Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

Bachelorarbeit Nr. 58

Datenverwaltung für unifizierte Service Komposition

Oliver Naumann

Studiengang: Informatik

Prüfer: Prof. Dr. Frank Leymann

Betreuerin: Dipl.-Inf. Katharina Görlach

begonnen am: 29.05.13

beendet am: 28.11.13

CR-Klassifikation: H.3.2, H.4.1

Kurzfassung

Diese Arbeit stellt ein Konzept zur Datenverwaltung innerhalb von Service Composition Engines auf Basis von Referenzen vor. Typischerweise werden von Geschäftsprozessen verwendete Daten in einer internen Datenbank gespeichert und by-value an Services übergeben. Dieses Vorgehen birgt vor allem bei der Verarbeitung von großen Datenmengen einige Nachteile. In dieser Arbeit wird diskutiert, welche Vorteile eine Auslagerung der Daten hat und wie diese realisiert werden kann. Basierend auf Erkenntnissen über die Datenverwaltung in bestehenden Service Composition Engines wird ein Konzept entwickelt, wie Variablen in eine externe Datenbank ausgelagert und anschließend durch Referenzen verwaltet werden können. Zentral dabei ist das Reference Resolution System (RRS). Das RRS ist eine Komponente, die als Schnittstelle zwischen der Datenbank und der Composition Engine fungiert und verantwortlich für das Speichern, Abrufen, Modifizieren und Löschen der gespeicherten Daten ist. Teil dieser Arbeit ist die Implementierung eines solchen RRS, das von einer bestehenden, grammatikbasierten Service Composition Engine eingebunden wird. Es werden Anforderungen und Erweiterungsmöglichkeiten eines RRS erörtert und kategorisiert. Die Korrektheit der Funktionalität und die Performanz des Systems wird anhand von Beispielen demonstriert.

Inhalt

1 Einleitung

Das Prinzip der serviceorientierten Architektur (im Folgenden SOA) findet bereits seit vielen Jahren vermehrt Einzug in die IT-Landschaften großer Unternehmen. Die sich verstärkende, weltweite Vernetzung von Firmen, die Auslagerung von IT-Dienstleistungen und der anhaltende Konkurrenzdruck durch das stetige Wachstum der Informatikbranche bringen neue Anforderungen mit sich: Die Konzerne müssen mehr als zuvor dazu in der Lage sein, schnell und flexibel auf Änderungen am Markt sowie gesetzliche Regulierungen zu reagieren [KO05]. In der Praxis werden Unternehmensprozesse oft automatisiert ausgeführt. Dabei muss deren Ablauf schnell und kostengünstig modifiziert werden können; neue Komponenten sollen ins Gesamtsystem integriert werden können, ohne dass dieses insgesamt großen Anpassungen unterzogen werden muss. Dies macht es nötig, dass einzelne Systemteile lose gekoppelt sind [WC05]. Die SOA schafft einen Rahmen zur Erfüllung dieser Anforderungen, indem Komponenten als Services modelliert und diese durch Dienstkompositionen (Service Compositions) orchestriert werden [JO08].

Um diese Dienstkompositionen automatisiert ablaufen zu lassen, müssen entsprechende Modelle in Sprachen überführt werden, deren Instanzen anschließend von sogenannten Service Composition Engines ausgeführt werden können. Hierfür wurden bereits zahlreiche Sprachen entwickelt, wie zum Beispiel die Web Services Business Process Execution Language (im Folgenden WS-BPEL) [JE07], das ebXML Business Process Specification Schema [UO01], YAWL [YA12] und ConDec [PA06].

[GL13] stellt einen Ansatz zur Unifizierung von Service Compositions vor, der formale Grammatiken nutzt, um verschiedene Konzepte solcher Sprachen zu vereinheitlichen. Diese formalen Grammatiken können anschließend von formalen Automaten ausgeführt werden. Die resultierende Service Composition Engine nutzt zur Verwaltung aller Daten, die zwischen den einzelnen Services ausgetauscht werden, ein Reference Resolution System, das jedes Datum extern speichert und den späteren Zugriff auf dieses per Referenz erlaubt.

In dieser Arbeit soll ein Konzept zur Verwaltung der Daten innerhalb der grammatikbasierten Service Composition Engine entwickelt werden. Hierbei soll ein Reference Resolution System (im Folgenden RRS) als ein Web-Service entworfen und implementiert werden. Eine ge-

eignete Datenbank soll an das RRS angebunden werden, in der von der Service Composition Engine erzeugte Variablen abgespeichert werden können.

Im Folgenden stellt die Arbeit vorerst die grammatikbasierte Service Composition Engine vor und erörtert bisherige Ansätze zur Datenverwaltung per Referenz im Umfeld der SOA. Kapitel 2 geht daraufhin kurz auf die zum Verständnis dieser Arbeit nötigen Grundlagen ein und beschreibt die Wahl der für das RRS verwendeten Datenbank. Anschließend diskutiert Kapitel 3 den Referenzbegriff und wie eine Verwaltung der Daten per Referenz möglich ist. Kapitel 4 stellt das RRS als Komponente konzeptionell vor und beschreibt deren Entwurf und Details der Implementierung. Anschließend soll Kapitel 5 die Funktionalität evaluieren, bevor Kapitel 6 mit einem Fazit über die gewonnenen Erkenntnisse und einem Ausblick über weitere mögliche Themen schließt.

