Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Visualisierung von generierten Plänen des OpenTOSCA Plangenerators

Timm Pankratz

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	M. Sc. Kálmán Képes
Beginn am:	27. August 2020
Beendet am:	26. Februar 2021

Kurzfassung

Die automatische Bereitstellung von Cloud-Anwendungen gewinnt immer mehr an Bedeutung, da diese zunehmend komplexer werden. Deshalb ist es notwendig, die Modellierung und Bereitstellung zu erleichtern, um die Entwicklungszeit solcher Anwendungen zu verringern und sie schnellstmöglich Verfügbar zu machen. Das OpenTOSCA System, welches auf dem TOSCA Standard von OASIS beruht, liefert Werkzeuge für die Modellierung von Cloud-Anwendungen. Das System kann mithilfe von Plänen, die im Plangenerator erstellt werden, die Anwendungen bereitstellen. Da die automatisch generierten Bereitstellungspläne aber aus vielen verschieden Befehlen und Operationen bestehen, ist es schwierig mit diesen zu arbeiten, da sie unübersichtlich sind. Muss ein Fehler innerhalb des Bereitstellungsplans ausfindig gemacht werden, ist es notwendig den Plan komplett zu durchsuchen. Um Zeit bei der Suche zu sparen und diese zu erleichtern, wird in dieser Arbeit ein Konzept zur Visualisierung von generierten Plänen vorgestellt. Dazu wird gezeigt, welche Vorteile eine Visualisierung bei der Arbeit mit einem generierten Plan hat. Außerdem werden die einzelnen Schritte, die bei der Visualisierung vorgenommen werden müssen, beschrieben. Eine Implementierung dieses Konzepts wird mithilfe des Camunda BPMN-Modelers durchgeführt.

Inhaltsverzeichnis

1 Einleitung **13**
- 1.1 Motivation . . . 14
- 1.2 Struktur . . . 16

2 Grundlagen **17**
- 2.1 Cloud Computing . . . 17
- 2.2 DevOps . . . 18
- 2.3 TOSCA und OpenTOSCA . . . 18
- 2.4 BPEL . . . 21
- 2.5 BPMN und Camunda . . . 21
- 2.6 Angular . . . 22
- 2.7 SAX-Parser . . . 23
- 2.8 Docker . . . 23

3 Verwandte Arbeiten **25**
- 3.1 EPK-Visualisierung von BPEL4WS Prozessdefinitionen . . . 25
- 3.2 Zusammenpassen von BPMN und BPEL . . . 27
- 3.3 BPMN4TOSCA . . . 28

4 Use Case Szenario **31**
- 4.1 Topologie von MyTinyToDo . . . 31
- 4.2 Workflow . . . 32
- 4.3 MyTinyToDo als Beispiel Use Case . . . 32
- 4.4 Lösungsansätze . . . 33

5 Konzept **35**
- 5.1 Vorraussetzungen . . . 35
- 5.2 Informationsgewinnung aus den Plänen . . . 36
- 5.3 Transformieren der Informationen . . . 37
- 5.4 Visualisierung im Viewer . . . 37
- 5.5 Zusammenfassung und Eingehen auf den Use-Case . . . 38

6 Implementierung **41**
- 6.1 Parser . . . 41
- 6.2 Transformator . . . 44
- 6.3 Modeler . . . 45
- 6.4 Anmerkungen . . . 48

7 Zusammenfassung und Ausblick **49**

Literaturverzeichnis **51**

Abbildungsverzeichnis

1.1 BPEL Code aus dem MyTinyToDo Bereitstellungsplan 14
1.2 Visualisierung des Codes aus Abbildung 1.1 15

2.1 Aufbau einer TOSCA Service Template [OAS13] 19
2.2 Topologymodeler in der Winery [KBBL13] 20
2.3 BPMN-Prozess zur Zubereitung einer Tasse Tee 22

3.1 Beispiel EPK-Visualisierung für einen BPEL Flow [MZ05] 25
3.2 BPMN4TOSCA Elemente [KBBL12] 28

4.1 Topologie von MyTinyToDo 31
4.2 Informationen innerhalb der Topologie 33
4.3 Ausschnitt aus der Visualisierung der MyTinyToDo Topologie 34

5.1 Vorgehensweise des Konzepts 35
5.2 Architektur des Systems 36

6.1 Darstellung von Informationen mithilfe des *BPMN-properties-panel* 46
6.2 Visualisierung des *bpel:assign* Falls 47
6.3 Visualisierung des *bpel:if* Falls 47

Tabellenverzeichnis

3.1 Tabelle mit EPK Transformationen 26

6.1 Tabelle mit BPMN Transformationen 45

Verzeichnis der Algorithmen

6.1 Sortieren der scopes 43

1 Einleitung

Mithilfe von Cloud Computing werden Anwendungen über das Internet bereitgestellt und auf vermeintlich unendlichen Ressourcen verteilt. Durch die zunehmende Vernetzung von Cloud Anwendungen innerhalb immer komplexeren Anwendungsfällen, wird die Entwicklung, die Verwaltung, sowie die Wartung zunehmend komplexer. Um mit dieser Komplexität umzugehen ist es notwendig, die Arbeitsmethoden zu verbessern und Cloud Anwendungen automatisch bereitzustellen. Hier kommt das *DevOps* Prinzip ins Spiel, welches Methoden zur Entwicklung solcher Anwendungen liefert, indem verschiedene Aspekte wie Qualitätssicherung und Operationalität zusammen betrachtet werden [EGHS16]. Zur Bereitstellung von Cloud Anwendungen wird das *OpenTOSCA System* [BEK+16] entwickelt, welches auf *TOSCA* [OAS13], einer Cloud Anwendungsmodellier- und Deployment Sprache, basiert und DevOps Prinzipien anwendet. Anwendungen werden deklarativ modelliert, indem der Nutzer angibt, welche Komponenten und Verknüpfungen bei der Bereitstellung miteinander interagieren und wie diese konfiguriert werden sollen. Damit die Bereitstellung ausgeführt werden kann, werden von OpenTOSCA sogenannte imperative Pläne generiert, welche die einzelnen Schritte enthalten die zur Bereitstellung notwendig sind. Anwendungen besitzen aber häufig eine sehr komplexe Topologie und bestehen aus vielen verschiedenen Bestandteilen. Um den Ablauf dieser Pläne zu verstehen, muss sich durch die viele Operationen, Befehle und Zuordnungen von Variablen innerhalb der Pläne durchgearbeitet werden. Der Überblick kann schnell verloren gehen, was zu Missverständnissen und daraus folgenden Fehlern führen kann. Um Klarheit zu schaffen ist ein Ansatz nötig, der Verständnis über die Vorgehensweise der Anwendungen und des Systems ermöglicht. Durch den Ansatz soll es einfacher sein Fehler zu finden, möglichst bevor die Anwendung bereitgestellt wird. Zusätzlich verringert die frühe Fehlererkennung innerhalb einer Bereitstellung Zeit und Kosten, da Fehler dadurch nicht erst zu Anwendungs- oder Bereitstellungszeit erkannt werden. Diese Ziele können durch die Erzeugung einer Visualisierung des Bereitstellungsplans erreicht werden. Eine Visualisierung stellt Informationen übersichtlich dar und kann leichter verstanden werden als der zugehörige Code. Die Informationen können schneller identifiziert und ausgelesen werden als beim alternativen Suchen im Bereitstellungsplan. Das Suchen im Plan wird dadurch unnötig, was die Arbeitsgeschwindigkeit erhöht und dadurch Zeit spart.

Das Ziel dieser Arbeit ist deshalb das Visualisieren von Bereitstellungsplänen, um eine bessere Übersicht über die Vorgänge zu erhalten und Fehler einfacher finden zu können. Dafür wird ein Konzept entwickelt, dass eine visuelle Darstellung aus den gegebenen Bereitstellungsplänen erzeugen kann. Außerdem soll eine Implementierung des Konzepts erstellt werden, mit der das Konzept validiert wird. Diese Implementierung basiert auf dem BPMN-Modeler von Camunda.

1.1 Motivation

Das Arbeiten mit automatisch generiertem Code ist oft mühsam, da dieser meist nicht für Menschen gedacht ist. Fehler in solcher Art von Code zu finden ist schwierig. Vor allem für diejenigen, die sich nicht mit dem Generierungsvorgang auskennen. Leider ist es trotzdem notwendig damit zu arbeiten, da die automatische Generierung erst funktionieren muss, bevor sie anwendungsbereit ist. Cloud Anwendungen sind davon oft betroffen, da sie bereitgestellt werden müssen. Ihre Funktionsweise muss gesichert sein, da sonst viele Nutzer betroffen sind, falls die Anwendungen ausfallen oder Fehler verursachen. Fallen Fehler erst bei Anwendungszeit auf, müssen diese so schnell wie möglich behoben werden damit die Anwendung wieder nutzbar ist. Deshalb sind Konzepte wichtig, die die Arbeitszeit bei der Fehlersuche verringern. Das Konzept der Visualisierung ist hierbei ein guter Ansatz, da eine visuelle Darstellung leichter zu Verstehen ist als der Code. Dieses Konzept wird in vielen verschiedenen Anwendungsbereichen verwendet. Zum Beispiel werden Visualisierungen bei der Modellierung von Prozessen verwendet. Dies trägt zum Verständnis des Prozessablaufs bei und ist besser verständlich als eine textuelle Beschreibung. Ein weiteres Beispiel ist die Erklärung der Funktionsweise von Anwendungen. Werden Bilder zur Erklärung ergänzt, versteht man die Funktionsweise leichter. In manchen Fällen liefern Bilder an sich eine bessere Erklärung als die eigentliche Beschreibung. Deshalb spricht nichts dagegen, etwas das automatisch generiert wird zu Visualisieren. Ein solcher Fall kommt bei den Bereitstellungsplänen von Anwendungen, die im OpenTOSCA System erstellt werden, vor [BEK+16]. Diese bestehen aus vielen Zeilen BPEL Code [OAS07], der aufgrund der Menge von verschiedenen Befehlen und Variablen schwer lesbar ist. Trotzdem muss mit diesem Code gearbeitet werden, um die korrekte Funktion zu gewährleisten.

```
<bpel:extensionActivity xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
xmlns:bpel4RestLight="http://www.apache.org/ode/bpel/extensions/bpel4restlight">
    <bpel4RestLight:POST accept="application/xml" contenttype="text/plain"
    request="OpenTOSCAContainerAPIServiceInstanceID160189543448816018954344 88"
    response="bpel4restlightVarResponse22"
    uri="$bpelvar[OpenTOSCAContainerAPIServiceTemplateURL160189543448816018954344 88]
    /nodetemplates/MyTinyToDoDockerContainer/instances"/>
</bpel:extensionActivity>
<bpel:assign xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
name="$assignName" validate="no">
    <bpel:copy>
        <bpel:from variable="bpel4restlightVarResponse22">
            <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0">
            <![CDATA[string(//*[local-name()='url']/text())]]></bpel:query>
        </bpel:from>
        <bpel:to variable="nodeInstanceURL__http___opentosca_org_servicetemplates_
        MyTinyToDo_Bare_Docker_MyTinyToDoDockerContainer_1601895434479"/>
    </bpel:copy>
    <bpel:copy>
        <bpel:from variable="bpel4restlightVarResponse22">
            <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0">
            <![CDATA[string(codepoints-to-string(reverse(string-to-codepoints
            (substring-before(codepoints-to-string(reverse(string-to-codepoints
            ($nodeInstanceURL__http___opentosca_org_servicetemplates_MyTinyToDo_Bare_Docker_
            MyTinyToDoDockerContainer_1601895434479))), '/'))))))]]></bpel:query>
        </bpel:from>
        <bpel:to variable="nodeInstanceID__http___opentosca_org_servicetemplates_
        MyTinyToDo_Bare_Docker_MyTinyToDoDockerContainer_1601895434479"/>
    </bpel:copy>
</bpel:assign>
```

Abbildung 1.1: BPEL Code aus dem MyTinyToDo Bereitstellungsplan

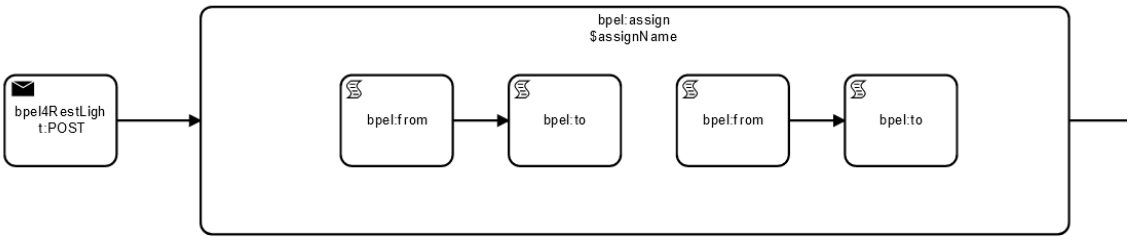

Abbildung 1.2: Visualisierung des Codes aus Abbildung 1.1

Selbst der Bereitstellungsplan einer kleinen Topologie wie der des MyTinyToDo Beispiels, welcher als Use Case in Kapitel vier dieser Arbeit verwendet wird, ist schon so lang, dass er schwer überschaubar ist. Somit wäre eine Visualisierung dieses Plans beim Bearbeiten eine große Hilfe.

In Abbildung 1.1 ist ein Codeabschnitt vom Bereitstellungsplan des MyTinyToDo Beispiels zu sehen. Schon auf den ersten Blick ist zu erkennen, dass selbst in einem kleinen Abschnitt des Plans viele Informationen geliefert werden. Zu Beginn des Codes wird ein HTTP Aufruf durchgeführt. Anschließend laufen verschiedene Vorgänge ab, die Variablen kopieren und mit *cdata* arbeiten. Wegen der Autogenerierung ist der Code oft nicht optimal eingerückt und es kommen syntaktische Informationen vor, die durch die Visualisierung auch implizit verstanden werden können. Deshalb müssen nur die wichtigsten Bestandteile dargestellt werden. Ein Beispiel dafür ist der *bpel:copy* Befehl, der das Kopieren von Informationen einleitet. Dieser Befehl wird in der Visualisierung durch den Pfeil zwischen *bpel:from* und *bpel:to* dargestellt. Die zugehörige Visualisierung des Codeabschnitts aus Abbildung 1.1 wird in Abbildung 1.2 gezeigt. Damit werden nur die für das Verständnis notwendigen Teile des Codes visualisiert. Informationen wie die Namen der Variablen fehlen dabei noch. Diese können bei der Darstellung im Viewer durch das Anklicken der zugehörigen Elemente eingesehen werden. Dadurch bleibt die Visualisierung übersichtlich, da nicht alle Informationen auf einmal gezeigt werden.

Das Beispiel aus Abbildung 1.1 und 1.2 ist eine weitere Bestätigung dafür, dass das Visualisieren von komplexen Abläufen und Operationen hilfreich für das Verständnis ist. Deshalb wird in dieser Arbeit ein passendes Konzept entwickelt, um dies Möglich zu machen.

1.2 Struktur

Diese Arbeit ist wie folgt strukturiert:

Kapitel 2 - Grundlagen

Kapitel zwei liefert die Grundlagen, die zum Verständnis dieser Arbeit notwendig sind. Dazu werden die Themengebiete erläutert und ihre Eigenschaften werden beschrieben. Anschließend wird ihre Bedeutung in Hinsicht auf die Arbeit begründet.

Kapitel 3 - Verwandte Arbeiten

Kapitel drei zeigt verwandte Arbeiten in ähnlichen Forschungsbereichen, die einen Einfluss auf diese Arbeit haben. Ihre Relevanz zu dieser Arbeit wird beschrieben.

Kapitel 4 - Use Case Szenario

Kapitel vier beschreibt ein Use Case Szenario und zeigt anhand Diesem, wie die Vorgehensweise beim Suchen von Fehlern innerhalb des Bereitstellungsplans abläuft. Es werden Lösungsansätze vorgeschlagen, um diesen Ablauf zu verbessern.

Kapitel 5 - Konzept

Kapitel fünf geht genauer auf das Konzept dieser Arbeit ein. Anschließend wird anhand des Use Case Szenarios die Anwendung des Konzepts beschrieben.

