Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3058

Planungsverfahren im scientific Workflow Management

Diana Przybylski

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Dimka Karastoyanova
Betreuer:	Dipl.-Inf. Katharina Görlach
begonnen am:	13. Juli 2010
beendet am:	12. Januar 2011
CR-Klassifikation:	H.4.1, I.2.9

Inhaltsverzeichnis

Web Services, scientific Workflows und Cloud Computing sind wichtige, aktuelle Forschungsgebiete ([1]). Die Ausführung von scientific Workflows in einer Cloud hat den Vorteil der unbeschränkten Ressourcen durch die Illusion unendlicher Rechenkapazität in der Cloud. Scientific Workflows, die häufig Web Services aufrufen, können effizienter ausgeführt werden, wenn diese großen Workflows in Teilprozesse (Fragmente) aufgeteilt werden, die verteilt und dynamisch in der Cloud ausgeführt werden. Dies ist effizienter, da auf eine Änderung der Infrastruktur reagiert werden kann, wenn es nötig ist. Dies bedeutet entweder eine erneute Zuordnung eines Fragments ohne Server auf einen anderen Server oder eine Refragmentierung eines zu großen Fragments.

1 Einleitung

Scientific Workflows und die Nutzung von Clouds sind in der aktuellen Forschung wichtige Themengebiete [1]. Ein scientific Workflow ist ein spezieller Workflow, dessen charakteristische Merkmale die Berechnung eines wissenschaftlichen Problems und die lange Ausführungsdauer sind. Cloud Computing stellt IT-Infrastrukturen (unter anderem Software oder Ressourcen) vollautomatisch zur Verfügung, die eventuell über die ganze Welt verteilt sind. Die Menge der Ressourcen beispielsweise kann dynamisch, der Nachfrage entsprechend, angepasst werden. Aufgrund der Eigenschaften von scientific Workflows ist es sinnvoll, diese in einer Cloud auszuführen, da es möglich ist, immer die nötigen Ressourcen bereitgestellt zu bekommen. Um eine optimale Ausführung des Workflows in der Cloud zu gewährleisten, ist es sinnvoll, die Workflows in einem ersten Schritt in kleinere Teilworkflows oder Fragmente zu zerlegen. Diese können dann durch Anwendung eines Planungsalgorithmus aus dem Bereich der künstlichen Intelligenz auf die zur Verfügung stehenden Ressourcen in der Cloud gemappt werden. Durch die Aufteilung des Workflows und die Nutzung eines Planungsalgorithmus ist es insbesondere möglich, das Deployment der Workflowteile dynamisch zu gestalten. Somit kann die Ausführung des Workflows optimal den aktuell verfügbaren Ressourcen angepasst werden. Darüber hinaus kann auf Veränderungen der Infrastruktur zeitnah reagiert werden, indem je nach Veränderung der Infrastruktur neue Teilworkflows erstellt werden oder die bereits vorhandenen neu gemappt werden.

In diesem Kapitel wird die Arbeit grob vorgestellt. In Abschnitt 1.1 werden die nötigen Grundlagen kurz dargelegt, worauf die Motivation dieser Arbeit in Abschnitt 1.2 und deren Zielsetzung in Abschnitt 1.3 folgen. Dieses Kapitel wird von verwandten Arbeiten (Abschnitt 1.4) abgeschlossen. In Kapitel 2 werden die grundlegenden Themengebiete, die für diese Arbeit benötigt werden, genau betrachtet. In dem darauf folgenden Kapitel 3 wird Augenmerk auf die konzeptionelle Lösung gelegt, die unter anderem die Fragmentierung und die verteilte, dynamische Ausführung enthält. In Kapitel 4 ist die Umsetzung der konzeptionellen Lösung zu finden. Abschließend werden in Kapitel 5 die Arbeit und ihre Ergebnisse zusammengefasst und ein Ausblick auf weitere mögliche Arbeiten geboten.

1.1 Einordnung in das Umfeld

Zur Beschreibung dieses Prozesses, muss zuerst ein grundlegendes Wissen vorhanden sein. Um dieses zu erlangen, werden im Folgenden die wichtigsten Themengebiete kurz erläutert. Diese Gebiete umfassen die Service Oriented Architecture, Workflows, Workflow Management und scientific Workflows sowie Business Process Execution Language (BPEL) und Web-Service Definition Language (WSDL). Sie hängen folgendermaßen zusammen: Ein Workflow wird im technischen Sinne durch BPEL und WSDL implementiert. WSDL implementiert darüber hinaus auch SOA. Außerdem werden die künstliche Intelligenz und Cloud Computing kurz vorgestellt.

Service Oriented Architecture (SOA) ist ein Architekturmuster, das Service Oriented Computing (SOC) realisiert. SOC stellt ein Paradigma dar, das Services benutzt. Services sind Funktionen, die an Netzwerk-Adressen zur Verfügung gestellt werden. Durch die plattformunabhängigen Standards, die bei den Services verwendet werden, ist es möglich, den Service von verschiedensten Plattformen aus aufzurufen und dabei zum Beispiel auch unterschiedliche Kommunikationsprotokolle zu benutzen. Die Web Service Technologie ist ein Standard- und Technologie-Stack, der SOA unterstützt.

Um die Funktionsweise von SOA zu verstehen, ist das SOA Dreieck hilfreich. Dieses ist in Abbildung 1 zu sehen. Der Service Requestor ist auf der Suche nach einem Service, den er mit Hilfe der Service Discovery findet. Die Service Discovery kennt den Service Provider, der zuvor seinen angebotenen Service öffentlich gemacht hat. Hat der Service Requestor nun den passenden Service gefunden, kann er die Funktionen des Services aufrufen und benutzen. Die Services und deren Funktionen werden beschrieben, so dass das Ergebnis der Funktion einsichtig ist, die Umsetzung, die dahinter steht, bleibt aber verborgen.

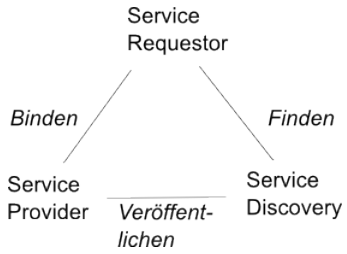

Abbildung 1: SOA Dreieck

SOA ist die Grundlage für Workflows und **Workflow Management**. SOA stellt verteilte, heterogene Dienste (Services) zur Verfügung. Sie sind lose gekoppelt, was bedeutet, dass die Anzahl der Annahmen, die zwei Parteien beim Informationsaustausch übereinander machen (beispielsweise wie viele Parameter in welchen Datentypen an die aufgerufene Funktion übergeben werden müssen), reduziert ist.

Diese Services werden durch einen Workflow kombiniert. Ein Workflow implementiert in den meisten Fällen einen Geschäftsprozess durch einen Graphen ([2]). Eine mögliche Anwendung von Workflows und Workflow Management ist der Einsatz im Geschäftsbereich. Die Dienste des Unternehmens können mit Hilfe von Workflows kombiniert werden.

Workflows beschreiben Abläufe eines Unternehmens entlang der gesamten Wertschöpfungskette. Der Steuerung der Ablauffolge wird somit durch das Workflow Management automatisiert. Darüber hinaus besteht die Möglichkeit, Arbeitsschritte, die automatisch durch Programme oder Services erledigt werden können, direkt aus dem Workflow aufzurufen und sie auszuführen.

Ein Workflow kann durch einen Graphen dargestellt werden. Dabei sind Knoten einzelne Aufgaben, die beispielsweise während eines Geschäftsprozesses ausgeführt werden müssen. Kanten stellen die Abhängigkeiten zwischen diesen Aufgaben dar. Knoten können Aufgaben (sogenannte *Aktivitäten*) ganz unterschiedlicher Art sein. Beispielsweise kann eine Aktivität nur beinhalten, dass eine Meldung erscheinen oder ein Zugriff auf ein Medium getätigt werden muss. Diese Aktivitäten können automatisch ausgeführt werden. Im Gegensatz dazu stehen Aktivitäten, die nur durch Menschenhand erledigt werden können.

Scientific Workflows sind spezielle Workflows, die Fragestellungen der Wissenschaft behandeln; genauer gesagt Berechnungen durchführen. Ihre Kennzeichen sind, dass sie im Allgemeinen langlaufende Workflows sind, die nur einmal instantiiert werden und die Funktionen, die sie nutzen, Web Services sind. Darüber hinaus sind sie datenzentriert und verarbeiten große Datenmengen.

Um die Unterschiede zwischen Workflows und Scientific Workflows nochmals hervorzuheben, muss das Augenmerk speziell auf das Deployment und die Instantiierung gelegt werden. Bei einem Workflow kommt das Deployment zeitlich vor der Instantiierung, die wiederum beliebig oft durchgeführt werden kann. Bei einem scientific Workflow findet das Deployment auch einmal statt, er wird aber meist nur einmal ausgeführt.

Workflow Management (siehe Abbildung 2) beinhaltet die Modellierung, die IT-Verfeinerung, das Deployment, die Ausführung und die Überwachung und Analyse der Überwachungsergebnisse von Workflows [2].

Während der Modellierungphase wird ein Prozess modelliert, der den zu beschreibenden Ablauf abbildet. In den Modellierungsprozess fließen zusätzlich die Key Performance Indicators (KPIs) ein. Sie gliedern sich in die Kategorien Kosten, Zeit, Qualität und Flexibilität und werden oft für Vergleiche durch Metriken benutzt. Der Modellierungsprozess geschieht meist nicht auf der IT-Ebene, weshalb eine IT-Verfeinerung benötigt wird, die das Modell verfeinert und ergänzt. Ist dieser Prozess abgeschlossen, folgt das Deployment und die Ausführung des Prozesses. Von diesem Workflow werden so viele Instanzen, wie nötig, erzeugt und ausgeführt. Die Ausführung wird überwacht, um den Workflow nach einer Analyse der Ergebnisse zu verbessern. Die dadurch gewonnenen Erkenntnisse können in der Modellierungsphase mit einbezogen werden. Somit kann der Workflow iterativ verbessert werden.

BPEL ist eine Sprache, die den Workflow implementiert. **WSDL** hingegen ist eine Sprache, die als Interface gesehen werden kann. Sie definiert Web Services, spezifiziert den Ort des Services und seine Operationen.

Die **Künstliche Intelligenz** ist ein Fachbereich, der sehr viele Facetten und Anwendungsgebiete hat. Zur künstlichen Intelligenz gehören unter anderem die Gebiete der Wahrnehmung und Verarbeitung von Informationen, Fortbewegung und Bewegungsabläufe sowie das Planen von Abläufen. Die Planung findet beispielsweise in Compu-

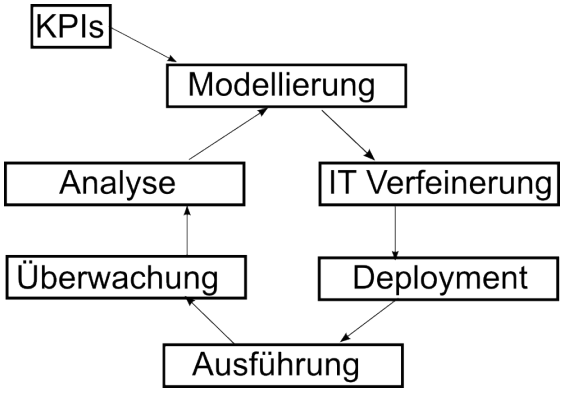

Abbildung 2: Der Business Process Model Lebenszyklus

terspielen Verwendung. Wenn man gegen einen Schachcomputer spielt, reagiert dieser durch Anwendung eines Planungsalgorithmus. Er erkennt den Spielzug, hat somit eine neue Ausgangssituation, um sein Ziel zu erreichen und wählt den besten nächsten Zug aus einer Menge von gültigen Regeln aus. Auch bei Computerspielen, die Menschen simulieren, werden diese Algorithmen benötigt, um ihr Verhalten möglichst real wirken zu lassen.

Die Planung und alle anderen Gebiete der künstlichen Intelligenz finden ihre Anwendung in dem wohl bekanntesten Beispiel der künstlichen Intelligenz: der Robotik. Auf dem Gebiet der Robotik wird Forschung betrieben, die Roboter immer menschen-ähnlicher machen soll. Hierzu gehören außer der Wahrnehmung und der Motorik vor allem die Fähigkeiten, Entscheidungen aufgrund von Tatsachen zu treffen und das Handeln an diesen Tatsachen auszurichten. Um dies zu ermöglichen, werden neben anderen Algorithmen auch Planungsalgorithmen eingesetzt. Sie verwenden Annahmen, die die Ausgangssituation beschreiben und ein definiertes Ziel haben. Auf der Basis dieses Wissens erstellen sie einen Plan, um das Ziel zu erreichen. Treten während der Ausführung des Plans unvorhergesehene Ereignisse auf, ist es dem Planer möglich, einen anderen Weg des Plans einzuschlagen, der nicht Teil der ursprünglichen Lösung war. Durch diese dynamische Änderung kann er auf das unvorhergesehene Ereignis reagieren. Diese spezielle Art der Planung ist die stetige Planung, die die Arbeit überwacht, bis sie beendet ist.

Cloud Computing stellt IT-Infrastrukturen zur Verfügung [3]. Im Speziellen können dies Rechen-Ressourcen sein. Diese sind über die ganze Welt verteilt und können dynamisch an den Bedarf der Benutzer angepasst werden. Dies ist die unterste der *drei Ebenen*, die es im Cloud Computing gibt. Sie wird „Infrastructure as a Service“ genannt. Die zwei weiteren Ebenen sind „Platform as a Service“ und „Software as as Service“.

Durch die Bereitstellung von Ressourcen können eigene Ressourcen eingespart werden. Darüber hinaus steht immer die nötige Menge an Ressourcen in der Cloud zur Verfügung.

1.2 Motivation der Arbeit

Die charakteristische Eigenschaften von scientific Workflows, die in dieser Arbeit adressiert werden, ist die lange Ausführungsdauer und eine daraus resultierende lange Reservierung oder Benutzung von Ressourcen. Diese Eigenschaft spricht für eine Aufteilung des Workflows in kleinere Teile (Fragmente), um diese Fragmente verteilt ausführen zu können. Durch eine verteilte Ausführung der Fragmente werden Ressourcen kürzer benutzt und schneller wieder frei gegeben. Alle Fragmente können auf unterschiedlichen Servern ausgeführt werden, wodurch eine parallele Ausführung unterstützt und eventuell beschleunigt wird.

Durch die Verwendung eines Planungsalgorithmus der künstlichen Intelligenz während der Ausführung von scientific Workflows wird diese flexibler gestaltet und somit optimiert. Das Deployment kann durch den Planungsalgorithmus dynamisch angewendet und zum spätest möglichen Zeitpunkt ausgeführt werden. Es wird eine optimale Verteilung der Workflowfragmente auf die zur Verfügung stehenden Server erreicht, da vor jedem Deployment überprüft werden kann, ob sich die Infrastruktur so verändert hat, dass neu fragmentiert und geplant oder nur neu geplant werden muss.

1.3 Zielsetzung der Arbeit

In dem nachfolgend beschriebenen Ablauf wird die Fragmentierung und das stetige Planen zur Ausführung eines scientific Workflows in einer Cloud kombiniert.

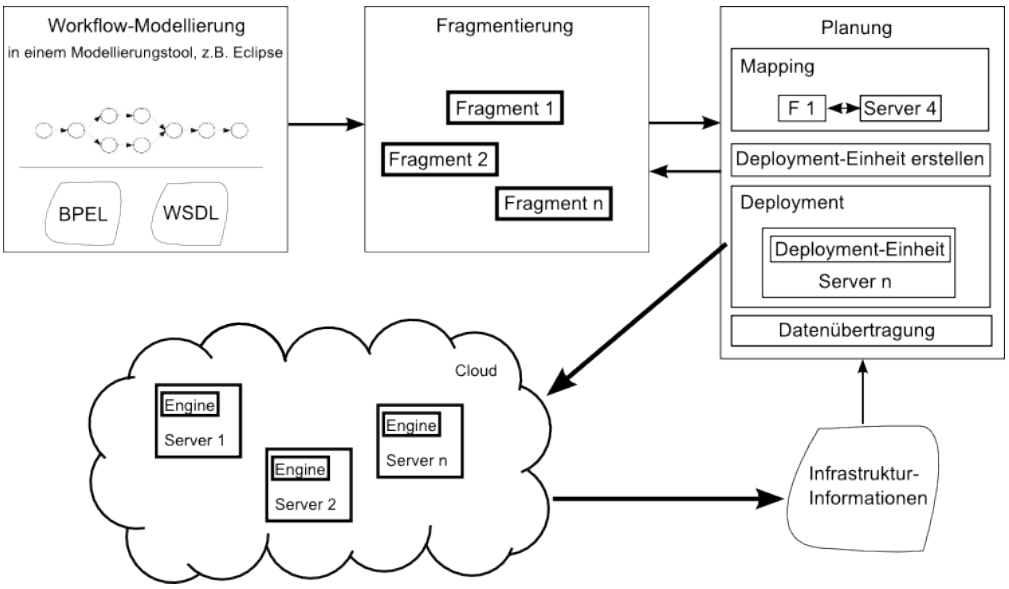

Abbildung 3: *Zielsetzung der Arbeit:* Für die Fragmentierung wird ein Workflow und Infrastruktur-Informationen der Cloud benötigt. Anschließend folgt das Deployment der Fragmente auf die Server der Cloud. Das Deployment erfolgt dynamisch; bei Änderungen der Infrastruktur kann darauf reagiert werden.

Zu Beginn steht die Modellierung eines Workflows mit einem Modellierungswerkzeug (siehe Abbildung 3). Der Workflow ist hierbei durch eine BPEL- und eine oder mehrere WSDL-Datei(en) beschrieben. Nun wird er durch eine Fragmentierung in Teile zerlegt, die in einer Cloud ausgeführt werden sollen. Die *Fragmentierung* wird aufgrund der Struktur der BPEL-Datei durchgeführt. Sie teilt den Prozess, der durch die BPEL-Datei beschrieben wird, in kleinere Teilprozesse, die nur Sequenzen von Basis-Aktivitäten, einzelne alternative Zweige oder Schleifenkörper des ursprünglichen Prozesses enthalten können. Alle Basis-Aktivitäten, die während der Fragmentierung vorkommen, werden unverändert in die Fragmente übernommen. Die Ausführungsreihenfolge der Aktivitäten bleibt erhalten. Während der Fragmentierung werden die Vorgänger- und Nachfolgerfragmente jedes einzelnen Fragments festgehalten, damit die Ausführungsreihenfolge gewährleistet werden kann.

Eine Komponente, die erst während des dynamischen Deployments zum Tragen kommt, koordiniert die Übertragung der benötigten Daten zwischen den Fragmenten. Diese Komponente ist in Abbildung 3 bei der Planung mit inbegriffen. Ist hierbei die Menge der Übertragung größer als ein fester Schwellwert, müssen diese beiden Fragmente auf einem Server ausgeführt werden, damit die Daten nicht übertragen werden müssen, sondern nur auf einem Server benötigt werden.

Ist die Fragmentierung abgeschlossen, kann das Deployment gestartet werden (siehe Abbildung 3). Zu Beginn müssen Informationen über die Größe der zur Verfügung stehenden Server der Cloud und die Größe der Fragmente eingeholt werden. Diese Informationen werden zum Mapping der Fragmente auf die Server benötigt. Während des Mappings wird für jedes Fragment zuerst ein Server passender Größe gesucht. Ist dieser nicht vorhanden, wird der Server mit dem größten freien Speicherplatz für das Mapping herangezogen. Ist für jedes Fragment ein Server gefunden worden, kann das Deployment gestartet werden. Um die Fragmente zu erstellen, muss eine Deployment-Einheit erstellt werden. Diese enthält das Prozessfragment, die zugehörige(n) WSDL-Datei(en) und die von diesem Fragment aufgerufenen Services. Diese Services bestehen wiederum aus einer oder mehreren WSDL-Datei(en) und der Implementierung. Die Aufgaben des Deployments sind, die Deployment-Einheiten in der richtigen Reihenfolge dynamisch auf die gemappten Server zu legen und zu instantiieren. Das dynamische Deployment endet, wenn alle Deployment-Einheiten und somit der gesamte Prozess erfolgreich ausgeführt worden sind. Die Umsetzung des dynamischen Deployments wird durch die Anwendung eines stetigen Planungsalgorithmus umgesetzt, der dafür verantwortlich ist, dass alle Fragmente in der richtigen Reihenfolge korrekt ausgeführt werden. Da der Prozess mehrmals instantiiert werden kann, kann es auch mehrere Instanzen der Fragmente geben. Die Instanzen eines Fragments sind immer jeweils genau einem Prozess zugeordnet.