1.1 Einordnung ins Gesamtsystem

In Abbildung 1 ist die Architektur der grammatikbasierten Service Composition Engine dargestellt:

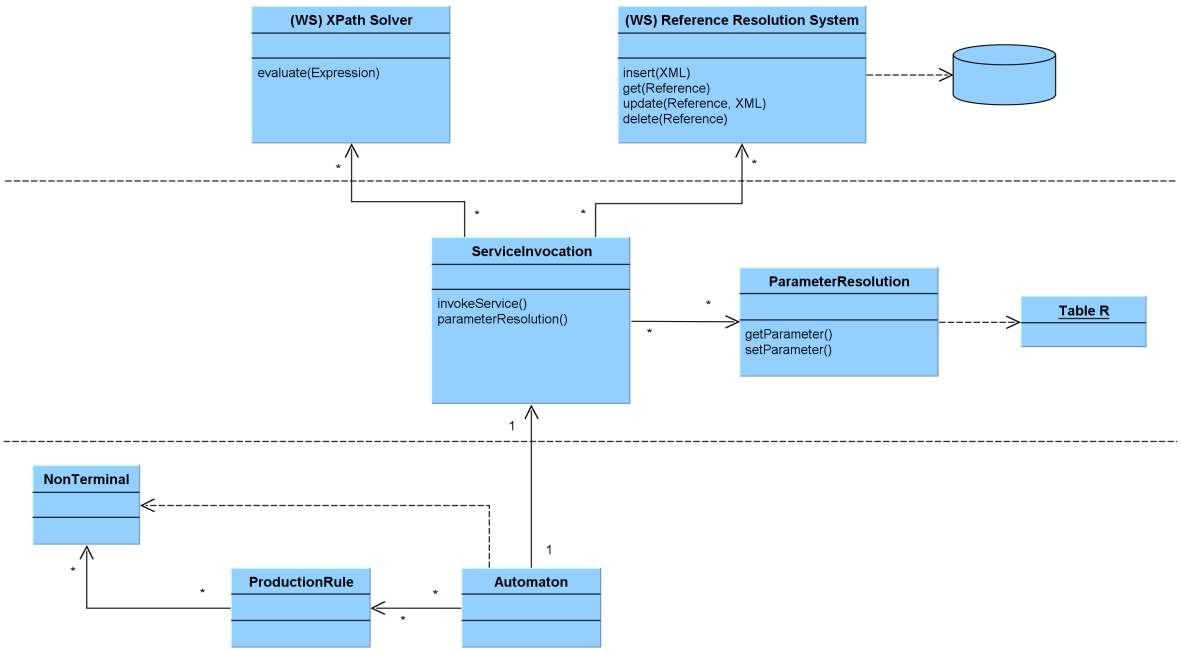

Abbildung 1: Architektur einer grammatikbasierten Engine für unifizierte Service-Komposition

Auf der untersten Ebene befindet sich die formale Grammatik, die den auszuführenden Geschäftsprozess beschreibt. Diese besteht aus Produktionsregeln, die Nichtterminale beinhalten, welche Aktivitäten innerhalb des Geschäftsprozesses kodieren, wie etwa den Aufruf eines

Services. Muss ein Service aufgerufen werden, ist hierfür die Komponente ServiceInvocation verantwortlich. Diese muss vor dem Aufruf zunächst die nötigen Parameter, die für den Aufruf des Services nötig sind, ermitteln. Hierzu schlägt die Komponente ParameterResolution in der Liste lokal gehaltener Referenzen nach und sucht die Referenz heraus, die auf die gesuchten Daten verweist. Mit dieser kann durch die Komponente ServiceInvocation nun ein Aufruf an das RRS ausgeführt werden, das die Referenz auflöst und die Parameterdaten zurückgibt, die anschließend zum Aufrufen des ursprünglichen Services verwendet werden können. Analog kann die Service Composition Engine auch neue Daten in das RRS einfügen oder bestehende Variablen löschen oder modifizieren. Kapitel 3.3 geht genauer auf die Verwaltung von Daten und Referenzen innerhalb der grammatikbasierten Composition Engine ein. Um eine lose Kopplung zu gewährleisten, wird das RRS als ein Web-Service bereitgestellt. Details hinsichtlich dieser architektonischen Wahl werden in Kapitel 4 diskutiert.

1.2 Verwandte Arbeiten

Es existieren verschiedene Ansätze für eine erweiterte Datenverwaltung in Service Composition Engines. Davon kommt das in [WG09] diskutierte Konzept dem hier vorgestellten Ansatz am nächsten. Dort werden Daten ebenfalls von einem RRS verwaltet, auf die per Referenzen, die als Endpoint References (siehe [WW06]) repräsentiert werden, zugegriffen werden kann. Allerdings verwendet dabei nicht nur die Service Composition Engine das RRS, sondern es werden Konzepte vorgestellt, mit denen auch aufzurufende Web-Services Referenzen auflösen können. Ferner werden verschiedene Benutzer- und Prozessrechte diskutiert, die die Sicherheit bei der Verwaltung der Referenzen erhöhen.

[WG09] stellt lediglich ein Konzept zur Datenverwaltung vor, jedoch keinen Prototypen. Diese Bachelorarbeit kann daher in vielerlei Hinsicht als eine Hinführung zu einer technischen Realisierung des vorgeschlagenen Konzepts angesehen werden. Es wird eine Implementierung vorgestellt, die als Basis eines komplexeren System fungieren kann und anhand derer evaluiert werden kann, ob die Herangehensweise an das Problem der Datenverwaltung mithilfe von Referenzen generell erfolgversprechend ist.

