Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

TEST4TOSCA: Konzept und Implementierung einer Komponente zur Generierung von WS-BPEL 2.0 Testplänen

Simon Matejetz

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	Michael Wurster, M.Sc.
Beginn am:	1. November 2017
Beendet am:	15. Juni 2018

Kurzfassung

Der OASIS Standard TOSCA ermöglicht die Modellierung der Topologie von Cloud Anwendungen sowie deren automatisierte Provisionierung und Management über implizite, also manuell erstellte, oder deklarative, also automatisch generierte, Pläne in standardisierten Formaten wie WS-BPEL 2.0. Die deklarative Generierung solcher Pläne findet bislang zum Beispiel für Build- und Terminationpläne statt, welche den Ablauf des Deployments beziehungsweise der Terminierung einer zugehörigen TOSCA-Anwendung beschreiben. Gerade wenn ein solcher Buildplan deklarativ generiert wurde stellt sich die Frage, ob eine Anwendung, beziehungsweise einzelne Komponenten der Anwendung, erfolgreich provisioniert wurden und ob vorgegebene Richtlinien, sogenannte Compliance Rules, eingehalten werden. Bisher existiert für TOSCA-Anwendungen jedoch keine Möglichkeit um automatisiert zuvor definierte Tests auszuführen nachdem diese deployt wurde. Um dies zu ermöglichen wurde in der vorliegenden Arbeit eine Softwarekomponente konzipiert und entwickelt, welche auf Grundlage einer TOSCA-Anwendung zusätzlich zu den bisher generierbaren Plänen einen von der TOSCA-Laufzeitumgebung unabhängigen Testplan erzeugt um die Anwendung nach erfolgreicher Provisionierung auf die Einhaltung von Compliance Rules und ihre Funktionalität zu testen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

API Programmierschnittstelle. 39

BPMN Business Process Model and Notation. 23

BPS Business Process Server. 28

CSAR Cloud Service Archive. 5, 19

DAG Directed Acyclic Graph. 40

DOM Data Object Model. 40

HTTP Hypertext Transfer Protocol. 26

IA Implementation Artifact. 5, 22

ID Identifier. 26

JRE Java Runtime Environment. 37

OSGi Open Services Gateway initiative. 46

PaaS Platform as a Service. 21

PoC Proof of Concept. 47

RAM Arbeitsspeicher. 21

RPC Remote Process Invocation. 26

SOAP Simple Object Access Protocol. 26

TOSCA Topology and Orchestration Specification for Cloud Applications. 5, 17

URL Uniform Resource Locator. 26

VM Virtuelle Maschine. 19

W3C World Wide Web Consortium. 25

WAR Web Archive. 22

WS-BPEL 2.0 Web Services Business Process Execution Language 2.0. 5, 17

WSDL Web-Service Description Language. 5, 18

XML Extensible Markup Language. 19

XSD XML Schema Definition. 25

YAML YAML Ain't Markup Language. 23

1 Einleitung

In den letzten Jahren ist Cloud Computing ein immer wichtigeres Feld im Bereich der Informatik geworden. Mehr und mehr Unternehmen verkürzen die Zeit zwischen Entwicklung und Auslieferung neuer Softwareprodukte, beziehungsweise neuer Versionen dieser, durch den Einsatz von Continuous Delivery und automatischem Deloyment. Zusätzlich zu kürzeren Zyklen in der Auslieferung hat dies neben der gewonnenen Schnelligkeit im Entwicklungsprozess den Vorteil, dass automatische Prozesse weit weniger fehleranfällig sind als die manuelle Durchführung dieser [OGP03].

Zur automatischen Durchführung des Deployments werden sogenannte Deployment Models, welche entweder imperativ definiert oder deklarativ auf Grundlage der Struktur und Eigenschaften einer Anwendung generiert werden, verwendet. Diese Modelle beschreiben alle auszuführenden Schritte welche nötig sind um eine Anwendung erfolgreich zu deployen und können von einer unterstützenden Laufzeitumgebung ausgeführt werden um das tatsächliche Deployment oder, im Falle anderer Models, eine andere Managementaufgabe der zugehörigen Anwendung zu übernehmen [BBF+18].

Bei den Deployment Models steht dabei die technisch korrekte Ausführung des Deployments, sprich die Installation, Konfiguration und Orchestrierung einer Anwendung und deren Komponenten im Mittelpunkt. Das Ergebnis dieser Ausführung wird anschließend jedoch nicht auf seine Funktionalität geprüft. Diese Überprüfung wäre allerdings sinnvoll, da auch bei einem technisch korrekten Deployment eine korrekte Funktionalität nicht garantiert ist. Das kann beispielsweise der Fall sein, wenn trotz erfolgreicher Installation, zum Beispiel wegen blockierter Ports oder ähnlichem, keine Verbindung der Komponenten untereinander zustande kommt. Ein weiter Faktor welcher zu mangelnder Funktionaltiät führen kann sind mögliche Unterschiede der verschiedenen Cloud-Umgebungen in welchen eine Anwendung ausgebracht wird [Pet11]. Ein solcher Funktionsfehler wird dann im schlimmsten Fall erst von einem Endnutzer bei der Verwendung der Anwendung entdeckt.

Zudem stellt sich nach dem erfolgreichen Deployment einer Anwendung die Frage ob die erstellte Anwendung vorgegebene Richtlinien einhält. Solche meist unternehmensinternen Vorgaben werden auch Compliance Rules genannt. Es handelt sich dabei um, vor dem Deployment definierte, gewünschte Eigenschaften einer Anwendung. Dabei kann es sich beispielweise um die Version einer verwendeten Komponente oder eine bestimmte Art der Verschlüsselung von Dateien handeln. Das Einhalten dieser Compliance Rules ist von höchster Wichtigkeit, da Verletzungen gegebenenfalls sogar strafbar vor den 2018 erlassenen Datenschutzgesetzen [Amt16] sind. Diese Gesetzte gewinnen im Moment immer mehr an Relevanz, da bei bei einer Verletzung hohe Geldstrafen drohen [Dat18]. Das Verletzen dieser Compliance Rules führt in der Regel jedoch nicht zu einem Fehlschlagen des Deployments oder der Beeinträchtigung der Funktionalität einer Anwendung und ist daher nur schwer manuell festzustellen.

Bei Topology and Orchestration Specification for Cloud Applications (TOSCA) handelt es sich um einen Standard nach welchem Cloud-Anwendungen einheitlich modelliert werden können. Eine nach diesem Standard modellierte Anwendung kann, ebenfalls via Models in Form von Web Services Business Process Execution Language 2.0 (WS-BPEL 2.0)-Plänen, deployt, gemanagt und terminiert werden. Diese Anwendungen nach dem Deployment zu testen ist bisher jedoch nur manuell möglich, da es keine standardisierte Möglichkeit gibt Tests für eine TOSCA-Anwendung zu definieren und diese nach dem Deployment automatisiert auszuführen.

Ziel dieser Arbeit ist es eine Möglichkeit zu schaffen um bereits bei der Modellierung von TOSCA-Anwendungen Tests zu definieren. Die Tests sollen dann nach erfolgreicher Instanziierung der Gesamtanwendung automatisiert ausgeführt werden können um die Anwendung beispielsweise auf ihre Funktionalität oder die Einhaltung zuvor festgelegter Compliance Regeln zu testen. Dadurch soll festgestellt werden ob nur das technische Deployment erfolgreich war oder die Anwendung tatsächlich wie gewünscht initialisiert wurde, um die zuvor genannten Risiken zu minimieren. Die TOSCA Spezifikation nennt einige Standards an Plantypen, welche von TOSCA-Laufzeitumgebungen als Beschreibung des Deployment Model unterstützt werden müssen. Aus den zuvor definierten Tests wird ein Testplan generiert welcher dem von TOSCA unterstützten WS-BPEL 2.0-Standard entspricht und somit von jeder TOSCA-Laufzeitumgebung ausgeführt werden kann.

Gliederung

In Kapitel 2 werden zunächst die der Arbeit zugrundeliegenden Technologien und Standards behandelt. Dabei handelt es sich hauptsächlich um den TOSCA Standard, auf welchem das OpenTOSCA Ecosystem, welches durch diese Arbeit erweitert wird, aufbaut, sowie dessen für die Arbeit relevante Bestandteile. Desweiteren werden die WS-BPEL 2.0 und Web-Service Description Language (WSDL) Standards näher erläutert, welche vor allem für das Verständnis des Konzepts und der Funktionalität der Implementierung essenziell sind. Nachdem die Grundlagen erklärt wurden, werden anschließend in Kapitel 3 verwandte Arbeiten vorgestellt, auf welchen die Implementierung teilweise aufbaut, beziehungsweise welche eine starke Überschneidung im behandelten Themenfeld aufweisen. Das der Umsetzung zugrundeliegende Konzept wird dann in Kapitel 4 erklärt und anhand von Beispielen verdeutlicht um anschließend in Kapitel 5 darauf aufbauend die Implementierung als Erweiterung des OpenTOSCA Ecosystem vorzustellen und technische Details weiter zu erläutern. Abschließend folgt in Kapitel 6 eine abschließende Zusammenfassung der Arbeit sowie ein Ausblick möglicher zukünftiger Änderungen und Erweiterungen der entwickelten Software.

2 Grundlagen

In diesem Kapitel werden die Grundlagen der TOSCA Spezifikation [LMPS13] sowie dazugehörende, für diese Arbeit relevanten, Bestandteile, dem Cloud Service Archive (CSAR) Format, sowie den WS-BPEL 2.0- und WSDL-Standards.

2.1 Topology and Orchestration Specification for Cloud Applications

Die TOSCA-Spezifikation [LMPS13] beschreibt den TOSCA-Standard als ein auf Extensible Markup Language (XML) basierendes Metamodel mit welchem Cloud-Anwendungen beziehungsweise Web-Services abstrakt und mit wiederverwendbaren Elementen modelliert werden können. Dabei werden einerseits die Struktur aller der Anwendung zugrunde liegenden Komponenten modelliert, sowie Managementoperationen zur späteren Steuerung der instanziierten Anwendung definiert. Die auf diesem Wege modellierten Anwendungen werden als CSAR gepackt und können anschließend in einer TOSCA-Laufzeitumgebung deployt, gemanagt und terminiert werden.

2.1.1 Komponenten

Eine TOSCA-Anwendung besteht aus verschiedenen, in der Spezifikation definierten, Komponenten. Diese Komponenten können, nachdem sie einmal definiert wurden, aufgrund ihrer verschiedenen Abstraktionsebenen auf unterschiedliche Art und Weise für die Modellierung weiterer TOSCA-Anwendung wiederverwendet werden.

Im Folgenden werden die für diese Arbeit relevanten Komponenten des TOSCA-Standards vorgestellt und auf deren Eigenschaften und Funktionen eingegangen.

Service Template Eine TOSCA-Anwendung wird in ihrer Gesamtheit durch ein Service Template beschrieben. Das Service Template enthält, wie in Abbildung 2.1 dargestellt, ein *Topology Template* sowie Node Templates und Relationship Templates welche jeweils durch einen Node Type beziehungsweise Relationship Type typisiert werden. Desweiteren werden die Management-Pläne einer TOSCA Anwendung im Service Template referenziert oder definiert.

Es ist zudem möglich, dass ein Service Template selbst ein weiteres Service Template referenziert. Das referenzierte Service Template wird dabei wie ein herkömmliches Node Template behandelt. Eine deployte TOSCA-Anwendung kann daher als aus der Gesamtheit ihres Service Templates abgeleitete Instanz betrachtet werden.

