Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Diplomarbeit Nr. 3473

Dynamische Provisionierung von Web Services für Simulationsworkflows

Valeri Schneider

Studiengang: Informatik

Prüferin: Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
Betreuerin: Dipl.-Inf., Dipl.-Wirt.Ing.(FH) Karolina Vukojevic

begonnen am: 13.03.2013
beendet am: 12.09.2013

CR-Klassifikation: C.2.4, D.2.11, H.4.1, H.3.5, H.5.3, I.6.7, J.2

Inhaltsverzeichnis

Abbildungsverzeichnis .... **III**

Listingverzeichnis .... **IV**

Tabellenverzeichnis .... **V**

Abkürzungsverzeichnis .... **VI**

1 Einleitung **1**

1.1 Aufgabenstellung 3

1.2 Gliederung der Arbeit 4

2 Hintergrund **5**

2.1 SimTech sWfMS 5

2.2 Infrastruktur hinter Cloud-Services 6

2.3 TOSCA 7

2.4 OpenTOSCA 9

2.5 Enterprise Service Bus 9

2.6 Verwandte Arbeiten 10

3 Vorgehensweise für Erweiterungen von WfMS **12**

3.1 Ansatz I (Externer Service) 13

3.2 Ansatz II (BPEL-Erweiterung) 15

3.3 Ansatz III (Sonstige Infrastruktur-Komponenten) 18

4 Architektur für On-Demand Provisionierung **20**

4.1 Architektur-Überblick 20

4.2 Simulationsservices 23

4.3 Service Repository 25

4.4 Service Registry 25

4.5 Provisioning Engine 26

4.6 Enterprise Service Bus 27

5 Erweiterung der Service-Provisionierung **29**

5.1 Provisioning Manager 31

5.2 Provisionierungstrategien 33

5.3 Visualisierung des Provisionierungsprozesses 34

6 Realisierung **36**

6.1 SimTech sWfMS....................36
6.1.1 ODE-PGF....................37
6.1.2 Fragmento....................41
6.1.3 Opal....................42
6.1.4 Auditing Application....................43
6.2 Erweiterung vom SimTech sWfMS....................45
6.3 Service Repository....................46
6.4 Service Registry....................47
6.5 Enterprise Service Bus....................47
6.5.1 Prozess für Service-Provisionierung....................48
6.6 ODE-PGF....................49
6.7 Provisioning Manager....................51
6.7.1 Adapter für MockServiceRepository....................53
6.7.2 Adapter für OpenTosca....................54
6.7.3 OpenTOSCA ContainerAPI....................57

7 Zusammenfassung und Ausblick....................69

7.1 Ausblick....................69

Literaturverzeichnis....................71

Abbildungsverzeichnis

Abbildung 1-1: Cloud-basierte Infrastruktur eines sWfMS (schematisch) 2
Abbildung 2-1: Architektur eines sWfMS [4] 5
Abbildung 2-2: Infrastruktur hinter Cloud-Services 6
Abbildung 2-3: TOSCA Service Template [12] 7
Abbildung 2-4: Struktur einer CSAR-Datei [12] 8
Abbildung 2-5: OpenTOSCA Architektur (vereinfacht) 9
Abbildung 2-6: Integration über ESB (schematisch) 10
Abbildung 2-7: Architektur für automatisierte Provisionierung von Cloud-Services [23] 11
Abbildung 3-1: Architektur eines sWfMS (vereinfacht) 12
Abbildung 3-2: Ansatz I – Externer Service 13
Abbildung 3-3: Ansatz I - Explizite Modellierung 14
Abbildung 3-4: A1 – Erweiterung von BPEL + Process Engine [24] 16
Abbildung 3-5: Architektur von SWoM [24] 17
Abbildung 3-6: A2 - Erweiterung von BPEL + Transformation [24] 18
Abbildung 3-7: Ansatz III – Sonstige Infrastruktur-Komponenten 19
Abbildung 4-1: Architektur für On-Demand Provisionierung [11] 20
Abbildung 4-2: Refactoring von Legacy Simulationsanwednungen 23
Abbildung 4-3: Arten von Simulationsservices 24
Abbildung 4-4: Prozess für die Service-Provisionierung [11] 27
Abbildung 5-1: Architektur für die Provisionierung von Simulationsservices [11] (vereinfacht) 29
Abbildung 5-2: Mögliche Problembereiche der Architektur für Service-Provisionierung 30
Abbildung 5-3: Provisioning Manager (externer Service) 31
Abbildung 5-4: Provisioning Manager (Erweiterter ESB) 32
Abbildung 5-5: Realisierung von Provisionierungsstrategien 34
Abbildung 6-1: Architektur vom SimTech Prototyp 36
Abbildung 6-2: ODE Process Management View 38
Abbildung 6-3: Integration von ODE-PGF (Web Services) 38
Abbildung 6-4: SimTech BPEL Designer (Steuerung) 39
Abbildung 6-5: SimTech BPEL Designer (Überwachung) 39
Abbildung 6-6: SimTech BPEL Designer (Breakpoints) 40
Abbildung 6-7: Integration von ODE-PGF (Messaging) 41

Abbildung 6-8: Integration von Fragmento 42
Abbildung 6-9: Integration von Opal 43
Abbildung 6-10: Integration von Auditing Application (ODE-PGF) 43
Abbildung 6-11: Integration von Auditing Application (BPEL Designer) 44
Abbildung 6-12: Erweiterung des SimTech sWfMS 45
Abbildung 6-13: Prozess für die Service-Provisionierung [11]...........................48
Abbildung 6-14: Architektur vom Provisioning Manager..................................51
Abbildung 6-15: Ablauf vom Service Deployment..52
Abbildung 6-16: Ablauf vom Service Deployment (direkter Download)..........53
Abbildung 6-17: Adapter für MockServiceRepository 53
Abbildung 6-18: Adapter für OpenTosca 54
Abbildung 6-19: Ablauf von deployService()-Methode 56

Listingverzeichnis

Listing 3-1: Beispiel für BPEL-Erweiterung 15
Listing 6-1: Download einer CSAR-Datei - HTTP-Request 46
Listing 6-2: Beispiel für TenantContext (vollständig) [44] 48
Listing 6-3: Beispiel für ODE-PGF Endpunkt-Konfiguration..........................50
Listing 6-4: Upload CSAR (lokal) - HTTP-Request 57
Listing 6-5: Upload CSAR (Web Formular) – HTTP-Request 58
Listing 6-6: Upload CSAR (URL) – HTTP-Request 58
Listing 6-7: Upload CSAR (lokal) - HTTP-Response...59
Listing 6-8: Zustand vom Deployment-Prozess – HTTP-Request 60
Listing 6-9: Zustand vom Deployment-Prozess – HTTP-Response 60
Listing 6-10: TOSCA-Verarbeitung - HTTP-Request...61
Listing 6-11: Service Templates abfragen – HTTP-Request 62
Listing 6-12: Service Templates abfragen – HTTP-Response 62
Listing 6-13: Implementation Artifacts Deployment - HTTP-Request............63
Listing 6-14: Plan Deployment - HTTP-Request...64
Listing 6-15: Build-Pläne auflisten - HTTP-Request 65
Listing 6-16: Plan Ausführung - HTTP-Request..66
Listing 6-17: Aktive Pläne ermitteln - HTTP-Request 67
Listing 6-18: Service Instanzen ermitteln - HTTP-Request................................68

Tabellenverzeichnis

Tabelle 6-1: Farbliche Zuordnung der Aktivitätszustände....................................40
Tabelle 6-2: Endpunkt-Eigenschaften (Auszug) [46]..50
Tabelle 6-3: Zustände vom Deployment Prozess 61

Abkürzungsverzeichnis

WfMS	Workflow Management System
IAAS	Institut für Architektur von Anwendungssystemen
TOSCA	OASIS Topology and Orchestration Specification for Cloud Applications
ESB	Enterprise Service Bus
BPEL	Business Process Execution Language
Mayflower	Model-as-you-go Workflow Developer
sWfMS	Scientific Workflow Management System
CSAR	Cloud Service Archive
EAI	Enterprise Application Integration

1 Einleitung

Seit längerer Zeit haben die Workflow-Technologien [1] den Einzug in die Geschäftswelt gehalten und sich inzwischen etabliert. Dabei werden sie von Unternehmen dazu benutzt, um eigene Geschäftsprozesse zu modellieren und automatisiert ablaufen zu lassen. In der Wissenschaft werden die Workflow-Technologien noch nicht so flächendeckend eingesetzt wie in der Geschäftswelt. Ein Grund dafür ist sicherlich eine Reihe von spezifischen Anforderungen, die von der Wissenschaft kommen und durch Geschäftsworkflows nicht ohne weiteres abgebildet werden können. Für wissenschaftliche Zwecke existiert eine Reihe von speziell dafür entwickelter Workflowsprachen wie Taverna [2] oder Triana [3], die nicht auf den Workflow-Technologien für Geschäftsprozesse [1] basieren. Für die Geschäftsworkflows gibt es jedoch zahlreiche Standards und Werkzeuge, daher kann sich deren Einsatz für wissenschaftliche Zwecke als vorteilhaft erweisen [4].

SimTech

Der Excellenzcluster SimTech [5] an der Universität Stuttgart betreibt Forschung im Bereich der Simulationstechnologien. Durch die enge Zusammenarbeit von Wissenschaftlern aus unterschiedlichen Fachgebieten können komplexe ganzheitliche Simulationen innovativ und effizient erforscht werden. Neben der angestrebten Grundlagenforschung der Simulationstechnologie werden fünf folgende Anwendungsgebiete (SimTech Visionen) [6] intensiv erforscht: Computergestütztes Materialdesign, Virtuelles Prototyping, Umwelttechnik, Systembiologie und Menschenmodell.

Das Institut für Architektur von Anwendungssystemen der Universität Stuttgart (IAAS) [7] befasst sich in SimTech mit der Weiterentwicklung der klassischen Workflow-Technologie, damit sie für die Modellierung, Ausführung, Überwachung und Analyse von wissenschaftlichen Simulationen eingesetzt werden kann. Eine Architektur eines Scientific Workflow Management Systems (sWfMS) [4] wurde entworfen, das den Anforderungen und Bedürfnissen von Wissenschaftlern gerecht wird.

Cloud-basierte Infrastruktur für Simulationsworflows

Der Fokus aktueller Forschung am IAAS liegt auf der Entwicklung einer Architektur für On-Demand Provisionierung von Workflow-Ausführungsumgebung und Services für Simulationsworkflows in einer Cloud-Umgebung [8]. Dadurch soll folgendes erreicht werden:

Ein Wissenschaftler ist nicht zwingend an die eigene Infrastruktur gebunden, sondern kann sich die benötigten bzw. fehlenden Ressourcen von einem Cloud-Provider für die Dauer einer Simulation bereitstellen lassen.

Die Cloud-Infrastruktur sollte gemeinsam genutzt werden können, d.h. einzelne Komponenten in der Infrastruktur werden von einem Cloud-Provider nicht jedes Mal neu und exklusiv bereitgestellt, sondern können von mehreren Benutzern gleichzeitig verwendet werden. Die Voraussetzung dafür sind aller-

dings multi-tenant-fähige Komponenten, wie z.B. ein multi-tenant-fähiger Enterprise Service Bus [9]

Die Kollaboration der Wissenschaftler soll auf ein höheres Niveau gehoben werden, indem nicht nur die Infrastruktur gemeinsam genutzt wird, sondern auch durch die Wiederverwendung und gemeinsame Eintwicklung von komplexen wissenschaftlichen Simulationen. Die Ergebnisse einer Simulation lassen sich dann leichter nachvollziehen und von anderen Wissenschaftlern reproduzieren und bewerten.

Community-getriebene Entwicklung, die man von vielen Open-Source Software Projekten kennt, soll durch die cloud-basierte Infrastruktur für Simulationsworkflows ermöglicht werden.

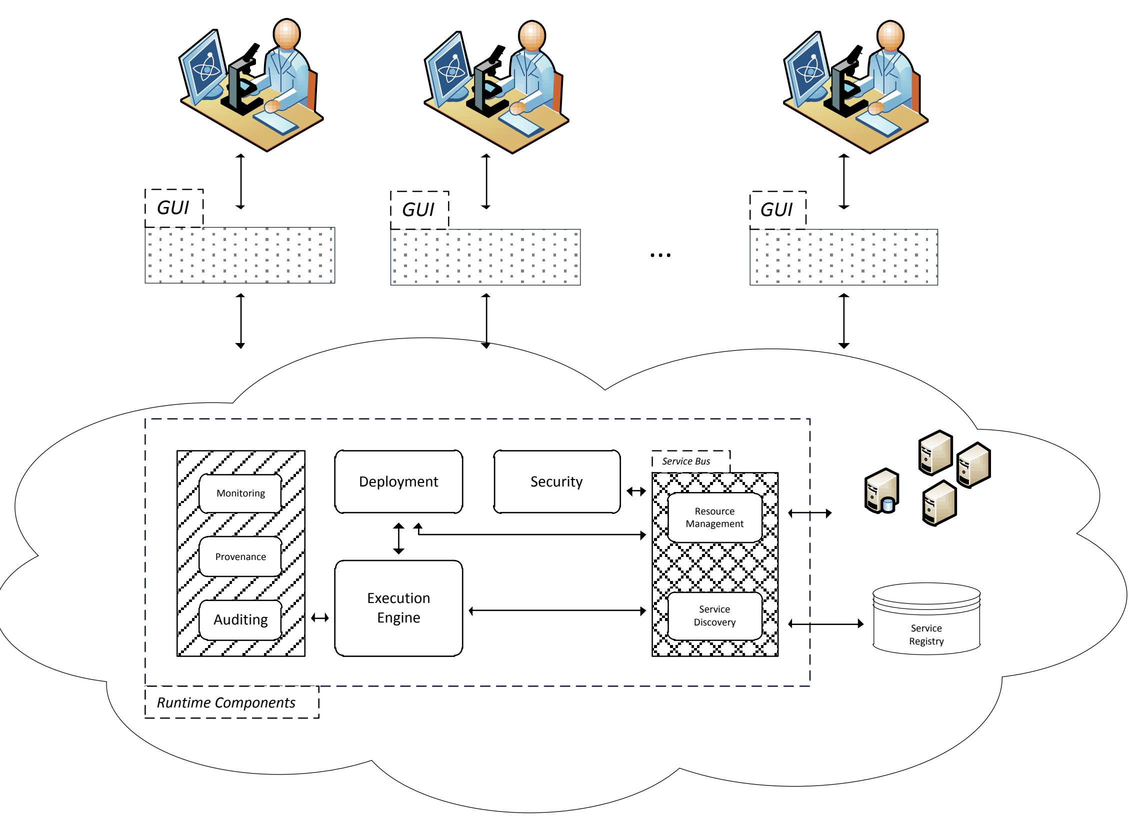

Abbildung 1-1: Cloud-basierte Infrastruktur eines sWfMS (schematisch)

Die cloud-basierte Infrastruktur (schematisch dargestellt in der Abbildung 1-1) wird in mehreren Schritten aufgebaut. Zum einen wurden die vorhandenen Komponenten vom SimTech sWfMS untersucht, inwieweit sie in eine Cloud-Umgebung migriert werden können [10]. Zum anderen wurde eine generische Architektur entwickelt, die die On-Demand Provisionierung von Workflowausführungsumgebung und Services ermöglicht [11]. Dadurch soll die benötigte Instrastruktur bei Bedarf komplett automatisiert in einer Cloud-Umgebung provisioniert und direkt für eine Simulations -Modellierung bzw. -Ausführung verwendet werden können.

Betrachtet man den Zeitpunkt, wann eine bestimmte Komponente während der Entwicklung einer Simulation tatsächlich benötigt wird, kann man folgendes erkennen: Komponenten wie Execution Engine oder Enterprise Service Bus müssen spätestens beim Starten einer Simulation verfügbar sein, damit ein Simulationsprozessmodell deployt, instanziiert und ausgeführt werden kann. Der Ressourcenverbrauch bleibt dabei relativ konstant für die Dauer einer Simulation und ist unabhängig von den Inhalten der tatsächlichen Simulation.

Simulationsservices

Im Gegensatz dazu sieht es bei den Simulationsservices, die die eigentlichen Simulationsschritte darstellen, ganz anders aus. Die Dauer dieser Schritte ist sehr stark von der konkreten Simulationslogik abhängig, es können also kurzläufge sowie langläufige Schritte sein. Auch der Ressourcenverbrauch kann stark variieren. Benötigt wird ein bestimmter Simulationsservice tatsächlich erst dann, wenn während der Simulationsausführung eine Aktivität an der Reihe ist, die diesen Web Service aufruft. Daher ist die Möglichkeit, die benötigten Simulationsservices bei Bedarf dynamisch zu provisionieren, von großer praktischer Bedeutung, denn so verbrauchen die Simulationen nur die Ressourcen, die sie auch wirklich benötigen.

Diese Arbeit beschäftigt sich mit der dynamischen Provisionierung von Simulationsservices in einer Cloud-Umgebung.

1.1 Aufgabenstellung

In dieser Arbeit soll die On-Demand Provisionierung von Simulationsservices zur Laufzeit eines Simulationsworkflows behandelt werden. Dabei sollen die benötigten Simulationsservices mit der darunterliegenden Infrastruktur in einer Cloud Umgebung dynamisch bereitgestellt werden. Die Provisionierung eines Simulationsservices soll erst dann erfolgen, wenn dieser Service tatsächlich während einer Simulationsausführung benötigt wird. Analog dazu soll die erzeugte Service-Instanz beendet und die dadurch allokierten Ressourcen freigegeben werden, wenn sie nicht mehr von der Simulation benötigt werden.

Am IAAS wurde eine Architektur entwickelt, die On-Demand Provisionierung von Workflow-Ausführungsumgebung und Services zur Laufzeit eines Simulationsworkflows in einer Cloud-Umgebung ermöglicht. Diese Architektur soll als Basis für diese Diplomarbeit dienen und bei Bedarf erweitert werden, damit der beschriebene Anwendungfall effektiv unterstützt werden kann.

Die entworfene Architektur soll prototypisch implementiert und in das bereits vorhandene SimTech sWfMS integriert werden. Zusätzlich sind bei der Implementierung folgende Vorgaben zu beachten: Die Realisierung der Beispiel-Simulationsservices soll mit TOSCA [12] erfolgen und als Service Bus ist der am IAAS entwickelte multi-tenant-fähige Service Bus ESB^{MT} [9] einzusetzen. Als Provisioning Engine soll OpenTOSCA [13] verwendet werden.

1.2 Gliederung der Arbeit

Die Arbeit ist wie folgt aufgebaut. Das *Kapitel 2* gibt einen Überblick über Technologien, Konzepte, Zusammenhänge und verwandten Arbeiten, die für diese Arbeit von Bedeutung sind. Das *Kapitel 3* beleuchtet an einem Beispiel verschiedene Möglichkeiten ein WfMS zu erweitern und diskutiert ihre Vor- und Nachteile. Im *Kapitel 4* wird der am IAAS entworfene Architektur-Ansatz für die On-Demand Provisionierung von Workflow-Ausführungsumgebung und Services für wissenschaftliche Workflows erläutert. Anschließend wird im *Kapitel 5* eine Erweiterung der vorgestellten Architektur im Bezug auf die Provisionierung von Simulationsservices beschrieben. Das *Kapitel 6* betrachtet ausgewählte Aspekte der Realisierung der Architektur. Schließlich werden im *Kapitel 7* die Ergebnisse der Arbeit zusammenfassend dargelegt und die Ideen für weitere nachfolgende Arbeiten erläutert.

2 Hintergrund

In diesem Kapitel wird eine Wissensbasis über eine Reihe von involvierten Technologien, Konzepten sowie Software-Produkten vermittelt. Die Beschreibungen von Technologien oder Produkten erheben keinen Anspruch auf Vollständigkeit, viel mehr werden die relevanten Aspekte hervorgehoben, die für das Verständnis dieser Diplomarbeit von Vorteil sind.

Im Abschnitt 2.1 wird die Architektur eines Scientific Workflow Management Systems vorgestellt. Der Abschnitt 2.2 erläutert die Infrastruktur hinter einem Cloud-Service. Konzepte von TOSCA - einer Beschreibungssprache für portable Cloud-Anwendungen, werden im Abschnitt 2.3 beschrieben. Im Abschnitt 2.4 wird mit OpenTOSCA eine Laufzeitumgebung für TOSCA vorgestellt. Der Abschnitt 2.5 erklärt den Enterprise Service Bus als Integrationsansatz. Schließlich folgt die Vorstellung von ausgewählten verwandten Arbeiten im Bereich der Service-Provisionierung im Abschnitt 2.6.

2.1 SimTech sWfMS

Am IAAS wurde eine Architektur eines Scientific Workflow Management Systems (sWfMS) [4] entworfen, das den Anforderungen und Bedürfnissen von Wissenschaftlern gerecht wird. Ein besonderes Augenmerk wurde dabei auf die iterative Vorgehensweise bei der Entwicklung von Simulationen gelegt und die damit verbundenen häufigen Anpassungen vom Simulationsmodell zur Laufzeit. Bei den Simulationen werden in der Regel Berechnungen auf großen Datenmengen durchgeführt, daher steht bei der Modellierung der Datenfluss im Mittelpunkt im Gegensatz zu Geschäftsworkflows, die kontrollflussorientiert sind. Die Abbildung 2-1 gezeigt die entworfene Architektur.

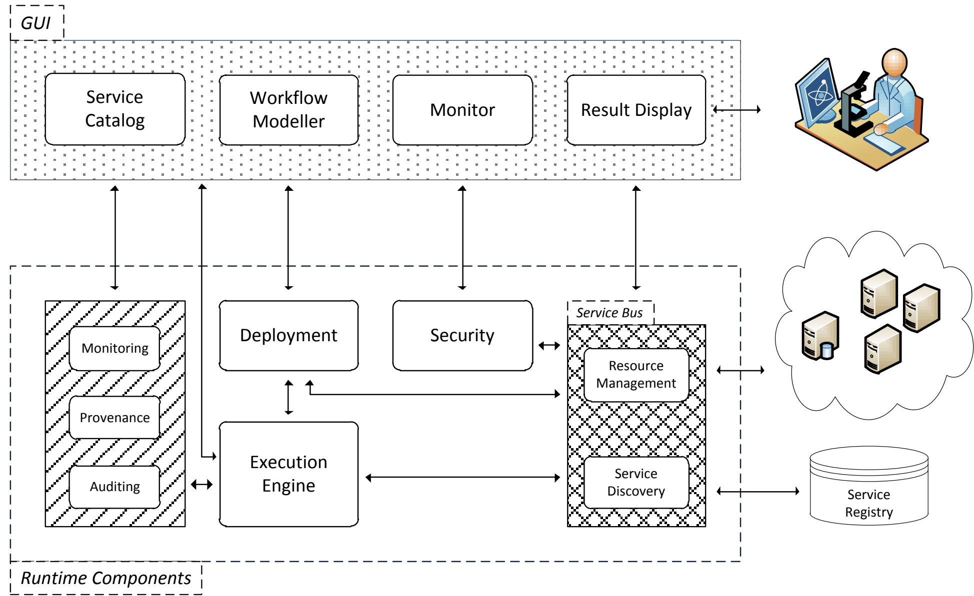

Abbildung 2-1: Architektur eines sWfMS [4]

Im Wesentlichen kann man hieraus drei Komponenten hervorheben: Ein mächtiges GUI soll vor dem Benutzer die Komplexität eines sWfMS verbergen und den Wissenschaftler in jeder Phase der Simulation unterstützen. Simulationsservices kapseln die Logik einzelner Simulationsschritte. Sie werden als als Webservices [14] zur Verfügung gestellt. Und schließlich die Execution Engine ermöglicht die Zusammensetzung einzelner Simulationsschritte in eine sinnvolle Simulation.

Die Architektur wurde prototypisch implementiert und in einer Reihe von studentischen Arbeiten evaluiert. Dabei wurden unter anderem Simulationen in folgenden Bereichen Festkörpersimulation [15], Verhalten von Zellkomponenten in biologischen Netzwerken [16] und Proteinmodellierung [17] realisiert.

2.2 Infrastruktur hinter Cloud-Services

In dieser Arbeit geht es um die dynamische Provisionierung von Simulationsservices in einer Cloud-Umgebung. Dabei soll nicht nur die Service-Anwendung allein, sondern auch die gesamte darunterliegende Infrastruktur ebenfalls in einer Cloud-Umgebung bereitgestellt werden. In der Abbildung 2-2 werden beispielhaft die typischen Infrastruktur-Komponenten dargestellt, die ein Service in einer Cloud-Umgebung benötigt.

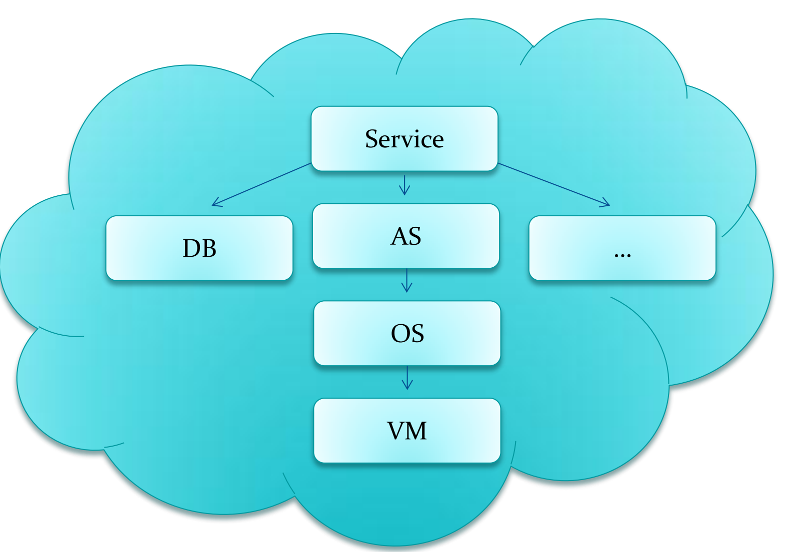

Abbildung 2-2: Infrastruktur hinter Cloud-Services

Als erstes wird ein virtueller Server benötigt, der hinsichtlich seiner Konfiguration an die Anforderungen des Services angepasst sein soll. D.h. die Eigenschaften vom virtuellen Rechner bezüglich CPU, Arbeitsspeicher und Speicher passen müssen. Auf dem virtuellen Server wird ein bestimmtes Betriebssystem gebraucht, das eventuell noch angepasst werden muss um das Zusammenspiel der Software-Komponenten zu optimieren, z.B. durch die Nachinstallation von bestimmten Updates, Patches oder Libraries.