Kapitel 6 - Implementierung

Kapitel sechs zeigt, wie das Konzept in dieser Arbeit implementiert wird. Jeder Implementierungsschritt wird dabei genauer beschrieben.

Kapitel 7 - Zusammenfassung und Ausblick

Kapitel sieben fasst die gesamte Arbeit zusammen und weist auf die Einschränkungen und Limitationen hin. Zum Schluss wird noch ein Ausblick auf zukünftige relevante Erweiterungsmöglichkeiten und Themengebiete gegeben.

2 Grundlagen

Dieses Kapitel dient dazu, das Hintergrundwissen zu den verschiedenen Themengebieten, die zum Nachvollziehen dieser Arbeit notwendig sind, zu liefern. Im ersten Abschnitt wird das Konzept des Cloud Computing erklärt. Der zweite Abschnitt beschäftigt sich mit dem Begriff *DevOps* und die Bedeutung in Hinsicht auf diese Arbeit. Der dritte Abschnitt dient der Einführung des TOSCA Standards für Cloud Anwendungen. Anschließend wird OpenTOSCA, eine Open-Source Implementierung zur Bereitstellung und Bearbeitung von Cloud-Anwendungen nach dem TOSCA-Standard, beschrieben. Im vierten Abschnitt wird auf die WS-Business Process Execution Language, kurz BPEL, eingegangen, auf welchem die zur Bereitstellung von Cloud-Anwendungen notwendigen Pläne basieren. Im nächsten Abschnitt wird auf den Bpmn-Modeler der Firma Camunda, sowie die *Business Process Model and Notation*, kurz BPMN, eingegangen, welche zum Visualisieren der Bereitstellungspläne verwendet wird. Der sechste Abschnitt dient der Beschreibung von Angular, eine Plattform zum Erstellen von Webapplikationen, welche auf TypeScript basiert und zusammen mit dem Bpmn-Modeler von Camunda zur Implementierung verwendet wird. Im siebten Abschnitt wird die Verwendung des SAX-Parsers in dieser Arbeit begründet, welcher zur Gewinnung der für die Visualisierung relevanten Informationen verwendet wird. Der letzte Abschnitt beschreibt Docker, eine Anwendung die im Zusammenhang mit dem Use Case in Kapitel vier verwendet.

2.1 Cloud Computing

Cloud Computing ist ein Konzept, welches den Zugriff und die Verwendung von verschiedensten Ressourcen wie Server oder Datenbanken weltweit ermöglicht. Dabei wird alles als *Service* betrachtet, was hohe Flexibilität ermöglicht. Das heißt, dass Nutzer Programme, Speicher oder andere Ressourcen mieten können und nach dem Prinzip *pay-per-use* nur soviel bezahlen wie auch verwendet wird. Ein weiterer Aspekt der Flexibilität ist dadurch gegeben, dass Nutzer jederzeit mehr Ressourcen mieten können, falls nötig. Die *NIST* Definition beschreibt die Charakteristiken von Cloud Computing genauer [MG11]. Die fünf definierenden Eigenschaften von *NIST* werden im Folgenden aufgeführt und näher erläutert.

- **On-Demand-Self-Service** ist die erste beschriebene Charakteristik. Diese besagt, dass ohne menschliche Interaktion Ressourcen für Nutzer bereitgestellt werden können.
- Die zweite Charakteristik, **Broad Network Access**, beschreibt, dass Ressourcen über die üblichen Netzwerkprotokolle zur Verfügung gestellt werden.
- **Resource Pooling** ist die dritte Charakteristik. Diese beschreibt, dass Provider von Cloud-Anwendungen ihre Ressourcen vielen verschiedenen Nutzern zur Verfügung stellen. Dadurch wird eine effiziente Nutzung dieser Ressourcen ermöglicht, da verschiedene Nutzer eine unterschiedliche Anzahl von Ressourcen benötigen und dabei Schwankungen auftreten.

- Die vierte Charakteristik wird **Rapid Elasticity** genannt und beschreibt die Elastizität der zu bereitstellenden Ressourcen. Ändert sich in einem kurzen Zeitraum der Bedarf an Ressourcen stark, können diese schnell an den Bedarf angepasst werden. Deshalb scheinen Ressourcen in der Cloud für Nutzer unendlich zu sein, da die Quantität sich automatisch anpasst.
- Die letzte Charakteristik heißt **Measured Service**. Hier kommt das zu Beginn beschriebene Prinzip *pay-per-use* ins Spiel. Die Verwendung von Ressourcen wird überwacht und der Nutzer muss genau den Betrag zahlen, der für die Ressource in dem Zeitraum festgelegt ist.

Zur automatischen Bereitstellung von Cloud Anwendungen nach dem *NIST* Prinzip werden Plattformen benötigt, um solche Anwendungen zu Erstellen und verfügbar zu machen. Dazu kann der TOSCA Standard verwendet werden, auf den in Abschnitt 2.3 eingegangen wird.

2.2 DevOps

Der Begriff *DevOps* wird heutzutage oft im Zusammenhang mit Entwicklung und Operation von Cloud-Anwedungen verwendet. Der Kerngedanke ist hier die Erhöhung der Geschwindigkeit bei der Entwicklung beziehungsweise Anpassung von Anwendungen. Dies soll durch die Zusammenarbeit von Entwicklungs-, Qualitätssicherungs- und Operationsaspekten erreicht werden [EGHS16]. Eine genaue Definition für diesen Begriff und die entsprechende Vorgehensweise liegt aber nicht vor. Die Feinheiten der Definition sind in unterschiedlichen Umgebungen auf diese angepasst. Verschiedene Definitionen enthalten nach Jabbari et al. [JAK16] unter anderem die Verwendung von *DevOps* als Methode für verbesserte Arbeit in einer Gruppe, *DevOps* als moderne Entwicklungsmethode oder *DevOps* als Standard für die Integration verschiedener Bereiche. In dieser Arbeit wird *DevOps* als Verbesserung der Entwicklungs- und Anpassungsmethode aufgefasst, da das Ziel das Erstellen einer Visualisierung ist, die helfen soll, Zeit bei der Entwicklung, Anpassung und Fehlersuche zu sparen. Eine Rolle spielt ebenfalls die *DevOps* Definition *Infrastruktur als Code*, bei der Templates mit Bereitstellungsfunktionen verwendet werden, um Cloud Anwendungen bereitzustellen [ABD+17]. Die Bereitstellungspläne, dessen Visualisierung das Ziel dieser Arbeit ist, werden nach dem TOSCA Standard, der auf der *DevOps* Definition *Infrastruktur als Code* beruht, erstellt.

2.3 TOSCA und OpenTOSCA

TOSCA (*Topology and Orchestration Specification for Cloud Applications*) ist eine Sprache zur Modellierung und Bereitstellung von Cloud Anwendungen, welche von OASIS entwickelt wurde [OAS13]. Es wird ein Standard festgelegt, der die automatische Bereitstellung und das Anpassen von Cloud Anwendungen erleichtert. Das Ziel dabei ist, die Portabilität und Interoperabilität zu gewährleisten. Die nach dem Standard erstellten Anwendungen sollen möglichst gut mit anderen Anwendungen interagieren und zusammenarbeiten können. Dabei soll aber der Austausch jederzeit möglich sein, sodass ein bestimmter Grad von Unabhängigkeit zwischen den einzelnen Bestandteilen bestehen muss. Um dies zu bewerkstelligen, liefert TOSCA ein Konzept für die Spezifikation solcher Anwendungen. Dieses Konzept beinhaltet ein Meta-Model für die Definition und den Aufbau der Anwendungen. Die Struktur des Aufbaus wird im Folgenden beschrieben.

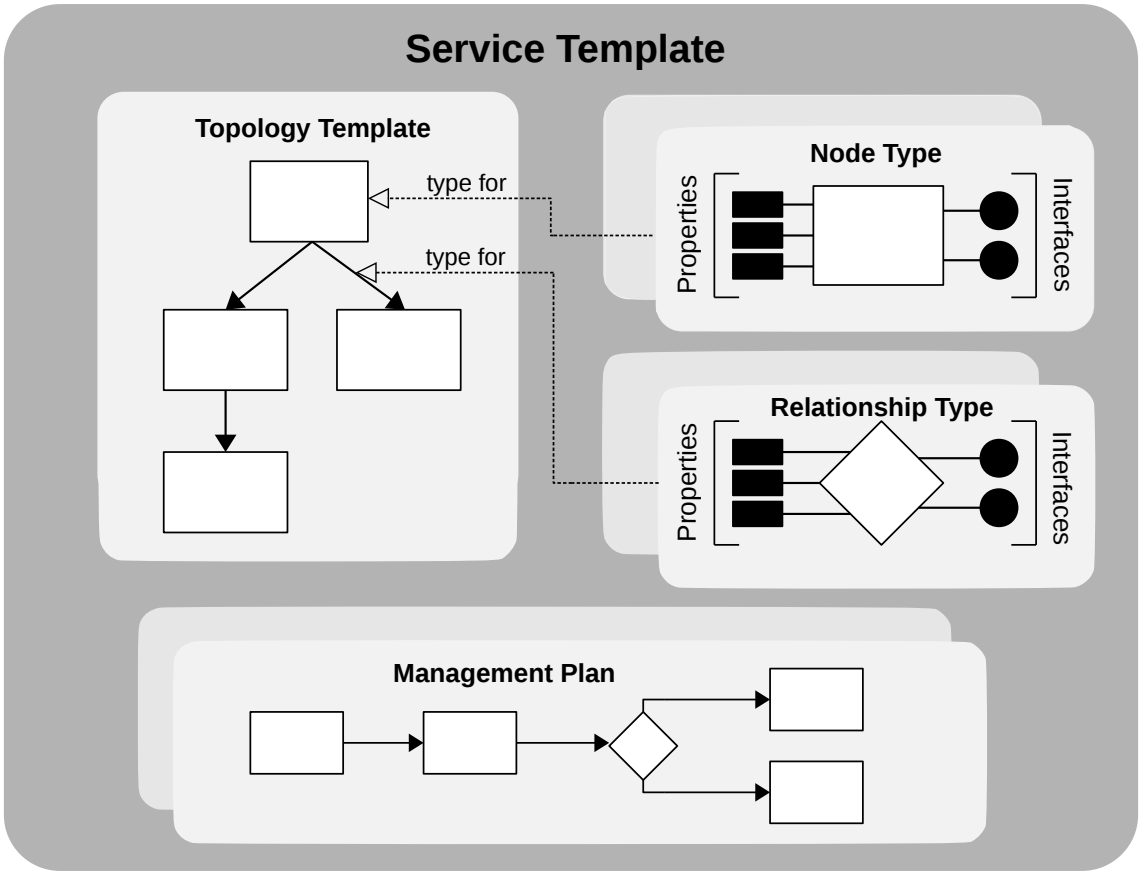

Abbildung 2.1: Aufbau einer TOSCA Service Template [OAS13]

Das Modellieren von Cloud Anwendungen erfolgt in TOSCA mithilfe von *Service Templates*. Eine *Service Template* besteht aus mehreren Bestandteilen, dem *Topology Template*, den *Node* und *Relationship Types*, sowie den *Management Plans*. Der Aufbau einer *Service Template* wird in Abbildung 2.1 gezeigt. Ein *Topology Template* besteht aus mehreren *Node Templates* die durch *Relationship Templates* miteinander verbunden sind. Die *Topology Template* stellt dabei einen Graph dar, dessen Knoten die *Node Templates* und dessen Kanten die Relationship Templates sind. In einer Anwendung entsprechen die *Node Templates* den einzelnen Komponenten, während die *Relationship Templates* die Beziehung zwischen den Komponenten, mit denen sie verbunden sind, darstellen. Die *Node Types* und *Relationship Types* innerhalb der *Service Template* beinhalten die Eigenschaften und Schnittstellen aller entsprechenden *Node Templates* und *Relationship Templates*, die in der *Topology Template* vorkommen. Die *Management Plans* einer *Service Template* beschreiben die Vorgehensweise bei Vorgängen wie dem Bereitstellen oder dem Terminieren des entsprechenden Plans. Die Sprache zum Beschreiben dieser Vorgänge ist dabei nicht festgelegt, was die Implementierungsmöglichkeiten flexibel macht. Da in dieser Arbeit der Fokus auf der Visualisierung des Bereitstellungsaspekts liegt, steht der *Management Plan* zum Bereitstellen der Anwendung im Vordergrund. Im Laufe dieser Arbeit wird dieser *Management Plan* bei der Verwendung als *Bereitstellungsplan* bezeichnet.

Um die Anwendung, welche durch die *Service Template* definiert ist, bereitzustellen und Instanzen dieser Anwendung erstellen zu können, verwendet TOSCA zwei Arten von Artefakten. Die *Deployment Artifacts* beinhalten die konkrete Implementierung der entsprechenden *Node Types* und *Relationship Types*. Ohne *Deployment Artifacts* ist die Anwendung nicht funktionsfähig. *Implementation Artifacts* werden zur Steuerung der Anwendung verwendet. Sie enthalten alle notwendigen Inhalte, um durch Operationen eine Instanz der Anwendung zu erstellen oder zu stoppen. Der Vorteil bei der Verwendung solcher Artefakte liegt genau wie bei den *Management Plans* in der Flexibilität. Die Wahl der Implementierungssprache und die Implementierungsdetails können passend zur Anwendungsart oder zum Anwendungszweck gewählt werden.

Damit die Anwendung verwendet werden kann ist es notwendig, dessen Modell an eine Laufzeitumgebung zu schicken. Um dies möglich zu machen wird im TOSCA Standard ein Dateityp eingeführt. Dieser wird CSAR (*Cloud Service Archive*) genannt und besteht aus dem *Service Template* der Anwendung, sowie allen relevanten *Deployment Artifacts* und *Implementation Artifacts*, die für die Bereitstellung der Anwendung notwendig sind. Dadurch ist die Portabilität der Anwendung gegeben. Da TOSCA keine konkrete Laufzeitumgebung definiert, kann somit jede beliebige Umgebung verwendet werden, die mit CSAR Dateien umgehen kann.

OpenTOSCA

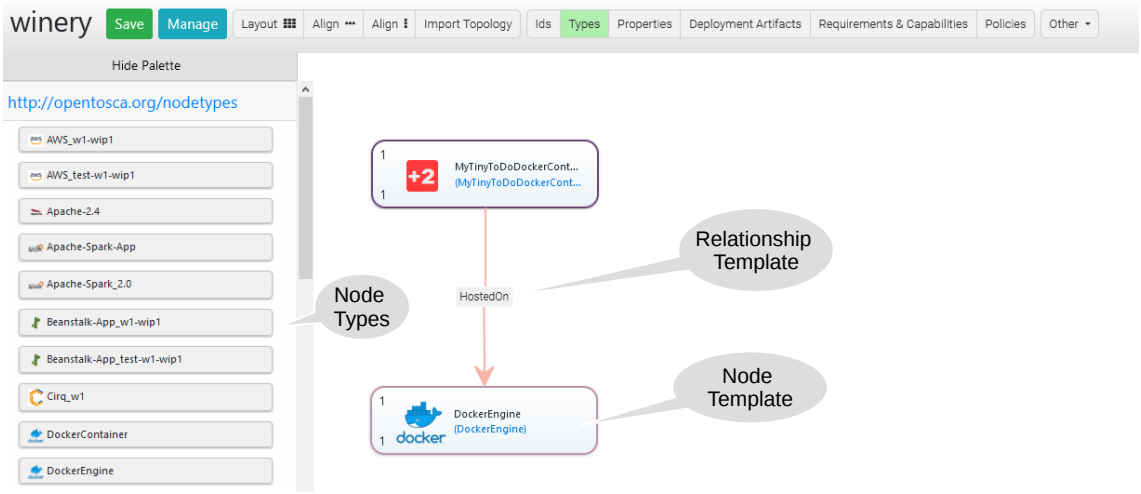

Abbildung 2.2: Topologymodeler in der Winery [KBBL13]

Eine Implementierung eines Systems auf der Basis des TOSCA Standards wird von der Universität Stuttgart entwickelt. Dieses System heißt OpenTOSCA und steht für *Open Source TOSCA Ecosystem* [BEK+16]. Es besteht aus drei Hauptbestandteilen, der Winery, dem OpenTOSCA Container und der OpenTOSCA UI. Die Winery wird dazu verwendet, um verschiedene TOSCA Bestandteile wie *Service Templates* und *Node Templates* zu bearbeiten. Spezifikationen können dort von Nutzern angepasst werden und es können zum Beispiel neue *Node Types* erstellt und verfügbar gemacht werden. Außerdem verfügt die Winery über einen *Topology Modeler*, der mithilfe vorhandener *Node-* und *Relationship Templates* zur Erstellung neuer *Service Templates* mit entsprechenden Plänen

verwendet werden kann. Eine Beispieltopologie im *Topology Modeler* wird in Abbildung 2.2 gezeigt. Auf der linken Seite der Abbildung ist eine Liste mit verschiedenen Node Types zu sehen, die zur Modellierung verwendet werden können. Zum Austausch von Informationen kann die Winery diese in CSAR Dateien verpacken und an andere OpenTOSCA Komponenten verschicken. Der Container wird im OpenTOSCA System dazu verwendet, um Anwendungen bereitzustellen. Er stellt die Laufzeitumgebung des Systems dar. Die Anwendungen, die bereitgestellt werden sollen, werden in der Form von CSAR Dateien aus der Winery in den Container geliefert. Die CSAR Dateien enthalten Bereitstellungspläne, die die notwendigen Informationen für die Bereitstellung enthalten. Die Bereitstellung und Instanziierung der Anwendungen werden mithilfe der OpenTOSCA UI gesteuert. Mit dieser graphischen Benutzerschnittstelle ist es möglich, die Anwendungen in den Container zu laden, zu starten und gegebenenfalls Parameter zu ergänzen. Alle laufenden Instanzen werden dabei aufgelistet und können beeinflusst werden.

