Seminar XML –Technologien

WSDL & WSFL

Danny Tschirner
(Matrikelnr.: 357 861 3)
tschirne@inf.fu-berlin.de

Franziska Liebsch
(Matrikelnr.: 362 328 1)
fliebsch@inf.fu-berlin.de

Freie Universität Berlin
Institut für Informatik

Inhaltsverzeichnis:

1. Einführung

”A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.”[1]
Dies ist die offizielle Definition des W3C für einen Web Service. Übersetzt könnte diese Definition so beschrieben werden: „Ein Web Service ist ein webbasiertes Programm mittels XML ausgedrückt, welches für den Client via Internet erreichbar gemacht wird.“ Da auch schon heute eine Vielzahl an Diensten existieren die man als Web Services bezeichnen könnte (z.B. das abfragen von Aktienkursen oder Suchmaschinen) und der Boom solcher Dienste einhergeht mit der Entwicklung des Internets in den 90er Jahren, wird auch in Zukunft davon auszugehen sein, dass Web Services mehr und mehr an Bedeutung gewinnen werden. Bei den meisten der momentanen Dienste existiert aber häufig nur eine Schnittstelle, die es dem Benutzer gestattet die zu verarbeitenden Daten einzugeben (z.B. mittels eines HTML-Formulars), es besteht also nicht die Möglichkeit mit anderen Programmen auf einen Web Service zuzugreifen. Aber gerade das wird in Zukunft von hoher Bedeutung sein, verschiedene Programme verschiedener Unternehmen sollen unabhängig vom Betriebssystem (bzw. der Programmiersprache) untereinander kommunizieren (bzw. interagieren) können. Durch diese vermehrte Nutzung von Web Services und dem damit verbundenen Nachrichtenaustausch, ist es eine Notwendigkeit diese Kommunikation in einer standardisierten (bzw. strukturierten) Form zu beschreiben.
Ein Standard zum Beschreiben von Web Services ist WSDL, dessen Struktur wir als erstes im Laufe dieser Ausarbeitung genauer beschreiben werden. WSDL ist ein XML (1.0) Format und ist die Abkürzung für Web Service Description Language, es ist aktuell in der Version 1.2 verfügbar. WSDL wurde gemeinsam von IBM und Microsoft entwickelt und erfreut sich daher großer Unterstützung. Auf der anderen Seite führt dies wiederum zu einer Vielzahl von Problemen, auf die wir im Laufe dieser Ausarbeitung genauer eingehen werden. Kurz gesagt enthält ein WSDL Dokument gerade die Informationen die ein „Client“ benötigt um mit dem Web Service zu interagieren. Es werden z.B. die SOAP Messages beschrieben und die Art und Weise wie mit ihnen umzugehen ist. WSDL beschreibt also das Interface des Web Service und bildet somit die Schnittstelle zu anderen Applikationen, z.B. welche Funktion muss mit welchen Parametern aufgerufen werden (Request) und wie sieht die Antwort aus (Response).
Um nun auch den internen Ablauf eines Web Services beschreiben zu können, werden wir als zweites auf WSFL eingehen. WSFL ist die Abkürzung für Web Service Flow Language. Sie wurde von IBM entwickelt, existiert seit Mai 2001 und ist aktuell in der Version 1.0 verfügbar.
WSFL ist eine XML – Sprache um die Zusammensetzung von Web Services zu beschreiben, das heißt die interne und externe Struktur eines Web Services. Hierbei handelt es sich meist um die Beschreibung eines Geschäftsprozesses der sich in mehrere Stufen gliedert. Diese Stufen haben zum Ziel die Erfüllung des Geschäftprozesses.
Durch die Zusammensetzung einzelner Web Services entsteht ein neuer Web Service, der wieder in andere Web Services eingebunden werden kann. Diese Funktionalität wird als rekursive Komposition bezeichnet.
So lassen sich nun komplexe und anspruchsvolle Geschäftsprozesse auf einfache Weise mittels Web Services realisieren.

2. Google Web Service

Da im Folgenden die Struktur eines kompletten WSDL Dokuments genauer beschrieben werden soll und dies sich am besten anhand eines durchgehenden Beispiels realisieren lässt, werde ich hier kurz etwas zu dem von mir gewählten Web Service und dessen Funktionen sagen.
Es handelt sich um einen von Google angebotenen Web Service, wobei dieser drei Funktionen zur Verfügung stellt. Zum einen die von www.google.de bekannte Funktion als Suchmaschine mittels eines Search Request. Man sendet eine Anfrage mit einem Suchbegriff an Google und erhält eine Reihe von URL's als Antwort (pro Anfrage erhält man aber nie mehr als 10 URL's zurückgesendet). Eine weitere Funktion des Google Web Service ist ein Cache Request. Der Benutzer hat hierbei die Möglichkeit eine URL an Google zu senden und erhält als Antwort den Inhalt der Seite aus der Google Datenbank (soweit dieser vorhanden ist).
Als letztes steht noch die Funktion eines Spelling Request zur Verfügung. Dabei handelt es sich um eine Funktion die man ebenfalls von der Google Suchmaschine kennt. Sollten wir uns Beispielsweise beim Eingeben des Suchbegriffes „Definition WSDL" verschreiben so würde uns auf der nächsten Seite ein Link fragen: „Meinten Sie: Definition WSDL". Genau diese Funktion der Rechtschreibprüfung bietet uns ein Search Request. Man sendet eine String an Google und erhält diesen unter Umständen korrigiert zurück.
Hier noch einmal alle drei Funktionen mit Ein- und Ausgabe im Überblick:

1.Search Request:
Eingabe: key, q, start, maxResults, filter,restrict, safeSearch, lr, io, oe
Ausgabe: GoogleSearchResult

2. Cache Request
Eingabe: key, url
Ausgabe: base64Binary

3. Spelling Request
Eingabe: key, phrase
Ausgabe: string

Bei der in allen Eingaben auftretenden Variable key handelt es sich um einen Identifikationsschlüssel den man von Google nach erfolgreicher Anmeldung zugesandt bekommt und der die Nutzung des Web Services auf 1000 Anfragen pro Tag beschränkt.

3. Syntax

Wir wissen nun welche Funktion uns der Google Web Service zur Verfügung stellt, aber wie sieht nun die genaue Definition des Web Services mittels eines WSDL Dokuments aus?
Wie in allen XML Dokumenten gibt es auch bei WSDL ein ausgezeichnetes Wurzelelement. Im Falle von WSDL handelt es sich um das Element *definitions.* Innerhalb dieses Wurzelelements kann es sechs Typen von Kindelementen (description Elemente) geben:

- *types* *(höchstens 1)*

- *message*
- *portType*
- *serviceType (kaum verwendet)*
- *binding*
- *service*

} *(0 oder mehrere)*

Wobei die Reihenfolgen der einzelnen Elemente in dem WSDL Dokument, wie zuvor beschrieben, einzuhalten ist, da die einzelnen Elemente auf zuvor definierte Elemente referenzieren.
Bis auf das Element *types* (welches maximal einmal innerhalb eines WDSL Dokuments verwendet werden darf) können alle anderen description Elemente mehrmals auftreten. Ein gültiges WSDL Dokument muss aber mindestens ein description Element beinhalten. Da aber wie bereits erwähnt, einzelne Elemente Referenzen auf zuvor definierte Elemente beinhalten und somit die Existenz des Anderen voraussetzen (z.B. referenziert das Element *binding* ein zuvor definiertes Element *portType*), wird ein WSDL Dokument meist aus den 5 Hauptelementen *types, messages, portType, binding und service* bestehen.

Die folgende Grafik stellt die 5 Hauptelemente und deren Referenzen zueinander dar.