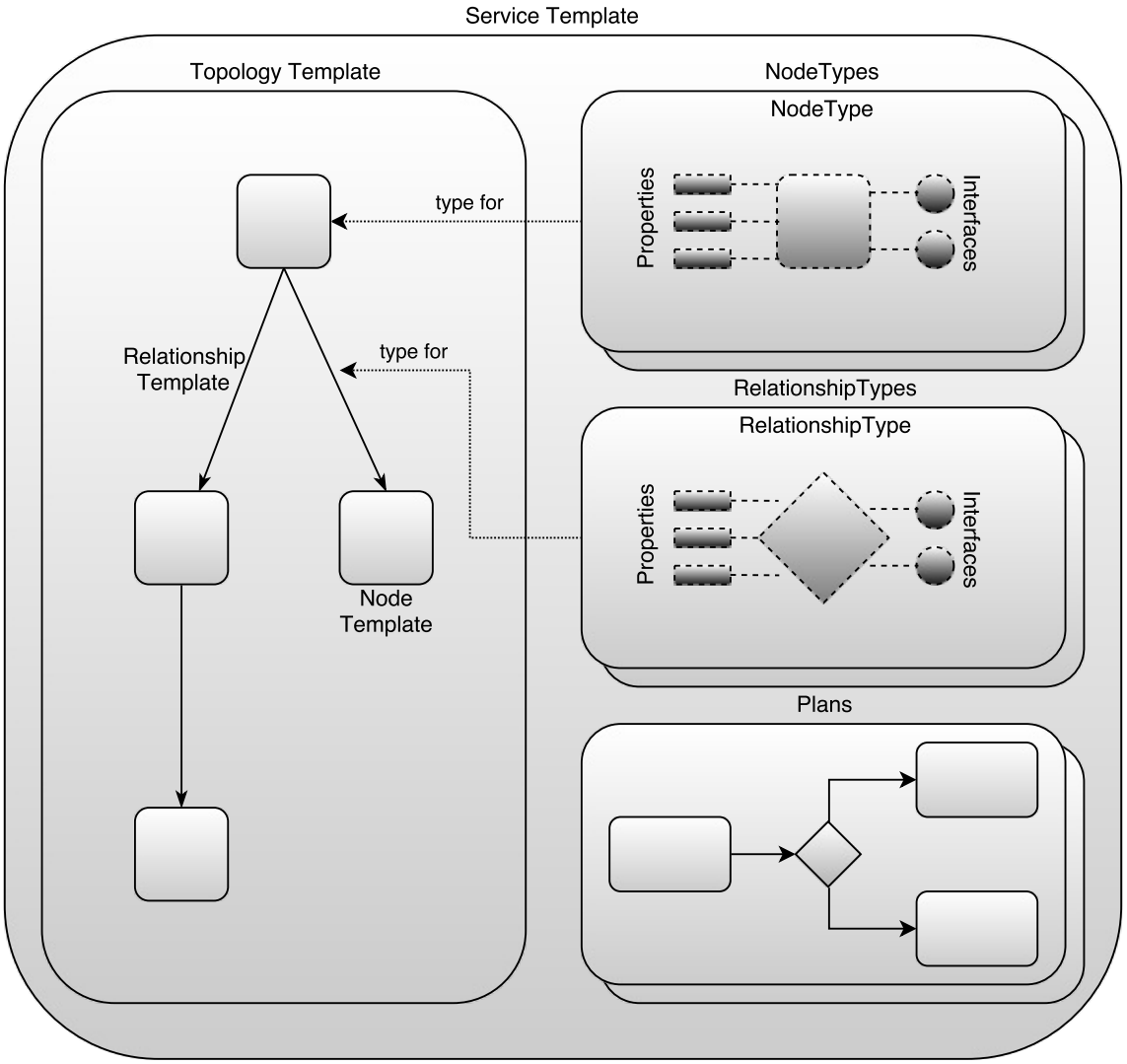

Abbildung 2.1: Struktur eines Service Templates und Zusammenhänge der einzelnen Komponenten [LMPS13]

Topology Template Ein Topology Template definiert die Strukur der Komponenten einer TOSCA-Anwendung in Form eines Graphs. Dies wird dadurch erreicht, dass das Topology Template die in der modellierten Anwendung enthaltenen Bestandteile in eine eindeutigen Hierarchie bringt. Bei diesen Bestandteilen handelt es sich um Node Templates, welche die Knoten des Graphs repräsentieren, sowie Relationship Templates, welche die Kanten darstellen. Wie eine solche Hierarchie aussehen kann ist ebenfalls in Abbildung 2.1 dargestellt. Die Node Templates sind dabei über Relationship Templates mit anderen Node Templates verbunden. Bei Node Templates welche keine ausgehenden Relationship Templates haben es sich um sogenannte Infrastruktur Node Templates wie beispielsweise eine lokale Ubuntu-Virtuelle Maschine (VM) oder eine externe Cloud auf welchen dann weitere Komponenten, beispielsweise mit einer *hosted-on* Relationship, aufbauen. Die Node Templates repräsentieren damit die einzelnen Komponenten einer TOSCA-Anwendung, die Relationship Templates hingegen beschreiben die Beziehungen der Komponenten untereinander.

Node Type Wie von den Konzepten objektorientierter Programmierung bekannt, entsprechen die Abstraktionslevel in TOSCA derer von Klassen und konkreter, von einer bestimmten Klasse abgeleiteter, Objekte. Wie auch in der objektorientierten Programmierung ist einer der größten Vorteile hierbei, dass auf diesem Wege einmal definierte Konstrukte auf verschiedene Art und Weise mit jeweils unterschiedlichen Eigenschaften aber der selben Grundstruktur wiederverwendet werden können.

In der TOSCA Definition repräsentieren hierbei die Node Types die abstrakten Klassen und definieren, wie in Abbildung 2.1 modelliert, Attribute, welche in TOSCA Properties genannt werden. Diese abstrakten Properties legen für die von dem jeweiligen Node Type abgeleiteten Node Templates fest welchen Namen und Typen die Properties besitzen, ohne diesen einen Wert zuzuweisen.

Desweiteren definiert ein Node Type ein oder mehrere Interfaces mit jeweils einer oder mehreren Operationen, zu welchen zum einen die TOSCA-Managementoperationen gehören, über welche sich eine Komponente beispielsweise starten und stoppen lässt und zum anderen Domain-Specific Operations bzw. Application Interfaces, welche die in Abschnitt 2.1.1 beschriebenen Implementation Artifacts referenzieren um bestimmte Funktionalitäten eines Node Templates bereitzustellen.

Node Template Bei einem Node Template handelt es sich um die konkrete Verwendung eines Node Types in einem konkreten Topology Template. Zusätzlich zur in Abbildung 2.1 dargestellten Modellierung der Struktur einer TOSCA-Anwendung besitzen Node Templates einige weitere wichtige Eigenschaften für die Beschreibung einer in TOSCA modellierten Cloud-Anwendung. Ein Node Template belegt die Properties des zugehörigen Node Types mit konkreten Werten des vom Node Type vorgegebenen Typs.

Da die Properties von den Node Types definiert, aber erst beim Erstellen eines Node Templates mit festen Werten versehen werden, ist es möglich zwei Node Templates mit verschiedenen Properties aus dem selben Node Type zu erzeugen.

Eine TOSCA-Anwendung kann zum Beispiel über zwei Platform as a Service (PaaS)-Komponenten verfügen, welche von dem selben *VM*-Node Type abgeleitet sind, jedoch mit einem unterschiedlich großen Arbeitsspeicher (RAM) erstellt werden sollen. Die Größe des RAM könnte in diesem Beispiel eine im *VM* Node Type definierte Property sein, welche dann in den beiden konkreten Node Templates mit dem jeweils gewünschten Wert belegt wird.

Zusätzlich können in Node Templates konkrete Policy Templates referenziert werden, welche dem Node Template zusätzliche Eigenschaften und Informationen, welche der Node Type nicht besitzt, hinzufügen können, ohne dass dafür der Node Type verändert werden muss.

Interfaces und Operations Ein Interface eines Node Types gruppiert eine oder mehrere Operationen unter einem gemeinsamen logischen Bezeichner. Dabei kann ein Node Type über beliebig viele Interfaces verfügen. Eine Operation definiert ihre für einen Aufruf benötigten Eingabeparameter, ihre Ausgabeparameter sowie deren Bezeichner und Typen.

Über diese Interfaces bzw. Operationen ist es möglich einzelne Node Templates bei der Ausführung eines Plans anzusprechen beziehungsweise zu steuern.

Node Type Implementation Bei einer Node Type Implementation handelt es sich um die konkrete Implementierung der Schnittstellen eines Node Types. Eine Node Type Implementation definiert hierbei, welches Implementation Artifact (IA) die Implementierung für welches zum entsprechenden Node Type gehörenden Interface, beziehungsweise für welche Operation, enthält. Eine Node Type Implementation ist damit ein Container für Referenzen zu Quellcode und anderen Bestandteilen, welche die Funktionalität der Schnittstellen eines Node Types bilden. Eine solche Schnittstelle, welche jeder Node Type anbieten muss, sind die die TOSCA Lifecycle Operationen „install“, „configure“, „start“, „stop“ und „uninstall“ über welche sich Komponenten standardmäßig starten, managen und beenden lassen [LM13].

Ein Node Type vom Typ Web Archive (WAR) würde also in seiner Node Type Implementation ein WAR Artifact referenzieren, welches die vom Node Type definierten Operationen als Java Methoden implementiert.

Implementation Artifact IAs sind die Implementierungen bzw. der Quellcode für die im Node Type definierten Operationen. Ein IA besitzt immer einen Artifact Type sowie das Artifact selbst als Datei. Die gebräuchlichsten Artifact Types sind WAR Artifacts sowie Script Artifacts.

Relationship Templates Wie in Abbildung 2.1 dargestellt können Node Templates mit anderen Node Templates in einer Beziehung stehen. Diese Beziehungen zwischen Node Templates werden über Relationship Templates realisiert. Beispielsweise kann ein Node Template Java8, welches auf einem Node Template Ubuntu-VM installiert wird mit einem *hosted-on* Relationship Template abgebildet werden. Dabei wäre das Java8 Node Template die Quelle und das Ubuntu-VM Node Template das Ziel dieses Relationship Templates.

Policy Type Ähnlich wie bei Node Types stellt ein Policy Type eine abstrakte Beschreibung einer Policy dar. Ein Policy Type definiert ebenfalls Propertybezeichner sowie deren Typ, jedoch keine festen Werte. Policy Types können voneinander abstammen, sie erben dann alle Properties des Eltern-Policy Types beziehungsweise überschreiben Properties mit identischem Bezeichner.

Policy Template Genau wie bei den in Abschnitt 2.1.1 vorgestellten Node Templates handelt es sich bei PolicyTemplates um konkrete Instanzen eines Policy Types. Policy Templates initialisieren die vom zugehörigen Policy Type definierten Properties mit konkreten Werten und können von Node Templates referenziert werden. Man kann sich Policy Templates als Erweiterung eines Node Templates vorstellen, welche zusätzliche Einschränkungen, Bedingungen oder Properties für ein das Policy Template referenzierendes Node Template definiert, unabhängig vom Node Type des Node Templates.

In der Arbeit [KBF+17] wird vorgestellt, wie man Policy Templates beispielsweise dafür benutzen kann um Policies wie *Public Access Policy*, *No Public Access Policy Only Modeled Ports Policy* oder *Secure Password Policy* in bestimmten Nodes zu referenzieren. Damit können Node Type unabhängige Eigenschaften für einzelne Node Templates definiert werden, ohne dass der Node Type selbst verändert werden muss.

So kann jedes beliebige Node Template, für welche diese Eigenschaft gewünscht ist, mit einer *No Public Access Policy* versehen werden. Node Templates, welche diese Policy referenzieren sind dann, in diesem Beispiel, nicht über öffentliche Schnittstellen erreichbar.

Plans Plans in TOSCA beschreiben das Management der Instanzen einer TOSCA-Anwendung. Typischerweise handelt es sich dabei um standardisierte Pläne, welche via Interfaces bzw. Operationen der Node Templates high-level Operationen umsetzen um unter anderem den Lebenszyklus einer Anwendung zu steuern. Bei solchen high-level Operationen kann es sich beispielsweise darum handeln, eine Instanz einer TOSCA-Anwendung zu erstellen oder zu beenden.

Die für diese Zwecke auszuführenden Operationen der Node Templates werden dabei in einem Business Process Model and Notation (BPMN)- oder WS-BPEL 2.0-Plan, welche in Abschnitt 2.1.1 tiefer behandelt werden, festgelegt.

2.1.2 Definitions

TOSCA verwendet den XML-Standard, was bedeutet dass alle Beschreibungen der Bestandteile einer TOSCA-Anwendung im XML Format verfasst werden. TOSCA bietet zudem die Möglichkeit diese Dateien im YAML Ain't Markup Language (YAML)-Format anzulegen, was hier jedoch nicht weiter behandelt wird. Bei Beschreibungen dieser Art nach dem TOSCA-Standard handelt es sich um sogenannte Definitions. Jeder Bestandteil einer TOSCA-Anwendung hat daher seine eigene Definition-Datei. Diese Definition-Datei definiert alle Eigenschaften, Abhängigkeiten und Referenzen des entsprechenden Bestandteils.

In Listing 2.1 ist abgebildet, wie eine solche Definitions-XML aussehen kann. Der durch die dargestellte Definitions Datei beschriebene Node Type definiert drei Properties sowie ein Interfaces mit einer Operation. Bei den Properties handelt es sich um *VMIP*, *VMUserName* und *VMUserPassword*, welche alle vom Typ *string* sind. Desweiteren wird ein Interface *OperatingSystemInterface* definiert, welche eine *installPackage* Operation anbietet. Diese Operation definiert wiederum ihre benötigten Input-Parameter *VMIP*, *VMUserName*, *VMPrivateKey* sowie *PackageNames* und zusätzlich einen Output-Parameter *InstallResult*.

Die Parameter *VMIP*, *VMUserName* und *VMPrivateKey* stimmen im Bezeichner mit Properties überein, und können somit bei einem tatsächlichen Aufruf aus den Properties des enstprechenden Node Templates befüllt werden. Das *PackageNames* Parameterfeld muss bei einem Aufruf immer zusätzlich mitgeliefert, beziehungsweise aus anderer Quelle beschafft, werden.

Sowohl die Input- als auch der Outputparameter sind für diese Operation als Pflichtfelder definiert und können somit bei einem Aufruf der Operation nicht leer bleiben.