Es ist während des dynamischen Deployments möglich, dass sich die Infrastruktur der Cloud verändert, zum Beispiel weil das dynamische Deployment sehr lange läuft. In diesem Fall ist es möglich, dass sich die Verfügbarkeit der Server in der Cloud oder deren Größe ändert. Wenn durch diese Änderung ein Fragment, das auf einen Server gemappt wurde, nicht mehr auf diesem Server ausgeführt werden kann, muss die Größe dieses Fragments erneut betrachtet werden. Ist diese größer als der Gesamtspeicher des größten Servers, wird die Fragmentierung neu ausgelöst. Bei dieser Fragmentierung muss

darauf geachtet werden, dass das Fragment nun so aufgeteilt wird, dass die entstehenden Fragmente kleiner als der Gesamtspeicher des größten Servers sind. Ist die Größe des Fragments aber kleiner als der Gesamtspeicher des größten Servers, wird das dynamische Deployment erneut ausgeführt, um für dieses Fragment einen anderen Server zu finden.

1.4 Related Work

Die bisherigen Ansätze in der Literatur befassen sich größtenteils nur mit der Fragmentierung oder Planung. Die Fragmentierung ist aus vielen verschiedenen Blickwinkeln betrachtet worden. Diese Arbeiten werden in Kapitel 3.3.1 genauer beleuchtet.

In Arbeit [4] ist die Fragmentierung nicht mehr das Thema. Hier geht es um Dynamik in Workflows. Es soll möglich sein, auf Änderungen des Modells reagieren zu können. Diese Änderungen müssen an den Workflows vorgenommen werden, bei denen das Deployment schon stattgefunden hat. Dieser Workflow ist als Petri-Netz repräsentiert. Auf Änderungen zu reagieren bedeutet, dass es Regionen im Workflow gibt, die verändert werden müssen. Dazu werden diese Regionen identifiziert. Sie sollten so schnell wie möglich durch die Regionen mit den enthaltenen Änderungen ersetzt werden. Eine Gemeinsamkeit zu dieser Arbeit ist die Reaktion auf Änderungen. Eine Änderung hat in den beiden Ansätzen aber unterschiedliche Auswirkungen. Dieser Ansatz reagiert bei einer Änderung mit einer erneuten Planung oder Fragmentierung, wohingegen die Methode aus [4] diese Änderung in alle bestehenden Workflows einbringt. Aus diesem Grund können diese beiden Ansätze nicht verglichen werden und haben als einzige Gemeinsamkeit die Reaktion auf Veränderungen.

Ansatz [5] wendet KI-Planungsverfahren auf Workflows an. Er bearbeitet Benutzeranfragen, die einen Zugriff auf autonome und heterogene Datenquellen benötigen. Die Umsetzung besteht aus einem Mediator, der Zugriff auf diese Daten bietet. Der Zugriff ist durch eine globale Anfragesprache möglich. In diesem Ansatz wird der Zugriff auf verschiedene Datenquellen durch Anwendung eines Mediators aus der künstlichen Intelligenz gewährleistet. Der Mediator, der in Ansatz [5] verwendet wird, hat Ähnlichkeit mit einem Workflow Management System, da auch dieses heterogene Aufrufe bewerkstelligen muss. Auch der Zugriff auf autonome und heterogene Datenquellen hat Ähnlichkeiten mit SOA. Da aber keine Planungsalgorithmen verwendet werden, sondern ein Mediator aus dem Bereich der künstlichen Intelligenz und auch keine Fragmentierung vorgenommen wird, sind mehr Unterschiede als Parallelen zu diesem Ansatz zu finden.

Im Folgenden werden die Ansätze betrachtet, die sich mit der Planung oder allgemein der künstlichen Intelligenz befassen und Ähnlichkeiten mit dem hier gewählten Vorgehen haben. Der Algorithmus aus dem Ansatz von [6] fragmentiert einen Workflow und führt ihn verteilt aus. Ein Workflow ist in einem oder mehreren Agenten gekapselt. Diese befinden sich an unterschiedlichen Standorten und kooperieren miteinander. Für die Umsetzung wird eine Technik angewendet, die den Workflow fragmentiert und die Fragmente in Agenten kapselt. Bei diesem Ansatz werden Agenten der künstlichen Intelligenz verwendet. Eine Gemeinsamkeit ist in der Kapselung des Workflows in unterschiedliche Teile zu sehen. Da die Teile aber nicht aufgrund der Struktur des Workflows hergestellt werden, sondern so, dass verwandte Aufgaben zusammengefasst werden, ist

der Algorithmus nicht mit der Fragmentierung dieser Arbeit zu vergleichen.

Pegasus [7] befasst sich auch mit der Fragmentierung und verteilter Ausführung von Workflows. Es bearbeitet auf der Grundlage spezieller Verteilungs- und Ausführungskriterien Aufgaben, die in Grids ausgeführt werden. Diese beiden Ansätze haben die Ausführung, die durch eine Planung durchgeführt werden und die vorangestellte Fragmentierung gemeinsam. Die Ausführungsumgebung ist aber eine andere und der Fragmentierung liegen andere Regeln zugrunde.

Arbeit [8] fällt auf, da in diesem Ansatz Dynamik durch Planungsverfahren hinzukommt. Der Ansatz plant die Ausführung eines Workflows, reagiert während der Ausführung auf Änderungen und plant gegebenenfalls neu. Hierzu werden die tatsächlichen Ergebnisse eines Teilschritts mit den erwarteten Ergebnissen verglichen. Bei einem unerwarteten Ergebnis wird dieses analysiert. Ist das Ergebnis nicht akzeptabel, so wird eine Neuplanung ausgelöst. Die in diesem Ansatz angewendeten Planungsalgorithmen unterscheiden sich von diesen, die in diesem Ansatz Verwendung finden, da eine andere Planungsaufgabe ausgeführt wird. Die gegebene Aufgabe wird mit Hilfe des partiell ordnenden Planens und der Neuplanung gelöst. In dem hier angewendeten Ansatz wird stetige Planung angewendet, um auf Änderungen der Infrastruktur reagieren zu können. Darüber hinaus wird in dieser Arbeit keine vorangestellte Fragmentierung angewendet.

2 Grundlagen

In diesem Kapitel werden die nötigen Grundlagen betrachtet. Diese umfassen Web Service Description Language (WSDL), Business Process Execution Language (BPEL), Cloud Computing und die Planungsalgorithmen der künstlichen Intelligenz.

2.1 Web Service Description Language (WSDL)

WSDL wird benutzt, um Web Services zu definieren. In einer WSDL-Datei wird der Ort des Web Services und die Operationen, die der Service bereitstellt, spezifiziert. Das Dokument besitzt XML-Struktur und enthält eine Menge von Definitionen, die den Web Service beschreiben. Diese Informationen sind abstrakt, die konkreten Informationen, also die Implementierung der Services, sind beispielsweise in einer BPEL-Datei (BPEL siehe Abschnitt 2.2) enthalten. Durch dieses Design ist die abstrakte Funktionalität von den Details der Service Beschreibung getrennt. Die Hauptelemente sind:

- *Types:* hier werden die benötigten Datentypen definiert
- *Message:* die auszutauschenden Daten werden hier abstrakt definiert
- *Operation:* hier sind die abstrakten Aktionen aufgelistet, die vom diesem Service unterstützt werden
- *Port Type:* Spezifikation der Menge der Operationen, die von einem Endpoint (Punkt, beispielsweise ein Prozessor oder eine Entität, zu dem Nachrichten geschickt werden können) unterstützt werden
- *Binding:* konkretes Protokoll und Datenformat, um einen Port Type zu implementieren
- *Port:* einzelner „Endpoint“ der von einer Netzwerkadresse identifiziert wird, die ein bestimmtes Binding unterstützt
- *Service:* Sammlung von (verwandten) Endpoints

2.2 Business Process Execution Language (BPEL)

Web Services Business Process Execution Language (BPEL) ist eine Sprache zur Spezifikation des Verhaltens von Geschäftsprozessen. BPEL ist eine Kombination einer graph-basierten und einer rechen-basierten Sprache, da BPEL aus den Sprachen IBM WSFL[1] (graph-basierte Sprache) und Microsoft XLANG[2] (rechen-basierte Sprache) entstanden ist.

In BPEL gibt es strukturierte und nicht-strukturierte (oder Basis-)Aktivitäten. Diese werden im Folgenden genauer erläutert. Die nicht-strukturierten Aktivitäten enthalten keine anderen Aktivitäten.

[1] http://xml.coverpages.org/wsfl.html
[2] http://msdn.microsoft.com/en-us/library/aa577463%28v=bts.70%29.aspx

Zu ihnen gehören:

- *receive: Receive* empfängt eine Nachricht, die in einer Variablen gespeichert wird. *Receive* kann eine neue Prozessinstanz instantiieren.
- *reply: Reply* versendet eine Nachricht. Diese Nachricht stellt die Antwort einer synchronen Kommunikation dar und antwortet somit auf ein vorangegangenes *Receive*.
- *invoke: Invoke* wird benutzt, um einen Web Service aufzurufen. Typischerweise ruft ein *Invoke* eine Operation des Services auf.
- *assign: Assign* weist einer Variablen einen Wert zu. Dieser kann in einem *Assign* berechnet werden oder durch Manipulation einer anderen Variablen entstehen. Die Operation hat einen *from*-Teil, der angibt von wo der Wert kopiert werden soll und einen *to*-Teil, in dem steht, wohin der Wert geschrieben wird.
- *empty: Empty* ist ein Platzhalter und stellt eine leere Operation dar.
- *exit:* Diese Aktivität beendet den Prozess.
- *throw: Throw* signalisiert explizit interne Fehler eines Geschäftsprozesses.
- *rethrow: Rethrow* propagiert Fehler weiter, die in einem Fault Handler gefangen wurden.
- *wait:* Verzögert die Ausführung für eine bestimmte definierte Zeitspanne oder bis ein definierter Zeitpunkt erreicht ist.

Zu den strukturierten Aktivitäten gehören:

- *Flow:* Ein *Flow* ist ein Konstrukt, in dem mehrere Aktivitäten enthalten sein können. Aktivitäten, die nicht durch Links verbunden sind, können parallel ausgeführt werden. Aktivitäten, die mit Links verbunden sind, müssen in einer bestimmten Reihenfolge ausgeführt werden. Ein Link hat eine Source- oder Ursprungsaktivität und eine Target- oder Zielaktivität. Eine Source ist eine ausgehende Kante einer Aktivität und Target eine eingehende Kante.
- *Sequence:* Aktivitäten, die in einer *Sequence* enthalten sind, müssen sequenziell in der spezifizierten Reihenfolge ausgeführt werden. Es ist aber durchaus möglich, dass eine *Sequence* beispielsweise einen *Flow* oder eine andere Aktivität enthält, die wieder Aktivitäten enthalten, die parallel ausgeführt werden können.
- *Scope:* Ein *Scope* unterstützt den Kontext, der das Ausführungsverhalten der enthaltenen Aktivitäten beeinflusst.
- *If: If* ist eine Aktivität, das eine Alternative des möglichen Kontrollflusses herbeiführt. Bei einem *If* gibt es zwei oder mehrere mögliche Pfade. Je nachdem, wie die Bedingung, die an das *If* geknüpft ist, ausgewertet wird, wird ein bestimmter Pfad zur Ausführung gewählt.

- *Repeat Until: Repeat Until* ist eine Schleife, die ausgeführt wird, bis die Abbruchbedingung wahr wird. Die Besonderheit bei dieser Schleife ist, dass die Abbruchbedingung am Schleifenende überprüft wird. Aus diesem Grund wird sie mindestens ein Mal ausgeführt.

- *While: While* ist eine Schleife, die ausgeführt wird, bis die Abbruchbedingung wahr wird. Der Unterschied zu *Repeat until* besteht darin, dass die Abbruchbedingung am Anfang des Schleifendurchlaufs ausgewertet wird. Es ist möglich, dass die Schleife nicht ausgeführt wird, wenn die Abbruchbedingung zu Beginn schon als wahr ausgewertet wird.

- *For Each:* Einer *For Each* Schleife kann sequenziell oder parallel ausgeführt werden. Die Anzahl, wie oft der Körper ausgeführt wird, richtet sich nach einem definierten Start- und Endwert. Wenn die Completion Condition wahr wird, kann die Ausführung abgebrochen werden, obwohl der Endwert noch nicht erreicht ist. Soll die *For Each* Schleife sequenziell ausgeführt werden, müssen alle Schleifendurchgänge nacheinander ausgeführt werden. Soll die Schleife aber parallel ausgeführt werden, kann jeder Durchlauf einzeln und somit parallel zu den anderen Durchläufen ausgeführt werden. Dies ist nur möglich, wenn sich die einzelnen Durchläufe nicht gegenseitig beeinflussen und der Zählwert für jeden Schleifendurchgang korrekt gesetzt wird.

- *Pick: Pick* wartet darauf, dass ein Event eintrifft, das in einer Menge definierter Events enthalten ist. Alle Events, die in der Menge enthalten sind, können eintreffen. Sobald das erste Event eingetroffen ist, werden alle weiteren Events ignoriert. Das eingetroffene Event ist ausschlaggebend dafür, welche Aktivität angestoßen wird. Somit ist eine Alternative aus allen möglichen Kontrollflüssen ausgewählt worden.

BPEL beinhaltet weitere Konstrukte (Isolated Scopes, Message Exchange Handling, Error Handling, Compensation Handlers, Fault Handlers, Termination Handlers, Event Handlers), die hier nicht beachtet werden.

In einem Prozess werden Aktivitäten in einer bestimmten Reihenfolge ausgeführt. Dies ist der Kontrollfluss. Durch den Kontrollfluss sind die Aktivitäten miteinander verbunden und müssen in der vorgeschriebenen Reihenfolge ausgeführt werden. Wenn der Ablauf des Workflows in einem DAG dargestellt ist, sind die Kanten der Kontrollfluss und die Knoten die Aktivitäten. Nun kann es bei der Navigation durch einen solchen Graphen dazu kommen, dass die Bedingung einer Kante zu „falsch“ ausgewertet wird. In einem solchen Fall wird der oder die Knoten (bei Workflows auch Aktivitäten genannt) unter Umständen nicht ausgeführt, die durch diese Kante aus erreichbar sind. Da bei der Ausführung eines Knotens aber jede eingehende Kante ausgewertet wird, kann es durchaus sein, dass folgende Knoten wieder ausgeführt werden, obwohl eine Kante als „falsch“ ausgewertet wurde. Die Auswertung der eingehenden Kanten erfolgt nach einer Bedingung. Ist diese Bedingung unter Berücksichtigung aller eingehenden Kantenwerte falsch, wird der Knoten nicht ausgeführt. Ist die Bedingung hingegen wahr, wird er

ausgeführt. Aufgrund der Tatsache, dass für die Bestimmung, ob ein Knoten ausgeführt wird oder nicht, alle Kanten ausgewertet werden müssen, muss abgewartet werden, bis alle Kantenbedingungen einen Wert erhalten haben. Aus diesem Grund ist es sinnvoll und für die weitere Ausführung des Geschäftsprozesses auch notwendig, ein „falsch“ weiterzugeben, um alle folgenden Bedingungen auswerten zu können. Diese Methode heißt Dead Path Elimination.

2.3 Cloud Computing

Cloud Computing stellt voll automatisiert IT-Infrastrukturen zur Verfügung. IT-Infrastrukturen bezeichnen unter anderem Maschinen oder Programme, die den Betrieb von beispielsweise Software ermöglichen. Die IT-Infrastruktur befindet sich unter der Ebene, in der eine Software ausgeführt wird und trägt zur automatisierten Informationsverarbeitung bei. Diese Infrastruktur kann über alle Kontinente verteilt sein. Die Menge von bereitgestellten Ressourcen kann ja nach Bedarf dynamisch angepasst werden. Sind bestimmte Rechner momentan nicht Teil einer Cloud, die Cloud braucht aber mehr Ressourcen, können so viele Rechner wie nötig hinzugenommen werden. Auf diese Weise stehen immer genügend Ressourcen zur Verfügung, sie müssen aber nicht ungenutzt in der Cloud verbleiben, da sie in diesem Fall aus der Cloud entfernt werden würden. Dadurch kann einem Nutzer die Bereitstellung von unendlichen Ressourcen vorgespielt werden. Dies hat zur Folge, dass es keine Beschränkung der Ressourcen gibt und die geforderte Menge von Ressourcen immer zur Verfügung gestellt werden kann. Dies ermöglicht eine Einsparung der eigenen Ressourcen.

Falls in einer Cloud Anwendungssoftware zur Verfügung gestellt wird, ist diese stets auf dem neusten Stand. Ein Benutzer muss nicht selbst ein Update installieren, sondern nutzt in der Cloud immer die aktuelle Version. Die Nutzung der Cloud ist gebührenpflichtig, aber nur die Zeit der Nutzung muss bezahlt werden. Der Erwerb von Lizenzen entfällt völlig. Dies bringt einen Kostenvorteil mit sich, da nur die tatsächliche Nutzung berechnet wird.

Es gibt drei Typen von Clouds. Der erste hier aufgeführte Typ beschreibt die unterste Ebene einer Pyramide. Diese Ebene stellt reine Ressourcen durch virtuelle Server zur Verfügung, die nicht konfiguriert sind. Da diese Ebene Infrastruktur zu Verfügung stellt, wird sie ***Infrastructure as a Service (IaaS)*** genannt. EC2 (Elastic Compute Cloud) von Amazon gehört zu dieser Ebene von Clouds.

Die darüber liegende Ebene heißt ***Platform as a Service (PaaS)***. Hier wird eine Anwendung erstellt und auf einen Server geladen. Der Server übernimmt die Aufteilung zur Ausführung auf die physischen Rechner. Somit ist in dieser Ebene der Server schon konfiguriert. Ein Beispiel für eine Cloud dieser Schicht ist force.com von Salesforce.com.

Die oberste Ebene ist ***Software as a Service (SaaS)***. In dieser Ebene wird eine komplette Software als Service zur Verfügung gestellt. Diese Software kann genutzt werden, ohne dass diese auf dem eigenen Rechner installiert werden muss oder schon installiert ist. Google mit GoogleDocs ist beispielsweise ein Anbieter dieser Ebene.

2.4 Planungsalgorithmen der künstlichen Intelligenz

Planen hat die Aufgabe, eine Folge von Aktionen zu finden, die ein definiertes Ziel erreicht. Um diese Aufgabe zu erfüllen, gibt es mehrere Ansätze (siehe Referenz [9]). Für das ***Planen mit Zustandsraumsuche*** werden Vorbedingungen, Aktionen und Effekte von Aktionen spezifiziert. Das Progressionsplanen geht dabei von einem Ausgangszustand aus und ermittelt Aktionsfolgen anhand von Vorbedingungen und Effekten, bis eine dieser Folgen den Zielzustand erreicht. Somit ist die Aufgabe gelöst. Bei einem Regressionsplan hingegen wird von dem Endzustand aus rückwärts geplant. Dies hat den Vorteil, dass auf diese Weise nur relevante Aktionen betrachtet werden. Bei dieser Planungsart werden nur streng lineare Aktionsfolgen betrachtet, weswegen die Vorteile der Problemzerlegung nicht genutzt werden können. Diese Vorteile kommen bei ***partiell ordnendem Planen*** zum Tragen. Bei dieser Planungsart wird unabhängig an mehreren Unterzielen gearbeitet, die anschließend kombiniert werden. So ergibt sich Flexibilität in der Ausführung.

