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

Diplomarbeit Nr. 2475

Optimization of XML Processing within a BPEL Engine

Dominique Xavier Kiefner

Studiengang: Softwaretechnik

Prüfer: Professor Leymann

Betreuer: Dipl-Phys. Dieter Roller

begonnen am: 09.03.2006

beendet am: 08.09.2006

CR-Klassifikation: D.3.4, E.1, H.3.3, H.4.1

Zusammenfassung

Wenn man eine Verarbeitung von XML optimieren will, hat man zwei Ansatzpunkte: eine lokale Optimierung und eine globale Optimierung. Die lokale Optimierung wählt man, um ein besseres Verfahren zum Verarbeiten von XML zu erhalten. Eine globale Optimierung verwendet man, um den Gesamtablauf eines Geschäftsprozesses innerhalb der BPEL engine zu verbessern. In dieser Diplomarbeit wurden beide Ansätze untersucht. Es wurde auch untersucht, wie man beide Optimierungen sinnvoll verbinden kann, damit sich die lokale Optimierung und die globale Optimierung ergänzen und nicht gegenseitig beeinträchtigen.
Bei der Entwicklung der lokalen Optimierung wurde der Fokus sowohl auf das Auswerten von XPath Anweisungen in Transaktionsbedingungen als auch auf das Kopieren von Bestandteilen innerhalb einer Assignment Aktivität gelegt. Die Ergebnisse können auch auf alle anderen Manipulationen von XML übertragen werden. Als Ansatzpunkt zur lokalen Optimierung wurden verschiedene Parserstrategien von XML untersucht und anschließend ein Streamparser als Ansatz gewählt.
Um die globale Optimierung eines Prozesses durchzuführen, braucht man Informationen über den Prozessablauf. Diese Informationen können am besten durch eine Datenflussanalyse gewonnen werden, da es die Implementierung von SWoM erlaubt, Erkenntisse früh zu gewinnen und früh einzubauen. Die statische Typbindung und der statische Datenzugriff von BPEL4WS unterstützen eine Datenflussanalyse weiterhin. Der letzte Punkt der für eine Datenflussanaylse spricht ist, dass man BPEL4WS auf eine konkrete Implementierungssprache übersetzt. Es gibt viele bewährte Datenflussanalyseverfahren aus dem Übersetzerbau. In der Diplomarbeit wurde der Fokus auf das Analyseverfahren der Verwendungsketten gelegt. Das Verfahren stellt fest, wann Werte von Variablen gesetzt und wann später auf diese zugegriffen wird. So kann man die Zugriffe besser planen und optimieren.
Der Streamparser Ansatz ist im Besonderen dazu in der Lage, die Erkenntnisse über den Zeitpunkt der Zugriffe umzusetzen, um dadurch eine wechselseitige Verstärkung zu erhalten. Es wird im Folgenden untersucht werden ob die Kombination beider Verfahren zu einem deutlich besseren Laufzeitverhalten von SWoM führt.

Inhaltsverzeichnis

1. Einleitung **3**

2. Überblick **4**
2.1. Geschäftsprozesse . . . 4
2.2. Programmieren im Großen . . . 4
2.3. XML . . . 5
2.3.1. XPath . . . 6
2.3.2. Parser von XML . . . 7
2.4. WSDL . . . 9
2.5. BPEL4WS . . . 9
2.5.1. XPath . . . 10
2.6. Datenflussanalyse . . . 10

3. Analyse von SWoM **12**
3.1. SWoM . . . 12
3.1.1. Zentrale Idee von SWoM . . . 12
3.1.2. Der grundlegende Entwurf von SWoM . . . 13
3.1.3. Realisierung von BPEL4WS in SWoM . . . 14
3.2. Eine genaue Analyse von SWoM . . . 22
3.2.1. Beispiel . . . 22
3.2.2. Einlesen des Beispiels . . . 24
3.2.3. Überprüfung des eingelesenen BPEL4WS-Prozesses . . . 25
3.2.4. Die Laufzeitverhalten des Beispiels . . . 26
3.3. Bewertung beider Abläufe . . . 30

4. Lösung **31**
4.1. Spezifikation . . . 31
4.1.1. XPath Umsetzung . . . 32
4.1.2. Datenflussanalyse . . . 34
4.2. Entwurf . . . 34
4.2.1. Erste Implemetierungsphase . . . 34
4.3. Zweite Implemetierungsphase . . . 37

5. Test **41**
5.1. Test während der Implementierung . . . 41
5.2. Testumgebung der Funktionstests . . . 43
5.3. Ergebnisse des Funktionstests . . . 43

6. Ergebnis **44**
6.1. Analyse und Bewertung der Testergebnisse . . . 44
6.2. Ausblick . . . 45

A. Codetabelle **46**

1. Einleitung

Die Webdienste gehören zum dienstorientierten Architecture Framework, der das Fundament der modernen verteilten heterogenen Anwendungen bildet. Die Webdienste bilden eine perfekte Ebene im Modell des Zwei-Ebenen-Programmierens, das eine Eigenschaft von geschäftsprozessbasierten Anwendungen ist.
Die geschäftsprozessbasierten Anwendungen sind aus zwei Teilen zusammengesetzt. Der erste Teil ist das Programmieren im Großen, damit wird das Prozessmodell erstellt. Das Prozessmodell beschreibt den Ablauf und die Anordnung der Aktivitäten, die den Prozess ausführen. Der zweite Teil ist das Programmieren im Kleinen, damit werden die Einzelteile erstellt. Die Einzelteile implementieren die verschiedenen Aktivitäten. Im Umfeld der Geschäftsprozesse wird die Business Process Execution Language for Web Services (BPEL4WS) als eine Spezifikation zum Beschreiben des Prozessmodells eingesetzt. Die Implementierung der Aktivitäten wird als Webdienst definiert. Die konkrete Implementierung einer Aktivität kann durch jede Programmiersprache erfolgen. Besonders zu beachten ist, dass BPEL4WS einen Geschäftsprozess von außen als einen Webdienst darstellt, so dass ein rekursives Aggregationsmodell unterstützt wird.
Die Aufgabe eines Geschäftsprozess Management Systems ist das Management des Lebenszyklus eines Geschäftsprozesses, indem durch das dazugehörige Prozessmodel navigiert wird und dabei die dazugehörigen Webdienste aufgerufen werden. Wenn das System die BPEL4WS Spezifiaktion implementiert, spricht man von einer BPEL4WS engine. Eine zentrale Aufgabe einer BPEL4WS engine ist die Manipulation der Daten, die in XML vorliegen. Das Manipulieren umfasst das Parsen von eingehenden Nachrichten, das Erstellen von Nachrichten zum Aufrufen von Webdiensten, das Auswerten von XPath Anweisungen in Transition Bedingungen und das Kopieren von Bestandteilen einer Variable innerhalb einer Assignment Aktivität.
Das Ausführen dieser Manipulationen beansprucht einen erheblichen Teil der Leistung der BPEL engine. Es ist deswegen wünschenswert, dass nur die jeweiligen Manipulationen ausgeführt werden, die notwendig sind. Desweiteren sollen nur auf die Teile eines XML Dokuments zugegriffen werden, die gerade benötigt werden.
Das Ziel dieser Diplomarbeit ist die Entwicklung von Algorithmen und Speicherstrukturen für eine optimierte XML Verarbeitung um eine passende Implementierung für die Stuttgart Workflow Mashine SWoM zu erstellen. Die SWoM verarbeitet BPEL4WS und wurde vom Institut für Architektur von Anwendungssystemen der Universität Stuttgart im WS 05 bis SoSe 06 entwickelt. Desweiteren sollen Tests durchgeführt werden, um die Optimierung nachzuweisen.

2. Überblick

Bei der Entwicklung von Geschäftsprozessen werden alte Fragestellungen und Lösungen anderer Gebiete der Informatik erneut untersucht, da sich die Eigenschaften von Geschäftprozessen von den anderen Gebieten unterscheiden. Es könnte sein, dass man für Geschäftsprozesse neue Lösungen findet oder alt bewährte wieder endeckt. Die Eigenschaften von Geschäftsprozessen werden im folgenden Abschnitt beschrieben sowie auch die Eigenschaften der verwendeten Technologie von Geschäftsprozessen. Dabei werden Zusammenhänge aufgezeigt, die diese Diplomarbeit wesentlich beeinflussen.

2.1. Geschäftsprozesse

Der Begriff Geschäftsprozesse hat viele verwandte Begriffe, die das gleiche bezeichnen, so z.B. Arbeitsablauf, Arbeitsplanung oder auch den im Deutschen verwendeten Englischen Begriff Workflow. Allgemein kann man sagen, dass es für einen Geschäftsprozess ein Modell und eine Beschreibungssprache gibt. Das Modell stellt die Abbildung der Realität in das Softwaresystem dar. Bei der Abbildung gibt es implizite und explizite Eigenschaften. Die expliziten Eigenschaften spiegeln sich in der Beschreibungssprache des Modells wieder, die verwendete Beschreibungssprache ist BPEL4WS. Implizite Eigenschaften der Abbildung durch BPEL4WS sind:

- Das Programmieren im Großen.
- Der Prozess wird in einer verteilten und heterogenen Welt ausgeführt.
- Der Prozess steht im Mittelpunkt des Modells.
- In der Regel werden sehr viele Prozessinstanzen gleichzeitig ausgeführt.

2.2. Programmieren im Großen

Das Programmieren im Großen ist eine zusätzliche Abstraktionsebene, um das Erstellen von großen Softwaresystemen zu erleichtern. Zuvor gab es zwei Ebenen als Trennung zwischen Daten und Ausführung. Die untere Ebene ist für die Datenhaltung des Programms zuständig, während die obere Ebene der Programmdurchführung dient. Beide Ebenen interagieren über wohldefinierte Schnittstellen. Die Implementierung einer Ebene ist für die jeweils andere Ebene verborgen.
Beim Programmieren im Großen wird die obere Programmebene in zwei Ebenen aufgeteilt. Die eine Ebene beschreibt das Programm auf einer höheren Ebene, um so die Funktionsweise des Programms zu beschreiben und dadurch das Gesamtsystem besser zu verstehen, während die zweite Ebene für das Ausführen von Aktionen zuständig ist.
Dieses Konzept wurde im Buch von [LR99] auf Geschäftsprozesse übertragen und inhaltlich erweitert. Es geht dabei im Wesentlichen um das Programmieren im Großen als primäres Modellieren eines Prozesses. Das Modellieren beinhaltet sowohl die Definition des Ablaufs und die Datenweitergabe als auch die Schnittstelle zu anderen Prozessen oder zu anderen Diensten.

Die zweite Ebene, also die Implemetierung der Webdienste, nennt man auch das Programmieren im Kleinen. Diese Dienste sollen dann wohldefinierte Aufgaben erfüllen, wie beispielsweise die Interaktion mit dem Anwender oder die Manipulation von Daten. Die Dienste interagieren in der Regel mit den Geschäftsprozessen auf Basis von Nachrichten. Als industrieller Standard zum Modellieren von Geschäftsprozessen hat sich die Beschreibungssprache BPEL4WS etabliert.

2.3. XML

XML [BPSM98] ist eine Beschreibungssprache für Dokumente. Es verwendet Auszeichnungen zur Beschreibung von Inhalten und Strukturen eines Dokuments. Ein gültiges XML-Dokument besteht aus drei Teilen: einem Prolog, einem Rumpf und einem Epilog. Der Prolog und der Epilog sind jedoch optional. Alle drei Teile haben eine gemeinsame Wurzel. Die Wurzel und die drei Teile sind Elemente, wobei jedes Element einen Namen hat. Das Element kann eine leere Auszeichnung </**Element**> sein oder auch ein Paar von einem Anfangs- und einem Endauszeichner <**Element**></**Element**> sein. Damit ein Dokument wohlgeformt ist, müssen die Elemente baumartig strukturiert sein. Ausgehend von dem Wurzelelement müssen alle Elemente korrekt geschachtelt sein. Die Elemente des Baums sind dann korrekt geschachtelt, wenn das umschließende Element das innere Element vollständig enthält.
Somit gibt es eine eindeutige hierarchische Beziehung zwischen den Elementen. XML verwendet als Terminologie für die Hierarchie die Begriffe: Vater, Sohn und Geschwister. Ein Vater enthält seine Söhne als Elemente. Alle Elemente der selben Tiefe sind Geschwister, wenn sie den gleichen Vater haben. Neben dem Namen können Elemente weitere Attribute besitzen. Die Attribute werden innerhalb des Anfangsauszeichners eines Elements definiert, wobei ein Attribut aus seinem Namen und seinem Wert besteht [Bir01].
Eine weitere Besonderheit von XML sind die Namensräume. Ein Namensraum ist eine definierte Menge von Namen, die in einem XML-Dokument für die Namen von Elementen und Attributen verwendet werden können. Ein Namensraum hat einen Uniform Resource Identifier, einen Geltungsbereich und optional einen Präfix. Die URI bezieht sich auf das Dokument, das die Namen definiert. Der Geltungsbereich legt eindeutig fest, auf welchen Namensraum sich ein Name bezieht, denn Namensräume können bei jedem Element deklariert werden. Es gibt immer einen Namensraum, alle Namen ohne Präfix beziehen sich auf diesen Namensraum. Der Geltungsbereich ist der Teil des Dokuments, der sich zwischen den beiden Auszeichnern des Elements an dem der Namensraum deklariert wurde, befindet. Namen mit dem Präfix eines Namensraumes ermöglichen es mehrere Namensräume gleichzeitig zu benutzen, denn nur der Präfix mit dem Namen muss eindeutig sein.
XML hat auch ein Datenmodell, das unabhängig von einem Betriebssystem gilt. Die grundlegenen Datentypen wurden vom World Wide Web Consortium festgelegt. Um komplexe Datentypen zu beschreiben kann man Schemas benutzen. Das XML-Schema ist eine Möglichkeit, um einen Namensraum eines XML-Dokuments zu beschreiben. Das XML-Schema kann aber mehr, denn der Hauptzweck des Schemas ist es, die Metadaten eines XML-Dokuments zu beschreiben [TBMM01]. Das XML-Schema ist wiederum in

XML geschrieben und verwendet als Dateiendung *.xsd. Das Wurzelelement eines XML-Schemas hat eine Präambel. Das Wichtigste an einem Schema ist das Inhaltsmodell, in dem die Struktur der Elemente und der Attribute des XML-Dokuments definiert wird. Zudem hat das Schema noch ein Datenmodell, das eine Übermenge des Datenmodells von XML ist. Alle in der Diplomarbeit verwendeten XML-Formate beschreiben ihre Metadaten und Namensräume durch XML-Schemas. Es kann also durchaus sein, dass es Definitionsketten von Schemas gibt, d.h ein Schema wird wieder über ein Schema definiert sein, so dass zwei XML-Formate teilweise das gleiche Schema verwenden.

2.3.1. XPath

XPath ist eine Sprache zum Auffinden von Elementen innerhalb eines XML-Dokuments. Die so gefundenen Elemente können auch manipuliert werden. XPath ist durch eine Erweiterte Backus Naur Form [CD99] definiert. Um Sie in einem XML-Dokument zu verwenden kann man XPath als ein XML-Schema einbinden. XPath beschreibt nur, wie etwas gemacht werden soll, jedoch braucht man immer eine Implementierung zum Ausführen von XPath. Man formuliert die Suche und Manipulation von Elementen in XPath und die gewählte Implementierung führt sie dann aus. Um die Verarbeitung von XPath verbessern zu können, muss man wissen, wie XPath funktioniert.
Der Zugriff auf Elemente erfolgt über Ausdrücke, die als Expr formuliert werden. Expr kann ein Funktionsaufruf sein, der in der Kontextmenge definiert wird. Die Argumente des Funktionsaufrufs sind wieder Expr. Es gibt vier Typen von Ergebnismengen eines Funktionsaufrufs oder einer Expr:

- Eine Menge von XPath Knoten
- Eine Zahl
- Ein boolscher Wert
- Eine Zeichenkette oder ein String

Zum Auffinden eines Elements oder einer Elementmenge innerhalb eines Dokuments kann ein Lokationspfad angegeben werden. Es gibt zwei Arten von Lokationspfaden. Die eine Art ist der relative Lokationspfad und die andere Art ist der absolute Lokationspfad, wobei auch der absolute Lokationspfad nur relativ bezüglich des Wurzelelements des Dokuments ist. Dies ist zu beachten, wenn über mehrere Dokumente oder Dokumentfragmente abgefragt werden soll. Deswegen gibt es bei einer Pfaddefinition immer ein Kontextelement. Beim absoluten Lokationspfad ist das Wurzelelement das Kontextelement und beim relativen Lokationspfad wird das Kontextelement explizit angegeben.
Jeder Lokationspfad besteht aus Schritten. Ein Schritt besteht aus einer Achse, einem Elementnamen und einer Menge von optionalen Filtern. Dabei gilt, dass der Schritt die Elementmenge filtert. Ein Lokationspfad wird von links nach rechts ausgewertet und mit jedem Schritt wird die Elementmenge kleiner. Die Problematik von XPath ist das Filtern, denn hierbei kann der Kontext des Weges eine Rolle bei der Entscheidung zum Filtern spielen. Dadurch können zuerst große Informationsräume erzeugt werden, die erst spät

gefiltert werden. Beim Datenmodell von XPath wird das Dokument als Baum aufgefasst. Der Baum besitzt unterschiedliche Knotentypen:

Wurzelknoten: Die Wurzel des XML-Dokuments, die alle anderen Knoten, also auch den Dokumentknoten enthält.

Elementknoten: Ein XML-Element wird als Knoten aufgefasst, an diesem Knoten können weitere Knoten hängen.

Textknoten: Der Textinhalt zwischen dem Anfangs- und Endauszeichner eines Elements.

