Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 43

Konzept und Implementierung einer Java-Komponente zur Generierung von WS-BPEL 2.0 BuildPlans für OpenTOSCA

Kálmán Képes

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Frank Leymann
Betreuer/in:	Dipl.-Inf. Uwe Breitenbücher
Beginn am:	2013-01-02
Beendet am:	2013-07-04
CR-Nummer:	I.7.2, H.4.1, D.3.2

Kurzfassung

Mit steigender Wichtigkeit und Verwendung von Cloud-basierten Lösungen für eine immer breitere Menge von Aufgaben und Einsatzmöglichkeiten, steigt die Komplexität des Aufsetztens, Verwaltens und Herunterfahrens von Cloud-Anwendungen. Um diesem Umstand entgegenzuwirken, wurde von OASIS die Standardisierung, in Form einer Beschreibungssprache, für das Beschreiben von Cloud-Anwendungen in die Wege geleitet. Die Anforderungen an solche Beschreibungssprachen ist unter anderem die Möglichkeit, die Topologie einer Cloud-Anwendung zu modellieren, darunter die Komponenten dieser und ihre Relationen zueinander. Zusätzlich muss der Lebenszyklus der ganzen Anwendung modellierbar sein. Die Beschreibungssprache TOSCA (Topology and Orchestration Specification for Cloud Applications) versucht alle Anforderungen für das Beschreiben von Cloud-Anwendungen zu erfüllen. Für das Verwalten bzw. Aufsetzen von sogenannten TOSCA *ServiceTemplates* werden Pläne verwendet, die in jeglicher Sprache verfasst werden können. Die manuelle Erstellung dieser Management Pläne erfordert jedoch detaillierte technische Kenntnisse über eingesetzte Technologien und Cloud Infrastrukturen und ist daher sehr fehleranfällig und aufwändig. Insbesondere die enge Kopplung dieser Pläne an die jeweiligen Anwendungen macht es zudem schwierig, diese auf andere Anwendungen zu übertragen. Diese Bachelorarbeit stellt ein Konzept und die Implementierung eines Plan-Generators vor, der Pläne für die Provisionierung von Anwendungen, basierend auf TOSCA Service Templates generiert. Der vorgestellte Ansatz ermöglicht somit die voll-automatische Provisionierung von Anwendungen. Das entwickelte Konzept wird anhand einer prototypischen Implementierung, im Rahmen von OpenTOSCA, validiert und mit bestehenden Ansätzen verglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Algorithmen

1 Einleitung

Mit zunehmender Verwendung von Cloud-basierten Lösungen in Unternehmen und von Privatkunden wird das automatisierte Management von Cloud-Anwendungen immer wichtiger und Verwaltung dieser immer komplexer. Management umfasst dabei das initiale Aufsetzen von Anwendungen, deren Administration, deren Verwaltung und finale Terminierung. Entwickler von Anwendungen auf Basis von Cloud Technologie müssen Entscheidungen bezüglich des Cloud Typs (*Public Cloud*, *Private Cloud*, *Community Cloud*, *Hybrid Cloud*) und des Service Modells (*IaaS*, *PaaS*, *SaaS*, *CaaS*) treffen. Zusätzlich zu diesen müssen die einzelnen Komponenten einer Anwendung auf der gewählten Cloud Infrastruktur korrekt installiert werden. Obwohl das Aufsetzen von einzelnen Komponenten, wie Datenbanken und WebServer, immer leichter wird, ist das für Composite-Cloud Anwendungen, welche aus mehreren Komponenten und Services bestehen können, weiterhin eine Herausforderung. Dabei nimmt die Komplexität, durch weitere Auslagerung von Anwendungen in die Cloud, zu. Anwendungen müssen heutzutage voll verwaltbar sein, dies bedeutet, dass der ganze Lebenszyklus, von Aufsetzen bis zum Herunterfahren, voll automatisiert durchgeführt werden kann. Die Struktur einer Cloud-Anwendung kann in Anwendungstopologien dargestellt werden. Dabei besteht eine Topologie aus Komponenten und ihren Beziehungen zueinander und können jeweils mit Attributen versehen werden.

Diese Bachelorarbeit beschäftigt sich mit dem Generieren von *Build Plänen* aus einer gegebenen Anwendungstopologie. Build Pläne sind Management Pläne, die eine Topologie einer Cloud-Anwendung instanziieren. Die Entwicklung solcher Pläne sind mit hohem Aufwand verbunden und erfordern detailliertes Wissen über einzelne Komponenten, Beziehungen und wie diese auf einer Cloud-Infrastruktur installiert werden. Es empfiehlt sich daher eine Trennung zwischen High-Level und Low-Level Wissen (Abschnitt 3.1) vorzunehmen. Dabei entspricht Low-Level Wissen handgeschriebenen Prozessen, um einzelne Elemente der Anwendung zu instanziieren. Diese Prozesse können anschliessend so orchestriert werden, dass diese der High-Level Schicht entsprechen.

Der Hauptbeitrag dieser Arbeit besteht darin, welche abstrakten Schritte getätigt werden müssen, um einzelne Komponenten einer Anwendungstopologie, verfasst in TOSCA (Abschnitt 2.1), zu instanziieren. Diese Schritte werden dann für einzelne Komponenten und Beziehungen zusammengefasst, um diese dann mittels eines Graph-Algorithmus so zu orchestrieren, dass diese die ganze Topologie provisionieren. Das vorgestellte Konzept wurde zur Validierung, im Rahmen von OpenTOSCA (Abschnitt 2.3) in einem Prototyp implementiert der Build Pläne in WS-BPEL 2.0 (Abschnitt 2.2) generiert.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: In diesem Kapitel werden Grundlagen für diese Arbeit erläutert. Darunter befinden sich TOSCA (Abschnitt 2.1), WS-BPEL 2.0 (Abschnitt 2.2) und OpenTOSCA (Abschnitt 2.3).

Kapitel 3 – Verwandte Arbeiten: Die vorgestellten Arbeiten in diesem Kapitel beschäftigen sich mit dem automatischem Provisioning von Anwendungen. Darunter werden *Set Cover* (Abschnitt 3.2), *Partial Order Planning* (Abschnitte 3.3, 3.1) und Graph (Abschnitt 3.4) Algorithmen verwendet, um einen Managment Plan zu generieren, der die Anwendung in den gewünschten Zustand transformiert.

Kapitel 4 – Konzept: In Kapitel 4 werden die Konzepte dieser Arbeit vorgestellt, dabei werden erst Probleme beim Generieren von BuildPlänen vorgestellt und anschliessend die einzelnen Konzepte erläutert. Nachträglich wird vorgestellt, wie sich der Plan-Generator in OpenTOSCA intergiert und eine Architektur für einen Plan-Generator erläutert.

Kapitel 5 – Design und Implementierung: Kapitel 5 erläutert die Implementierung. Darin wird das Design und Entwicklung der einzelnen Komponenten erläutert.

Kapitel 6 – Zusammenfassung und Ausblick Kapitel 6 fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Grundlagen

2.1 OASIS TOSCA

Das Konsortium „Organization for the Advancement of Structured Information Standards“(OASIS) [SISO] hat bezüglich des steigenden Interesses für und der wachsenden Komplexität von Cloud-Lösungen die Standardisierung einer Beschreibungssprache für Cloud-Anwendungen in die Wege geleitet. Diese Sprache ist die „OASIS Topology and Orchestration Specification for Cloud Applications “(TOSCA) [TCATTC] und befindet sich im Standardisierungsprozess durch das TOSCA Kommitee im Bearbeitungszeitraum dieser Bachelorarbeit. TOSCA ist eine in XML [BPSM+08] implementierte Sprache, welche die Portabilität und Verwaltung von Cloud-Anwendungen und Services während ihres ganzen Lebenszyklus zu beschreiben versucht. Die Topologie einer Cloud-Anwendung wird mittels eines *Enterprise Topology Graphs* [BFL+12] beschrieben, so werden Komponenten (Knoten) als *NodeTemplate* und Relationen (Kanten) zwischen Komponenten als *RelationshipTemplate* bezeichnet. Ein TopologyTemplate fasst nun mehrere solcher Knoten und Kanten zusammen und modelliert somit eine Anwendung. Beide Templates werden typisiert, dabei sind *NodeTypes* die Typen von *NodeTemplates* und *RelationshipTypes* die von *RelationshipTemplates*. Software Artefakte, aus denen die Anwendung besteht, werden in *ArtifactTemplates* beschrieben, die wiederrum einen *ArtifactType* besitzen. *ArtifactTemplates* werden innerhalb von *NodeTypeImplementations* und *RelationshipTypeImplementations* zusammengefasst und als *ImplementationArtifact* (IA) oder *DeploymentArtifact* (DA) verwendet. IA's stellen die Implementierung von Operationen eines *NodeTypes* dar, während DA's als Artefakte gelten, die benötigt werden, um einen NodeType zu instanziieren. Attribute der einzelnen Elemente werden mittels *Properties* als Instanzen eines XML Schemas dargestellt.

Der Lebenszyklus eines TOSCA *ServiceTemplates* wird durch *Plans* verwaltet, dabei werden diese in *Build Pläne*, *Management Pläne* und *Termination Pläne* kategorisiert. Ein *Build Plan* ist dafür zuständig das im *ServiceTemplate* definierte *TopologyTemplate* zu instanziieren, ein *Management Plan* ist für Verwaltungsaufgaben auf der Topologie zuständig, letztere ist ein *Termination Plan*, der eine Topologie aus der Cloud-Umgebung wieder herunterfährt.

2.2 OASIS WS-BPEL 2.0

„OASIS Web Services Business Process Execution Language Version 2.0“(WS-BPEL 2.0) [Org07] ist eine in XML [BPSM+08] implementierte Workflow Sprache, die Geschäftsprozesse, basierend auf *Web Services* [BHM+04], implementiert. BPEL benutzt dazu die WSDL

Schnittstellen [CCMW01] der Services, um die Funktionalität des Prozesses zu implementieren, um diese wiederum mittels einer WSDL Schnittstelle verfügbar zu machen. Dabei basiert das Typsystem für Variabeln in BPEL, wie bei WSDL, auf XML Schema [Falo1], um die Interaktion zwischen diesen mittels SOAP zu ermöglichen und mittels XPath 1.0 zu lesen und zu schreiben. BPEL kann block- und flussbasiert verwendet werden. Es können bekannte Konstrukte wie *if,for* und *while* verwendet werden, um so Blöcke (*scopes*) zu definieren. Zusätzlich ist es möglich einzelne BPEL *scopes* mit *links* zu verbinden, um so einen flussbasierten Ablauf zu implementieren. Dabei können die *links* mit Bedingungen versehen werden, um so den Gesamtfluss zu beeinflussen.

2.3 OpenTOSCA

OpenTOSCA [AI] ist eine am Institut für Architektur von Anwendungssystemen der Universität Stuttgart entwickelte Laufzeitumgebung für TOSCA Anwendungen. Der TOSCA Container verarbeitet Applikationen imperativ. Das bedeutet, dass die erforderlichen Pläne zum Aufsetzen, Verwalten und Runterfahren von Applikationen selber entwickelt und mitgeliefert werden müssen. Die Implementierung basiert auf dem OSGi Framework [OSG08] und besteht somit aus einer Reihe von OSGi-Bundles, die die Gesamtfunktionalität bereitstellen. Darunter lässt sich die High-Level Architektur in API (*ContainerAPI*), Control (*Control*, *TOSCAEngine*, *IAEngine* und *PlanEngine*) und Modell (*OpenTOSCA Core*) unterteilen. Die API wird verwendet um TOSCA Applikationen, als *CSAR* gepackt, zu deployen. CSAR Dateien sind ZIP Dateien mit konventionen bzgl. der Struktur.Beim Instanziieren der Anwendung wird das TOSCA Modell (*Definitions*) geladen und die einzelnen ImplementationArtifacts von der *IAEngine* in der OpenTOSCA Umgebung installiert (TOSCA-Management-Umgebung [TCATTC, S. 14]). Die Pläne der Anwendung müssen in BPEL verfasst sein und werden als letztes auf einem WSO2 Business Process [WSO] Server deployed, davor wurde dieser mit aktuellen Endpunkten, wie etwa einer WebService IA Addresse, aktualisiert. Die OpenTOSCA Core ist eine Menge von OSGi-Bundles, die das speichern von CSAR's und Instanzdaten übernimmt und von der alle weiteren Komponenten ihre Daten beziehen.

3 Verwandte Arbeiten

In diesem Kapitel werden Arbeiten vorgestellt, die sich mit dem Provisionieren von Composite-Cloud Anwendungen beschäftigen oder deren Konzepte sich dafür verwenden lassen. In diesen Arbeiten werden für das Provisionieren *Partial Order Planning* oder *Set Cover* Algorithmen verwendet. Anwendungen werden auf Modelle, die Ist- und Soll-Zustand repräsentieren, abgebildet. Provisioning Operationen werden verwendet, um dem Algorithmus Möglichkeiten zu geben, die Anwendung vom Ist-Zustand in den Soll-Zustand zu überführen. Es wird eine korrekte Reihenfolge von Operationsaufrufen berechnet, um den Soll-Zustand zu erreichen. Weiterhin wird ein Vorgehen vorgestellt, indem Abhängigkeiten zwischen Komponenten als Graph abgebildet werden und dessen Kanten invertiert werden, um einen Ablauf für das Provisioning zu gewinnen. Als letzter Abschnitt werden die einzelnen vorgestellten Arbeiten zusammengefasst und deren Einflüsse auf das Konzept beschrieben.

3.1 Pattern-based Runtime Management of Composite Cloud Applications

Uwe Breitenbücher et al. stellen in ihrem Paper *Pattern-based Runtime Management of Composite Cloud Applications* [BBKL13] ein Pattern-basiertes Vorgehen für das Management von Composite-Cloud Anwendungen vor. Dabei werden High-Level und Low-Level Wissen über das Management von Komponenten getrennt, um dadurch diese für verschiedene Anwendungen automatisiert zu verwenden.

Das Vorgehen wird konzeptionell in drei Schritte gegliedert: Anwenden von *Management Patterns*, Generieren von Management Plänen durch orchestrieren von *Management Planlets* und Ausführen des generierten Plans. Im ersten Schritt werden Management Patterns angewendet, diese entsprechen (Management-)Aufgaben, wie etwa das Skalieren einer Anwendung, und entsprechen einem High-Level Verständnis der auszuführenden Aufgaben, um den gewünschten Effekt zu erzielen. Dabei wird ein *Application State Model* (ASM) der Anwendung in ein *Desired Application State Model* (DASM) durch ein Management Pattern transformiert. Ein Application State Model repräsentiert dabei den aktuellen Zustand einer Anwendung, bestehend aus einer *Application Topology*, der einem Graphen entspricht, welcher die Komponenten und ihre Relationen zueinander beschreibt. Komponenten und Relationen können Eigenschaften besitzen, die einem Key-Value Modell entsprechen. Die Eigenschaften beschreiben Laufzeit-Informationen der entsprechenden Elemente im Topologie Graphen. Zusätzlich können Elemente des Graphen typisiert werden und ihre Eigenschaften vererben.

Komponenten besitzen zusätzlich Schnittstellen die Management-Operationen anbieten. Das Desired Application State Model (DASM) ist ein ASM, das den gewünschten Zustand nach Anwendung von Management Patterns entspricht. Dieses besitzt zusätzlich Annotationen auf Elementen, die verdeutlichen, welche Tasks auf diesen angewendet werden, beispielsweise das eine Komponente kreiert oder gelöscht werden soll.

Management Patterns entsprechen dem High-Level Wissen, um die gewünschte Managementaufgaben zu bewerkstelligen. Ein Pattern besteht aus zwei Teilen, dem *Target Topology Fragment* und einer *Topology Transformation*. Das Target Topology Fragment ist eine kleine Topologie, welche beschreibt auf welche Knoten und Kanten das Pattern angewendet werden kann. Im Beispiel des Papers wird ein Pattern vorgestellt, dass eine Komponente vom Typ *WAR*, die auf einer Komponente jeglichen Typs aufgesetzt und auf einer Public Cloud migriert werden kann. Eine Topology Transformation würde definieren, wie die Topologie aufgebaut ist, nachdem das Pattern angewendet wurde. Die *WAR* Komponente, aus dem Beispiel des Papers, wird so vom Pattern transformiert, dass diese auf einer Amazon EC2 Virtual Machine mit einem Linux und Tomcat installiert wird.

Beim Generieren von Management Plänen auf Basis der transformierten Application State Models wird das DASM, das beim Anwenden von Management Patterns auf das ASM generiert wurde, verwendet, um daraus einen Management Plan zu erstellen. Das Zusammensetzen und Wählen einer passenden Reihenfolge von Planlets wird mittels eines *Partial Order Planning* Algorithmus [Wel94] ermittelt. Dabei wird als Eingabe das ASM und das DASM übergeben. Daraus berechnet der Algorithmus eine passende Reihenfolge von Planlets, die dann zu einem Management Plan zusammengesetzt werden. Der resultierende Plan kann nun vollautomatisch ausgeführt werden und das Management der Anwendung vollziehen.

3.2 Pattern-based Composite Application Deployment

Eilam et al. stellen in ihrem Paper *Pattern-based Composite Application Deployment* [EEKS11] ein Vorgehen vor, dass das Workflow- und Modell-basierte Vorgehen für das Aufsetzen von Composite-Cloud Anwendungen vereint. Dabei sollen bspw. Skripte und Workflows separat entwickelt und anschliessend in einem Modell verwendet werden können.

Ihr Vorgehen basiert auf einem *Deployment Model*, das Ressourcen enthält, die schon installiert wurden und die noch installiert werden müssen. Das Modell wird anschliessend in ein *Workflow Model* transformiert. Die Transformation besteht aus zwei Schritten: Identifizieren der Operationen, die zum Soll-Zustand führen und eine Berechnung einer Reihenfolge in der sie ausgeführt werden können. Dabei werden Operationen als *Automation Signatures* repräsentiert. Automation Signatures bestehen aus einem Pattern Modell, das angibt wie sich die Ausführung der Operation auf die Topologie auswirkt. Zusätzlich enthalten sind Anforderungen an Ressourcen, die von der Operation betroffen sind. Automation Signatures sollen die Verbindung zwischen imperativen Konstrukten (Skripte, Workflows) und deklarativen Modellen darstellen.

Das Modell besteht aus einer Topologie die *Units* enthalten. Ein *Unit* stellt eine Ressource der Topologie die schon instanziiert wurde oder noch instanziiert werden muss, dar. Zusätzlich besitzen diese einen *life cycle state*, die den Zustand eines Units repräsentieren. Dabei besteht dies aus einem Tupel $(initstate, goalstate)$. Units können typisiert werden und mit Attributen versehen werden. Relationen zwischen Units erben von einer festen Menge von Typen: Diese sind *hosting*, *membership*, *dependency*, *propagation* und *realization*. Jede der Relationen besitzt eine eigene Semantik, wie z.B. das *hosting* angibt, das eine Komponente als Container für die andere verwendet werden kann.

Für das Generieren von Workflows aus dem Deployment Modell, werden die Patterns aus allen verfügbaren Automation Signatures auf die Topologie gematcht, um einen Ablauf von Operationen zu finden, der alle *initstates* der Units innerhalb der Topologie zu einem *goalstate* führen. Eilam et al. lösen dieses Problem mit *Set Cover* Algorithmen und benutzen Teilmengen des Deployment Modells als Universum $\mathcal{U}$, als $\mathcal{S}$ werden mögliche Realisierungen von den gegeben *Automation Signatures* definiert. Der Algorithmus muss nun passende Realisierungen finden, die nicht miteinander überlappen, also ein *Exact Cover Problem* lösen.

3.3 Model Driven Provisioning: Bridging the Gap Between Declarative Object Models and Procedural Provisioning Tools

In *Model Driven Provisioning: Bridging the Gap Between Declarative Object Models and Procedural Provisioning Tools*[MMEK06] stellen Maghraoui et al. ein Vorgehen vor, indem Soll-Zustands-Modelle und prozedurale Operationen für das Aufsetzen von Anwendungen in Rechenzentren vereint werden. Dabei werden Ist- und Soll-Zustand in Modellen festgehalten. Daraus werden mittels AI Planning Workflows generiert, welche die Umgebung (Rechenzentrum) aus dem Ist-Zustand in den Soll-Zustand überführen.

Für das Modellieren von Ist- und Soll-Zustand werden *Resource Object Models* verwendet. Diese bestehen aus attributierten, typisierten Knoten und Kanten. Dieses Modell wird in *First-Order Logic* transformiert. Operationen werden in PDDL (Planning Domain Definition Language) [GIP+98] modelliert, welche das Definieren von Vorbedigungen erlaubt. Im Weiteren ist es möglich, Einschränkungen auf den Zustand der Ressourcen, die Topologie selbst und Attributen zu definieren.