Das ***Planen mit Aussagenlogik*** plant mit dem Ansatz der Erfüllbarkeit eines logischen Satzes. Ein Modell, das den Satz erfüllt, weist allen Aktionen, die Teil einer korrekten Lösung sind „wahr“ zu.

Die bisher vorgestellten Planungsmöglichkeiten funktionieren in der Theorie, sind aber in der Praxis nur bedingt anwendbar, da in der realen Welt zusätzliche Faktoren beachtet werden müssen. Zu diesen Faktoren zählen die Dauer von Aktionen und die Ressourcen, die für die Bearbeitung der Aktionen benötigt werden. In der realen Welt ist ein weiteres Ziel, eine minimale Gesamtzeit der Ausführung zu erreichen. Um diesem Ziel seine nötige Beachtung zu schenken, gibt es mehrere Ansätze. Ein Ansatz ist das ***Hierarchische Task-Netzwerk-Planen***. Der ursprünglich problembeschreibende Plan wird als Beschreibung der Aufgabe auf höchster Ebene betrachtet. Dieser Plan wird verfeinert, indem die gegebene Aufgabe in Teilaufgaben zerlegt werden, bis diese Aufgaben nicht mehr zerlegbar sind. Nun ist die niedrigste Ebene erreicht und eine Aktion ist eine partiell geordnete Menge. Während des ***Planens in nichtdeterministischen Domänen*** muss ein Planungsagent seine Wahrnehmung nutzen. So kann er bei unerwarteten Vorgängen den Plan verändern oder ihn ersetzen. Wenn die Unbestimmbarkeit während des Planens begrenzt ist, kann ***sensorloses*** oder ***bedingtes Planen*** angewendet werden. Das sensorlose Planen erzeugt standardmäßige Pläne, die ohne Wahrnehmung ausgeführt werden können. Bedingtes Planen hingegen erzeugt einen bedingten Plan mit unterschiedlichen Verzweigungen für die verschiedenen Möglichkeiten, die auftreten können. Ist die Unbestimmbarkeit unbegrenzt kommt ***Ausführungsüberwachung und Neuplanung*** oder eine ***stetiges Planen*** zur Bearbeitung der Aufgabe in Frage. Bei der Ausführungsüberwachung und Neuplanung wird zusätzlich eine Ausführungüberwachung verwendet, die die aktuelle Situation bewertet. Macht der Zustand den Anschein, dass er erfolgreich weitergeführt werden könnte, wird die Planung fortgeführt, sonst wird sie überarbeitet und eventuell neu geplant. Das stetige Planen ist darauf ausgelegt, so lange zu arbeiten, bis die Aufgabe erledigt ist. Der Planer ist hier in der Lage mit unerwarteten Umständen in der Umgebung umzugehen und diese zu verarbeiten, einmal gesteckte Ziele wieder aufzugeben und dazukommende Ziele hinzuzunehmen.

Das ***Multiagenten-Planen*** bringt die Planung auf eine neue Ebene. Es können mehrere Agenten in einer Umgebung zusammenarbeiten. Ein Agent nimmt die anderen Agenten in sein Modell auf, ohne seine grundlegenden Algorithmen ändern zu müssen. Um konstruktiv zu arbeiten, müssen sich die Agenten auf einen Plan einigen, der ausgeführt werden soll. Die Agenten teilen die Aufgaben, die erledigt werden müssen, untereinander auf. Dies wird durch Kommunikation erreicht. Arbeiten die Agenten nicht zusammen, sind sie Konkurrenten. Hier stehen die Nutzenfunktionen der Agenten in Konkurrenz zueinander.

3 Dynamische, verteilte Ausführung von BPEL-Prozessen

Dieses Kapitel beschäftigt sich mit der dynamischen, verteilten Ausführung von BPEL-Prozessen. In Kapitel 3.1 wird der gesamte Ablauf beschrieben, um einen Überblick über die gesamte Funktionalität zu geben. Darauf folgt eine detailliertere Darlegung der Algorithmen, die die Datenflussanalyse (Abschnitt 3.2), die Fragmentierung (Abschnitt 3.3) und das dynamische Deployment (Abschnitt 3.4) umfassen. Abschließend wird dieser Ansatz in Kapitel 3.5 diskutiert.

In diesem Rahmen ist es nicht möglich, das gesamte Spektrum von BPEL-Prozessen zu betrachten. Darüber hinaus wird ein bestehender Algorithmus benutzt, der seinerseits Einschränkungen verlangt.

Der betrachtete Workflow muss ein gerichteter azyklischer Graph (DAG) sein. Schleifen werden gesondert behandelt. Eigentlich würde eine Schleife die Eigenschaften eines DAG zerstören. In BPEL sind sie aber ein einziges Konstrukt, weshalb die Eigenschaften des DAG gewahrt bleiben. Darüber hinaus muss das Bernstein Kriterium erfüllt sein, das besagt, dass zwischen parallel ausgeführten Pfaden keine Datenkanten existieren dürfen.

3.1 Allgemeiner Ablauf

Diese Arbeit realisiert ein stetiges Planungsverfahren zur dynamischen, verteilten Ausführung von BPEL-Prozessen. Der Ablauf, in dem die Fragmentierung und das dynamische Deployment enthalten ist, ist in Abbildung 4 zu sehen. Beginn des Planungs-

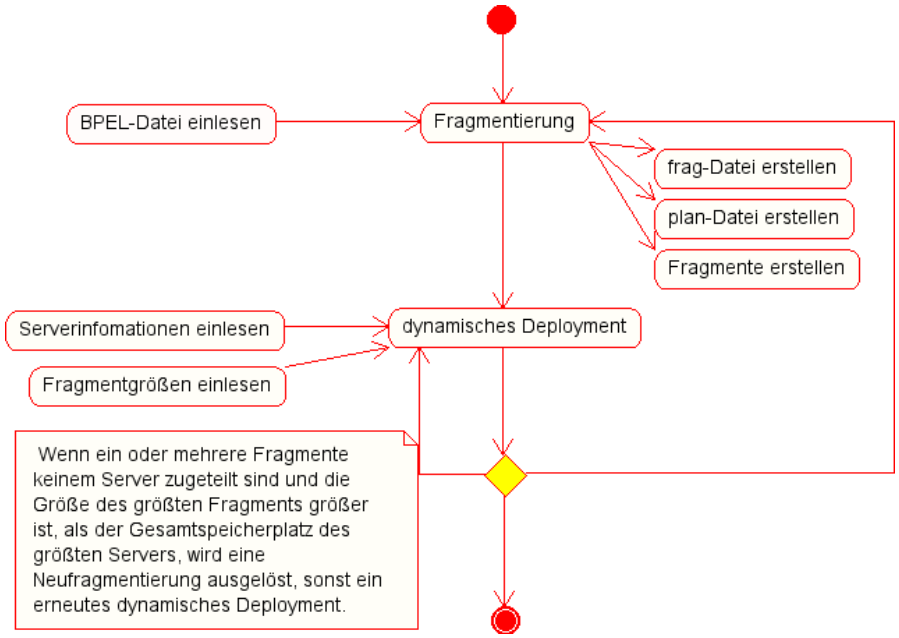

Abbildung 4: Stetige Planung zur dynamischen, verteilten Ausführung von BPEL-Prozessen

verfahrens ist die ***Fragmentierung***. Um sie auszuführen, muss eine BPEL-Datei eingelesen werden und die WSDL-Datei(en) zur Verfügung stehen. Es werden Fragmente auf Grundlage des BPEL-Prozesses erstellt, indem die einzelnen Aktivitäten, die in dem Prozess vorhanden sind, genauer betrachtet werden und nach festgelegten Regeln in kleinere Teile gruppiert werden. Dazu wird der BPEL-Prozess betrachtet. Dieser muss azyklisch sein, da eine Datenanalyse nötig ist, die dies fordert. Der Prozess liegt als eine BPEL-Datei vor. Diese kann alle Basis- und strukturierten Aktivitäten enthalten. Basis-Aktivitäten werden bei der Fragmentierung direkt in die Fragmente übernommen. Ein *Flow* wird nicht übernommen, bei diesem Konstrukt wird bei der Erstellung der Fragmente darauf geachtet, dass alle Aktivitäten, die nicht voneinander abhängen, parallel in verschiedenen Fragmenten ausgeführt werden. Eine *Sequence* und ein *Scope* wird nicht beachtet, nur die Ausführungsreihenfolge der enthaltenen Aktivitäten wird beibehalten. Alle Schleifenkonstrukte ergeben ein Fragment, von dem aus alle Fragmente aufgerufen werden, die den Schleifenkörper enthalten. Bei einer Alternative (*If*) oder einem *Pick* sind diese in einem Fragment, von dem aus die Fragmente aufgerufen werden, die die einzelnen Pfade enthalten. Die Bedingung der Alternative ist entscheidend dafür, welche Fragmente aufgerufen werden.

Die erstellten Fragmente enthalten nur Basis-Aktivitäten, Alternativen, *Pick* oder Schleifen des Ursprungsprozesses. Diese Aktivitäten sind durch eine *Sequence* in jedem Fragment umschlossen. Die Fragmentierung wird in Kapitel 3.3 genauer betrachtet.

Nach der Fragmentierung kann mit dem ***dynamischen Deployment*** begonnen werden. Dazu wird die von der Fragmentierung erstellte frag-Datei benötigt, Informationen über die Infrastruktur und die Größe der Fragmente ausgelesen.
Die Server der Cloud werden unterteilt in:

- nicht verfügbare Server
- verfügbare Server
  - aber kein freier Speicher mehr
  - freier Speicher ist noch vorhanden

Ist ein Server verfügbar und frei, wird der gesamte Speicherplatz des Servers und der momentan freie Speicherplatz ermittelt. Die Größe eines Servers wird exemplarisch für Kriterien herangezogen, die bei der Auswahl eines Servers von Bedeutung sind. Der Server muss also über genügend Speichervolumen verfügen, um für die Ausführung des Fragments in Frage zu kommen. Die Größe ist nur von exemplarischer Bedeutung, da es praktisch nicht möglich ist, dass ein Server zu wenig Speichervolumen für die Ausführung eines Fragments eines scientific Workflows hat, da auch für die Ausführung des Gesamtprozesses genug Speicher zur Verfügung stand, dies aber wegen langen Rechenzeiten unpraktisch ist. Es ist denkbar, diese Größe durch ein anderes Kriterium oder sogar Mehrere zu ersetzen. Beispielsweise ist es möglich, durch Berechnungen den benötigten Arbeitsspeicher zu ermitteln und mit dem des Servers abzugleichen. Zudem ist es vorstellbar, den vorhandenen externen Speicher zu betrachten, um eine Entscheidung treffen zu können, wo Daten gespeichert werden sollen. Diese Möglichkeiten sprengen aber den

zeitlichen Rahmen dieser Arbeit, werden aus diesem Grund nicht eingehend betrachtet und bleiben für die zukünftige Forschung offen.

Sind Infrastruktur-Informationen, die frag-Datei und die Größen der Fragmente nun verfügbar, beginnt das Mapping der Fragmente auf die zur Verfügung stehenden freien Server. Dabei wird zuerst ein Server mit exakt passender Größe gesucht. Ist ein solcher nicht vorhanden, wird das Fragment auf den größten freien, verfügbaren Server gemappt. Ist kein Server frei und verfügbar, der groß genug ist, wird analysiert, ob es einen verfügbaren Server gibt, der aber beschäftigt ist. Ist dies der Fall, wird mit dem Deployment des Fragments gewartet, bis dieser Server wieder frei ist. Gibt es auch einen solchen Server nicht, bedeutet dies die Auslösung einer Neufragmentierung. Diese wird in Kapitel 3.3.3 genauer betrachtet.

Das Deployment geschieht für jedes einzelne Fragment und wird genau dann gestartet, wenn das vorhergehende Fragment in den Zustand „run“ wechselt. Das dynamische Deployment wird direkt nach dem Mapping gestartet, ohne davor beispielsweise den Server zu reservieren. Da das Deployment und die zugehörige Auswahl des Servers gestartet wird, wenn das Fragment, dessen Deployment direkt vor dem aktuellen Fragment durchgeführt wurde, in den Zustand „run“ wechselt, bedeutet dies im Gegenzug, dass bei Beginn der Ausführung eines Fragments, dieser Zustandswechsel das Zeichen für das Deployment des darauf folgenden Fragments ist. Das (dynamische) Deployment wird von einer stetigen Planung (siehe 2.4) übernommen, die sich darum kümmert, dass alle Fragmente ausgeführt werden. Falls dies während der Ausführung zu Problemen führt, löst diese Planung die Probleme und führt die restliche Ausführung entsprechend fort, so dass am Ende der gesamte Prozess korrekt und vollständig ausgeführt wurde. Probleme, die auftreten können, sind im folgenden Abschnitt beschrieben.

Ist ein Fragment auf einen Server gemappt worden und noch nicht im Zustand „run“, können Fehler auftreten. Es ist unter Anderem möglich, dass ein Server plötzlich nicht mehr verfügbar ist, weil er zum Beispiel abgestürzt ist, oder dass er aufgrund einer anderen Ausführung zu klein für die Ausführung des Fragments geworden ist. Im ersten Fall muss ein neuer Server zur Ausführung gefunden werden. Es müssen also die Informationen über die Infrastruktur beachtet werden und das Deployment des Fragments wird neu durchgeführt. Im zweiten Fall werden auch die Infrastrukturinformationen beobachtet und abgewartet, bis ein passender Server zu Verfügung steht.

Obwohl durch diese Beschreibung eigentlich schon der komplette Ablauf dargelegt ist, wurde hierbei aber noch kein Augenmerk auf das dynamische Deployment gelegt. Es kommt zum Tragen, wenn die Fragmente auf die Server verteilt werden müssen. Wenn zu einem Zeitpunkt für mehr als ein Fragment Server gesucht werden, beginnt die Suche nach einem Server bei dem größten Fragment. Hier wird angenommen, dass die Ausführungsdauer proportional zur Größe des Fragments ist. Also braucht die Ausführung eines großen Fragments länger, als die eines Kleinen. Dadurch erhält die Zuteilung eines Servers für große Fragmente mehr Bedeutung als für Kleine, da die Möglichkeit besteht, dass während der Ausführung der großen Fragmente ein passender Server für eventuell übrig gebliebene kleine Fragmente frei wird. Durch die verspätete Ausführung und die kurze Dauer eines kleinen Fragments verzögert sich die Ausführung des kompletten Workflows entweder gar nicht oder nur wenig. Bei der Auswahl eines passenden

Servers wird nun zuerst nach einem Server gesucht, der genau die geforderte Größe hat. Ist ein solcher nicht vorhanden, wird das Fragment auf den größten freien verfügbaren Server gemappt. Wird so bei der Vergabe der Server verfahren, bleiben im schlechtesten Fall kleine Fragmente übrig. Möglicherweise können für diese Fragmente noch Server gefunden werden, auf denen im gleichen Schritt zwar schon andere Fragmente gemappt wurden, wo aber noch ausreichend Speicherplatz für ein kleines Fragment übrig geblieben ist.

Im Folgenden werden die Fälle betrachtet, bei denen sich die ***Infrastruktur*** verändert. Eine Veränderung zieht, je nachdem, was sich verändert hat, unterschiedliche Maßnahmen nach sich. Die Infrastrukturinformationen beinhalten Angaben zu den verfügbaren Servern und zu der Größe der Server. Verändert sich etwas, wird entweder eine Neufragmentierung oder ein erneutes dynamisches Deployment ausgelöst. Dieser Sachverhalt ist in Abbildung 5 dargestellt.

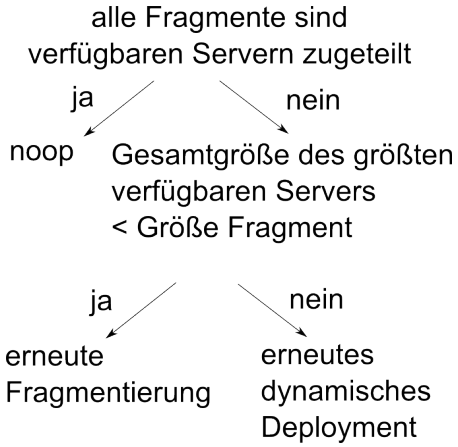

Abbildung 5: Auswirkung einer Änderung der Infrastruktur-Informationen: betrachtet werden die Verfügbarkeit der Server und die Änderung der Größe der Server.

Wenn eine Änderung der Infrastruktur vorliegt, muss als Erstes überprüft werden, ob alle Fragmente auf einen verfügbaren freien Server gemappt sind. Ist dies der Fall, hat die Änderung der Infrastruktur keine Auswirkungen. Gibt es jetzt aber ein Fragment, das keinem Server mehr zugeteilt ist, weil dieser soeben weggefallen ist, muss die Fragmentgröße mit der Gesamtgröße des Serverspeichers verglichen werden. Ist die Größe des Fragments größer als der gesamte Speicherplatz des größten Servers, so muss neu fragmentiert werden und dieses Fragment in kleinere Teile unterteilt werden. Ist dies nicht der Fall, genügt ein erneutes dynamisches Deployment und das serverlose Fragment wird auf einen anderen verfügbaren freien Server gemappt.

3.2 Berechnung der Datenkanten

Die Datenkanten zwischen den einzelnen Aktivitäten des BPEL-Prozesses müssen berechnet werden, um Datenübertragungsmengen zwischen Fragmenten unter einem be-

stimmten Schwellwert zu halten. Um die Datenkanten zu berechnen, wird ein bestehender Algorithmus benutzt. Er wird in Arbeit [10] beschrieben und ist in einer Diplomarbeit [11] implementiert. Die Idee wird hier kurz beschrieben. Die Datenkanten werden durch eine statische Analyse, die eine Tiefensuche durchführt, identifiziert, bei der Dead Path Elimination (DPE) berücksichtigt wird. Für die Berechnung muss der Workflow das Bernstein-Kriterium erfüllen, das besagt, dass in parallelen Zweigen nicht gleichzeitig Schreiber und Leser einer Variablen vorkommen dürfen. Um die möglichen Schreiber, die für das Erstellen der Datenkanten ausschlaggebend sind, zu berechnen, werden die Schreiber in Kategorien eingeordnet, die den möglichen Zuständen einer schreibenden Aktivität entsprechen. Diese drei Zustände sind im Folgenden aufgelistet:

- *möglicher Schreiber:* Ein möglicher Schreiber einer Variablen in einer Aktivität ist ein Schreiber, dessen Daten diese Aktivität erreichen können.
- *deaktivierter Schreiber:* Ein deaktivierter Schreiber ist ein Schreiber, dessen Daten von einem nachfolgenden Schreiber überschrieben werden. Der deaktivierte Schreiber kann wieder zu einem möglichen Schreiber werden.
- *ungültiger Schreiber:* Ein Schreiber wird zu einem ungültigen Schreiber, wenn der Wert, den er schreibt, immer von einem nachfolgenden Schreiber überschrieben wird.

Um den Zustand einer schreibenden Aktivität zu speichern, benötigt man eine Tabelle mit vier Spalten:

- *erste Spalte:* die Aktivität / der Link selbst
- *zweite Spalte:* die möglichen Schreiber
- *dritte Spalte:* die deaktivierten Schreiber
- *vierte Spalte:* Boolscher Wert, der angibt, ob die Aktivität tot sein könnte (ist immer in dieser Spalte enthalten, sobald die Auswertung das erste Mal ergibt, dass die Aktivität tot sein könnte).