Wenn das Betriebssystem funktionsfähig und passend eingerichtet ist, müssen nun zusätzliche Software-Komponenten, die ein Service benötigt installiert werden, z.B. eine Datenbank oder ein Application Server als eine Laufzeitumgebung für einen Service. Je komplexer die Cloud-Anwendung ist, desto mehr zusätzliche Komponenten und Einstellungen müssen gemacht werden. Wenn ein Service ausschließlich Standard-Software-Produkte nutzt, kann der Aufwand für die Installation und Einrichtung evtl. verringert werden, wenn die benötigten Software-Produkte in der Installation des Betriebssystems integriert sind. Falls man aber z.B. einen modifizierten Application Server benötigt, so wird er separat installiert werden müssen.

2.3 TOSCA

Mit *Topology and Orchestration Specification for Cloud Applications* (TOSCA) hat OASIS [18] im März 2013 eine Spezifikation in der Version 1.0 [12] veröffentlicht und damit eine XML-basierte Beschreibungssprache vorgestellt um Cloud-Anwendungen unabhängig von einem Cloud-Provider zu beschreiben. Die Spezifikation wurde unter Teilnahme von vielen namhaften Unternehmen wie IBM, Google Inc., Hewlett-Packard, Cisco Systems oder SAP AG entworfen, dadurch soll der Akzeptanz von TOSCA durch der Industrie nichts im Wege stehen. Dieser Abschnitt beschreibt Konzepte sowie Bestandteile von TOSCA und orientiert sich stark an der Spezifikation der Version 1.0 [12].

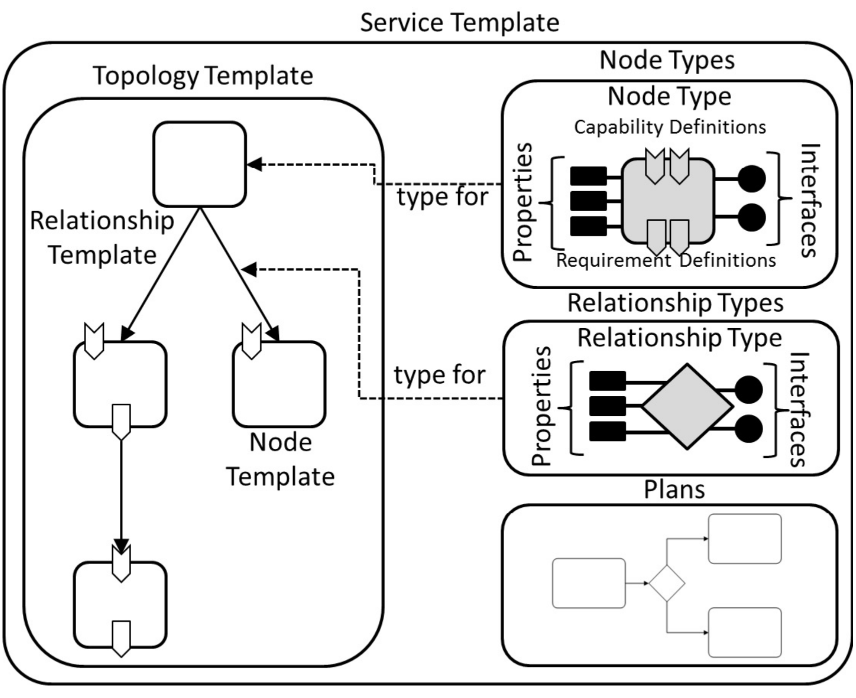

Abbildung 2-3: TOSCA Service Template [12]

In TOSCA werden Cloud-Services durch Service Templates beschrieben. Die Abbildung 2-3 zeigt Bestandteile eines Service Templates, das aus einem Topologie Template und einer Menge von Plänen besteht. Über ein Topologie Template wird die Struktur eines Services beschrieben. Durch Pläne lassen sich die Services verwalten, insbesondere auch Starten und Beenden einer Service-Instanz. Ein Topologie Template besteht aus einer Menge Node Templates, die Komponenten eines Services representieren, und einer Menge von Relationship Templates. Relationship Templates verbinden die Node Templates und bilden damit eine Topologie. Node Templates und Relationship Templates werden zwecks Wiederverwendung durch Node Types und Relationship Types definiert. Ein Node Type wäre dann z.B. “Application Server” oder “DB”. Als Beispiele für ein Relationship Type kann man „hosted on“ und „connects to“ nennen.

Pläne beschreiben Management-Aspekte eines Services. TOSCA definiert keine neue Sprache für die Definition von Plänen, sondern erlaubt den Einsatz beliebiger Sprachen für die Beschreibung von Prozessmodellen, z.B. BPEL oder BPMN. Durch Ausführung von Plänen können neue Service-Instanzen erzeugt oder bestehende beendet werden. Auch andere Management-Aspekte werden dadurch abgebildet. Management-Operationen, die aus einem Prozessmodell heraus aufgerufen werden sollen, können in Form von Implementation Artifacts bereitgestellt werden. Die Implementation Artifacts müssen in einer Management-Umgebung (TOSCA Container) deployt werden, damit die Operationen, die sie realisieren, aufgerufen werden können.

TOSCA definiert das Format einer Cloud Service Archive Datei (CSAR), die alle benötigten TOSCA-Dokumente, Pläne und Artifacts enthält. Die Struktur einer CSAR-Datei zeigt die Abbildung 2-4. Dabei sind die Ordner TOSCA-Metadata und Definitions obligatorisch und enthalten entsprechend eine CSAR-Manifest-Datei und TOSCA-Definitionsdateien.

Abbildung 2-4: Struktur einer CSAR-Datei [12]

2.4 OpenTOSCA

OpenTOSCA ist eine Referenz-Implementierung vom TOSCA-Container. Diese Laufzeitumgebung für TOSCA wurde am IAAS im Rahmen eines Studienprojekts und mehrerer studentischen Arbeiten entwickelt. OpenTOSCA ist eine webbasierte Anwendung, die mit Java realisiert wurde. Mit OpenTOSCA können CSAR-Dateien deployte werden um Service-Instanzen in einer Cloud-Umgebung zu erstellen.

Nachfolgend zeigt die Abbildung 2-5 eine vereinfachte Architektur von Open-TOSCA. Die Interaktion vom Container mit der Außewelt erfolgt über die ContainerAPI. Eine Service-Instanz in einer Cloud-Umgebung kann mit OpenTOSCA durch folgenden Ablauf erstellt werden:

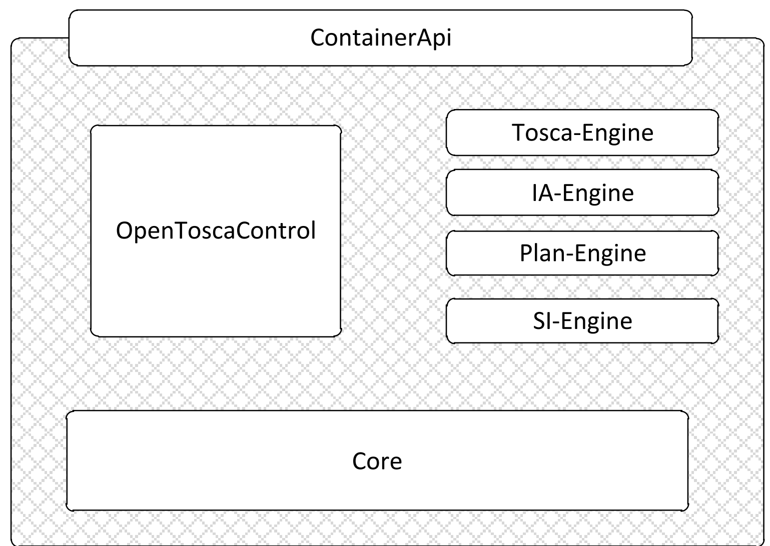

Abbildung 2-5: OpenTOSCA Architektur (vereinfacht)

(1) Die CSAR-Datei wird als erstes zum Container übertragen. (2) Im nächsten Schritt werden die TOSCA-Dokumente verarbeitet durch die Tosca-Engine. (3) Danach müssen Implementation Artifacts deployt werden mit der IA-Engine. (4) Nun folgt das Deployment von Plänen mit Hilfe der Plan-Engine. (5) Anschließend kann ein Build-Plan über die SI-Engine ausgeführt werden und eine neue Service-Instanz wird dadurch erzeugt.

2.5 Enterprise Service Bus

Enterprise Service Bus (ESB) ist nach der Definition von Chappell [19] eine auf Standards basierende Integrationsplattform, welche Web Services, Nachrichtenaustausch, Datentransformation und intelligentes Routing kombiniert um verteilte Anwendungen verlässlich miteinander zu verbinden.

Der ESB als ein Integrationsansatz entstand aus den IT-Bedürfnissen nach einer Integrationsplattform, die Vorteile der bestehenden Ansätze zur Enterprise Application Integration (EAI) wie z.B. vom Hub-and-Spoke-Ansatz besitzt und gleichzeitig die Nachteile, wie limitierte Skalierbarkeit vermeidet. Die Abbildung 2-6 zeigt schematisch den ESB-Integrationsansatz.

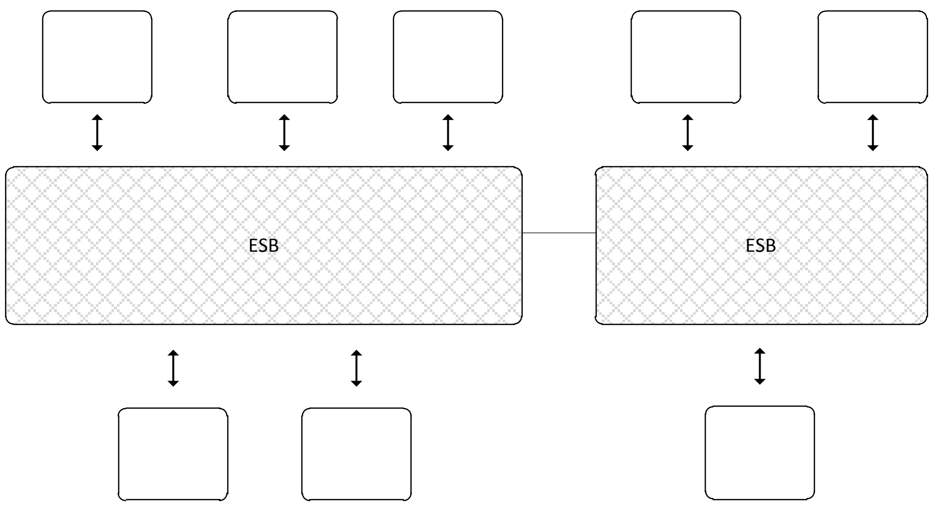

Abbildung 2-6: Integration über ESB (schematisch)

2.6 Verwandte Arbeiten

Die vorgestellten verwandten Arbeiten können in zwei Gruppen unterteilt werden. Einerseits sind es Arbeiten, die sich mit der Provisionierung von Services in verteilten Umgebungen beschäftigen ohne jedoch einen expliziten Bezug zu Cloud-Services zu haben. Daher behandeln sie die Aspekte der Provisionierung von Services in einer Cloud-Umgebung nicht. PUPPET [20] und CHEF [21] sind Lösungen für verteilte automatisierte Installation und Konfiguration von Software. Der Installations- bzw. Konfigurationsprozess wird durch eine Server-Komponente gesteuert, die über eigene Clients auf den verteilten Rechner agiert.

Die zweite Gruppe bilden Arbeiten, die sich explizit mit Ansätzen der Cloud-Provisionierung befassen. In [22] wird On-Demand Provisionierung von Ressourcen für BPEL-Workflows behandelt. Bei dem verwendeten Ansatz wurde die Open-Source BPEL-Engine ActiveBPEL erweitert um dynamische Lastverteilung zu realisieren. Bei einer großen Auslastung werden im Amazon EC2 dynamisch zusätzliche virtuelle Maschinen gestartet und die benötigten Middleware-Komponenten deployt. Die Provisionierung erfolgt durch eine Provisioner-Komponente, die die vom Cloud-Provider bereitgestellten Schnittstellen nutzt.

In [23] wird eine Architektur für die automatisierte Provisionierung von Services in einer Cloud Umgebung vorgestellt. Dabei deckt der vorgestellte Architekturansatz das ganze Spektrum an Aufgaben, die für die Provisionierung von Cloud-Services durchgeführt werden müssen, wie z.B. Starten und Stoppen von virtuellen Maschinen oder Installation, Konfiguration und Überwachung der Software auf den virtuellen Maschinen. Die Abbildung 2-7 zeigt den Architekturansatz.

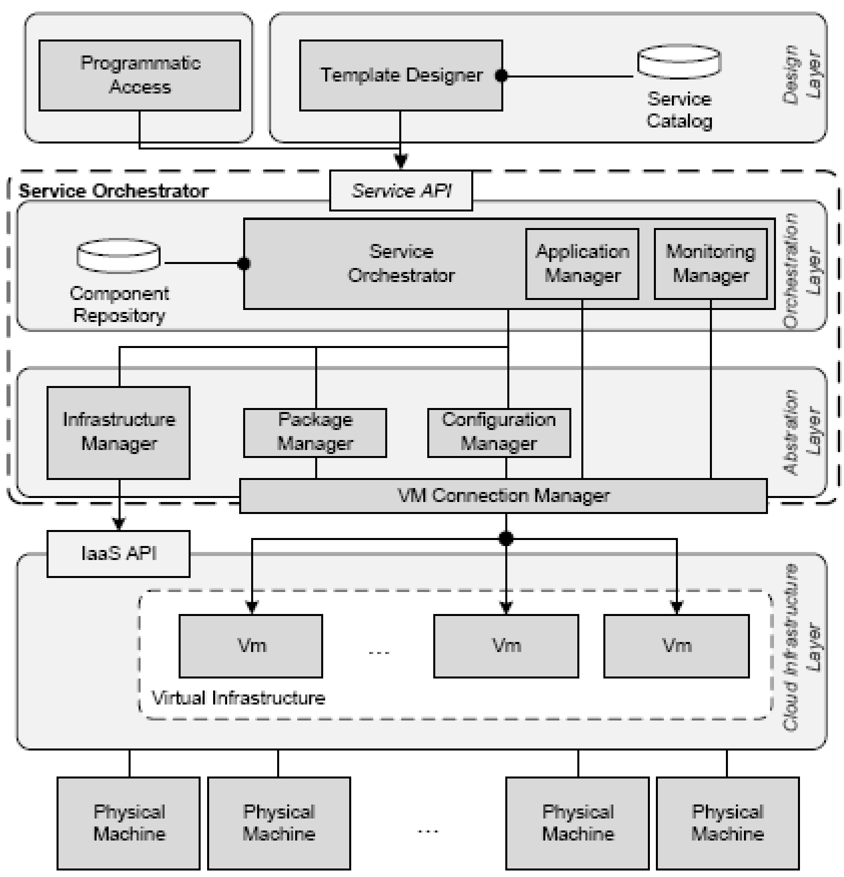

Abbildung 2-7: Architektur für automatisierte Provisionierung von Cloud-Services [23]

Wie man aus der Abbildung sehen kann, werden für die Provisionierung von Services die Schnittstellen eines Cloud-Providers für das Starten oder Beenden von virtuellen Maschinen (IaasS API) benutzt. Danach erfolgt den Zugriff auf die virtuellen Maschinen über den VM Connection Manager, der unterschiedliche Zugriffmechanismen wie z.B. SSH unterstützt. Der Service Orchestrator ist für das Erstellen von Services zuständig. Er sorgt dafür, dass alle Schritte, die für die Erstellung eines Services notwendig sind, in richtiger Reihenfolge ausgeführt werden.

3 Vorgehensweise für Erweiterungen von WfMS

Dieses Kapitel stellt unterschiedliche Ansätze für die Erweiterung eines WfMS um eine neue Funktionalität vor und erörtert deren Vor- und Nachteile. Die Vorgehensweise wird an einem Beispiel verdeutlicht, ist jedoch allgemeiner Natur und kann dann für jede Art von Erweiterungen eines WfMS herangezogen werden.

Die Abbildung 3-1 zeigt zunächst eine vereinfachte Architektur eines sWfMS, das im Kapitel 2.1 bereits behandelt wurde. Nun soll das vorhandene WfMS um eine neue Funktionalität erweitert werden. Die bisherige Annahme über Services als Utilities – Dienste, die immer da sind und bei Bedarf verwendet werden können, soll nun nicht mehr gelten. D.h. unter Umständen können die benötigten Services nicht vorhanden sein (nicht laufen), wenn man sie (bei Ausführung eines Simulationsworkflows) braucht. Eine neue Funktionalität soll entwickelt werden, damit die benötigten Services zur Laufzeit eines Workflows bei Bedarf in einer Cloud-Umgebung provisioniert werden können. Ebenfalls sollen die Services mit der darunterliegenden Infrastruktur wieder freigegeben werden, falls man sie nicht mehr braucht.

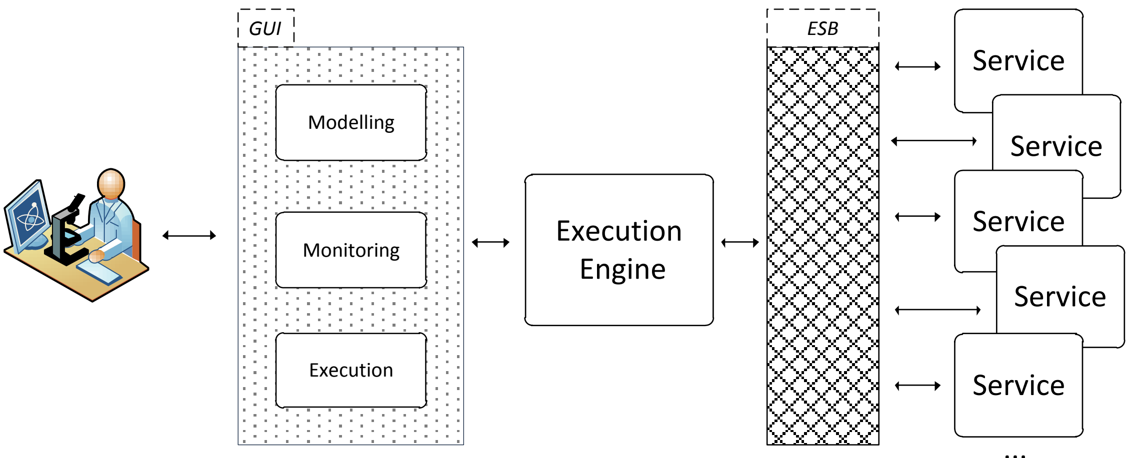

Abbildung 3-1: Architektur eines sWfMS (vereinfacht)

Im Bezug auf die Realisierung dieser Funktionalität im Rahmen eines WfMS stellt sich die Frage, auf welcher Ebene die Erweiterung stattfinden soll. Die Entscheidung ist nicht einfach und muss wohl überlegt sein, denn die Erweiterung der involvierten Komponenten kann unter Umständen einen hohen Entwicklungsaufwand bedeuten.

In der Abbildung 3-1 ist eine vereinfachte Architektur eines WfMS zu sehen, dabei werden folgende Bereiche betrachtet: ein GUI für Modellierung, Steuerung und Überwachung von Simulationen, Execution Engine für die tatsächli-

che Ausführung von Simulationsprozessen, Web Services als einzelne Simulationsschritte sowie weitere Infrastruktur-Komponenten wie ein ESB.

3.1 Ansatz I (Externer Service)

Als erstes kann untersucht werden, ob die neue Provisionierungsfunktionalität mit bereits vorhandenen Mitteln realisiert werden kann. Das würde dann heißen, dass die Erweiterung mit überschauberem Entwicklungsauswand verbunden wäre, denn keine der vorhandenen Komponenten muss erweitert werden. Wenn man etwas mit Workflow-Technologie umsetzen will, so ist folgende Vorgehensweise üblich: Man entwickelt eine neue Funktionalität und bietet sie als Web Service an. So kann die Funktionalität bei einer Prozess-Modellierung berücksichtigt und damit genutzt werden.

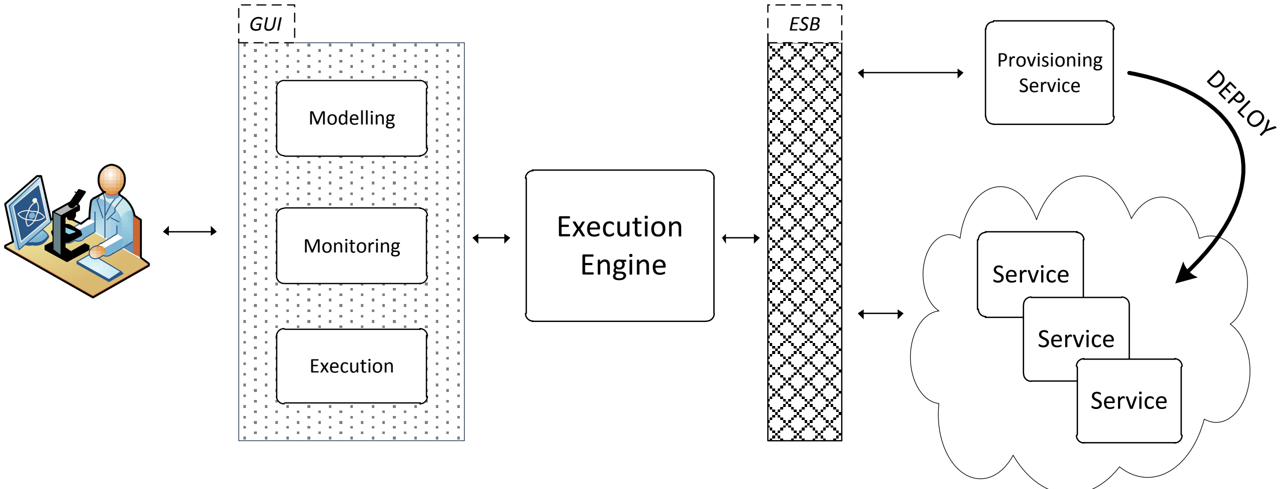

Abbildung 3-2: Ansatz I – Externer Service

Die Abbildung 3-2 zeigt einen Provisionierungsservice, der die neue Provisionierungsfunktinoalität kapselt und nach außen zwei Operationen anbietet. D*eploySimulationService* provisioniert einen gewünschten Simulationsservice in einer Cloud-Umgebung. *UndeploySimulationService* macht das Gegenteil und beendet den Service samt allen allokierten Ressourcen.

Nun kann der Provisionierungsservice bei der Modellierung von Simulationen explizit benutzt werden. Ein Simulationsprozess mit der explizit modellierten Provisionierung könnte dann z.B. so aussehen wie die Abbildung 3-3 zeigt.

Durch einen Aufruf der Operation *DeploySimulationService* wird der benötigte Simulationsservice provisioniert und steht dann zur Verfügung. Die Endpunkt-Referenz des provisionierten Services ist bekannt, somit kann der Simulationsservice aufgerufen werden und den entsprechenden Simulationsschritt ausführen. Danach kann die Service-Instanz über die Operation *UndeploySimulationService* wieder beendet werden, falls der Service im weiteren Verlauf der Simulation nicht benötigt wird.

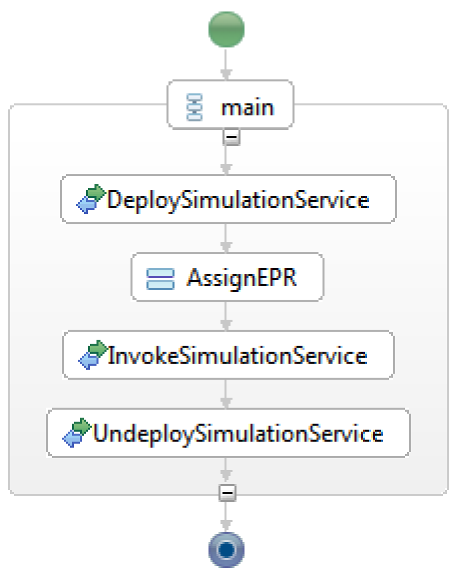

Abbildung 3-3: Ansatz I - Explizite Modellierung

Mit diesem Ansatz lässt sich also die gewünschte Erweiterung eines WfMS um die Provisionierungsfunktionalität realisieren, ohne dass bestehende Komponenten modifiziert werden müssen. Desweiteren bietet der Ansatz dem Benutzer die maximale Kontrolle über alle Provisionierungsprozesse, denn es passiert nur das, was vom Benutzer selbst modelliert wurde. Es können unterschiedliche, auch sehr komplexe Provisionierungsstrategien entwickelt und eingesetzt werden, z.B.:

- Provisionierung aller benötigten Services vor der Ausführung einer Simulation.
- Provisionierung eines Services unmittelbar vor dem jeweiligen Aufruf.
- Parallele Provisionierung von Services im Hintergrund während der Laufzeit.
- Falls Informationen über die Deployment-Dauer und Dauer von Simulationsschritten vorliegen, kann eine Provisionierungsstrategie flexibel angepasst werden.

Die volle Kontolle über die Provisionierung, die der Benutzer bei der expliziten Modellierung der Provisionierungsaspekte hat, ist jedoch mit einigen signifikanten Nachteilen verbunden. (1) Existierende Prozessmodelle können die Provisionierung nicht ohne weiteres nutzen, dazu muss ein Prozessmodell manuell erweitert werden. (2) Die Prozesslogik wird durch Provisionierungsaspekte aufgebläht, dadurch geht die Übersicht verloren. Man kann die Simulationslogik von der Provisionierungslogik nicht auf einen Blick unterscheiden, denn in beiden Fällen werden normale BPEL-Sprachkonstrukte benutzt. (3) Der Benutzer kann sich nicht auf die Modellierung von Simulationen konzentrieren, sondern muss sich auch über die dahinterliegende Infrastruktur im Klaren sein. Jedoch wollte man mit dem mächtigen Frontend genau das Gegenteil er-

reichen – die Komplexität eines WfMS vor dem Benutzer verstecken. (4) Iterative Vorgehensweise bei der Entwicklung von Simulationen wird durch die explizit modellierte Provisionierungsaspekte erheblich erschwert.