Attributsknoten: Alle Attribute eines Elements werden in Knoten umgewandelt. Die Reihenfolge der Attribute legt der Parser fest.

Namensraumknoten: Wenn an einem Element ein Namensraum definiert wird, ist der Namensraum als ein eigenständiger Knoten aufgefasst.

Verarbeitungsanweisungsknoten: Wenn an einem Elementknoten Verarbeitungsanweisungen stehen wird ein Knoten erzeugt.

Kommentarknoten: Für einen Kommentar wird ein eigener Knoten erzeugt und kann so ausgelesen werden.

Es gibt eine ähnliche Knotenmenge bei Document Object Model [WAB+98], doch es gibt Unterschiede in der Semantik. Bei den Element- und Wurzelknoten unterscheidet sich der Zeichenkettenwert der Knoten vom Wert der Zeichenkette, der von der DOM Methode *nodeValue* zurückgegeben wird. Die Auswertung von XPath mit den aktuellen DOM-Implementierungen liegt im P-Space und hat eine exponentielle Laufzeit [GKP05].

2.3.2. Parser von XML

Allgemein können die XML-Dateien als Bytestrom betrachtet werden. Der Bytestrom wird in Tokens umgewandelt, wobei diese wiederum in XML-Fragmente transformiert werden. Die Informationen zum Erstellen der Fragmente werden aus den Metadaten von XML gewonnen. Das Parsen von XML ist in der Regel zeitaufwendig und ist das größte Performance Problem von XML. Es gibt verschiedene Methoden zum Parsen von XML je nachdem wie XML weiterverarbeitet werden soll. Bei einem BPEL4WS Prozess verwenden die eingehenden Nachrichten ein XML-Format und müssen häufig geparst werden, um auf Teile der Nachrichten zugreifen zu können. Desweiteren wird XML zum Beschreiben der Variablen des Prozesses genützt. Deshalb wird eine Optimierung des Zeitaufwandes beim Parsen jeden BPEL4WS Prozess beschleunigen.

DOM bedeutet Document Object Model und ist eine Spezifikation, die das XML Dokument als einen Baum mit Knoten definiert. Es gibt verschiedene Implementierungen von DOM [WAB+98]. Eine davon ist JDOM [Hun01] und ist in Java geschrieben. Daher ist sie für eine DOM Implementierung vergleichweise schnell. Aber es wird eine Alternative

zu DOM gesucht, denn alle DOM Implementierungen bauen einen zu großen Informationsraum auf, der für die Aufgabenstellung unnötig ist.

VTD-Parser haben eine ähnliche Auffassung hinsichtlich eines XML-Dokuments wie DOM, nur dass man den Schwerpunkt auf einen lesenden Zugriff gelegt hat. Wird das Dokument verändert, so muss die Struktur komplett erneuert werden. Zudem ist die Suchstruktur optimiert. Die Baumstruktur ist in einer Liste gespeichert. Dabei wird auf eine effiziente Speicherverwaltung Wert gelegt. XPath wird auf die Struktur übersetzt und dann ausgewertet. Die Implementierung ist noch nicht fertig, aber für die Aufgabenstellungen von BPEL4WS ausreichend und deutlich schneller als alle DOM Implementierungen. Trotzdem stellt sich die Frage, ob der Neuaufbau der Struktur nach einem schreibenden Zugriff nicht einen zu negativen Einfluss auf das Laufzeitverhalten hat [Zha04].

SAX steht für Simple API XML [Meg00] und ist eine Spezifikation, die ein XML-Dokument als eine Menge von Tokens ansieht. Das Dokument wird sequentiell abgearbeitet und dabei wird je nach Typ des Tokens ein Ereignis ausgelöst. Die Ereignisse werden an einen Zuhörer übergeben, der dann reagieren kann. Dadurch können schnell bestimmte Elemente oder Attribute gefunden werden. Allerdings muss man dies von Hand steuern und eigenständig den XPath auswerten. Ohne eine Erweiterung für eine Automatisierung von XPath ist dieser Ansatz also nicht brauchbar. Durch eine Automatisierung sollte es nun möglich sein aus der Menge der unterstützten XPath Anweisungen eine auszuwählen. Die Anweisung sollte dann von der Implementierung ausgeführt werden. Die Implementierung übersetzt dann die Anweisung in einen Kontroller für den SAX Parser. Dieser verarbeitet dann ein beliebiges XML Dokument. Anschließend führt der Kontroller die Anweisung aus und liefert das Resultat. Der Kontroller wird also automatisch erzeugt.

Pull Parser [Slo02] sehen ein XML-Dokument als eine Menge von Tokens an und sind ähnlich wie die SAX Parser. Aber hier werden keine Ereignisse ausgelöst, sondern man treibt den Parser mit dem Aufruf: *„gib mir das nächste Token aus“* an. So spart man sich den Aufwand für die Benachrichtung des Zuhörers. Allerdings stellen sich hier alle Probleme, die auch für den SAX gelten. Sollten aber diese Probleme beseitigt sein, dann ist der Pull Parser schneller als eine SAX Implementierung. Für die Diplomarbeit wurde aus der Vielzahl von möglichen Implementierungen XPP3 ausgewählt.

XPath Auswertung Der klassische Ansatz zum Auswerten von XPath ist die Umwandlung des Dokuments in einen DOM Baum. Der VTD-Parser baut zwar eine schlankere Baumstruktur auf, doch wird auch hier der gesamte Baum aufgebaut, um eine XPath Anweisung auszuwerten.
Es gibt einen anderen Ansatz, der auf unterschiedliche Art umgesetzt wurde: Spex [OFB03] oder Taskforce XML [GKP05]. Beide verwenden den Ansatz, dass man die XPath Anweisungen als Steueranweisungen für eine Parser übersetzt. Dann reicht ein einfacher Parser wie ein SAX Parser oder wie ein Pull Parser um XPath auszuwerten.

Man bezeichnet diesen Ansatz als „automatisierten Stream Parser“. Wenn etwas kompiliert wird, kann es in der Regel für das Zielsystem optimiert werden, um so weitere Verbesserungen zu erhalten.

2.4. WSDL

WSDL verwendet XML als Format für die Nachrichten und ist ein abstraktes Protokoll für Netzdienste. Die Nachrichten werden mit Simple Object Access Protocol verteilt. Das Interessante an WSDL ist das Datenmodell [CCMW01]. Es gibt eine Input- und eine Output-Variable, die beide innerhalb der Nachricht definiert werden. Die Datentypen beider Variablen werden über das xsd-Schema und das xsl-Schema festgelegt. Zusätzlich kann man über den Namensraum auf andere WSDL Nachrichten bezüglich komplexer Datentypen verweisen, so dass hier eine Wiederverwendung möglich ist.

2.5. BPEL4WS

BPEL4WS ist eine Beschreibungssprache für Geschäftsprozesse und eine Weiterentwicklung von XLANG und WSFL. Die Diplomarbeit verwendet die Version 1.1 von BPEL4WS [IDB+03]. BPEL4WS erlaubt es, ausführbare sowie abstrakte Prozesse zu beschreiben. Die Konstrukte für einen abstrakten Prozess werden nicht berücksichtigt, da diese nicht ausführbar sind. Man kann die Konstrukte von BPEL4WS wie folgt ordnen:

Daten: Zu den Daten werden folgende Konstrukte gezählt: *Variables*, *Messages*, *Correlations*.

Aktivitäten: Zu den Aktivitäten gehören alle Aktivitätskonstrukte und Verknüpfungen.

Prozessinformationen: Alle Prozessinformationen stehen beim Wurzelelement des BPEL4WS Dokuments.

Kommunikationen: Zu der Kommunikationskonstrukten zählen die *PartnerlinkTyp*, *PartnerLink* und *Partners*.

Der Fokus der Diplomarbeit liegt auf den Daten von BPEL4WS und allen Aktivitäten, die die Daten von BPEL4WS manipulieren, da gerade hier immer wieder XML erzeugt oder gelesen werden muss. Deswegen ist eine Optimierung der Lese- und Schreibzugriffe auf die Daten sinnvoll. Grundsätzlich werden die Daten eines Prozesses in den Variablen gehalten und mit Hilfe der Variablen übertragen. Die Variablen können von drei unterschiedlichen Datentypen sein:

- der WSDL Nachrichtentyp,
- der XML *schema simple type*
- XML-elements.

Die Bindung zwischen Variable und Datentyp ist statisch. Der Name einer Variable ist eindeutig im Bezug auf den Geltungsbereich. Es kann aber vorkommen, dass eine Variable von einer gleichnamigen Variable, die in einem tieferliegenden *Scope* definiert wurde, verdeckt wird. Dadurch ist der Zugriff über den Namen der Variablen nur innerhalb eines Scopes eindeutig.
Zur Bestimmung der Gleichheit von Variablen wird auf die Namensäquivalenz des Datentyps der Variable geprüft. Zugriffe auf den Inhalt einer Variablen erfolgen entweder über ein Assignment oder über eine Expression in einer Aktivität. Als Sprache für die Expression wird XPath verwendet, die um folgende Ausdrücke erweitert wurde:

- *bpws:getVariableProperty ('variableName', 'propertyName')*
- *bpws:getLinkStatus ('linkName')*
- *bpws:getVariableData ('variableName', 'partName', 'locationPath')*

Die Bindung der Zugriffe hinsichtlich der Variablen erfolgt über den Namen und ist statisch. Also liegt eine statische Bindung sowohl der Zugriffe auf die Variablen wie auch die Typbindung der Variablen eines BPEL4WS Prozesses zur Laufzeit vor. Das bedeutet, dass der Typ oder der Zugriff auf eine Variable nicht während der Laufzeit verändert werden kann. Deswegen kann man den Prozess vor der Laufzeit analysieren und sinnvoll optimieren [ASU99].

2.5.1. XPath

In BPEL4WS wird XPath verwendet und entsprechend erweitert. Der Verwendungszweck ist der Zugriff, die Auswertung und die Manipulation der Variablen von BPEL4WS. Pfade sollten immer innerhalb des Funktionsausdrucks *bpws:getVariableData()* oder innerhalb eines *Assignment* benutzt werden. Das schränkt zwar die Freiheit des Entwicklers ein, aber nicht seine Möglichkeiten des Datenzugriffs. Somit wird die Analyse von XPath erheblich erleichtert. Die Verarbeitung ist so eindeutig.
Eine wichtige Frage ist die effiziente Auswertung von XPath. Die heute gebräuchlichen Verfahren haben ein exponentielles Laufzeitverhalten. Der Artikel [GKP05] zeigt, dass es auch besser geht, doch sein Schwerpunkt lag auf der Verarbeitung von großen XML Dateien. Zwar haben auch Geschäftsprozesse große Datenmengen, doch die sind auf viele Variablen verteilt. Daraus ergibt sich die zentrale Fragestellung dieser Diplomarbeit. Im Folgenden soll überprüft werden, ob bei der Verarbeitung großer, verteilter Datenmengen ein Stream Parser schneller arbeiten kann, als ein DOM Parser.

2.6. Datenflussanalyse

Durch die statischen Eigenschaften von BPEL4WS ist es möglich, schon Optimierungen während der Vorbereitung eines Prozesses auszuführen. Solche Fragestellungen sind im Compilerbau für Programmiersprachen bereits wohlbekannt. Dort werden diese Fragestellungen mit Hilfe von Datenflussanalysen gelöst. Sie bieten die Möglichkeit Aussagen über das Verhalten der Daten eines Programms zu erhalten.

Wie die Theorie bewiesen hat, sind die allgemeinen Fragestellungen unentscheidbar. Doch die Datenflussanalyse erlaubt es, eingeschränkte Aussagen über das Verhalten der Daten zu treffen. Entscheidend ist, wieviele Register benötigt werden, um alle aktiven Variablen im Speicher zu halten. [ASU99]
Die Grundlage der Datenflussanalyse ist die Bildung einer Übermenge, die ein Modell für die gesuchte Fragestellung darstellt. Jede Operation der realen Daten wird auf die Übermenge abgebildet. Die Aussagen gelten nur, wenn die Analyse eine positive Antwort erbringt, anderenfalls ist die Antwort nicht auf ein reales System übertragbar. Das bedeutet, dass wenn keine Fehler in der Übermenge für die Fragestellung gefunden werden, dies auch für die realen Daten gilt. Sollte hingegen ein Fehler gefunden werden, so kann dieser im realen System nicht existent sein. Durch die Fixpunkteigenschaft der Übermenge terminiert die Datenflussanalyse immer und in der Praxis sind die Antworten schnell berechnet.
Der Zweck der Datenflussanalyse in der Diplomarbeit ist zum einen die Überprüfung der Fragestellung, ob man am Beispiel einer *Assignment* Aktivität ohne einen DOM Parser auskommen kann, indem man die XPath Anweisungen analysiert und diese für einen einfachen Parser umsetzt. Zum anderen soll untersucht werden, ob durch die Datenflussanalyse das Parsern im Allgemeinen verbessert werden kann. Hieraus lassen sich weitere interessante Punkte ableiten wie beispielsweise der Bedarf an gleichzeitig verwendeten Parsern oder wann der richtige Zeitpunkt für die Durchführung des Parsens eines Dokuments ist.

3. Analyse von SWoM

Die bestehende Implementierung wurde im Rahmen eines Studienprojektes des Studiengangs Softwaretechnik am Institut für Architektur von Anwendungssystemen der Universität Stuttgart im WS 05 bis SoSe 06 erstellt. Die Zielsetzung des Studienprojektes war es, ein lauffähiges erweiterbares System zu erstellen, das BPEL4WS ausführen kann. Es sollte die Möglichkeit geben, den Prozess bei seiner Ausführung zu überwachen. Das Projekt dauerte zwölf Monate und wurde erfolgreich beendet. In der Endphase wurde erkannt, dass eine Optimierung der XML Verarbeitung das Laufzeitverhalten von SWoM verbessern würde. Deswegen wurden nach einer ersten Analyse zwei Diplomarbeiten [Fel06] begonnen, um das Laufzeitverhalten von SWoM zu verbessern.
Die Fragestellung dieser Diplomarbeit ist, ob man auf den Einsatz eines DOM bei der Verarbeitung einer *Transition Condition* oder bei einem *Assignment* verzichten kann. Es wird im Folgenden aufgezeigt, dass es durch den Einsatz eines Stream Parsers anstelle eines DOM Parsers möglich ist, das Laufzeitverhalten von SWoM positiv zu beeinflussen.

3.1. SWoM

SWoM steht für Stuttgart Workflow Mashine und führt einen Geschäftsprozess aus, der in BPEL4WS beschrieben ist. Die zugrundeliegende Idee für SWoM stammt aus dem Buch [LR99]. Es geht um die Ausführung des BPEL4WS Prozesses mit Hilfe von zustandlosen Navigatoren. Die Möglichkeit für eine Implementierung in Java wurde im Artikel [KKL+04] dargestellt. Neben der Implementierung wurden auch die im Rahmen des Studienprojektes erstellten Dokumente, der Entwurf [MS06] und die Spezifikation [CB05] zur Analyse von SWoM verwendet. Die Ideen und Konzepte wurden im Rahmen des Studienprojekts verwendet und weiterentwickelt.

3.1.1. Zentrale Idee von SWoM

Die zentrale Idee von SWoM ist die Verwendung von Navigatoren, da ein BPEL4WS Prozess viele parallele Instanzen besitzen kann. Deswegen ist eine Implementierung sinnvoll, die ohne einen Flaschenhals in der Verarbeitung eines Prozesses auskommt. Das wird erreicht, indem es mehrere Navigatoren gibt. Jeder Navigator verarbeitet eine Prozessinstanz. Alle Instanzen, die ausführbar sind, werden möglichst gleichzeitig von den Navigatoren bearbeitet. So können dann alle Prozessinstanzen gleichzeitig ausgeführt weden.
Damit die Kommunikation zwischen den Instanzen nicht zum Flaschenhals wird, werden Nachrichten und Warteschlangen verwendet. Alle Informationen zum Ausführen einer Aktivität einer Prozessinstanz müssen in der Nachricht enthalten sein, die der Navigator erhält. So kann jeder Navigator jede Aktivität einer Porzessinstanz ausführen. Daraus folgt der grobe Ablauf einer Prozessinstanz. Ein Navigator erhält eine Nachricht. Mit den darin enthaltenen Informationen der Aktivitäten einer Prozessinstanz führt der Navigator die Aktivität aus. Zusammen mit den Systemkomponenten für die Datenhaltung, die Kommunikation und dem Einlesen eines BPEL4WS Prozesses, erhält man den folgenden grundlegenden Entwurf von SWoM.

3.1.2. Der grundlegende Entwurf von SWoM

Die Abbildung 1 zeigt das Muster des Entwurfs.