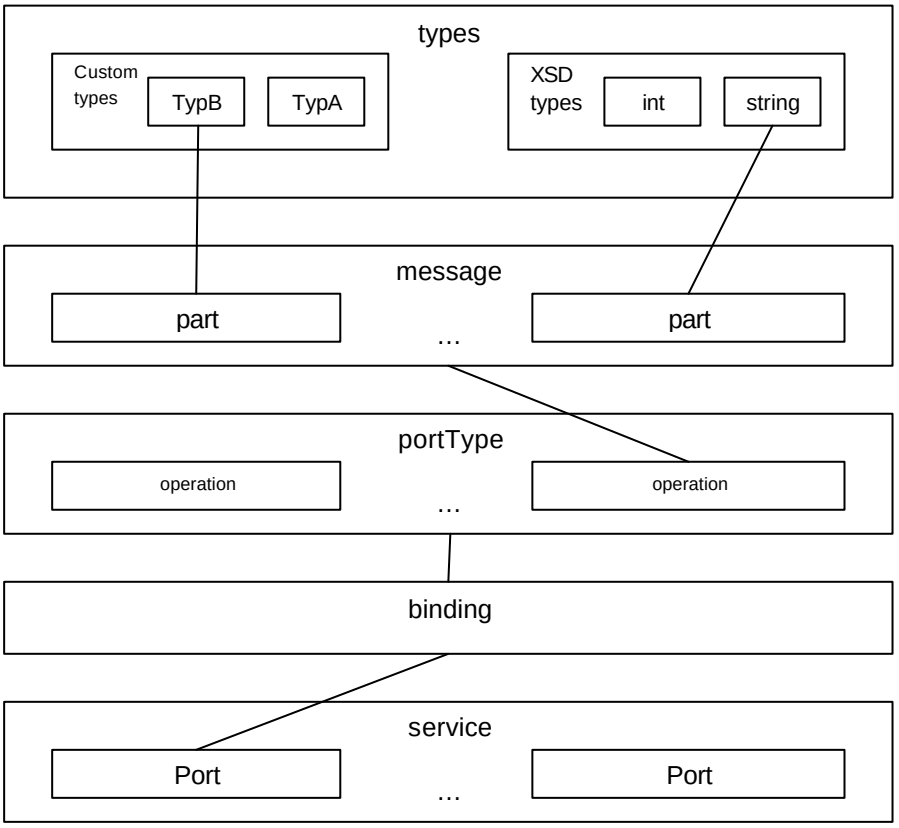

Abbildung 1: Zusammenhang der 5 description Elemente

Im Folgenden werden die einzelnen Elemente eines WSDL Dokuments allgemein beschrieben und anschließend der dazugehörige Abschnitt des Google Web Services angegeben.

3.1. Das Element definitions

Es ist das Wurzelelement eines WSDL Dokuments und hat den gleichen Zweck wie das Element *schema* in einer XML Schema Definition, denn wie dieses kann auch ein WSDL Dokument seinen eigenen Namensraum, mittels des optionalen Attributs *targetNamespace*, definieren. Dieser Namensraum ermöglicht die zuvor beschriebenen Referenzen (Verweise) der einzelnen Entitäten untereinander. Neben dem Attribut *name* und *targetNamespace* kann das Element *definitions* noch weitere Verweise auf Namensräume beinhalten. Im Beispiel des Google Web Services wird dem Element *definitions* zuerst das Attribut name (mit dem Wert „GoogleSearch“) hinzugefügt und der Zielnamespace auf „urn:GoogleSearch“ gesetzt. Dann wird ein Verweis auf den Zielnamespace selbst gemacht und ihm das Präfix *typens* zugewiesen. Durch dieses Präfix kann man nun die einzelnen Elemente in diesem Dokument direkt referenzieren. Anschließend wird noch auf verschiedene Namensräume verwiesen um später im Dokument auf einzelne Entitäten zu verweisen, die in diesen Namensräumen definiert sind (z.B. XSD, SOAP). Zuletzt wird der WSDL-Namespace auf den WSDL Standardnamespace gesetzt.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions          name="GoogleSearch"
                      targetNamespace="urn:GoogleSearch"
                       xmlns:typens="urn:GoogleSearch"
                       xmlns:xsd="http://www. w3.org/2001/XMLSchema"
                       xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                       xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
                       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                       xmlns="http://schemas.xmlsoap.org/wsdl/">

          <!--  hier stehen die Definitionen -->

</definitions>
```

Listing 1: definitions Element des Google Web Service

3.2. Das Element types

Innerhalb dieses Elements werden die Datentypen definiert, auf die später im WSDL Dokument referenziert wird. Das Standardtypensystem zur Beschreibung von Daten ist XML Schema und laut W3C erreicht man mit diesem auch eine maximale Kompatibilität mit WSDL. Es können aber auch beliebige andere Typensysteme zur Beschreibung der Datentypen verwendet werden.
Als einziges description Element benötigt das Element *types* kein *name* Attribut.
Im Folgenden Beispiel wird mittels XSD ein Datentyp „GoogleSearchResult“ definiert, der das Funktionsergebnis eines GoogleSearchRequests darstellt. Dieser Datentype beinhaltet wiederum mehrere Unterelemente, hier seien nur einige aufgeführt:

documentFiltering	Boolean Wert der angibt ob während der Suche ein Filter verwendet wurde oder nicht
estimatedTotalResultsCount	Integer Wert der die approximierte Anzahl der Suchergebnisse repräsentiert
estimateIsExact	Boolean Wert welcher angibt ob die Anzahl der Suchergebnisse exakt ist oder nicht
resultElements	Array der einzelnen Suchergebnisse
searchTime	Double Variable der Suchzeit

Weitere Typdefinitionen (wie z.B. ResultElementArray oder DirectoryCategoryArray) wurden der Übersichtlichkeit wegen weggelassen, erfolgen aber analog zur Definition von GoogleSearchResult.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions ... >
  <types>
    <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"  targetNamespace="urn:GoogleSearch">
      <xsd:complexType name="GoogleSearchResult">
        <xsd:all>
          <xsd:element name="documentFiltering"              type="xsd:boolean"/>
          <xsd:element name="searchComments"                 type="xsd:string"/>
          <xsd:element name="estimatedTotalResultsCount"     type="xsd:int"/>
          <xsd:element name="estimateIsExact"                type="xsd:boolean"/>
          <xsd:element name="resultElements"                 type="typens:ResultElementArray"/>
          <xsd:element name="searchQuery"                    type="xsd:string"/>
          <xsd:element name="startIndex"                     type="xsd:int"/>
          <xsd:element name="endIndex"                       type="xsd:int"/>
          <xsd:element name="searchTips"                     type="xsd:string"/>
          <xsd:element name="directoryCategories"            type="typens:DirectoryCategoryArray"/>
          <xsd:element name="searchTime"                     type="xsd:double"/>
        </xsd:all>
      </xsd:complexType>
      <!-- weitere Typdefinitionen -->
    </xsd:schema>
  </types>
<!-- weitere Definitionen -->
</definitions>
```