Andere Ansätze verwenden ebenfalls Datenbanken, um Daten zu externalisieren. Ein Überblick über bestehende Konzepte, in denen beispielsweise SQL von Geschäftsprozessen genutzt wird, um auf eine Datenbank zuzugreifen, wird in [VS08] gegeben.

2 Grundlagen

Im Folgenden werden einige grundlegende Begriffe erläutert, die zum Verständnis dieser Arbeit relevant sind. Kapitel 2.1 widmet sich vorerst der Spezifikation der Extensible Markup Language als ein wichtiges Werkzeug in der serviceorientierten Architektur. In Kapitel 2.2 wird daraufhin das Konzept von Web-Services und dessen Möglichkeiten der Realisierung vorgestellt. Kapitel 2.3 geht schließlich auf das XML-basierte Datenbanksystem BaseX ein, das in dieser Arbeit als Datenspeicher für das RRS Verwendung findet.

2.1 Extensible Markup Language (XML)

Die Extensible Markup Language (kurz: XML) ist eine plattformunabhängige Sprache, auf deren Basis neue Sprachen definiert werden können. Ausprägungen dieser Sprachen sind Dokumente, die einer bestimmten, zuvor meist mit XML Schema festgelegten Struktur folgen [BE99]. Mit XML ist es möglich, Daten in nicht-binärer, menschenlesbarer Form abzuspeichern. Dadurch und dank ihres standardisierten Formats, das vom World Wide Web Consortium spezifiziert wird, eignet sie sich insbesondere auch zur Kommunikation zwischen Systemen unterschiedlicher Plattformen. Dies macht XML zu einer geeigneten Grundlage für den Datenaustausch zwischen Web-Services [WC05].

Zur Definition neuer XML-basierter Sprachen wird heutzutage meist XML Schema eingesetzt. XML Schema spezifiziert die Art und Weise, wie Dokumente strukturiert und Datentypen definiert werden können [WC05]. In XML-Schema-Dokumenten werden sogenannte XML-Schema-Definitionen (kurz: XSDs) erstellt. Diese bestimmen, welche Form XML-Dokumente haben müssen, um als valide Instanzen der zugehörigen Schemata zu gelten. Ein XML-Dokument, für das ein Schema definiert ist und das den Anforderungen dieses Schemas genügt, wird valide genannt; ein XML-Dokument, das die Vorgaben der XML-Spezifikation erfüllt, nennt man wohlgeformt [WW08]. Ein XML-Dokument kann also zwar wohlgeformt aber dennoch nicht valide in Bezug auf ein bestimmtes Schema sein.

Listing 1 zeigt beispielhaft eine XML-Schema-Definition, Listing 2 eine dazugehörige, valide XML-Instanz.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Wurzel">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Kind1" type="xs:string"/>
        <xs:element name="Kind2">
          <xs:complexType>
            <xs:sequence>
             <xs:element name=“Datum“ type=“xs:date“/>
            </xs:sequence>
            <xs:attribute name=“ID“ type=“xs:integer“/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 1: Beispielhafte XML-Schema-Definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<Wurzel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="schema.xsd">

  <Kind1>Erstes Kindelement</Kind1>
  <Kind2 ID=“2“>
    <Datum>2000-08-17</Datum>
  </Kind2>

  <!-- Dies ist ein Kommentar und wird ignoriert. -->

</Wurzel>
```

Listing 2: XML-Instanz, die dem Schema aus Listing 1 folgt.

Zur Strukturierung der erzeugten Sprachen stehen verschiedene Konzepte zur Verfügung. Text, der diese Konzepte repräsentiert, wird Markup genannt [WW08]. Jeglicher Text, der kein Markup ist, heißt im Folgenden Zeichendaten. Die grundlegenden Markup-Bausteine vom XML werden Elemente genannt. Elemente können Attribute und weitere Kindelemente sowie Zeichendaten enthalten. Ein Beispiel hierfür ist das Element <Kind1> in Listing 2, das die Zeichendaten „Erstes Kindelement“ enthält. Durch die Verschachtelung von Elementen entstehen dabei baumartige Strukturen. Jedes XML-Dokument hat genau ein Element, das alle anderen enthält, das sogenannte Wurzelelement (engl. „root element“ bzw. „document element“) [GD05]. In obigem Beispiel ist dies das Element mit dem Namen <Wurzel>.

Elemente werden durch spitze Klammern dargestellt. Besitzt ein Element weitere Kinder, muss es in ein Start- und ein Endelement aufgeteilt werden, die zusammen alle Kinder umfassen. Attributen wird innerhalb des Startelements in Anführungszeichen ein Wert zugewiesen (siehe beispielsweise das Attribut ID des Elements <Kind2> in Listing 2). Ferner können Kommentare innerhalb des Dokuments verwendet werden, die bei der Verarbeitung ignoriert werden (siehe Listing 2) [WW08].

Insbesondere in großen Projekten kann es schnell dazu kommen, dass viele verschiedene XML-Sprachen definiert werden. Die Sprachen können miteinander kombiniert werden, wodurch es unter den Strukturbausteinen jedoch zu Namenskonflikten kommen [WC05]. Deshalb existiert in XML die Möglichkeit, Namensräume (engl. „namespaces“) festzulegen, die die Gültigkeitsbereiche von Namen bestimmen [WW08]. Listing 1 nutzt beispielsweise den Namensraum „xs“, der als die URL „http://www.w3.org/2001/XMLSchema“ definiert ist.

Im Rahmen dieser Arbeit wird außerdem ein weiteres Konstrukt verwendet: Enthalten die Elemente oder Attribute Zeichendaten, die vom Parser als Markup interpretiert werden (wie etwa '<' oder '>'), würde dies zu Fehlern in der Verarbeitung führen. Um diesen Konflikt aufzulösen, existieren sogenannte CDATA-Bereiche, innerhalb derer jegliche Zeichen als Zeichendaten aufgefasst werden [WW08]. Listing 3 zeigt die Verwendung von CDATA in einem solchen Fall. Das Element <Element1> enthält hierbei einen CDATA-Bereich – die darin enthaltenen spitzen Klammern markieren somit kein Markup, sondern werden als Zeichendaten interpretiert.

