Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Extraktion und Mapping der in Chef Kochbüchern beschriebenen Deployment Architekturen auf ein generisches Architekturmodell

Markus Schütterle

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	Lukas Harzenetter, M.Sc.
Beginn am:	09.11.2018
Beendet am:	09.05.2019

Kurzfassung

Zur Automatisierung der Bereitstellung von Anwendungen sind in den vergangenen Jahren verschiedene Technologien entstanden. Dazu gehören Werkzeuge zum automatischen Konfigurationsmanagement, wie Chef, Juju, Puppet oder Docker. Aufgrund starker Nutzung dieser Technologien existieren viele öffentliche Repositories, die ausführbare Artefakte enthalten und mit diesen Technologien verwendet werden können. Vor der Bereitstellung von Anwendungen müssen die Komponenten einer geplanten Anwendungsarchitektur auf Kompatibilität geprüft werden. Die Überprüfung dieser erfolgt aktuell manuell. Um die Kompatibilitätsüberprüfung zu automatisieren, ist Wissen über Softwarekomponenten und Plattformen sowie deren Beziehungen untereinander notwendig. Da Artefakte zum Konfigurationsmanagement für die automatische Bereitstellung und Konfiguration von Anwendungen konzipiert sind, ist das Wissen über kompatible Komponenten in Artefakten in öffentlichen Repositories vorhanden. Zur Verwendung dieses Wissens müssen die öffentlich verfügbaren Artefakte abgerufen und die enthaltenen Deployment Architekturen in ein wiederverwendbares Architekturmodell übersetzt werden.

Die vorliegende Arbeit bezieht sich auf die Technologie Chef. Chef verwendet zum automatischen Konfigurationsmanagement sogenannte Kochbücher als Artefakte. Um das Wissen abzurufen, wird ein Chef Kochbuch Crawler vorgestellt, mit dem Chef Kochbücher aus dem Chef Supermarket heruntergeladen und lokal gespeichert werden. Zur Generalisierung der extrahierten Deployment Architekturen, wird zur Abbildung dieser ein generisches Architekturmodell vorgestellt. Für die Extraktion der Deployment Architekturen aus Chef Kochbüchern, wird ein spezialisierter Chef Kochbuch Compiler vorgestellt. Dieser parst die Dateien eines Chef Kochbuches, in einer definierten Reihenfolge und übersetzt das Kochbuch in die enthaltenen Deployment Architekturen. Dabei werden Abhängigkeiten zu weiteren Kochbüchern durch Rekursion aufgelöst.

Um die Machbarkeit des erläuterten Ansatzes zu beweisen, wird dieser als Prototyp in die Eclipse Winery, einem webbasierten Modellierungswerkzeug für TOSCA Elemente, implementiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

API Application Programming Interface

BNF Backus-Naur-Form

ChefDK Chef Developement Kit

DMMN Declarative Application Management Modelling and Notation

DSL Domain specific language

ECAs Environment-centric Artefakte

ETGs Enterprise Topology Graphen

NCAs Node-centric Artefakte

REST Representational State Transfer

TOSCA Topology and Orchestration Specification for Cloud Applications

VM virtuelle Maschine

XML Extensible Markup Language

YAML YAML Ain't Markup Language

1 Einleitung

In diesem Kapitel wird die Motivation und das Ziel der vorliegenden Arbeit in Abschnitt 1.1 deutlich gemacht. Nachfolgend wird in Abschnitt 1.2 der Aufbau dieser Arbeit umrissen. Zum Abschluss dieses Kapitels wird in Abschnitt 1.3 ein laufendes Beispiel vorgestellt, das zur Erläuterung der vorgeschlagenen Ansätze in dieser Arbeit verwendet wird.

1.1 Motivation

Da Cloud Computing für die meisten Menschen auf der Erde nicht mehr wegzudenken ist, wird die Technologie immer bedeutender. Dass Anbieter von Anwendungen und Diensten immer mehr auf Cloud Computing setzen, liegt vor allem daran, dass sich der Betrieb der Informationstechnik deutlich flexibler gestalten lässt [Ley09]. Da Kunden und Nutzer von solchen Diensten und Anwendungen erwarten, dass neue Funktionen und Fehler so schnell wie möglich verfügbar und behoben sind, ist es für Anbieter überlebensnotwendig, Updates kontinuierlich und so schnell wie möglich bereitzustellen. Um wettbewerbsfähig zu bleiben und den Erwartungen der Nutzer gerecht zu werden, müssen neue Releases in immer kürzeren Zyklen bereitgestellt werden. Eine häufige und kontinuierliche Bereitstellung lässt sich nur durch eine weitgehende Automatisierung der beteiligten Prozesse realisieren [HF10].

Aufgrund unterschiedlicher Ziele, Prozessen und Denkweisen von Menschen steht der kontinuierlichen Auslieferung häufig die Barriere zwischen Entwicklern und Betriebspersonal im Weg [Hüt12]. So wollen beispielsweise Entwickler Änderungen so schnell wie möglich ausliefern, wohingegen das Betriebspersonal den stabilen Betrieb gewährleisten möchte [Hüt12]. Um die Barriere zwischen Entwicklern und Betriebspersonal zu beseitigen, wurde das DevOps-Paradigma entwickelt [HM11; WBL14a]. Durch den DevOps-Ansatz verspricht man sich bei der kontinuierlichen Bereitstellung von Software schnellere und häufigere Releases. Um eine schnelle kontinuierliche Auslieferung zu ermöglichen, ist eine durchgängige Automation sowie eine enge Zusammenarbeit von Entwicklern und Betriebspersonal notwendig. Zur Automatisierung gibt es Werkzeuge, die diesen Prozess unterstützen.

Um die Bereitstellung von Cloud-Anwendungen zu automatisieren, gibt es heutzutage eine Vielfalt an DevOps-Werkzeugen, Diensten und wiederverwendbaren Artefakten. Unter diesen Technologien gibt es unter anderem verschiedene Werkzeuge zur automatisierten Konfiguration von Netzwerkressourcen. Bekannte Beispiele zur Automatisierung und Unterstützung der Konfiguration sind Ansible[1], Chef[2],

[1] https://www.ansible.com/
[2] https://www.chef.io/

Docker[3], Juju[4] und Puppet[5]. Aufgrund intensiver Nutzung dieser Technologien sind öffentliche Repositories entstanden, die ausführbare und häufig verwendete Artefakte enthalten. Diese können von den genannten Technologien zum Bereitstellen der gewünschten Anwendungen wiederverwendet werden. Chef stellt mit dem Chef Supermarket[6] ein öffentliches Repository zur Verfügung. Dieses beinhaltet eine Vielzahl von Kochbüchern, die für eigene Chef Projekte verwendet werden können. Weitere Beispiele sind Puppetforge[7] für Puppet Manifeste oder Ansible Galaxy[8] für Ansible Playbooks.

Aktuell muss bei der Planung von Anwendungsarchitekturen manuell überprüft werden, ob die Versionen der verwendeten Softwarekomponenten miteinander kompatibel sind. Dabei stellt sich zum Beispiel die Frage, ob eine bestimmte Java Version auf einem gegebenen Ubuntu Betriebssystem verfügbar ist. Konkret stellt sich dabei beispielsweise die Frage, wie Java 5 auf Ubuntu 18.04 läuft. Als Domänenexperte weiß man, dass das nicht möglich ist. Die Vision ist eine automatisierte Aussage über die Kompatibilität von verwendeten Softwarekomponenten treffen zu können. Um dies zu automatisieren, ist Wissen über kompatible Softwarekomponenten notwendig. Informationen dazu finden sich oftmals in der Dokumentation der Softwarekomponenten. Ein möglicher Ansatz ist, die Informationen von den entsprechenden Internetquellen zu crawlen. Da die genannten DevOps-Artefakte für die automatische Konfiguration von Anwendungen und Services zusammen mit ihren Abhängigkeiten konzipiert sind, ist das Wissen über kompatible Softwarekomponenten auch in öffentlichen Repositories mit lauffähigen DevOps-Artefakten vorhanden. Um die vorhanden Informationen aus öffentlichen Repositories zu verwenden, müssen die in den Artefakten enthaltenen Deployment Architekturen extrahiert und auf ein Architekturmodell abgebildet werden, mit dem sich die Kompatibilitätsüberprüfung von geplanten Architekturen automatisieren lässt.

Der Aufbau von DevOps-Artefakten unterschiedlicher Technologien unterscheidet sich. Der Inhalt dieser Arbeit bezieht sich auf die Technologie Chef. Zur Automatisierung der Konfiguration von Infrastruktur verwendet Chef sogenannte Kochbücher. Da es zu diesem Zeitpunkt keinen Ansatz dazu gibt, wird ein Konzept benötigt, welches die verwendeten Softwareversionen und Abhängigkeiten aus einem Chef Kochbuch extrahiert und auf ein Architekturmodell abbildet. In dieser Arbeit wird ein solcher Ansatz vorgestellt, die enthaltene Deployment Architekturen aus Chef Kochbüchern zu extrahieren und auf ein Architekturmodell abzubilden. Damit wird die Grundlage geschaffen, die Kompatibilitätsüberprüfung der verwendeten Softwareversionen einer geplanten Architektur zu automatisieren. Um aus Kochbüchern die verwendeten Konfigurationen zu extrahieren und diese in einer Datenbank abzulegen, werden diese aus öffentlichen Repositories, wie dem Chef Supermarket, abgerufen. Dadurch entsteht eine Datenbank mit Wissen über Softwareversionen mit ihren Abhängigkeiten untereinander. Die extrahierten Informationen dienen als Basis für eine Automatisierung der Voraussage, ob die Softwareversionen einer geplanten Architektur miteinander kompatibel sind.

[3] https://www.docker.com/
[4] https://jujucharms.com/
[5] https://puppet.com/de
[6] https://supermarket.chef.io/
[7] https://forge.puppet.com/
[8] https://galaxy.ansible.com/

1.2 Struktur

Die vorliegende Arbeit erläutert in Kapitel 2 zuerst Hintergrundinformationen und Grundlagen, die für das weitere Verständnis der Arbeit notwendig sind. Nachfolgend werden in Kapitel 3 verwandte Arbeiten zusammengefasst. In Kapitel 4 wird ein Ansatz vorgestellt, die Deployment Architekturen von Chef Kochbüchern auf ein Architekturmodell abzubilden. Dazu wird ein Ansatz, zum Crawlen von Kochbüchern aus öffentlichen Repositories und ein allgemeines Architekturmodell, vorgestellt. In Kapitel 5 werden die Details der Transformation erläutert. Anschließend wird in Kapitel 6 der in der Eclipse Winery [KBBL13] implementierte Prototyp des vorgestellten Ansatzes erläutert. In Kapitel 7 werden die Limitationen des vorgestellten Ansatzes erläutert. Die Arbeit endet in Kapitel 8 mit einer Zusammenfassung und einem Ausblick.

1.3 Laufendes Beispiel

Abbildung 1.1 zeigt die Anwendungstopologie, die während der Arbeit als laufendes Beispiel verwendet wird. Die Topologie besteht aus einer beispielhaften Applikation namens MyApp. Diese hängt von Java 8[9] ab und kann auf Ubuntu 16.04[10], Fedora 29[11] und weiteren Plattformen bereitgestellt werden. Damit die Konfiguration der Topologie nicht manuell erfolgen muss, wird sie mit Chef[12] automatisiert. Dazu wird ein Chef Kochbuch implementiert, mit dem die Konfigurationsaufgaben für die unterschiedlichen Plattformen ausgeführt werden können. Das entwickelte MyApp-Kochbuch enthält alle notwendigen Metadaten, Attribute und Rezepte für die Konfiguration. Da es für die Konfiguration von Java ein öffentlich verfügbares Kochbuch[13] gibt, wird im MyApp-Kochbuch eine Abhängigkeit zum Java-Kochbuch deklariert und das Java-Kochbuch wiederverwendet.

[9] https://www.java.com/de/
[10] https://www.ubuntu.com/
[11] https://getfedora.org/de/
[12] https://docs.chef.io/chef_overview.html
[13] https://github.com/sous-chefs/java

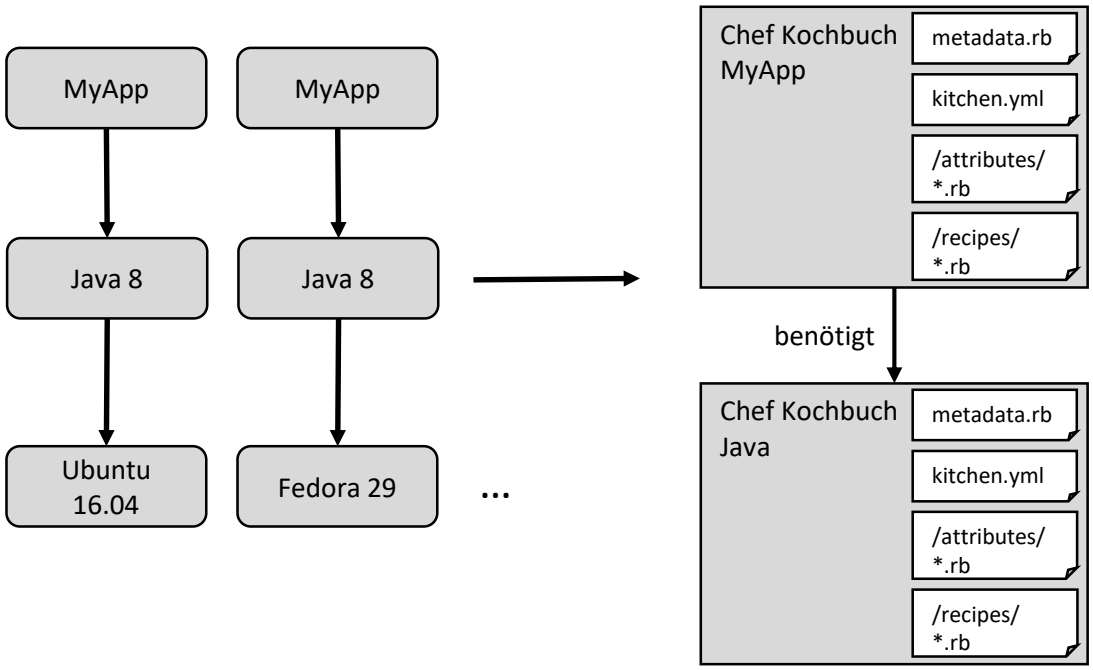

Abbildung 1.1: Architektur des laufenden Beispiels

2 Hintergrund und Grundlagen

In diesem Kapitel werden grundlegende Sachverhalte eingeführt, die für das Verstehen des Ansatzes und der Implementierung notwendig sind. Dabei werden das DevOps-Werkzeug Chef (Abschnitt 2.1) sowie das Open-Source Modellierungswerkzeug Eclipse Winery (Abschnitt 2.2) vorgestellt, in das der Prototyp integriert wird, vor. Zudem werden in Abschnitt 2.3 Grundlagen von Parsern eingeführt.

2.1 Chef

Chef[1] ist ein Automatisierungsunternehmen, mit dem gleichnamigen Produkt Chef[2]. Der Inhalt dieser Arbeit bezieht sich auf das Produkt Chef, mit dem das Unternehmen versucht Entwickler und Systemadministratoren zu vereinen. Das Produkt Chef ist eine Automatisierungsplattform zur automatischen Einrichtung von Betriebssystemen und Programmen, die auf Hardware in Rechenzentren laufen. Um sicherzustellen, dass die Konfigurationen in jeder Umgebung und in jeder Größenordnung konsistent angewendet werden kann, wird Infrastruktur mithilfe von Code deklariert. Unabhängig davon, ob in der Cloud, vor Ort oder in einer Hybridumgebung gearbeitet wird, lässt sich mithilfe von Chef die Konfiguration, Bereitstellung und Verwaltung der Infrastruktur in einem Netzwerk, unabhängig von dessen Größe, automatisieren [Inc19a]. Chef wurde in Ruby und Erlang geschrieben und ist als Open-Source Anwendung unter der Apache Lizenz 2.0 veröffentlicht. Um Infrastruktur mittels Code zu beschreiben, hat Chef eine eigene, auf Ruby[3] basierende, Domain specific language (DSL) entwickelt [Mar15; TV14]. Chef-Code besteht aus *Ressourcen*, welche jeweils eine einzige Konfigurationsaufgabe definieren, die ausgeführt werden muss [Inc19a]. Eine Ressource beschreibt zum einen den gewünschten Zustand eines Konfigurationselements und zum anderen die dazu notwendigen Konfigurationsschritte. So lässt sich, wie in Auflistung 2.1 gezeigt, das beispielhafte Paket `myapp` mit der `package`-Ressource installieren.

Auflistung 2.1 Chef Ressource

```
1  package 'myapp' do
2    action: install
3  end
```

Ressourcen werden in sogenannten *Rezepten* zusammengefasst [Inc19a]. Chef-Rezepte sind ein fundamentales Element in Kochbüchern. In diesen wird die Konfiguration von Infrastruktur, mithilfe von Chef-Code, deklariert. Rezepte werden in der Chef-DSL geschrieben. In einem Rezept können dabei mehrere Ressourcen zusammengefasst werden, die jeweils eine Konfigurationsaufgabe beschreiben. Dies können Pakete sein, die installiert, Dienste, die ausgeführt oder Dateien die geschrieben werden.

[1] https://www.chef.io
[2] https://www.chef.sh
[3] https://www.ruby-lang.org/de/

Insgesamt beschreibt ein Rezept zum Beispiel die Installation und Konfiguration von Serverapplikationen oder Werkzeugen wie Apache HTTP Server[4], MySQL[5] oder Hadoop[6]. Rezepte können abhängig vom zugrundeliegenden Betriebssystem für die Ausführung bestimmter Softwareversionen konfiguriert werden und gewährleisten, dass die Software, aufgrund von verschiedenen Abhängigkeiten, in der richtigen Reihenfolge installiert wird. Rezepte können in anderen Rezepten wiederverwendet werden und müssen Teil eines Chef Kochbuches sein.

Chef Kochbücher sind zweckspezifische Artefakte, in denen Chef-Code zusammengefasst wird [Inc19a]. Ein Kochbuch definiert ein Szenario und alle zugehörigen Konfigurationsschritte dazu, wie beispielsweise das Installieren eines Web Servers. Ein Kochbuch besteht aus Rezepten, Attributen, Dateien und Vorlagen sowie einigen Metadaten. Da Kochbücher oft für verschiedene Konfigurationsmöglichkeiten entwickelt wurden, werden Attribute zur Änderung des Verhaltens eines Kochbuches verwendet. Mit den Metadaten wird sichergestellt, dass ein Kochbuch richtig installiert wird. Zum Beispiel werden in den Metadaten Abhängigkeiten zu anderen Kochbüchern deklariert. Weitere Details über den Aufbau von Chef Kochbüchern werden in Abschnitt 4.5 erläutert.

Abbildung 2.1 zeigt eine Übersicht des Konzepts von Chef und die Schritte wie Chef-Code entwickelt, getestet und bereitgestellt wird. Chef-Code wird mit dem Chef Developement Kit (ChefDK) entwickelt [Inc19a]. Hier befinden sich alle Werkzeuge zur Entwicklung von Kochbüchern. Nach der Entwicklung eines Kochbuches wird dieses auf den *Chef-Server* hochgeladen. Dieser agiert als zentrale Management Instanz und speichert zudem alle notwendigen Konfigurationsdaten wie Kochbücher und die *run-list*. Diese Liste spezifiziert, in welcher Reihenfolge die aufgelisteten Rezepte auf einem bestimmten *Chef-Client* ausgeführt werden sollen. Chef-Clients sind Rechenressourcen der Infrastruktur, wie virtuelle Maschinen, Containerinstanzen oder physische Server, auf denen der Chef-Client installiert ist. Im Betrieb rufen die Chef-Clients in regelmäßigen Abständen die neusten Kochbuchinformationen vom Chef-Server ab. Wenn der Zustand eines Netzwerkknotens nicht dem im Kochbuch definierten Zustand entspricht, führt der Chef-Client die im Kochbuch deklarierten Anweisungen aus, die den Knoten wieder in den gewünschten Zustand bringen. Damit wird sichergestellt, dass die konfigurierten Rechenressourcen zu jeder Zeit ordnungsgemäß konfiguriert sind und korrigiert Knoten, die sich nicht im gewünschten Zustand befinden.

Zusammengefasst lassen sich mit Chef Kochbüchern Rechenressourcen, wie beispielsweise virtuelle Maschinen, konfigurieren. Chef Kochbücher deklarieren die dazu notwendigen Konfigurationsschritte in Rezepten. Damit lässt sich beispielsweise ein bestimmtes Java JDK/ JRE, mit dem Java-Kochbuch [sou19] zusammen mit allen Abhängigkeiten, auf einer beliebigen Plattform die Java unterstützt, automatisch installieren und konfigurieren. Im weiteren Verlauf der Arbeit werden häufig die Begriffe Kochbuch oder Rezept verwendet, welche sich im Rahmen der vorliegenden Arbeit auf Chef Kochbücher und Rezepte der Plattform Chef[7] beziehen. Diese sind nicht mit anderen Definitionen von Kochbüchern und Rezepten zu verwechseln.

[4] https://httpd.apache.org/
[5] https://www.mysql.com/de/
[6] https://hadoop.apache.org/
[7] https://www.chef.sh/

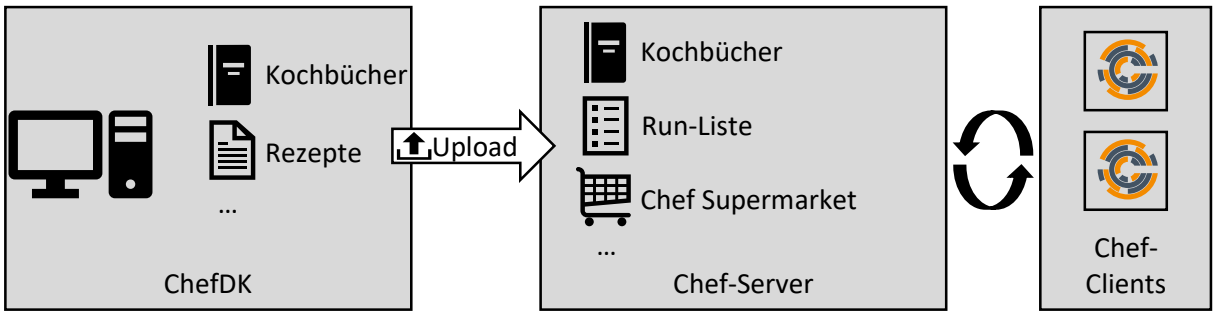

Abbildung 2.1: Entwickeln, Testen und Implementieren von Chef-Code [Inc19a]

2.2 Eclipse Winery

Die Eclipse Winery ist ein webbasiertes Werkzeug zur grafischen Modellierung von TOSCA Topologien [KBBL13]. Das Werkzeug Winery bietet die Möglichkeit, TOSCA Elemente zu erstellen, zu managen und zu modifizieren. Abbildung 2.2 zeigt die aktuellen Hauptkomponenten der Winery. Die Komponenten können über eine webbasierte grafische Benutzeroberfläche bedient werden. Der Topologie-Modelleditor ermöglicht die Erstellung von Topologien als gerichtete Grafiken. Topologien bestehen aus Instanzen von Knotentypen und Beziehungstypen. Die Elemente einer Topologie können mit Anforderungen, Fähigkeiten, Eigenschaften usw. versehen werden. Die Instanzen der Knoten und Beziehungen werden aus Vorlagen erstellt, die im Repository der Winery vorhanden sind. Als Datenbank verwendet die Winery ein Git-basiertes Dateisystem. Darin werden Knotentypen, Beziehungstypen usw. als Vorlagen in XML-Dateien (Extensible Markup Language (XML)) gespeichert. Ein Repository Management bietet Funktionen wie das Zugreifen, Speichern oder Löschen von Vorlagen in der Datenbank.

Bis jetzt ermöglicht die Winery das Modellieren von Cloud-Anwendungen. Dazu werden Vorlagen von Komponenten aus einer Datenbank verwendet. Eine Anwendungs-Topologie besteht aus Knoten mit Fähigkeiten, Anforderungen und Beziehungen untereinander. Um eine Anforderung eines Knotens aufzulösen, müssen die Anforderungen eines Knotens mit den Fähigkeiten eines anderen übereinstimmen. Dabei wird nicht überprüft, ob die Softwareversionen, die sich hinter den verbundenen Knoten verbergen, miteinander kompatibel sind. Zusätzlich stellt die Winery unterstützende Funktionen wie den *Consistency Check* oder die *Topology Completion* [HBBL+14] zur Verfügung. Mit dem Consistency Check wird automatisch überprüft, ob erstellte Vorlagen ihrer Spezifikation entsprechen. Mit der Topology Completion können unvollständige Topologien modelliert und automatisch vervollständigt werden. Dabei werden die Anforderungen eines Knotens in der modellierten Topologie mit Knoten aus der Datenbank, welche die passenden Fähigkeiten besitzen, vervollständigt. Keine der unterstützenden Funktionen bietet die Möglichkeit, automatisch die Kompatibilität der verwendeten Softwareversionen zu überprüfen. Als Grundlage dafür wird Wissen benötigt, welche Softwareversionen miteinander kompatibel sind. Dieses Wissen ist aktuell nicht vorhanden. Die vorliegende Arbeit löst dieses Problem teilweise, in dem miteinander verwendete Softwareversionen mit ihren Fähigkeiten und Anforderungen aus Chef Kochbüchern extrahiert werden. Dabei wird angenommen, dass die abgerufenen Kochbücher lauffähig sind, also syntaktisch und semantisch korrekt sind.

Das Modellierungswerkzeug Winery ist Teil des größeren Ökosystems *OpenTOSCA* [BBH+13]. Neben der Winery besteht das OpenTOSCA Ökosystem aus der TOSCA Runtime *OpenTOSCA Container* [BBH+13] und der *Vinotek* [BBKL14b]. Abbildung 2.3 zeigt die Abhängigkeiten dieser drei Hauptkomponenten des Ökosystems. Die Winery als erste Komponente besitzt Funktionalitäten, für die Entwicklung und das Management von TOSCA Definitionen. OpenTOSCA Container ist eine TOSCA Runtime, die für das Bereitstellen und das Management der Anwendungen zuständig ist. Die Vinotek, als dritte und letzte Komponente im Ablauf, ermöglicht die Bereitstellung und Außerbetriebnahme dieser Anwendungen.

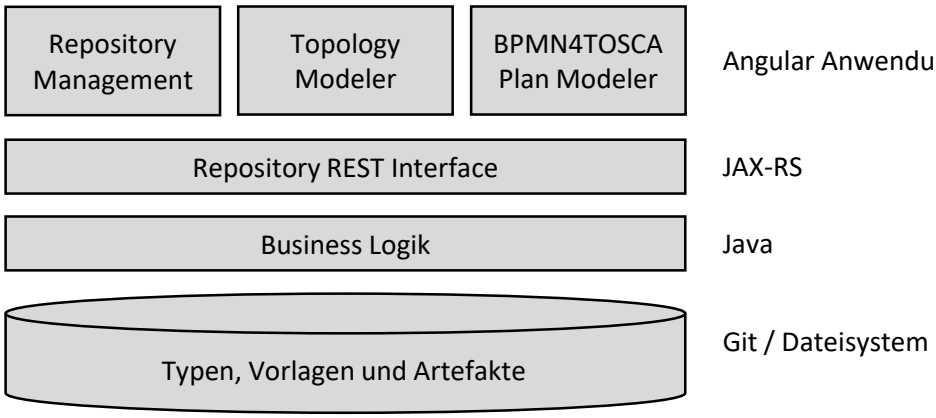

Abbildung 2.2: Aktuelle Komponenten der Winery

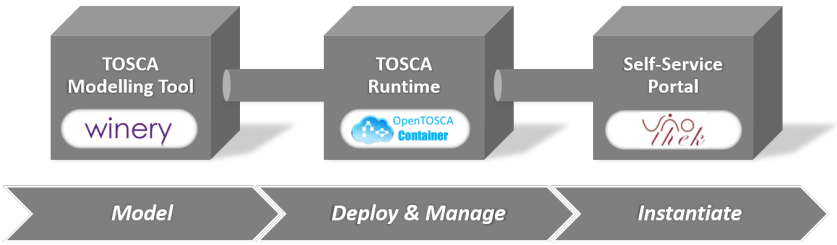

Abbildung 2.3: OpenTOSCA Ökosystem, genommen von https://opentosca.org/

2.3 Parser

Für die Abbildung von Chef Kochbüchern auf ein Architekturmodell, wird ein Verfahren benötigt, das die Dateien der Kochbücher liest, Eigenschaften erkennt und extrahiert sowie in ein Architekturmodell übersetzt. Das Lesen und Interpretieren von Quellcode wird wie im Compilerbau mit einem Parser realisiert, mit dem die auf Ruby basierte Sprache von Chef Kochbüchern erkannt wird. Dieser Abschnitt führt, für das bessere Verständnis der vorliegenden Arbeit, einige wichtige Begriffe aus der Spracherkennung sowie dem Compilerbau ein.

Definition 2.1 (Sprache) Eine Sprache ist eine Menge gültiger Sätze. Sätze bestehen aus Phrasen, welche aus Unterphrasen bestehen und so weiter [Par13]. Chef Rezepte werden in einer ChefDSL geschrieben, welche auf der Sprache Ruby basiert.

Definition 2.2 (Grammatik) Jede Sprache folgt einer Grammatik. Die Grammatik definiert formal die Syntaxregeln einer Sprache [Par13]. Jede Regel in einer Grammatik drückt die Struktur einer Subphrase aus [Par13]. Die Grammatik der ChefDSL basiert auf der Ruby Grammatik [Mat19b]. Diese ist in Pseudo Backus-Naur-Form (BNF) dargestellt. Weitere Details finden sich in der Datei *parse.y* von Ruby Distributionen [Mat19a].

Definition 2.3 (Syntaxbaum) Ein Syntaxbaum oder Parsebaum ist die abstrakte Darstellung eines Textes in einer hierarchischen Struktur. Dabei gibt jede Unterverzeichniswurzel den darunterliegenden Elementen einen abstrakten Namen [Par13]. Die Unterverzeichnisstämme entsprechen dabei den Namen der Grammatikregeln. Die Blattknoten des Baumes sind Symbole oder Token des Satzes [Par13].

Definition 2.4 (Token) Ein Token oder ein Symbol ist die kleinste logische Einheit einer Sprache. Dies können Schlüsselwörter, Bezeichner, Operatoren oder Konstanten sein. Ein Token besteht aus einem oder mehreren Zeichen. Token enthalten mindestens zwei Informationen. Zum einen den Typ des Tokens und zum anderen den repräsentierten Text.

Definition 2.5 (Lexer) Der Lexer oder Tokenizer übernimmt die lexikalische Analyse eines Parsers und zerteilt einen Text in eine Abfolge von Token.

Definition 2.6 (Parser) Anhand der definierten Grammatik überprüft der Parser, ob mehrere aufeinanderfolgende Token einen aussagekräftigen Ausdruck bilden. Dabei prüft der Parser, ob mehrere aufeinanderfolgende Token einer bestimmten Sprache, zu einer Regel der Grammatik passt. Darauf basierend wird der Syntaxbaum der geparsten Eingabe aufgebaut. Dieser Schritt wird im Compilerbau syntaktische Analyse genannt [Aho03]. Anschließend muss die Semantik überprüft werden, bei der die Bedeutung eines Ausdrucks bestimmt wird und entsprechende Maßnahmen getroffen werden [Aho03].

Abbildung 2.4 bildet den Standard Datenfluss eines Parsers ab. Auflistung 2.2 zeigt die Grammatik, die dabei verwendet wird. In dieser werden Regeln für Lexer und Parser definiert. Die gezeigte Grammatik ist nicht vollständig aber genügt zur Demonstration der Funktionsweise. Der Lexer liest den Text und erkennt zuerst die Zahlen eins und zwei, gefolgt von einem Leerzeichen. Aufgrund der deklarierten Lexer-Regel (siehe Zeile 12 von Auflistung 2.2), interpretiert der Lexer die Zahlen zusammen als das Token *NUM*. Gleiches gilt für die Zahlen eins und drei, welche der Lexer ebenfalls als *NUM*-Token erkennt. Dazwischen erkennt der Lexer das Multiplikationszeichen. Nachdem die Eingabe in Token zerlegt wurde, verwendet der Parser diese für den Aufbau einer logischen Struktur. In den Parser-Regeln wird definiert, wie eine Multiplikation aussehen muss (siehe Zeile 6 Auflistung 2.2). Da die Eingabe in diesem Beispiel zur Multiplikationsregel passt, baut der Parser den Syntaxbaum entsprechend. Wenn anschließend der Syntaxbaum durchlaufen und die Semantik interpretiert wird, trifft der Parser auf den Multiplikationsknoten. Dieser lässt sich dann als Multiplikation interpretieren, das bedeutet beide Zahlen werden miteinander multipliziert und als Wert zurückgegeben. Mit dem Lexer können auch Token übersprungen werden (siehe Zeile 15 Auflistung 2.2). Die Zeile deklariert, dass Leerzeichen vom Lexer ignoriert werden. Da diese andernfalls gleichermaßen in die Parser-Regeln integriert werden müssten, werden die Parser-Regeln dadurch vereinfacht.

Auflistung 2.2 Beispiel Grammatik

```
1  /*
2  * Parser-Regeln
3  */
4
5  expr : multiplication ;
6  multiplication : NUM MUL NUM;
7
8  /*
9  * Lexer Regeln
10 */
11
12 NUM   : [0-9]+ ;
13 MUL : '*';
14
15 WHITESPACE : ' ' -> skip ;
```

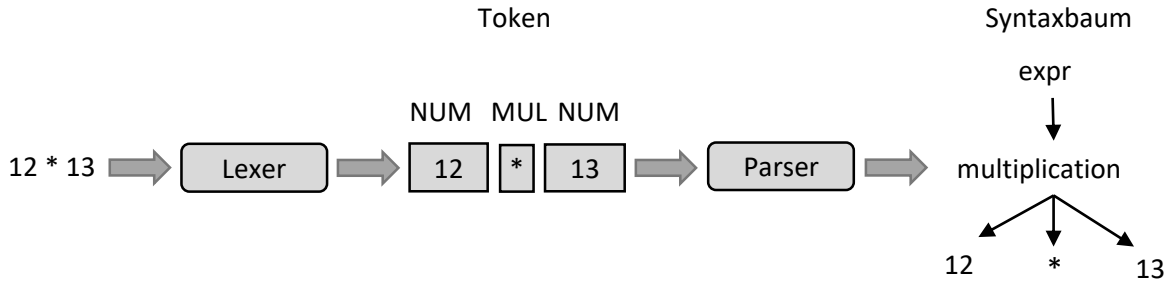

Abbildung 2.4: Standard Datenfluss eines Parsers

3 Verwandte Arbeiten

Dieses Kapitel fasst den Stand der Wissenschaft anhand von verwandten Arbeiten zusammen. In Abschnitt 3.1 werden verschiedene Ansätze von Webcrawlern vorgestellt, worauf Abschnitt 3.2 zu Topologiemodellen folgt. Danach werden in Abschnitt 3.3 Ansätze zur automatisierten Erkennung von Topologien vorgestellt. Dabei liegt der Fokus auf dem Erkennen der Topologien aus DevOps-Artefakten wie Dockerfiles, Chef Kochbüchern und Puppet Manifesten. In jedem Abschnitt wird erläutert, inwiefern sich diese Arbeit von den verwandten Arbeiten abgrenzt.

3.1 Crawler

Um lauffähige Deployment Architekturen aus DevOps-Artefakten zu extrahieren, werden lauffähige Artefakte benötigt. Lauffähige Artefakte sind solche, mit denen sich das beinhaltete Szenario vollständig auf einem Knoten installieren und konfigurieren lässt. Fertige DevOps-Artefakte finden sich in öffentlichen Repositories wie Github, dem Chef Supermarket oder Puppet Forge. Um diese zu transformieren, müssen sie zuerst aus den öffentlichen Repositories gecrawlt werden. Die Grundidee, Daten aus dem Web zu Crawlen, folgt sechs Schritten [Mat14]:

1. wird eine URL ausgewählt, von der gecrawlt wird,
2. wird die Seite heruntergeladen und geparst,
3. wird der relevante Inhalt gespeichert,
4. werden die URLs der Seite extrahiert,
5. werden die URLs der Warteschlange hinzugefügt und
6. werden die Schritte wiederholt.

Abbildung 3.1 zeigt die typische Architektur eines Webcrawlers basierend auf den sechs vorgestellten Schritten. Ein Crawler gelangt von einer zuvor festgelegten URL auf eine Webseite und läd diese herunter. Danach wird der Inhalt der Webseite geparst und relevanter Inhalt wird gespeichert. Typischerweise finden sich auf Webseiten URLs zu weiteren Webseiten. Diese werden extrahiert und einer Warteschlange hinzugefügt. Durch Wiederholen des Vorgangs, kommt der Crawler auf weitere Webseiten und durchsucht, ausgehend von der anfangs spezifizierten URL, alle verlinkten Webseiten.

Trotz des scheinbar einfachen Ablaufs der Grundidee von Webcrawlern, gibt es bei der Implementierung von Webcrawlern viele Herausforderungen, wie beispielsweise die Skalierbarkeit oder Kompromisse bei der Inhaltsauswahl [ON10]. Für die Implementierung von Webcrawlern finden sich in der Literatur folglich mehrere Ansätze. So präsentieren Boldi et al. [BCSV04] mit dem UbiCrawler einen skalierbaren und vollständig verteilten Webcrawler. Castillo [Cas05] präsentiert einen Ansatz für einen Webcrawler,

der eng mit der Suchmaschine verbunden ist. Weitere Ansätze für hochperformante Webcrawler sind beispielsweise der von Thelwall [The01] für Data-Mining Zwecke oder der von Edwards, McCurley und Tomlin [EMT01], die einen Ansatz für inkrementelle Webcrawler vorstellen. da Silva et al. [SVG+99] präsentieren einen automatischen Crawler von Dokumenten, namens CoBWeb. Shkapenyuk und Suel [SS02] zeigen einen Ansatz für einen verteilten Webcrawler, der auf einem Netzwerk von Workstations ausgeführt wird und Heydon und Najork [HN99] präsentieren einen skalierbaren sowie erweiterbaren Webcrawler namens Mercator.

Nach Endres et al. [EBLW17] lassen sich Crawler in zwei Arten kategorisieren:

Definition 3.1 (Allgemeine Crawler) Allgemeine Crawler dienen einem allgemeinen Zweck und analysieren jede Art von Dokument. Diese werden zum Beispiel bei Suchmaschinen oder für Data Mining Zwecke eingesetzt [EBLW17].

Definition 3.2 (Fokussierte Crawler) Fokussierte Crawler sind für einen bestimmten Zweck spezifiziert [CVD99; EBLW17]. Sie sind für eine Art von Inhalt spezialisiert und crawlen dementsprechend nur bestimmte Inhalte von ausgewählten Quellen.

Da sich die vorliegende Arbeit ausschließlich mit dem Analysieren von Chef Kochbüchern befasst, wird ein fokussierter Crawler verwendet. Es wird demnach ein Crawler benötigt, der ausschließlich auf das Crawlen von Chef Kochbüchern spezialisiert ist, welche anschließend analysiert werden. Dazu müssen im ersten Schritt Repositories spezifiziert werden, in denen Chef Kochbücher gespeichert sind. Endres et al. [EBLW17] präsentieren in ihrer Arbeit ein auf Chef Kochbücher spezialisiertes Crawling Framework, weshalb die Idee an sich nicht neuartig ist. Da es in der Eclipse Winery aktuell noch keinen Crawler für Chef Kochbücher gibt, stellt der in dieser Arbeit entwickelte Crawler eine Erweiterung der Eclipse Winery dar.