Ähnlich wie die XML-Datei in Listing 2.1 sind auch die Definitions anderer Bestandteile aufgebaut, die detailierte Aufzählung und Gegenüberstellung ist für die weitere Arbeit jedoch nicht relevant.

Listing 2.1 Definition eines Node Types am gekürzten Beispiel des *Ubuntu-VM* Node Types

```
<Definitions>
      <NodeType name="Ubuntu-VM" abstract="no" final="no">
            <PropertiesDefinition>
                  <properties>
                        <key>VMIP</key>
                        <typ>xsd:string</typ>
                  </properties>
                  <properties>
                        <key>VMUserName</key>
                        <typ>xsd:string</typ>
                  </properties>
                  <properties>
                        <key>VMUserPassword</key>
                        <typ>xsd:string</typ>
                  </properties>
            </PropertiesDefinition>
            <Interfaces>
            <Interface name="OperatingSystemInterface">
                  <Operation name="installPackage">
                  <InputParameters>
                        <InputParameter name="VMIP" type="xsd:String" required="yes"/>
                        <InputParameter name="VMUserName" type="xsd:String"  required="yes"/>
                        <InputParameter name="VMPrivateKey" type="xsd:String" required="yes"/>
                        <InputParameter name="PackageNames" type="xsd:String" required="yes"/>
                  </InputParameters>
                  <OutputParameters>
                        <OutputParameter name="InstallResult" type="xsd:String"
                            required="yes"/>
                  </OutputParameters>
                  </Operation>
            </Interfaces>
      </NodeType>
</Definitions>
```

2.1.3 Cloud Service Archive

Bei einem CSAR handelt es sich um ein ZIP-Archiv mit der Dateiendung *.csar*, welches eine nach TOSCA modellierte Anwendung mit allen zugehörigen Beschreibungen und sonstigen Dateien repräsentiert. Ein CSAR enthält dabei mindestens zwei Unterordner, bei einem davon handelt es sich um den *TOSCA-Metadata*-Ordner, der andere ist der *Definitions* Ordner [LMPS13].

Ein CSAR ist in sich geschlossen. Das bedeutet, dass zusätzlich zu den genannten Pflichtbestandeteilen alle für den gesamten Lifecycle der TOSCA-Anwendung benötigten Bestandteile ebenfalls im CSAR enthalten sind. Dazu gehören neben den Metadaten und Definitions alle zur Anwendung gehörenden Artefakte und Pläne. Die Struktur eines CSARs kann zusätzlich beliebig erweitert werden. Abbildung 2.2 stellt beispielhaft die Struktur eines validen CSARs für eine TOSCA-Anwendung dar. Das abgebildete CSAR enthält neben der Pflichtbestandteile ein WAR-Artifact, einen Build-Plan und einen Termination-Plan. „/“ stellt das Rootverzeichnis des CSARs dar. Durchgezogene Linien repräsentieren den Teil des CSAR welcher zwingend vorhanden sein muss, die gestrichelte Linie den Teil, welcher zur Vollständigkeit des CSARs hinzugefügt wurden. Drei Punkte am Ende eines Pfades kennzeichnen, dass hier noch weitere

Ordner oder Dateien erlaubt wären. Auf dieser Weise können einer TOSCA-Anwendung weitere TOSCA-Laufzeitumgebung spezifische Eigenschaften hinzugefügt werden indem ein neuer Ordner, welcher dann die zusätzlichen Informationen enthält, angelegt wird. TOSCA-Laufzeitumgebungs, welche diese Eigenschaften nicht unterstützen können die Anwendung dann gegebenenfalls trotzdem ausbringen indem der zusätzliche Ordner ignoriert wird.

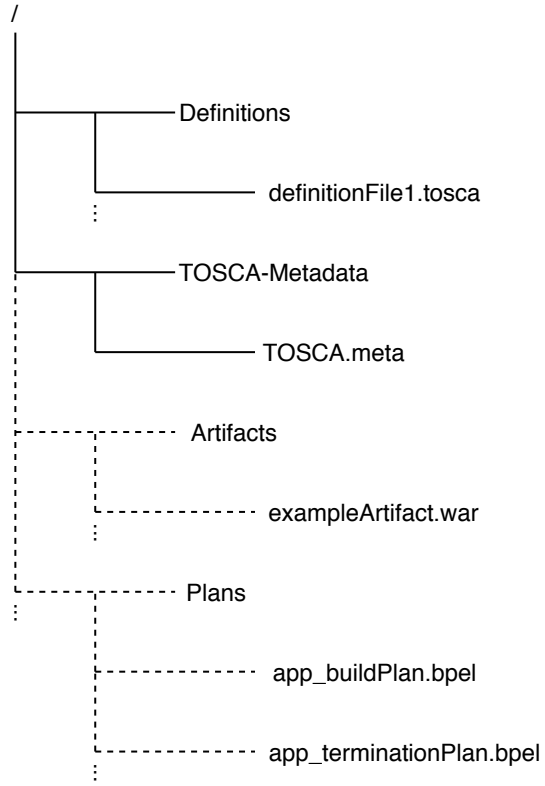

Abbildung 2.2: Beispiel eines validen CSARs

2.2 Web-Service Description Language 1.1

Bei WSDL handelt es sich um einen, vom World Wide Web Consortium (W3C) im Jahre 2001 hervorgebrachten, Standard [CCMW01] zur Beschreibung von Web-Services. Es gibt eine neuere Version 2.0 [CMRW07], welche hier jedoch nicht behandelt wird, da im behandelten Planbuilder die Version 1.1 zum Einsatz kommt. Jede spätere Erwähnung meint daher immer die WSDL Version 1.1.

Durch Dokumente welche diesem XML erweiternden Standard folgen werden die Schnittstellen und Eigenschaften eines Web-Services beschrieben. Im Folgenden werden die wichtigsten, in Listing 2.2 dargestellten, Bestandteile eines solchen Dokuments sowie deren jeweilige Verwendung beschrieben.

<types> Mit dem *<types>*-Element eines WSDL-Files können beliebig viele eigene Datentypen, via *<schema>*s, definiert werden. Bei diesen Typen handelt es sich im Endeffekt um interne XML Schema Definition (XSD), welche die XML-Standardtypen um komplexere Datentypen, die selbst wiederum aus XML-Standardtypen aufgebaut sind, erweitern und später für die Definition der Schnittstellen genutzt werden können.

Im Beispiel in Listing 2.2 wird ein neues *<schema>* mit dem root-element namens *ItemInfo* definiert. Dieses Element besteht nach dieser beispielhaften Definition aus zwei weiteren Elementen bei welchen es sich um die Identifier (ID) als Dezimalzahl sowie dem Itemname als String in einer beliebigen Reihenfolge handelt.

<message> Ein *<message>*-Element beschreibt einen Nachrichtentyp, welcher später zum Datenaustautsch zwischen den Schnittstellen genutzt werden kann. Dieser festgelegte Nachrichtentyp besteht aus mindestens einem *<part>*, wobei es sich bei einem *<part>* wiederum sowohl um einen vorher eigens definierten *<type>* als auch um einen der XML-Standardtypes handeln kann. Auch Kombinationen aus beidem können zu einem Messageformat zusammengefügt werden.

In Listing 2.2 wird auf diese Weise ein neuer Nachrichtentyp namens *BuyItemRequest* definiert, welcher später zur Anfrage an den Service verwendet werden kann. Dieser Nachrichtentyp besteht aus einem Element des zuvor definierten *ItemInfo* Typs sowie einem zusätzlichen Timestamp welcher vom XML-Standardtype String ist.

<portType> Ein *<portType>* definiert die Schnittstelle des beschriebenen Web-Services. Dabei werden abstrakt Operationen sowie deren Parameter beschrieben. Es können hierbei sowohl *<input>* als auch *<output>* Parameter definiert werden. Hierbei ist zu beachten, dass es sich bei dem beschriebenen *<portType>* um eine One-Way Schnittstelle handelt sofern der *<output>* Parameter nicht definiert wird und um eine Request-Response Schnittstelle, falls beide Elemente vorhanden sind.

In Listing 2.2 wird demnach eine One-Way Schnittstelle definiert, welche als Eingabeparameter eine Nachricht im Format des zuvor definierten Nachrichtenformats *BuyItemRequest* erwartet.

<binding> Das *<binding>*-Element bindet den zuvor definierten *<portType>* an eine Simple Object Access Protocol (SOAP)-Schnittstelle. Dabei wird sowohl die Art der Übertragung in Form von Remote Process Invocation (RPC) oder document ausgewählt, als auch das Übertragungsprotokoll sowie der Name der aufzurufenden SOAP-Operation und ob, beziehungsweise wie, die Nachricht bei der Übertragung verschlüsselt werden soll.

Im Beispiel von Listing 2.2 wird somit der *<portType>* an eine SOAP-Schnittstelle gebunden welche eine *buyItem*-Operation anbietet. Zudem wird Hypertext Transfer Protocol (HTTP) als Protokoll gewählt sowie mit dem Schlüsselwort *literal* keine Verschlüsselung bei der Übertragung verwendet.

Listing 2.2 Minimalbeispiel einer WSDL-Datei für einen Webshop

```
<definitions>
      <types>
            <schema>
                  <element name="ItemInfo">
                        <complexType>
                              <all>
                                    <element name = "id" type = "xsd:decimal"/>
                                    <element name = "itemname" type = "xsd:string"/>
                              </all>
                        </complexType>
                  </element>
            </schema>
      </types>

      <message name="BuyItemRequest">
            <part name="item" element="ItemInfo"/>
            <part name="timestamp" type="xsd:string"/>
      </message>

      <portType name="BuyItemPortType">
            <operation name="buyItem">
                  <input message="BuyItemRequest"/>
            </operation>
      </portType>

      <binding name="SOAPBinding" type="BuyItemPortType">
            <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
            <operation name="buyItem">
                  <soap:operation soapAction="buyItem"/>
                  <input>
                        <soap:body use="literal"/>
                  </input>
            </operation>
      </binding>

      <service name ="ShopService">
            <port binding = "SOAPBinding" name = "ShopPort">
                  <soap:address location = "http://www.example.com/MyShopAPI/"/>
            </port>
      </service>
</definitions>
```

<service> Letztendlich werden alle zuvor definierten Elemente zu einem *<service>*-Element zusammengefasst, indem eine konkrete Uniform Resource Locator (URL), hinter welcher sich die reale SOAP-Schnittstelle befindet, mit einem zuvor definierten *<portType>* zusammengeführt wird und somit konkrete *<ports>* gebildet werden.

2.3 Web Services Business Process Execution Language 2.0

Die WS-BPEL 2.0-Spezifikation [OAS07] definiert ein Modell in Form einer XML-Erweiterung um das Verhalten eines Business Prozesses basierend auf seinen Interaktionen zu beschreiben. Genauer

beschreibt ein solcher Plan die Interaktion zwischen den, via nach dem zuvor beschriebenen WSDL-Standard definierten, Web-Service Schnittstellen des Business Prozesses mit anderen, ebenfalls in WSDL beschriebenen, Web-Services. Dabei wird nicht nur der Ablauf der Interaktion zwischen einzelnen Schnittstellen sowie die Reihenfolge deren Ausführung festgelegt, sondern auch die Formate der Nachrichten welche dabei ausgetauscht werden.

Wie bereits dem Namen zu entnehmen können korrekt definierte WS-BPEL 2.0 Pläne in einem entsprechenden Business Process Server (BPS) ausgeführt werden. Auf diese Weise ausgeführte Pläne agieren dann wiederum selbst als eigenständiger Web-Service und fungieren als vermittelnde Instanz zwischen den zu integrierenden, in der Regel SOAP, Schnittstellen.

Ein auf diesem Wege erzeugte WS-BPEL 2.0 Web-Service kann dann, z.B. auf Basis von Message-Events, weitergehende Event-Chains auslösen, die empfangenen Nachrichten synchron oder asynchron weiterleiten beziehungsweise auch im Prozess der Weiterleitung verändern und somit beispielsweise die Integration zweier Web-Services, welche per se nicht kompatibel sind realisieren.

3 Verwandte Arbeiten

In diesem Kapitel werden Arbeiten vorgestellt, welche sich mit ähnlichen Themen wie dem in dieser Arbeit behandelten beschäftigen. Dazu gehören unter anderem Arbeiten welche sich zumindest teilweise mit der Entwicklung von vergleichbaren beziehungsweise in der Implementierung verwendeten Komponenten und Erweiterungen für OpenTOSCA befassen und teilweise als Grundlage für diese Arbeit dienten oder die Entwicklung und Konzeption durch vorhandene Ideen, welche in abgewandelter Art für den Entwurf und die Umsetzung des Konzepts übernommen werden konnten beziehungsweise die Konzeptfindung und Implementierung beeinflusst haben.

Konzept und Implementierung einer Java-Komponente zur Generierung von WS-BPEL 2.0 BuildPlans für OpenTOSCA In seiner Arbeit [Kép13] aus dem Jahre 2013 beschäftigt sich Kálmán Képes mit der Konzeption und Implementierung eines Planbuilders in der Programmiersprache Java, welcher Build Pläne im WS-BPEL 2.0-Format für TOSCA Anwendungen generiert.