2.4 BPEL

Die auf XML basierende Sprache BPEL (*WS-Business Process Execution Language*) [OAS07] wird zum Beschreiben von ausführbaren Prozessen verwendet. BPEL Prozesse kommunizieren ausschließlich mit anderen Services und Prozessen und sind deshalb nicht darauf angepasst, für Nutzer ohne Hilfsmittel leicht zugänglich und verständlich zu sein. Deshalb ist eine Alternative zum Durchsuchen des BPEL Prozesscodes für Nutzer, die spezifische Informationen aus diesem Code brauchen, oft sehr hilfreich. Den Prozesscode graphisch darzustellen ist hier meistens der beste Ansatz. Im Fall von OpenTOSCA wird BPEL zur Beschreibung von Bereitstellungsplänen und ihrem Ablauf verwendet. Ohne diese Pläne ist es nicht möglich im OpenTOSCA System eine Anwendung bereitzustellen. Deswegen sollten die Pläne möglichst fehlerfrei sein. Zum Überprüfen der Korrektheit eines Plans sind Vorgänge wie das Durchsuchen des Plans nach bestimmten Variablen und ihre Verwendung unvermeidbar. Deshalb ist eine Visualisierung des Plans sehr hilfreich um Zeit zu sparen. Die naheliegendste und oft gewählte Möglichkeit zur Visualisierung von BPEL ist die Verwendung von BPMN zur Darstellung von Prozessabläufen.

2.5 BPMN und Camunda

Die *Business Process Model and Notation*, kurz BPMN genannt, ist eine graphische Spezifikationssprache zur Beschreibung von Workflows und Geschäftsprozessen [OMG11]. Mithilfe von verschiedenen Events und Aktivitäten können komplexe Abläufe modelliert und beschrieben werden, um das Verständnis des entsprechend modellierten Prozesses zu erleichtern.

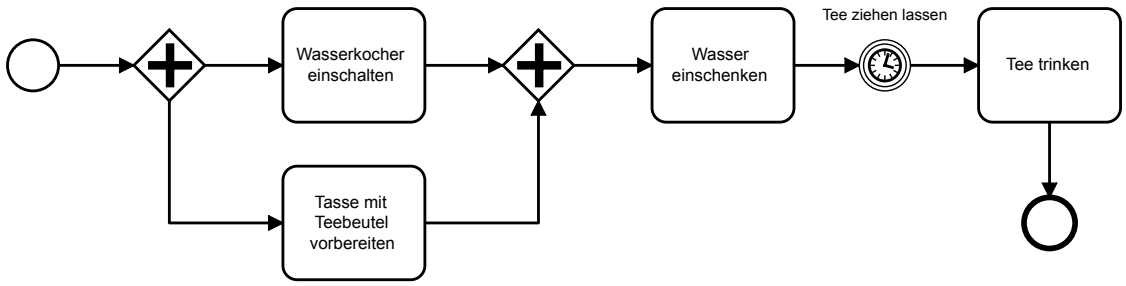

Abbildung 2.3: BPMN-Prozess zur Zubereitung einer Tasse Tee

Ein Beispielprozess für die Modellierung mit BPMN wird in Abbildung 2.3 gezeigt. Hier wird der Ablauf modelliert, der zum Zubereiten und Trinken einer Tasse Tee notwendig ist. Zuerst wird parallel zum Einschalten des Wasserkochers eine Tasse mit einem Teebeutel vorbereitet. Sind beide Aktionen abgeschlossen, wird das Wasser in die Tasse eingeschenkt. Nachdem der Tee etwas ziehen gelassen wird, wird er getrunken. Damit ist der Prozess abgeschlossen. Die Vielseitigkeit von BPMN beim Modellieren von Prozessen ist ein ausschlaggebender Grund für die Wahl eines BPMN-Modelers zur Visualisierung der Bereitstellungspläne.

Der BPMN-Modeler, der in dieser Arbeit verwendet wird, heißt *bpmn-js*[1]. Dieser Modeler wird von der Firma Camunda[2] entwickelt und ist leicht anpassbar und erweiterbar. Die Möglichkeiten, die dieser Modeler zur Verfügung stellt, eignen sich gut für den Anwendungszweck dieser Arbeit. Das *Modeling-API* des Modelers trivialisiert das Erstellen und Hinzufügen von BPMN-Elementen. Die erstellten Elemente lassen leicht verknüpfen und, falls nötig, anpassen. Die *properties-panel* Erweiterung von *bpmn-js* eignet sich gut zum Darstellen der Informationen spezifischer BPMN-Elemente, da diese ein Panel hinzufügt, welches Attribute der Elemente anzeigen kann. Sollten weitere Funktionen notwendig sein, lassen sich diese auch im Nachhinein einfach hinzufügen.

2.6 Angular

Angular ist eine Open-Source Platform zum Erstellen von Webapplikationen. Die ursprüngliche Version von Angular wird AngularJS genannt und verwendet JavaScript [JBM14]. Die neueren Versionen basieren dagegen auf TypeScript[3], was Typisierung ermöglicht. Diese sind außerdem modularer, weshalb Angular in dieser Arbeit gegenüber AngularJS vorgezogen wird. Angular wird in dieser Arbeit gemeinsam mit dem Camunda BPMN-Modeler verwendet, um Bereitstellungspläne zu visualisierten. Angular-Applikationen bestehen aus verschiedenen Modulen, welche in Komponenten und Services eingeteilt sind. Dadurch können neue Module schnell und einfach eingebunden werden. Im Falle dieser Arbeit wird der Camunda Modeler, sowie entsprechende Erweiterungen des Modelers, als Angular Komponente verwendet. Die Implementierung, welche die Visualisierung der Bereitstellungspläne ermöglicht, wird mithilfe mehrerer Services hinzugefügt. Ein Service ist

[1] https://github.com/bpmn-io/bpmn-js
[2] https://camunda.com/
[3] https://www.typescriptlang.org/

flexibler als eine Komponente und kann somit leichter angepasst und erweitert werden. Außerdem kann die Anwendung, durch Verwendung der HTTP-Module von Angular, Informationen mit anderen Anwendungen austauschen, sollte dies benötigt werden.

2.7 SAX-Parser

Das *Simple API for XML*, welches mit SAX abgekürzt wird, ist eine Technik zum Parsen von XML-Dateien. Ein SAX-Parser ist eventbasiert und arbeitet den zu parsenden XML-Code sequentiell ab. Dies erfordert das manuelle Behandeln jedes Events, seien es Strings, XML-Tags oder sonstige Vorkommnisse. Im Gegensatz zum DOM-Parser, welcher die zu parsende XML in eine DOM-Struktur nach dem *Document Object Model* umwandelt und alle Informationen jederzeit aufrufbar macht, kann der SAX-Parser nur auf bestimmte Informationen zugreifen, wenn sie zu dem Zeitpunkt in der Sequenz vorkommen. Da aber bei richtiger Verwendung des SAX-Parsers die gesamte XML nur einmal durchgegangen werden muss, ist der Verbrauch von Ressourcen deutlich geringer als beim alternativen DOM-Parser. Der DOM-Parser benötigt genug Speicherplatz, um die erzeugte DOM-Struktur zu speichern. Deshalb ist der DOM-Parser für das Parsen von großen XML nicht geeignet, da der Speicherplatzverbrauch entsprechend der Länge des XML-Codes ansteigt. Dies ist auch der Grund, der die Benutzung des SAX-Parsers in dieser Arbeit veranlasst. Selbst kleine Bereitstellungspläne besitzen eine Länge, bei der die Benutzung des SAX-Parsers vorteilhaft ist. Außerdem können nicht benötigte Informationen beim Verwenden des SAX-Parsers direkt herausgefiltert werden, indem die entsprechenden Events beim sequentiellen Abarbeiten des XML-Codes ignoriert werden. Das erleichtert die weitere Verwendung der gewonnenen Informationen, da jede einzelne Information relevant ist.

2.8 Docker

Docker ist eine Applikation, die einen Rahmen zur Bereitstellung und Verteilung von Anwendungen darstellt [Mer14]. Der Kern von Docker ist die DockerEngine, welche die Anwendung und ihre Abhängigkeiten verpacken und in der Form eines Containers bereitstellen kann. Diese Docker Container werden isoliert auf dem Kernel vom Betriebssystem ausgeführt. Durch diese Eigenschaften eignen sich Docker Container gut zum Bereitstellen von Cloud-Anwendungen. In dieser Arbeit werden die Docker Engine und ein Docker Container als Beispielkomponenten eines Bereitstellungsplans verwendet. Sowohl die Engine als auch der Container stellen hierbei *Node Templates* nach dem TOSCA Standard dar und sind deshalb als Beispiel geeignet.

3 Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten vorgestellt, die das Konzept und die Implementierung in dieser Arbeit beeinflusst haben. Dazu gehören EPKs, Vergleiche von BPMN und BPEL, sowie die graphische Sprache BPMN4TOSCA. Jeder Abschnitt betrachtet solch eine Arbeit.

3.1 EPK-Visualisierung von BPEL4WS Prozessdefinitionen

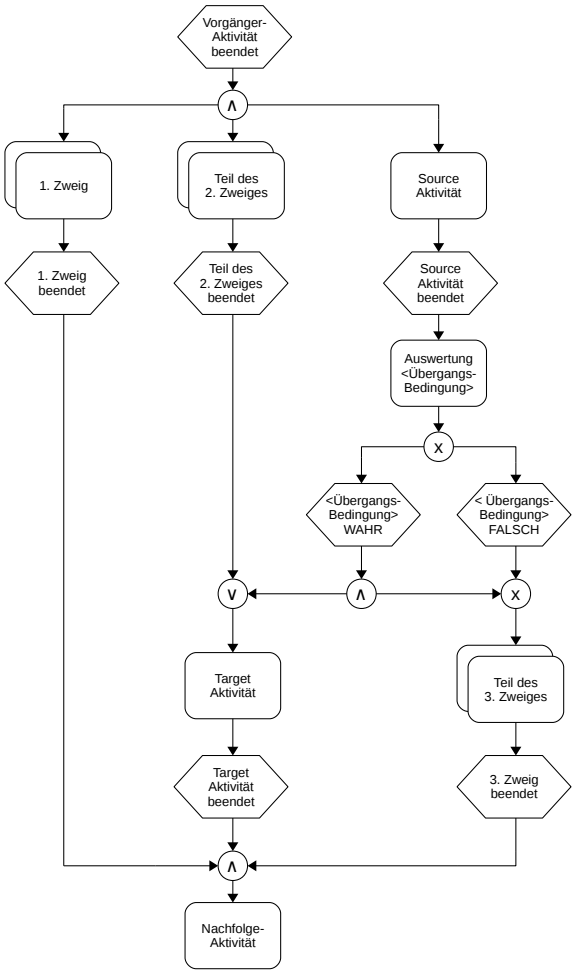

Abbildung 3.1: Beispiel EPK-Visualisierung für einen BPEL Flow [MZ05]

Die Visualisierung von BPEL Prozessen ist schon immer ein wichtiges Forschunsgebiet gewesen. Um das Verständnis dieser Prozesse zu verbessern werden viele verschiedene Ansätze vorgeschlagen. Bereits im Jahr 2005 wurde von Mendling und Ziemann ein Konzept zur Visualisierung von BPEL Prozessen, damals noch BPEL4WS Prozesse genannt, vorgeschlagen [MZ05]. Dieses Konzept bildet BPEL Prozesse auf Ereignisgesteuerte Prozessketten, kurz EPKs, ab [NR02]. EPKs besitzen Elemente, die Vorteilhaft für eine BPEL Abbildung sind. Funktionen werden in EPKs verwendet, um Aktivitäten darzustellen. Ereignisse sind Vor- und Nachbedingungen von Funktionen, die die Aktivitäten eingrenzen können. Damit können auch Unterprozesse dargestellt werden, was hierarchische Prozessabläufe ermöglicht. Außerdem besitzen EPKs Konnektoren, um die Elemente miteinander zu verbinden und Verzweigungen darzustellen. Abbildung 3.1 zeigt ein Beispiel für die Visualisierung eines BPEL Prozesses. In dieser Visualisierung ist der Ablauf des Prozesses übersichtlich, aber es fehlt noch eine Möglichkeit, Informationen innerhalb der einzelnen Bestandteile darzustellen. Mendling und Ziemann schlagen deshalb für die Transformation von BPEL zu EPK das EPML Format zur Darstellung der EPKs vor [MN06]. Dieses Format beinhaltet zusätzlich zu den normalen EPK Elementen Erweiterungen wie Datenfelder, um zusätzliche Informationen und Spezifikationen, die Transformation notwendig sind, darzustellen.

Die Transformation der verschiedenen BPEL Elemente und Blöcke wird im Folgenden beschrieben. Eine Übersicht dazu findet sich in Tabelle 3.1. Der Start des Prozessablaufs wird durch ein *AND-split* Element dargestellt. Mithilfe eines *AND-join* Elements wird dieser beendet. BPEL Blöcke wie der *bpel:invoke* Block werden auf Funktionen, die mit Ereignissen kombiniert werden gemappt. Besitzten die BPEL-Blöcke Eingaben oder Ausgaben, werden diese mithilfe von Datenfeldern dargestellt, die mit einer korrespondierenden Eingabe- oder Ausgabefunktion verbunden sind. Da der *bpel:invoke* Block beides besitzt, sind beide Funktionen und Datenfelder enthalten. *bpel:receive* besitzt dafür nur die Eingabefunktion und das ensprechende Datenfeld. Ebenso wie *bpel:invoke* besitzt *bpel:assign* beide Funktionen und Datenfelder, da eine Zuordnung durchgeführt wird.

BPEL Elemente	**EPK Elemente**
flow	startet mit AND-split, endet mit AND-join
scope	sub-process innerhalb der flow Darstellung
fault scope	fault handler, sub-process
compensation scope	compensation handler, sub-process
bpel:assign	Eingabefunktion, Ausgabefunktion, Eingabefeld, Ausgabefeld, Ereignisse
bpel:invoke	Eingabefunktion, Ausgabefunktion, Eingabefeld, Ausgabefeld, Ereignisse
bpel:receive	Eingabefunktion, Eingabefeld, Ereignisse
bpel:from	Eingabefeld
bpel:to	Ausgabefeld
bpel:if	Ereignis
bpel:throw	Funktion, Ereignis, Fehlerdatenfeld
bpel:while	XOR-join, Ereignis, Funktion, XOR-split
bpel:switch	XOR-split, XOR-join

Tabelle 3.1: Tabelle mit EPK Transformationen

Der *bpel:reply* Block besitzt im Gegensatz zu *bpel:receive* nur die Ausgabefunktion und das Datenfeld dazu. Der *bpel:throw* Block besteht aus einer Kombination aus Funktion und Ereignissen, wobei die Funktion den Fehler mithilfe eines Datenfelds speichert. Die Bedingungsabfrage *bpel:if* findet dabei innerhalb des Ereignisses statt. Ist die Bedingung erfüllt, führt dies zum Abbruch des Prozessablaufs und der *fault handler* wird ausgeführt. Kommen Schleifen oder Aufteilungen innerhalb des Prozessablaufs vor, werden diese mithilfe von *XOR-Splits* und *XOR-Joins* dargestellt.