Die ungültigen Schreiber müssen nicht gespeichert werden, da sie für die Bestimmung der möglichen Schreiber nicht mehr in Frage kommen und somit auch für die zu identifizierenden Datenkanten nicht mehr in Betracht gezogen werden müssen. Der Algorithmus erstellt eine Tabelle, in der alle möglichen Schreiber einer Aktivität enthalten sind. Abbildung 6 zeigt einen Graphen, an dem diese Analyse durchgeführt wird. Die Tabelle 3.2 ergibt sich durch diese Analyse. Die Aktivitäten 1 und 2 schreiben theoretisch in eine Variable x. Nach Ausführung der Aktivität 1 wird die explizite Transition Condition gesetzt. Wird aufgrund dieser Auswertung die Aktivität 2 nicht ausgeführt, ist nur die erste Aktivität ein Schreiber und die zweite Aktivität ist tot. Wurde Aktivität 2 aber ausgeführt, so ist Aktivität 1 ein deaktivierter Schreiber. Wenn Aktivität 6 x liest, kommen beide Schreiber als mögliche Schreiber in Frage, da das Join vor dieser Aktivität ein *OR*

ist. Dies bedeutet, dass Aktivität 6 auch ausgeführt werden kann, wenn nur die Transition Condition zwischen Aktivität 5 und 6 als *wahr* ausgewertet wird. Wurde die zweite Aktivität ausgeführt, ist sie der mögliche Schreiber. Ist hingegen nur die erste Aktivität ausgeführt worden, ist sie der mögliche Schreiber. Da nicht bestimmt werden kann, durch welchen Pfad die sechste Aktivität zur Ausführung angestoßen wurde, können auch keine Rückschlüsse gezogen werden, welche der beiden ersten Aktivitäten ausgeführt wurde. Somit müssen diese beiden Aktivitäten als mögliche Schreiber in Betracht kommen.

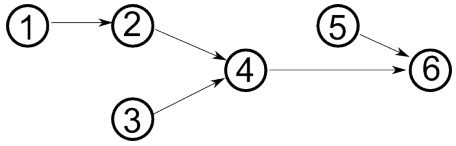

Abbildung 6: Prozess mit Join-Conditions: Aktivität 1 schreibt x (w_1^x), Aktivität 2 schreibt x (w_2^x), Aktivität 4 liest x (r_1^x), Aktivität 6 liest x (r_2^x). Die Verbindung zwischen Aktivität 1 und 2 hat eine explizite Transition Condition (link tc_1), alle anderen Verbindungen haben eine Default Transition Condition *wahr* (l). Der zweite Join ist ein *OR*.

Aktivität / Link	**Mögl. Schreiber**	**Deakt. Schreiber**	**evtl. tot**
Aktivität 1 (w_1^x)	null	null	false
tc_1 (zwischen A1 und A2)	w_1^x	null	false
Aktivität 2 (w_2^x)	w_1^x	null	true
l_1 (zwischen A2 und A4)	w_2^x	w_1^x	false
Aktivität 3 (a_1)	null	null	false
l_2 (zwischen A3 und A4)	null	null	false
Aktivität 4 (r_1^x)	w_2^x	w_1^x	false
l_3 (zwischen A4 und A6)	w_2^x	w_1^x	false
Aktivität 5 (a_2)	null	null	false
l_4 (zwischen A5 und A6)	null	null	false
Aktivität 6 (r_2^x)	w_1^x, w_2^x	null	false

Mit Hilfe dieser Information ist es nach der Fragmentierung möglich, die Menge der zwischen den Fragmenten zu übertragenden Daten zu erkennen. Dazu wurde eine Liste erstellt, die die Fragmente und ihre enthaltenen Aktivitäten umfasst. Die erzeugte Tabelle zeigt also alle Abhängigkeiten zwischen den Schreibern und den Aktivitäten. Dadurch ist eindeutig, zwischen welchen Fragmenten Daten übermittelt werden müssen.

3.3 Fragmentierung

Der hier verwendete Ansatz der Fragmentierung hat viele verwandte Ansätze in der Literatur. Diese werden im nächsten Abschnitt beschrieben, um eine Grundlage für den hier entwickelten Algorithmus zu schaffen. Der Algorithmus wird im darauf folgenden Unterkapitel genau dargelegt.

3.3.1 Einführung

Es gibt viele Arbeiten, die sich mit der Fragmentierung beschäftigen. Die Arbeiten [12], [10] und [13] untersuchen BPEL und leiten die Datenkanten her. Durch diese Herleitung entsteht BPEL-D. BPEL-D steht für BPEL, das Datenkanten enthält.

Die Algorithmen von den Arbeiten [12] und [10] arbeiten beide mit BPEL[3] (Sprache, die zur Spezifikation von ausführbaren Workflow-Modellen genutzt wird). Ziel der beiden Ansätze ist die Berechnung von Datenkanten. Dazu wird der BPEL-Prozess traversiert, eine statische Analyse auf den Daten durchgeführt und somit die Datenkanten bestimmt. In Ansatz [10] ist die Idee des Algorithmus beschrieben, wohingegen Arbeit [12] die Umsetzung beinhaltet. Diese Datenanalyse wird bei der Fragmentierung und anschließenden Planung dieses Ansatzes benötigt und verwendet.

In Arbeit [13] wird ein anderer Algorithmus zur Berechnung von Datenkanten vorgestellt. Diese Analyse kann auf BPEL angewendet werden. Der verwendete Algorithmus wird mit Hilfe eines Petri-Netzes verifiziert. Zu Beginn des Algorithmus wird ein CSSA (Concurrent Single Static Assignment)-basierter Graph aufgestellt. Dieser enthält durch die Transformation sowohl explizit den Kontroll- als auch den Datenfluss. Anschließend sammelt der Algorithmus kommunikationsrelevanten Datenfluss und weist jedem CSSA-Knoten eine Menge von Nachrichten-Abhängigkeiten zu.

Der Ansatz aus Arbeit [14] nimmt eine Fragmentierung eines Workflows anhand von Swimlanes vor. Der Algorithmus arbeitet mit BPEL-D (siehe Referenz [10]). BPEL-D ist einer Obermenge von BPEL. BPEL enthält Datenkanten zwischen Aktivitäten nur implizit, wohingegen bei BPEL-D explizite Datenkanten vorhanden sind. Ist der Workflow fragmentiert, sind Datenkanten zerbrochen worden, die zwischen Aktivitäten verliefen, die nun in verschiedenen Fragmenten sind. Diese Datenkanten werden durch einen Nachrichtenaustausch und zusätzliche FaultHandler zwischen den Fragmenten wieder hergestellt. Die Fragmentierung dieses Ansatzes wendet andere Fragmentierungsregeln als die Fragmentierung dieser Arbeit an. Zusätzlich zu den unterschiedlichen Fragmentierungsregeln verwendet der in Referenz [14] vorgestellte Ansatz für die Rekonstruktion zerbrochener Datenkanten immer eine Datenübermittlung basierend auf Nachrichtenaustausch. Der in dieser Arbeit angewendete Ansatz nutzt zwar auch Nachrichtenaustausch, dieser wird aber nur zum Anstoßen der Berechnung von Fragmenten benutzt, die beispielsweise in einem Schleifenfragment enthalten sind.

Die im Folgenden aufgeführten Arbeiten befassen sich mit der Fragmentierung bei dezentraler Ausführung oder etwas vergleichbarem.

Ansatz [15] fragmentiert ein Petri-Netz zur verteilten Ausführung. Außerdem wird eine dynamische Fragmentierung während der Ausführung vorgestellt. Die Fragmente ergeben sich durch die Zerlegung eines Petri-Netzes aufgrund der zugrunde liegenden Struktur. Es wird an jeder Verzweigung ein neues Fragment erstellt. Das Fragment, das den ersten Pfad der Verzweigung enthält, endet erst am Ende des gesamten Graphen. Die restlichen Pfade dieser Verzweigung ergeben jeweils ein Fragment, das bei der Zusammenführung der Äste, die sich geteilt haben, wieder endet. Es wird eine Kombination der statischen Fragmentierung und der dynamischen Fragmentierung zur Laufzeit vorge-

[3] http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html

schlagen. Die Arbeit betrachtet nur die einfachste Art eines Petri-Netzes, weil der Fokus dieser Arbeit auf der strukturellen Partitionierung liegt und beachtet keine Datenflüsse. Die Gemeinsamkeit zu dieser Arbeit ist die strukturelle Fragmentierung, der hier aber andere Regeln zugrunde liegen.

Ein weiterer Ansatz ist in Referenz [16] dargestellt. Hier wird ein BPEL-Prozess zur dezentralen Ausführung partitioniert. Die Aktivitäten des Prozesses werden in fixe und portable Knoten eingeteilt, die unter Beachtung bestimmter Regeln zusammengefasst werden. Diese Knoten können nun auf verschiedenen Servern, also nicht mehr zentral auf dem Hauptserver, ausgeführt werden. Hier sind die Parallelen zu diesem Ansatz deutlich. Der BPEL-Prozess wird unter Betrachtung der Struktur partitioniert, wobei der partitionierte Prozess auf verschiedenen Servern ausgeführt wird.

Die Idee von Referenz [17] zielt auch auf eine dezentrale Ausführung eines Workflows ab. Sie wird durch eine Fragmentierung erreicht, die die Koordinationslogik auf die verschiedenen Teilnehmer verteilt. Die dezentrale Ausführung wird durch die Verwendung von Tuplespace [18] umgesetzt. Der Einsatz von Tuplespace ermöglicht, dass alle Teile des Workflows Zugriff auf Daten haben, die für die Ausführung nötig sind. Die Verifikation des Ansatzes geschieht mit Hilfe von „executable Workflow Netzen“ (EWFN) und Petri-Netzen für die Modellierung. Das Augenmerk liegt hier auf der dezentralen Ausführung, was zu dem Ansatz dieser Arbeit passt, bei [17] aber durch Tuplespace ermöglicht wird.

Ansatz [19] nutzt Executable Workflow Networks (EWFN), um BPEL Prozesse verteilt und dezentral auszuführen. Der Workflow wird aufgeteilt, so dass es zur gegebenen Infrastruktur passt. Dieser Ansatz zielt auf die Orchestrierung von Workflows ab, also die Auswertung des Kontrollflusses und die Ausführung der Aktivitäten. Diese ist normalerweise zentral und wird hier dezentralisiert. Im Gegensatz zu dem hier vorgestellten Ansatz wird dieser Workflow also nicht unter Beachtung seiner Struktur aufgeteilt, sondern unter Gesichtspunkten, die die Teilprozesse gut auf die Infrastruktur abbilden. Durch diese Abbildung ist die dezentrale Ausführung gegeben.

[20] beschäftigt sich mit Workflows zur Modellierungzeit und Transaktionen. Bei diesem Ansatz wird kein großer Wert auf das zugrundeliegende Modell gelegt, sondern auf Transaktionen. Es wird versucht, eine optimale Stratifizierung von globalen Transaktionen zu erreichen. Eine Berechnung wird ausgeführt, die die Basisaktivitäten der globalen Transaktion in Straten gruppiert. Dies geschieht basierend auf den Eigenschaften der Transaktionen und der Ressourcen, die diese benutzen. Die Straten werden koordiniert, damit die Semantik der ursprünglichen Transaktion gewahrt wird. Das Prinzip dieses Algorithmus hat ein ähnliches Vorgehen wie der Ansatz dieser Arbeit, da die Koordination der Straten vergleichbar mit der Koordination der Fragmente ist.

3.3.2 Algorithmus

Der als BPEL-Prozess gegebene Workflow wird in Teilstücke zerlegt. Diese Teilstücke sind Teilprozesse des BPEL-Prozesses; alle Teilprozesse ergeben einen eigenständigen Prozess. Aus diesem Grund sind alle Teilprozesse, die im Folgenden als *Fragmente* bezeichnet werden, nach dem gleichen Schema aufgebaut. Der Definitionsteil des Prozesses

ist in jeder Fragmentdatei zu finden, der Dateiname, der darin enthalten ist, ergibt sich aus dem ursprünglichen Namen des Prozesses, der durch „Part_i“ ergänzt wird; wobei i die Fragmentnummer ist, die auch im Namen der Fragmentdatei zu finden ist. Der Name der Fragmentdatei lautet beim i-ten Fragment „Fragment_i.bpel“. Jede Fragmentdatei enthält eine Sequenz, in der alle BPEL-Aktivitäten dieses Fragments enthalten sind.

Da sich alle Fragmente durch die Zerlegung des Ursprung-Prozesses berechnen lassen, hängen alle Fragmente logisch zusammen. Um ein bei der Ausführung äquivalentes Verhalten zum Ursprungsprozess zu gewährleisten, wird die Navigation des Ursprungsprozesses auf die Ausführung der Fragmente übertragen. Die Navigation wird in zwei Dateien gespeichert. Der Inhalt dieser Dateien ist identisch. Die Erzeugung und das Format unterscheiden sich aber, weshalb beide Dateien benötigt werden. Die erste der beiden Dateien hat die Endung „frag“ und stellt ein Format speziell für den Export dar, welches für den Austausch von Fragmenten und deren Navigation bereitgestellt wird. Die zweite Datei hat die Endung „plan“ und verwendet ein Format, das ausschließlich zum internen Gebrauch bestimmt ist.

Die Dateien, die während der Fragmentierung mit der Endung „plan“ und „frag“ erstellt werden, enthalten eine Auflistung der erstellten Fragmente. Das Dateiformat ist XML, weshalb alle Fragmente unter dem Wurzelelement „Fragments“ zusammengefasst sind. Die Informationen, die über jedes Fragment gespeichert werden, sind Folgende:

- der Verzeichnispfad, an dem die Fragmentdatei zu finden ist
- die Vorgänger des Fragments
- die Nachfolger des Fragments
- die Aktivitäten (Art und Name), die in dem Fragment enthalten sind

Diese Datei wird im gleichen Verzeichnis des zu fragmentierenden Prozesses gespeichert, während für die erstellten Fragmente ein Ordner mit dem Namen „Fragments“in diesem Verzeichnis angelegt wird.

Um die Fragmente zu ermitteln, beginnt der Fragmentierungsalgorithmus am Anfang der Prozessdatei und besucht alle Aktivitäten nacheinander. Der Algorithmus arbeitet wie eine Tiefensuche, indem er zuerst alle in einer strukturierten Aktivität enthaltenen Aktivitäten betrachtet, bevor er die nachfolgende Aktivität besucht. Die Eigenschaften eines Fragments sind Folgende:

- es enthält keine anderen Fragmente, es sei denn sie werden von einem Fragment aus mit einem *invoke* aufgerufen
- es überlappt nicht mit einem anderen Fragment
- es enthält nur Basis-Aktivitäten, eine Alternative (*If*) oder Schleifenkonstrukt
- es beginnt
  - vor einem *If* oder *Pick*
  - vor einem *While* oder *Repeat Until*

  - vor einem *For Each*
  - bei der ersten Basis-Aktivität in einer strukturierten Aktivität

- es endet
  - vor Beginn einer strukturierten Aktivität
  - vor Beginn eines *If* oder *Pick*
  - vor Beginn eines *While* oder *Repeat Until*
  - vor Beginn eines *For Each*
  - wenn der Prozess keine folgenden Aktivitäten mehr enthält

Während der Fragmenterstellung wird anhand der Erstellungsreihenfolge der Fragmente deren Vorgänger und Nachfolger ermittelt. Während der Fragmenterstellung werden alle gleichzeitig erstellten Fragmente gespeichert. Werden die nächsten Fragmente erstellt, haben sie als Vorgänger die unmittelbar vorher gespeicherten Fragmente, die nun nicht mehr gespeichert werden müssen. Alle Fragmente, bei denen diese Vorgänger eingetragen werden, müssen als Nachfolger dieser Fragmente eingetragen werden. In Abbildung 7 sind Rechtecke zu sehen, die Fragmente darstellen.

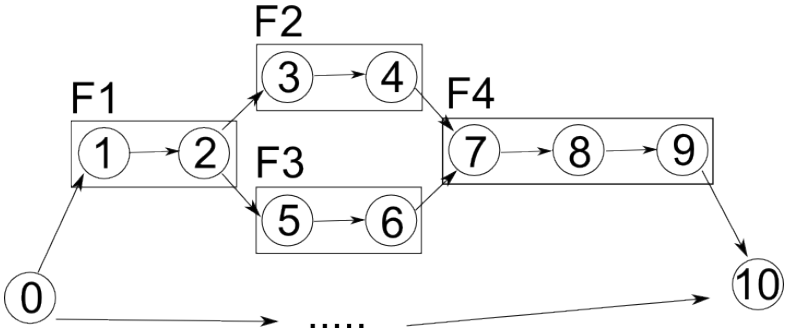

Abbildung 7: Workflow mit Fragmenten

Die folgende Tabelle enthält die Vorgänger und Nachfolger dieser Fragmente.

Fragment	**Vorgänger**	**Nachfolger**
Fragment 1	Fragment, das Knoten 0 enthält	Fragment 1, Fragment 2
Fragment 2	Fragment 1	Fragment 4
Fragment 3	Fragment 1	Fragment 4
Fragment 4	Fragment 2, Fragment 3	Fragment, das Knoten 10 enthält

Die Arbeitsweise des Fragmentierungsalgorithmus wird im Folgenden näher beleuchtet. Wenn er auf ***Scope, Sequence oder Flow*** trifft, sind dies strukturierte Aktivitäten, die mehrere Aktivitäten enthalten können. Wenn eine dieser Aktivitäten in einem Prozess vorkommt, wird diese nicht beachtet. Es wird die erste Basis-Aktivität in diesen Aktivitäten gesucht. Ist diese gefunden, beginnt ein Fragment, in dem alle Basis-Aktivitäten enthalten sind, bis wieder eine strukturierte Aktivität beginnt. Ein *Flow* hat in dieser Betrachtung eine Sonderstellung. Hier muss unterschieden werden, ob der *Flow* Links

enthält, oder nicht. Alle Aktivitäten, die keine Links enthalten, ergeben je ein Fragment. Aktivitäten, die Links enthalten, ergeben auch ein Fragment.

Bei ***If, Pick, While, Repeat Until oder For Each*** wird der Inhalt dieser Aktivitäten in jeweils ein Fragment übernommen. Aktivitäten, die in diesen Konstrukten enthalten sind, werden nach dem beschriebenen Algorithmus der jeweiligen Aktivität auch in Fragmente unterteilt, die aber durch ein *invoke* aus diesem Fragment heraus aufgerufen werden. Das *invoke* ist der Beginn einer synchronen Kommunikation. Dies bedeutet, dass alle Fragmente, die von diesem aus aufgerufen werden, um ein *receive* zu Beginn und ein *reply* am Ende des Fragments erweitert werden. Durch diese Erweiterung enthalten Fragmente mit solchen Konstrukten zwar andere Fragmente, der Aufruf wird aber von dem Konstrukt selbst übernommen.

Bei einem ***If*** muss für jeden möglichen Pfad ein Fragment erstellt werden, weil noch nicht vorhergesagt werden kann, welches der Fragmente ausgeführt werden wird. Aufgrund der Bedingung des *If* wird schließlich nur ein Pfad ausgewählt und somit nur ein Fragment ausgeführt. Die Erstellung eines Fragments im Falle einer Alternative ist im Folgenden zu sehen. In Folgenden ist das *If* im Ursprungs-Prozess dargestellt:

```
<if>
  <condition>
    bpel:getVariableProperty('shipRequest',
    'props:shipComplete')
  </condition>

  <sequence>
    <invoke name="invoke1"
      partnerLink="customer"
      operation="shippingNotice"
      inputVariable="shipNotice">
      <correlations>
        <correlation set="shipOrder" pattern="request" />
      </correlations>
    </invoke>
  </sequence>

  <else>
    <sequence>
      <assign name="assign2">
        <copy>
          <from>0</from>
          <to>$itemsShipped</to>
        </copy>
      </assign>
    </sequence>
  </else>
```

```
</if>
```

If nach der Fragmentierung. Das *If* ist in einem Fragment enthalten.

```
<if>
  <condition>
    bpel:getVariableProperty('shipRequest',
        'props:shipComplete')
  </condition>
  <sequence>
    <invoke>
      Fragment_i
    </invoke>
  </sequence>

  <else>
    <sequence>
      <invoke>
        Fragment_i+1
      </invoke>
    </sequence>
  </else>
</if>
```

Pick wird wie ein *If* behandelt. Für jeden möglichen Pfad muss ein Fragment erstellt werden, da zu der Zeit, wenn die Fragmente erstellt werden, noch nicht klar ist, welcher Pfad ausgeführt werden wird. Zur Ausführungszeit wird aber nur ein Fragment tatsächlich ausgeführt, die anderen können ignoriert werden.