Ziel der Arbeit war es, auf der Grundlage eines CSARs einen prototypischen WS-BPEL 2.0-Plan zu erzeugen, welcher später dem CSAR hinzugefügt werden kann und das Deployment der modellierten TOSCA-Anwendung beschreibt. Dabei wird im Dokument insbesondere formal auf die high-level Algorithmen zum finden einer geeigneten Provisionierungsreihenfolge sowie das anschließende low-level Erstellen des eigentlichen Buildplans im WS-BPEL 2.0-Format eingegangen. Zudem wurden im Laufe der Arbeit verschiedene Java-Komponenten für die Generierung von WS-BPEL 2.0-Fragmenten entwickelt. Das Ergebnis der Arbeit war demnach ein plugin-based WS-BPEL 2.0-Planbuilder, welcher bereits über einige Plugins für die Erstellung von ersten Prototyp Build Plänen verfügte.

Die dabei entwickelte Planbuilder Komponente diente als Grundlage für die Implementierung dieser Arbeit und enthält bereits einige hilfreiche Funktionen zur Generierung von WS-BPEL 2.0-Elementen und der Verwaltung und Erstellung von WS-BPEL 2.0-Dokumenten.

Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA Bei dieser Arbeit handelt es sich um ein von Breitenbücher et al. verfasstes Paper [BBK+14], welches inhaltlich stark auf der zuvor vorgestellten Arbeit von Kálmán Képes aufbaut. Zusätzlich wird jedoch auf die Möglichkeiten der deklarativen und imperativen Provisionierung eingegangen. Deklarative Provisionierung beschreibt hierbei die Möglichkeit, dass bereits bei der Modellierung der Anwendung ein, die Provisionierung beschreibender, Plan in einem von TOSCA unterstützten Format beigelegt werden kann um darin die Reihenfolge und Art der Provisionierung zu definieren. Die, für diese Arbeit interessantere, imperative Provisionierung beschreibt im Gegensatz dazu die Modellierung einer Anwendung nach dem TOSCA-Standard ohne dabei explizit einen Plan für die Provisionierung zu erstellen, beziehungsweise beizulegen. In diesem Fall wird auf Basis der vorhandenen TOSCA-Anwendung, gepackt in einer CSAR-Datei, ein passender, ausführbarer

Buildplan für das Deployment der Anwendung generiert. Zur Verdeutlichung der Funktionsweise werden die Ideen aus Kapitel 3 in überarbeiteter Form vorgebracht.

Die Idee, dass sowohl imperative als auch deklarative Erstellung der Pläne möglich ist wurde für das Konzept dieser Arbeit übernommen.

Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing In der 2013 vorgestellten Arbeit [WWB+13] von Waizenegger et al. wird die Idee verfolgt, das Verhalten von TOSCA-Komponenten über die Zuweisung von Policies zu bestimmten Nodes zu steuern. Dabei beschäftigt sich die Arbeit hauptsächlich mit den beiden verschiedenen Wegen wie durch die definierten Policies eine höhere Sicherheit der Komponenten erreicht werden kann. Beispiele für solche Policies sind die *Region Policy* oder auch die *Encrypted Database Policy*, welche jeweils die Region der Cloud, auf welcher eine Komponente ausgebracht werden kann, einschränkt beziehungsweise dafür sorgt, dass die Datenbank einer Komponente verschlüsselt wird.

Zusätzlich zu dem markieren von bereits erstellten Node Templates, für welche Policies in der vorgestellten Arbeit genutzt wurden, werden diese TOSCA-Policies in dieser Arbeit dafür verwendet bereits vorhandene Node Templates mit Tests zu versehen.

A Framework for Component Deployment Testing In ihrer Arbeit [BPS+] stellen Bertolino et al. das Konzept und die Entwicklung eines Frameworks für das Testen des Deployments einer Softwarekomponente vor. Das entwickelte Framework ermöglicht es Tests außerhalb der Komponente zu definieren und anschließend auszuführen. Ein Test wird dabei gegen eine virtuelle Komponente programmiert, welche im Endeffekt ein Interface der zu testenden Komponente darstellt. Bei der Ausführung der so definierten Tests werden die Methodenaufrufe der virtuelle Komponente dann an die zu testende Komponente weitergeleitet. Damit dies funktionieren kann muss eine entsprechende XML-Datei erstellt werden in welcher die Eigenschaften und Schnittstellen der zu testenden Komponente festgehalten sind. Die Rückgabewerte der tatsächlichen Komponente können dann mit den erwarteten Werten verglichen werden.

Das Konzept entspricht dem Aufruf von Managementoperationen einer TOSCA-Komponente. Der Vorteil dabei ist jedoch, dass das Interface einer TOSCA-Komponente nicht explizit in einer XML-Datei definiert werden muss sondern implizit den Operations der Node Type Definition einer modellierten Anwendung entnommen werden kann. Mit einem entsprechenden Plugin wäre es demnach möglich die Funktionalität des Frameworks über den Aufruf der Managementoperationen einer TOSCA-Komponente zu erreichen. Dies hätte zusätzlich zum eingesparten Aufwand durch die nicht vorhandene Notwendigkeit der manuellen Erstellung von Beschreibungungen der Komponentenschnittstellen den Vorteil, dass die Testpläne nach ihrer Generierung der TOSCA-Anwendung hinzugefügt werden. Neben der gewonnen Portabilität kann die Adresse der Schnittstellen somit auch bei einem erneuten Ausbringen der Komponente über die Instanzdaten der jeweils genutzten TOSCA-Laufzeitumgebung bezogen werden [Eis13]. Im Gegensatz dazu muss die XML-Beschreibung der Schnittstellen bei jedem Umgebungswechsel angepasst werden.

Desweiteren ist die Definition von Infrastrukturtests, Umgebungstests sowie Compliancetests, wie sie beispielweise mit dem in dieser Arbeit entwickelten Script Test Plugin möglich sind, in

diesem Framework nicht vorgesehen. Eine Erweiterung der verschiedenen Testmöglichkeiten durch Plugins ist ebenfalls nicht vorhanden.

4 Konzept

In diesem Kapitel werden die Konzepte und Ideen, auf welchen die anschließend implementierte Komponente aufbaut, vorgestellt.

Dabei werden zunächst die Anforderungen an die zu implementierende Software präsentiert sowie die Problemstellung vorgestellt um anschließend auf den Lösungsansatz und die Verwendung des bestehenden TOSCA-Standards einzugehen.

4.1 Anforderungen

Die in dieser Arbeit entwickelte Software soll die Funktionalität anbieten einen WS-BPEL 2.0-Testplan für eine TOSCA-Anwendung zu generieren. Die TOSCA-Anwendung, für welche dieser Testplan generiert wird, muss dafür im CSAR-Format vorliegen. Der generierte Testplan soll nach dem Deployment die zuvor definierten Tests der Anwendung ausführen.

Es soll dabei möglich sein diese Tests ohne feste Kopplung an bestimmte Node Templates zu definieren. Das bedeutet, dass es nicht nötig sein darf den bestehenden Sourcecode bzw. die vorhandenen Artefakte der TOSCA-Anwendung zu verändern um eine Möglichkeit bereitzustellen diese zu testen. Desweiteren soll es möglich sein, diese Tests erst am Ende oder nach der Modellierung einer solchen Anwendung hinzuzufügen oder gar zu erstellen.

Der aus den einer Anwendung hinzugefügten Tests generierte Testplan soll im WS-BPEL 2.0-Format und somit unabhängig von der TOSCA-Laufzeitumgebung sein. Das bedeutet, dass der Testplan von jeder TOSCA-konformen Laufzeitumgebung ausgeführt werden kann.

Tests sollen in einen Test-Typ und einen davon abgeleiteten konkreten Test aufgetrennt sein, wie es auch bei anderen TOSCA-Komponenten, wie zum Beispiel Node Types und Node Templates, der Fall ist. Genau wie bei den TOSCA Node Types soll es demnach Test Type Komponenten geben, welche abstrakt gehalten werden um für ähnliche Testfälle wiederverwendbar zu sein. Ein konkreter Test soll dann, genau wie Node Templates, als Test Template lediglich die Eigenschaften, beziehungsweise Parameter, der zugehörigen Test Type Komponente definieren.

Ein Test für ein Node Template besteht demnach aus einem Test-Typ, welcher durch die Test Type Komponente repräsentiert wird, sowie einem davon abgeleiteten Test Template.

Es sollen neue Test Templates und Test Types definierbar sein, ohne dass dafür Änderungen oder Erweiterungen der Komponente, welche den Plan generiert, notwendig sind. Der erzeugte Testplan soll außerdem dem Format der bekannten Build-, Termination- und Scalingplänen folgen und daher ebenfalls im WS-BPEL 2.0-Format sein um später von jeder TOSCA-Laufzeitumgebung ausführbar zu sein.

Test-Komponente	TOSCA-Komponente
Test Type	Policy Type
Test Template	Policy Template
Test Implementation Type	Artifact Type
Test Implementation Artifact	IA

Tabelle 4.1: Mapping der Testbestandteile auf TOSCA-Komponenten

Die Tests sollen mit bestehenden TOSCA-Komponenten definierbar sein, ohne dabei die Struktur oder den Aufbau der TOSCA-Anwendung, -Komponenten und des CSARs auf spezifikationsunkonforme Weise zu verändern.

4.2 Verwendung bestehender TOSCA-Komponenten

Tests sollen mit bereits existenten TOSCA-Komponenten definierbar sein und die vorhandenen, zu testenden, Node Templates nicht verändern sondern lediglich erweitern, sodass die zu testenden Komponenten, sowie der verwendete Test Type und das Test Template später für die Testplan generierende Komponente ersichtlich und von den tatsächlichen TOSCA-Komponenten zu unterscheiden sind. Die Zuordnung der Test-Komponenten zur jeweiligen TOSCA-Komponente ist hierbei aus Tabelle 4.1 zu entnehmen.

4.2.1 Policies

Wie bereits in Kapitel 2 erwähnt, bieten TOSCA-Policies eine Möglichkeit um Node Templates weitere Eigenschaften oder Einschränkungen hinzuzufügen, welche die Node Templates vorher nicht besessen haben. beispielsweise können Nodes mit Policies wie z.B. der *No Public Access* Policy versehen werden. Die Policy im angeführten Beispiel kann dann, falls sie von der ausführenden TOSCA-Laufzeitumgebung unterstützt wird, dafür sorgen, dass Schnittstellen der zugehörigen Node nicht öffentlich erreichbar sind.

Außerdem existieren Policy Types und Policy Templates, welche genutzt werden können um die gewünschte Abstraktion in Test Types und davon abgeleitete Test Templates zu repräsentieren. Die Policy Types geben dabei, genau wie Node Types, mögliche Parameter der zugehörigen Policy Templates an. Wie auch Node Templates belegen Policy Templates anschließend die, zuvor von dem zugehörigen Policy Type definierten, Eigenschaften mit konkreten Werten. Ein Test Type, welcher durch einen Policy Type repräsentiert wird kann somit beliebig viele Verwendungen mit verschiedenen Eigenschaften in Form von Test Templates, bei welchen es sich um Policy Templates handelt, besitzen. Auf diese Weise erstellte Tests können dann, genau wie Policies in ihrer urpsprünglichen Verwendung, von Node Templates referenziert werden.

4.2.2 Implementation Artifacts

Tests bestehen jedoch in der Regel nicht ausschließlich aus den via Test Type und Test Template definierten Eigenschaften, sondern je nach Art des Tests möglicherweise auch aus Quellcode. Das bedeutet, dass dieser ebenso verfügbar gemacht werden muss wie die definierten Eigenschaften eines Tests. Bei Test Quellcode kann es sich dabei um alle Typen der definierten TOSCA Artifact Types handeln, wobei für die Konzeption dieser Arbeit exemplarisch ein Script Artifact als Artifact Type angenommen wird. Das bedeutet, dass der Testquellcode in Form von IAs vom Artifact Type Script Artifact bereitgestellt wird. Diese Artifacts müssen dem zugehörigen Policy Type zugeordnet werden können um eine spätere Ausführung der Tests mit den im Test Type, beziehungsweise Test Template, definierten Eigenschaften zu ermöglichen.

Falls ein Test Type mit einem zugehörigen IA gewählt wird, so muss dieses über den Node Type des zu testenden Node Template verknüpft werden.

Es sind jedoch auch Arten von Tests ohne eigene Implementierung in Form eines IAs denkbar, wenn ein Plugin für den entsprechenden Test Type bei der Generierung des Plans verfügbar ist, welches lediglich die Eigenschaften eines Test Templates benötigt.

4.2.3 Node Type

Damit ein Test später ausführbar gemacht werden kann, muss, wie bereits erwähnt, eine Zuordnung vom Test Implementation Artifact zum Test Type als auch dem Test Template. Da weder Policy Types noch Policy Templates derzeit ein IA referenzieren können, ist dies auch für Tests nicht möglich. Daher muss diese Verknüpfung an einem zentralen Element erfolgen. Dafür kommen, wie in Abbildung 4.1 dargestellt, nur die Node Templates infrage, da sie sowohl Test Type und Test Template als auch die für die Tests relevanten IAs über ihren Node Type referenzieren können.