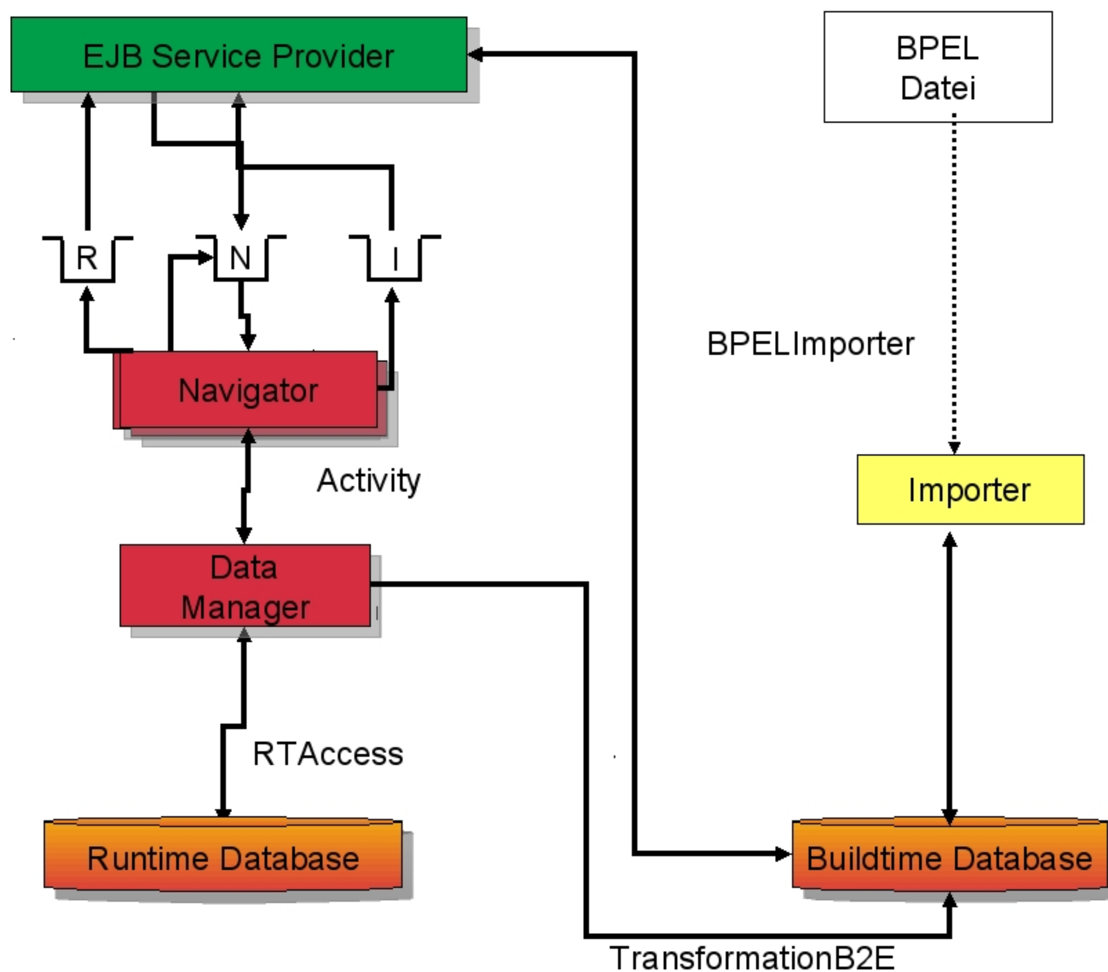

Abbildung 1: Schaubild von SWoM

- Es gibt einen *Service Provider*, der die eingehenden Nachrichten über SOAP empfängt und die Nachrichten mittels des *Correlation* Konzepts von BPEL4WS zu einer Prozessinstanz verbindet. Es wird für jeden Webdienst eines Prozesses eine eigene *Enterprise Java Bean* erzeugt. Diese wird aus den Informationen des Prozesses erzeugt. Zum *Service Provider* wird auch der *Correlation Handler* gerechnet. Der *Service Provider* hat drei Schnittstellen:
  - Zu der *Buildtime* Datenbank, um Informationen abzufragen.
  - Zu der *Runtime* Datenbank, um Informationen zu speichern.

  - Zu den Warteschlangen, in die die Nachrichten an die Navigatoren abgelegt werden und von denen die Nachrichten abgeholt werden, die an die Webdienste gehen sollen.

- Die Navigatoren empfangen Nachrichten aus den Warteschlangen und senden Nachrichten an die Warteschlangen. Die Nachrichten enthalten die Informationen über die Aktivität. Mit den Informationen holt sich der Navigator vom *Data-Manager* die Aktivitäten. Die Aktivitäten werden von den Navigatoren ausgeführt. Jeder Navigator führt genau eine Aktivität aus.

- Der *Data-Manager* ist ein Zwischenspeicher und sorgt dafür, dass die Daten aus der *Buildtime* Datenbank einer aktiven Prozessinstanz im Speicher gehalten werden. Hier werden alle Daten aus der *Buildtime* Datenbank, die für die Ausführung einer Prozessinstanz notwendig sind, abgelegt. Desweiteren wird über den *Data-Manager* auf die *Runtime* Datenbank zugegriffen, um so alle Daten, die bei der Ausführung einer Prozessinstanz benötigt werden, zu speichern oder zu laden. Diese Daten sind zum einen der Inhalt der Variablen und zum anderen die Kontrollinformationen der Prozessinstanz. Der *Data-Manager* hat Schnittstellen zu den beiden Datenbanken, um die Daten persistent zu speichern und ebenso zu laden. Die Schnittstellen zu den Navigatoren stellen die Aktivitäten und die Warteschlangen der Navigatoren dar. Die Aktivitäten sind Bestandteile des *Data-Managers*.

- Die *Runtime* Datenbank dient zur persistenten Speicherung aller Informationen, die eine Prozessinstanz zur Laufzeit erzeugt. Über die *RTAcess* wird auf die Datenbank zugegriffen.

- Die *Buildtime* Datenbank ermöglicht das persistente Speichern des Prozessmodells. Die BPEL4WS Datei und alle weiteren notwendigen Dateien werden in Tabellen abgebildet. Aus diesen Tabellen werden dann die Prozessinstanzen und der *Service Provider* erzeugt. Auf dieser Datenbasis wird auch ein Prozess auf seine Gültigkeit hin überprüft.

- Der *Importer* lädt alle Dateien und speichert sie in der *Buildtime* Datenbank ab.

- Die Aufzeichnungskomponente *Audit* soll das Verhalten des Gesamtsystems aufzeichnen und speichern. Die Komponente ist nicht für diese Diplomarbeit relevant und wird daher aus Gründen der Überschaubarkeit nicht weiter verfolgt.

3.1.3. Realisierung von BPEL4WS in SWoM

SWoM bildet im Moment nur Teile der BPEL4WS Spezifikation ab, diese Teile sind jedoch ausführbar. Im Folgenden sollen mehrere Aspekte der Implementierung von BPEL4WS aufgezeigt werden. Zuerst wird das Importieren von BPEL4WS in die *Buildtime* Datenbank, dann wird die Datenverarbeitung und zuletzt der grobe Ablauf eines Prozesses erläutert.

Importieren vom BPEL4WS Das Importieren von BPEL4WS erfolgt in zwei Schritten:

Schritt eins: Der Geschäftsprozess und alle seine Konstrukte werden in eine relationale Datenbank *(Buildtime)* importiert. Die Aktivitäten und die *Partnerlinks* werden jeweils in einer Tabelle gespeichert. Die Variablen werden ebenfalls in einer Tabelle gespeichert, ebenso wie die Beziehung zweier Aktivitäten zueinander über einen *Link*. Jedes Konstrukt bekommt eine interne *ID*.

Schritt zwei: Eine Projektinstanz wird mit Java-Objekten ausgeführt. Die Aktivtäten sind als Objekte realisiert. Die *Partnerlinks* sind Attribute der Aktivitätsobjekte. Die Variablen werden als Objekte abgebildet, die Daten werden in XML gespeichert. Neben den Namen der Aktivitäten und Variablen besitzen die Objekte noch die *ID* aus der Tabelle. Die Links werden als Attribute der Aktivitätsobjekte abgebildet.

Datenverarbeitung Die Datenverarbeitung von BPEL4WS besteht aus den Variablen, den XPath-Anweisungen und der *Assignment* Aktivität. Die Umsetztung der Variablen von BPEL4WS erfolgt in SWoM folgendermaßen:

- Die Variable wird als ein Objekt implementiert. Die Daten der Variablen werden mit einer Hashtable verwaltet, die in der Variable abgelegt ist. SWoM unterstützt nur Variablen, die als WSDL Nachrichtentyp definiert sind. So besteht jede Variable aus Teilbäumen mit dem Schlüsselwort *part*. Die Unterbäume werden auch als *Parts* bezeichnet. Da der Name eines *Parts* innerhalb des Nachrichtentyps eindeutig ist, verwendet SWoM den Namen eines *Parts* als Schüssel für die Hashtable. Das Datenobjekt innerhalb der Hasahtable ist ein String. So kann mit dem Namen des *Parts* auf den Datenstring des Teilbaums zugegriffen werden (siehe Abbildung 2 Seite 16).

- Wenn eine Nachricht bei der SWoM eintrifft, wird die betreffende Nachrichtenvariable ausgelesen. Dies erfolgt über die Methoden von XSUL. Die Methoden liefern eine Liste der *Parts* der Variablen der Nachricht. Die Liste wird dann in die Hashtable der SWoM-Variable gespeichert.

- Wenn SWoM eine Nachricht verschickt, wird eine Liste der *Parts* aus der Nachrichtentypdefinition erzeugt. Für jedes Element der Liste wird die Hashtable abgefragt, die so erhaltenen Datenstrings werden dann zu einer Variable zusammengefügt.

Eine Variable in SWoM besteht also aus XML Fragmenten, die über den Namen des Teilbaums in einer Hashtable verwaltet werden. Wenn zum Auffinden von Daten einer Variable ein Lokationspfad verwendet wird, hat SWoM folgendes Laufzeitverhalten:

1. Aus der XPath Anweisung wird der Name der Variable ausgelesen.

2. Mit dem Namen erhält man vom *Data-Manager* die *ID* der Variable.

3. Mit der *ID* erhält man vom *Data-Manager* die Variable.

Hashtable

Key	Data
part1	<part1>... </part1>
part2	<part2>... </part2>
part3	<part3>... </part3>
part4	<part4>... </part4>

Abbildung 2: Schaubild einer gefüllten Hashtable

4. Von der Variable erhält man den Datenstring.
5. Der Datenstring wird in einen DOM Baum umgewandelt.
6. Auf diesem DOM Baum wird der XPath ausgeführt
7. Man erhält als Ergebnis den Teilbaum, der durch die XPath Anweisung beschrieben wird.
8. Der Ergebnisbaum wird dann wieder in einen Datenstring umgewandelt.

Zur Verdeutlichung wird aus dem Beispiel *travelOrder* die Variable *Hotelreservation* 3.1.3 genommen. Auf der Variable wird folgende XPath Anweisung ausgeführt. Verdeutlicht wird die Aktion durch die Abbildung 3.

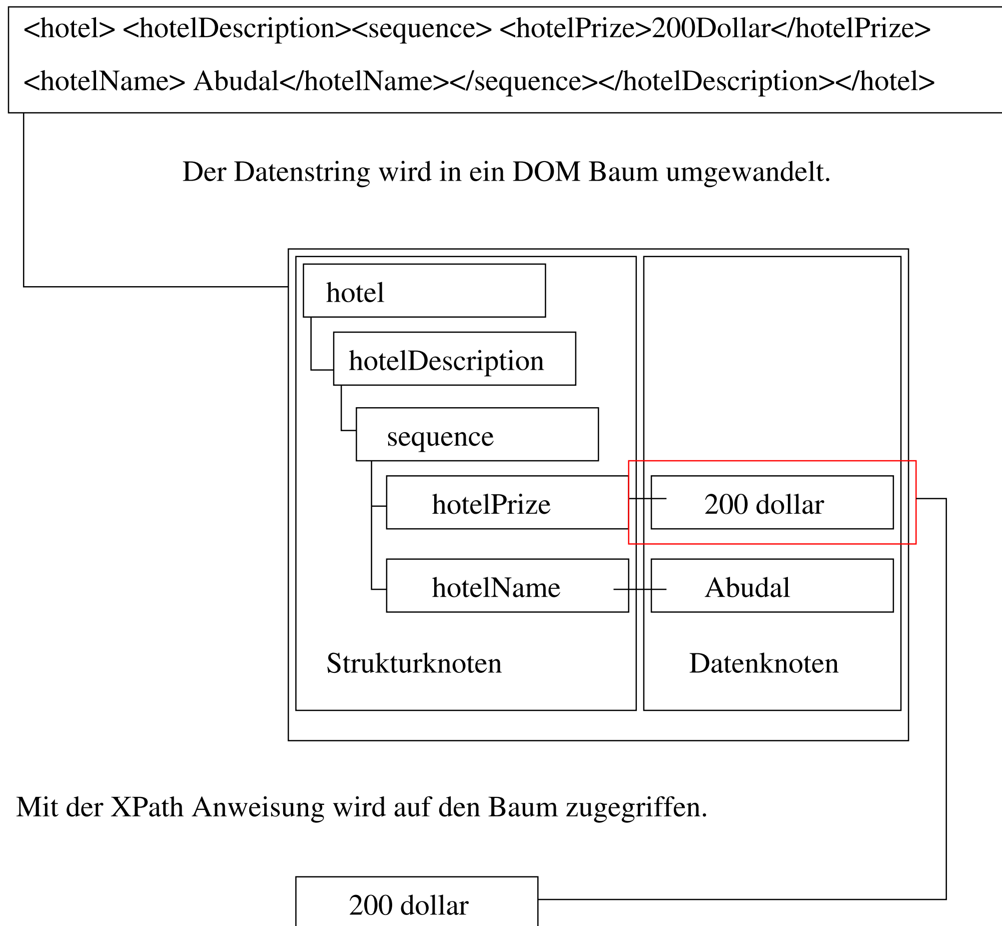

Abbildung 3: Schaubild der Verarbeitung einer XPath

```
expression="bpws:getVariableData('hotelReservation','hotel','/
    child::hotelDescription/child::sequence/child::hotelPrize')"

<hotelReservation>
    <hotelReservationInput>
        <hotel>
            <hotelDescription>
                <sequence>
                    <hotelName> Abudal</hotelName>
                    <hotelPrize>200 Dollar</hotelPrize>
                </sequence>
```

```
        </hotelDescription>
      </hotel>
    </hotelReservationInput>
</hotelReservation>
```

Wenn man das Verfahren analysiert, stellt man fest, dass der Zwischenschritt über den Namen zum Auffinden der Variable unnötig ist, da man schon beim Importieren des BPEL4WS Prozesses den Namen durch die *ID* ersetzen kann, da bei einer Transition Condition der Datenzugriff nur über eine BPEL4WS XPath Funktion erfolgt. So kann man alle Datenzugriffe schon beim Importieren erkennen. So wird der Name der Variable bei einer Assignment Aktivität schon durch die ID ersetzt. Der Aufbau des DOM Baums braucht viel Zeit, während die XPath Auswertung durch die Baumstrukur schnell ist. Es liegt nahe keinen DOM Baum aufzubauen, um ein besseres Laufzeitverhalten zu erhalten.

1. Der Name wird nicht erst zur Laufzeit sondern schon beim Importieren durch eine *ID* ersetzt, sobald die Beziehung zwischen dem Namen und der internen *ID* vorliegt.

2. Der Aufbau eines DOM Baums braucht viel Zeit. Als Alternative sollte ein Stream Parser verwendet werden. Man benutzt einen Pull Parser und übersetzt die XPath Anweisung in Steueranweisungen für den Pull-Parser. Der übersetzte XPath Ausdruck steuert den Pull-Parser und speichert das Ergebnis der XPath Anweisung. Da der Pull Parser sehr schnell ist, sollte diese Lösung sehr effizient sein. Die offene Frage ist, ob man so alle in der BPEL4WS Spezifikation beschriebenen Funktionen einer XPath Anweisung erfüllen kann.

Würde man die Alternative auf das Beispiel anwenden, dann erhält man folgende Abbildung 4:

Eine XPath Anweisung wird zur Laufzeit ausgewertet.

1. Die XPath Anweisung holt sich mit der *ID* die Variable vom *Data-Manager*.
2. Von der Variable erhält man den Datenstring.
3. Der Datenstring wird mit Hilfe der übersetzten Steueranweisung durch den Pull Parser ausgewertet und gibt das Ergebnis als Datenstring aus.

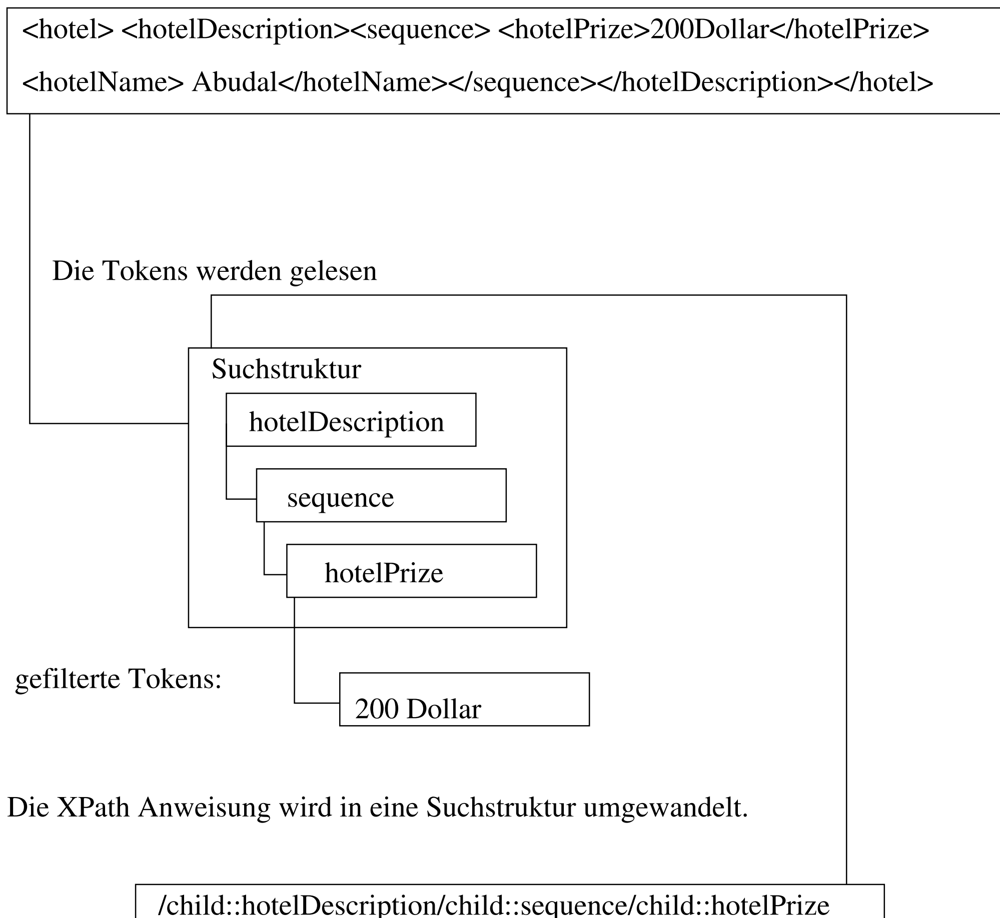

Abbildung 4: Schaubild der neuen XPath

Die schnellere Auswertung zur Laufzeit wird durch eine längere Laufzeit beim Importieren von BPEL4WS nach SWoM erkauft. Das hat aber keinen Einfluss auf das Laufzeitverhalten von SWoM, wenn es eine Prozessinstanz ausführt. Bei einer *Assignment* Aktivität gibt es noch eine weitere Nutzung von Lokationspfaden. Es wird hier der Lokationspfad nicht nur zum Auffinden von Daten, sondern auch zum Einfügen von Daten

verwendet. Im folgenden Beispiel wird das Verhalten von SWoM beschrieben und dann das Verhalten wenn man den Pull Parser verwendet. SWoM verwendet hierbei wieder die JDOM Bibliothek. Es kommt zu folgendem Ablauf der Operationen:

1. Mit der *ID* erhält man vom *Data-Manager* die Variable.
2. Von der Variable erhält man den Datenstring.
3. Der Datenstring wird in einen DOM Baum umgewandelt.
4. Auf diesem DOM Baum wird die XPath Anweisung ausgeführt.
5. Als Ergebnis der Anweisung erhält man die Position im DOM Baum.
6. An der Position werden dann die neuen Daten in den DOM Baum eingefügt.
7. Der so veränderte DOM Baum wird dann in einen Datenstring umgewandelt.

