Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Automatisierte Aktualisierung bereitgestellter Anwendungen mit TOSCA

Sergej Nisin

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Dr. h.c. Frank Leymann
Betreuer/in:	Lukas Harzenetter, M.Sc.
Beginn am:	3. August 2020
Beendet am:	3. Februar 2021

Kurzfassung

In den vergangenen Jahren sind verschiedene Technologien entstanden zur automatischen Bereitstellung von Cloud Anwendungen. Dazu gehören auch die Modellierungssprache TOSCA und die Laufzeitumgebung OpenTOSCA Container. Da die damit verwalteten Cloud Anwendungen beliebig komplex sein können und aus beliebig vielen Komponenten bestehen können, kann auch das manuelle Aktualisieren von Komponenten kompliziert und zeitaufwendig werden. Jedoch sind Aktualisierungen wichtig, weil ansonsten Sicherheitslücken offen bleiben, die ein Risiko für das System darstellen können.

In der vorliegenden Arbeit wird ein Ansatz vorgestellt, der automatische Aktualisierungen von TOSCA-Anwendungen ermöglicht mit Fokus auf Sicherheitsupdates. Zum Erreichen des Ziels wird ein Algorithmus vorgestellt zur Generierung von ausführbaren Plänen, die die Komponenten von bereitgestellten Anwendungen aktualisieren.

Um die Machbarkeit des Ansatzes zu beweisen, wurde ein Prototyp in OpenTOSCA Container, einer TOSCA-Laufzeitumgebung, implementiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

BPEL Business Process Execution Language. Siehe [OASIS07]. 13

BPMN Business Process Model and Notation. Siehe [OMG11]. 18

CSAR Cloud Service Archive. 19

DBMS Datenbankmanagementsystem. 21

TOSCA Topology and Orchestration Specification for Cloud Applications. Siehe [OASIS13]. 13

XML Extensible Markup Language. 18

1 Einleitung

In diesem Kapitel wird zunächst in Abschnitt 1.1 die Motivation und das Ziel der Arbeit verdeutlicht. Anschließend wird in Abschnitt 1.2 ein laufendes Beispiel vorgestellt, das in dieser Arbeit zur Verdeutlichung von Ansätzen verwendet wird. Zuletzt wird in Abschnitt 1.3 der Aufbau dieser Arbeit erklärt.

1.1 Motivation

Cloud Anwendungen bestehen typischerweise aus mehreren Komponenten. Beispielsweise kann eine Datenbank auf einem anderen Server liegen als ein Java Programm, das auf diese Datenbank zugreift. Hierbei muss aber nicht nur sichergestellt werden, dass auf dem Server des Java-Programms eine Java-Laufzeitumgebung installiert ist, sondern auch, dass das Programm die Datenbank finden und darauf zugreifen kann. Auf diese Weise wird, vor allem bei größeren Anwendungen, das manuelle Bereitstellen oft komplex und fehleranfällig. Um den Prozess des Bereitstellens somit zu vereinfachen, wurden in den vergangenen Jahren mehrere Technologien entwickelt, die das automatische Provisionieren von Cloud-Anwendungen ermöglichen. Die Topology and Orchestration Specification for Cloud Applications (TOSCA) [OASIS13] ist ein Standard zur Beschreibung solcher Anwendungen und das OpenTOSCA Ökosystem [BEK+16] ermöglicht dabei die Modellierung und automatisierte Bereitstellung von Anwendungen in TOSCA. Dafür werden sogenannten Pläne generiert, welche das Deployment der Anwendungen in standardisierten Business Process Execution Language (BPEL) Workflows [OASIS07] beschreibt. Workflows können auch dazu verwendet werden Anwendungen zu managen [BBK+14] [KBL17].

Da die Komponenten einer Anwendung typischerweise Aktualisierungen erhalten, zum Beispiel zur Behebung von Programmierfehlern oder zur Schließung von Schwachstellen, ist es wichtig diese Regelmäßig zu aktualisieren. Aktuell gibt es im OpenTOSCA Ökosystem keine Möglichkeit, die Komponenten einer bereitgestellten Applikation auf eine automatisierte Weise zu aktualisieren. Daher soll in dieser Arbeit ein Konzept entwickelt werden, um in TOSCA modellierte und mithilfe von OpenTOSCA bereitgestellten Anwendungen automatisiert zu aktualisieren.

1.2 Laufendes Beispiel

Abbildung 1.1 zeigt einen Aufbau einer Anwendung, die in dieser Arbeit als ein laufendes Beispiel verwendet wird.

Der Aufbau wird als eine Topologie in Form eines gerichteten Graphen angegeben. Die Knoten des Graphen stellen einzelne Komponenten der Anwendung dar. Die Kanten sind Beziehungen, die die Abhängigkeiten zwischen den Komponenten beschreiben.

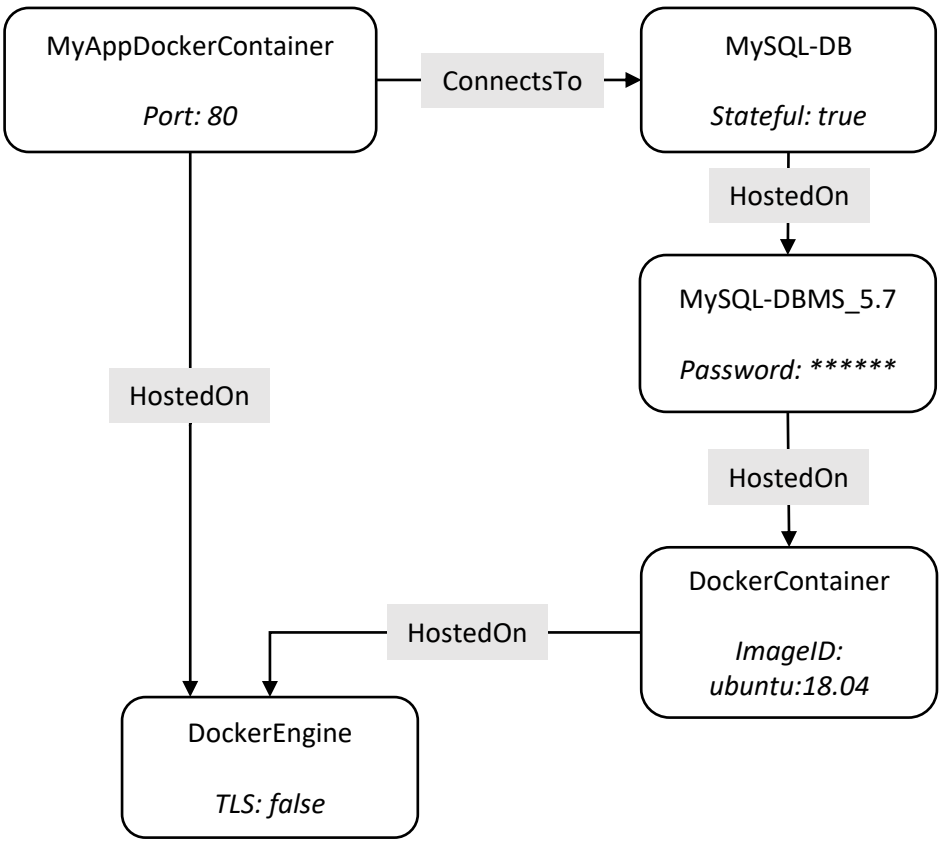


Abbildung 1.1: Aufbau der Anwendung des laufenden Beispiels

Die Beziehung „HostedOn“ bezeichnet, dass eine Komponente zur Ausführung auf eine andere Komponente angewiesen ist. Hier muss sichergestellt sein, dass beim Bereitstellen der abhängenden Komponenten, die andere Komponente bereits bereitgestellt und gestartet wurde.

Die Beziehung „ConnectsTo“, beschreibt, dass zur korrekten Ausführung einer Komponente, eine andere Komponente benötigt wird. Diese können aber unabhängig voneinander bereitgestellt und gestartet werden.

In dieser Topologie ist „MyAppDockerContainer“ eine Applikation, die als ein Docker[1] Container von einer „Docker Engine“ bereitgestellt wird, und über das Netzwerk auf dem Port 80 erreichbar ist.

Nebenbei wird auf der Docker Engine ein weiterer Container mit Ubuntu[2] der Version 18.04 ausgeführt. Auf diesem Container läuft das Datenbankmanagementsystem MySQL[3], das eine Datenbank „MySQL-DB“ verwaltet.

[1] https://www.docker.com/
[2] https://hub.docker.com/_/ubuntu
[3] https://www.mysql.com/

Der Zugriff auf die Docker Engine soll nicht verschlüsselt werden, was über „TLS: false“ dargestellt wird. Der Zugriff auf das Datenbangmanagementsystem soll dagegen mit einem Passwort geschützt werden.