```
<Element1>
  <![CDATA[Dies ist kein Markup: <a>B</a>]]>
</Element1>
```

Listing 3: Verwendung von CDATA, um Markup als Zeichendaten zu interpretieren.

2.2 Web-Services und JAX-WS

Als einen Web-Service bezeichnet man ein Softwaresystem, das für eine direkte Maschine-zu-Maschine-Kommunikation innerhalb eines Netzwerks bereitgestellt wird. Ein Web-Service besitzt eine Schnittstelle, oft im XML-Format, die bestimmt, auf welche Art der Web-Service aufgerufen werden kann [WW04].

Web-Services spielen oftmals eine zentrale Rolle innerhalb der SOA. In dieser Architektur werden alle Softwarekomponenten als Services modelliert. Diese können dann lose gekoppelt auf Schnittstellenebene orchestriert werden, was das entstehende System flexibel und robust macht [GD05]. Hierbei ist anzumerken, dass diese Services nicht unbedingt Web-Services sein müssen – die SOA macht keinerlei Aussagen über die technische Realisierung.

Web-Services als gängige Realisierung der Services in der SOA können auf verschiedene Arten implementiert werden. Zwei der verbreitetsten Möglichkeiten sind SOAP-Web-Services und RESTful-Web-Servies [PZ08]. Der in dieser Arbeit entwickelte Web-Service wird als ein

SOAP-Web-Service implementiert. Diese Art von Web-Service nutzt eine XML-gestützte Repräsentation der ausgetauschten Nachrichten, genannt SOAP-Nachrichten. Die Schnittstelle eines solchen Web-Services wird ebenfalls in einem XML-Format beschrieben, der Web Services Description Language (WSDL) [WW04].

Durch die Verwendung von Web-Services ergeben sich vor allem folgende Vorteile:

- Plattformunabhängigkeit: Dank der standardisierten Schnittstelle kann der entstandene Web-Service von allen Systemen genutzt werden, die das SOAP-Nachrichtenprotokoll unterstützen.

- Lose Kopplung: Das System macht sich nur durch seine Schnittstelle bekannt, Details der Implementierung und des internen Datenflusses bleiben transparent.

Die SOAP-Nachrichten enthalten alle für den Aufruf nötigen Daten, insbesondere die Parameter für den Web-Service-Aufruf bzw. den Rückgabewert im Falle der Antwortnachricht. In der WSDL-Schnittstellenbeschreibung können ferner Fehler, sogenannte Faults, definiert werden, die im Falle einer Ausnahme während des Aufrufs in der SOAP-Nachricht zurückgegeben werden [WC05].

Ein weiterer Vorteil von Web-Services ist der standardisierte Verzeichnisdienst Universal Description, Discovery and Integration (kurz: UDDI). Dieser bietet die Möglichkeit, Web-Services mit bestimmten Eigenschaften innerhalb von Netzwerken dynamisch aufzufinden („Find“), sofern diese zuvor registriert wurden („Publish“) [NE02]. Abbildung 2 stellt dies schematisch dar.

Vor allem wenn ein Web-Service dynamisch gebunden wird, ist es manchmal wichtig, nicht nur zu wissen, welche Schnittstelle der Web-Service bietet und wie er syntaktisch aufzurufen ist, sondern auch, wie er sich verhält und unter welchen Rahmenbedingungen ein Aufruf möglich ist. Diese Klasse an Eigenschaften wird unter dem Begriff der nichtfunktionalen Eigenschaften zusammengefasst, die beispielsweise Sicherheitsaspekte beinhalten oder Randbedingungen angeben, die bei einem Aufruf des Services einzuhalten sind. Web-Services unterstützen eine Feststellung dieser Eigenschaften durch die ebenfalls XML-basierte Spezifikation WS-Policy [WC05].

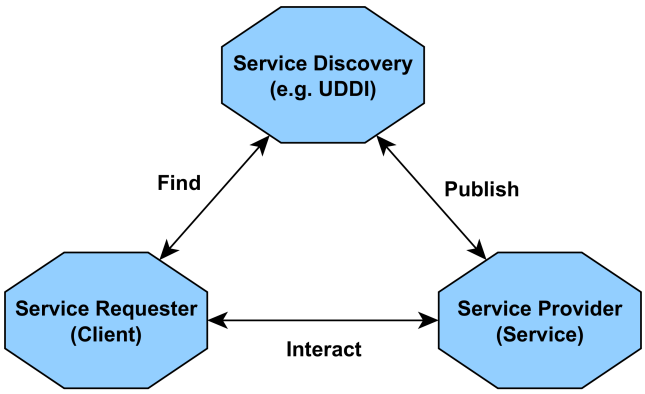

Abbildung 2: Service-Dreieck nach [WW04].

Um den Web-Service im Rahmen dieser Arbeit zu entwickeln, wird auf die Java API for XML Web Services, kurz JAX-WS, zurückgegriffen [EM06]. Die exakte Definition der Web-Service-Schnittstelle mit WSDL und die Implementierung der Verarbeitung von SOAP-Nachrichten ist eine technisch komplexe Aufgabe. JAX-WS bietet dagegen die Möglichkeit, Web-Services direkt aus Java-Code zu generieren, und erlaubt so die Entwicklung eines Web-Services auf abstrakterer Ebene. In einem Bottom-Up-Ansatz werden hierbei erst gewöhnliche Java-Klassen und Methoden erstellt und anschließend durch Annotationen ergänzt. Die so markierten Einheiten können daraufhin exportiert werden, wobei die WSDL-Beschreibung für den Web-Service automatisch erzeugt wird. Zusätzliche Informationen werden durch Deployment-Deskriptoren zur Verfügung gestellt. Der Web-Service kann dann auf dem gewünschten Server bereitgestellt werden.