Ebenfalls in Abbildung 4.1 abgebildet ist, dass die einzige Möglichkeit ein IA zu referenzieren über die Node Implementations der Node Types sowie ihren Interfaces und Operations, besteht. Es liegt daher nahe, ein neues Interface für die Ausführung von Tests zu definieren, welches als Operationen die Namen der Test Types der für diesen Node Type nutzbaren Tests zusammen mit ihrem jeweiligen Test IAs referenziert. Die Zuordnung der Operation zum PolicyType erfolgt dann über den Operationsbezeichner sowie dem Namen des Test Types. Das IA eines Test Types *CheckJavaVersion* würde, der Definition in Abbildung 4.1 nach, über eine Operation einer Node Type Implementation namens *CheckJavaVersion* verfügbar gemacht.

4.2.4 Node Template

Die Node Templates der zu testenden Komponenten werden insofern erweitert, als dass ihnen das gewünschte Test Template und der zugehörige Test Type hinzugefügt wird.

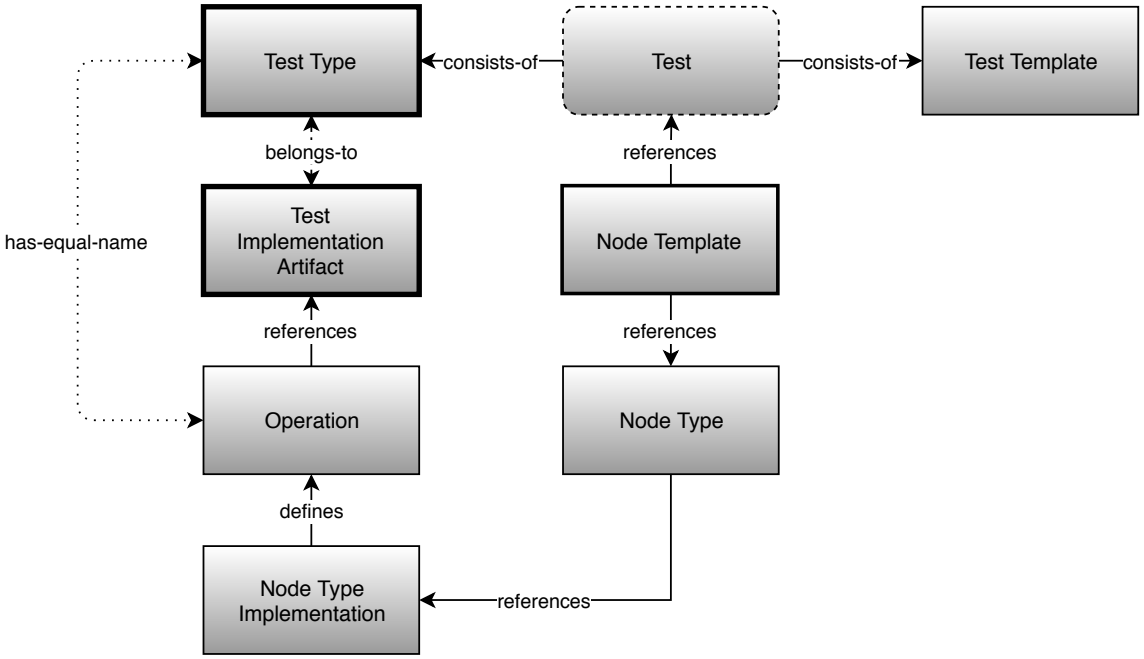

Abbildung 4.1: Verknüpfung von Test Type und Test Implementation Artifact ausgehend von einem Node Template

4.3 Erweiterbarkeit

Die in dieser Arbeit entwickelte Software und auch die Auswahl an Tests soll leicht erweiterbar sein. Im folgenden werden die Möglichkeiten der Erweiterung um einzelne Bestandteile nach zuvor vorgestelltem Konzept näher beleuchtet.

Test Types Um einen neuen Test-Typ zu definieren soll, abgesehen vom Erstellen der neuen Bestandteile, keine Veränderung der TOSCA-Modellierungsumgebung notwendig sein. Hierfür wird lediglich ein neuer Policy Type angelegt und die möglichen Eigenschaften sowie deren Typen definiert. Anschließend wird ein zugehöriges IA eines unterstützten Artifact Types angelegt, in welchem die im Test Type definierten Properties je nach Artifact Type als Funktionsparameter oder Variable benutzt werden können.

Wurde der neue Test Typ und das zugehörige Test IA angelegt können diese in Node Types, welche diesen Typ von Test anwenden können sollen, referenziert um sie dann für abgeleitete, zu testende, Node Templates zu nutzen.

Test Templates Sofern der zugehörige Test Type des zu erstellenden Test Templates bereits vorhanden ist, soll lediglich die Neuanlage des Test Templates in Form eines Policy Templates sowie die Belegung der Parameter mit einem konkreten Wert notwendig sein.

Test Implementation Types Der Test Implementation Type hängt vom Artifact Type des zum Test Type gehörenden IAs ab. Im Zuge dieser Arbeit sollen lediglich Test Types mit Test Implementation Type Script Artifact unterstützt werden. Allerdings soll die zu entwickelnde Komponente auf Plugins basieren und somit leicht erweiterbar sein. Sollten später also weitere Test Implementation Types gewünscht sein so genügt das Hinzufügen eines Plugins um Artifakte dieses Typs zu unterstützen.

Node Template Tests Um einen bestehenden Test zu einem bestehenden Node Template hinzuzufügen genügt eine Referenzierung der zugehörigen Test Definitions via Policies sowie das Hinzufügen des gewünschten Test Typen zum Test Interface der Node Type Implementation, falls noch nicht vorhanden.

4.4 Zusammenfassung

Ein mit dem Ergebnis dieser Arbeit definierter Test besteht aus einem Test Type, einem Test Template sowie einer Test Implementation. Um einen Test zu erstellen müssen demnach alle drei Komponenten existieren und im zu testenden Node Template referenziert werden.

Bei allen Test-Komponenten handelt es sich dabei, wie in Tabelle 4.1 definiert, um bestehende TOSCA-Komponenten deren Bedeutung lediglich im Kontext der Testgenerierung verändert wurde, welche jedoch auf TOSCA-konforme Art verwendet werden.

Policy Templates können Policy Types referenzieren. Die in Abbildung 4.1 abgebildete *belongs-to* Beziehung zwischen Test Type und Test Implementation Artifact kann jedoch, aufgrund von Einschränkungen der TOSCA-Spezifikation, nicht via direkter Verknüpfung von Policy Type zu IA erfolgen. Node Implementation Artifacts müssen daher über den Node Type des zu testenden Node Type Templates referenziert sein.

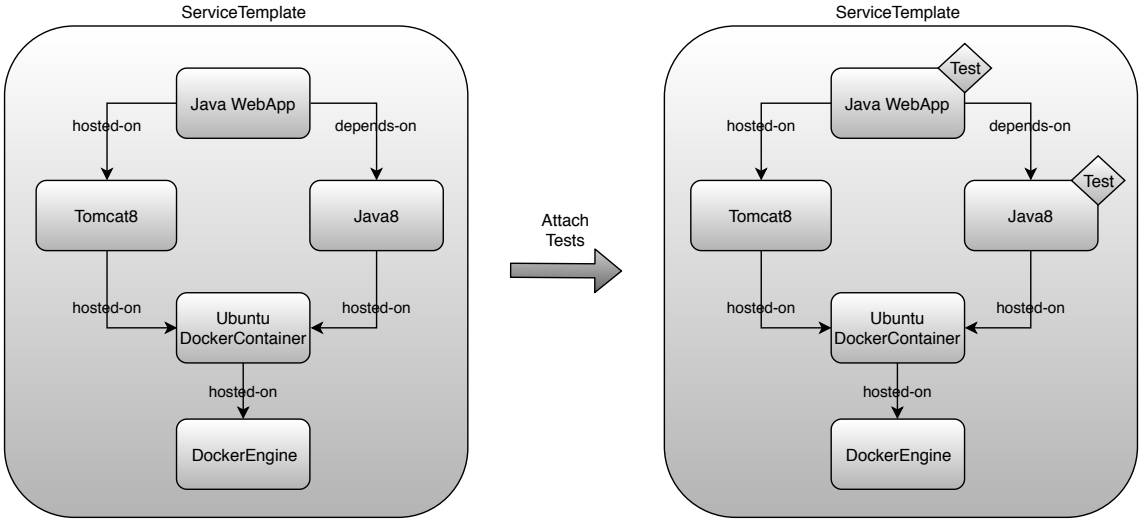

Abbildung 4.2: Für Komponenten eines bereits fertig modellierten Service Templates werden per Test Templates Tests hinzugefügt

Das konzipierte Modell erlaubt es somit, wie in Abbildung 4.2 dargestellt, einer bereits modellierten TOSCA-Anwendung neue Tests hinzuzufügen, ohne die bestehende Anwendung dabei strukturell zu verändern. Bei den in Abbildung 4.2 dargestellten Tests könnte es sich beispielsweise um einen Test *Test Shop API* vom Typ *HTTP-Test* handeln, um die Erreichbarkeit der Java WebShop-Komponente zu testen, sowie um einen *JavaVersionTest* vom Typ *EnvironmentVariableTest* für das Java8 Node Template, welcher die Infrastruktur auf die korrekte Version des installierten Java Runtime Environment (JRE) überprüft.

5 Implementierung

Im folgenden Kapitel wird die Umsetzung des zuvor präsentierten Konzepts sowie Implementierungsdetails als auch das OpenTOSCA Ecosystem und die Integrierung der zu entwickelnden Erweiterung in dieses vorgestellt.

5.1 OpenTOSCA

OpenTOSCA[1] ist ein Ecosystem für den gesamten Lifecycle einer TOSCA-Anwendung von der Modellierung über die Erstellung eines CSARs bis hin zur Instanziierung und der Verwaltung der Instanzen [BBH+13].

5.1.1 Bestandteile

OpenTOSCA ermöglicht es mit dem Winery-Tool TOSCA-Anwendungen über eine grafische Benutzerschnittstelle zu modellieren und im CSAR-Format zu exportieren. Für die so modellierten Anwendungen werden dann in der Planengine fehlende Pläne generiert um sie dann über die TOSCA-Laufzeitumgebung, genannt Container, zu instanziieren und via der generierten oder manuell erstellten Pläne zu verwalten.

Winery Winery ist eine web-basierte grafische Benutzeroberfläche für die Modellierung von TOSCA-Anwendungen und dient dazu, auf einfachem Wege neue TOSCA Bestandteile wie zum Beispiel Node Types oder Policy Types anzulegen und basierend auf den angelegten Bestandteilen neue Topologien beziehungsweise ganze TOSCA-Anwendungen nach dem erwähnten Baukastenprinzip zu erstellen.

Die so modellierten Anwendungen sind valide unter dem TOSCA-Standard und können im CSAR-Format exportiert werden. Es ist somit möglich per Winery eine TOSCA-Anwendung zu modellieren, ohne dass eine XML-Datei manuell erstellt werden muss.

Um die genutzten TOSCA-Komponenten als Tests zu erkennen und von den Elementen zu unterscheiden, welche durch diese Komponenten eigentlich dargestellt werden, werden die Test-Komponenten unter anderen Namespaces definiert. So werden Policy Types, welche Test Types beschreiben, nicht wie sonst für Policy Types üblich unter dem Namespace „http://opentosca.org/policytypes“ sondern stattdessen unter der Erweiterung „http://opentosca.org/policytypes/tests“ definiert.

[1] http://www.iaas.uni-stuttgart.de/OpenTOSCA/

Container Der OpenTOSCA Container enthält unter anderem die TOSCA-Laufzeitumgebung und bietet somit über seine Programmierschnittstelle (API) die Funktionalität an um eine TOSCA-Anwendung, welche als CSAR gepackt ist, zu instanziieren und die erstellten Instanzen zu verwalten.

Wie bereits erwähnt, sind CSARs in sich geschlossen und erlauben dem Container somit die Provisionierung der modellierten Anwendung ohne zusätzlich weitere Informationen zu benötigen. Die Ausführung der Managementoperationen zur Instanziierung bzw Terminierung einer Anwendung erfolgt dabei mittels WS-BPEL 2.0-Plänen, welche den genauen Ablauf einer Managementoperation durch die Ausführung der Operationen von Nodes Templates in einer festgelegten Reihenfolge definiert.

Ein zuvor erstellter Plan kann einer TOSCA-Anwendung bereits bei der Modellierung imperativ in den Plans-Ordner des CSARs hinzugefügt werden [BBK+14] und wird vom Container für die Realisierung der entsprechenden Managementoperation genutzt indem der Plan beispielsweise bei der Instanziierung der Anwendung in eine WS-BPEL 2.0-Engine ausgebracht und ausgeführt wird.