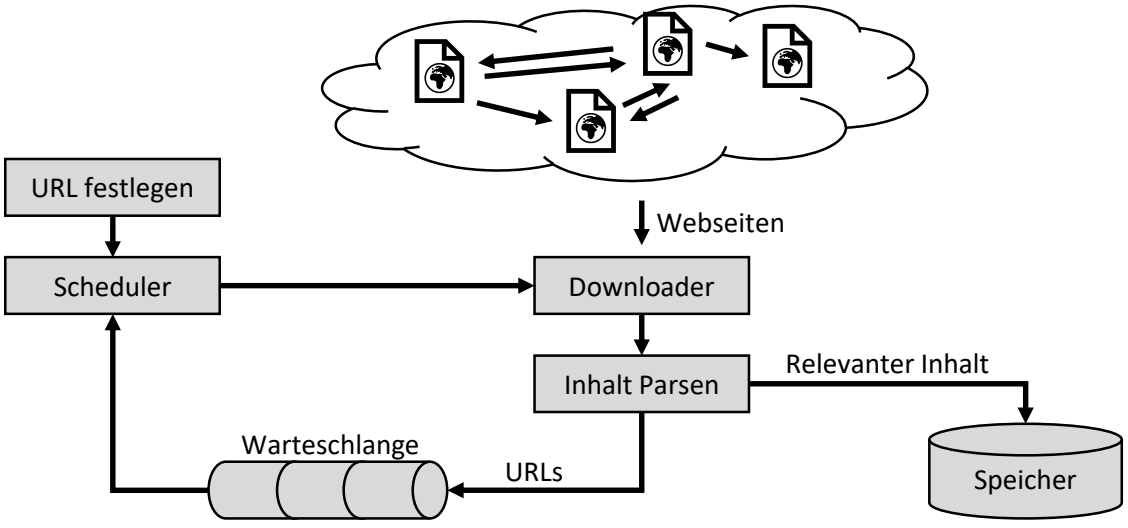

Abbildung 3.1: Typische Architektur von Webcrawlern

3.2 Topologien

Zur Darstellung der Deployment Architekturen aus Chef Kochbüchern wird ein Architekturmodell benötigt mit dem sich die Komponenten der Architektur und ihre Beziehungen zueinander generisch darstellen lassen. Chef Kochbücher dienen der automatisierten Bereitstellung von Anwendungen in der Cloud, auf privaten Ressourcen oder in Hybridumgebungen [Inc19a]. Cloud-Anwendungen werden oft auf eine Funktionalität reduziert, die in einen Container oder eine virtuelle Maschine (VM) gepackt wird [LFWW16]. Tatsächlich können Cloud-Anwendungen aus einer einfachen bis sehr komplexen Architektur bestehen [LFWW16]. Um eine detaillierte Übersicht über eine Anwendung zu bekommen, können Anwendungsarchitekturen in einem Graphen, mit ihren Komponenten und Beziehungen untereinander, dargestellt werden [EBLW17]. Dies wird realisiert, indem die Anwendungskomponenten die Knotenpunkte des Graphen und die Beziehungen zwischen den Komponenten die Kanten des Graphen darstellen. Die Knoten und Beziehungen können mit weiteren Informationen wie Fähigkeiten, Anforderungen und Eigenschaften verfeinert werden. Solch ein Graph wird auch Topologie genannt [EBLW17]. Ein Chef Kochbuch enthält meist ein Szenario zur Installation und Konfiguration einer Anwendung oder eines Services auf einem Netzwerkknoten. Dieses kann aus einer komplexen Struktur von mehreren Chef Kochbüchern bestehen (siehe laufendes Beispiel in Abschnitt 1.3). Um diese technologieunabhängig mit anderen DevOps-Artefakten zu verwenden, können Chef Kochbücher in den industriegetriebenen Standard Topology and Orchestration Specification for Cloud Applications (TOSCA) transformiert werden, wie Wettinger, Breitenbücher und Leymann [WBL14b] sowie Wettinger et al. [WBKL16] zeigen.

TOSCA ist ein OASIS Standard, der im Jahr 2013 veröffentlicht wurde [OAS13a; OAS13b]. Dieser ermöglicht die Darstellung von Cloud-Anwendungen in einer Topologie, wie in Abbildung 3.2 gezeigt [BBKL14a]. Mit TOSCA lassen sich Komponenten und Beziehungen der Anwendung darstellen [BBKL14a]. Der ursprüngliche TOSCA Standard wurde in XML-Dateien definiert und mit dem „TOSCA Simple Profile in YAML“ [OAS19] erweitert. Nach wie vor gelten die Definitionen in XML sowie in YAML [OAS19]. TOSCA definiert ein generisches Topologiemodell, welches die detaillierte Darstellung einer Cloud-Anwendung erlaubt. Wie in Abbildung 3.2 gezeigt, haben TOSCA Knotentypen Fähigkeiten, Anforderungen und Eigenschaften. Fähigkeiten enthalten die Eigenschaften, welche ein Knoten bereitstellt. Anforderungen enthalten Referenzen zu weiteren Komponenten, die für die Funktionalität eines Knotens notwendig sind. Wird eine Anwendungstopologie geplant, müssen die Anforderungen eines Knotens durch weitere Knoten aufgelöst werden, welche die passenden Fähigkeiten besitzen, die Anforderungen aufzulösen (siehe Abbildung 3.2). Dadurch werden die Anforderungen eines Knotens aufgelöst. Der TOSCA Standard eignet sich für diese Arbeit. Das Mapping von Chef Kochbüchern ist dann allerdings an den TOSCA Standard gebunden. Um das Mapping auf ein Architekturmodell generisch zu halten, wird nicht der TOSCA Standard verwendet. Allerdings kann das Mapping vom allgemeinen Architekturmodell auf den TOSCA Standard übertragen werden. Dies gilt auch für andere Modelle, die dem TOSCA Standard ähnlich sind. Dabei müssen den Knoten des Modells zwingend Anforderungen und Fähigkeiten zugeordnet werden können.

Mit TOSCA lässt sich der Bauplan einer Anwendung modellieren. Ein Unternehmen hat oft eine komplexe Anwendungslandschaft mit vielen Instanzen. Diese können zum Beispiel aus TOSCA Modellen stammen. Um eine bessere Sicht auf die gesamte IT Infrastruktur zu erhalten, definieren Enterprise Topology Graphen (ETGs) ein formales Modell zur Darstellung der IT Infrastruktur eines Unternehmens [BFL+12]. Darin lassen sich alle Entitäten einer IT Infrastruktur sowie deren Beziehungen

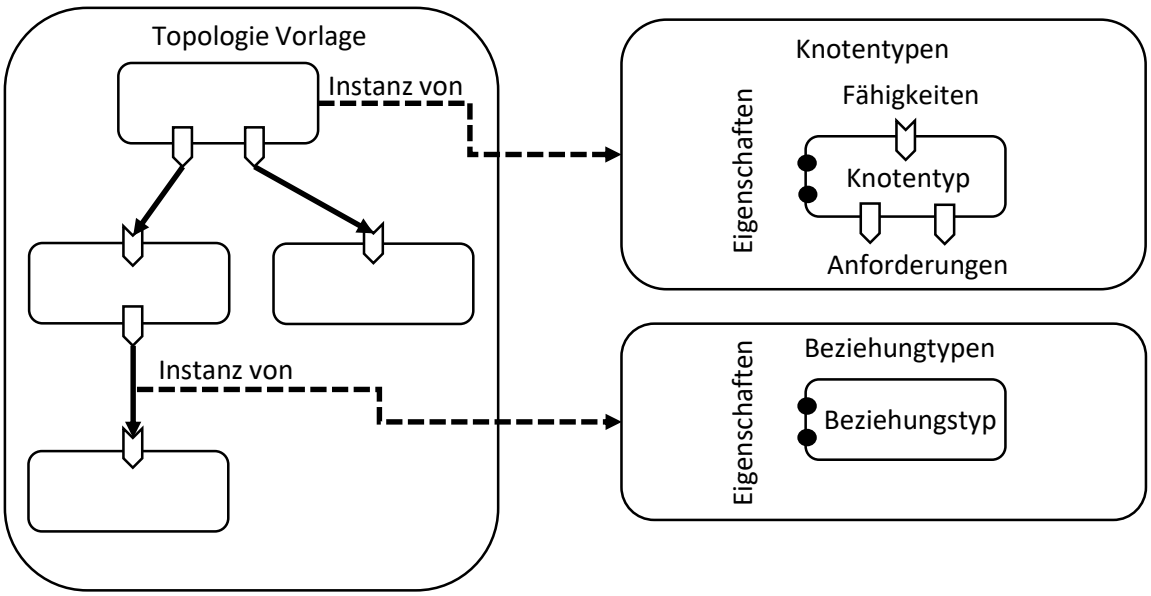

Abbildung 3.2: Struktur einer TOSCA Topologie

untereinander darstellen. Das ETG Modell ist stark an das TOSCA Modell angelehnt. Es spiegelt auf Instanzebene im Gegensatz zum TOSCA Modell, viele verschiedene (TOSCA) Anwendungsinstanzen wider, die in der Unternehmenstopologie vorhanden sind [BFL+12]. Die Darstellung der IT Infrastruktur in einem Graphen ermöglicht den Einsatz von Algorithmen zur Graphenverarbeitung [BLNS12]. Mit ETGs lassen sich komplette Unternehmensarchitekturen modellieren. Zur Modellierung einer einzelnen Anwendungsarchitektur sind diese für die vorliegende Arbeit zu wenig detailliert.

Breitenbücher [Bre16] stellt die Modellierungssprache Declarative Application Management Modelling and Notation (DMMN) vor. Diese dient in erster Linie zur Modellierung von deklarativen Managementmodellen. „Das DMMN-Metamodell ermöglicht die deklarative Modellierung auszuführender Managementaufgaben in Abhängigkeit der Anwendungsstruktur, sowie die Abbildung aller verfügbaren technischen Managementoperationen der Anwendungskomponenten und deren Relationen." [Bre16, S. 122] Das Metamodell unterteilt sich in eine Topologieschicht, eine technische Managementschicht und eine deklarative Managementschicht. Da das Modell allgemein gehalten und an keinen Standard gebunden ist, ist für diese Arbeit vor allem die Topologieschicht relevant (siehe Abbildung 3.3). Zur Kompatibilitätsprüfung von verwendeten Softwareversionen müssen Komponententypen ihre Fähigkeiten und Anforderungen als Eigenschaften haben, um zu überprüfen, ob die Anforderungen eines Knotens zu den Fähigkeiten eines unterliegenden verbundenen Knoten passen. Um Komponenten anhand ihrer Anforderungen und Fähigkeiten zu einer Topologie zu verbinden, muss das Metamodell um diese Eigenschaften erweitert werden.

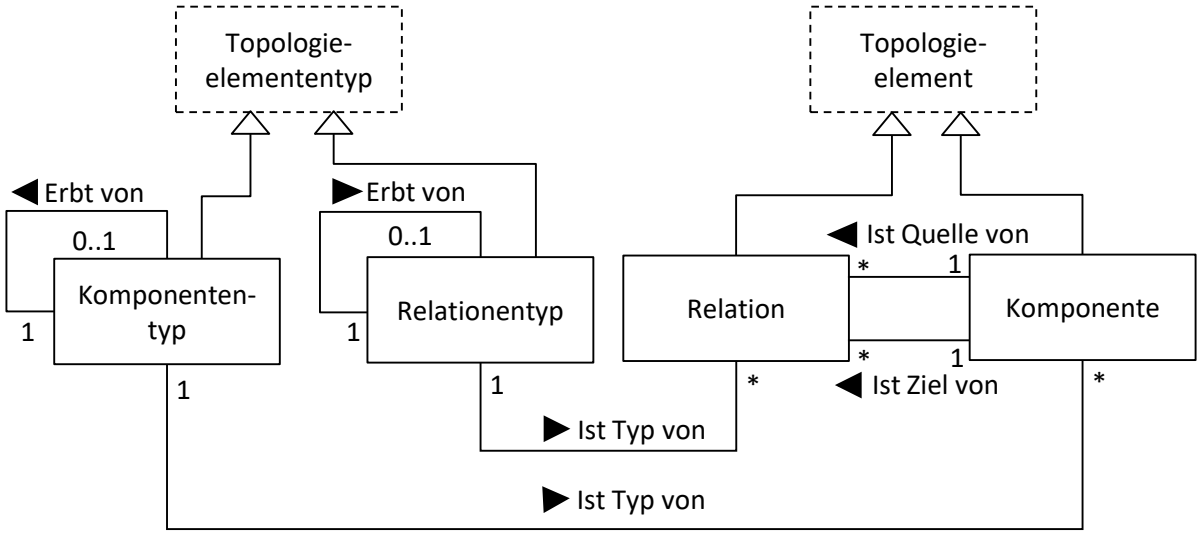

Abbildung 3.3: DMMN Topologieschicht [Bre16]

3.3 Automatisierte Erkennung von Deployment Topologien

Der Hauptbeitrag dieser Arbeit ist die Extraktion und das Mapping der Deployment Architekturen von Chef Kochbüchern auf ein Architekturmodell. Dies ist eine Transformation von Chef Kochbüchern in ein Architekturmodell mit der zugrundeliegenden Deployment Architektur der Chef Kochbücher. Modelltransformationen spielen vor allem in Bereichen wie modellgetriebenen Architekturen oder modellgetriebene Entwicklung eine bedeutende Rolle [KWB03]. Nach Kleppe, Warmer und Bast [KWB03] ist eine Transformation das automatische Generieren eines Zielmodells aus einem Quellmodell, wobei die Transformation einer Definition folgt. Diese Definition beschreibt anhand von Regeln, wie ein in einer Quellsprache formuliertes Modell in die Zielsprache übersetzt wird. Nach Sendall und Kozaczynski [SK03] lassen sich Modelltransformationen nach drei unterschiedlichen Ansätzen klassifizieren:

1. Die direkte Modell Manipulation, bei der sofort die Darstellung des Modells geändert werden kann.
2. Die Zwischendarstellung, bei der das Quellmodell zuerst in ein standardisiertes Zwischenmodell exportiert wird, um dann weiter transformiert zu werden.
3. Eine Transformationssprache, die eine Reihe von Konstrukten oder Mechanismen für das explizite Ausdrücken, Zusammenstellen und Anwenden von Transformationen bereitstellt.

In dieser Arbeit wird die zweite Kategorie nach Sendall und Kozaczynski [SK03] verwendet.

DevOps-Automatisierungsansätze unterscheiden sich voneinander. Sie zu integrieren und zu kombinieren, um Anwendungen in der Cloud bereitzustellen, ist anspruchsvoll. Wettinger, Breitenbücher und Leymann [WBL14b] zeigen zum einen eine erste Klassifizierung von DevOps-Artefakten und beschreiben deren Verwendung. Zum anderen wird ein Ansatz präsentiert, der durch eine Transformation die nahtlose und interoperable Orchestrierung beliebiger Artefakte zum Modellieren und Bereitstellen von Anwendungstopologien ermöglicht [WBL14b]. Dabei wird die zweite Kategorie der

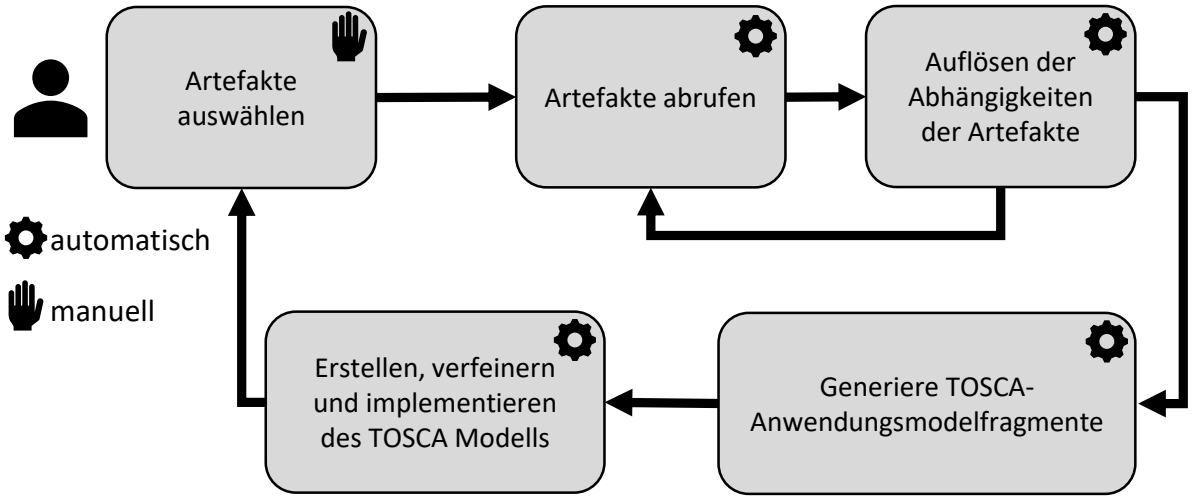

Abbildung 3.4: DevOps-Artefakt Transformation [WBL14b]

Modelltransformationen nach Sendall und Kozaczynski [SK03] verwendet. Nach Wettinger, Breitenbücher und Leymann [WBL14b] lassen sich DevOps-Artefakte in Node-centric Artefakte (NCAs) und Environment-centric Artefakte (ECAs) unterteilen. NCAs bestehen aus Skripten, Images, Modulen, deklarativen Konfigurationsdefinitionen usw., die auf einem Knoten, wie einem physikalischen Server oder einer virtuellen Maschine, ausgeführt werden [WBL14b]. Da Beziehungen zu anderen Knoten nicht dargestellt werden, sind NCAs nicht dazu gedacht eine vollständige Anwendungstopologie bereitzustellen. ECAs bestehen aus Skripten, Paketen oder Vorlagen, die in einer Umgebung von einem oder mehren Netzwerkknoten ausgeführt werden [WBL14b]. Beziehungen zu anderen Knoten werden dabei explizit angegeben. Folglich kann mit ECAs eine komplette Anwendungstopologie bereitgestellt werden [WBL14b]. Chef Kochbücher werden der Klasse der NCAs zugeordnet [WBL14b]. Wettinger, Breitenbücher und Leymann [WBL14b] präsentieren zudem eine Methode zur Transformation von DevOps-Artefakten verschiedener Technologien in ein gemeinsames Zwischenmodell zur nahtlosen und interoperablen Nutzbarkeit. Als gemeinsames Zwischenmodell wird hier der Standard TOSCA [OAS13b] verwendet. Der Ablauf der Methode ist in Abbildung 3.4 dargestellt. Im ersten Schritt werden manuell die Artefakte ausgewählt, die für die gewünschte Applikationstopologie notwendig sind. Das können Chef Kochbücher, Docker Images, Puppet Manifeste oder Ansible Playbooks sein. Danach werden die Artefakte von den öffentlichen Repositories abgerufen. Im dritten Schritt werden automatisch die Abhängigkeiten geprüft, die in den abgerufenen Artefakten enthalten sind, was dazu führen kann, dass weitere Artefakte abgerufen werden müssen, bis alle Abhängigkeiten aufgelöst sind. Im nächsten Schritt werden TOSCA Knotentypen und TOSCA Beziehungstypen, basierend auf den ausgewählten und abgerufenen Artefakten, generiert. Nachdem die Artefakte in TOSCA Typen transformiert wurden, können diese weiterverwendet werden. So können zum Beispiel mit dem vorgestellten Modellierungswerkzeug Winery, Anwendungstopologien mit den generierten TOSCA Typen erstellt und später automatisch bereitgestellt werden. Im Gegensatz zu dieser Arbeit werden in dem Ansatz zum Zeitpunkt der Applikationsplanung die Artefakte gecrawlt, die zur Bereitstellung der Anwendung notwendig sind. Dabei ist das Ziel die automatische Bereitstellung von Anwendungen, die Artefakte unterschiedlicher Technologien verwenden und nicht die automatische Überprüfung der Kompatibilität der Anwendungen.

Wettinger, Breitenbücher und Leymann [WBL14b] zeigen zudem wie die technische Transformation von Chef Kochbüchern stellvertretend für NCAs und Juju charms für ECAs funktioniert. Dabei wird eine feine und alternativ eine grobe Transformation vorgestellt, welche in Abbildung 3.5 beispielhaft anhand des laufenden Beispiels dieser Arbeit gezeigt wird. Da sich diese Arbeit nur auf Chef Kochbücher bezieht, wird an dieser Stelle nur die Transformation von Kochbüchern erläutert. Bei der feinen Transformation (siehe Abbildung 3.5 a)) wird für das ausgewählte Chef Kochbuch sowie für die enthaltenen Abhängigkeiten zu anderen Kochbüchern jeweils ein Knotentyp erstellt. Am laufenden Beispiel wird beispielsweise das MyApp-Kochbuch in einen Knoten transformiert und da dieses eine Abhängigkeit zum Java-Kochbuch hat, wird dieses ebenfalls in einen Knoten transformiert. Bei der groben Transformation (siehe Abbildung 3.5 b)) werden die Abhängigkeiten des ausgewählten Kochbuches in den Knoten selbst integriert. Am laufenden Beispiel wird beispielsweise das MyApp-Kochbuch und das Java-Kochbuch in den MyApp Knoten transformiert. Bei der feinen Transformation wird für jedes Kochbuch ein Knotentyp erstellt. Jeder Knoten hat Anforderungen (A) und Fähigkeiten (F), mit denen mehrere Knoten zu Graphen zusammengefügt werden können, sodass die Fähigkeiten eines Knotens mit den Anforderungen des verbundenen Knotens übereinstimmt. Als Fähigkeit eines Knotens wird der Name des Kochbuches verwendet. Zudem kann ein Kochbuch Abhängigkeiten zu anderen Kochbüchern enthalten. Für jede Abhängigkeit zu einem anderen Kochbuch wird eine Anforderung am Knoten generiert, der auf das abhängige Kochbuch verweist. Die Rezepte und Attributdateien eines Kochbuches werden den Eigenschaften der erstellten Knoten angehängt. Die grobe Transformation funktioniert weitestgehend analog zur feinen Transformation. Der einzige Unterschied ist, dass Abhängigkeiten zu anderen Kochbüchern keinen extra Knoten ergeben, sondern in den des ursprünglichen Kochbuches integriert werden [WBL14b]. Da der Ansatz von Wettinger, Breitenbücher und Leymann [WBL14b] Abhängigkeiten zu weiteren Kochbüchern ebenfalls rekursiv auflöst, ist dieser aufgrund des Ablaufes ähnlich zur vorliegenden Arbeit. Ziel des Ansatzes ist es aber DevOps-Artefakte in ein gemeinsames Zwischenmodell zu transformieren, um verschiedene Technologien nahtlos und interoperabel nutzbar zu machen. Dabei werden nur die Metadaten der Kochbücher analysiert, um Namen und Abhängigkeiten zu extrahieren. Alle weiteren Dateien werden nicht analysiert. Verglichen mit dieser Arbeit erhält ein extrahierter Knoten als Fähigkeiten den Namen des transformierten Chef Kochbuches und als Abhängigkeiten nur die Namen der abhängigen Kochbücher. Für die Bereitstellung des Knotens werden dem Knoten die notwendigen Attributdateien sowie die Rezepte des Kochbuchs angehängt. Daraus werden, im Gegensatz zu dieser Arbeit, keine weiteren Informationen über installierte Pakete und Abhängigkeiten untereinander extrahiert, weshalb sich die Transformation klar von der Transformation der vorliegenden Arbeit abgrenzt.

Es existieren verschiedene DevOps-Ansätze, die meistens auf individuellen Ansätzen basieren. Wenn eine Anwendungstopologie Artefakte von verschiedenen Anbietern enthält, wird meist viel Code benötigt, um diese gemeinsam zu verwenden. Um dieses Problem zu lösen, wurde von Wettinger et al. [WBKL16] ein Ansatz vorgestellt, dessen Ziel es ist, verschiedene Arten von DevOps-Artefakten auf einen gemeinsamen Standard zu bringen. Dadurch sollen die verschiedenen DevOps-Artefakte zusammen kombinierbar und wiederverwendbar sein. Der Ansatz basiert auf dem bereits erläuterten von Wettinger, Breitenbücher und Leymann [WBL14b]. Im Gegensatz dazu, werden mit einem Crawler DevOps-Artefakte von öffentlichen Repositories abgerufen, bevor Anwendungen geplant werden. Dieser Schritt folgt dem Ansatz von Wettinger, Andrikopoulos und Leymann [WAL15], bei dem DevOps-Artefakte von öffentlichen Repositories gecrawlt werden und in einer Wissensdatenbank abgelegt werden. Damit können Entwickler sofort einen Überblick über die existierenden Artefakte verschiedener DevOps-Ansätze bekommen. Die gecrawlten Artefakte werden eindeutig identifizierbar in einem

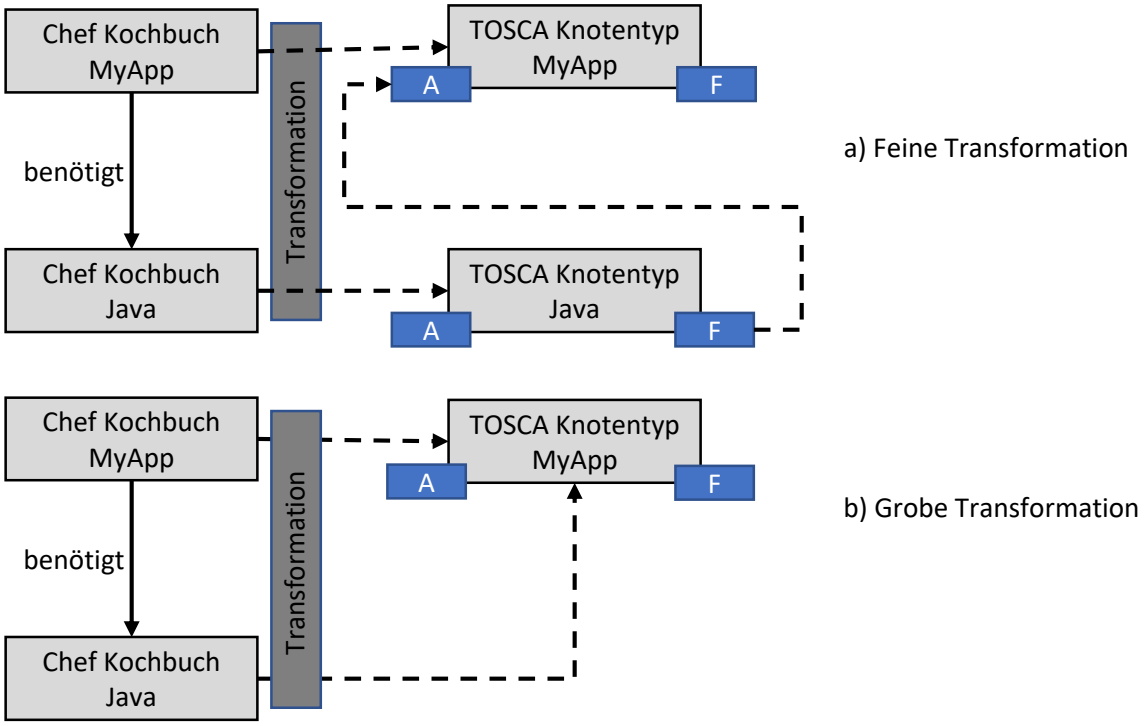

Abbildung 3.5: Feine und grobe Artefakt Transformation am laufenden Beispiel [WBL14b]

gemeinsamen Format in einer Wissensdatenbank abgelegt. Im zweiten Schritt werden die Artefakte in ein gemeinsames Modell transformiert. Die Transformation erfolgt nach dem bereits erläuterten Ansatz von Wettinger, Breitenbücher und Leymann [WBL14b]. Die transformierten Artefakte werden in einer gemeinsamen Datenbank abgelegt. Das Ergebnis ist eine Datenbank mit standardisierten DevOps-Artefakten verschiedener Ansätze, die miteinander kombinierbar und wiederverwendbar sind. Da die Transformation der Chef Kochbücher die Gleiche ist wie die des vorherigen Ansatzes [WBL14b], lässt sich der Ansatz von Wettinger et al. [WBKL16] im Bezug auf diese Arbeit insgesamt ähnlich bewerten. Die Idee, Artefakte aus festgelegten öffentlichen Quellen abzurufen und zu transformieren ist demnach keine neuartige Idee. Da die Artefakte in einen gemeinsamen Standard transformiert werden, um sie auf einer gemeinsamen Plattform nutzen zu können, unterscheidet sich die Art der Transformation der DevOps-Ansätze maßgeblich zu der vorliegenden Arbeit. In der vorliegenden Arbeit werden im Unterschied dazu die enthaltenen Deployment Architekturen extrahiert und nur diese gespeichert, weshalb sich die vorliegende Arbeit davon abgrenzt. Bei der Transformation werden nur der Name des Kochbuches sowie die abhängigen Kochbücher aus den Metadaten extrahiert und die anderen Dateien des Kochbuchs als Eigenschaften angehängt. Im Gegensatz zu der vorliegenden Arbeit werden die verwendeten Komponenten und Plattformen nicht aus den Artefakten extrahiert, weshalb sich die Transformation zur Extraktion der enthalten Deployment Architekturen klar davon abgrenzt.

Endres et al. [EBLW17] stellen in ihrer Arbeit die *Topologize*-Methode vor, mit der DevOps-Artefakte von öffentlichen Quellen abgerufen werden und in Topologiemodelle transformiert werden. Die Methode wurde mit dem Ziel entwickelt, verschiedene Arten von DevOps-Artefakten in ein gemeinsames Topologiemodell zu übersetzen, damit komplexere, aus unterschiedlichen Artefakten bestehende Anwen-

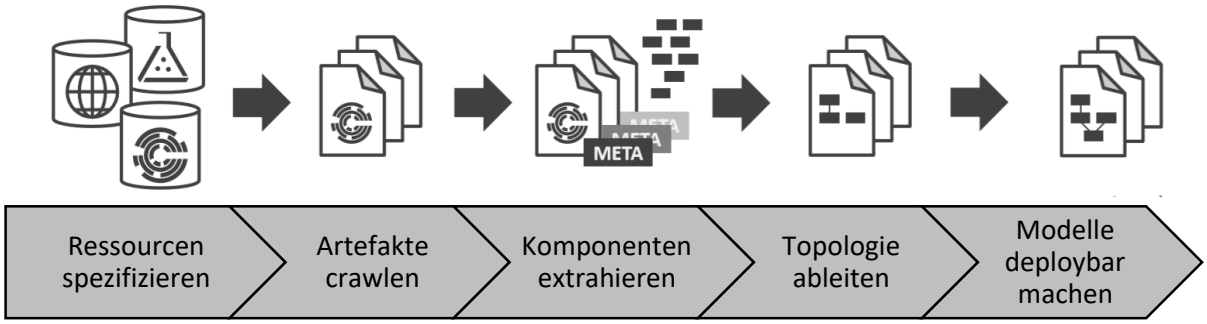

Abbildung 3.6: Schritte der *Topologize* Methode [EBLW17]

dungen einfacher modelliert und bereitgestellt werden können. Die Topologize-Methode von Endres et al. [EBLW17] besteht aus fünf Schritten, die in Abbildung 3.6 dargestellt sind. Im ersten Schritt werden Repositories spezifiziert von denen Artefakte abgerufen werden. Wie bereits erläutert, gibt es verschiedene öffentliche Repositories, in denen die Artefakte der unterschiedlichen Technologien, wie beispielsweise Chef, Ansible oder Puppet, bereitgestellt werden. Die Topolgize-Methode wird anhand einer Fallstudie mit Chef Kochbüchern validiert. In diesem Fall werden Kochbücher vom Chef Supermarket[1] abgerufen, in dem Kochbücher für verschiedene Anwendungen und Szenarien von der Chef Community bereitgestellt werden. Im zweiten Schritt werden die Artefakte von den festgelegten Repositories abgerufen. Im dritten Schritt werden die Komponenten der abgerufenen Artefakte extrahiert und in den Standard TOSCA transformiert. In dem Anwendungsfall Chef geschieht das Mapping auf die TOSCA Typen nach dem Ansatz von Wettinger, Breitenbücher und Leymann [WBL14b], welcher zu Beginn des Abschnitts erläutert wurde. Im vierten Schritt werden die analysierten Artefakte und ihre Metainformationen interpretiert, um die Struktur der Anwendung, welche durch das Artefakt installiert wird, abzuleiten [EBLW17]. Das Topologiemodell ist ein gerichteter, beschrifteter Graph, wobei Knoten die Komponenten und Kanten die Abhängigkeiten beschreiben [EBLW17]. Der fünfte und letzte Schritt der Topolgize-Methode, bei dem die Topologien einsetzbar gemacht werden, ist optional. Hier werden die extrahierten Topologien so weit mit Informationen angereichert, dass sie automatisch bereitgestellt werden könnten. Wenn zum Beispiel ein Chef Kochbuch nur die Installation einer Datenbank beschreibt, wird die Topologie durch ein Betriebssystem und einen Knoten einer virtuellen Maschine erweitert [EBLW17]. Die Topologize-Methode ist ein Ansatz zur Umwandlung von Artefakten aus öffentlichen Repositories in ein technologieunabhängiges Topologiemodell. Dabei werden die enthaltenen Komponenten sowie ihre Beziehungen beschrieben. Dies beinhaltet allerdings nicht die Extraktion der installierten Pakete und Versionen aus den Artefakten. Stattdessen werden die gecrawlten Artefakte in ein technologieunabhängiges Topologiemodell gewrappt, ohne den eigentlichen Inhalt des Artefakts zu analysieren.

Shambaugh et al. [SWG16] präsentieren einen Ansatz zur statischen Verifizierung von Puppet Manifesten, welche eine starke Ähnlichkeit zu Chef Kochbüchern aufweisen. Um Puppet Manifeste auf Determinismus zu überprüfen, werden diese in einen gerichteten azyklischen Graphen kompiliert. Dabei

[1] https://supermarket.chef.io/

sind die Knoten die Ressourcen, welche mit gerichteten Kanten verbunden sind. Die Kanten drücken eine *depends on* Beziehung aus. Puppet Manifeste können zur Konfiguration von unterschiedlichen Maschinen verwendet werden. Wenn diese in Manifesten definiert werden, wird für jeden Knoten ein Ressourcengraph erstellt. Im Gegensatz zu der vorliegenden Arbeit werden die Konfigurationsaufgaben betrachtet. Dabei werden die verwendeten Komponenten nicht extrahiert, sondern es wird überprüft ob die Konfigurationsschritte deterministisch sind. Weitere Arbeiten zur Überprüfung der Idempotenz von Konfigurationsskripten finden sich bei Hummer et al. [HROE13], am Beispiel von Chef Kochbüchern und bei Hanappi et al. [HHD16], am Beispiel von Puppet Manifesten. In beiden Ansätzen werden die Artefakte in ein Zustandsübergangsdiagramm transformiert. Im Unterschied zur vorliegenden Arbeit werden die Artefakte im Hinblick auf die Zustandsänderungen transformiert. Dabei werden die Ressourcen der Artefakte zwar in einen Graphen mit Abhängigkeiten transformiert, aber die verwendeten Softwareversionen sind dabei nicht von Bedeutung.

Bis an dieser Stelle wurde die automatische Erkennung von Topologien aus DevOps-Artefakten thematisiert, welche häufig nur die Bereitstellung einer Anwendung beschreiben. Unternehmen haben oft eine komplexe IT Infrastruktur, die aus vielen Komponenten bestehen kann. Zur besseren Übersicht lassen sich diese in einem Graphen darstellen, der alle Anwendungen, Prozesse, Services und Komponenten sowie deren Beziehungen untereinander beinhaltet. Diese manuell zu erfassen und darzustellen wäre eine zeitintensive und fehleranfällige Aufgabe. Zur Automatisierung dieser stellen Binz et al. [BBKL13] einen Ansatz zur automatischen Generierung solcher Topologiegraphen vor, welcher die automatische Erfassung der IT Infrastruktur in einem Topologiegraphen ermöglicht. Der Ansatz zielt jedoch auf die Darstellung einer laufenden IT Infrastruktur ab, wohingegen die vorliegende Arbeit den Fokus auf das Extrahieren der Deployment Architekturen einzelner Anwendungen aus DevOps-Artefakten legt.

Beim Planen von Anwendungsarchitekturen muss zu diesem Zeitpunkt manuell überprüft werden, ob die verwendeten Versionen miteinander kompatibel sind. Um dies zu automatisieren sind Versionsinformationen über die verwendeten Komponenten, mit deren Fähigkeiten und Anforderungen notwendig. Zur Realisierung sind Kenntnisse über die Kompatibilität von verschiedenen Softwareversionen erforderlich. Diese Informationen können beispielsweise in einer Wissensdatenbank (vgl. [WAL15]), die als Basis dient, bereitgestellt werden. Bis jetzt existiert kein Ansatz Versionsinformationen, die in Chef Kochbüchern enthalten sind, zu extrahieren und diese in ein generisches Architekturmodell zu transformieren. Darum ist der in Kapitel 4 und Kapitel 5 präsentierte Ansatz ein neuartiger Beitrag. Dabei werden die verwendeten Komponenten und Versionen aus Chef Kochbüchern extrahiert und in ein Architekturmodell transformiert, um diese in eine Wissensdatenbank mit Versionsinformationen von Komponenten abzulegen. Zum Füllen der Wissensdatenbank werden Chef Kochbücher, von denen deren Lauffähigkeit angenommen wird, benötigt. Diese finden sich in öffentlichen Repositories, wie dem Chef Supermarket, in denen Chef Kochbücher zur freien Verwendung bereitgestellt werden. Zum Abrufen dieser Chef Kochbücher wird ein fokussierter Crawler verwendet, der darauf spezialisiert ist, Chef Kochbücher aus öffentlichen Quellen abzurufen.

4 Ansatz für das Mapping von Chef Kochbüchern auf ein generisches Architekturmodell

In diesem Kapitel wird das Konzept für die Extraktion der in Chef Kochbüchern enthaltenen Deployment Architekturen vorgestellt. Dabei ist das Ziel die Extraktion der verwendeten Softwareversionen und Plattformversionen. Die extrahierten Architekturen dienen als Basis für die automatische Überprüfung der Kompatibilität der verwendeten Komponenten von geplanten Anwendungsarchitekturen. Dazu werden Informationen über kompatible Softwareversionen benötigt. Dieses Kapitel stellt einen Ansatz vor, mit dem dieses Wissen aus Chef Kochbüchern extrahiert werden kann. Abschnitt 4.1 zeigt die Architektur des Gesamtkonzeptes, mit dem sich Chef Kochbücher aus öffentlichen Repositories crawlen lassen, um anschließend die enthaltenen Deployment Architekturen zu extrahieren und auf das generische Architekturmodell abzubilden. Um umfangreiches Wissen zu generieren, werden fertig implementierte Chef Kochbücher benötigt. Dazu wird in Abschnitt 4.2 der Ansatz für das Crawlen von Chef Kochbüchern aus öffentlichen Repositories erläutert. Anschließend wird in Abschnitt 4.3 das Metamodell vorgestellt, auf das die extrahierten Deployment Architekturen aus Chef Kochbüchern abgebildet werden. Abschnitt 4.4 stellt den allgemeinen Ansatz für die Transformation von Kochbüchern in das Architekturmodell vor. Dazu werden in Abschnitt 4.5 Eigenschaften von Kochbüchern identifiziert, mit denen sich die zugrundeliegende Deployment Architektur darstellen lässt. Im Anschluss werden in Kapitel 5 Details zu der Transformation in das generische Architekturmodell erläutert.