Die Applikation „MyAppDockerContainer“ nutzt eine Datenbank zur Verwaltung von Daten. Dies wird über die Beziehung „ConnectsTo“ dargestellt.

1.3 Struktur

In Kapitel 2 dieser Arbeit werden zunächst grundlegende Informationen gegeben, die für das Verständnis dieser Arbeit notwendig sind. Anschließend werden in Kapitel 3 verwandte Arbeiten beschrieben. Der Ansatz der das automatische Aktualisieren von TOSCA Anwendungen ermöglicht wird in Kapitel 4 vorgestellt. In Kapitel 5 wird der implementierte Prototyp des Ansatzes gezeigt. Zuletzt sind in Kapitel 6 eine Zusammenfassung und ein Ausblick zu finden.

2 Grundlagen

Dieser Kapitel bietet einen Überblick über grundlegende Begriffe und Technologien an, die in dieser Arbeit verwendet werden und für das Verständnis der Arbeit notwendig sind.

Zuerst werden in Abschnitt 2.1 zwei Ansätze zur Beschreibung von Anwendungen vorgestellt. Anschließend wird in Abschnitt 2.2 kurz in die Sprache BPEL eingeführt. Als nächstes wird in Abschnitt 2.3 die Spezifikation der Modellierungssprache TOSCA besprochen. Abschnitt 2.4 beschreibt die Möglichkeit der Generierung von Workflows. Abschnitt 2.5 stellt das OpenTosca Ökosystem vor, das zur Implementierung eines Prototyps verwendet wird. Danach wird in Abschnitt 2.6 die Unterscheidung von zustandslosen und zustandsbehafteten Komponenten erklärt und zuletzt wird in Abschnitt 2.7 das Immutable Infrastructure Paradigma beschrieben.

2.1 Bereitstellungsmodelle

Zur automatischen Bereitstellung von Anwendungen wird zuerst ein *Modell* benötigt, das vorher vom Benutzer erstellt werden muss. Ein Modell ist hierbei eine Beschreibung einer Anwendung, die verwendet werden kann, um diese Anwendung bereitzustellen. Es gibt zwei Ansätze, wie so ein Modell erstellt werden kann:

Beim *deklarativen* Ansatz wird im Modell nur beschrieben, wie die Struktur und der Zustand einer Anwendung nach dem Bereitstellen sein sollen. Eine *deployment engine* bringt die Applikation anschließend in den gewünschten Zustand [EBF+17]. Zur deklarativen Beschreibung von Anwendungen, können die Strukturen von Anwendungen in einem Graphen dargestellt werden [EBLW17] wie wir schon in Abbildung 1.1 gesehen haben. Solche Graphen nennt man auch Topologie [EBLW17]. Die Knoten des Graphen stellen einzelne Komponenten der Anwendung dar. Die Kanten sind Beziehungen, die die Abhängigkeiten zwischen den Komponenten beschreiben. Den Knoten und Beziehungen können auch weiteren Informationen angegeben werden, wie Eigenschaften oder Fähigkeiten.

Beim *imperativen* Ansatz wird stattdessen eine Sequenz an Operationen angegeben, die die Anwendung den Wünschen entsprechend aufsetzt. Diese Sequenz wird anschließend von einer *process engine* ausgeführt [EBF+17]. Eine robuste Technologie zur Erstellung und Ausführung solcher imperativer Modelle ist der Standard BPEL [OASIS07].

Der *imperative* Ansatz hat gegenüber dem *deklarativen* Ansatz zum Vorteil, dass der Entwickler jedes Detail genau definieren kann. Der größte Nachteil ist aber, dass das Verfassen von *imperativen* Plänen mit großem Zeit- und Arbeitsaufwand verbunden ist [BBK+14].

Beide Ansätze sind für diese Arbeit relevant, weil wie in den nachfolgenden Abschnitten genauer erklärt wird, kombiniert TOSCA und damit auch OpenTOSCA beide Ansätze. Zum Erreichen des Ziels der Arbeit wird unser Ansatz sein, deklarative Modelle in imperative Pläne umzuwandeln.

2.2 Business Process Execution Language (BPEL)

Die Business Process Execution Language (BPEL) [OASIS07] ist ein industrieller Standard zur Modellierung von Geschäftsprozessen und stellt eine Möglichkeit zur Beschreibung von imperativen Plänen dar.

Die erste Version der Spezifikation wurde im Jahr 2002 von OASIS veröffentlicht. Die aktuelle Version ist WS-BPEL 2.0 aus dem Jahr 2007. Bei BPEL wird keine graphische Darstellung spezifiziert, sondern Geschäftsprozesse werden ausschließlich mit Extensible Markup Language (XML) beschrieben.

BPEL erlaubt Variablen zu verwenden, Kontrollfluss auszudrücken und Schnittstellen aufzurufen. Durch den Sprachbestandteil $< flow >$ ist es auch möglich die Reihenfolge der Ausführung von Aktivitäten als einen Graph anzugeben.

Um einen BPEL-Prozess auszuführen, wird eine BPEL-Engine benötigt. Es existieren sowohl kommerzielle als auch Open-Source Engines. Beispiel für eine Open-Source Engine ist Apache ODE[1].

In dieser Arbeit wird BPEL zur Beschreibung von imperativen Plänen verwendet und der Prototyp generiert Pläne in dieser Sprache. Der Kern dieser Arbeit lässt sich jedoch auch auf andere Sprachen übertragen. Zum Beispiel auf Business Process Model and Notation (BPMN) [OMG11].

2.3 TOSCA

Topology and Orchestration Specification for Cloud Applications (TOSCA) ist ein Standard zur Beschreibung von Cloud Anwendungen mit XML [OASIS13].

Bei TOSCA werden die *deklarativen* und *imperativen* Ansätze kombiniert. Es kann hierbei sowohl *deklarativ* modelliert werden, aber auch *imperative* ausführbare Pläne eingebunden werden.

In TOSCA wird jede Anwendung als ein *Service Template* modelliert. Die Service Template enthält ein *Topology Template*, das die Topologie, sprich die Struktur der Anwendung, definiert. Die Topology Template besteht hierbei aus *Node Templates* und *Relationship Templates*, die die Knoten bzw. Kanten eines Graphen darstellen. Jedes Node und Relationship Template hat einen Typen sog. *Node Type* bzw. *Relationship Type*, die Operationen und Properties definieren. *Properties* sind Schlüssel-Wert-Paare, die die Eigenschaften und das Verhalten einer Komponente beschreiben. Mögliche Properties werden von Node und Relationship Types definiert und die Werte von den entsprechenden Templates festgelegt. Zum Beispiel kann in einem Node Type eine Property „Port“ definiert werden und ein Node Template dieser Property den Wert „80“ zuweisen. *Operationen* verändern eine Komponente. Ein Beispiel ist eine Operation $start$, die die Komponente startet. Über ein oder mehrere *Interfaces* wird angegeben, welche Operationen in einem Node oder Relationship Type vorhanden sind. Abbildung 2.1 verdeutlicht das genannte. Durch die Trennung von Templates und Types ist es möglich die Types wiederzuverwenden. Beispiele für Relationship Types sind „wird gehosted auf“ oder „verbindet zu“. Beispiele für Node Types sind „Ubuntu 18.04“, „Java