Repeat Until und While sind Schleifen, die in BPEL als ein Konstrukt betrachtet werden. Somit sind die Eigenschaften eines gerichteten azyklischen Graphen gewahrt, die durch eine Schleife zerstört worden wären. Als Folge dieser Interpretation dieser Konstrukte wird für jedes ein Fragment erstellt. Der Unterschied zwischen den beiden Schleifenarten, der darin liegt, dass *Repeat Until* (im Gegensatz zu *While*) auf jeden Fall mindestens einmal ausgeführt wird, fällt bei der Fragmentierung nicht ins Gewicht. Die Bedingung, die prüft, ob die Schleife ausgeführt wird, befindet sich im Fragment, wodurch es nicht vorkommen kann, dass ein Fragment gar nicht ausgeführt wird, da die Bedingung immer überprüft werden muss. Ist die Abbruchbedingung zu Beginn wahr, wird zwar die Schleife nicht ausgeführt, das Fragment musste aber trotzdem den Wert der Bedingung berechnen. Also können die beiden Schleifenarten äquivalent behandelt werden. Aktivitäten, die in diesem Konstrukten enthalten sind, werden auch in Fragmente unterteilt. Der Aufruf der Fragmente geschieht durch ein *invoke* nach dem gleichen Prinzip, das bei einem *If* angewendet wird. Stellvertretend für alle Schleifen ist im Folgenden die Erstellung eines Fragments zu sehen, wenn der Algorithmus auf ein *while* trifft. Im Folgenden ist das *While* im Ursprungs-Prozess zu sehen:

```
<while>
  <condition>
    $itemsShipped
    &lt;
    bpel:getVariableProperty('shipRequest',
    'props:itemsTotal')
  </condition>

  <sequence>
    <invoke name="invoke2"
      partnerLink="customer"
      operation="shippingNotice"
      inputVariable="shipNotice">
    <correlations>
      <correlation set="shipOrder"
        pattern="request" />
      </correlations>
    </invoke>

    <assign name="assign4">
       <copy>
         <from>
           $itemsShipped
           +
           bpel:getVariableProperty('shipNotice',
           'props:itemsCount')
         </from>
         <to>$itemsShipped</to>
       </copy>
     </assign>
   </sequence>
</while>
```

While nach der Fragmentierung. Das *While* ist in einem Fragment enthalten.

```
<while>
  <condition>
    $itemsShipped < bpel:getVariableProperty('shipRequest',
    'props:itemsTotal')
  </condition>
  <sequence>
    <invoke>
      Fragment_i
    </invoke>
  </sequence>
```

</while>

For Each hebt sich von den beiden vorigen Schleifenarten ab. Der Unterschied besteht darin, dass Schleifendurchgänge auch parallel ausgeführt werden können. Zusätzlich können alle Schleifendurchgänge auch sequenziell, d. h. nacheinander ausgeführt werden. Bei einer sequenziellen Ausführung wird ein Fragment erstellt, das eine analoge Struktur zu allen anderen Schleifen besitzt.

In diesem Fragment werden Fragmente mit *invoke* aufgerufen, die in der Schleife ausgeführt werden sollen. Bei der parallelen Ausführung wird berechnet, wie viele Fragmente entstehen müssen, indem der Anfangswert der Schleife von dem Endwert, bei dem die Schleife abgebrochen werden soll, abgezogen und Eins addiert wird. Der entsprechende Zählerwert wird jedem Fragment mitgegeben. Alle diese Fragmente können nun parallel und sequenziell ausgeführt werden. Nach jedem ausgeführten Fragment wird ein Fragment aufgerufen, das die *Completion Condition* enthält. Ist diese erfüllt, werden die restlichen Fragmente der Schleife nicht weiter ausgeführt.

Trifft der Algorithmus auf Basis-Aktivitäten nimmt er diese ohne jegliche Änderung in das aktuelle Fragment auf. Basisaktivitäten sind neben Alternativen (*If*) und Schleifenkonstrukten die einzige Art von Aktivitäten, die aus dem Ursprungsprozess übernommen werden.

Eine exemplarische Fragmentierung ist im Folgenden in Abbildung 8 zu sehen. Es wird der „shippingService“-Prozess fragmentiert, der unter in der BPEL Spezifikation[4] zu finden ist. Zusätzlich sind im Anhang (A) die wichtigsten Auszüge aufgeführt. Es sind alle Aktivitäten zu sehen, die im Ursprungsprozess enthalten sind. Durch die Fragmentierung,

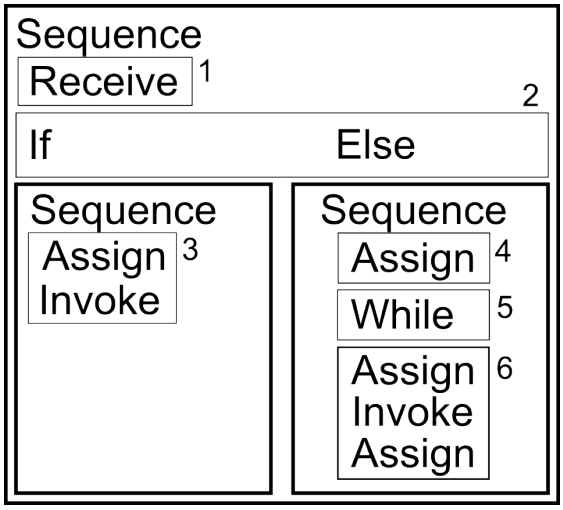

Abbildung 8: Struktur des Prozesses shippingService

bei der *Sequenzen* und *Flows* nicht in die Fragmente übernommen werden, sind diese auch nicht in den dünnen Rechtecken enthalten. Sie sind die erstellten Fragmente, die

[4]http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html

durchnummeriert sind. Das Fragment 2, das die Alternative (*if* und *else*) enthält, ruft im jeweiligen Zweig das Fragment mit einem *invoke* auf, das im Bild darunter abgebildet ist (Fragment 3 oder 4). Das Fragment 5, das das *while*-Konstrukt enthält, ruft das darunter liegende Fragment 6 ebenfalls durch ein *invoke* auf.

In diesem Beispiel sieht die Auflistung der Vorgänger und Nachfolger folgendermaßen aus:

Fragment	**Vorgänger**	**Nachfolger**
Fragment 1	keine	Fragment 2
Fragment 2	Fragment 1	keine
Fragment 3	if	keine
Fragment 4	if	Fragment 5
Fragment 5	Fragment 4	Fragment 5
Fragment 6	while	keine

Als Vorgänger taucht das jeweilige Konstrukt auf (Null wäre nicht geeignet, da diese Fragmente sonst mit den Startfragmenten verwechselt werden könnten). Nachfolger des jeweiligen Konstrukts (Schleife, *If*, *Pick*) ist das Fragment, das auf dieses folgt. Die Fragmente innerhalb eines solchen Konstrukts werden nicht in der Vorgänger-/Nachfolgerliste beachtet, da sie direkt aus dem Fragment des Konstrukts aufgerufen werden.

Aktivitäten, die nicht betrachtet werden, sind Isolated Scopes, Message Exchange Handling, Error Handling, Compensation Handlers, Fault Handlers, Termination Handlers, Event Handlers.

Aufgrund des Bernstein-Kriteriums, das für den zu fragmentierenden Workflow gelten muss, ist garantiert, dass zwischen den parallelen Fragmenten, die bei der Fragmentierung eines *Flows* mit Aktivitäten ohne Links oder eines parallelen *For Each* entstehen, keine Kontroll- oder Datenflusskanten verlaufen. Diese Kanten kann es nur zwischen Fragmenten geben, die aufeinander folgen. Ein anderer Schluss daraus ist, dass die Verzweigung nach Knoten 2 in Abbildung 7 wieder zusammengehen muss, bevor die Verzweigungen von Knoten 0 wieder zusammengeführt werden kann. Dadurch ist es möglich so zu fragmentieren, dass Fragmente nach einer Verzweigung beginnen und vor einer Zusammenführung wieder enden. Einige Fragmente sind in Abbildung 7 zu sehen. Sie umfassen die enthaltenen Knoten und sind durch Rechtecke dargestellt.

Die ***Datenübermittlung zwischen den Fragmenten*** kann während der Fragmenterstellung noch nicht beachtet werden. Die Daten müssen zu den Servern übermittelt werden, auf denen die Fragmente zu finden sind. Da zum Zeitpunkt der Fragmentierung diese Zuordnung noch nicht durchgeführt wurde, kann auch noch nicht bestimmt werden, wohin die Daten übermittelt werden müssen. Erst wenn diese Zuordnung gemacht wird, kann auch die Übermittlung der Daten geplant werden. Dies hat zur Folge, dass die Datenübermittlung erst während dem dynamischen Deployment vorgenommen werden kann. Aus diesem Grund wird während des dynamischen Deployments überprüft, ob zwischen Fragmenten auf unterschiedlichen Servern die Datenübertragungsmenge den vorgegebenen Schwellwert übersteigt. Ist dies der Fall, werden beide Fragmente auf einem Server ausgeführt, um die Übermittlung der Datenzwischen verschiedenen Servern zu vermeiden.

3.3.3 Neufragmentierung

Eine Neufragmentierung wird benötigt, wenn ein Fragment, das schon einem Server zugeordnet war, nun keinem Server mehr zugeordnet ist und die Größe des Fragments größer ist, als die Gesamtgröße des größten Servers. Würde nur nach einem neuen Server gesucht werden, würde aufgrund der Größe keiner gefunden werden. Da das Fragment also zu groß für die verfügbaren Server ist, muss es zerlegt werden, damit die entstehenden Teile den Servern zugeordnet werden können.

Zu Beginn der Neufragmentierung muss überprüft werden, ob das Fragment überhaupt weiter zerlegt werden kann. Dies wird anhand der enthaltenen Aktivitäten entschieden. Handelt es sich um ein *If*, *Pick*, *While*, *Repeat Until* oder *For Each*, ist nur eine Aktivität in diesem Fragment enthalten, die durch ein oder mehrere *invokes* alle auszuführenden Aktivitäten des Konstrukts aufruft. Solche Aktivitäten können nicht weiter aufgeteilt werden und die Neufragmentierung kann nicht ausgeführt werden. Es gibt nun keine andere Möglichkeit als zu warten, bis wieder ein Server verfügbar wird, der groß genug ist, um dieses Fragment auszuführen.

Ein Fragment kann weiter zerlegt werden, wenn die enthaltenen Aktivitäten Basisaktivitäten sind. Diese werden durch die vorhergehende Fragmentierung sequenziell ausgeführt. Deshalb ist es möglich, die Aktivitäten in kleinere Fragmente zu unterteilen, die wiederum sequenziell ausgeführt werden müssen. Wenn es bei dieser erneuten Fragmentierung darum geht, wie viele Fragmente aus diesem einen erstellt werden sollen, können die Größen des Fragments und des größten verfügbaren Server herangezogen werden. Ist beispielsweise das Fragment genau doppelt so groß, wie der größte verfügbare Server, müssen drei neue Fragmente erstellt werden. Auf den ersten Blick könnte man denken, dass zwei Fragmente ausreichen würden. Dass dies aber nicht genug Fragmente sind, wird deutlich, wenn man bedenkt, dass jedes der neuen Fragmente wieder ein eigenständiger Prozess mit eigenen Prozessdefinitionen sein muss. Daher haben zwei Fragmente, die aus einem entstanden sind, nicht die Hälfte der Größe des ursprünglichen Fragments, sondern sind etwas größer.

Ist nun das ursprüngliche Fragment aufgeteilt, müssen die „plan“ und „frag“-Datei angepasst werden. Alle Fragmente, die bis zu diesem Zeitpunkt erfolgreich ausgeführt wurden, müssen entfernt werden, damit sie nicht erneut ausgeführt werden und die neu erstellten Fragmente müssen hinzugefügt werden, damit der Prozess korrekt ausgeführt werden kann. Anschließend muss die Planung auf Basis der neu erstellten „plan“-Datei neu angestoßen werden.

3.4 Dynamisches Deployment

Bei der Fragmentierung ist eine Datei mit der Endung „plan“ erstellt worden. Diese Datei ist Grundlage für die Planung. Sie enthält die gleichen Informationen wie die frag-Datei, ist aber für den internen Gebrauch aufbereitet. Die Informationen, die aus dieser Datei gebraucht und ausgelesen werden müssen, sind die Vorgänger und Nachfolger. Zusätzlich werden weitere Informationen benötigt. In der folgenden Auflistung sind alle

Informationen aufgelistet, die für die Planung notwendig sind.

- Fragmentnummer und -größe
- Servernummer
- freier und gesamter Speicherplatz jedes Servers
- Anzahl der CPUs

Die Aufgaben der Planung sind:

- Fragmentnummer und -größe ermitteln und speichern
- Serverinformationen (siehe obere Auflistung) ermitteln und speichern
- Interessante Informationen der plan-Datei speichern (beispielsweise Vorgänger, Nachfolger)
- Mapping der Fragmente auf die aktuell verfügbaren Server
- Dynamisches Deployment der Fragmente

Die zwei Hauptaufgaben des dynamischen Deployments sind das *Mapping* und das *Deployment*. Das *Mapping* hat die Aufgabe, alle Fragmente den verfügbaren Servern zuzuteilen. Dazu wird die Größe eines jeden Fragments mit dem freien Speicherplatz der verfügbaren Server verglichen. Zuerst wird versucht, einen Server passender Größe zu finden. Existiert dieser nicht, wird der Server mit dem größten freien Speicherplatz gesucht und das Fragment diesem Server zugeteilt. Die Zuordnung der Server- und Fragmentnummern wird in einer Tabelle gespeichert. Zusätzlich wird die Größe des Fragments von der Servergröße abgezogen. Ist auf diesem Server noch Speicherplatz verfügbar, ist es möglich, dass weitere Fragmente diesem Server zugeteilt werden. Bei dieser Zuteilung ist es aber wichtig, dass die Anzahl der Fragmente, die parallel ausgeführt werden sollen, nicht die Anzahl der CPUs auf dem Server übersteigen, da sie sonst nicht mehr parallel ausgeführt werden können.

Das dynamische Deployment führt das *Deployment* und die gleichzeitige Instantiierung der Fragmente aus. Zu Beginn werden alle Fragmente ausfindig gemacht, die keinen Vorgänger haben. Von diesen Fragmenten werden nun die Nachfolger ermittelt, deren dynamisches Deployment im nächsten Schritt ansteht. Dieser Vorgang wird so lange ausgeführt, bis alle Fragmente ausgeführt wurden. Dies ist der Fall, wenn es kein Fragment mehr gibt, das noch einen Nachfolger hat. Wenn das Deployment für ein Fragment gestartet wird, muss davor eine Deployment-Einheit erstellt worden sein. Diese enthält ein Betriebssystem, eine Engine und das Fragment selbst. Diese Deployment-Einheit wird auf den Server kopiert, der während des Mappings ausgesucht wurde. Ist die Einheit kopiert, wird sie instantiiert, sobald alle Vorgänger-Fragmente erfolgreich ausgeführt wurden. Ist die Einheit das erste Fragment, das ausgeführt werden soll, kann es sofort instantiiert und und die Berechnung gestartet werden.

Da erst während der Planung diese Zuordnung der Fragmente zu den verfügbaren Servern gemacht wird, kann auch erst zu diesem Zeitpunkt die Datenübermittlung geplant werden. Hierzu werden alle Variablen, die übermittelt werden müssen, im Planer zwischengespeichert, damit sie bei der Instantiierung der Fragmente mit übergeben werden können. Werden die Daten von einem anderen Fragment benötigt, werden die Werte dieser Variablen dem Fragment übermittelt. Ist die Planung beendet und somit der Prozess komplett ausgeführt, werden diese Daten gelöscht.

Bevor das dynamische Deployment ausgeführt wird, wird überprüft, ob die Datenübertragungsmenge über einen bestimmten Schwellwert hinausgeht. Ist dies der Fall und die betroffenen Fragmente sind auf verschiedene Server gemappt worden, muss anders gemappt werden, nämlich so, dass diese Fragmente auf dem gleichen Server ausgeführt werden. So ist das Problem der zu großen Datenübertragungsmenge zwischen Fragmenten auf unterschiedlichen Servern behoben, da nun die Daten auf dem gleichen Server gebraucht werden. In diesem Fall werden die Daten, die nur von den Fragmenten auf diesem Server benötigt werden, nur dort zwischengespeichert.

3.4.1 Wiederholtes Dynamisches Deployment

Das erneute dynamische Deployment wird ausgelöst, wenn ein Fragment einem Server zur Ausführung zugeordnet war und dieser Server nun nicht mehr verfügbar ist. Um zu ermitteln, ob in dem spezifischen Fall das erneute dynamische Deployment die richtige Wahl ist, wird die Größe des Fragments mit der Gesamtgröße des größten verfügbaren Server verglichen. Ist das Fragment größer, muss es zerteilt werden und ein erneutes dynamisches Deployment würde hier nichts nützen, da kein Server für das Fragment gefunden werden würde. In diesem Fall muss eine Neufragmentierung gestartet werden. Diese ist in Kapitel 3.3.3 beschrieben.

Ist das Fragment aber kleiner als die Gesamtgröße des größten verfügbaren Servers reicht ein erneutes dynamisches Deployment aus. Das Mapping und alle Schritte, die auf das Mapping folgen, werden zum wiederholten Mal ausgeführt. Somit ist sichergestellt, dass das Fragment einem neuen Server zugeteilt wurde. Der Unterschied zu dem dynamischen Deployment, das zu Beginn direkt nach der Fragmentierung ausgeführt wurde, liegt darin, dass hier nicht für alle Fragmente ein Server gesucht wird, sondern nur für das Fragment, das keinem Server mehr zugeteilt ist.

3.5 Diskussion

Die Fragmentierung und das dynamische Deployment werden aufgrund von Kriterien durchgeführt, die in der Theorie zwar sinnvoll sind, in der Praxis aber durch praxisnahere Kriterien ersetzt werden können. Beispielsweise kann der Server ausgewählt werden, indem die Lage der zur Berechnung benötigten Daten mit in die Auswahl einbezogen wird. Ein anderes Auswahlkriterium kann die Anzahl der CPUs sein. Müssen mehrere parallele Fragmente Daten zu einem gemeinsamen nachfolgenden Fragment übertragen, wobei die Datenübertragungsmenge zu groß ist, müssen alle Fragmente auf einem Server ausgeführt werden. Hierbei kommt nur ein Server für die optimale Ausführung in Frage,

der genug CPUs hat, um alle parallelen Fragmente echt parallel auszuführen. Zusätzlich muss die Größe des Arbeitsspeichers so groß sein, dass dadurch eine schnelle Berechnung der Fragmente garantiert werden kann. Darüber hinaus kann bei der Auswahl von mehreren Servern beachtet werden, dass diese in verschiedenen Clustern liegen. Dies erhöht die Quality of Services (QoS).

Die jetzige Umsetzung kann noch verbessert werden, indem die Datenübertragung anders gestaltet wird. Die optimale Lösung wäre, dass die Daten direkt von dem Server auf dem sie manipuliert oder erzeugt wurden auf den Server übermittelt werden, auf dem sie von einem Nachfolger-Fragment benötigt werden. Dieser Ansatz lässt sich wegen des dynamischen Deployments nicht ohne Hindernisse umsetzen. Da die Manipulation oder Erzeugung der Daten schon fertiggestellt sein kann, bevor das Nachfolgerfragment, das die Daten benötigt noch keinem Server zugeordnet ist und somit nicht feststeht, wohin die Daten übermittelt werden sollen. Eine mögliche Lösung wäre, dass das Nachfolgerfragment weiß, auf welchen Server die Vorgänger-Fragmente zu finden sind und sich die Daten selbst besorgt. Da ein Schwellwert für die Datenübertragung einmal festgelegt wird, muss dies für jede Cloud neu festgelegt werden, weil eventuell andere Datenübertragungsmengen möglich sind. Da dies unflexibel und umständlich ist, könnte die Datenübertragungsmenge dynamisch nach der verfügbaren Bandbreite berechnet werden. Des Weiteren ist es denkbar, bei der Fragmentierung die Fragmente unter zusätzlicher Berücksichtigung der Servergrößen zu erstellen. Dazu muss bekannt sein, wie viel Speicherplatz eine Basisaktivität durchschnittlich hat. Mit dieser Information kann die voraussichtliche Größe des Fragments berechnet werden und das Fragment beendet werden, wenn es sonst zu groß für den größten Server wird.