4.1 Gesamtkonzept

In diesem Abschnitt wird das entwickelte Gesamtkonzept der vorliegenden Arbeit vorgestellt. Abbildung 4.1 zeigt eine Übersicht der gesamten Architektur des vorgestellten Ansatzes, welche hauptsächlich aus einem Chef Kochbuch Crawler und einem Chef Kochbuch Compiler besteht. Der Chef Kochbuch Crawler (siehe Abschnitt 4.2) crawlt alle verfügbaren und nicht veralteten Kochbücher aus öffentlichen Chef Repositories. Die Kochbücher werden dabei entpackt und auf der lokalen Festplatte gespeichert. Der Crawler agiert dabei als ein eigenes unabhängiges Modul. Der Chef Kochbuch Compiler übersetzt die gecrawlten Kochbücher von der Festplatte und arbeitet diese der Reihe nach ab. Ein Kochbuch besteht aus mehreren Dateien, die der Chef Kochbuch Compiler einliest und übersetzt. Der Chef Kochbuch Compiler arbeitet bei der Übersetzung der Dateien in mehreren Schritten, die an den Compilerbau angelehnt sind (siehe Abschnitt 4.4). Im ersten Schritt zerlegt der Lexer die Ruby-Dateien in Token. Danach transformiert der Parser die Token in einen abstrakten Syntaxbaum. Dies geschieht nach den Regeln der Ruby-Grammatik. Der Syntaxbaum wird anschließend semantisch analysiert. Dabei werden relevante Informationen (siehe Abschnitt 4.5) zu der Abbildung der, im Kochbuch enthaltenen, Deployment Architekturen extrahiert und auf ein generisches Architekturmodell abgebildet. Dies geschieht nach

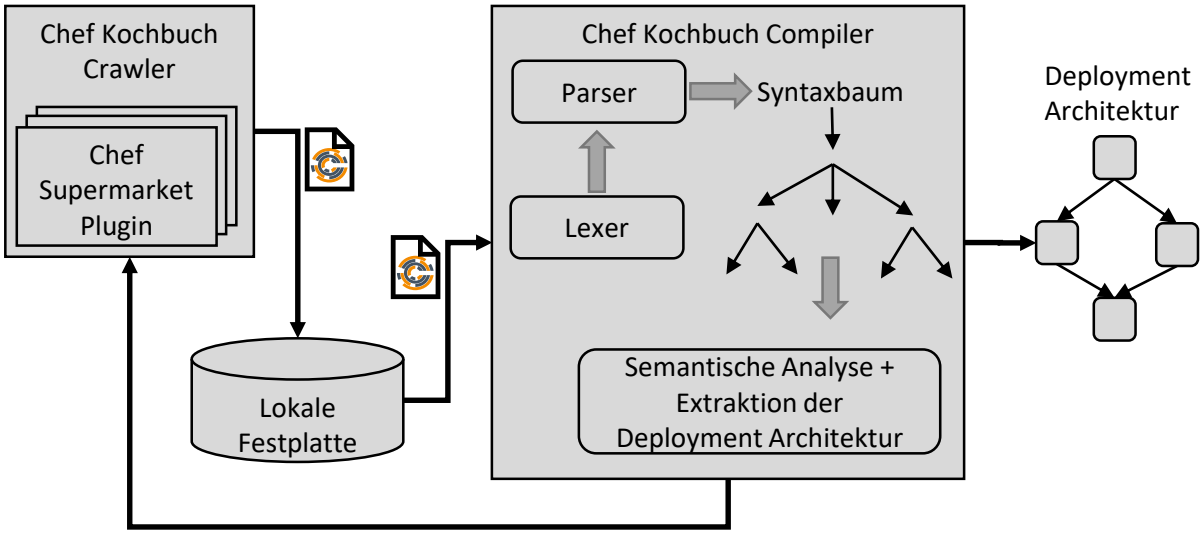

Abbildung 4.1: Gesamtkonzept der Extraktion von Deployment Architekturen aus Kochbüchern

dem in Kapitel 5 vorgestellten Ansatz, zu der Transformation von Kochbüchern in das enthaltene Architekturmodell. Chef Kochbücher können Abhängigkeiten zu weiteren Kochbüchern besitzen, welche bei der semantischen Analyse erkannt und vom Chef Kochbuch Crawler auf der lokalen Festplatte gespeichert werden. Abhängige Kochbücher werden somit ebenfalls in die Analyse miteinbezogen. Nach der Transformation des Kochbuches werden die abhängigen Kochbücher ebenfalls rekursiv transformiert. Dieser Vorgang wird wiederholt, bis ein Kochbuch erreicht wird, dass keine weiteren Abhängigkeiten besitzt. Details zu dem Vorgang werden in Kapitel 5 erläutert. Das Resultat der Transformation sind die enthaltenen Deployment Architekturen des transformierten Kochbuches in Form eines generischen Architekturmodells (siehe Abschnitt 4.3). Dabei stellt jeder Knoten ein transformiertes Kochbuch dar. Das genaue Mapping auf das Architekturmodell wird in Kapitel 5 erläutert.

4.2 Chef Kochbuch Crawler

Vor der Extraktion der Deployment Architekturen aus Chef Kochbüchern werden fertige, lauffähige Chef Kochbücher benötigt. Lauffähig sind Kochbücher, wenn sie sich zur Konfiguration von Rechenressourcen verwenden lassen. Fertige Kochbücher werden von der Chef Community in öffentlichen Repositories wie Github[1] bereitgestellt. Für diese Arbeit wird angenommen, dass Kochbücher aus öffentlichen Repositories lauffähig sind, was gleichbedeutend mit der syntaktischen sowie semantischen Korrektheit der gecrawlten Kochbücher ist. Dazu wird ein fokussierter Crawler verwendet, der auf das Crawlen von Chef Kochbüchern aus öffentlichen Repositories spezialisiert ist. Die Aufgabe dieses ist es, Chef Kochbücher aus öffentlichen Repositories zu crawlen und für weitere Analyseschritte lokal zu speichern. Der Zugriff auf öffentliche Repositories unterscheidet sich untereinander. Der entwickelte Crawler kann

[1] https://github.com/

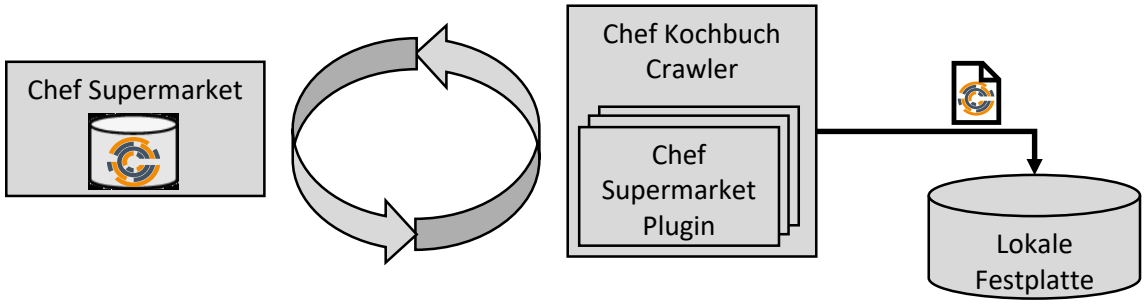

Abbildung 4.2: Chef Kochbuch Crawler

durch ein domänenspezifisches Plugin, das den Zugriff auf das entsprechende Repository implementiert, erweitert werden (analog zu Algorithmus 4.1).

Das Unternehmen Chef bietet speziell für deren Kochbücher den Chef Supermarket[2], als öffentliches Repository für Kochbücher, an. Dieser beinhaltet einen Großteil der allgemein zugänglichen Kochbücher. Zudem finden sich dort viele der auf Github veröffentlichten Kochbücher wieder. Abbildung 4.2 zeigt die Architektur des Chef Kochbuch Crawlers mit einem Plugin für den Chef Supermarket, der die verfügbaren Kochbücher crawlt und auf der lokalen Festplatte speichert.

Der Zugriff auf die im Chef Supermarket verfügbaren Kochbücher erfolgt über eine REST API[3] (Representational State Transfer (REST) Application Programming Interface (API)). Mit der API lassen sich Informationen zu allen Kochbüchern abrufen, die im Chef Supermarket verfügbar sind. Die Methode für das Crawlen aller verfügbaren Kochbücher aus dem Chef Supermarket wird in Algorithmus 4.1 beschrieben. Zur Abfrage mehrerer Kochbücher über die Supermarket-API muss ein Startwert und eine Anzahl an Kochbüchern angegeben werden. Für das Crawlen aller Kochbücher, wird die Gesamtanzahl der Kochbücher im Chef Supermarket benötigt. Die Anfrage dazu wird in Zeile 4 von Algorithmus 4.1 deklariert. Antworten auf Anfragen an den Chef Supermarket über die API sind im JSON-Format[4]. Mit der Methode `getJsonFromUrl(url)` wird der Inhalt aus der URL gelesen und in ein JSON-Objekt[5] gespeichert (siehe Zeile 5). Der Inhalt des Objekts wird beispielhaft in Auflistung 4.1 abgebildet und beinhaltet den Startwert der Abfrage im `start` Feld, die Anzahl aller im Chef Supermarket verfügbaren Kochbücher im Feld `total` sowie einen Array mit Informationen über die angefragten Kochbücher ab dem Startpunkt der Abfrage im Feld `items`. Werden mehr als 1000 Kochbücher abgefragt sind in dem Feld `items` trotzdem maximal 1000 Elemente enthalten. Die Informationen im Array enthalten die URL zu dem Kochbuch im Feld *cookbook*. Die Gesamtanzahl der im Chef Supermarket verfügbaren Kochbücher wird aus dem `total` Feld ausgelesen (siehe Zeile 4 von Algorithmus 4.1).

Mit der GET-Anfrage aus der Überschrift von Auflistung 4.1 lassen sich alle Kochbücher auf einmal abfragen, indem der *start*-Wert auf 0 und die *items* auf die Gesamtanzahl der Kochbücher gesetzt werden. In der Antwort des Chef Supermarkets sind trotzdem nur die ersten 1000 Kochbücher ab dem Startwert enthalten. Folglich werden die Kochbücher in mehreren Iterationen gecrawlt. Dazu wird ein Offset für den Startpunkt der ersten Iteration und die Anzahl der gecrawlten Kochbücher pro Iteration deklariert

[2] https://supermarket.chef.io/
[3] https://docs.chef.io/supermarket_api.html
[4] http://json.org/
[5] https://docs.oracle.com/javame/8.0/api/json/api/com/oracle/json/JsonObject.html

(siehe Zeile 2 und 3 in Algorithmus 4.1). Durch die `while`-Schleife in Zeile 7 wird sichergestellt, dass alle Kochbücher aus dem Chef Supermarket gecrawlt werden. Zu Beginn jeder Iteration wird die URL zur Abfrage der Kochbücher mit dem Offset und den Kochbüchern pro Iteration gesetzt (siehe Zeile 8). Nach Durchlauf einer Iteration wird der Offset für die API-Abfrage der nächsten Iteration so gesetzt, dass die folgenden 1000 Kochbücher gecrawlt werden (siehe Zeile 17). Damit wird in Zeile 8 von Algorithmus 4.1 in jeder Iteration eine neue API-Anfrage generiert und die Informationen gespeichert (siehe Zeile 9). Die Kochbuchinformationen werden in Zeile 10 aus dem `items`-Feld der Antwort (siehe Auflistung 4.1 Zeile 4-12) ausgelesen und die Informationen jedes Kochbuches in einem Feld des Arrays abgelegt. Jedes Feld des Arrays hält Informationen eines Kochbuches mit der weiterführenden URL zu dem Kochbuch aus dem Feld *cookbook*, aus der API-Antwort von Auflistung 4.1. Mit der `for`-Schleife in Zeile 11 von Algorithmus 4.1 wird über alle Kochbücher im Array iteriert und jeweils die weiterführende URL aus den Feldern im Array ausgelesen und gespeichert (siehe Zeile 13). Die weiterführende URL aus Zeile 13 dient als neue API-Anfrage, mit der in Zeile 14 detailliertere Informationen über ein Kochbuch im JSON-Format (siehe Auflistung 4.2) abgerufen werden. Der Inhalt der Antwort wird an die Methode `processCookbook(cookbookJSONObject)` (siehe Algorithmus 4.2) übergeben, welche den Crawling Vorgang der aktuellsten Version des Kochbuches aus den übergebenen Informationen implementiert (siehe Zeile 15). Zum Parallelisieren des Crawling Vorgangs kann die Methode `processCookbook` in Threads aufgerufen werden.

Algorithmus 4.1 Crawling Vorgang aller Kochbücher

```
1:  function getAvailableCookbooksFromSupermarket()
2:      startOffset := 0
3:      itemsPerLoop := 1000                                   //Maximum 1000
4:      urlAll := "https://supermarket.chef.io/api/v1/cookbooks?start=" + startOffset + "&items=5000"
5:      totalCookbooks := getJsonFromUrl(urlAll)
6:      totalNumCookbooks := totalCookbooks.getInt("total")
7:      while startOffset < totalNumCookbooks do
8:          urlIteration := "https://supermarket.chef.io/api/v1/cookbooks?start=" + startOffset +
    "&items=" + itemsPerLoop
9:          cookbooksToCrawl := getJsonFromUrl(urlIteration)
10:         cookbooksToCrawlArray := cookbooksToCrawl.getJSONArray("items")
11:         for all cookbook ∈ cookbooksToCrawlArray do
12:             cookbookJSON := cookbooksToCrawlArray.getJSONObject(cookbook)
13:             urlCookbook := cookbookJSON.getString("cookbook")
14:             cookbookJSONObject := getJsonFromUrl(urlCookbook)
15:             processCookbook(cookbookJSONObject)            //Kann parallelisiert werden
16:         end for
17:         startOffset = startOffset + itemsPerLoop
18:     end while
19: end function
```

Algorithmus 4.2 zeigt die Methode `processCookbook`. Diese erhält als Übergabeparameter den Inhalt aus Auflistung 4.2. Zunächst wird das Feld `deprecated` aus den Informationen ausgelesen (siehe Algorithmus 4.2 Zeile 2). Das Feld liefert den booleschen Wert *true*, wenn das Kochbuch veraltet und

Auflistung 4.1 REST-API Antwort auf Abfrage aller Kochbücher
GET https://supermarket.chef.io/api/v1/cookbooks?start=0&items=5000

```
{
"start":0,
"total":3838,
"items":[
      {
      "cookbook_name":"1password",
      "cookbook_maintainer":"jtimberman",
      "cookbook_description":"Installs 1password",
      "cookbook":"https://supermarket.chef.io/api/v1/cookbooks/1password"
      },
      {
      "cookbook_name""":"301",
      ...
}
```

Auflistung 4.2 REST-API Antwort auf Abfrage des Java-Kochbuches
GET https://supermarket.chef.io/api/v1/cookbooks/java

```
{
"name":"java",
"maintainer":"sous-chefs",
"description":"Recipes and resources for installing Java and managing
    certificates",
"category":"Other",
"latest_version":"https://supermarket.chef.io/api/v1/cookbooks/java
    /versions/4.0.0",
"external_url":"https://github.com/sous-chefs/java",
"source_url":"https://github.com/sous-chefs/java",
"issues_url":"https://github.com/sous-chefs/java/issues",
"average_rating":null,"created_at":"2009-10-25T23:51:43.000Z",
"updated_at":"2019-04-24T19:26:45Z","up_for_adoption":null,
"deprecated":false,
"versions":[
      "https://supermarket.chef.io/api/v1/cookbooks/java/versions/4.0.0",
      "https://supermarket.chef.io/api/v1/cookbooks/java/versions/3.2.0",
      ...
      ]
}
```

Auflistung 4.3 REST-API Antwort auf Abfrage des Java-Kochbuches Version 4.0.0
GET https://supermarket.chef.io/api/v1/cookbooks/java/versions/4.0.0

```
1 {
2 "license":"Apache-2.0",
3 "tarball_file_size":35088,
4 "version":"4.0.0",
5 "published_at":"2019-04-24T19:26:45Z",
6 "average_rating":null,
7 "cookbook":"https://supermarket.chef.io/api/v1/cookbooks/java",
8 "file"
      :"https://supermarket.chef.io/api/v1/cookbooks/java/versions/4.0.0/download",
9 "quality_metrics":[{...}],
10 "supports":{
11       "centos":">= 0.0.0",
12       "fedora":">= 0.0.0",
13       ...
14       },
15 "dependencies":{
16       "homebrew":">= 0.0.0",
17       "windows":">= 0.0.0"
18       }
19 }
```

false, wenn das Kochbuch aktuell ist. Veraltete Kochbücher werden durch Zeile 3 ignoriert und nicht heruntergeladen. Bei einem aktuellen Kochbuch, wird das Feld `name` und das Feld `latest_version` aus den übergebenen Daten ausgelesen, welches den Kochbuchname und die URL zu der aktuellsten Version des Kochbuches liefert (Zeile 4 und 5). Mit der URL werden, analog zu den bisherigen Anfragen, Informationen über die aktuellste Version des Kochbuches abgefragt und in einem weiteren JSON-Objekt in Zeile 6 gespeichert. Der Inhalt des JSON-Objekts enthält Detailinformationen zur aktuellsten Version des Kochbuches in der Form, wie sie in Auflistung 4.3 beispielhaft abgebildet ist. Dort findet sich im Feld *version* die Version des Kochbuches und im Feld *file* die URL für den Download des Kochbuches. Algorithmus 4.2 extrahiert aus dem JSON-Objekt (siehe Auflistung 4.3) die URL für den Download des Kochbuches sowie die Version des Kochbuches und übergibt sie zusammen mit dem Kochbuchnamen an die Methode `downloadChefCookbook` (siehe Algorithmus 4.2 Zeile 7-9). Diese verwendet die URL für den Download des Kochbuches und speichert das Kochbuch mit dem Namen und der Version eindeutig in ein Zielverzeichnis. Die aus dem Chef Supermarket heruntergeladen Kochbücher befinden sich in einem komprimierten Ordner und müssen vor der Weiterverwendung entpackt werden. Die Kochbücher werden in ein Zielverzeichnis entpackt und befinden sich am Ende des Crawlingvorgangs auf der lokalen Festplatte. Die gecrawlten Kochbücher können nun von der lokalen Festplatte abgerufen und weiterverarbeitet werden. Für die Weiterverarbeitung wird in Abschnitt 4.3 zunächst das Architekturmodell vorgestellt, in das die Kochbücher transformiert werden.

Algorithmus 4.2 Crawling Vorgang eines Kochbuches

```
1:  function processCookbook(cookbookJSONObject)
2:      cookbookDeprecated = cookbookJSONObject.getBoolean("deprecated")
3:      if !cookbookDeprecated then
4:          cookbookName := cookbookJSONObject.getString("name")
5:          urlLatestVersion = cookbookJSONObject.getString("latest_version")
6:          latestVersionJSONObject = getJsonFromUrl(urlLatestVersion)
7:          cookbookVersion = latestVersionJSONObject.getString("version")
8:          fileUrl = latestVersionJSONObject.getString("file")
9:          downloadChefCookbook(cookbookName, cookbookVersion, fileUrl)
10:     end if
11: end function
```

4.3 Generisches Architekturmodell

Ziel der Arbeit ist die Abbildung der in Chef Kochbüchern enthaltenen Deployment Architekturen auf ein generisches Architekturmodell. In Abschnitt 3.2 wurden verschiedene Ansätze zur Darstellung von Cloud-Anwendungen vorgestellt. Dieser Abschnitt stellt ein Architekturmodell vor, mit dem sich die Deployment Architektur von Anwendungen generisch darstellen lässt. In Abschnitt 4.3.1 werden zuerst die Anforderungen an das Modell erläutert. Anschließend wird in Abschnitt 4.3.2 das generische Architekturmodell zur Abbildung der Deployment Architekturen vorgestellt. Das Architekturmodell wird bewusst allgemein gehalten, damit der präsentierte Ansatz auf alle Deploymenttechnologien, die auf dieses Modell abgebildet werden können, übertragbar ist. Gleichzeitig lässt sich das Modell auf speziellere Architekturmodelle übertragen, welche die den Anforderungen entsprechen, wie zum Beispiel das TOSCA Modell.

4.3.1 Anforderungen

In diesem Abschnitt werden Anforderungen an ein Architekturmodell zur Darstellung der Deployment Architektur einer Anwendung definiert. Diese werden aus dem Aufbau und den Eigenschaften von Softwarearchitekturen abgleitet. Jede Anforderung bekommt eine eindeutige Nummer zugeordnet, um das in Abschnitt 4.3.2 vorgestellte Architekturmodell zu verifizieren.

Die Deployment Architektur einer Cloud-Anwendung kann aus mehreren Komponenten bestehen, die verschiedene Abhängigkeiten untereinander haben. So kann die Installation einer Komponente eine weitere voraussetzen, die zuerst installiert werden muss. Daraus folgt Anforderung 1: Das Modell kann eine Anwendung in Form eines gerichteten Multigraphen darstellen kann.

Der in dieser Arbeit präsentierte Ansatz dient als Basis zur automatischen Kompatibilitätsprüfung der verwendeten Komponenten einer geplanten Softwarearchitektur. Dazu werden Kompatibilitätsinformationen zu den beteiligten Softwarekomponenten benötigt. Abbildung 4.3 zeigt beispielhaft Komponente 2, welche von Komponente 1 abhängt. Dies bedeutet, dass zur Funktionalität von Komponente 2 zuerst Komponente 1 installiert sein muss. Komponente 2 wird also die *Anforderung (A)* zugeordnet, dass Komponente 1 vorher installiert sein muss. Analog dazu wird Komponente 1 ihre eigene *Fähigkeit (F)* zugeordnet, welche sie für andere Komponenten anbietet. Eine Verbindung zwischen zwei Komponenten

besteht nur, wenn die Anforderung von Komponente 2 mit der Fähigkeit von Komponente 1 übereinstimmt. Um diese Informationen generisch darzustellen, müssen sich einer Komponente Fähigkeiten und Anforderungen zuordnen lassen. Daraus lassen sich die Anforderungen 2, 3 und 4 ableiten.

Modellanforderung 1 (MA-1): Deployment Architektur einer Anwendung lässt sich in Form eines gerichteten Multigraphen darstellen. Dabei sind die Knoten des Graphen die Komponenten. Die Beziehungen untereinander werden durch gerichtete Kanten ausgedrückt. Die Richtung bestimmt die Abhängigkeit zu einer Komponente.

Modellanforderung 2 (MA-2): Fähigkeiten und Anforderungen an Knoten. Einer Komponente müssen sich Fähigkeiten und Anforderungen zuordnen lassen.

Modellanforderung 3 (MA-3): Verbindungen nur bei Abhängigkeiten. Verbindungen zu anderen Komponenten bestehen nur, wenn eine Anforderung einer Komponente mit der Fähigkeit der Komponente, zu der die Abhängigkeit besteht, übereinstimmt.

Modellanforderung 4 (MA-4): Automatisierbarkeit der Auflösung von Abhängigkeiten zwischen Komponenten. Die Auflösung der Abhängigkeiten zu anderen Komponenten muss automatisierbar sein. Wenn ein Knoten eine Anforderung hat müssen sich automatisch Knoten finden lassen, die mit ihren Fähigkeiten diese Abhängigkeit auflösen.

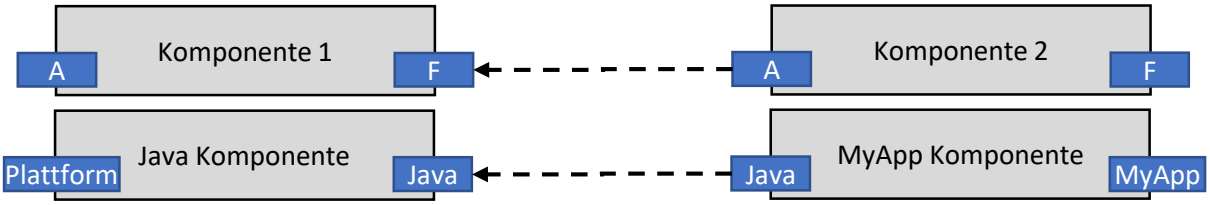

Abbildung 4.3: Verbindung zwischen Komponenten

4.3.2 Generisches Architekturmodell

Dieser Abschnitt stellt ein generisches Architekturmodell vor, mit dem sich die Deployment Architektur einer Anwendung darstellen lässt. Das vorgestellte Architekturmodell ist aus der Topologieschicht des DMMN-Modells von Breitenbücher [Bre16] abgeleitet.

Mit der Topologieschicht des DMMN-Modells lässt sich die Struktur einer Anwendung formal darstellen [Bre16]. Abbildung 4.4 zeigt das daraus abgeleitete generische Architekturmodell zur Darstellung der Deployment Architektur von Anwendungen. Eigenständigen Bestandteile einer Anwendung werden durch die *Komponenten* repräsentiert. Eine *Relation* repräsentiert die Beziehung zwischen genau zwei Komponenten. Die Bedeutung von Komponenten und Relationen wird durch die eindeutigen Typen Komponententyp und Relationentyp spezifiziert [Bre16]. Zur Modellierung einer Architektur werden die Komponenten und Relationen aus den entsprechenden Typen instantiiert. Komponenten oder Relationen, die eine Architektur beschreiben, sind dabei Instanzen ihres Typs. Für diese Arbeit ist der linke Teil von Abbildung 4.4 wichtig, der mit einer gestrichelten Linie umrandet ist. Die extrahierten Komponenten aus Kochbüchern stellen Komponententypen dar, mit denen eine geplante Anwendung überprüft wird die aus Instanzen verschiedener Komponententypen und Relationentypen besteht.

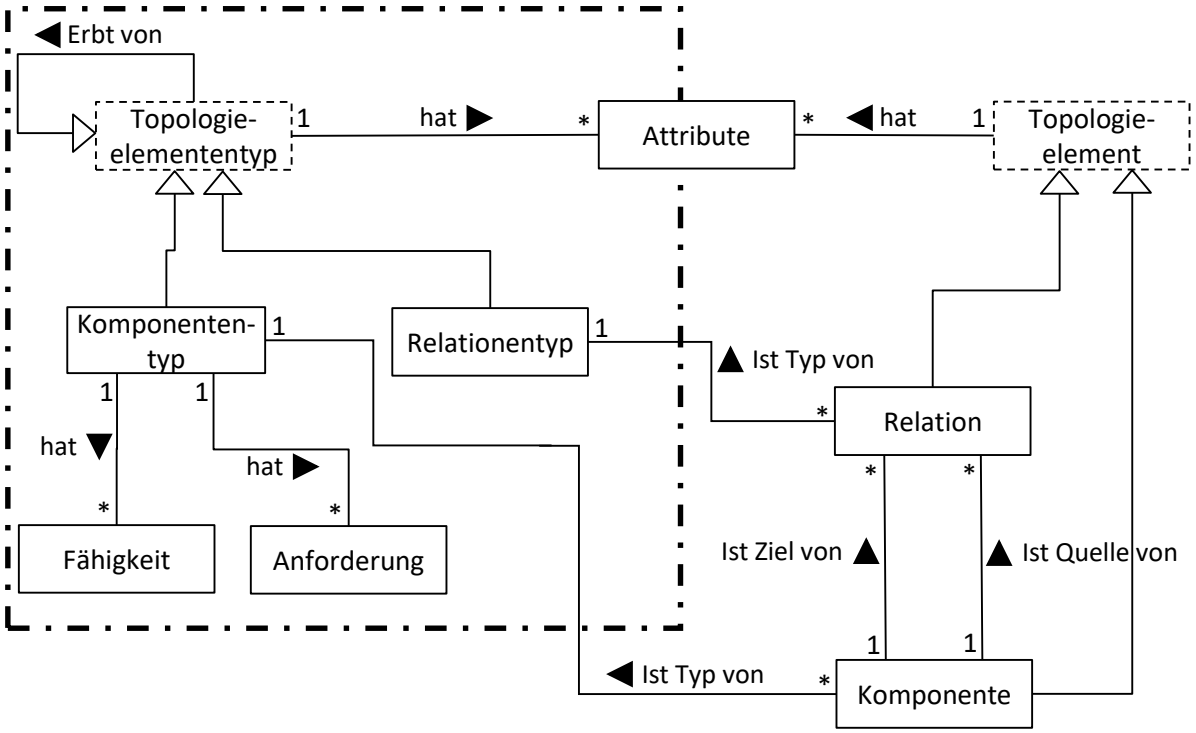

Abbildung 4.4: Generisches Metamodell für Deployment Architekturen (angelehnt an [Bre16])

Diese Arbeit verfolgt das Ziel, die Basis für einen automatischen Kompatibilitätscheck von geplanten Anwendungsarchitekturen zu legen. Wie in den Anforderungen in Abschnitt 4.3.1 beschrieben, kann zwischen zwei Komponenten nur eine Relation bestehen, wenn Anforderungen einer Komponente mit Fähigkeiten der Komponente aufgelöst werden, von der sie abhängt. Ein Komponententyp hat folglich Fähigkeiten und Anforderungen mit denen sich automatisch Relationen zwischen Komponententypen generieren lassen. Der in dieser Arbeit beschriebene Ansatz extrahiert die in Chef Kochbüchern enthaltenen Softwarekomponenten als Komponententypen und ordnet ihnen die entsprechenden Fähigkeiten und Anforderungen zu. Aus dem laufenden Beispiel (siehe Abschnitt 1.3) werden zum Beispiel die Komponententypen MyApp und Java extrahiert (siehe Abbildung 4.3). Dem Komponententyp MyApp werden die im MyApp-Kochbuch extrahierten Fähigkeiten zugeordnet und da das Kochbuch vom Java-Kochbuch abhängt, werden dem MyApp Komponententyp Anforderungen zugeordnet, die sich aus dem Java-Kochbuch extrahieren lassen. Der Java Komponententyp erhält die installierten Komponenten aus dem Kochbuch als Fähigkeiten. Da als einzige Anforderung die zur Plattform besteht, erhält der Java-Komponententyp die entsprechenden Anforderungen auf die Plattform. Die genauen Details werden an dieser Stelle nicht weiter erläutert. Stattdessen wird auf Kapitel 5 verwiesen, in dem die Details der Transformation erläutert werden.

4.4 Allgemeiner Ansatz für die Extraktion der in Kochbüchern enthaltenen Deployment Architekturen

Um die Deployment Architekturen eines Chef Kochbuches auf ein generisches Modell abzubilden, müssen die Informationen dazu zuerst aus dem Kochbuch extrahiert werden. Die Informationen dazu finden sich im Chef-Code in den Dateien und Rezepten eines Kochbuches. Chef-Code wird in Ruby geschrieben und kann folglich komplexe Anweisungen enthalten. Zur Interpretation wird ein spezialisierter Chef Kochbuch Compiler verwendet, der ein lauffähiges Chef Kochbuch in die enthaltenen Deployment Architekturen übersetzt. Dieser Abschnitt stellt den allgemeinen Ablauf des entwickelten Ansatzes für das Extrahieren der Deployment Architekturen aus einem lauffähigen Chef Kochbuch vor. Dabei werden die extrahierten Deployment Architekturen auf das in Abschnitt 4.3.2 vorgestellte generische Architekturmodell abgebildet.

Abbildung 4.5 zeigt die einzelnen Schritte des Chef Kochbuch Compilers, der ein Chef Kochbuch in das enthaltene Architekturmodell übersetzt. Der Ansatz wurde aus dem allgemeinen Aufbau von Compilern nach Aho [Aho03] abgeleitet. Die Schritte werden im Folgenden mit den Zahlen aus Abbildung 4.5 benannt. In Schritt (1) wird das Kochbuch spezifiziert, aus dem die Deployment Architekturen extrahiert werden. Wie in Abschnitt 2.1 erläutert, wird Chef-Code in einer Chef-DSL implementiert, die auf Ruby basiert. Der genaue Aufbau und der Inhalt der Dateien wird in Abschnitt 4.5 erläutert. In den Dateien wird die Architektur eines zu installierenden Szenarios mit Code konfiguriert. Folglich müssen für die Extraktion der Deployment Architekturen die Dateien mit dem relevanten Code analysiert werden. Um den Code in den Dateien zu analysieren, wird dieser in Schritt (2) zuerst in seine Bestandteile zerlegt. Dies geschieht wie im Compilerbau durch eine lexikalische Analyse [Aho03]. Die im Kochbuch enthaltenen Dateien werden entsprechend einer Reihenfolge (siehe Kapitel 5) an den Lexer (2) übergeben, der die lexikalische Analyse durchführt. Der Lexer zerlegt den enthaltenen Code in eine Token-Folge aus den Basiselementen von Ruby Code. Im Compilerbau (siehe Abschnitt 2.3) folgt nach der lexikalischen Analyse die syntaktische Analyse, in der die Token-Folge in einen abstrakten Syntaxbaum übersetzt wird, welcher die grammatische Struktur der Token-Folge darstellt [Aho03]. Die Token werden in Schritt (3) an einen Parser übergeben, der die syntaktische Analyse implementiert. Bei der syntaktischen Analyse, erstellt der Parser aus der Token-Folge einen Syntaxbaum (4), welcher anhand der Grammatikregeln erstellt wird, auf der die Programmiersprache Ruby basiert. Die Grammatik der Chef-DSL basiert auf der Ruby Grammatik [Mat19b]. Weitere Details zur Grammatik von Ruby finden sich in der Datei *parse.y* der Ruby Distribution [Mat19a]. Die Extraktion der Deployment Architekturen des spezifizierten Kochbuches und das Mapping auf das Architekturmodell erfolgt in Schritt (5), in dem der Syntaxbaum semantisch analysiert und die gefundenen Eigenschaften in ein generisches Architekturmodell (6) übersetzt werden. Der Code des Kochbuches wird an dieser Stelle nicht in ausführbaren Maschinencode übersetzt, wie dies bei herkömmlichen Compilern der Fall ist, sondern in die enthaltenen Deployment Architekturen. Der in dieser Arbeit präsentierte Ansatz zur semantischen Analyse weicht deshalb von der semantischen Analyse von Compilern, wie sie in der Literatur, wie beispielsweise bei Aho [Aho03], beschrieben wird, ab. Die Details der semantischen Analyse werden in Kapitel 5 erläutert.

Der in diesem Abschnitt vorgestellte Chef Kochbuch Compiler extrahiert die in Kochbüchern enthaltenen Deployment Architekturen und übersetzt sie in das vorgestellte generische Architekturmodell (siehe Abschnitt 4.3). Um die enthaltenen Deployment Architekturen zu extrahieren, müssen Eigenschaften identifiziert werden, mit denen sich die im Kochbuch enthaltenen Deployment Architekturen beschreiben lassen. In Abschnitt 4.5 werden die Eigenschaften, zur Darstellung der im Kochbuch enthaltenen

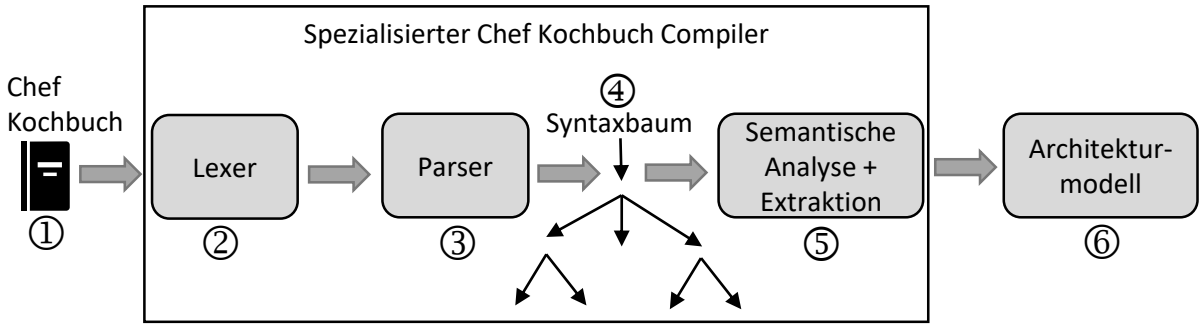

Abbildung 4.5: Chef Kochbuch Compiler

Deployment Architekturen, identifiziert und erläutert, die während der semantischen Analyse extrahiert werden müssen. Nach der Identifikation der relevanten Eigenschaften, wird die semantische Analyse, sprich die Details zur Transformation in das Architekturmodell, in Kapitel 5 erläutert. Das Ergebnis des vorgestellten Kompilierungsvorgangs sind die im spezifizierten Kochbuch enthaltenen Deployment Architekturen (6), in Form des vorgestellten generischen Architekturmodells aus Abschnitt 4.3.

4.5 Identifikation von Eigenschaften

Um die Deployment Architekturen in der semantischen Analyse zu extrahieren, müssen zunächst Eigenschaften identifiziert werden, mit denen sich diese abbilden lassen. In diesem Abschnitt wird der Aufbau von Chef Kochbüchern analysiert. Dabei werden die einzelnen Bestandteile eines Kochbuches untersucht und relevante Eigenschaften identifiziert, nach denen die Dateien analysiert werden müssen. Relevante Eigenschaften sind alle Eigenschaften, die zur Darstellung der enthaltenen Deployment Architektur beitragen. Dazu werden Anforderungen und Fähigkeiten des konfigurierten Szenarios durch das Kochbuch gesucht. Dabei sind die Informationen von Bedeutung, die sich zur Kompatibilitätsprüfung von verwendeten Softwareversionen untereinander verwenden lassen. Zu relevanten Informationen gehören zum Beispiel installierte Softwarekomponenten, vorausgesetzte Softwarekomponenten, ihre Abhängigkeiten untereinander sowie Systemanforderungen.

4.5.1 Chef Kochbuch Aufbau

Ein Kochbuch ist eine Konfigurationseinheit, in der spezifische Aufgaben definiert und konfiguriert werden [Inc19a]. Diese Konfigurationen lassen sich auf einem Knoten ausführen, welcher in den, im Kochbuch beschriebenen, Zustand überführt wird. Ein Chef Kochbuch definiert meistens ein *Szenario* und enthält alles, was zur Unterstützung dieses erforderlich ist. Der Begriff Szenario bedeutet in diesem Kontext die Installation und vollständige Konfiguration eines Dienstes oder einer Anwendung. Es kann beispielsweise ein Java OpenJDK [6] unter Verwendung des Java-Kochbuches[7] installiert werden. Dabei werden durch das Kochbuch die richtigen Softwarepakete für die unterliegende Plattform installiert

[6] https://openjdk.java.net/
[7] https://github.com/sous-chefs/java

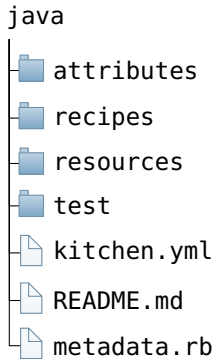

Abbildung 4.6: Verzeichnisstruktur des Java-Kochbuches

und Konfigurationsaufgaben, wie das Anlegen von Verzeichnissen oder Umgebungsvariablen, werden automatisch durchgeführt.

Kochbücher sind in einer vollständig in sich geschlossenen Verzeichnisstruktur organisiert. Sie besteht aus Verzeichnissen und Dateien, die für unterschiedliche Zwecke verwendet werden. Abbildung 4.6 zeigt den allgemeinen Aufbau der Verzeichnisstruktur von Chef Kochbüchern, welche die für diese Arbeit relevanten Dateien und Verzeichnisse in der Verzeichnisstruktur eines Kochbuches zeigt. In den nachfolgenden Abschnitten werden die Dateien und Verzeichnisse erläutert und die für diese Arbeit bedeutsamen Eigenschaften zur Darstellung der enthaltenen Deployment Architekturen identifiziert.

4.5.2 metadata.rb

Auf der obersten Ebene der Verzeichnisstruktur eines Kochbuches befindet sich grundsätzlich die Datei *metadata.rb* [Inc19a]. Diese enthält Informationen, mit denen Kochbücher für einen Netzwerkknoten korrekt bereitgestellt werden. In den Metadaten werden verschiedene Einstellungen wie Name, Beschreibung oder Lizenz des Kochbuches vorgenommen. In einigen der Felder ist es erlaubt, Versionen einzuschränken, wozu die in Tabelle 4.1 gezeigten Operatoren verwendet werden [Inc19a].