Der Ansatz von Mendling und Ziemann weist Ähnlichkeiten zum Ansatz für die Visualisierung von BPEL Prozessen, der in dieser Arbeit verwendet wird, auf. Die Art der Transformation und die Nutzung von Datenfeldern zum Speichern von benötigten Informationen ist auch übergreifend bei anderen Visualisierungsmöglichkeiten eine gute Idee. Diese Ideen können auch bei der Transformation von BPEL zu BPMN verwendet werden, da der Übergang von EPKs zu BPMN Prozessen ohne Probleme oder großen Verlusten durchgeführt werden kann. EPKs können als Vorgänger von BPMN betrachtet werden, da der Prozessablauf von BPMN dem von EPKs stark ähnelt. BPMN ist auch übersichtlicher und benutzerfreundlicher als EPKs. Deswegen wurde die Benutzung von EPKs, die vor allem im deutschen Raum weit verbreitet waren, von BPMN abgelöst, da der Übergang einfach zu bewerkstelligen ist [DT09]. Der nächste Schritt wäre die direkte Transformation von BPEL zu BPMN. Dabei ist aber zu beachten, dass diese im Gegensatz zur Transformation von EPK zu BPMN nicht verlustfrei möglich ist. [HS10].

3.2 Zusammenpassen von BPMN und BPEL

Sollen BPEL Prozesse mithilfe von BPMN visualisiert werden, müssen die Eigenschaften der beiden Sprachen genauer betrachtet werden. Hündling und Schmiedel vergleichen in ihrer Arbeit die Sprachen und fassen die Kernpunkte zusammen [HS10]. Die Sprache BPEL ist nach Hündling und Schmiedel hauptsächlich zur Darstellung von Komplexen Prozessen und Transaktionsszenarien geeignet. Vor allem bei voll automatisierten Vorgängen die keinen menschlichen Eingriff benötigen ist BPEL eine gute Wahl. Die Trennung zwischen technischen Abläufen und Geschäftsprozessen, die optimierbar sind, ist der Einsatzbereich von BPEL, wobei letzteres durch andere Sprachen wie BPMN durchgeführt werden sollte. Das Optimieren solcher Prozesse durch kleine Anpassungen und Änderungen am Prozess ist der Einsatzbereich von BPMN. Außerdem ist BPMN flexibler als BPEL, da in BPMN unter anderem parallele Abläufe desselben Tasks schneller integriert werden können. Was in BPEL eine komplette Ausmodellierung erfordert, geschieht in BPMN durch setzten eines Attributs. Deswegen ist BPMN auch besser zum Erstellen von Modellen geeignet. Prozesse können in BPMN auch auf Arten gestartet werden, die in BPEL Prozessen nicht möglich sind. Beispiele hierfür sind *BPMN-timer-events* oder *BPMN-fault-events*.

Trotz der verschiedenen Einsatzbereiche lassen sich BPEL Prozesse bei Beachtung der semantischen Unterschiede mithilfe von BPMN-Elementen darstellen. Da keine feste Definition für die Transformation existiert und wegen der Unterschiede auch nicht möglich ist, unterscheidet sich die Visualisierung in jedem Anwendungsfall. Eine funktionsfähige Übersetzung von BPMN zu BPEL oder umgekehrt ist unter Beachtung bestimmter Richtlinien durchführbar [OMG11]. Es dürfen keine Verklemmungen vorkommen, da sonst der Prozessablauf stoppt. Außerdem darf während des Ablaufs die Synchronisation nicht verloren gehen. Dies ist der Fall, wenn derselbe Prozessabschnitt mehrmals zur selben Zeit ausgeführt wird. Der kombinierte Einsatz von BPMN und BPEL, was im Fall dieser Arbeit der Visualisierung entspricht, lohnt sich nach Hündling und

Schmiedel nur wenn bestimmte Kriterien zutreffen. Ein solches Kriterium ist das Vorhandensein eines BPEL-Prozesses beziehungsweise einer Implementierung, die solche Prozesse erzeugt. Ist es notwendig solche Prozesse genauer zu überwachen oder Teile des Prozesses zu überprüfen, lohnt sich der gemeinsame Einsatz der beiden Sprachen. Vor allem, wenn die Komplexität der Prozesse das Verständnis erschwert. Die Bereitstellungspläne, welche in dieser Arbeit visualisiert werden sollen, sind solch ein Anwendungsfall. Dementsprechend ist eine Visualisierung nach den Kriterien lohnenswert und trägt zum Verständnis bei.

3.3 BPMN4TOSCA

BPMN4TOSCA ist eine graphische Sprache, die eine Erweiterung von BPMN darstellt. Diese TOSCA spezifische Erweiterung von BPMN wurde von Kopp et al. entwickelt, um die Modellierung des *TOSCA Management Plans* mithilfe von TOSCA Operationen zu erleichtern [KBBL12]. Außerdem soll die Darstellung der TOSCA spezifischen Informationen übersichtlicher sein. Dazu werden vier neue Elemente eingeführt, die dies möglich machen sollen. Diese werden in Abbildung 3.2 gezeigt. Der *TOSCA Topology Management Task* wird benutzt, um Operationen, die in der Laufzeitumgebung des Systems definiert sind, verwendbar zu machen. Im Fall des *OpenTOSCA* Systems ist die Laufzeitumgebung der TOSCA Container. Der *TOSCA Node Management Task* ermöglicht das Ausführen von TOSCA Operationen der *Node Templates*. Sowohl dieser Task als auch der *TOSCA Topology Management Task* sind Erweiterungen des *BPMN-Service-Tasks*. Der nächste Task ist eine Erweiterung des *BPMN-Script-Tasks* und wird *TOSCA Script Task* genannt. Zusätzlich zu der normalen Funktion des *BPMN-Script-Tasks* kann dieser Scripts verwenden, die in TOSCA spezifischen Dateien definiert sind. Die letzte Erweiterung ist das *TOSCA Data Object*, welches eine Erweiterung des *BPMN-Data-Objects* darstellt. Dieses *TOSCA Data Object* liefert

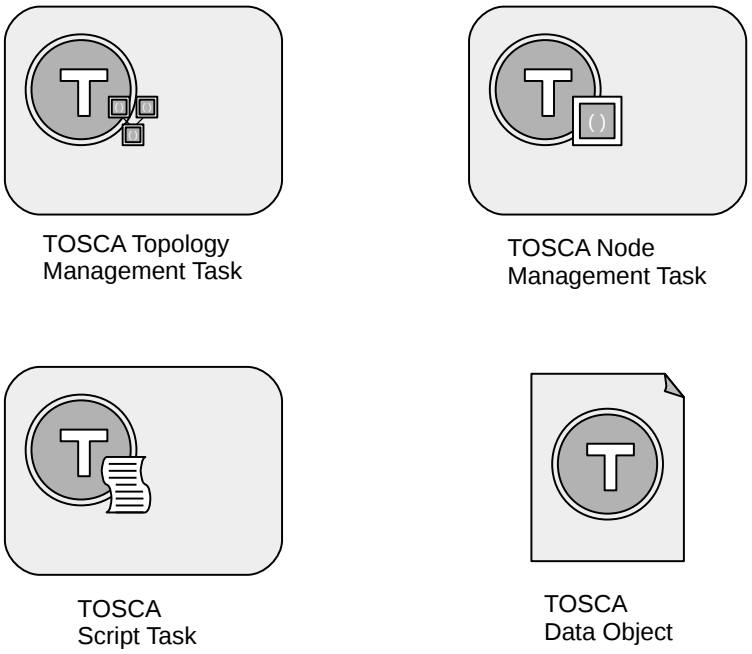

Abbildung 3.2: BPMN4TOSCA Elemente [KBBL12]

Informationen von *Nodes* und *Relationships*, die während der Laufzeit erzeugt werden. Dazu gehört zum Beispiel die *NodeInstance ID*, die bei der Instanziierung des Plans für ein bestimmtes *Node Template* erzeugt wird. Die Kommunikation und Übertragung der Informationen zwischen dem TOSCA Container und dem *Node Data Object* ist dabei nicht sichtbar. Dadurch fallen Bestandteile des Plans weg und dieser wird übersichtlicher.

Die Verwendung von BPMN4TOSCA als graphische Sprache zur Implementierung des Konzepts dieser Arbeit kann in Betracht gezogen werden. Dies hat sowohl Vor- als auch Nachteile im Hinblick auf das Ziel der Visualisierung. Werden die generierten Pläne, die Visualisiert werden sollen, mithilfe einer *ServiceTemplate* aus einem auf TOSCA basierenden System, wie beispielsweise OpenTOSCA, erzeugt, besitzen diese TOSCA Operationen. In diesem Fall können die TOSCA Operationen in der Visualisierung durch den *TOSCA Toplogy Management Task* oder den *TOSCA Node Management Task* optimal dargestellt werden. Dazu wird aber ein Viewer oder Modeler benötigt, der BPMN4TOSCA Elemente darstellen, beziehungsweise modellieren kann. Durch die Funktionalität der BPMN4TOSCA Elemente, mit anderen Komponenten wie dem TOSCA Container im OpenTOSCA System zu interagieren, ist dies umso wichtiger. Aufgrund der Neuheit von BPMN4TOSCA sind noch nicht viele Möglichkeiten dafür vorhanden. Ein Beispiel für einen BPMN4TOSCA Modeler wird von Michelbach gezeigt [Mic15]. Sollen BPMN4TOSCA Pläne ausführbar gemacht werden, müssen diese nach bestimmten Regeln, die von Kopp et al. festgelegt werden, in BPMN umgewandelt werden [KBBL12]. Dabei müssen die Ausführungsschritte, die zum Beispiel beim *TOSCA Data Object* bei der Darstellung wegfallen, ebenfalls angezeigt werden. Da das Ziel beim Visualisieren von Plänen in dieser Arbeit die Darstellung des kompletten Prozessablaufs mit allen wichtigen Informationen ist, kann diese auch direkt mithilfe von BPMN erfolgen. BPMN4TOSCA Elemente, die nur eine bestimmte TOSCA Operation darstellen, wären trotzdem verwendbar. Elemente wie das *TOSCA Data Object*, die einen Teil des Prozessablaufs weglassen, um die Übersichtlichkeit zu verbessern, können aber nicht verwendet werden. Sonst wird der Prozessablauf dadurch fehlerhaft dargestellt. Enthält ein Plan keine TOSCA Operationen, sind die BPMN4TOSCA Elemente ebenfalls nicht notwendig.

Hängt die Visualisierung eng einem auf TOSCA basierenden System zusammen, können passende BPMN4TOSCA Elemente zur Darstellung von TOSCA Operationen verwendet werden. Ansonsten eignen sich andere graphische Sprachen wie BPMN mehr, da diese nicht auf TOSCA beschränkt sind. Außerdem existieren bereits mehrere Werkzeuge und Viewer, die für die Visualisierung verwendet werden können. Die Funktionalität der Elemente, die für die Visualisierung verwendet werden, muss nicht ihrer Definition entsprechen, sondern soll Elemente der Sprache des Plans darstellen. Die speziellen Funktionen der BPMN4TOSCA Elemente werden deshalb nicht benötigt. Deswegen wird sich in dieser Arbeit gegen eine Verwendung von BPMN4TOSCA als graphische Sprache zur Visualisierung entschieden. Eine Implementierung des Konzepts mithilfe von BPMN4TOSCA ist trotzdem möglich und sollte für ähnliche Anwendungszwecke bedacht werden, bei denen die Funktionalität der Elemente eine größere Rolle spielt.

4 Use Case Szenario

In diesem Kapitel wird gezeigt, wie die Bereitstellung eines Plans für eine Cloud Anwendung in OpenTOSCA abläuft und wie Benutzer die Korrektheit des Plans verifizieren können. Im ersten Abschnitt wird eine Beispieltopologie beschrieben, die als Use Case verwendet wird. Der zweite Abschnitt beschreibt die Vorgehensweise bei der Bereitstellung einer neu erstellten Anwendung. Im nächsten Abschnitt wird anhand des Use Case gezeigt, was ein Nutzer beim Anpassen eines Bereitstellungsplans beachten muss. Dadurch werden die einzelnen Schritte die der Nutzer befolgen muss, sowie die Probleme, die auftreten können, gezeigt. Am Ende des Kapitels werden Lösungsansätze besprochen, die an die auftretenden Probleme angepasst sind.

4.1 Topologie von MyTinyToDo

Dieser Abschnitt verwendet die *MyTinyToDo_Bare_Docker Service Template* zur Darstellung des Ablaufs und möglicher Probleme beim Bereitstellen. Beim Erstellen einer *MyTinyToDo_Bare_Docker*[1] Instanz wird eine DockerEngine erstellt, auf der ein Docker Container, der eine einfache To-do-Liste[2] beinhaltet, installiert wird. Diese To-do-Liste kann dann vom Benutzer bearbeitet werden.

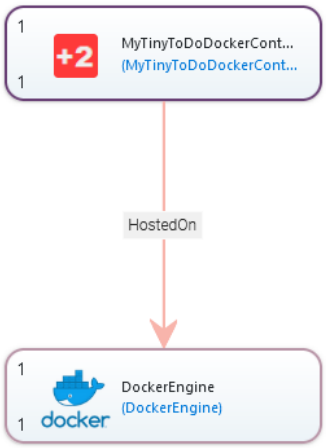

Abbildung 4.1: Topologie von MyTinyToDo

[1] https://github.com/ptrckkk/myTinyTodo
[2] https://www.mytinytodo.net/

MyTinyToDo_Bare_Docker besteht aus zwei *Node Templates* und nur einem *Relationship Template*. Damit ist diese Topologie recht klein gehalten, aber beinhaltet die wichtigsten Bestandteile, welche in größeren Topologien in größerem Maße vorkommen. Die *MyTinyToDoDockerContainer Node Template* benötigt Informationen, die für Operationen wie das Starten oder das Entfernen des Containers notwendig sind. Diese müssen korrekt übergeben werden, da es sonst zu Fehlern bei der Bereitstellung oder Instanziierung kommt. Diese Node Template ist über die *Relationship Template "Hosted-On"* mit der *Node Template DockerEngine* verbunden. Die *DockerEngine Node Template* benötigt ebenfalls die richtigen Informationen, um den Docker Container starten zu können. Ist alles funktionsfähig, kann der Benutzer nun, nach Erstellen einer Instanz, die To-do-Liste auf der entsprechenden Adresse bearbeiten (sei es localhost oder konkrete IP-Adresse).

4.2 Workflow

In diesem Abschnitt wird die Vorgehensweise beim Bereitstellen einer Cloud Anwendung im OpenTOSCA System beschrieben. Zu Beginn wird eine Topologie im OpenTOSCA System erstellt. Dazu muss der Benutzer diese im *Topology Modeler* der Winery erstellen. Alle wichtigen Variablen und Eigenschaften sollten in diesem Schritt in die entsprechenden Templates eingefügt werden. Anschließend werden diese in eine CSAR Datei gepackt und an den Container gesendet. Diese Datei dient dem gemeinsamen Informationsaustausch zwischen verschiedenen OpenTOSCA Komponenten. Mithilfe des Plangenerators im OpenTOSCA Container wird der Plan zum Bereitstellen der Anwendungen generiert. Dieser wird ebenfalls in der CSAR Datei gespeichert. Im nächsten Schritt kann die Anwendung nun mithilfe des Plans bereitgestellt und instanziiert werden. Falls notwendig können hier noch weitere Parameter vom Benutzer übergeben werden.

