{platform}"
      cmm.getPlatforms().add(platform)
}

metadata.dependencies.each { |cookbook, version|
      cookbook="#{cookbook}"
      version=" #{version}"
      dependency=Dependency.new
      dependency.setCookbook(cookbook)
      dependency.setVersion(version)
      cmm.getDependencies().add(dependency)
}
```

```
metadata.attributes.each { |name, options|
      name="#{name}"
      attribute=Attribute.new
      attribute.setName(name)
      options.each{|optionname, option|
            optionname="#{optionname} "
            option="#{option}"
            if /display_name /=~optionname then
                  attribute.setDisplay_name(option)
            end
            if /description/=~optionname then
                  attribute.setDescription(option)
            end
            if /default /=~optionname then
                  attribute.setDefault_value(option)
            end
            if /choice /=~optionname then
                  attribute.setChoice(option)
            end
            if /calculated/=~optionname then
                  attribute.setCalculated(option)
            end
            if /type /=~optionname then
                  attribute.setType(option)
            end
            if /required /=~optionname then
                  attribute.setRequired(option)
            end
            if /recipes/=~optionname then
                  attribute.setRecipes(option)
            end
      }
      cmm.getAttributes().add(attribute)
}

return cmm
```

Anhang 3

Die Datei "ScriptArtifact.xsd"

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema
      targetNamespace="http://docs.oasis-open.org/tosca/.../Artifacts"
      elementFormDefault="qualified"
      attributeFormDefault="unqualified"
       xmlns="http://docs.oasis-open.org/tosca/ns/2011/12/Artifacts"
       xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="ScriptArtifactProperties">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ScriptLanguage" type="xs:anyURI"/>
        <xs:element name="PrimaryScript" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Anhang 4

Die Datei "ChefArtifact.xsd"

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.example.com/ChefArtifacts"
  elementFormDefault="qualified" attributeFormDefault="unqualified"
  xmlns="http://www.example.com/ChefArtifacts"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="documentation" type="tDocumentation"/>
  <xs:complexType name="tDocumentation" mixed="true">
    <xs:sequence>
      <xs:any processContents="lax" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="source" type="xs:anyURI"/>
  </xs:complexType>
  <xs:complexType name="tExtensibleElements">
    <xs:sequence>
      <xs:element ref="documentation" minOccurs="0"
        maxOccurs="unbounded"/>
      <xs:any namespace="##other" processContents="lax" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>
  <xs:element name="ChefArtifactProperties">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="tChefArtifactProperties"/>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="tChefArtifactProperties">
    <xs:complexContent>
      <xs:extension base="tExtensibleElements">
        <xs:sequence>
          <xs:element name="Cookbooks">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="Cookbook" maxOccurs="unbounded">
                  <xs:complexType>
                    <xs:attribute name="name" type="xs:string"
                      use="required"/>
                    <xs:attribute name="location" type="xs:anyURI"
                      use="required"/>
```

```
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Roles" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Role" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string"
            use="required"/>
          <xs:attribute name="location" type="xs:anyURI"
            use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Mappings" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="PropertyMapping"
        type="tPropertyMapping" minOccurs="0"
        maxOccurs="unbounded"/>
      <xs:element name="SourcePropertyMapping"
        type="tPropertyMapping" minOccurs="0"
        maxOccurs="unbounded"/>
      <xs:element name="TargetPropertyMapping"
        type="tPropertyMapping" minOccurs="0"
        maxOccurs="unbounded"/>
      <xs:element name="InputParameterMapping"
        type="tParameterMapping" minOccurs="0"
        maxOccurs="unbounded"/>
      <xs:element name="OutputParameterMapping"
        type="tParameterMapping" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="RunList">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Include" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="RunListEntry"
```

```
                        type="tRunListEntry" minOccurs="1"
                        maxOccurs="unbounded"/>
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
                <xs:element name="Exclude" minOccurs="0">
                  <xs:complexType>
                    <xs:sequence>
                      <xs:element name="RunListEntry"
                        type="tRunListEntry" minOccurs="1"
                        maxOccurs="unbounded"/>
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="tRunListEntry">
    <xs:attribute name="cookbookName" type="xs:string"/>
    <xs:attribute name="recipeName" type="xs:string"/>
    <xs:attribute name="roleName" type="xs:string"/>
  </xs:complexType>
  <xs:complexType name="tParameterMapping">
     <xs:attribute name="parameterName" type="xs:string"
       use="required"/>
     <xs:attribute name="cookbookAttribute" type="xs:string"
       use="required"/>
  </xs:complexType>
  <xs:complexType name="tPropertyMapping">
    <xs:attribute name="propertyPath" type="xs:string" use="required"/>
     <xs:attribute name="cookbookAttribute" type="xs:string"
       use="required"/>
    <xs:attribute name="mode" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="input"/>
          <xs:enumeration value="output"/>
          <xs:enumeration value="input-output"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:schema>
```

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.

Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.

Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.

Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.

Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, den 8. Januar 2014 _____