Maghraoui et al. nehmen für das Berechnen einer Reihenfolge einen *Partial Order Planning* Algorithmus. Der Input für diesen ist das generierte First-Order Logic Modell und die in PDDL beschriebenen Operationen.

3.4 Cafe

In *A Method and Implementation to Define and Provision Variable Composite Applications, and its Usage in Cloud Computing* [Mie10], der Dissertation von Ralph Mietzner, und im Beitrag *Cafe: A Generic Configurable Customizable Composite Cloud Application Framework* [MUL09] von

Ralph Mietzner et al. wird Cafe vorgestellt. Cafe ist ein Framework, um konfigurierbare, zusammengesetzte, service-orientierte Anwendungen zu entwickeln und diese automatisiert aufzusetzen. Dabei besteht eine Cafe Anwendung (*Application Template*) aus einem *Application Model*, *Variability Model* und den verwendeten Code Artefakten bzw. Referenzen zu diesen. Das Application Model bildet Komponenten, ihre Beziehungen zueinander, die Implementierung einer Komponente und Typen von Komponenten und Implementierungen ab. Zusätzlich werden Komponenten mit einem Pattern beschrieben, dieses kann vom Typ *Single Instance Component*, *Single Configurable Instance Component* oder *Multiple Instances Component* sein, dabei geben die Patterns Aufschluss, ob eine Komponente bspw. Multi-Tenancy unterstützt und so in anderen Anwendungen verwendet werden kann. Die Beziehungen zwischen Komponenten werden mittels *Deployment Relations* und dem Variability Model modelliert. Erstere gibt an, dass eine Komponente auf der anderen installiert werden muss. Letztere ermöglicht es gezielt Daten in Fragmente von Code Artefakten einzufügen, bspw. kann in eine WSDL ein Endpunkt einer anderen Komponente eingefügt werden. Dazu werden *Variability Points* definiert, die unter bestimmten Bedingungen und Phase des ganzen Cafe Prozesses gesetzt werden. Dadurch lassen sich implizite Abhängigkeiten abbilden.

Für das Provisioning von Cafe Apllikationen wird das „Cafe Provisioning and Management Interface“(CPMI) [Mie10, Kapitel 5.2] verwendet, um mit den verschiedenen Granularitäten[Mie10, Kapitel 5.1] für das Managment von Komponenten umzugehen. Dabei werden Workflows (*Component Flows*) verwendet, um das CPMI zu implementieren, die das Nutzen von Komponenten vereinheitlichen. Das CPMI hat als Basis Operationen/-Messages *reserve*, *cancelReservation*, *provision* und *deProvision* um den, vom CPMI definierten, Zustand einer Komponente zu ändern. Reserve/cancelReservation wird dafür genutzt, um eine Komponente zu reservieren, damit beim Provisioning sicher gestellt werden kann, dass alle Komponenten verfügbar sind. Provision/deProvision wird verwendet, um eine Komponente zu starten bzw. zu stoppen, dabei übernimmt der Component Flow diese Aufgaben. Zusätzlich wird das CPMI erweitert für *Provider Components*, welche zusätzlich zu den anderen Operationen noch *reserveDeployment*, *deploy*, *cancelDeployment* und *undeploy* besitzen. Verwendet werden diese, um Komponenten auf Provider Komponenten zu deployen. Es werden eine Reihen von Dokumenten definiert, die beim jeweiligen Aufrufen der Operationen mitgeschickt werden, bspw. wird ein *Customization Document* in einer *customize* Message/Operation mitgeliefert.

Der ganze Ablauf des Provisionings von Cafe Anwendungen wird in folgenden Schritten bewerkstelligt: *Find Realizations*, *Find Correct and Complete Component Bindings*, *Select the Realization Component Binding*, *Component Provisioning* und *Executing Provisioning Activities*. Find Realizations versucht passende schon vorhandene Komponenten und Provider Komponenten für noch zu installierende Komponenten zu finden, um diese für die Anwendung zu nutzen, dabei werden mehrere Möglichkeiten errechnet. Find Correct and Complete Component Bindings wählt aus der Find Realizations Phase korrekte Kombinationen aus vorhandenen Komponenten und noch zu startenden Komponenten aus, dabei bedeutet korrekt, dass alle Abhängigkeiten (von Deployment Relations und Variability Model) erfüllt sind. Im Select the Realization Component Binding Schritt werden aus den Kombination das kostengünstigste ausgewählt und mittels reserve Message/Operation reserviert. Nur wenn alle Komponenten reserviert wurden, ist die Kombination realisierbar. In der Component

Provisioning Phase wird die gewählte Realization in eine *Provisioning Order* transformiert, dabei werden die Abhängigkeiten durch Deployment Relations und durch die Variability Points verwendet, um eine Reihenfolge für die Instanziierung von Komponenten zu generieren. Für Deployment Relations wird der Graph des Application Templates von Blättern zur Wurzel nacheinander instanziiert. Zusätzlich werden Abhängigkeiten des Variability Models zwischen Komponenten als Graph dargestellt und mit dem Application Template Graph vereinigt, so dass eine Abhängigkeit eines Variability Points zu einem anderen zwischen zwei Komponenten als Kante dargestellt wird[Mie10, Definition 104]. Die Kanten des Graphen werden nun invertiert [Mie10, Definition 105], um einen Ablauf für das Provisioning zu haben. Beginnend nun von Wurzeln zu den Blättern, werden die einzelnen Komponenten (Knoten) instanziiert. Dies ist die *Executing Provisioning Activities* Phase, hierbei wird für jede Komponente die Operationen des CPMI genutzt, darunter provision, um die Komponenten zu starten und *customize*, um die gewählte Lösung aus den vorherigen Phasen zu konfigurieren.

3.5 Zusammenfassung und Einflüsse auf das Konzept

Für die Partial Order Algorithmen werden Modelle für Ist- und Soll-Zustand benötigt, zusätzlich müssen Effekte von Operationen definiert werden. TOSCA bietet für Ist- und Soll-Zustand keine dedizierte Möglichkeit an, diese anzugeben. Die einzige Möglichkeit als Ist-Zustand ist, das keine Teile der Topologie in der Umgebung vorhanden sind (im Gegensatz zu schon instanziierten Teilen der Topologie). Für die Effekte einer Operation fehlen Konstrukte jeglicher Art. Zwar können mit dem TOSCA Lifecycle Interface [TCa] interpretiert werden, welche Auswirkungen eine Operation auf die Umgebung hat, doch mit diesem ist es dann nicht nötig auf komplexe Planning Algorithmen zurückzugreifen, welche durch nicht-deterministische Selektionen von Operationen eine hohe theoretische Komplexität mit sich bringen (typischerweise liegen diese in NP [Byl91]). Für Set Cover Algorithmen wird davon ausgegangen, das Teilworkflows für verschiedene Submengen, verschiedener Granularität, der Topologie vorhanden sind, um dann die einzelnen Teile für einen Gesamt-Workflow zu orchestrieren. Diese Teilworkflows sind speziell auf Komponenten und Relationen zugeschnitten und eingeschränkt wiederverwendbar. Die Möglichkeit Effekte von Operationen und die Granularität der Interaktion mit Komponenten mittels eines Interfaces (bspw. CPMI in Abschnitt 3.4) zu generalisieren, erlaubt die Komplexität des Provisionierens zu reduzieren. Jedoch kann ein großer Aufwand entstehen, falls die gewünschte Granularität zu hoch bzw. zu niedrig für die zu provisierenden Komponenten ist.

4 Konzept

In diesem Kapitel werden die Kernkonzepte dieser Arbeit erklärt, die nötig sind, um *Build-Pläne* zu generieren. Pläne welche die Provisionierung einer Cloud-Anwendung beschreiben, werden hier Build-Pläne genannt. Dabei werden erst übergeordnete Eigenschaften der Topologie betrachtet. Darunter fallen Eigenschaften des Topologie-Graphen und die in ihr enthaltenen Komponenten. Aus der Analyse werden dann Definitionen aufgeführt, die für Build-Pläne nötig sind (Abschnitt 4.1). Die in folgenden Abschnitten präsentierten Konzepte, beschreiben eine Methode, wie diese, unter Beachtung der Eigenschaften einer TOSCA Topologie, Teilworkflows aus einer Topologie generiert werden können, die sich aus feiner granulierten Workflows zusammensetzen lassen. (Siehe Abschnitte 4.2.2 und 4.3) Dabei werden Ideen aus Abschnitt 3.1 verwendet, um High-Level Wissen von Low-Level Wissen zu trennen. Zusätzlich können die kleinsten Teile des Konzepts (IA/DA Provisioning Activities,..) (siehe Abschnitt 4.2.2) in Partial Order Algorithmen verwendet werden, da diese als Pre-Conditions und Operations für den Algorithmus bezeichnet werden können (Siehe 3.3). Dabei können die vorgestellten Konstrukte auch in Set Cover Algorithmen verwendet werden, um in einem Exact Cover Problem als einzelne Teilmengen ($\in S$) zu nutzen (Abschnitt 3.2). Jedoch wird hier, anstatt Planning und Set Cover Algorithmen, ein Algorithmus, angelehnt an Cafe aus Abschnitt 3.4, vorgestellt. Dieser basiert auf der Idee, die Kanten der Topologie zu invertieren und dann Schritt für Schritt Low-Level Teilworkflows auszuführen, um das gewünschte Provisioning zu erfüllen.

4.1 Konzepte anhand der TOSCA Topologie für die Plan-Generierung

In diesem Abschnitt werden Herausforderungen, die das maschinelle Generieren von TOSCA Build-Plänen erschweren, erläutert und konzeptionelle Lösungsmöglichkeiten für diese beschrieben. Darunter fallen die Herausforderungen beim Aufsetzen von Softwareartefakten (in TOSCA *ArtifactTemplates* genannt), eine geeignete Wahl der Reihenfolge von Operationen auf Komponenten (*NodeTemplates* in TOSCA) und Beziehungen (*RelationshipTemplates*), das Ermitteln der nötigen Eingabeparameter für Operationen und die Art der Operationsaufrufe. Das Gesamtkonzept baut auf eine klare Definition von *BasisTypen* auf, um somit klare Grenzen für das Entwickeln von Low-Level Workflows zu schaffen und diese dann fürs ganze Provisioning zu verwenden. Falls auf die Möglichkeit von BasisTypen verzichtet wird, fehlen den Konstrukten die nötige Semantik, um die Topologie korrekt zu interpretieren. Anschliessend werden Eigenschaften einer TOSCA Topologie definiert, welche die Komplexität

der Problemstellung reduzieren bzw. Anforderungen definieren, welche beim Erstellen von TOSCA Build-Plänen berücksichtigt werden müssen.

Um die Konzepte in diesem Kapitel zu verdeutlichen, wird auf folgenden Definitionen aufgebaut:

Ein TOSCA TopologyTemplate ist ein gerichteter Graph $G(V, E)$ für den folgende Funktionen und Mengen zusätzlich definiert werden:

- $V := \{Alle\ NodeTemplates\ innerhalb\ der\ TOSCA\ Topologie\}$
- $E := \{(v, w) \mid v, w \in V\}$
- $NodeTypes := \{Alle\ vorhandenen\ NodeTypen\ innerhalb\ der\ TOSCA\ Definitions\}$
- $NodeTypeImplementations := \{Alle\ vorhandenen\ NodeTypeImplementations\ innerhalb\ der\ TOSCA\ Definitions\}$
- $nodeTypeImplementations : NodeTypes \rightarrow NodeTypeImplementations$, bildet NodeTypeImplementations eines NodeTypes ab.
- $RelationshipTypes := \{Alle\ vorhandenen\ RelationshipTypen\}$
- $RelationshipTypeImplementations := \{Alle\ vorhandenen\ RelationshipTypeImplementations\ innerhalb\ der\ TOSCA\ Definitions\}$
- $ArtifactTypes := \{Alle\ vorhandenen\ ArtifactTypes\ innerhalb\ der\ TOSCA\ Definitions\}$
- $ArtifactTemplates := \{Alle\ vorhandenen\ ArtifactTemplates\ innerhalb\ der\ TOSCA\ Definitions\}$
- $artifactTypes : NodeTypeImplementations \rightarrow ArtifactTypes$, bildet die verwendeten ArtifactTypen einer NodeTypeImplementation ab.
- $artifactType : ArtifactTemplates \rightarrow ArtifactTypes$, bildet ein ArtifactTemplate auf ein ArtifactType ab.
- $Parameter := \{(x, y) | x \in Menge\ von\ Parameternamen,\ y \in Menge\ von\ Parametertypen\}$
- $Operation \subseteq 2^{Parameter}$
- $Interface \subseteq 2^{Operation}$
- $Interfaces \subseteq 2^{Interface}$
- $ImplementationArtifacts := \{(x, y) |\ x \in Interface, y \in ArtifactTemplates\}$
- $Keys := \{Alle\ Schlüssel\ von\ Properties\ innerhalb\ der\ TOSCA\ Topologie\}$
- $Values := \{Alle\ Werte\ von\ Properties\ innerhalb\ der\ TOSCA\ Topologie\}$
- $type : V \cup E \cup ArtifactTemplates \rightarrow NodeTypes \cup RelationshipTypes \cup ArtifactTypes$
- $typeRef : NodeTypes \cup RelationshipTypes \rightarrow NodeTypes \cup RelationshipTypes \cup \emptyset$, bildet das *derivedFrom* der NodeTypes und RelationshipTypes ab.

- $parentTypes : NodeTypes \rightarrow 2^{NodeTypes}$
- $childTypes : NodeTypes \rightarrow \{x \mid x \in NodeTypes\}, p \mapsto \{c \mid parentTypes(c) = p\}$
- $keys : V \cup E \cup ArtifactTemplates \rightarrow Keys$
- $value : Keys \rightarrow Values$
- $implementations : V \cup E \rightarrow NodeTypeImplementations \cup RelationshipTypeImplentations$
- $interfaces : V \rightarrow Interfaces$
- $sourceInterfaces : E \rightarrow Interfaces$
- $targetInterfaces : E \rightarrow Interfaces$
- $ias : NodeTypeImplementations \cup RelationshipTypeImplementations \rightarrow ImplementationArtifacts$
- $das : NodeTypeImplementations \cup RelationshipTypeImplementations \rightarrow ArtifactTemplates$

4.1.1 Anforderung an TOSCA Topologien

Basis von TOSCA Typen

TOSCA bietet die Möglichkeit, da die Topologie einem *Resource Object Model* [MMEK06] oder *Enterprise Topology Graph* [BFL+12] entsprechen, Typen, darunter NodeTypes, RelationshipTypes, etc. voneinander erben zu lassen. Dies erlaubt die Wiederverwendung von bestehenden Typen und das Einteilen in Kategorien. Unter den Kategorien können sich generische NodeTypen für Server, WebApplication, virtuelle Maschine bis hinzu speziellen Typen wie Apache HTTP Server, WARApplikation oder Amazon EC2 Instance befinden. Für RelationshipTypes empfiehlt sich häufig auftretende Beziehungen zwischen NodeTypen zu definieren, bspw. *connectsTo*, *deployedOn*, *dependsOn*, etc. Dabei ist es von entscheidender Rolle das spezifischere Typen von generischen Typen, Eigenschaften erben und die Semantik dieser generischen *BaseTypes* nicht verletzen. Ein Grund für diese Anforderung sind im nächsten Abschnitt beschriebene Unterteilung der NodeTypes und RelationshipTypes (siehe Abschnitt 4.1.1). Weitere Gründe hierfür sind auch Aspekte der Interoperabilität. Diese wird Mithilfe von BasisTypen verstärkt, da *Type Architects*[TCb], Experten für Typen von Komponenten und Relationen, die Typen die für eine Anwendung benötigt werden, von diesen erben lassen können. So kann es dem *Application Architect*, Experten für gesamt Struktur einer Cloud-Anwendung und deren Lebenszyklen [TCb], (bzw. einem Plan-Generator) leichter ermöglichen TOSCA Pläne zuschreiben.

Dabei ist die Wahl der Basistypen kein triviales Unterfangen und kann evtl. die Mehrdeutigkeiten einzelner Komponenten einer Cloud-Applikation nicht formal erfassen oder reduzieren. In dieser Ausarbeitung wird nicht darauf eingegangen, wie sich eine Menge von Basistypen wählen lässt. Es wird bei der Vorstellung der einzelnen Konzepte, falls zum Verständnis nötig, teilweise darauf eingegangen. Eine umfassende Untersuchung ist aber nicht Gegenstand dieser Arbeit.

Für BaseTypes lassen sich folgende Definitionen verwenden:

- $BaseTypes \subseteq \{x \in NodeTypes \cup RelationshipTypes | \ typeRef(x) = \emptyset\}$
- $hasBasetype(x) \mapsto \begin{cases} true & \exists y \in parentTypes(x) \mid y \in BaseTypes \\ false & else \end{cases}$
- $baseType(x) := \{y \mid y \in parentTypes(x) \wedge y \in BaseTypes\}$

Infrastructure Nodes und Edges

In diesem Abschnitt werden *Inrastructure NodeTypes* und *Infrastructure RelationshipTypes* definiert. Diese geben an, ob ein NodeTemplate ein *Infrastructure Node* und ein RelationshipTemplate eine *Infrastructure Edge* ist. Diese Definitionen werden verwendet, um einem Plan-Generator anzugeben, ob eine Komponente zur Infrastruktur der Topologie beiträgt. Dabei werden Infrastructure Edges verwendet, um einen Stack von Infrastrukturen für ein NodeTemplate zu bilden. Diese Informationen können von einem Plan-Generator verwendet werden, um zu bestimmen auf welchen Knoten ein Artefakt aufgesetzt werden muss.

Für das Aufsetzen von Software Artefakten auf Knoten innerhalb einer TOSCA Topologie ist es nötig diese zu kategorisieren: In Knoten, die erlauben, dass Software Artefakte auf ihnen installiert werden und in Knoten, die dies nicht ermöglichen. Dabei sind die vorhandenen Basistypen einer Topologie die Grundmenge von möglichen Kandidaten für eine Menge von *Infrastructure NodeTypes*, die Artefakte akzeptieren. *Infrastructure NodeTypes* sind bspw. Server. Knoten, auf denen das Materialisieren von ImplementationArtifacts und DeploymentArtifacts möglich ist, werden hier *Infrastructure Nodes* genannt und haben einen *Infrastructure NodeType* als Basistyp. Unter Materialisieren versteht sich, das DAs und IAs auf einer Komponente der Topologie installiert werden, bspw. Zip Dateien oder Skripte. Diese IAs/DAs müssen auf einer Infrastructure Node entlang eines *Infrastructure Paths* materialisiert werden. Man beachte, dass nicht alle IAs auf einem Knoten der Topologie installiert werden dürfen, sondern in der TOSCA-Managed Umgebung installiert werden müssen (siehe dazu „Aufsetzen von DeploymentArtifacts und ImplementationArtifacts “).

Infrastructure NodeTypes sind NodeTypes, die eine bestimmte Menge an *ArtifactTypes* auf sich installieren lassen und sich so generell als Infrastrukturkomponenten klassifizieren lassen. Dabei ist die Menge größer desto genereller der NodeType, bspw. kann ein NodeType *Linux* alle möglichen Artefakte auf sich aufsetzen, wo hingegen ein *Apache Tomcat* sich für ArtifactTypen *WAR* eignet. *Infrastructure NodeTypes* können zu den *BaseTypes* gehören, da Typen für gängige Infrastrukturkomponenten wie, bspw. einen Server, vorkommen sollten. In TOSCA können solche Eigenschaften bspw. mit *CapabilityTypes* und *RequirementTypes* dargestellt werden. So könnte man einen Basistyp *ContainerCapability* und einen *ContainerRequirement* definieren, die es semantisch erlauben alles auf einen NodeType zu materialisieren. Spezialisierungen dieser könnten, für WAR Dateien, als *WARContainerCapability* und *WARContainerRequirement* definiert werden. Eine weitere Möglichkeit wäre, die verwendeten Infrastructure NodeTypes mit *deploy*-Operationen zu versehen. Dabei könnte die Operationen teil eines Interfaces sein, entsprechend dem TOSCA Lifecylce Interface [TCa], und

angeben welche DeploymentArtifacts mittels dieser einer NodeType Instanz zur Verfügung gestellt werden können. Ein Beispiel wäre im *type*-Attribut, eines Parameters der Operation, eine Referenz eines ArtifactTypes anzugeben.