Der vorgestellte Ansatz ist auf Grund der zahlreichen damit verbundenden Probleme keine zufriedenstellende Lösung. Nun werden andere Ansätze betrachtet, die eine Modifizierung bzw. Erweiterung vorhandener WfMS-Komponenten voraussetzen.

3.2 Ansatz II (BPEL-Erweiterung)

Der vorgestellte Ansatz I konnte zwar die Provisionierung in ein bestehendes WfMS integrieren, die Lösung hat sich jedoch als umständlich und nicht praktikabel erwiesen. Eine Ursache dafür liegt darin, dass die vorhandenen BPEL-Sprachmittel keine speziellen Konstrukte für die Service-Provisionierung haben. Der Ansatz II versucht das Problem durch eine Erweiterung der Sprache BPEL zu lösen.

Durch spezielle Provisionierungskonstrukte in der dafür erweiterten BPEL-Sprache können Provisionierungsaspekte viel kürzer und eleganter beschrieben werden, wie das folgende Beispiel zeigt.

```
<bpel:invoke
   name="InvokeSimulationService"
   provisioning="yes">
   ...

</bpel:invoke>
```

Listing 3-1: Beispiel für BPEL-Erweiterung

Hier wurde die Invoke-Aktivität durch ein optionales `provisioning`-Attribut erweitert, das folgende Bedeutung hat: Ist das Attribut vorhanden und dessen Wert ist gleich „yes“, so wird der entsprechende Simulationsservice zuerst provisioniert, spätestens vor dem Aufruf dieser Invoke-Aktivität.

Damit eine BPEL-Erweiterung auch produktiv genutzt werden kann, müssen beteiligte WfMS-Komponenten mit dem erweiterten BPEL umgehen können. Die beteiligten Komponenten wären in unserem Fall die Modellierungskomponente vom GUI und die Execution Engine (auch Process oder Workflow Engine genannt).

In [24] werden verschiedene Möglichkeiten, eine BPEL-Erweiterung zu realsiieren diskutiert. Die in der Abbildung 3-4 gezeigte Möglichkeit (A1) besteht darin, dass das Modellierungswerkzeug sowie die Process Engine erweitert werden (Abbildung 3-4).

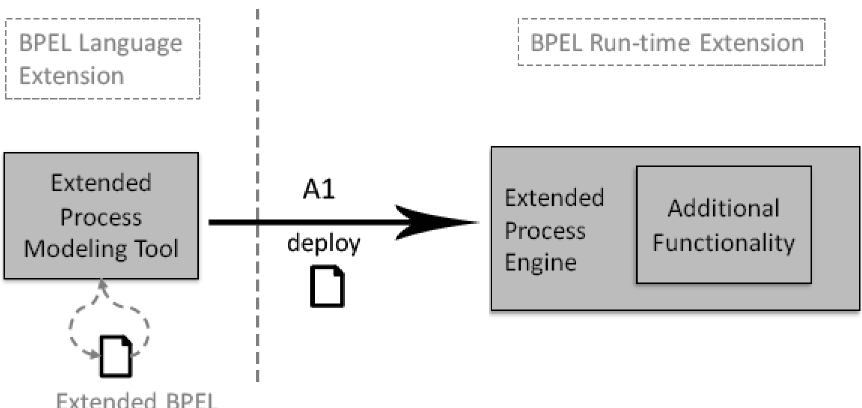

Abbildung 3-4: A1 – Erweiterung von BPEL + Process Engine [24]

Das Modellierungswerkzeug wird dahingehend erweitert, dass die Modellierung eines erweiterten BPEL-Prozessmodells unterstützt wird. Das Prozessmodell wird auf der ebenfalls erweiterten Process Engine deployt. Die Process Engine muss die neuen BPEL-Konstrukte verstehen und die Provisionierungslogik (und damit auch die eigentliche Provisionierungsfunktionalität) implementieren bzw. geeignet einbinden.

Bei dieser Vorgehensweise ist der notwendige Entwicklungsaufwand für die Realisierung einer Erweiterung als hoch einzuschätzen, in erster Linie bedingt durch die Komplexität einer Process Engine. Dies verdeutlicht folgende Abbildung, die den Aufbau von Stuttgarter Workflow Maschine (SWoM) [25] zeigt.

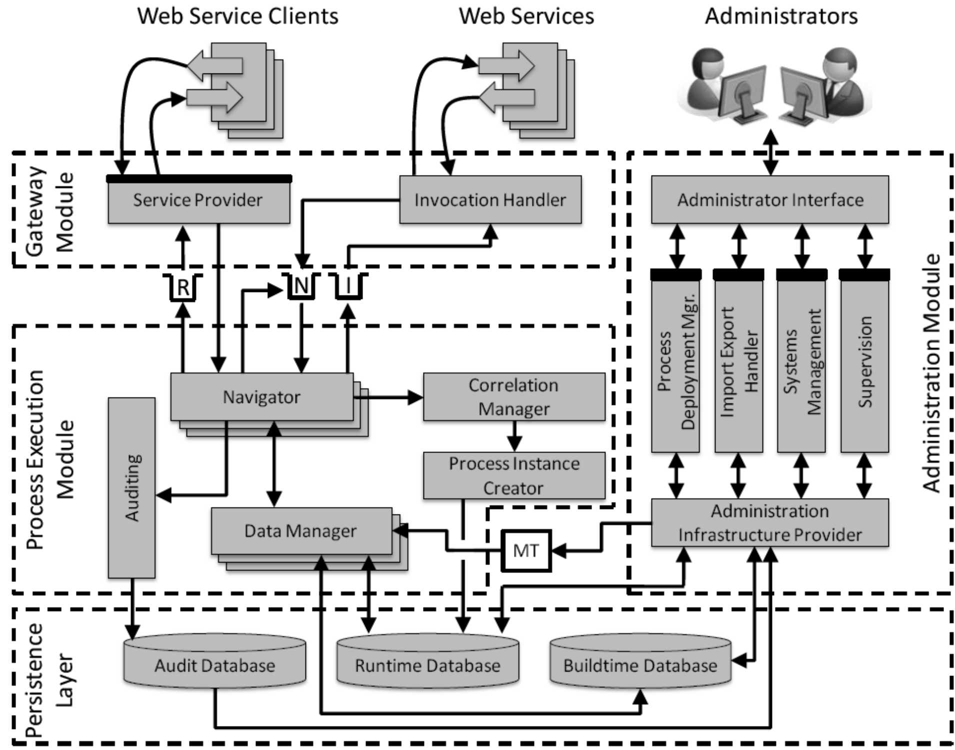

Abbildung 3-5: Architektur von SWoM [24]

Die Entscheidung über eine Erweiterung von Process Engine sollte deswegen gut überlegt sein. Dagegen spricht auch die Tasache, dass die Erweiterung einer bestimmten Process Engine zu einer Einschränkung die Portabilität der Prozessmodelle führen kann. Denn die erweiterten Prozessmodelle können nur auf einer entsprechend erweiterten Process Engine laufen.

Ein anderer ebenfalls in [24] beschriebener Ansatz (A2) versucht die Erweiterung der Process Engine zu vemeiden und beschränkt sich auf die Erweiterung von BPEL und von dem Modellierungswerkzeug. Für die Prozessausführung soll weiterhin die Standard Process Engine eingesetzt werden. In einem zusätzlichen Zwischenschritt wird das erweiterte BPEL-Prozessmodell in ein Standard BPEL-Prozessmodell transformiert, welches dann auf einer Standard Process Engine deployt und ausgeführt werden kann. Die Abbildung 3-6 zeigt schematisch den Ablauf.

Das Kernstück bei dieser Variante ist die Transformation der Prozessmodelle, wobei die neuen Elemente vom erweiterten BPEL durch vorhandene Sprachkonstrukte ersetzt werden. Ein wesentlicher Nachteil ist jedoch, dass die Transformation nicht immer möglich ist, daher ist die Vorgehensweise nicht universell einsetzbar.

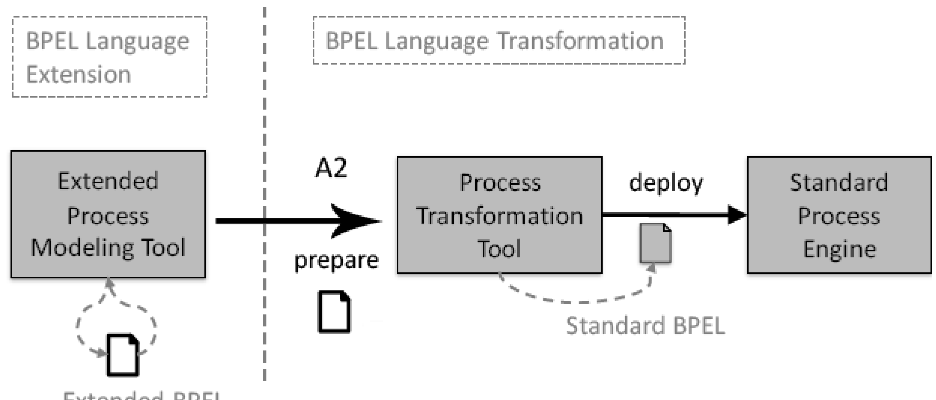

Abbildung 3-6: A2 - Erweiterung von BPEL + Transformation [24]

Der Ansatz A2 eignet sich gut für solche Fälle, wenn die benötigte Funktionalität sich mit Standardkonstrukten von BPEL beschreiben lässt, die BPEL-Erweiterung aber sinnvoll wäre, z.B. wenn dadurch Lesbarkeit der Prozessmodelle erhöht oder die Modellierung vereinfacht wird.

Auf Grund der Diskrepanz zwischen dem modellierten Prozessmodell und dem deployten und ausgeführten Prozessmodell kann die Überwachung der laufenden Prozessinstanzen nicht ohne weiteres erfolgen. Die Überwachung im GUI für Simulationsworkflows basiert auf dem grafischen Prozessmodell. Daher ist eine Erweiterung der Überwachungskomponente notwendig, indem eine Zuordnung zwischen den Events des deployten Prozessmodells und dem angezeigten Modell realisiert wird.

Nach der Empfehlung in [24] soll eine BPEL-Erweiterung für die Realisierung einer neuen Funktionalität eines WfMS nur dann zum Einsatz kommen, wenn die Spracherweiterung von BPEL unumgänglich ist bzw. von vorne rein als Anforderung gilt. In den anderen Fällen soll die Realisierung der Funktionalität nach Möglichkeit durch Erweiterung anderer Infrastruktur-Komponenten erfolgen, wie z.B. von einem Enterprise Service Bus.

3.3 Ansatz III (Sonstige Infrastruktur-Komponenten)

Bei diesem Ansatz sollen die Probleme der vorherigen Ansätze I und II vermieden werden, also lassen sich die Kernideen wie folgt zusammenfassen:

(i) Das Prozessmodell einer Simulation soll keine Provisionierungsaspekte beinhalten, dadurch kann sich der Benutzer auf die Entwicklung der Simulationen konzentrieren. Aus dieser Sicht ist die Provisionierung für den Benutzer transparent.

(ii) Dennoch sollen Möglichkeiten vorgesehen werden, wodurch der Benutzer die Provisionierungsprozesse auf eine vereinfachte Art und Weise (dadurch evtl. eingeschränkt) steuern kann. Z.B durch die Auswahl einer bestimmten Provisionierungsstrategie aus einer vorgegebenen Liste.

(iii) Als Folgerung aus (i) ist eine BPEL-Erweiterung nicht notwendig, auch eine Erweiterung der Process Engine soll vermieden werden, wenn die gewünschte Erweiterung des WfMS durch Erweiterungen anderer Infrastruktur-Komponenten realisiert werden kann.

Bei diesem Ansatz wird die Provisionierungsfunktionalität auf der ESB-Ebene angesiedelt, wie aus der Abbildung 3-7 ersichtlich ist. Bisher wurde der ESB eingesetzt um die benötigten Services über eine Service Discovery Komponente zu finden und aufzurufen. Der Ablauf von Service Discovery muss um die neue Provisionierungslogik erweitert werden. Dadurch kann die Erweiterung transparent für den Benutzer erfolgen, bei der Modellierung von Simulationen finden keine Änderungen statt.

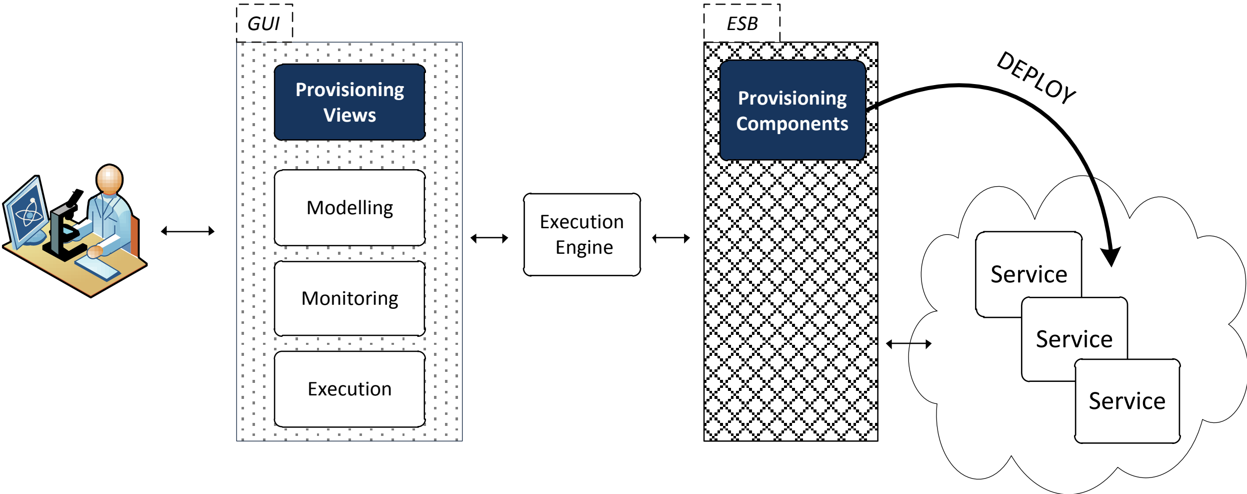

Abbildung 3-7: Ansatz III – Sonstige Infrastruktur-Komponenten

Die eigentliche Provisionierungsfunktionalität kann als ESB-Erweiterung realisert werden oder alternativ als externe Komponente an ESB angebunden werden. Um dem Benutzer die Kontrolle über die Provisionierungsprozesse zu ermöglichen ist eine Erweiterung vom GUI notwendig.

Dieser Ansatz wurde als eine geeignete Lösung für die Erweiterung eines WfMS um die Provisionierungsfunktionalität ausgewählt und wird daher im weiteren Verlauf der Arbeit verfolgt.

4 Architektur für On-Demand Provisionierung

Dieser Abschnitt stellt zunächst die am IAAS entworfene Architektur für die On-Demand Provisionierung von Workflow Ausführungsumgebung und Services für wissenschaftliche Workflows in Cloud-Umgebungen [11] vor. Die Bestandteile der Architektur sowie die Zusammenhänge zwischen den Komponenten werden beschrieben. Anschließend werden Aspekte der Provisionierung von Simulationsservices näher betrachtet und auf die Erweiterung der Architektur bezüglich der Service-Provisionierung konzeptionell eingegangen.

4.1 Architektur-Überblick

Die Komponenten der entworfenen Architektur für On-Demand Provisionierung werden in der Abbildung 4-1 dargestellt. Dabei sind sie in zwei Dimensionen aufteilt. Vertikal werden die Komponenten nach Ort unterschieden, wo sie laufen - lokal oder in einer Cloud-Umgebung. Horizontal erfolgt die Trennung nach Zeit, wann die Komponenten benötigt werden bzw. bereits laufen müssen. Hier unterscheidet man zwischen drei Bereichen: Modellierungszeit (Modeling time), Laufzeit von Workflow Ausführungsumgebung (Middleware runtime) und Laufzeit von Services (Service runtime).

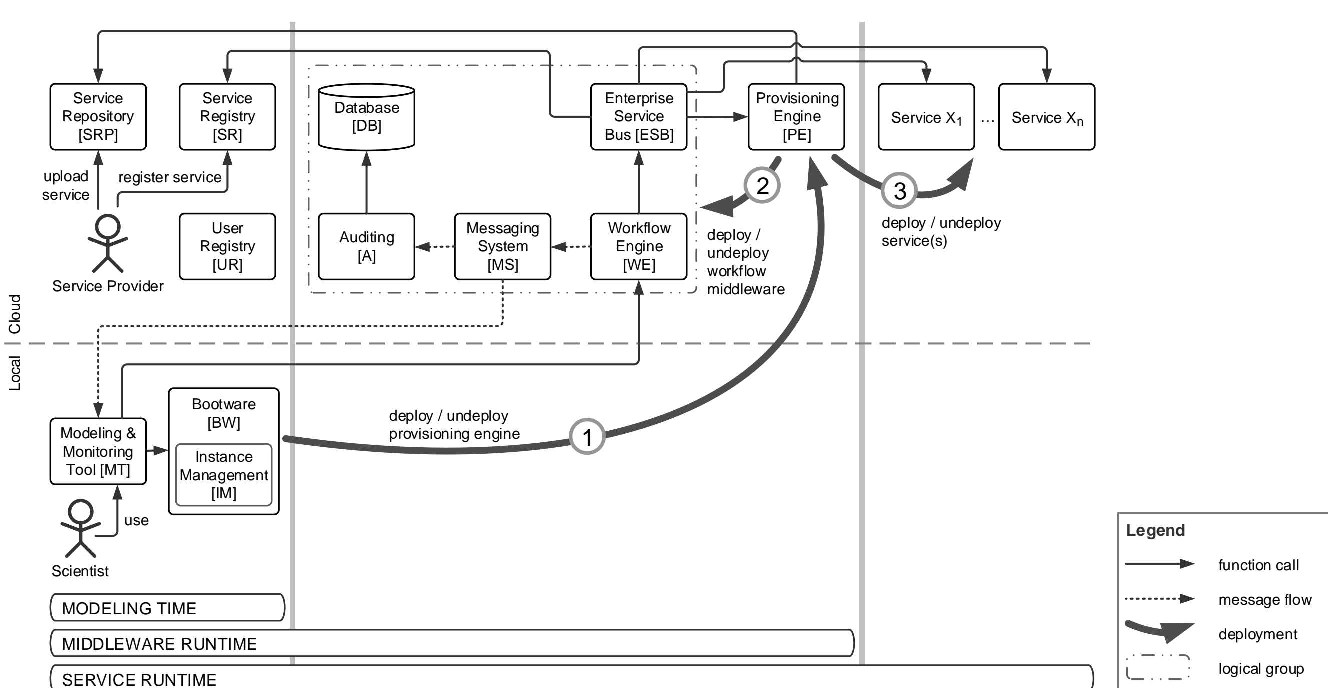

Abbildung 4-1: Architektur für On-Demand Provisionierung [11]

Der Architekturansatz sieht vor, dass möglichst viele Komponenten in einer Cloud-Umgebung laufen und der Benutzer nur das notwenige Minimum an Software auf seinem lokalen Rechner installieren und einrichten muss. Im Wesentlichen braucht der Benutzer das kombinierte Werkzeug für die Modellie-

rung, Steuerung und Überwachung von wissenschaftlichen Workflows (Modeling & Monitoring Tool). Für die initiale Provisionierung der Workflow-Ausführungsumgebung wird zusätzlich die Bootware-Komponente benötigt. Diese zwei Komponenten müssen bereits zur Modellierungszeit laufen.

Für die Modellierung von wissenschaftlichen Workflows ist ein Service Katalog erforderlich, der die Beschreibungen vorhandener Simulationsservices im Modellierungswerkzeug zur Verfügung stellt. Damit kann der Wissenschaftler die Services im eigenen Simulationsprozess nutzen. Da die Modellierung gegen die Schnittstellen von Simulationsservices erfolgt, ist ein Zugriff auf Services während der Modellierung nicht notwendig.

Service Registry Für die Registrierung von Services durch einen Service Provider steht die Service Registry zur Verfügung. Die Service Registry ist in erster Linie ein Verzeichnisdienst, wie man ihn aus der service-orientierten Architektur (SOA) [26] kennt. Dort werden alle Informationen über einen Service abgebegt, damit dieser gefunden und benutzt werden kann. Bei dem behandelten Architekturansatz werden in der Service Registry zusätzliche Informationen über so genannte *Not Provisioned Services* abgelegt bzw. zur Verfügung gestellt. Im Unterschied zu „normalen" Web Services müssen solche Services vor der Nutzung zuerst provisioniert werden.

Service Repository Die Service Repository Komponente dient als Ablageort für Service-Distributionen, die durch einen Service Provider hochgeladen werden. Sollte ein *Not Provisioned Service* im Service Registry registriert werden, ist mit den anderen benötigten Metadaten auch die Information darüber abzulegen, wie die entsprechende Service-Distribution aus dem Service Repository zu bekommen ist. Außer der Upload-Funktionalität sind die Versionierung sowie der geregelte Zugriff auf die Inhalte zwei wichtige Eigenschaften vom Service Repository. Der Zugriff auf die Service Registry sowie auf das Service Repository wird mit Hilfe von der User Registry Komponente geregelt.

Bis jetzt wurden Komponenten behandelt, die spätestens zur Modellierungszeit entweder lokal oder in einer Cloud-Umgebung laufen müssen. Die Ausführung von modellierten Simulationsprozessen benötigt einerseits eine laufende Ausführungsumgebung, damit das Prozessmodell instanziiert und ausgeführt werden kann. Andererseits werden Simulationsservices benötigt, die aus dem Prozess heraus über Invoke-Aktivitäten aufgerufen werden. Beides muss zuerst in einer Cloud-Umgebung provisioniert werden.

Provisioning Engine Die Provisionierung von der Workflow Ausführungsumgebung und von Simulationsservices erfolgt mit einem generischen Ansatz. Die Provisioning Engine ist in beiden Fällen für die Provisionierung zuständig, sie muss aber davor selbst in einer Cloud-Umgebung provisioniert werden. Dafür kommt die bereits erwähnte Bootware Komponente zum Einsatz, die lokal auf dem Benut-

zer-Rechner läuft und angetriggert vom Modeling & Monitoring Tool das Deployment der Provisioning Engine übernimmt.

Nachdem die Provisioning Engine deployt wurde und arbeitsbereit ist, wird das Deployment der Workflow-Ausführungsumgebung durchgeführt. Eine Workflow-Ausführungsumgebung zeigt die Abbildung 4-1 als eine logische Gruppe zusammenhängender Komponenten. Das abgebildete Workflow-Middleware ist an den SimTech sWfMS angelehnt und besteht aus einer Workflow Engine, einer Auditing Application mit einer Datenbank dahinter und einem Enterprise Service Bus. Dabei kommunizieren das Modeling & Monitoring Tool, die Auditing Application und die Workflow Engine über ein Messaging System.

Enterprise Service Bus

Auf der Workflow Engine wird ein modellierter Simulationsprozess deployt und beim Starten einer Simulationsausführung instanziiert. Ein Simulationsprozess besteht aus mehreren Schritten, die durch Simulationsservices realsiert sind. Die Workflow Engine ruft die entsprechenden Simulationsservices nicht direkt auf, sondern nutzt den Enterprise Service Bus dafür. Das wird erreicht durch die Weiterleitung der Web Service Aufrufe an den ESB.

Der ESB nimmt die eingehenden Web Service Aufrufe entgegen und hat die Aufgabe einen passenden Web Service dazu zu finden. Auf der Suche nach dem passenden Service konsultiert der ESB die Service Registry, bei der alle verfügbaren Services registriert sein müssen. Abhängig von der Service Information, die über die Service Registry veröffentlicht wurden, trifft der ESB die Entscheidung über die weitere Vorgehensweise im Bezug auf den eingegangenen Serviceaufruf. Handelt es sich hierbei um einen Service, der noch nicht provisioniert ist, stoßt der ESB die Provisionierung an. Dadurch wird eine Service-Instanz in einer Cloud-Umgebung mit der gesamten darunterliegenden Infrastruktur deployt.

Die Provisionierung der Simulationsservices wird von der Provisioning Engine umgesetzt, die auch die Provisionierung der Worklfow Ausführungsumgebung vorgenommen hat. Die Provisioning Engine holt die entsprechende Service-Distribution von dem Service Repository zu sich und deployt den Service in einer Cloud-Umgebung. Auf Grund dieser Vorgehensweise muss die Provisioning Engine wissen, wie die Service-Distributionen vom Service Repository geholt werden können.

Als Ergebnis vom Deployment wird eine Endpunkt-Referenz von der erstellten Service-Instanz an den ESB zurückgegeben. Danach kann der Service aufgerufen werden und die Ergebnisse des Aufrufes gehen zurück an den Aufrufenden, in dem Fall an die Workflow Engine. Das Beenden der Service-Instanzen erfolgt ebenfalls über die Provisioning Engine, angetriggert von der ESB-Provisionierungslogik.

In folgenden Abschnitten werden ausgwählte Komponenten etwas detaillierter beschrieben.

4.2 Simulationsservices

Simulationsservices kapseln einzelne Schritte einer Simulation und bieten diese gekapselte Funktionalität als Web Service an. Der Einsatz von Simulationsservices als wiederverwendbare Bausteine ist aber nur dann möglich wenn bereits bei der Entwicklung von Services der Aspekt der Wiederverwendbarkeit berücksichtigt wird.

Die Abbildung 4-2 zeigt einen typischen Ansatz, wie man bestehende Simulationsanwendungen umbauen kann, damit sie durch einen Workflow-Prozess beschrieben werden können. Dabei wird die Logik einer Simulationsanwendung in kleinere Teile aufzugesplitten, die nach Möglichkeit keine direkten Abhängigkeiten voneinander haben sollten. Ist das erreicht, werden um diese Simulationsschritte herum spezielle Wrapper gebaut, die die Simulationslogik als Web Service Operationen nach außen anbieten. Somit sind die einzelnen Simulationsschritte als Web Services erreichbar und können in einem BPEL-Prozessmodell benutzt werden. Das BPEL Prozessmodell beschreibt die gesamte Logik einer Simulation.