4.3 MyTinyToDo als Beispiel Use Case

Angenommen ein Nutzer möchte die To-do-Liste von *MyTinyToDo_Bare_Docker* um eine Funktion erweitern. Dafür ist es notwendig, dass beim Erstellen des Docker Containers eine zusätzliche Variable, welche notwendige Daten für die neue Funktion enthält, hinzugefügt werden muss. Diese Variable ist schnell ergänzt, aber ob der Docker Container hinterher noch richtig funktioniert ist fragwürdig. Um dies zu testen, muss der Nutzer nun den angepassten Plan bereitstellen und versuchen diesen zu instanziieren. Oft funktioniert dies aber nicht direkt und zwingt den Nutzer, in diesem Fall den Fehler zu finden. Das erfordert aber in den meisten Fällen das Durchsuchen des Bereitstellungsplans, welcher aus BPEL-Code besteht. Für unerfahrene Nutzer bedeutet dies, den schwer lesbaren Code nach einer Stelle zu durchsuchen, an der eine Variable falsch, oder zu früh gesetzt wird. Das ist oft sehr anstrengend, da viele verschiedene Operationen, Zuweisungen und HTTP Aufrufe im Plan mehrfach vorkommen. Beispiele dafür sind in Abbildung 4.2 zu sehen. Zusätzlich dazu muss auf die TOSCA Operationen geachtet werden, da diese bei der Ausführung im richtigen TOSCA Zustände sein müssen. Selbst für erfahrene Nutzer ist dies anstrengend, da selbst kleine Pläne wie der von *MyTinyToDo_Bare_Docker*, welcher nur zwei *Node Templates* besitzt, schon über tausend Zeilen an BPEL-Code enthalten. Ist der vermeintliche Fehler gefunden, ist es nötig den Plan wieder neu bereitzustellen und zu instanziieren. Tritt wieder ein Fehler auf, muss die Prozedur wiederholt werden. Deshalb wäre es von Vorteil, das Durchsuchen des Bereitstellungsplans zu vereinfachen, um schneller das gewünschte Ergebnis zu erreichen.

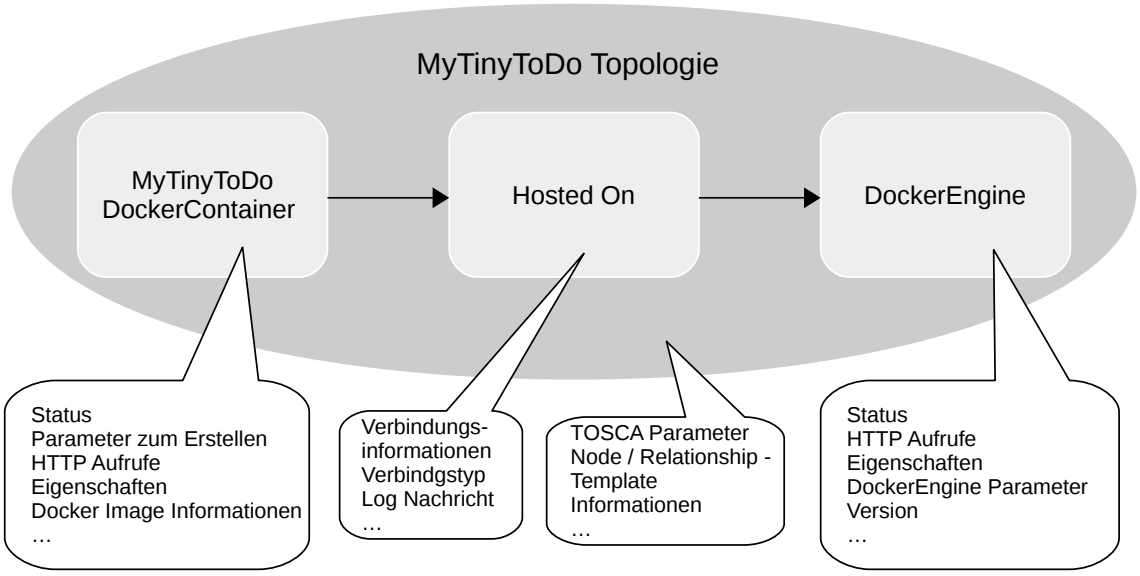

Abbildung 4.2: Informationen innerhalb der Topologie

4.4 Lösungsansätze

Für das Ziel, den Bereitstellungsplan einfacher zugänglich und lesbar zu machen, gibt es leider keine einfach durchführbare Lösung. Der Plan besteht aus sehr viele Zeilen Code, was das Verständnis von sich aus schon erschwert. Dazu kommt die Menge an verschiedenen Operationen und Befehlen, die im Plan vorkommen und alle spezifische Informationen besitzen. Im Fall des Use Cases müssen beim Hochfahren des Docker Containers, zusätzlich zu den Parametern der Docker Engine, die TOSCA-Zustände beachtet werden, was die Menge an Information noch weiter erhöht. Die verschiedenen Abläufe innerhalb des Prozesses im Plan sind nicht der Reihe nach sortiert, weshalb diese erst erschlossen werden muss. Außerdem ist der BPEL-Code meist nicht formatiert, was zusätzliche Schwierigkeiten bereitet. Die trivialste Lösung für letzteres wäre, den BPEL-Code des Bereitstellungsplans zu formatieren beziehungsweise automatisch richtig einzurücken. Dazu sind bereits Werkzeuge vorhanden, die diese Aufgabe erfüllen können. Da der Sinn und Zweck von BPEL-Code in der Kommunikation mit anderen Prozessen und Services besteht, und eine direkte Interaktion mit menschlichen Nutzern nicht vorgesehen ist, sind andere Ansätze vorzuziehen. Ein anderer Ansatz wäre, den Prozessablauf des BPEL-Codes graphisch darzustellen um dem Benutzer das Verständnis zu erleichtern. Hierfür gibt es mehrere Lösungen. Eine dieser Lösungen ist die Erzeugung einer Graphik, die den Prozessablauf beschreibt. Das Problem der Formatierung des Codes fällt damit weg. Es muss aber darauf geachtet werden den Prozessablauf in richtiger Reihenfolge darzustellen, da dieser nicht sortiert vorliegt. Außerdem müssen die verschiedenen Arten von Befehlen und Operationen entsprechend behandelt und passend dargestellt werden. Damit kann der Benutzer den Ablauf des Prozesses einfach verstehen und ist nicht dazu gezwungen den BPEL-Code des Plans zu durchsuchen. Der Nachteil hierbei besteht in der fehlenden Übersichtlichkeit der Graphik. Die Graphik kann zwar den Prozess darstellen, aber zusätzliche Informationen wie bestimmte Variablen bei BPEL Befehlen oder Operationen sind schwer darzustellen, ohne die Übersichtlichkeit zu verlieren. Dank der Menge an Informationen in einem Plan, die beispielsweise in Abbildung 4.2 dargestellt wird, kann dies leicht passieren.

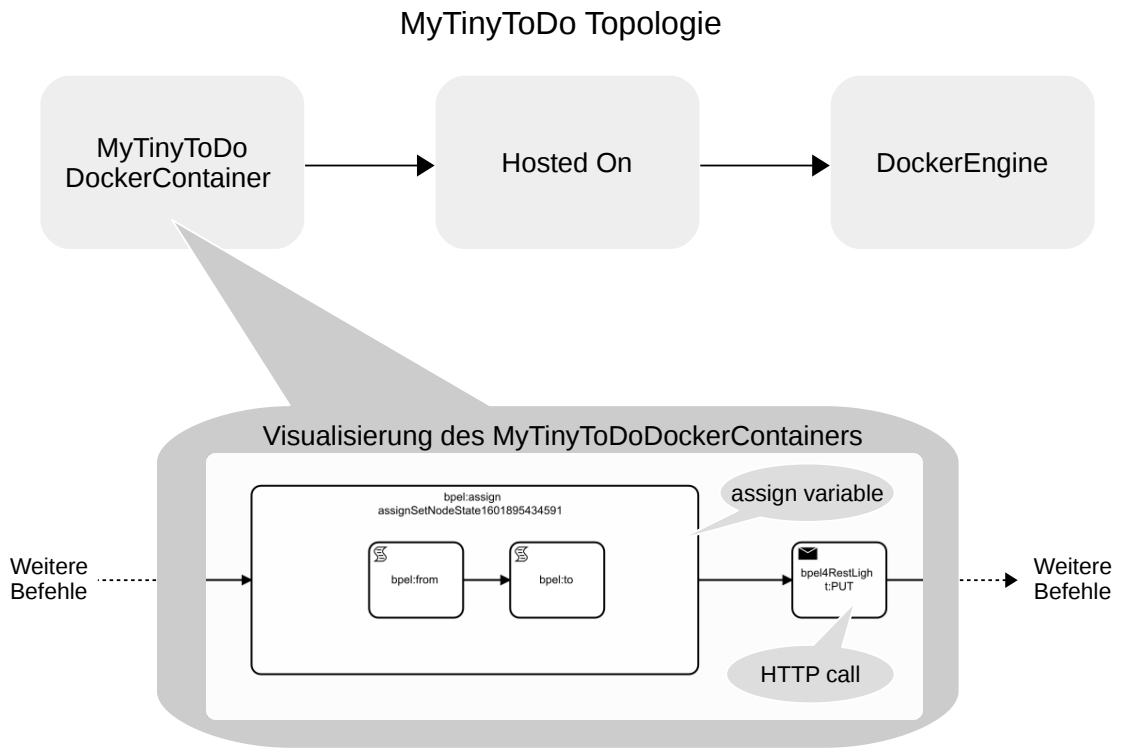

Abbildung 4.3: Ausschnitt aus der Visualisierung der MyTinyToDo Topologie

Könnte man mit den verschiedenen BPEL Befehlen die den Code ausmachen interagieren, um die Details auszulesen, wäre das Problem der fehlenden Übersichtlichkeit gelöst. Diese Interaktionen werden mit der Verwendung eines Web-Renderers möglich gemacht. Dies ist der Lösungsansatz, der in dieser Arbeit verwendet wird. Der BPEL Prozess wird klar dargestellt und die Variablen jedes BPEL Befehls können durch das Anklicken des jeweiligen Elements angezeigt werden. Es müssen aber dieselben Punkte wie beim Ansatz der Erzeugung einer Graphik beachtet werden. Als Web-Renderer wird in dieser Arbeit der BPMN-Modeler von Camunda verwendet. Mithilfe dieses Modelers werden die verschiedenen BPEL Befehle als BPMN-Elemente dargestellt. Die Sprache BPMN wird zur Visualisierung verwendet, da ihre Elemente intuitiv verständlich und Prozessabläufe gut darstellbar sind. In Abbildung 4.3 ist ein Ausschnitt zu sehen welcher zeigt, wie die Visualisierung eines Bestandteils der Topologie des Use Case Beispiels aussieht. Zuordnungen von Variablen und Informationsaustausch mithilfe von HTTP Aufrufen können auf einen Blick erkannt werden. Damit kann der korrekte BPEL Befehltyp ausfindig gemacht werden und es muss nur noch der gesuchte Befehl gefunden werden. Die Suche wird von einem großen Codeblock auf einen vergleichsweise kleineren Teil der BPMN Visualisierung reduziert.

Um das Durchsuchen des Plans nach Fehlern komplett zu trivialisieren, wäre die endgültige Lösung die automatisierte Fehlersuche. Der Zeitgewinn beim Testen eines Plans wäre immens, aber die Realisierung des Konzepts der automatischen Fehlersuche ist schwierig. Verschiedenste Variationen von Änderungen müssen dabei erkannt werden, um die Fehlersuche möglich zu machen. Außerdem ist das Garantieren der Korrektheit oft nicht möglich, da manche Fehler erst beim Bereitstellen oder Instanziieren auftreten. Deshalb ist dies ein Projekt für die Zukunft.

5 Konzept

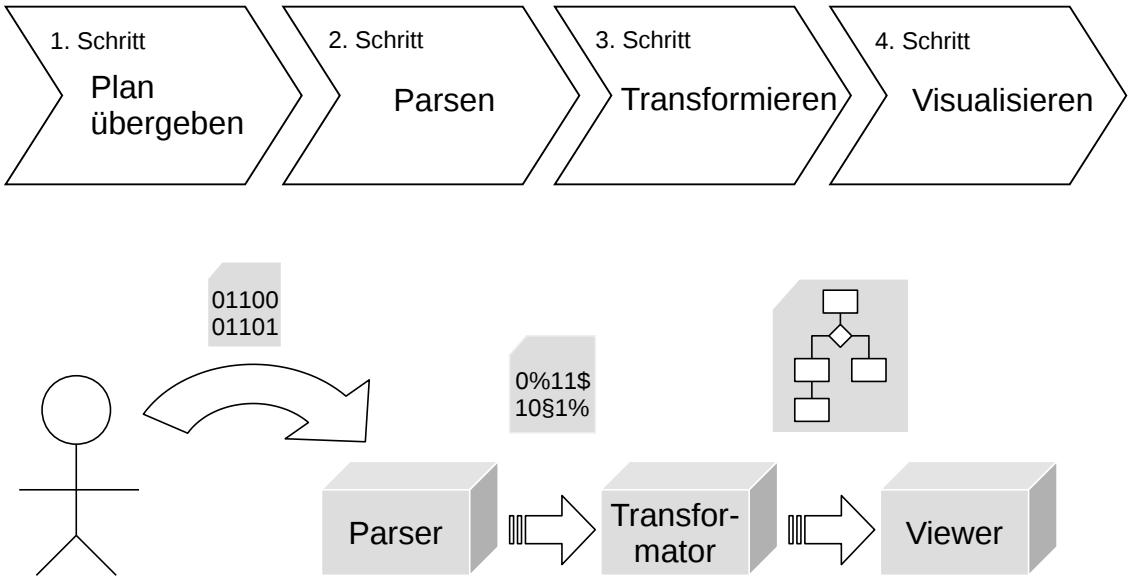

Abbildung 5.1: Vorgehensweise des Konzepts

In diesem Kapitel wird ein Konzept zur Visualisierung von generierten Plänen vorgestellt. Um sinnvolle Ergebnisse bei der Visualisierung zu erhalten ist es wichtig, alle notwendigen Schritte, die zum Endergebnis führen, aufeinander anzupassen. Deshalb werden die Hauptbestandteile des Konzepts zunächst einzeln betrachtet. Der erste Abschnitt beinhaltet die Voraussetzungen, die im Rahmen des Konzepts erfüllt werden müssen. Im zweiten Abschnitt wird beschrieben, wie mithilfe eines Parsers die zur Visualisierung notwendigen Informationen gewonnen werden. Der dritte Abschnitt beschreibt die Transformation der vom Parser gewonnenen Informationen in eine graphische Sprache. Im vierten Abschnitt wird erläutert, wie ein Viewer verwendet wird, um die Visualisierung darzustellen. Im letzten Abschnitt wird das Konzept als Ganzes betrachtet und es wird nochmal auf den Use Case des vorherigen Kapitels eingegangen.