2.3 Das XML-basierte Datenbanksystem BaseX

Wie in Kapitel 2.2 beschrieben ist XML ein wichtiges Konzept vieler Web-Services, da es sowohl für die Beschreibung der Schnittstelle als auch, im Falle von SOAP-Web-Services, für die Strukturierung der ausgetauschten Nachrichten eingesetzt wird. Um ein möglichst einheitliches Umfeld zur Ausführung von Geschäftsprozessen zu schaffen, bietet es sich daher an, von Service Composition Engines verwendete Daten ebenfalls in Form von XML abzuspeichern, was in der Praxis das gängige Vorgehen ist. Innerhalb dieser Arbeit soll zum Speichern solcher Daten eine Datenbank verwendet werden. Dies macht die Suche nach einem geeigneten Datenbanksystem notwendig, in der die Daten zuverlässig verwaltet werden können.

Bereits früh nach der ersten Spezifikation von XML stellte sich die Frage, in welcher Form Daten im XML-Format abgespeichert werden können, um eine effiziente Verarbeitung und

schnelle Datenbankanfragen zu gewährleisten. Darunter fallen unter anderem Ansätze, XML in relationalen Datenbanken abzuspeichern. Beispielhaft wurde die Performanz eines solchen Vorgehens etwa in [FK99] untersucht. Während dort sowohl das Speichern von Daten als auch Anfragen an die Datenbank schnell ausgeführt werden konnten, bestand dennoch das Problem, dass die Rekonstruktion ganzer XML-Dokumente aus der in der Datenbank vorhandenen Repräsentation der Daten zu viel Zeit in Anspruch nahm.

Ein anderer Ansatz ist das Ablegen von XML-Daten in Speichern, die XML nativ unterstützen, wie zum Beispiel in [KM99]. Diesem Ansatz folgt auch die quelloffene Datenbankengine BaseX, die ab 2007 aus der Arbeit der Database and Information Systems Group der Universität Konstanz hervor ging[1]. Für diese Arbeit wird eine Datenbank auf Basis der BaseX-Engine zum Ablegen der Daten verwendet. BaseX bietet im Vergleich zu anderen XML-Datenbanken eine gute Performanz beim Ausführen von Anfragen [GR10]. Sie unterstützt transaktionale Operationen nach dem ACID-Paradigma [GR81]. Die Einrichtung eines Servers ist binnen weniger Minuten möglich, was das System sehr flexibel macht und die Einbindung der Datenbank in das Gesamtsystem erleichtert. Ferner stellt BaseX bereits einen frei verwendbaren Client-Code in Java zur Verfügung, mit dem die Integration in das RRS ohne Anpassungen möglich ist. Die Verwendung einer XML-Datenbank hat den Vorteil, dass die Daten, die das RRS in Form von XML erhält, nicht transformiert werden müssen, sondern direkt an die Datenbank weitergegeben werden können. Implizit findet dabei eine Prüfung dieser Daten statt, sodass nur wohlgeformtes XML der Datenbank hinzugefügt werden kann. BaseX bietet außerdem eine grafische Benutzeroberfläche, die einen visuellen Eindruck vom aktuellen Zustand der Datenbank gewährt und zur Administration eingesetzt werden kann [GH07].

1 „BaseX | The XML Database“, http://www.basex.org, aufgerufen am 03.10.2013

3 Referenzen

In diesem Kapitel soll das Konzept einer Referenz im Kontext der Datenverwaltung innerhalb von Service Composition Engines erläutert und dessen Notwendigkeit dargelegt werden. Auf dem Gebiet der Informatik wird der Begriff Referenz oftmals mit dem programmiersprachlichen Element eines Zeigers gleichgesetzt oder verglichen [ST10]. Zeiger sind in diesem Zusammenhang Datentypen, die auf Speicheradressen zeigen [HE01]. Für den Einsatz innerhalb dieser Arbeit ist dieser Begriff allerdings unpassend. Stattdessen wird der folgende Referenzbegriff verwendet:

> **Definition (Referenz):**
> Eine Referenz ist ein wohlgeformtes, XML-basiertes Konstrukt, das indirekt und eindeutig auf eine in einem externen Speicher abgelegte Variable verweist. Es enthält alle zur Auffindung der Variablen benötigten Informationen.