[1] http://ode.apache.org/

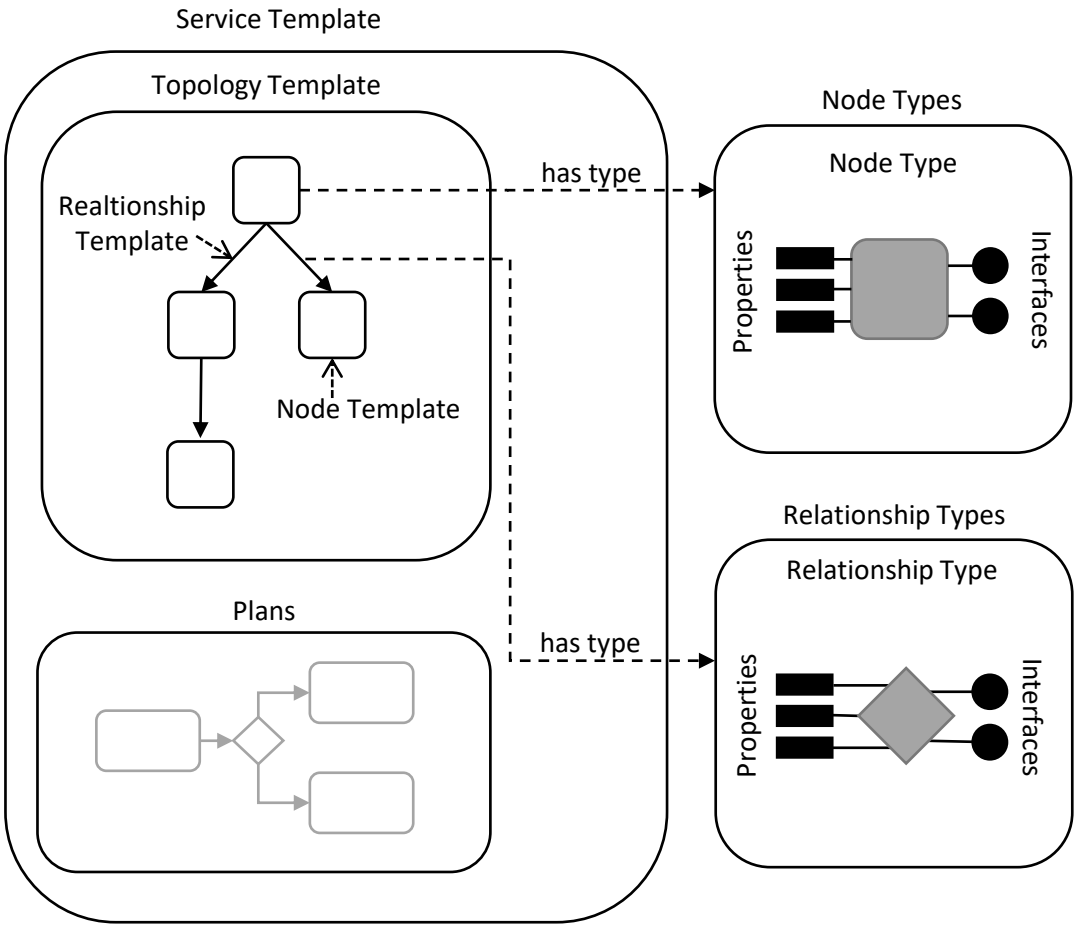


Abbildung 2.1: Aufbau einer TOSCA-Anwendung

8“ oder auch „Docker Container“. Ein *Artifact Template* mit einem Typen *Artifact Type* ist eine Referenz auf eine Datei. Beispielsweise ist eine ausführbare Datei „start.sh“ ein Artifact Template vom Typen „ScriptArtifact“ oder eine Datei „dbTopology.sql“ ein Artifact Template vom Typen „SQLArtifact“. Diese Artifact Templates können in Node Templates eingebunden werden, wenn diese vom Node Template benötigt werden. Node und Relationship Types können *Interfaces* enthalten, die die vorhandenen Operationen definieren. Diese Operationen werden implementiert durch *Node* bzw. *Relationship Type Implementation* indem es den Operationen ausführbare Artifact Templates zuweist.

Zusätzlich zur Topology Template kann ein Service Template auch *Plans* enthalten, die Abläufe zum Verwalten der Anwendung beschreiben. Beispielsweise kann ein BPEL-Plan „UpdateApp“ angehängt werden, der die Anwendung aktualisiert.

Zur Übertragung und Weitergabe von TOSCA-Anwendungen, spezifiziert TOSCA den Format Cloud Service Archive (CSAR), um Topology Templates, Types, Artifacts, Pläne und alle benötigten Dateien in ein in sich geschlossenes Paket zu verpacken.

2.4 Workflow Generierung

In TOSCA können Anwendungen deklarativ beschrieben. Diese können allerdings in imperative *Workflows* umgewandelt werden [BBK+14], was in Abschnitt 3.2 nochmal genauer beschrieben wird. Algorithmen, die beschreiben wie imperative Workflows von deklarativen Modellen abgeleitet werden können, bezeichnen wir in dieser Arbeit als *PlanBuilder*.

In [HBKL20] und [KBL17] wurde dieser Konzept erweitert um *Workflow*-Pläne zu generieren, die die Anwendung verwalten können. In dieser Arbeit werden diese Konzepte erweitert um die Generierung von Update-Plänen, die die Komponenten einer Anwendung aktualisieren können.

2.5 OpenTosca Ökosystem

Das OpenTosca Ökosystem [BEK+16] ist eine open-source Implementierung zur Modellierung und Deployment von Anwendungen und besteht aus mehreren Teilen. Die für uns relevanten Komponenten sind:

Winery[2] ist ein web-basiertes Tool zum graphischen Entwickeln von TOSCA Anwendungen [KBBL13]. Hier können Topology Templates über drag-and-drop in einem Graphen erstellt und verwaltet werden. Auch Node Types, Relationship Types, Artifact Types lassen sich erstellen und verändern. Als Datenbank verwendet Winery ein Git-Repository, in dem die TOSCA Types und Templates in XML-Dateien gespeichert werden.

Der **OpenTosca Container**[3] [BEK+16] ist eine TOSCA *Laufzeitumgebung*, die die mit TOSCA definierten Anwendungen automatisiert aufsetzten und ausführen kann. Hier lassen sich TOSCA-Anwendungen aus dem Repository laden, wobei mithilfe von PlanBuildern BPEL-Pläne erstellt und in die TOSCA Anwendung eingebunden werden. Es werden hierbei unter anderem Pläne zum Provisionieren und Terminieren erstellt. Diese Pläne lassen sich mit OpenTOSCA Container ausführen um Instanzen der Anwendungen zu erstellen und zu verwalten. Benutzt werden kann OpenTosca Container über eine REST-API oder eine Web-Oberfläche[4].

Zur Beweis der Machbarkeit des in dieser Arbeit vorgestellten Ansatzes, wird im Rahmen dieser Arbeit ein Prototyp als Teil von **OpenTosca Container** implementiert.

2.6 Komponenten Arten

In der Cloud können Komponenten in zustandslose (engl. stateless) und zustandsbehaftete (engl. stateful) Komponenten kategorisiert werden [FLR+14]. Zustandslose Komponenten speichern keine Daten zur weiteren Verarbeitung, weswegen diese einfach horizontal skaliert werden können [FLR+14]. Beispielsweise ist *MyAppDockerContainer* in Abbildung 1.1 auf Seite 14 zustandslos, weil es keine Daten bei sich speichert, sondern diese in die Datenbank leitet.

[2] https://eclipse.github.io/winery
[3] https://opentosca.github.io/container/
[4] https://github.com/OpenTOSCA/ui

Andererseits haben zustandsbehaftete Komponenten einen internen Zustand. Zum Beispiel ist die Datenbank *MySQL-DB* in Abbildung 1.1 zustandsbehaftet, weil es persistente Daten verwaltet, die von *MyAppDockerContainer* gesendet oder angefragt werden.

Ziel dieser Arbeit ist es Anwendungen zu aktualisieren, ohne Veränderung des Zustandes. Ferner kann bei unerwarteter Änderung des Zustandes, die Anwendung in einen inkonsistenten Zustand kommen. Zum Beispiel, wenn Einträge einer Datenbank auf Daten einer anderen Datenbank verweisen, werden die Verweise ungültig, wenn nur die verwiesene Datenbank zurückgesetzt wird. Deshalb müssen wir beim Aktualisieren von Komponenten sichergehen, dass der Zustand beibehalten wird.

2.7 Immutable Infrastructure

Es existieren die Paradigmen „Mutable Infrastructure“ und „Immutable Infrastructure“ [Mor16]

Bei Mutable Infrastructure darf eine Infrastruktur nach dem Bereitstellen verändert werden. Beispielsweise kann man auf einem Debian-System mit „apt-get upgrade“ Pakete auf den neusten Stand bringen.

Dies ist bei „Immutable Infrastructure“ so nicht erlaubt. Wenn ein Server verändert werden soll, dann muss der alte Server gestoppt und stattdessen ein neues System auf die gewünschte Weise aufgesetzt werden.

Kubernetes[5] ist ein Orchestrierungs-System das die beschriebene „Immutable Infrastructure“ benutzt. Die einzelnen Komponenten einer Kubernetes-Applikation sind Docker-Container, die in sich geschlossene Systeme darstellen. Möchte man nun eine Komponente aktualisieren oder die Konfiguration verändern, dann wird Kubernetes einen neuen Container mit der gewünschten Konfiguration starten und den alten Container stoppen.

Der Vorteil von „Immutable Infrastructure“ ist, dass es weniger komplex als mutable ist. Wenn man beispielsweise einen Server mutable aktualisieren möchte, kann es passieren, dass dabei ein Fehler auftritt und der Server in einen Zustand kommt, der weder der Anfangszustand noch der gewünschte Zustand nach der Aktualisierung ist. Bei „Immutable Infrastructure“ kann dies nicht passieren. Falls bei beim Bereitstellen ein Fehler auftritt, wird Server gestoppt und es kann von gleichen Zustand aus erneut versucht werden.

Eines der Nachteile von „Immutable Infrastructure“ ist allerdings, dass bei Veränderung die Daten und der Zustand der Anwendung nicht beibehalten werden. Beispielsweise gehen beim Neuaufsetzten eines Servers mit einem Datenbankmanagementsystem (DBMS) die Dateien des Dateisystems und somit auch die Daten der Datenbank verloren. Bei Kubernetes wurde dies gelöst durch Persistent Volumes, wobei hier ein Verzeichnis bereitgestellt wird, in dem Zustandsinformationen gespeichert werden, die beim Update beibehalten werden.