Zusätzlich zur imperativen Methode, bei welcher während der Modellierung der TOSCA-Anwendung manuell WS-BPEL 2.0-Pläne hinzugefügt werden, ist es möglich keine Pläne selbst zu erstellen und diese Aufgabe somit deklarativ dem OpenTOSCA Planbuilder zu überlassen [BBK+14].

Wird ein CSAR einer TOSCA-Anwendung in den Container geladen, ohne dass dieses bereits alle benötigten Pläne enthält übernimmt der Planbuilder die Aufgabe der Planerstellung. Derzeit in OpenTOSCA existierende Planarten sind Build-, Scale- und Terminationplan, welche jeweils zum Initialisieren, Skalieren beziehungsweise Terminieren der Instanz einer TOSCA-Anwendung dienen.

Der Planbuilder berücksichtigt bei der Generierung der Pläne unter anderem die Struktur des Topology Templates sowie die verwendeten Node Types und greift dabei auf entsprechende Plugins zurück.

In einem ersten Schritt wird dabei, wie in Abbildung 5.1 dargestellt, ein Directed Acyclic Graph (DAG) aus dem Topology Template erstellt. Dabei wird die Reihenfolge der auszuführenden Operationen sowie der zu instanziierenden Komponenten von der Topologie abgeleitet. Die für die Provisionierung der TOSCA-Anwendung relevanten Operationen eines Node Templates werden dabei vor den Operationen der weiteren, auf ihnen gehosteten (*hosted-on*-Relationship) und auf sie angewiesenen (*depends-on*-Relationship) Node Templates ausgeführt [Kép13]. Dass das sinnvoll ist, lässt sich am Beispiel einer Java-Anwendung veranschaulichen, welche auf einer Ubuntu-VM gehostet wird. In diesem Fall ist es nur logisch und sinnvoll, dass die VM funktionsfähig instanziiert sein muss bevor Java beziehungsweise die Anwendung darauf installiert wird.

5.1.2 Integration in das OpenTOSCA-Ecosystem

Die entwickelte Komponente erweitert den OpenTOSCA Planbuilder. Der Planbuilder ist eine in Java implementierte Komponente welche bereits WS-BPEL 2.0-Pläne für andere Plantypen generiert und daher bereits einige hilfreiche Schnittstellen und Klassen anbietet um ein WS-BPEL 2.0-Dokument über deren Methoden zu erstellen und zu verändern.

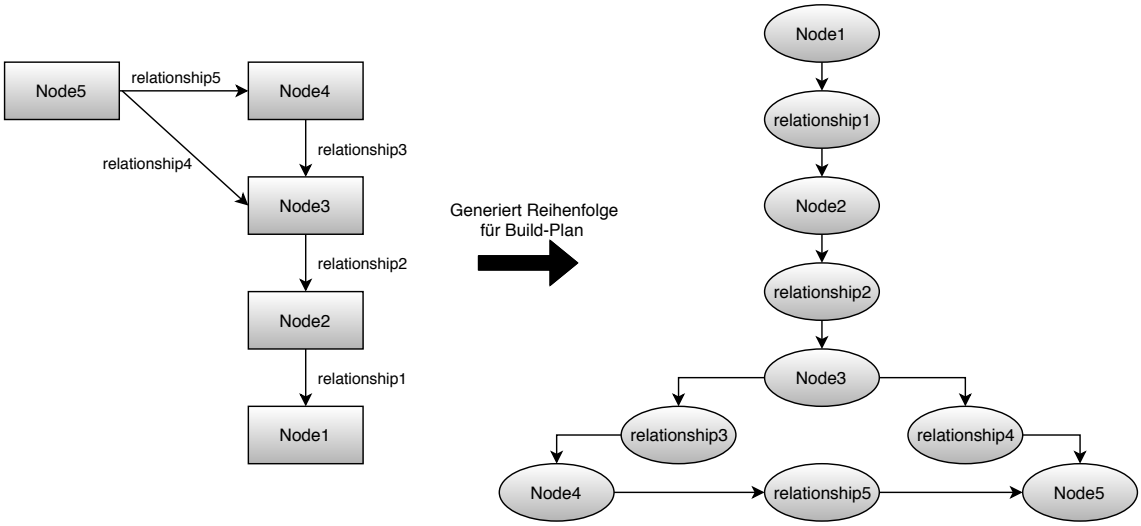

Abbildung 5.1: Ableitung der Reihenfolge für den BuildPlan auf Grundlage eines Topology Templates [Kép13]

Der Planbuilder ist derzeit, wie in Abbildung 5.2 dargestellt, in den OpenTOSCA Container integriert und reagiert beim Hochladen neuer CSARs um fehlende Pläne zu generieren. Grundsätzlich ist der Planbuilder jedoch eine alleinstehende Komponente, welche in der Theorie auch ohne den Container benutzt werden könnte um Pläne für eine beliebige TOSCA-Anwendung zu generieren um sie dieser anschließend hinzuzufügen. Die die Integration in den Container hat jedoch den Vorteil, dass das, aus dem CSAR generierte Data Object Model (DOM) des Containers benutzt werden kann. Bei diesem DOM handelt es sich um die Repräsentation des CSARs und seiner Bestandteile als Java Objekte. Weiterhin besitzt der Planbuilder bereits einige Plugins an welchen sich die Implementierung des entwickelten Test Plugins strukturell orientiert. Zusätzlich werden vom Planbuilder bereits WSDL-Dateien für erstellte WS-BPEL 2.0-Pläne generiert.

5.2 Generierung eines Testplans auf Grundlage eines CSARs

Nachdem ein CSAR erfolgreich in den OpenTOSCA Container geladen wurde, werden vom Planbuilder automatisch die fehlenden WS-BPEL 2.0-Pläne erstellt.

Die im Klassendiagramm in Abbildung 5.3 dargestellten Klassen übernehmen bei der Generierung des Plans dabei Aufgaben auf verschiedenen Abstraktionsebenen. Die oberste Klasse in der Hierarchie ist *AbstractPlanBuilder* und referenziert alle im Planbuilder vorhandenen Plugins. Diese Komponente wird von allen plangenerierenden Klassen des Planbuilders als Vaterklasse genutzt und wurde um die *getTestPlugin* erweitert, welche die Suche nach passenden Test Plugins ermöglicht. Die *AbstractTestPlanBuilder*-Klasse enthält die Funktionalität um einen DAG aus den zu testenden Node Templates zu einem Plan hinzufügt, die *BPELTestProcessBuilder*-Klasse generiert letztendlich den fertigen Testplan auf Grundlage des DAGs und durch Ausführung der Test Plugins.

Das folgende Kapitel stellt die Algorithmen vor, welche in den implementierten Klassen zur Ableitung eines WS-BPEL 2.0 Test Plans aus einem Service Template umgesetzt wurden.

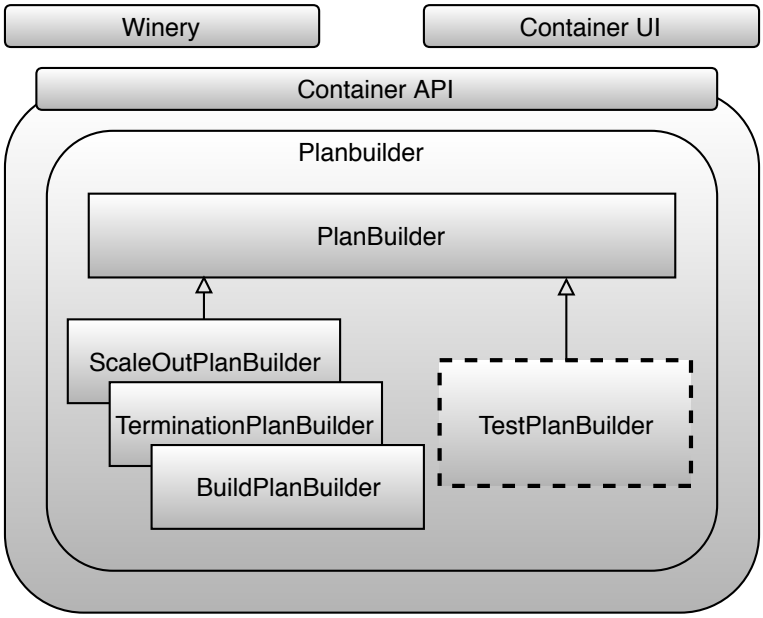

Abbildung 5.2: Integration der zu entwickelnden Testplanbuilder Komponente in den vereinfacht dargestellten OpenTOSCA Container

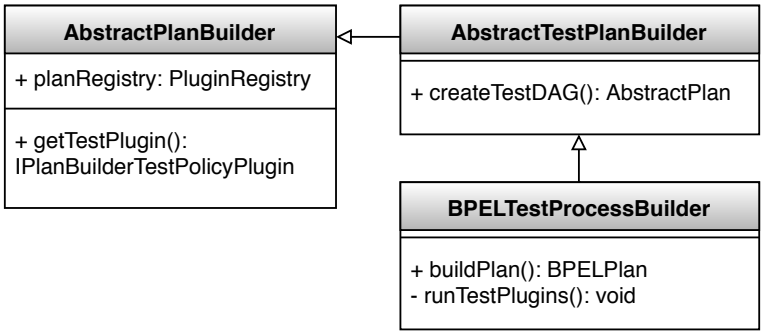

Abbildung 5.3: Vereinfachtes UML-Klassendiagramm der implementierten Komponente

5.2.1 Abfolge der Umwandlungsschritte

Der in Algorithmus 5.1 abgebildete Algorithmus beschreibt die Reihenfolge und die grob abstrahierten Operationen welche ausgeführt werden um das gewünschte Ergebnis zu erhalten.

Zunächst wird über den im folgenden Abschnitt 5.2.2 beschriebenen Algorithmus, welcher in der *createTestDAG*-Methode der *AbstractTestPlanBuilder*-Klasse implementiert wurde, eine Reihenfolge bestimmt in welcher die Tests nach dem Deployment der Anwendung ausgeführt werden sollen. Aus der daraus resultierenden Reihenfolge wird dann in der *BPELTestProcessBuilder*-Klasse, wie in Abschnitt 5.2.3 dargestellt, ein erstes WS-BPEL 2.0-Skelett erstellt. Abschließend wird dieses Skelett durch die Vervollständigung der jeweils passenden Plugins zu einem voll funktionsfähigen WS-BPEL 2.0-Plan vervollständigt. Dieser letzte, finalisierende Schritt wird in Abschnitt 5.2.4 genauer beschrieben.

Algorithmus 5.1 High-level Algorithmus für die Erstellung eines WS-BPEL 2.0 Test Plans ausgehend von einem Service Template mit Tests

procedure GENERATE TESTPLAN
serviceTemplate ← ServiceTemplate with defined Tests to generate the Test Plan for
testDAG ← $computeTestingOrder$(*serviceTemplate*)
bpelSkeleton ← $generateBPELSkeleton$(*testDAG*)
bpelPlan ← $executeTestPlugins$(*bpelSkeleton*, *serviceTemplate*)
return *bpelPlan*
end procedure

5.2.2 Bestimmung der Test-Ausführungsreihenfolge

Der Algorithmus zu Erstellung der Test-Ausführungsreiheinfolge ist ein Partial Planning Algorithmus [Wel94], welcher bereits in ähnlicher Form im OpenTOSCA Planbuilder für Buildpläne [Kép13] eingesetzt wird. Der Algorithmus wurde jedoch um einige Details verändert beziehungsweise erweitert, um möglichst genau den Use-Case für die Testplan Generierung umzusetzen.

Um eine sinnvolle Reihenfolge für die Testausführung zu finden wird in Algorithmus 5.2 mithilfe einer Tiefensuche ein DAG aus den Node Templates, welche einen zugeordneten Test besitzen, erstellt. Dieser DAG repräsentiert dann die Ausführungsreihenfolge der Tests. Folgend wird dargelegt, wie ein die Ausführungsreiheinfolge bestimmender Test-DAG aus einem Service Template erstellt wird.

Verbinden von Test Node Templates

Zuerst werden alle Node Templates mit einem definierten Test aus dem Service Template extrahiert um anschließend über die entstandene Liste zu iterieren und entlang der Relationship-Richtung per Tiefensuche alle nächstgelegenen und mit ausschließlicher Benutzung ausgehender Relationships, also in Richtung Infrastruktur, zu finden. Anschließend wird jedes zu testende Node Template mit jedem auf diesem Wege gefundenen anderen zu testenden Node Template verknüpft, wobei die hergestellte Verbindung umgekehrt zur Richtung der im Service Template, eventuell über weitere Node Templates, gegebenen Verbindung angelegt wird. Der Grund dafür ist, dass die Node Templates, welche näher an der Infrastruktur liegen, vor den darauf aufbauenden Node Templates getestet werden sollen, da der Test eines Node Templates bei einem bereits fehlerhaft getesteteten unterliegenden Node Template sinnlos beziehungsweise nicht aussagekräftig ist. Daher muss ein Node Template, welche in der Topologie per Relationship auf einem anderen Node Template aufbaut im DAG für die Testreihenfolge hinter diesem stehen.