Listing 2: Auszug der types Definition des Google Web Service

3.3. Das Element message

Da in WSDL (insbesondere im Element *types*) mehrere Schemadefinitionsformate verwendet werden können, es für eine Sprache zur Beschreibung von Web Services aber unerlässlich ist Nachrichten zu beschreiben, muss es also möglich sein Nachrichten allgemein zu identifizieren bzw. zu beschreiben. Genau aus diesem Grund gibt es das Element *message* in der WSDL Spezifikation. Es beschreibt die Nachrichten die zwischen dem Client und dem Server übermittelt werden. Ein WSDL Dokument kann mehrere *message* Elemente enthalten, wobei jedes einen unter allen *message* Elementen eindeutigen Namen haben muss.
Innerhalb eines *message* Elements gibt es wiederum ein oder mehrere *part* Elemente, von denen ebenfalls jedes einen eindeutigen Namen innerhalb dieses message Elementes haben muss. Diese *part* Elemente beschreiben nun die eigentlichen Parameter (Daten) der zu definierenden Nachricht. Dies geschieht durch eines der beiden Attribute *type* oder *element*.
Im Listing 3 werden nun sechs Nachrichten definiert, wobei immer zwei Nachrichten zu einer Funktion des Google Web Service gehören. Eine Nachricht beschreibt die Eingabe (die Informationen die zum Server gesendet werden), die mit do[Funktion] bezeichnet wird, und die andere die Ausgabe (die Antwort des Servers an den Client), die mit do[Funktion]Response bezeichnet wird. Da XSD über einfache Datentypen wie boolean, int und string verfügt, können alle Parameter mit solchen Datentypen direkt aus der Spezifikation der Ein- und Ausgabe des Google Web Services übernommen werden. Lediglich die Nachricht doGoogleSearchResponse referenziert einen Typ (GoogleSearchResult) der zuvor im Element *types* definiert wurde.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions ... >
<!-- weitere Definitionen -->
  <message name="doGoogleSearch">
    <part name="key"          type="xsd:string"/>
    <part name="q"            type="xsd:string"/>
    <part name="start"        type="xsd:int"/>
    <part name="maxResults"   type="xsd:int"/>
    <part name="filter"       type="xsd:boolean"/>
    <part name="restrict"     type="xsd:string"/>
    <part name="safeSearch"   type="xsd:boolean"/>
    <part name="lr"           type="xsd:string"/>
    <part name="ie"           type="xsd:string"/>
    <part name="oe"           type="xsd:string"/>
  </message>
  <message name="doGoogleSearchResponse">
    <part name="return"  type="typens:GoogleSearchResult"/>
  </message>
  <message name="doGetCachedPage">
    <part name="key"      type="xsd:string"/>
    <part name="url"      type="xsd:string"/>
  </message>
  <message name="doGetCachedPageResponse">
    <part name="return"   type="xsd:base64Binary"/>
  </message>
  <message name="doSpellingSuggestion">
    <part name="key"      type="xsd:string"/>
    <part name="phrase"   type="xsd:string"/>
  </message>
  <message name="doSpellingSuggestionResponse">
    <part name="return"   type="xsd:string"/>
    </message>
<!-- weitere Definitionen -->
</definitions>
```

Listing 3: message Element des Google Web Service

Genau zwischen den Nachrichten doGoogleSearchResponse und doGoogleSearch wird auch schon ein entscheidender Unterschied der Nachrichten Definition sichtbar: Zum einen kann man die zu übergebenden Parameter einer Nachricht jeweils als eigenes *part* Element definieren (wobei alle Parameter bereits vorhanden Typen von XML Schema sein müssen) oder man definiert alle Parameter als neuen Typ mittels XML Schema im Element *types* und referenziert dieses dann im Element *part*. Von Microsoft (speziell im Visual Studio .Net) wird nur die letztere Variante der *message* Definition verwendet.
Um diesen Unterschied der Definition noch einmal zu verdeutlichen zeigen Listing 4a und Listing 4b zwei unterschiedliche *message* Elemente der jeweils gleichen Nachricht. Es handelt sich jeweils um eine Nachricht „AddMsgIn“, die zwei Parameter x und y jeweils vom Typ Integer besitzt.

```
<types>
  <xsd:schema …>
    <xsd:complexType name="Add">
      <xsd:all>
        <xsd:element name="x"   type="xsd:int"/>
        <xsd:element name="y"   type="xsd:int"/>
      </xsd:all>
    </xsd:complexType>
  </xsd:schema>
</types>
<message name="AddMsgIn">
  <part name="parameters" type="s:Add"/>
</message>
```

Listing 4a: message „AddMsgIn“ Beispiel 1

```
<message name="AddMsgIn">
  <part name="x" type="xsd:int"/>
  <part name="y" type="xsd:int"/>
</message>
```

Listing 4b: message „AddMsgIn“ Beispiel 2

3.4. Das Element portType

Dieses Element enthält abstrakte Operationen um die Art des Nachrichtenaustauschs darzustellen die zwischen dem Client und dem Server stattfindet.
Jedes *portType* Element hat ein oder mehrere *operation* Elemente, wobei jedes dieser *operation* Elemente einen unter allen *operation* Elementen eindeutiges *name* Attribut haben muss (dies gilt laut W3C, es gibt aber Abweichungen, siehe unten). Innerhalb jedes *operation* Elementes muss ein *input* und/oder *output* Element existieren.
Durch diese Festlegung der Verwendung von *input* und *output* Elementen innerhalb einer *operation* Entität ergeben sich unterschiedliche Anordnungen und somit vier verschiedene Arten von *portType* Operationen:

1. Input-Output Operation (Request-Response):
   Operation im klassische RPC Stil,
   der Client stellt eine Anfrage an den Server
   und der Server sendet eine entsprechende Antwort
   (oder eine Fehlernachricht) zurück an den Client.

```
<operation name=..>
  <input message= .../>
  <output message= .../>
</operation>
```

2. Input-Only Operation (One-Way):
   Der Client sendet eine Nachricht an den Server,
   der Server sendet aber keine Antwort zurück
   an den Client.

```
<operation name=..>
  <input message= .../>
</operation>
```

3. Output-Input Operation (Solicit-Response):
   Der Server sendet eine Anfrage an den Client
   und der Client schickt einen Antwort (oder
   eine Fehlernachricht) zurück an den Server.

```
<operation name=..>
  <output message= .../>
  <input message= .../>
</operation>
```

4. Output-Only Operation (Notification):
   Der Server sendet eine Nachricht an den Client,
   dieser erhält aber keine Antwort vom Selbigen.

```
<operation name=..>
  <output message= .../>
</operation>
```

Außerdem können bei Operationen mit *input* und *output* Elementen noch mehrere optionale *fault* Element angegeben werden um verschieden Fehlermeldungen, die während der Verarbeitung aufgetreten sind, weiterzuleiten. Diese Operationen können außerdem noch das optionale Attribut *parameterOrder* verwenden um die Parameterreihenfolge zu beschreiben. Da SOAP aber dafür eine geeignete Methode zur Verfügung stellt, wird dieses Attribut selten verwendet.
Laut W3C dürfen verschiedene *operation* Elemente nicht den gleichen Namen besitzen. Da dies aber gerade bei Web Services mit überladenen Methoden (d.h. mehrere Methoden mit dem gleichen Namen aber verschiedenen Parametern) unpraktisch ist, gibt es von Microsoft für solche Fälle die folgende Konvention: Die *input*, *output* oder *fault* Elemente der jeweiligen Operationen müssen eindeutige *name* Attribute erhalten.
Im Fall des Google Web Services haben wir drei RPC Operationen, innerhalb dieser wird auf die jeweilige zuvor definierte Nachricht verwiesen. Die drei *operation* Elemente innerhalb des *portTypes* bilden also eine abstrakte Definition der drei Methoden des Google Web Service.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions ... >
<!-- weitere Definitionen -->
  <portType name="GoogleSearchPort">
    <operation name="doGetCachedPage">
      <input message="typens:doGetCachedPage"/>
      <output message="typens:doGetCachedPageResponse"/>
    </operation>
    <operation name="doSpellingSuggestion">
      <input message="typens:doSpellingSuggestion"/>
      <output message="typens:doSpellingSuggestionResponse"/>
    </operation>
    <operation name="doGoogleSearch">
      <input message="typens:doGoogleSearch"/>
      <output message="typens:doGoogleSearchResponse"/>
    </operation>
  </portType>
<!-- weitere Definitionen -->
</definitions>
```

Listing 5: portType Element des Google Web Service

3.5. Das Element serviceType

Obwohl dieses Element laut W3C noch zu den *definition* Elementen innerhalb eines WSDL Dokuments zählt und auch von folgenden Elementen (*service*) referenziert werden muss, findet dieses Element in der Praxis nur selten Verwendung. Da aber wie bei anderen Elementen der WSDL Spezifikation, die eigentliche Verwendung sich von der Vorgabe innerhalb der Spezifikation erheblich unterscheidet, wird dieses Element wohl in absehbarer Zeit so nicht mehr in der Spezifikation zu finden sein, da dieses Element nur einen Zwischenschritt von der *portType* Definition zur service Definition darstellt. Denn in einem Element *serviceType* können eine oder mehrere *portType* Elemente existieren, die auf einen zuvor definierten *portType* verweisen. Auch innerhalb des Google Web Services wurde auf die Verwendung des Elementes *serviceType* verzichtet.