Die Versionsabhängigkeit `>= 1.0.0, <2.0.0` lässt sich mit dem pessimistischen Operator in der Form `~>1.0` schreiben. Diese Einschränkung erlaubt alle 1.x Versionen.

Operator	Beschreibung
~>	Pessimistischer Operator zur Festlegung eines Versionsbereichs
=	Gleich
>=	Größer oder gleich
>	Größer
<	Kleiner
<=	Kleiner oder gleich

Tabelle 4.1: Versionsabhängigkeits-Operatoren [Inc19a]

Auflistung 4.4 Kommandos in Metadaten

```
1 # Kochbuch Name
2 name 'myapp'
3
4 # Version
5 version '1.0.0'
6
7 # Abhaengigkeit zu weiteren Kochbuechern
8 depends 'java', '>= 4.0.0'
```

Nachfolgend werden für diese Arbeit relevante Inhalte der Metadaten erläutert. Für vollständige Informationen zu den Möglichkeiten in den Metadaten wird auf die Chef Dokumentation verwiesen [Inc19a].

In den Metadaten eines Kochbuches ist das Feld `name` immer erforderlich und definiert den Namen eines Kochbuches. Das Kommando `name` ist beispielhaft in Zeile 2 von Auflistung 4.4 dargestellt. Der Name des Kochbuches identifiziert das Kochbuch, was beim Auflösen von Abhängigkeiten wichtig ist. Der Name entspricht meistens dem Namen des Szenarios oder der Anwendung, die durch das Kochbuch konfiguriert wird. Das Java-Kochbuch[8] dient beispielsweise der Installation von Java oder das Openssh-Kochbuch[9] der Installation und Konfiguration von Openssh[10]. Der Name des Kochbuches wird für das Auflösen von Abhängigkeiten und zur Benamung der extrahierten Komponenten benötigt.

Im Feld `version` wird die aktuelle Version des Kochbuches festgelegt. Die Versionen von Chef Kochbüchern werden nach der semantischen Versionierung [Pre19] vergeben. Die Versionsnummer folgt immer einem drei-Nummern-Prinzip, wie in Zeile 5 von Auflistung 4.4 gezeigt. Die Version wird für eine eindeutige Versionierung der extrahierten Komponenten extrahiert.

Die Versionsnummer ist vor allem für das Feld `depends` notwendig, mit dem Abhängigkeiten zu anderen Kochbüchern angegeben werden. Wird damit ein Kochbuch als Abhängigkeit eingetragen, ist das Vorhandensein eines Kochbuches mit übereinstimmendem Namen und Version auf dem Chef-Server erforderlich. Die Dateien des Kochbuches lassen sich dann im eigentlichen Kochbuch wiederverwenden. Da die Abhängigkeiten bei fehlerhaften Inhalten falsch aufgelöst werden, ist der Inhalt dieses Feldes sensibel. Auf das `depends` Feld sind alle Versionseinschränkungs-Operatoren anwendbar. Findet man eine Versionsbeschränkung wie im gezeigten Beispiel in Zeile 8 von Auflistung 4.4 in den Metadaten eines Kochbuches, bedeutet dies, dass im Kochbuch Rezepte des Kochbuches *java* in einer Version größer oder gleich 4.0.0 verwendet werden können. Diese Informationen sind für das Auflösen von Abhängigkeiten notwendig.

Um einem Kochbuch eine kurze Beschreibung des konfigurierten Szenarios zu geben, wird das Kommando `description` verwendet (siehe Zeile 2 von Auflistung 4.5). Für eine ausführlichere Beschreibung wird das Kommando `long_description` verwendet, welches in zwei Arten Anwendung findet. Entweder wird eine Datei mit der ausführlichen Beschreibung gelesen, wie im gezeigten Beispiel aus der Readme-Datei des Kochbuches (siehe Zeile 5 von Auflistung 4.5), oder die Beschreibung wird direkt in das Feld eingebettet wie in Zeile 8 von Auflistung 4.5 gezeigt. Da in der langen Beschreibung beliebiger Inhalt

[8] https://github.com/sous-chefs/java
[9] https://github.com/chef-cookbooks/openssh
[10] https://www.openssh.com/

Auflistung 4.5 Beschreibung eines Kochbuches festlegen

```
1  # Kurze Beschreibung
2  description 'Ein Kochbuch, das MyApp installiert und konfiguriert.'
3
4  # Lange Beschreibung
5  long_description IO.read(File.join(File.dirname(__FILE__), 'README.md'))
6
7  # Alternative lange Beschreibung
8  long_description <<-EOH
9  = Beschreibung:
10 Hier koennte eine Beschreibung stehen
11 = Anforderungen:
12 Debian oder Ubuntu praeferiert.
13 ....
14 EOH
```

Auflistung 4.6 Unterstütze Plattform festlegen

```
1  # Unterstuetzt alle Ubuntu Versionen
2  supports 'ubuntu'
3
4  # Unterstuetzt spezielle Ubuntu Versionen
5  supports 'ubuntu', '>= 14.04'
6  supports 'ubuntu', '= 16.04'
7
8  # Mehrere Plattformen mit Schleife
9  %w(aix amazon arch centos fedora freebsd opensuse opensuseleap oracle redhat scientific smartos
       suse ubuntu zlinux).each do |os|
10       supports os
11 end
```

definiert werden kann, folgt die lange Beschreibung keinen Regeln. Das Extrahieren von Informationen aus der langen Beschreibung wird deshalb aus dem Inhalt der Arbeit ausgegrenzt.

Mit dem Kommando `supports` wird deklariert, welche Plattformen ein Kochbuch unterstützt. Unterstützt das Kochbuch zum Beispiel die Plattform Ubuntu, wird dies definiert wie in Zeile 2 von Auflistung 4.6. Da dort keine Versionsbeschränkung angegeben ist, bedeutet dies, dass alle Ubuntu Versionen unterstützt werden. Um zu zeigen, dass alle Versionen größer oder gleich 14.04 unterstützt werden, wird eine entsprechende Versionsbeschränkung angegeben (siehe Zeile 5 von Auflistung 4.6). Wird nur Ubuntu 16.04 unterstützt, geschieht dies analog dazu mit dem geeigneten Operator (siehe Zeile 6 von Auflistung 4.6). Zur Deklaration von mehreren unterstützten Versionen, wird das `supports` Kommando mehrfach verwendet oder innerhalb einer Schleife verwendet (siehe Zeile 8-11 von Auflistung 4.6).

4.5.3 Chef Attribute

Chef Attribute sind Details eines Knotens, der mit einem Kochbuch konfiguriert wird [Inc19a]. Sie werden vom Chef-Client verwendet, um zu verstehen, was der aktuelle Status des Knotens ist, wie der

Status des Knotens am Ende des vorherigen Client-Laufs war, oder wie der Status des Knotens am Ende des aktuellen Client-Laufs ist [Inc19a]. Im Kontext der statischen Analyse eines Kochbuches, wie in dieser Arbeit, ermöglichen die Attribute zu verstehen, wie ein Knoten durch das Kochbuch konfiguriert wird. Ein Knoten ist die Maschine, die mit dem Kochbuch konfiguriert wird. Da der konfigurierte Knoten bei der statischen Analyse eines Kochbuches nicht zur Verfügung steht, steht der aktuelle Status des Knotens sowie der Zustand nach dem letzten Chef-Client-Lauf nicht zur Verfügung.

Attribute werden dem Chef-Client an verschiedenen Orten zur Verfügung gestellt. Es existieren Attribute, wie die Plattform oder die Plattformfamilie, die vom Zielknoten abhängen und von Ohai zu Beginn eines jeden Chef-Client-Laufs ermittelt werden [Inc19a]. Ohai ist ein Werkzeug zur Erfassung von Systemkonfigurationsdaten wie plattformabhängigen Attributen, die dem Chef-Client für die Verwendung in Kochbüchern zur Verfügung gestellt werden [Inc19a]. Die von Ohai automatisch erfassten Attribute stehen deshalb bei der statischen Analyse eines Kochbuches nicht zur Verfügung. Zusätzlich können Attribute in Kochbüchern definiert werden. Dies geschieht in Attributdateien, welche sich im *attributes* Unterordner eines Kochbuches befinden, oder in den Rezepten selbst. Weitere Attribute werden in Umgebungen und Rollen definiert [Inc19a]. Diese befinden sich im *chef-repo* Verzeichnis auf dem Arbeitsplatz, auf dem das ChefDK installiert wurde [Inc19a]. Diese stehen bei der statischen Analyse eines Kochbuches nicht zur Verfügung und deshalb kann auf diese Attribute bei der Analyse eines Chef Kochbuches nicht zurückgegriffen werden.

Attribute kommen, wie in Tabelle 4.2 gezeigt, in sechs verschiedenen Typen vor, welche jeweils eine unterschiedliche Priorität aufweisen. Attribute kommen in Attributdateien, Rezepten, Umgebungen und Rollen vor. Die Kombination von Attributtypen (siehe Tabelle 4.2) und Quellen ermöglicht bis zu 15 verschiedene Möglichkeiten, welche sich eindeutig priorisieren lassen. Da ein Chef Kochbuch keine Rollen und Umgebungen definiert, kann bei dessen statischer Analyse nur auf die Attribute aus Attributdateien und Rezepten zugegriffen werden. Aus diesem Grund werden die Prioritäten von Attributen aus Umgebungen und Rollen in der folgenden Auflistung bewusst außer Acht gelassen. Die Priorisierung der Attribute erfolgt von eins mit der niedrigsten Priorität bis zu 11 mit der höchsten Priorität, wie folgt [Inc19a]:

1. Ein default Attribut in einer Attributdatei.
2. Ein default Attribut in einem Rezept.
3. Ein force_default Attribut in einer Attributdatei.
4. Ein force_default Attribut in einem Rezept.
5. Ein normal Attribut in einer Attributdatei.
6. Ein normal Attribut in einem Rezept.
7. Ein override Attribut in einer Attributdatei.
8. Ein override Attribut in einem Rezept.
9. Ein force_override Attribut in einer Attributdatei.
10. Ein force_override Attribut in einem Rezept.
11. Ein automatic Attribut, das von Ohai erfasst wird.

Attribut Typ	Beschreibung
default	Ein *default* Attribut wird bei jedem Start des Chef-Client-Laufs automatisch zurückgesetzt und hat die niedrigste Priorität.
force_default	Mit *force_default* wird sichergestellt, dass in Kochbüchern deklarierte Attribute eine höhere Priorität wie *default* Attribute von Rollen oder Umgebungen aufweisen.
normal	Ein *normal* Attribut ist eine Einstellung, die im Knoten Objekt persistiert und Vorrang über *default* Attribute hat.
override	Ein *override* Attribut wird beim Start jedes Chef-Client-Laufs automatisch zurückgesetzt und hat eine höhere Priorität wie default, force_default und normal.
force_override	Analog zu *force_default*.
automatic	Ein *automatic* Attribut enthält Daten, die von Ohai bei Beginn jedes Chef-Client-Laufs identifiziert werden. Automatische Attribute lassen sich nicht bearbeiten und haben die höchste Priorität.

Tabelle 4.2: Typen von Chef Attributen [Inc19a]

Auflistung 4.7 Implizite und nicht implizite Verwendung des Knotenobjekts

```
1  % Implizite Verwendung des Knotenobjekts
2  default['java']['jdk_version'] = '8'
3
4  % Knotenobjekt ist Teil des Attributs
5  node.default['java']['jdk_version'] = '8'
```

Wie zu Beginn des Abschnitts erläutert, lassen sich Attribute in Attributdateien deklarieren. Diese befinden sich im *attributes* Verzeichnis eines Kochbuches. Wenn ein Kochbuch auf einem Knoten ausgeführt wird, wird zuerst die *default.rb* Attributdatei geladen wenn diese existiert. Anschließend werden alle Attribute aus den weiteren Attributdateien in lexikalischer Reihenfolge geladen. Auflistung 4.7 zeigt einen Ausschnitt aus der *default.rb* Attributdatei des Java-Kochbuches [sou19], anhand dem der Aufbau von Chef Attributen erläutert wird. Chef Attribute werden dem Knotenobjekt (node) zugeordnet, welches alle notwendigen Informationen über den konfigurierten Knoten enthält. In Attributdateien wird das Knotenobjekt entweder implizit verwendet (siehe Zeile 2 von Auflistung 4.7), oder es wird explizit als Teil des Attributs definiert (siehe Zeile 5). Die Chef Attribute aus den Zeilen 2 und 6 sind deshalb identisch.

Eine Attributzuweisung in einem Chef Kochbuch folgt festen Regeln. Diese lässt sich wie folgt verallgemeinern: `Attributtyp ['Kochbuchname'] (['Attribut-Dateiname'])? ['Attributname']+`. Ein Attribut besteht aus einem Attributtyp gefolgt von Literalen, die sich in eckigen Klammern befinden. Dabei repräsentiert der erste Literal den Namen des Kochbuches, in dem sich die Attributdatei befindet. Das folgende Literal fungiert als Name der Datei, in dem sich das Attribut befindet. Befindet sich das Attribut in der *default.rb* Attributdatei, ist das zweite Literal optional. Das letzte Literal repräsentiert den eigentlichen Attributnamen.

Im Beispiel aus Auflistung 4.8 handelt es sich bei den Attributen in den Zeilen 2 und 3 um das gleiche Attribut, welches in unterschiedlicher Schreibweise deklariert ist. Da sich das Attribut in der Datei *default.rb* befindet, ist der Ort optional. Dies hat die gleiche Bedeutung wie das zweite Attribut in Zeile 3,

Auflistung 4.8 Zusammenhang Attributname und Attributdatei

```
1  % Folgende zwei Attribute sind in der default.rb Attributdatei.
2  default['java']['jdk_version'] = '8'
3  default['java']['default']['jdk_version'] = '8'
4
5  % Folgendes Attribut ist in der windows.rb Attributdatei
6  default['java']['windows']['jdk_version'] = '8'
7
8  case node['platform_family']
9  when 'windows'
10   default['java']['install_flavor'] = 'windows'
11 when 'mac_os_x'
12   default['java']['install_flavor'] = 'homebrew'
13 else
14   default['java']['install_flavor'] = 'openjdk'
15 end
```

Attribut	Beschreibung
`node['platform']`	Die Plattform auf dem der Knoten läuft.
`node['platform_family']`	Die Familie der Plattform, auf dem der Knoten läuft.
`node['platform_version']`	Die Version der Plattform, auf dem der Knoten läuft.
`node['kernel']['machine']`	Der Kernel des Betriebssystems.

Tabelle 4.3: Automatische Attribute [Inc19a]

in der das gleiche Attribut mit expliziter Ortsangabe deklariert ist. Man liest: Attribut mit Typ *default* im Java-Kochbuch in der Datei *default.rb* des *attributes* Unterverzeichnisses mit Attributnamen *jdk_version*. Dem Attribut wird in allen Fällen der String *'8'* zugewiesen. Analog dazu befindet sich das Attribut aus Zeile 6 (siehe Auflistung 4.8) in der Datei *windows.rb* im *attributes* Verzeichnis des Java-Kochbuches.

Da die Chef-DSL auf Ruby basiert, kann alles, was mit Ruby ausgeführt werden kann, bei Attributdefinitionen verwendet werden, wie beispielsweise If-Anweisungen oder Switch-Case-Anweisungen. Ein Beispiel dafür liefert Auflistung 4.8 in den Zeilen 8-15, welches ebenfalls aus dem Java-Kochbuch [sou19] stammt. Abhängig von der Plattformfamilie wird dem Attribut `['java']['install_flavor']` ein unterschiedlicher Wert zugewiesen. Dabei fällt in Zeile 8 das Attribut `node['platform_family']` auf, welches von der bisherigen Definition abweicht. Es handelt sich um den Zugriff auf ein automatisches Attribut, welches vom Betriebssystem des Knotens abhängt, auf dem das Kochbuch ausgeführt wird. Automatische Attribute sind Informationen zu dem Knoten selbst, wie die Plattform, IP-Adresse oder Kernelmodule. Automatische Attribute werden zu der Laufzeit von Ohai erkannt und lassen sich somit in Kochbüchern verwenden. Häufig verwendete automatische Attribute finden sich in Tabelle 4.3. Weitere werden in den Chef Dokumenten erläutert [Inc19a]. Da automatische Attribute von dem konfigurierten Knoten abhängen, kann bei der statischen Analyse eines Kochbuches auf diese nicht zugegriffen werden.

Auflistung 4.9 Chef DSL Methode include_recipe

```
1  # Ruft windows.rb Rezept aus dem Java-Kochbuch auf
2  include_recipe 'java::windows'
3
4  # Ruft default.rb Kochbuch aus MyApp-Kochbuch auf
5  include_recipe 'myapp::default'
6
7  # Ruft default.rb Kochbuch aus MyApp-Kochbuch auf
8  include_recipe 'myapp'
```

4.5.4 Rezepte

Ein *Rezept* stellt das grundlegendste Element in einem Chef Kochbuch dar, in dem die Konfigurationsaufgaben deklariert werden [Inc19a]. Rezepte müssen Bestandteil eines Kochbuches sein und befinden sich im *recipes* Verzeichnis eines Kochbuches. Da Code in Rezepten in Ruby[11] geschrieben wird, enden diese auf **.rb*.

Rezepte lassen sich in anderen Rezepten aufrufen. Dies ist kochbuchübergreifend möglich. Der Aufruf eines weiteren Rezeptes in einem Rezept erfolgt mit der Methode `include_recipe 'recipe'`. Dadurch wird das aufgerufene Rezept, exakt an der Stelle, an dem die Methode aufgerufen wird, ausgeführt. Die Funktionsweise wird mit einem beispielhaften Ausruf aus dem Java-Kochbuch verdeutlicht (siehe Zeile 2 von Auflistung 4.9). Der Rezeptaufruf erfolgt durch Übergabe eines Parameters, bestehend aus Kochbuchname, in dem das Rezept enthalten ist und dem Namen des Rezeptes getrennt durch zwei Doppelpunkte: `'java::windows'`. In dem Beispiel wird das *windows.rb* Rezept aus dem Java-Kochbuch aufgerufen. Der Inhalt des aufgerufenen Rezeptes wird exakt an dieser Stelle ausgeführt. Danach wird in Zeile 5 das *default.rb* Kochbuch aus dem MyApp-Kochbuch ausgeführt. Der Aufruf des *default* Rezeptes eines Kochbuches kann in einer Kurzversion, wie in Zeile 8, erfolgen. Dabei wird der Methode der Name des Kochbuches übergeben. Wird ein Rezept eines anderen Kochbuches verwendet, hat die Datei *metadata.rb* einen entsprechenden Eintrag. In diesem Fall wäre dort die Anweisung `depends 'myapp'` enthalten. Dies wurde in Abschnitt 4.5.2 erläutert.

Der Inhalt von Chef Rezepten ist eine Sammlung von *Ressourcen* gepaart mit Ruby Code [Inc19a]. Eine Ressource ist ein Ruby Block, bestehend aus einem Typ (type), einem Name (name), einer oder mehreren Eigenschaften (attribute) und einer oder mehreren Aktionen (action) (siehe Zeile 2-5 von Auflistung 4.10). Mit Ressourcen werden Konfigurationsaufgaben, wie das Installieren von Softwarepaketen oder das Konfigurieren von Verzeichnissen, durchgeführt. Die Funktionsweise der verschiedenen, für diese Arbeit relevanten Ressourcen, werden in Abschnitt 4.5.5 erläutert. In Rezepten werden zudem Chef Attribute (siehe Abschnitt 4.5.3) wie in den Attributdateien definiert. Dies geschieht mit der expliziten Verwendung des Knotenobjekts, wie in Zeile 11 von Auflistung 4.10 gezeigt. Darüber hinaus kann in Rezepten alles benutzt werden, was mit der Sprache Ruby möglich ist [Inc19a]. Es lassen sich beispielsweise Variablen definieren, wie in Zeile 8, oder es lassen sich Ausdrücke (if, unless, usw.), Schleifen-Anweisungen, Switch-Case-Anweisungen sowie Arrays verwenden (siehe Zeile 14-27 von Auflistung 4.10).

[11] https://www.ruby-lang.org/de/

Auflistung 4.10 Chef Rezept Aufbau

```
1  # Ressource
2  type 'name' do
3    attribute 'value'
4    action :type_of_action
5  end
6
7  # Variablen
8  package_name = 'openjdk-8-jdk'
9
10 # Chef Attribute
11 node.default['java']['package_name'] = package_name
12
13 # Ruby Code
14 if (node['platform' == 'ubuntu')
15   # Mache etwas wenn node['platform'] ubuntu ist
16 else
17   # Mach andere Dinge
18 end
19
20 case node['platform']
21 when 'centos', 'redhat', 'fedora', 'suse'
22   # Mache etwas wenn node['platform'] centos, redhat, fedora oder suse ist.
23 when 'debian', 'ubuntu'
24   # Mache etwas wenn node['platform'] debian oder ubuntu ist
25 when 'arch'
26   # Mache etwas wenn node['platform'] arch ist
27 end
```

4.5.5 Ressourcen

Eine *Ressource* ist eine Anweisung zur Konfiguration, mit der ein Konfigurationselement in den gewünschten Zustand gebracht wird [Inc19a]. Eine Ressource erklärt alle dazu erforderlichen Schritte. In der Chef DSL gibt es verschiedene Ressourcentypen, wie *package*[12], *template*[13] oder *service*[14]. Diese unterscheiden sich in ihrer Konfigurationsaufgabe. Mit der *package*-Ressource werden Softwarepakete installiert, deinstalliert oder aktualisiert. Mit der *template*-Ressource lassen sich Dateien an einem konfigurierten Ort aus einer Quelldatei erstellen sowie die Berechtigungen konfigurieren und mit der *service*-Ressource lassen sich Dienste managen [Inc19a].

Eine Ressource ist ein Ruby Block, wie in Auflistung 4.11 Zeile 2-5 gezeigt. Dieser besteht aus einem Typ (type), einem Namen (name) und einer oder mehrere Eigenschaften (attribute) denen Werte zugewiesen werden. Jede Chef Ressource hat zudem eine oder mehrere mögliche Aktionen (action), mit der die Aktion der Ressource definiert wird. Jeder Ressourcentyp hat dabei seine eigenen Eigenschaften und Aktionen. Auflistung 4.11 demonstriert dies in Zeile 8-11 am Beispiel der *package*-Ressource. Da die

[12] https://docs.chef.io/resource_package.html
[13] https://docs.chef.io/resource_template.html
[14] https://docs.chef.io/resource_service.html

Auflistung 4.11 Aufbau von Chef Ressourcen Blöcken

```
1  # Aufbau von Ressourcen
2  type 'name' do
3    attribute 'value'
4    action :type_of_action
5  end
6
7  # Installation des myapp Pakets in der Version 1.1
8  package 'myapp' do
9    version '1.1'
10   action :install
11 end
```

Aktion *:install* deklariert ist, installiert die Ressource in der gezeigten Konfiguration das beispielhafte Paket *myapp* in der Version 1.1.

Da Ressourcen, deren Aufgabe das Installieren und Deinstallieren von Softwarepaketen ist, die Deployment Architektur beeinflussen, beschränkt sich diese Arbeit auf diese Ressourcen. Ressourcen wie *template* oder *service* werden zu Konfigurationszwecken von Verzeichnissen und Services eingesetzt und haben dementsprechend auf die Deployment Architektur keinen Einfluss. Zur Verfeinerung des Architekturmodells lässt sich der, im Anschluss an dieses Kapitel, vorgestellte Ansatz, analog zu den vorgestellten Ressourcen, erweitern.

Ressourcen werden in Chef Ressourcen und benutzerdefinierte Ressourcen kategorisiert. Chef Ressourcen sind von Chef zur Verfügung gestellte Ressourcen wie die vorgestellte *package*-Ressource. Benutzerdefinierte Ressourcen sind eine Erweiterung von Chef zur Definition von eigene Ressourcen. Die Verwendung erfolgt analog zu Chef Ressourcen.

Chef Ressourcen

Dieser Abschnitt erläutert die für diese Arbeit bedeutsamen Chef Ressourcen. Dabei liegt der Fokus auf Ressourcen welche einen Einfluss auf die Architektur des installierten Szenarios haben, wie beispielsweise die Installation von Softwarekomponenten. Wenn Softwarepakete installiert, deinstalliert oder aktualisiert werden, wird die *package*-Ressource verwendet [Inc19a]. Sie dient als Basisressource für weitere Ressourcen, die zu der Paketverwaltung auf verschiedenen Plattformen verwendet werden. Weitere Ressourcen für das Installieren von Softwarepaketen für bestimmte Plattformen finden sich in Abbildung 4.7, welche der Chef Dokumentation entnommen sind [Inc19a]. In der Chef Dokumentation wird darauf hingewiesen, die *package*-Ressource so oft wie möglich zu verwenden, obwohl es möglich ist jede dieser spezifischen Ressourcen für Installationen zu verwenden [Inc19a]. Trotz allem ist in besonderen Fällen die Verwendung einer spezialisierten Ressource unumgänglich. Folglich muss die semantische Analyse für alle Abwandlungen der *package*-Ressource sensibilisiert werden.

Auflistung 4.12 zeigt für diese Arbeit relevante Teile der Syntax der *package*-Ressource. Dabei wurden nur die für die vorliegende Arbeit relevanten Teile der Syntax abgebildet. Die vollständige Syntax der *package*-Ressource findet sich in der Chef Dokumentation [Inc19a]. Das Kommando `package` in Zeile 1 identifiziert die Ressource. Wird eine der vorgestellten spezialisierten Ressourcen, wie *apt_package*, verwendet, wird

- apt_package
- bff_package
- cab_package
- chef_gem
- chocolatey_package
- dnf_package
- dpkg_package
- freebsd_package
- gem_package
- homebrew_package
- ips_package
- macports_package
- openbsd_package
- pacman_package
- paludis_package
- portage_package
- rpm_package
- smartos_package
- solaris_package
- windows_package
- yum_package
- zypper_package

Abbildung 4.7: Auf der package-Ressource basierte spezialisierte Ressourcen

Auflistung 4.12 Allgemeiner Aufbau der package-Ressource [Inc19a]

```
1 package 'name' do
2   package_name        String, Array # defaults to 'name' if not specified
3   version             String, Array
4   action              Symbol # defaults to :install if not specified
5 end
```

an dieser Stelle der entsprechende Ressourcenname eingesetzt. Das Feld `'name'` entspricht dem Namen der Ressource. Dieser wird als Standardwert für die Eigenschaft `package_name` verwendet, wenn diese Eigenschaft nicht explizit angegeben ist (siehe Zeile 2). Die Eigenschaft `version` stellt die Version des Paketes dar, welches installiert oder aktualisiert wird und ist optional (siehe Zeile 3). Die Eigenschaft `action` gibt an, welche Aktion, mit der Ressource, durchgeführt werden soll (siehe Zeile 4). Die möglichen Aktionen der *package*-Ressource sind:

1. `:install` zur Installation eines Pakets in der definierten Version.
2. `:nothing`, um das Paket vorzumerken und erst durch eine Benachrichtigung eine Aktion durchzuführen.
3. `:purge` zum Entfernen eines Pakets (nur Debian Plattformen).
4. `:upgrade`, um ein Paket zu installieren oder sicherzustellen, dass die neueste Version installiert ist.
5. `:remove` zum Entfernen eines Pakets.
6. `:reconfig`, um ein Paket neu zu konfigurieren. Dazu ist aber eine Datei notwendig.

Alle nicht genannten Eigenschaften der *package*-Ressource dienen zwar zur Konfiguration, haben allerdings keinen für diese Arbeit relevanten Einfluss auf die Deployment Architektur.

Auflistung 4.13 Aufbau von benutzerdefinierten Ressourcen [Inc19a]

```
1  # Name der Ressource
2  resource_name : example
3
4  # Eigenschaften der Ressource
5  property :property_name, RubyType, default: 'value'
6
7  # Aktionen definieren
8  action :install do
9    # Ein Mix aus Ruby Code und Chef Ressourcen
10 end
11
12 action :another_action_name do
13   # Ein Mix aus Ruby Code und Chef Ressourcen
14 end
```

Benutzerdefinierte Ressourcen

Benutzerdefinierte Ressourcen sind eine Erweiterung von Chef, mit der eigene Ressourcen deklariert werden [Inc19a]. Sie sind in Ruby-Dateien deklariert und befinden sich im *resources*-Verzeichnis eines Kochbuches. Die Verwendung der benutzerdefinierten Ressourcen ist analog zu Chef Ressourcen.

Benutzerdefinierte Resssourcen haben einen Namen, der aus dem Kochbuch und aus dem Dateinamen, in dem die Ressource deklariert wurde, besteht, wobei ein Unterstrich sie voneinander trennt. Befindet sich die Ressource aus Auflistung 4.13 beispielsweise im MyApp-Kochbuch und trägt den Dateinamen *example.rb* wird die Ressource über den Typ `myapp_example` aufgerufen. Die Verwendung benutzerdefinierter Ressourcen ist dabei analog zu der von Chef Ressourcen (siehe Auflistung 4.11). Der Name kann durch die Methode `resource_name` anpasst werden (siehe Zeile 2 von Auflistung 4.13), weshalb diese Ressource über den Typ `example` aufgerufen wird.

In einer benutzerdefinierten Ressource werden die benötigten Eigenschaften deklariert (siehe Zeile 5 von Auflistung 4.13). Diese deklarierten Eigenschaften sind vergleichbar mit beispielsweise der `version`-Eigenschaft der erläuterten *package*-Ressource.

Eine benutzerdefinierte Ressource muss mindestens eine Aktion definieren, die in einem Aktionsblock definiert ist (siehe Auflistung 4.13 Zeile 8-10). Dabei ist der Name der Aktion ein Wert, welcher der Aktionseigenschaft (action) zugewiesen werden kann, wenn die Ressource in einem Rezept verwendet wird. Innerhalb einer Aktion wird durch einen Mix aus Ruby Code und Chef Ressourcen die Konfigurationsaufgabe deklariert.

Für diese Arbeit ist an dieser Stelle relevant, wenn durch eine benutzerdefinierte Ressource ein Softwarepaket installiert wird. Dies wird wie in Rezepten durch die Verwendung der *package*-Ressource oder einer Anwandlung davon deklariert. Der Möglichkeiten zur Deklaration von benutzerdefinierten Ressourcen sind vielfältig, weshalb für detailliertere Informationen zu dem Aufbau von benutzerdefinierten Ressourcen auf die Chef Dokumentation verwiesen wird [Inc19a].

Auflistung 4.14 Aufbau der kitchen.yml Datei [sou19]

```
1  # Teil 1
2  driver:
3        name: vagrant
4
5  # Teil 2
6  provisioner:
7        name: chef_zero
8        deprecations_as_errors: false
9
10 # Teil 3
11 platforms:
12       - name: centos-7
13       - name: debian-9
14       - name: freebsd-11
15       - name: fedora-29
16       - name: ubuntu-14.04
17       - name: ubuntu-16.04
18
19 # Teil 4
20 suites:
21       - name: openjdk-8
22       run_list:
23             - recipe[test::openjdk8]
```

4.5.6 kitchen.yml

In einem Kochbuchverzeichnis kann die Datei *kitchen.yml* existieren (siehe Abbildung 4.6). Diese ist optional und existiert deshalb nicht in jedem Kochbuch. Die Datei *kitchen.yml* dient zur Konfiguration von *Kitchen*, welches eine Testumgebung zum isolierten Ausführen von Infrastrukturcode auf einer oder mehreren Plattformen bereitstellt [Inc19b]. Dazu werden in der Datei *kitchen.yml* Plattformen und Rezepte angegeben und über jede Kombination von Plattformen und angegebenen Rezepten getestet. Da das Kochbuch auf den deklarierten Plattformen getestet wird, sind die deklarierten Plattformen aus der Datei *kitchen.yml* für diese Arbeit von Interesse.

Die Datei *kitchen.yml* besteht aus vier Teilen [Mar15], die in Auflistung 4.14, am Beispiel des Java-Kochbuches, dargestellt werden. Im ersten Teil wird der Treiber konfiguriert, den Kitchen zum Starten von virtuellen Maschinen für die Tests verwendet. In Teil zwei wird definiert, wie Chef in den virtuellen Maschinen für die Tests verwendet wird (siehe Zeile 6-8 von Auflistung 4.14). Teil drei definiert die Plattformen, auf denen das Kochbuch getestet wird (siehe Zeile 11-17 von Auflistung 4.14). Dabei steht unter der Eigenschaft *name* die Plattform und die Version getrennt durch einen Bindestrich. Teil vier definiert die Testsuiten (siehe Zeile 20-23 von Auflistung 4.14). Am Beispiel des Java-Kochbuches in Auflistung 4.14 wurde eine Testsuite mit dem Namen *openjdk-8* definiert. Jede Testsuite hat eine sogenannte *run-list*, in der die auszuführenden Rezepte angegeben sind. Am gezeigten Beispiel ist in der Liste das *opendjk8.rb* Rezept aus dem Testkochbuch des *test*-Verzeichnisses des Java-Kochbuches angegeben (siehe Abbildung 4.6).

5 Details zur Transformation in ein Architekturmodell

In diesem Abschnitt wird der Ansatz zur technischen Transformation von einem Chef Kochbuch in die enthaltenen Deployment Architekturen erläutert. Dazu wird die Vorgehensweise erläutert, wie sich die in Abschnitt 4.5 identifizierten Eigenschaften auf das vorgestellte Architekturmodell abbilden lassen. Dabei wurden, anhand der Struktur von Chef Kochbüchern, Eigenschaften identifiziert, mit denen sich die Deployment Architektur darstellen lässt. Abbildung 5.1 zeigt das daraus abgeleitete Modell eines Chef Kochbuches mit den identifizierten Eigenschaften, die extrahiert werden. Das Modell enthält zusätzliche, für die Extraktion der Deployment Architekturen notwendige, Eigenschaften, wie beispielsweise Chef Attribute und Ruby Variablen.

Bei der Konfiguration eines Knotens mit einem Kochbuch werden die im Kochbuch deklarierten Aufgaben ausgeführt. Konfigurationsaufgaben, wie das Installieren eines Softwarepaketes, können sich abhängig von der unterliegenden Plattform unterscheiden. Wird zum Beispiel Java auf einem windowsbasierten Betriebssystem installiert, hat das installierte Softwarepaket einen anderen Namen als bei der Installation von Java auf einem Ubuntu Betriebssystem. Unterstützt ein Kochbuch nur eine Plattform, sind alle Konfigurationsaufgaben auf diese Plattform zugeschnitten. Häufig unterstützt ein Kochbuch mehrere Plattformen und konfiguriert die Aufgaben abhängig von der zugrundeliegenden Plattform. Dazu ermittelt Ohai zu Beginn die für das Betriebssystem spezifischen, automatischen Attribute wie Plattformversion oder Plattformfamilie. Diese werden in den Kochbüchern verwendet, um den Knoten unter der Verwendung von Ruby Anweisungen abhängig von der Plattform zu konfigurieren. Aus den Metadaten (siehe Abschnitt 4.5.2) lassen sich die Plattformen ableiten, die ein Chef Kochbuch unterstützt. Bei der Extraktion der Deployment Architektur wird deshalb zwischen den verschiedenen Plattformen differenziert. Ein Kochbuch hat folglich mehrere Kochbuch-Konfigurationen, wie in Abbildung 5.1 dargestellt. Jede Kochbuch-Konfiguration bekommt als Anforderung jeweils eine der Plattformen zugeordnet, die das Kochbuch unterstützt. Die Anzahl der extrahierten Kochbuch-Konfigurationen ist somit abhängig von der Anzahl der unterstützten Plattformen.

Kochbücher werden über einen Namen identifiziert, der an die Kochbuch-Konfigurationen weitergegeben wird. Damit lässt sich zuordnen, zu welchem Kochbuch die extrahierte Kochbuch-Konfiguration gehört. Analog dazu wird jeder Kochbuch-Konfiguration die Version des Kochbuches zugeordnet.

Die Rezepte des analysierten Kochbuches bestehen aus Ressourcen (siehe Abschnitt 4.5.5). In Rezepten erkannte Ressourcen, die eine Softwarekomponente installieren, werden als Fähigkeit einer Kochbuch-Konfiguration abgebildet. Werden mehrere Komponenten durch die Rezepte installiert, hat eine Kochbuch-Konfiguration mehrere Fähigkeiten. Installierte Softwarekomponenten werden nur als Fähigkeit identifiziert, wenn die Ressource in den eigenen Rezepten definiert ist. Eigene Rezepte oder Ressourcen sind diejenigen, die in der eigenen Verzeichnisstruktur des Kochbuches abgelegt sind. Wie in

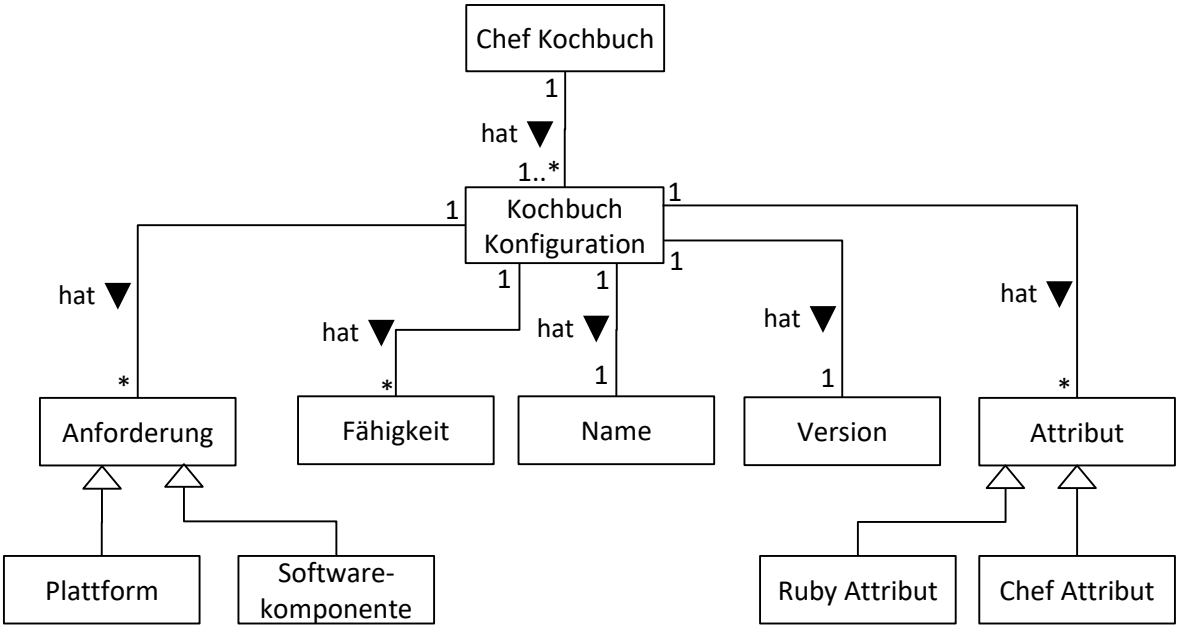

Abbildung 5.1: Kochbuchmodell

Abschnitt 4.5.4 erläutert, lassen sich Rezepte eines Kochbuches in Rezepten fremder Kochbücher verwenden. Dies wird in Abbildung 5.2 anhand des laufenden Beispiels mit dem MyApp-Kochbuch verdeutlicht. Dieses verwendet Rezepte des Java-Kochbuches und hat deshalb eine Abhängigkeit zum Java-Kochbuch. Wird ein Rezept des Java-Kochbuches in einem Rezept des MyApp-Kochbuches aufgerufen, wird der beinhaltete Code an der Stelle ausgeführt (siehe Abschnitt 4.5.4). Deklarieren die aufgerufenen Java-Rezepte die Installation einer Softwarekomponente, wird diese nicht den Fähigkeiten der MyApp-Kochbuch-Konfigurationen, sondern den Anforderungen der MyApp-Kochbuch-Konfigurationen, zugeordnet. Abbildung 5.2 zeigt diese Zuordnung schematisch. Softwarekomponenten, die durch Rezepte des Java-Kochbuches innerhalb des MyApp-Kochbuches installiert werden, werden den Anforderungen (A) der MyApp-Kochbuch-Konfigurationen zugeordnet. Softwarekomponenten aus den Rezepten des MyApp-Kochbuches werden den Fähigkeiten (F) der MyApp-Kochbuch-Konfigurationen zugeordnet. Einer Kochbuch-Konfiguration werden somit Anforderungen, die vom Typ Plattform oder Softwarekomponente sind und Fähigkeiten, die vom Typ Softwarekomponente sind, zugeordnet (siehe Abbildung 5.1).