Infrastructure RelationshipTypes sind RelationshipTypes, die semantisch Beziehungen zwischen Knoten darstellen, die einer Komponente-zu-Container Beziehung entsprechen. Dabei entspricht der *Source*-Knoten der Komponente und der *Target*-Knoten dem Container auf dem das Artefakt der Komponente installiert werden kann. Ein Beispiel für ein *Infrastructure RelationshipType*, der zugleich als BaseType verwendet werden kann, wäre eine *deployedOn*-Relation [BFL+12]. Dabei können Infrastructure RelationshipTypes die im oberenen Paragraph beschriebenen *CapabilityTypes/RequirementTypes* verwenden.

Infrastructure Nodes haben durch die Infrastrukturtypen folgende Eigenschaften:

InfrastructureNode

- Ein NodeTemplate ist ein Infrastructure Node, falls sein NodeType als Basistyp ein Infrastructure NodeType besitzt.
- Ein Infrastructure Node beherrscht eine bestimmte Menge an ArtifactTypes. Dies bedeutet das ermittelt werden kann, welche Art von Artefakten auf diesen Knoten materialisiert werden können.
- Ein Infrastructure Node kann eine *deploy* Operation besitzen. Diese Operation ist semantisch so zu verstehen, dass mittels Ausführen dieser und mitgeben einer DeploymentArtifact, diese auf der angegebenen NodeType-Instanz der Infrastructure Node installiert wird.

Zusätzlich muss es möglich sein die Beziehungen zwischen den verschiedenen Infrastructure Nodes zu identifizeren, daher sind *Infrastructure Edges* zu definieren, die wiederrum einen Infrastructure RelationshipType besitzen. Ein Beispiel für so eine Relation hätte als BaseType einen *deployedOn* Typ:

InfrastructureEdge

- Eine *Infrastructure Edge* sind RelationshipTemplates, die einen RelationshipType besitzen, der als Basistyp einen *Infrastructure RelationshipType* besitzt.
- Eine *Infrastructure Edge* hat immer eine *Infrastructure Node* als Ziel (in TOSCA als Target bezeichnet).

Ein BuildPlan-Generator muss für jegliches Aufsetzen von Artefakten innerhalb der Topologie ermitteln, welche Knoten es erlauben Artefakte auf ihnen zu installieren. Beispielsweise muss dieser, um eine PHPApplikation zu provisionieren, wissen, an welcher Stelle im Graphen dies geschehen kann. Das Beispiel in Abbildung 4.1 erläutert das Vorgehen. Der Generator muss dafür den Pfad von *PHPApplication* nach *VM* über die *deployedOn* Kanten einsehen, um dort den möglichst besten *Infrastructure Node* für das Installieren zu wählen. Dabei sind die Kriterien für eine optimalen Knoten, die Länge des Pfades vom NodeTemplate zur *Infrastructure Node* und die Menge der eingehaltenen Eigenschaften dieses Knotens von

Bedeutung. Ein Plan-Generator nimmt idealerweise immer den ersten Knoten auf dem Pfad, der eine *deploy* Operation für das zu installierende Artefakt anbietet.

Folgende Definitionen fassen Eigenschaften einer *Infrastructure Node* und *Infrastructure Edge* zusammen und definieren einen *Infrastructure Path*:

- $deployableTypes : NodeTypes \rightarrow ArtifactTypes$, hier werden ArtifactTypes zurückgegeben, die ein NodeType auf sich deployen lässt.
- $InfrastructureNodeTypes := \{x \mid x \in BaseTypes \wedge deployableTypes(x) \neq \emptyset\}$, Infrastructure NodeTypes sind NodeTypes, die es erlauben, Artefakte auf diese zu deployen.
- $InfrastructureNodes := \{x \in V \mid \exists y \in parentTypes(x) : y \in InfrastructureNodeTypes\}$, ein NodeTemplate ist ein Infrastructure Node, sobald in seiner NodeType Typ-Hierarchie, ein Infrastructure NodeType vorhanden ist.
- $InfrastructureRelationshipTypes := \{x \mid baseType(x)$ *stellt eine Beziehung zwischen Komponente und einem Container für diese dar*$\}$
- $InfratructureEdge := \{x \in E \mid \exists y \in parentTypes(x) : y \in InfrastructureRelationshipTypes\}$
- $InfrastructurePath(x) := \{xv_1v_2v_3...v_n \mid type((x,v_1)), type((v_1,v_2)), .., type((v_{n-1},v_n)) \in InfrastructureRelationshipTypes \wedge type(v_1), type(v_2), .., type(v_n) \in InfrastructureNodeTypes\}$, $x, v_1, .., v_n \in V$

4.1.2 Instanziierung von NodeTemplates und RelationshipTemplates

NodeTemplates entsprechen Komponenten in einer Cloud-Topologie in TOSCA und werden anhand von NodeTypes typisiert. Dabei ist es möglich alle Arten von Typen zu definieren, doch es empfiehlt sich Typen zu definieren, die auch tatsächlich in einer Cloud-Applikation auftreten, bspw. Application Server, Datenbank, etc. (siehe Abschnitt „Basis von TOSCA Typen“). Ein NodeType bietet die Möglichkeit Operationen und Properties zu definieren. Properties stellen Eigenschaften der Komponente dar, während Operationen dafür vorgesehen sind, Instanzen eines NodeTypes zu manipulieren [TCATTC, Seite 15]. Beispielsweise würde ein NodeType *ApplicationServer* in seinen Properties eine IP-Addresse bereitstellen und eine Operation *Start* um eine ApplicationServer in der TOSCA-Management-Umgebung [TCATTC, Seite 14] zu starten. Properties werden mithilfe von XML-Schema [Falo1] definiert und Instanzen dieser werden in NodeTemplates angegeben. Implementierung für Operationen werden durch *ImplementationArtifacts* (IA) bereitgestellt, diese können aus einfachen Skripten oder ganzen WAR Dateien bestehen und werden in *NodeTypeImplementation* Elementen definiert. NodeTemplates besitzen darüber hinaus noch die Möglichkeit *DeploymentArtifacts* (DA) zu definieren. Diese werden verwendet, um ggf. benötigte Dateien für die Instanziierung bereitzustellen (Bsp.: gezippte PHP-Applikation).

Um nun Instanzen von NodeTypes zu erstellen, müssen Operationen vorhanden sein, die dies ermöglichen, bspw. eine *Install* Operation. Zusätzlich zu Operationen, die eine Instanz eines NodeTypes erstellen, werden Operationen benötigt, die die Instanz konfigurieren

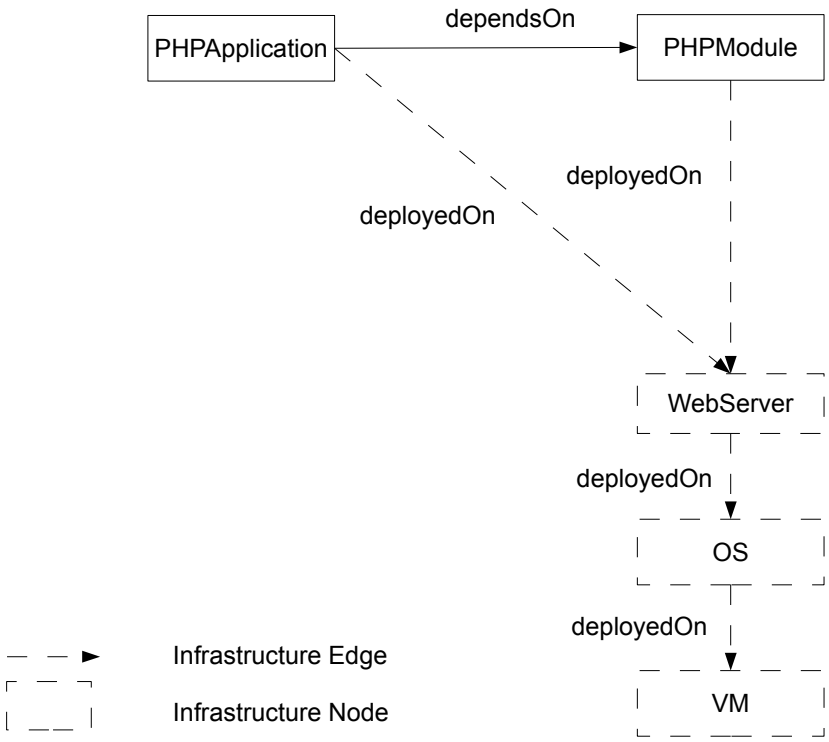

Abbildung 4.1: Beispiel einer Topologie mit Infrastructure Nodes und Edges

und verwalten. Beispielsweise muss ein Port geöffnet bzw. geschlossen werden, um Zugriff zu ermöglichen bzw. zu verhindern. Operationen, die dafür zuständig sind eine Instanz zu starten oder zu stopppen, werden benötigt, da nicht jede Komponente ohne ein explizites Starten verfügbar ist. Operationen können pro NodeType definiert werden und verschiedene Parameter verlangen. Diese können in Properties stehen oder ausserhalb der TOSCA-Definition, also nicht aus der Topologie ermittelbar. Weiterhin muss sichergestellt werden, dass Implementierungen der Operationen verfügbar sind, bevor ein Build-Plan das Instanziieren des einzelnen Topologie Elements beginnt.

Für RelationshipTemplates sind die selben Probleme vorhanden, mit dem Zusatz, dass entschieden werden muss auf welchen Interfaces (TOSCA *Source-* und *TargetInterface*) mit dem Instanziieren begonnen wird und auf welcher Komponente Artefakte des Templates manifestiert werden.

Für Build-Pläne ergeben sich durch die beschriebenen Eigenschaften folgende Probleme beim Erstellen von *NodeType* und *RelationshipType* Instanzen:

- Verwendung von *DeploymentArtifacts* und *ImplementationArtifacts*

- Identifizierung der Art eines Operations-Aufrufs
- Ermitteln einer Reihenfolge von Operations-Aufrufen
- Wahl der Parameter

Diese Herausforderungen werden im nachfolgenden Abschnitten behandelt und ein Konzept präsentiert, dass diese zusammenfasst und konstruktiv behandelt. Dabei werden Verantwortlichkeiten, zwischen den einzelnen Komponenten und Workflows eines TOSCA-Containers, wie OpenTOSCA, definiert, um einen klaren Ablauf auf den verschieden Ebenen, wie etwa Low-Level Workflows, zu erreichen. In folgenden Abschnitten werden oft Definitionen bzgl. der Umgebung erwähnt. Die Umgebung entspricht prinzipiell allen möglichen TOSCA-Management-Umgebungen, wobei hier konkret die OpenTOSCA Umgebung als Ziel verwendet wird. OpenTOSCA bietet an, IAs vom Typ WAR zu installieren und somit den BPEL Plänen verfügbar zu machen. Dies wird durch die IAEngine (Siehe 2.3) bewerkstelligt, die erweitert werden kann, um weitere ArtifactTypen zu verarbeiten.

4.1.3 Aufsetzen von DeploymentArtifacts und ImplementationArtifacts

ImplementationArtifacts (IA) und *DeploymentArtifacts* (DA) in TOSCA stellen konkrete Software-Artefakte der TOSCA Topologie dar. Dabei ist die Semantik der beiden verschieden. ImplementationArtifacts stellen die Implementierung für NodeType/RelationshipType Operationen dar, während DeploymentArtifacts dazu verwendet werden, um die funktionale Software einer Anwendung bereitzustellen[TCATTC, Seite 14].

Ein Beispiel für ein DeploymentArtifact wäre eine PHP-Applikation. Dabei wird eine ZIP Datei benötigt, die auf einem System entpackt werden muss. Dies bedeutet auch, dass DeploymentArtifacts zur Laufzeit von Build-Plänen verwendet werden, um diese in der Topologie zu installieren. Hierbei muss der BuildPlan-Generator anhand des Artefakttyps des DeploymentArtifacts und der Topologie ermitteln, wie er das Artefakt deployed. DAs müssen nicht immer auf eine Komponente geladen werden. Beispielsweise können mit *deploy/install*-Operationen, eine URI zu dem Artefakt übergeben werden und das Hochladen wird von diesen erledigt.

Ein ImplementationArtifact unterscheidet sich in der Handhabung in der Hinsicht, dass diese möglichst vor Beginn der Topologie-Instanziierung zur Verfügung stehen. Ein Beispiel dafür wäre eine WAR Datei, die einen WebService enthält, mit dem Instanzen von Virtuellen Maschinen gestarten werden können. Dies soll die TOSCA Management-Umgebung bewerkstelligen [TCATTC, Seite 14]. Dabei ist das Vorgehen nicht verschieden zudem eines Build-Plans. Die TOSCA Management-Umgebung muss die Topologie interpretieren und das IA korrekt installieren. Ein Spezialfall sind IAs, die nicht vor Beginn der Instanziierung verfügbar gemacht werden können. Darunter fallen bspw. Skript Artefakte die auf einer laufenden Komponente des Zielsystems (eine Instanz eines NodeTypes) installiert werden müssen. Hierbei lassen sich diese von Build-Plänen zur Laufzeit auf das System übertragen. Alle anderen IAs, die nicht von Komponenten abhängig sind, können direkt vor Beginn des Provisionings installiert werden.

Somit kann folgende Definition für ein IA verwendet werden:

$$deployableIntoEnvironment : ArtifactTypes \rightarrow \{true, false\},$$

$$artifact \mapsto \begin{cases} true & \textit{TOSCA Management-Umgebung erlaubt ArtifactTyp zu installieren} \\ false & else \end{cases}$$

Das Vorgehen beim Installieren von Artefakten ist beschränkt durch den Artefakttyp dieser, den Typ der Infrastructure Nodes entlang der Pfade vom Template zu den Blättern der Topologie. Zusätzlich muss beachtet werden, dass IAs auf Knoten der Topologie installiert werden oder in der TOSCA Management-Umgebung. Der *ArtifactType* gibt an, wie man grundlegend diesen zu handhaben hat. Beispielsweise ob das Artefakt eine ZIP Datei ist und diese, auf einen Infrastructure Node, entpackt werden muss. Es muss entschieden werden, ob das Artefakt in der Management-Umgebung installiert werden muss, da es bspw. einen WebService darstellt. Der NodeType des NodeTemplates gibt an was materialisiert wird, also ob es sich um eine Applikation, einen Application Server, Betriebssystem, etc. handelt. Dabei ist es nicht von Belang ob dieser ein Infrastructure Node ist oder nicht. Falls das NodeTemplate ein Infrastructure Node ist, müssen entweder Artefakte auf einen tieferen Infrastructure Node installiert werden oder in der TOSCA Umgebung. Falls das NodeTemplate kein Infrastructure Node ist, wird das gleiche Vorgehen gewählt. Die Infrastructure Nodes entlang der Pfade von Template zu den Blättern sind beim Aufsetzen der Artefakte von großer Bedeutung. Diese müssen es erlauben anhand ihrer Definition zu ermitteln, welche Arten von Artefakten auf ihnen installiert werden können. Dafür können mehrere Ansätze verfolgt werden:

Auf Basis des NodeTypes Auf der Basis von NodeTypes können Mengen an ArtifactTypes definiert werden, die ein NodeType akzeptiert. Dabei können diese in Meta-Daten in der NodeType Definition hinterlegt oder anhand des Typs selber ermittelt werden, die dann ein Plan-Generator verarbeiten muss, um festzustellen, ob ein Artefakt auf dem Knoten installiert werden kann. Dabei kann auch die Vererbungshierarchie der NodeTypes in Betracht gezogen werden, bspw. kann ein Basetype Server alle Artefakte akzeptieren und ein davon erbender LinuxServer NodeType eine beschränktere Menge, damit keine .EXE Dateien auf diesen hochgeladen werden.

Bei diesem Ansatz müsste der Plan-Generator die Meta-Daten interpretieren können oder für die Basetypes einen Katalog an akzeptierenden Artefakten pro Typ besitzen.

Ein Plan-Generator würde in diesem Fall folgende Definition verwenden:

$$InfrastructueNodeMap : InfrastructureNodeTypes \rightarrow 2^{ArtifactTypes}, p \mapsto deployableTypes(p)$$

Auf Basis von CapabilityTypes CapabilityTypes werden dafür verwendet, um jegliche Art von Fähigkeiten von NodeTypes zu definieren. Hierbei gibt die Spezifikation als Beispiel die Fähigkeit eines NodeTypes einen Datenbank-Endpunkt zur Verfügung zu stellen an, mit dem andere NodeTypes eine Datenbankverbindung aufbauen können. CapabilityTypes können hier auch dafür verwendet werden, um anzugeben, dass ein NodeType

Artefakte eines bestimmten Typs akzeptiert. Ein Plan-Generator kann so eine Menge von $(CapabilityType, NodeType)$ Tupeln besitzen, die generisch Artefakte auf Komponenten laden.

Hierbei muss der Plan-Generator einen Katalog an $(CapabilityType, NodeType)$-Handlern besitzen.

- $capabilityTypes := \{Alle\ CapabilityTypes\ der\ TOSCA\ Definitions\}$
- $deployableTypes : CapabilityTypes \rightarrow ArtifactTypes$, bildet CapabilityTypes auf ArtifactTypes ab, diese geben dann an welche Artefakte ein NodeType akzeptiert.

Auf Basis von Operationen Ein NodeType, der ermöglichen soll Artefakte aufzunehmen, kann auch eine Operation zur Verfügung stellen, die genau für den jeweiligen Artefakttyp vorgesehen ist. Dabei muss dem Plan-Generator klar sein, dass so eine Operation auch dafür vorgesehen ist, welches möglich wäre mittels einer wiederverwendbaren Schnittstelle (Siehe dazu auch 4.1.6). Einschränkung dazu ist das die Endknoten jeglicher Infrastruktur Pfade der Topologie, IA's besitzen die in der Umgebung installiert werden müssen.

Dieser Ansatz reduziert die Komplexität des Plan-Generators und verteilt das Verarbeiten/Aufsetzen von Artefakten auf die jeweiligen Komponenten der Topologie.

Für diesen Ansatz gelten folgende Definitionen:

- $deployableTypes : Operations \rightarrow ArtifactTypes$, gibt an ob eine Operation für das Deployen von ArtifactTypen verwendbar ist.
- $InfrastructureNodeSinks := \{x \mid x \in InfrastructureNodes : \exists (x,y) \notin E\ \}$
- $\forall x \in InfrastructureNodeSinks \mid deployableTypes(z) \neq \emptyset,\ z \in Interfaces(x)$, für alle Infrastructure Nodes die Blätter der Topologie sind, müssen Operationen vorhanden sein, die es erlauben Artefakte auf diese zu laden.
- $\forall x \in V \setminus InfrastructureNodeSinks \mid artifactTypes(nodeTypeImplementations(x)) \subseteq deployableTypes(InfrastructureNodeSinks)$, für alle Komponenten der Topologie müssen deren Artefakte, mindestens auf die Infrastruktur Komponenten Blätter installiert werden können.

4.1.4 Identifizierung der Art eines Operations-Aufrufs

Operationen von Nodetypes und RelationshipTypes werden in TOSCA durch ImplementationArtifacts implementiert. Diese wiederum werden durch ArtifactTypes typisiert. Dabei können ArtifactTypes von anderen erben und erlauben dadurch eine Verfeinerung der Typen anhand der Vererbungskette. Ein Beispiel dafür wäre, dass ein ArtifactType WAR von einem ArtifactType ZIP erbt. Darüber hinaus werden Mengen von IAs in *NodeTypeImplementations* bzw. *RelationshipTypeImplementations* zusammengefasst.

Die Art in der eine Operation, bspw. eines NodeTypes, aufgerufen wird hängt somit von der verwendeten ImplementationArtifact ab und dessen ArtifactType. Um die zu ermitteln müssen die verwendeten Typen in einer Topologie so interpretierbar sein, dass ein BuildPlan-Generator aus diesen einen passenden Operations-Aufruf generieren kann, den dann ein Build-Plan verwendet, um dann die Operation auf dem NodeType aufzurufen. TOSCA definiert keine eigenen Elemente, die direkt angeben, dass ein Artefakt eine gewisse Schnittstellenart anbietet. Ein Beispiel für eine Schnittstellenart wäre WSDL. Dies wird deutlich, wenn man einen ArtifactType WAR für ein IA verwendet, um einen WebService als Implementierung für einen NodeType anzugeben. Eine WSDL-Schnittstelle in der gegebenen WAR Datei müsste also innerhalb dieser, vorher identifiziert werden können, um einen Operations-Aufruf zu generieren. Die grundlegende Herausforderung bei der Identifizierung der Aufrufsarten ist die Tatsache, dass sich nicht immer deterministisch feststellen lässt, dass ein ArtifactType genau eine Schnittstellenart anbietet. Eine WAR Datei kann bspw. auch eine REST-basierte Schnittstelle anbieten.