[5] https://kubernetes.io

In unserem Beispiel in Abbildung 1.1 ist DockerContainer eine „immutable“ Komponente. Mit dem Ansatz wie in Kubernetes, wird in DockerContainer ein Verzeichnis bereitgestellt, in dem die Zustandsdaten von MySQL-DB gespeichert werden. Bei einem Update von DockerContainer, wird dieses Verzeichnis beibehalten, sodass bei Neuinstallation der darauf gehosteten Komponenten, der Zustand erhalten bleibt.

3 Verwandte Arbeiten

Die nachfolgenden Abschnitte beschreiben Arbeiten, die mit dem in dieser Arbeit vorgestellten Ansatz verwandt sind. Zuerst wird in Abschnitt 3.1 eine Arbeit vorgestellt, die das Problem von Beibehalten von Zuständen angeht. Danach wird in Abschnitt 3.2 analysiert, wie vorherige Arbeiten Workflow-Pläne generiert haben.

3.1 Freeze und Defrost

Ein Ansatz um Daten bei einer Aktualisierung beizubehalten, ist die Verwendung des *Freeze and Defrost* Ansatzes von Harzenetter et al. [HBKL20].

In [HBKL20] wurde ein Ansatz vorgestellt, eine zustandsbehaftete Anwendung so zu stoppen, dass die Anwendung in den gleichen Zustand wieder neugestartet werden kann. Der Ausdruck *freeze an application* (zu deutsch Anwendung einfrieren) wurde hier definiert als das Stoppen einer Anwendung und das Speichern der Zustandes. Der Ausdruck *defrost an application* (zu deutsch Anwendung auftauen) wurde definiert als das Starten einer Anwendung in den vorher gespeicherten Zustand.

Aufbauend darauf ist der Ansatz, nur Teile einer Anwendung einzufrieden oder aufzutauen. In unserem Beispiel in Abbildung 1.1 ist DockerContainer eine „immutable“ Komponente, weswegen bei Aktualisierung von DockerContainer die Zustandsdaten verloren gehen. Falls aber zuvor der Teil der Anwendung mit MySQL-DB und MySQL_DBMS_5.7 eingefroren und nach Aktualisierung von DockerContainer dieser Teil wieder aufgetaut wird, bleiben die Zustandsdaten erhalten.

3.2 Generierung von Management-Workflows

Breitenbücher et al. [BBK+14] haben einen Ansatz beschreiben, wie aus deklarativen TOSCA-Topologien imperative Workflows zur Provisionierung einer Anwendung automatisch generiert werden können. In [KBL17] wurde dieser Ansatz erweitert um die Generierung von *Scale-Out Plänen*, die die Konfiguration von IoT automatisieren. Auch Harzenetter et al. [HBKL20] erweiterten diesen Ansatz um die Generierung von freeze und defrost-Plänen.

Alle diese Ansätze funktionieren nach der gleichen Methode, welches von Abbildung 3.1 illustriert wird. Zuerst wird das Topology Template in einen *Order Graph* umgewandelt. Dieser ist ein gerichteter Graph, der die Reihenfolge von abstrakten Aktivitäten angibt. Anschließend wird aus dem Order Graph ein *Plan Skeleton* generiert. Im Gegensatz zum Order Graph ist der Plan abhängig von der verwendeten Sprache (z.B. BPEL). Die Reihenfolge wird beibehalten. Der Skeleton enthält jedoch statt abstrakten Aktivitäten, Platzhalter der Ziel-Sprache. Zum Beispiel können in BPEL

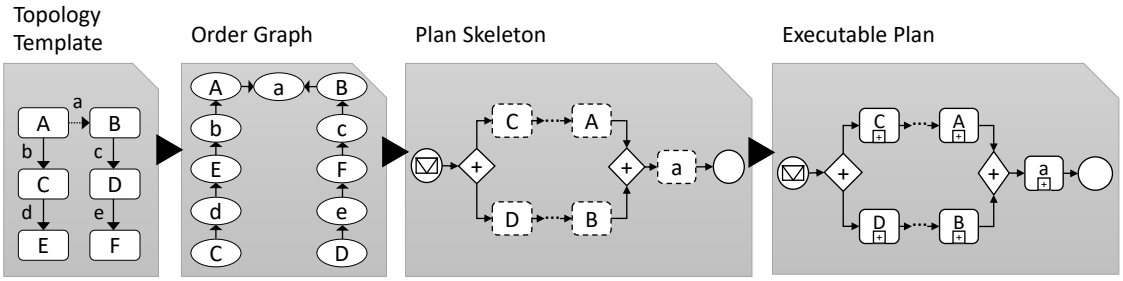


Abbildung 3.1: Methode mit drei Schritten zur Generierung von ausführbaren Plänen

diese Platzhalter $< empty >$ - Aktivitäten in einem $< flow >$ sein, deren Reihenfolge durch $< links >$ definiert wird. Der letzte Schritt ist das Ersetzen der Platzhalter durch sprachenabhängige und ausführbare Logik. Sogenannte *Logic Provider* füllen die Platzhalter abhängig von Node Template, Ziel-Sprache und Aktivität.

4 Ansatz

In diesem Kapitel wird ein Ansatz zum automatischen Aktualisieren von Anwendungen beschrieben. Zuerst wird in Abschnitt 4.1 eine Übersicht gegeben, wie der Ansatz funktionieren wird. Anschließend werden in Abschnitt 4.2 Operationen für Node Types definiert, die für den Ansatz erfordert werden. Als nächstes werden in Abschnitt 4.3 mögliche Szenarien vorgestellt und beschrieben, wie in diesen Fällen vorgegangen werden soll, um damit Regeln zu formulieren, die eingehalten werden müssen. Schließlich wird in Abschnitt 4.4 eine Notation für TOSCA definiert, um damit in Abschnitt 4.5 die Details des Ansatzes zu beschreiben.

4.1 Übersicht

Da die Anwendungen mit Winery *deklarativ* modelliert werden, muss OpenTOSCA Container zum Bereitstellen und Terminieren einen Ablauf an Operationen finden, die das gewünschte Ergebnis bewerkstelligen. Bei OpenTOSCA Container werden dazu aus der Topologie BPEL Pläne generiert.

Auch zum Aktualisieren von den Anwendungen werden wir nun einen PlanBuilder erstellen, der einen Plan zum Aktualisieren der Anwendungen generieren kann. Zu diesem Zweck beschreiben wir nun wie so ein Plan generiert werden kann.

Wir konzentrieren uns in dieser Arbeit auf Sicherheitsupdates und werden deshalb im Folgenden fordern, dass alle Aktualisierungen abwärtskompatibel sind. Beispielsweise darf in Abbildung 1.1 „Ubuntu 18.04“ nicht auf „Ubuntu 20.04“ aktualisiert werden, weil „MySQL-DBMS_5.7“ nicht mit dieser Version kompatibel ist.

Ferner werden wir auch davon ausgehen, dass bei der Installation einer Komponente, die gewünschte Version installiert wird. Somit muss nach einer Neuinstallation, die Komponente nicht upgedatet werden.

Zur Beibehaltung des Zustandes bei stateful Komponenten wurden in den vorherigen Kapiteln zwei Ansätze gezeigt. Der Freeze und Defrost Ansatz in Abschnitt 3.1 und die Verwendung von Persistent Volumes wie bei Kubernetes in Abschnitt 2.7. Ein Vorteil von Persistent Volumes ist, dass wenn die Komponente in einem Container ist, dass die Zustandsdaten direkt auf dem Hostsystem gespeichert werden können ohne Netzwerkübertragungen. Bei TOSCA kann aber nicht davon ausgegangen werden, dass Container verwendet werden, da TOSCA auch Bereitstellung von Servern ohne Container erlaubt. Ein Vorteil von Freeze und Defrost ist, dass Zustandsdaten nicht ständig in Dateien gespeichert sein müssen, sonden erst bei Ausführung einer Operation erstellt werden können. Zum Beispiel können hierbei Daten aus dem Arbeitsspeicher gespeichert werden. In unserem Ansatz entscheiden wir uns für die Verwendung des Freeze und Defrost Ansatzes, weil es eine große Flexibilität bietet und die Kompatibilität mit TOSCA bereits von Harzenetter et al. [HBKL20] bewiesen wurde.

4.2 Operationen

In TOSCA können *Node Types* auch *Implementation Artifacts (IA)* besitzen, die Implementierungen von Operationen enthalten können.

Wir definieren folgende folgende Operationen ein, die wir für unsere Lösung verwenden.