Das in Abschnitt 5.2.2 abgebildete Service Template resultiert nach dem Durchlaufen des beschriebenen Algorithmus demnach in den zwei abgebildeten Connections. *connection_1* entstammt der umgedrehten *relationship1-2* aus dem Service Template und *connection_2* kommt über *relationship1-5* sowie *relationship5-6* zustande. *Node5* wird dabei nicht zu den Connections hinzugefügt, da für diese Node keine Tests definiert sind.

Algorithmus 5.2 Verbinden von Test Node Templates

procedure GENERATE TEST DAG
  testNodes ← All Node Templates with Tests assigned
  for *testNode* in *testNodes* **do**
    closestTestNodes ← deepSearchAllClosestTestNodes(*testNode*)
    connections ← add connection from each Node in *closestTestNodes* to *testNode*
  end for
  connections ← remove duplicates
  return *connections*
end procedure

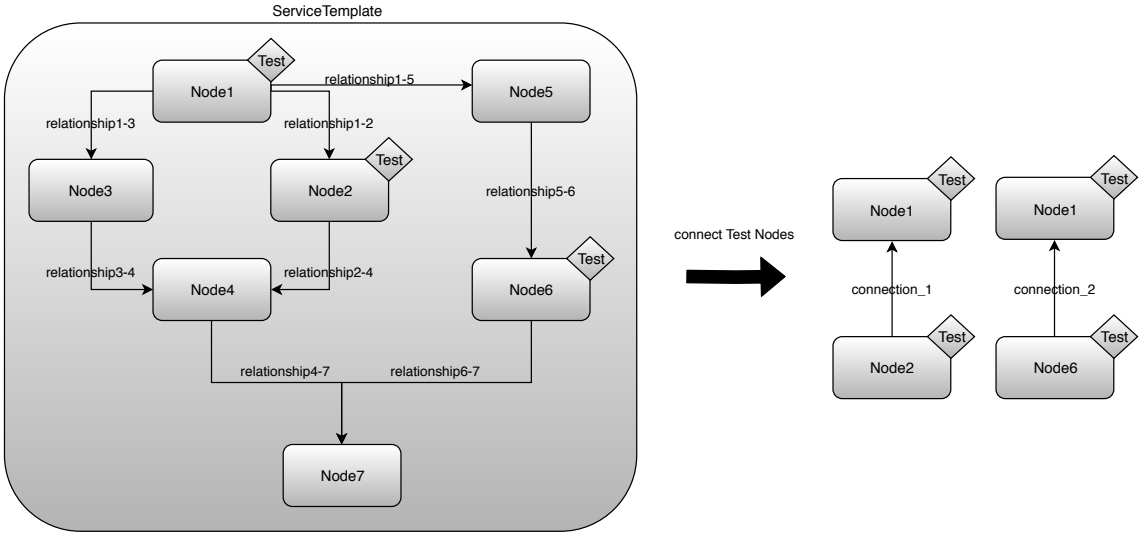

Abbildung 5.4: Verbinden der Test Node Templates eines Service Templates

Verschmelzen der Test-Connections

Nachdem alle Connections zwischen Test Node Templates gefunden und korrekt hergestellt wurden, müssen diese zu einem DAG verschmolzen werden. Conntections lassen sich dabei durch folgende Gleichungen beschreiben:

- $N = \{\text{Node Templates with tests assigned}\}$
- $Connection = \{(a, b) | a, b \in N\}$

Zwei Connections A und B werden nach folgender Regel verschmolzen:

- $A = Connection(a, b)$
- $B = Connection(c, d)$
- $$Verschmelzung(A, B) = \begin{cases} \{(X, Y) | X = (a \cup c), Y = b = d\} & \text{if } b = d \\ \{(X, Y) | X = a = c, Y = (b \cup d)\} & \text{if } a = c \\ false & \text{else} \end{cases}$$

Das Resultat ist also ein DAG, welcher die möglichen Testreihenfolgen einschränkt, jedoch nicht eindeutig vorgibt. Im Beispiel von Abschnitt 5.2.2 werden demnach nach zuvor definierter Regel *connection1* mit *connection2* verschmolzen. Der entstehende DAG gibt dann vor, dass sowohl Node2 als auch Node6 vor Node1 getestestet wird, welche der beiden Nodes Node2 und Node6 zuerst getestet wird, wird hierbei nicht definiert.

Sieht man sich erneut das Service Template in Abschnitt 5.2.2 an, macht der entstandene DAG, beziehungsweise die dadurch bestimmte Testreihenfolge, durchaus Sinn, da die unterliegenden Node Templates Node2 und Node6 vor der von diesen Node Templates abhängigen Node1 getestet werden.

Auf diese Weise entstehen jedoch lediglich Sub-DAGs mit einer Tiefe von 1, da nur die jeweils nächsten Test Node Templates miteinander verknüpft werden. Um den Gesamt-DAG zu erzeugen werden diese Sub-DAGs anschließend miteinander verknüft indem sie an der Stelle, an welcher identische Node Templates auftreten, vereint werden. Angenommen Node4 der Topologie aus Abschnitt 5.2.2 wird ebenfalls ein Test hinzugefügt, dann würde zusätzlich zu den in Abschnitt 5.2.2 dargestellten Verbindungen eine Verbindung von Node4 nach Node2 existieren. Die so zusätzlich entstehende Verbindung würde nicht mit den anderen beiden gemergt werden, da Node2 in keiner der Verbingungen die Node mit der eingehenden Relation ist und Node4 nicht vorkommt. Anhand des letzten Schrittes würde diese Verbindung jedoch mit dem Ergebis aus Abschnitt 5.2.2 verknüpft, da Node2 in beiden existiert, und Node4 würde damit im finalen DAG noch vor Node2 stehen.

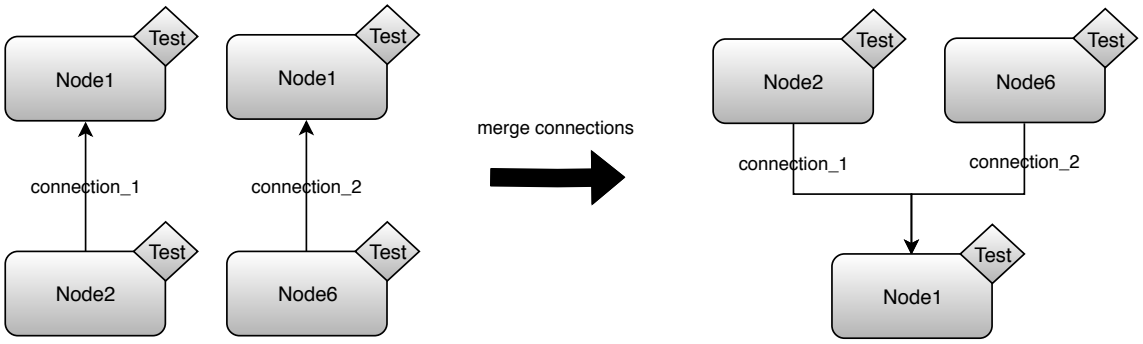

Abbildung 5.5: Verschmelzen der Test Node Connections

5.2.3 Erstellung des WS-BPEL 2.0-Skeletts

Auf Grundlage des zuvor erstellten Test-DAG wird anschließend das WS-BPEL 2.0-Skelett erstellt. Das nach diesem Schritt erstellte Dokument beihnhaltet noch keine Logik zum Testen der Node Templates sondern lediglich die Struktur beziehungsweise die Reihenfolge der Ausführung dieser.

Bei dem WS-BPEL 2.0-Skelett handelt es sich damit zunächst lediglich um <*scope*>s. Diese <*scope*>s enthalten leere *sequence*s, welche anschließend von den Plugins mit der konkreten Logik zum Testen der einzelnen Node Templates befüllt werden. Diese <*scope*>s sind bereits via <*link*>s verknüpft. Diese <*link*>s bezwecken eine Synchronisation parallel ablaufender Tasks [OAS07]

Listing 5.1 Gekürztes WS-BPEL 2.0-Skelett mit leeren *<sequence>*s und gesetzten *<link>*s abgeleitet von dem im vorherigen Beispiel erzeugten DAG

```
<process>
      <flow>
            <links>
                  <link name="link2-1"/>
                  <link name="link6-1"/>
            </links>
            <scope name="TestNode2">
                  <sources>
                        <source linkName="link2-1"/>
                  </sources>
                  <sequence>
                  </sequence>
            </scope>

            <scope name="TestNode6">
                  <sources>
                        <source linkName="link6-1"/>
                  </sources>
                  <sequence>
                  </sequence>
            </scope>

            <scope name="TestNode1">
                  <targets>
                        <target linkName="link2-1"/>
                        <target linkName="link6-1"/>
                        <joinCondition>$link2-1 and $link6-1</joinCondition>
                  </targets>
                  <sequence>
                  </sequence>
            </scope>
      </flow>
</process>
```

und werden im Falle der Testpläne dafür genutzt um die Testausführung nach Reihenfolge des DAGs zu realisieren.

In Listing 5.1 ist das aus dem resultierenden DAG in Abschnitt 5.2.2 abgeleitete WS-BPEL 2.0-Skelett vereinfacht abgebildet.

5.2.4 Plugins

Um validen WS-BPEL 2.0 Code zur Ausführung der zuvor definierten Tests zu generieren wird, je nach Test Typ, ein anderes Plugin benötigt, welches den Test Typ unterstützt.

Die Test Plugins basieren alle auf dem *IPlanBuilderTestPolicyPlugin* Interface, welches die Methoden *canHandle* und *handle* vorgibt.

Bereits bei anderen Plugins des Containers und des Planbuilders wird der Open Services Gateway initiative (OSGi)-Standard verwendet. Daher werden auch Test Plugins in Java und als OSGi-Bundles implementiert um global in den Planbuilder eingebunden zu werden. Alle so registrierten

Plugins können dann über die *PluginRegistry*-Klasse mithilfe der *getTestPlugins*-Methode über ihren registrierten Bezeichner gefunden werden. Bei allen Planbuilder Plugins handelt es sich bei diesem Bezeichner um das jeweils implementierte Interface.

Auf diesem Wege werden alle verfügbaren Test Plugins nach der Erstellung des DAG und des WS-BPEL 2.0-Skeletts entdeckt und über die *canHandle*-Methode für jeden Test jedes Node Templates das Plugin gesucht, welches diesen Test Typ mit diesem Node Template, beziehungsweise seiner Infrastruktur, unterstützt.

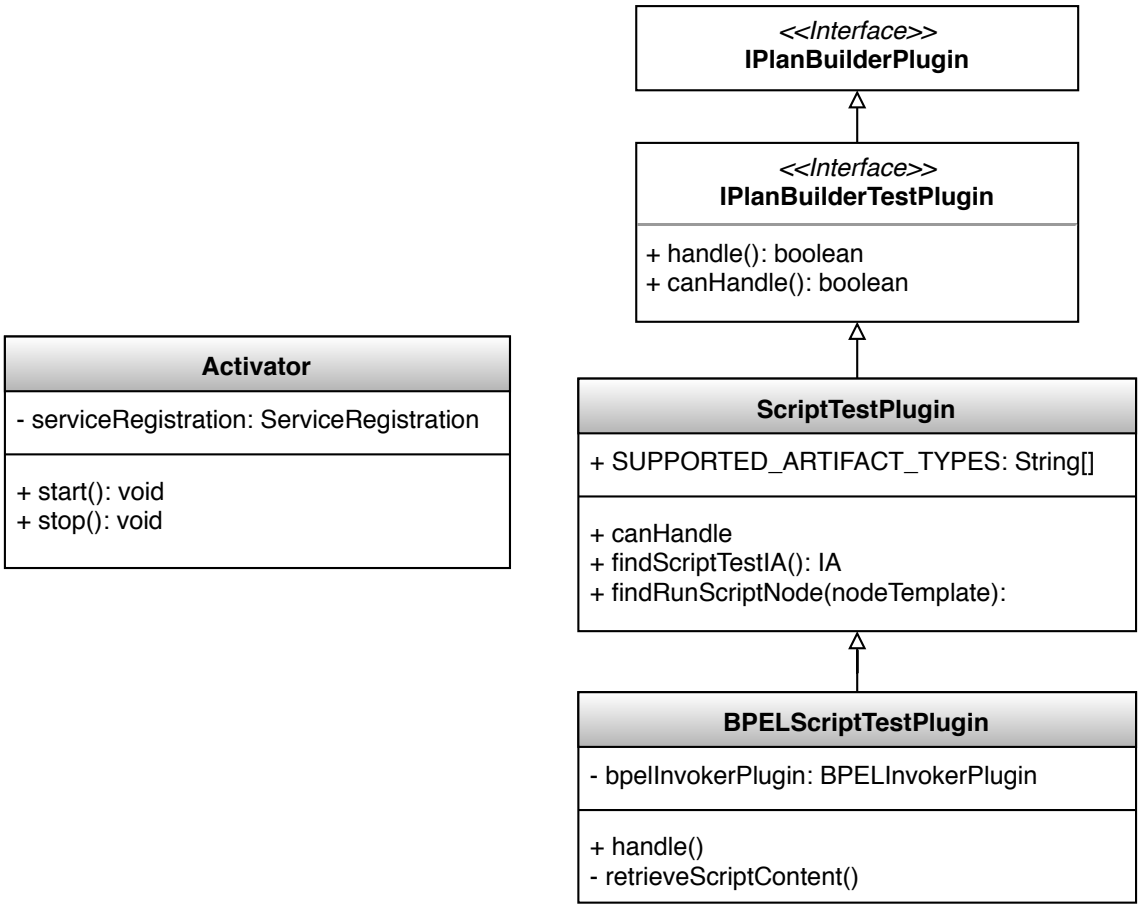

Abbildung 5.6: Vereinfachtes UML-Klassendiagram des Script Test Plugins

Script Test Plugin

Als Proof of Concept (PoC) für diese Arbeit wurde prototypisch ein erstes *Script Test Plugin* erstellt, welches WS-BPEL 2.0-Code für Tests mit dem Test Implementation Type Script Artifact generiert.