Die TOSCA Spezifikation empfiehlt Artefakte mit Meta-Daten zu versehen, die von einem TOSCA Container verstanden werden um die ImplementationArtifacts in der TOSCA-fähigen Umgebung zu installieren [TCATTC, Seite 14]. Dadurch können Build-Pläne über den Container NodeTypes instanziieren und/oder Operationen auf den Instanzen ausführen. Diese Vorgehensweise kapselt die Komplexität von Build-Plänen ab und ermöglicht Aufrufe über eine einheitliche Schnittstelle abzuwickeln. Jedoch müsste die Komponente auch mit IA's umgehen, die nicht in der Umgebung installiert werden (siehe 4.1.3), um alle Operationsaufrufe abstrahieren zu können.

Eine Alternative wäre einen Build-Plan selbst, mit generischen Fragmenten anzureichern, um geeignete Aktivitäten zu generieren, die sich mit den Meta-Daten innerhalb eines Artefakts kümmern und so Operationen aufrufen. Dabei kann das Vorgehen dem des TOSCA Containers entsprechen. Dazu muss das ImplementationArtifact anhand ihres ArtifactTypes, den gegeben Meta-Daten und das Artefakt selbst betrachtet werden, um eine genaue Schnittstellenart zu identifizieren. Dabei können auch Informationen darüber auf welcher Komponente das Artefakt installiert wurde von nutzen sein, wie etwa, dass eine WAR Datei auf einem JEE Application Server installiert wurde.

Für die Identifizierung der Schnittstellenart einer Operation gelten folgende Definitionen:

- $InterfaceTypes := \{$*Alle auftretenden Schnittstellenarten innerhalb der TOSCA Topologie*$\}$
- $OperationType : Operations \rightarrow InterfaceTypes$, es muss definiert werden, welche Schnittstellenart für eine Operation verwendet werden muss.

4.1.5 Parameter Handling

In TOSCA können Parameter für Operationen definiert werden. Dabei können diese als Eingabe- und Ausgabe-Parameter definiert, sowie deren Namen und Typ angegeben werden. Die Typen sind nicht auf XML Schema beschränkt und es können reine Strings verwendet werden, die Hinweise auf den tatsächlichen Typ geben. Ein Build-Plan-Generator muss im

Stande sein, diese Typen zu identifizieren und korrekte Transformationen dieser für die jeweiligen Schnittstellentypen zu erzeugen.

Ein Plan-Generator muss, bevor er passende Aufrufe für Management Operationen generiert, ermitteln woher und welche Werte er für die einzelnen Parameter erhält. Dabei können Parameterwerte direkt in den Properties einzelner Templates liegen, in externen Komponenten für Instanzverwaltung (bspw. in einer Plan Portability API [PPA]), Eingaben von Benutzern, etc.. Die verschiedenen Quellen können wiederum in verschiedenen Formaten vorliegen und verschiedene Zugriffsmöglichkeiten vorraussetzen. Diese Werte müssen darüber hinaus in passende Repräsentationen und Formate gebracht werden, damit diese für Operationen verwendet werden können.

Bei der Wahl der Parameter für eine TOSCA Operation lässt sich unterscheiden zwischen internen und externen Parameterwerten. Dabei sind interne Parameterwerte, die sich aus der gegebenen Topologie ermitteln lassen. Zu diesen gehören hauptsächlich Properties von allen Elementen der Topologie, Artefakte, etc. Unter externen Parameterwerten verstehen sich Werte, die sich nicht aus der Topologie bestimmen lassen, und die Eingabe von aussen oder das Anfordern bei externen Komponenten benötigen. Ein Beispiel dafür wären die Zugangsdaten eines Cloud-Anbieters oder Instanzdaten einer Komponente, die über eine externe Schnittstelle angefordert werden.

Handhabung interner Parameter Interne Parameterwerte einer Topologie bezeichnen Werte von Parametern, die aus der Topologie entnommen werden können, z.B. durch Properties. Das bedeutet, dass zum Beispiel Properties die ein NodeTemplate definiert hat, als Eingabe bzw. Ausgabe für Operationen verwendet werden können. DeploymentArtifacts und ImplementationArtifacts fallen auch unter die Kategorie der internen Parameterwerte, da in TOSCA die Möglichkeit existiert *deploy*-Operationen zu definieren, um mittels dieser Artefakte auf eine Komponente zu laden. Ein Beispiel für das Setzen einer Property wäre ein NodeTemplate mit BaseType Server. Eine Operation, die das NodeTemplate instanziiert, würde als Ausgabe Parameter eine Referenz auf eine *Server IP*-Property besitzen. Diese Property würde nach dem Instanziieren gesetzt werden. Als Beispiel für das Lesen einer Property, wäre eine *connectsTo*-Relation, die eine WebApplication mit einer Datenbank verbindet. Dabei müsste eine ServerIp Property entlang eines Infrastrukturpfades vorhanden sein, die unterhalb der Datenbank Komponente liegt. Wenn diese gefunden wurde, wird diese der WebApplication Komponente in einer Operation (Bspw. *connect*) mitgeliefert.

Um ermitteln zu können welche internen Werte auf welche Parameter angesetzt werden müssen, ist es nötig diese dem Plan-Generator anzugeben. Dafür bietet TOSCA keine native Methode an, die sich dafür eignet. Deshalb müssen Erweiterungen vorgenommen werden, die das Werte ausfindig machen ermöglicht. Dazu können einfache Mechanismen verwendet werden, wie in etwa Parameter-Name entspricht einem Property-Namen des Templates oder man verwendet Technologien wie XPath [BBC+07] oder URIs [BFL+12] um eine Property zu referenzieren, den gewünschten Property-Wert auszulesen und diesen dann als Parameterwert weiterzugeben. Weiterhin sollte es Möglichkeiten geben Artefakte an Operationen weiterzugeben, bspw. durch Angabe welche Artifacttypen diese akzeptieren.

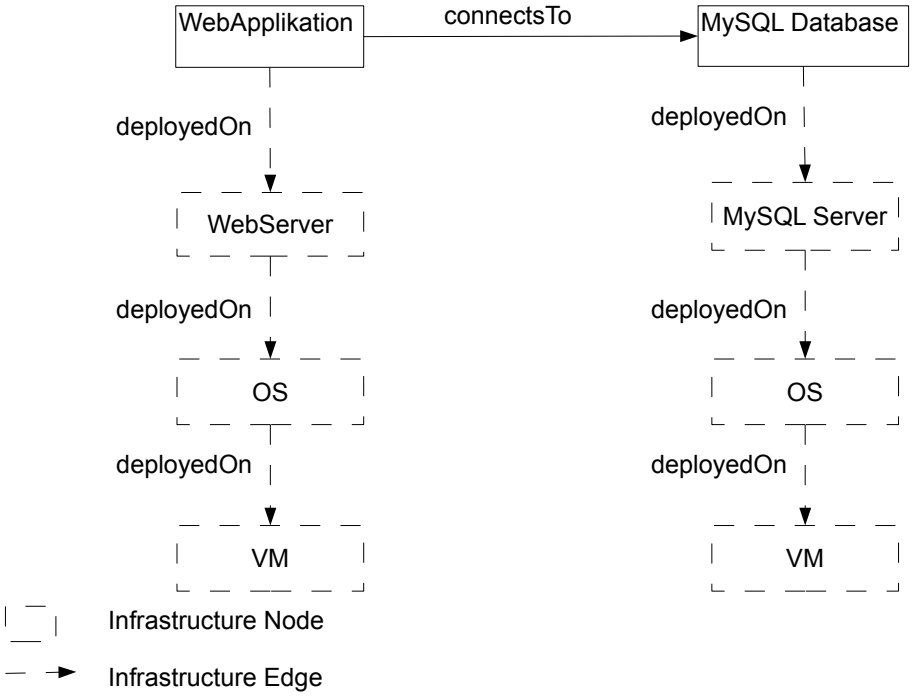

Abbildung 4.2: Beispiel einer Topologie mit Infrastructure Nodes, Edges und einer connectsTo-Relation

Um die Wiederverwendbarkeit von NodeTypes und RelationshipTypes zu erhalten empfiehlt es sich interne Parameterwerte im jeweiligen Template anzugeben und zu halten. Jedoch gibt es Fälle bei dem dies nicht möglich ist: Eine WebApplikation die Zugriff auf eine Datenbank benötigt und dies mittels einer „connectsTo“-Relation modelliert wurde, benötigt zwangsläufig eine IP-Addresse des Servers auf der die Datenbank aufgesetzt ist. Hierbei wird die Bedeutung von *Infrastructure Nodes*, *Edges* und wohldefinierten BaseTypes einsichtlich. An Abbildung 4.2 wird dies verdeutlicht. Ein Plan-Generator kann Anhand der RelationshipType connectsTo den beiden Knoten „WebApplication“und „MySQL Database“feststellen, dass eine IP-Addresse benötigt wird, da die beiden Komponenten nicht auf dem selben Infrastrukturpfad liegen. Falls nun die IP-Addresse nicht in „MySQL Database“vorhanden ist, muss der Generator entlang des Infrastrukturpfades, der MySQL Datenbank, nach dieser suchen. Wenn diese gefunden wurde, bspw. in der VM Komponente, wird die IP-Addresse der Operation innerhalb des Source-Interfaces, der „connectsTo“-Relation verfügbar gemacht. Wegen diesem Fall muss es auch möglich sein, Parameterwerte anhand von CapabilityTypes (bspw. IP Capability) und/oder generellen Queries (bspw. mit XPath), Properties von Infrastrukturen zu ermitteln.

Für interne Parameter gilt folgende Definition:

$internalParameter : Parameter \rightarrow Keys \cup ImplementationArtifacts \cup DeploymentArtifacts$, bildet einen Parameter auf eine Property, IA oder DA ab.

Handhabung externer Parameter Parameter deren Werte nicht anhand der Topologie ermittelt werden können, werden hier als externe Parameter bezeichnet. Zu den Mitgliedern dieser Menge gehören bspw. Zugangsdaten eines Cloudanbieters und jegliche Konfigurationsoption, die in der Topologie abgebildet werden.

Ein Build-Plan muss die Möglichkeit haben diese Werte zu erhalten, ob diese per Request mitgeliefert werden, dieser sie selber von einer API oder diese zur Laufzeit mittels BPEL4People [FEK07] erhält ist implementierungsabhängig. Dabei beeinflusst die Wahl der Implementierung inwieweit der Build-Plan eigenständig die Topologie instanziieren kann.

Die Ermittlung von externen Parametern gestaltet sich in Hinsicht auf interne Parameter, als genau die Parameter, die für die in der Topologie keine Werte ermittelt werden konnten.

Für externe Parameter gelten nun folgende Definition:

$externalParameters := \{x \mid x \in Parameter \wedge internalParameter(x) = \emptyset\}$, alle Parameter die nicht als Wert eine Property, IA oder DA haben.

4.1.6 Ermitteln der Reihenfolge von Operations-Aufrufen

Beim Generieren eines Build-Plans muss die Reihenfolge der Operations-Aufrufe auf einer NodeType/RelationshipType Instanz ermittelt werden. Diese Reihenfolge muss im Falle eines NodeTypes semantisch einem *Installieren der Komponente*, *Konfigurieren der Komponente*, evtl. *Starten einer Komponente* entsprechen. Ein RelationshipType nutzt Operationen auf NodeTypes über zwei TOSCA Schnittstellen, das Source- und TargetInterface. Die Reihenfolge dieser Operation muss ermöglichen, dass etwaige Konfigurationen auf der Target-Node ausgeführt werden bevor gleiches auf der Source-Node vorgenommen wird. Diese Arbeit beschäftigt sich mit Build-Plänen, daher wird hier nur auf das korrekte Instanzieren, Konfigurieren und evtl. Starten einer Komponente eingegangen.

Für die Ermittlung der Operationen, die genau den beschriebenen Vorgang ermöglichen, können zwei Konzepte verwendet werden. *Property-Getrieben* und/oder *Interface-Getrieben*.

Property-Getrieben Bei einer Property-getriebenen Ermittlung, einer passenden Reihenfolge von Operationen, ist das Ziel, die noch nicht gesetzten Werte einer NodeTemplate Property mit dem Aufrufen von Operationen valide zum gegebenen XML-Schema zu setzen. Dafür können Planning Algorithmen gezielt verwendet werden. Dieses Konzept folgt aus der Definition der TOSCA Spezifikation, in der nach Instanziierung einer Komponente die Properties valide zum Schema sein sollen[TCATTC, Seite 30]. Dieses Vorgehen setzt

vorraus, dass ermittelt werden kann, welche Property auf welchen Parameter einer Operation gesetzt wird (Siehe Abschnitt 4.1.5). Wenn ermittelt werden kann, welche Property auf welche (In/Out)-Parameter einer Operation gesetzt wird, lassen sich dadurch Abhängigkeiten bzgl. des Ablaufs ermitteln. Bspw. kann eine „Configuration-Operation“ nicht verwendet werden, weil eine Property „ServerIP“ noch nicht gesetzt ist, bis eine entsprechende „Install-Operation“ dies getan hat. Eine Reihenfolge würde also darauf aufbauen, dass nur Operationen verwendet werden, deren Properties bereits vorhanden sind. Erst nachdem diese Operationen aufgerufen wurden, können andere folgen. Das gleiche Vorgehen ist auch bei RelationshipTypes möglich, mit der Ausnahme, dass TargetInterface Operationen Vorrang vor SourceInterface Operationen haben.

Der Property getriebene Ansatz kann bei korrektem Entwerfen von Properties und Operationen der NodeTypes korrekte Ergebnisse liefern, doch ist immer eine Wahrscheinlichkeit da, das evtl. eine semantisch falsche Reihenfolge ermittelt wird, bspw. durch Parameter die externe Werte benötigen die selbst mit eines Planning Algorithmus nicht zum gewünschten Ergebniss führen können.

Interface-Getrieben Die Interface getriebene Ermittlung einer Reihenfolge baut auf der Möglichkeit auf, das TOSCA es ermöglicht, Interface Elemente wiederverwendbar zu entwerfen/entwickeln. Dabei werden generische Operationen verwendet, die genau die Semantik des Installierens, Konfigurierens und Startens einer Komponente widerspiegeln. Diese Operationen müssen auf allen NodeTypes verwendbar sein und auch eine sinnvolle Reihenfolge implizit vermitteln (Bsp.: Kein Konfigurieren vor einem Installieren). Das TOSCA Komitee hat dies bezüglich ein TOSCA Interface erstellt, dass die Operationen *Install*, *Configure*, *Start*, *Stop* und *Uninstall* definiert, aber noch nicht offiziell in die Spezifikation eingeflossen ist[TCa]. Mit dem beschriebenen Interface ist es offensichtlich welche Reihenfolge verwendet werden muss. In TOSCA können auch Operationen aus dem Interface weggelassen werden, ohne dabei das Ermitteln einer Reihenfolge zu gefährden. Vorraussetzung ist, dass ein NodeType mindestens eine Operation für das Instanziieren anbietet (*Install*,*Start*). Für RelationshipTypes eignet sich das Interface bedingt, da ausser der *Configure* Operationen schwer die Semantik eines Starten und Stoppens auf einer Relation abbilden lässt.

Der Interface getriebene Ansatz ist vereinfacht die Ermittlung einer Reihenfolge durch einen passenden Entwurf einer solchen Schnittstelle, welches die Lifecycle Phasen einer Komponente bzw. Relation genau definiert. Dabei bedeutet „passend“, dass immer eine explizit definierte Reihenfolge von Operationen gegeben ist. Das beinhaltet Semantik der Operationen und deren Prioritäten gegeneinander, wie etwa das eine *Install-Operation* einen NodeType in der TOSCA-Umgebung installiert und diese immer vor einer *Configure-Operation* aufgerufen wird.

4.2 Template Build Plan Fragmente

Dieser Abschnitt beschreibt *Template Build Plan Fragmente*. Template Build Plan Fragmente sind Workflow-Fragmente, die einzelne Provisioning-Schritte, einer TOSCA Komponente oder Relation, ausführen. Dabei bestehen diese aus drei Phasen: *Pre-Phase* (Abschnitt 4.2.2), *Provisioning-Phase* (Abschnitt 4.2.2) und *Post-Phase* (Abschnitt 4.2.2). Jeder der Phasen benutzt fertige Abläufe von Aktivitäten, die sich an den Konzepten aus Abschnitt 4.1 orientieren. Diese Abläufe werden *Provisioning Activities* genannt und werden in *IA Provisioning Activities* (Abschnitt 4.2.1), *DA Provisioning Activities* (Abschnitt 4.2.1), *Operation Activities* (Abschnitt 4.2.1) und *Consistency Activities* (Abschnitt 4.2.1) kategorisiert.

4.2.1 Provisioning Activities

Provisioning Activities sind Abläufe von Workflow-Aktivitäten. Diese werden in Aktivitäten kategorisiert, die ein DA verfügbar machen (*DA Provisioning Activities*), ein IA installieren (*IA Provisioning Activities*), TOSCA Management Operationen aufrufen (*Operation Activities*) und Konsistenz von TOSCA Daten wiederherstellen (*Consistency Activities*), falls diese während des Provisionierens verletzt wurden. Diese Aktivitäten sind stark mit den gewählten Konzepten verbunden. Das heisst, falls für das Referenzieren von Properties XPath verwendet wurde, müssen Provisioning Activities das verarbeiten können. Darüber hinaus ist, bei der Implementierung von Provisioning Activities, zu beachten, dass es zusammen gehörende Mengen von IA/DA Provisioning Activities und Operation Activities geben kann. Dies wird am Beispiel von Skript Artefakten klar. Wenn Skripte von DA/IA Aktivitäten auf ein System hochgeladen werden, müssen passende Operation Activities existieren, die wissen wo diese auf dem System liegen, um diese bspw. per SSH auszuführen.

IA Provisioning Activities *IA Provisioning Activities* sind Aktivitäten, die ein ImplementationArtifact für den Build-Plan aufrufbar machen. Diese Aktivitäten sind ein Ablauf, um ein IA aufzusetzen, dabei müssen diese das Artefakt aus der jeweiligen Template Implementierung (NodeTypeImplementation bzw. RelationshipTypeImplementation) auslesen, auf einen Infrastructure Node laden, dort installieren und eventuelle Konfigurationen vornehmen. Dabei hängt der Ablauf von der Topologie ab, da verschiedene Aktivitäten benötigt werden, um ein Artefakt auf einen speziellen NodeType zu deployen. Deshalb sind IA Aktivitäten speziell für einen Tupel $(IA, NodeType)$ ausgelegt. Ein BuildPlan-Generator wählt anhand dieses Tupels den passenden Ablauf. Dabei entspricht der NodeType eines der Komponenten entlang eines Infrastruktur Pfades, von dem NodeTemplate des IAs. Ein Beispiel für dieses Vorgehen, dass ein Build Plan IA Provisioning Activities benötigt, lässt sich anhand von OpenTOSCA beschreiben. OpenTOSCA ist in der Lage WebService IA's in dessen TOSCA-Management Umgebung zu deployen, jedoch keine Skripte. Als Test-Cases für OpenTOSCA wurden 1-Tier und 2-Tier Cloud-Anwendungen verwendet, bei denen VM's für die Infrastruktur verwendet wurden. Die Applikationen auf diesen, wurden mittels Skripten installiert. Es wurden WebService IA's benutzt, um die VM's zu provisionieren, jedoch mussten für Skript IA's handgeschriebene Abläufe verwendet werden. Mit Skript

IA Provisioning Aktivities, können die benötigten Artefakte, auf die konkreten VMs dieser Anwendung, automatisiert geladen und von Build-Plänen verwendet werden.

Für IA Provisioning Activities werden durch folgende Definition typisiert:

$IAProvisioningActivitiesTypes := \{(x,y) \in ArtifactTypes \times (NodeTypes \cup \emptyset) \mid \exists z \in parentTypes(y) : z \in InfrastructureNodeTypes \wedge x \in deployableTypes(y)\}$

$IAProvisioningActivitiesInput := \{(IA, NodeTemplate) \in ImplementationArtifacts \times (InfrastructureNodes \cup \emptyset)\}$

Falls ein IA auf einer Komponente der Topologie installiert werden muss, muss diese Komponente ein Infrastructure Node sein und den ArtifactType akzeptieren. Falls das IA auch in der Umgebung installiert werden kann, sind Aktivitäten auch dafür zulässig.