5.1 Vorraussetzungen

Das hier vorgestellte Konzept setzt voraus, dass der Plan, der an den Parser übergeben wird, bestimmte Kriterien erfüllt. Es wird davon ausgegangen, dass der übergebene Plan valide ist, da sonst das Parsen nicht möglich ist. Die Übergabe des Plans wird von einem Nutzer durchgeführt, kann aber auch automatisch implementiert werden. Der Parser und der Transformator sollten aufeinander abgestimmt sein um die Durchführung zu erleichtern. Zur Visualisierung sollte eine

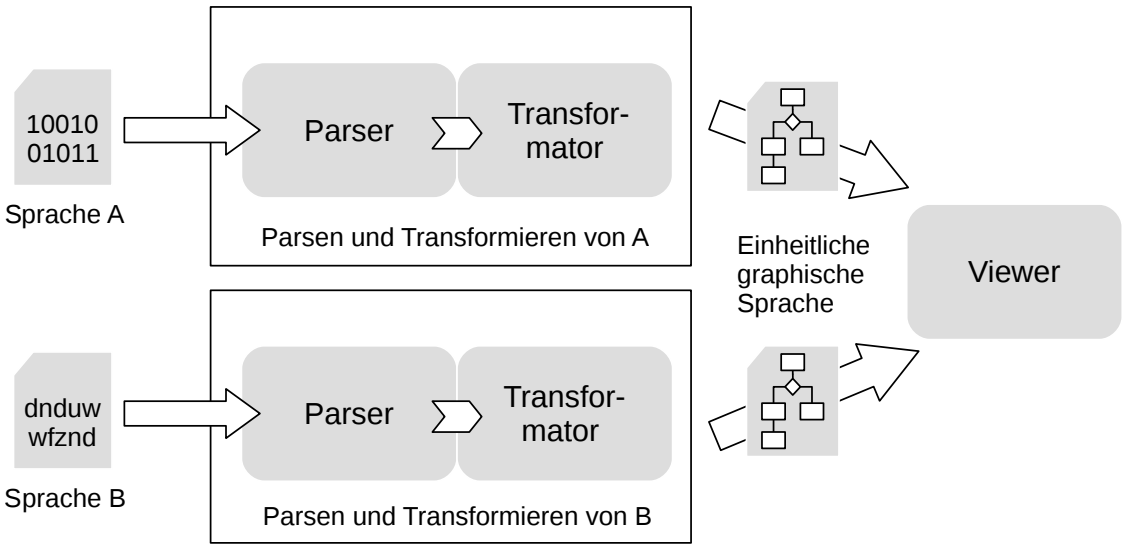

Abbildung 5.2: Architektur des Systems

geeignete graphische Sprache gewählt werden, um eine übersichtliche Darstellung des Plans zu ermöglichen. Die gewählte Sprache muss das Ziel aller Transformationen, unabhängig von der zu parsenden Sprache, sein. Dadurch wird eine einheitliche Darstellung erzeugt. Zuletzt sollte ein geeigneter Viewer zum Visualisieren verwendet werden, der die graphische Sprache, in die transformiert wird, darstellen kann. Diese Darstellung sollte möglichst übersichtlich sein. Wenn diese Voraussetzungen erfüllt sind, ist die Ausführung des Konzepts möglich.

5.2 Informationsgewinnung aus den Plänen

Wird der übergebene Plan korrekt geparsed, werden die wichtigsten Informationen aus diesem gewonnen und gespeichert. Hierbei müssen die Eigenschaften des Plans bei der Wahl des Parsers beachtet werden, die Probleme verursachen können. Die Länge des Plans und die Anzahl der Operationen, welche in diesem enthalten sind, sind entscheidend für die Wahl des Parsers. Ist der Plan eher klein und besitzt nur wenige Operationen, wären Parser Arten wie der DOM-Parser eine gute Wahl, da einfacher mit dieser Art von Parsern zu arbeiten ist. Die Informationen werden bei diesem Parser mithilfe einer Baumstruktur gespeichert und können jederzeit abgerufen werden. Da dieser aber meist den gesamten Plan abspeichern muss, ist die Verwendung dieser Parser Art bei großen Plänen mit vielen Operationen keine effiziente Lösung. In diesem Fall wären Parser, die auf bestimmte Events im Plan reagieren, eine bessere Wahl. Wird ein spezifischer Abschnitt innerhalb des Plans erreicht, wird eine Markierung gesetzt. Verlässt der Parser den Abschnitt, wird die Markierung wieder entfernt. Dadurch wird der Plan Schritt für Schritt geparst. Mithilfe solcher Markierungen kann sich der Parser orientieren und die genaue Position im Plan ist jederzeit bekannt. Außerdem kann auf diese Weise zwischen verschiedenen Operationen differenziert werden. Werden bei Operationen nicht alle Informationen benötigt, kann mithilfe der Markierungen bestimmt werden, was ausgelesen wird. Dies ist vor allem bei ähnlichen Operationen mit unterschiedlichem Inhalt nützlich. Bei der Wahl des Parsers muss natürlich auch auf die Sprache des zu parsenden Plans geachtet werden. Sollten Pläne in verschiedenen Sprachen vorliegen, muss für jede Sprache ein

passender Parser gewählt werden. Nachdem das Parsen abgeschlossen ist, werden die gewonnenen Informationen an den Transformator weitergegeben. Eventuell müssen manche Inhalte zuvor noch sortiert werden, um später die richtige Reihenfolge bei der Visualisierung zu erzeugen. In diesem Konzept sind Parser und Transformator aneinander angepasst und benutzen ein einheitliches Informationsformat. Dadurch besteht ein starker Zusammenhang der beiden Bestandteile. Alternativ könnten die Parser die gewonnenen Informationen in eine einheitliche Übergangssprache umwandeln, wodurch nur eine Art von Transformator notwendig ist, der diese Sprache transformieren kann. Dies ist aber schwieriger umzusetzen, da die ein zusätzlicher Umwandlungsschritt notwendig ist, um die mithilfe des Parsers gewonnenen Informationen in der Übergangssprache darzustellen. Außerdem erfordert die Spezifikation einer solchen Übergangssprache zusätzlichen Aufwand. Deswegen wurde sich in diesem Konzept dagegen entschieden.

5.3 Transformieren der Informationen

Die Informationen, die vom Parser geliefert werden, müssen nun verarbeitet werden, um das Ziel der Visualisierung zu erreichen. Dies ist aber nicht mit der Sprache des übergebenen Plans möglich, da diese nicht zur graphischen Darstellung geeignet ist. Deshalb müssen die Informationen aus dem Plan erst in eine Sprache, die zur Visualisierung verwendet werden kann, transformiert werden. Da das Format der Informationen durch Anpassen von Parser und Transformator schon passend ist, kann das Transformieren direkt durchgeführt werden, ohne die Informationen umwandeln zu müssen. Sind Pläne in verschiedenen Sprachen vorhanden, ändert sich das Vorgehen nicht, da Parser und Transformator in diesem Konzept für jede Sprache aneinander angepasst sind (siehe Abbildung 5.2). Die Sprache, die zur Visualisierung verwendet wird, soll dabei einheitlich sein. Gibt es mehrere Paare aus Parsern und Transformatoren, müssen diese als Ergebnis dieselbe graphische Sprache liefern. Dabei soll eine Sprache gewählt werden, mit der Prozesse und Operationen klar dargestellt werden können. Die Wahl der Elemente, die zur Visualisierung verwendet werden, soll Sinn ergeben, damit Nutzer keine zusätzliche Zeit verbringen müssen, um diese zu Verstehen. Deshalb müssen Elemente gewählt werden, deren Bedeutung intuitiv mit der ursprünglichen Bedeutung der Informationen in Verbindung gebracht werden kann. In dieser Arbeit fällt die Wahl der Sprache deshalb auf BPMN. So kann zum Beispiel ein Prozess aus dem mithilfe von OpenTOSCA erzeugten Bereitstellungsplan als Unterprozess in BPMN dargestellt werden. Befehlsblöcke innerhalb eines Prozesses werden durch korrespondierende *Tasks* innerhalb des Unterprozesses dargestellt. Der Inhalt der Variablen von Befehlsblöcken wird in die Taskbeschreibung übertragen. Somit erfolgt die Transformation ohne Probleme und die Visualisierung kann durchgeführt werden. Wichtig ist es auch, dass die Namen von Prozessen und Operationen ebenfalls übertragen werden, um die Übersichtlichkeit zu verbessern und die Zuordnung zu erleichtern. Nutzer können dadurch auf den ersten Blick erkennen, welchen Prozess oder welche Operation visualisiert wird.

5.4 Visualisierung im Viewer

Wurden alle relevanten Informationen aus dem Plan beim Parsen entnommen und in die gewählte graphische Sprache transformiert, müssen diese nun visualisiert werden. Die Art der Visualisierung ist für Nutzer der wichtigste Teil dieses Konzepts, da die Interaktion fast nur mit dieser erfolgt. Ist die Visualisierung nicht klar strukturiert, oder nur schwer verständlich, ist der Nutzer eher abgeneigt

diese zu verwenden. Deshalb dürfen auch keine wichtigen Informationen bei der Darstellung fehlen, da der Nutzer sonst gezwungen ist, diese im Plan zu suchen. Damit wäre das Ziel, die Suche nach Fehlern zu verkürzen, verfehlt. Deshalb werden die wichtigsten Informationen beim Parsen herausgefiltert und sortiert, um eine gut strukturierte, übersichtliche Visualisierung zu erzeugen. Soll die Visualisierung für andere Zwecke weiterverwendet werden ist oft eine Anpassung nötig. Um Anpassungen an der Darstellung vorzunehmen und einfache Erweiterbarkeit möglich zu machen ist die Verwendung eines Web-Renderers als Viewer optimal. Im Gegensatz zu Graphiken kann der Nutzer direkt im Renderer Änderungen vornehmen. So können beispielsweise vom Nutzer bei bestimmten Elementen Kommentare hinzugefügt werden, oder neue Elemente ergänzt werden. Dadurch kann der Nutzer seine Funde festhalten und Lösungsvorschläge spekulieren. Mithilfe von Plugins kann ein Renderer noch weiter personalisiert werden, um jedem Nutzer eine bequeme Arbeitsumgebung zu bieten. Zum Visualisieren wird in dieser Arbeit deshalb, wie in Kapitel 2.4 beschrieben, ein BPMN-Modeler verwendet, der als Web-Renderer agiert. Mithilfe dieses Modelers wird das Visualisieren der transformierten Prozesse und Operationen aus dem TOSCA Bereitstellungsplan vereinfacht. Außerdem wird die *properties-panel* Erweiterung des Modelers verwendet, um Informationen der visualisierten BPMN-Elemente besser darzustellen. Dadurch erhält der Nutzer durch anklicken des Elements alle Informationen auf einen Blick. Sollen andere Viewer als Web-Renderer verwendet werden ist es empfehlenswert, zur Übersichtlichkeit der Informationen eine ähnliche Darstellungsweise für diese zu verwenden. Zuletzt ist bei der Visualisierung zu beachten, dass zwar valide BPMN-Elemente erzeugt werden, diese aber in ihrer Bedeutung den transformierten Operationen aus dem Bereitstellungsplan entsprechen. Das Ziel ist nicht ein äquivalenter, funktionsfähiger BPMN-Prozess, sondern das Visualisieren des Prozessablaufs im Bereitstellungsplan mithilfe der BPMN-Elemente.

5.5 Zusammenfassung und Eingehen auf den Use-Case

Im Großen und Ganzen betrachtet ist das Konzept einfach zu verstehen. Es wird ein Plan, der Visualisiert werden soll als Input gegeben. Dieser wird geparsed, wobei die relevanten Informationen entnommen, sortiert und in eine geeignete graphische Sprache transformiert werden. Können Pläne in verschiedenen Sprachen vorliegen, werden diese entsprechend geparsed und in dieselbe, einheitliche graphische Sprache transformiert. Ein passender Viewer wird zur Darstellung der Visualisierung verwendet. Der Viewer stellt die Visualisierung in der gewählten graphischen Sprache dar. Dies ist in dieser Arbeit ein BPMN-Modeler, der den Prozessablauf des Bereitstellungsplans mithilfe von sinnvollen BPMN-Elementen darstellt. Auf die detaillierten Informationen kann mithilfe des Panels zugegriffen werden. Das macht die Benutzung des Konzeptes intuitiv. Um das zu zeigen, wird nochmals auf das Beispiel aus Kapitel 4.3 eingegangen. Hier möchte ein Nutzer eine zusätzliche Variable beim Erstellen des Dockercontainers hinzufügen. Das Konzept kommt hier ins Spiel, da der Nutzer nicht jedes Mal den Plan bereitstellen und instanziieren will, um diesen auf Funktionsfähigkeit zu testen. Der Nutzer übergibt den Plan an den Parser und muss nun auf die Visualisierungstaste im Modeler drücken. Damit wird der Plan automatisch geparsed, transformiert und an den Modeler zum Visualisieren übergeben. Dieser Prozess muss nicht durch den Nutzer beeinflusst werden. Die Interaktion des Nutzers beschränkt sich auf den Modeler, da der Rest automatisch erfolgt. Der Nutzer sieht nur den fertigen Plan und kann mit diesem durch Anklicken der Elemente interagieren. Die Visualisierung kann nun vom Nutzer beliebig bearbeitet werden. Um nach der extra Variable zu suchen, die der Nutzer einbauen möchte, kann dieser bequem die Visualisierung nach der richtigen

Stelle durchsuchen. Da in diesem Fall die Variable im Dockercontainer sein muss, können alle anderen scopes ignoriert werden. Eventuell anzupassende TOSCA Parameter befinden sich ebenfalls in dieser scope. Der Nutzer kann sich also auf den Abschnitt konzentrieren, der relevant ist, ohne diesen jedes Mal im Bereitstellungsplans suchen zu müssen, da er in der Visualisierung einfach identifiziert werden kann. Dieser Abschnitt kann leicht markiert werden, wodurch der Nutzer auch nach einer längeren Pause die richtige Stelle schnell wiederfindet. Somit kann die Zeit bei der Fehlersuche reduziert werden und die Effizienz bei der Suche wird erhöht.

6 Implementierung

In diesem Kapitel wird auf die Implementierung des im vorherigen Kapitel beschriebenen Konzepts eingegangen. Die Vorgehensweise beim Parsen des Bereitstellungsplans, welcher mithilfe des Plangenerators in der OpenTOSCA Umgebung erzeugt wird, wird beschrieben. Dabei spielt die Funktionsweise des SAX-Parsers eine große Rolle, da die Implementierung auf diesen angepasst ist. Die Transformation der mithilfe des Parsers gewonnenen Informationen in die graphische Sprache BPMN wird im nächsten Abschnitt beschrieben. Im dritten Abschnitt wird die Implementierung der Visualisierung mithilfe des BPMN-Modelers von Camunda beschrieben. Das *Modeling API* des Modelers spielt hierbei eine wichtige Rolle, da die Visualisierung mithilfe dieses APIs direkt durch das Auslesen der vom Parser gelieferten Informationen erzeugt werden kann. Am Ende des Kapitels werden noch einige Anmerkungen zur Funktionsweise, sowie zu den Einschränkungen und Limitierungen der Implementierung gemacht.

6.1 Parser

Wie schon im Konzept angesprochen wird, ist der Parser, welcher in dieser Arbeit verwendet wird, ein eventbasierter Parser. Dieser verwendet die SAX Technik zum Parsen und wird daher SAX-Parser genannt. Die speziellen Eigenschaften dieses Parsers, die in Kapital 2.7 beschrieben werden, sind für die Implementierung wichtig. Dies liegt vor allem an der Länge der zu parsenden Bereitstellungspläne und dem Filtern der zur Visualisierung relevanten Informationen. Der SAX-Parser, der in dieser Implementierung verwendet, wird heißt *sax-ts*[1], ein SAX-Parser der für TypeScript aufgelegt ist. Da der Modeler in der Implementierung als Angular Komponente vorliegt und Angular auf TypeScript basiert, ist ein passender SAX-Parser notwendig, um die Korrektheit der geparsten Informationen zu garantieren. Dieser wird mithilfe des *npm* Paketmanagers[2] hinzugefügt. Die eigentliche Parser Implementierung erfolgt als Angular Service, welcher einfach in eine andere Angular Komponente eingefügt werden kann. In diesem Fall ist diese Komponente der Camunda Modeler, welche gleichzeitig auch als Hauptkomponente dient.

Beim Parsen des Bereitstellungsplans reagiert der SAX-Parser auf verschiedene Events. In dieser Implementierung sind die wichtigsten Events diejenigen, die durch Start- und End-Tags im BPEL-Code ausgelöst werden. Diese Events sind dafür zuständig bestimmte Markierungen zu setzen, die verwendet werden, um festzustellen, in welchem Teil des BPEL-Codes sich der Parser befindet. Befindet sich zum Beispiel in der aktuellen Zeile ein Start-Tag von *bpel:assign*, wird diese Markierung mithilfe einer Variable gespeichert, wodurch der Parser das Wissen erhält, dass er sich

[1] https://github.com/Maxim-Mazurok/sax-ts
[2] https://www.npmjs.com

in einem *bpel:assign* Befehlsblock aufhält. Erreicht der Parser den End-Tag von *bpel:assign*, weiß der Parser, dass das Ende des Befehlsblocks erreicht ist. Somit weiß der Parser immer, wo er sich befindet und wie der Inhalt des aktuellen Blocks behandelt werden muss.