Die Lösung mit dem Pull Parser hat folgenden Ablauf:

1. Die XPath Anweisung holt sich mit der *ID* die Variable vom *Data-Manager*.
2. Von der Variablen erhält man den Datenstring.
3. Der Datenstring wird mit Hilfe der übersetzten Steueranweisung durch den Pull Parser ausgewertet und gibt so das gefundene Element als einen String aus.
4. Mit Hilfe der Methoden der String Klasse wird der Datenstring eingefügt. Zuerst wird die Position des Elementstrings im alten Datenstring gefunden, dann wird der alte Datenstring mit Hilfe der Position in drei Teile gespalten und das mittlere Stück wird durch den neuen Datenstring ersetzt.

Ablauf eines Prozesses

Das Vorbereiten eines Prozesses Dabei werden zuerst alle notwendigen Dateien eingelesen und in die *Buildtime* Datenbank geschrieben. Hierbei werden die Konstrukte in eine relationale Datenbank abgebildet. Beim Importieren werden die Daten innerhalb der *Buildtime* Datenbank für das gleichzeitige Ausführen mehrerer Prozessinstanzen vorbereitet. Dann wird der Prozess validiert und anschließend werden die notwendigen Komponenten im *Service Provider* erzeugt, um die Nachrichten für einen Prozess richtig zu behandeln. Somit ist der Prozess ausführbar.

Ausführen einer Prozessinstanz Ein Prozess wird von außen gestartet, dieser wird durch eine eingehende Nachricht gestartet. Wenn sie beim *Service Provider* eintrifft, stellt der fest, dass noch keine Prozessinstanz läuft, aber der Prozess ausführbar ist. Dann wird diese Nachricht in eine interne Nachricht für die Warteschlange umgewandelt und ihr werden die Variable der Nachricht und alle Informationen für den Navigator mitgegeben. So bekommt ein Navigator die Nachricht von der Warteschlange

übergeben. Der Navigator holt sich die Aktivität beim *Data-Manager*. Beim Holen der Aktivität wird festgestellt, dass diese Aktivität eine Start Aktivität der Prozessinstanz ist.

Der *Data-Manager* erzeugt daraufhin eine Prozessinstanz, die bei ihm und in der *Runtime* Datenbank angelegt wird. Nachdem die Prozessinstanz erzeugt wurde, wird die angeforderte Aktivität endgültig erzeugt und dem Navigator übergeben. Danach führt der Navigator die Aktivität aus. Die Aktivität speichert ihre Daten ab und stellt den Nachfolger fest. Dies kann entweder die umschließende Aktivität sein, eine Aktivität, die über einen *Link* zu erreichen ist oder die Termination des Prozesses sein. Die Aktivität stellt den Nachfolger fest und erzeugt die passende Nachricht, die der Navigator in die Warteschlange legt. Jede Aktivität wird wie folgt abgearbeitet: Der Navigator erhält eine Nachricht mit der eindeutigen *ID* einer Aktivität, die er sich vom *Data-Manager* holt. Der *Data-Manager* gibt die Aktivität zurück. Die Aktivität wird dann vom Navigator ausgeführt. Danach wird dann der Nachfolger bestimmt.

Bewertung Aus diesem groben Ablaufschema wird klar, dass die beiden Phasen eindeutig getrennt sind. Eine Verlagerung der Last in die erste Phase beschleunigt die zweite Phase. Somit kann man es sich leisten, teure Analyseverfahren zu verwenden, um die Ausführung der Prozesse zu beschleunigen, da die Durchführung der Analyse zu keinem veränderten Verhalten des Prozesses führt. Für eine bessere Darstellung des Ablaufs wird ein kleiner vollständiger BPEL4WS Prozess mit SWoM zur Laufzeit durchgespielt.

3.2. Eine genaue Analyse von SWoM

Für eine genaue Analyse des Ablaufs eines BPEL4WS Prozesses auf der SWoM nehmen wir das Beispiel einer Reisebuchung. Kurz gefasst das Beispiel sieht so aus, dass eine eingehende Anfrage in zwei Anfragen aufgespaltet wird. Die erste Anfrage geht an ein Hotel und die zweite geht an eine Fluggesellschaft. Beide schicken eine Bestätigung. Die beiden Bestätigungen werden zu einer Bestätigung zusammengefasst und zurück an den ursprünglichen Anfrager geschickt. Der Prozess *travel order* ist mit BPEL4WS beschrieben. Das Beispiel wird mit der bestehenden Implementierung von SWoM auf Klassenebene durchgespielt. Zum Aufzeigen der Schwächen der bisherigen Implementierung wird an den entscheidenden Stellen der alternative Ablauf dargestellt. Der alternative Ablauf spiegelt die gefundene Lösung der Diplomarbeit wieder, so dass deutlich wird, warum der alternative Ablauf optimaler ist als die alte Implementierung.

3.2.1. Beispiel

Die Abbildung 5 verdeutlicht das Beispiel. Die BPEL-Datei befindet sich im Anhang A.

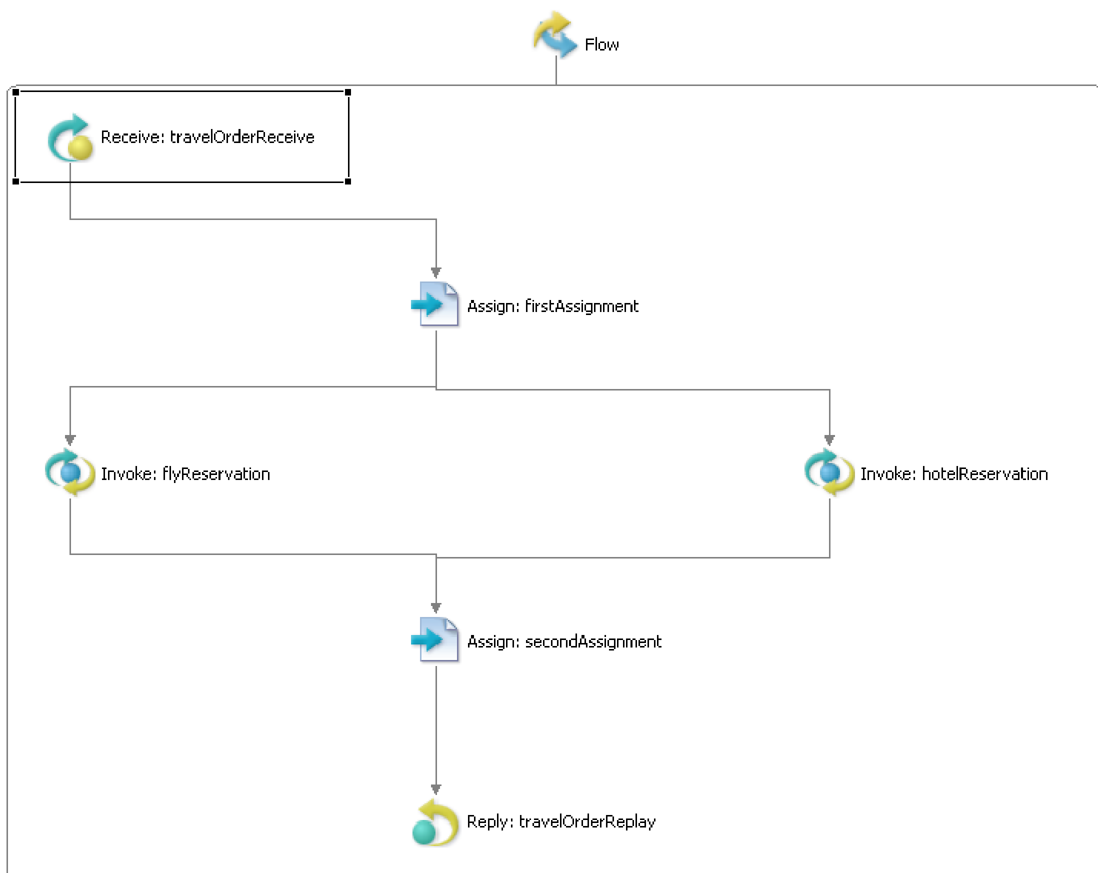

Abbildung 5: Schaubild des Reisebuchungs Prozesses

3.2.2. Einlesen des Beispiels

Um einen BPEL4WS Prozess auszuführen, liest die SWoM die BPEL4WS-Datei und alle dazugehörigen WSDL-Dateien ein. SWoM hat hierfür ein eigenes Archiv festgelegt. Das Archiv umfasst die WSDL-Dateien, die BPEL4WS Datei sowie eine Datei, die alle notwendigen Konfigurationen enthält. Den genauen Aufbau findet man in der SWoM Spezifikation [CB05]. Der Anwender gibt die Lokalisation der Archivdatei an und startet den Import der Archivdatei. Zuerst wird die Konfigurationsdatei gelesen, danach die WSDL-Datei und dann die BPEL4WS-Datei. SWoM ruft den BPEL4WS-Importer auf, um eine BPEL4WS-Datei einzulesen. Der BPEL4WS-Importer holt sich mit Hilfe der XML-Beans alle BPEL4WS Konstrukte. Die XML-Beans sind automatisch erzeugte Klassen. Um die Klassen zu erzeugen, werden sie mit XML Schemas beschrieben. Mit Hilfe dieser Klasse kann der Parser so gesteuert werden, dass er automatisch alle gesuchten BPEL4WS Konstrukte identifiziert. Diese werden dann umgewandelt und in die jeweilige Liste gespeichert. Das Erstellen des Schemas und der notwendigen Daten wird vom SWoM Entwickler vorgenommen, das heißt, wenn neue Konstrukte eingefügt werden sollen, muss eine neue SWoM erstellt werden.
So kann der BPEL4WS-Importer mit einfachen Funktionsaufrufen auf die jeweilige Liste jedes Konstrukttyps zugreifen und dabei jedes Konstrukt in einer Tabelle der *Buildtime* Datenbank abbilden. Wenn man den BPEL4WS-Importer auf das Beispiel anwendet, so erhält man sieben Aktivitäten: eine *Flow* Aktivität, zwei *Assignment* Aktivitäten, eine *Replay* Aktivität, eine *Recieve* Aktivität und zwei *Invoke* Aktivitäten. Dazu werden sieben *Links*, sechs Variablen und ein Prozessmodell in die Datenbank als Tabelleneinträge erstellt. Im neuen Ablauf wird eine zusätzliche Tabelle in der *Buildtime* Datenbank für alle Zugriffe erzeugt. Die Zugriffstabelle erleichtert die spätere Datenflussanalyse. Wir erzeugen mit dem Beispiel folgende Tabelle 1.
Bisher wurden die Informationen über die Zugriffe verteilt in der *Buildtime* Datenbank gespeichert, denn die Bearbeitung des Zugriffs passiert nicht in der Variable, sondern an verschiedenen Stellen. Dies ergibt für die Datenflussanalyse das Problem, dass man an verschiedenen Stellen suchen muss und dann auch die *Transition Condition* parsen muss. Zudem weiß man erst zur Laufzeit, ob es einen Datenzugriff außerhalb einer *Assignment* Aktivität gibt. Deswegen wird schon beim Einlesen eines Prozesses nach allen Zugriffen auf eine Variable gesucht und jeder Zugriff mit einer eindeutigen ID versehen. Daher braucht man eine Tabelle der Zugriffe.

ZugriffID	AID	ZielID	VariableID	Part	Query	lesend schreibend
zu000001	assign01	zu000002	travelOrderID	fly		l
zu000002	assign01		flyOrderID	fly		s
zu000003	assign01	zu000004	travelOrderID	hotel		l
zu000004	assign01		hotelOrderID	hotel		s
zu000005	assign02	zu000006	flyReservationId	fly		l
zu000006	assign02		travelReservationID	fly		s
zu000007	assign02	zu000008	hotelReservationID	hotel		l

ZugriffID	AID	ZielID	VariableID	Part	Query	lesend schreibend
zu000008	assign02		travelReservationID	hotel		s
zu000009	assign02		hotelReservationID	hotel	prize	l
zu000010	assign02		flyReservationID	fly	prize	l
zu000011	assign02		travelReservationID	totalprize		s
zu000012	invoke01		hotelOrderID			l
zu000013	invoke01		hotelReservationID			s
zu000014	invoke02		flyOrderID			l
zu000015	invoke02		flyReservationID			s
zu000016	replay01		travelReservationID			l
zu000017	recieve01		travelOrder			s

Tabelle 1: Zugriffstabelle

3.2.3. Überprüfung des eingelesenen BPEL4WS-Prozesses

Die bisherige Implementierung überprüft einige in der BPEL4WS Spezifikation [IDB+03] beschriebenen Bedingungen für einen gültigen BPEL4WS-Prozess sowie auch die Gültigkeit aller *Links*. Hierbei werden Informationsstrukturen aufgebaut, die man auch für Aussagen über die Abhängigkeit der lesendenen Zugriffe bzgl. des schreibenden Zugriffs auf die Variable nutzen kann. Wenn die Informationen jetzt gefunden werden, kann man die Prozessinstanzierung und den Prozessablauf beschleunigen.
Für die globale Optimierung der Verarbeitung von XML ist es notwendig zu wissen, ob man schon vorab den lesenden Zugriff vorbereiten kann. Dazu braucht man eine Datenflussanalyse. Mit der Datenflussanalyse können die Beziehungen zwischen lesenden und schreibenden Zugriffen ermittelt werden. Das Analyseverfahren wird im Kapitel 4.3 genau beschrieben. Zudem kann man erkennen, ob auch auf alle Bestandteile einer Variable zugegriffen wird. Das kann man auch aus der Datenflussanalyse zum Ermitteln der Beziehung zwischen den lesenden und schreibenden Zugriffen ableiten. Wenn man die Datenflussanalyse auf das Beispiel anwendet, erhält man folgende Tabelle:

Variable	letzter schreibender Zugriff	davon abhängige lesende Zugriffe
travelReservation	zu000011	zu000016
hotelReservation	zu000013	zu000009 zu000007
flyReservation	zu000015	zu000005 zu000010
hotelOrder	zu000004	zu000012
flyOrder	zu000002	zu000014
travelOrder	zu000017	zu000001 zu000003

Tabelle 2: letzte schreibende Zugriffe

Die Tabelle 2 sagt aus, dass es für jeden schreibenden Zugriff abhängige lesende Zu-

griffe gibt. Wenn zum Beispiel der Variable travelOrder mit der Zugriffs ID zu000017 die Daten übergeben werden, dann können schon beide Zugriffe zu000001 und zu000003 vorbereitet werden, d.h. die jeweiligen Datenstrings der Zugriffe werden ermittelt und mit der Zugriffs ID gespeichert. Aus Übersichtsgründen wird aber im folgenden Beispiel nicht mit dem Konzept des frühzeitigen Parsens durchgespielt, da das Beispiel deutlich komplexer geworden wäre. Bei der Schaffung einer Prozessinstanz wird beim Erstellen einer Variable die Tabelle 2 abgefragt und alle abhängig lesenden Zugriffe in einer Liste gespeichert. Wenn dann auf eine Variable mit einer schreibenden ZugriffsID zugegriffen wird, dann wird die Liste der abhängigen Zugriffe dieser ZugriffsID abgearbeitet. Die Tabelle 2 wird bei der Gültigkeitsprüfung der Prozessinstanz erstellt.