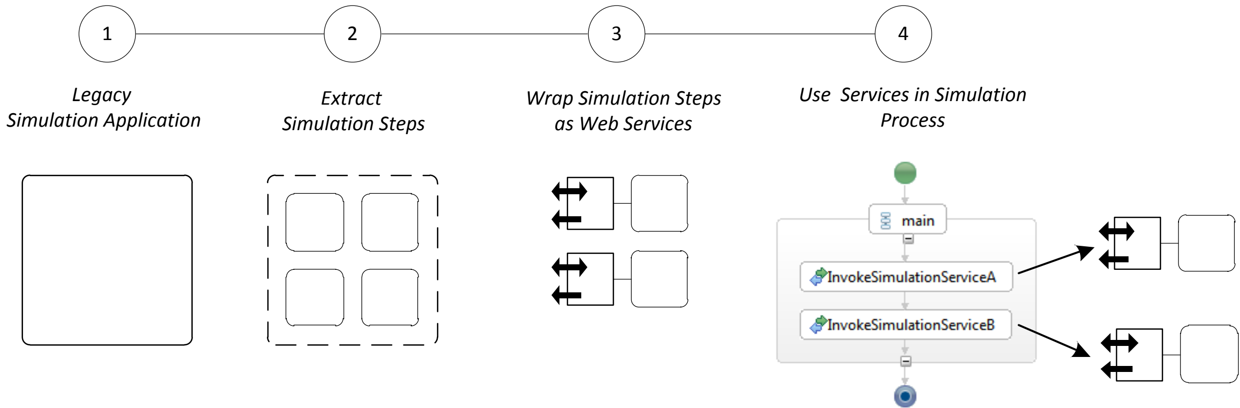

Abbildung 4-2: Refactoring von Legacy Simulationsanwednungen

Laufende Instanzen eines Simulationsservices, der als Wrapper für Teile einer Legacy Simulationsanwendung geschrieben wurde, können in der Regel nicht wiederverwendet bzw. parallel genutzt werden. Der Grund dafür liegt in der eigentlichen Simulationslogik, die aus einer Legacy Simulationsanwendung extrahiert wurde und daher die angestrebte Wiederverwendung bzw. parallele Nutzung nicht unterstützt. Wird dagegen ein Simulationsservice direkt als Web Service entwickelt, können Aspekte der gemeinsamen Verwendung von den bereitgestellten Operationen bei der Entwicklung berücksichtigt werden.

Es ist wichtig, zwischen diesen beiden Arten von Simulationsservices zu unterscheiden, denn im Bezug auf die Provisionierung ergeben sich daraus deutliche Unterschiede.

Bei dem vorgestellten Architekureinsatz sind eine Reihe weiterer Unterscheidungsmerkmale von Simulationsservices von Bedeutung. Die Abbildung 4-3 gibt einen Überblick über die relevanten Arten von Simulationsservices.

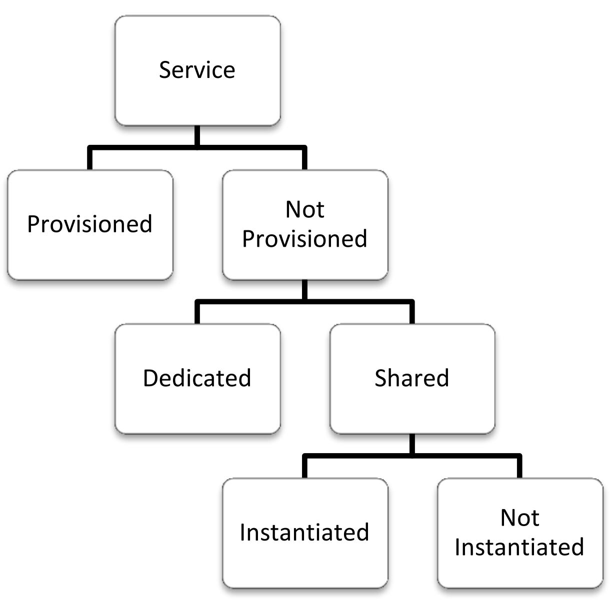

Abbildung 4-3: Arten von Simulationsservices

Zunächst werden Services abgegrenzt, die nicht von der Provisionierung betroffen sind (*Provisioned Services*). Bei dieser Art von Services handelt es sich um „normale" Services, die bereits laufen, in der Service Registry verzeichnet sind und nicht provisioniert werden müssen, sondern einfach benutzt werden können. Bei einem Aufruf eines solchen Services wird die Provisionierungslogik auf der ESB-Ebene nicht angewendet.

Die restlichen Services werden als *Not Provisioned Services* bezeichnet. Für diese Art von Services existiert eine entsprechende Service-Distribution im Service Repository, die von der Provisioning Engine verwendet wird um eine Service-Instanz in einer Cloud-Umgebung zu erzeugen. Vor der Registrierung eines Services bei der Service Registry muss die dazugehörige Service-Distribution ins Service Repository hochgeladen werden und die Referenz darauf bei der Registrierung mitangegeben werden.

Not Provisioned Services werden noch mal hinsichtlich ihrer Wiederverwendbarkeit unterteilt. *Dedicated Services* sind Services, die in der Regel als Wrapper für Legacy Simulationsanwendungen geschrieben wurden und erlauben daher keine Mehrfachnutzung. Aus diesem Grund müssen sie jedes Mal neu provisioniert werden, wenn sie benötigt werden. Im Gegensatz dazu können *Shared*

Services mehrfach genutzt werden. Die Instanziierung erfolgt nur dann, wenn noch keine Instanz dieses Services zu diesem Zeitpunkt provisioniert ist. Abhängig von dem beschriebenen Zustand ist ein Service dann *Instantiated* oder *Not Instantiated*.

4.3 Service Repository

Mit Service Repository wurde die Architektur eines sWfMS erweitert um eine Komponente für die Verwaltung von Service-Distributionen. Dabei ist eine Service-Distribution ein Paket, das alle benötigten Informationen enthält um einen Service in einer Cloud-Umgebung zu provisionieren. Der Architekturansatz definiert an dieser Stelle keine Einschränkungen an das tatsächliche Format. Allerdings besteht ein direkter Zusammenhang mit der Provisioning Engine, die das Format einer solchen Service-Distribution verstehen muss. Ein Beispiel für eine Service-Distribution wäre eine CSAR-Datei, die einen oder mehrere Services in TOSCA [12] beschreibt.

Das Service Repository bietet das Upload und den Download von Service-Distributionen an. Ein Service Provider kann die Service-Distributionen hochladen. Der Download erfolgt durch die Provisioning Engine. Neben den genannten Operationen wird die Versionierung der hochgeladenen Dateien unterstützt. Die Zugriffkontrolle schützt mit Hilfe von User Registry vor unbefugten Zugriffen auf die Inhalte.

Bei der Registrierung eines Services in der Service Registry wird eine Referenz auf die entsprechende Service-Distribution im Service Repository benötigt, damit die Provisionierung eines Services erfolgen kann.

4.4 Service Registry

Die Service Registry ist in SOA [26] von zentraler Bedeutung, denn sie bietet die Möglichkeit, Services zu veröffentlichen sowie nach geeigneten Services zu suchen. Damit verbindet sie den Service Provider mit dem Service Consumer und ermöglicht die Wiederverwendung von Services.

In der vorgestellten Architektur hat die Service Registry die Aufgabe, Simulationsservices in einem Verzeichnis zu organisieren sowie die Provisionierung von Services in einer Cloud-Umgebung zu unterstützen. Bei der Veröffentlichung von Services werden unter anderem Metadaten abgelegt, die vom ESB für Entscheidungen bezüglich der Provisionierung von Simulationsservices benötigt werden. Insbesondere sind Informationen über die unterschiedlichen Service-Arten, wie sie im Abschnitt 4-2 beschrieben sind, sehr wichtig.

Für die Provisionierung von Simulationsservices ist folgender Funktionsumfang der Service Registry vorgesehen:

(1) Das Holen von Metadaten eines Services ist die grundlegende Operation.

(2) Bei *Not Provisioned Services* können neue Service-Instanzen, die durch die Provisioning Engine erzeugt wurden, registriert und bei Nichtnutzung wieder abgemeldet werden.

(3) Bei *Shared Services* können zusätzlich die Service Aufrufe registriert und nach einem erfolgten Aufruf abgemeldet werden. Für die Entscheidung ob eine Service-Instanz beendet werden soll, kann die Anzahl der aktiven Aufrufe eines *Shared Services* bei der Service Registry angefragt werden.

4.5 Provisioning Engine

Die Provisioning Engine ist für das Deployment und das Undeployment der Workflow-Ausführungsumgebung und Simulationsservices in einer Cloud-Umgebung zuständig. Beides sind aus der Sicht der Provisioning Engine Distributionen von Cloud-Anwendungen und können damit einheitlich behandelt werden solange sie in einem Format vorliegen, das die Provisioning Engine versteht.

Das Deployment der Workflow-Aussführungsumgebung wird von der Bootware Komponente angestossen. Das passiert wenn der Benutzer der Modellierung eines Simulationsprozesses abgeschlossen hat und die Simulation startet. Die Bootware muss zunächst die Provisioning Engine in einer Cloud-Umgebung deployen. Dann kann das Deployment der Workflow-Ausführungsumgebung von der Provisioning Engine durchgeführt werden.

Im Gegensatz dazu wird das Deployment von Simulationsservices indirekt von der Workflow Engine initiiert. Bei einer Simulationsausführung navigiert die Workflow Engine durch die Prozessinstanz und ruft entsprechende Services über den ESB auf. Der ESB entscheidet anhand von Service Informationen aus der Service Registry über die Notwendigkeit einer Service-Provisionierung.

Für die Erzeugung einer Service-Instanz in einer Cloud-Umgebung braucht die Provisioning Engine Informationen über den Speicherort der entsprechenden Service-Distribution. Dazu wird eine Referenz auf die Service-Distribution im Service Repository aus den Service-Metadaten benutzt. Die Provisioning Engine muss in der Lage sein, das Service Repository anzusprechen um den Download der Service-Distribution vorzunehemen.

Neben dem Deployen der Workflow-Ausführungsumgebung bzw. von Service-Instanzen gehört auch das Undeployment derselben zu den Aufgaben einer Provisioning Engine. Die Entscheidung über das Beenden von Service-Instanzen trifft der ESB, bei der Workflow-Ausführungsumgebung wird das Undeployment von der Bootware-Komponente gesteuert.

Als Beispiel für eine Provisioning Engine kann OpenTOSCA [27] genannt werden – eine Laufzeitumgebung für TOSCA, die als CSAR-Dateien vorliegende Cloud-Services instanziieren kann.

4.6 Enterprise Service Bus

Der Enterprise Service Bus ist als Teil von Workflow-Middleware für die Discovery von Simulationsservices zuständig. Bei einem Web Service Aufruf aus der Execution Engine heraus muss der ESB einen passenden Web Service über die Service Registry finden.

Um die on-demand Provisionierung von Simulationsservices zu bewerkstelligen, wurde der ESB mit einer Provisionierungslogik ausgestattet. Bei jedem eingehenden Web Service Aufruf wird der in der Abbildung 4-4 gezeigte Prozess durchlaufen. Um Entscheidungen über die Provisionierung zu treffen, werden die in der Service Registry abgelegten Metadaten geholt und analysiert. Das Deployment von Services in einer Cloud-Umgebung erfolgt über die Provisioning Engine. Der abgebildete Prozess wird nachfolgend erläutert.

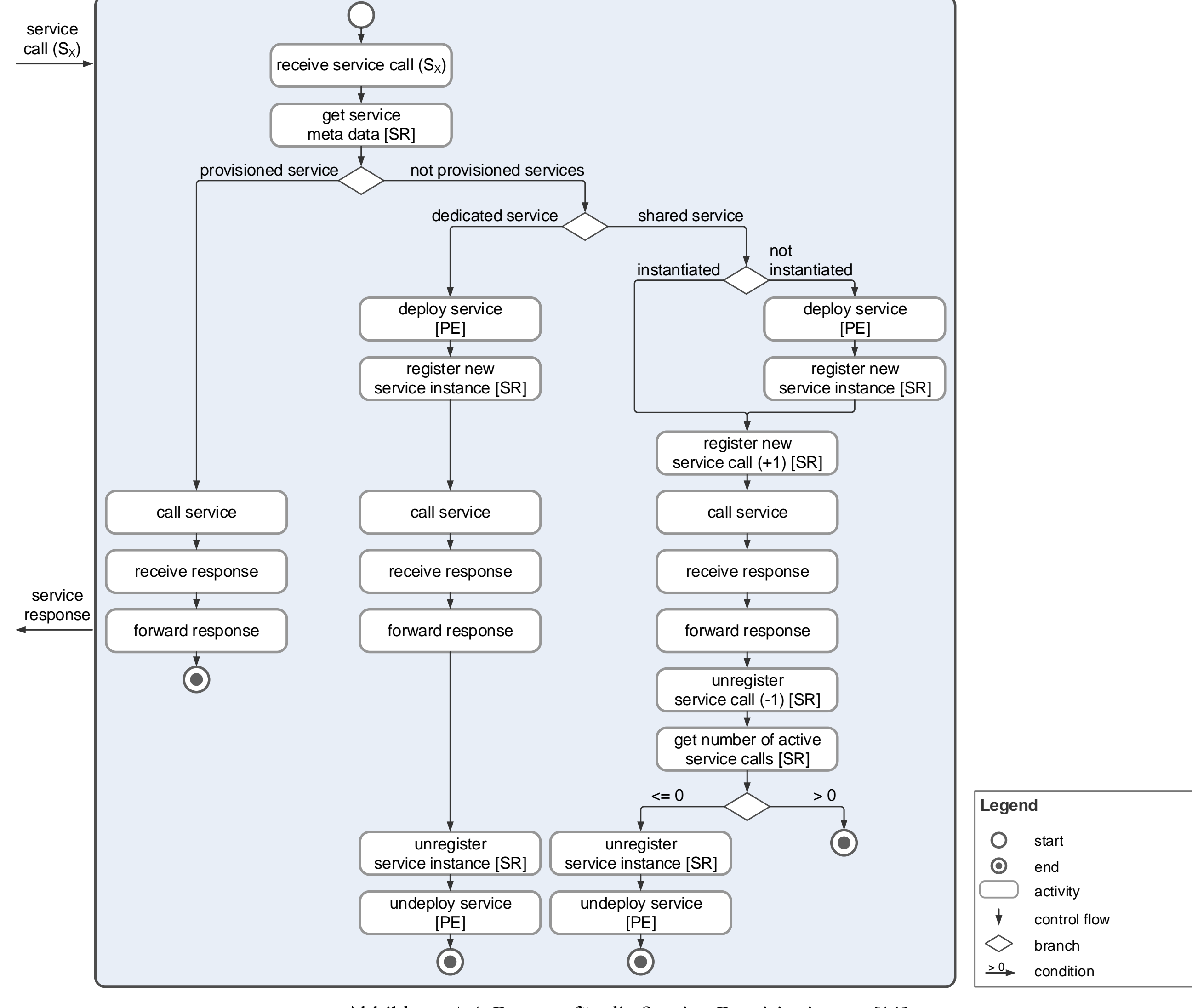

Abbildung 4-4: Prozess für die Service-Provisionierung [11]

Wenn die Execution Engine einen Web Service aufruft, wird der Aufruf an den ESB weitergeleitet und vom ESB konsumiert und verarbeitet. Im ersten Schritt werden Metadaten von dem benötigten Service über die Service Repository angefragt. Wie bereits im Abschnitt 4.4 angedeutet, sind in den Metadaten Angaben über die Art von diesem Service enthalten. Anhand dieser Information wird die erste Unterscheidung gemacht. Handelt es sich hierbei um einen *Provisioned Service* (linker Pfad in der Abbildung), kann der Service direkt aufgerufen werden. Nach dem Erhalt der Antwort-Nachricht wird sie an die Execution Engine weitergeleitet und die Verarbeitung durch den ESB ist damit beendet.

Bei einem *Not Provisioned Service* wird noch einmal unterschieden zwischen einem *Dedicated Service* und einem *Shared Service*. Ein *Dedicated Service* kann nicht wiederverwendet werden, daher muss er in jedem Fall vor dem Aufruf deployt werden (mittlerer Pfad in der Abbildung). Dabei spielt keine Rolle, ob bereits Instanzen dieses Services laufen oder nicht. Das Deployment findet über die Provisioning Engine statt, die mit den Angaben aus den Metadaten versorgt werden muss. Genauer wird die Referenz auf die entsprechende Service-Distribution im Service Repository benötigt. Nachdem der Service in einer Cloud-Umgebung deployt wurde, wird die Service Instanz bei der Service Registry registriert. Anschließend kann der eigentliche Service Aufruf erfolgen. Nach der Weiterleitung der Antwort-Nachricht an die Workflow Engine muss die Service Instanz wiederum über die Provisioning Engine beendet werden. Davor wird die Instanz noch bei der Service Registry abgemeldet. Nach dem Undeployment der Service Instanz ist die Verarbeitung beendet.

Bei einem *Shared Service* wird die Logik noch mal verfeinert (rechter Pfad in der Abbildung). Ist zur Zeit des Service-Aufrufes keine laufende Service-Instanz vorhanden, wird der Service zunächst deployt und die Service-Instanz wird bei der Service Registry registriert. Instanzen dieser Service-Art können mehrere Service-Aufrufe gleichzeitig verarbeiten. Daher wird jeder Service-Aufruf ebenfalls bei der Service Registry angemeldet und nach der Verarbeitung der Antwort-Nachricht wieder abgemeldet. Dieses Handling wird benötigt um Service-Instanzen zu erkennen, die nicht mehr in Verwendung sind und somit beendet werden können. Dazu wird die Service Registry nach der Anzahl der aktiven Service-Aufrufe abgefragt. Sollten noch aktive Aufrufe vorhanden sein, kann die Service-Instanz nicht beendet werden. Nach der Verarbeitung vom letzten aktiven Aufruf wird die Service-Instanz bei der Service Registry abgemeldet und über die Provisioning Engine beendet.

5 Erweiterung der Service-Provisionierung

Im voherigen Abschnitt wurde die Architektur für On-Demand Provisionierung von Workflow-Middleware und Simulationsservices in einer Cloud-Umgebung vorgestellt. Dieser Abschnitt beschäftigt sich ausschließlich mit Aspekten der Provisionierung von Simulationsservices. In diesem Zusammenhand wird die Architektur auf mögliche Problemstellen untersucht und daraus resultiertende Verbesserungen bzw. Erweiterungen der Architektur werden vogeschlagen.

Die Abbildung 5-1 stellt zunächst in vereinfachter Form den behandelten Architekuransatz dar, wobei lediglich die für die Provisionierung von Simulationsservices relevanten Komponenten dargestellt werden.

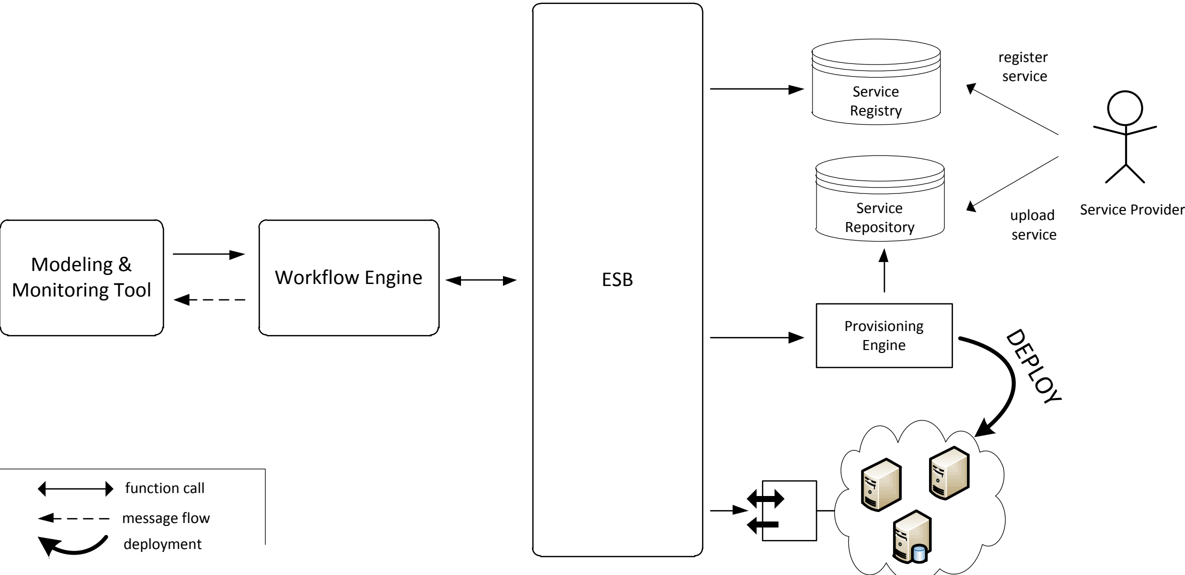

Abbildung 5-1: Architektur für die Provisionierung von Simulationsservices [11] (vereinfacht)

Die Provisionierung von Simulationsservices wird dann benötigt, wenn eine Instanz eines Simulationsprozesses ausgeführt wird und dabei eine Invoke-Aktivität an der Reihe ist. An dieser Stelle versucht die Workflow Engine über den ESB einen passenden Simulationsservice zu finden und aufzurufen. Nach der Feststellung durch den ESB um welche Service-Art es sich handelt, wird die Entscheidung über eine mögliche Provisionierung getroffen. Handelt es sich hierbei um einen Service, der provisioniert werden muss, wird die Provisioning Engine mit der Erstellung einer Service-Instanz in einer Cloud-Umgebung beauftragt. Die Provisioning Engine erhält dabei alle Informationen, die sie für das Deployment benötigt wie etwa die Referenz auf die passende Service-Distribution in dem Service Repository. Darauf holt die Provisioning Engine die Service-Distribution zu sich und erzeugt daraus eine Service-Instanz in einer Cloud-Umgebung.

Aus diesem Ablauf können Annahmen über bestimmte Funktionalität der Komponenten bzw. Anforderungen an die Komponenten abgeleitet werden. (1) Die Provisioning Engine versteht das Format von Service-Distributionen, die im Service Registry durch einen Service Provider abgelegt werden und kann sie verarbeiten. (2) Die Provisioning Engine ist in der Lage, benötigte Service-Distributionen vom Service Repository selbstständig zu holen, d.h. sie kennt die dafür vorgesehene Schnittstelle des Service Repository und kann sie nutzen. (3) Der Enterprise Service Bus kann Operationen der Provisioning Engine im Bezug auf das Deployment und Undeployment von Services nutzen. Der Endpunkt einer erzeugten Service-Instanz ist dem ESB nach der erfolgten Provisionierung bekannt und der Service kann aufgerufen werden.

Die Abbildung 5-2 verdeutlicht die möglichen Problemstellen in der Architektur für die Provisionierung von Simulationsservices, die sich aus den beschriebenen Annahmen ergeben.

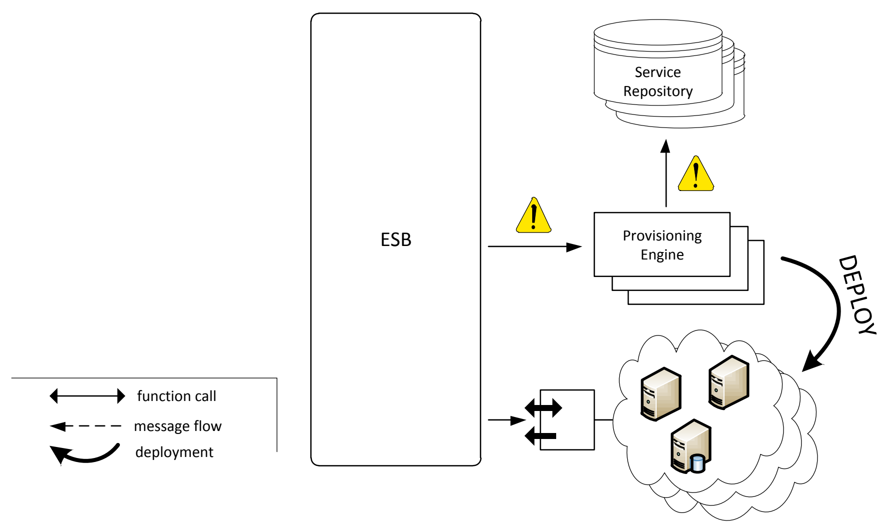

Abbildung 5-2: Mögliche Problembereiche der Architektur für Service-Provisionierung

Eine Provisioning Engine wird in der Regel von einem Cloud-Provider bereitgestellt, damit man in seiner Cloud-Umgebung on-demand Service-Instanzen erzeugen kann. Also hat ein Cloud-Provider, der solche Funktionalität anbietet, eine eigene Implementierung der Provisioning Engine. Es kann davon ausgegangen werden, dass der Funktionsumfang je nach Provisioning Engine unterschiedlich sein kann. Aus diesem Grund kann die Annahme (2) zu einem Problem führen, wenn die eingesetze Provisioning Engine nicht mit einem bestimmten Service Repository umgehen kann. Für die Realisierung vom Service Repository als Verwaltungswerkzeug für Service-Distributionen der Simulationsservices gibt es keine Standards, daher kann die Annahme (2) in einem bestimmten Fall gelten ist aber nicht allgemeingültig für alle möglichen Service Repositories und Provisioning Engines.

Auch die Annahme (1) kann unter Umständen ein Problem werden, falls ein Format für die Speicherung von Service-Distributionen benutzt wird, das von der Provisioning Engine nicht oder nicht komplett unterstützt wird. Hier kann die Nutzung von standardisierten Formaten wie CSAR und TOSCA Abhilfe schaffen. Aber auch dann kann keine vollständige Unterstützung gewährleistet werden, da es in der Hand des Herstellers liegt, inwieweit die eigene Implementierung ein Standard vollständig unterstützt.

Aus der Annahme (3) folgt, dass der ESB eventuell mehrere unterschiedliche Provisioning Engines unterstützen muss, wenn Services in unterschiedlichen Cloud-Umgebungen erzeugt werden sollen. Das kann der Fall sein, wenn von Cloud-Providern bereitgestellte Provisioning Engines zum Einsatz kommen. Unterstützung von unterschiedlichen Provisioning Engines ist mit Änderungen vom ESB-Provisionierungsprozesses verbunden. Denn der ESB ruft die Operationen einer Provisioning Engine direkt aus seinem Provisionierungsprozess heraus, was die Anpassung vom Prozess notwendig macht. Bei unterschiedlichen Provisionierungsstrategien, die ebenfalls als Prozessbeschreibungen vorliegen, sind auch sie von den Anpassungen betroffen, was kein optimales Handling ist.