Die zuvor deployten Node Templates sollen für einen solchen Test nicht verändert werden. Deswegen wird für Script Tests die *runScript* Operation des nächsten unterliegenden Node Templates genutzt, falls das zu testende Node Template diese selbst nicht anbietet. Der Grund

dafür ist, dass für diese Art von Test ein Node Template benötigt wird, auf welchem Skripte ausgeführt werden können.

In der *canHandle*-Methode des Plugins wird daher überprüft, ob ein solches Node Template in Richtung Infrastruktur vorhanden ist, ob das Test Script IA im CSAR enthalten ist sowie ob es sich bei diesem IA um ein Script Artifact handelt. Falls alle diese Voraussetzungen erfüllt sind, kann das Plugin die Generierung des WS-BPEL 2.0-Codes für diesen Test übernehmen.

Wurde ein passendes unterliegendes Node Template gefunden, so wird der Inhalt des zugehörigen Script Artifacts entnommen und die Variablen, deren Bezeichner mit einem Parameterbezeichner des Testtyps übereinstimmt, durch den, im Test Template definierten, Wert ersetzt. Anschließend wird WS-BPEL 2.0-Code erstellt um das resultierende Skript über die *runScript*-Operation auf dem Node Template auszuführen.

Das in Abschnitt 5.2.4 dargestellt Klassendiagramm stellt die Struktur des erstellten Plugins dar. Die unabhängige *Activator* Klasse ist hierbei ein Konstrukt des OSGi-Frameworks und registriert das Plugin im *BundleContext* der Anwendung. Das *IPlanBuilderPlugin*-Interface muss von jedem Plugin Interface des Planbuilders implementiert werden und wird daher auch von dem in dieser Arbeit erstellten *IPlanBuilderTestPlugin* implementiert, welches wiederum von jedem erstellten Test Plugin implementiert werden muss. Der funktionale Teil eins Test Plugins wird, wie bei anderen bestehenden Plugins in zwei Klassen aufgeteilt. Die *ScriptTestPlugin*-Klasse implementiert die *canHandle*-Methode. Die *BPELScriptTestPlugin*-Klasse hingegen implementiert mit der *handle*-Methode die Logik welche dann, auf Grundlage des übergebenen Test Templates, ein WS-BPEL 2.0-Fragment generiert um den jeweiligen Test mit den übergebenen Parametern auszuführen. Dies wurde unter anderem durch die Verwendung des *BPELInvokerPlugin*s umgesetzt, welches bereits im Planbuilder vorhanden ist und WS-BPEL 2.0-Code für den Aufruf einer Operation, in diesem Fall *runScript*, eines Node Templates erstellen kann. Für die spätere Kommunikation mit dem Node Template wird zudem eine WSDL-Datei generiert.

Erstellen weiterer Plugins

Um weitere Plugins zu erstellen muss lediglich ein OSGi-Bundle angelegt werden welches eine Klasse beinhaltet die, analog zur *ScriptTestPlugin*-Klasse in Abschnitt 5.2.4, das *IPlanBuilderTestPolicyPlugin*-Interface implementiert. Das bedeutet, die *canHandle*- und *handle*-Methoden müssen umgesetzt werden. In der *canHandle*-Methode wird dann überprüft, ob es sich beim Test des übergebenen Node Templates um einen vom entwickelten Plugin unterstützten Test Type oder Test Implementation Type handelt und dementsprechend ein boolscher Wert zurückgegeben. In der *handle*-Methode wird die Logik implementiert, welche für das Node Template dann die im WS-BPEL 2.0-Plan fehlenden Logik zum Ausführen des Tests generiert.

Denkbare weitere Plugins wären beispielsweise ein Plugin für Tests vom Test Implementation Type WAR, deren Implementierung also aus einem WAR-Archiv besteht. Außerdem können Plugins für Tests ohne IA angelegt werden, welche die *canHandle*-Methode ausschließlich auf Grundlage des Test Types implementieren und bereits vorgefertigte WS-BPEL 2.0-Fragmente enthalten um diese auf Basis der im jeweiligen Test Template definierten Properties zu vervollständigen und dann wie alle anderen Plugins an geeigneter Stelle in den WS-BPEL 2.0-Plan einfügen.

Auch wäre es denkbar Plugins zu erstellen, welche die *canHandle*-Methode auf Basis des Test Implementation Types und des Test Types auswerten. Somit könnten beispielsweise Plugins erstellt werden, welche zusätzlich zum Setzen der Variablen im übergebenen Script IA über weitere Eigenschaften verfügen, welche über zusätzliche Properties des zugehörigen Test Types gesetzt werden können. Die Plugins, welche lediglich auf den Test Implementation Type überprüfen, wie zum Beispiel das in dieser Arbeit erstellte Script Test Plugin, würden dann als „Fallback“ dienen falls kein passenderes Plugin, welches sowohl den Test Implementation Type als auch den Test Type unterstützt, gefunden werden kann.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein neues Konzept zum Testen von TOSCA-Anwendungen vorgestellt. Anstatt die Tests fest in eine TOSCA-Laufzeitumgebung einzubringen wurde eine Komponente entwickelt, welche auf Basis einer modellierten TOSCA-Anwendung einen Testplan im WS-BPEL 2.0-Format erstellt. Der Testplan ist dabei, im Gegensatz zur alternativen Entwicklung einer Testkomponente in einer TOSCA-Laufzeitumgebung, unabhängig von einer bestimmten Laufzeitumgebung. Somit kann der Plan von jeder TOSCA-Laufzeitumgebung, welche einen BPS zur Ausführung von WS-BPEL 2.0-Plänen anbietet, genutzt werden. Bei WS-BPEL 2.0 handelt es sich um einen der Standards, welche in der TOSCA-Spezifikation zur Definition von Plänen beschrieben werden, daher ist anzunehmen, dass jede TOSCA-Laufzeitumgebung diesen auch Unterstützt und die generierten Pläne somit von jeder beliebigen TOSCA-Laufzeitumgebung ausgeführt werden können.

Die Testpläne werden, wie bereits erwähnt, aus einer bereits modellierten TOSCA-Anwendung abgeleitet, welche als CSAR vorliegt. Anhand der Topologie der Anwendung wird dann die Reihenfolge der Tests festgelegt und Plugins übernehmen je nach Art des zu testenden Node Templates sowie des dazugehörigen Tests die Generierung der WS-BPEL 2.0-Logik zum ausführen des Tests.

Teil der Arbeit war neben der Implementierung des Testplanbuilders als Teil des OpenTOSCA Ecosystems die Erstellung eines ersten exemplarischen Plugins, mit welchem als Test definierte Skripte auf Node Templates ausgeführt werden können. Ein Anwendungsfall hierfür ist beispielsweise die Überprüfung der Umgebungsvariablen eines Node Templates auf die gewünschten Werte.

Ausblick

Die auf Plugins basierende Konstruktion des Planbuilders erlaubt es diesen um weitere Features zu erweitern. Auch die in dieser Arbeit entwickelte Testplanbuilder Komponente folgt diesem Prinzip. Bisher existiert nur das Script Test Plugin mit welchem als Script definierte Tests ausgeführt werden können, weitere sinnvolle Plugins wären beispielsweise ein WAR Test Plugin und Plugins welche die Testlogik bereits besitzen und über die Test Templates der zu testenden Node Templates lediglich vervollständigen.

Mit dem Script Test Plugin ist momentan ausschließlich die Ausführung von lokalen Tests möglich. Das bedeutet, dass die Tests immer auf dem Node Template selbst, beziehungsweise dem in Richtung Infrastruktur nächsten Node Template welches Skripte ausführen kann, durchgeführt werden. Mit dem WAR Test Plugin könnte es dann beispielsweise möglich werden einen Webservice nur für Testzwecke zu starten, Schnittstellentests von diesem durchzuführen und ihn anschließend wieder zu beenden.

Literaturverzeichnis

[Amt16] Amtsblatt der Europäischen Union. „Verordnung des Europäischen Parlaments und des Rates zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten, zum freien Datenverkehr und zur Aufhebung der Richtlinie 95/46/EG (Datenschutz-Grundverordnung) vom 27. April 2016 zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten, zum freien Datenverkehr und zur Aufhebung der Richtlinie 95/46/EG (Datenschutz-Grundverordnung)“. In: 2016 (zitiert auf S. 17).

[BBF+18] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann. „A Systematic Review of Cloud Modeling Languages“. In: *ACM Computing Surveys* 51.1 (Feb. 2018), S. 1–38. DOI: 10.1145/3150227 (zitiert auf S. 17).

[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: *11th International Conference on Service-Oriented Computing*. LNCS. Springer, 2013 (zitiert auf S. 39).

[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 of the IEEE International Conference on Cloud Engineering (IEEE IC2E 2014)*. IEEE Computer Society, März 2014, S. 87–96. DOI: DOI10.1109/IC2E.2014.56 (zitiert auf S. 29, 40).

[BPS+] A. Bertolino, A. Polini, I. di Scienza, e Tecnologie, dell'Informazione, " Faedo, " „A Framework for Component Deployment Testing“. In: IEEE (zitiert auf S. 30).

[CCMW01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. *Web Services Description Language (WSDL) 1.1*. 2001 (zitiert auf S. 25).

[CMRW07] R. Chinnici, J.-J. Moreau, A. Ryman, S. Weerawarana. *Web Services Description Language (WSDL) Version 2.0*. 2007 (zitiert auf S. 25).

[Dat18] Datenschutz.org. *Verstoß gegen den Datenschutz: Hohe Bußgelder möglich!* 2018. URL: https://www.datenschutz.org/verstoss/ (zitiert auf S. 17).

[Eis13] M. Eisele. „Verwaltung von Instanzdaten eines TOSCA Cloud Services“. IAAS Universität Stuttgart, 2013 (zitiert auf S. 30).

[KBF+17] K. Képes, U. Breitenbücher, M. P. Fischer, F. Leymann, M. Zimmermann. „Policy-Aware Provisioning Plan Generation for TOSCA-based Applications“. In: *Proceedings of The Eleventh International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2017), 10-14 September 2017, Rome, Italy*. Xpert Publishing Services, Sep. 2017, S. 142–149. ISBN: 978-1-61208-582-1 (zitiert auf S. 22).

[Kép13] K. Képes. „Konzept und Implementierung einer Java-Komponente zur Generierung von WS-BPEL 2.0 BuildPlans für OpenTOSCA“. IAAS Universität Stuttgart, 2013 (zitiert auf S. 29, 40, 41, 43).

[LM13] P. Lipton, S. Moser. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. Techn. Ber. Committee: OASIS, 2013 (zitiert auf S. 22).

[LMPS13] P. Lipton, S. Moser, D. Palma, T. Spatzier. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Committee: 2013 (zitiert auf S. 19, 20, 24).

[OAS07] OASIS. *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*. Organization for the Advancement of Structured Information Standards (OASIS). 2007 (zitiert auf S. 27, 45).

[OGP03] D. Oppenheimer, A. Ganapathi, D. A. Patterson. „Why do Internet services fail and what can be done about it?“ In: *USITS*. 2003 (zitiert auf S. 17).

[Pet11] D. Petcu. „Portability and Interoperability Between Clouds: Challenges and Case Study“. In: *Proceedings of the 4th European Conference on Towards a Service-based Internet*. ServiceWave'11. Poznan, Poland: Springer-Verlag, 2011, S. 62–74. ISBN: 978-3-642-24754-5. URL: http://dl.acm.org/citation.cfm?id=2050869.2050876 (zitiert auf S. 17).

[Wel94] D. S. Weldt. „An Introduction to Least Commitment Planning“. In: (1994) (zitiert auf S. 43).

[WWB+13] T. Waizenegger, M. Wieland, T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, B. Mitschang, A. Nowak, S. Wagner. „Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing“. In: *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*. Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-41029-1. DOI: 10.1007/978-3-642-41030-7_26 (zitiert auf S. 30).

Alle URLs wurden zuletzt am 14. 06. 2018 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