3.2.4. Die Laufzeitverhalten des Beispiels

Die Architektur des Laufzeitsystems von SWoM wurde schon im vorherigen Abschnitt beschrieben. Wir gehen nun davon aus, dass SWoM bereit ist. Der *Service Provider* empfängt eine Nachricht über SOAP. Er holt sich den *Message Part* der SOAP-Nachricht, erzeugt eine SWoM Variable und setzt die *Parts* der Nachricht in die Variable. Dann liest der *Service Provider* weitere Informationen aus, die für die Anfrage beim *CorrelationManager* benötigt werden. Der *Service Provider* holt sich vom *CorrelationManager* die *Correlation* mit den Parametern der Nachricht und bekommt so eine *piid (process instance identifier)*. Zudem wird festgestellt, dass es sich um eine *Receive* Nachricht handelt. Mit diesen Informationen wird eine *FromFacadeBeanMessage* erstellt und der Navigatorwarteschlange übergeben.
Der Navigator erhält eine Nachricht aus der Navigatorwarteschlange. Er stellt den Typ und die Herkunft der Nachricht fest. Die Nachricht stammt von der *Fassade* und ist eine Navigator Nachricht. Die Attribute der Nachricht werden der neuen Aktivität *Receive travelOrder* übergeben. Danach wird die neue Aktivität ausgeführt. Dabei wird festgestellt, dass die Aktivität den Prozess startet. Deswegen wird das Prozessmodell von der *Buildtime* Datenbank in den *Data Manager* geladen. Danach werden die *Partnerlinks*, die Variablen, Aktivitäten und *Links* aus der Datenbank in den *Data Manager* geladen. Dafür wird die Klasse *TranformationeB2E* verwendet, die den Prozess, die Variablen, die Aktivitäten, den *Faulthandler* und den *Compensationhandler* aus den Tabellen der *Buildtime* Datenbank erzeugt. Im neuen Ablauf werden den Variablen jetzt die Informationen über die Zugriffe und Steueranweisungen für die Parser übergeben. Zugleich werden die Informationen zur Laufzeit in die *Runtime* Datenbank gespeichert. Nachdem alles geladen und gespeichert ist, wird die Aktivität *Receive* und die Variable *travelOrder* der *Runtime* Datenbank übergeben. Jetzt wird die Variable *travelOrder* auf die Zugriffe vorbereitet.
Der Variable werden die *Parts* der Variable übergeben. Für jeden Part des Dokuments wird ein Eintrag in der Hashtable der Variable erzeugt. Der Name des Parts ist der Schlüssel innerhalb der Hashtable (siehe Abbildung 2 Seite 16) Nachdem die Daten der Variable in der *Runtime* Datenbank gespeichert worden sind, beendet sich die *Receive* Aktivität selbst. Dabei wird eine Nachricht an den Navigator mit der *ID* der Vateraktivität geschickt. Ein Navigator erhält die Nachricht mit der *ID* und holt sich die dazugehörige

Aktivität, in diesem Beispiel die *Flow* Aktivität und führt diese aus. Die Flow-Aktivität schickt an einen Navigator eine Nachricht mit der *ID*, die erste *Assignment* Aktivität. Ein Navigator erhält eine Nachricht mit der *ID* der *Assignment* Aktivität, holt sich die dazugehörige *Assignment* Aktivität vom *Data-Manager* und führt sie aus.

alter Ablauf	neuer Ablauf
Die *Assignment* Aktivität wird ausgeführt. Die *Copy* Elemente eines *Assignments* sind hier als eine Liste von *Copy*-Objekten abgebildetet. Die Liste der *Copy*-Objekte wird iteriert. Das erste *Copy*-Objekt wird ausgeführt. SWoM hat für jede unterstützte *from-* oder *to-* Anweisung eines *Assignment* ein eigenes Objekt. Das *VariableQueryFS*-Objekt stellt die Implementierung einer *from variable = ncname part=ncname query=queryString* Anweisung dar und das *VariableQueryTO*-Objekt stellt die Anweisung *to variable = ncname part=ncname query=queryString* dar. Zuerst wird das *VariableQueryFS*-Objekt aus dem *Copy*-Objekt geholt. Dann wird die Variable *travelOrder* geholt. Der Inhalt der Variablen wird von der *Runtime* Datenbank abgefragt. Als nächstes wird die Methode *getContent()* der Variable *travelOrder* ausgeführt. Man erhält Zugriff auf die Hashtable der Variable. Der Name des *Parts* ist der Schlüssel, um den Datenstring aus der Hashtable zu erhalten. Der String wird dann in ein JDOM Dokument umgewandelt. Auf diesen JDOM Dokument wird die *Query* ausgeführt und erhält ein Dokumentfragment zurück. Das Dokumentfragment wird an die *Assignment* Aktivität zurückgegeben.	Die *Assignment* Aktivität wird ausgeführt. Die Liste der *Copy*-Objekte wird iteriert. Das erste *Copy*-Objekt wird ausgeführt. Die Variable *travelOrder* wird geholt und die Methode *getContent()* wird ausgeführt, dabei wird die Zugriffs *ID* (zu000001) übergeben. Innerhalb der Variable wird mit Hilfe der *ID* die *Query* und der Teilstring geholt. Aus der *Query* wird eine Suchstruktur gebaut und dem Parser mit den Datenstrings übergeben. Der Parser liefert das Gesuchte als String zurück und wird an die *Assignment* Aktivität weitergereicht.

alter Ablauf	neuer Ablauf
Nun wird aus dem *Copy*-Objekt das *VariableQueryTS*-Objekt geholt. Dem *VariableQueryTS*-Objekt wird das Ergebnisobjekt übergeben. Das *VariableQueryTS*-Objekt holt sich von der *Runtime* Datenbank den Inhalt der Variable *flyOrder*. Aus dem Inhalt der Variable wird mit dem Namen des *Parts* der Datenstring des *Parts* geholt. Der Datenstring wird in ein JDOM Dokument umgewandelt. Auf diesem wird die *Query* ausgführt. So findet man die Position im JDOM Dokument, wo das Ergebnisobjekt eingefügt wird. Hierzu wird das Ergebnisobjekt in ein Dokumentfragment umgewandelt und eingefügt. Das JDOM Dokument wird in einen String umgewandelt und der String wird in die Hashtable der Variablen *flyOrder* gesetzt, dabei ist der Name des *Parts fly* der Schlüssel innerhalb der Hashtable.	Dann wird die Variable *flyOrder* geholt und ihr wird der String und die Zugriffs *ID* (zu000002) übergeben. Innerhalb der Variable wird mit Hilfe der *ID* die *Query* und der Teilstring, auf die sich die *Query* bezieht geholt. Aus dem *Query* wird eine Suchstruktur gebaut und dem Parser mit den Datenstrings übergeben. Der Parser liefert das Gesuchte als String zurück. Mit Hilfe der Stringoperation wird der alte String durch den übergebenen String ersetzt.
Das nächste *Copy*-Objekt wird ausgeführt. Zuerst wird das *VariableQueryFS*-Objekt aus dem *Copy*-Objekt geholt. Dann wird die Variable *travelOrder* geholt. Der Inhalt der Variable wird von der *Runtime* Datenbank abgefragt. Als nächstes wird die Methode *getContent()* der Variable *travelOrder* ausgeführt. Man erhält Zugriff auf die Hashtable der Variable. Der Name des *Parts* ist der Schlüssel, um den Datenstring aus der Hashtable zu erhalten. Der String wird dann in ein JDOM Dokument umgewandelt. Auf diesem JDOM Dokument wird die *Query* ausgeführt und man erhält ein Dokumentfragment zurück. Das Dokumentfragment wird an die *Assignment* Aktivität zurückgegeben.	Das nächste *Copy*-Objekt wird ausgeführt. Die Variable *travelOrder* wird geholt und die Methode *getContent()* wird ausgeführt, dabei wird die Zugriffs *ID* (zu000003) übergeben. Innerhalb der Variable wird mit Hilfe der *ID* die *Query* und der Teilstring geholt. Aus der *Query* wird eine Suchstruktur gebaut und dem Parser mit den Datenstrings übergeben. Der Parser liefert das Gesuchte als String zurück und wird an die *Assignment* Aktivität weitergereicht.

alter Ablauf	neuer Ablauf
Nun wird aus dem *Copy*-Objekt das *VariableQueryTS*-Objekt geholt. Dem *VariableQueryTS*-Objekt wird das Ergebnisobjekt übergeben. Das *VariableQueryTS*-Objekt holt sich von der Datenbank den Inhalt der Variable *hotelOrder*. Aus dem Inhalt der Variable wird mit dem Namen des *Parts* der Datenstring des *Parts* geholt. Der Datenstring wird in ein JDOM Dokument umgewandelt. Auf diesem wird die *Query* ausgführt. So findet man die Position im JDOM Dokument, wo das Ergebnisobjekt eingefügt wird. Hierzu wird das Ergebnisobjekt in ein Dokumentfragment umgewandelt und eingefügt. Das JDOM Dokument wird in einen String umgewandelt und der String wird in die Hashtable der Variable *hotelOrder* gesetzt, dabei ist der Name des *Parts hotel* der Schlüssel innerhalb der Hashtable.	Dann wird die Variable *hotelOrder* geholt und ihr wird der String und die Zugriffs *ID* (zu000004) übergeben. Innerhalb der Variable wird mit Hilfe der *ID* die *Query* und der Teilstring, auf die sich die *Query* bezieht, geholt. Aus der *Query* wird eine Suchstruktur gebaut und dem Parser mit den Datenstrings übergeben. Der Parser liefert das Gesuchte als String zurück. Mit Hilfe der Stringoperation wird der alte String durch den übergebenen String ersetzt.

Tabelle 3: Ablauf des ersten Assignments

Jeweils ein Navigator erhält eine Nachricht mit einer der beiden *Invoke* Aktivitäten. Die Nachrichten können gleichzeitig eintreffen. Es wird aus Gründen der Übersicht zuerst der Ablauf der *Invoke* Aktivität *FlyReservation* beschrieben und danach die *Invoke* Aktivität *HotelReservation*. Der Navigator bekommt die *ID* der *Invoke* Aktivität *FlyReservation* in einer Nachricht, holt sich die Aktivität und führt diese aus. Die *Invoke* Aktivität *FlyResveration* holt sich die Variable *flyOder* und schickt eine *Invoke* Nachricht mit der Variable an den *Service Provider*. Die Antwort vom Client kommt über den *Service Provider* als Nachricht zurück.
Die *Invoke* Aktivität *FlyResveration* wird erneut ausgeführt. Dabei wird festgestellt, dass die Aktivität auf eine Antwort wartet. Daher wird aus der Nachricht die Rückgabevariable *flyReservation* ausgelesen und in die Datenbank geschrieben. Die Aktivität beendet sich dann selbst. Die *Invoke* Aktivität *hotelReservation* wird analog ausgeführt. Die Variable wird ausgelesen, indem jeder Part aus der Hashtable geholt und dem Nachrichtenersteller hinzugefügt wird.
Die beiden *Invoke* Aktivitäten beenden sich abschließend selbst und schicken eine Nachricht an den Navigator. Erst wenn beide fertig sind, wird die nächste *Assignment* Aktivität vom Navigator ausgeführt. Der Navigator führt die *Assignment* Aktivität aus. Die Operation erfolgt anlog zur ersten *Assignment* Aktivität. Die zweite *Assignment* Aktivität wird ausgeführt. Der Navigator erhält eine Nachricht mit der *Replay* Aktivität

und führt sie aus. Die *Replay* Aktivität speichert die Variable ab und erstellt eine Nachricht, die dem *Service Provider* übergeben wird. Die *Parts* der Variable werden aus der Hashtable ausgelesen. Danach hört die Aktitivät auf und der Prozess wird beendet.

3.3. Bewertung beider Abläufe

Wie am Beispiel deutlich wird, ist die bisherige Lösung nicht optimal und erzeugt eine große Datenmenge, wenn auf einer Variable eine *Query* ausgeführt wird. Selbst wenn man hier optimiert, wird man schnell an Grenzen stoßen. Diese Grenzen können zum einen durch eine Analyse der statischen Eigenschaften eines BPEL4WS Prozesses überwunden werden und zum anderen durch den Einsatz von schellen Parsern, die durch Suchstrukturen gesteuert werden. Die wichtigsten Punkte des Beispiels sind hier nochmals zusammengefasst:

- Die Verwendung von *IDs* ermöglicht ein direktes Holen und Setzen der Variable während der Laufzeit. Die Rückumwandlung des Namens einer Variablen in den Namensraum vom XML fällt weg.

- Die Analyse der Beziehungen zwischen lesenden und schreibenden Zugriffen ermöglicht eine vorzeitige Bereitstellung der Daten, ohne dabei gegen das *Compensation-* und *Fault*-Handler Konzept von BPEL4WS zu verstoßen.

- Die Umwandlung der XPath Ausdrücke in Steueranweisungen zum Parsen der XML-Strings ermöglicht die Benutzung von schnelleren und einfacheren Parsern. So erhält man ein besseres Laufzeitverhalten als wenn man XPath mit Hilfe einer DOM Implementierungen wie JDOM auswertet.

4. Lösung

Die Beschreibung der Lösung erfolgt in zwei Schritten. Zuerst wird die Lösung spezifiziert, dabei wird auch der Weg zur Lösung dargelegt und dann wird auf Klassen- und Methodenebene die konkrete Lösung aufgezeigt. Die hier vorgestellte Lösung schränkt sich selbst ein, indem sie das Datenmodell von SWoM übernimmt. Somit verhält sich die Lösung genauso wie die ursprüngliche Implementierung bei einer *Assignment* Aktivität, wenn es keinen Lokationspfad in der Kopieranweisung gibt. Man könnte auch auf die Hashtable verzichten und auch jeden *Part* parsen. Dafür wäre ein großer Umbau notwendig, der den Rahmen der Diplomarbeit sprengen würde und keinen deutlichen Informationsgewinn erbringen würde.

4.1. Spezifikation

Wie im Kapitel zwei aufgezeigt, gibt es unterschiedliche Ansätze um die Verarbeitung von XML zu verbessern.

- Die Verwendung von *IDs* ermöglicht ein direktes Holen und Setzen einer Variable während der Laufzeit. Die Rückumwandlung des Namens einer Variable in den Namensraum von XML fällt weg. Der Name einer Variable wird durch die *IDs* ersetzt. Dies passiert bereits nach dem Einlesen der BPEL4WS Datei und während des Importierens der Konstrukte in die *Buildtime* Datenbank. Es müssen dann alle Zugriffe auf alle Variablen gefunden werden und der Zugriff über den Namen durch Zugriffe über die *IDs* ersetzt werden. Ein besonderes Augenmerk muss auf die XPath Anweisung von *transition Condtionen* von *Links* gelegt werden. Die XPath Anweisung wird geparst, so dass man die Zugriffe über BPEL4WS Funktionen auf die Variablen finden kann.

- Die Analyse der Beziehungen zwischen lesenden und schreibenden Zugriffen ermöglicht eine vorzeitige Bereitstellung der Daten, ohne dabei gegen das *Compensation-* und *Faulthandler* Konzept von BPEL4WS zu verstoßen. Denn zum einen werden die Variablen der beiden Aktivitäten gleich behandelt und zum anderen wird der lesende oder schreibende Zugriff nur vorbereitet, so dass beim Auftreten eines Fehlers kein veränderter Ablauf vorliegt, den man beachten muss. Die Analyse ermöglicht es nach einem schreibenden Zugriff bereits alle abhängigen lesenden Zugriffe vorzubreiten. Dies ist zum Beispiel der Fall, wenn es eine *Transition Condition* gibt, die auf zwei Variablen lesend zugreift und beide Variablen Ausgabenvariablen zweier *Invoke* Aktivitäten sind. So kann dann die Variable der *Invoke* Aktivität, die früher eintrifft schon geparst und für den Zugriff vorbereitet werden. Trifft dann die zweite Nachricht ein wird, dementsprechend auch die nächste Variable vorbereitet. Wenn dann die *Transition Condition* ausgewertet wird, liegen die Daten beider Variablen bereit.

- Die Umwandlung der XPath-Ausdrücke in Steueranweisungen zum Parsen des XML-Strings ermöglicht die Benutzung von schnelleren und einfacheren Parsern.

Hierbei ist zu klären, in welchem Umfang der von der BPEL4WS Spezifikation geforderte Funktionsumfang von XPath unterstützt wird und ob man die XPath Ausdrücke, die man unterstützt, schon beim Einlesen der BPEL4WS Datei erkennen kann.

In der Anfangsphase der Diplomarbeit wurde angenommen, dass zuerst die Datenflussanalyse zu implementieren ist, um die Informationen zum Erstellen der Steueranweisungen zu erhalten. Doch es stellte sich heraus, dass der Teil der Analyse bereits von bestehenden Implementierungen gelöst worden ist. Somit kann man zuerst die einfachen Parser mit den Steueranweisungen implementieren und gleichzeitig kann man die ausschließlichen Nutzung der *IDs* implementieren. Bei der ersten Implementierung wurde darauf geachtet, dass man diese um die Ergebnisse der Datenflussanalyse erweitern kann. Erst in der zweiten Implementierungsphase wurde die Datenflussanalyse eingearbeitet.