3.6. Das Element binding

Dieses Element soll genutzt werden um dem Benutzer eines Web Services zu beschreiben, wie Nachrichten die vom Server kommen bzw. zum Server gesendet werden, formatiert werden müssen und welches Transportprotokoll für den Nachrichtenaustausch verwendet wird.
Es kann Bindungsdefinitionen verschiedener Protokolle wie z.B. SOAP enthalten. Innerhalb einer *binding* Komponente darf aber nicht mehr als ein Protokoll verwendet werden. Um also einen Web Service zu beschreiben der in der Lage ist mittels verschiedener Transportprotokolle zu kommunizieren, ist es notwendig für jedes Protokoll ein separates *binding* Element zu definieren.
Jedes *binding* Element muss eine Attribut *type* enthalten, welches auf einen zuvor definierten *portType* referenziert. Die Bindung an ein bestimmtes Protokoll wird dann mit der Verwendung von bestimmten Erweiterungselementen erreicht, wobei jedes Protokoll über einen eigenen Satz an Erweiterungselementen verfügt. Die Namen der *opertion* Elemente und die Anzahl der *input*, *output* und *fault* Kindelemente, innerhalb einer *binding* Definition, müssen exakt mit denen des referenzierten *portTypes* übereinstimmen. In manchen Fällen werden die Erweiterungselemente schon auf der *portType* Ebene verwendet.
Die konkreten Definitionen der einzelnen Erweiterungselemente wird aber nicht in der WSDL Spezifikation gegeben sondern in einem extra Dokument des W3C (http://www.w3.org/TR/wsdl12-bindings). WSDL-Bindings definiert darin Erweiterungs – elemente für SOAP, HTTP und MIME.

Der Nachrichtenaustausch des Google Web Service geschieht über SOAP. Demzufolge werden auch die entsprechenden SOAP Erweiterungselemente verwendet. Zuerst wird der im letzten Abschnitt definierte GoogleSearchPort referenziert und die Art des Nachrichtenaustauschs auf rpc (Remote-Procedure-Call) gesetzt. Die Namen der Operationen und die Anzahl der *input* bzw. *output* Elemente entsprechen denen des GoogleSearchPort. Der Übersicht wegen wurden hier nur die Definitionen der GoogleSearch Nachrichten angegeben, die des SearchRequest bzw. CacheRequest entstehen aber analog.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions ... >
<!-- weitere Definitionen -->
  <binding name="GoogleSearchBinding" type="typens:GoogleSearchPort">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <!-- weitere Operationen -->
    <operation name="doGoogleSearch">
      <soap:operation soapAction="urn:GoogleSearchAction"/>
      <input>
        <soap:body use="encoded"
                   namespace="urn:GoogleSearch"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                   namespace="urn:GoogleSearch"
                   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>
<!-- weitere Definitionen -->
</definitions>
```

Listing 6: binding Element des Google Web Service

3.7. Das Element service

Hier wird nun der eigentliche Web Service definiert. Ein „Service" ist eine Ansammlung von *ports*, wobei jeder *port* ein abgeschlossener *portType* ist, dieser beinhaltet dann einen spezifischen Endpunkt (Adresse), welcher angibt wo der Service zu erreichen ist.
Laut W3C muss jede *service* Komponente ein *serviceType* Attribut haben, welches auf einen zuvor definierten *serviceType* verweist. Aber wie bereits erwähnt sind Verstöße gegen die WSDL Spezifikation eher die Regel als die Ausnahme und da in den meisten Fällen auf die Verwendung des *serviceType* Elements verzichtet wird, wird man in den selbigen WSDL Dokumenten das *serviceType* Attribut nicht vorfinden.
Jedes *port* Element hat ein *binding* Attribut, das auf ein vorher definiertes *binding* referenziert. Außerdem muss in einem Anschluss (*port*) immer ein konkreter Endpunkt definiert werden. Diese Definition des Endpunktes erfolgt wie schon bei der Protokollbindung über spezielle Erweiterungselemente die ebenfalls vom Protokoll abhängig sind. Das heißt also, es besteht innerhalb des Elements *service* die Möglichkeit mehrere Anschlüsse zu definieren, die entweder auf das gleiche *binding* verweisen und/oder deren Endpunkte sich an verschiedenen Adressen befinden. Die verschiedenen *ports* sollten sich also alle gleiche verhalten, können ihre Funktionalität aber über unterschiedliche Transportprotokolle oder an verschiedenen Endpunkten anbieten. Ein Client kann dann beispielsweise über die verschiedenen Anschlüsse iterieren um eine kompatible Bindung mit einem geeigneten *portType* und Protokoll zu finden.
Der Google Web Service verweist demzufolge auf die zuvor definierte SOAP-Bindung und definiert dann den Endpunkt mit der Adresse http://api.google.com/search/beta2.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions ... >
<!-- weitere Definitionen -->
  <service name="GoogleSearchService">
    <port name="GoogleSearchPort" binding="typens:GoogleSearchBinding">
      <soap:address location="http://api.google.com/search/beta2"/>
    </port>
  </service>
</definitions>
```

Listing 7: service Element des Google Web Service

3.8. Das Element import

Da ein WSDL Dokument auch bei einem Web Service mit geringer Funktionalität sehr schnell sehr umfangreich werden kann, besteht durch das Element *import* (wie auch bei XSD) die Möglichkeit andere Dokumente in ein WSDL Dokument zu importieren. Somit wäre es beispielsweise denkbar, die Elemente *types*, *messages* und *portType* in einem eigenen Dokument zu definieren, da diese drei Definitionen von der Art des Protokolls unabhängig sind. Anschließend definiert man die einzelnen *bindings* für die verschiedenen Protokolle in separaten Dokumenten und speichert die eigentliche *service* Definition in einer eigenen WSDL Datei welche die anderen Dokumente dann importiert. Wobei noch zu beachten ist, dass *import* Elemente immer vor den description Elementen stehen müssen. Dies ist ein in der Praxis häufig verwendetes Verfahren um die Lesbarkeit und Portabilität von WSDL Dokumenten zu verbessern.

4. Überleitung

Da wir nun beschrieben haben, wie man mit einem Web Service kommunizieren kann, werden wir jetzt auf die genaue Zusammensetzung eines Web Services eingehen. Wie eingangs beschrieben, nutzen wir dazu WSFL.
Dazu unterteilt WSFL die Beschreibung in zwei Arten von Zusammensetzungen:

- *Flow Models:*
  Es wird der Ablauf einer Reihe von Aktivitäten von einem oder mehreren Akteuren beschrieben. Dieser Ablauf bildet als Ergebnis das Erreichen einer übergeordneten Aufgabe.

- *Global Models:*
  Hier werden die Zusammenhänge und Verbindungen einzelner Web Services unter – einander beschrieben. Als Ergebnis erhält man eine Gesamtbeschreibung der Inter – aktionen der Partner.

Jedes Flow und Global Model wird in ein public und ein private Interface unterteilt, wobei das public Interface in WSDL beschrieben wird und das private Interface in WSFL.
Im Folgenden werden wir diese Zusammenhänge genauer erläutern.