Es ist möglich, dass das Java-Kochbuch weitere Abhängigkeiten zu Kochbüchern aufweist. Somit können aus Rezepten des Java-Kochbuches die Rezepte der Abhängigkeit aufgerufen werden. Die Inhalte dieser Rezepte werden nicht den Anforderungen der MyApp-Kochbuch-Konfigurationen zugeordnet, sondern ignoriert. Abhängigkeiten wie das Java-Kochbuch werden, wie in Abbildung 5.2 dargestellt, rekursiv, in einer eigenen Transformation, in die enthaltenen Kochbuch-Konfigurationen, übersetzt. Abhängigkeiten des Java-Kochbuches werden somit in der separaten Analyse des Java-Kochbuches den Anforderungen der Java-Kochbuch-Konfigurationen zugeordnet. Die in Abbildung 5.2 dargestellte Transformation des MyApp-Kochbuches wird folglich in zwei Schritten ausgeführt. Im ersten Schritt werden die MyApp-Kochbuch-Konfigurationen aus dem MyApp-Kochbuch extrahiert. Wenn die Analyse des MyApp-Kochbuches beendet ist, wird eine weitere Analyse auf dem Java-Kochbuch ausgeführt. Diese läuft analog zu der Analyse des MyApp-Kochbuches ab. Dies beutetet, dass installierte Kom-

ponenten durch Rezepte des Java-Kochbuches den Fähigkeiten der Java-Kochbuch-Konfigurationen zugeordnet werden und eventuelle Abhängigkeiten zu weiteren Kochbüchern den Anforderungen der Java-Kochbuch-Konfigurationen zugeordnet werden. Da sich dies analog zu der Transformation des MyApp-Kochbuches verhält, wurde dies in Abbildung 5.2 nicht dargestellt. Sind im Java-Kochbuch weitere Abhängigkeiten zu anderen Kochbüchern angegeben, werden diese ebenfalls rekursiv analysiert, bis ein Kochbuch erreicht ist, das keine weiteren Abhängigkeiten hat. Im gezeigten Beispiel in Abbildung 5.2 ist der Vorgang nach der Analyse des Java-Kochbuches beendet. Das Ergebnis sind mehrere MyApp-Kochbuch-Konfigurationen, die alle eine unterschiedliche Plattform als Anforderung haben. Sie haben als Fähigkeiten die installierten Softwarekomponenten des MyApp-Kochbuches und als Anforderungen die installierten Komponenten durch die aufgerufenen Rezepte des Java-Kochbuches. Analog dazu werden bei der Transformation des Java-Kochbuches die Java-Kochbuch-Konfigurationen extrahiert, welche entsprechende Fähigkeiten und Anforderungen haben.

Zur Konfiguration werden in Rezepten Chef Attribute verwendet. Diese werden entweder in Attributdateien (siehe Abschnitt 4.5.3) oder in den Rezepten (siehe Abschnitt 4.5.4) deklariert. Definierte Chef Attribute sind innerhalb eines Kochbuches global, was bedeutet, dass der Zugriff ist an jeder Stelle, an der Chef Attribute erlaubt sind, möglich ist. Um die Konfiguration anzupassen ohne den Code in Rezepten zu ändern, werden Chef Attribute verwendet. Darüber hinaus werden sie für die plattformabhängige Zuweisung von Werten verwendet. Dies kann zum Beispiel der Name einer Softwarekomponente wie Java sein, die bei der Installation auf einem Ubuntu Betriebssystem eine andere Bezeichnung hat, als bei der Installation auf einem windowsbasierten Betriebssystem. Um Chef Attribute bei Benutzung korrekt aufzulösen, werden diese jeder Kochbuch-Konfiguration zugeordnet (siehe Abbildung 5.1). Da die Chef-DSL auf Ruby basiert, treten im Code von Kochbüchern Ruby Variablen auf. Um die Werte der Ruby Variablen bei möglicher Verwendung abzurufen, werden diese, analog zu den Chef Attributen, der jeweiligen Kochbuch-Konfiguration zugeordnet. Chef Attribute und Ruby Variablen werden hauptsächlich zu der korrekten Extraktion der gesuchten Informationen verwendet und dienen nicht direkt der Darstellung der Deployment Architektur.

Abbildung 5.1 zeigt die identifizierten Eigenschaften, die aus einem Chef Kochbuch zur Extaktion und Modellierung der Deployment Architekturen extrahiert werden. Die Eigenschaften sind auf verschiedene Dateien innerhalb eines Kochbuches verteilt. Abbildung 5.2 zeigt das konzeptionelle Mapping von einem Chef Kochbuch auf das generische Architekturmodell am laufenden Beispiel (siehe Abschnitt 1.3). Dabei deuten die Pfeile an, welche Informationen aus den Dateien extrahiert werden. Aus den Metadaten wird beispielsweise der Name und die Version des Kochbuches extrahiert und den Eigenschaften der MyApp-Konfigurationen zugeordnet. Zudem werden unterstützte Plattformen extrahiert. Für jede dieser extrahierten Plattformen wird die MyApp-Konfiguration dupliziert, sodass jede Kochbuch-Konfiguration eine andere Plattform als Anforderung hat. Um die extrahierten Plattformen aus den Metadaten mit Versionen zu verfeinern, wird die Datei *kitchen.yml* in die Transformation einbezogen. Wenn die Datei *kitchen.yml* vorhanden ist, werden aus dieser weitere Plattformen extrahiert. Chef Attribute werden aus den Attributdateien extrahiert und den Eigenschaften der Kochbuch-Konfigurationen zugeordnet. Aus den Rezepten des Kochbuches werden die Softwarekomponenten den Fähigkeiten oder Anforderungen der Kochbuch-Konfigurationen zugeordnet.

Die Dateien eines Kochbuches müssen in einer definierten Reihenfolge analysiert werden, damit die Deployment Architekturen korrekt extrahiert werden. Aus der Analyse des Aufbaus von Kochbüchern in Abschnitt 4.5 lassen sich zwei Ansätze für die Extraktion der enthaltenen Deployment Architekturen ableiten. Die erste Möglichkeit ist eine Analyse, bei der das Kochbuch in einer ähnlichen Art und

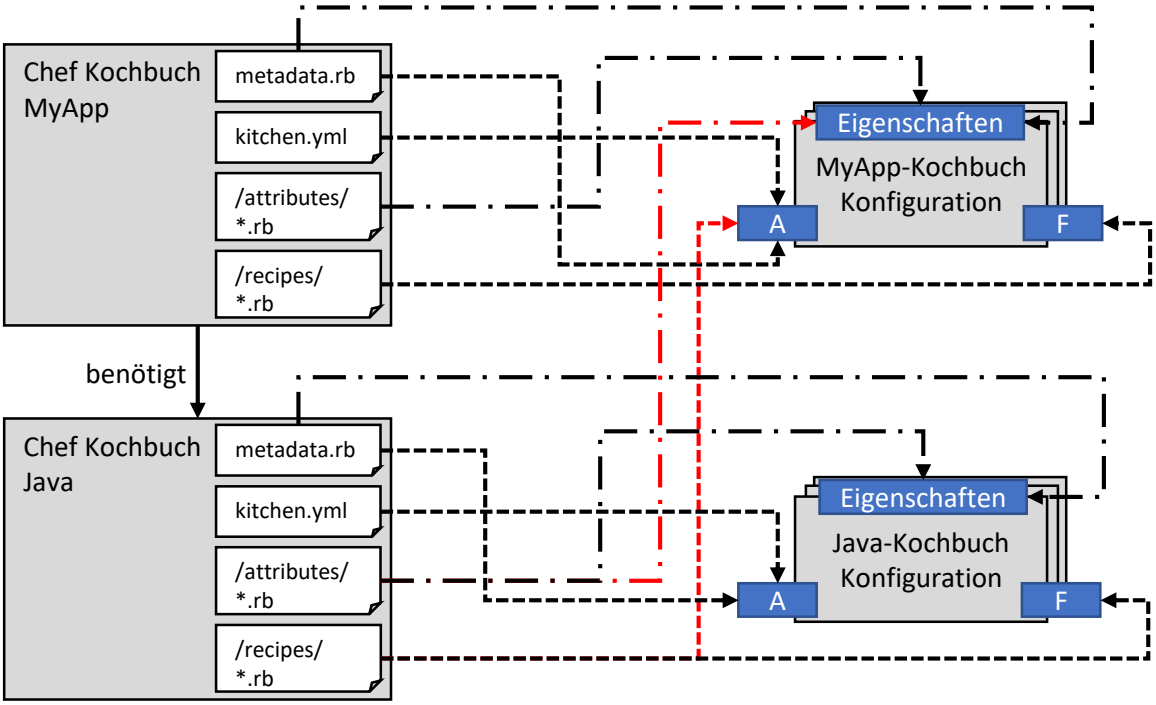

Abbildung 5.2: Konzeptionelles Mapping der im MyApp-Kochbuch enthaltenen Deployment Architekturen auf das Architekturmodell

Weise kompiliert wird, wie es bei der Konfiguration eines physischen Knotens mit Chef passiert. Der Ansatz dieser Transformation wird in Abschnitt 5.1.1 präsentiert. Zusätzlich lässt sich ein zweiter Ansatz ableiten. Dieser ist eine Transformation ausgehend von den Integrationstests aus der Datei *kitchen.yml*. Da Integrationstests ein optionaler Bestandteil eines Kochbuches sind, ist diese Art der Transformation nur möglich, wenn die Datei *kitchen.yml* im Kochbuch existiert. In den Tests werden die Konfigurationsaufgaben über verschiedene Plattformen getestet. Dabei wird das Kochbuch zur Konfiguration von virtuellen Knoten verwendet, sowie das Ergebnis der Konfiguration überprüft. Der Ablauf ist derselbe wie bei der Konfiguration echter Knoten. Folglich baut der Ansatz auf Basis der Integrationstests auf dem Ablauf der normalen Transformation auf. Dieser optionale Ansatz wird in Abschnitt 5.1.2 vorgestellt.

5.1 Ablauf der Transformation

Zu Beginn von Kapitel 5 wurde der allgemeine Ansatz für das Mapping der in Chef Kochbüchern enthaltenen Deployment Architekturen auf das vorgestellte generische Architekturmodell (siehe Abschnitt 4.3) vorgestellt. Chef Kochbücher bestehen aus mehreren Dateien, wobei die für die vorliegende Arbeit relevanten Dateien in Abschnitt 4.5 erläutert wurden. Um die Deployment Architektur aus Kochbüchern zu extrahieren, müssen diese Dateien in einer bestimmten Reihenfolge analysiert werden.

Zuerst müssen grundlegende Informationen über das Kochbuch aus den Metadaten ausgelesen werden. Anschließend werden die extrahierten Plattforminformationen mit denen der Datei *kitchen.yml* verfei-

nert. Um Attributaufrufe in den Rezepten aufzulösen, müssen danach die Attributdateien kompiliert werden. Die Rezepte werden deshalb als letztes übersetzt. Aus dem Aufbau von Kochbüchern wurden zwei Möglichkeiten dazu abgeleitet. In diesem Abschnitt wird der Ablauf beider Möglichkeiten erläutert. Die erste Möglichkeit wird in Abschnitt 5.1.1 und die zweite in Abschnitt 5.1.2 erläutert.

5.1.1 Normale Transformation

Dieser Abschnitt beschreibt den Ablauf der normalen Transformation. Dieser ist ähnlich zu dem Ablauf, wie er bei der Konfiguration eines Knotens durch ein Kochbuch ist. Abbildung 5.3 zeigt den allgemeinen Ablauf der Transformation. Dabei wird vorausgesetzt, dass das zu analysierende Kochbuch in der vorgestellten Form (siehe Abbildung 4.6) verfügbar ist, weil andernfalls die benötigten Dateien nicht gefunden werden.

Zu Beginn der Analyse wird überprüft, ob die Datei *metadata.rb* auf der obersten Ebene der Verzeichnisstruktur (siehe Abbildung 4.6) vorhanden ist. Da Informationen, wie Abhängigkeiten zu anderen Kochbüchern, unterstützte Plattformen und der Name des Kochbuches, nicht extrahiert werden können, wenn die Datei *metadata.rb* nicht vorhanden ist, wird die Analyse in diesem Fall beendet. Wenn die Datei *metadata.rb* im Kochbuch enthalten ist, wird diese kompiliert. Die extrahierten Informationen werden in einem Kochbuch-Objekt gespeichert, welches eine Instanz des vorgestellten Kochbuchmodells ist (siehe Abbildung 5.1). Der Ablauf des Kompilierungsvorgangs der Metadaten wird in Abschnitt 5.2 detailliert beschrieben.

Nachdem die Metadaten kompiliert wurden, wird überprüft, ob auf der obersten Ebene des Kochbuchverzeichnisses die Datei *kitchen.yml* vorhanden ist (siehe Abschnitt 4.5.6). Diese ist optional und die Datei existiert nur im Kochbuch, wenn das Kochbuch Integrationstests beinhaltet. Wenn die *kitchen.yml* im Kochbuch enthalten ist, wird diese kompiliert. Die extrahierten Informationen werden in das Kochbuch-Objekt integriert, das bei der Extraktion der Informationen aus den Metadaten erstellt wurde. Der Ablauf des Kompilierungsvorgangs der *kitchen.yml* Datei, sowie der Vorgang für das Extrahieren der relevanten Informationen, wird in Abschnitt 5.3 erläutert.

Nachdem die Dateien *metadata.rb* und *kitchen.yml* analysiert und die relevanten Informationen extrahiert wurden, folgt die Überprüfung des Vorhandenseins von Attributdateien in der Verzeichnisstruktur des Kochbuches. Dabei wird auf das Vorhandensein von Attributdateien, die auf *.rb enden, überprüft, die sich im *attributes*-Verzeichnis (siehe Abbildung 4.6) des Kochbuches befinden. Wenn das kompilierte Kochbuch Abhängigkeiten zu weiteren Kochbüchern hat, werden deren Attribut-Verzeichnisse ebenfalls überprüft. Wird beispielsweise das MyApp-Kochbuch aus dem laufenden Beispiel (siehe Abschnitt 1.3) kompiliert, wird das *attributes*-Verzeichnis des MyApp-Kochbuches auf Attributdateien überprüft. Da das MyApp-Kochbuch von dem Java-Kochbuch abhängt, wird das *attributes*-Verzeichnis des Java-Kochbuches ebenfalls auf Attributdateien überprüft. Existieren Attributdateien, werden diese kompiliert und die enthaltenen Chef Attribute in das Kochbuch-Objekt extrahiert. Die Reihenfolge und der Ablauf der Extraktion der Chef Attribute aus den Attributdateien wird in Abschnitt 5.5 erläutert.

Nachdem alle Attributdateien kompiliert sind und die Attribute in das Kochbuch-Objekt extrahiert wurden, wird überprüft, ob sich die Datei *default.rb* im *recipes*-Verzeichnis des Kochbuches befindet. Die Datei *default.rb* ist das Standard-Rezept des Kochbuches. Wenn das Rezept vorhanden ist, wird der Ablauf der Konfiguration per Konvention vom *default.rb*-Rezept gestartet. Befindet sich im Kochbuch keine *default.rb*-Datei, ist nicht ersichtlich, welches Rezept das Start-Rezept ist. Diese Information ist in

diesem Fall in der Readme-Datei des Kochbuches angegeben, welche aus unstrukturiertem Text besteht, wodurch sich diese Information dort nicht auslesen lässt. Ist das *default.rb*-Rezept nicht vorhanden wird die Analyse deshalb abgebrochen. Ausgehend vom Start-Rezept können wiederum weitere Rezepte aufgerufen werden. Der Kompilierungsvorgang von Rezepten, ausgehend vom *default.rb*-Rezept, wird in Abschnitt 5.6 detailliert erläutert.

Nachdem die Rezepte, ausgehend vom *default.rb*-Rezept, kompiliert wurden und die extrahierten Informationen, analog zu den vorherigen Schritten zu dem Kochbuch-Objekt hinzugefügt wurden, ist die Analyse des Kochbuches abgeschlossen. Das MyApp-Kochbuch aus dem laufenden Beispiel ist an dieser Stelle in seine enthaltenen Komponententypen transformiert, welche jeweils einer Kochbuch-Konfiguration entsprechen (siehe Abschnitt 4.3.2). Da das Kochbuch eine Abhängigkeit zum Java-Kochbuch hat, wird das Java-Kochbuch ebenfalls in einer eigenen Transformation kompiliert. Dieses Vorgehen wurde in Abbildung 5.2 schemenhaft dargestellt. Das Java-Kochbuch wird analog zu dem erläuterten Ablauf (siehe Abbildung 5.3) kompiliert und in Java-Kochbuch-Konfigurationen übersetzt. Dabei wird die Annahme getroffen, dass abhängige Kochbücher über das Rezept *default.rb* aufgerufen werden. Tatsächlich können auch andere Rezepte des Kochbuches aufgerufen werden. Da Rezeptaufrufe in plattformabhängigen Bedingungen vorkommen können, erhöht dies die Komplexität des Ansatzes deutlich, was aus Zeitgründen in dieser Arbeit nicht weiter behandelt wird.

5.1.2 Testfall Transformation

Chef Kochbücher können die optionale Datei *kitchen.yml* enthalten. Mit dieser lassen sich Kochbücher automatisch für jede Kombination von deklarierten Plattformen und Testsuiten testen. Die Datei besteht aus vier Teilen (siehe Abschnitt 4.5.6). Auflistung 5.1 zeigt Teil vier, der aus einer Liste mit dem Namen `suites` besteht. Die Liste enthält eine Sammlung von Tests, mit denen verschiedene Aspekte eines Kochbuches getestet werden. Ein Test besteht mindestens aus einem Namen (siehe Zeile 2 und 5 in Auflistung 5.1) sowie aus der sogenannten `run_list` (siehe Zeile 3-4 in Auflistung 5.1). In dieser werden die Rezepte angegeben, die in dem Test ausgeführt und getestet werden. Im abgebildeten Beispiel wird für den Test mit dem Namen *openjdk-8* das Rezept *openjdk8* aus dem Test-Kochbuch angegeben, welches sich im *test*-Verzeichnis eines Kochbuches befindet. In den aufgerufenen Rezepten des Test-Kochbuches wird ein Testfall geschaffen, in dem die eigentlichen Rezepte des Kochbuches aufgerufen werden. Testfälle können folglich von der Standardkonfiguration abweichen. Am Beispiel des Java-Kochbuches wird in der Standardkonfiguration Java 8 installiert. Durch Anpassung der Chef Attribute lässt sich mit dem Kochbuch zusätzlich Java 6, 7 und 11 konfigurieren. Für den Anwender finden sich Informationen dazu in der Readme-Datei des Kochbuches, die aus unstrukturiertem Text besteht und nicht analysiert wird. Da das Java-Kochbuch Tests für jede unterstützte Java-Version besitzt, lassen sich alternativ die Informationen auch aus den Integrationstests des Java-Kochbuches ableiten.

Der vorgeschlagene Ansatz für die Extraktion der Deployment Architekturen aus den Tests eines Kochbuches gleicht zu einem Großteil der normalen Transformation aus Abschnitt 5.1.1. Dementsprechend kann auch der vorgestellte Ablauf der normalen Transformation aus Abbildung 5.3 weitgehend übernommen werden. Die Transformation weicht bei der Kompilierung der Datei *kitchen.yml* und bei der Annahme des Startrezeptes von der normalen Transformation ab.

Dieser Absatz erläutert die Abweichungen zu der normalen Transformation aus Abschnitt 5.1.1, welche die Transformation der Datei *kitchen.yml* betreffen. Jeder Test muss eine `run_list` haben, in der die

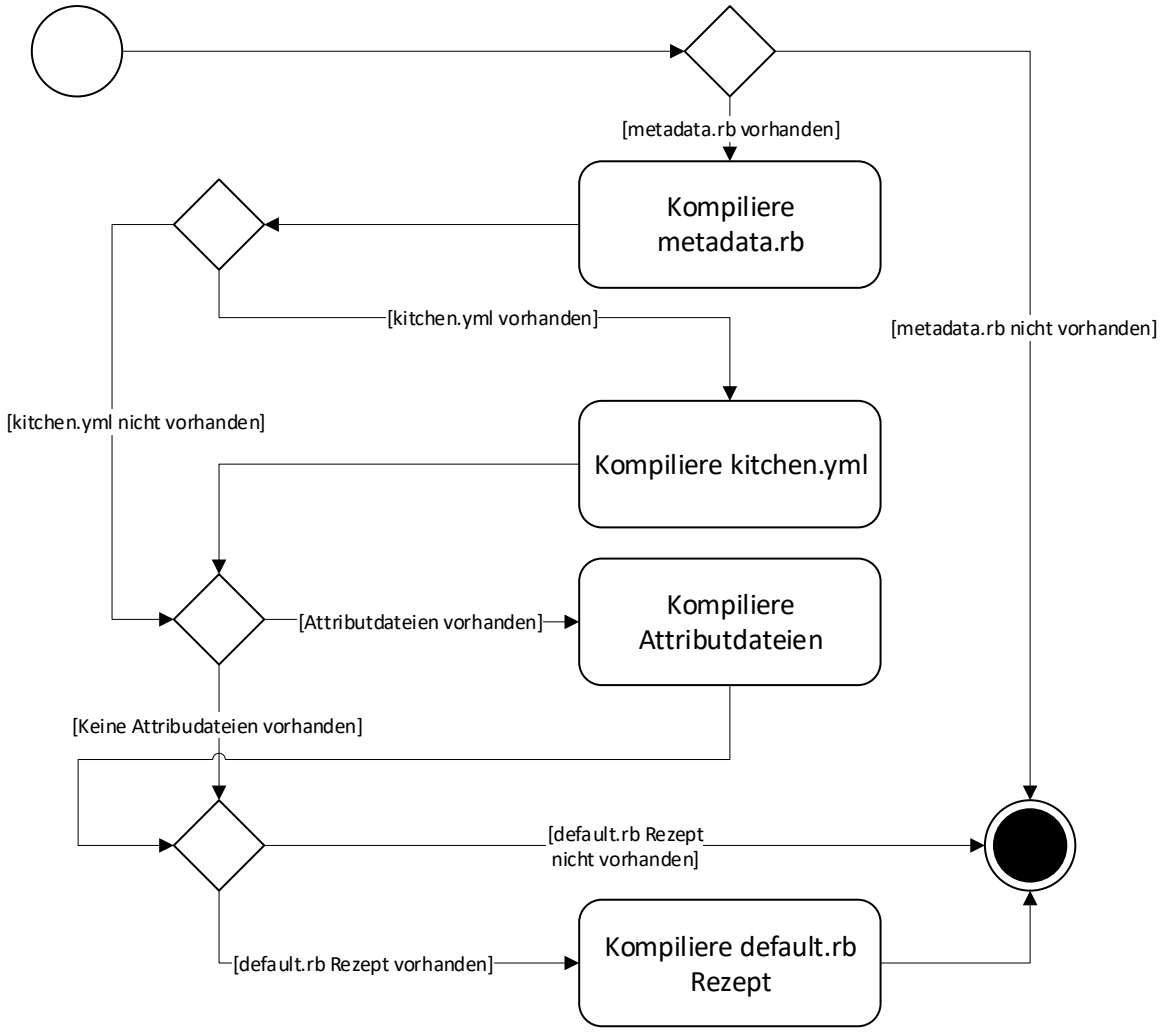

Abbildung 5.3: Ablauf der normalen Transformation

ausgeführten Rezepte deklariert sind. Am Beispiel von Auflistung 5.1 ist beim Test mit dem Namen *openjdk-8* das Rezept *test::openjdk8* angegeben. In der Liste können auch mehrere Rezepte enthalten sein. Die Namen der Rezepte müssen extrahiert werden und markieren den Startpunkt der Transformation bei den Rezepten. Da das Rezept *default.rb* das Startrezept ist, wenn nichts anderes angegeben ist und es keine weiteren Anhaltspunkte zu den aufgerufenen Rezepten gibt, wird bei der normalen Transformation das Rezept *default.rb* als Startrezept verwendet. Bei der Transformation auf Basis der Tests wird dies durch die aus der Liste extrahierten Rezepte ersetzt. Dieses Vorgehen müsste für jeden Testfall wiederholt werden. Die vorgestellte Möglichkeit für die Extraktion der enthaltenen Deployment Architekturen auf Basis der Tests eines Kochbuches bietet den Vorteil, dass sich weitere lauffähige Konfigurationen extrahieren lassen. Am Beispiel des Java-Kochbuches werden aus der normalen Transformation Konfigurationen mit Java 8 als Fähigkeiten extrahiert. Durch die Analyse der Tests können zusätzliche Konfigurationen mit Java 6, Java 7 und Java 11 als Fähigkeiten extrahiert werden. Die Möglichkeit basiert weitgehend auf der vorgestellten normalen Transformation in Abschnitt 5.1.1. Der einzige Unterschied zu der normalen

Auflistung 5.1 Testsuiten in Datei kitchen.yml (Teil 4)

```
1 suites:
2 - name: openjdk-8
3       run_list:
4             - recipe[test::openjdk8]
5 - name: openjdk-7
6       run_list:
7             - ...
```

Transformation besteht in der Extraktion der ausgeführten Rezepte aus der Datei *kitchen.yml*. Im Gegensatz zu der Testfall Transformation verwendet die normale Transformation das Rezept *default.rb* als Startrezept. Die Testfälle lassen sich somit analog zu der normalen Transformation in die enthaltene Deployment Architektur übersetzen. Diese Methode hat einen Nachteil, dass nicht klar ist, welches Szenario in dem Test überprüft wird. Bei den Tests kann nicht davon ausgegangen werden, dass das Szenario des Kochbuches mit unterschiedlichen Einstellungen getestet wird. Die Tests enthalten häufig spezielle Integrationstests, die in fehlerhaften extrahierten Informationen resultieren würden. Eine Transformation aller Testfälle ohne Kenntnis des getesteten Szenarios ist deshalb wenig sinnvoll. Bei den Testfällen muss eine Vorauswahl getroffen werden, bei denen sichergestellt werden kann, dass die Testfälle mit dem Szenario des Kochbuches übereinstimmen. Die Auswahl der geeigneten Testfälle wird in dieser Arbeit nicht weiter behandelt.

5.2 Metadaten

Die Extraktion der Deployment Architekturen aus einem Kochbuch besteht, wie in Abbildung 5.3 dargestellt, aus mehreren Schritten. Dieser Abschnitt erläutert den Kompilierungsvorgang der Metadaten, die sich in der Datei *metadata.rb*, auf der obersten Verzeichnisebene eines Kochbuches, befinden. Der Inhalt der Datei *metadata.rb* enthält Informationen, anhand derer sich Kochbücher auf Knoten korrekt bereitstellen lassen [Inc19a]. Der Aufbau der Metadaten wurde bereits in Abschnitt 4.5.2 erläutert.

Während der Transformation wird das Kochbuch in ein Modell übersetzt, wie in Abbildung 5.1 gezeigt. Das Modell wird in einem Kochbuch-Objekt abgebildet, welches eine Instanz von dem Modell ist. In dem Kochbuch-Objekt werden die extrahierten Informationen gespeichert. Wie in Abbildung 5.1 gezeigt, besteht ein Kochbuch aus mindestens einer Kochbuch-Konfiguration. Zu Beginn der Transformation ist das Kochbuch-Objekt eine leere Instanz, die keine Informationen gespeichert hat. Gleichzeitig enthält es eine leere Kochbuch-Konfiguration.

In Abschnitt 4.5.2 wurden die Ausdrücke erläutert, mit denen in den Metadaten Eigenschaften deklariert werden. Es handelt sich dabei um Befehle, mit denen eine Eigenschaft des Kochbuches deklariert wird. Die erlaubten Kommandos folgen der Grammatik von Ruby und besitzen alle den gleichen Aufbau. Auflistung 5.2 zeigt eine Übersicht über die Kommandos, die bei der vorgestellten semantischen Analyse der Metadaten extrahiert werden. Der Aufbau der Ausdrücke besteht immer aus einem Kommando und mindestens einem Argument: `Kommando arg (,arg)*`. Im ersten Argument steht immer der Wert, der durch das Kommando gesetzt wird. Hat das Kommando weitere Argumente werden diese getrennt

Auflistung 5.2 Kommandos die aus metadata.rb extrahiert werden

```
1  # Name des Kochbuches
2  name 'myapp'
3
4  # Beschreibung
5  description 'Installs and configures MyApp'
6
7  # Version
8  version '1.0.0'
9
10 # Unterstuetzte Plattformen
11 %w(ubuntu centos fedora smartos suse).each do |os|
12   supports os
13 end
14
15 # Unterstuetzt spezielle Ubuntu Version
16 supports 'ubuntu', '= 16.04'
17
18 # Abhaengigkeit zum Java-Kochbuch
19 depends 'java', '>= 4.0.0'
```

durch ein Komma angegeben. Auflistung 5.2 zeigt einen Auszug der Metadaten des MyApp-Kochbuches aus dem laufenden Beispiel (siehe Abschnitt 1.3).

Im Folgenden wird anhand des Auszugs beispielhaft erläutert, wie der Kompilierungsvorgang der Metadaten ausgeführt wird. Das Kommando *name* (siehe Zeile 2 in Auflistung 5.2) ist laut der Dokumentation erforderlich und somit immer in den Metadaten enthalten [Inc19a]. Das Kommando hat immer nur ein Argument, welches den Namen des Kochbuches festlegt. Der Wert wird den Eigenschaften der Kochbuch-Konfigurationen zugeordnet, wie in Abbildung 5.4 gezeigt.

Jedes Kochbuch hat eine eindeutige Version, die mit dem Kommando *version* in den Metadaten deklariert wird. Die Version wird in den Eigenschaften der Kochbuch-Konfigurationen gespeichert. Sie dient der eindeutigen Namensgebung der extrahierten Kochbuch-Konfigurationen (siehe Abbildung 5.4).

Wenn das analysierte Kochbuch eine kurze Beschreibung enthält, wird das mit dem Kommando *description* in den Metadaten deklariert, welches ebenfalls nur ein Argument haben kann (siehe Zeile 5 in Auflistung 5.2). Die Beschreibung muss nicht extrahiert werden, wird aber zur besseren Übersicht den Eigenschaften Kochbuch-Konfigurationen zugeordnet, wie in Abbildung 5.4 gezeigt. Sie gibt meistens einen kurzen Überblick, welches Szenario durch das Kochbuch konfiguriert wird.

Mit dem Kommando *supports* (siehe Zeile 11 und 16 in Auflistung 5.2) wird deklariert, welche Plattformen durch das Kochbuch unterstützt werden. Dabei wird zwischen zwei Fällen unterschieden. Im ersten Fall enthält das Kommando als Argument nur den Namen der Plattform, wie in Zeile 12 von Auflistung 5.2. Das bedeutet, dass alle Versionen der Plattformen unterstützt werden. Im zweiten Fall enthält das erste Argument ebenfalls den Namen der Plattform. In den weiteren Argumenten sind Versionsbeschränkungen (siehe Abschnitt 4.5.2) angegeben, wie in Zeile 16 von Auflistung 5.2. Da sich Konfigurationen zwischen den Plattformen unterscheiden können, wie zu Beginn in Kapitel 5 erläutert, kann eine Kochbuch-Konfiguration eines Kochbuches nur eine Plattform als Anforderung haben. Da Plattformen in den Metadaten nur über das *supports*-Kommando identifiziert werden und die

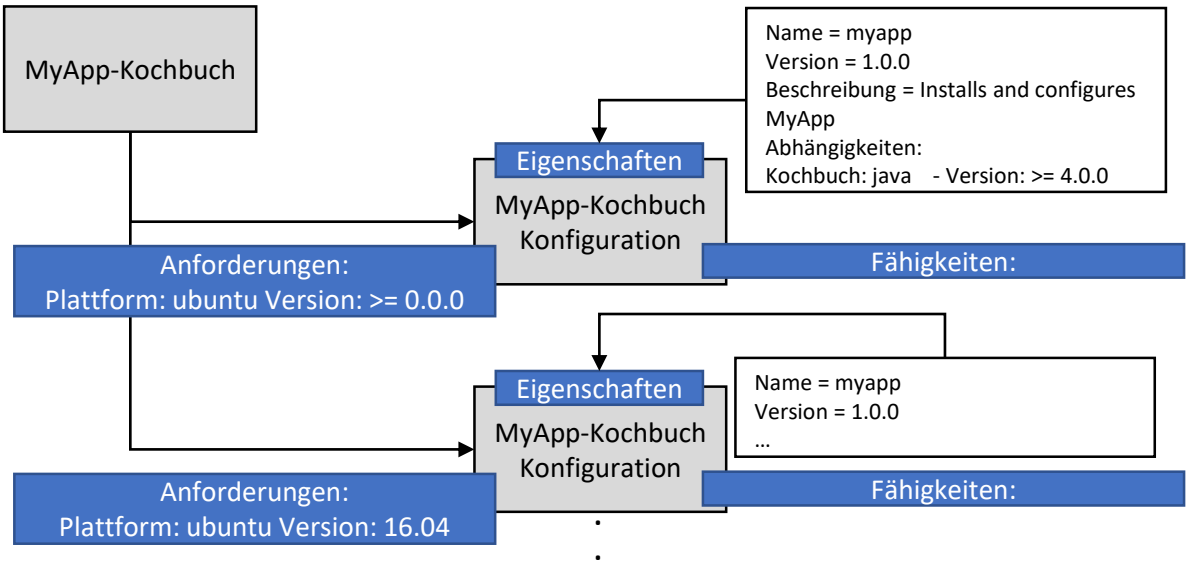

Abbildung 5.4: Transformation der Metadaten

Datei *metadata.rb* die erste analysierte Datei ist, enthält das Kochbuch-Objekt an dieser Stelle nur eine Kochbuch-Konfiguration, die keine Plattform als Anforderung hat. Die erste identifizierte Plattform wird folglich der ersten und bis an dieser Stelle einzigen Kochbuch-Konfiguration zugeordnet. Für jede weitere erkannte Plattform wird eine bestehende Kochbuch-Konfiguration dupliziert und die alte durch die neue identifizierte Plattform ersetzt. Die Anzahl der Kochbuch-Konfigurationen entspricht somit der Anzahl der extrahierten Plattformen, die das Kochbuch unterstützt (siehe Abbildung 5.4). Wird im Argument des *supports*-Kommandos eine Versionsbeschränkung angegeben, wird diese als Version der Plattform eingetragen. Enthält das Argument keine Versionsbeschränkung, wird der Wert „>= 0.0.0“ eingetragen. Dies bedeutet, dass alle Versionen der angegebenen Plattform unterstützt werden.

Abhängigkeiten zu weiteren Kochbüchern werden mit dem Kommando *depends* angegeben. Im ersten Argument steht der Name des Kochbuches, zu dem die Abhängigkeit besteht. In optionalen weiteren Argumenten wird eine Versionsbeschränkung angegeben, mit der erlaubte Versionen des angegebenen Kochbuches deklariert werden. In Zeile 19 von Auflistung 5.2 wird das Java-Kochbuch in einer Version größer oder gleich 4.0.0 als Abhängigkeit angegeben. Die Abhängigkeit bedeutet, dass Rezepte und Attribute des Java-Kochbuches im MyApp-Kochbuch verwendet werden. Folglich werden Kochbücher, die als Abhängigkeit angegeben werden, auf der Festplatte gespeichert, damit diese in die Transformation einbezogen werden können. Da der Crawler aus Abschnitt 4.2 nur das Crawlen aller Kochbücher unterstützt, wird dieser um die Fähigkeit erweitert, ein Kochbuch mit dem übergebenen Namen und angegebener Versionsbeschränkung herunterzuladen. Dazu wird die Methode in Algorithmus 5.1 verwendet, der Name des Kochbuches sowie die extrahierte Versionsbeschränkung übergeben werden. Die Chef Supermarket-API bietet einen Endpunkt, der den Zugriff auf ein bestimmtes Kochbuch ermöglicht. Die Anfrage dazu wird in Algorithmus 5.1 in Zeile 2 deklariert. Die Antwort wird analog zu den Anfragen in Abschnitt 4.2 gespeichert. Die Antwort auf die Anfrage hat die Form wie in Auflistung 4.2 gezeigt. Die Daten werden zusammen mit Namen und Versionsbeschränkung des Kochbuches an die Methode `processCookbook` übergeben (vgl. Algorithmus 4.2). Statt der aktuellsten Version des Kochbuches wird dabei die aktuellste, erlaubte Version des Kochbuches heruntergeladen,

die mit der angegebenen Versionsbeschränkung möglich ist. Dies wird in Tabelle 5.1 gezeigt. In den Daten, die der Methode `processCookbook` übergeben werden, finden sich alle Versionen des Kochbuches, die im Chef Supermarket verfügbar sind. Die URLs zu den Versionen finden sich in den übergebenen Daten (siehe Auflistung 4.2) im Feld *versions*, in Form eines Arrays mit einem Feld für jede URL. Die URL ist in der Form `https://supermarket.chef.io/api/v1/cookbooks/java/versions/4.0.0`. Die URLs zu den Kochbuchversionen sind in absteigender Version geordnet. Zur Auswahl der Version wird über alle URLs iteriert und in jeder Iteration die Version extrahiert, die sich nach dem letzten Schrägstrich befindet. In der genannten URL wird die Version `4.0.0` extrahiert. Die Version wird mit der Versionsabhängigkeit aus dem `depends` Kommando der Metadaten verglichen. Am Beispiel aus Auflistung 5.2 wird die Version `4.0.0` mit der Versionsabhängigkeit `>= 4.0.0` verglichen. Wenn diese wahr ist, wird das Kochbuch über die URL heruntergeladen. Da über die Versionen in absteigender Reihenfolge iteriert wird, ist sichergestellt, dass die höchstmögliche, erlaubte Version des abhängigen Kochbuches heruntergeladen wird. Als Versionsbeschränkung können auch mehrere Beschränkungen angegeben werden. Die Überprüfung der erlaubten Version funktioniert analog dazu, mit dem Zusatz, dass die Version mit allen Versionsbeschränkungen verglichen wird. Wird beispielsweise die Versionsbeschränkung `>= 1.0.0 , <= 4.0.0` angegeben, wird die Version mit beiden Beschränkungen verglichen. Wenn beide wahr sind, wird die Version des Kochbuches über die URL heruntergeladen. Die abhängigen Kochbücher werden in ein Verzeichnis innerhalb der Kochbuchstruktur gespeichert.