DA Provisioning Activities Für das Aufsetzen von DeploymentArtifacts werden fertige Abläufe verwendet, die durch einen Tupel aus $(DA, NodeType)$ typisiert werden. Diese werden *DA Provisioning Activities* genannt. Diese Abläufe laden das DA auf den nächsten *Infrastructure Node* entlang eines Pfades aus *Infrastructure Edges* vom NodeTemplate des DAs zu einem Blatt der Topologie. Die Aktivitäten für ein DA müssen dieses nicht installieren oder ausführen, dafür werden die Operation des jeweiligen NodeTypes benutzt. Eines der Test-Cases für OpenTOSCA ist eine PHP-Applikation, die als DA eine Zip-Datei besitzt, in der die Applikation selbst gepackt ist. Skripte mussten dann diese Zip-Datei, auf der VM Komponente, an die richtige Stelle entpacken. DA Provisioning Activities werden nur dazu verwendet, DA's für IA Operationen verfügbar zu machen. Somit können die Aktivitäten denen von IA Provisioning Activities ähneln.

Für DA Provisioning Activities gilt folgende Definition:

$DAProvisioningActivitiesTypes := \{(x,y) \in ArtifactTypes \times NodeTypes \mid \exists z \in parentTypes(y) : z \in InfrastructureNodeTypes \wedge x \in deployableTypes(y)\}$

$DAProvisioningActivitiesInput := \{(DA, NodeTemplate) \in DeploymentArtifacts \times InfrastructureNodes\}$

Falls ein DA auf einer Komponente der Topologie installiert werden muss, muss diese Komponente ein Infrastructure Node sein und den ArtifactType akzeptieren.

Operation Activities Ein Ablauf von Aktivitäten, die die jeweiligen Operationen eines NodeTypes aufrufen, werden durch einen Tupel $(Operation, IA, NodeTemplate)$ typisiert und *Operation Activities* genannt. Die Operation wird benötigt, um eventuelle Property Mappings auszulesen (Siehe Abschnitt 4.1.5). Das IA wird verwendet, um die Schnittstellenart zu identifizieren und evtl. Meta-Daten in dieser zu verwenden. Das NodeTemplate wird mitgegeben, um Properties aus dieser und aus Templates der darunter liegenden Infrastruktur verfügbar zu machen. Für ein WebService IA würde der WSDL Porttype und die verwendete Operation im IA referenziert sein. So können Operation Activities passende SOAP Aufrufe generieren. Bei einem Skript IA, müssten diese den Namen des Skripts aus dem IA lesen können. Weiter

muss es möglich sein, die Variablen aus der Operation für den Skript Aufruf und mittels des Infrastruktur Pfads des NodeTemplates die korrekte Addresse für eine SSH-Verbindung auslesen zu können.

Für Operation Activities gilt folgende Definition zum typisieren:

$OperationActivitiesTypes := \{x \in ArtifactTypes\}$

$OperationActivitiesInput := \{(op, ia, t) \in Operations \times ImplementationArtifacts \times (V \cup E) |\ (op \in interfaces(t) \oplus (op \in sourceInterfaces(t) \cup targetInterfaces(t))) \wedge ia \in ias(implementations(t))\}$

Consistency Activities *Consistency Activities* werden verwendet, um der Umgebung aktuelle Properties zur Verfügung zu stellen. Dabei kann die Umgebung eine Instanzdatenbank sein oder im Build-Plan verwendete Variablen. Die Aktivitäten brauchen nur das jeweilige Template, um ihre Arbeit zu verrichten. Diese Aktivitäten sind optional, aber können von Nöten sein. Je nach Implementierung der IA/DA/Operation Activities, also ob bspw. Properties Workflow-intern gehandhabt werden, müssen Consitency Activities, Instanzen auf verschiedenen Datenbanken updaten. Zusätzlich sind diese Aktivitäten dafür verantwortlich, dem System indem der Plan-Generator integriert ist, NodeType Instanzdaten von neu instanziierten Komponenten zur Verfügung zu stellen.

Für Consistency Activities werden anhand des NodeTypes (bzw. RelationshipTypes), eines NodeTemplates (bzw. RelationshipTemplates), typisiert. Das bedeutet, dass ein Template von einer festen Menge von Consistency Activities behandelt wird.

$ConsistencyActivitiesTypes := \{x \in RelationshipTypes \cup NodeTypes\}$

$ConsistencyActivitiesInput := (NodeTemplate) \in V \wedge (RelationshipTemplate) \in E$

4.2.2 Template Build Plan Fragment

Hier wird das Konzept von *Template Build Plan Fragmenten* vorgestellt, um die Probleme beim Instanziieren von TOSCA Templates zu lösen. Ein Build Plan verwendet ein Template Build Plan Fragment pro NodeTemplate oder RelationshipTemplate, um jeweils diese zu provisionieren. Ein Template Build Plan Fragment ist eine Reihe von IA Provisioning Activities, DA Provisioning Activities, Operation Activities und Consistency Activities die in drei verschiedene Phasen eingeordnet werden: *Pre-Phase* Aktivitäten, *Provisioning-Phase* Aktivitäten und *Post-Phase* Aktivitäten (Siehe 4.3). Weiterhin werden in einem Fragment erst die Pre-Phase Aktivitäten, danach die Provisioning-Phase Aktivitäten und zuletzt die Post-Phase Aktivitäten ausgeführt.

Pre-Phase *Ein Pre-Phase ist eine Menge von Aktivitäten, die alle nötigen DeploymentArtifacts und ImplementationArtifacts eines Templates verfügbar machen.*

Ein Pre-Phase eines Template Build Plan Fragments besteht aus einer Reihe sequenziell ablaufender Aktivitäten, die sich in zwei Kategorien unterteilen lassen: Aktivitäten die ein DeploymentArtifact oder ein ImplementationArtifact verfügbar machen/deployen. Dabei ist jeweils genau ein Ablauf von Aktivitäten zulässig, um jeweils ein Artefakt verfügbar zu machen. Der Plan-Generator muss entscheiden, auf welchem Infrastructure Node er das Artefakt installiert. Dazu muss er Informationen des Artefakts, das Template indem es liegt und dessen Abhängigkeiten in Betracht ziehen. Ein Pre-Phase verwendet IA Provisioning Activities und DA Provisioning Activities, um die Artefakte zu verarbeiten. Dabei werden erst alle IAs mit ihren jeweiligen Aktivitäten ausgeführt und danach Aktivitäten der DAs. Dies folgt aus der Forderung, dass IAs vor jeglicher Managementaktivität verfügbar sein sollen. Darüber hinnaus müssen evtl. DAs für das Instanziieren eines Nodes benutzt werden und so können DA Aktivitäten die Operationen des Nodes direkt verwenden. Ein Beispielablauf für eine Pre-Phase wird in Abbildung 4.4 gegeben. Dort wird ein Skript IA Install.sh auf den nächsten Infrastructure Node „OS“ hochgeladen, so dass es für die Operation Activities verfügbar ist. Danach wird die WebServer.zip DA auf die selbe Komponente geladen.

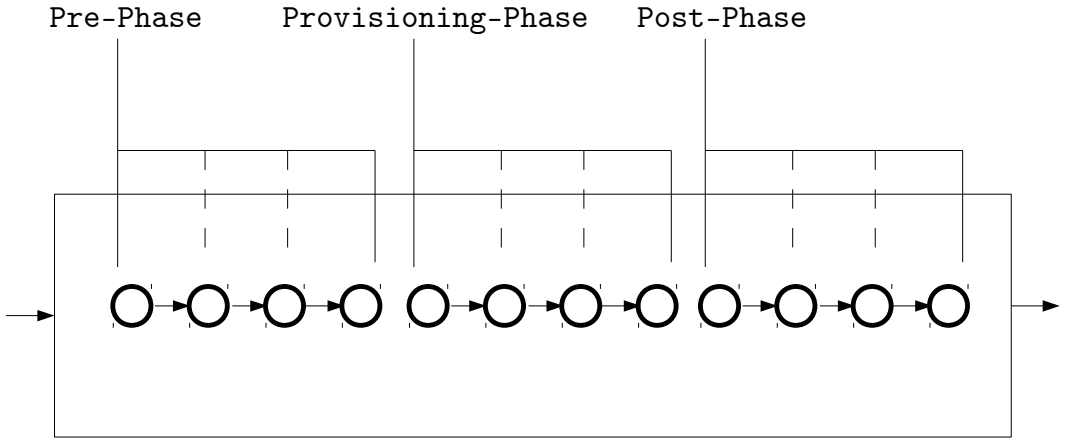

Abbildung 4.3: Aufbau Template Build Plan Fragment

Nach diesen Aktivitäten können Operation Activities das Install.sh Skript nutzen, um die WebServer.zip zu entpacken und somit den WebServer Knoten zu instanziieren.

Für ein Pre-Phase gilt folgende Definition:

$Pre - Phase := \{ia_1 ia_2..ia_n da_1 da_2..da_m \mid ia_1, ia_2, .., ia_n \in IAProvisioningActivities, da_1, da_2, .., da_m \in DAProvisioningActivities\}$, ein Pre-Phase ist ein Ablauf von IA Provisioning Activities und DA Provisioning Activities, wobei IA Aktivitäten vor DA Aktivitäten ausgeführt werden.

$Pre - Phase\ Input := \{iaInput_1,iaInput_2,..,iaInput_n,daInput_1,daInput_2,..,daInput_m \mid iaInput_1,iaInput_2,..,iaInput_n \in IAProvisioningActivitiesInput, daInput_1,daInput_2,..,daInput_m \in DAProvisioningActivitiesInput\}$

Provisioning-Phase *Ein Provisioning-Phase ist ein Ablauf von Aktivitäten, die mittels Management-Operationen eines Templates dieses instanziieren, konfigurieren und starten.*

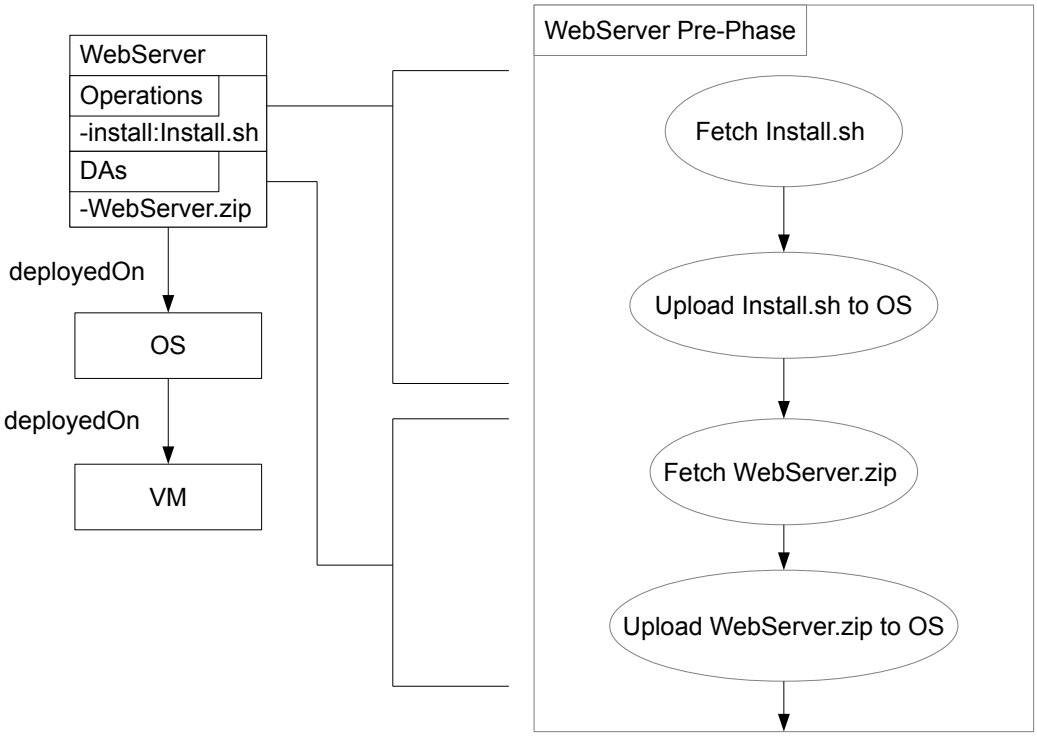

Abbildung 4.4: Beispiel einer Pre-Phase mit einem WebServer NodeTemplate

In einem Provisioning-Phase wird eine Reihenfolge von Management-Operationen ausgeführt, dabei werden Operation Activities für jeweils eine Operation verwendet. Die Reihenfolge wird vom Plan-Generator ermittelt und mittels passenden Aktivitäten in der Phase ausgeführt. Beispielsweise würde bei einem Interface-getriebenen Ansatz, Aktivitäten für eine Install-Operation in dieser Phase liegen und nach diesen Aktivitäten für eine Start- oder Konfigurier-Operation.

Für ein Provisioning-Phase gilt folgende Definition:

$Provisioning - Phase := \{op_1 op_2..op_n \mid op_1, op_2, .., op_n \in OperationActivities\}$, eine Provisioning Phase führt Operation Activities entsprechend, der ermittelten Reihenfolge des Plan-Generators, aus.

$Provisioning - Phase\ Input := \{\ opInput_1,opInput_2,..,opInput_n \mid opInput_1,..,opInput_n \in OperationActivitiesInput\}$

Post-Phase *Ein Post-Phase ist ein Ablauf von Aktivitäten, die die Konsistenz wiederherstellt, die in der jeweiligen Pre- oder Post-Phase verletzt wurde.*

Dafür werden Consistency Aktivitäten verwendet. Der Plan-Generator wird bspw. Aktivitäten in die Post-Phase einsetzen, bei denen Properties als Parameterwerte für Operations-Parameter verwendet wurden. Dabei werden Ausgabe Parameter, die evtl. innerhalb des Workflows verwaltet werden, an externe Komponenten gesendet. Ein Post-Phase ermöglicht somit, dass Aktivitäten aus den vorherigen Phasen, sich nicht mit verschiedenen Komponenten des gesamt Systems auseinander setzen müssen und sich dadurch deren Komplexität verringert.

Für einen Post-Phase gilt folgende Definition:

$Post - Phase := \{con_1 con_2..con_n \mid con_1, con_2, .., con_n \in ConsistencyActivities\}$

$Post - Phase\ Input := \{\ conInput_1,conInput_2,..,conInput_n \mid conInput_1,conInput_2,..,conInput_n \in ConsistencyActivitiesInput\}$

4.3 Topology Build Plan

In diesem Kapitel wird vorgestellt, wie die Grundstruktur eines Build-Plans (hier auch *Topology Build Plan* genannt) aufgebaut ist und wie Template Build Plan Fragmente darin genutzt werden. Der Einfluss von RelationshipTemplates (bzw. deren RelationshipType) auf den Ablauf innerhalb des Plans wird zusätzlich erläutert. In einem Topology Build Plan werden fertig generierte Template Build Plan Fragmente für jeweils ein Template (Node- und RelationshipTemplate) verwendet.

Grundstruktur Der Ablauf eines Build Plans wird direkt aus der Topologie generiert, wobei die RelationshipTemplate-Abhängigkeiten zwischen den NodeTemplates den Gesamtablauf bestimmen. Die NodeTemplates, die die Blätter der Topologie darstellen, werden als erstes instanziiert. Diese stellen die untersten Infrastruktur Knoten, auf denen alles weitere aufgebaut wird. Darauf folgen die eingehenden RelationshipTemplates und darauf die NodeTemplates von denen diese ausgehen. Das Vorgehen ist angelehnt, an den von Cafe (Abschnitt 3.4). In Cafe werden die Kanten der Topologie invertiert und einem neuen Graph verwendet (Provisioning Order Graph), da durch die Deployment-Relations Abhängigkeiten bezüglich der Infrastruktur abgebildet werden. Durch das Invertieren werden die Komponenten zu erst instanziiert, die verwendet werden, um auf ihnen andere Komponenten zu deployen. In TOSCA jedoch besitzen RelationshipTemplates eigene Operationen (auf NodeTypes) und besitzen semantische Aspekte durch ihren RelationshipType. Der RelationshipType beeinflusst in welcher Reihenfolge die NodeTemplates und das RelationshipTemplate instanziiert werden. Dabei soll die Reihenfolge der Semantik entsprechen. Um die Operationen eines RelationshipTemplate auszuführen, werden diese, aus Sicht des BuildPlan-Generators, wie NodeTemplates behandelt. NodeTemplates und RelationshipTemplates werden, mittels Template Build Plan Fragment, als Knoten in einem neuen Graph dargestellt. Die Kanten des Graphen werden, anhand der RelationshipTemplates, innerhalb der Topologie gesetzt.

Ein Topology Build Plan ist nun ein Graph mit folgender Definition:

- $G_{plan}(V_{plan}, E_{plan})$ *ist ein Topology Build Plan mit der Knotenmenge* V_{plan} *und der Kantenmange* $E_{plan} \subseteq V_{plan} \times V_{plan}$.
- $V_{plan} := V \cup \{v_e \mid \forall e \in E\}$, für jeden Knoten aus der Topology und für jede Kante der Topology, existiert ein neuer Knoten im Topology Build Plan.
- $E_{plan} := V_{plan} \times V_{plan}$
- $\forall e \in E$: *e ist eine Relation, bei der Target Node vor Relation und Source Node instaziiert werden muss* $E_{plan} \cup (t, v_e) \cup (v_e, s), s, t, v_e \in V_{plan}$
- $\forall e \in E$: *e ist eine Relation, bei der Target Node und Source Node vor der Relation instaziiert werden muss* $E_{plan} \cup (t, v_e) \cup (s, v_e), s, t, v_e \in V_{plan}$

Abbildung 4.5 veranschaulicht einen vereinfachten Prozess, um einen Topology Build Plan Skelett zu generieren. Ein Build Plan Skelett ist ein Workflow Graph mit Platzhalten für Template Build Plan Fragmente, bestehend aus Knoten für NodeTemplates und RelationshipTemplates. Der Plan-Generator würde als erstes, für jedes NodeTemplate und RelationshipTemplate innerhalb der Topologie, einen Knoten im Graphen erstellen. Danach würde er in Topologie nach Senken suchen. Hier wäre die einzige Senke der VM Knoten. Ausgehend von diesem Knoten, geht nun der Plan-Generator die eingehenden Kanten des VM NodeTemplates entlang und verbindet als erstes im Build Plan Graphen VM mit dem relation1 Knoten. Darauf wird der relation1 Knoten mit OS Knoten im Build Plan Graph verbunden. Dies wird bis zum WebServer Knoten fortgeführt. Bei mehreren eingehenden Kanten werden im Beispiel, Pfade (WebServer, relation4, PHPApplication) und (WebServer, relation3, PHPModule, relation5, PHPApplication) gebildet, um ein PHPModule vor der PHPApplication auf dem WebServer zu installieren.

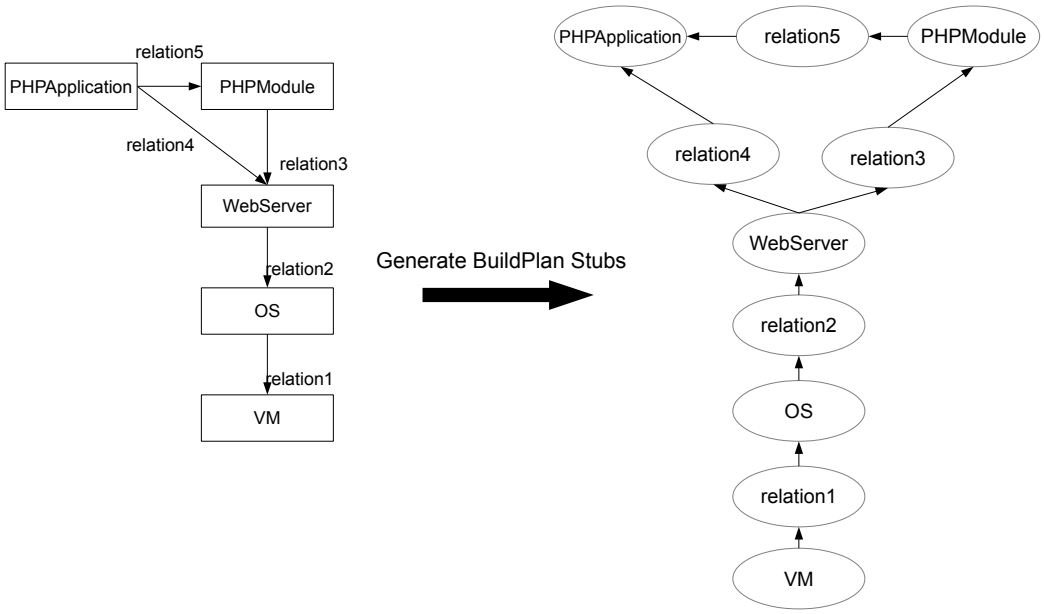

Abbildung 4.5: Beispiel eines Build-Plans

Einfluss von RelationshipTypes auf den Ablauf RelationshipTemplates sind die Kanten des TOSCA Topologie Graphen und entsprechen den Beziehungen zwischen den NodeTemplates. RelationshipTemplates werden durch RelationshipTypes typisiert, dabei soll der Typ den Graphen mit Semantik anreichern. Darunter können Kanten Beziehungen, wie etwa *deployedOn*, *connectsTo*, *dependsOn*, etc., darstellen. Bei der Generierung von Build Plänen müssen solche Beziehungen entsprechend ihres Types behandelt werden, dass bedeutet der Ablauf muss entsprechend generiert werden. Dabei nehmen BaseTypes (Siehe 4.1.1) eine wichtige Rolle ein. Diese geben vor, in welcher Reihenfolge das RelationshipTemplate und die beiden NodeTemplates, die es verbindet, instanziiert werden. Es gibt somit zwei Möglichkeiten dies zu tun. Dabei müssen BaseTypes vermitteln, welche Möglichkeit ausgewählt werden muss. Beispielsweise würde ein *connectsTo* RelationshipType vermittelt, dass einer der beiden NodeTemplates auf den anderen Zugriff benötigt. Dabei wäre das NodeTemplate, das als Source für das RelationshipTemplate definiert ist, die Komponente, die zum Target-Template Verbindung erhalten soll. So müssten erst beide NodeTemplates instanziiert werden und danach das RelationshipTemplate. Im Gegensatz dazu, muss die *dependsOn* RelationshipType anders behandelt werden. Der Knoten, welcher als Source Knoten definiert ist, wird nach dem RelationshipTemplate instanziiert. Vor beiden wird erst

das Target NodeTemplate instanziiert. Für alle RelationshipTypes müssen diese Abläufe klar definiert werden, damit ein Plan-Generator der Semantik entsprechend Pläne generiert.