- Die Operation **install** provisioniert eine Instanz der Komponente.
- Die Operation **start** startet die Komponente.
- Die Operation **stop** stoppt die Komponente.
- Die Operation **uninstall** entfernt die gestoppte Instanz der Komponente. Zustandsdaten werden gelöscht.
- Die Operation **freeze** speichert den aktuellen Zustand der Komponente in einem persistenten Speicher.
- Die Operation **defrost** stellt den Zustand der Komponente aus den im persistenten Speicher gespeicherten Daten wieder her.
- Die Operation **update** aktualisiert eine gestoppte Komponente auf die aktuellste abwärtskompatible Version. Falls die Komponente als „immutable“ annotiert ist, werden die Zustandsdaten dieser und der darauf gehosteten Komponenten gelöscht. Beispielsweise kann der Komponente „DockerContainer“ in Abbildung 1.1 eine Operation „update“ hinzugefügt werden, die bei Ausführung den Docker Container gestoppt, das neuste Image aus dem Repository holt und den Container mit dem neuen Image wieder startet. Wir können hier ausgehen, dass die neue Version abwärtskompatibel ist, weil angegeben ist, dass der Versionszweig „18.04“ verwendet werden muss und die Update-Politik[1] von Ubuntu besagt, dass nur sichere Änderungen zugelassen werden, die nicht das Verhalten des Systems verändern.

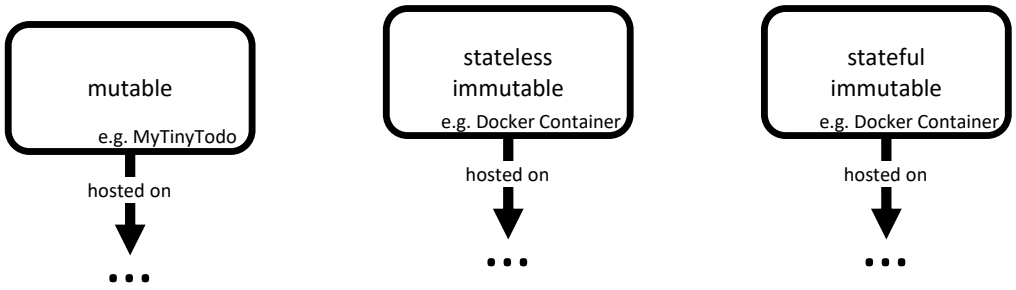
4.3 Szenarien

Nun betrachten wir mögliche Szenarien und beschreiben, wie in diesen Fällen vorgegangen werden muss. Die Abbildung 4.1 zeigt die möglichen Fälle, wie ein Blattknoten sein kann, der aktualisiert wird. Blattknoten beschreibt hier eine Komponente, auf der keine anderen Komponenten gehostet werden. Beispiele für solche Komponenten sind „MyAppDockerContainer“ und „MySQL-DB“ des laufenden Beispiels in Abbildung 1.1 auf Seite 14.

In Abbildung 4.1a ist ein mutable Node Template dargestellt, das ein Blattknoten ist, und somit keine anderen Komponenten davon abhängen. In diesem Fall kann der Knoten durch die folgenden Schritte aktualisiert werden:

1. Stoppe die Komponente, weil Komponenten üblicherweise nicht im laufenden Zustand aktualisiert werden können. Zum Beispiel muss zum Aktualisieren einer Java-Anwendung, diese vorher gestoppt werden.

[1] https://wiki.ubuntu.com/StableReleaseUpdates



(a) Fall, dass ein mutable Blattknoten upgedatet werden soll

(b) Fall, dass ein stateless und immutable Blattknoten upgedatet werden soll

(c) Fall, dass ein stateful und immutable Blattknoten upgedatet werden soll

Abbildung 4.1: Mögliche Fälle, wenn ein Blattknoten aktualisiert werden soll.

2. Führe die Operation `update` der Komponente aus. Für diesen Zweck muss vorausgesetzt werden, dass bei allen zu aktualisierenden Komponenten, diese Operation implementiert wurde.
3. Starte die Komponente, weil diese in Schritt 1 gestoppt wurde.

Da die Komponente mutable upgedatet werden kann, werden Daten, falls vorhanden, auch beibehalten, weil hier die Komponente verändert wird, und nicht neu erstellt.

In Abbildung 4.1b ist ein immutable und stateless Node Template, das ein Blattknoten ist. Hier kann mit den gleichen Schritten wie im vorherigen Fall vorgegangen werden:

1. Stoppe die Komponente.
2. Führe die Operation `update` der Komponente aus.
3. Starte die Komponente.

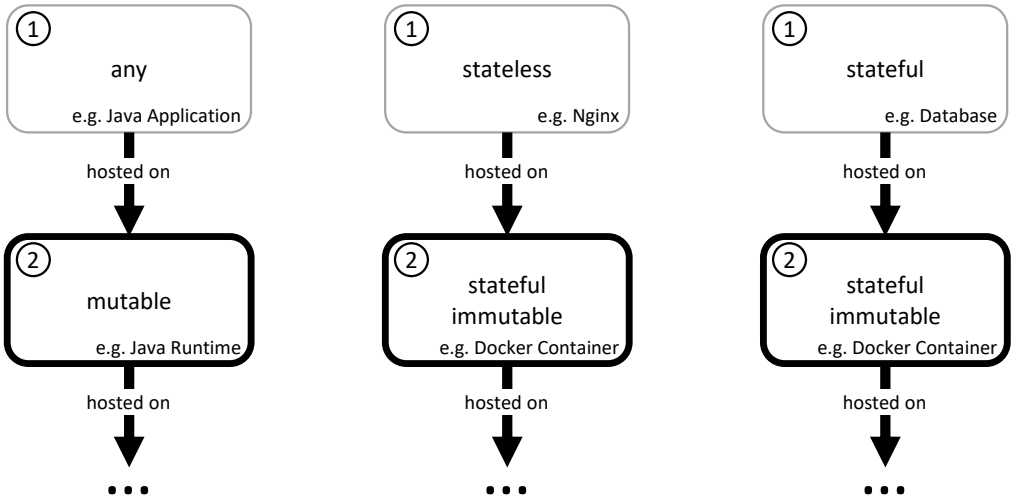
Da dieser Knoten stateless ist, besitzt die Komponente keine Zustandsdaten, die beibehalten werden müssen.

In Abbildung 4.1c ist ein immutable und stateful Node Template, das ein Blattknoten ist. Hier werden nun weitere Schritte benötigt, weil immutable Komponenten beim Aktualisieren neu aufgesetzt werden müssen, wobei Zustandsdaten verloren gehen können:

1. Stoppe die Komponente.
2. Führe die Operation `freeze` der Komponente aus.
3. Führe die Operation `update` der Komponente aus.
4. Führe die Operation `defrost` der Komponente aus.
5. Starte die Komponente.

Durch den Aufruf der Operationen `freeze` und `defrost` werden Zustandsdaten gespeichern bzw. wiederhergestellt. Dadurch wird sichergestellt, dass der Zustand der Komponente nach der Aktualisierung beibehalten wird.

Als nächstes betrachten wir Fälle in Abbildung 4.2, in denen die zu aktualisierende Komponente kein Blattknoten mehr ist. Die dick umrandete Komponente ist jeweils die zu aktualisierende Komponente.



(a) Mutable Komponente hostet eine beliebige andere Komponente.

(b) Stateful, immutable Komponente hostet eine stateless Komponente.

(c) Stateful, immutable Komponente hostet eine stateful Komponente.

Abbildung 4.2: Mehrere Fälle, in denen eine Komponente upgedatet werden soll, der eine andere Komponente hostet

In Abbildung 4.2a ist ein mutable Node Template (2), auf dem eine beliebige andere Komponente (1) gehostet wird. In diesem Fall müssen folgende Schritte vorgenommen werden:

1. Stoppe (1), weil Komponente (2) während der Aktualisierung kurzzeitig nicht zur Verfügung steht. Beispielsweise kann eine Java Applikation nicht während einer Aktualisierung der Java Laufzeitumgebung ausgeführt werden.
2. Stoppe die Komponente (2).
3. Führe Operation `update` der Komponente (2) aus. Das Vorhandensein der Operation wird vorausgesetzt.
4. Starte die Komponente (2).
5. Starte (1), weil diese in Schritt 1 gestoppt wurde.

Da die Komponente (2) mutable upgedatet wird, werden Daten, falls vorhanden, auch beibehalten, weil hier die Komponente verändert und nicht neu erstellt wird.

In Abbildung 4.2b ist ein immutable und stateful Node Template (2), auf dem eine stateless Komponente (1) gehostet wird. In diesem Fall müssen folgende Schritte vorgenommen werden:

1. Stoppe (1), weil Komponente (2) während der Aktualisierung kurzzeitig nicht zur Verfügung steht.
2. Deinstalliere die Komponente (1).
3. Stoppe die Komponente (2).
4. Führe Operation `freeze` der Komponente (2) aus.
5. Führe Operation `update` der Komponente (2) aus.
6. Führe Operation `defrost` der Komponente (2) aus.