4.1.1. XPath Umsetzung

Durch die schrittweise Importierung von BPEL4WS in die SWoM ist auch eine schrittweise Behandlung von XPath möglich. XPath wird in einem BPEL4WS Prozess zum Auffinden und zur Manipulation von Daten verwendet. Um XPath schrittweise zu behandeln, muss man die Verwendung von XPath bei BPEL4WS genauer analysieren. Durch das Konzept von BPEL4WS, dass auf Variablen nur mit XPath zugegriffen wird, kann man alle Zugriffe auf Variablen schon beim Importieren von BPEL4WS in die *Buildtime* Datenbank feststellen. Wenn man das *Assignment* und die BPEL4WS Funktion genauer untersucht, stellt man fest, dass man keinen XPath braucht, denn bei einem Zugriff auf eine gesamte Variable oder auf einen *Part* werden diese komplett kopiert. Bei einem Zugriff auf einen *Part* liegt dieser bereits so vor, dass auch er nur kopiert wird. Man muss also dafür sorgen, dass der Name der Variable durch deren *ID* ersetzt wird.
Eine besondere Behandlung brauchen nur die BPEL4WS XPath *Query*. Daher liegt ein besonderes Augenmerk dieser Diplomarbeit auf der *Query*. Die Sprache, in der eine *Query* geschrieben wird, ist XPath. Aber die BPEL4WS Spezifikation schränkt XPath dermaßen ein, dass eine *Query* nur ein eindeutiger Lokationspfad sein darf. Es darf nur ein Element in dem Dokumentfragment geben, auf den sich der Lokationspfad beziehen darf. Andernfalls muss eine Ausnahme ausgelöst werden. Dadurch wird die Ausführung einer *Query* deutlich erleichtert. Es bleibt noch das Problem, dass einige Sprachbestandteile von XPath einen Rückwärtsbezug haben und dies für einen Stream Parser schwer zu handhaben ist. Jedoch hat man festgestellt, dass bei alleiniger Verwendung formal eindeutiger Lokationspfade bei XPath jeder Rückwärtsbezug eines Lokationspfads in einen Vorwärtsbezug umwandelbar ist. Zudem wurde gezeigt, dass man mit den formalen Lokationspfaden jedes Element eines XML-Dokuments auffinden kann. Die Umwandlung aller Rückwärtsbezüge in Vorwärtsbezüge kann schon beim Importieren von BPEL4WS in die *Buildtime* Datenbank erfolgen. Es gab verschiedene Implementierungen zu vergleichen:

Phantom XML: Eine Lösung für XPath von IBM [RV05], die leider nicht zugänglich war.

VTD-Parser: Eine Lösung von [Zha04] zum Aufbau einer schlanken Suchstruktur aus dem XML-Dokument. Auf diesem Dokument wird dann XPath ausgeführt. Leider war es nicht möglich, die BPEL4WS-Funktionen einzubinden. Ein weiterer negativer Aspekt ist, dass man die Suchstruktur über das Dokument aufbaut, denn in der Regel sind solche Suchstrukturen deutlich größer als jene, die von der XPath Query abgeleitet werden. Daher hat man diesen Ansatz in der Diplomarbeit nicht weiter verfolgt.

XMLTaskforce: Diese Lösung von [GKP05] baut aus XPath einen Suchbaum auf. Beim Aufbau des Suchbaums wird darauf geachtet, dass keine doppelten Strukturen erzeugt werden. Die vorgestellte Lösung ist für ein Linux/Unix System entwickelt worden und es gab keinen Zugriff auf den Sourcecode, so dass ein Einbinden in SWoM kaum möglich gewesen wäre.

Galax: Diese Lösung von [FSW00] ist in der Sprache Objective Caml geschrieben und wurde deswegen nicht weiter untersucht.

Spex: Diese Lösung von [OFB03] wurde in Java 1.5 unter der GPL Lizenz geschrieben. Jedoch war der Sourcecode zugänglich. Da dieses Konzept der Zielsetzung der Diplomarbeit angepasst werden konnte kam diese Lösung zur Anwendung. Sie baut aus einer *Query* eine Suchstruktur auf und umfasst alle von der BPEL4WS Spezifikation geforderten Elemente. Zudem können alle *Queryies* in ausschließlich vorwärtsbezoge Lokationspfade umgewandelt werden. **Spex** hat folgende Komponenten:

- Der Umwandler modifiziert in einer XPath Anweisung alle Rückwärtsbezüge in Vorwärtsbezüge. Die so umgewandelte XPath Anweisung kann wieder als String gespeichert werden.
- Der Übersetzer transformiert eine XPath Anweisung mit Vorwärtsbezug in eine Steuerstruktur für den XPath Parser um. Der Übersetzer kann allerdings nur mit einer Teilmenge von XPath umgehen, die eine Übermenge der von der BPEL4WS Spezifikation gefoderten Konstrukte einer *Query* ist.
- Der Parser durchsucht ein XML Dokument mit Hilfe der Steuerstruktur der XPath *Query*. Dadurch kann man also den XPath Ausdruck automatisch mit einem Stream Parser auswerten.

Bei der *Transition Condition* wird XPath zur Formulierung von booleschen Ausdrücken verwendet. Dabei ist wichtig, dass die Ausdrücke keine Lokationspfade enthalten, die nicht durch eine BPEL4WS Funktion gekapselt sind. So kann man XPath in zwei Teilmengen aufteilen.
Die eine Menge wird durch Spex implementiert und stellt die Lokationspfade dar. Die andere Menge stellt Ausdrücke dar, die ein Ergebnis liefern. Diese können effizient in Java implementiert werden. Beim Einlesen von XPath Ausdrücken aus einer BPEL4WS Datei muss man jedoch parsern, um so die Elemente des Ausdrucks der passenden Menge zuzuordnen. Dadurch lässt sich der Ausdruck auf beide Teilmengen aufteilen. Dies

wird durch eine modifizierte Grammatik von XPath erreicht. Sie enthält keine Elemente eines Lokationspfads und die BPEL4WS Funktionen werden besonders behandelt. Mit der Grammatik wird dann ein LALR-Parser erzeugt. Hierzu wurde der Parsergenerator Sablecc verwendet.

4.1.2. Datenflussanalyse

Bei der Datenflussanalyse sollen aus den statischen Informationen von BPEL4WS die Informationen gewonnen werden, mit denen man das Laufzeitverhalten von SWoM verbessern kann. Eine Möglichkeit ist, dass man festellt, ab wann ein Datenzugriff vorzubereiten ist. Eine weitere Möglichkeit ist, dass man festellen kann, ob überhaupt auf den *Part* einer Variable zugegriffen wird. Wenn nicht, kann man den *Part* löschen. Dazu muss man zuerst einen Kontrollgraph für den BPEL4WS Prozess erzeugen. Die Umwandlung der Aktivitäten erfolgt nach [Hei03]. Nachdem ein Kontrollgraph aufgebaut ist, erfolgt die Datenflussanalyse.

4.2. Entwurf

4.2.1. Erste Implemetierungsphase

Parsen allgemeiner Ausdücke Ein allgemeiner Ausdruck muss zwischen den Operationen und den BPEL4WS-Funktionen aufgeteilt werden. Die Aufteilung erfolgt durch das Parsen des allgemeinen Ausdrucks. Der Parser wurde mit einer modifizierten Grammatik erstellt. Ihr fehlen aber alle Elemente für einen Lokationspfad. Also wird die XPath Grammatik modifiziert. In ihrer ursprünglichen Form sieht sie wie folgt aus:

```
unaryexpr = minus* unionexpr ;
unionexpr =  pathexpr | unionexpr pipe pathexpr;
  ...
filterexpr =  primaryexpr|..
```

Die neue Grammatik hat keine pathexpr, sondern zeigt direkt auf die primaryexpr. Sie ergibt nachfolgendes Bild:

```
unaryexpr = minus* unionexpr;
unionexpr = primaryexpr  | unionexpr pipe primaryexpr;
```

Der Parsergenerator erzeugt mit Hilfe der neuen Grammatik einen Parser. Dieser erzeugt aus einem XPath Ausdruck einen Syntaxbaum. Nun läuft das Programm den Baum mit einem Analysator ab, der beim Durchlauf alle BPEL4WS-Funktionen über das Präfix und den Namen der Funktion findet. Anschließend erzeugt der Analysator aus dem Syntaxbaum eine Liste der Expressionen. Die erste Expression in der Liste ist diejenige, wenn man den XPath Ausdruck von links nach rechts liest. Jede Expression kennt ihren Nachfolger und weiß durch welche Operation beide Expressionen miteinander verküpft sind. Es gibt aber auch Expressionen, die keine Nachfolger haben, zum Beispiel die letzte Expression oder die Expression, die ein Funktionsargument ist.

```
primaryexpr = variablereference
| paren_l expr paren_r
| literal
| number
| functioncall;
```

Wenn die Expression eine Funktion ist, so hat sie eine Liste ihrer Argumente. Wenn eine Expression eine BPEL4WS Datenfunktion ist, wird sie wie ein Zugriff mit Lokationspfad behandelt, wobei ihre Argumente in der Zugriffstabelle gespeichert werden. Die Liste der Expressionen wird in einer Tabelle der *Buildtime* Datenbank gespeichert. Der Aufruf des Parsers, das Analysieren des Syntaxbaums und das Speichern der Liste der Expression sind in einer Methode zusammengefasst. Die Methode wurde dem *BPELImporter* von SWoM hinzugefügt.

Vorbehandlung der Lokationspfade Die Lokationspfade werden mit Hilfe von Spex so umgeschrieben, dass der Lokationspfad nur noch Vorwärtsbezüge besitzt. Die zurückgegebene Knotenliste wird in einen String rückgeschrieben. denn ein String lässt sich so besser in die *Buildtime* Datenbank speichern. Die Frage wie man die Knotenliste direkt in die Datenbank speichert und ausliest wurde ausgeklammert. Für die Vorbehandlung wurde eine eigene Methode geschrieben und dem *BPELImporter* von SWoM hinzugefügt.

Speichern von Zugriffen Bei einem *Assignment* oder bei einem allgemeinen Ausdruck kann es lesende und schreibende Zugriffe geben. Jeder dieser Zugriffe wird in eine Zeile der Zugriffstabelle mit folgenden Parametern gespeichert:

- Es wird eine neue eindeutige *ID* für den Zugriff erzeugt und dient als Primärschlüssel.
- Die *ID* des Konstrukts wird als Fremdschlüssel mitgespeichert, um so beim Laden eines Prozessmodells oder bei der Analyse alle Zugriffe dem zugehörigen Konstrukt zuzuordnen.
- Es wird die *ID* der Variable mitgespeichert, um so die Zugriffe nach den Variablen zu ordnen.
- Wenn der Zugriff ein lesender ist und in Beziehung zu einem schreibenden steht, wird die *ID* des schreibenden Zugriffs beim lesenden gespeichert.
- Wenn der Zugriff einen *Part* hat, wird dieser als String gespeichert.
- Wenn der Zugriff eine *Query* hat, wird diese als String gespeichert.
- Es wird als Zahlenwert gespeichert, ob der Zugriff ein lesender oder schreibender Zugriff ist.

Laden allgemeiner Ausdrücke Wenn ein Konstrukt einen allgemeinen Ausdruck besitzt, wird mit dessen *ID* in der Expressions-Tabelle abgefragt. Das Ergebnis ist eine Liste. Wenn die Liste nicht leer ist, wird für jede Expression ein Glied erzeugt und miteinander verbunden. Wenn ein Glied auf eine Variable oder einen Systemzustand zugreift wird dies zusätzlich in die Liste *load* gespeichert.

Laden von Zugriffen Bei der Erzeugung eines *Assignment*s oder einer Variable wird dabei auch auf die Zugriffstabelle zugegriffen. Der Ablauf unterscheidet sich von der alten Implementierung, da beim Assignment andere Daten als bei der Variable gespeichert werden. Im Gegensatz zur alten Implementierung von SWoM haben die Variablen die nötigen Methoden um XPath ausführen zu können. In der ersten Phase ist diese Änderung noch nicht nötig, aber wenn die Ergebnisse der Datenflussanalyse verwendet werden sollen, muss die Methode zur Behandlung von XPath in der Variable implementiert sein. Deswegen gibt es ein verändertes Laden der Zugriffe.

Beim Assignment werden durch die *ID* des *Assignment*s alle Zugriffe des *Assignment*s aus der *Buildtime* Datenbank abgefragt. Die benötigten Daten sind die *IDs* der Variablen, die *ID* des Zugriffs und die *ID* des Ziels. Wenn es kein Ziel gibt, ist dieser Wert leer. Dann wird ein Objekt erzeugt und zwischengespeichert. Nachdem alle Objekte erzeugt worden sind, werden die Beziehungen zwischen den lesenden und schreibenden Zugriffen hergestellt. Anschließend werden die beiden Objekte in ein *Copy* Objekt gespeichert. Dieses wird wiederum in einer Liste gespeichert, die ein Attribut des *Assignment* Objekts ist.

Bei der Variable werden alle Zugriffe auf eine Variable mit ihrer *ID* abgefragt. Die Liste mit allen Zugriffen auf die Variable wird wie folgt abgearbeitet. Zuerst wird unterschieden, ob es sich um einen lesenden oder einen schreibenden Zugriff handelt. Dabei wird jeweils eine unterschiedliche Methode aufgerufen, um der Variable die Informationen über den Zugriff zu übergeben. Der jeweiligen Methode wird dann der *Part*, die *Query* und die Zugriffs *ID* übergeben. Die Variable erzeugt mit den übergebenen Daten ein Zugriffsobjekt. Die Variable speichert die Zugriffsobjekte in einer Hashtable, dabei dient die Zugriffs *ID* als Schüssel. Wenn die Liste abgearbeitet ist, gilt das Setzen der Daten in die Variable als abgeschlossen.

Auffinden von Elementen Bei einer Variable wird die Methode *getContent(String accessID)* aufgerufen. Anschließend wird in der Hashtable *access* mit der *accessID* das Zugriffsobjekt geholt. Dann wird überprüft, ob das Zugriffsobjekt einen Lokationspfad besitzt. Wenn das der Fall ist, wird geparst. Dafür wird zuerst mit dem *Part* aus dem Zugriffsobjekt der Datenstring aus der Hashtable *Content* geholt. Der Lokationspfad und der Datenstring wird Spex übergeben. Als Resultat erhält man von Spex den passenden Ergebnisstring. Dieser wird dann zurückgegeben. Im Fall, dass das Zugriffsobjekt nur einen *Part* besitzt, wird mit dem *Part* der Datenstring aus der Hashtable geholt und dieser dann zurückgegeben. Im letzten möglichen Fall hat das Zugriffsobjekt weder einen Lokationspfad noch einen *Part*. Dann wird die gesamte Hashtable zurückgegeben.

Einfügen von Elementen Bei einer Variable wird die Methode *setContent(String accessID, Object data)* aufgerufen. Mit der *accessID* wird aus der Hashtable *access* das Zugriffsobjekt geholt, sofern es einen Lokationspfad besitzt, wird mit dem *Part* aus der Hashtable *Content* der dazugehörige Datenstring geholt. Der Datenstring und der Lokationspfad wird Spex übergeben, das dann einen String als Resultat ausgibt. Mit der Stringmethode *getSubString* wird die Position des Resultatstrings bestimmt. Mit der Position und der Länge des Resultatsstrings wird der Datenstring in drei Teile gespalten: Anfang bis Position -1, das ist der Vorstring. Position bis Länge plus der Position, das ist der alte Datenstring. Von Länge plus Position plus 1 bis zum Ende des Datenstrings, das ist der Nachstring.
Das übergebene Objekt *data* wird in einen String umgewandelt und mit dem Vorstring und dem Nachstring verknüpft. Der neue Datenstring wird in der Hashtable *Content* mit dem *Part* als Schlüssel gespeichert.
Im nächsten Fall hat das Zugriffsobjekt nur einen *Part*. Hier wird das übergebene Objekt mit dem Schlüssel des *Parts* in die Hashtable gespeichert.
Im letzten möglichen Fall hat das Zugriffsobjekt weder einen *Part* noch einen Lokationspfad. Damit ist das übergebene Objekt eine gefüllte Hashtable. Diese ersetzt die alte Hashtable *Content*.

Ausführen eines allgemeinen Ausdrucks Wenn ein allgemeiner Ausdruck ausgewertet werden soll, wird zunächst die Liste *load* mit den Zugriffen auf die Daten und die Systeminformation abgearbeitet. Mit der *ID* der Variable wird diese von *Cache-Manager* geholt. Danach wird der Methode *getContent(accessID)* der Variable die Zugriffs *ID* übergeben und man erhält einen Datenstring zurück. Dieser wird in das dazugehörige Glied kopiert. Nachdem die Liste abgearbeitet ist, werden die Glieder des Ausdrucks von links nach rechts ausgewertet.

Anmerkungen Die verwendete Bibliothek Spex wurde verändert. Die gesamte Bibliothek wurde von Hand von Java 1.5 nach Java 1.4 portiert um so zu gewährleisten, dass es keine Probleme mit der SDK von der Websphere gibt. Die Ausgabe wurde so verändert, dass dem gefundenen Element nichts hinzugefügt wird. Der Pull Parser wurde als Standartparser festgelegt und so erweitert, dass auch Dokumentfragmente von Spex geparst werden können. Dies ist deshalb wichtig, weil die *Parts* von Variablen nur als Fragmente vorliegen. Es wurden zwei Bibliotheken erstellt, die eine enthält den modifizierten Spex und die andere die Methoden für die allgemeinen Ausdrücke.

4.3. Zweite Implemetierungsphase

Datenflussanalyse Die Datenflussanalyse wurde in der Klasse *ProcessModelValidator* hinzugefügt, da zu diesem Zeitpunkt des Vorbereitens einer Prozessinstanz alle relevanten Informationen vorliegen. Es wird ein Kontrollgraph erstellt, der Kanten und Knoten besitzt. Die Kanten sind gerichten und werden über Listen in den Knoten realisiert. Da es bei einer Datenflussanalyse auch rückwärtsgerichtete Analysen gibt, wird auch das Inverse der Kante in einer Liste gespeichert. Somit hat jeder Knoten zwei Knotenlisten,

welche die Kanten darstellen. Jeder Knoten enthält die *ID* der Aktivität oder des *Links*, die der Knoten darstellt. Wenn das Element, das den Knoten darstellt, Datenzugriffe hat, werden die *ID*s der Variablen, der *Part* und welcher Art der Zugriff ist als Attribute des Knotens gespeichert.