Algorithmus 4.1 Algorithmus in Pseudocode für die Generierung eines Build Plans mit Platzhaltern für Templates

procedure GENERATEBUILDPLANSKELETON(TopologyTemplate as directed graph $G(V, E)$)
  $G_{plan}(V, E)$:= new empty directed graph;
  for all $v \in G.V$ **do**
    // Add all vertices from the topology
    $G_{plan}.V \cup v$
  end for
  for all $e \in G.E$ **do**
    // Add a vertex for each edge in topology
    v_e := new vertex for e
    $G_{plan}.V \cup v_e$
    if e is kind of e.source and e.target before e relation **then**
      // add two edges from source vertex and target vertex to the relation vertex
      $G_{plan}.E \cup \{(e.source, v_e), (e.target, v_e)\}$
      // add two edges from target vertex to relation and relation to source vertex
    else if e is kind of e.source and e before e.target relation **then**
      $G_{plan}.E \cup \{(v_e, e.source), (e.target, v_e)\}$
    end if
  end for
end procedure

Verwendung Template Build Plan Fragmente Die in Abschnitt 4.2.2 definierten Fragmente werden verwendet, um die einzelnen Templates zu instanziieren. Für die Generierung eines Build-Plan Skeletts kann Algorithmus 4.1 verwendet werden, um ein Grundgerüst für die einzelnen Fragmente zu haben, in denen jeweils die passenden Fragmente eingebettet werden können. Template Build Plan Fragmente verwenden Elemente der TOSCA Topologie und referenzierte Daten aus der ganzen TOSCA Definitions dieser Elemente. Je nach Implementierung der Fragmente und des Build Plans müssen verschiedene Daten aus der TOSCA Definitions, innerhalb des Build Plans verfügbar sein. Ein Minimum an Input-Daten für den Plan werden alle externen Parameter sein, die einzelne Fragmente benötigen. Darunter werden bspw. Zugangsdaten für einen Cloud-Anbieter sein, die von Operation Activities innerhalb eines VM NodeType Fragments, für die Ausführung benötigt werden. Alle anderen Daten können extern gehalten werden, jedoch empfiehlt sich, Properties innerhalb des Plans als Variablen darzustellen. So reduziert sich die Komplexität der Fragmente, da sie keine zusätzlichen Aufrufe tätigen müssen.

Algorithmus 4.2 beschreibt in Pseudocode eine Möglichkeit, wie ein Plan-Generator DA/IA Provisioning Activities, für ein NodeTemplate finden kann. Dabei werden alle Knoten entlang eines Infrastruktur Pfades vom gegebenen NodeTemplate verwendet, um ein passendes

Fragment zu identifizieren. Pro Knoten wird zusätzlich dessen NodeType Hierarchie verwendet, um eine passende Menge von IA/DA Provisioning Activitites zu finden. Dabei liegt der Fokus des Algorithmus darauf, dass möglichst der erste Knoten für das Deployment verwendet wird.

Algorithmus 4.2 Algorithmus in Pseudocode für das Ermitteln von DA/IA Provisioning Activities für ein gegebenen Artefakttyp und NodeTemplate

```
procedure MatchArtifactActivities(ArtifactType, NodeTemplate, matchForIAFragment)
    // Input: ArtifactType of a DA or IA of the given NodeTemplate
    // and boolean value to distinguish betwenn IA Fragment and DA Fragment search
    for ∀x ∈ InfrastructurePath(NodeTemplate) do
        // search on every infrastructure path of the given NodeTemplate
        for i=1 to |x| do
            // every NodeTemplate on the infrastructure path
            for ∀ y ∈ parentTypes(π_i(x)) do
                // based on it's type hierarchy for a matching fragment
                if matchForIAFragment then
                    if ∃(ArtifactType,y)∈ IAProvisioningActivities then
                        return (ArtifactType,y) ∈ IAProvisioningActivities
                    end if
                else
                    if ∃(ArtifactType,y)∈ DAProvisioningActivities then
                        return (ArtifactType,y) ∈ DAProvisioningActivities
                    end if
                end if
            end for
        end for
    end for
    return null
end procedure
```

Für Operation und Consistency Activities müssen keine speziellen Algorithmen verwendet werden, um eine passende Menge zu identifizieren. Diese müssen nicht, bezogen auf die Topologie, ermittelt werden und werden direkt mit den entsprechenden Konstrukten (bspw. $(Operation,\ IA,\ NodeTemplate)$) verwendet.

4.4 Integration in OpenTOSCA

In diesem Abschnitt werden die gewählten Konzepte und konkrete Konstrukte vorgestellt, die für die Integration in OpenTOSCA verwendet werden. Dabei wird beschrieben, welche Mengen für BaseTypes, Infrastructure Nodes und Edges gewählt wurden. Zusätzlich werden

Template Build Plan Fragmente definiert, die benötigt werden, um die OpenTOSCA Test-Cases zu implementieren.

Überblick OpenTOSCA wird anhand von zwei Test-Cases getestet: Einer 1-Tier PHP Applikation, bei dem Datenbank und Applikation auf einer VM deployed werden. Eine 2-Tier Variante, bei der Datenbank und Applikation auf jeweils einer VM deployed werden. Beide Test-Cases nutzen WAR IA's, um eine VM auf Amazon EC2 zu instanziieren und das darauf vorinstallierte Linux zu managen. Abbildung 4.2 ist der 2-Tier Variante nachempfunden. Dabei bestehen beide Tiers aus einem Stack aus VM, OS, und Server, auf der dann jeweils die Applikation bzw. die SQL Datenbank installiert werden muss. Für die Bachelorarbeit wurde ein eigener Test-Case entworfen, der dieser 2-Tier Topologie nachempfunden wurde, um diesen dann auf den OpenTOSCA Test-Case zu übertragen. Dieses Vorgehen wurde gewählt, da die Test-Cases für OpenTOSCA Mehrdeutigkeiten und Annahmen besitzen, die ohne große Veränderungen der Topologie Elemente, nicht zu einem gewünschten Ergebnis geführt hätten. Für den Test-Case der Bachelorarbeiten, werden die nun folgenden Instanzen der Konzepte gewählt.

BaseTypes Für die Menge von BaseTypes werden folgende NodeTypes und RelationshipTypes verwendet:

$$baseTypes := \{\text{server, os, vm, application, node, connectsTo, deployedOn, dependsOn}\}$$

Server, OS, VM, Application stellen die grundlegenden Komponenten dar. Zusätzlich wird ein Node BaseType verwendet, um Komponenten einen BaseType zu geben, die sich nicht in andere Kategorien einteilen lassen, wie bspw. ein Modul. Für RelationshipTypes, die als BaseType verwendet werden, wurden Typen aus [BFL+12] gewählt. So kann connectsTo verwendet werden, um eine Verbindung zwischen zwei Komponenten zu modellieren. DeployedOn kann für Infrastruktur Beziehungen verwendet werden und dependsOn als allgemeine Abhängigkeit verwendet werden.

Infrastructure NodeTypes und RelationshipTypes Für Infrastructure NodeTypes/RelationshipTypes werden folgende Teilmengen der BaseTypes gewählt:

- $InfrastructureNodeTypes := \{\text{server, os, vm}\} \subseteq baseTypes$
- $InfrastructureRelationshipTypes := \{\text{deployedOn}\} \subseteq baseTypes$

Server, OS, VM sind offensichtliche Infrastruktur Komponenten. Mit diesen können alle möglichen Server, Betriebssysteme und Virtuelle Maschinen von diesen erben. So können NodeTypes dem Plan-Generator/Fragment-Entwickler vermitteln, dass diese als Kandidaten für das deployen von IA's und DA's verwendet werden können. Die deployedOn Relation ist ein allgemeiner Typ, um zu vermitteln dass eine Komponente auf eine andere deployed werden muss. So bildet diese Informationen, wie sich die Infrastruktur der Topologie zusammensetzt, ab.

Spezifische NodeTypes Für spezifische NodeTypes werden folgende Mengen verwendet:

- *NodeTypes* := {PhpApplication, PhpModule, MySQL Server, MySQL Database, ApacheWebServer, LinuxOS, AmazonVM}
- *parentTypes*(PhpApplication) := {application}
- *parentTypes*(PhpModule) := {node}
- *parentTypes*(MySQL Server) := {server}
- *parentTypes*(MySQL Database) := {node}
- *parentTypes*(ApacheWebServer) := {server}
- *parentTypes*(LinuxOS) := {os}
- *parentTypes*(AmazonVM) := {vm}

AmazonVM, LinuxOS, MySQL Server bzw. ApacheWebServer werden mittels deployedOn Relationen verbunden, um zwei Infrastruktur Stacks für jeweils die MySQL Database und PhpApplication zu modellieren. PhpModule wird mittels dependsOn Relation als Abhängigkeit für eine PhpApplication verwendet, um sicherzustellen, dass auf dem ApacheWebServer ein PhpModule intalliert wird.

ArtifactTypes Für ArtifactTypes wurde folgende Menge deklariert:

- *ArtifactTypes* := {WAR, Skript, ZIP, SQL, Package }
- *deployableIntoEnvironment*(WAR) := *true*
- *deployableIntoEnvironment*(Skript, ZIP, SQL, Package) := *false*

OpenTOSCA ermöglicht es WAR Files in dessen TOSCA-Management-Umgebung zu installieren. Während Skripte und ZIP Dateien, für den Test-Case, auf die Komponenten installiert werden müssen. SQL sind Datenbank Dateien, die auf einem MySQL Server deployed werden können und Package stehen für Linux Packete, die die Implementierung von ApacheWebServer und MySQL Server stellen.

IA/DA Deployment Um zu bestimmen welche Artefakte auf eine Infrastruktur Node deployed werden können, wählen wir „Auf Basis von NodeTypes“(Siehe Abschnit 4.1.3) und definieren folgende Mengen:

- *deployableTypes*(ApacheWebServer) := {ZIP}
- *deployableTypes*(LinuxOS, AmazonVM) := {*}
- *deployableTypes*(MySQL Server) := {SQL}

Die folgenden Mengen erlauben es, dass auf ApacheWebServern Applikationen im ZIP Format installiert werden können und auf einem MySQL Server SQL Dateien die Datenbank Definitionen enthalten. Für LinuxOS und AmazonVM wurde eine Wildcard * gesetzt, das bedeutet, dass auf ihnen prinzipiell alles installiert werden kann.

Wahl der Parameter Bei der Wahl von Parametern, wurde der simpelste Ansatz gewählt. Wenn eine Operation eines NodeTypes, einen Parameter besitzt, der den gleichen Namen eines Keys innerhalb der NodeType Properties hat, wird dieser als Parameterwert verwendet. Somit gilt das ein Kind-Element des Property Root-Elements, möglichst keine weiteren Kind-Elemente besitzt. So kann der Wert innerhalb des Elements direkt als Parameterwert verwendet werden. Parameter für die keine Properties gefunden werden, werden als Input des Plans definiert und somit als externe Parameter verwendet.

Template Build Plan Fragmente Um nun anhand der gegebenen Typen, eine Topologie automatisiert zu provisionieren, werden folgende Template Build Plan Fragmente definiert:

- $iaFragments := \{$(Skript, {LinuxOS, AmazonVM})$\}$
- $daFragments := \{$(ZIP, {LinuxOS, AmazonVM, ApacheWebServer}), (SQL, MySQL Server), (Package, LinuxOS)$\}$
- $opFragments := \{$(WAR), (Skript)$\}$
- $conFragments := \{$(*)$\}$

Im Test-Case werden für das Instanziieren der Datenbank und Applikation Skripte verwendet, die auf einem Zielsystem innerhalb der Topologie installiert werden müssen. Dazu wird ein Fragment das Skripte, die in einem IA definiert sind, auf ein LinuxOS oder eine AmazonVM hochladen können, benutzt. DA's werden hier ZIP/SQL Dateien und Linux Packete sein. Für ZIP wird ein Fragment, dass diese auf ein LinuxOS, AmazonVM oder ApacheWebServer installieren kann, verwendet. Für SQL Dateien wird ein Fragment verwendet, dass solche auf einen MySQL Server installieren kann. Für Packete wird eins entwickelt, dass LinuxOS beherrscht. Operations Fragmente sollen möglichst generisch sein, so werden zwei Fragmente benutzt. Eine kann Web Services aufrufen und die andere Skripte auf einem Zielsystem. Für Consistency Fragmente wird eins verwendet, dass für jedes NodeTemplate (bzw. RelationshipTemplate) die Instanzdaten auf einer externen Komponente verwalten kann.

Operationsreihenfolge Die Operationsreihenfolge eines NodeTemplates oder RelationshipTemplates wird Interface-getrieben ermittelt. Es wird das TOSCA Lifecycle Interface [TCa] verwendet. So gilt folgende feste Menge von Interfaces:

$$Interfaces := P(\{\text{Install, Configure, Start, Stop, Uninstall}\}) \setminus (\{\text{Stop}\} \cup \{\text{Uninstall}\} \cup \{\text{Stop, Uninstall}\})$$

Wir erlauben alle Kombinationen des Lifecycle Interfaces, bis auf diese, die nur Operationen besitzen, die für das Beenden einer Komponente bzw. Relation vorgesehen sind.

4.5 Architektur

In diesem Abschnitt werden Anforderungen an Architektur und Implementierung beschrieben und die Architektur für den Plan-Generator erläutert. Es werden erst Anforderungen, wie etwa das WS-BPEL 2.0 zur Implementierung von Build Plänen verwendet werden soll, aufgelistet. Anschließend wird ein Überblick über die Architektur und die enthaltenen Komponenten dieser gegeben.

4.5.1 Anforderungen

Anforderungenen, die an die Implementierung gestellt wurden, werden in diesem Abschnitt aufgelistet. Zusätzlich werden Entscheidungen bzgl. der Implementierung erläutert.

Eine Anforderung an die Implementierung war, dass diese eigenständig lauffähig und zusätzlich in andere Software integrierbar ist. So soll es möglich sein, dass TOSCA Modellierungswerkzeuge Pläne generieren können, um eventuell fehlende Aktivitäten identifizieren zu können, bspw. wenn ein Template Build Plan Fragment fehlt. Als eigenständige Komponente kann diese, mit einer geeigneten API, verwendet werden, um Pläne zur Laufzeit eines Provisionings anzufordern. Dabei sollte primär eine spätere Integration in den OpenTOSCA Container [AI] erfolgen, der im Laufe des Cloud Cycle Projekts [Cyc] entwickelt wurde. Der OpenTOSCA Container ist ein TOSCA-Container, der es erlaubt CSARs zu importieren und zu verarbeiten, um die in den CSARs vorhanden Topologien zu instanziieren. Dieser wurde als OSGi Anwendung entwickelt und besteht somit aus einer Reihe von OSGi Bundles. OSGi mit seinem Komponentenmodell erlaubt es einzelne OSGi Bundles zu verwenden. Für den Prototyp wurden die *Core*-Komponenten des openTOSCA Containers als Basis für den Prototyp gewählt. Diese implementieren die Datenhaltung des Containers, um CSARs zu speichern und zu laden. Dabei wird die Core per *Importern* und *Exportern* eingebunden, um eine weitere Trennung von Logik und Integration zu erreichen (siehe 4.5).

Das vorgestellte Konzept dieser Arbeit ermöglicht eine erweiterbare Implementierung, welche die Verarbeitung verschiedener Tupel (bspw. $(DA, InfrastructureNodeType)$) durch Plugins erledigen kann. Da die Anforderung den Prototyp in den openTOSCA Container zu integrieren OSGi benötigt, lässt sich ein Pluginmechanimus mittels *OSGi Services* realisieren, um genau die gewünschte Erweiterbarkeit zu ermöglichen.

Eine weitere Anforderung an den Prototyp war, dass generierte Pläne in WS-BPEL 2.0 verfasst sind. BPEL ist eine in XML implementierte Workflow Sprache, das standardisierte XML-Technologien wie WSDL 1.1, XML Schema 1.0, XPath 1.0, XSLT 1.0 und Infoset verwendet. Die Implementierung muss dabei mindestens Funktionalitäten für BPEL, WSDL und XML Schema zur Verfügung stellen. Für BPEL selbst muss es Möglichkeiten geben Elemente der Sprache einzufügen. WSDL wird in BPEL genutzt, um Schnittstellen zwischen einem

WebService und dem BPEL Prozess zu deklarieren. XML Schema ist das Typsystem von BPEL (wie auch von WSDL). WSDL Schnittstellen müssen für den Prozess registriert werden können und sowie XML Schema Typen.

Um einen Überblick über die Anforderungen zu geben, wurden diese nochmal hier zusammen gefasst:

- Eigenständig Lauffähig
- Integrierbar
- Erweiterbar durch Plugins
- Leichte Integration in den openTOSCA Container
- Generierung von TOSCA BuildPlänen implementiert in WS-BPEL 2.0

4.5.2 Architektur Überblick

Für die Architektur musste beachtet werden, dass der Prototyp als eigenständige Komponente lauffähig ist. Darüber hinaus muss der Generator in andere Software integrierbar sein. Für diesen Umstand wurde ein *Integration Layer* vorgesehen, dieser soll ermöglichen die eigentliche Logik, Build Pläne zu generieren, von den möglichen Umgebungen zu trennen. Dabei besteht der Integration Layer aus *Importern* und *Exportern*. Importer werden verwendet, um eine CSAR aus der Umgebung dem *Plan Builder* zu übergeben. Exporter werden verwendet, um die CSAR, gepackt mit dem neuen Build Plan, wieder in ihre Umgebung zu laden. Importer und Exporter Paare müssen speziell für die Umgebung, aus der sie CSARs für den Plan Builder bereitstellen, entwickelt und im Integration Layer registriert werden. Der *Plan Builder* ist dafür zuständig, die erhaltene CSAR mit einem Build Plan zu erweitern. Dabei wird dieser ein Plan Modell Skelett aus leeren Template Build Plan Fragmenten generieren. Der Builder wird dann mittels der *Template Fragment Library* und den darin registrierten *Plugins*, die leeren Template Build Plan Fragmente mit Aktivitäten ausfüllen. Der Plan Builder und die Plugins bearbeiten den Build Plan mittels der *Facade*. Diese ist dafür vorgesehen, die Komplexität von Plansprachen und ihren Gegebenheiten zu reduzieren. Die Funktionalität der *Facade* besteht aus dem Erstellen von Plan Skeletten mit leeren *Template Build Plan Fragmenten*, dem Bearbeiten der Fragmente, dem Registrieren von *PortTypes* im Falle eines BPEL Plans, etc. Das Ziel der *Facade*, neben dem Reduzieren von Komplexität, ist auch die vorgestellten Konzepte einzuhalten. Nachdem der *Plan Builder* den Build Plan fertig gestellt hat, wird dieser über den *Integration Layer* an einen *Exporter* übergeben, der die nun erweiterte CSAR wieder in seiner Umgebung ablegt. In Abbildung 4.6 wird die Architektur dargestellt.