Die beschriebenen Problemfälle können durch folgenden Vorschlag für Erweiterung der Architektur gelöst werden.

5.1 Provisioning Manager

Durch die Hereinnahme einer neuen Komponente - Provisioning Manager - wird eine zusätzliche Abstraktionsschicht geschaffen. Wie aus der Abbildung 5-3 hervorgeht, versteckt der Provisioning Manager die konkreten Provisioning Engines vor dem ESB.

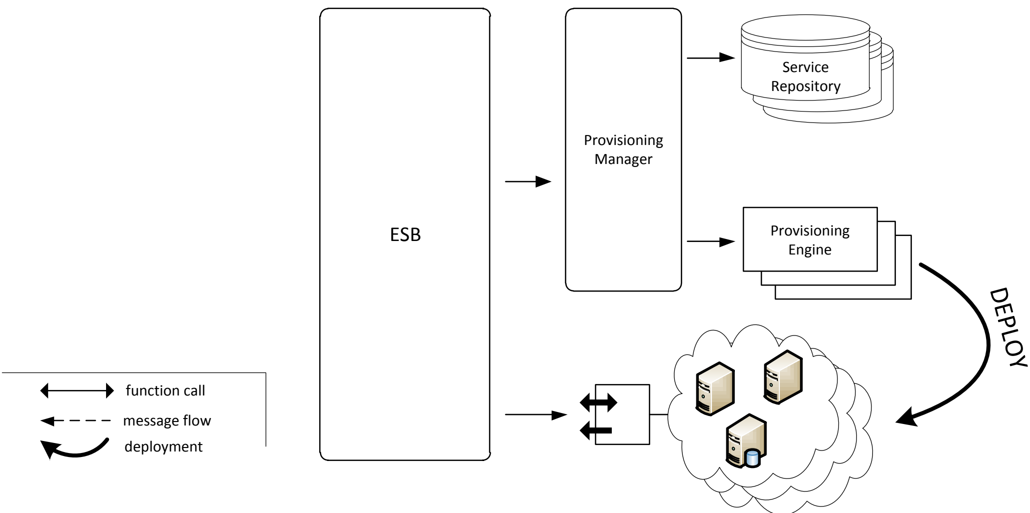

Abbildung 5-3: Provisioning Manager (externer Service)

Dadurch kann der ESB-Provisionierungsprozess gegen die stabilen Schnittstellen des Provisioning Managers agieren und muss sich nicht um die Interaktionen mit der tatsächlich verwendeten Provisioning Engine kümmern. Damit wird der Provisionierungsprozesses bei Nutzung mehrerer Provisioning Engines nicht beeinflusst und muss nicht geändert werden. Dieser Ansatz ermöglicht gleichzeitige Unterstützung und Nutzung mehrerer Provisioning Engines.

Der Provisioning Manager einerseits kapselt die Logik für den Umgang mit unterschiedlichen Provisioning Engines, andererseits wird dadurch eine direkte Anhängigkeit zwischen der Provisioning Engine und dem Service Repository aufgelöst. Da die Annahme (2) beim ursprünglichen Ansatz nicht gewährleistet werden kann, wird der Zugriff auf die Service-Distributionen im Service Repository nun durch den Provisioning Manager geregelt. Der Provisioning Manager kann mit unterschiedlichen Service Repositories umgehen.

Die Erweiterung der Architektur durch den Provisioning Manager hat auch im Bezug auf die Annahme (1) einen signifikanten Vorteil gegenüber dem ursprünglichen Architekturansatz. Das Format von Service-Distributionen im Service Registry muss nicht mehr das gleiche Format sein, das die Provisioning Engine unterstützt. In diesem Fall kann der Provisioning Manager um geeignete Transformationslogik erweitert werden, die das Format aus dem Service Repository in das evtl. abweichende Format der Provisioning Engine überführt.

Die Abbildung 5-3 zeigt den Provisioning Manager als eine eigenständige Komponente, die als externer Service mit dem ESB integriert ist. Alterativ kann die Integration auch durch die in der Abbildung 5-4 dargestellte Erweiterung vom ESB erfolgen.

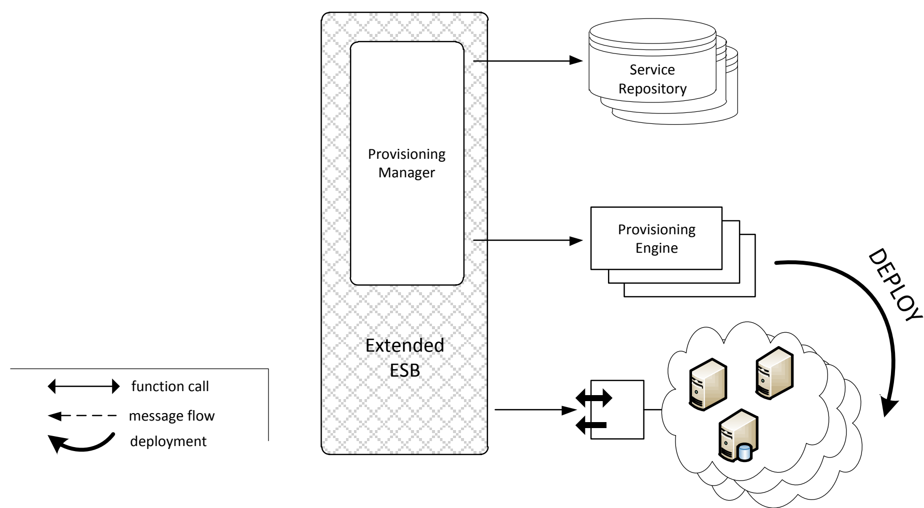

Abbildung 5-4: Provisioning Manager (Erweiterter ESB)

5.2 Provisionierungstrategien

Durch den vorgestellten Provisionierungsprozess, der im ESB abläuft, wird die dynamische Provisionierungsstrategie angewendet. Die Grundidee von dieser Strategie besteht darin, dass die Provisionierung eines Simulationsservices durch den Aufruf von diesem Service ausgelöst werden kann. Diese Strategie ist die Basis für dynamische Provisionierung von Services. Weiterhin sind aber eine Reihe weiterer Strategien denkbar, daher soll die Architektur den Einsatz unterschiedlicher Strategien unterstützen. Im Folgenden wird anhand von einer Beispiel-Strategie untersucht, inwiefern die Möglichkeiten für eine flexible Nutzung von Provisionierungsstrategien durch den vorgestellten Architekturansatz gegeben sind.

Als Beispiel-Strategie wird statische Provisionierung aller Services betrachtet. Das bedeutet: alle involvierten Services müssen beim Start einer Simulation provisioniert werden, falls sie nicht bereits laufen. Nachdem die Simulation beendet ist, sind alle provisionierten Service-Instanzen zu beenden und die Ressourcen sind freizugeben. Diese Strategie kann bei der iterativen Vorgehensweise eingesetzt werden, denn wenn die Services bereits laufen, können bestimmte Simulationsschritte wiederholt werden, ohne dass die Services vor jedem Aufruf wiederholt deployt werden müssen. Bei *Dedicated Services* kann der Einsatz jedoch problematisch sein.

Die dynamische Provisionierungsstrategie reagiert auf getätigte Service-Aufrufe, die an den ESB von der Workflow Engine weitergeleitet werden. D.h. ein eingehender Service-Aufruf ist der Auslöser für die Anwendung der Provisionierungslogik. Es ist klar, dass bei dieser Vorgehensweise die statische Strategie nicht ohne weiters realsiert werden kann, denn alle involvierten Services sind im Voraus vor dem ersten Service-Aufruf zu provisionieren. Auch für das Beenden der Service-Instanzen fehlt dem ESB die Information, wann die komplette Simulation zu Ende ist.

Die statische Provisionierungsstrategie ist also ausschließlich auf der ESB-Ebene nicht zu realisieren. Die benötigten Informationen können aber von anderen Infrastruktur-Komponenten zum ESB kommen. Z.B. die Workflow Engine könnte beim Deployment von einem Prozessmodell die Provisionierung im Voraus anstoßen. Genauso wäre möglich, das Beenden von betroffenen Service-Instanzen nach der Ausführung der letzten Prozessaktivität durchführen zu lassen. Auch das Modeling & Monitoring Tool besitzt die benötigten Informationen und kann ebenfalls für die Realsierung der statischen Provisionierungsstrategie eingesetzt werden.

Die Abbildung 5-5 verdeutlicht den Vorschlag, die Erweiterung vom Modeling & Monitoring Tool der Erweiterung der Workflow Engine vorzuziehen. Der Grund sind die im Abschnitt 3 beschriebenen Probleme, die eine Erweiterung einer existierenden Workflow Engine mit sich bringen kann.

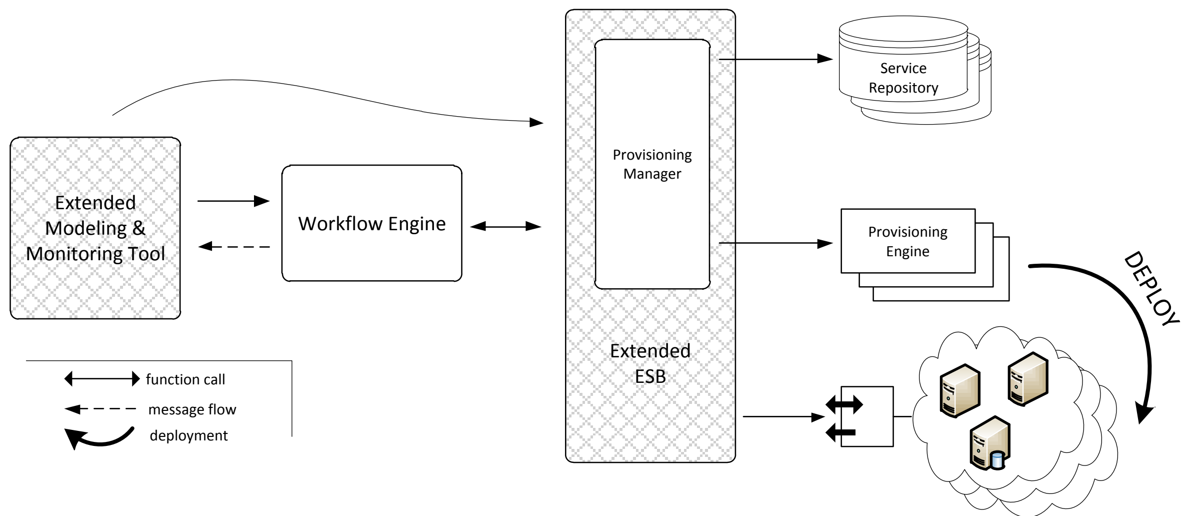

Abbildung 5-5: Realisierung von Provisionierungsstrategien

5.3 Visualisierung des Provisionierungsprozesses

Der beschriebene Architekuransatz für die Provisionierung von Simulationsservices in einer Cloud-Umgebung basiert auf einer Erweiterung der Workflow-Ausführungsumgebung um die Provisionierungslogik auf der ESB-Ebene sowie auf dem Einsatz von Provisioning Engine. Für den Benutzer sind die Provisionierungsaspekte transparent, daher kann die Modellierung und Ausführung von Simulationen weiterhin wie gewohnt durchgeführt werden.

Web Services, die zur Laufzeit eines Worfklows bereits laufen, können direkt aufgerufen werden können. Andere Simulationsservices müssen dagegen erst in einer Cloud-Umgebung provisioniert werden. Die Provisionierung eines Simulationsservices ist ein komplexer Vorgang, dessen Dauer von der Komplexität der benötigten Cloud-Infrastruktur abhängt. Zusätzlich kann die Impementierung der Provsioning Engine ein Faktor sein für Dauer des Provisionierungsprozesses.

Aus der Benutzersicht läuft die Provisionierung im Hintergrund ab. Der Benutzer sieht anhand vom Zustand einer Aktivität, dass diese gerade ausgeführt wird. Der Zustand wird von der Workflow Engine an den GUI propagiert und dargestellt. Die Workflow Engine leitet den entsprechenden Service-Aufruf an den ESB weiter und wartet auf die Antwort. Von dem laufenden Provisionierungsprozess bekommt die Workflow Engine nichts mit und kann daher nicht unterscheiden, ob zurzeit ein Service provisioniert wird oder eine Operation des Services bereits ausgeführt wird. Aus diesem Grund kann auch der GUI eine laufende Provisionierung hinter einem Service-Aufruf nicht erkennen.

Die Entwicklung von Simulationen zeichnet sich durch eine iterative und explorative Vorgehensweise aus. Dabei wechseln sich die Ausführung einzelner Schritte mit der Analyse der Ergebnisse und erneuter Ausführung ab. Bei der Analyse der Laufzeit der Simulationsschritte ist die Kenntnis über die im Hintergrund laufende Provisionierung von Bedeutung, denn sie kann die gemessenen Werte der Laufzeit eines Service-Aufrufes verfälschen. Auch bei Problemen mit der Ausführung von Service-Aufrufen durch die Workflow Engine muss der Wissenschaftler bei der Fehlersuche beachten, dass auch die Provisionierung als eine mögliche Ursache in Frage kommen kann.

Die Abbildung 5-5 deutet eine mögliche Lösung für beschriebene Probleme an. Das Modeling & Monitoring Tool kann erweitert werden um die auf dem ESB laufende Provisionierungsprozesse dem Benutzer in geeigneter Form darzustellen. Damit kann der Wissenschaftler bei Bedarf auch die Provisionierungsaspekte überwachen oder in Form einer Strategieauswahl beeinflussen.

6 Realisierung

Während bis jetzt die abstrakte Architektur für die on-demand Provisionierung der Workflow-Middleware und Simulationsservices behandelt wurde, geht es in diesem Kapitel um die Realisierung der Architektur. Dabei musste ein Teil der vorgestellten Architektur, der sich mit Provisionierung von Simulationsservices beschäftigt, prototypisch umgesetzt werden. Als Basis für die Realisierung wurde das aktuelle SimTech sWfMS verwendet, das im Abschnitt 6.1 detaillierter beschrieben wird. Danach wird im Abschnitt 6.2 die Erweiterung des SimTech sWfMS um die neuen Komponenten nach dem Architekturentwurf vorgestellt. Anschließend werden Realisierungsaspekte der jeweiligen Komponenten in den Abschnitten 6.3 bis 6.7 beschrieben.

6.1 SimTech sWfMS

Dieser Abschnitt gibt einen Überblick über den Ist-Zustand vom SimTech sWfMS, der als Basis für den praktischen Teil der Arbeit gilt. Der SimTech Prototyp wird kontinuierlich weiterentwickelt, daher beziehen sich folgende Informationen auf einen Entwicklungsstand, wie dieser zum Anfangszeitpunkt dieser Diplomarbeit war. Die Abbildung 6-1 zeigt die vorhandenen Komponenten und wie sie miteinander interagieren.

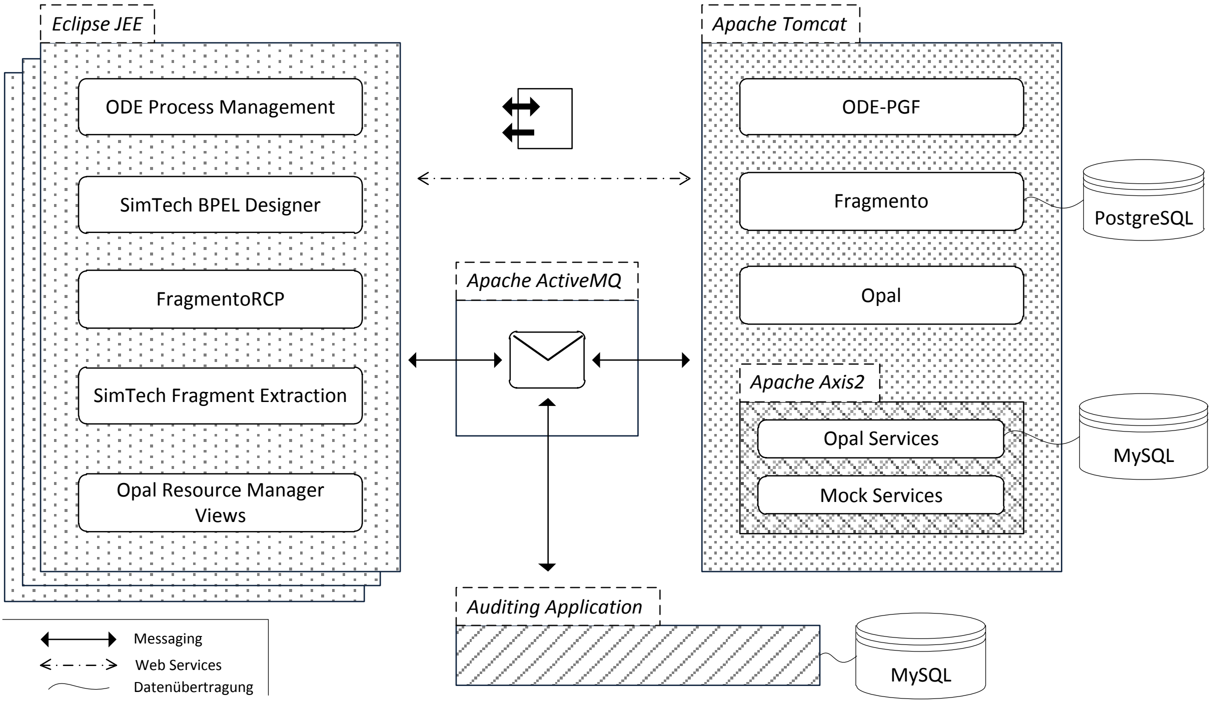

Abbildung 6-1: Architektur vom SimTech Prototyp

Der GUI ist die Java Entwicklungsumgebung Eclipse [28], erweitert durch zahlreiche SimTech Eclipse Plug-Ins wie den *SimTech BPEL Designer*, der *FragmentoRCP* oder der *ODE Process Management* Plug-In. Mehrere GUIs können parallel auf die Workflow-Middleware-Komponenten zugreifen, wie die Abbildung 6-1 verdeutlicht.

Apache Tomcat [29] Application Server wird als Laufzeitumgebung für die als Web-Anwendungen lauffähigen Workflow-Middleware-Komponenten benutzt. Dazu zählen die erweiterte Workflow Engine *ODE-PGF*, das Repository für Prozessfragmente *Fragmento*, die Festkörpersimulationsanwendung *Opal* sowie das *Apache Axis2* [30] mit den Simulationsservices.

Auditing Application

Schließlich ist die *SimTech Auditing Application*, die als Java Stand-Alone Anwendung mit einer Anbindung an die MySQL [31] Datenbank realisiert ist, für die Protokollierung und persistente Speicherung der Ereignisse der Workflow Engine zuständig. Sie ermöglicht den GUIs die Überwachung aller Prozessinstanzen, unabhängig davon, von welchem GUI die eine oder andere Prozessinstanz erstellt oder gestartet wurde.

Integration

Die Integration der Komponenten erfolgt teilweise direkt über die von den Komponenten angebotenen Web Service Schnittstellen und über den *Apache ActiveMQ* [32] Messaging Server. Bei dem Architektur-Entwurf wurde ein großer Wert auf die lose Kopplung der Komponenten gelegt, dadurch können die Komponenten unabhängig voneinander verteilt betrieben werden. Übersichtshalber zeigt die Abbildung 6-1 mehrere Komponenten innerhalb einer Instanz von Apache Tomcat.

6.1.1 ODE-PGF

Mit dem Apache ODE Pluggable Framework (ODE-PGF) [33] Workflow-Managementsystem lässt sich nicht nur das Standard-Verhalten einer BPEL-Engine abbilden, sondern auch darüber hinausgehende Sachverhalte. Das ODE-PGF entstand im Rahmen einer Diplomarbeit [34]. Dabei wurde die Open Source BPEL-Engine Apache Orchestration Director Engine (Apache ODE) [35] in der Version 1.1.1 erweitert. Die Erweiterung hatte zur Folge, dass Ereignisse des BPEL-Event-Modells [36] nach außen sichtbar gemacht wurden. Z.B. sind es Ereignisse, die beim Deployment oder bei der Ausführung von Prozessinstanzen bzw. Aktivitäten auftreten. Das ermöglicht nun Erweiterungen zu bauen, die durch die Interpretation der auftretenden Ereignisse zusätzliche Funktionalität abbilden können. Die BPEL-Engine muss nicht modifiziert werden.

ODE Process Management

Durch das *ODE Process Management* Plug-in können auf dem ODE-PGF deployte Prozesse verwaltet werden, wie die Abbildung 6-2 zeigt. Der Benutzer kann eine Liste aller aktuell deployten Prozessmodelle angezeigen lassen und Operationen darauf ausführen, z.B. können einzelne Prozessmodelle gezielt mit „undeploy“-Operation entfernt werden.

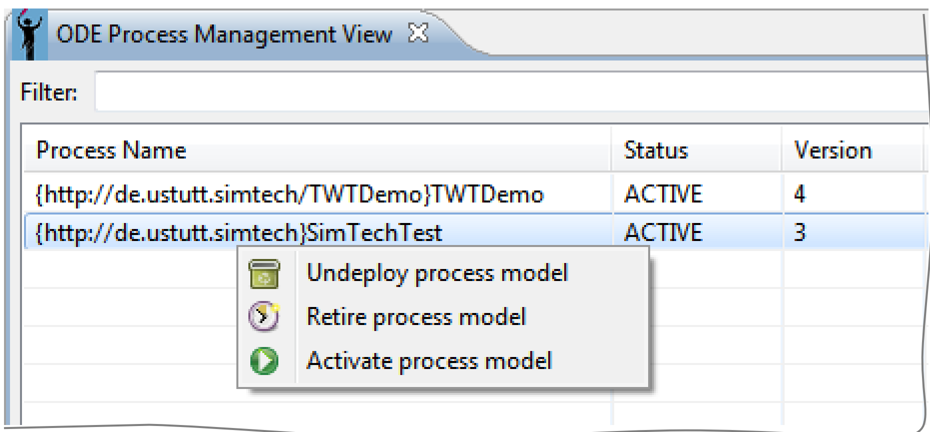

Abbildung 6-2: ODE Process Management View

Der Benutzer muss sich nicht um das Deployment der Prozessmodelle kümmern, dies geschieht automatisch im Hintergrund beim Starten einer Prozessinstanz. An dieser Stelle werden die Informationen darüber benötigt, ob das entsprechende Prozessmodell bereits deployt ist oder nicht. Ist das nicht der Fall, wird das Prozessmodell zuerst auf dem ODE-PGF deployt.

Die Abbildung 6-3 zeigt die Kommunikation zwischen beiden Komponenten, die über die von ODE-PGF bereitgestellte Web Service Schnittstelle stattfindet.

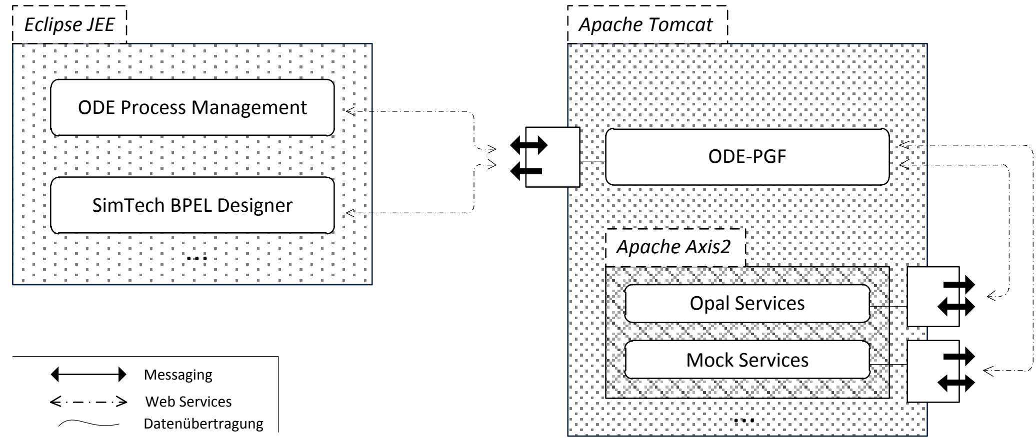

Abbildung 6-3: Integration von ODE-PGF (Web Services)

SimTech BPEL Designer

Der SimTech BPEL Designer ist das Werkzeug für Modellierung, Überwachung und Steuerung der Simulationsworkflows. Als Basis wurde der Eclipse BPEL Designer [37] verwendet – ein Modellierungswerkzeug für BPEL. Für die Nutzung in wissenschaftlichen Simulationen sind einige Erweiterungen realisiert worden wie z.B. die Steuerung der Ausführung einer Prozessinstanz – Abbildung 6-4 – mit den Operationen *Start*, *Suspend*, *Resume*, *Terminate, Iterate* und *Reexecute*. Die ersten 4 der genannten Operationen sind selbsterklärend, mit *Iterate* lassen sich Teile einer Prozessinstanz wiederholt ausführen. *Reexecute* kann ebenfalls zur wiederholten Ausführung von Teilen einer Prozessinstanz benutzt werden. Der Unterschied liegt darin, dass bei *Reexecute* alle Änderungen, die ab der ausgewählten Aktivität durchgeführt wurde, rückgängig gemacht werden.

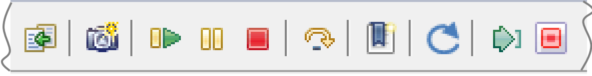

Abbildung 6-4: SimTech BPEL Designer (Steuerung)

Im Modellierungseditor lassen sich der Gesamtzustand der Ausführung sowie die Zustände jeder einzelnen Aktivität überwachen. Wie in der Abbildung 6-5 dargestellt, wird der Gesamtzustand textuell angezeigt, während die Aktivitäten abhängig vom Zustand mit unterschiedlichen Farben dargestellt werden.