Aufbau des Kontrollgraphs Aus der *Buildtime* Datenbank wird mit der *ID* des Prozesses die Wurzelaktivität abgefragt. Mit dieser *ID* werden zunächst deren Daten aus der Datenbank abgefragt. Wenn die Wurzelaktivität eine *Flow* Aktivität ist, werden zuerst alle enthaltenen Aktivitäten geladen. Für jede wird dann ein Knoten mit der *ID* der Aktivität erstellt. Mit ihr wird auf der Zugriffstabelle nach Zugriffen der Aktivität gesucht und das Ergebnis im Knoten gespeichert. Anschließend werden die Aktivitäten danach sortiert, ob sie eine *Start* oder *Start-End* Aktivität sind und es werden die Beziehungen zwischen diesen Aktivitäten und der *Flow* Aktiviät als Kanten gespeichert. Diese Knoten werden in die Liste „*nicht fertig*“ gespeichert. Dann sind alle *End* und *Start-End* Aktivitäten zu holen und es werden die Beziehungen zwischen diesen Aktivitäten und der *Flow* Aktivität als Kante gespeichert. Die *ID* der *Flow* Aktivität wird nun in die Hashtable „*abgearbeitet*“ gespeichert. Damit sind die ersten Schritte abgearbeitet. Nun werden folgende Schritte wiederholt bis die Liste „*nicht fertig*“ leer ist.

- Es wird ein Knoten aus der Liste „*nicht fertig*“ entfernt.
- Wenn der Knoten eine Flow Aktivität darstellt, werden alle enthaltenen Aktivitäten wie oben beschrieben behandelt.
- Danach werden mit der *ID* des Knotens all jene *Links* abgefragt, bei denen die Aktivität eine Quelle ist.
- Nun werden die *Links* abgearbeitet. Dabei wird für jeden *Link* ein Knoten erstellt und die Datenzugriffe des Links werden abgefragt.
- Dann wird eine Kante zwischen der Quelle und dem *Link* gezogen. Im weiteren Verlauf wird noch eine Kante zwischen dem *Link* und dem Ziel gezogen.
- Wenn das Ziel des *Link*s noch nicht in der Hashtable „*abgearbeit*“ zu finden ist, wird der Knoten in die Liste „*nicht fertig*“ gespeichert.

Sobald die Liste *nicht fertig* leer ist, soll der Kontrollgraph erstellt werden.

Aufbau der Datenstruktur Man verwendet einen Bitvektor als Datenstruktur für die Datenflussanalyse. Der Bitvektor hat dieselbe Länge wie die Prozessvariable. Die Position innerhalb des Vektors wird über eine Hashtable realisiert. Mit der ID der Variablen erhält man die Position innerhalb des Bitvektors. Somit hat man die Datenstruktur für die Datenflussanalyse erzeugt.

Durchführung der Datenflussanalyse In dem man die Fragestellung der globalen Optimierung auf ein klassisches Datenflussproblem abbildet, beispielsweise das sogenannte *Set Use* Problem, kann man daraus eine Steuerstruktur für die Parser ableiten. Das verwendete Analyseverfahren besteht aus zwei Schritten und wurde aus den Buch [ASU99] entnommen und leicht modifiziert. Dabei werden zunächst Verwendungsdefinitionsketten gebildet. Dann wird der Kontrollgraph mit den Ketten und den Informationen über die tatsächlichen schreibenden Zugriffe für jede Variable modifiziert, um so den letzten schreibenden Zugriff zu ermitteln

1.Schritt In[X] = Aufwärts[X] vereinigt mit(Out[X] minus Zuweisung[X]) Out [B] = die Vereinigung aller Nachfolger von X In[S]

- x ist der Knoten
- Aufwärts ist die Menge der lesenden Zugriffs *IDs* auf eine Variable
- Zuweisung ist die Menge der lesenden Zugriffe, die nicht im gleichen Knoten x eines schreibenden Zugriffs sind.

2. Schritt Der schreibende Zugriff muss der letzte zu erreichende Zugriff für den lesenden Zugriff sein und auf dem Pfad darf kein weiterer schreibender Zugriff existieren. Für jeden lesenden Zugriff muss die Verwendungskette bis zu den schreibenden Zugriffen durchsucht werden, um den Zeitpunkt eindeutig zu ermitteln und ob kein andere schreibende Zugriff in der Verwendungskette liegt.

Beispiel Mit dem Beispiel travelOrder gibt es folgende Aufwärtsmenge und Zuweisungsmenge auf die Variablen:

Knoten	Zuweisung	aufwärts
Recieve Travel Order	travelOrder,all	
Assignment 1	hotelOrder,all flyOrder,all	travelOrder,zu000001 travelOrder,zu000002
InvokeHotelOrder	hotelReservation, all	hotelOrder, zu000012
InvokeFlyOrder	flyReservation,all	flyOrder,zu000014
Assignment2	TravelReservation,all	flyReservation,zu000005 flyReservation, zu000010 hotelReservation,zu000009 hotelReservation,zu000007
Replay TravelConfirmation		travelReservation,zu000016

Tabelle 4: Zuweisungs und Aufwärtsmengen des Beispiels

So erhält man folgendes Ergebnis der Datenflussgleichung:

Variable	letzter schreibender Zugriff	davon abhängige lesende Zugriffe
travelReservation	zu000011	zu000016

Variable	letzter schreibender Zugriff	davon abhängige lesende Zugriffe
hotelReservation	zu000013	zu000009 zu000007
flyReservation	zu000015	zu000005 zu000010
hotelOrder	zu000004	zu000012
flyOrder	zu000002	zu000014
travelOrder	zu000017	zu000001 zu000003

Tabelle 5: letzte schreibende Zugriffe

Das Ergebnis wird dann in die *Buildtime* Datenbank gespeichert.

Verändertes Laden der Zugriffe Die Implementierung beim Laden der Zugriffe zur Laufzeit wird erweitert. Wenn eine Variable erstellt wird, dann wird bei jedem schreibenden Zugriff eine Abfrage an die Tabelle 5 gesendet. Als Resultat erhält man eine Liste. Diese wird dann im Zugriffsobjekt gespeichert.

Verändertes Auffinden von Elementen Die Methode „get content“ der Variable wird aufgerufen. Dabei wird zuerst überprüft, ob die Hashtable die Zugriffs *ID* als Schlüssel enthält. Wenn dies der Fall ist wird der dazugehörige Datenstring zurückgegeben. Ansonsten muss zum Auffinden des Objekts geparst werden.

Verändertes Einfügen von Elementen Wenn ein Objekt mit einer Zugriffs *ID* eingefügt wird, kommt es zum gleichen Verfahren wie beim Einfügen der Elemente der ersten Implementierungsphase. In der neuen Implementierungsphase wird die Liste der abhängigen Zugriffe abgearbeitet. Jeder dieser lesenden Zugriffe wird geparst und das Ergebnis mit der Zugriffs *ID* als Schlüssel in der Hashtable *content* abgespeichert.

5. Test

In dieser Diplomarbeit wurden Tests durchgeführt um deutliche Laufzeitverbesserungen festzustellen. Deshalb muss man die gefundenen Lösungen implementieren und mit den Eckdaten des alten Systems vergleichen. Es sollte zudem das Testsystem vom Studienprojekt für einen Gesamttest verwendet werden, so dass man in der Lage ist, SWoM unter realistischen Bedingungen zu testen. Die Informationen über das Testsystem stehen im folgenden Dokument [BH05]. Vor dem Gesamttest wurden bereits kleinere Komponenten miteinander verglichen.

Funktionstest Der Funktionstest überprüft die Korrektheit von SWoM bzgl. des übergebenen Prozesses. Es ist klar, dass beide Implementierungen das gleiche Ergebnis liefern sollen. Der Test liefert auch Ergebnisse über das Laufzeitverhalten.

5.1. Test während der Implementierung

Der erste Test sollte ein einfaches Assignment simulieren. Die verwendeten Testdaten waren:

```
Lokationspfad eins: /child::hotel/child::hotelQuery
Part eins:
<hotel>
  <hotelQuery>
    <sequence>
        <hotelName>Abudal</hotelName>
        <cityName>NewYork</cityName>
    </sequence>
  </hotelQuery>
 </hotel>
Lokationspfad zwei: /child::hotel/child::hotelQuery
Part zwei:
<hotel><hotelQuery></hotelQuery></hotel>
```

Der erste Lokationspfad holt sich aus dem ersten Part die Daten, die mit dem zweiten Lokationspfad in den zweiten Part eingefügt werden. Zuerst wurde der Test mit Java Development Kit 1.5 ausgeführt. Das Ergebnis war, dass die SWoM Implementierung eine Zeitspanne von 700-900 Millisekunden brauchte, um alle Operationen auszuführen. Die neue Implementierung hingegen brauchte nur eine Zeitspanne von 250-350 Millisekunden für alle Operationen. Es war notwendig, beide Implementierungen unter dem JDK 1.4.1 auszuführen. weil die Websphere Plattform nur diese JDK 1.4.1 unterstützt. Nachdem beide Implementierungen unter JDK 1.4.1 ausgeführt wurden, veränderte sich bei der alten Implementierung unter der alten Sprachversion die untere Schranke auf 500 Millisekunden. Leider jedoch veränderte sich die obere Schranke von 900 Millisekunden nicht. Auch die neue Implementierung wurde schneller, aber nur um 100 Millisekunden.
Die Testergebnisse der beiden Implementierungen wichen voneinander ab. Bei der neuen

Implementierung entsprach das Testergebnis dem Erwartungswert. Bei der alten Implementierung wurde noch ein Wurzelelement hinzugefügt, das man noch entfernen muss. Das Verhalten der alten Implementierung wäre korrekt, wenn mit vollständigen XML Dokumenten gearbeitet würde. Doch es wird mit XML Fragmenten gearbeitet.

Optimierung mit den Ergebnissen der Datenflussanalyse Es wurde bereits während der Implementierung der Fragestellung nachgegangen, wie durch ein frühzeitiges bereitstellen von Komponenten zum Parsen das Laufzeitverhalten verbessert werden könnte. Die Ergebnisse der globalen Optimierung wurden verwendet, um die lokale Optimierung zu verbessern. Um dies Nachzuweisen wurden folgende Tests durchgeführt:

```
Lokationspfad /child::hotel/child::hotelQuery
Part
<hotel>
  <hotelQuery>
    <sequence>
        <hotelName>Abudal</hotelName>
        <cityName>NewYork</cityName>
    </sequence>
  </hotelQuery>
 </hotel>
```

Zuerst wurde Untersucht, wie sich das Erstellen der XPath Steueranweisung beim Einlesen des Prozesses auf die lokale Optimierung auswirkt. Es wurde festgestellt, dass wenn man die benötigten Schritte für das Erstellen der XPath Anweisungen herausnimmt und so bereistellt, wie wenn man die globale Optimierung verwenden würde, so ist die lokale Optimierung im Durchschnitt um 60 Millisekunden schneller.
Desweiteren wurde untersucht, ob sich das Laufzeitverhalten der neuen Implementierung durch das Verwenden eines Pools von Parsern verbessern würde. Die benötigte Anzahl der Parser im Pool kann über die Datenflussanalyse abgeleitet werden. Die so abgeänderte Implementierung war im Test durchschnittlich 40 Millisekunden schneller.
Die beiden Ergebnisse zeigen, dass man ein verbessertes Laufzeitverhalten durch die Wahl des richtigen Zeitpunkts zum Vorbereiten des Parsens erreichen kann. Zudem führten die Ergebnisse zu der Frage, wie das Verhalten beider Implementierungen bei größeren Datenmengen ausfällt.

Test mit größeren Datenmengen In einem weiteren Test wurden mehrere größere XML-Dateien von 2.5 Kb bis 25.5 Kb verwendet. Die XPath Anweisungen waren Lokationspfade, die bis zu fünf Schritte umfassten, um so Rückschlüsse auf das Verhalten bei größeren Datenmengen zu erhalten. Das Testresultat ergab, dass beide Implementierungen im Durchschnitt länger brauchten um die Anweisungen zu verarbeiten. Sogar bei der 25.5 Kb großen XML Datei wurde die neue Implementierung nur um 100 Millisekunden langsamer. So ist der Einfluss der Größe der Datei auf das Laufzeitverhalten im Vergleich zum Aufwand der Parservorbereitung noch relativ klein.
Bei der alten Implementierung war auffällig, dass sich die Spanne zwischen der unteren

und oberen Schranke deutlich um 800 Millisekunden erweitert hat, so dass sich die untere Schranke nur um den gleichen Wert veränderte wie bei der neuen Implementierung, wohingegen der Wert der oberen Schranke deutlich zugenommen hat.

Anmerkungen zu den Tests Während der Durchführung der Tests sind einige interessante Beobachtungen aufgetreten. Beim ersten Testaufbau wurde der Fehler gemacht, dass die vom Studienprojekt verwendete Bibliothek xml-api.jar nicht vorhanden war. Daher verwendete der Compiler für die alte Implementierung die Klassen der Sun Runtime Bibliothek. So verbesserte sich das Laufzeitverhalten der alten Implementierung erheblich und erreichte das Laufzeitverhalten der neuen Implementierung. Nur mit allen Optimierungen war die neue Implementierung deutlich schneller. Die alte Bibliothek hat weitere Funktionen, die für den Service Provider benötigt werden, die die Sun Runtime nicht zur Verfügung stellt.
Bei allen drei Tests wurden XML Fragmente verwendet. Im letzten Test wurden die Testdaten aus externen Dateien eingelesen, wobei sich die alte Implementierung fehlerhaft verhielt.
Wenn beim letzten Test echte XML Dokumente verwendet wurden, verhielt sich die neue Implementierung erwartungsgemä"s fehlerhaft. Die Ursache war die Veränderung des Parsertreibers um sicher Fragmente zu parsen. Wenn nun in einer zukünftigen Erweiterung wieder XML Dokumente geparst werden sollen, muss der Treiber angepasst werden, so dass man am besten zwei Treiber hat. Den einen um Fragmente und den anderen um Dokumente zu parsen.

5.2. Testumgebung der Funktionstests

Auf einem Rechner wird jeweils die neue und alte Implementierung der SWoM installiert und auf einem zweiten wird der Testclient installiert. Beide Rechner sind über ein Netzwerk miteinander verbunden. Die Datenbank ist eine DB2 8.02 Express und als Server wird der Websphere Applikationsserver 6.0.2.7 verwendet. Beides sind Produkte von IBM.

5.3. Ergebnisse des Funktionstests

Leider war es im Zeitrahmen der Diplomarbeit nicht möglich, die fertige Implementierung mit dem vom Studienprojekt gelieferten Testrahmen zu testen, da es noch erhebliche Probleme mit dem Testrahmen von SWoM gegeben hat. Ein realistischer Test war daher nicht mehr möglich.

6. Ergebnis

Die Diplomarbeit startete in der Endphase des Studienprojekts, das SWoM erstellt hat. Die Diplomarbeit hatte so einen konkreten Rahmen. Dieser Rahmen hatte einen negativen Aspekt, denn der wichtige Vergleichsstest zwischen der neuen und der alten Implementierung war in dem strikten Zeitrahmen der Diplomarbeit mit dem fehlerhafen Testrahmen der SWoM nicht möglich.
Die wichtigste Anforderung der Diplomarbeit war es, die Verarbeitung von XML zu optimieren. Die hierfür gewählten Maßnahmen ermöglichten den von BPEL4WS geforderten Funktionsumfang von XPath zu unterstützen. Zur Implementierung der lokalen Optimierung wurde ein Streamparser verwendet. Diese Implementierung führt zu einer lokalen Optimierung der Auswertung von XPath, leider konnte wie bereits genannt kein Gesamttest durchgeführt werden, um auch die globale Optimierung direkt nachzuweisen. Aber aus den Testergebnissen und den Eigenschaften eines Streamparsers, mit dessen Hilfe man XPath schon vorab vorbereiten kann, kann man schlussfolgern, dass auch der Ansatz der globalen Optimierung einen positiven Einfluss auf die Verarbeitung von XML hat. Denn wenn bereits geparst werden kann sobald das System Ressourcen frei hat, kann man die Manipulationen der Daten schneller ausführen. Als ein Nebenprodukt der statischen Analyse kann auch der Prozess besser verifiziert werden und Fehler werden auch vor einem Test entdeckt.

6.1. Analyse und Bewertung der Testergebnisse

Es war positiv, dass schon der erste Test einen deutlichen Unterschied beim Laufzeitverhalten zwischen alter und neuer Implementierung aufzeigte. Der minimale Abstand im Laufzeitverhalten war ca. 200 Millisekunden groß, der schon beim Parsen einer sehr kleinen XML Datei und mit einer kleinen XPath Anweisung auftrat. Der Test zeigt, dass die neue Implementierung eine lokale Optimierung von XPath ist. Absolut unerwartet war, dass der Abstand zwischen der oberen und unteren Schranke des Laufzeitenverhaltens der alten Implementierung 400 Millisekunden betrug. Hier zeigte sich die neue Implementierung deutlich stabiler und somit deutlich besser.
Ein negativer Aspekt der neuen Implementierung ist der große Faktor der Laufzeitkosten um das Parsen vorzubereiten. Diese Tatsache war so nicht erwartet, da hier die in der Diplomarbeit verwendete Literatur zu anderen Vermutungen Anlaß gab. Auch die Tests mit größeren Dateien zeigten den hohen Aufwand der Parservorbereitung, so dass der Ansatz der globalen Optimierung sinnvoll ist. Daher kann man das Laufzeitverhalten hier deutlich verbessern.
Bei den Tests zum Nachweisen der globalen Optimierung wurden deutliche Verbesserungen des Laufzeitverhaltens festgestellt. Es wurde aber nur eine Datenflussanalyse für die globale Optimierung verwendet. Es ist gut möglich, dass es noch weitere Datenflussanalysen für die globale Optimierung gibt.
Ein positver Aspekt der neuen Implementierung ist, dass die Laufzeitkosten bezüglich der Größe der Daten kaum gewachsen sind. Noch ein weiterer positiver Aspekt der neuen Implementierung ist die minimale Abweichung der Laufzeitkosten beim Durchführen des

gleichen Tests.