Zusammenfassend ist die Fragmentierung sinnvoll, da das Fragment auf dem Server dem bzw. einem Web Service, der aufgerufen wird, ausgeführt werden kann. Dies ist sinnvoll, wenn die Datenübertragung zwischen einzelnen Fragmenten kostengünstiger ist als der Aufruf des Web Services. Durch die Fragmentierung werden in die Ausführung des scientific Workflows Zwischenschritte eingefügt. Dadurch wird es möglich, Zwischenergebnisse der Berechnung zu kontrollieren. Bei Fehlern hat dies den Vorteil, dass die Berechnung nicht komplett neu ausgeführt werden muss, sondern bei dem letzten Zwischenergebnis wieder gestartet werden kann. Ist ein Fehler im Algorithmus vorhanden, der korrigiert werden soll, kann er im betroffenen Fragment geändert werden. Um die Berechnung erneut durchzuführen, kann auch in diesem Fall die Ausführung bei dem geänderten Fragment gestartet werden.

Durch die Verwendung der stetigen Planung wird das Deployment dynamisch gestaltet. Es ist möglich, auf Änderungen sofort zu reagieren und dies ohne jeglichen menschlichen Eingriff. Wenn diese Planung mit anderen Ansätzen kombiniert wird, die mit Fehlerfällen des Prozesses umgehen können, kann ein scientific Workflow automatisch und ohne menschliche Überwachung ausgeführt werden.

4 Implementierung

Die Arbeit dieser Diplomarbeit wird in zwei Eclipse-Plugins umgesetzt. Als Erstes wird die Benutzerschnittstelle und die Cloud Test-Umgebung betrachtet. Anschließend werden die Implementierungsdetails der Fragmentierung und der Planung dargelegt.

4.1 Benutzerschnittstelle

Die Benutzerschnittstelle ist Eclipse. Eclipse ist eine Entwicklungsumgebung, die erweiterbar ist. Das Herz von Eclipse besteht nur aus einer Plattform, die die grundlegende Funktionalität für die Implementierung bereitstellt. Um weitere Funktionen bereitstellen zu können, gibt es das Konzept der Plugins. Möchte man Zusatzfunktionalitäten nutzen, die bereits in Plugins zur Verfügung stehen, müssen die gewünschten Plugins nur in Eclipse integriert werden. Zudem ist es möglich, selbst ein Plugin zu erstellen, das Funktionalität anbietet, die es noch nicht gibt.

Diese Benutzerschnittstelle wird hier um Funktionalität erweitert, die in zwei Plugins enthalten ist. Ein Plugin ist folgendermaßen aufgebaut: Es gibt eine Extension (Erweiterung), die zu dem von Eclipse angebotenen Extension Point (Erweiterungspunkt) passt. Dies sind definierte Punkte, an denen es möglich ist, Eclipse zu erweitern. Das Plugin setzt an diesem Punkt an und stellt neue Funktionalität zur Verfügung. Soll es darüber hinaus auch noch möglich sein, das Plugin selbst zu erweitern, besteht auch die Möglichkeit, dass das Plugin Extension Points anbietet. Ein Beispiel für die Erweiterung von Eclipse durch einen Extension Point ist das Kontextmenü. Dieses kann durch neue Menüeinträge erweitert werden, die durch Extensions beschrieben werden und zu dem zugehörigen Extension Point passen.

Um die Funktionalität für die Fragmentierung und die Planung einzubringen, wird der Extension Point „org.eclipse.ui.popupMenus“ genutzt. Durch einen Rechtsklick auf die BPEL-Datei öffnet sich ein Kontextmenü, das den Menüpunkt „Fragmentation“ enthält. Wird dieser ausgewählt, startet die Fragmentierung. Die Planung hingegen wird ausgelöst, wenn der Menüpunkt „Planning“ im Kontextmenü der bei der Fragmentierung erstellten plan-Datei ausgewählt wird.

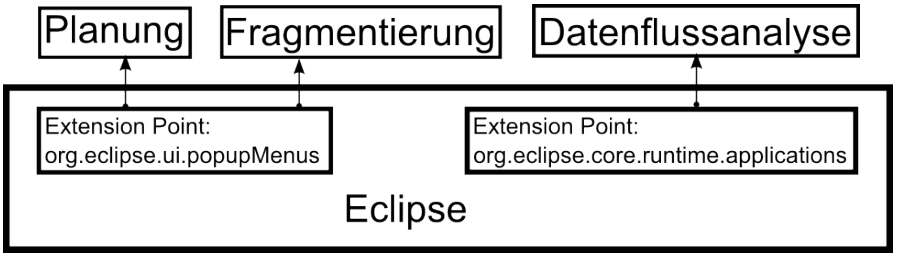

Abbildung 9: Erweiterung von Eclipse

4.2 Cloud Test-Umgebung

Eucalyptus [21] ist eine Open Source Anwendung, um eine Cloud zu simulieren. Sie setzt auf einem Hypervisor auf.

Ein Hypervisor ist eine Software zur Erstellung und Verwaltung von virtuellen Maschinen. Es gibt zwei verschiedene Virtualisierungsmöglichkeiten. Die erste Möglichkeit ist die Paravirtualisierung. Hier wird die Virtualisierung dadurch erreicht, dass die zur Verfügung stehenden Ressourcen unter den virtuellen Maschinen aufgeteilt werden, und jede virtuelle Maschine so arbeitet, als ob sie die Ressourcen für sich alleine hätte, obwohl sie diese noch mit den anderen virtuellen Maschinen teilt. Bei der Vollvirtualisierung werden die Ressourcen wirklich so unter den virtuellen Maschinen aufgeteilt, dass jede virtuelle Maschine ihre eigenen Ressourcen besitzt. Je nach Rechner, auf dem die Virtualisierung umgesetzt werden soll, wird eine Virtualisierungsmöglichkeit ausgewählt. Besitzt ein Rechner beispielsweise nur eine CPU, ist nur die Paravirtualisierung möglich, da bei einer Vollvirtualisierung für eine zweite virtuelle Maschine keine CPU mehr zur Verfügung stehen würde. Unabhängig davon, welche Virtualisierungsmöglichkeit gewählt wird, ist der Hypervisor für die Zuteilung der Ressourcen an die virtuellen Maschinen zuständig. Er erstellt und löscht sie, kann sie hochfahren, herunterfahren oder anhalten und sie nach einem Halt fortfahren lassen. Der Hypervisor verwaltet die Ressourcen, die auf einem Rechner zur Verfügung stehen und teilt sie den laufenden virtuellen Maschinen je nach Bedarf zu. Dieser Mechanismus simuliert die Präsenz von mehreren physikalischen Rechnern, obwohl nur ein physikalischer Rechner vorhanden ist. Der Hauptrechner interagiert mit dem Hypervisor und wird „dom0“ genannt. Zur Kommunikation über „ssh“ zwischen den virtuellen Maschinen und dem Hauptrechner werden die Schnittstellen der Brücke „br0“ und „eth0“ benutzt. Dabei befindet sich „br0“ auf dem Hauptrechner und „eth0“ ist die dazu passende Schnittstelle auf jedem einzelnen Knoten.

In Abbildung 10 ist der Virtualisierungsmanager des Hypervisors Xen zu sehen. Es sind alle virtuellen Maschinen mit ihren Namen aufgelistet, es ist zu sehen, wie viel Prozent des zugeteilten Speichers und der CPU von den Maschinen genutzt wird. Über diese Oberfläche ist es möglich, die virtuellen Maschinen zu starten, Details über sie zu erfahren, neue virtuelle Maschinen zu erstellen und bestehende zu löschen. Neben dem Hypervisor Xen ist KVM ein häufig verwendeter Hypervisor. Eucalyptus baut auf ihm auf.

Mit Eucalyptus ist es möglich, eine Cloud zu simulieren, die Infrastruktur zur Verfügung stellt. Dies ist möglich, da als Voraussetzung für Eucalyptus mit Hilfe eines Hypervisors virtuelle Maschinen aufgesetzt sein müssen. Diese virtuellen Maschinen werden von Eucalyptus genutzt, um ein oder mehrere Cluster zu erstellen, die eine Cloud darstellen. Durch ein Kommandozeilenwerkzeug kann mit den Maschinen gearbeitet werden, wie es auch bei einer realen Cloud der Fall ist. Die Anwendung bietet die Möglichkeit, Informationen über die zur Verfügung stehenden Ressourcen zu erhalten. Eucalyptus besteht aus fünf Komponenten, die im Folgenden aufgezählt und beleuchtet werden.

- *Cloud Controller:* Bietet Web-Interfaces an, bearbeitet Anfragen des Administrators oder von Benutzern, führt Ressourcen-Zuordnungen durch und verwaltet die

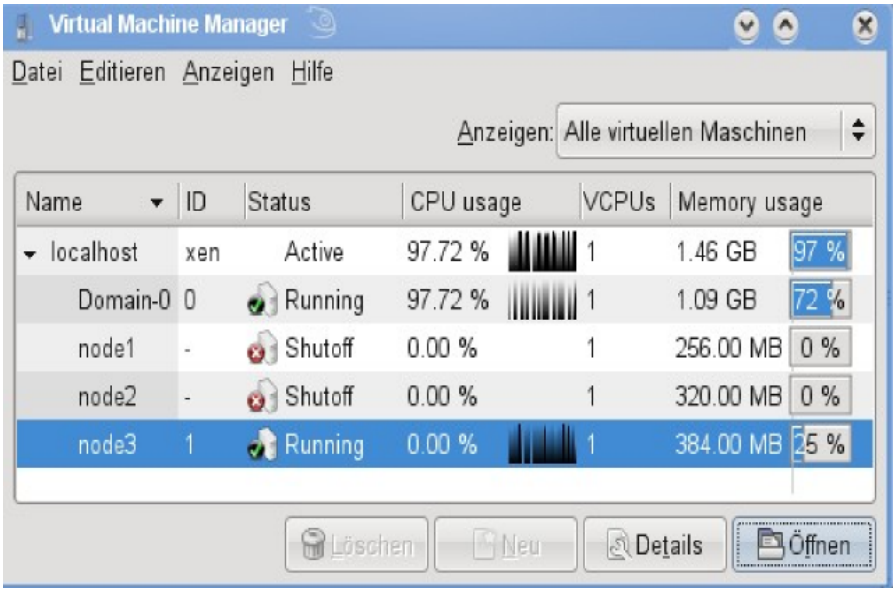

Abbildung 10: Virtualisierungsmanager von Xen

Accounts.

- *Walrus:* Implementiert eine bucket-basierte Speicherung, die inner- und außerhalb der Cloud verfügbar ist.
- *Cluster Controller:* Jedes Cluster benötigt einen Cluster Controller, der das Scheduling auf der Cluster-Ebene und die Netzwerkkontrolle übernimmt. Eine Cloud kann mehrere Cluster enthalten, die Knoten enthalten. Es kann mehrere Cluster geben, um die Quality of Service (QoS) zu steigern.
- *Storage Controller:* Jedes Cluster benötigt einen Storage Controller, der für die blockweise Speicherung verantwortlich ist.
- *Node Controller:* Auf jedem Knoten muss ein Node Controller vorhanden sein, der den Hypervisor kontrolliert. Jeder Knoten muss in einem Cluster der Cloud enthalten sein.

Zum Testen dieser Arbeit wird eine Maschine verwendet, auf der ein Hypervisor und Eucalyptus installiert ist. Es wird eine Single-Cluster Installation benutzt, bei der alle Komponenten - mit Ausnahme des Node Controllers - auf einer Maschine platziert sind. Die virtuellen Maschinen, die als Knoten (engl. Nodes) bezeichnet werden, haben nur einen Node Controller. In Abbildung 11 ist die Architektur des zum Testen benutzten Rechners zu sehen. Es wurden drei virtuelle Maschinen unterschiedlicher Größe erstellt. Sie werden durch Eucalyptus verwaltet. Ihre Daten sind im Folgenden aufgelistet:

- Node1

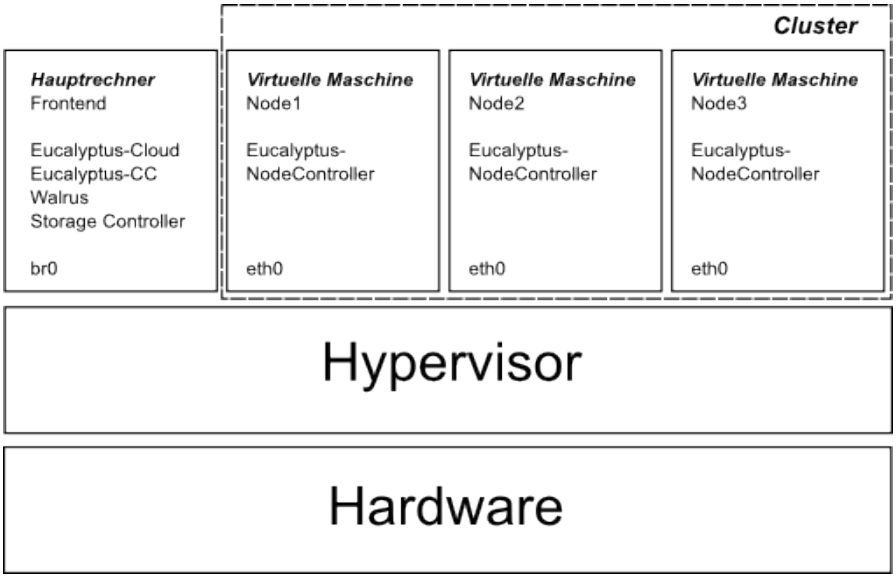

Abbildung 11: Rechner-Architektur für Tests

  - Hauptspeicher: 8 GB
  - RAM: 256 MB
  - Anzahl virtueller CPUs: 1

- Node2
  - Hauptspeicher: 8 GB
  - RAM: 320 MB
  - Anzahl virtueller CPUs: 1

- Node3
  - Hauptspeicher: 8 GB
  - RAM: 384 MB
  - Anzahl virtueller CPUs: 1

Um die Fragmentierung und die Planung effizient durchführen zu können, müssen Informationen der Infrastruktur zur Verfügung stehen. Wie in Kapitel 3.1 erläutert, wird der freie und gesamte Speicherplatz der verfügbaren Server benötigt. Die Informationen können bei Eucalyptus abgefragt werden. Der Befehl „euca_describe_availability_zones“ gibt eine Liste aus, die alle nötigen Informationen enthält. Diese Liste ist in Abbildung 12 zu sehen. Sie enthält alle virtuellen Maschinen. Diese sind nach Typen klassifiziert, die etwas über die Größe der virtuellen Maschine aussagen. Die Spalte „free“ gibt an, wie viele virtuelle Maschinen (VMs) von einem Typ momentan verfügbar sind. Diese Typen sind von Eucalyptus zur Klassifikation der Knoten eingeführt worden. Die Spalte „max“ hingegen gibt Auskunft darüber, wie viele virtuelle Maschinen insgesamt zur Verfügung

stehen. Hierzu zählen auch die, die momentan beschäftigt sind. Darüber hinaus enthält die Liste noch Informationen zu den einzelnen VM Typen, zu denen die Anzahl der CPUs, der RAM und die Speichergröße gehören. Diese Informationen werden in unterschied-

```
AVAILABILITYZONE        cluster 192.168.1.65
AVAILABILITYZONE        |- vm types     free / max   cpu   ram  disk
AVAILABILITYZONE        |- m1.small     0000 / 0000   1    128     2
AVAILABILITYZONE        |- c1.medium    0000 / 0000   1    256     5
AVAILABILITYZONE        |- m1.large     0000 / 0000   2    512    10
AVAILABILITYZONE        |- m1.xlarge    0000 / 0000   2   1024    20
AVAILABILITYZONE        |- c1.xlarge    0000 / 0000   4   2048    20
```

Abbildung 12: Ausgabe des Befehls „euca_describe_availability_zones“

lichen Algorithmen des dynamischen Deployments benötigt. Das erste Mal, wenn auf diese Angaben zugegriffen werden muss, ist vor Beginn der Fragmentierung. Das zweite Mal geschieht der Zugriff auf diese Daten während des dynamischen Deployments und weitere Zugriffe folgen, wenn neu fragmentiert oder neu geplant werden muss.

Eucalyptus selbst besitzt die Möglichkeit, einen Scheduling Algorithmus zu setzen, der die zuvor erstellten Images (enthält ein Betriebssystem und die gewünschte Konfiguration samt aller nötigen Programme und Dateien) auf die zur Verfügung stehenden Knoten der Cloud verteilt. Dieser Algorithmus kann in eucalyptus.conf (zu finden in eucalyptus-1.6.2/etc/eucalyptus/) gesetzt werden, indem eine Variable auf den Namen der Scheduling Strategie gesetzt wird. Diese Variable wird in der Funktion „schedule_instance“ in eucalyptus-1.6.2/cluster/handlers.c aufgerufen, die den dazugehörenden Algorithmus aufruft. In der Funktion, die diese Funktion aufruft, muss alle Funktionalität dieser Arbeit - bis auf die eigentliche Scheduling Strategie - zu finden sein. Diese Funktion ist „doRunInstances“, die in der gleichen Datei zu finden ist. Eine Erweiterung, die nicht vergessen werden darf, ist die Hinzunahme des neuen Werts der Enumeration, die die möglichen Scheduling Strategien enthält. Diese Enumeration ist in eucalyptus-1.6.2/cluster/handlers.h zu finden. In dieser Datei muss auch die Definition der neu hinzugekommenen Scheduling Strategie aufgenommen werden, da diese in einer eigenen Funktion implementiert sein muss.

4.3 Fragmentierung

Die Fragmentierung soll aus einer BPEL-Datei mehrere BPEL-Dateien erstellen. Um die Fragmentierung auszulösen, kann über das Kontextmenü der bpel-Datei in Eclipse (siehe Kapitel 4.1) der Punkt „Fragmentierung“ ausgewählt werden (siehe Abbildung 13).

Dazu nutzt das Plugin den Extension Point „org.eclipse.ui.popupMenus“. Um den Workflow zu fragmentieren, wird in diesem Plugin die Eclipse-interne Repräsentation Eclipse Modeling Framework (EMF) genutzt, die speziell für BPEL ausgelegt ist. Diese Repräsentation ist ein Baum, der traviersiert werden kann. Die Ausgabe des Fragmentierungsalgorithmus ist die frag- und die plan-Datei frag. Die plan-Datei ist der Einstiegspunkt für die Planung.