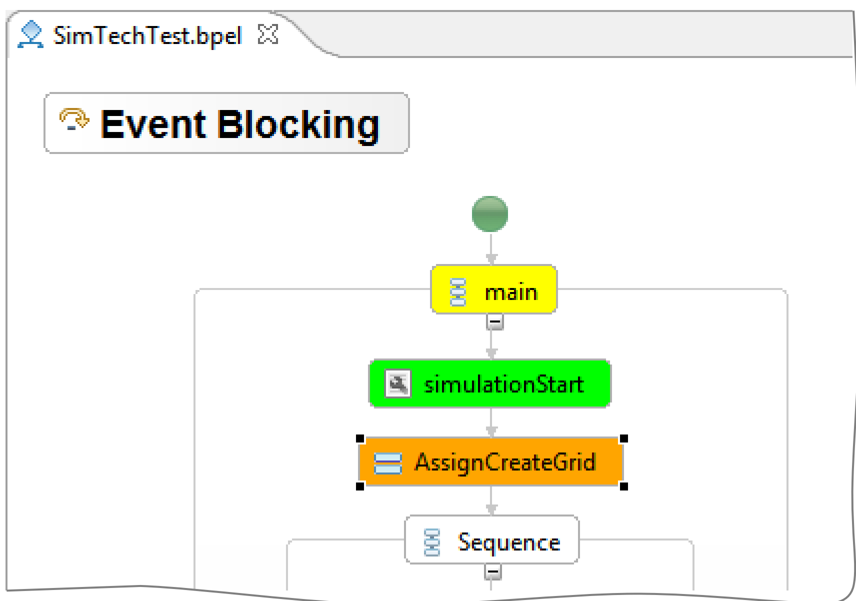

Abbildung 6-5: SimTech BPEL Designer (Überwachung)

Die Zuordnung der Aktivitätsfarbe zu einem Aktivitätszustand ist in der Tabelle 6-1 aufgeführt.

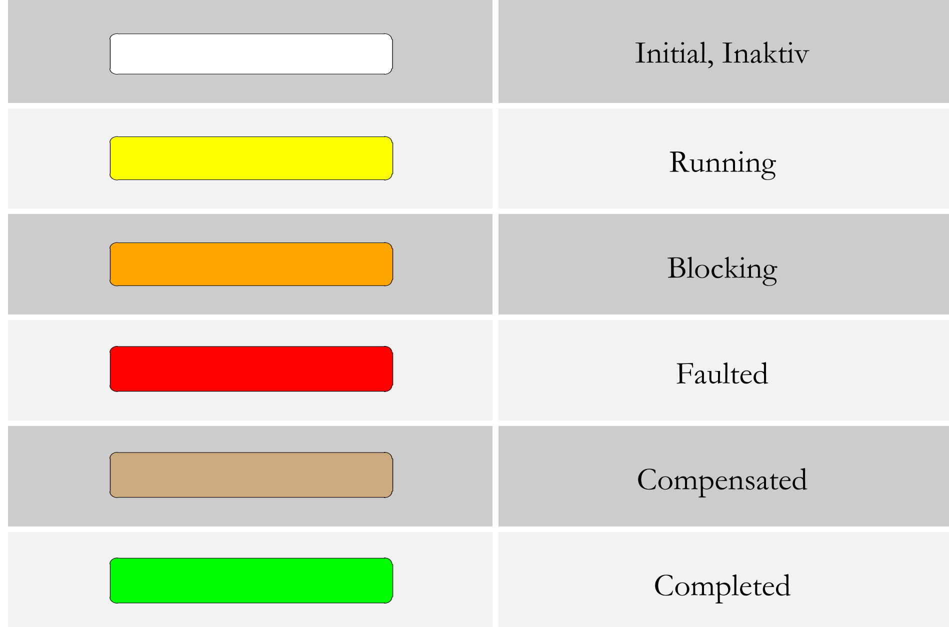

Tabelle 6-1: Farbliche Zuordnung der Aktivitätszustände

Die Ausführung einer Prozessinstanz kann durch Setzen von Breakpoints unterbrochen werden. Die Aktivität befindet sich dann im Zustand *Blocking* und die auf dieser Weise angehaltene Simulation (bzw. ein Simulationszweig) kann näher untersucht werden. Breakpoints ermöglichen damit das Debugging von laufenden Simulationen. Wie die Abbildung 6-6 zeigt, Breakpoints werden zu einzelnen Aktivitäten definiert. Dabei kann die Auslösung eines Breakpoints an bestimmte Bedingungen gekoppelt sein, z.B.wenn die Aktivität fehlschlägt.

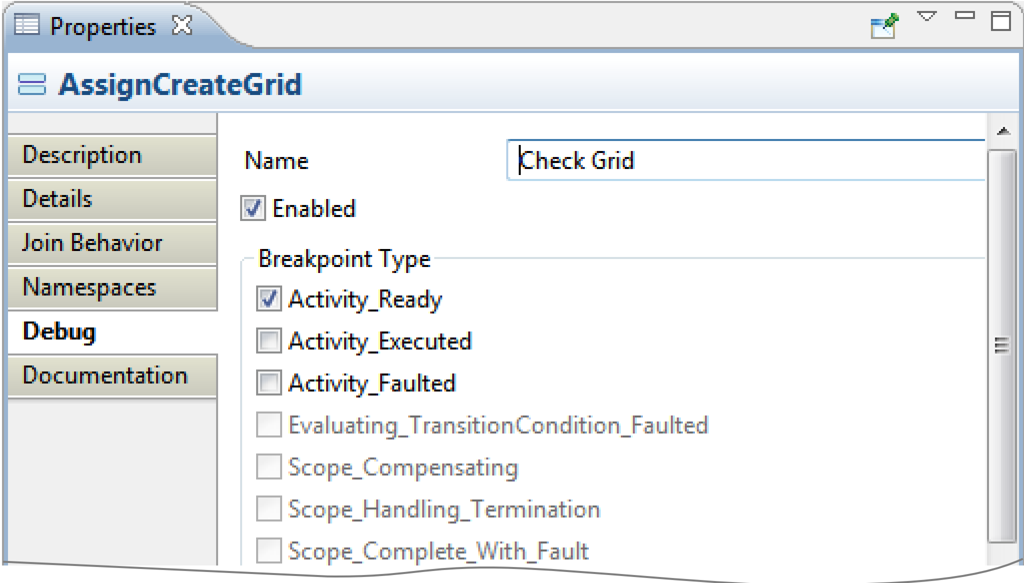

Abbildung 6-6: SimTech BPEL Designer (Breakpoints)

Für die Integration von ODE-PGF mit dem BPEL Designer wird die Web Service Schnittstelle von ODE-PGF sowie das Messaging über den *Apache ActiveMQ* Messaging Server benutzt.

Die Abbildung 6-7 skizziert die Integration über Messaging. Dabei registriert sich der BPEL Designer beim Topic (org.apache.ode.events) für die Ereignisse des ODE-PGFs um Informationen für die Überwachung der Ausführung zu bekomment. Zum Setzen von Breakpoints versendet der BPEL Designer eine Nachricht an die entsprechende Queue (org.simtech.ode.in). Die Antwort erfolgt dann ebenfalls über eine weitere temporäre Queue (punktierte Queue in der Abbildung).

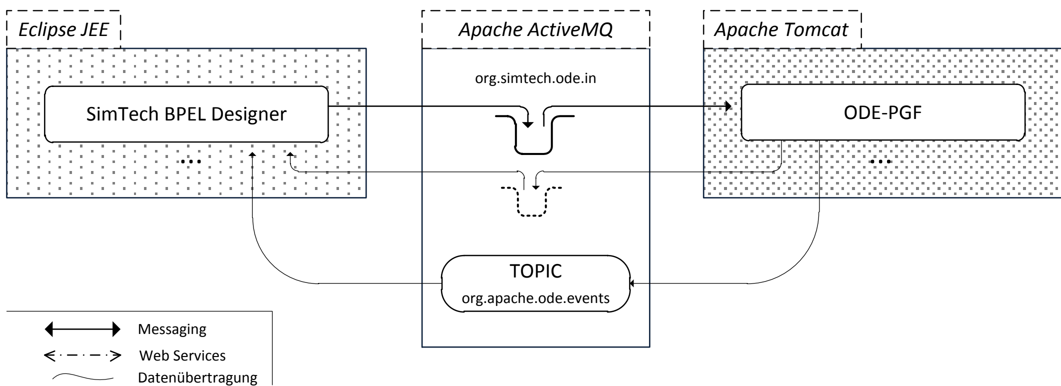

Abbildung 6-7: Integration von ODE-PGF (Messaging)

6.1.2 Fragmento

Fragmento [38] ist ein am IAAS entwickeltes Repository für die versionierte Verwaltung von Prozessfragmenten [39]. Dabei ist ein Prozessfragment nach der Definition von [40] ein zusammenhängender Prozessgraph, der aus Aktivitäten, Platzhaltern der Aktivitäten und Kontrollkanten besteht. Dabei kann er unvollständig, inkonsistent, nicht direkt ausführbar und teilweise undefiniert sein.

Neben der bereits erwähnten Versionierung bietet Fragmento weiterhin eine Reihe nützlicher Features. Z.B. durch Locks sollen mögliche Inkonsistenzen beim gleichzeitigen Bearbeiten eines Fragments vermieden werden. Desweiteren können Beziehungen zwischen Artefakten sowie Validierungen definiert werden.

FragmentoRCP Fragmento bietet neben dem Repository selbst eine Verwaltungsoberfläche, die man für das Laden oder Entfernen einzelner Prozessgrafmente nutzen kann, allerdings hat Fragmento keine integrierte Modellierungsumgebung für die

Fragmente. Für diesen Zweck ist eine WebService-Schnittstelle vorgesehen, die von Modellierungswerkzeugen verwendet werden kann um auf die Fragmente aus dem Repository zuzugreifen und sie in die Prozessmodellierung einzubeziehen. *FragmentoRCP* [41] ist ein Eclipse Plug-In, der die Integration von Fragmento in das Fromtend realisiert und dadurch Prozessfragmente aus dem Repository dem *SimTech BPEL Designer* zur Verfügung stellt. Dieser Zusammenhang wird durch die Abbildung 6-8 verdeutlicht.

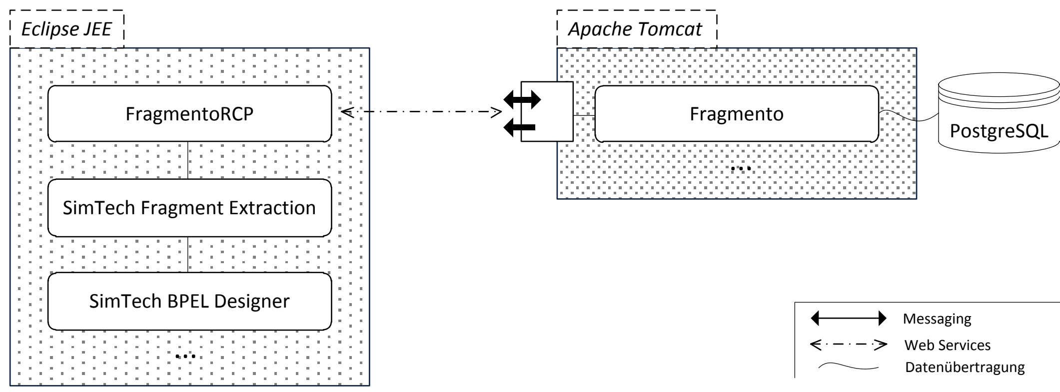

Abbildung 6-8: Integration von Fragmento

SimTech Fragment Extraction

Die *SimTech Fragment Extraction* Komponente bietet wichtige Funktionalität für die Extraktion von Prozessfragmenten aus modellierten Prozessmodellen an. Die extrahierten Prozessfragmente können über den *FragmentoRCP* im Fragmento Repository abgelegt werden und dadurch für andere sichtbar bzw. wiederverwendbar gemacht werden. Es ist aber auch möglich die Ergebnisse der Extraktion direkt lokal bei der Modellierung zu verwenden.

6.1.3 Opal

Bei Opal handelt es sich um eine Simulationsanwendung für Festkörpersimulationen, die im Rahmen der studentischen Arbeit von S. Hotta [15] entstanden ist. Dabei wurde eine bestehende Fortran77-Simulationsanwendung in ein Simulationsworkflow eingebunden. Die einzelnen Simulationsschritte wurden dazu als Java Wrapper für native Fortran77-Anwendungen realisiert und als Web Services zur Verfügung gestellt. Die Abbildung 6-9 skizziert die im SimTech Prototyp vorhandenen Opal-Komponenten und deren Integration.

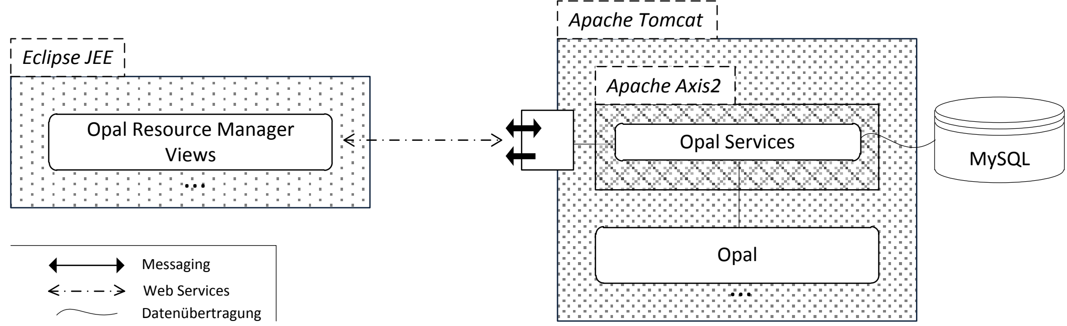

Abbildung 6-9: Integration von Opal

Da Opal im Rahmen dieser Diplomarbeit nicht behandelt wird, verzichtet der Autor an dieser Stelle auf eine ausführliche Beschreibung von Opal und verweist den interessierten Leser auf die Arbeit [42], die als Basis für die Opal-Realisierung genommen wurde und Festkörpersimulationen ausführlich behandelt.

6.1.4 Auditing Application

Wie im Abschnitt 4.1.1 bereits angedeutet, hat die Erweiterung von Apache ODE um den Pluggable Framework (PGF) zum ODE-PGF signifikante Vorteile gegenüber dem Standard-Produkt. Die Weiterleitung der internen Events nach außen ermöglicht die Erstellung von Erweiterungen, die durch die Interpretation der auftretenden Ereignisse zusätzliche Funktionalität abbilden.

Auch die SimTech Auditing Application nutzt diese Möglichkeiten für die Protokollierung und persistente Speicherung der Ereignisse von ODE-PGF.

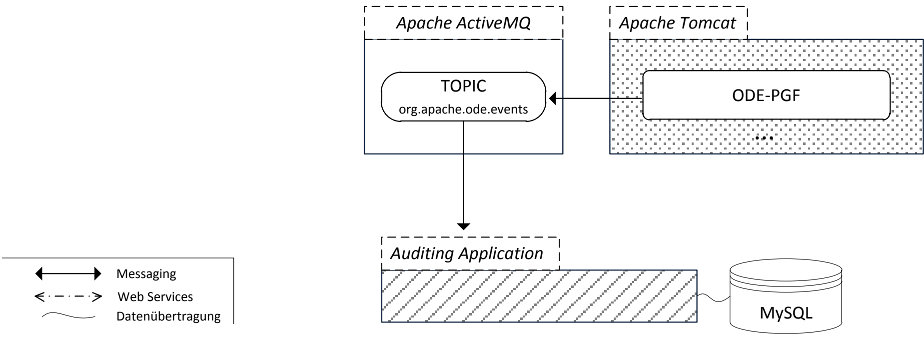

Abbildung 6-10: Integration von Auditing Application (ODE-PGF)

Die emittierten Events werden auf dem dafür vorgesehenen Topic (org.apache.ode.events) im Messaging Server durch ODE-PGF publiziert. Die Auditing Application ist als Subscriber dieses Topics automatisch über die Er-

eignisse informiert und legt diese Informationen in einer MySQL-Datenbank persistent ab.

Der SimTech BPEL Designer fragt die gespeicherten Daten über die entsprechende Queue (org.simtech.ode.eclipse) bei der Auditing Application an und nutzt sie für die Überwachung. Die Daten werden über eine temporäre Queue übertragen, die der SimTech BPEL Designer speziell dafür initiiert.

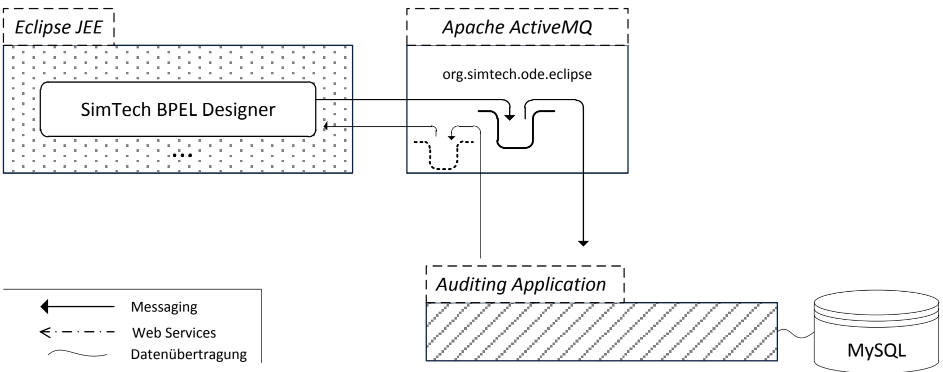

Abbildung 6-11: Integration von Auditing Application (BPEL Designer)

6.2 Erweiterung vom SimTech sWfMS

Basierend auf der im Abschnitt 4 vorgestellten Architektur für on-demand Provisionierung von Workflow-Middleware und Simulationsservices wurde das SimTech sWfMS um neue Komponenten erweitert. Das Ziel der Erweiterung war es die on-demand Provisionierung der Simulationsservices in einer Cloud-Umgebung zur Laufzeit eines Workflows zu realisieren. Die Verbesserungen der Architektur aus dem Kapitel 5 wurden ebenfalls bei der Realisierung berücksichtigt.

Wie die Abbildung 6-12 zeigt, sind mit Service Registry, Service Repository, Provisioning Engine, Provisioning Manager und Enterprise Service Bus fünf neue Komponenten dazugekommen.

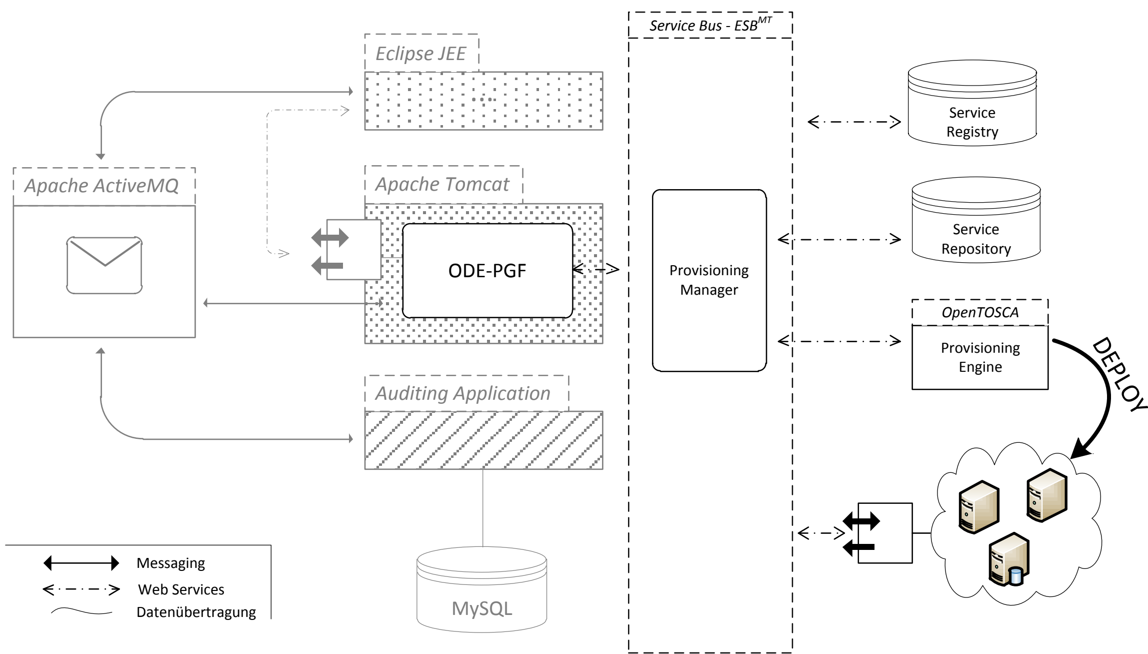

Abbildung 6-12: Erweiterung des SimTech sWfMS

Das **Service Repository** dient als Ablageort für CSAR-Dateien, die Simulationsservices mit TOSCA beschreiben. CSAR-Dateien können in das Service Repository hochgeladen und bei der Provisionierung verwendet werden, um daraus in einer Cloud-Umgebung Service-Instanzen zu erzeugen.

Die **Service Registry** stellt ein Verzeichnis für Simulationsservices da. Services können in der Service Registry an- und abgemeldet werden. Registrierte Services können für die Modellierung und Ausführung von Simulationen verwendet werden. Die Service Registry hält Metadaten über die registrierten Services sowie bestimmte Runtime-Daten für *Not Provisioned Services*. Die Runtime-Daten, z.B. Informationen über erzeugte Service-Instanzen oder über aktive

Service-Aufrufe, werden von dem Provisionierungsprozess auf der ESB-Ebene benötigt.

Als **Provisioning Engine** wird OpenTOSCA [27] verwendet– eine Laufzeitumgebung für TOSCA. Damit lassen als CSAR-Dateien beschriebene Cloud-Services verarbeiten um daraus Service-Instanzen in einer Cloud-Umgebung zu erstellen. Auch andere Management-Aspekte von Services werden dadurch abgebildet, wie z.B. das Beenden einer Service-Instanz.

Der **Enterprise Service Bus** ist die zentrale Komponente der Architektur für die Provisionierung von Simulationsservices. Der ESB integriert die beteiligten Komponenten und steuert die Provisionierung von Services über einen Provisionierungsprozess. Als ESB wird ein am IAAS entwickeltes Produkt ESB^{MT} eingesetzt.

Mit dem **Provisioning Manager** wurde eine neue Komponente entworfen und realisiert, die den Zugriff auf unterschiedliche Provisioning Engines und Service Repositories auf eine einheitliche Art und Weise regelt.

6.3 Service Repository

Ein Service Repository sollte im Rahmen dieser Arbeit nicht ausgearbeitet und realisiert werden. Die Komponente wird aber für die Realisierung der Architektur für dynamische Provisionierung von Simulationsservices benötigt. Für diesen Zweck wurde an dieser Stelle ein MockServiceRepository als ein RESTful Web Service erstellt und als Platzhalter für die zukünftige Lösung in den SimTech sWfMS eingebracht.

Aus dem genannten Grund wurde die Schnittstelle vom MockServiceRepository sehr schlicht gehalten und erlaubt nach außen eine einzige Operation für den Download der im Service Repository abgelegten CSAR-Dateien. Der Zugriff erfolgt über einen Identifikator (Dateiname inklusive Dateiendung) mit der folgenden HTTP-GET-Anfrage.

HTTP-Request

Headers:

```
GET /repositoryapi/csars/{id}

Host: localhost:1488

Content-Type: text/plain
```

Body:

Listing 6-1: Download einer CSAR-Datei - HTTP-Request

6.4 Service Registry

Der praktische Teil dieser Arbeit basiert auf der Annahme, dass die Service Registry gegeben ist, da die Komponente im Rahmen einer gesonderten studentischen Arbeit behandelt werden soll. Für die Realisierung des Provisionierungsprozesses auf der ESB-Ebene wird die Service Registry benötigt, daher wird dafür eine Platzhalter-Komponente (wie auch im Fall des Service Repository) verwendet.

Für den Provisionierungsprozess wird von der Service Registry folgende Funktionalität erwartet: (1) Die Service-Metadaten können geholt werden. (2) Für Services, die vor der Nutzung provisioniert werden müssen (*Not Provisioned Services*), enthalten Service-Metadaten eine Referenz auf die entsprechende CSAR-Datei im Service Repository. (3) Für *Not Provisioned Services* können Runtime-Daten abgefragt und geändert werden. Als Runtime-Daten werden unter anderem Informationen über die erzeugten Service-Instanzen sowie über die Anzahl der aktiven Service-Aufrufe gespeichert.

Diese Platzhalter-Komponente wurde als ein SOAP Web Service realisiert.

6.5 Enterprise Service Bus

Als eine der Vorgaben für den praktischen Teil der Arbeit war der Einsatz von ESB^{MT} [9] als Enterprise Service Bus. Der ESB^{MT} basiert auf dem ESB-Produkt ApacheServiceMix [43], welcher in Rahmen von mehreren studentischen Arbeiten erweitert wurde. Das Augenmerk bei der Erweiterung lag an der Mandantenfähigkeit (Multi-tenancy) von ESBs. Eine der Anforderungen an den erweiterten ESB wurde die Rückwärtskompatibilität der Lösung definiert [9]. D.h auch Anwendungen, die nicht multi-tenant-fähig sind, sollen weiterhin ohne Einschränkungen auf dem ESB^{MT} lauffähig sein.

Der Aspekt der Mandantenfähigkeit der Architektur für die Provisionierung von Simulationsservices wurde im Rahmen dieser Arbeit nicht behandelt. Die Mandantenfähigkeit wird jedoch ein Thema von weiterführenden Arbeiten sein, daher wird in diesem Abschnitt der Nachrichtenaustausch in einer Multi-tenant-fähigen Umgebung kurz erläutert.

Alle Nachrichten, die an den ESB, vom ESB oder innerhalb vom EBS verschickt werden, müssen eindeutig einem Mandanten und einem Benutzer zugeordnet werden. Dies wird erreicht durch den *TenantContext*, der in jeder Nachricht mitübertragen wird. Das Listing 6-2 zeigt beispielhaft den Aufbau von einem TenantContext. Der Manadant wird über die `tenantId` und der Benutzer über die `userId` identifiziert, beides sind eindeutige globale Werte. Zusätzlich können optionale Key-Value-Paare angegeben werden, wie z.B. die Email-Adresse vom Benutzer.

```
<tenantContext>
  <tenantId>16c2025386054b679001935c50c8b707</tenantId>
  <userId>dc0b71dd4c994efb964bbc30efd552cc</userId>

  <optionalEntry>
    <key>tenantName</key>
    <value>Example Inc.</value>
  </optionalEntry>

  <optionalEntry>
    <key>userEmailAddress</key>
    <value>user@example.org</value>
  </optionalEntry>

</tenantContext>
```

Listing 6-2: Beispiel für TenantContext (vollständig) [44]

6.5.1 Prozess für Service-Provisionierung

Im Abschnitt 4.6 wurde der unten abgebildete Prozess für die Provisionierung von Simulationsservices vorgestellt, der das Kernstück der Provisionierungsfunktionalität bildet.