5. Ausgangssituation

Als Ausgangssituation dieses Abschnittes wird nun folgendes Beispiel eingeführt:
Wir stellen uns vor, dass wir endlich unseren wohlverdienten Sommerurlaub planen wollen und schließlich auch realisieren wollen. Dieser sollen aus verschiedenen Stufen bestehen, da wir uns viele Orte und Sehenswürdigkeiten ansehen wollen.
Dazu könnten wir jetzt in sämtliche Reisebüros gehen um uns über Preise und in Frage kommende Urlaubziele zu informieren. Dies ist sehr zeitaufwendig und deshalb wollen wir unser Vorhaben vereinfachen:
Wir schalten unseren Rechner ein, loggen uns ins Internet ein und gehen dort auf die Suche nach einem passenden Angebot, werden dort natürlich auch fündig.
Jetzt fallen eine Reihe von Aktivitäten seitens des Reisenden (*Traveler*), des virtuellen Reisebüros (*Agent*) und der Airline, mit der das Reisebüro kommunizieren muss, damit die Buchung der Reise vollkommen wird.
Der Reisende muss die geplante Reiseroute nun zum Reisebüro schicken, dazu gehören natürlich auch persönliche Angaben zur Person, Kreditkartennummer und die detaillierten Reisedaten.
Das Reisebüro muss dann die genauen Etappen der Reise bestimmen (z.B. genauer Aufenthalt am jeweiligen Ort), diese Daten an die Airline schicken, damit diese die Sitze reservieren kann.
Die Airline muss dann die angeforderten Daten dem Reisbüro bestätigen. Dann kann die endgültige Reiseroute vom Reisebüro festgelegt und dem Reisenden bestätigt werden.
Die Airline kann dann die Tickets an den Reisenden schicken.
Diese Kommunikation unter den Akteuren ist in der folgenden Grafik ersichtlich:

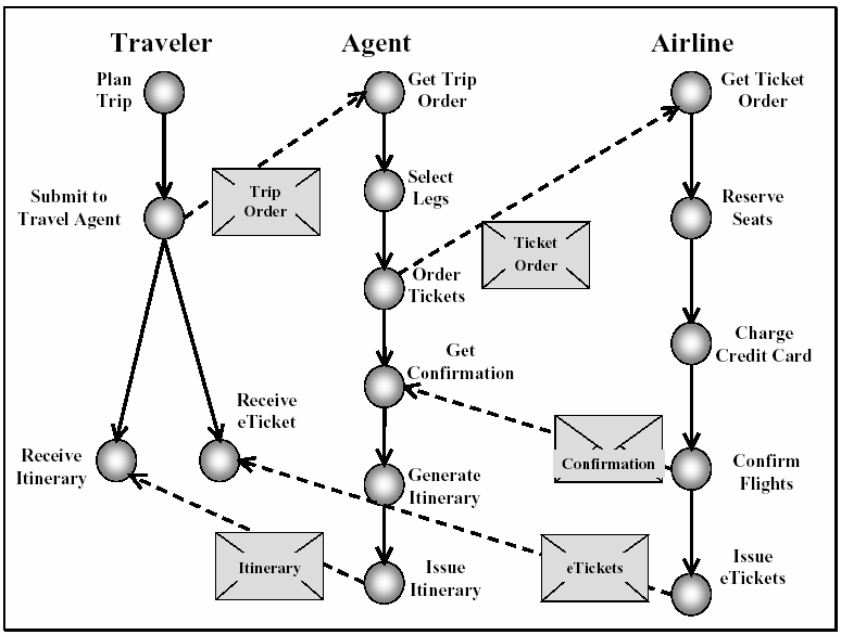

Abbildung 2: Definition der Beziehungen

In der Grafik (Abb.2) stellen die Kreise die Aktivitäten (*activities*) dar und die Kästchen die Interaktion zwischen den einzelnen Partnern.

6. Modellierung der Ausgangssituation

Wie man ebenfalls in der Grafik (*Abb.2*) sehen kann, kann der Geschäftsprozess als Graph repräsentiert werden:
Die Aktivitäten der Akteure werden als Knoten dargestellt und können Nachrichten bekommen (*input – oder output messages*) oder senden (gerichtete durchgezogene Kanten). Jede Nachricht kann beliebig viele Aufgaben haben.
Die gerichteten Kanten (gestrichelte Pfeile) des Graphen stellen die Verbindungen der einzelnen Web Services untereinander dar.
Die Kanten können Bedingungen enthalten, die dann Einfluss auf die jeweiligen folgenden Aktivitäten haben können.
Erkennbar ist hier auch die notwendige Einhaltung der Reihenfolge der Definitionen in den WSFL – Dokumenten. Dadurch sind auch Schleifen innerhalb eines Prozesses realisierbar, jedoch sei hier auf die WSFL – Spezifikation in der Quellenangabe verwiesen [5].

7. Die Sprache

Die Web Service Flow Language wird nun anhand des oben angegebenen Beispiels genauer betrachtet, wobei die in der Grafik (*Abb. 2*) vorhandenen Elemente in der WSFL – Grammatik zuerst als XML – Elemente definiert. Danach erfolgt die Definition der private und public Interfaces in WSFL und WSDL.

7.1 Definition der Nachrichten inklusive der Typdefinitionen

Als erstes braucht man ein Grundgerüst an Typdefinitionen, auf denen dann die Nachrichten (*messages*) zugreifen können.
In dem folgenden Listing 8 sieht man die Definition der Typen *CreditCard, Person, Recipient* und *Journery*. Bei der Definition des Elements *tripOrder* sieht man, dass hier die zuvor definierten Elemente *Journey, CreditCard* und *Recipient* zur Definition von *tripOrder* wieder verwendet werden.
Aus Platzgründen sind hier nicht alle Typdefinitionen aufgeführt, aber die anderen werden gemäß dem XML – Schema genauso definiert.
Rot umrandet findet man eine Definition der Nachricht *receivedTicketOrder*, welche zwischen den Akteuren ausgetauscht wird.
Ähnlich werden dann die anderen benötigten Nachrichten definiert, die dann die Signatur für die einzelnen Aktivitäten darstellen, in denen die *messages* dann zur Kommunikation genutzt werden.

```
<?xml version="1.0"?>
  <definitions name="totalTravel"
   targetNamespace=
http://www.Tck.com/WebServices/Messages/TotalTravel
   xmlns:tio=
http://www.Tck.com/WebServices/Messages/TotalTravel
   xmlns="">

<types>
  <schema
     xmlns= http://www.w3.org/2000/10/XMLSchema
     targetNamespace=
http://www.Tck.com/WebServices/Messages/TotalTravel
     xmlns:tio=
http://www.Tck.com/WebServices/Messages/Totaltravel>
    <element name="CreditCard">
     <complexType>
       <sequence>
         <element name="CardNumber"
                  type="nonNegativeInteger"/>
         <element name="ExpiryDate" type="month"/>
         <element name="Company" type="string"/>
       </sequence>
     </complexType>
    </element>

<complexType name="Person">
  <sequence>
    <element name="FirstName" type="string"/>
    <element name="LastName" type="string"/>
    <element name="Address" type="tio:Address"/>
    <element name="BirthDate" type="date"/>
  </sequence>
</complexType>

<element name="Recipient" type="tio:Person"/>
<element name="Journey">
  <complexType>
    <all>
      <element name="Stage"
         minOccurs="1" maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="Location" type="string"/>
            <element name="Begin" type="date"/>
            <element name="End" type="date"/>
          </sequence>
        </complexType>
      </element>
      <element ref="tio:participants"/>
    </all>
  </complexType>
</element>
```

```
<!--================================== -->
<!-- tripOrder is the basic "where to" -->
<!-- information supplied by the       -->
<!-- traveller to the travel agent     -->
<!--================================== -->
<element name="tripOrder">
  <complexType>
    <all>
      <element ref="tio:Journey"/>
      <element ref="tio:CreditCard"/>
      <element ref="tio:Recipient"/>
    </all>
  </complexType>
</element>
...
</schema>
</types>

<!--================================== -->
<!-- messages externalized by airline -->
<!-- and agent processes              -->
<!--================================== -->
<message name="tripOrderMsg">
  <part name="order" element="tio:tripOrder"/>
</message>
...
<!--================================== -->
<!-- messages used internally to define -->
<!-- the signatures of activities in the -->
<!-- travel agent business process      -->
<!--================================== -->
<message name="receivedTripOrder">
 <part name="request" element="tio:tripOrder"/>
 <part name="agentWorkId"
       element="wsfl:FlowInstanceId"/>
</message>

</definitions>
```