Die enthaltenen Information, die zur Auffindung der Variablen nötig sind, können vielfältig sein und unterscheiden sich je nach verwendetem Datenspeicher. Werden die Variablen beispielsweise in einer einfachen Textdatei gespeichert, könnten diese Informationen den Pfad zur Datei und deren Namen beinhalten. Wird dagegen eine Datenbank als Speicher verwendet, muss die Referenz den Hostnamen und weitere Verbindungsinformationen wie den Namen der Datenbank und das Passwort enthalten. Ferner muss die Referenz, unabhängig vom gewählten Datenspeicher, auch einen Identifikator enthalten, der die Variable im Gültigkeitsbereich des Speichers eindeutig bestimmt. Andernfalls könnten einzelne Variablen innerhalb des Datenspeichers nicht adressiert werden. Diese Information wird in dieser Arbeit als Name der Variablen bezeichnet.

Im Rahmen dieser Arbeit wird als Speicher eine Datenbank auf Basis des in Kapitel 2.3 vorgestellten Datenbanksystems BaseX verwendet. Entsprechende Referenzen enthalten den Host, auf dem die Datenbank erreichbar ist, den Namen der Datenbank, den Benutzernamen und das Passwort zur Authentifizierung, den Port, durch den auf die Datenbank zugegriffen werden kann sowie den oben angesprochenen Namen der referenzierten Variablen.

In Kapitel 3.1 soll vorerst dargestellt werden, in welcher Form Daten für gewöhnlich in Ser-

vice Composition Engines verwaltet werden. Anschließend werden in Kapitel 3.2 Vorteile einer alternativen Datenverwaltung mit Referenzen aufgezeigt. Kapitel 3.3 geht auf die spezifische Verwaltung der Referenzen innerhalb der in dieser Arbeit verwendeten grammatikbasierten Service Composition Engine ein, während Kapitel 3.4 schließlich den Aufbau der Referenzen detailliert beschreibt.

3.1 Datenverwaltung in Service Composition Engines

Geschäftsprozesse sind aufgrund der Tatsache, dass sie Kompositionen verschiedener Services modellieren, in den meisten Fällen zustandsbehaftet, um den Datenaustausch zwischen einzelnen Komponenten möglich zu machen [HA05]. Service Composition Engines müssen daher eine Möglichkeit zur Verfügung stellen, Daten während der Ausführung von Prozessen zu speichern und bei Bedarf abzurufen. Außerdem muss auch die Sprache, in der die von der Service Composition Engine ausgeführte Instanz formuliert wurde, über Konstrukte verfügen, die das Halten eines Zustands unterstützen [JE07].

In den meisten Fällen werden prozessrelevante Daten intern in der Composition Engine gespeichert und verwaltet. Sie liegen in expliziter Repräsentation vor und können so direkt adressiert werden. Häufig werden für diesen Zweck interne Datenbanken verwendet. Ein bekannter Vertreter hierfür ist zum Beispiel die quelloffene Composition Engine Apache ODE. Diese erlaubt es zwar auch, Daten per WS-BPEL-Erweiterung auf externen Speichern abzulegen, allerdings ist dies nur für Datenquellen möglich, die von der Java Database Connectivity [RE00] unterstützt werden, und auch nur für die standardmäßigen SQL-Datentypen[2].

Ebenfalls eine interne Datenbank benötigt die Workflow Engine jBPM, die Geschäftsprozesse auf Basis von Business Process Model and Notation (BPMN) ausführt. Die Nutzung einer externen Datenbank wird hierbei nicht ermöglicht, stattdessen setzt die Engine eine lokale Instanz der verwendeten Datenbank voraus[3]. Selbiges gilt für für WS-BPEL Engine OW2 Orchestra, die alle Daten ebenfalls in in einer vorkonfigurierten Datenbank ablegt[4].

Einen ähnlichen Ansatz verfolgt beispielsweise auch YAWL und deren zugehörige Composition Engine. In dieser werden alle Daten als XML-Dokumente repräsentiert. Einzelne Datenelemente werden dann innerhalb von Variablen abgelegt, die in verschiedenen Ausprägungen

2 „Apache ODE“, http://ode.apache.org/index.html, abgerufen am 23.11.2013
3 „jBPM – Jboss Community“, http://www.jboss.org/jbpm, abgerufen am 24.11.2013
4 „Orchestra: The Open Source BPEL Solution“, http://orchestra.ow2.org, abgerufen am 24.11.2013

(beispielsweise Input- und Output-Variablen oder lokale Variablen) existieren und einem strengen Typsystem auf Basis von XML Schema folgen. Die Daten werden in einer internen Datenbank abgelegt – standardmäßig in einer PostgreSQL-Datenbank. Eine Rekonfiguration zur Nutzung anderer Datenbankverwaltungssysteme ist jedoch möglich. Datentransfer und -manipulation erfolgen direkt mit XQuery [YA12].

3.2 Vorteile einer referenzbasierten Datenverwaltung