6.2. Ausblick

Die Optimierung von der XML-Verarbeitung ist ein weites Feld und bei weitem noch nicht vollständig erforscht. Es ist interessant, dass mit einem Teil der Techniken zur Optimierung auch eine Verifikation des Prozesses möglich ist.

Offene Punkte bei der Lösung Im Rahmen dieser Diplomarbeit wurde nicht untersucht, welchen Einfluss die Death Path Elimination DPE auf die Datenflussanalyse hat. Um DPE zu berücksichtigen, muss man alle im Kontrollgraph befindlichen Aktivitäten, die elminiert werden können, auffinden. Für jede so gefundene Aktivität müssen zusätzliche Kanten eingefügt werden. Jeder Vorgänger der Aktivitiät und jeder Nachfolger der Aktivität müssen miteinander verbunden werden, um die Elminierungen durch DPE zu simulieren. Das kann dazu führen, dass man nur wenig Informationen gewinnen kann, um global zu optimieren. Das hat keinen Einfluss auf die lokale Optimierung.
Mit der Datenflussanalyse kann schon vor der Ausführung des Prozesses ermittelt werden, ob es die Möglichkeit gibt, mit einem Lokationspfad auf eine leere Variable lesend oder schreibend zuzugreifen. Dass mit einem Lokationspfad auf eine leere Variable zugegriffen wird, kann dadurch verhindert werden, dass die Variable über ein Literal gesetzt wird. Der Ersteller des Prozesses sollte auf einen möglichen Fehler hingewiesen werden. Es kann sein, dass die Datenflussanalyse eine Ausführung des Prozesses erkennt, die so aber niemals in der Realität ausgeführt wird. Deswegen sollte nur eine Warnung und nicht ein Fehler ausgegeben werden.

Alternative Implementierungsansätze Es wurde nur die lokale Optimierung bei der Transaktionsbedingung und dem Kopieren innerhalb einer Assignment Aktivität untersucht. SWoM verwendet DOM auch zum Erstellen und zum Parsen von Nachrichten. Auch hier könnte man einen Streamparser und die Datenflussanalyse einsetzen. So könnte man dann auch die Datenhaltung innerhalb der Variable verändern, wenn nur auf die gesamte Variable zugegriffen wird, braucht man keine Hastable, sondern ein String würde reichen. Wenn nicht auf die gesamte Variable sondern nur auf einige Teile zugegriffen wird können nur diese Teile aus der Nachricht ausgelesen werden und in der Hashtable gespeichert werden.

Zukünftige Entwicklung von SWoM Es ist gerade ein zweites Studienprojekt gestartet worden, dass die bestehende SWoM erweitern soll, um so BPEL4WS vollständig zu unterstützen. In diesem Rahmen können die Ergebnisse der Diplomarbeit in die Hauptentwicklungslinie von SWoM eingearbeitet werden. Hierbei wird die Fragestellung bezüglich der alternativen Implementierung des *Service Providers* eine wichtige Rolle spielen. Zudem sollte ein Funktionstest der Neuimplementierung durchgeführt werden und anschließend auch mit den Ergebnissen der Diplomarbeit von [Fel06] verglichen werden um die beste Lösung für SWoM zu finden.

A. Codetabelle

In der folgende Tabelle ist aufgezeigt, welche Klasse hinzugekommen, welche Klasse verändert und welche Klasse überholt worden, damit man die Änderung im Code nachvoll ziehen kann.

Paketpfad und Name der Klasse	Status der Klasse	Kommentar
xmloperation.jar	neu	Die Datei wird als externe Bibliothek eingebunden und enthält alle Klassen zum Parsen allgemeiner Ausdrücke.
xmlstream.jar	neu	Die Datei wird als externe Bibliothek eingebunden und enthält alle Klassen zum Parsen von XML-Fragmenten.
SharedEJB / ejbModule / iaas / swom / shared / buildtime / importer / BPELImporter	verändert	Zwei Methoden wurden hinzugefügt um die XPath Anweisungen zu verändern. Desweiteren wurden bei allen Aktivitäten die Zugriffe auf Variablen gespeichert in die Tabelle der Zugriffe gespeichert. Zuletzt wird beim Einlesen der Links die Transaktionsbedingung behandelt und gespeichert.
SharedEJB / ejbModule / iaas / swom / shared / buildtime / importer /DatabaseFunctions.java	verändert	Zwei Methoden wurden hinzugefügt, um auf die neuen Tabellen schreiben zu können.
SharedEJB / ejbModule / iaas / swom / shared / Variable.java	verändert	Die gesamten Methoden zum Arbeiten mit Zugriffs *ID* hinzugefügt.
SharedEJB / ejbModule / iaas / swom / shared/ Access.java	neu	Die Klasse besitzt alle Informationen über einen Zugriff.
ProcessExecution / ejbModule / iaas / swom / pe / datam / TransformationB2E.java	verändert	Das gesamte Erstellen einer Prozessinstanz wurde dem neuen Konzept angepasst.
ProcessExecution / ejbModule / iaas / swom / pe / nav / NavigatorBean.java	verändert	Das Speichern und das Laden einer Variable wurde verändert.
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / Activity.java	verändert	Die Auswertung der Join Bedingung wurde verändert.
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / basic / AssignActivity	verändert	Das Laden der Variable wurde angepasst.

Paketpfad und Name der Klasse	Status der Klasse	Kommentar
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / basic / InvokeActivity	verändert	Das Laden der Variable wurde angepasst.
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / basic / ReceiveActivity	verändert	Das Laden der Variable wurde angepasst.
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / basic / ReplayActivity	verändert	Das Laden der Variable wurde angepasst.
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / basic / assign /VariableFS	überholt	Die Klasse wird nicht mehr verwendet.
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / basic / assign /VariablePartFS	überholt	Die Klasse wird nicht mehr verwendet.
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / basic / assign /VariablePartQueryFS	überholt	Die Klasse wird nicht mehr verwendet.
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / basic / assign /VariableTS	überholt	Die Klasse wird nicht mehr verwendet.
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / basic / assign /VariablePartTS	überholt	Die Klasse wird nicht mehr verwendet.
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / basic / assign /VariablePartQueryTS	überholt	Die Klasse wird nicht mehr verwendet.
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / basic / assign /VariableCopyFrom	neu	Die Klasse ersetzt alle VariableFS Klassen.
ProcessExecution / ejbModule / iaas / swom / pe / pmodel / basic / assign /VariableCopyTo	neu	Die Klasse ersetzt alle TS Klassen.
ProcessExecution / ejbModule / iaas / swom / pe / xpath /StreamEvaluator.java	neu	Die Klasse führt die Auswertung von Bedingungen durch.
AIP / ejbModule / iaas / swom / admin / aip / ProcessModelValidator.java	verändert	Der Klasse wurden die Methoden zur Datenflussanalyse hinzugefügt.

Tabelle 6: Die Codetabelle

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
BPEL Process Definition
Edited using ActiveBPEL(tm) Designer Version 2.1.0 (http://www.
    active-endpoints.com)
-->
<process xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-
    process/" xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/
    business-process/" xmlns:ns1="http://www.example.org/
    travelOrderWSDLFile/" xmlns:ns2="http://www.example.org/
    InvokeHotel/" xmlns:ns3="http://www.example.org/NewWSDLFile/
    " xmlns:ns4="http://www.example.org/
    TravelReservationResponse/" xmlns:ns5="http://www.example.
    org/Airline/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    expressionLanguage="http://www.w3.org/TR/1999/REC-xpath
    -19991116" name="travelOrder" suppressJoinFailure="yes"
    targetNamespace="http://127.0.0.1/travelOrder">
    <partnerLinks>
        <partnerLink myRole="start" name="travel" partnerLinkType=
            "ns3:travel"/>
        <partnerLink name="HotelReservation" partnerLinkType="
            ns2:HotelReservation" partnerRole="reservate"/>
        <partnerLink name="airline" partnerLinkType="ns5:airline"
            partnerRole="reservation"/>
    </partnerLinks>
    <variables>
        <variable messageType="ns3:travelOrderRequest" name="
            travelOrder"/>
        <variable messageType="ns2:NewOperationRequest" name="
            HotelOrder"/>
        <variable messageType="ns2:NewOperationResponse" name="
            HotelDescription"/>
        <variable messageType="ns3:travelOrderResponse" name="
            travelDescription"/>
        <variable messageType="ns5:NewOperationRequest" name="
            flyOrder"/>
        <variable messageType="ns5:NewOperationResponse" name="
            flyDescription"/>
    </variables>
    <correlationSets>
        <correlationSet name="CS1" properties="ns3:piid"/>
    </correlationSets>
    <flow>
        <links>
```

```
    <link name="L6"/>
    <link name="L2"/>
    <link name="L3"/>
    <link name="L1"/>

    <link name="L5"/>
    <link name="L4"/>
</links>

<assign name="Assignment2">
    <target linkName="L5"/>
    <target linkName="L4"/>
    <source linkName="L6"/>
    <copy>
        <from part="HotelInvokOutput" variable="
            HotelDescription"/>
        <to part="Hotel" variable="travelDescription"/>
    </copy>
    <copy>
        <from part="Flydescription" variable="flyDescription
            "/>
        <to part="Fly" variable="travelDescription"/>
    </copy>
</assign>
<assign name="firstAssignment">
    <target linkName="L1"/>
    <source linkName="L2"/>
    <source linkName="L3"/>
    <copy>
        <from part="travel" query="/travel/cityName"
            variable="travelOrder"/>
        <to part="HotelInvokeInput" query="/travel/cityName"
             variable="HotelOrder"/>
    </copy>
    <copy>
        <from part="travel" query="/travel/hotelName"
            variable="travelOrder"/>
        <to part="HotelInvokeInput" query="/travel/hotelName
            " variable="HotelOrder"/>
    </copy>
    <copy>
        <from part="travel" query="/travel/flyID" variable="
            travelOrder"/>
```

```
          <to part="Flyreservation" query="/travel/flyID"
              variable="flyOrder"/>
        </copy>
        <copy>
          <from part="travel" query="/travel/airlineName"
              variable="travelOrder"/>
          <to part="Flyreservation" query="/fly/airlineName"
              variable="flyOrder"/>
        </copy>
      </assign>
      <receive createInstance="yes" name="travelOrderReceive"
          operation="travelOrder" partnerLink="travel" portType=
          "ns3:travelOrderWSDL" variable="travelOrder">
          <source linkName="L1"/>
      </receive>
      <reply name="travelOrderReplay" operation="travelOrder"
          partnerLink="travel" portType="ns3:travelOrderWSDL"
          variable="travelDescription">
          <target linkName="L6"/>

      </reply>
      <invoke inputVariable="HotelOrder" name="hotelReservation"
           operation="hotelReservationOperation" outputVariable=
          "HotelDescription" partnerLink="HotelReservation"
          portType="ns2:InvokeHotel">
          <target linkName="L2"/>
          <source linkName="L5"/>
      </invoke>
      <invoke inputVariable="flyOrder" name="flyReservation"
          operation="flyReservationOperation" outputVariable="
          flyDescription" partnerLink="airline" portType="
          ns5:Airline">
          <target linkName="L3"/>
          <source linkName="L4"/>
      </invoke>
   </flow>
</process>
```

Abbildungsverzeichnis

1. Schaubild von SWoM 13
2. Schaubild einer gefüllten Hashtable 16
3. Schaubild der Verarbeitung einer XPath 18
4. Schaubild der neuen XPath 20
5. Schaubild des Reisebuchungs Prozesses 23

Tabellenverzeichnis

1. Zugriffstabelle 25
2. letzte schreibende Zugriffe 25
3. Ablauf des ersten Assignments 29
4. Zuweisungs und Aufwärtsmengen des Beispiels 39
5. letzte schreibende Zugriffe 40
6. Die Codetabelle 47

Literatur

[ASU99] Aho, Alfred ; Sethi, Ravi ; Ullman, Jeffrey: *Compilerbau, 2 Bd., Bd.2.* Oldenbourg, 1999. – ISBN 3486252941

[BH05] Birgitt Heubach, SWoM-Team: *SWoM AdminstrationGuide-Test Client.* Mai 2005. – http://www.iaas.uni-stuttgart.de/lehre/vorlesung/2005_ss/stupro/index.p%hp

[Bir01] Birbeck, Mark: *Professional XML.* Zweite Edition. pub-WROX, 2001 (Programmer to programmer). – ISBN 1–86100–505–9 (paperback)

[BPSM98] Bray, Tim ; Paoli, Jean ; Sperberg-McQueen, C. M.: Extensible Markup Language (XML) 1.0 - W3C Recommendation 10-February-1998 / W3C. Version: 1998. citeseer.nj.nec.com/bray98extensible.html. 1998 (REC-xml-19980210). – Forschungsbericht. – http://www.w3.org/TR/2000/REC-xml-20001006

[CB05] Christoffer Bromberg, SWoM-Team: *SWoM Specification.* Juli 2005. – http://www.iaas.uni-stuttgart.de/lehre/vorlesung/2005_ss/stupro/index.p%hp

[CCMW01] Christensen, Erik ; Curbera, Francisco ; Meredith, Greg ; Weerawarana, Sanjiva: *Web Services Description Language (WSDL) 1.1.* World Wide Web Consortium, Note NOTE-wsdl-20010315, März 2001

[CD99] Clark, James ; DeRose, Steve: *XML Path Language (XPath) Version 1.0, W3C Recommendation.* November 1999. – http://www.w3c.org/TR/xpath

[Fel06] Felipe, Erias M.: *Compilation of BPEL conditions and assign activities into Java code.* Stuttgart, Stuttgart, Univ., Studiengang Informatik, Diplomarbeit, 2006. – Diplomarbeit Nr. 2455

[FSW00] Fernandez, Mary ; Simeon, Jerome ; Wadler, Philip: An Algebra for XML Query. In: *Lecture Notes in Computer Science* 1974 (2000), 11–?? citeseer.ist.psu.edu/fernandez00algebra.html

[GKP05] Gottlob, Georg ; Koch, Christoph ; Pichler, Reinhard: Efficient algorithms for processing XPath queries. In: *ACM Trans. Database Syst.* 30 (2005), Nr. 2, 444–491. http://doi.acm.org/10.1145/1071610.1071614. – ISSN 0362–5915

[Hei03] Heidinger, Thomas: *Statische Analyse von BPEL4WS-Prozessmodellen*, Humboldt-Universität zu Berlin, Studienarbeit, Dezember 2003. http://www2.informatik.hu-berlin.de/top/download/publications/heidinger%03.pdf

[Hun01] Hunter, Jason: *JDOM 1.0.* Java Specification Request (JSR) 102, Februar 2001

[IDB+03] Ibm, Francisco C. ; Dholakia, Hitesh ; Bea, Yaron G. ; Microsoft, Johannes K. ; Ibm, Frank L. ; Sap, Kevin L. ; Ibm, Dieter R. ; Smith, Doug ; Systems, Siebel ; Thatte, Satish ; Sap, Ivana T. ; Ibm, Sanjiva W.: *Business Process Execution Language for Web Services.* http://citeseer.ist.psu.edu/669609.html;ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf. Version: Mai 13 2003

[KKL+04] Kloppmann, Matthias ; König, Dieter ; Leymann, Frank ; Pfau, Gerhard ; Roller, Dieter: Business process choreography in WebSphere: Combining the power of BPEL and J2EE. In: *IBM Systems Journal* 43 (2004), Nr. 2, 270–296. http://dx.doi.org/10.1147/sj.432.0270

[LR99] Leymann, F. ; Roller, D.: *Production Workflow: Concepts and Techniques.* Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999

[Meg00] Megginson, David: *Simple API for XML (SAX).* 2000. – http://www.saxproject.org/

[MS06] Mirko Sonntag, SWoM-Team: *SWoM Design.* Januar 2006. – http://www.iaas.uni-stuttgart.de/lehre/vorlesung/2005_ss/stupro/index.p%hp

[OFB03] Olteanu, D. ; Furche, T. ; Bry, F.: *Evaluating complex queries against XML streams with polynomial combined complexity.* citeseer.ist.psu.edu/olteanu03evaluating.html. Version: 2003

[RV05] Rose, Kristoffer H. ; Villard, Lionel: *Phantom XML.* 2005. – http://www.idealliance.org/proceedings/xml05/ship/80/paper.PDF/

[Slo02] Slominski, Aleksander: *Design of a Pull and Push Parser System for Streaming.* Februar 07 2002. – http://citeseer.ist.psu.edu/614792.html

[TBMM01] Thompson, H. ; Beech, D. ; Maloney, M. ; Mendelsohn, N.: XML Schema 1.1 / W3C. 2001 (REC-xmlschema-1-20041028). – Forschungsbericht. – http://www.w3.org/TR/xmlschema-0/

[WAB+98] Wood, Lauren ; Apparao, Vidur ; Byrne, Steve ; Champion, Mike ; Isaacs, Scott ; Jacobs, Ian ; Hors, Arnaud L. ; Nicol, Gavin ; Robie, Jonathan ; Sutor, Robert ; Wilson, Chris ; Wood, Lauren: Document Object Model (DOM) Level 1 Specification / W3C Recommendation. 1998 (REC-DOM-Level-1-19981001). – Forschungsbericht. – http://www.w3.org/TR/REC-DOM-Level-1/

[Zha04] ZHANG, Jimmy: SOAP Processing: A Non-extractive Approach. In: *Web Services, European Conference, ECOWS 2004, Erfurt, Germany, September 27-30, 2004, Proceedings* Bd. 3250, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–23202–8, 152–167

Erklärung

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

(Dominique Xavier Kiefner)