Algorithmus 5.1 Crawlingvorgang eines Kochbuches mit Name und Versionsbeschränkung

1: **function** GETCOOKBOOK($cookbookName, versionRestriction$)
2: `url` := "$https://supermarket.chef.io/api/v1/cookbooks/$" + $coobookName$
3: `cookbookJSONObject` := GETJSONFROMURL(url)
4: PROCESSCOOKBOOK(`cookbookJSONObject`, $cookbookName, versionRestriction$)
5: **end function**

Versionsbeschränkung	gecrawlte Version des Kochbuches
> 1.0.0	Aktuellste Version des Kochbuches.
<= 1.0.0	Version 1.0.0 des Kochbuches.
> 1.0.0 , <= 4.0.0	Version 4.0.0 des Kochbuches.

Tabelle 5.1: Aktuellste Version des Kochbuches abhängig von Versionsbeschränkung

5.3 kitchen.yml

Da die Angaben zu unterstützten Plattformen in den Metadaten eines Kochbuches meistens ungenau sind, wird die Datei *kitchen.yml* für eine detailliertere Extraktion der unterstützten Plattformen analysiert. Da die Datei optional ist, wird nach der Transformation der Metadaten überprüft, ob die Datei *kitchen.yml* auf der obersten Ebene des Kochbuchverzeichnisses vorhanden ist. Die Datei *kitchen.yml* lässt sich inhaltlich in vier Bestandteile gliedern (siehe Abschnitt 4.5.6). Für die Transformation ist der Teil notwendig, indem die Plattformen deklariert werden, auf dem das Kochbuch getestet wird. Der Teil beginnt mit dem Schlüsselwort *platforms* und beinhaltetet eine Liste von Plattformen, auf denen getestet wird (siehe Teil 3 in Auflistung 4.14).

Die Datei *kitchen.yml* ist nicht, wie die bisherigen Dateien, in Ruby geschrieben, sondern im YAML-Format[1] (YAML Ain't Markup Language (YAML)). Bei der Transformation der Datei *kitchen.yml* werden die Plattformen aus der Liste mit dem Schlüsselwort *platforms* extrahiert. Diese enthält Listeneinträge mit den Plattformen, auf denen das Kochbuch getestet wird. Folglich kann davon ausgegangen werden, dass das Kochbuch die Plattformen in den Tests unterstützt. Ein Element in der Liste der Plattformen hat immer die Form: `- name: ubuntu-16.04`. Dabei wird die Plattform und die Version immer durch einen Bindestrich getrennt angegeben. Die extrahierten Plattformen und Versionen werden mit den extrahierten Plattforminformationen aus den Metadaten zusammengeführt.

Algorithmus 5.2 zeigt die Extraktion der Plattforminformationen aus der Datei *kitchen.yml* und die Zusammenführung mit den bestehenden Kochbuch-Konfigurationen. Der Methode wird das bestehende Kochbuch-Objekt übergeben, das die Informationen aus den Metadaten enthält. Im Kochbuch-Objekt ist demnach mindestens eine Kochbuch-Konfiguration vorhanden, wie in Abbildung 5.5 gezeigt. Diese werden in einer Liste von Kochbuch-Konfigurationen gespeichert (siehe Zeile 2 von Algorithmus 5.2). In Zeile 3 werden die Plattformen aus der Datei *kitchen.yml* extrahiert und in einer Liste gespeichert, die Elemente in der Form `ubuntu-16.04` enthält. Die extrahierten Plattformen müssen den bestehenden Kochbuch-Konfigurationen zugeordnet werden, sodass keine Duplikate entstehen und die Informationen vervollständigt werden. Dies geschieht mit der Schleife ab Zeile 4, mit der jede extrahierte Plattform den bestehenden Kochbuch-Konfigurationen zugeordnet wird. Da die Plattformbezeichnungen von Metadaten und *kitchen.yml* nicht den gleichen Standard haben, werden die extrahierten Plattformen in Zeile 6 normalisiert. In den Metadaten schreibt sich die Plattform openSUSE Leap[2] beispielsweise als `opensuseleap`, wohingegen sie in der Datei *kitchen.yml* als `opensuse-leap` deklariert wird. Gleiches gilt für die Plattform MacOS, welche sich in den Metadaten als `mac_os_x` schreibt und in der Datei *kichen.yml* als `macos`. Die Plattformen werden auf die Form der Metadaten normalisiert.

Für jede extrahierte Plattform muss überprüft werden, ob die Plattform schon einer bestehenden Kochbuch-Konfiguration zugeordnet wird. Dies geschieht durch die Schleife ab Zeile 7, die über alle bestehenden Kochbuch-Konfigurationen iteriert und die neuen Informationen integriert. In jeder Iteration wird der bestehende Name und die Version der Plattform extrahiert und überprüft, ob die neue Plattform aus der Datei *kitchen.yml* mit dem existierenden Plattformnamen beginnt. Die Variable `index` nimmt den Wert „-1“ an, wenn die neue Plattform nicht mit dem existierenden Namen beginnt, und den Wert „0“, wenn dies der Fall ist. Zwischen den Werten wird mit einer Switch-Case-Anweisung unterschieden (Zeile 12-23). Nimmt die Variable `index` den Wert „0“ an, wurde die neue Plattform in den bestehenden Kochbuch-Konfigurationen gefunden. Anschließend wird die Version überprüft, wobei zwischen zwei Fällen unterschieden wird. Entweder wurde in den Metadaten die Plattform ohne Versionsbeschränkung angegeben, dann ist die Version „>= 0.0.0“enthalten, oder es wird eine Version angegeben. Ist die Version der bestehenden Plattform „>= 0.0.0“, wird die Version der bestehenden Plattform mit der Version aus der extrahierten Plattform aktualisiert (siehe Zeile 14). Andernfalls existiert in der bestehenden Plattform eine Version. Folglich wird die neu extrahierte Plattform in einer neuen Kochbuch-Konfiguration gespeichert (siehe Zeile 16). Dazu wird eine bestehende Kochbuch-Konfiguration kopiert und Name sowie Version der Plattform mit der neuen ersetzt. Die neue Kochbuch-Konfiguration wird den bestehenden zugeordnet. Schließlich wir in Zeile 18 eine boolesche Variable auf wahr gesetzt, mit der sichergestellt wird, dass die Plattform nicht noch einmal hinzugefügt wird. Nimmt die Variable `index` den Wert „-1“ an,

[1] https://yaml.org/
[2] https://software.opensuse.org/distributions/leap?locale=de

Algorithmus 5.2 Extraktion der Plattforminformationen aus Datei kitchen.yml

```
1:  function addPlatformVersionInformationFromKitchen(cookbookObject)
2:      cookbookConfigurationList := cookbookObject.getCookbookConfigurationsAsList()
3:      platforms := ymlParser.getPlatformsFromKitchen()
4:      for all platform ∈ platforms do
5:          platformFound := false
6:          platform.correctName()
7:          for all cookbookConfig ∈ cookbookConfigurationList do
8:              existingPlatformName := cookbookConfig.getPlatformName()
9:              existingPlatformVersion := cookbookConfig.getPlatformVersion()
10:             index := platform.indexOf(existingPlatformName)
11:             switch index do
12:                 case 0
13:                     if existingPlatformVersion ==">= 0.0.0" then
14:                         cookbookConfig.updatePlatformVersion(platform)
15:                     else
16:                         cookbookConfigurationList.addConfigWithNewVersion(platform)
17:                     end if
18:                     platformFound := true
19:                 case −1
20:                     if !platformFound AND cookbookConfig.isLastConfig() then
21:                         cookbookConfigurationList.addConfigWithNewVersion(platform)
22:                     end if
23:             end switch
24:             if platformFound then
25:                 break
26:             end if
27:         end for
28:     end for
29:     return cookbookObject.replaceCookbookConfigs(cookbookConfigurationList)
30: end function
```

ist die neue Plattform eine andere wie die der überprüften Kochbuch-Konfiguration. Dies wird solange überprüft, bis die Plattform gefunden ist oder die letzte bestehende Kochbuch-Konfiguration erreicht wurde. In Algorithmus 5.2 wird folglich in Zeile 19-22 überprüft, ob die neue Plattform noch keiner Kochbuch-Konfiguration zugeordnet wurde und ob die Kochbuch-Konfiguration die letzte in der Liste ist. Wenn dies der Fall ist, wird eine neue Kochbuch-Konfiguration mit der neuen Plattform hinzugefügt. Die Methode gibt das Kochbuch-Objekt mit den hinzugefügten Informationen zurück.

Die Funktionsweise von Algorithmus 5.2 wird anhand des Beispiels in Abbildung 5.5 verdeutlicht. Vor der Analyse der Datei *kitchen.yml* wurden die Metadaten des MyApp-Kochbuches kompiliert. Daraus resultiert eine Kochbuch-Konfiguration mit der Plattform Ubuntu in Version „>= 0.0.0“. Dies besagt, dass alle Ubuntu Versionen unterstützt werden. Die Datei *kitchen.yml* enthält die Plattformen `- name: ubuntu-16.04` und `- name: debian-9`.

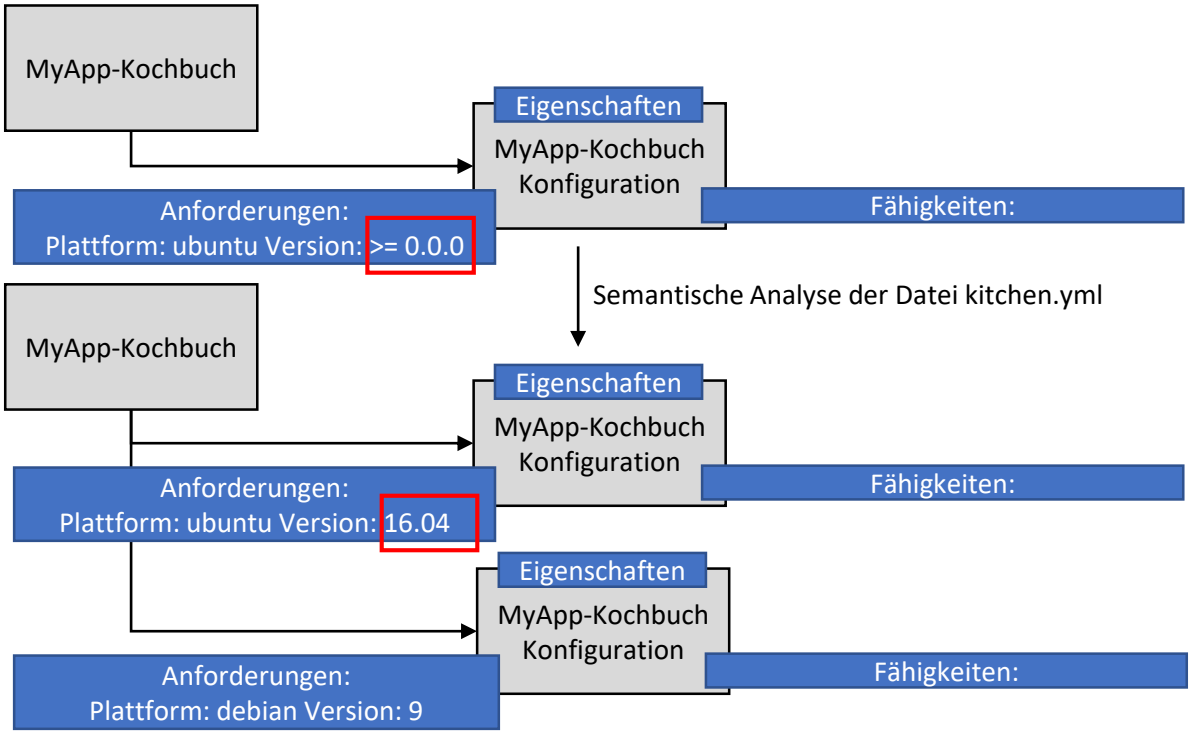

Abbildung 5.5: Mapping der extrahierten Plattformen aus kitchen.yml

Der vorgestellte Algorithmus 5.2 extrahiert als erste Plattform `ubuntu-16.04`. Da die bestehende Kochbuch-Konfiguration Ubuntu unterstützt, aber keine Version spezifiziert ist, wird die Version aktualisiert, wie in Abbildung 5.5 gezeigt. Besteht in den Anforderungen der Kochbuch-Konfiguration bereits eine Plattform mit einer Version, wie beispielsweise die Plattform Ubuntu in Version 14.04, wird eine neue Kochbuch-Konfiguration hinzugefügt. Als nächste Plattform wird `debian-9` extrahiert. Diese ist in keiner bestehenden Kochbuch-Konfiguration. Folglich wird eine bestehende Kochbuch-Konfiguration kopiert und die bestehende Plattform durch die neue ersetzt.

5.4 Automatische Chef Attribute

Um abhängig von den Eigenschaften des Knotens selbst Konfigurationen spezifisch für den Knoten zu deklarieren, werden in Kochbüchern automatische Chef Attribute verwendet. Zur Spezialisierung von Konfigurationsaufgaben werden automatische Chef Attribute zum Beispiel in Case-Anweisungen verwendet (siehe Auflistung 4.8). Die Funktionsweise sowie der Aufbau von automatischen Attributen wurde in Abschnitt 4.5.3 anhand von Tabelle 4.3 erläutert. Automatische Attribute werden von Ohai ermittelt, wenn ein Knoten mit dem Kochbuch konfiguriert wird. Da das Kochbuch im vorgestellten Ansatz statisch analysiert wird, steht Ohai nicht zur Verfügung und es kann auf diese Attribute nicht zugegriffen werden. Im Folgenden werden Ansätze für die Lösung dieses Problems beschrieben und in Tabelle 5.2 mit ihren Vorteilen und Nachteilen zusammengefasst.

Eine Möglichkeit ist, die im Kochbuch aufgerufenen automatischen Chef Attribute zu ignorieren. Da die Attribute nicht generiert werden müssen, hat diese Möglichkeit den Vorteil des geringeren Wartungsaufwands. Durch das Ignorieren der automatischen Attribute lassen sich vom Knoten abhängige Bedingungen nicht auflösen. Folglich können Fallunterscheidungen auf Basis der automatischen Chef Attribute bei der semantischen Analyse nicht aufgelöst werden. Da Kochbücher oft für mehrere Plattformen konzipiert sind, enthalten sie plattformabhängige Bedingungen, mit denen solche Fallunterscheidungen deklariert werden. Da Bedingungen nicht aufgelöst werden, folgt daraus, dass während der semantischen Analyse entscheidende Informationen möglicherweise nicht extrahiert werden können.

Alternativ zu dem Ignorieren der Attribute besteht die zweite Möglichkeit im Generieren der automatischen Chef Attribute. Dabei werden diese aus den Informationen abgeleitet, die aus dem Kochbuch bei der semantischen Analyse extrahiert werden. Da sich beispielsweise die Namen von Softwarekomponenten plattformübergreifend unterscheiden können, werden automatische Attribute zur Unterscheidung solcher Bedingungen verwendet. Der Vorteil dieser Alternative ist die daraus resultierende Verfeinerung der semantischen Analyse, durch Unterscheidung von solchen Bedingungen. Da die Attribute aus bestehendem Wissen generiert werden müssen, hat diese Möglichkeit einen einhergehenden höheren Wartungsaufwand als Nachteil. Da normalerweise Ohai für die Erfassung der automatischen Attribute zuständig ist, müssen Teile der Funktionalität von Ohai simuliert werden. Da eine veränderte Zuordnung dann auch in der Implementierung des vorgestellten Chef Kochbuch Compilers angepasst werden muss, entsteht Wartungsaufwand durch Änderungen der Funktionalität bei Ohai, wenn zum Beispiel eine Plattform einer anderen Familie zugeordnet wird.

Alternative 1 hat den Nachteil, dass knotenabhängige Bedingungen nicht aufgelöst werden können, wodurch Informationen nicht extrahiert werden. Alternative 2 hat den Vorteil, dass knotenabhängige Bedingungen teilweise aufgelöst werden können. Da sich beide Alternativen aus Tabelle 5.2 im Wartungsaufwand unterscheiden, muss abgewogen werden, ob die Vorteile von Alternative 2 den höheren Wartungsaufwand rechtfertigen. Wartungsaufwand entsteht, wenn sich die Zuordnung der automatischen Attribute in Ohai ändert aber verfeinert die semantische Analyse. Da sich erhöhte Wartungsaufwand mit den Vorteilen von Alternative 2 vereinbaren lässt, wird ein erhöhter Wartungsaufwand akzeptiert. Alternative 2 wird nur für automatische Attribute verwendet, die sich bei der statischen Analyse eines Kochbuches generieren lassen. Für andere automatische Attribute wird Alternative 1 verwendet. Die generierten automatischen Chef Attribute werden im Folgenden erläutert.

Automatische Chef Attribute werden behandelt wie Chef Attribute (siehe Abschnitt 5.5). Wie in Abbildung 5.6 gezeigt, werden sie den Eigenschaften einer Kochbuch-Konfiguration zugeordnet. Die Werte der automatischen Chef Attribute lassen sich nicht aus Dateien auslesen und werden, wenn möglich, aus vorhandenen Informationen generiert. Nachfolgend werden automatische Chef Attribute erläutert, die mit dem im Kochbuch vorhandenen Wissen generiert werden können.

Dem automatischen Chef Attribut `['platform']` wird als Wert der Name der Plattform zugeordnet. Der Plattformname wird aus der Datei *metadata.rb* und der Datei *kitchen.yml* extrahiert (siehe Abschnitt 5.2 und Abschnitt 5.3). Bei der Extraktion einer Plattform wird dieser jeweils einer neuen Kochbuch-Konfiguration zugeordnet. Dabei wird der Kochbuch-Konfiguration das Chef Attribut `['platform']` mit dem Namen der Plattform als Wert, in kleinen Schriftzeichen, zugeordnet, wie Ohai dies bei der normalen Verwendung des Kochbuches tun würde. In Abbildung 5.6 wird der Kochbuch-Konfiguration mit Plattform Ubuntu beispielsweise das Chef Attribut `['platform'] = ubuntu` zugeordnet und der Kochbuch-Konfiguration mit der Windows Plattform das Chef Attribut `['platform'] = windows`.

Dem automatischen Chef Attribut `['platform_family']` wird als Wert die Plattformfamilie zugeordnet. Diese wird aus dem Namen der Plattform generiert. Die Zuordnung von Plattform zu Plattformfamilie findet sich in der aktuellen Implementierung von Ohai in den Plugins[3]. Die Plattformfamilie wird folglich auf Basis des Namens der Plattform generiert. Der Plattformname wird aus der Datei *metadata.rb* und der Datei *kitchen.yml* extrahiert (siehe Abschnitt 5.2 und Abschnitt 5.3). Bei der Extraktion der Plattform wird diese einer neuen Kochbuch-Konfiguration als Anforderung zugeordnet. Dabei wird der Kochbuch-Konfiguration das Chef Attribut `['platform_family']` mit der entsprechenden Plattformfamilie als Wert zugeordnet. In Abbildung 5.6 wird dem Chef Attribut `['platform_family']` der Kochbuch-Konfiguration mit Ubuntu Plattform beispielsweise der Wert *debian* zugeordnet und der Konfiguration mit Plattform Windows der Wert *windows*.

Dem automatischen Chef Attribut `['platform_version']` wird die Plattformversion als Wert zugewiesen. Wird eine Plattformversion aus den Metadaten (siehe Abschnitt 5.2) oder der Datei *kitchen.yml* (siehe Abschnitt 5.3) extrahiert und einer Kochbuch-Konfiguration zugeordnet, wird der Kochbuch-Konfiguration das Chef Attribut `['platform_version']` mit der entsprechenden Version als Wert zugeordnet.

#	Alternative	Vorteile	Nachteile
1	Automatische Attribute ignorieren	Kein Wartungsaufwand.	Knotenabhängige Bedingungen lassen sich nicht auflösen.
②	Automatische Attribute generieren	Knotenabhängige Bedingungen lassen sich größtenteils auflösen.	Da sich Werte ändern können, folgt ein höherer Wartungsaufwand.

Tabelle 5.2: Alternativen für Automatische Attribute

5.5 Attributdateien

Nachdem die Dateien *metadata.rb* und optional *kitchen.yml* kompiliert wurden, werden die Attributdateien kompiliert, sofern diese existieren. Attributdateien finden sich im *attributes*-Verzeichnis (siehe Abbildung 4.6). Der Aufbau der beinhalteten Chef Attribute wurde in Abschnitt 4.5.3 erläutert.

Attributdateien werden nacheinander eingelesen und semantisch analysiert. Die Reihenfolge ist analog zu der Reihenfolge wie sie Chef während dem Konfigurationsvorgang kompiliert. Zuerst wird die Datei *default.rb* aus dem *attributes*-Verzeichnis geladen und kompiliert. Danach werden die weiteren Attributdateien von A-Z geordnet kompiliert.

Wenn das analysierte Kochbuch Abhängigkeiten hat, werden die abhängigen Kochbücher heruntergeladen. Die dort abgelegten Kochbücher liegen ebenfalls in der bisher bekannten Struktur von Chef Kochbüchern vor. Folglich können diese ebenfalls ein *attributes*-Verzeichnis mit Attributdateien beinhalten. Da Rezepte von abhängigen Kochbüchern zusammen mit ihren Chef Attributen aufgerufen werden können, werden die Attributdateien der abhängigen Kochbücher ebenfalls geladen. Da in der Chef Dokumentation keine Aussage über die Reiehnfolge getroffen wird, werden die Attributdateien der

[3] https://github.com/chef/ohai/tree/master/lib/ohai/plugins

Auflistung 5.3 Transformation von Attributdateien

```
1  # Verschiedene Attributtypen
2  default['myapp']['attribut1'] = 'Wert Attribut 1'
3
4  force_default['myapp']['attribut2'] = 'Wert Attribut 2'
5
6  normal['myapp']['attribut3'] = 'Wert Attribut 3'
7
8  override['myapp']['attribut4'] = 'Wert Attribut 4'
9
10 force_override['myapp']['attribut5'] = 'Wert Attribut 5'
11
12 # Dieses Attribut wird nur bei Ubuntu Plattformen gesetzt
13 if (node['platform'] == 'ubuntu')
14   default['myapp']['attribut6'] = 'Wert Attribut 6'
15 end
16
17 # Attribut wird abhaengig von der Plattformfamilie gesetzt
18 case node['platform_family']
19 when 'windows'
20   default['myapp']['install_flavor'] = 'windows'
21 when 'mac_os_x'
22   default['myapp']['install_flavor'] = 'homebrew'
23 else
24   default['myapp']['install_flavor'] = 'openjdk'
25 end
```

Kochbücher in der Reihenfolge, wie auch die abhängigen Kochbücher in den Metadaten deklariert wurden, geladen. Die Reihenfolge der geladenen Attributdateien innerhalb eines abhängigen Kochbuches ist analog zu der genannten Reihenfolge.

Chef Attribute kommen in unterschiedlichen Typen vor (siehe Abschnitt 4.5.3). Die Typen sind (i) *default*, (ii) *force_default*, (iii) *normal*, (iv) *override* und (v) *force_override*. Da automatische Chef Attribute von Ohai erfasst werden und nicht überschrieben werden dürfen, kann der Attributtyp *automatic* nicht vorkommen. Wie mit automatischen Attributen verfahren wird, wurde in Abschnitt 5.5 erläutert. Die verschiedenen Attributtypen haben unterschiedliche Bedeutungen (siehe Abschnitt 4.5.3). Auflistung 5.3 zeigt einen beispielhaften Ausschnitt einer Attributdatei, an dem die Transformation der Attributdateien erläutert wird. Um Chef Attribute bei der Verwendung in Rezepten aufzulösen, müssen bei der semantischen Analyse die Attributtypen *default* (Zeile 2), (ii) *force_default* (Zeile 4), (iii) *normal* (Zeile 6), (iv) *override* (Zeile 8) und (v) *force_override* (Zeile 10) identifiziert und extrahiert werden. Alle Attributtypen bestehen aus einem Typ gefolgt von dem Namen des Attributs. Das Chef Attribut in Zeile 2 ist von dem Typ *default* und besitzt den Namen `['myapp']['attribut1']`. Diesem Chef Attribut wird der String „Wert Attribut 1“ als Wert zugewiesen.

Chef Attribute werden, wie in Abbildung 5.1 gezeigt, den Eigenschaften einer Kochbuch-Konfiguration zugeordnet. Am laufenden Beispiel mit dem MyApp-Kochbuch wird das Attribut `['myapp']['attribut1']` den Eigenschaften jeder Kochbuch-Konfiguration zugeordnet (siehe Abbildung 5.6). Das Beispiel aus Zeile 17-25 von Auflistung 5.3 zeigt das Attribut `['myapp']['install_flavor']`, welches abhängig von der Plattformfamilie einen anderen Wert zugewiesen bekommt. Chef Attribute müssen deshalb jeder Kochbuch-

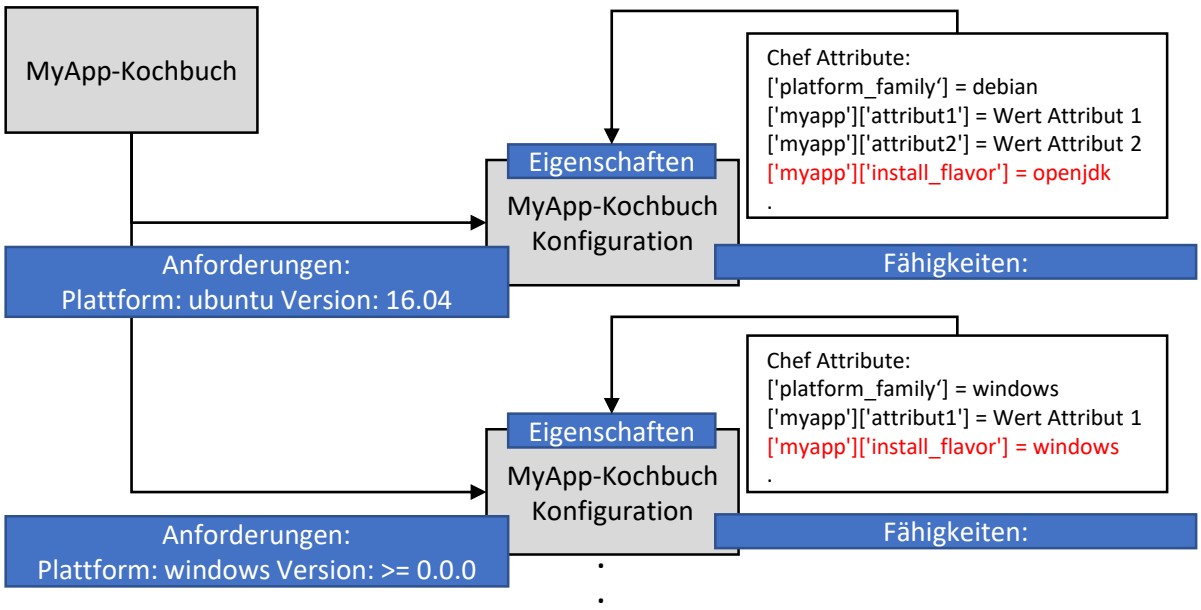

Abbildung 5.6: Chef Attribute werden den Eigenschaften der Kochbuch-Konfigurationen zugeordnet

Konfiguration einzeln zugeordnet werden. Jede Kochbuch-Konfiguration hat eine Plattform als Anforderung. Aus dieser wurde das automatische Attribut `['platform_family']` generiert, auf das in der gezeigten Case-Anweisung in Zeile 18 zugegriffen wird. Der Wert des Attributs `['myapp']['install_flavor']` nimmt in der gezeigten Case-Anweisung, abhängig von der Plattformfamilie der Kochbuch-Konfiguration, einen anderen Wert an. Abbildung 5.6 hebt die Zuordnung von diesem Chef Attribut in Rot hervor.

Für die semantische Analyse lässt sich daraus ableiten, dass der abstrakte Syntaxbaum einer Attributdatei für jede Kochbuch-Konfiguration durchlaufen und semantisch analysiert werden muss. Dies wird in Algorithmus 5.3 zur Transformation einer Attributdatei beispielhaft gezeigt. In Zeile 2 werden die Kochbuch-Konfigurationen als Liste aus dem Kochbuch-Objekt gespeichert. In Zeile 3 wird die Attributdatei in einen abstrakten Syntaxbaum übersetzt. Mit der For-Schleife in Zeile 4-7 wird die Attributdatei für jede Kochbuch-Konfiguration einzeln analysiert. Dazu wird in Zeile 5 der Syntaxbaum durchlaufen und die gefundenen Attribute werden der Kochbuch-Konfiguration hinzugefügt. Die Kochbuch-Konfiguration mit den neu gefundenen Informationen wird nun in Zeile 6 in einer neuen Liste aus Kochbuch-Konfigurationen gespeichert. Die Methode gibt das Kochbuch-Objekt mit den neu extrahierten Informationen aus der Attributdatei zurück. Dies wird für jede Attributdatei, die geparst werden muss, nach der am Anfang des Abschnitts erläuterten Reihenfolge, wiederholt.

Attributdateien in Chef Kochbüchern können alle Anweisungen enthalten, die in der Programmiersprache Ruby möglich sind. In Attributdateien kommen beispielsweise If-Bedingungen oder Case-Anweisungen vor, wie das Beispiel aus Auflistung 5.3 in Zeile 12-25 zeigt. Um die Attribute in den Anweisungen korrekt aufzulösen, müssen solche Anweisungen korrekt interpretiert werden. Die Methodik, solche Anweisungen zu interpretieren, wurde in der Literatur zum Bau von Compilern, wie zum Beispiel Aho [Aho03], ausführlich diskutiert. Das Vorgehen, solche Anweisungen semantisch zu interpretieren, wird deshalb in dieser Arbeit nicht weiter thematisiert, sondern vorausgesetzt. Dies gilt

auch für die Transformation von weiteren Dateien in Kochbüchern, die in Ruby geschrieben werden, wie Rezepte und benutzerdefinierte Ressourcen.

Algorithmus 5.3 Kompilieren einer Attributdatei

```
1: function compileAttributeFile(attributeFile, cookbookObject)
2:     cookbookConfigurationList := cookbookObject.getCookbookConfigurationsAsList()
3:     parseTree := parseRubyFile(attributeFile)
4:     for all cookbookConfiguration ∈ cookbookConfigurationList do
5:         cookbookConfiguration = analyzeAttributeFile(parseTree, cookbookConfig)
6:         newCookbookConfigurationList.add(cookbookConfig)
7:     end for
8:     return cookbookObject.replaceCookbookConfigs(newCookbookConfigurationList)
9: end function
```

5.6 Rezepte

Wie im Ablauf der Transformation in Abbildung 5.3 gezeigt, werden im letzten Schritt die Rezepte des Kochbuches kompiliert. Rezepte befinden sich im *recipes*-Verzeichnis eines Kochbuches (siehe Abbildung 4.6). Bei der Verwendung von Chef Kochüchern wird die Reihenfolge der auszuführenden Rezepte zur Konfiguration eines Knotens in einer Liste auf dem Chef-Server festgelegt [Inc19a]. Diese wird manuell konfiguriert und befindet sich nicht innerhalb der Verzeichnisstruktur eines Kochbuches. Wird keine Liste festgelegt, ist die Datei *default.rb* das Standardrezept, welches den Startpunkt der Konfiguration markiert. Da es innerhalb eines Kochbuches keinen weiteren Anhaltspunkt für das Startrezept gibt, wird die Datei *default.rb* als erstes kompiliert. Dieses ist in Kochbüchern nicht erforderlich, weshalb es Kochbücher ohne die Datei *default.rb* im *recipes*-Verzeichnis gibt. Existiert diese nicht, kann kein Startpunkt der Analyse festgelegt werden. In solchen Fällen stehen meistens Informationen über das Startrezept in der Readme-Datei, welche meistens genaue Informationen zur Konfiguration und Anwendung des Kochbuches enthält. Da sie aus unstrukturiertem Text besteht, extrahiert dieser Ansatz keine Informationen aus der Readme-Datei. Folglich werden Rezepte nur analysiert, wenn im Kochbuch das Standardrezept enthalten ist. Andernfalls wird die Analyse abgebrochen.

In einem Rezept lassen sich weitere Rezepte aufrufen. Der Aufruf geschieht dabei mit dem Kommando `include_recipe 'cookbook::recipe'`. Im Argument der Methode wird das aufzurufende Rezept angegeben. Die Funktionsweise wurde in Abschnitt 4.5.4 anhand von Auflistung 4.9 erläutert. Findet in einem Rezept der Aufruf eines weiteren Rezeptes statt, wird der Inhalt des aufgerufenen Rezeptes exakt an der deklarierten Stelle des Aufrufs ausgeführt. Rezepte lassen sich auch kochbuchübergreifend aufrufen. Die Funktionsweise wird anhand von Tabelle 5.3 verdeutlicht. Diese zeigt Rezeptaufrufe am Beispiel des MyApp-Kochbuches aus dem laufenden Beispiel. Dabei besteht weiterhin die Annahme, dass eine, in den Metadaten des Kochbuches deklarierte, Abhängigkeit zu dem Java-Kochbuch besteht. Dementsprechend liegt das Kochbuch an dieser Stelle der Transformation in der vorgestellten Verzeichnisstruktur vor (siehe Abbildung 4.6). Durch die Abhängigkeit zu dem Java-Kochbuch befindet sich in der Verzeichnisstruktur ein zusätzliches Verzeichnis (hier *dependencies*), in dem sich das Java-Kochbuch unter dem Verzeichnis *java* befindet. Das Herunterladen von abhängigen Kochbüchern wurde in Abschnitt 5.2 erläutert. Durch die Deklaration des Java-Kochbuches als Abhängigkeit des MyApp-Kochbuches, kann in Rezepten des

#	Rezeptaufruf	Pfad zu dem aufgerufenen Rezept am Beispiel des MyApp-Kochbuches
1	`include_recipe 'myapp'`	myapp/recipes/default.rb
2	`include_recipe 'myapp::default'`	myapp/recipes/default.rb
3	`include_recipe 'java'`	myapp/dependencies/java/recipes/default.rb
4	`include_recipe 'java::default'`	myapp/dependencies/java/recipes/default.rb
5	`include_recipe 'example::default'`	Da keine Abhängigkeit zum example-Kochbuch besteht, wird es nicht aufgerufen.

Tabelle 5.3: Rezeptaufrufe am Beispiel des MyApp-Kochbuches

MyApp-Kochbuches der Aufruf von Rezepten des Java-Kochbuches vorkommen, wie die gezeigten Aufrufe 3 und 4 in Tabelle 5.3. Da das Java-Kochbuch als Abhängigkeit des MyApp-Kochbuches deklariert ist, werden die aufgerufenen Java-Rezepte analysiert und die Informationen extrahiert. Das Java-Kochbuch kann wiederum indirekte Abhängigkeiten zu weiteren Kochbüchern, wie dem fiktiven Example-Kochbuch haben. In den aufgerufenen Java-Rezepten können folglich Rezepte des Example-Kochbuches aufgerufen werden, wie in Rezeptaufruf 5 in Tabelle 5.3 gezeigt. Bei der Analyse des MyApp-Kochbuches werden Aufrufe zu indirekten Abhängigkeiten ignoriert. Der Rezeptaufruf wird folglich übergangen und das Rezept nicht analysiert. Da abhängige Kochbücher rekursiv transformiert werden, wie zu Beginn in Kapitel 5 erläutert, werden die Abhängigkeiten des Java-Kochbuches in einer separaten Analyse berücksichtigt (siehe Abbildung 5.2). Zusammengefasst werden bei der Transformation des MyApp-Kochbuches eigene Rezepte und Rezepte zu Abhängigkeiten, wie dem Java-Kochbuch, analysiert. Anschließend werden die Abhängigkeiten des MyApp-Kochbuches (Java-Kochbuch) transformiert. Dieses Vorgehen wird wiederholt, bis alle Abhängigkeiten aufgelöst sind.

In Rezepten können Chef Attribute deklariert werden (siehe Abschnitt 4.5.4). Dies geschieht analog zu der Deklaration von Chef Attributen in Attributdateien, mit dem Unterschied, dass das Knotenobjekt explizit angegeben wird (`node.default['name']['attributname']`). In Rezepten deklarierte Chef Attribute werden analog zu Attributen in Attributdateien extrahiert (siehe Abschnitt 5.5). Folglich werden in Rezepten deklarierte Chef Attribute den Eigenschaften den extrahierten Kochbuch-Konfigurationen zugeordnet. Um die Konfiguration eines Knotens zu individualisieren, werden Chef Attribute in Rezepten verwendet. Dies geschieht, wie in Zeile 11 von Auflistung 5.4 gezeigt, durch Aufrufen des Knotenobjekts, gefolgt von dem Namen des Chef Attributs (`node['myapp']['package_name']`). Um das Attribut aufzulösen, wird es durch den Wert des aufgerufenen Chef Attributs ersetzt. Chef Attribute werden, wie in Abschnitt 5.5 erläutert, in den Eigenschaften der extrahierten Kochbuch-Konfigurationen gespeichert. Folglich lassen sich zu jeder Zeit der Analyse die Werte der Chef Attribute aus den Kochbuch-Konfigurationen auslesen, die alle Chef Attribute bis an der aufgerufenen Stelle im Code in den Eigenschaften gespeichert haben.

Chef Attributen können abhängig von der Plattform unterschiedliche Werte zugewiesen werden (siehe Abschnitt 5.5). Folglich kann sich die Konfiguration von verschiedenen Plattformen unterscheiden, was innerhalb der Rezepte mit Chef Attributen realisiert wird. Rezepte werden deshalb ausgehend vom Startrezept (default.rb), analog zu Attributdateien, für jede Kochbuch-Konfiguration einzeln semantisch analysiert, was durch Algorithmus 5.4 schemenhaft verdeutlicht wird. Dieser übersetzt das Startrezept des Kochbuches zunächst in einen abstrakten Syntaxbaum, welcher für jede Kochbuch-Konfiguration durchlaufen wird. Die dabei gefundenen Informationen werden in die bestehenden integriert. Der be-

schriebene Ablauf stellt sicher, dass Fallunterscheidungen auf Basis plattformabhängiger Chef Attribute korrekt aufgelöst und extrahiert werden.

Algorithmus 5.4 Kompilieren von Rezepten ausgehend vom Startrezept

```
1: function compileRecipe(startrecipe, cookbookObject)
2:     cookbookConfigurationList := cookbookObject.getCookbookConfigurationsAsList()
3:     parseTree := parseRubyFile(startrecipe)
4:     for all cookbookConfiguration ∈ cookbookConfigurationList do
5:         cookbookConfig = analyzeRecipe(parseTree, cookbookConfig)
6:         newCookbookConfigurationList.add(cookbookConfig)
7:     end for
8:     return cookbookObject.replaceCookbookConfigs(newCookbookConfigurationList)
9: end function
```

Da Rezepte in Ruby geschrieben werden, können zusätzlich zu Chef Attributen Ruby Variablen deklariert werden (siehe Auflistung 4.10 Zeile 8). Diese werden, vergleichbar mit Chef Attributen, den Eigenschaften einer Kochbuch-Konfiguration zugeordnet, um Aufrufe von Ruby-Variablen an anderer Stelle aufzulösen. Damit wird sichergestellt, dass der Zugriff auf diese zu jedem Zeitpunkt der Transformation möglich ist. Wird eine Variable an einer anderen Stelle im Rezept verwendet, wird auf die bestehenden Ruby Variablen zugegriffen und der Variablenaufruf durch den Wert der Variable ersetzt.

Der Inhalt von Rezepten ist eine Sammlung von Ressourcen, mit denen Konfigurationsaufgaben durchgeführt werden (siehe Abschnitt 4.5.4). Für diese Arbeit sind vor allem Ressourcen relevant, mit denen Softwarepakete installiert, deinstalliert und gemanagt werden. In Abschnitt 4.5.5 wurde die *package*-Ressource sowie Abwandlungen davon identifiziert, mit der diese Art von Konfigurationsaufgaben in Rezepten deklariert werden. Auflistung 5.4 zeigt in Zeile 4-8 anhand dem laufenden Beispiel die beispielhafte Deklaration einer *package*-Ressource. Die Ressource mit dem Namen „package1“ installiert das Softwarepaket „myapp“ in der Version „1.1“. Der Fakt, dass das Softwarepaket installiert wird, lässt sich aus der Aktion (action) „:install“ folgern.