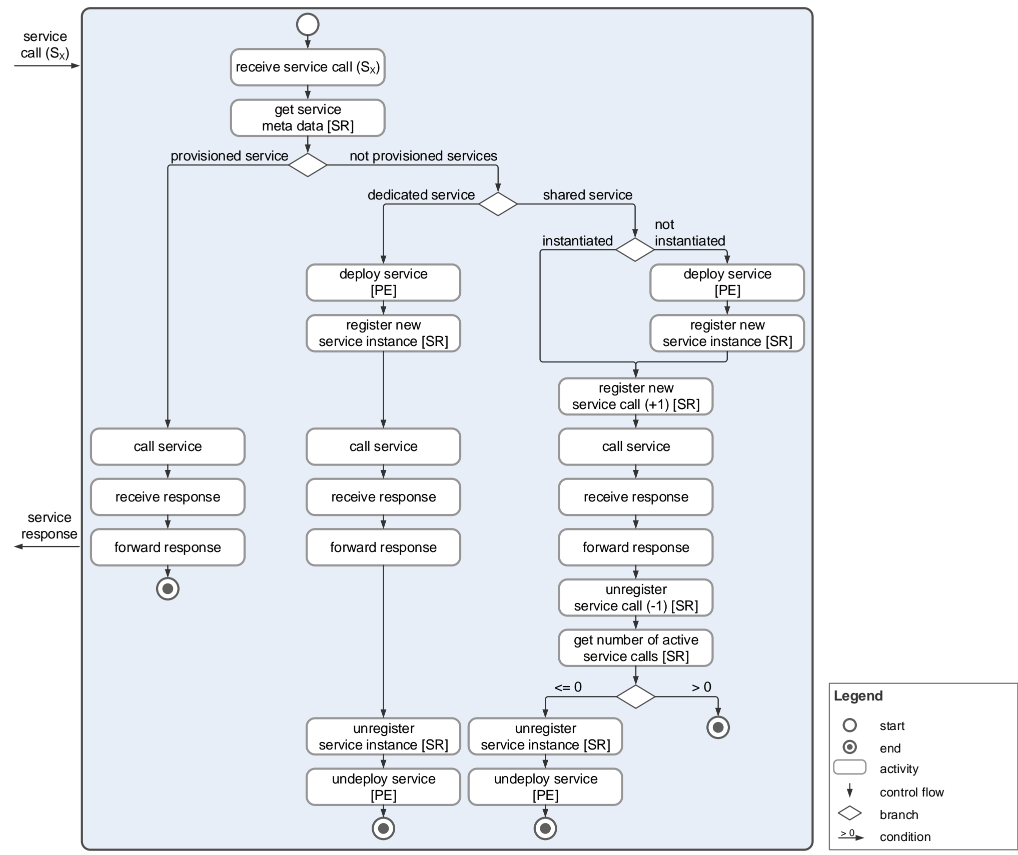

Abbildung 6-13: Prozess für die Service-Provisionierung [11]

Für die Realisierung dieses Prozesses im ESB^{MT} standen mehrere Alternativen zur Auswahl. Zum einen steht im ESB^{MT} das Integrationsframework Apache Camel [45] zur Verfügung. Damit können Routing- und Transformationsregeln für den Nachrichtenfluss definiert werden, basierend auf den bekannten Enterprise Integration Patterns. Die Regeln können in mehreren domänenspezifischen Sprachen wie Java oder XML-basiert definiert werden.

Eine andere Möglichkeit den Provisionierungsprozess abzubilden, wäre die Nutzung der eingebauten Workflow Engine und die Realisierung der Provisionierungslogik als einen BPEL-Prozess. Es ist davon ausgehen, dass die aktuelle Provisionierungslogik in Zukunft erweitert und optimiert werden muss, daher war es wichtig eine Realisierungsvariante auszuwählen, die flexible und schnelle Anpassungen der Logik erlaubt. Aus diesem Grund wurde die Variante mit dem BPEL-Prozess vorgezogen und realisiert.

6.6 ODE-PGF

Durch die Hereinnahme vom ESB in das SimTech sWfMS musste auch die verwendete Workflow Engine ODE-PGF angepasst werden. Bis jetzt wurden entsprechende Simulationsservices vom ODE-PGF direkt aufgerufen. Bedingt durch die Erweiterung der Architekur müssen die Simulationsservices über den ESB aufgerufen werden, damit die auf der ESB-Ebene angesiedelte Provisionierungslogik greifen kann.

Die Weiterleitung der Web Service Aufrufe an den ESB wurde durch die Konfiguration von ODE-PGF realisiert. In ODE-PGF können externe SOAP- oder HTTP-Endpunkte durch spezielle Property-Dateien angepasst werden. Die Dateien müssen die Endung `*.endpoint` haben und in dem Verzeichnis für die globale ODE-PGF Konfiguration liegen (vordefiniertes Verzeichnis: `ode/WEB-INF/conf`). Die Dateien werden dynamisch geladen und zur Laufzeit aktualisiert (etwa jede 30 Sekunden) [46].

Durch solche Property-Dateien können Eigenschaften von Endpunkten überschrieben werden. Die Tabelle 6-2 zeigt einen Auszug über ausgewählte Endpunkt-Eigenschaften. Die komplette Tabelle kann auf [46] eingesehen werden.

Context	Property name	Accepted values	Description/Notes
Outbound Services	address	a URL	overrides the soap:address or http:address. The order of precedence is: process, property files, wsdl.
	http.request.chunk	true/false	This will enable/disable chunking support. Will not apply to http-bound services TBD
	http.protocol.content-charset	a string	
	http.default-headers.{your-header}		You must define one property per header, prefixed with 'http.default-headers'. These values will be appended to any previous value already set for a given header.
	http.connection.timeout	an int	timeout in milliseconds until a connection is etablished
	http.socket.timeout	an int	timeout in milliseconds for waiting for data
	http.protocol.max-redirects	an int	the maximum number of redirects to be followed
	ws-adddressing.headers	true(default)/false	Enable/disable the WS-Addressing headers for outgoing soap requests

Tabelle 6-2: Endpunkt-Eigenschaften (Auszug) [46]

Für die Weiterleitung der Web Service Aufrufe an den ESB kann die `address`-Eigenschaft benutzt werden. Das Listing 6-3 zeigt bespielhaft einen Auszug einer Endpunkt-Property-Datei mit der Weiterleitung von Aufrufen eines bestimmten Web Services an einen EBS-Endpunkt, der HTTP-Anfragen konsumieren und verarbeiten kann.

```
alias.ustutt=http://service.iaas.ustutt.de

ustutt.ServiceMock.ode.address=http://localhost:8079/esbhttpc
onsumer/
```

Listing 6-3: Beispiel für ODE-PGF Endpunkt-Konfiguration

6.7 Provisioning Manager

Der Provisioning Manager hat die Aufgabe das Deployment bzw. das Undeployment von Simulationsservices zu vereinheitlichen und unabhängig vom Einsatz konkreter Provisioning Engines und Service Repositories zu machen. Der Provisioning Manager bietet nach außen eine stabile Schnittstelle für die Provisionierung von Services an und kapselt die Provisionierungsabläufe sowie die Kommunikationsdetails eingesetzter Provisioning Engines.

Die Abbildung 6-14 zeigt die Architektur vom Provisioning Manager. Demnach besteht er aus einer Menge von Repository Adapter, die Zugriffe auf unterschiedliche Service Repositories kapseln und von der Control-Komponente über ein einheitliches `Repository Interface` verwendet werden.

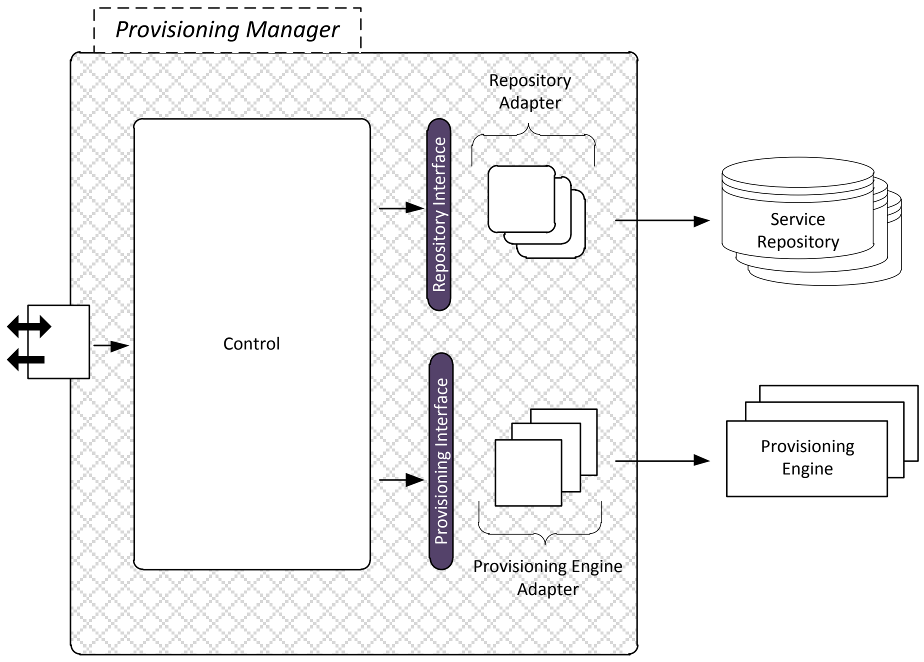

Abbildung 6-14: Architektur vom Provisioning Manager

Analog dazu bilden Provisioning Engine Adapter, auf die über das `Provisioning Interface` zugegriffen wird, eine Basis für einheitliche Verwendung unterschiedlicher Provisioning Engines. Die Control-Komponente steuert die Abläufe innerhalb vom Provisioning Manager. Die beiden Interfaces sowie die Zusammenhänge zwischen den Adapter werden in den Abschnitten 6.7.1 und 6.7.2 dargestellt. Der Provisioning Manager wurde als ein SOAP Web Service mit zunächst 2 Operationen (`deployService` und `undeployService`) realisiert.

Durch die Entkopplung der Provisioning Engine vom Service Repository sieht ein Ablauf der `deployService`-Operation nach der Abbildung 6-15 folgendermaßen aus:

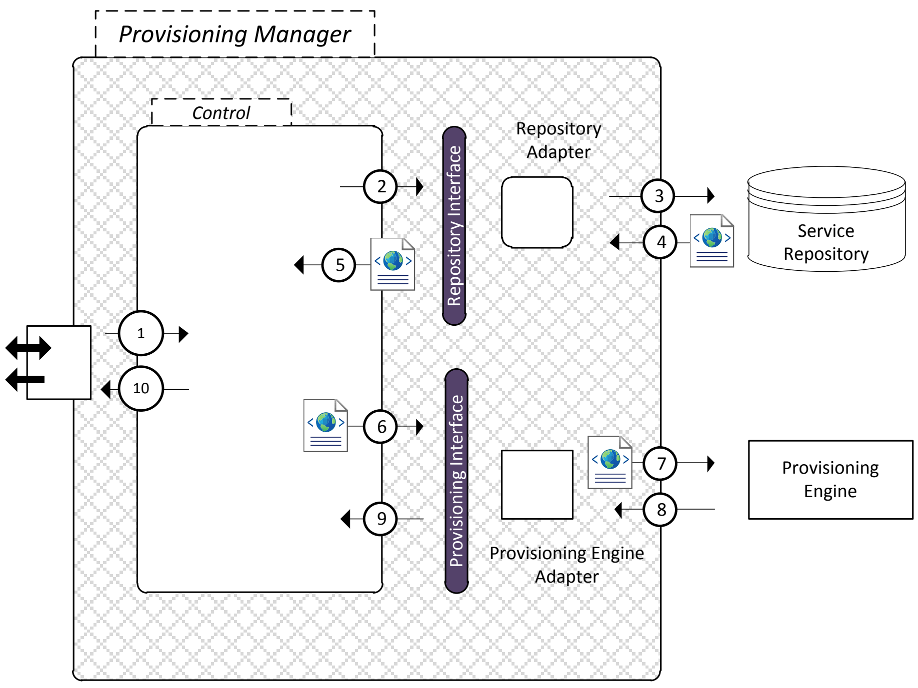

Abbildung 6-15: Ablauf vom Service Deployment

Die Control-Komponente nimmt einen Aufruf der deployService-Operation entgegen (1) und holt über einen entsprechenden Repository Adapater die CSAR-Datei zu sich (5). Jetzt kann die CSAR-Datei bei Bedarf einer Verarbeitung unterzogen werden. Im einfachsten Fall wird die CSAR-Datei an die Provisioning Engine über den entsprechenden Provisioning Engine Adapter übergeben und mit der Erzeugung einer neuen Service-Instanz beauftragt (6).

Bei diesem Ablauf sieht man, dass die CSAR-Datei zweimal übertragen werden muss, ein Mal vom Service Repository zum Provisioning Manager und das zweite Mal vom Provisioning Manager zur Provisioning Engine. Dieses Verhalten kann optimiert werden wie die Abbildung 6-16 zeigt. Die Voraussetzung dafür ist, dass die eingesetzte Provisioning Engine den direkten Download von CSAR-Dateien aus dem Service Repository unterstützen muss. Die CSAR-Datei wird in diesem Fall nur einmal übertragen, allerdings kann die CSAR-Datei vom Provisioning Manager nicht mehr zwischenverarbeitet werden.

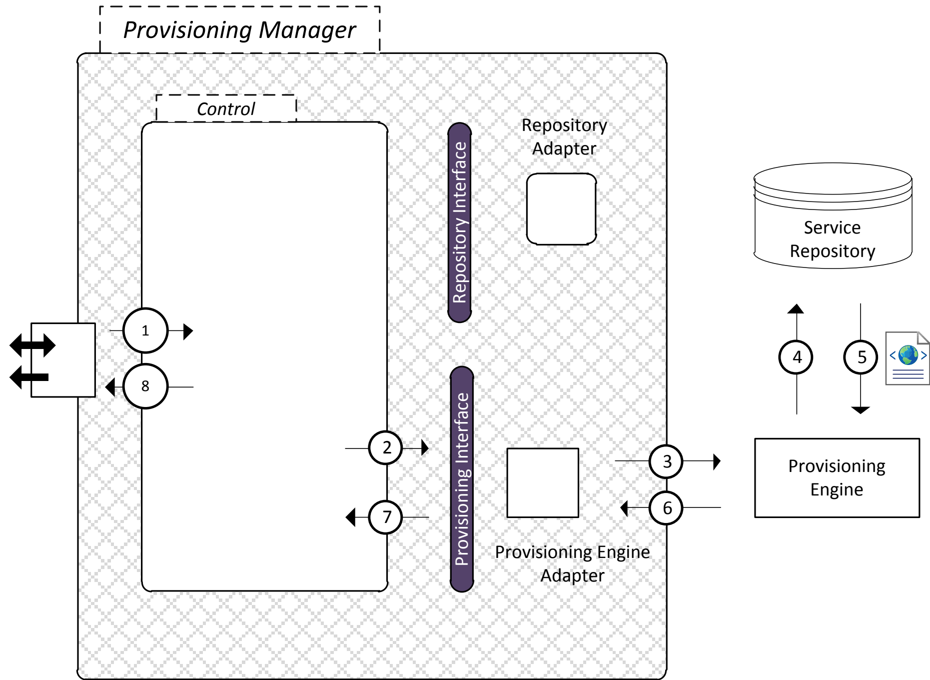

Abbildung 6-16: Ablauf vom Service Deployment (direkter Download)

6.7.1 Adapter für MockServiceRepository

Repository Adapter im Provisioning Manager werden dazu verwendet um Zugriff auf konkrete Service Repositories zu vereinheitlichen. Ein Repository Adapter kapselt die Kommunikation und die spezifische Zugriffslogik. Von der Control-Komponente können damit alle Repository Adapter über das Repository Interface angesprochen werden. Als Service Repository wurde bei der Realisierung das MockServiceRepository verwendet. Dazu wurde der entsprechende Adapter geschrieben. In der Abbildung 6-17 wird das Interface für den Zugriff auf unterschiedliche Repositories abgebildet, sowie der Zusammenhang mit dem MockServiceRepositoryAdapter.

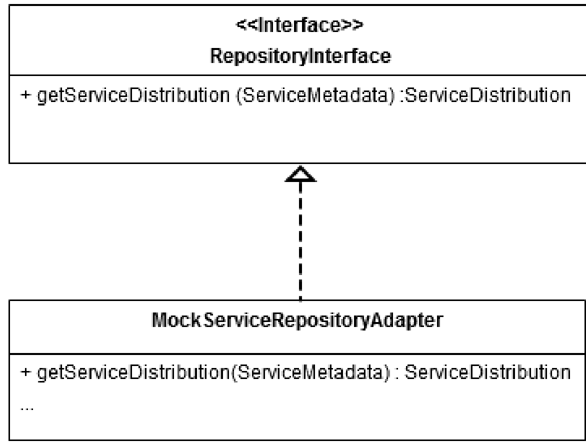

Abbildung 6-17: Adapter für MockServiceRepository

6.7.2 Adapter für OpenTosca

Die Einbindung konkreter Provisioning Engines in den Provisioning Manager erfolgt über die Realisierung von Provisioning Engine Adapter. Ein Provisioning Engine Adapter implementiert einerseits das Provisioning Interface und kapselt damit die Zugriffsdetails einer konkreten Provisioning Engine.

Um OpenTOSCA als Provisioning Engine nutzen zu können, wurde im Provisioning Manager ein Adapter für OpenTOSCA realisiert. Die Abbildung 6-18 zeigt anhand von einem vereinfachten UML-Klassendiagramm den Zusammenhang zwischen dem OpenToscaAdapter und dem Provisioning Interface.

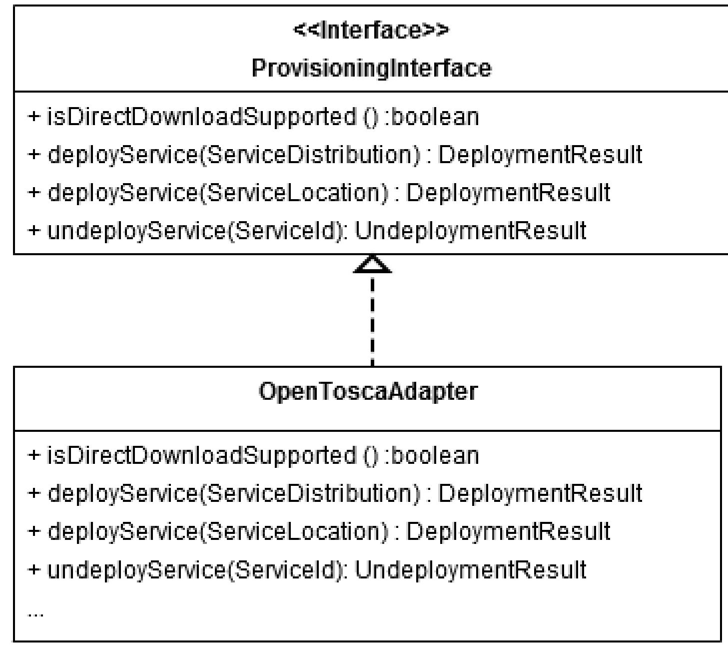

Abbildung 6-18: Adapter für OpenTosca

Die Abbildung 6-19 zeigt exemplarisch den Ablauf für die Realisierung der deployService-Methode durch den OpenToscaAdapter. Um die Übesichtlichkeit zu verbessern, wurden in der Abbildung Überprüfungen sowie die Fehlerbehandlung weggelassen.

Der OpenToscaAdapter nutzt für die Realisierung der Provisioning Interface-Methoden die ContainerAPI von OpenTOSCA, deren ausgewählte Operationen im nächsten Abschnitt ausführlich beschrieben sind. Der Ablauf für die Realisierung von deployService- oder undeployService-Methoden orientiert sich stark an die Gegebenheiten der verwendeter Provisioning Engine.

Im Falle von OpenTOSCA müssen eine Reihe von Schritten in fest definierter Reihenfolge durchgeführt werden, wie die Abbildung 6-18 zeigt. Demnach wird als erstes die CSAR-Datei zum OpenTOSCA Container hochgeladen. Danach kann die CSAR-Datei in mehreren Schritten verarbeitet werden. Die Operationen von OpenTOSCA ContainerAPI werden durch Versenden von HTTP-Anfragen ausgeführt. Nach jeder Verarbeitungsoperation, wie das Up-

load der CSAR-Datei oder die Verarbeitung der TOSCA-Dokumente läuft der jeweilige Schritt innerhalb von OpenTOSA ab. Eine erfolgreiche Ausführung einer Operation wird durch das Setzen des Zustandes für den gesamten Deployment-Prozess signalisiert. Um an diesen Zustand zu kommen, muss dieser durch eine entsprechende Operation periodisch abgefragt werden. Danach kann mit der Ausführung vom nächsten Schritt verfahren werden, ebenfalls angetriggert durch den OpenToscaAdapter.

Nach dem Upload der CSAR-Datei wird die Verarbeitung der TOSCA-Dokumente ausgeführt. Wurde dieser Schritt erfolgreich beendet, erfolgt das Deployment von Implementation Artifacts und Plänen. Danach kann durch die Auswahl eines Build-Planes die Instanziierung eines Service Templates angestossen werden.

Bei der Ausführung von Plänen unterscheidet sich das Handling von den vorherigen Operationen im Bezug auf die Übermittlung des Ergebnisses einer Operation. Musste beim Deployment von Implementation Artifacts der Zustand periodisch durch den OpenToscaAdapter abgefragt werden, wird bei der Ausführung von Plänen ein Callback registriert. Über dieses Callback wird der OpenToscaAdapter über die Ergebnisse der Pläneausführung informiert. Im Erfolgsfall wird die Endpunkt-Referenz der erstellten Service-Instanz an die Control-Komponente und im Endeffekt an den ESB übergeben.

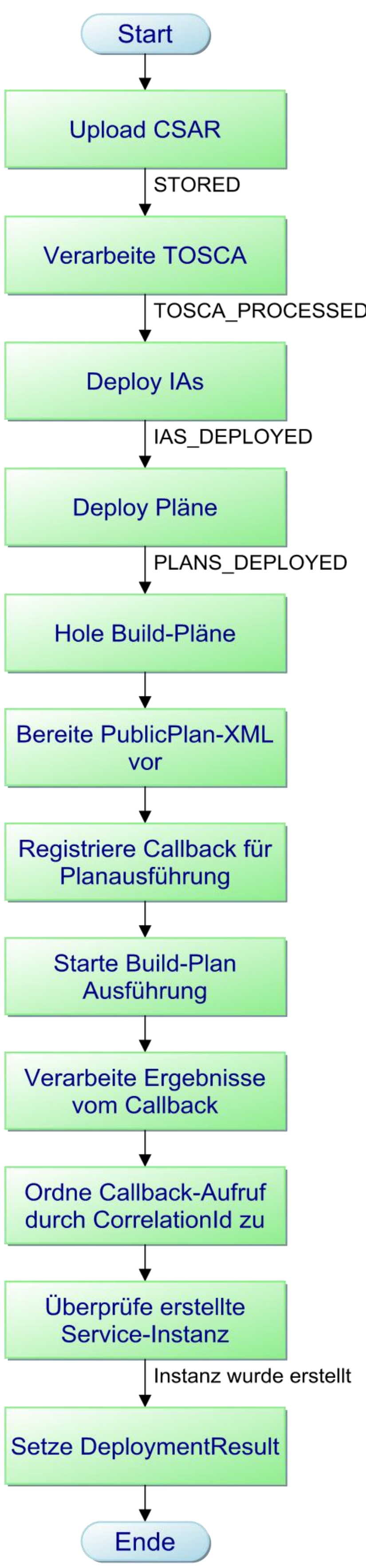

Abbildung 6-19: Ablauf von deployService()-Methode

6.7.3 OpenTOSCA ContainerAPI

Dieser Abschnitt beschreibt ausgewählte Operationen vom OpenTOSCA ContainerAPI, die für die Realisierung des OpenToscaAdapters von Bedeutung sind. Wie bereits erläutert sind folgende Schritte notwendig um eine in TOSCA beschriebene Cloud-Anwendung (in diesem Fall ein Simulationsservice) mit Hilfe von OpenTOSCA in einer Cloud-Umgebung zu provisionieren:

- Übertragung der CSAR-Datei in den OpenTOSCA Container
- Verarbeitung der CSAR-Datei durch den Container:
  - Verarbeitung der TOSCA Definitionen
  - Deployment von Implementation Artifacts
  - Deployment von Plänen
- Ausführung eines Build-Plans, der eine Service-Instanz erstellt

6.7.3.1 Upload von CSAR-Dateien

Das Upload von CSAR-Dateien kann auf unterschiedliche Art und Weise erfolgen. Sollte sich die CSAR-Datei bereits auf dem gleichen Rechner wie der TOSCA Container befinden, kann eine HTTP-Anfrage mit der Methode POST und der Angabe des absoluten Pfades auf dieser Maschine abgesetzt werden. Auch wenn diese Möglichkeit für den produktiven Einsatz weniger relevant ist, eignet sie sich jedoch gut für die Tests.

HTTP-Request

Headers:

```
POST /containerapi/CSARs

Host: localhost:1337

Content-Type: text/plain
```

Body:

```
C:/Documents/openTOSCA/MockService.csar
```

Listing 6-4: Upload CSAR (lokal) - HTTP-Request

Eine andere Möglichkeit die CSAR-Datei zum TOSCA Container zu übertragen wäre die Übertragung der ganzen Datei direkt im Body der (POST) HTTP-Anfrage. Diese Vorgehensweise wird üblicherweise in Webseiten-Formularen genutzt um dem Benutzer den Upload von Dateien zu ermöglichen. Wichtig ist hierbei den Content-Type-Header auf *multipart/form-data* zu

setzen. Der Inhalt der HTTP-Nachricht ist dann auch entsprechend dem angegebenen Content-Typ zu befüllen [47].

HTTP-Request

Headers:

```
POST /containerapi/CSARs
Host: localhost:1337
Content-Type: multipart/form-data
```

Body:

...

Listing 6-5: Upload CSAR (Web Formular) – HTTP-Request

Die dritte Möglichkeit erlaubt die Angabe einer URL im Body der entsprechenden (POST) HTTP-Anfrage.