Listing 8: Definition der Nachrichten und Typen

7.2 Definition des public Interfaces für das Flow Model

Jetzt müssen die Schnittstellen des Flow Models (public Interface) definiert werden, die so genannten *port types*. Dies geschieht in der Web Service Description Language, damit andere *service provider* den Webservice auch nutzen können. WSFL erbt diese Spracheigenschaft von WSDL.
Ein *port type* besteht aus mindestens einer Operation. Eine Operation ist die Verbindung zwischen dem *port type* des *service providers* und einer internen Aktivität. Es wird auf die zuvor definierten *messages* zugegriffen.
Im folgenden Listing 9 sehen wir die Schnittstellendefinition der Airline. Die Definition des Reisebüros und des Reisenden wird analog vorgenommen, aber hier nicht mit aufgeführt.
Die Airline hat zwei *port types*. Im *port type ticketHandler* befinden sich zwei Operationen, zum einen wartet die Airline auf die Ticketbestellung (*receiveTicketOrder*) und zum anderen sendet sie die Bestätigung an das Reisebüro, bei erfolgreicher Buchung (*sendConfirmation*).
Der *port type ticketDelivery* sendet mit der Operation *sendETicket* die Tickets zum Reisenden.

```
<!--This is the standalone Airline interface -->
<portType name=”ticketHandler”>
       <operation name=“receiveTicketOrder”>
              <input name=“receiveTicketOrderInput”
              message=“tio:ticketOrderMsg”/>
              <output name=“receiveTicketOrderOutput”
              message=“tio:ticketOrderAck”/>
       </operation>
       <operation name=“sendConfirmation”>
              <ouput name=“sendConfirmationOutput”
              message=“tio:confirmationMsg”/>
       </operation>
</portType>
<portType name=“ticketDelivery”>
       <operation name=“sendETicket”>
              <output name=“sendETicketOutput”
              message=“tio:eTicketMsg”/>
       </operation>
</portType>
```

Listing 9: public Interface des Flow Models

7.3 Definition der Service Provider für das Flow Model

Ein Service Provider stellt seinen Web Service als Komposition von mehreren Aktivitäten zur Verfügung und muss deshalb als solcher definiert (*Service Provider Type*) werden.
Jeder Webservice muss eine Schnittstelle (*port Type*) haben, damit diese von anderen Webservices verwendet werden kann. Der Reisende verwendet in unserem Fall die Schnittstelle *ticketBuyer* im nachfolgenden Listing 10:

```
<definitions name="totalTravel">
  ...
  <serviceProviderType name="travelerType">
    <portType name="abc:ticketBuyer/>
  </serviceProviderType>

  <serviceProviderType name="agentType">
    <portType name="abc:tripHandler/>
    <portType name="abc:ticketBuyer/>
  </serviceProviderType>

  <serviceProviderType name="airlineType">
    <portType name="abc:ticketHandler/>
    <portType name="abc:ticketDelivery/>
  </serviceProviderType>
</definitions>
```

Listing 10: Definition der Service Provider

Die Airline und das Reisebüro (*agent*) werden analog definiert. Da diese jeweils zwei Nachrichten austauschen (siehe gestrichelten Kanten der Abb.2), brauchen sie auch zwei Schnittstellen.
Die *port types* der Airline sind genau die Schnittstellen, die zuvor im Listing 9 definiert wurden.
Wenn sich nun noch Anbieter dazwischen stellen möchte, z.B. ein Webservice, der als Suchmaschine zwischen den Reisebüros agiert um das günstigste Angebot zu finden, muss er diesen Schnittstellenanforderungen genügen.

7.4 Definition des private Interfaces des Flow Models Airline

Um ein WSFL Flow Model komplett zu definieren, braucht man folgende Elemente:

- das <*flowSource*> und <*flowSink*> Element definiert den input und output des Flow Models.
- <*serviceProvider*> Elemente, die die teilhabenden Services definieren.
- Eine externe Schnittstelle (<*export*>) für den Prozess, durch die er Informationen mit der Außenwelt austauschen kann (z.B. eine SOAP – Nachricht).
- <*activity*> Elemente, die die individuellen Nachrichten der Service Provider ent – halten.
- <*controlLink*> und <*dataLinks*> Elemente, die den Kontrollfluss und den Datenfluss steuern.

Listing 11 zeigt das private Interface des Flow Models Airline, es fehlt hier noch das Flow Model des Reisebüros (*agent*). Es wird der Prozess der Ticketausstellung dargestellt.

```
<!--============================================= -->
<!-- definition of bookTickets flow model -->
<!-- using airlineFlow serviceProviderType -->
<!--============================================= -->
<flowModel name="bookTickets"
           serviceProviderType="airlineFlow">
  <flowSource name="ticketFlowSource">
    <output name="processInstanceData"
            message="tio:receivedTicketOrder"/>
  </flowSource>

  <serviceProvider name="agent" type="agentFlow"/>
  <serviceProvider name="traveler"
                   type="travelerType"/>                    1

  <export lifecycleAction="spawn">
    <target portType="tio:ticketHandler"
            operation="receiveTicketOrder">
      <map sourceMessage="receiveTicketOrderInput"
           targetMessage="processInstanceData"
           targetPart="request"/>
      <map sourceMessage="processInstanceData"
           sourcePart="airlineWorkId"
           targetMessage="receiveTicketOrderOutput"
           targetPart="airlineWorkId"/>
    </target>
  </export>                                                 2
...
  <activity name="chargeCreditCard" >
    <input name="dataIn" message="tio:reservation" />
    <output name="dataOut"
            message="tio:chargedReservation"/>
    <performedBy serviceProvider="local"/>
    <implement>
      <internal>
        <!-- .. call to credit card service .. -->
      </internal>
    </implement>
  </activity>                                               3
```

```
<activity name="confirmFlights" >
  <input name="confirmFlightsInput"
         message="tio:chargedReservation"/>
  <output name="confirmFlightsOutput"
          message="tio:eTicketMsg"/>
  <performedBy serviceProvider="agent"/>
  <implement>
    <export portType="tio:ticketHandler"
            operation="sendConfirmation">
      <map sourceMessage="confirmFlightsInput"
           sourcePart="confirmationInfo"
           targetMessage="sendConfirmationOutput"
           targetPart="confirmationInfo"/>
    </export>
  </implement>
</activity>                                                 3

  <controlLink
    name="rS-cC"
    source="reserveSeats"
    target="chargeCreditCard"/>
  <dataLink
    name="rS-cCdata"
    source="reserveSeats"
    target="chargeCreditCard"/>                             4
</flowModel>
```

Listing 11: private Interface des Flow Models Airline

Ein Flow Model namens *bookTickets* wird erstellt und als *service provider* wird *airlineFlow* zugewiesen und die anderen beteiligten *service provider* werden mitdefiniert (Listing 11.1). Dann haben wir eine <*export*> Anweisung, die eine *lifecycleAction* namens *spawn* definiert (Listing 11.2). Jedes Flow Model kann immer in Verbindung mit einem *port type* den „Lebenskreis“ der Instanz eines Flow Models managen. Mit *spawn* startet ein neues unabhängiges Flow Model als *activity* in einem anderen Flow Model.
Mit *map* wird die angegebene *activity* auf eine Operation des *port types* des jeweiligen *service provider types* angewendet.

Die in Listing11.3 beschriebenen *activities* beschreiben den eigentlichen „*Flow*“ der Web Services, dort sind die individuellen Nachrichten der Provider definiert.
Zu beachten ist hier die Reihenfolge in der die *activities* definiert werden, denn nach dieser Reihenfolge werden sie ausgeführt. Das heißt hier konkret, dass erst die Kreditkartendaten (*chargeCreditCard*) benötigt werden und dann die Flüge reserviert werden (*confirmFlights*).
Eine Aktivität kann Eingangs – und Ausgangsmessages haben, die dann zur Verarbeitung weiter genutzt werden, sowie von welchem Web – Service sie ausgeführt wird.
Das <*implement*> - Element in der Aktivität sagt, woher die Aktivität kommt

In Listing 11.4 beschreibt der <*contolLink*> die Reihenfolge der auszuführenden *activities*. Der <*dataLink*> besagt, dass die Daten von *reserveSeats* in *chargeCreditCard* gebraucht werden und deshalb zuerst *reserveSeats* und dann *chargeCreditCard* ausgeführt werden muss. Hier fehlen natürlich noch die anderen *dataLinks* und *controlLinks*, die die Reihenfolge der anderen Aktivitäten festlegen.