Die in Kapitel 3.1 dargestellte Datenverwaltung hat einige Nachteile, die vor allem zum Tragen kommen, wenn die Web-Services sehr viele Daten untereinander austauschen. Dies ist vor allem der Fall in Domänen, in denen Kalkulationen auf großen Datenmengen stattfinden; ein Beispiel hierfür sind wissenschaftliche Arbeitsabläufe, die etwa detaillierte Simulationen beinhalten [BC08].

Werden in solchen Anwendungsbereichen Geschäftsprozesse zur Orchestrierung von Web-Services verwendet, wird es zur Aufgabe der Service Composition Engine, diese großen Datenmengen zu verwalten. Dies kann zu Problemen wie Performanz-Einbrüchen oder Speicherknappheit führen, wenn die Engine nicht zur Verwaltung von Daten solcher Größe ausgelegt ist, insbesondere wenn die Menge an Daten sich in einer Größenordnung befindet, in der die Ressourcen nicht zentral, sondern verteilt abgespeichert werden müssen [FU01]. Ferner ist die Service Composition Engine dafür verantwortlich, dass diese Datenmengen an die einzelnen Web-Services weitergegeben werden, was je nach Netzwerk zu Engpässen durch zu geringe Bandbreite führen kann und so gegebenenfalls die Ausführungszeit stark verlängert. Ebenso können sich durch die zentrale Verwaltung der expliziten Daten Einbußen in der Wartbarkeit und Benutzerfreundlichkeit ergeben. Dies ist vor allem bei Variablen mit komplexen Typen der Fall, deren Darstellung die Dokumente, die die Ausführungslogik beinhalten, aufblähen und unübersichtlich machen kann.

Stattdessen schlägt diese Arbeit eine Verwaltung der Daten durch Referenzen vor. Hierbei sind zwei Vorgehensweisen zu unterscheiden:

- (1) Die Referenzen werden lediglich von der Service Composition Engine genutzt und verwaltet, die aufgerufenen Web-Services selbst bleiben unverändert. Dieser Ansatz genießt die Vorteile der Datenauslagerung, macht es aber dennoch nötig, dass die Referenzen schon vor dem Aufruf eines Web-Services aufgelöst werden; damit müssen

die für den Web-Service nötigen Parameter noch immer by-value übertragen werden.

- (2) Sowohl die Service Composition Engine als auch die aufgerufenen Web-Services werden so angepasst, dass sie mit Referenzen umgehen und diese auflösen können. Dies erlaubt sowohl die Auslagerung der Daten als auch eine effiziente Parameterübergabe an die Web-Services, da nun nicht mehr die eigentlichen Daten selbst, sondern lediglich die Referenzen auf den tatsächlichen Speicherort übergeben werden müssen.

Im Rahmen dieser Arbeit wird lediglich der Ansatz nach (1) verfolgt. Dieser kann aber durchaus als Vorarbeit für Ansatz (2) angesehen werden, da die Service Composition Engine in jedem Fall für das Management von Referenzen angepasst werden muss. Die Verwaltung und Auflösung der Referenzen übernimmt hierbei das bereits erwähnte RRS. Der Entwurf des RRS als Web-Service macht dieses als lose gekoppelte Komponente flexibel genug, dass es kein fester Bestandteil der Service Composition Engine ist, sondern als eigenständige Instanz fungiert. So ist später die Möglichkeit gegeben, dass es auch von den am Geschäftsprozess beteiligten Services zur Auflösung der Referenzen benutzbar ist und ein einzelnes RRS von mehreren Prozessinstanzen verwendet werden kann.

Sollen nach Ansatz (2) auch die Web-Services mit den Referenzen umgehen können, müssen diese entsprechend angepasst werden. Als Möglichkeit bietet sich hier zum einen an, die Web-Services zu modifizieren, dass diese selbstständig Referenzen auflösen und so an benötigte Daten gelangen. Ein anderer Ansatz ist das Umschließen der Web-Services mit einem Wrapper, dessen Aufgabe lediglich das Auflösen der Referenzen und die anschließende Weitergabe der Daten an den unmodifizierten Web-Service ist. [WG09] geht detailliert auf eine Lösung der Problematik nach Ansatz (2) ein, in der eine Erweiterung der Sprache WS-BPEL um Referenzen und eine Transformation der Web-Services durch Wrapper vorgeschlagen wird.

Vorteile einer Datenverwaltung nach (1) sind:

- Durch die Entwicklung des Referenzen-Konzepts, eines RRS und einer referenzenunterstützenden Service Composition Engine wird eine erweiterbare Basis für eine durch Referenzen unterstützte Datenverwaltung für die gesamte Umgebung der Geschäftsprozesse inklusive der Web-Services gelegt.