7. Starte die Komponente (2).
8. Installiere die Komponente (1).
9. Starte (1), weil diese in Schritt 1 gestoppt wurde.

Gegenüber dem vorherigen Fall werden hier zusätzlich die Operationen `freeze` und `defrost` ausgeführt. Dies wird benötigt, weil (2) immutable ist und somit bei Aktualisierung Zustandsdaten nicht beibehalten werden. Durch die zusätzlichen Schritte wird sichergestellt, dass der Zustand wiederhergestellt wird. (1) ist stateless und besitzt somit keinen Zustand, der verloren gehen kann. Da allerdings (2) immutable ist und bei Aktualisierung neu erstellt werden muss, geht die Installation von (1) verloren. Deshalb deinstallieren wir die Komponente vorher und installieren diese anschließend neu.

In Abbildung 4.2c ist ein immutable und stateful Node Template (2), auf dem eine stateless Komponente (1) gehostet wird. In diesem Fall müssen folgende Schritte vorgenommen werden:

1. Stoppe (1).
2. Führe Operation `freeze` der Komponente (1) aus.
3. Deinstalliere die Komponente (1).
4. Stoppe die Komponente (2).
5. Führe Operation `freeze` der Komponente (2) aus.
6. Führe Operation `update` der Komponente (2) aus.
7. Führe Operation `defrost` der Komponente (2) aus.
8. Starte die Komponente (2).
9. Installiere die Komponente (1).
10. Führe Operation `defrost` der Komponente (1) aus.
11. Starte (1), weil diese in Schritt 1 gestoppt wurde.

Da im Gegensatz zum vorherigen Fall die Komponente (1) stateful ist, besitzt diese Zustandsdaten. Durch die Operationen `freeze` und `defrost` werden hier nun auch bei (1) die Zustandsdaten wiederhergestellt.

Im Allgemeinen können wir folgende Regeln ableiten:

1. Wenn ein Knoten aktualisiert werden soll, müssen zuvor alle darauf gehosteten Knoten zuerst gestoppt werden, weil der zu aktualisierende Knoten während der Aktualisierung nicht verfügbar ist. Beispielsweise kann eine Java Applikation nicht während einer Aktualisierung der Java Laufzeitumgebung ausgeführt werden
2. Nach der Aktualisierung eines Knotens müssen die darauf gehosteten Knoten wieder gestartet werden, damit nach der Aktualisierung alle Komponenten wieder verfügbar sind.
3. Falls ein Knoten Immutable ist, müssen von allen darauf gehosteten, Knoten deinstalliert werden.
4. und im Anschluss neuinstalliert werden.
5. Falls ein Knoten Immutable ist, müssen die Zustandsdaten von allen darauf gehosteten, zustandsbehafteten Knoten, beim Stoppen gesichert und
6. im Anschluss wiederhergestellt werden, um den Zustand der Komponente nach der Aktualisierung beizubehalten.

Algorithmus 4.1 beschreibt einen Ablauf, der die angegebenen Regeln erfüllt. Der Algorithmus besteht aus aus zwei Schleifen, die nacheinander ausgeführt werden. Die Schreibung $\text{nodes}_{\text{topDown}}$ bedeutet hier, dass in einer Reihenfolge iteriert wird, dass bevor in einem Schleifendurchlauf ein

Knoten aus *nodes* drankommt, vorher zuerst alle darauf gehosteten Knoten drangekommen sein müssen. Bei $\text{nodes}_{\text{bottomUp}}$ ist die Reihenfolge genau umgekehrt. `node.ancestors` ist die Menge aller Knoten, die von node über hostedOn-Kanten erreichbar sind, also auf denen node gehosted wird. In der ersten Schleife werden alle Knoten, die von einem Knoten abhänging sind, der aktualisiert werden soll, gestoppt. Damit wird Regel 1 erfüllt. Falls ein stateful Knoten auf einem immutable Knoten gehostet wird, wird bei bei diesem die Operation Freeze aufgerufen. Damit wird die Regel 5 sichergestellt. Falls ein Knoten auf einem immutable Knoten gehostet wird, wird bei bei diesem die Operation Uninstall aufgerufen. Damit wird die Regel 3 sichergestellt. Diese Schleife nennen wir „Stop-Phase“. In der zweiten Schleife werden alle zu aktualisierenden Knoten aktualisiert und die in der Stop-Phase gestoppten Knoten wieder gestartet, was die Regel 2 erfüllt. Falls ein Knoten auf einem immutable Knoten gehostet wird, wird bei bei diesem die Operation Install aufgerufen. Damit wird die Regel 4 sichergestellt. Falls ein stateful Knoten auf einem immutable Knoten gehostet wird, wird bei bei diesem die Operation Defrost aufgerufen. Damit wird die Regel 6 sichergestellt. Diese Schleife nennen wir „Start-Phase“.

4.4 Notation

Zum besseren Verständnis benutzen wir im Folgenden eine Notation für TOSCA, basierend auf dem in [HBKL20] beschriebenen Metamodells:

Sei T die Menge aller Topology Templates, dann sind für für eine Topology Template $t \in T$ folgende Elemente definiert:

- C_t ist die Menge aller Node Templates.
- $R_t \subseteq C_t \times C_t$ ist die Menge aller Relationship Templates.
- CT_t ist die Menge aller Node Types in t.
- RT_t ist die Menge aller Relationship Types in t.
- $type_t$ weist jedem Node Template und Relationship Template einen Node- bzw. Relationship-Type zu.

  $$type_t : C_t \to CT_t$$
  $$type_t : R_t \to RT_t$$

- Da Typen Supertypen haben können, gibt $allTypes_t$ alle Typen eines Node bzw. Relationship Templates zurück.

  $$allTypes_t : C_t \to \wp(CT_t)$$
  $$allTypes_t : R_t \to \wp(RT_t)$$

- $A_t \subseteq \Sigma^+ \times \Sigma^+$ ist die Menge aller Properties als (Schlüssel, Wert)-Paare.
- $attributes_t : C_t \to \wp(A_t)$ weist jedem Node Template eine Menge von Properties zu.
- $O_t \subseteq \Sigma^+$ ist die Menge aller auf Node-Templates ausführbare Operationen.
- $operations_t : C_t \to \wp(O_t)$ gibt jedem Node Template eine Menge von verfügbaren Operationen.

Algorithmus 4.1 Update Algorithmus

```
for all node ∈ nodes_topDown do
    if |{p|p ∈ node.ancestors, p.toUpdate = true}| > 0) then
        STOP(node)
        if |{p|p ∈ node.ancestors, p.immutable = true}| > 0 then
            if node.stateful then
                FREEZE(node)
            end if
            UNINSTALL(node)
        end if
    else if node.toUpdate then
        STOP(node)
        if node.immutable ∧ node.stateful then
            FREEZE(node)
        end if
    end if
end for
for all node ∈ nodes_bottomUp do
    if |{p|p ∈ node.ancestors, p.toUpdate = true}| > 0) then
        if |{p|p ∈ node.ancestors, p.immutable = true}| > 0 then
            INSTALL(node)
            if node.stateful then
                DEFROST(node)
            end if
        end if
        START(node)
    else if node.toUpdate then
        UPDATE(node)
        if node.immutable ∧ node.stateful then
            DEFROST(node)
        end if
        START(node)
    end if
end for
```

4.5 Details des Ansatzes

4.5.1 Schritt 1: Modellierung

Der erste Schritt damit ein Update Plan generiert werden kann, ist es eine Anwendung mit TOSCA zu Modellieren. Zusätzlich zu den zum Bereitstellen und Terminieren benötigten Operationen wie install, start, stop und uninstall, ist bei den Komponenten, die aktualisiert werden sollen, eine update operation notwendig. Wenn eine Komponente aktualisiert werden soll, die immutable ist, müssen

bei allen darauf gehosteten stateful Komponenten, `freeze` und `defrost` Operationen vorhanden sein, weil bei Aktualisierung von immutable keine Zustandsdaten beibehalten werden, und diese deshalb gesichert und wieder geladen werden müssen.

Diese Modellierung kann mit Winery erfolgen, das eine graphische Benutzeroberfläche bereitstellt.

4.5.2 Schritt 2: Generierung eines Update-Order-Graphen

Der nächste Schritt ist die Generierung eines Order-Graphen. Durch die Ausführung von Algorithmus 4.2 wird ein Update-Order-Graph generiert. Der Update-Order-Graph ein gerichteter Graph, der angibt in welcher Reihenfolge *Aktivitäten*, die die Knoten des Graphen darstellen, ausgeführt werden. *Aktivitäten* sind hier zunächst nur eine abstrakte Angabe. Die genauen Operationsaufrufe werden erst im nächten Schritt vervollständigt. Die Aktivitäten, die zum Aktualisieren benötigt werden sind: $ActivityTypes$:= {NONE, TERMINATE, PROVISION, FREEZE-AND-TERMINATE, PROVISION-AND-DEFROST, UPDATE, FREEZE-UPDATE-DEFROST}.