7.5 Definition des public und private Interfaces des Global Models

Wie schon in der Einführung beschrieben, beschreibt das Global Model die Zusammenhänge und Verbindungen einzelner Web Services untereinander. Als Ergebnis erhält man eine Gesamtbeschreibung der Interaktionen der Partner.

In unserem Beispiel handelt es sich nun um eine Verbindung zwischen dem Reisebüro (*agent*) und der Airline. Dies geschieht wieder in WSDL, dazu muss eine neue Schnittstelle (*port type*) definiert werden.
In dem folgenden Listing 12 sieht man die Definition der Schnittstelle (*port type*) zwischen dem neu definierten *tripNTicketHandler* zu dem *Traveler*, welches das public Interface des Global Models darstellt.
Der neue *port type* enthält genau 3 *output* – Operationen, und das sind genau die 3 Verbindungen die in Abb.2 dargestellt sind, nämlich *reveiveTripOrder, sendItinerary und sendETickets* (zwischen Reisebüro, Airline und Reisenden).

```
<portType name="tripNTicketHandler">
  <operation name="receiveTripOrder">
    <input name="receiveRequest"
           message="tio:tripOrderMsg"/>
    <output name="returnResponse"
           message="tio:tripOrderAck"/>
  </operation>
  <operation name="sendItinerary">
    <output name="sendMessage"
           message="tio:Itinerary"/>
  </operation>
  <operation name="sendETickets">
    <output name="sendMessage"
           message="tio:ETickets" />
  </operation>
</portType>
```

Listing 12: public Interface des Global Models

Nun können wir das zugehörige private Interface des Global Models definieren (Listing 13), dieses kann dann mit dem Reisenden kommunizieren:

```
<globalModel name="bookTripNTickets"
             serviceproviderType="agentNAirlineFlow">
  <serviceProvider name="travelAgent"
                   serviceProviderType="tio:agentFlow">
    <export>
      <source portType="tio:tripHandler"
              operation="sendItinerary"/>
      <target portType="tio:tripNTicketHandler"
              operation="sendItinerary"/>
    </export>
    <export>
      <source portType="tio:tripHandler"
              operation="receiveTripOrder"/>
      <target portType="tio:tripNTicketHandler"
              operation="receiveTripOrder"/>
    </export>
    <locator type="static"
             service="Traveluck.com"/>
  </serviceProvider>
  <serviceProvider name="airline"
serviceProviderType="tio:airlineFlow">
    <export>
      <source portType="tio:ticketDelivery"
              operation="sendETicket"/>
      <target portType="tio:tripNTicketHandler"
              operation="sendETickets"/>
    </export>
  </serviceProvider>
  <plugLink>
    <source serviceProvider="airline"
            portType="tio:ticketHandler"
            operation="sendConfirmation"/>
    <target serviceProvider="travelAgent"
            portType="tio:tripHandler"
            operation="waitForConfirmation"/>
  </plugLink>
  <plugLink>
    <source serviceProvider="travelAgent"
            portType="tio:tripHandler"
            operation="requestTicketOrder"/>
    <target serviceProvider="airline"
            portType="tio:ticketHandler"
            operation="receiveTicketOrder"/>
  </plugLink>
</globalModel>
```

1

Listing 13: private Interface des Global Models

In dem Listing 13 sehen wir, dass genau zwei *service provider* definiert sind, nämlich das Reisebüro (*travelAgent*) und die *airline.*
Diese beinhalten wieder externe Schnittstellen (*export*), die die Beziehungen der *port types* beschreiben.
Die Operation *sendItinerary* aus dem neuen *port type tripNTicketHandler* soll die gleiche sein, wie die ursprüngliche Operation *sendItinerary* aus dem *port type triphandler*.

Es werden die Bindungen zwischen Service und Service Provider angegeben (*locator*). Hier wird „*static*“ benutzt, d.h., dass in das Global Model ein Verweis auf die WSDL – Beschreibung des zu verwendeten Web – Service eingetragen.
Es gibt noch die Möglichkeit einen Web – Service local, per UDDI und dynamisch zur Laufzeit einzubinden. Bei der lokalen Variante wird ein lokales Programm ausgeführt wird, wie z.B. eine Java – Klasse.
Bei der Einbindung über die Suche per UDDI (Universal Description, Discovery and Integration). Hierbei wird im Global Model die Operation find_service definiert, die ein UDDI – Verzeichnis kontaktiert und die passenden Web – Services heraussucht, die den Anforderungen genügen. Dort kann dann auch festgelegt werden, wann die Suche ausgeführt wird, entweder bereits zur Entwicklungszeit oder zur Laufzeit (dynamische Variante).

Außerdem sind noch zwei *plugLinks* (Listing 13.1) definiert, diese beschreiben die Interaktionen zwischen den *service providern*.
Es bestehen in unserem Beispiel genau zwei Verbindungen zwischen dem Reisebüro und der Airline: die Bestellung der Tickets (*ticketOrder*) und die Sendung der Bestätigung (*confirmation*).
Der erste *plugLink* besagt konkret, dass der Service Provider Airline mit der Schnittstelle tickethandler die Operation *sendConfirmation* an die Operation *waitForConfirmation* des Reisebüro schickt.
Diese Verbindungen ist wieder klar der Abb. 2 zu entnehmen.

8. Rekursive Komposition

Eine weitere Stärke von WSFL ist die Möglichkeit der rekursiven Komposition. Dies bedeutet nichts anderes, als dass auf einfache Weise neue Geschäftsprozesse durch die Kombination von bestehenden definiert können.
Dazu müssen nur die bestehenden WSFL – Dokumente mit Hilfe von WSDL zu einem Web – Service zusammen gepackt werden, so wie dies in unserem Beispiel mit dem Flow Model der Airline und dem Reisebüro gemacht wurde.

9. Fehlerbehandlung in WSFL

In WSFL gibt es die Möglichkeit Fehler zu behandeln, welche sich auf den Kontext von *messages* beziehen, indem man, aus der Sicht des Models, die gerichteten Kanten der verbundenen Aktivitäten mit Übergangsbedingungen versieht.

Allerdings bieten sich weiter keine Möglichkeiten den Transfer der Daten sicher und verlässlich zu machen.

10. Einbindung von ausführbaren Einheiten

Die in WSFL genutzten *activities* können EXE / CMD – Dateien, Java – Klassen, CICS (Customer Information Control System) – Programme sein, wobei die konkrete Einbindung in einer WSDL – Datei erfolgt. Hier ein Beispiel wie die Datei WORD.exe eingebunden werden kann:

```
<definitions name="DocumentProcessing"
targetNamespace="http://example.com/documentProcessing.wsdl"
xmlns:exe="http//schemas.xmlsoap.org/wsdl/exe"/>
<types>
       <element name="Document">
              <complexType>
                     <all>
                            <element name="DocumentName" type="string"/>
                     </all>
              </complexType>
       </element>
</types>
<message name="DocumentIdentification">
       <part name="body" element="Document"/>
</message>
<portType name="DocumentProcessingPortType"
       <operation name="CreateDocument"/>
       <operation name="EditDocument">
              <input message="tns:DocumentIdentification"/>
       </operation>
       <operation name="DeleteDocument">
              <input message="tns:DocumentIdentification"/>
       </operation>
</portType>                                                    1
<binding name="DocumentProcessingBinding"
       type="DocumentProcessingPortType">
<operation name="CreateDocument">
<exe:operation pathAndFileName="word.exe"
                  startInForeground="yes"
                  style="visible"
                  executionMode="normal"/>
</operation>
<operation name="EditDocument">
<exe:operation pathAndFileName="word.exe"
               startInForeground="yes"
               inheritEnvironment="yes"
               style="visible"
               executionMode="normal"/>
```

```
<input>
<exe:input encoding=”commandParameters”/>
</input>
</operation>
<operation name=”DeleteDocument”>
       <cmd:operation pathAndFileName=”deldoc.cmd”/>
</operation>
</binding>
<service name=”DocumentProcessingService”>
       <documentation>Document Processing Service</documentation>
       <port name=”DocumentProcessingPort”
       binding=”tns:DocumentProcessingBinding/>
       </service>
</definitions>
```

2

Listing 14: Einbindung einer EXE – Datei

Man sieht wieder in Listing 14.1, dass eine Schnittstelle zum externen Programm definiert werden muss (*port type*) . In Listing 14.2 sieht man bereits weitere WSDL – Elemente, mit denen dann die exe – Datei eingebunden wird.
Für die anderen Dateiarten sei auf die WSFL – Spezifikation in der Quellenangabe verwiesen [5].