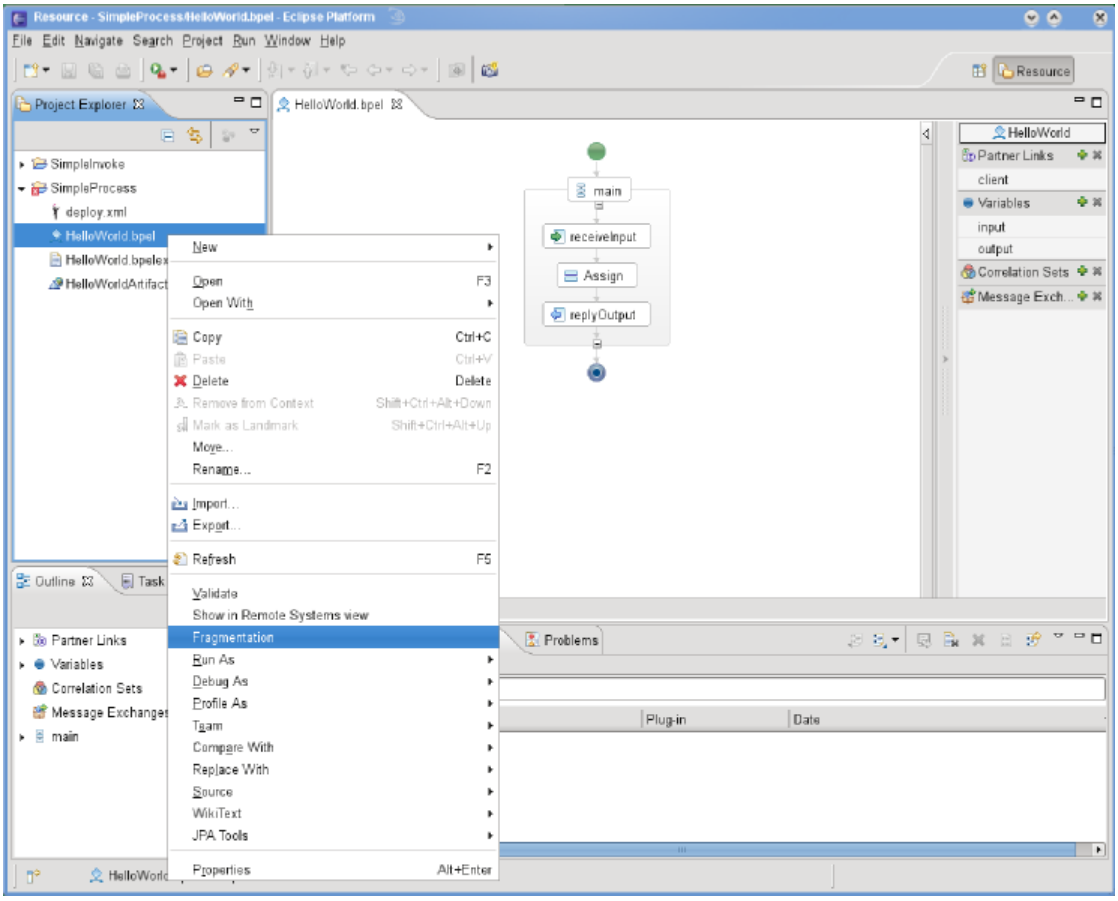

Abbildung 13: Start der Fragmentierung in Eclipse

Die Klassen der Fragmentierung und ihre Methoden sind im Folgenden aufgelistet:
Klasse Start: implementiert das Interface „IActionDelegate“

- public void run(IAction action) ruft startFragmentation der Klasse Fragmentation auf
- public void selectionChanged(IAction action, ISelection selection)

Klasse Fragmentation: implementiert die Fragmentierung

- protected static void startFragmentation (org.eclipse.bpel.model.Process pro, String loc, String locFrag)
  Stößt die Fragmentierung an und erstellt die frag- und plan-Datei. Die Methoden clearVariables(), die Hauptmethode fragment(process) und writeSuccessors() werden dazu aufgerufen.
- private static void clearVariables()
  Setzt alle Variablen zurück und wird von startFragmentation (org.eclipse.bpel.-model.Process pro, String loc, String locFrag) aufgerufen.
- private static void fragment(ExtensibleElement element)
  Hauptmethode der Fragmentierung, die von startFragmentation (org.eclipse.bpel.-model.Process pro, String loc, String locFrag) aufgerufen wird. Führt die Fragmentierung durch und ruft direkt oder indirekt alle weiteren Methoden auf.
- private static String fillFragFileHeader (org.eclipse.bpel.model.Process process)
  Befüllt die frag-Datei mit dem Definitionsteil des Prozesses und wird von fragment(ExtensibleElement element) aufgerufen.
- private static String makeInvoke (int counter)
  Generiert das *invoke*, das ein anderes Fragment aufruft. Diese Methode wird von fragment(ExtensibleElement element) aufgerufen.
- private static void makeFragFile (String ff)
  Erschafft die bpel-Dateien für die einzelnen Fragmente. Wenn es noch keinen Ordner gibt, der diese Fragmente enthalten soll, wird dieser ebenfalls erstellt. Der Aufruf erfolgt durch closeFragFile().
- private static void openFragment ()
  Diese Methode schreibt alle nötigen Informationen in die frag-Datei, wenn ein neues Fragment beginnt. Dies beinhaltet den Aufruf der Methoden, die die Vorgänger und Nachfolger des aktuellen Fragments schreiben. Der Aufruf erfolgt durch fragment(ExtensibleElement element).
- private static void writePredecessors(Vector<String> fragPred)
  Schreibt die Vorgänger des aktuellen Fragments und wird von openFragment () aufgerufen.

- private static void writeSuccessors(Vector<String> pred, int in)
  Schreibt die Nachfolger von Fragmenten. Der Aufruf erfolgt durch writePredecessors (Vector<String> fragPred).
- private static void writeSuccessors()
  Schreibt die Nachfolger von Fragmenten, bei denen am Ende der Fragmentierung noch kein Nachfolger eingetragen ist. Diese Methode wird von startFragmentation (org.eclipse.bpel.model.Process pro, String loc, String locFrag) aufgerufen.
- private static void closeFragment()
  Schließt ein Fragment. Der Aufruf erfolgt durch fragment(ExtensibleElement element).
- private static void closeFragFile()
  Schreibt eine frag-Datei. Der Aufruf erfolgt durch fragment(ExtensibleElement element).
- protected static void handleRecursion(ExtensibleElement element)
  regelt die Rekursion bei der Traversierung des Prozesses. Der Aufruf erfolgt durch fragment(ExtensibleElement element).

Klasse Activities: behandelt alle Aktivitäten, die in die bpel-Datei eines Fragments geschrieben werden. Alle Methoden werden von der Methode Fragmentation.fragment() aufgerufen, abhängig davon welche Aktivität momentan in der Fragmentierung behandelt wird.

- protected static String handleAssign (ExtensibleElement element)
- protected static String handleReceive (ExtensibleElement element)
- protected static String handleInvoke (ExtensibleElement element)
- protected static String handleReply (ExtensibleElement element)
- protected static String handleWait (ExtensibleElement element)
- protected static String handleEmpty (ExtensibleElement element)
- protected static String handleThrow (ExtensibleElement element)
- protected static String handleRethrow (ExtensibleElement element)
- protected static String handleExit (ExtensibleElement element)
- protected static String handleWhile (ExtensibleElement element)
- protected static String handleRepeatUntil (ExtensibleElement element)
- protected static String handleIf (ExtensibleElement element)
- protected static String handlePick (ExtensibleElement element)
- protected static String handleForEach (ExtensibleElement element)

4.4 Planung

Für den Beginn der Planung kann man durch einen Rechtsklick auf die plan-Datei den Punkt „Planning“ auswählen (siehe Abbildung 14). Die Klassen der Planung und ihre

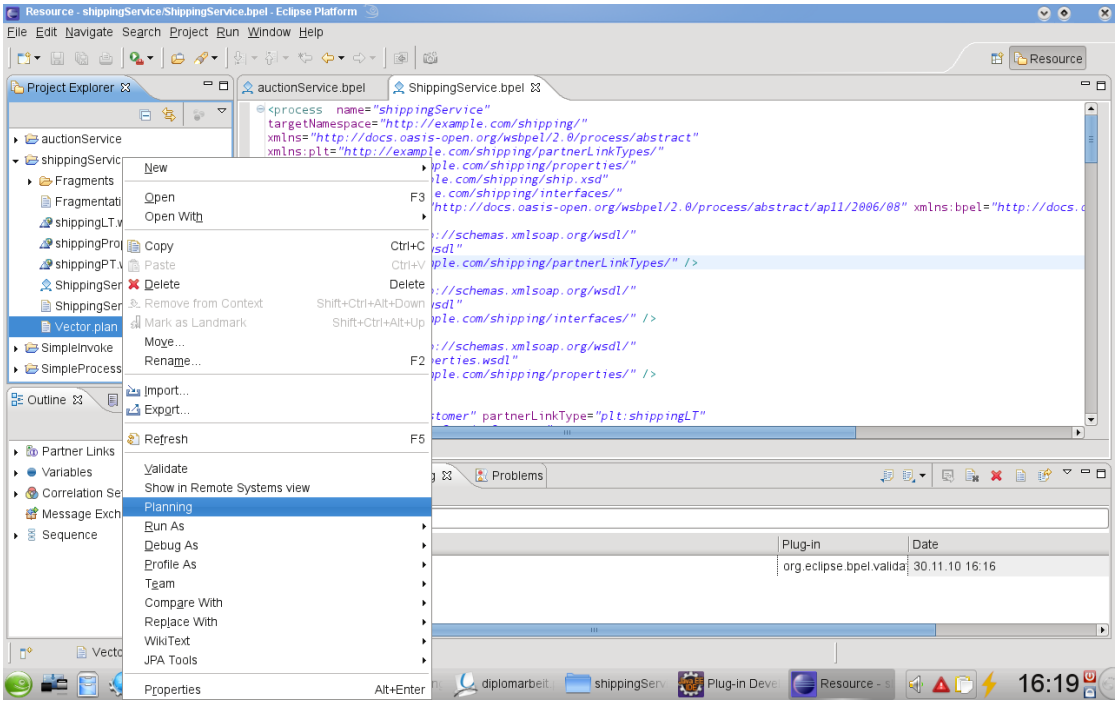

Abbildung 14: Start der Planung

Methoden sind im Folgenden aufgelistet:
Klasse Start: implementiert das Interface „IActionDelegate“

- public void run(IAction action) ruft startPlanning der Klasse Planning auf
- public void selectionChanged(IAction action, ISelection selection)

Klasse Planning:

- protected static void startPlanning(IPath p) throws IOException
  stößt die Planung an. Zu Beginn werden die Variablen zurückgesetzt, indem clearVariables() aufgerufen wird. Anschließend wird die frag-Datei durch readFragFile() ausgelesen, der BPEL-Prozess für die Datenanalyse bestimmt, die Datenanalyse ausgeführt (makeDataAnalysis(), readAnalysisResult()), die Größen der Fragmente (sizeOfFragmentsToArray()) und Server gespeichert (serverInformationToArray()), das Mapping aufgerufen (mapping(fragmentSizes, 0)) und schließlich das Deployment gestartet (deployment()).
- private static void clearVariables()
  Alle Variablen werden zurückgesetzt.

- private static void sizeOfFragmentsToArray()
  Die Größe der Fragmente wird ermittelt und gespeichert. Diese Methode wird von startPlanning(IPath p) aufgerufen und ruft ihrerseits mergeSort(Comparable <Integer> [ ][ ] a) auf.

- private static void serverInformationToArray() throws FileNotFoundException
  Verfügbare Informationen über die Server der Cloud werden ausgelesen und gespeichert. Diese Methode wird von startPlanning(IPath p) aufgerufen.

- public static void mergeSort(Comparable <Integer> [ ][ ] a)
  MergeSort Algorithmus. Er sortiert die Größen der Fragmente und ruft hierzu die Methoden mergeSort(Comparable <Integer> [ ][ ] a, Comparable <Integer> [ ][ ] tmpArray, int left, int right) und merge(Comparable <Integer> [ ][ ] a, Comparable <Integer> [ ][ ] tmpArray, int leftPos, int rightPos, int rightEnd) auf. Dieser Algorithmus wird von sizeOfFragmentsToArray() aufgerufen.

- private static void mergeSort (Comparable <Integer> [ ][ ] a, Comparable <Integer> [ ][ ] tmpArray, int left, int right)
  MergeSort Algorithmus

- private static void merge(Comparable <Integer> [ ][ ] a, Comparable <Integer> [ ][ ] tmpArray, int leftPos, int rightPos, int rightEnd)
  MergeSort Algorithmus

- private static void mapping(Comparable <Integer> [ ][ ] frags, int place)
  Hier werden die Fragmente auf Server gemappt. Diese Methode ruft searchFittingServer(Comparable <Integer> size), getBiggestServer(Comparable <Integer> size) und writeToArray (Comparable <Integer> serverNumber, Comparable <Integer> fragmentNumber, int i) auf und wird von startPlanning(IPath p) aufgerufen.

- private static void writeToArray (Comparable <Integer> serverNumber, Comparable <Integer> fragmentNumber, int i)
  Speichert das Ergebnis des Mappings und wird von mapping (Comparable <Integer> [ ][ ] frags, int place) aufgerufen.

- private static Comparable <Integer> searchFittingServer(Comparable <Integer> size)
  Der Aufruf erfolgt durch mapping(Comparable <Integer> [ ][ ] frags, int place). Ein verfügbarer Server passender Größe wird gesucht.

- private static Comparable <Integer> getBiggestServer(Comparable <Integer> size)
  Der Aufruf erfolgt durch mapping(Comparable <Integer> [ ][ ] frags, int place). Der größte verfügbare Server wird gesucht.

- private static void deployment()
  Diese Methode wird von startPlanning() aufgerufen und stößt das Deployment an.

Dazu werden die Methoden deploy (String fragment) und eventuell startRefragmentation(String fragment) oder rePlanning (int fragNumber) aufgerufen.

- private static void deploy (String fragment)
  Der Aufruf erfolgt durch die Methode Deplpoyment(). Hier wird das Deployment wird mit stetiger Planung durchgeführt. Es wird eventuell rePlanning (int fragNumber) oder startRefragmentation(String fragment) aufgerufen. Die Methode makeDeployXml (IPath bundle) wird zusätzlich genutzt.

- private static void makeDeployXml (IPath bundle)
  Die Methode wird durch deploy (String fragment) aufgerufen. Sie erzeugt die Datei deploy.xml für jedes Fragment.

- private static void readFragFile() throws IOException
  Die Methode wird von startPlanning() aufgerufen, um aus der frag-Datei alle relevante Informationen, wie Vorgänger und Nachfolger auszulesen und zu speichern.

- private static void writePredecessorsAndSuccessors (String fragment, boolean bool)
  Diese Methode wird von readFragFile() aufgerufen und erneuert die Vorgänger und Nachfolger, nachdem die plan-Datei neu eingelesen wurde.

- private static void rePlanning (int fragNumber)
  Das erneute dynamische Deployment wird durchgeführt, indem ein neuer verfügbarer Server für das aktuelle Fragment gesucht wird. Diese Methode wird von deploy (String fragment) oder deployment() aufgerufen.

Klasse Refragmenation: In dieser Klasse wird die erneute Fragmentierung behandelt.

- public static void startRefragmentation(String fragment)
  Die Refragmentierung wird durch deploy (String fragment) oder deployment() gestartet. Diese Methode ruft refragment () auf.

- private static boolean refragment () throws IOException
  Der Aufruf erfolgt durch startRefragmentation (String fragment). Die Methode refragmentiert das aktuelle Fragment und die plan-Datei wird neu geschrieben. Um den Definitionsteil der Fragmente zu erhalten, wird fillHeader (int k) aufgerufen.

- private static String fillHeader (int k) throws IOException
  Diese Methode wird von refragment () aufgerufen und liest den Definitionsteil für die neu erstellten Fragmente ein.

4.5 Datenflussanalyse

Dieser Algorithmus wird während des dynamischen Deployment verwendet, um zu überprüfen, ob alle Datenübertragungsmengen kleiner als der festgelegte Schwellwert (der definiert wird) ist und um die Datenweiterleitung zwischen den Fragmenten gewährleisten zu können. Der Algorithmus ist in Kapitel 3.2 genauer betrachtet worden, wird in Arbeit [10] beschrieben und in einer Diplomarbeit [11] implementiert. Der Algorithmus ist

seinerseits in einem Plugin enthalten, das für diese Arbeit in Eclipse integriert sein muss. Die Verwendung des Plugins wird durch einen Aufruf der Hauptmethode eingeleitet.

Das Plugin erweitert Eclipse am Extension Point „org.eclipse.core.runtime.applications“. Die Methode die aufgerufen wird, ist „de.uni_stuttgart.iaas.bpel_d.algorithm.analysis.Process.analyzeProcessModel(process)“. Anschießend wird „de.uni_stuttgart.iaas.bpel_d.algorithm.analysis.output()“ aufgerufen. Diese Methode wurde erweitert, so dass sie die Ergebnisse der Analyse in eine Datei scheibt, die zur Benutzung wieder eingelesen werden kann.

5 Zusammenfassung und Ausblick

In diesem Kapitel werden die Ansätze und die Ergebnisse der Arbeit in Abschnitt 5.1 skizziert, bevor in Kapitel 5.2 erörtert wird, welche Möglichkeiten es gibt, diese Arbeit weiterzuführen.

5.1 Zusammenfassung der Ergebnisse

Der vorgestellte Algorithmus legt einen Grundstein, um die Gebiete des Workflow Managements und der künstlichen Intelligenz in einem Algorithmus zu nutzen und diese Nutzung weiter auszubauen. Der Vorteil ist, dass die Ausführung eines scientific Workflows in Umgebungen, die ihre Infrastruktur ändern, durch die Verwendung einer Fragmentierung und von Planungsalgorithmen effektiver und effizienter gestaltet werden kann. Die Ursache liegt darin, dass die Fragmentgrößen an die verfügbaren Ressourcen dynamisch angepasst werden können und auch die Zuordnung zwischen den Fragmenten und Servern dynamisch gestaltet ist.

Der Algorithmus führt eine Fragmentierung und verteilte, dynamische Ausführung von BPEL-Prozessen wie folgt durch. Um einen BPEL-Prozess zu fragmentieren, muss dieser ausführbar, also ohne syntaktische Fehler, sein. Darüber hinaus muss das Bernstein-Kriterium gelten.

Die Fragmentierung erfolgt aufgrund der im Prozess enthaltenen Aktivitäten und deren Reihenfolge. Alle Fragmente enthalten - außer einer umschließenden Sequence - nur Basis-Aktivitäten, Schleifenkonstrukte, Alternativen oder *Picks*. Die Konstrukte *Scope*, *Sequence* und *Flow* des Ursprungsprozesses dienen nur der korrekten Erstellung der Fragmente und alle Arten von Handlern werden nicht beachtet. Nur bei Fragmenten mit enthaltenen Schleifenkonstrukten, Alternativen oder *Picks* kommen verschachtelte Fragmente vor. Alle diese Konstrukte sind in einem Fragment enthalten, dass keine weiteren Aktivitäten des Ursprungsprozesses enthält. Alle Aktivitäten, die von diesen Konstrukten ausgeführt werden sollen, sind in Fragmenten enthalten, die von dem Fragment mit dem Konstrukt aus aufgerufen werden. Diese *invokes*, mit denen die enthaltenen Fragmente aufgerufen werden, werden während der Fragmentierung hinzugefügt. Die Ausgabe der Fragmentierung sind alle Fragmente als BPEL-Dateien und zwei Dateien, die den Ablauf der Fragmentierung beschreiben. Die frag-Datei wird zu Export-Zwecken erstellt, wohingegen die plan-Datei für den internen Gebrauch bestimmt ist und Grundlage für das dynamische, verteilte Deployment ist. Der Inhalt der beiden Dateien ist identisch, nur die Erstellung unterschiedet sich. Sie enthalten eine Auflistung der Fragmente, deren Speicherort und deren enthaltenen Aktivitäten sowie alle Vorgänger und Nachfolger der einzelnen Fragmente.

Das verteilte, dynamische Deployment setzt auf der Fragmentierung auf, indem die plan-Datei, die Größe der Fragmente und Informationen über die Infrastruktur der Cloud eingelesen werden. Aufgrund dieser Daten wird zu Beginn ein Mapping durchgeführt, das die Fragmente den verfügbaren Servern zur Ausführung zuordnet. Um das Deployment durchzuführen, muss im nächsten Schritt eine Deployment-Einheit erstellt werden, die alle nötigen Dateien enthält. Zu dieser Einheit gehört das Fragment selbst mit seiner bzw.

seinen WSDL-Datei(en) und einer deploy.xml, die zur Ausführung benötigt wird, ein Betriebssystem und eine Engine, um das Fragment ausführen zu können. Ist diese Einheit erstellt, kann sie auf den Server kopiert werden, auf dem sie ausgeführt werden soll. Ist der Server in der Zwischenzeit nicht mehr verfügbar, weil er beispielsweise abgestürzt ist, gibt es zwei Möglichkeiten für die weitere Ausführung des Fragments. Wenn es einen Server gibt, dessen Größe ausreichend für die Ausführung des Fragments ist, wird nur das Mapping und die nachfolgenden Schritte erneut ausgeführt. Gibt es aber keinen Server, dessen gesamte Größe groß genug ist, muss das Fragment weiter zerlegt werden, falls dies möglich ist. In diesem Fall wird eine Refragmentierung eingeleitet. Sind aus diesem Fragment kleinere Teilfragmente entstanden, können diese alle nötigen Schritte, bis hin zur Ausführung erneut durchlaufen. Ist eine weitere Aufteilung des Fragments nicht möglich, muss gewartet werden, bis ein Server zur Verfügung steht, der für die Ausführung dieses Fragments in Frage kommt.