Der Update-Order-Graph ist im Algorithmus angegeben als $G_{uog} = (Activities, Order)$. Elemente der Menge *Activities* stellen die Knoten der erstellten Graphen dar und werden angegeben als ein Tripel (te, at, ph) eines Topologie-Elements $te \in C_t \cup R_t$, eines Aktivitäten-Typen $at \in ActivityTypes$ und der Phase $ph \in \{STOP, START\}$. *Order* ist eine Menge an Tupeln von zwei Aktivitäten und gibt an, in welcher Reihenfolge die Aktivitäten ausgeführt werden sollen. Die Funktion `hasUpdatableAncestor` überprüft ob ein Knoten von einem Knoten abhängt, der upgedatet werden kann und wird definiert in Algorithmus 4.3.

In Abschnitt 4.3 haben wir in Algorithmus 4.1 die Schleifen aufgeteilt in Stop-Phase und Start-Phase. Auch dieser Algorithmus generiert einen Plan mit diesen zwei Phasen. Mit $Activities_{stop}$ wird eine Menge erstellt mit den Aktivitäten, die der Stop-Phase zugehören und $Activities_{start}$ mit den Aktivitäten der Start-Phase. Zusätzlich wird jeder Knoten mit der entsprechenden Phase ph annotiert.

Für jeden Node Template wird abhängig von den Properties, Operationen und den Node Templates von denen es abhängt, zwei Aktivitäten in den Graphen G_{uog} hinzugefügt. Jeweils einen in die Stop-Phase und die Start-Phase. Auch falls zu einem Knoten nichts gemacht werden soll, werden NONE-Aktivitäten hinzugefügt, weil dies die Verbindung durch Kanten in der nächten Schleife erleichtert. Als nächstes werden die Kanten hinzugefügt. Auch hier werden immer zwei Kanten, jeweils eins in jeder Phase, erstellt. ConnectsTo-Relationen werden temporär auch terminiert, da bei einer Aktualisierung nicht garantiert werden kann, dass beide Knoten der Relation im gestarteten Zustand bleiben.

Der letzte Abschnitt des Algorithmus fügt Kanten hinzu, die die Stop- und Start-Phase so miteinander verbinden, dass nachdem die Stop-Phase fertig ausgeführt wurde, die Start-Phase startet.

Abbildung 4.3 zeigt eine Darstellung eines Graphen, der mit Algorithmus 4.2 aus unserem laufenden Beispiel aus Abbildung 1.1 generiert wurde. Jeder Kasten stellt einen Knoten dar. **Fett** geschrieben ist der Aktivitäten-Typ $at \in ActivityTypes$, *Kursiv* geschrieben ist das Node oder Relationship Template $te \in C_t \cup R_t$, und in Klammern die Phase $ph \in \{STOP, START\}$. Die fett umrandeten Knoten gehören zur zu aktualisierenden Komponente.

Algorithmus 4.2 generateUpdatePlan($t \in T$)

$G_{uog} = (Activities_{stop} \cup Activities_{start}, Order_{uog})$
for all $c_i \in C_t$ **do**
  if HASUPDATABLEANCESTOR(c_i, t) **then**
    if (stateful, true) $\in$ $attributes(c_i)$ **then**
      $Activities_{stop} := Activities_{stop} \cup (c_i, \text{FREEZE-AND-TERMINATE}, \text{STOP})$
      $Activities_{start} := Activities_{start} \cup (c_i, \text{PROVISION-AND-DEFROST}, \text{START})$
    else
      $Activities_{stop} := Activities_{stop} \cup (c_i, \text{TERMINATE}, \text{STOP})$
      $Activities_{start} := Activities_{start} \cup (c_i, \text{PROVISION}, \text{START})$
    end if
  else if update $\in$ $operations(c_i)$ **then**
    $Activities_{stop} := Activities_{stop} \cup (c_i, \text{NONE}, \text{STOP})$
    if (stateful, true) $\in$ $attributes(c_i) \wedge$ (immutable, true) $\in$ $attributes(c_i)$ **then**
      $Activities_{start} := Activities_{start} \cup (c_i, \text{FREEZE-UPDATE-DEFROST}, \text{START})$
    else
      $Activities_{start} := Activities_{start} \cup (c_i, \text{UPDATE}, \text{START})$
    end if
  else
    $Activities_{stop} := Activities_{stop} \cup (c_i, \text{NONE}, \text{STOP})$
    $Activities_{start} := Activities_{start} \cup (c_i, \text{NONE}, \text{START})$
  end if
end for
for all $r_j = (c_1, c_2) \in R_t$:
    $(c_1, op_{stop,1}), (c_2, op_{stop,2}, \text{STOP}) \in Activities_{stop} \wedge$
    $(c_1, op_{start,1}), (c_2, op_{start,2}, \text{START}) \in Activities_{start}$ **do**
  if hostedOn $\in$ $allTypes(r_j)$ **then**
    $Order_{uog} := Order_{uog} \cup \{((c_1, op_{stop,1}, \text{STOP}), (c_2, op_{stop,2}, \text{STOP}))\}$
    $Order_{uog} := Order_{uog} \cup \{((c_2, op_{start,2}), (c_1, op_{start,1}, \text{START}))\}$
  else if connectsTo $\in$ $allTypes(r_j)$ **then**
    $act_{stop} := (r_j, \text{TERMINATE}, \text{STOP})$
    $act_{start} := (r_j, \text{PROVISION}, \text{START})$
    $Activities_{stop} := Activities_{stop} \cup act_{stop}$
    $Activities_{start} := Activities_{start} \cup act_{start}$
    $Order_{uog} := Order_{uog} \cup \{(act_{stop}, (c_1, op_{stop,1})), (act_{stop}, (c_2, op_{stop,2}))\}$
    $Order_{uog} := Order_{uog} \cup \{((c_2, op_{start,2}), act_{start}), ((c_1, op_{start,1}), act_{start})\}$
  end if
end for
// Stop-Phase zu Start-Phase verbinden
for all $(c_i, act_1) \in Activities_{stop} : ((c_i, uct, \text{STOP}), \_) \notin Order_{uog}$ **do**
  for all $(c_j, act_2) \in Activities_{start} : (\_, (c_j, uct, \text{START})) \notin Order_{uog}$ **do**
    $Order_{uog} := Order_{uog} \cup \{((c_i, act_1, \text{STOP}), (c_i, act_2, \text{START}))\}$
  end for
end for

Algorithmus 4.3 hasUpdatableAncestor($c \in C_t, t \in T$)

```
ancestorQueue := {c}
while |ancestorQueue| > 0 do
    ancestor ∈ ancestorQueue
    ancestorQueue := ancestorQueue \{ancestor}
    if ancestor ≠ c∧ update ∈ operations(ancestor) then
        return true
    else
        for r_i = (c_1, c_2) ∈ R_t : c_1 = ancestor do
            if hostedOn ∈ allTypes(r_i) then
                ancestorQueue := ancestorQueue ∪{c_2}
            end if
        end for
    end if
end while
return false
```

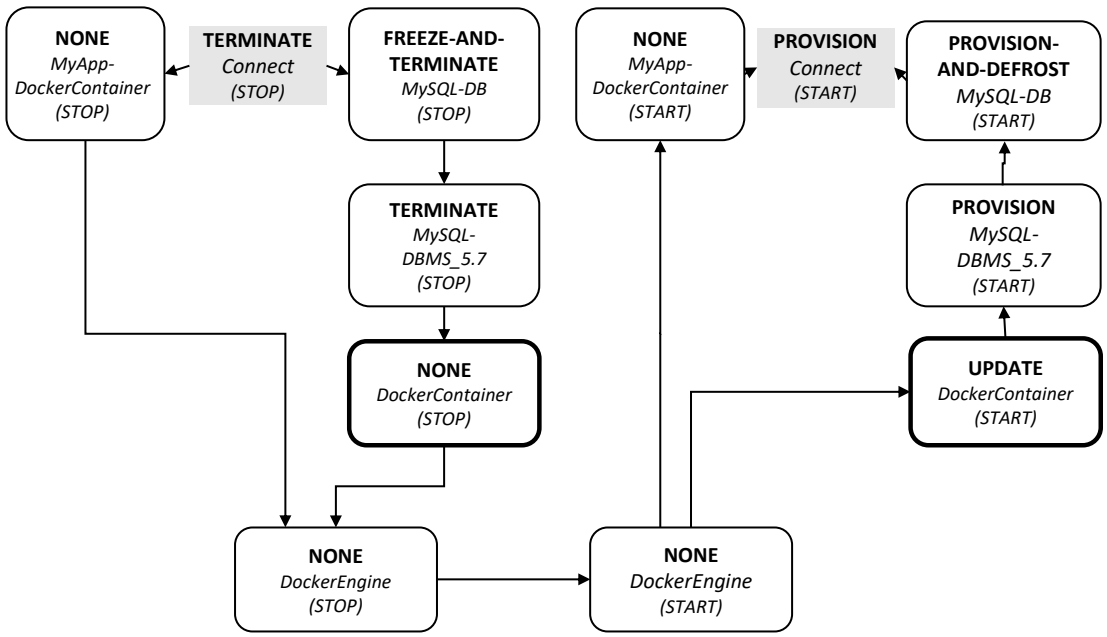


Abbildung 4.3: Update-Order-Graph, der aus dem Beispiel in Abbildung 1.1 generiert wird.