HTTP-Request

Headers:

```
POST /containerapi/CSARs
Host: localhost:1337
Content-Type: text/plain
```

Body:

```
http://www.example.com/repository/MockService.csar
```

Listing 6-6: Upload CSAR (URL) – HTTP-Request

Der TOSCA Container holt dann die CSAR-Datei über die angegebene URL selbstständig zum Container.

Beim erfolgreichen Upload lässt sich aus der entsprechenden HTTP-Response der Ort der CSAR-Datei im Container herauslesen. Die Antwort-Nachricht ist bei allen drei Varianten gleich.

HTTP-Response

Headers:

```
HTTP/1.1 201 Created

Location:
http://localhost:1337/containerapi/CSARs/MockService.csar

Content-Length: 0
```

Body:

Listing 6-7: Upload CSAR (lokal) - HTTP-Response

6.7.3.2 Verarbeitung von CSAR-Dateien

Nachdem die CSAR-Datei in den Container geladen wurde, wird sie in mehreren Schritten verarbeitet. Die Verarbeitungsschritte müssen in der richtigen Reihenfolge durchgeführt und vom Benutzer gestartet werden. Für jeden Verarbeitungsschritt sind eigene Operationen vorhanden.

Um den ganzen Ablauf zu koordinieren, benötigt man auch aktuelle Status-Informationen von dem Deployment-Prozess. Die Verarbeitung erfolgt sequentiell, daher werden die Status-Informationen zwischen den Verarbeitungsschritten geholt. Erst wenn der jeweilige Schritt tatsächlich erfolgreich beendet ist, was durch den entsprechenden Deployment-Status belegt ist, kann der nächste Schritt ausgeführt werden.

Statusabfrage

Die Status-Informationen werden über eine GET-Anfrage geholt mit der Angabe von *csarId*, die eine geladene CSAR-Datei im Container eindeutig identifiziert. Die *scarId* ist in der Regel als der CSAR-Dateiname zusammen mit der Datei-Endung definiert. Die Listings 6-8 und 6-9 beinhalten die entsprechenden HTTP-Request und -Response für die Statusabfrage.

HTTP-Request

Headers:

```
GET /containerapi/CSARControl/{csarId}

Host: localhost:1337

Content-Type: text/plain
```

Body:

Listing 6-8: Zustand vom Deployment-Prozess – HTTP-Request

Der Inhalt der Antwort-Nachricht enthält die zugehörige ID der CSAR-Datei (`scarId`), den aktuellen Deployment-Zustand (`deploymentState`) und eine Liste der möglichen Operationen (`OPERATION`).

HTTP-Response

Headers:

```
HTTP/1.1 200 OK

Content-Type: text/plain
```

Body:

```
<DeploymentProcess DeploymentState="TOSCA_PROCESSED">
  <ProcessId/>
  <Operations>
      <Operation>INVOKE_IA_DEPL</Operation>
  </Operations>

</DeploymentProcess>
```

Listing 6-9: Zustand vom Deployment-Prozess – HTTP-Response

Das Deployment von CSAR-Dateien lässt sich mit 3 Opeartionen steuern:

- `PROCESS_TOSCA` startet die Verarbeitung der TOSCA-Dokumente einer CSAR-Datei
- `INVOKE_IA_DEPL` ruft das Deployment von Implementation Artifacts auf
- `INVOKE_PLAN_DEPL` startet das Deployment von Plänen

Die folgende Tabelle 6-3 gibt Aufschluss über die vorhandenen Zustände sowie über ihre Bedeutung.

STORED	Die CSAR-Datei ist im Container gespeichert
TOSCAPROCESSING_ACTIVE	TOSCA-Dateien einer CSAR werden gerade verarbeitet
TOSCA_PROCESSED	Verarbeitung von TOSCA-Dateien einer CSAR ist abgeschlossen
IA_DEPLOYMENT_ACTIVE	Implementation Artifacts einer CSAR werden gerade deployt
IAS_DEPLOYED	Deployment von Implementation Artifacts einer CSAR ist abgeschlossen
PLAN_DEPLOYMENT_ACTIVE	Pläne einer CSAR-Datei werden gerade deployt
PLANS_DEPLOYED	Deployment von Plänen einer CSAR-Datei ist abgeschlossen

Tabelle 6-3: Zustände vom Deployment Prozess

TOSCA Verarbeitung

Im ersten Deployment-Schritt werden TOSCA Definition-Dokumente eingelesen, geprüft und verarbeitet. Die Auflösung von Imports und Referenzen findet dabei statt und die TOSCA-Definitionen werden im internen Modell abgelegt. Die Operation wird über eine POST-Anfrage mit der Angabe der `csarId` und der Operationsbezeichnung `PROCESS_TOSCA` angetriggert.

HTTP-Request

Headers:

```
POST /containerapi/CSARControl/{csarId}
Host: localhost:1337
Content-Type: text/plain
```

Body:

```
PROCESS_TOSCA
```

Listing 6-10: TOSCA-Verarbeitung - HTTP-Request

Nachdem die Operation erfolgreich ausgeführt wurde (Deployment-Status: `TOSCA_PROCESSED`) kann mit dem Deployment von Implementation Artifacts fortgefahren werden. Generell kann eine CSAR-Datei mehrere Service Templates, also Beschreibungen von mehreren Services, beinhalten. Aus diesem Grund muss die CSAR auf vorhandene Service Templates untersucht werden. Die Liste der Service Templates kann mit dem folgenden GET-Request abgefragt werden.

HTTP-Request

Headers:

```
GET /containerapi/CSARControl/{csarId}/ServiceTemplates

Host: localhost:1337
```

Body:

Listing 6-11: Service Templates abfragen – HTTP-Request

Nun kann aus der Antwort-Nachricht ein bestimmtes Service Template, das für die Erzeugung einer Service Instanz benutzt werden soll, ausgewählt werden. Ein Service Template (`servicetemplate`) innerhalb einer CSAR-Datei wird in folgender Form identifiziert: `{namespace}LocalPart`, z.B.: `{http://www.example.com/tosca}MockService_ServiceTemplate`

HTTP-Response

Headers:

```
HTTP/1.1 200 OK

Content-Type: text/plain
```

Body:

```
{serviceTemplate}&{serviceTemplate}&…
```

Listing 6-12: Service Templates abfragen – HTTP-Response

Implementation Artifacts Deployment

Damit die Management-Operationen während der Ausführung von Management-Plänen aufgerufen werden können, müssen die Implementation Artifacts zuerst deployt werden, z.B. auf einem Apache Tomcat. Diese Aufgabe wird mit der folgenden POST-Anfrage realisert. Als Parameter im Body sind der

Name der Operation (`INVOKE_IA_DEPL`) sowie die ID (Qualified Name [48]) vom Service Template anzugeben.

HTTP-Request

Headers:

```
POST /containerapi/CSARControl/{csarId}

Host: localhost:1337

Content-Type: text/plain
```

Body:

```
INVOKE_IA_DEPL&

{http://www.example.com/tosca}MockService_ServiceTemplate
```

Listing 6-13: Implementation Artifacts Deployment - HTTP-Request

Während die Implementation Artifacts deployt werden, ist der Gesamtzustand vom Deployment-Prozess `IA_DEPLOYMENT_ACTIVE`. Beim erfolgreichen Abschluss der Operation wird der Zustand auf `IAS_DEPLOYED` geändert. Ist das der Fall, kann im nächsten Schritt mit dem Deployment der Pläne begonnen werden.

Plan Deployment

Durch Pläne werden Management-Aspekte von Service Instanzen beschrieben, insbesondere auch die Erstellung einer Service Instanz. OpenTOSCA kann mit Plänen, die in Form von BPEL Prozessen vorliegen, umgehen. Damit die BPEL-Pläne ausgeführt werden können, müssen sie zunächst auf einer BPEL Engine deployt werden. Bei OpenTOSCA wird dafür der WSO2 Business Process Server [49] eingesetzt. Das Deployment von Plänen erfolgt über eine POST-Anfrage mit der Operation `INVOKE_IA_DEPL` und der Angabe vom Service Template Identifikator.

HTTP-Request

Headers:

```
POST /containerapi/CSARControl/{csarId}

Host: localhost:1337

Content-Type: text/plain
```

Body:

```
INVOKE_PLAN_DEPL&

{http://www.example.com/tosca}MockService_ServiceTemplate
```

Listing 6-14: Plan Deployment - HTTP-Request

Beim laufenden Deployment von Plänen ist der Gesamtzustand entsprechend `PLAN_DEPLOYMENT_ACTIVE`, nach der erfolgreichen Ausführung der Operation - `PLANS_DEPLOYED`.

Mit diesem Schritt ist die komplette Verarbeitung der CSAR-Datei zu Ende, das Management vom Lebenszyklus einer oder mehreren Service-Instanzen kann nun über die Ausführung der entsprechenden Management-Pläne erfolgen.

6.7.3.3 Ausführung von Plänen

Ein Service Template kann mehrere Management-Pläne enthalten, das sinnvolle Minimum stellen ein Build-Plan und ein Termination-Plan dar. Ein Build-Plan soll in der Lage sein eine neue Service-Instanz zu erstellen. Dagegen ist ein Termination-Plan dafür gedacht, eine existierende Service-Instanz zu beenden und die dadurch belegten Cloud-Resourcen wieder freizugeben.

In OpenTOSCA redet man in diesem Bezug von Public Plans, das sind alle Pläne, die nach außen sichbar sind. Dabei sind Pläne in drei Bereiche aufgeteilt:

- Build-Pläne
- Termination-Pläne
- Sonstige Managementpläne

Durch die folgende GET-Anfrage lassen sich vorhandene Build-Pläne auflisten bzw. ihre IDs. Für eine Plan-Ausführung wird unter anderem die Plan-ID benötigt, daher soll diese Operation vor der eigentlichen Plan-Ausführung durchgeführt werden.

HTTP-Request

Headers:

```
GET /containerapi/CSARs/{csarId}/PublicPlans/BUILD

Host: localhost:1337
```

Body:

Listing 6-15: Build-Pläne auflisten - HTTP-Request

Analog dazu können Termination- und sonstige Management-Pläne aufgelistet werden mit:

```
/containerapi/CSARs/{csarId}/PublicPlans/TERMINATION

/containerapi/CSARs/{csarId}/PublicPlans/OTHERMANAGEMENT
```

Nachdem die ID vom benötigen Build-Plan ermittelt wurde, kann die Ausführung dieses Plans gestartet werden. Dies geschieht über die folgende POST-Anfrage, mit einer relativ komplexen XML als Body, daher auch der Content-Typ soll passend gesetzt werden (`application/xml`).

HTTP-Request

Headers:

```
POST /containerapi/CSARs/{csarId}/Instances

Host: localhost:1337

Content-Type: application/xml
```

Body:

```
<PublicPlan
  CSARID="MockService.csar"
  PlanType="http://docs.oasis-pen.org/tosca/ns/2011/12/PlanTypes/BuildPlan"
  InternalPlanID="0"
  PlanID="ns2:TestLifeCycleDemoBUILDPlan"
  InternalInstanceInternalID="0"
  InterfaceName="TestBuildPlan"
  OperationName="process"
  InputMessageID="ns3:TestBuildPlanRequest"
  PlanLanguage="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
  isActive="false"
  hasFailed="false"
  xmlns="http://www.opentosca.org/ConsolidatedTOSCA"
  xmlns:ns2="org.opentosca.demo"
  xmlns:ns3="planinvocation/test/buildplan">

  <InputParameter name="input" type="string" required="true">
    test2
  </InputParameter>

  <InputParameter name="CorrelationID" type="correlation" required="true">
    1337
  </InputParameter>

  <InputParameter name="CallbackAddress" type="callbackaddress" required="true">
    http://localhost:8080/dummy
  </InputParameter>

  <OutputParameter name="result" type="string" required="true"/>
  <OutputParameter name="CorrelationID" type="correlation" required="true"/>
  <OutputParameter name="CallbackAddress" type="callbackaddress" required="true"/>

</PublicPlan>
```

(Markierungen am Rand: 1 = Attribute von PublicPlan, 2 = InputParameter, 3 = OutputParameter)

Listing 6-16: Plan Ausführung - HTTP-Request

Die abgebildete XML stellt einerseits ein Interface für die Kommunikation von außen dar, andererseits enthält sie aber in der aktuellen Version eine Reihe von internen Daten, die für die Kommunikation nicht gesetzt werden müssen, die Struktur jedoch ist so vorgegeben und muss eingehalten werden. Es ist davon

auszugehen, dass das Interface in den neueren Versionen von OpenTOSCA bereinigt wird.

Die XML besteht aus einer Reihe von Attributen (1) eines Public Plans, einer Menge von Input- (2) und Output-Parameter (3). Aus dem eben beschriebenen Grund werden nicht alle Elemente der XML beschrieben, sondern nur auf die relevanten eingegangen.

Durch die Angabe von Plan-ID (Attribut `InternalPlanID`) wird entschieden, welcher Plan ausgeführt werden sollte. Die Ausführung der Pläne erfolgt asynchron. Nach der erfolgreichen Ausführung eines Planes kann das Ergebnis an eine zuvor definierte Callback-Adresse gesendet werden (Input-Parameter: `CallbackAddress`). Um eine an die Callback-Adresse gesendete Nachricht der Ausführung eines Planes zuzuordnen, wird eine Correlation-ID benötigt (Input-Parameter: `CorrelationID`). Laufende Pläne können durch folgende GET-Anfrage ermittelt werden.

HTTP-Request

Headers:

```
GET /containerapi/CSARs/{csarId}/Instances/
{instanceId}/ActivePublicPlans
Host: localhost:1337
```

Body:

Listing 6-17: Aktive Pläne ermitteln - HTTP-Request

Nach der erfolgreichen Ausführung eines Build-Planes erscheint die erzeugte Instanz in der Liste vorhandener Instanzen. Die vorhandenen Instanzen können wie folgt aufgelistet werden:

HTTP-Request

Headers:

```
GET /containerapi/CSARs/{csarId}/Instances

Host: localhost:1337
```

Body:

Listing 6-18: Service Instanzen ermitteln - HTTP-Request

Existierende Service Instanzen können durch die Ausführung eines Termination-Planes beenden werden. Die Vorgehensweise dafür ist analog: das entsprechende preparierte Public-Plan-XML soll gesendet werden an

`/containerapi/CSARs/{csarId}/Instances/{InstanceId}.`

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde die am IAAS entworfene Architektur für die On-Demand Provisionierung von Workflowausführungsumgebungen und Services für wissenschaftliche Workflows in Cloud-Umgebungen behandelt. Dabei stand die dynamische Provisionierung von Simulationsservices (samt der darunterliegenden Infrastruktur) in einer Cloud-Umgebung zur Laufzeit eines Workflows im Fokus. Die Architektur wurde erweitert und basierend auf TOSCA prototypisch umgesetzt. Für diesen Zweck wurde der SimTech Prototyp durch die Hereinnahme von neuen Komponenten gemäß dem Architekurentwurf (Service Registry, Service Repository, ESB^{MT}, Provisioning Manager), sowie durch die neue Provisionierungslogik auf der ESB-Ebene erweitert. Als Ergebnis können Simulationsservices abhängig von der ausgewählten Provisionierungsstrategie dynamisch in einer Cloud-Umgebung bereitgestellt werden.

7.1 Ausblick

Im Rahmen dieser Arbeit wurden nicht alle Aspekte vom Aufbau und von der Nutzung einer cloud-basierten Infrastruktur für wissenschaftliche Workflows ausführlich behandelt, da dies den Umfang der Arbeit sprengen würde. Die Themen sind jedoch als mögliche weiterführende Arbeiten interessant und werden daher in diesem Abschnitt noch mal kurz erläutert.

Benutzerkontrolle über die Provisionierung

Der realisierte Ansatz für die dynamische Provisionierung hat zunächst den Benutzer des sWfMS (den Wissenschaftler) aus dem Provisionierungsprozess ausgeschlossen, indem die ganze Provisionierung für ihn transparent gehalten wird. Es ist zu untersuchen, inwiefern die Kontrolle über die Provisionierungsprozesse an den Benutzer weitergegegen werden kann und bei bestehendem Bedarf eine geeignete Form dafür zu finden.

Reele Simulationsservices mit TOSCA

Für den praktischen Teil der Arbeit wurden in TOSCA beschriebene Beispiel-Services benutzt. Der nächste Schritt wäre dann die Umsetzung der reelen Simulationsservices mit TOSCA und die Evaluierung der Lösung anhand von reelen Simulationen.

Service Repository

Das Service Repository für die Verwaltung von TOSCA-CSARs (Cloud Service Archive) wurde lediglich am Rande behandelt und soll Teil weiterführender Arbeiten werden im Bezug auf die Community-basierte Entwicklung von Simulationen.

Provisionierung anderer Teile der Infrastruktur

Der Fokus der Arbeit lag an der dynamischen Provisionierung von Simulationsservices zur Laufzeit eines Workflows. Die Provisionierung der Workflow-Ausführungsumgebung in einer Cloud-Umgebung wurde nicht behandelt. Der Zeitpunkt der Provisionierung sowie die Nutzungsdauer unterscheiden sich stark von dem behandelten Fall.

Provisionierunsstrategien

Die Flexibilität der Architektur im Hinblick auf den Provisionierungsprozess ermöglicht den Einsatz von unterschiedlichen Strategien für die Provisionierung. Abhängig von bestimmten Kriterien wie z.B. die Dauer eines Web-Service-Aufrufs, können verschiedene Strategien eingesetzt werden. Besonders interessant sind automatisierte Optimierungen und Strategieauswahl bzw. Anpassungen zu Laufzeit. Als Basis für solche Verfahren könnten z.B. Sammlung und Auswertung statistischer Daten dienen.

Feingranulare Wiederverwendung der Infrastruktur

Die Nutzung der für die Simulationsservices provisionierten Infrastruktur kann auch deutlich feingranularer sein. In diesem Fall ist bei der Freigabe von Ressourcen zu berücksichtigen, ob bestimmte Komponenten wie z.B. ein Application Server derzeit nicht anderweitig in Verwendung sind und daher die Ressource nicht freigegeben werden darf.

Literaturverzeichnis

[1] F. Leymann und D. Roller, Production Workflow - Concepts and Techniques, Englewood Cliffs: PTR Prentice Hall, 2000.

[2] T. Oinn, M. Greenwood, M. Addis, M. Nedim Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. Pocock, M. Senger, A. Wipat und C. Wroe, „Taverna: Lessons in Creating a Workflow," in *Concurrency and Computation: Practice and Experience*, 18(10):1067-110, 2006.

[3] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor und I. Wang, „Programming Scientific and Distributed Workflow with Triana with Services," in *Concurrency and Computation: Practice and Experience. Special Issue on Scientific Workflows*, 2005.

[4] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann und M. Reiter, „Conventional Workflow Technology for Scientific Simulation," in *Guide to e-Science*, X. Yang, L. Wang und W. Jie, Hrsg., Springer-Verlag, 2011.

[5] „Exzellenzcluster Simulation Technology," [Online]. Available: http://www.simtech.uni-stuttgart.de/. [Zugriff am 10 09 2013].

[6] „SimTech Visionen," [Online]. Available: http://www.simtech.uni-stuttgart.de/forschung/visionen/index.html. [Zugriff am 10 09 2013].

[7] „Institut für Architektur von Anwendungssystemen der Universität Stuttgart," [Online]. Available: http://www.iaas.uni-stuttgart.de/. [Zugriff am 10 09 2013].

[8] K. Vukojevic, D. Karastoyanova und F. Leymann, „Choreographies and Cloud Infrastructure for Simulation Workflows," *SimTech Status.*

[9] S. Strauch, V. Andrikopoulos, F. Leymann und D. Muhler, „ESB^MT: Enabling Multi-Tenancy in Enterprise Service Buses," in *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science*, 2012.

[10] L. Reinfurt, Migrating the SimTech Simulation Workflow Management System to a Cloud Infrastructure, Studienarbeit, IAAS, Universität Stuttgart, 2013., 2013.

[11] „Vukojevic-Haupt Karolina, Institut für Architektur von Anwendungssystemen (IAAS), Universität Stuttgart," [Online]. Available: http://www.iaas.uni-stuttgart.de/institut/mitarbeiter/vukojevic/index.php. [Zugriff am 10 09 2013].

[12] „OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC," [Online]. Available: https://www.oasis-open.org/committees/tosca/. [Zugriff am 10 09 2013].

[13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak und S. Wagner, „OpenTOSCA – A Runtime for TOSCA-based Cloud Applications," *ICSOC,* 2013.

[14] „Web Services Architecture,“ [Online]. Available: http://www.w3.org/TR/ws-arch/. [Zugriff am 10 09 2013].

[15] S. Hotta, Ausführung von Festkörpersimulationen auf Basis der Workflow Technologie, Universität Stuttgart, Diplomarbeit, 2010.

[16] Y. Zou, Simulation des Verhaltens von Zellkomponenten in biologischen Netzwerken mit Hilfe von Workflow Technologie, Universität Stuttgart, Diplomarbeit, 2012.

[17] F. Bernd und D. Wagner, Webservice und Workflow-Technologie für Proteinmodellierung, Universität Stuttgart, Studienarbeit, 2010.

[18] „OASIS | Advanced open standards for the information society,“ [Online]. Available: https://www.oasis-open.org/. [Zugriff am 10 09 2013].

[19] D. A. Chappell, Enterprise Service Bus: Theory in Practice, O'Reilly, 2004.

[20] J. Turnbull, Pulling Strings with Puppet, FirstPress, 2007.

[21] A. Jacob, „Infrastructure in the cloud era,“ in *Proceedings at International O'Reilly Conference Velocity*, 2009.

[22] T. Dörnemann, E. Juhnke und B. Freisleben, On-Demand Resource Provisiong for BPEL Workflows Using Amazon's Elastic Compute Cloud.

[23] J. Kirschnick, J. M. Alcaraz Calero, L. Wilcock und N. Edwards, Towards an Architecture for the Automated Provisioning of Cloud Services.

[24] O. Kopp, K. Görlach, D. Karastoyanova, F. Leymann, M. Reiter, D. Schumm, M. Sonntag, S. Strauch, T. Unger, M. Wieland und R. Khalaf, „A Classification of BPEL Extensions," *Journal of Systems Integration,* 2011.

[25] „Stuttgarter Workflow Maschine," [Online]. Available: http://www.iaas.uni-stuttgart.de/forschung/projects/swom/.

[26] „Reference Architecture Foundation for Service Oriented Architecture Version 1.0," [Online]. Available: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.pdf. [Zugriff am 10 09 2013].

[27] „OpenTOSCA - Open Source Runtime Environment for TOSCA," [Online]. Available: http://www.iaas.uni-stuttgart.de/OpenTOSCA/indexE.php. [Zugriff am 10 09 2013].

[28] „The Eclipse Foundation open source cummunity website.," [Online]. Available: http://www.eclipse.org/. [Zugriff am 10 09 2013].

[29] „Apache Tomcat," [Online]. Available: http://tomcat.apache.org/. [Zugriff am 10 09 2013].

[30] „Apache Axis2,“ [Online]. Available: http://axis.apache.org/axis2/java/core/. [Zugriff am 10 09 2013].

[31] „MySQL,“ [Online]. Available: http://www.mysql.com/. [Zugriff am 10 09 2013].

[32] „Apache ActiveMQ,“ [Online]. Available: http://activemq.apache.org/. [Zugriff am 10 09 2013].

[33] „Pluggable Framework for Apache ODE,“ [Online]. Available: http://www.iaas.uni-stuttgart.de/forschung/projects/ODE-PGF/. [Zugriff am 10 09 2013].

[34] T. Steinmetz, Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE, Universität Stuttgart, Diplomarbeit, 2008.

[35] „Apache ODE (Orchestration Director Engine),“ [Online]. Available: http://ode.apache.org. [Zugriff am 10 09 2013].

[36] D. Karastoyanova, R. Khalaf, R. Schroth, M. Paluszek und F. Leymann, „BPEL Event Model,“ Institut für Architektur von Anwendungssystemen der Universität Stuttgart, 2006.

[37] „Eclispe BPEL Designer Project,“ [Online]. Available: http://www.eclipse.org/bpel/. [Zugriff am 10 09 2013].

[38] „Fragmento: Process Fragment Library," [Online]. Available: http://www.iaas.uni-stuttgart.de/forschung/projects/fragmento/start.htm. [Zugriff am 10 09 2013].

[39] D. Schumm, D. Karastoyanova, F. Leymann und S. Strauch, „Advanced Process Fragment Library," in *Proc. of the 19th International Conference on Information Systems Development (ISD'10)*, 2010.

[40] D. Schumm, F. Leymann, Z. Ma, T. Scheibler und S. Strauch, „Integrating Compliance into Business Processes: Process Fragments as Reusable Compliance Controls," in *Proc. of the Multikonferenz Wirtschaftsinformatik (MKWI'10)*, 2010.

[41] D. Dentsas, Integration von Fragmento in eine Rich Client Plattform, Universität Stuttgart, Diplomarbeit, 2011.

[42] P. Binkele und S. Schmauder, „An atomistic Monte Carlo simulation for precipitation," *Zeitschrift für Metallkunde,* 49:1-6, 2003.

[43] „Apache ServiceMix," [Online]. Available: http://servicemix.apache.org/. [Zugriff am 10 09 2013].

[44] S. Essl, Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support, Universität Stuttgart, Masterarbeit.

[45] „Apache Camel," [Online]. Available: http://camel.apache.org/. [Zugriff am 10 09

2013].

[46] „Apache ODE - Endpoint Configuration," [Online]. Available: http://ode.apache.org/endpoint-configuration.html. [Zugriff am 10 09 2013].

[47] L. Masinter, „RFC2388 Returning Values from Forms: multipart/form-data," 08 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2388.txt. [Zugriff am 10 09 2013].

[48] „Namespaces in XML 1.0 - QName," [Online]. Available: http://www.w3.org/TR/REC-xml-names/#NT-QName. [Zugriff am 10 09 2013].

[49] „WSO2 Business Process Server," [Online]. Available: http://wso2.com/products/business-process-server. [Zugriff am 10 09 2013].

Alle URLs wurden zuletzt am 10.09.2013 geprüft.

Erklärung

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

Gäufelden, 10.09.2013

Ort, Datum Unterschrift

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Gäufelden, 10.09.2013

Place, Date Signature