Die Umsetzung dieser Idee erfolgt in einem Eclipse-Plugin, das das Kontext-Menü des Navigators erweitert. Durch einen Rechtsklick auf die BPEL-Datei, die fragmentiert werden soll, kann diese angestoßen werden. Durch einen Rechtsklick auf die plan-Datei kann nach der Fragmentierung auch die Planung gestartet werden.

Der Algorithmus enthält die grundlegende Funktionalität. Durch die Unterstützung von allen Basis-Aktivitäten, *Flows*, *Sequenzen*, *Scopes*, Alternativen (*If*), *Picks* und Schleifen (*While*, *Repeat Until*, *For Each*) ist eine Fragmentierung und ein dynamisches Deployment eines einfachen scientific Workflows möglich. Dieser Ansatz kann weiter ausgebaut werden, so dass auch alle Arten von Handlern unterstützt werden. Einschränkungen, die sich bei diesem Ansatz - beispielsweise durch die Erfüllung des Bernsteinkriteriums - ergeben, können verringert werden, wodurch der hier entwickelte Algorithmus auf einem größeren Gebiet angewendet werden kann.

5.2 Weiterführende Arbeiten und Ausblick

Der Algorithmus kann durch eine Vielzahl von Ansätzen erweitert oder weiter verfeinert werden. Wie durch die Nennung folgender Beispiele ersichtlich wird, hat der hier vorgestellte Ansatz Erweiterungsmöglichkeiten in vielen Bereichen. Daher ist dieser Ansatz eine Grundlage, um weiter Forschung zu betreiben.

Ansätze zur Erweiterung könnten die Folgenden sein. Es wäre denkbar, eine manuelle Fragmentierung und Teile des dynamischen Deployments manuell auszuführen. Dabei ist der Benutzer nicht ganz auf sich allein gestellt, sondern erhält Unterstützung zur manuellen Bearbeitung. Im Falle der manuellen Fragmentierung würde eine Unterstützung Hilfe bei der Erstellung der plan- und frag-Datei bedeuten. Dies ist möglich, wenn der Benutzer die Stelle markiert, an der fragmentiert werden soll und die so erstellten Fragmente automatisch die richtige Nummerierung erhalten und dabei in die plan- und frag-Datei aufgenommen werden. Bei einem manuellen dynamischen Deployment ist es beispielsweise möglich, den Benutzer auswählen zu lassen, welche Ressourcen er nutzen will. Diese könnten eine Untermenge aller zur Verfügung stehender Ressourcen sein. Nach dieser Auswahl wird das ausgewählte Bundle automatisch auf der Ressource ausgeführt. In diesem Prozess muss der Benutzer zusätzlich durch die Vorgabe der Ausführungsreihen-

folge der Bundles unterstützt werden.

Auf der anderen Seite ist es sinnvoll, eine Ausführungsdauer im Voraus zu berechnen, um dem Benutzer eine Abschätzung der Ausführungsdauer zur Verfügung zu stellen. Dadurch kann der Benutzer darauf reagieren, falls ihm die Dauer zu lange ist. Die Ausführungsdauer kann anhand des kritischen Zeitpfad berechnet werden, indem man beispielsweise jeder Basis-Aktivität eine Dauer zuweist. Nachdem alle möglichen Pfade durch den Workflow berechnet worden sind, ist der kritische Zeitpfad der Pfad, der die längste Ausführungsdauer hat.

Ein weiteres Beispiel zur Erweiterung des Algorithmus stellt die Möglichkeit dar, den Workflow vor Beginn der Fragmentierung auf Fehler zu untersuchen (Ansatz siehe Referenz [22]). Bei diesem Ansatz wird vorausgesetzt, dass die Fragmentierung auf Basis von ausführbaren Dateien durchgeführt wird, der Prozess also eine korrekte Syntax besitzt. Der Prozess kann trotzdem auf Syntaxfehler untersucht werden. Darüber hinaus kann die semantische Korrektheit des Prozesses überprüft werden. Beispielsweise durch statische Analysen kann der Prozess auf das Vorhandensein von Endlosschleifen überprüft werden. Enthält der Workflow syntaktische oder semantische Fehler, ist die Ausführung zwecklos, sie wird nicht gestartet und der Benutzer wird darüber informiert. Darüber hinaus könnte der Workflow mit Hilfe von Kontroll- und Datenfluss und DPE optimiert werden, indem Sequenzen, die nicht voneinander abhängen, parallel ausgeführt werden können. Diese Erweiterung könnte durch einen Schalter an- und ausschaltbar sein und durch eine partiell ordnende Planung realisiert sein.

Ein Beispiel, das nur die Erweiterung der Fragmentierung betrifft, ist die Ermittlung von fixen und portablen Knoten eines Fragments, wie sie in dem Ansatz von [16] vorgestellt wird. Durch diesen Ansatz können einzelne Fragmente weiter zerteilt werden. Diese Methode könnte statt der Neufragmentierung angewendet werden. In diesem Ansatz wird diese Methode nicht verwendet, weil die Fragmentierung aufgrund der im Prozess enthaltenen Aktivitäten und der Struktur des Prozesses durchgeführt wird. Die Zuordnung von portablen Knoten zu einem fixen Knoten hingegen beachtet nicht die Ausführungsreihenfolge, ermöglicht aber mehrere Möglichkeiten eine bestimmte portable Aktivität zu einem fixen Knoten zuzuordnen.

A Anhang

A.1 Der Prozess „auctionService“

Als erstes Beispiel ist der Prozess „auctionService“ gewählt worden, der sich in der BPEL Spezifikation[5] befindet. Der Quellcode der bpel- und wsdl-Datei ist hier nicht erneut aufgeführt, er kann in der Spezifikation eingesehen werden.

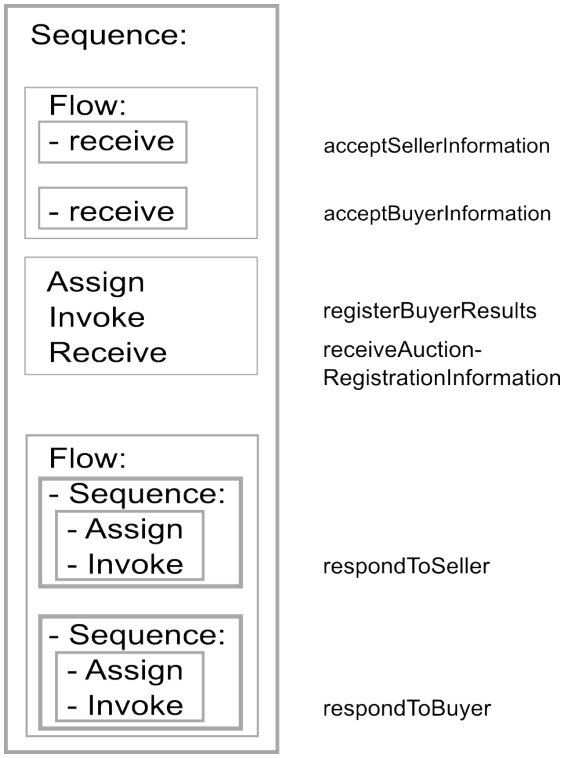

Abbildung 15: Struktur von auctionService.bpel: In der linken Spalte sind die Aktivitäten mit den erstellten Fragmenten zu sehen. Die Fragmente, die als oberstes Element eine Sequenz oder einen Flow enthalten, sind nur der Übersichtlichkeit halber eingefügt. Diese Rechtecke stellen keines der erstellten Fragmente dar. Die rechte Spalte enthält die Namen der entsprechenden Aktivitäten. Ist kein Name neben einer Aktivität abgebilet, hat diese keinen Namen.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<Fragments>
  <Fragment1>
    <bpelFile>
      <"/home/eclipse/Workspace/auctionService
```

[5] http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html

```
        /Fragments/Fragment1.bpel"/>
    </bpelFile>
    <Activities>
      <receive:acceptSellerInformation/>
    </Activities>
    <Predecessors>
      <null/>
    </Predecessors>
    <Successors>
      <Fragment3/>
    </Successors>
  </Fragment1>

  <Fragment2>
    <bpelFile>
      <"/home/eclipse/Workspace/auctionService
        /Fragments/Fragment2.bpel"/>
    </bpelFile>
    <Activities>
      <receive:acceptBuyerInformation/>
    </Activities>
    <Predecessors>
      <null/>
    </Predecessors>
    <Successors>
      <Fragment3/>
    </Successors>
  </Fragment2>

  <Fragment3>
    <bpelFile>
      <"/home/eclipse/Workspace/auctionService
        /Fragments/Fragment3.bpel"/>
    </bpelFile>
    <Activities>
      <assign:null/>
      <invoke:registerAuctionResults/>
      <receive:receiveAuctionRegistrationInformation/>
    </Activities>
    <Predecessors>
      <Fragment1/>
      <Fragment2/>
    </Predecessors>
    <Successors>
```

```
      <Fragment4/>
      <Fragment5/>
    </Successors>
  </Fragment3>

  <Fragment4>
    <bpelFile>
      <"/home/eclipse/Workspace/auctionService
        /Fragments/Fragment4.bpel"/>
    </bpelFile>
    <Activities>
      <assign:null/>
      <invoke:respondToSeller/>
    </Activities>
    <Predecessors>
      <Fragment3/>
    </Predecessors>
    <Successors>
      <null/>
    </Successors>
  </Fragment4>

  <Fragment5>
    <bpelFile>
      <"/home/eclipse/Workspace/auctionService
        /Fragments/Fragment5.bpel"/>
    </bpelFile>
    <Activities>
      <assign:null/>
      <invoke:respondToBuyer/>
    </Activities>
    <Predecessors>
      <Fragment3/>
    </Predecessors>
    <Successors>
      <null/>
    </Successors>
  </Fragment5>
</Fragments>
```

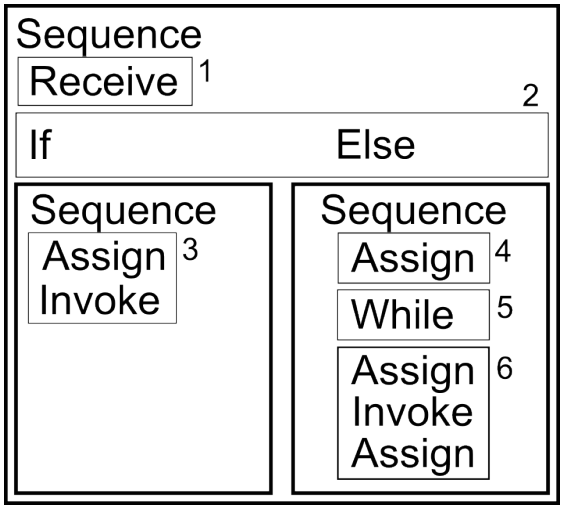

Abbildung 16: Struktur des Prozesses shippingService

A.2 Der Prozess „shippingService“

In Abbildung 16 ist erneut die Struktur des Prozesses dargestellt. Im Folgenden ist die frag-Datei zu sehen.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Fragments>
  <Fragment1>
    <bpelFile>
      <"/home/eclipse/Workspace/shippingService/
        Fragments/Fragment1.bpel">
    </bpelFile>
    <predecessorsFragment1>
      <null>
    </predecessorsFragment1>
    <successorsFragment1>
      <Fragment2>
    </successorsFragment1>
    <Activities>
      <receive:receive1/>
    </Activities>
  </Fragment1>

  <Fragment2>
    <bpelFile>
      <"/home/eclipse/Workspace/shippingService/
        Fragments/Fragment2.bpel">
```

```
  </bpelFile>
  <predecessorsFragment2>
    <Fragment1>
  </predecessorsFragment2>
  <successorsFragment2>
    <null>
  </successorsFragment2>
  <Activities>
    <If:null>
  </Activities>
</Fragment2>

<Fragment3>
  <bpelFile>
    <"/home/eclipse/Workspace/shippingService/
      Fragments/Fragment3.bpel">
  </bpelFile>
  <predecessorsFragment3>
    <if>
  </predecessorsFragment3>
  <successorsFragment3>
    <null>
  </successorsFragment3>
  <Activities>
    <assign:assign1/>
    <invoke:invoke1/>
  </Activities>
</Fragment3>

<Fragment4>
  <bpelFile>
    <"/home/eclipse/Workspace/shippingService/
      Fragments/Fragment4.bpel">
  </bpelFile>
  <predecessorsFragment4>
    <if>
  </predecessorsFragment4>
  <successorsFragment4>
    <null>
  </successorsFragment4>
  <Activities>
    <assign:assign2/>
  </Activities>
</Fragment4>
```

```
  <Fragment5>
    <bpelFile>
      <"/home/eclipse/Workspace/shippingService/
        Fragments/Fragment5.bpel">
    </bpelFile>
    <predecessorsFragment5>
      <if>
    </predecessorsFragment5>
    <successorsFragment5>
      <null>
    </successorsFragment5>
    <Activities>
      <while:null>
    </Activities>
  </Fragment5>

  <Fragment6>
    <bpelFile>
      <"/home/eclipse/Workspace/shippingService/
        Fragments/Fragment6.bpel">
    </bpelFile>
    <predecessorsFragment6>
      <while>
    </predecessorsFragment6>
    <successorsFragment6>
      <null>
    </successorsFragment6>
    <Activities>
      <assign:assign3/>
      <invoke:invoke2/>
      <assign:assign4/>
    </Activities>
  </Fragment6>
</Fragments>
```

Literatur

[1] Brian Hayes. Cloud computing. *Commun. ACM*, 51(7):9–11, 2008.

[2] F. Leymann and D. Roller. *Production Workflow, Concepts and Techniques.* Prentice Hall, Upper Saddle River, New Jersey 07458, 2000.

[3] A. Lenk et. al. What's inside the cloud? an architectural map of the cloud landscape. In *ICSE Workshop on Software Engineering Challenges of Cloud ComputingVancouver*, pages 23–31, Canada, 2009.

[4] C. Ellis and G. Rozenberg. Dynamic change within workflow systems. In *COCS '95: Proceedings of conference on Organizational computing systems*, pages 10–21, New York, NY, USA, 1995. ACM.

[5] R. Petzschmann. Entwicklung eines planungsalgorithmus für mediatorbasierte informationssysteme unter berücksichtigung eingeschränkter anfragemöglichkeiten. Master's thesis, Technische Universität Berlin, 2005.

[6] C. Reese, J. Ortmann, S. Offermann, D. Moldt, K. Lehmann, and T. Carl. Architecture for distributed agent-based workflows. pages 42–49, 2005.

[7] Gurmeet Singh Mei-Hui Su Ewa Deelman, Gaurang Mehta and Karan Vahi. *Pegasus: Mapping Large-Scale Workflows to Distributed Resources.* Springer, 2006.

[8] Hilmar Schuschel and Mathias Weske. Plaengine: Ein system zur planung und ausführung von workflows. In *BTW*, pages 225–234, 2005.

[9] Stuard Russell and Peter Norvig. *Künstliche Intelligenz: Ein moderner Ansatz.* Pearson Studium, München, 2007.

[10] Oliver Kopp, Rania Khalaf, and Frank Leymann. Deriving explicit data links in ws-bpel processes. In *Proceedings of the International Conference on Services Computing, Industry Track, SCC 2008*, pages 367–376. IEEE Computer Society, 2008.

[11] Sebastian Breier. Extended Data-flow Analysis on BPEL Processes. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Juli 2008.

[12] R. Khalaf, O. Kopp, and F. Leymann. Maintaining data dependencies across bpel process fragments. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing - ICSOC 2007*, volume 4749 of *LNCS*, pages 207–219. Springer, 2007.

[13] Simon Moser, Axel Martens, Katharina Görlach, Wolfram Amme, and Artur Godlinski. Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis. pages 98–105. IEEE Computer Society, 2007.

[14] Rania Khalaf and Frank Leymann. Role-based decomposition of business processes using bpel. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 770–780, Washington, DC, USA, 2006. IEEE Computer Society.

[15] W.Tan and Y. Fan. Dynamic workflow model fragmentation for distributed execution. *Comput. Ind.*, 58(5):381–391, 2007.

[16] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *In OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–187. ACM Press, 2004.

[17] D. Wutke, D. Martin, and F. Leymann. Model and infrastructure for decentralized workflow enactment. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 90–94, New York, NY, USA, 2008. ACM.

[18] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[19] D. Martin, D. Wutke, and F. Leymann. A novel approach to decentralized workflow enactment. In *EDOC '08: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 127–136, Washington, DC, USA, 2008. IEEE Computer Society.

[20] O. Danylevych, D. Karastoyanova, and F. Leymann. Optimal stratification of transactions. *Internet and Web Applications and Services, International Conference on*, 0:493–498, 2009.

[21] Daniel Nurmi, Richard Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *CCGRID*, pages 124–131, 2009.

[22] Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. Faster and more focused control-flow analysis for business process models through sese decomposition. In *ICSOC*, pages 43–55, 2007.

[23] Wolfram Amme, Axel Martens, and Simon Moser. Advanced verification of distributed ws-bpel business processes incorporating cssa-based data flow analysis. *International Journal of Business Process Integration and Management*, 4(1):47–59, 2009.

[24] Ganna Monakova, Oliver Kopp, Frank Leymann, Simon Moser, and Klaus Schaefers. Verifying business rules using an smt solver for bpel processes. *International Journal of Cooperative Information Systems (IJCIS)*, 17(3):259–282, 2008.

[25] Scott Callaghan, Ewa Deelman, Dan Gunter, Gideon Juve, Philip Maechling, Christopher Brooks, Karan Vahi, Kevin Milner, Robert Graves, Edward Field, David Okaya, and Thomas Jordan. Scaling up workflow-based applications. *J. Comput. Syst. Sci.*, 76(6):428–446, 2010.

[26] Ewa Deelman. Grids and clouds: Making workflow applications work in heterogeneous distributed environments. *International Journal of High Performance Computing Applications*, pages 1–15, 2009.

[27] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P. Berman, and Philip Maechling. Scientific workflow applications on amazon ec2. *CoRR*, abs/1005.2718, 2010.

[28] Paul T. Groth, Ewa Deelman, Gideon Juve, Gaurang Mehta, and G. Bruce Berriman. Pipeline-centric provenance model. *CoRR*, abs/1005.4457, 2010.

[29] Rizos Sakellariou, Henan Zhao, and Ewa Deelman. Mapping workflows on grid resources: Experiments with the montage workflow. *CoreGrid*, pages 1–14, 2009.

[30] Christina Hoffa, Gaurang Mehta, Timothy Freeman, Ewa Deelman, Kate Keahey, Bruce Berriman, and John Good. On the use of cloud computing for scientific workflows. *3rd International Workshop on Scientific Workflows and Business Workflow Standards in e-Science (SWBES) in conjunction with Fourth IEEE International Conference on e-Science (e-Science 2008)*, 2008.

[31] Gurmeet Singh, Carl Kesselman, and Ewa Deelman. Optimizing grid-based workflow execution. *J. Grid Comput.*, 3(3-4):201–219, 2005.

[32] Jim Blythe, Ewa Deelman, and Yolanda Gil. Planning for workflow construction and maintenance on the grid. *ICAPS 2003 Workshop on Planning for Web Services*, 2003.

[33] Ewa Deelman, James Blythe, Yolanda Gil, and Carl Kesselman. Pegasus: Planning for execution in grids. *GriPhyN technical report 2002-20*, 2002.

[34] Yolanda Gil, Ewa Deelman, Jim Blythe, Carl Kesselman, and Hongsuda Tangmunarunkit. Artificial intelligence and grids: Workflow planning and beyond. *IEEE Intelligent Systems*, 19(1):26–33, 2004.

[35] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Scott Koranda, Albert Lazzarini, Gaurang Mehta, Maria Alessandra Papa, and Karan Vahi. Pegasus and the pulsar search: From metadata to execution on the grid. In *PPAM*, pages 821–830, 2003.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Diana Przybylski)