Der Anfang eines BPEL-Prozesses, in diesem Fall der Beginn des Plan-Ablaufs, ist das *flow* Element. Dementsprechend beginnt der Parser an dieser Stelle, festgelegt durch eine Markierung. Als nächstes folgen alle Verbindungen zwischen den Hauptbestandteilen. Diese Verbindungen werden *links* genannt. Mithilfe der links kann die genaue Reihenfolge, in der die Hauptbestandteile vorkommen, festgelegt werden. Die Hauptbestandteile, welche *scopes* genannt werden, kommen nach den links im flow vor. Alle scopes beinhalten entsprechende links, mit denen sie mit anderen scopes verbunden sind. Da die scopes standardmäßig nicht sortiert im Plan vorliegen, ist es ohne die links nicht möglich eine Reihenfolge aufzustellen. Deshalb werden die links extra gespeichert, um hinterher die scopes zu sortieren. Jede scope stellt ein bestimmtes *Node Template* oder *Relationship Template* dar. Alle scopes bestehen aus drei Ablaufsequenzen, die in bestimmten Fällen benutzt werden. Die Hauptsequenz, welche *mainsequence* im Plan genannt wird, stellt den Normalfall dar und wird standardmäßig ausgeführt. Die anderen Fälle treten auf, falls ein Fehler vorkommt oder eine Kompensation erforderlich ist. Diese sind Spezialfälle der scope und heißen im Plan jeweils *fault scope* und *compensation scope*. Es ist wichtig, die Informationen der fault- und compensation scope vom Standardfall zu trennen, da sonst der Ablauf nicht korrekt wiedergegeben wird.

Innerhalb der scopes liegen die verschiedenen Befehlsblöcke vor. Befindet sich der Parser innerhalb eines Befehlsblocks, muss der Inhalt dieses Blocks ausgelesen und gespeichert werden, sofern dieser relevant und nicht leer ist. Ist der Befehlsblock beispielsweise wieder ein *bpel:assign* Block, so werden innerhalb dieses Blocks Informationen von einer Variable zu einer anderen übertragen. Das Speichern dieser Informationen wird jeweils durch die Markierung der Start-Tags von *bpel:from* und *bpel:to* ausgelöst. Da aber auch Fälle vorkommen, bei denen Daten aus HTTP-Aufrufen übertragen werden, nur Teilinformationen benötigt werden, oder *Strings* ausgelesen werden, müssen diese extra behandelt werden. Die meisten dieser Fälle werden durch das Auslösen des *cdata* Events des Parsers behandelt. Im Fall von Strings, müssen diese ebenfalls durch ein extra Event des Parsers behandelt werden, da Strings oft nicht sichtbare, leere Strings sein können. Dies ist in dieser Implementierung ein großes Problem, da im Bereitstellungsplan bestimmte TOSCA-Variablen wie etwa die auszuführende Operation oder die ID der *Service Template* vorkommen und als String vorliegen. Diese TOSCA-Variablen kommen dabei direkt hintereinander vor, weshalb der Parser einen String nach dem anderen auslesen muss. Zwischendurch kommen hierbei oft leere Strings vor, die nicht relevant sind. Da aber auch TOSCA-Variablen leer sein können muss der Parser zwischen relevanten und irrelevanten leeren Strings Unterscheiden können. Deshalb wird bei der Implementierung ein Zähler verwendet, der beim Auslesen der Informationen den richtigen String zur entsprechenden Variable zuordnet.

Beim Behandeln einiger Befehlsblöcke ist es außerdem notwendig, dass der Parser das Elternelement im BPEL-Code kennt. Da ein SAX-Parser aber Zeile für Zeile arbeitet, sind Zusammenhänge zwischen den Elementen unbekannt. In diesem Fall liefern diese Zusammenhänge aber die benötigten Informationen. Dieses Problem wird durch die Verwendung eines Stapels gelöst, auf den nach der Behandlung eines Start-Tags, dieser Tag auf den Stapel gelegt wird. Tritt der entsprechende End-Tag auf, wird dieser wieder vom Stapel genommen. Dadurch sind die Elternelemente immer bekannt und können durch das Anschauen des obersten Stapelelements verwendet werden.

Algorithmus 6.1 Sortieren der scopes

Input:
dataArr[] - Array mit unsortierten scopes
seqEdges[] - Array mit allen links
Output:
sortedArray[] - Array mit sortierten scopes

```
1:  procedure SORTIEREN(dataArr[],seqedges[])
2:      sortedArray[] ← ∅
3:      for seqEdges[].lenght + 1 do
4:          for each scope ∈ dataArr[] do
5:              for each edge ∈ scope.incomingEdges[] do
6:                  if edge ∉ seqEdges[] then
7:                      incomingEdges[].remove(edge)
8:                  end if
9:              end for
10:         end for
11:         for each scope ∈ dataArr[] do
12:             if scope.incomingEdges[].lenght == 0 then
13:                 if scope.outgoingEdges[].lenght > 0 then
14:                     for each egde ∈ outgoingEdges[] do
15:                         seqEdges[].remove(edge)
16:                     end for
17:                 end if
18:                 sortedArray[].push(scope)
19:                 dataArray[].remove(scope)
20:             end if
21:         end for
22:     end for
23:     return sortedArray[]
24: end procedure
```

Das Speichern von relevanten Informationen aus den Befehlsblöcken erfolgt mithilfe von Javascript-Objekten und Arrays. Die links zur Bestimmung der Reihenfolge der scopes werden in einem Array gespeichert. Jede scope wird innerhalb eines Javascript-Objektes gespeichert und diese werden ebenfalls in ein Array eingefügt. Zusätzlich werden alle zum scope gehörigen eingehenden und ausgehenden links, innerhalb eines Arrays zum Objekt hinzugefügt. Die Befehlsblöcke innerhalb einer scope werden ebenfalls als Objekte mit ihren Informationen gespeichert. Damit besitzt jedes scope Objekt mehrere Befehlsblockobjekte und zwei Arrays mit links. Da innerhalb einer Scope schon die richtige Ablaufreihenfolge vorliegt, müssen nur noch die *fault* und *compensation* Abschnitte gekennzeichnet werden. Dies passiert mithilfe einer Markierung bei Beginn des entsprechenden Abschnitts. Im letzten Schritt müssen die scopes noch nach Ablaufreihenfolge sortiert werden. Dazu wird ein Algorithmus ausgeführt, der in 6.1 zu sehen ist.

Der Ablauf des Algorithmus erfolgt in jeweils zwei Schritten: Im ersten Schritt wird für jede eingehende Kante in jedem scope überprüft, ob sie noch im Array mit allen links enthalten ist. Ist dies nicht der Fall, wird sie im Array der eingehenden Kanten des entsprechenden scopes gelöscht.

Das Array mit allen links dient dabei zur Kontrolle, ob die Kante noch als eingehende Kante existieren darf. Im zweiten Schritt wird für jede scope überprüft, ob diese noch eingehende Kanten hat, also das entsprechende Array mit eingehenden links nicht leer ist. Ist dies der Fall, ist die aktuelle scope die nächste im Prozessablauf, da sie keine Vorgänger hat. Die scope wird dann in ein neues Array eingefügt, welches alle scopes und somit alle Informationen in richtiger Reihenfolge beinhaltet, und aus dem alten Array gelöscht. Alle ausgehenden Kanten dieser scope werden dann aus dem Array, welches alle links enthält, entfernt. Damit können im ersten Schritt des nächsten Durchlaufs alle scopes angepasst werden, um die nächste scope im Prozessablauf zu finden. Dies funktioniert, da jede Kante, also jeder link, immer jeweils zwei scopes miteinander verbindet. Durch diese Vorgehensweise werden alle scopes nach spätestens der Anzahl aller links addiert mit eins sortiert. Das sortierte Array wird dann an den Service weitergegeben, der für die Transformation und die Visualisierung im BPMN-Modeler zuständig ist.

6.2 Transformator

Um die vom Parser gelieferten Informationen zu visualisieren, ist ein Transformator notwendig, der diese in eine graphische Sprache umwandelt. In dieser Arbeit wird BPMN als graphische Sprache verwendet, um Bereitstellungspläne zu visualisieren. Die Aufgabe des Transformators ist nun, die verschiedenen Arten von Befehlsblöcke aus dem Bereitstellungsplan in passende BPMN-Elemente umzuwandeln. Wie bereits in Kapitel 5.3 erwähnt, werden BPMN-Elemente gewählt, deren intuitive Bedeutung dem Inhalt des entsprechenden Befehlsblocks entspricht. So werden unter anderem *bpel:invoke* und *bpel:receive* entsprechend durch *BPMN-SendTask* und *BPMN-ReceiveTask* dargestellt. Dies ist ein gutes Beispiel für eine gelungene Transformation, da die Bedeutungen der Befehlsblöcke und Elemente sehr ähnlich ist. Der *bpel:receive* Block besitzt sogar einen fast identischen Namen wie der *BPMN-ReceiveTask*. Die Transformation von scopes in BPMN-Unterprozesse ist ebenfalls intuitiv, da beide weitere untergeordnete Informationen im Inneren besitzen, seien es Befehlsblöcke bei scopes oder weitere Tasks bei Unterprozessen. Sogar für die Sonderfälle wie fault- und compensationscopes gibt es passende BPMN-Elemente wie *BPMN-fault-event* und *BPMN-compensation-event*. Da es sich aber um grundlegend verschiedene Sprachen handelt, können nicht alle Befehlsblöcke aus dem BPEL-Plan mithilfe eines passenden BPMN-Elements dargestellt werden. Für *bpel:from* und *bpel:to* Blöcke, welche verwendet werden, um zum Beispiel Informationen aus einer Variablen in eine andere zu kopieren, gibt es keine BPMN-Elemente mit ähnlicher Bedeutung. In diesem Fall werden BPMN-Elemente verwendet, deren Bedeutung am nächsten angrenzt. Bei *bpel:from* und *bpel:to* fällt die Wahl auf den *BPMN-ScriptTask*, die das Kopieren von Variablen theoretisch mithilfe eines Skripts ausgeführt werden kann. Dieses BPMN-Element macht deshalb mehr Sinn als ein zufällig gewähltes BPMN-Element, wie etwa der *BPMN-HumanTask*, der hier nicht in den Kontext passt. Eine komplette Übersicht über alle transformierten Elemente kann in Tabelle 6.1 eingesehen werden.

In der Implementierung erfolgt die Transformation, indem die vom Parser gelieferten Informationen direkt vor der Visualisierung in BPMN-Elemente umgewandelt werden. Da die Informationen bereits in der richtigen Reihenfolge geliefert werden, müssen diese einfach nur der Reihe nach abgearbeitet werden. Deshalb ist es sinnvoll, die Transformation und Visualisierung in einem Schritt durchzuführen, anstatt die Elemente einzeln zu transformieren, wieder zu speichern und erst im nächsten Schritt zu visualisieren.

BPEL Elemente	BPMN Elemente
scope	Subprocess
fault scope	Subprocess + Error Start Event
compensation scope	Subprocess + Compensation Start Event
bpel:assign	Subprocess
bpel:invoke	Send Task
bpel:receive	Receive Task
bpel:from	Script Task
bpel:to	Script Task
bpel:if	Exclusive Gateway
bpel:throw	Task
bpel4RestLight:PUT	Send Task
bpel4RestLight:GET	Send Task
bpel4RestLight:POST	Send Task

Tabelle 6.1: Tabelle mit BPMN Transformationen

6.3 Modeler

Zur Implementierung der eigentlichen Darstellung der Visualisierung wird der bereits in Kapitel 2.4 erwähnte BPMN-Modeler von Camunda als Web-Renderer verwendet. Spezifisch wird *bpmn-js*[3] verwendet, die Browservariante des Modelers. Die Wahl fällt auf diesen Modeler, da er einfach verwendbare Möglichkeiten zur Darstellung von BPMN-Prozessen, sowohl von Hand als auch programmatisch, bietet. Diese werden sich in der Implementierung zu Nutze gemacht. Das sortierte Array, welches vom Parser übergeben wird, wird Schritt für Schritt ausgelesen und transformiert. Mithilfe des *Modeling-APIs* des Modelers werden die BPEL-Befehlsblöcke anschließend als BPMN-Elemente dargestellt, indem über die scopes iteriert wird. Durch die Verwendung des *Modeling-APIs* wird die Transformation und Visualisierung in einem Schritt ermöglicht.

In jeder Iteration erfolgt die Darstellung einer scope mit ihrem kompletten Inhalt. Dabei muss in jedem Schritt das zuletzt generierte Element gespeichert werden, um eine Verbindung zwischen den Elementen mithilfe des *BPMN-sequenceflows* herzustellen. Für die jeweils ersten Elemente in jeder scope wird ein Dummy-Element erzeugt, welches beim Erstellen der Verbindungen ignoriert wird, um Fehler zu vermeiden. Dies gilt ebenfalls für die Verbindungen zwischen den scopes an sich. Dieses Element ist notwendig um valide Verbindungen zu erhalten, die auch Sinn ergeben. Sollten in einer scope fault- oder compensationscopes auftreten, werden diese mithilfe von BPMN-Unterprozessen innerhalb der Hauptscope dargestellt. Dabei wird eine Markierung gesetzt die kennzeichnet, dass der nächste Schritt sich innerhalb der entsprechenden scope befindet. Außerdem wird vor Beginn des BPMN-Unterprozesses ein *BPMN-fault-event* oder ein *BPMN-compensation-event* gesetzt, um die Art der scopes zu verdeutlichen. Dadurch ist die Abgrenzung zu den anderen Komponenten der Hauptscope klarer. Beim Erzeugen der BPMN-Elemente aus den Informationen der BPEL-Befehlsblöcke werden zusätzlich zum Namen, welcher auf dem BPMN-Element angezeigt wird, die restlichen Informationen mithilfe der Dokumentation des Elements gespeichert. Diese

[3] https://github.com/bpmn-io/bpmn-js

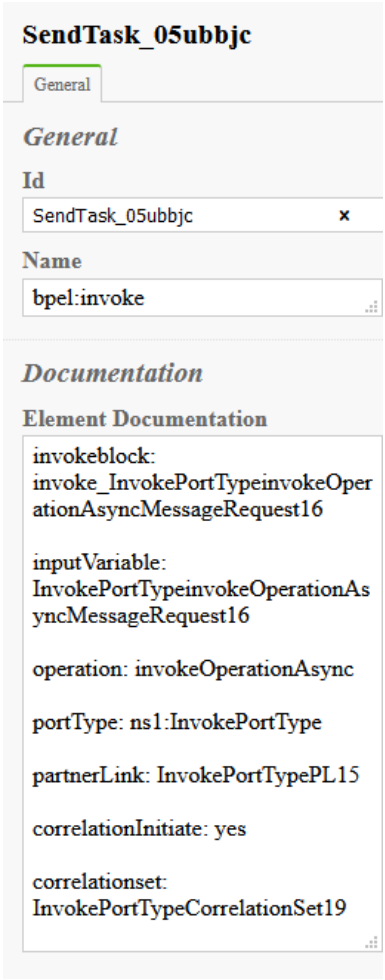

Abbildung 6.1: Darstellung von Informationen mithilfe des *BPMN-properties-panel*

Dokumentation ist im Modeler normalerweise nicht auf den ersten Blick sichtbar. Deshalb wird die *bpmn-js-properties-panel* Erweiterung verwendet. Der Modeler wird um ein anpassbares Panel erweitert, welches zusätzlich zum Namen die ID des BPMN-Elements und die Dokumentation innerhalb des Dokumentationsfelds anzeigt. Das Dokumentationsfeld des Panels wird zum Anzeigen der Informationen verwendet, da dieses genug Platz aufweist, um alles klar darzustellen. Ein Beispiel dafür ist in Abbildung 6.1 zu sehen. Hier werden die Informationen eines *bpel:invoke* Blocks gezeigt. Um dem Nutzer alle Informationen übersichtlich zu präsentieren, ist die Verwendung des Panels notwendig. Durch das Anklicken von BPMN-Elementen wird das Panel, sowie die entsprechenden Informationen des Elements, angezeigt. Dadurch erhält der Benutzer alle relevanten Informationen auf einen Blick. Beim Speichern der Informationen im Dokumentationsfeld muss aber bei einigen Blöcken die Darstellung angepasst werden, da diesem Feld nur ein String übergeben werden kann. Als Beispiel können die TOSCA spezifischen Variablen, welche schon im vorherigen Abschnitt erwähnt wurden, genommen werden. Wird der String nicht angepasst, liegen diese Variablen unleserlich vor und das Panel erfüllt seinen Zweck nicht. Deshalb wird der String vor der Übergabe an das Panel noch einmal je nach Blocktyp formatiert.