4.5.3 Schritt 3: Aktivitäten vervollständigen

Der letzte Schritt ist aus dem Order Graph einen ausführbaren Plan zu generieren.

Zuerst wird ein Plan Skeleton in der Zielsprache erstellt. Da bei BPEL innerhalb eines $< flow >$ ein Graph beschrieben wird, lässt sich ein BPEL Plan Skeleton erstellen, indem für jeden Knoten ein $< empty >$ und für jede Kante ein $< link >$ hinzugefügt wird.

Zuletzt wird der Plan Skeleton mit Hilfe von Logic Providers um die Logik vervollständigt. Bei BPEL können hierfür $< empty >$ durch Sequenzen an Arbeitsschritten ersetzt werden. Hier können Logic Providers aus vorherigen Arbeiten wiederverwendet werden. Provisioning Logic Providers aus [BBK+14] verarbeiten Node Templates und erstellen ausführbare PROVISION-Aktivitäten. Auch TERMINATE, FREEZE-AND-TERMINATE, PROVISION-and-DEFROST und UPDATE-Aktivitäten werden von entsprechenden Logic Providers generiert, welche in der Regel die entsprechenden Operationen der Komponente aufrufen. Zum Beispiel wird in der UPDATE-Aktivität die update-Operation ausgeführt.

Sobald alle Aktivitäten vervollständigt wurden, ist der Plan fertig und und kann von einer process-engine ausgeführt werden.

5 Prototyp

Zum Beweis der Machbarkeit wurde in OpenTOSCA Container ein Prototyp implementiert[1]. Es wurde eine PlanBuilder hinzugefügt der nach dem in Kapitel 4 vorgestellten Ansatz funktioniert. Bei OpenTOSCA Container werden beim Hinzufügen einer Anwendung automatisch alle verfügbaren PlanBuilder ausgeführt. Neben *Provision*- und *Termination*-Plänen wird nun auch ein *Update*-Plan mit einem PlanBuilder erzeugt, der mit BPEL spezifiziert wird. Sobald eine Instanz der Anwendung mit einem *Provision*-Plan erstellt wurde, ist es möglich den *Update*-Plan zu starten, der die Anwendung aktualisiert.

Die Implementierung wurde in der Programmiersprache Java geschrieben. Die Klasse `AbstractUpdatePlanBuilder` im Package `org.opentosca.planbuilder.core` enthält die Methode `generateUOG`, die aus einem Topology Template einen Update-Order-Graph generiert und implementiert somit Schritt 2 des Ansatzes in Abschnitt 4.5.2 auf Seite 32. Die Klasse `BPELUpdateProcessBuilder` im Package `org.opentosca.planbuilder.core.bpel` ist eine Unterklasse von `AbstractUpdatePlanBuilder`. Die Methode `buildPlan` dieser Klasse nutzt die Methode `generateUOG` der Oberklasse um einen Update-Order-Graphen zu generieren und erstellt anschließend daraus einen fertigen BPEL-Plan. Diese Methode ist somit eine Implementierung von Schritt 3 des Ansatzes in Abschnitt 4.5.3 auf Seite 35.

Damit die Operation „freeze“, erfolgreich ausgeführt werden kann, wird ein Speicherort benötigt, an den die Zustandsdaten geschrieben werden können. Deshalb wurde in der Klasse `NodeTemplateController` im Package `org.opentosca.container.api` der Endpunkt „/uploadDA“ der HTTP-Api von OpenTOSCA Container hinzugefügt. Update-Operationen können über eine HTTP PUT-Anfrage an diesen Endpunkt eine Datei übergeben, die als Artifact Template gespeichert und im Node Template eingebunden wird. OpenTOSCA Container stellt sicher, dass auf alle in einem Node Template eingebundenen Artifact Templates von der Komponente und somit auch von der Operation defrost zugegriffen werden kann.

[1] https://github.com/OpenTOSCA/container/pull/164

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Ansatz vorgestellt um Komponenten einer bereitgestellten TOSCA Anwendung automatisiert zu aktualisieren, indem ein ausführbarer Plan generiert wird, der die Komponenten aktualisiert. Dies ermöglicht (i) ein großes Zeitersparnis bei Aktualisierung von Anwendungen, (ii) das Ausführen von Aktualisierungen auch von unerfahrenen Benutzern, und damit verbunden ist (iii) höhere Sicherheit durch schnellere Aktualisierungen, die Sicherheitslücken beheben.

Die Anwendbarkeit des Ansatzes wurde demonstriert mit einem Prototypen in OpenTOSCA Container. Der Prototypen generiert ausführbare BPEL-Pläne, die bei Ausführung die Komponenten aktualisiert. Als Vorraussetzung ist gefordert, dass aktualisierbare Komponenten eine update Operation implementieren, die eine einzelne Komponente aktualisieren und dass zustandsbehaftete Komponenten freeze und defrost Operationen implementieren, die den Zustand speichern und laden.

Eine Limitation ist die Annahme, dass die Anwendung während der Aktualisierung untätig ist. Während der Aktualisierung werden Komponenten temporär gestoppt, dadurch kann die Anwendung währenddessen nicht im vollen Umfang benutzt werden. Ferner können externe Anfragen, während des Prozesses den internen Zustand verändern und zu einem inkonsistenten Zustand führen. Deshalb ist dieser Ansatz nicht für durchgehend verwendete Anwendungen geeignet, sondern erfordert Zeiträume in der die Anwendung nicht verwendet wird.

Literaturverzeichnis

[BBK+14] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. „Combining declarative and imperative cloud application provisioning based on TOSCA“. In: *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014* (2014), S. 87–96. DOI: 10.1109/IC2E.2014.56 (zitiert auf S. 13, 17, 20, 23, 35).

[BEK+16] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. „The OpenTOSCA Ecosystem - Concepts & Tools“. In: *European Space project on Smart Systems, Big Data, Future Internet-Towards Serving the Grand Societal Challenges* 1 (2016), S. 112–130. DOI: 10.5220/0007903201120130 (zitiert auf S. 13, 20).

[EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. „Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications“. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017, S. 22–27. ISBN: 978-1-61208-534-0 (zitiert auf S. 17).

[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: *CLOSER*. 2017, S. 180–190 (zitiert auf S. 17).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014 (zitiert auf S. 20).

[HBKL20] L. Harzenetter, U. Breitenbücher, K. Képes, F. Leymann. „Freezing and defrosting cloud applications: automated saving and restoring of running applications“. In: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (2020), S. 101–114. DOI: 10.1007/s00450-019-00415-8 (zitiert auf S. 20, 23, 25, 30).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „Winery–a modeling tool for TOSCA-based cloud applications“. In: *International Conference on Service-Oriented Computing*. Springer. 2013, S. 700–704 (zitiert auf S. 20).

[KBL17] K. Képes, U. Breitenbücher, F. Leymann. „Integrating IoT Devices Based on Automatically Generated Scale-Out Plans“. In: *Proceedings - 2017 IEEE 10th International Conference on Service-Oriented Computing and Applications, SOCA 2017* 2017-Janua (2017), S. 155–163. DOI: 10.1109/SOCA.2017.29 (zitiert auf S. 13, 20, 23).

[Mor16] K. Morris. *Infrastructure as code: Managing servers in the cloud*. First edition. Safari Books Online. Sebastopol, CA: O'Reilly Media, 2016. ISBN: 9781491924396 (zitiert auf S. 21).

[OASIS07] *Web Services Business Process Execution Language Version 2.0: OASIS Standard*. 11. Apr. 2007. URL: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf (zitiert auf S. 11, 13, 17, 18).

[OASIS13] *Topology and Orchestration Specification for Cloud Applications Version 1.0: OASIS Standard*. 25. Nov. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf (zitiert auf S. 11, 13, 18).

[OMG11] *Business Process Model and Notation (BPMN), Version 2.0*. 3. Jan. 2011. URL: https://www.omg.org/spec/BPMN/2.0/PDF (zitiert auf S. 11, 18).

Alle URLs wurden zuletzt am 26. 01. 2021 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