Die Extraktion dieser Informationen aus Rezepten, wie in Auflistung 5.4, wird anhand von Abbildung 5.7 erläutert. Extrahierte Pakete aus Chef Rezepten werden in einer Liste gespeichert, welche einen Eintrag mit dem Namen der Chef Ressource als Schlüssel enthält. Dieses Grundprinzip wird in Abbildung 5.7 mit dem ersten Listeneintrag demonstriert. Dieser zeigt die extrahierten Informationen aus Zeile 16-20 von Auflistung 5.4. Mit dem Namen der Ressource als Schlüssel zeigt der Eintrag auf die extrahierten Informationen. Dabei werden die Eigenschaften `package_name`, `version` und `action` mit den zugehörigen Werten extrahiert. Der zweite Eintrag der Liste in Abbildung 5.7 zeigt das extrahierte Paket aus Zeile 4-8 von Auflistung 5.4. Der dritte Eintrag der Liste entspricht dem extrahierten Paket aus Zeile 10-13. Bei der Ressource werden Chef Attribute als Name und Version verwendet. Das Attribut `['myapp']['package_name']` wird durch den Wert „myapp-addon“ und das Attribut `['myapp']['package_version']` durch den Wert „1.2“ ersetzt. Die Ressource enthält im Vergleich zu den bisherigen Ressourcen keine Eigenschaft `package_name`. Entsprechend der Chef Dokumentation wird der Name der Ressource als Standardwert verwendet, wenn die Eigenschaft `package_name` nicht deklariert ist. Da keine Eigenschaft `package_name` angegeben ist, nimmt diese den Wert „myapp-addon“ an. Die Eigenschaft `action` ist ebenfalls nicht angegeben. Folglich wird für die Aktion, entsprechend der Chef Dokumentation, der Standardwert *:install* angenommen.

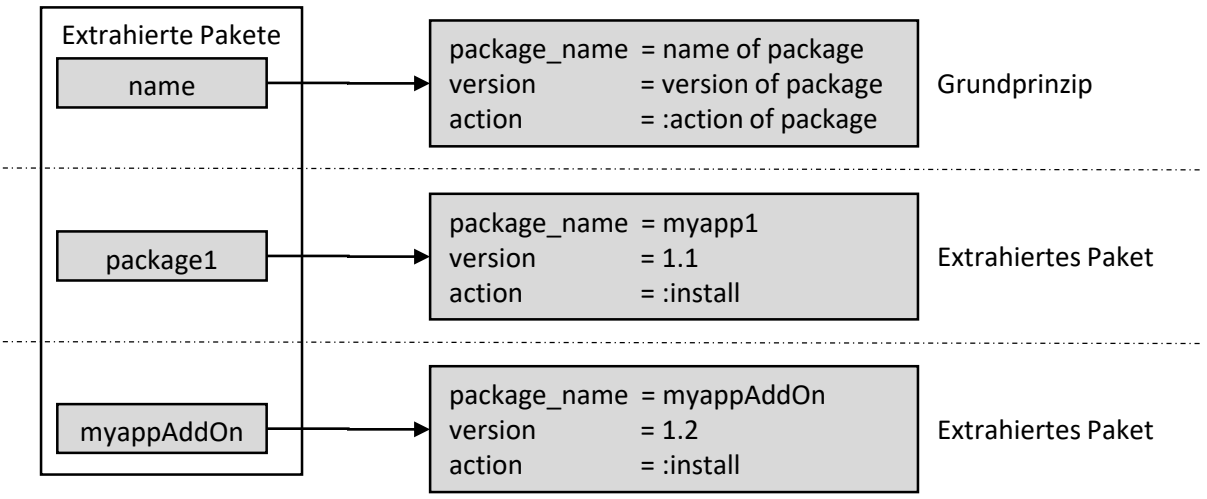

Abbildung 5.7: Extraktion von Softwarekomponenten

Die Chef Ressource *package* hat weitere mögliche Aktionen (siehe Abschnitt 4.5.5). Abgesehen von der Aktion *:install* werden Pakete auch durch die Aktion *:upgrade* installiert. Durch die Aktion *:nothing* wird das Paket vorgemerkt und durch die Aktionen *:purge* und *:remove* werden Pakete vom Knoten entfernt. Folglich werden die Pakete bei diesen Aktionen aus der Liste gelöscht. Die Aktion *:reconfig* konfiguriert ein Paket unter Verwendung einer speziellen Datei. Diese Aktion wird deshalb ignoriert.

Mit der *package*-Ressource lassen sich auch mehrere Komponenten installieren oder managen, wie in den Zeilen 23-27 von Auflistung 5.4 abgebildet. Durch das Beispiel in Zeile 23 werden 2 Pakete installiert, die an Stelle des Ressourcennamen als Array angegeben sind. Folglich werden aus der Ressource zwei Pakete extrahiert. Dabei sind keine weiteren Eigenschaften in der Ressource angegeben. Für die Eigenschaft `package_name` wird folglich jeweils der Name verwendet. Da die Eigenschaft `action` nicht angegeben ist, wird der Standardwert *:install* angenommen. Zudem wird keine Eigenschaft `version` angegeben, wodurch keine Version der Pakete extrahiert werden kann. Das Beispiel in Zeile 25-28 zeigt dasselbe Beispiel mit angegebenen Versionen für die Pakete „package1“ und „package2“.

Jedes Kochbuch wird, wie in Abbildung 5.2 gezeigt, in die enthaltenen Konfigurationen übersetzt. Hat das transformierte Kochbuch Abhängigkeiten zu weiteren Kochbüchern, werden diese heruntergeladen und in die Transformation einbezogen. Nach der Transformation werden die abhängigen Kochbücher ebenfalls rekursiv transformiert. Dadurch werden die Kochbuch-Konfigurationen aus den abhängigen Kochbüchern extrahiert, wodurch das vorgestellte Architekturmodell resultiert (siehe Abbildung 5.2). Aus den Rezepten eines Kochbuches werden Softwarekomponenten extrahiert, die dort in verschiedenen Abwandlungen der *package*-Ressource deklariert werden (siehe Abschnitt 4.5.5). Die extrahierten Softwarekomponenten werden jeweils einer Kochbuch-Konfiguration zugeordnet, wobei dabei mit den extrahierten Komponenten unterschiedlich verfahren wird.

Extrahierte Komponenten werden entweder den Fähigkeiten oder den Anforderungen einer Kochbuch-Konfiguration zugeordnet. Die Zuordnung ist dabei abhängig vom Kochbuch, in dem sich das Rezept befindet. Diese Zuordnung wird anhand von Auflistung 5.4 erläutert, bei dem es sich um das Standardrezept des MyApp-Kochbuches aus dem laufenden Beispiel handelt, welches eine Abhängigkeit zu dem Java-Kochbuch aufweist. Die erste Anweisung des Rezeptes in Zeile 2 ruft das Standardrezept des

Auflistung 5.4 Default Rezept des MyApp-Kochbuches

```
1  # Aufruf von Java default.rb Rezept
2  include_recipe 'java::default'
3
4  package 'package1' do
5    package_name 'myapp'
6    version '1.1'
7    action :install
8  end
9
10 package node['myapp']['package_name'] do
11   version node['myapp']['package_version']
12   action :install
13 end
14
15 # Ab hier zur Demonstration der Extrahtion von package-Ressourcen
16 package 'name' do
17   package_name 'name of package'
18   version 'version of package'
19   action :action of package
20 end
21
22 # Mehrere Komponenten installieren
23 package %w(package1 package2)
24
25 package %w(package1 package2) do
26   version [ '1.3.4-2', '4.3.6-1']
27 end
```

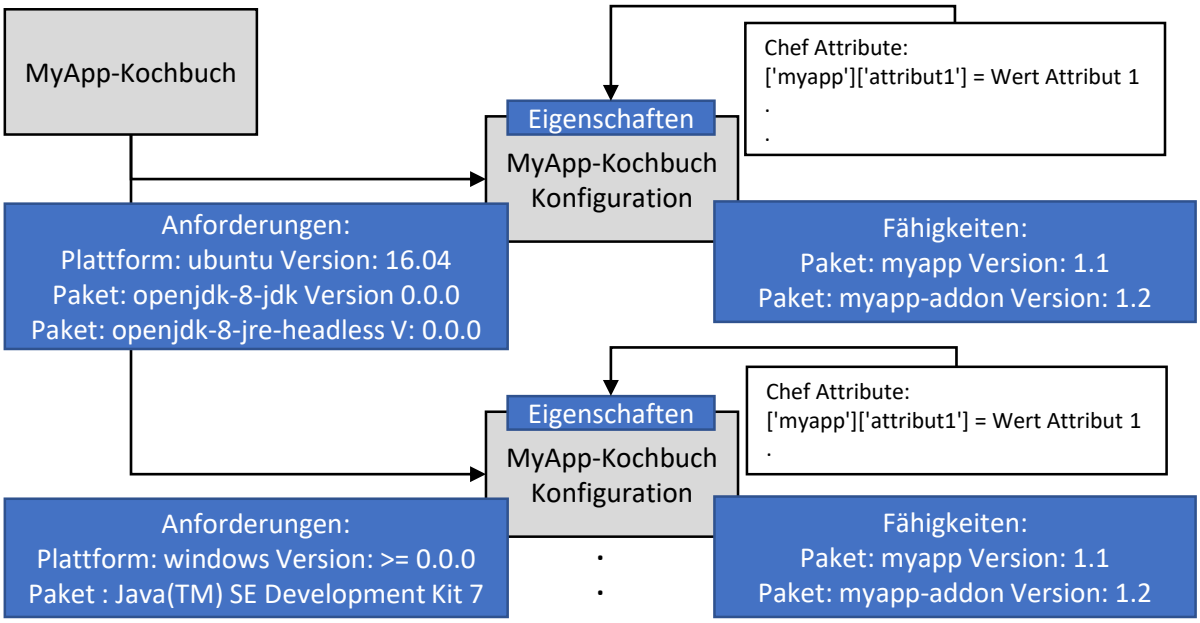

Abbildung 5.8: Softwarekomponenten werden aus Rezepten extrahiert und auf Fähigkeiten und Anforderungen der Konfigurationen abgebildet

Java-Kochbuches auf. Dadurch wird der Inhalt dieses Rezeptes genau an dieser Stelle ausgeführt. Der Inhalt enthält *package*-Ressourcen für das Installieren von Java-Komponenten. Da das Java-Kochbuch eine Abhängigkeit des MyApp-Kochbuches ist, werden die extrahierten Softwarepakete den Anforderungen der MyApp-Kochbuch-Konfigurationen zugeordnet. Aus dem aufgerufenen Java-Rezept können wiederum Rezepte von Abhängigkeiten des Java-Kochbuches aufgerufen werden. Da die Abhängigkeiten des Java-Kochbuches bei dessen Analyse aufgelöst werden, werden diese Rezeptaufrufe bei der Analyse des Kommandos `include_recipe` ignoriert. Die extrahierten Pakete aus eigenen Rezepten eines Kochbuches werden den Fähigkeiten der Kochbuch-Konfigurationen zugeordnet (siehe Auflistung 5.4 Zeile 4-13). Die Zuordnung von Softwarepaketen zu den Anforderungen und Fähigkeiten von Kochbuch-Konfigurationen wird in Abbildung 5.8 visualisiert. Diese zeigt zwei der extrahierten Kochbuch-Konfigurationen des MyApp-Kochbuches. Dabei wurden die Softwarepakete aus dem Java-Rezept den Anforderungen der MyApp-Kochbuch-Konfigurationen zugeordnet. Für die Konfiguration der Plattform Ubuntu-16.04 wurden als Anforderungen die Pakete *opendjk-8-jdk* und *opendjk-8-jre* aus dem Java-Kochbuch extrahiert. Da für die Pakete keine Version angegeben war, wird der Platzhalter *0.0.0* als Version verwendet. Die extrahierten Softwarepakete aus den Rezepten des MyApp-Kochbuches wurden den Fähigkeiten der MyApp-Kochbuch-Konfigurationen zugeordnet. Dabei wurde das Paket *myapp* in der Version *1.1* sowie das Paket *myapp-addon* in der Version *1.2* extrahiert.

In Chef Kochbüchern können benutzerdefinierte Ressourcen verwendet werden (siehe Abschnitt 4.5.5). Aus diesen muss extrahiert werden, welche Softwarepakete durch die aufgerufene Ressourcen installiert werden. Die extrahierten Pakete müssen analog zu normalen Chef Ressourcen, wie die *package*-Ressource, den Anforderungen und Fähigkeiten von Kochbuch-Konfigurationen zugeordnet werden. Ein Ansatz zum Parsen und Kompilieren von benutzerdefinierten Ressourcen würde den zeitlichen Rahmen der vorliegenden Arbeit überschreiten und wird deshalb nicht behandelt. An dieser Stelle wird deshalb auf weitere Forschungsmöglichkeiten hingewiesen.

5.7 Mapping auf das Architekturmodell

Bis an dieser Stelle wurde die Extraktion der Informationen zu der Darstellung der Deployment Architekturen aus Kochbüchern erläutert. Dabei wurden unter anderem Informationen wie Chef Attribute extrahiert, die nicht zu der Darstellung der Deployment Architekturen notwendig sind. Informationen die nicht zu der Darstellung der Deployment Architektur notwendig sind, werden nicht auf das Architekturmodell abgebildet.

Während der Transformation eines Kochbuches durch den Chef Kochbuch Compiler, werden den Kochbuch-Konfigurationen Eigenschaften wie Abhängigkeiten zu Kochbüchern, Chef Attribute und Ruby Variablen zugeordnet. Diese werden nach der Transformation nicht mehr benötigt und entfernt. Die Eigenschaften, die in den Kochbuch-Konfigurationen erhalten bleiben, sind Name und Version des Kochbuches. Diese dienen der eindeutigen Identifizierbarkeit der extrahierten Kochbuch-Konfigurationen.

Das Mapping der extrahierten Komponenten auf ein generisches Architekturmodell wird anhand von Abbildung 5.9 erläutert. Diese zeigt zwei Kochbuch-Konfigurationen sowie eine Plattform, die aus der Transformation des laufenden Beispiels resultieren. Aus dem MyApp-Kochbuch wurden die entsprechenden MyApp-Kochbuch-Konfigurationen extrahiert, wobei Abbildung 5.9 eine davon zeigt. Diese hat als Anforderung die Plattform Ubuntu 16.04. Da das MyApp-Kochbuch eine Abhängigkeit

zum Java-Kochbuch hat, wurden entsprechend die Java-Kochbuch-Konfigurationen extrahiert. Abbildung 5.9 zeigt zusätzlich die passende Java-Kochbuch-Konfiguration mit Plattform 16.04 als Anforderung. Beide Kochbuch-Konfigurationen ergeben die Deployment Architektur aus dem laufenden Beispiel auf der Plattform Ubuntu 16.04. Die extrahierten Kochbuch-Konfigurationen sind somit die Komponententypen im Architekturmodell (siehe Abschnitt 4.3.2). Die MyApp-Kochbuch-Konfiguration hat als Anforderungen die Komponenten *openjdk-8-jdk* und *openjdk-8-jre-headless*. Diese werden durch die Java-Kochbuch-Konfiguration aufgelöst, die als Fähigkeiten die Komponenten *openjdk-8-jdk* und *openjdk-8-jre-headless* hat. Da die Java-Kochbuch-Konfiguration als Anforderung die Plattform hat, wird die Plattform durch die Anforderungen der Java-Kochbuch-Konfiguration prinzipiell schon aufgelöst, sodass die MyApp-Kochbuch-Konfiguration keine Plattform mehr als Anforderung benötigt. Dabei ist wichtig von welcher Richtung die Abhängigkeiten der Komponenten aufgelöst werden. Werden diese von oben nach unten aufgelöst, also von der MyApp-Kochbuch-Konfiguration in Richtung Plattform, kann die Plattform aus den Anforderungen der MyApp-Kochbuch-Konfigurationen entfernt werden. Werden Abhängigkeiten von der Plattform her aufgelöst, geht dies nicht. In diesem Fall ist über die Anforderungen einer Kochbuch-Konfiguration nicht ersichtlich, welche die Richtige ist, da sich die Fähigkeiten auch zwischen Plattformen unterscheiden können. In dieser Arbeit wird angenommen, dass die Abhängigkeiten von oben aufgelöst werden. Dabei wird die Plattform entfernt, sobald die Kochbuch-Konfiguration eine Softwarekomponente eines anderen Kochbuches als Anforderung hat, da diese dann die Anforderung an die Plattform hat.

Aus den Kochbüchern lassen sich, abgesehen von Namen und Versionen der unterstützten Plattformen, keine weiteren Informationen extrahieren. Damit die Deployment Architektur vollständig ist, werden aus den extrahierten Plattformen ebenfalls Knotentypen generiert. Abbildung 5.9 zeigt deshalb, abgesehen von den Kochbuch-Konfigurationen, einen weiteren Knoten, der die Plattform *Ubuntu-16.04* repräsentiert. Die aus den Plattformen generierten Knotentypen erhalten als Fähigkeit den Namen und die Version der Plattform. Da aus einem Kochbuch keine Anforderungen an eine Plattform extrahiert werden, besitzt ein Plattformknoten keine Anforderungen. Im gezeigten Beispiel löst dieser die Anforderung der Java-Kochbuch-Konfiguration auf.

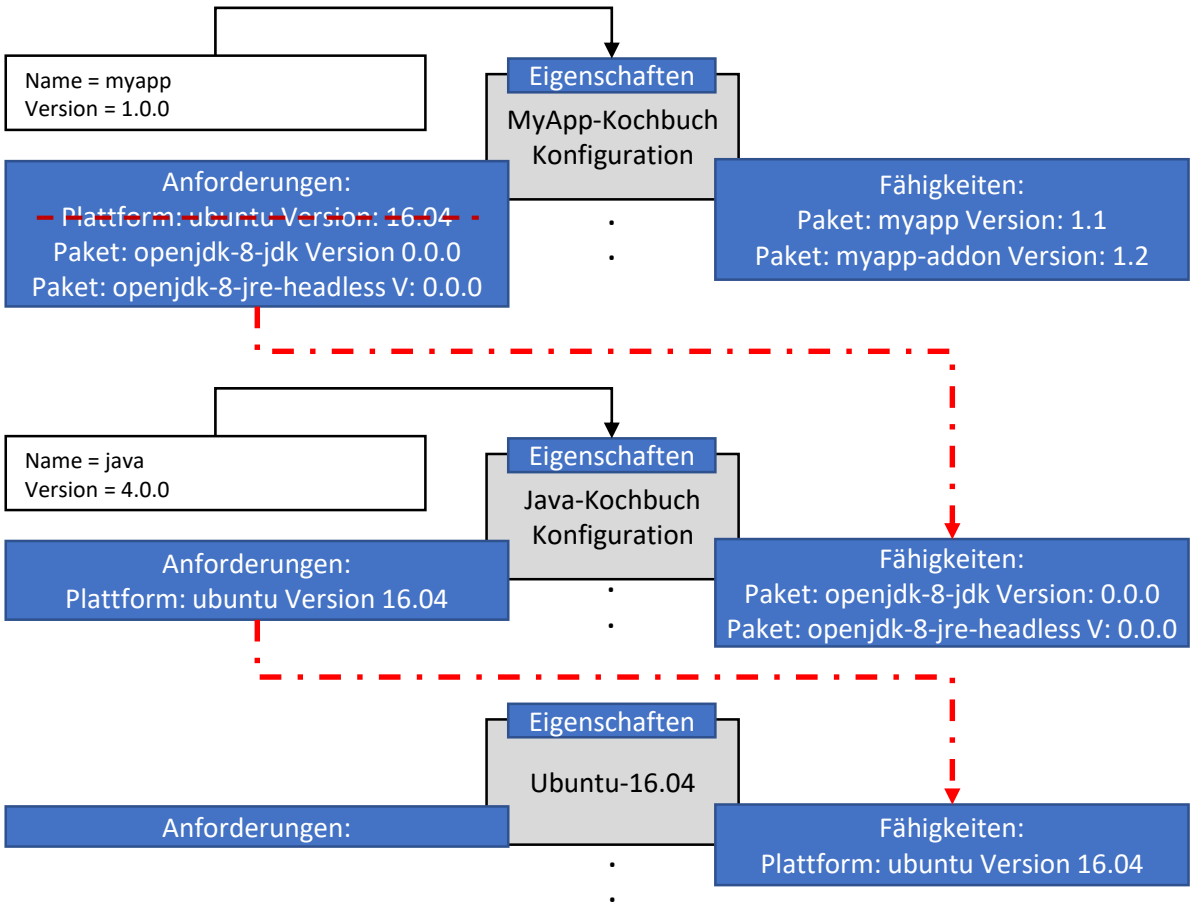

Abbildung 5.9: Finales Mapping der extrahierten Informationen auf das Architekturmodell

6 Prototyp und Validierung

In diesem Kapitel wird in Abschnitt 6.1 die konzeptionelle Machbarkeit für den vorgestellten Ansatz zur Extraktion und des Mappings von Chef Kochbüchern auf ein generisches Architekturmodell erläutert (siehe Kapitel 4 und Kapitel 5). In Abschnitt 6.2 wird validiert, ob der vorgestellte Ansatz die Deployment Architekturen korrekt aus den Kochbüchern des laufenden Beispiels extrahiert und auf das vorgestellte generische Architekturmodell abbildet (siehe Abschnitt 4.3).

6.1 Erweiterung der Winery

Die Machbarkeitsstudie wurde als Prototyp in die Eclipse Winery implementiert[1]. Für den Crawler wurde ein Plugin entwickelt, mit dem sich Chef Kochbücher aus dem Chef Supermarket crawlen lassen (siehe Abschnitt 4.2). Der in Abschnitt 4.4 und Kapitel 5 vorgestellte Ansatz, für die Extraktion der Deployment Architektur aus Chef Kochbüchern, ist der Hauptbeitrag dieser Arbeit, welcher als Erweiterung der Winery implementiert wurde. Die entwickelten Funktionalitäten arbeiten in einem Gesamtkonzept zusammen, welches Chef Kochbücher aus öffentlichen Repositories crawlt und auf der lokalen Festplatte speichert. Die gecrawlten Kochbücher werden in ihre enthaltenen Deployment Architekturen übersetzt. Der implementierte Prototyp löst dabei mögliche Abhängigkeiten zu weiteren Kochbüchern rekursiv auf, indem diese automatisch gecrawlt und ebenfalls analysiert werden.

Die Erweiterungen der Winery wurden nach dem Prinzip der testgetriebenen Entwicklung implementiert [Bec03]. Da bei der testgetriebenen Entwicklung klar ist, wann die Erweiterung fertig ist, macht das Schreiben der Tests vor der Implementierung der eigentlichen Funktionalität die Entwicklung vorhersehbar. Gleichzeitig wird mithilfe der Tests sichergestellt, dass andere Funktionalitäten von den aktuell durchgeführten Änderungen nicht beeinflusst werden. Durch die entwickelten Tests konnten Grenzfälle getestet werden, wodurch unvorhersehbare Fehler deutlich reduziert werden.

6.1.1 Crawler

Um das Wissen über lauffähige Architekturen zu extrahieren, muss es möglich sein lauffähige Chef Kochbücher aus öffentlichen Repositories zu crawlen. In Kapitel 4 wurde ein Ansatz dazu für den Chef Supermarket erläutert. Da die Eclipse Winery noch keine Funktionalität für das Crawlen von Chef Kochbüchern besitzt, wurde ein Crawler nach dem in Abschnitt 4.2 erläuterten Ansatz implementiert.

Die Firma Chef selbst stellt ein öffentlich zugängliches Repository für Chef Kochbücher zur Verfügung: den Chef Supermarket. Aus dem in Kapitel 4 und Kapitel 5 erläuterten Ansatz lassen sich für den

[1] https://github.com/OpenTOSCA/winery/pull/130

Crawler zwei Funktionalitäten ableiten. Zum einen muss der Crawler alle verfügbaren Kochbücher aus dem Chef Supermarket über die angebotene REST-API crawlen. Um in den Metadaten angegebene, abhängige Kochbücher herunterzuladen, müssen zum anderen Kochbücher auch durch Angabe von Name und Versionsbeschränkung gecrawlt werden können.

Für das Crawlen von lauffähigen Chef Kochbüchern wurde der beschriebene Ansatz aus Abschnitt 4.2 implementiert, mit dem alle verfügbaren Kochbücher aus dem Chef Supermarket gecrawlt werden können. Dabei wird die aktuellste Version jedes Kochbuches heruntergeladen. Die Antworten der API des Chef Supermarkets sind im JSON-Format[2]. In den Informationen, die der Crawler über die API als Antwort über jedes Kochbuch bekommt, ist ein boolescher Wert enthalten, der besagt, ob ein Kochbuch veraltet ist oder nicht. Um veraltete Informationen auszuschließen, werden veraltete Kochbücher nicht heruntergeladen. Der implementierte Crawler lässt sich parallelisieren. Folglich können die Kochbücher entweder alle der Reihe nach heruntergeladen werden oder der Crawlingvorgang der Kochbücher wird durch Threads parallelisiert.

Um Abhängigkeiten aufzulösen, wurde die Funktionalität implementiert, Kochbücher durch Übergabe von Name und Versionsbeschränkung zu crawlen. Dazu implementiert der Prototyp die Möglichkeit, Versionen, die in der semantischen Versionierung vorliegen, mit Versionsbeschränkungen zu vergleichen. Die Versionsbeschränkungen, die in Chef Kochbüchern vorkommen, sind in Tabelle 4.1 aufgeführt. Sie können kombiniert werden, sodass abhängige Kochbücher mit einem oder zwei Versionsbeschränkungen angegeben werden, wie in Tabelle 5.1 dargestellt. Dazu implementiert der Prototyp die Funktionalität, bis zu zwei angegebene Versionsbeschränkungen mit einer Kochbuchversion zu vergleichen. Diese kann für mehr als zwei Versionsbeschränkungen erweitert werden. Durch die Versionsvergleiche wird sichergestellt, dass die neueste, mögliche Version des abhängigen Kochbuches heruntergeladen wird.

6.1.2 Chef Kochbuch Compiler

Wie in Abschnitt 4.4 erläutert, wird für die Extraktion der Deployment Architektur ein spezialisierter Chef Kochbuch Compiler benötigt. Dieser übersetzt die Chef Kochbücher in die enthaltenen Architekturmodelle, welche die Deployment Architekturen repräsentieren.

Für die Umsetzung des vorgeschlagenen Ansatzes wurde ein spezialisierter Chef Kochbuch Compiler, nach dem vorgestellten Ansatz aus Abschnitt 4.4, entwickelt. Dieser hat die Funktionalität, Ruby-Quellcode aus Kochbüchern zu analysieren und die gesuchten Informationen in ein Zielmodell zu übersetzen (siehe Kapitel 5). Abbildung 6.1 zeigt den Aufbau sowie den Ablauf, des entwickelten Chef Kochbuch Compilers. Dieser wurde nach dem in Abbildung 4.5 dargestellten Ansatz entwickelt, hat jedoch die Vorverarbeitung der Dateien als zusätzlichen Schritt. Die Funktionalitäten des implementierten Chef Kochbuch Compilers werden in den folgenden Abschnitten erläutert.

Vorverarbeitung

Die Vorverarbeitung wurde als zusätzlicher Schritt in den Prototyp implementiert. In diesem wird der Inhalt der *.rb Dateien für die weitere Verarbeitung aufbereitet. Die Aufbereitung dient der Verein-

[2] https://www.json.org/json-de.html

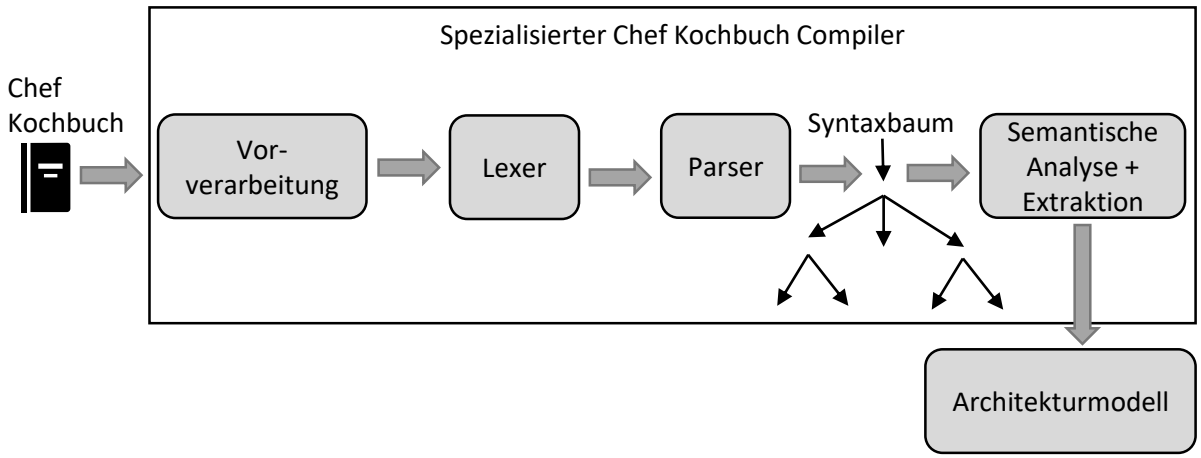

Abbildung 6.1: Prototyp des Chef Kochbuch Compilers

fachung des Parsers, welcher eine Folge aus Token erhält und den Syntaxbaum nach vorgegebenen Regeln erstellt. Dabei müssen alle Token aus dem Lexer zu den Grammatikregeln passen. Zur Vereinfachung der Grammatikregeln werden Leerzeichen und Tabs vom Lexer ignoriert. Folglich kommen diese in den Grammatikregeln nicht vor. Da anderer Leerraum wie Zeilenumbrüche an manchen Stellen das Ende einer Ruby-Anweisung markieren, kann dieser nicht ignoriert werden. Der Lexer erkennt dementsprechend Zeilenumbrüche als Token, was zur Folge hat, dass diese in den Grammatikregeln an jeder erlaubten Stelle vorkommen müssen. Allerdings kann an verschiedenen Stellen auf Zeilenumbrüche verzichtet werden. Um die Grammatik des Parsers zu vereinfachen, werden die überflüssigen Zeilenumbrüche in der Vorverarbeitung entfernt. Da bei der Vorverarbeitung in jedem Fall über alle Zeilen der Ruby-Dateien iteriert wird, hat diese eine Erweiterung, welche überflüssige Zeilen des Quellcodes entfernt. Daraus resultiert für den Lexer sowie den Parser weniger Arbeit und geringeres Fehlerpotential. Die vorverarbeiteten Dateien werden temporär gespeichert und weiterverarbeitet, sodass die Originaldateien durch die Vorverarbeitung nicht verändert werden.

Lexer und Parser

Lexer und Parser werden mit dem Parsergenerator ANTLR4[3] generiert. Die Regeln des Lexers wurden von der Ruby-Grammatik[4] übernommen und für die Syntax des Parsergenerators angepasst. Gleiches gilt für die Regeln des Parsers.

Lexer und Parser übersetzen den Ruby-Quellcode von Chef Kochbüchern in einen Syntaxbaum. Abbildung 6.2 zeigt den Syntaxbaum des in Auflistung 5.4 gezeigten Rezeptes, von Zeile 1 bis 9. Die Blattknoten des Baumes sind die Token, in die der Lexer den Quellcode zerlegt hat. Jedes Token wird einer übergeordneten Parserregel zugeordnet, die wiederum Teil einer weiteren Regel sein kann. Alle Regeln laufen an der Spitze des Syntaxbaumes zusammen. Eine geparste Datei mit Ruby-Quellcode wird somit unter dem obersten Knoten neu geordnet. Der in Abbildung 6.2 abgebildete Syntaxbaum

[3] https://www.antlr.org/
[4] https://ruby-doc.org/docs/ruby-doc-bundle/Manual/man-1.4/yacc.html

zeigt die zwei ersten Anweisungen aus Auflistung 5.4. Die linke Anweisung im Syntaxbaum ist der Rezeptaufruf und die zweite Anweisung ist die *package*-Ressource. Der Syntaxbaum lässt sich über Besuchermethoden durchlaufen, wodurch jeder Knoten im Syntaxbaum besucht werden kann. Die dafür notwendigen Methoden werden aus der Grammatik von Lexer und Parser durch ANTLR generiert. Diese geben in der Basisimplementierung den Inhalt des unterliegenden Knotens zurück. Um den Quellcode in die enthaltene Deployment Architektur zu übersetzen, müssen die Besuchermethoden entsprechend implementiert werden. Dieser Teil stellt die semantische Analyse dar.

Semantische Analyse

In der semantischen Analyse wird der Syntaxbaum durchlaufen und der Inhalt auf die Bedeutung überprüft. Die semantische Analyse erfolgt mittels der Besuchermethoden, die mit ANTLR generiert werden. Dazu werden für jede Parserregel entsprechende Methoden generiert. Mit den Methoden lässt sich jeder Knoten im Syntaxbaum besuchen. Standardmäßig werden die Knoten des Syntaxbaumes aus Abbildung 6.2 von links nach rechts besucht.

Durch Überschreiben der Besuchermethoden wird zum einen die Reihenfolge angepasst, wie die Knoten besucht werden. Zum anderen wird die Funktionalität angepasst, was geschieht, wenn ein Knoten besucht wird. Die erste Anweisung in Abbildung 6.2 enthält den Rezeptaufruf `include_recipe 'java::default'`. Wird zum Beispiel der Knoten *command:OperationCallArgs* in der linken Anweisung aus Abbildung 6.2 besucht, muss die Besuchermethode die Funktionalität implementieren, das angegebene Rezept aufzurufen. Durch die Grammatik wird definiert, dass der Knoten aus einer Operation sowie Argumenten besteht. Wenn die Operation den Wert *include_recipe* hat, wird dies durch entsprechende Bedingungen erkannt und die Methode schließt daraus, dass im Argument des Kommandos das aufgerufene Rezept enthalten ist.

Das erläuterte Vorgehen am Beispiel des Rezeptaufrufs ist auf alle weiteren Anweisungen übertragbar, die für die technischen Transformation von Kochbüchern in Kapitel 5 erkannt werden müssen. Der Prototyp implementiert alle Besuchermethoden, die dazu notwendig sind.

Der Quellcode von Chef Kochbüchern kann alles enthalten, was in der Programmiersprache Ruby möglich ist. Um die semantische Analyse zu verfeinern, implementiert der Prototyp die semantische Analyse von häufig verwendeten Ruby-Anweisungen. Dazu sind Besuchermethoden für If-Anweisungen, Switch-Case-Anweisungen und For-Schleifen implementiert. If-Anweisungen starten mit dem Schlüsselwort *if*, gefolgt von einem booleschen Ausdruck. Um den booleschen Ausdruck zu interpretieren, sind Besuchermethoden implementiert, die boolesche Ausdrücke als wahr oder falsch interpretieren können. Ruby erlaubt die Einbettung von Ruby-Code in Strings mittels String Interpolation. Bei Verwendung dieser Syntax wird alles zwischen den öffnenden `#{-` und schließenden `}` Bits als Ruby-Code ausgewertet und das Ergebnis in den umgebenden String eingebettet. Der entwickelte Chef Kochbuch Compiler implementiert diese Funktionalität, sodass Ruby-Code in Strings aufgelöst wird. Die Funktionalität wird häufig dazu verwendet, Strings mit Chef Attributen zusammenzubauen. Der Chef Kochbuch Compiler ersetzt aufgerufene Chef Attribute und Ruby Variablen mit dem entsprechenden Wert.

Jedes Chef Kochbuch muss die Datei *metadata.rb* im Verzeichnis haben (siehe Abschnitt 4.5.2). Diese wird kompiliert, wenn das Kochbuch auf einen Chef-Server hochgeladen wird und in der Datei *metadata.json* gespeichert. Beim Hochladen eines Kochbuches in den Chef Supermarket wird, abhängig vom Benutzer, entweder die Datei *metadata.rb*, die Datei *metadata.json* oder beide Dateien hochgeladen. Es muss

Abbildung 6.2: Syntaxbaum

mindestens eine von beiden Dateien in den Chef Supermarket hochgeladen werden. Folglich existieren im Chef Supermarket Kochbücher, welche beide oder nur eine von beiden Dateien aufweisen. Da ein Kochbuch ohne Metadaten nicht analysiert werden kann und dies in der Chef Dokumentation so publiziert wird, wurde in Kapitel 5 angenommen, dass die Datei *metadata.rb* immer vorhanden ist. Dies ist im Chef Supermarket aber nicht der Fall, weshalb der Prototyp mit der Funktionalität erweitert wurde, die Informationen auch aus der Datei *metadata.json* zu extrahieren. Ist keine der beiden Dateien vorhanden fehlen zur Transformation notwendige Informationen und der Vorgang wird abgebrochen. Der Prototyp implementiert damit die notwendigen Methoden für die Extraktion der gesuchten Informationen aus der Datei. Das Mapping der extrahierten Informationen ist analog zu der Datei *metadata.rb*. Zudem wurde die Funktionalität implementiert, aus der Datei *kitchen.yml* Plattforminformationen zu extrahieren, wenn die Datei in der Verzeichnisstruktur des Kochbuches existiert (siehe Abschnitt 5.3).

6.2 Validierung

In diesem Abschnitt wird die Implementierung validiert sowie die Anwendbarkeit der Implementierung beschrieben. In Abschnitt 6.2.1 wird dazu die Funktionalität sowie Leistungsfähigkeit des entwickelten Crawlers ermittelt und in Abschnitt 6.2.2 die Qualität der gecrawlten Kochbücher überprüft. In Abschnitt 6.2.3 wird der entwickelte Prototyp des Chef Kochbuch Compilers am laufenden Beispiel validiert und in Abschnitt 6.2.4 werden die extrahierten Deployment Architekturen auf TOSCA gemappt. Die in diesem Abschnitt präsentierten Ergebnisse wurden auf einem Windows 10 Rechner mit 2 Prozessorkernen mit 2,5 Ghz und 8 GB RAM erzielt.

6.2.1 Crawler

Unter Verwendung des implementierten Prototyps wurden am 12. April 2019 alle verfügbaren Kochbücher aus dem Chef Supermarket gecrawlt. Von den 3868 Kochbüchern waren 178 veraltet. Veraltete Kochbücher wurden dabei nicht gecrawlt. Auf ein Kochbuch war der Zugriff nicht möglich. Die heruntergeladenen Kochbücher sind in einem komprimierten Dateiformat. Diese werden vom Crawler entpackt. 13 Kochbücher konnten, aufgrund von ungültigen Zeichen, nicht entpackt werden. Damit wurden insgesamt 3676 Kochbücher heruntergeladen und entpackt.

Der entwickelte Crawler hat die Funktionalität, Kochbücher der Reihe nach oder parallelisiert herunterzuladen. Werden alle Kochbücher der Reihe nach heruntergeladen, dauert dies zwischen 2 und 2.5 Stunden. Für die Messung der Leistungsfähigkeit des Crawlers wurde der Crawlingvorgang parallelisiert. Wie in Abschnitt 4.2 erläutert, werden die Kochbücher in mehreren Iterationen heruntergeladen. Zur Parallelisierung des Vorgangs, werden die Kochbücher einer Iteration parallel heruntergeladen. Dabei wurde die maximale Anzahl von parallelen Downloads auf 50 begrenzt. Mit dieser Begrenzung lief der Crawlingvorgang in allen 10 durchgeführten Messungen stabil. Insgesamt wurde eine mittlere Dauer von 7 Minuten und 25 Sekunden ermittelt. Die Zahlen resultieren aus einem DSL-Anschluss mit 22 MBit/s Bandbreite. Zum Zeitpunkt der Messung waren keine weiteren Teilnehmer im Netzwerk und die real verfügbare Bandbreite des Anschlusses lag im Mittel bei 21 MBit/s.