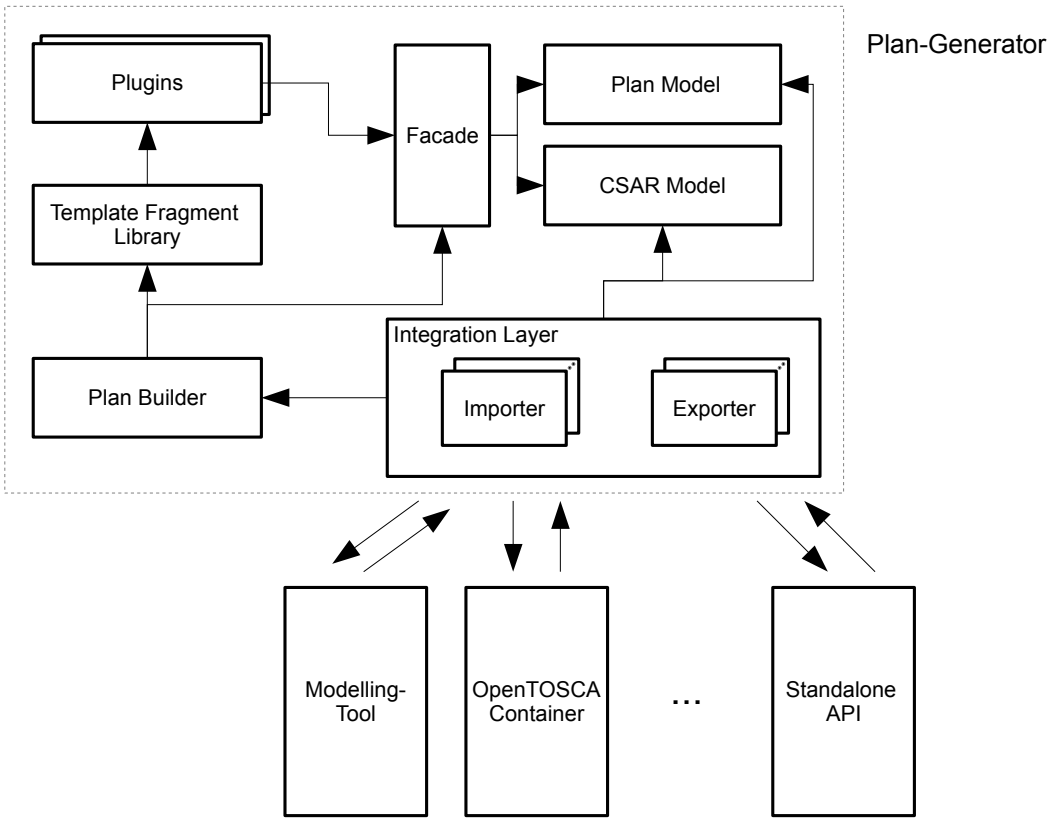

Abbildung 4.6: Plan-Generator Architektur

Plan Builder Die Plan Builder Komponente ist für die Orchestrierung von einzelnen Template Build Plan Fragmenten zuständig. Diese werden aus der Template Fragment Library angefordert. Dabei muss der Plan Builder nicht entspechend Algorithmus 4.1 aus Abschnitt 4.3 arbeiten, dieser kann auch Set Cover und Planning Algorithmen verwenden, um mit den Fragmenten aus der Library einen Build Plan zu generieren.

Template Fragment Library Die Template Fragment Library ist dafür vorgesehen, IA/DA Provisioning Activities, Operation Activitites und Consistency Activities, mittels Plugins, dem Plan Builder verfügbar zu machen. Die Plan Builder Komponente würde mit TOSCA Elementen anfragen, ob in der Library passende Fragmente vorhanden sind. Wenn ein Plugin ein passendes Fragment besitzt, würde der Plan Builder dieser vermitteln, dass die Aktivitäten des Fragments über die Facade dem Build Plan hinzugefügt werden sollen.

Plugins Ein Plugin registriert sich bei der Template Fragment Library, um ein oder mehrere Fragmente anbieten zu können. So kann ein Plugin generische Fragmente zur Verfügung stellen, bspw. ein Operation Activities Fragment für WSDL Operationen.

Facade Die Facade ist dafür gedacht, Low-Level Logik auf den Modellen zu kapseln. Somit ist es eines der Hauptaufgaben, das Entwickeln von Plugins zu erleichtern. Die Methoden der Facade beeinflussen auch welche Algorithmen in der Plan Builder Komponente implementiert werden können. So werden Methoden in ihr benötigt, die nicht nur leere Template Build Plan Fragmente generieren, falls Planning Algorithmen verwendet werden.

Plan Model Das Plan Model entspricht dem Modell, indem der Build Plan implementiert wird. Das Modell kann alle Konstrukte der Implementierungssprache enthalten, da die Facade Low-Level Operationen auf dem Modell abkapselt.

CSAR Model Die CSAR Model Komponente entspricht einem Modell, einer ganzen CSAR. Unter einer ganzen CSAR versteht sich bspw. die Möglichkeit, aus der Manifest dieser auszulesen, welches Definitions Dokument das *Entry-Definitions* Dokument ist. Zudem soll es möglich sein alle Elemente einer TOSCA Definitions, verwenden zu können. Importierte Elemente (Definitions, NodeTypes, etc.) sollen schon innerhalb der Entry-Definitions aufgelöst sein. Es soll möglich sein, Plan Elemente dem Modell hinzuzufügen.

Integration Layer Der Integration Layer ist dazu gedacht, den Plan-Generator in andere Software integrieren zu können. Im Layer werden Importer und Exporter verwendet, die jeweils für die Umgebung in die, der Plan-Generator integriert wird, entwickelt werden. So kann es Importer/Exporter Paare geben die den Generator, in ein TOSCA Modelling-Tool, in den OpenTOSCA Container integrieren oder eigenständig als standalone Komponente anbieten, um Build Plan Generierung als einen Dienst anzubieten.

5 Design und Implementierung

Dieser Abschnitt beschreibt den in dieser Bachelorarbeit entwickelten Prototyp der TOSCA Build-Pläne, implementiert in WS-BPEL 2.0, erstellt. Dabei werden Entwurf der Komponenten der Architektur und Implementierung dieser erläutert.

5.1 Build Plan BPEL Modell

Für das Build Plan Modell wurde ein DOM-Document, mit den darin vorgesehenen DOM-Elementen, in Klassen zusammengefasst und mit zusätzlichen Klassen für WSDL und einem Apache ODE Deployment Descriptor gewählt. Die *BuildPlan*-Klasse besteht aus einem *DOM-Document*, einer Reihe von *DOM-Elementen*, einem *GenericWsdlWrapper*, einem Deployment Descriptor *Deploy*, einer *AbstractDefinitions* und einer Liste von *TemplateBuildPlan*. Das DOM-Document repräsentiert den ganzen BPEL Prozess und alle Elemente, die innerhalb der Erstellung des Build Plans benötigt werden, werden daraus generiert und verwendet. Die DOM-Elemente entsprechen Elementen aus BPEL. Darunter ist ein *process* Element für den Prozess selbst, ein *extensions* Element für Extensions die Plugins evtl. registrieren. Weiterhin existiert eine Liste von *import* Elementen fürs Importieren von WSDL oder XSD, ein *partnerLinks* Element um importierte WSDL Schnittstellen dem Prozess verfügbar zu machen. Das *variables* Element existiert für Variablen, ein *sequence* Element für den Haupt-Kontrollfluss und einem *assign* Element, um Property Variablen zu initialisieren. Der Input/Output wird mit einem *receive* und *invoke* für asychrone Kommunkation realisiert. Für die Template Build Plan Fragmente, wird ein *flow* Element verwendet, so kann der Plan Builder einen Build Plan Graph darin realisieren. Für die Abstraktion einer WSDL (die für den BPEL Prozess), wurde eine generische WSDL Datei gewählt, die mit Platzhaltern der Form *{TAGNAME}* versehen wurde. Die Klasse GenericWsdlWrapper lädt bei Instanziierung diese Datei und füllt passende Stellen mit Daten, die mittels den Methoden der Klasse überreicht werden. Nachdem der Build Plan fertiggestellt wurde, werden die Tags entfernt und als *String* dem System übergeben. Der Deployment Descriptor entspricht dem von Apache ODE [ODEb] und wurde mit *xjc* [XBBC] generiert. ODE stellt dazu ein XML Schema bereit [odea]. Wenn ein Plugin ein *BPEL-Invoke*-Element intergriert, muss dieser evtl. angepasst werden. Das BPEL Build Plan Modell besitzt noch eine Referenz auf ein *AbstractDefinitions* (Siehe Abschnitt 5.2) Objekt, da ein Build Plan möglicherweise Informationen aus dem ganzen Definitions Dokument braucht, darunter NodeTypes, RelationshipTypes, etc.. Dabei können diese auch in importierten Dokumenten deklariert worden sein. Oft wird deswegen das Definitions Dokument dem *Entry-Definitions* Dokument in der *Tosca.meta* Datei in der CSAR entsprechen.

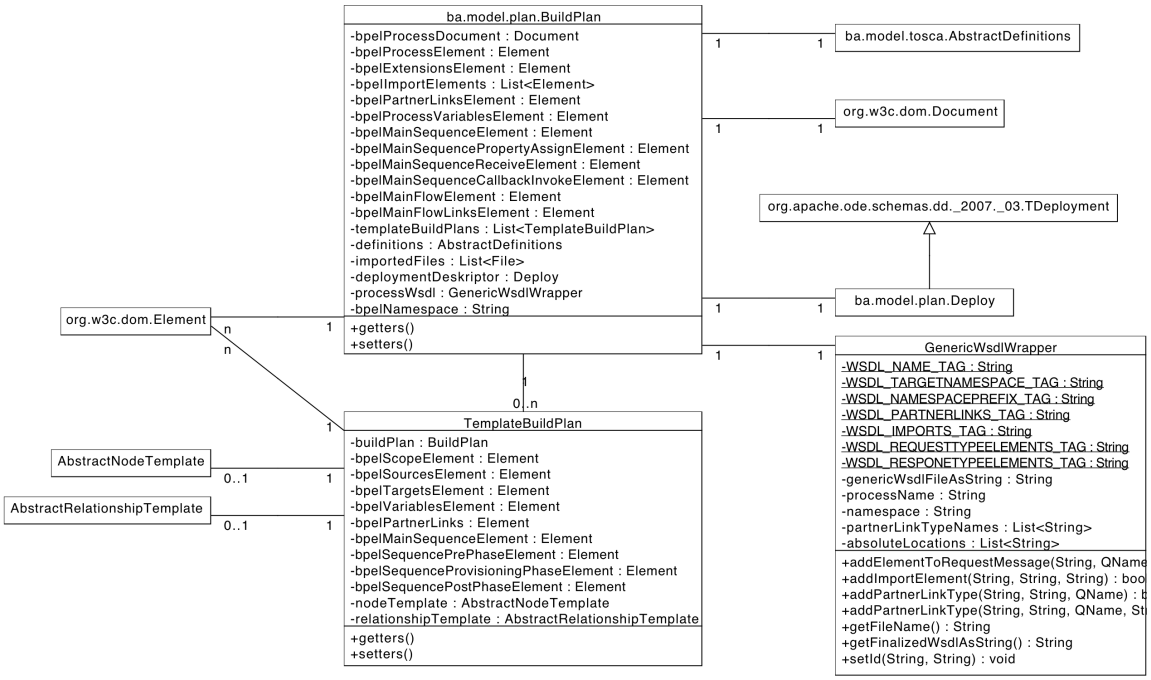

Abbildung 5.1: UML Klassen Diagramm eines Build Plan BPEL Modells innerhalb des Prototyps

Ein BuildPlan Objekt hat eine Liste von Referenzen auf TemplateBuildPlan Objekte. Diese entsprechen den *TemplateBuildPlanFragments* aus Kapitel 4 (4.2.2). Ein TemplateBuildPlan besteht aus einer Reihe von DOM-Elementen, einer Referenz auf den BuildPlan in dem sie verwendet werden und besitzen entweder eine Referenz auf ein *AbstractNodeTemplate* oder *AbstractRelationshipTemplate*, da ein TemplateBuildPlan für ein Template zuständig ist. Ein Template Build Plan Fragment wurde in BPEL als *scope* mit einer *sequence* mit jeweils einer *sequence* für *Pre Phase*, *Provisioning Phase* und *Post Phase* modelliert. *Sources*, *targets*, *variables* und *partnerLinks* des *scopes* werden auch als DOM-Element festgehalten. Die Methoden eines TemplateBuildPlan sind reine Getter- und Setter-Methoden, da die Logik in der Fassade gehalten wird. In Abbildung 5.1 wird das Build Plan Modell anhand von einem UML Diagramm verdeutlicht.

5.2 CSAR Modell

Das CSAR Modell entspricht einer Menge von abstrakten Klassen, die die Struktur und Semantik eines TOSCA Definitions Dokuments erfassen. Dabei wurden bewusst nicht JAXB Klassen verwendet (bzw. durch *xjc* generierte Klassen), damit der Plan Builder sich nicht um die Eigenarten dieser kümmern muss. Zusätzlich erlaubt es den *Importern* aus allen

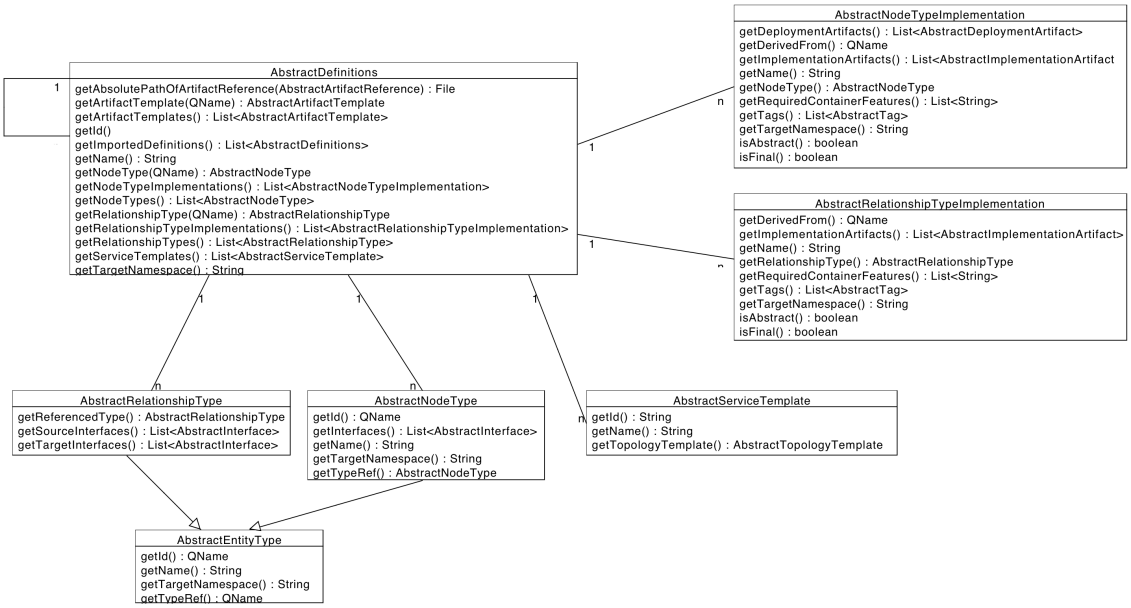

Abbildung 5.2: UML Klassen Diagramm Auschnitt eines CSAR/Definitions Modells innerhalb des Prototyps

möglichen Umgebungen das Modell zu implementieren. Das Modell wird hier CSAR Modell genannt, da zusätzliche Funktionen implementiert wurden, wie das Dereferenzieren von relativen Pfaden innerhalb von *ArtifactTemplate* zu absoluten Pfaden. Ein weiterer Grund für die Entscheidung, dass das Modell abstrakt implementiert wurde, ist, dass komfortabel *NodeTypeImplementation* für *NodeTemplates* ermittelt werden können. Abbildung 5.2 gibt einen Ausschnitt aus dem Datenmodell, da das gesamte TOSCA Modell hier den Rahmen sprengen und hier nur ein Einblick zum Verständnis nötig ist.

Die Topologie ist durch die Wahl von abstrakten Klassen effizient traversierbar und erlaubt so *Infrastructure Nodes* und *Infrastructure Edges* leicht zu finden. Zusätzlich sind Methoden vorhanden, die Quellen und Blätter einer Topologie finden.

5.3 Facade

Die Fassade ist dafür gedacht die große Anzahl von XML Elementen des Build Plans und das Verwalten dieser zu vereinfachen. Die Fassade ist an das gleichnamige Design-Pattern [GHJV95] angelehnt. Hierbei bedeutet angelehnt, dass die Fassade aus vier Klassen besteht, mit jeweils verschiedener Granularität bzgl. der Funktionen auf dem Datenmodell. Die Klassen sind namentlich *BuildPlanHandler*, *TemplateBuildPlanHandler*, *BpelProcessHandler* und *BPELTemplateScopeHandler*. Die ersten beiden Klassen sind High-Level Abstraktionen für den Planbuilder und die Plugins. Die letzteren sind näher zum XML des BPEL Prozesses

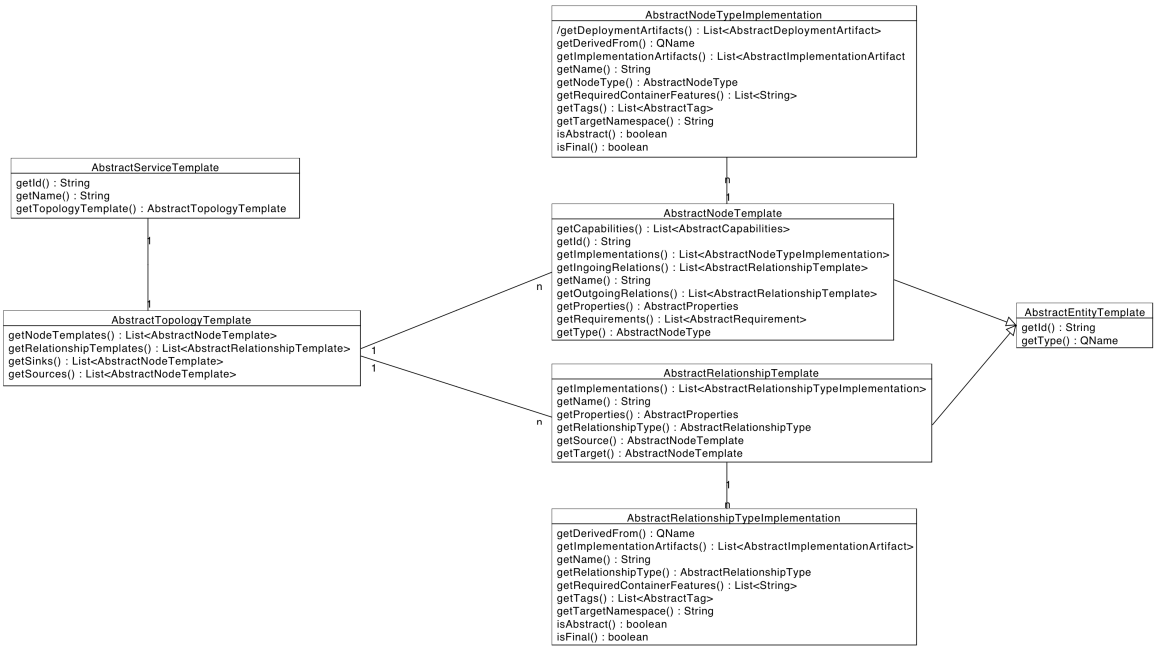

Abbildung 5.3: UML Klassen Diagramm Auschnitt eines Topology Modells innerhalb des Prototyps

und sind somit für fein-granulare Funktionen vorgesehen. Die Abbildung 5.4 gibt einen Überblick als UML Diagramm.

Die *BPELTemplateScopeHandler* Klasse ist dafür zuständig, die XML Elemente innerhalb eines BPEL Template Scopes zu verwalten. Die Klasse kann einen Template Scope entsprechend eines *Template Build Plan Fragments* (*Pre Phase*, *Provisioning Phase*, *Post Phase*) initialisieren, BPEL *sources* und *targets* hinzufügen und *variables* und *partnerLinks* registrieren. *BPELTemplateScopeHandler* ist für fein-granulare Funktionen zuständig.

BPELProcessHandler ist eine Klasse, die das BPEL Dokument (also XML) behandelt. Dabei ist die Zuständigkeit eine Stufe höher als die der *BPELTemplateScopeHandler* Klasse. Sie kann einen leeren BPEL Prozess initialisieren. Dabei wird das oberste *sequence* eingefügt, dessen *assign* für Property-Variablen und ein *flow* Element für Template Build Plan Scopes generiert. Die Klasse ist auch zuständig für globale Partnerlinks, Variablen, Extensions und importierte Dokumente verantwortlich.