11. Ähnliche Technologien anderer Entwickler

Außer WSFL von IBM gibt es noch andere Business Prozess „Standards“. Zu Ihnen gehören z.B. ebXML BPSS, XLANG, BMPL, WSCI, BPEL4WS und andere. Auf die genannten wird hier kurz eingegangen.

11.1 ebXML BPSS

Dies bedeutet electronic business XML Business Process Spezifikation Schema.
BPSS ist ein Teil der ebXML B2B Spezifikationen und in BPSS werden nur die öffentlichen Prozesse auf eine einfache Art und Weise beschrieben. Diese Beschreibung erfolgt mittels UML und es wird dabei nicht darauf Rücksicht genommen wie die Daten zwischen den Akteuren fließen. Weitere Informationen sind unter

http://www.ebxml.org/specs/ebBPSS.pdf

zu finden.

11.2 XLANG

XLANG ist von Microsoft entwickelt und legt den Schwerpunkt ebenfalls nur auf die öffentlichen Prozesse. Um die Service Interfaces jedes Teilnehmers zu beschreiben wird WSDL benutzt. Weitere Informationen findet man unter :

http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

11.3 BPML

BPML bedeutet Business Process Modeling Language und ist eine Spezifikation von der Business Process Management Initiative (BPMI).

Ziel dieser Spezifikation ist das Abbilden der Prozesse in einer Organisation. BMPL ist komplementär zu anderen Standards wie BPSS, die ja nur den öffentlichen Prozess abbilden, angeordnet. Informationen dazu unter:

http://www.bpmi.org/

11.4 WSCI

WSCI bedeutet Web Service Choreography Interface und ist eine auf XML basierende Interface Sprache um die Interaktion zwischen einem Web Service und den teilnehmenden anderen Choreographen zu beschreiben und kann ebenfalls mit WSDL zusammen arbeiten.
Wieder handelt es sich um eine Beschreibung der öffentlichen Prozesse und nicht um das interne Verhalten. Weitere Informationen unter:

http://www.w3.org/TR/wsci/

11.5 BPEL4WS

BPEL4WS bedeutet Business Process Execution Language for Web Services und wurde von BEA, IBM und Microsoft entwickelt.
BPEL ist eine Vereinigung von XLANG und WSFL und eine Sprache um Geschäftsprozesse und die Kommunikationsprotokolle zu beschreiben. Durch die Vereinigung beider Spezifikationen erhält man hier die interne und öffentliche Beschreibung von Geschäftsprozessen und man kann WSDL einbinden. Nähere Informationen unter:

http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

12. Zusammenfassung

Wie bei allen XML Formaten liegt auch bei WSDL der große Vorteil in der Plattformunabhängigkeit. Wir haben ein komplettes WSDL Dokument eines Web Service betrachtet, ohne die eigentliche Implementierung des Selbigen zu kennen. Genau dieser Vorteil ermöglicht es, dem Anwender Funktionalitäten eines Web Services in seinem Programm zu verwenden, ohne an eine bestimmte Programmiersprache gebunden zu sein. Das einzige was der Nutzer wissen muss, ist, über welches Protokoll und mit welchen Parametern muss ich mit diesem Web Service kommunizieren und wie implementiere ich dies in meiner verwendeten Programmiersprache. Ein weiterer Vorteil der XML – Sprachen, und somit auch von WSDL, ist die Struktur der Grammatik, die so aufgebaut ist, dass viele Dokumente automatisch von Programmen generiert bzw. eingelesen werden können und so der Mensch als Fehlerquelle ausgeschlossen werden kann. Im Visual Studio .NET ist z.B. das Erzeugen eines WSDL Dokuments mittels eines Mausklicks möglich.
Aber wie bei allen Standards ist auch bei WSDL das Problem, dass die Entwicklung der Spezifikation nicht einhergeht mit der Entwicklung auf dem Markt. Wenn eben bestimmte Funktionalitäten benötiget werden, die aber innerhalb der Spezifikation nicht vorhanden sind,

werden diese von diesen Firmen als eigener Standard bzw. Konvention herausgebracht (siehe Microsoft und die Überlagerung von Methoden). Das genaue Gegenteil bewirkt der Darwinismus, Definitionen die von keinem genutzt werden und demzufolge wertlos sind, werden in späteren Versionen wohl kaum weiterverwendet, höchstens wegen der Abwärtskompatibilität (siehe *serviceType*).
Es ist mit WSDL also eine Möglichkeit entstanden, die es dem Benutzer gestattet Web Services in einer „standardisierten" Form zu beschreiben, diese also der Außenwelt zu Verfügung zu stellen oder selbst Web Services zu nutzen. Es sollten trotzdem externe WSDL Dokument immer mit Skepsis betrachten werden. Und bei der Erstellung eigener Dokumente sollte man darauf vorbereitet sein, dass es trotz Verwendung des WSDL Standards, immer zu Problemen mit verschiedenen Applikationen kommen kann.
Um die gleiche Freiheit der Plattformunabhängigkeit für die Zusammensetzung eines Web Services zu erlangen, muss man sich erst durch einen Dschungel von Spezifikationen wühlen und die passende für seine eigenen Bedürfnisse zu finden. Hier wurde mit WSFL ein kleiner Teil abgedeckt. WSFL wird in zwei große Teile gespalten: Der erste Teil, das Flow Model, umfasst den internen Ablauf eines Web Services. Der zweite Teil, das Global Model, behandelt die Interaktion von Web Services untereinander. So erhält man mit WSFL ein großes Maß an Abstraktion.
Die beiden Modelle können mehrmals geschachtelt ausgeführt werden (rekursive Komposition). Durch die leichte Schnittstellendefinition können schnell neue Web Services eingebunden werden.
Mit WSFL können auch Fehler behandelt werden, allerdings bietet es keine Möglichkeit der Sicherheit und Zuverlässigkeit, dies wird dann erst in der WSEL Spezifikation berücksichtigt.
Es lassen sich ausführbare Einheiten in Form von Java – Klassen, EXE / CMD – basierende Implementierungen und CICS – Programme einbinden.

Allerdings sind so viele dieser Technologien auf dem Markt, dass sich für die Zukunft nur hoffen lässt, dass sich ein gemeinsamer Standard einstellen wird und nicht jeder sein eigenes Süppchen weiter kocht.

13. Abbildungsverzeichnis

14. Quellen

[1] Web Service Definition
www.w3c.org (09.02.2003)
[2] WSDL Spezifikation
http://www.w3.org/TR/2003/WD-wsdl12-20030124 (09.02.2003)
[3] .NET
http://www.mspress.mircosoft.de/mspress/assets/leseproben/6441.pdf (03.01.2003)
[4] Google API
http://www.google.de/api (09.02.2003)
[5] WSFL – Spezifikation
http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf (09.02.2003)
[6] Übersicht der Technologien
http://www.webservicesarchitect.com/content/extras/Chapter10.pdf (09.02.2003)
[7] ebXML BPSS
http://www.ebxml.org/specs/ebBPSS.pdf (09.02.2003)
[8] XLANG
http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm (09.02.2003)
[9] BMPL
http://www.bpmi.org/ (09.02.2003)
[10] WSCI
http://www.w3.org/TR/wsci/ (09.02.2003)
[11] BPEL4WS
http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/ (09.02.2003)