Die komprimierten Kochbücher haben eine Größe von ungefähr 318 Megabyte und die entpackten Kochbücher eine Größe von ungefähr 650 Megabyte. Insgesamt werden knapp 1 Gigabyte Speicher für die Kochbücher benötigt.

6.2.2 Qualität der gecrawlten Kochbücher

Für die Extraktion der enthaltenen Deployment Architekturen wird eine bestimmte Beschaffenheit der Kochbücher vorausgesetzt, von der die Qualität der extrahierten Deployment Architekturen maßgeblich abhängt. In diesem Abschnitt wird die Qualität der gecrawlten Kochbücher im Bezug auf den vorgestellten Ansatz überprüft.

Der entwickelte Chef Kochbuch Compiler setzt die Datei *metadata.rb* voraus. Diese ist laut der Chef Dokumentation in jedem Kochbuch vorhanden. Wie in Abschnitt 6.1.2 erläutert, werden die Metadaten kompiliert, wenn das Kochbuch auf einen Chef-Server hochgeladen und in der Datei *metadata.json* gespeichert. Insgesamt wurden 3676 Kochbücher aus dem Chef Supermarket gecrawlt. Davon haben 2825 Kochbücher die Dateien *metadata.rb* und *metadata.json*. Bei 851 Kochbüchern ist nur die Datei *metadata.json* vorhanden und 0 Kochbücher haben ausschließlich die Datei *metadata.rb*. Durch die Erweiterung des Chef Kochbuch Compilers für die Datei *metadata.json* können zusätzlich 851 Kochbücher analysiert werden, weshalb die Erweiterung sinnvoll ist.

Um die Qualität der extrahierten Informationen zu erhöhen, wird versucht exakte Plattforminformationen aus der Datei *kitchen.yml* zu extrahieren. Von den gecrawlten Kochbüchern besitzen 1293 Kochbücher die Datei *kitchen.yml*, was einem Anteil von 35 % entspricht.

Für die Extraktion von Fähigkeiten und Anforderungen werden die Rezepte der gecrawlten Kochbücher analysiert. Die Rezepte befinden sich im *recipes*-Verzeichnis eines Kochbuches. Von den 3676 gecrawlten Kochbüchern besitzen 3444 das Verzeichnis mit Rezepten. Als Startrezept wird das Rezept *default.rb* benötigt, welches in 3294 der 3676 gecrawlten Kochbücher vorhanden ist. Da Anforderungen und Fähigkeiten der Kochbuch-Konfigurationen, abgesehen von der Plattform, nur aus Rezepten extrahiert werden, lassen sich aus 89.6 % der gecrawlten Kochbücher Kochbuch-Konfigurationen extrahieren.

Wie in Abschnitt 4.5.5 erläutert, lassen sich Rezepte mit eigenen Ressourcen erweitern. Diese befinden sich im Verzeichnis *resources* eines Kochbuches. Insgesamt verwenden 1025 der 3676 Kochbücher benutzerdefinierte Ressourcen. Zukünftige Forschungsarbeiten können eine Erweiterung des entwickelten Chef Kochbuch Compilers für benutzerdefinierte Ressourcen thematisieren.

6.2.3 Extraktion der Deployment Architektur aus Kochbüchern

Als Anwendungsfall für die Extraktion der Deployment Architekuren von Chef Kochbüchern aus öffentlichen Repositories wurde das laufende Beispiel aus Abschnitt 1.3 verwendet. Das MyApp-Kochbuch ist ein beispielhaftes Kochbuch, welches zur Erläuterung der Funktionsweise des in dieser Arbeit vorgestellten Ansatzes verwendet wurde. Es besitzt eine Abhängigkeit zu der aktuellsten Version des Java-Kochbuches, welches zu dem Zeitpunkt der Validierung die Version 4.0.0[5] ist. Das MyApp-Kochbuch wurde zu der Erklärung der Funktionsweise von Kochbüchern bewusst einfacher gehalten. Um den

[5] https://supermarket.chef.io/api/v1/cookbooks/java/versions/4.0.0

Ansatz auch für komplexere Kochbücher zu validieren, wurde das Java-Kochbuch als Abhängigkeit verwendet, bei dem es sich um ein komplexes Kochbuch mit vielen plattformabhängigen Bedingungen und mehreren Rezepten handelt. Zudem verwendet das Java-Kochbuch in den Rezepten keine benutzerdefinierten Ressourcen oder Shell-Skripte für die Installation von Softwarekomponenten. Das Java-Kochbuch wurde im Standardrezept des MyApp-Kochbuches aufgerufen (siehe Auflistung 5.4).

Tabelle 6.1 zeigt die extrahierten Kochbuch-Konfigurationen aus dem MyApp-Kochbuch. Insgesamt wurden 13 Kochbuch-Konfigurationen extrahiert. Die Anzahl stimmt mit der Anzahl der deklarierten Plattformen überein. Dabei werden die extrahierten Anforderungen und Fähigkeiten aufgeführt. Bei den Anforderungen wurden für die bessere Nachvollziehbarkeit die zugehörigen Plattformen in Klammern angegeben. Da das MyApp-Kochbuch keine plattformabhängigen Bedingungen besitzt und es so im MyApp-Kochbuch deklariert ist, sind die extrahierten Fähigkeiten der Kochbuch-Konfigurationen alle identisch, was manuell auf syntaktische Korrektheit überprüft wurde. Bei den extrahierten Anforderungen sind Unterschiede in den Namen der extrahierten Komponenten zu sehen. Diese resultieren aus den plattformabhängigen Bedingungen, die im Java Kochbuch im Rezept *set_attributes_from_version.rb* deklariert sind und dort auf Basis des automatischen Attributes `['platform_family']` formuliert wurden, welches bei der Extraktion der Plattformen automatisch generiert wird. Die extrahierten Anforderungen wurden syntaktisch korrekt nach der Deklaration im Java-Kochbuch extrahiert. Dies wurde manuell anhand der Deklarationen im Java-Kochbuch validiert. Da Java auf der Plattform *mac_os_x-10.12* vorinstalliert ist, hat beispielsweise Konfiguration 8 als Anforderung nur die Plattform [Ora19]. Analog dazu zeigt Tabelle 6.2 einen Ausschnitt aus den extrahierten Java-Kochbuch-Konfigurationen. Da Softwarekomponenten zur Installation im Java-Kochbuch deklariert werden, haben die extrahieren Kochbuch-Konfigurationen die installierten Komponenten als Fähigkeiten. Da die Java-Komponenten keine weiteren Abhängigkeiten zu anderen Softwarekomponenten haben, haben diese als Anforderungen die entsprechenden Plattformen. Da durch Tabelle 6.1 gezeigt wurde, dass die Komponenten des Java-Kochbuches korrekt extrahiert werden, sind die Java-Konfigurationen nicht alle aufgelistet.

Die Extraktion der MyApp-Kochbuch-Konfigurationen dauerte in zehn Versuchen zwischen 41-51 Sekunden. Die Extraktion der Java-Kochbuch-Konfigurationen dauerte in den zehn Versuchen zwischen 51-58 Sekunden. Da die Rezepte ausgehend vom Startrezept für jede Kochbuch-Konfiguration wiederholt geparst und analysiert werden müssen, ist die Dauer hauptsächlich auf die Komplexität des Java-Kochbuches zurückzuführen. Das Ziel dieser Arbeit ist ein effizienter Prozess für die Extraktion der Deployment Architekturen aus Chef Kochbüchern. Da das zeitaufwändige Parsen und Analysieren der Kochbücher nur einmal durchgeführt werden muss, ist der vorgeschlagene Ansatz dafür geeignet.

#	**Anforderungen**	**Fähigkeiten**
1	java-1_8_0-openjdk-0.0.0 java-1_8_0-openjdk-devel-0.0.0 (suse->= 0.0.0)	myapp-1.1 myappaddon-1.2
2	java-1_8_0-openjdk-0.0.0 java-1_8_0-openjdk-devel-0.0.0 (opensuseleap-42)	myapp-1.1 myappaddon-1.2
3	sun-jdk8-0.0.0 sun-jre8-0.0.0 (smartos->= 0.0.0)	myapp-1.1 myappaddon-1.2
4	openjdk-8-jdk-0.0.0 openjdk-8-jre-headless-0.0.0 (ubuntu-16.04)	myapp-1.1 myappaddon-1.2
5	openjdk-8-jdk-0.0.0 openjdk-8-jre-headless-0.0.0 (ubuntu-14.04)	myapp-1.1 myappaddon-1.2
6	java-1_8_0-openjdk-0.0.0 java-1_8_0-openjdk-devel-0.0.0 (fedora-29)	myapp-1.1 myappaddon-1.2
7	openjdk8-0.0.0 (freebsd-11)	myapp-1.1 myappaddon-1.2
8	mac_os_x-10.12	myapp-1.1 myappaddon-1.2
9	openjdk-8-jdk-0.0.0 openjdk-8-jre-headless-0.0.0 (debian-8)	myapp-1.1 myappaddon-1.2
10	openjdk-8-jdk-0.0.0 openjdk-8-jre-headless-0.0.0 (debian-9)	myapp-1.1 myappaddon-1.2
11	Java(TM) SE Development Kit 7 (64-bit)-0.0.0 (windows-10)	myapp-1.1 myappaddon-1.2
12	java-1_8_0-openjdk-0.0.0 java-1_8_0-openjdk-devel-0.0.0 (centos-6)	myapp-1.1 myappaddon-1.2
13	java-1_8_0-openjdk-0.0.0 java-1_8_0-openjdk-devel-0.0.0 (centos-7)	myapp-1.1 myappaddon-1.2

Tabelle 6.1: Extrahierte Kochbuch-Konfigurationen aus dem MyApp-Kochbuch

#	**Anforderungen**	**Fähigkeiten**
1	suse->= 0.0.0	java-1_8_0-openjdk-0.0.0 java-1_8_0-openjdk-devel-0.0.0
2	opensuseleap-42	java-1_6_0-openjdk-0.0.0 java-1_8_0-openjdk-devel-0.0.0
3	smartos->= 0.0.0	sun-jdk8-0.0.0 sun-jre8-0.0.0
4	ubuntu-16.04	openjdk-8-jdk-0.0.0 openjdk-8-jre-headless-0.0.0
...	...	...

Tabelle 6.2: Auszug der extrahierten Kochbuch-Konfigurationen aus dem Java-Kochbuch

6.2.4 Mapping auf TOSCA Topologien

Der Prototyp wurde in die Eclipse Winery (siehe Abschnitt 2.2) implementiert. Dies ist ein Modellierungs-Werkzeug für TOSCA-basierte Cloud-Anwendungen, welches hauptsächlich mit dem TOSCA-Standard arbeitet. Es bietet die Möglichkeit, die Topologie von Cloud-Aapplikationen standardisiert und unabhängig vom Anbieter zu beschreiben [BBKL14a]. Da die Winery zur Modellierung von Topologien den TOSCA-Standard verwendet, werden die extrahierten Deployment Architekturen auf diesen abgebildet. Das in dieser Arbeit in Abschnitt 4.3 erläuterte generische Architekturmodell wurde bewusst allgemein gehalten und kann auf TOSCA gemappt werden. In Abschnitt 4.3.1 wurden Anforderungen an ein Architekturmodell formuliert. Der TOSCA-Standard erfüllt diese Anforderungen, weshalb sich die extrahierten Kochbuch-Konfigurationen auf diesen abbilden lassen. Ein Komponententyp ist in TOSCA ein *NodeType*. Eine Anforderung eines Knotentyps entspricht einer *Requirement Definition* und eine Fähigkeit einer *Capability Definition*. Das Mapping wird anhand von Auflistung 6.1 und Auflistung 6.2 gezeigt. Auflistung 6.1 zeigt eine extrahierte MyApp-Kochbuch-Konfiguration auf Basis der Plattform Ubuntu 16.04 im TOSCA-Standard und Auflistung 6.3 zeigt die extrahierte Java-Kochbuch-Konfiguration auf Basis der Plattform Ubuntu 16.04, mit der die Anforderungen der extrahierten MyApp-Kochbuch Konfiguration aufgelöst werden können. Auflistung 6.2 zeigt den Knoten der Plattform Ubuntu 16.04, mit dem die Anforderung der Java-Kochbuch-Konfiguration an die Plattform aufgelöst wird. Die drei Komponenten ergeben zusammen eine der extrahierten Deployment Architekturen, die im laufenden Beispiel in Abschnitt 1.3 enthalten sind.

Auflistung 6.1 MyApp-Kochbuch-Konfiguration im TOSCA Standard

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <Definitions xmlns="http://docs.oasis-open.org/tosca/ns/2011/12"
       xmlns:winery="http://www.opentosca.org/winery/extensions/tosca/2013/02/12"
       xmlns:selfservice="http://www.eclipse.org/winery/model/selfservice"
       xmlns:testwineryopentoscaorg="http://test.winery.opentosca.org"
       targetNamespace="https://supermarket.chef.io/api/v1/cookbooks/myapp/versions/1.0.0"
       id="winery-defs-for_ns1-myapp-w1">
3        <NodeType name="myapp-w1"
             targetNamespace="https://supermarket.chef.io/api/v1/cookbooks/myapp/versions/1.0.0">
4              <RequirementDefinitions>
5                    <RequirementDefinition
                         xmlns:ns1="https://supermarket.chef.io/api/v1/cookbooks/myapp/versions/1.0.0"
                         name="package-w1" requirementType="ns1:openjdk-8-jdk-0.0.0"/>
6                    <RequirementDefinition
                         xmlns:ns1="https://supermarket.chef.io/api/v1/cookbooks/myapp/versions/1.0.0"
                         name="package-w2" requirementType="ns1:openjdk-8-jre-headless-0.0.0"/>
7              </RequirementDefinitions>
8              <CapabilityDefinitions>
9                    <CapabilityDefinition
                         xmlns:ns1="https://supermarket.chef.io/api/v1/cookbooks/myapp/versions/1.0.0"
                         name="package-w1" capabilityType="ns1:myapp-1.1"/>
10                   <CapabilityDefinition
                         xmlns:ns1="https://supermarket.chef.io/api/v1/cookbooks/myapp/versions/1.0.0"
                         name="package-w2" capabilityType="ns1:myappaddon-1.2"/>
11             </CapabilityDefinitions>
12       </NodeType>
13 </Definitions>
```

Auflistung 6.2 Plattform im TOSCA Standard

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <Definitions xmlns="http://docs.oasis-open.org/tosca/ns/2011/12"
       xmlns:winery="http://www.opentosca.org/winery/extensions/tosca/2013/02/12"
       xmlns:selfservice="http://www.eclipse.org/winery/model/selfservice"
       xmlns:testwineryopentoscaorg="http://test.winery.opentosca.org"
       targetNamespace="https://supermarket.chef.io/api/v1/platforms/"
       id="winery-defs-for_platforms-ubuntu-16.04-w1">
3        <NodeType name="ubuntu-16.04-w1"
             targetNamespace="https://supermarket.chef.io/api/v1/platforms/">
4              <CapabilityDefinitions>
5                    <CapabilityDefinition
                         xmlns:platforms="https://supermarket.chef.io/api/v1/platforms/"
                         name="platform" capabilityType="platforms:ubuntu-16.04-w1"/>
6              </CapabilityDefinitions>
7        </NodeType>
8  </Definitions>
```

Auflistung 6.3 Java-Kochbuch-Konfiguration im TOSCA Standard

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <Definitions xmlns="http://docs.oasis-open.org/tosca/ns/2011/12"
       xmlns:winery="http://www.opentosca.org/winery/extensions/tosca/2013/02/12"
       xmlns:selfservice="http://www.eclipse.org/winery/model/selfservice"
       xmlns:testwineryopentoscaorg="http://test.winery.opentosca.org"
       targetNamespace="https://supermarket.chef.io/api/v1/cookbooks/java/versions/4.0.0"
       id="winery-defs-for_ns0-java-w1">
3        <NodeType name="java-w1"
             targetNamespace="https://supermarket.chef.io/api/v1/cookbooks/java/versions/4.0.0">
4              <RequirementDefinitions>
5                    <RequirementDefinition
                         xmlns:ns0="https://supermarket.chef.io/api/v1/cookbooks/java/versions/4.0.0"
                         name="supported plattform" requirementType="ns0:ubuntu-16.04-w1"/>
6              </RequirementDefinitions>
7              <CapabilityDefinitions>
8                    <CapabilityDefinition
                         xmlns:ns0="https://supermarket.chef.io/api/v1/cookbooks/java/versions/4.0.0"
                         name="package-w1" capabilityType="ns0:openjdk-8-jdk-0.0.0"/>
9                    <CapabilityDefinition
                         xmlns:ns0="https://supermarket.chef.io/api/v1/cookbooks/java/versions/4.0.0"
                         name="package-w2" capabilityType="ns0:openjdk-8-jre-headless-0.0.0"/>
10             </CapabilityDefinitions>
11       </NodeType>
12 </Definitions>
```

7 Limitationen

In diesem Kapitel werden die Limitationen des vorgestellten Ansatzes zur Extraktion der Deployment Architekturen aus Chef Kochbüchern erläutert. In Rezepten sowie Attributdateien von Kochbüchern werden häufig automatische Attribute verwendet, die plattformabhängig sind, weshalb die Qualität der extrahierten Kochbuch-Konfigurationen von den extrahierten Plattformen abhängt. In den meisten Kochbüchern werden in den Metadaten die unterstützten Plattformen ohne Versionsbeschränkung angegeben. Um die Deployment Architekturen detaillierter zu extrahieren, wird versucht zusätzliche Plattformen aus der Datei *kitchen.yml* zu extrahieren. In der Datei *kitchen.yml* werden die Plattformen mit Name und Version angegeben. Um plattformabhängige Bedingungen innerhalb von Rezepten und Attributdateien aufzulösen, sind diese Angaben genau genug. Dabei werden Bedingungen aufgelöst, die von Name und Version der Plattformen abhängig sind. Bedingungen, die von speziellen Eigenschaften, wie der Kernelversion, abhängig sind, können nicht aufgelöst werden. Wenn in den Metadaten keine Versionen angegeben sind und die Datei *kitchen.yml* nicht existiert, lassen sich plattformabhängige Bedingungen, die von der Plattformversion abhängig sind, nicht auflösen. Ein Ansatz dies zu lösen, ist das Generieren aller von Chef unterstützen Plattformversionen, wenn in den Metadaten angegeben ist, dass alle Versionen einer Plattform unterstützt werden. Wenn beispielsweise angegeben ist, dass alle Versionen der Plattform Ubuntu unterstützt werden, können aus dieser Information die Plattformen Ubuntu-14.04, Ubuntu-16.04 und Ubuntu-18.04 generiert werden. Die Liste aller von Chef unterstützten Plattformen findet sich in der Chef Dokumentation[1]. Da die Angaben zu den unterstützten Plattformen in den Metadaten nicht bindend sind, hat der Ansatz den Nachteil, dass die Richtigkeit dieser Annahme nicht gegeben sein muss. Außerdem macht dies den Chef Kochbuch Compiler abhängig von Informationen außerhalb der Kochbücher, wodurch der Wartungsaufwand erhöht wird. Die Abwägung von Vorteilen und Nachteilen dieses Ansatzes bietet Forschungsmöglichkeiten für zukünftige Arbeiten.

Eine weitere Limitation des vorgestellten Ansatzes ist die Validität der extrahierten Plattformen. Die Angaben zu unterstützten Plattformen innerhalb eines Kochbuches sind nicht bindend und folglich nicht zwingend korrekt. Da die Extraktion der Deployment Architekturen ohne die Plattforminformationen aus den Metadaten und der Datei *kitchen.yml* stark beeinträchtigt, bis unmöglich ist, müssen die Plattforminformationen jedoch in der Analyse verwendet werden. Die extrahierten Deployment Architekturen sind deshalb als Richtwert zu behandeln und es wird davon abgeraten, diese in die vollautomatische Entscheidung von möglicherweise sicherheitsrelevanten Entscheidungen einzubeziehen.

Eine der Hauptlimitationen des Chef Kochbuch Compilers ist die Unterstützung eingebetteter Shell-Skripte und benutzerdefinierter Ressourcen durch Chef Kochbücher, welche der Chef Kochbuch Compiler aktuell nicht interpretieren kann. Shell-Skripte werden über Chef Ressourcen wie *bash*, *script*, *execute*, *batch*, *perl*, *ruby* oder *python* in Chef Rezepte eingebaut. In der Chef Dokumentation wird

[1] https://docs.chef.io/platforms.html

darauf hingewiesen, die *package*-Ressource für die Installation von Softwarekomponenten zu verwenden [Inc19a]. Für bestimmte Aufgaben sind Shell-Skripte allerdings unverzichtbar, wie beispielsweise um Software zu installieren, die nicht in freigegebenen Software-Repositories verfügbar ist. Die behandelten Ressourcentypen haben eine klare Semantik, die Shell-Skripte oft nicht aufweisen. Das Parsen von benutzerdefinierten Ressourcen wurde aus Zeitgründen nicht behandelt. Zur vollständigen Extraktion der verwendeten Softwarekomponenten innerhalb eines Kochbuches ist weitere Forschung, bei der semantischen Analyse von Shell-Skripten und benutzerdefinierten Ressourcen, notwendig.

Bei der Transformation von abhängigen Kochbüchern wurde angenommen, dass Rezepte des abhängigen Kochbuches über das Standardrezept (default.rb) eingebunden werden. Bei der anschließenden rekursiven Transformation der Abhängigkeiten wurde ebenfalls das Standardrezept als Startpunkt der Transformation angenommen. Tatsächlich können auch andere Rezepte des abhängigen Kochbuches aufgerufen werden. Werden beispielsweise andere Rezepte in den Rezepten des MyApp-Kochbuches aufgerufen, wird dies bei der Transformation des MyApp-Kochbuches zwar berücksichtigt, bei der anschließenden Transformation des Java-Kochbuches wird allerdings nur von dem Standardrezept ausgegangen. Da das zusätzliche Rezept nicht transformiert wird, kann es in diesem Fall passieren, dass in den extrahierten Java-Kochbuch-Konfigurationen Softwarepakete in den Fähigkeiten fehlen. Ein Ansatz dies zu lösen, ist die Übergabe der aufgerufenen Rezepte an die rekursive Transformation des abhängigen Kochbuches. Da Rezeptaufrufe auch in plattformabhängigen Bedingungen vorkommen können, erhöht dies die Komplexität deutlich und die Lösung ist nicht trivial. Aus Zeitgründen wurde dieser Ansatz in der vorliegenden Arbeit nicht weiterverfolgt. Zudem führt die Annahme in dieser Arbeit zwar zu fehlenden, nicht aber zu falschen Informationen. Zur Optimierung der Transformation wird an dieser Stelle auf weitere Forschungsmöglichkeiten hingewiesen.

Die beispielhafte Anwendung MyApp aus dem laufenden Beispiel verwendet Rezepte des Java-Kochbuches, welches in der Standardkonfiguration für die Installation und Konfiguration von Java 8 vorgesehen ist. Am laufenden Beispiel wurde das Java-Kochbuch in seiner Standardkonfiguration verwendet. Es kann der Fall auftreten, dass die Anwendung MyApp statt Java 8 beispielsweise Java 11 benötigt. Dies lässt sich über die Chef Attribute des Java-Kochbuches anpassen. Dazu gibt es zwei Möglichkeiten. Möglichkeit eins ist die Anpassung der Chef Attribute im Java-Kochbuch bevor der Knoten konfiguriert wird. Da das Attribut erst im Nachhinein angepasst wird, gibt es in diesem Fall keinerlei Möglichkeit für den vorgestellten Ansatz diese Anpassung zu extrahieren. Möglichkeit zwei ist die Überschreibung des entsprechenden Chef Attributes innerhalb der Rezepte des MyApp-Kochbuches. Da bei der Extraktion der MyApp-Kochbuch-Konfigurationen die entsprechenden Chef Attribute extrahiert werden, wird dieser Fall teilweise von dem vorgestellten Ansatz erfasst, womit die Java 11 Komponenten als Anforderungen der MyApp-Kochbuch-Konfigurationen extrahiert werden. Bei der nachfolgenden Transformation des Java-Kochbuches wird es allerdings in der Standardkonfiguration transformiert, weshalb die extrahieren Java-Kochbuch-Konfigurationen die Java 8 Komponenten als Fähigkeiten besitzen. Um die Anforderungen der MyApp-Kochbuch-Konfigurationen aufzulösen, fehlen in diesem Fall die passenden Java-Kochbuch-Konfigurationen. Dieser Fall wird von dem vorgestellten Ansatz für die Extraktion der Deployment Architekturen nicht gelöst und es wird an dieser Stelle auf weitere Forschungsmöglichkeiten hingewiesen. Dies könnte beispielsweise durch eine Übergabe der, während der Transformation des MyApp-Kochbuches extrahierten, dem Java-Kochbuch zugehörigen, Chef Attribute an die rekursive Transformation des Java-Kochbuches, gelöst werden. Da die Chef-Attribute sich aufgrund plattformabhängiger Bedingungen zwischen den Kochbuch-Konfigurationen unterscheiden können, weist dieser Teil eine ähnliche Komplexität auf, wie die im vorherigen Absatz erläuterte Annahme des Startrezepts bei abhängigen Kochbüchern.

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Ansatz präsentiert, um (i) Chef Kochbücher aus öffentlichen Repositories, wie dem Chef Supermarket zu crawlen und (ii) die Deployment Architekturen der gecrawlten Kochbücher zu extrahieren und auf ein generisches Architekturmodell abzubilden.

Für die Extraktion von Kenntnissen über lauffähige Deployment Architekturen aus Chef Kochbüchern werden lauffähige Kochbücher benötigt, welche alle Information für die Installation und Konfiguration einer Anwendung oder eines Szenarios enthalten. Lauffähige Kochbücher werden in öffentlichen Repositories, wie dem Chef Supermarket, veröffentlicht. Dazu wurde ein Crawler entwickelt, der alle verfügbaren Chef Kochbücher aus dem Chef Supermarket crawlt und auf der lokalen Festplatte speichert. Zur Darstellung der Deployment Architekturen wurde ein generisches Architekturmodell vorgestellt, mit dem sich die Komponenten einer Anwendungsarchitektur und ihre Beziehungen untereinander darstellen lassen. Für die Transformation der Kochbücher wurde ein spezialisierter Chef Kochbuch Compiler vorgestellt, mit dem die Deployment Architekturen aus einem Kochbuch extrahiert und auf das Architekturmodell abgebildet werden. Dieser liest und analysiert die Dateien aus der Verzeichnisstruktur eines Kochbuches in einer definierten Reihenfolge, crawlt automatisch abhängige Kochbücher und transformiert die Abhängigkeiten rekursiv.

Die Vorteile sind (a) ein Ansatz für die Extraktion von Wissen über Softwarekomponenten und Beziehungen untereinander aus Chef Kochbüchern, das als Basis für eine automatisierte Kompatibilitätsüberprüfung dient, (b) ein generisches Architekturmodell für die Darstellung der Deployment Architekturen von Chef Kochbüchern, dass auch für die Darstellung von Deployment Architekturen von DevOps-Artefakten anderer Technologien anwendbar ist, (c) ein erweiterbarer Ansatz zur Extraktion von weiterem Wissen aus Kochbüchern und (d) eine Erweiterung der Eclipse Winery um ein Crawling Framework für Chef Kochbücher.

Die Validierung des Ansatzes wurde mit einer konzeptionellen Implementierung als Erweiterung des TOSCA-Modellierungswerkzeugs Eclipse Winery gezeigt. Dabei wurde gezeigt, dass der Ansatz, Kochbücher aus öffentlichen Repositories zu crawlen und die enthaltenen Deployment Architekturen zu extrahieren, anwendbar ist. Wie in Kapitel 7 erläutert, bietet der entwickelte Chef Kochbuch Compiler Optimierungsmöglichkeiten bei der Transformation der Kochbücher, zur Verbesserung von Qualität und Quantität der extrahierten Informationen.

Eine Limitation des vorgestellten Ansatzes ist die Annahme, dass die Konfigurationen in den gecrawlten Kochbücher korrekt deklariert sind. Aus Zeitgründen wurde die Überprüfung der syntaktischen und semantischen Korrektheit nicht behandelt. Eine weitere Limitierung der Arbeit ist zudem, dass in Chef Kochbüchern auch eingebettete Shell-Skript verwendet werden dürfen. Auf Shell-Skripte soll bei der Installation von Softwarekomponenten, laut den Chef Richtlinien, verzichtet werden, aber für bestimmte Aufgaben sind sie unverzichtbar. Zukünftige Arbeiten könnten die semantische Analyse solcher Skripte einschließen. Shell-Skripte werden auch in DevOps-Artefakten anderer Technologien,

wie beispielsweise Dockerfiles oder Puppet Manifesten, verwendet, weshalb deren Interpretation auch bei der Transformation dieser Artefakte hilfreich sein kann.

Bei der rekursiven Transformation von abhängigen Kochbüchern gibt es weitere Forschungsmöglichkeiten, welche die Übergabe von Chef Attributen und aufgerufener Rezepte aus dem übergeordneten Chef Kochbuch betreffen. Da eine Transformation aller Testfälle ohne Kenntnis des getesteten Szenarios zu fehlerhaften extrahierten Kochbuch-Konfigurationen führt, kann weitere Forschung an der Auswahl der Tests in der Datei *kitchen.yml* zur Extraktion von zusätzlichen lauffähigen Konfigurationen durchgeführt werden. Da Puppet Manifeste im Aufbau Ähnlichkeiten zu Chef Kochbüchern aufweisen, könnte der vorgestellte Ansatz auf diese angewendet werden.

Literaturverzeichnis

[Aho03] A. V. Aho. *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003 (zitiert auf S. 9, 30, 62).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. „OpenTOSCA–a runtime for TOSCA-based cloud applications". In: *International Conference on Service-Oriented Computing*. Springer. 2013, S. 692–695 (zitiert auf S. 8).

[BBKL13] T. Binz, U. Breitenbucher, O. Kopp, F. Leymann. „Automated discovery and maintenance of enterprise topology graphs". In: *Service-Oriented Computing and Applications (SOCA), 2013 IEEE 6th International Conference on*. IEEE. 2013, S. 126–134 (zitiert auf S. 20).

[BBKL14a] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. „TOSCA: portable automated deployment and management of cloud applications". In: *Advanced Web Services*. Springer, 2014, S. 527–549 (zitiert auf S. 13, 80).

[BBKL14b] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. „Vinothek-A Self-Service Portal for TOSCA." In: *ZEUS*. Citeseer. 2014, S. 69–72 (zitiert auf S. 8).

[BCSV04] P. Boldi, B. Codenotti, M. Santini, S. Vigna. „Ubicrawler: A scalable fully distributed web crawler". In: *Software: Practice and Experience* 34.8 (2004), S. 711–726 (zitiert auf S. 11).

[Bec03] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003 (zitiert auf S. 71).

[BFL+12] T. Binz, C. Fehling, F. Leymann, A. Nowak, D. Schumm. „Formalizing the cloud through enterprise topology graphs". In: *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE. 2012, S. 742–749 (zitiert auf S. 13, 14).

[BLNS12] T. Binz, F. Leymann, A. Nowak, D. Schumm. „Improving the manageability of enterprise topologies through segmentation, graph transformation, and analysis strategies". In: *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*. IEEE. 2012, S. 61–70 (zitiert auf S. 14).

[Bre16] U. Breitenbücher. „Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements". In: (2016) (zitiert auf S. 14, 15, 28, 29).

[Cas05] C. Castillo. „Effective web crawling". In: *Acm sigir forum*. Bd. 39. 1. Acm. 2005, S. 55–56 (zitiert auf S. 11).

[CVD99] S. Chakrabarti, M. Van den Berg, B. Dom. „Focused crawling: a new approach to topic-specific Web resource discovery". In: *Computer networks* 31.11–16 (1999), S. 1623–1640 (zitiert auf S. 12).

[EBLW17] C. Endres, U. Breitenbücher, F. Leymann, J. Wettinger. „Anything to Topology - A Method and System Architecture to Topologize Technology-Specific Application Deployment Artifacts“. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017), Porto, Portugal*. SCITEPRESS, Apr. 2017, S. 180–190. ISBN: 978-989-758-243-1 (zitiert auf S. 12, 13, 18, 19).

[EMT01] J. Edwards, K. McCurley, J. Tomlin. „An adaptive model for optimizing performance of an incremental web crawler“. In: *Proceedings of the 10th international conference on World Wide Web*. ACM. 2001, S. 106–113 (zitiert auf S. 12).

[HBBL+14] P. Hirmer, U. Breitenbücher, T. Binz, F. Leymann et al. „Automatic Topology Completion of TOSCA-based Cloud Applications.“ In: *GI-Jahrestagung*. 2014, S. 247–258 (zitiert auf S. 7).

[HF10] J. Humble, D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010 (zitiert auf S. 1).

[HHD16] O. Hanappi, W. Hummer, S. Dustdar. „Asserting reliable convergence for configuration management scripts“. In: *ACM SIGPLAN Notices* 51.10 (2016), S. 328–343 (zitiert auf S. 20).

[HM11] J. Humble, J. Molesky. „Enterprises Must Adopt Devops to Enable Continuous Delivery“. In: Cutter IT Journal, vol. 24, 2011 (zitiert auf S. 1).

[HN99] A. Heydon, M. Najork. „Mercator: A scalable, extensible web crawler“. In: *World Wide Web* 2.4 (1999), S. 219–229 (zitiert auf S. 12).

[HROE13] W. Hummer, F. Rosenberg, F. Oliveira, T. Eilam. „Testing idempotence for infrastructure as code“. In: *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2013, S. 368–388 (zitiert auf S. 20).

[Hüt12] M. Hüttermann. „Beginning DevOps for Developers“. In: *DevOps for Developers*. Springer, 2012, S. 3–13 (zitiert auf S. 1).

[Inc19a] C. S. Inc. *Chef Dokumentation*. 2019. URL: https://docs.chef.io (zitiert auf S. xi, 5–7, 13, 31–42, 52, 53, 63, 84).

[Inc19b] C. S. Inc. *Infrastructure Code Deserves Tests Too*. 2019. URL: https://kitchen.ci/ (zitiert auf S. 43).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „Winery – Modeling Tool for TOSCA-based Cloud Applications“. In: *11th International Conference on Service-Oriented Computing*. LNCS. Springer, 2013 (zitiert auf S. 3, 7).

[KWB03] A. Kleppe, J. Warmer, W. Bast. *MDA Explained: The Model Driven Architecture : Practice and Promise*. Addison Wesley, 2003 (zitiert auf S. 15).

[Ley09] F. Leymann. „Cloud Computing: The Next Revolution in IT“. In: *Photogrammetric Week '09*. Wichmann Verlag, 2009, S. 3–12 (zitiert auf S. 1).

[LFWW16] F. Leymann, C. Fehling, S. Wagner, J. Wettinger. „Native cloud applications: Why virtual machines, images and containers miss“. In: *Proceedings of the 6th International Conference on Cloud Computing and*. SciTePress. 2016, S. 7–15 (zitiert auf S. 13).

[Mar15] M. Marschall. *Chef infrastructure automation cookbook*. Packt Publishing Ltd, 2015 (zitiert auf S. 5, 43).

[Mat14] K. Matsudaira. „Capturing and structuring data mined from the web“. In: *Communications of the ACM* 57.3 (2014), S. 10–11 (zitiert auf S. 11).

[Mat19a] Y. Matsumoto. *parse.y*. 2019. URL: https://github.com/ruby/ruby/blob/trunk/parse.y (zitiert auf S. 9, 30).

[Mat19b] Y. Matsumoto. *Pseudo BNF Syntax of Ruby*. 2019. URL: https://ruby-doc.org/docs/ruby-doc-bundle/Manual/man-1.4/yacc.html (zitiert auf S. 9, 30).

[OAS13a] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. OASIS, 2013. URL: http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.html (zitiert auf S. 13).

[OAS13b] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. OASIS, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html (zitiert auf S. 13, 16).

[OAS19] OASIS. *TOSCA Simple Profile in YAML Version 1.2*. OASIS, 2019. URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.html (zitiert auf S. 13).

[ON10] C. Olston, M. Najork. „Web Crawling“. In: *Information Retrieval* 4.3 (2010), S. 175–246 (zitiert auf S. 11).

[Ora19] Oracle. *Informationen und Systemanforderungen zur Installation und Verwendung von Oracle Java auf Mac OS X*. 2019. URL: https://www.java.com/de/download/faq/java_mac.xml (zitiert auf S. 78).

[Par13] T. Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013 (zitiert auf S. 8, 9).

[Pre19] T. Preston-Werner. *Semantic Versioning 2.0.0*. 2019. URL: https://semver.org/ (zitiert auf S. 33).

[SK03] S. Sendall, W. Kozaczynski. „Model transformation: The heart and soul of model-driven software development“. In: *IEEE software* 20.5 (2003), S. 42–45 (zitiert auf S. 15, 16).

[sou19] sous-chefs. *Java Cookbook*. 2019. URL: https://github.com/sous-chefs/java (zitiert auf S. xi, 6, 36, 37, 43).

[SS02] V. Shkapenyuk, T. Suel. „Design and implementation of a high-performance distributed web crawler“. In: *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE. 2002, S. 357–368 (zitiert auf S. 12).

[SVG+99] A. da Silva, E. Veloso, P. Golgher, B. Ribeiro-Neto, A. Laender, N. Ziviani. „CoBWeb-a crawler for the Brazilian Web“. In: *String Processing and Information Retrieval Symposium, 1999 and International Workshop on Groupware*. IEEE. 1999, S. 184–191 (zitiert auf S. 12).

[SWG16] R. Shambaugh, A. Weiss, A. Guha. „Rehearsal: a configuration verification tool for puppet“. In: *ACM SIGPLAN Notices*. Bd. 51. 6. ACM. 2016, S. 416–430 (zitiert auf S. 19).

[The01] M. Thelwall. „A web crawler design for data mining“. In: *Journal of Information Science* 27.5 (2001), S. 319–325 (zitiert auf S. 12).

[TV14] M. Taylor, S. Vargo. *Learning Chef: A Guide to Configuration Management and Automation*. O'Reilly Media, Inc., 2014 (zitiert auf S. 5).

[WAL15] J. Wettinger, V. Andrikopoulos, F. Leymann. „Automated Capturing and Systematic Usage of DevOps Knowledge for Cloud Applications“. In: *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*. IEEE Computer Society, 2015, S. 60–65 (zitiert auf S. 17, 20).

[WBKL16] J. Wettinger, U. Breitenbücher, O. Kopp, F. Leymann. „Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel“. In: *Future Generation Computer Systems* 56 (2016), S. 317–332 (zitiert auf S. 13, 17, 18).

[WBL14a] J. Wettinger, U. Breitenbücher, F. Leymann. „DevOpSlang – Bridging the Gap between Development and Operations“. In: *Service-Oriented and Cloud Computing (ESOCC 2014)*. Springer, Berlin, Heidelberg, 2014, S. 108–122 (zitiert auf S. 1).

[WBL14b] J. Wettinger, U. Breitenbücher, F. Leymann. „Standards–based DevOps Automation and Integration Using TOSCA“. In: *Proceedings of the 7th International Conference on Utility and Cloud Computing (UCC 2014)*. IEEE Computer Society, 2014, S. 59–68 (zitiert auf S. 13, 15–19).

Alle URLs wurden zuletzt am 05. 05. 2019 geprüft.

Erklärung

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

Ort, Datum, Unterschrift