Die Klasse *TemplateBuildPlanHandler* bietet Planbuilder und den Plugins Funktionen an, die XML abkapseln. Dafür benutzt die Klasse Funktionen von *BPELTemplateScopeHandler* und ermöglicht somit Funktionalität eine Stufe über dieser, bspw. kann sie zwei Template Build Pläne mit einem BPEL Link verbinden. Darüber hinaus ist die Klasse dafür verantwortlich Template Build Pläne für NodeTemplates und RelationshipTemplates zu initialisieren.

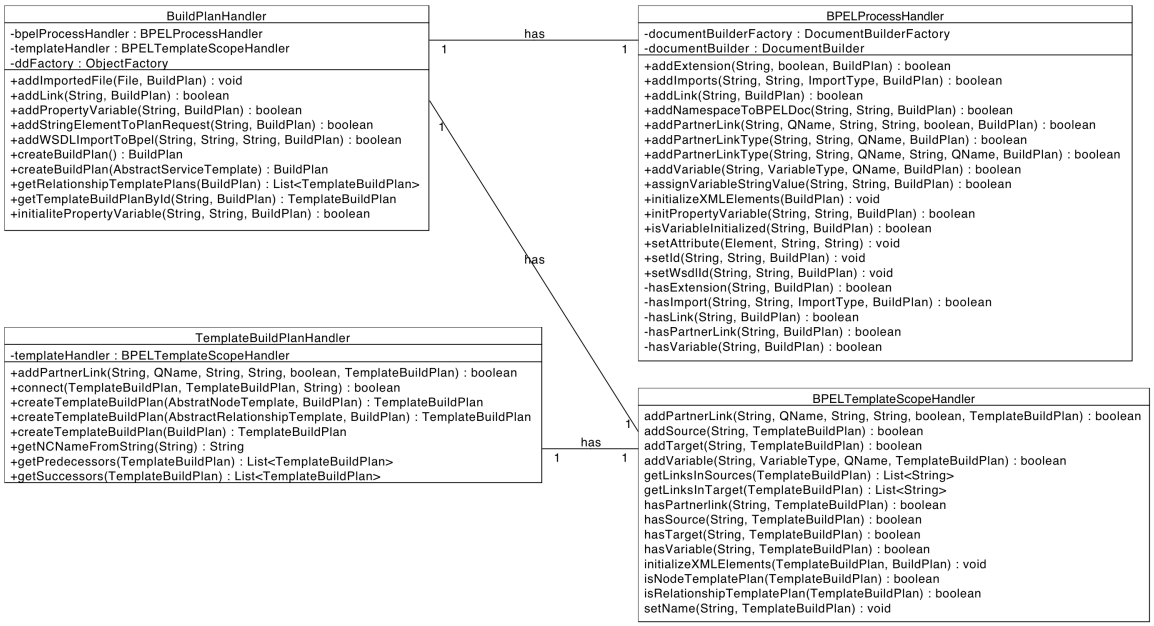

Abbildung 5.4: UML Klassen Diagramm der Fassade zwischen Modell und Planbuilder Komponenten

BuildPlanHandler wird verwendet, um Funktionen auf Build Plan/Graph Ebene auszuführen. Bspw. können Variablen für die Input-Message des Plans registriert werden. Die Klasse benutzt die Funktionen der Klassen *BPELTemplateScopeHandler* und *BPELProcessHandler*.

5.4 Plugin Context

Damit Plugins nicht Zugriff auf die ganze Topologie besitzen, wurde die Klasse *TemplatePlanContext* im Sinne des Konzepts des Information Hidings entwickelt. Die Klasse hat vollen Zugriff auf die Fassade des Modells, kapselt jedoch den Zugriff auf ein NodeTemplate bzw. sein Template Build Plan und dessen Phasen ab. Darüber hinaus werden Methoden angeboten, die Plugins das Hinzufügen von Aktivitäten sowie das Registrieren von Extensions und Variablen, vereinfachen soll. *TemplatePlanContext* verwaltet den Zustand eines Template Build Plan innerhalb des Build Plans und wird vom Plan Builder instanziiert und den Plugins übergeben, damit sie ihre Arbeit mit dem gegeben Kontext verrichten. Abbildung 5.5 gibt eine Veranschaulichung in UML.

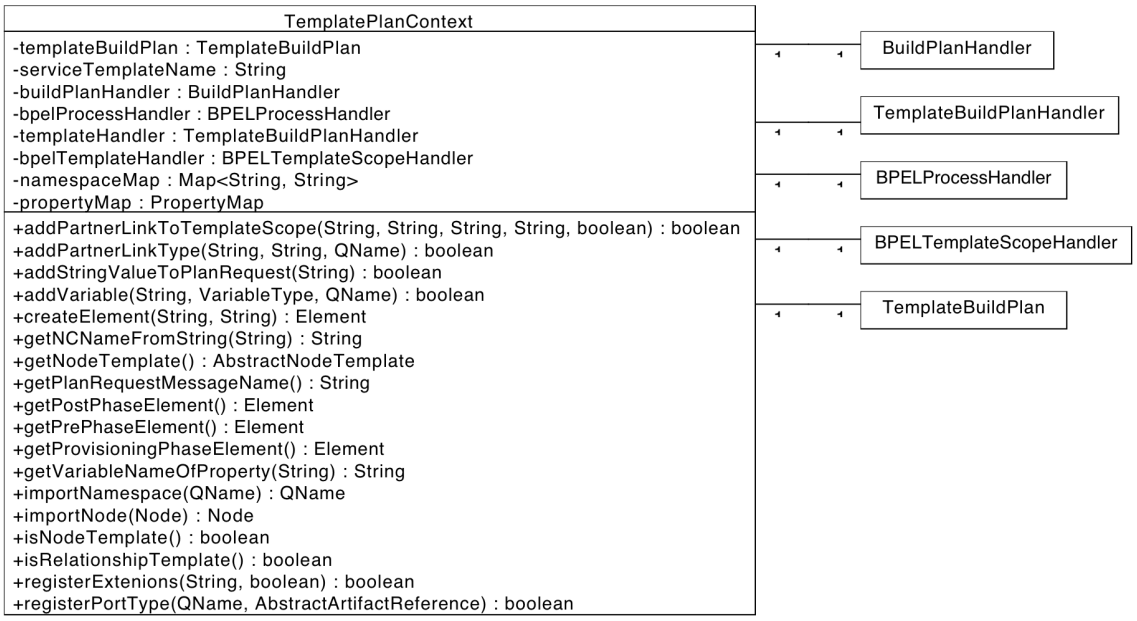

Abbildung 5.5: UML Klassen Diagramm eines TemplatePlanContexts für Plugins, die mithilfe dieser BPEL Fragmente zu einem Template Build Plan hinzufügen können

5.5 Plan Builder und Plugin Registry

PlanBuilder ist die Klasse, die für High-Level Logik beim Erstellen eines Build Plans zuständig ist. Diese generiert ein BPEL Prozess Skelett, entsprechend Algorithmus 4.1, initialisiert Properties in globalen BPEL Variablen mithilfe der *PropertyVariableInitializer* Klasse. Das Skelett besteht nach diesen Schritten aus BPEL Scopes innerhalb des Haupt Flow Elements mit leeren Sequence Elementen für Pre Phase, Provisioning Phase und Post Phase. Der *PlanBuilder* beginnt danach mittels der *PluginRegistry* passende Plugins für die Phasen zu wählen und diese auf die Phasen mit einem *TemplatePlanContext* anzusetzen. Dabei sind Plugins dafür verantwortlich passende Schritte zu tätigen, um einen korrekten Ablauf innerhalb einer Phase zu erhalten. Die Plugins werden durch das Interface, das sie implementieren kategoriesiert (*IPlanBuilderPrePhaseIAPlugin*, *IPlanBuilderPrePhaseDAPlugin*, *IPlanBuilderProvPhaseOperationPlugin* und *IPlanBuilderPostPhasePlugin*) und müssen entsprechend der Semantik dieser arbeiten. Nachdem alle Knoten des Plans bearbeitet wurden, wird der gesamt BPEL Prozess noch angepasst, bspw. werden an Stellen im Plan in der keine Aktivität eingefügt werden konnte (bspw. weil kein Plugin fähig dazu war), mit *empty* Aktivitäten aufgefüllt. So kann der Plan-Generator auch genutzt werden, um Templates für Topologien zu generieren.

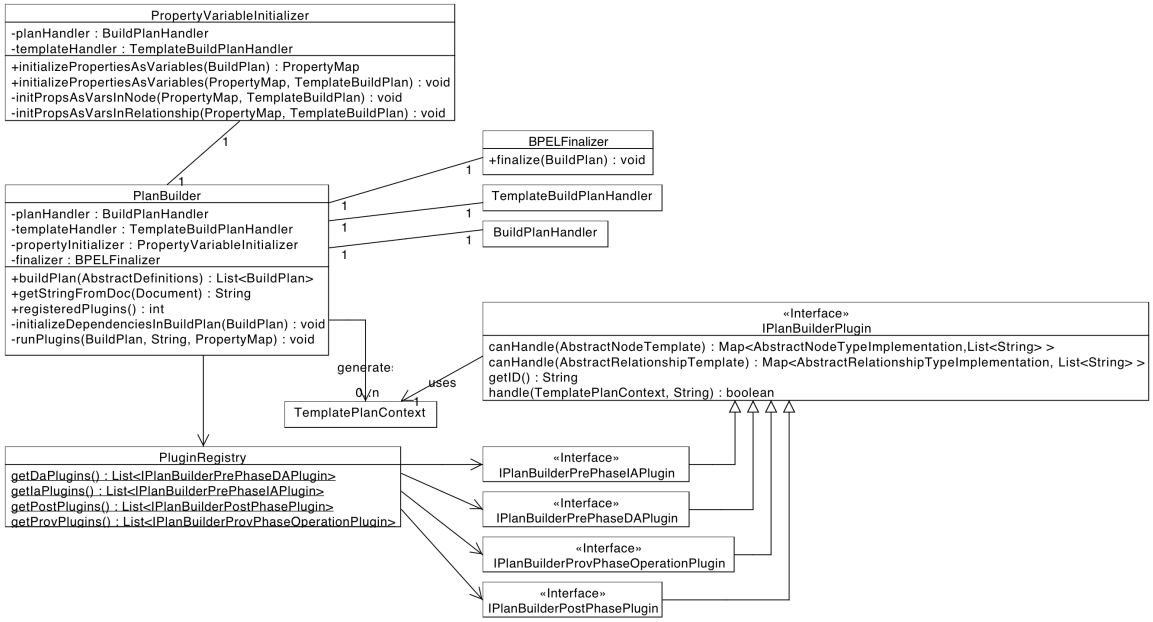

Abbildung 5.6: UML Klassen Diagramm der PlanBuilder Komponente

6 Zusammenfassung und Ausblick

Im Laufe dieser Bachelorarbeit wurden aktuelle Herausforderungen bei der Generierung von Build Plänen für TOSCA Topologien beschrieben. Dabei wurde auf die Rolle von Basistypen eingegangen und wie diese verwendet werden können, um die Infrastruktur einer TOSCA Topologie zu identifizieren. Mit diesen Definitionen wurden nachträglich Lösungsvorschläge gegeben, um das Aufsetzen von Artefakten und Aufrufen von Operationen zu ermöglichen. Auf diesen Konzepten aufbauend, können Low-Level Workflow-Fragmente entwickelt werden. Dabei können diese so kategorisiert werden, dass High-Level Workflows aus diesen Fragmenten generiert werden könnnen. Es existieren Fragmente um Artefakte auf eine Komponente zu laden, wie auch solche, die für das Aufrufen einer Operation einer Komponente verantwortlich sind. Diese Fragmente sind, da sie auf eine bestimmte Aktion beschränkt sind, in verschiedenen Vorgehen wiederverwendbar, so etwa in Set Cover und Planning Algorithmen. Es wurde jedoch ein Vorgehen vorgestellt, aufbauend auf Basistypen und den Fragmenten, dass einen Build Plan aus der Topologie generiert werden kann, indem Kanten der Topologie invertiert werden. Dabei werden einzelne Komponenten und Relationen in verschiedenen Phasen, die Fragmente enthalten, provisioniert.

Ausblick

Diese Arbeit beschreibt ein konzeptionelles Vorgehen für das Provisioning von TOSCA Topologien und einen Prototyp für diese Konzepte. Viele dieser Konzepte bauen auf eine wohldefinierte Menge von Basistypen auf, jedoch ist noch nicht wirklich klar, wie diese für einen Standard wie TOSCA auszusehen haben. Zudem ist die Rolle von Capability/Requirement-Types bzgl. Basistypen nicht klar. Beispielsweise stellt sich die Frage, wie man mit NodeTypes umgehen soll. Ein Beispiel dafür wäre etwa ein BaseType „Server“ mit einem CapabilitiyType „WAR“ und einem NodeType „Tomcat“, der impliziert das er WAR Dateien auf sich deployen lässt. Somit ist hier die Kernfrage ob, NodeType Semantik durch CapabilityTypes substituiert werden können.

Zusätzlich fehlen in TOSCA Konstrukte, die bspw. für das Identifizieren von Schnittstellen der NodeType Operationen nötig sind. Entwickler von TOSCA Templates, müssen diese mit Meta-Daten anreichern, um jegliche Topologie mit einer automatischen Generierung von Plänen zu provisionieren. Jedoch verhindert das die Interoperabilität zwischen Plan-Generatoren, da für jeden neuen Datensatz an Meta-Daten, dieser erweitert werden müsste. Um dem entgegenzuwirken, wäre es eine Möglichkeit, die in Abschnitt 4.1 beschriebenen Probleme, durch eine einheitliche Definition zu umfassen. Dabei sollte diese für möglichst

flexible Entwicklung geeignet sein. So wäre es von Vorteil wenn, verschiedene Ansätze unterstützt würden.

Wenn der TOSCA Standard, der noch in Entwicklung ist, es ermöglicht die Probleme aus Abschnitt 4.1 zu lösen, wäre es möglich, bereits geschriebene Fragmente für verschiedene Plan-Generatoren verfügbar zu machen. So könnten Fragment-Datenbanken existieren die zur Laufzeit angefragt werden, um eine Topologie zu provisionieren.

Der entwickelte Prototyp beherrscht durch die, verschiedenen Möglichkeiten die Topologie zu interpretieren, eine begrenzte Verwendbarkeit (namentlich Topologien bestehenden aus Skripten, WebServices, Zips und Packages). Dies kann zwar durch Entwickeln von Fragmenten erweitert werden, jedoch ist mit Fehlen von weiteren Konstrukten innerhalb von TOSCA, nie eine volle Interoperabilität möglich. Dies wäre der wichtigste Aspekt, den man als nächstes Angehen sollte.

Literaturverzeichnis

[AI] U. S. I. für Architektur von Anwendungssystemen (IAAS). OpenTOSCA - Open Source Laufzeitumgebung für TOSCA. http://www.iaas.uni-stuttgart.de/OpenTOSCA/. (Zitiert auf den Seiten 12 und 47)

[BBC+07] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, J. Siméon. XML Path Language (XPath) 2.0 (W3C Recommendation), 2007. URL http://www.w3.org/TR/xpath20/. (Zitiert auf Seite 30)

[BBKL13] U. Breitenbücher, T. Binaz, O. Kopp, F. Leymann. Pattern-based Runtime Management of Composite Cloud Applications. In *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013*. SciTePress, 2013. (Zitiert auf Seite 13)

[BFL+12] T. Binz, C. Fehling, F. Leymann, A. Nowak, D. Schumm. Formalizing the Cloud through Enterprise Topology Graphs. In *Proceedings of 2012 IEEE International Conference on Cloud Computing*. IEEE Computer Society Conference Publishing Services, 2012. (Zitiert auf den Seiten 11, 21, 23, 30 und 44)

[BHM+04] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard. Web Services Architecture. W3C Note NOTE-ws-arch-20040211, World Wide Web Consortium, 2004. URL http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/. (Zitiert auf Seite 11)

[BPSM+08] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). World Wide Web Consortium, Recommendation REC-xml-20081126, 2008. (Zitiert auf Seite 11)

[Byl91] T. Bylander. Complexity Results for Planning. In *IJCAI*, S. 274–279. 1991. (Zitiert auf Seite 17)

[CCMW01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web Service Definition Language (WSDL). Technischer Bericht, W3C, 2001. URL http://www.w3.org/TR/wsdl. (Zitiert auf Seite 12)

[Cyc] C. Cycle. http://www.cloudcycle.org/. (Zitiert auf Seite 47)

[EEKS11] T. Eilam, M. Elder, A. V. Konstantinou, E. C. Snible. Pattern-based composite application deployment. In N. Agoulmine, C. Bartolini, T. Pfeifer, D. O'Sullivan, Herausgeber, *Integrated Network Management*, S. 217–224. IEEE, 2011. URL http://dblp.uni-trier.de/db/conf/im/im2011.html#EilamEKS11. (Zitiert auf Seite 14)

[Falo1] D. C. Fallside. *XML Schema Part o: Primer*. W3C, 2001. URL http://www.w3.org/TR/xmlschema-0. W3C Recommendation. (Zitiert auf den Seiten 12 und 24)

[FEKo7] M. Ford, A. Endpoints, C. Keller. WS-BPEL Extension for People (BPEL4People), Version 1.0. 2007. (Zitiert auf Seite 32)

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. (Zitiert auf Seite 53)

[GIP+98] M. Ghallab, C. K. Isi, S. Penberthy, D. E. Smith, Y. Sun, D. Weld. PDDL - The Planning Domain Definition Language. Technischer Bericht, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212. (Zitiert auf Seite 15)

[Mie10] R. Mietzner. *A method and implementation to define and provision variable composite applications, and its usage in cloud computing*. Dissertation, University of Stuttgart, 2010. URL http://elib.uni-stuttgart.de/opus/volltexte/2010/5614/. Http://d-nb.info/1012539598. (Zitiert auf den Seiten 15, 16 und 17)

[MMEKo6] K. E. Maghraoui, A. Meghranjani, T. Eilam, E. V. Konstantinou. Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In *In Proceedings of Middleware 2006, ACM/IFIP/USENIX, 7th International Middleware Conference*. 2006. (Zitiert auf den Seiten 15 und 21)

[MULo9] R. Mietzner, T. Unger, F. Leymann. Cafe: A Generic Configurable Customizable Composite Cloud Application Framework. In R. Meersman, T. Dillon, P. Herrero, Herausgeber, *CoopIS 2009 (OTM 2009)*, Band 5870 von *Lecture Notes in Computer Science*, S. 357–364. Springer-Verlag, Berlin, Heidelberg, 2009. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2009-75&engl=0. (Zitiert auf Seite 15)

[odea] http://svn.apache.org/viewvc/ode/trunk/bpel-schemas/src/main/xsd/dd.xsd. (Zitiert auf Seite 51)

[ODEb] A. ODE. http://ode.apache.org/. (Zitiert auf Seite 51)

[Orgo7] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. (Zitiert auf Seite 11)

[OSGo8] OSGi Alliance. OSGi Service Platform Core Specification, 2008. URL http://www.osgi.org/Specifications/HomePage/. (Zitiert auf Seite 12)

[PPA] T. N. for the Plan Portability API. https://www.oasis-open.org/committees/download.php/46756/TOSCA-24%20-%20Why.pptx. (Zitiert auf Seite 30)

[SISO] O. for the Advancement of Structured Information Standards OASIS. https://www.oasis-open.org/. (Zitiert auf Seite 11)

[TCa] O. T. TC. TOSCA Lifecycle Interface. http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html#_Toc347920704. (Zitiert auf den Seiten 17, 22, 33 und 46)

[TCb] O. T. TC. TOSCA Roles. http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html#_Toc347920696. (Zitiert auf Seite 21)

[TCATTC] O. Topology, O. S. for Cloud Applications TOSCA Technical Commitee. OASIS Topology and Orchestration Specification for Cloud Applications. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca. (Zitiert auf den Seiten 11, 12, 24, 26, 29 und 32)

[Wel94] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, 1994. (Zitiert auf Seite 14)

[WSO] WSO2. WSO2 Business Process Server. http://wso2.com/products/business-process-server/. (Zitiert auf Seite 12)

[XBBC] xjc Java Architecture for XML Binding Binding Compiler. http://docs.oracle.com/javase/6/docs/technotes/tools/share/xjc.html. (Zitiert auf Seite 51)

Alle URLs wurden zuletzt am 03. 07. 2013 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