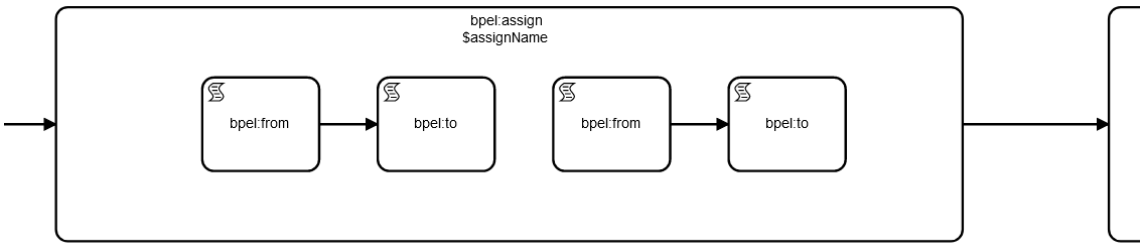

Abbildung 6.2: Visualisierung des *bpel:assign* Falls

Die BPEL-Befehlsblöcke haben beim Darstellen noch zwei Fälle, auf die besonders geachtet werden muss. Der erste Fall ist der in dieser Arbeit schon mehrfach erwähnte *bpel:assign* Block. Dieser wird ähnlich wie fault- und compensationscopes behandelt, da er ebenfalls mithilfe von BPMN-Unterprozessen dargestellt wird. Anders als die fault- und compensationscopes ist dieser aber direkt mit den anderen BPMN-Elementen verbunden. Innerhalb des *bpel:assign* Blocks kommen immer beliebig viele Paare von *bpel:from* und *bpel:to* vor. Diese Paare sind miteinander, aber nicht mit anderen Paaren verbunden. Obwohl keine Verbindungen vorliegen ist die Ablaufreihenfolge trotzdem standardmäßig von links nach rechts, wie bei allen anderen Elementen der Visualisierung. Der zweite Fall ist der *bpel:if* beziehungsweise *bpel:throw* Block. Die If-Bedingung wird mit einem exklusiven BPMN-gateway dargestellt, welches zwischen einem BPMN-Task, der das *throw* darstellt, und dem nächsten Element verzweigt.

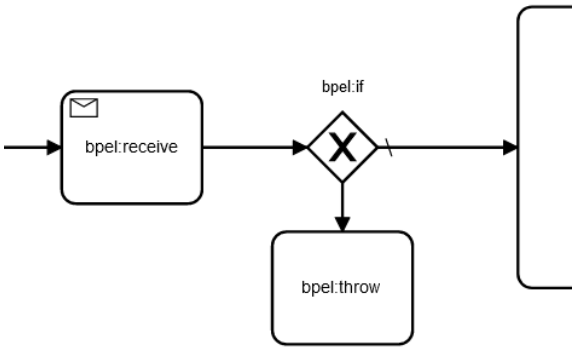

Abbildung 6.3: Visualisierung des *bpel:if* Falls

Wurde durch alle scopes iteriert, ist die Visualisierung abgeschlossen. Das Layout der Visualisierung ist auf eine bestimmte Art aufgebaut. Die scopes werden von oben nach unten nach der Ablaufreihenfolge des Plans dargestellt. Dabei steht die erste scope ganz oben. Die Befehlsblöcke innerhalb der scope werden nach der Ablaufreihenfolge von links nach rechts dargestellt. Enthält die scope im inneren fault- oder compensationscopes, werden diese zu Beginn der scope angezeigt, gekennzeichnet durch entsprechende *BPMN-fault-events* und *BPMN-compensation-events*. Der restliche scope Inhalt folgt danach. Alle vorliegenden Informationen zu bestimmten Elementen können durch das Anklicken dieser mithilfe des Panels erhalten werden.

6.4 Anmerkungen

Bei der Entwicklung dieser Arbeit sind einige Einschränkungen und Limitierungen eingetreten, die erwähnt werden sollten. Da der Schwerpunkt der Arbeit in der Visualisierung von Bereitstellungsplänen liegt, wurde die Implementierung auf diesen beschränkt. Der nächste Schritt der vorgenommen werden muss ist die Integration in das OpenTOSCA System. Dies überschreitet die Einschränkung des Schwerpunktes der Arbeit und bringt weitere Fragen auf, die zur Diskussion offen stehen. Unter anderem beinhalten diese Fragen die Art und Weise der Integration. Der Modeler zur Visualisierung kann zum Beispiel direkt in die OpenTOSCA Winery integriert werden, oder als alleinstehende Anwendung verwendet werden. Eine weitere Limitierung sind die Regeln von BPMN im Modeler. Diese beschränken die durch das Modeling-API erstellbaren Verbindungen und BPMN-Elemente. Da BPMN und BPEL verschiedenen Regeln und Konzepten zugrunde liegen, müssen auf diese bei der Implementierung besonders geachtet werden. Ein wichtiger Punkt bei der Darstellung der Visualisierung ist der Fall, bei dem mehrere scopes parallel ausgeführt werden. In diesem Fall werden diese in der Visualisierung sequentiell dargestellt, wobei die scope, die während dem Ausführen des Algorithmus 6.1 innerhalb des Bereitstellungsplans zuerst vorkommt, als erstes dargestellt wird. Als Anmerkung sollte noch das Layout der Visualisierung genannt werden. Bei dem verwendeten Layout enthalten scopes teilweise leere Bereiche. Dies kommt meist bei fault- und compensationscopes vor. Als Alternative können Layoutplugins wie zum Beispiel *dagre*[4] verwendet werden, wodurch die Visualisierung kompakter wird. Ob die Übersichtlichkeit und Lesbarkeit erhalten bleibt ist dabei eine wichtige Frage. Aus diesem Grund wird standardmäßig kein Layoutplugin in dieser Implementierung verwendet.

[4] https://github.com/dagrejs/dagre

7 Zusammenfassung und Ausblick

Durch die zunehmende Nutzung von Cloud-Applikationen wird das Erstellen und Bereitstellen dieser Anwendungen immer wichtiger. Die Flexibilität solcher Cloud-Applikationen ist eines ihrer Vorteile und kommt in verschiedenen Bereichen zum Einsatz. Oft müssen Anpassungen vorgenommen werden um mit der sich schnell entwickelnden Technologien und neuen Entwicklungen kompatibel zu sein. Hier kommt der TOSCA Standard ins Spiel, der Cloud-Anwendungen plattformunabhängig beschreibt, wodurch diese überall verwendet werden können. Dieser Standard wird von OpenTOSCA, dem Open-Source Ökosystem, welches an der Universität Stuttgart entwickelt wird, verwendet. Cloud-Anwendungen können durch die Verwendung des *Topology Modelers* der Winery im OpenTOSCA System modelliert werden. Anhand dieser Modellierung werden mithilfe des OpenTOSCA Plangenerators Bereitstellungspläne erzeugt. Diese können dann innerhalb des Systems bereitgestellt und instanziiert werden.

Die Nutzung des Plangenerators erleichtert das Erzeugen von Cloud-Anwendungen, aber garantiert nicht deren Ausführbarkeit. Sollte ein Fehler beim Bereitstellen auftreten, ist es erforderlich den Bereitstellungsplan zu durchsuchen, um diesen zu finden. Dies erfordert Zeit und Konzentration, da Bereitstellungspläne in OpenTOSCA BPEL-Dateien sind, die keine Rücksicht auf Benutzerfreundlichkeit nehmen. Diese Dateien enthalten viele verschiedene Operationen und Befehle, was das Suchen nach dem Fehler schwierig macht. Deshalb erfordert die Fehlersuche meist mehr Aufwand als die Erstellung des Bereitstellungsplans an sich.

Das Ziel dieser Arbeit ist die Erleichterung bei der Arbeit mit solchen Bereitstellungsplänen im Sinne von *DevOps*. Es wurde ein Konzept entwickelt, um die Visualisierung von Bereitstellungsplänen zu ermöglichen. Mithilfe dieser Visualisierung soll der Plan übersichtlicher sein, um die Fehlersuche zu vereinfachen. Dabei wird der Bereitstellungsplan durch die Verwendung eines SAX-Parsers geparst, um die wichtigsten Informationen des BPEL-Prozesses herauszufiltern. Es wird ein BPMN-Modeler als Web-Renderer verwendet, um den Plan und die gefilterten Informationen visuell darzustellen. Hierbei wird ein Web-Renderer verwendet, da dieser einfacher zu erweitern und zu integrieren ist als die Alternativen. Diese bestehen aus Graphikgeneratoren, welche in diesem Anwendungsfall nicht so flexibel sind wie Web-Renderer. Im BPMN-Modeler werden BPEL-Befehlsblöcke aus dem Bereitstellungsplan durch passend gewählte BPMN-Elemente dargestellt. Zusätzliche Informationen werden durch das Anklicken der Elemente angezeigt. Eine Implementierung des Konzepts wurde mithilfe des BPMN-Modelers *bpmn-js* von Camunda, sowie dem Sax-Parser *sax-ts* erstellt. Diese verwendet das *properties-panel* Modul des BPMN-Modelers, um die relevanten Informationen aus dem Bereitstellungsplan in einem Panel anzuzeigen.

Ausblick

Das Ziel dieser Arbeit, eine Visualisierung von Bereitstellungsplänen zu erzeugen, wurde weitgehend erfüllt. Der nächste Schritt wäre die Integration in das OpenTOSCA System, wie schon in Kapitel 6.3 erwähnt. Die optimale Art der Integration lässt sich noch ausführlicher diskutieren. Weitere Zukunftsmöglichkeiten wären die Erweiterung des als Web-Renderer verwendeten Modelers. Die Verwendung eines Web-Renderers wurde bewusst gewählt um eine zukünftige Anpassung oder Erweiterung der Implementierung zu erleichtern. So können zukünftig notwendige Informationen beim Parsen hinzugefügt und visualisiert werden. Eine weitere Möglichkeit zum Erleichtern der Benutzung wäre die Funktion, im bereits visualisierten Plan Änderungen an den Informationen zu übernehmen und diese Änderungen direkt in den Bereitstellungsplan zu übertragen. Eine Möglichkeit zum Speichern der Änderungen liegt bereits vor, da bei der Visualisierung des Plans BPMN-Elemente erstellt werden, die ihre normale Funktionalität beibehalten. Würde dies funktionieren, kann man noch einen Schritt weiter gehen und die Bereitstellung des Plans direkt aus der Visualisierung im Renderer starten. Das erfordert aber eine enge Verknüpfung der Visualisierung mit dem Bereitstellungsplan in beide Richtungen. Das bereits in Kapitel 4.4 beschriebene Ziel des automatischen Testens auf Korrektheit des Bereitstellungsplans ist natürlich die beste Lösung für die Identifizierung von Fehlern im Plan. Die Schwierigkeit dieses Vorhabens ist offensichtlich, da die Korrektheit unter anderem abhängig vom Aufbau der verwendeten *TOSCA Templates* ist. Wird dieses Ziel aber erreicht, kann dadurch von der Vereinfachung der Fehlersuche direkt zum Beheben des Fehlers übergegangen werden, da die Fehlersuche komplett wegfällt.

Literaturverzeichnis

[ABD+17] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, D. A. Tamburri. „DevOps: Introducing Infrastructure-as-Code“. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, S. 497–498. DOI: 10.1109/ICSE-C.2017.162 (zitiert auf S. 18).

[BEK+16] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. „The OpenTOSCA Ecosystem - Concepts Tools“. In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016,* INSTICC. SciTePress, 2016, S. 112–130. ISBN: 978-989-758-207-3. DOI: 10.5220/0007903201120130 (zitiert auf S. 13, 14, 20).

[DT09] G. Decker, W. Tscheschner. „Migration von EPK zu BPMN“. In: (Nov. 2009). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.204.4290&rep=rep1&type=pdf#page=91 (zitiert auf S. 27).

[EGHS16] C. Ebert, G. Gallardo, J. Hernantes, N. Serrano. „DevOps“. In: *IEEE Software* 33.3 (2016), S. 94–100. DOI: 10.1109/MS.2016.68 (zitiert auf S. 13, 18).

[HS10] D. J. Hündling, D. Schmiedel. „Prozessautomatisierung am Beispiel: Wie passen BPMN und BPEL zusammen?“ In: (2010). URL: https://www.doag.org/formes/pubfiles/2262953/docs/Konferenz/2010/vortraege/SOA%20/%20BPM/245-2010-K-SOA-Huendling-Prozessautomatisierung.pdf (zitiert auf S. 27).

[JAK16] R. Jabbari, N. Ali, B. T. K. Petersen. „What is DevOps?: A Systematic Mapping Study on Definitions and Practices“. In: *XP '16 Workshops: Scientific Workshop Proceedings of XP2016 Edinburgh Scotland UK* (Mai 2016), Article No. 12, S. 1–11. DOI: 10.1145/2962695.2962707 (zitiert auf S. 18).

[JBM14] N. Jain, A. Bhansali, D. Mehta. „AngularJS: A modern MVC framework in JavaScript“. In: *Journal of Global Research in Computer Science* 5.12 (2014), S. 17–23 (zitiert auf S. 22).

[KBBL12] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications“. In: *Business Process Model and Notation*. Hrsg. von J. Mendling, M. Weidlich. Bd. 125. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2012, S. 38–52. ISBN: 978-3-642 -33154-1. DOI: 10.1007/978-3-642-33155-8_4 (zitiert auf S. 28, 29).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „Winery – A Modeling Tool for TOSCA-Based Cloud Applications“. In: *Service-Oriented Computing*. Hrsg. von S. Basu, C. Pautasso, L. Zhang, X. Fu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 700–704. ISBN: 978-3-642-45005-1. DOI: https://doi.org/10.1007/978-3-642-45005-1_64 (zitiert auf S. 20).

[Mer14] D. Merkel. „Docker: lightweight linux containers for consistent development and deployment“. In: *Linux journal* 2014.239 (2014), S. 2. URL: https://www.seltzer.com/margo/teaching/CS508.19/papers/merkel14.pdf (zitiert auf S. 23).

[MG11] P. Mell, T. Grance. „The NIST Definition of Cloud Computing“. In: *National Institute of Standards and Technology* (Sep. 2011). DOI: 10.6028/NIST.SP.800-145 (zitiert auf S. 17).

[Mic15] T. Michelbach. „Ein Modellierungswerkzeug für BPMN4TOSCA“. Diplomarbeit. Universität Stuttgart, 2015. DOI: http://dx.doi.org/10.18419/opus-3497 (zitiert auf S. 29).

[MN06] J. Mendling, M. Nüttgens. „EPC markup language (EPML): An XML-based interchange format for event-driven process chains (EPC)“. In: *Inf. Syst. E-Business Management* 4 (Juli 2006), S. 245–263. DOI: 10.1007/s10257-005-0026-1 (zitiert auf S. 26).

[MZ05] J. Mendling, J. Ziemann. „Epk-visualisierung von bpel4ws prozessdefinitionen“. In: (Jan. 2005). URL: https://www.researchgate.net/publication/228408533_Epk-visualisierung_von_bpel4ws_prozessdefinitionen (zitiert auf S. 25, 26).

[NR02] M. Nüttgens, F. J. Rump. „Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK)“. In: *Promise 2002 – Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen*. Hrsg. von J. Desel, M. Weske. Bonn: Gesellschaft für Informatik e.V., 2002, S. 64–77 (zitiert auf S. 26).

[OAS07] OASIS. *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*. Organization for the Advancement of Structured Information Standards (OASIS). 2007. URL: https://www.oasis-open.org/committees/download.php/23974/wsbpel-v2.0-primer.pdf (zitiert auf S. 14, 21).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Nov. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (zitiert auf S. 13, 18, 19).

[OMG11] OMG. *Business Process Model and Notation (BPMN) Version 2.0*. Object Management Group (OMG). 2011. URL: https://www.omg.org/spec/BPMN/2.0/PDF (zitiert auf S. 21, 27).

Alle URLs wurden zuletzt am 25. 02. 2021 geprüft.

Erklärung

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

Ort, Datum, Unterschrift