- Es wird eine simple Repräsentation der Daten innerhalb der Service Composition En-

gine geschaffen. Ein einzelnes Datum wird nicht mehr durch eine (eventuell komplexe) XML-Instanz dargestellt, sondern durch einen einfachen Namen (siehe auch Kapitel 3.3 zur Verwaltung der Referenzen in der Service Composition Engine).

- Der Speicherbedarf der Service Composition Engine wird reduziert, da Daten ausgelagert werden können und immer nur die Parameter für den aktuell aufzurufenden Web-Service geladen werden müssen. Dies macht gegebenenfalls die Ausführung von großen, datenintensiven Geschäftsprozessen überhaupt erst möglich.

3.3 Verwaltung der Referenzen

In diesem Kapitel wird beschrieben, wie Daten und Referenzen in der in Kapitel 1.1 vorgestellten grammatikbasierten Service Composition Engine verwaltet werden. In dieser Engine werden von formalen Automaten Modelle ausgeführt, die auf formalen Grammatiken basieren. Um eine eindeutige Terminologie zu schaffen, müssen zunächst zwei Begriffe voneinander abgegrenzt werden. Auf Referenzen wurde oben bereits eingegangen; ferner verwendet die grammatikbasierte Service Composition Engine jedoch auch Bezeichner.

Definition (Bezeichner):

Ein Bezeichner ist ein innerhalb eines Geschäftsprozesses eindeutiger Name, der auf eine Referenz abgebildet werden kann.

Diese Bezeichner sind die Namen, anhand derer Variablen innerhalb der formalen Grammatik definiert und identifiziert werden. Die Abbildung der Bezeichner auf tatsächliche Referenzen (und im Anschluss deren Abbildung auf das Datum selbst) geschieht erst zur Laufzeit. Anstatt also Referenzen direkt zu verwenden, wird auf eine Referenz durch ihren Bezeichner verwiesen, dessen Name konstant und eindeutig in der Grammatik festgehalten wird. Die Bezeichner sind notwendig, da die Referenzen unter Umständen sehr groß werden können und eine Verwendung von ihnen innerhalb der Grammatik damit unhandlich wäre. Zudem ist a priori die genaue Gestalt einer Referenz meist nicht bekannt, weshalb diese nicht als Verweis auf die zugehörige Variable in der Grammatik verwendet werden kann. Abbildung 3 stellt das Zusammenspiel zwischen Bezeichnern und Referenzen bei Aufruf eines Services schematisch dar.

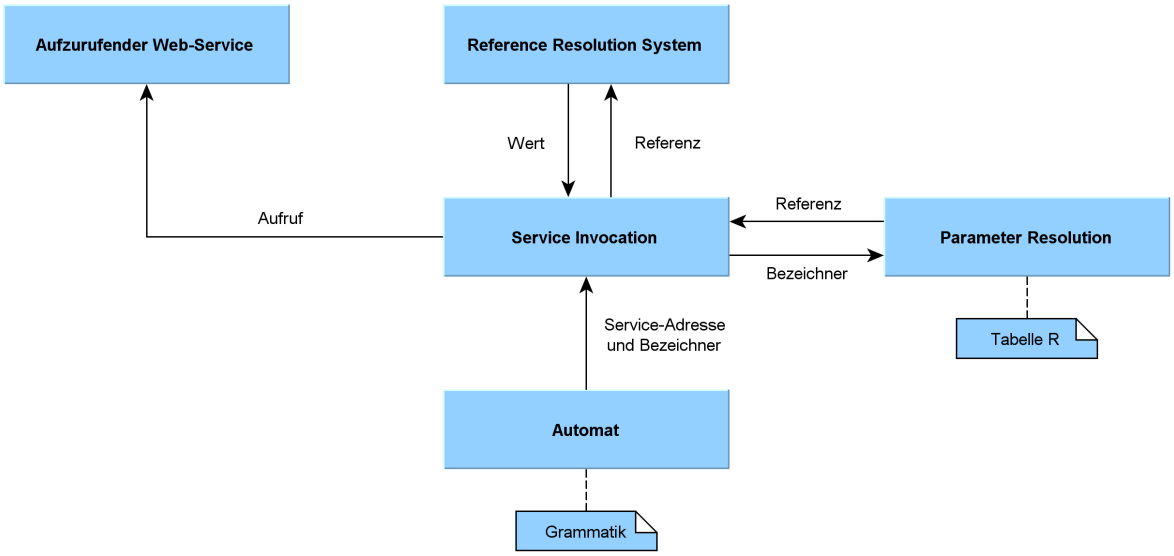

Abbildung 3: Grundlegender Datenfluss beim Aufruf eines Web-Services durch die grammatikbasierte Service Composition Engine.

Da die Bezeichner nur innerhalb eines Geschäftsprozesses eindeutig sind, besitzen sie lediglich lokale Gültigkeit (im Gegensatz zu Referenzen, die zu Beginn von Kapitel 3 als global eindeutig definiert wurden). Eine Liste aller Bezeichner, denen eine Referenz zugeordnet wurde, ist die Tabelle R. Diese enthält die Abbildung von Bezeichnern auf Referenzen und wird zur Auflösung genutzt, wenn auf eine bestimmte Referenz zugegriffen werden soll. Die Tabelle R ist Teil der Service Composition Engine. Jede Prozessinstanz verwaltet dabei ihre eigene Tabelle und ist selbst dafür verantwortlich, die Einträge für verwendete Bezeichner zu generieren und später wieder zu löschen. Abbildung 4 zeigt den schematischen Aufbau der Tabelle R mit möglichen Inhalten.

Bezeichner	Referenz
X	<reference><storageLocation>...</storageLocation><variableName>e17h1</variableName></reference>
Y	<reference><storageLocation>...</storageLocation><variableName>aew5g</variableName></reference>
Z	<reference><storageLocation>...</storageLocation><variableName>jw8ihz</variableName></reference>
...	...

Abbildung 4: Aufbau und mögliche Ausprägung der Tabelle R.

Bevor ein Bezeichner auf eine Referenz abgebildet werden kann, muss die entsprechende Variable zuvor initialisiert worden sein. Andernfalls existiert in der Tabelle R keine Referenz für den Bezeichner und er kann nicht aufgelöst werden können. Entweder muss also die Prozessinstanz eine zugehörige Referenz aus einer externen Quelle erhalten haben, oder sie fügt die

Variable mit den gewünschten Initialwert selbst in das RRS ein. Wird ein neuer Wert in das RRS eingefügt, liefert dieses eine global eindeutige Referenz zurück, mit der der gespeicherte Wert später wieder abgerufen werden kann. Diese Referenz kann nun in der Zeile des entsprechenden Bezeichners eingefügt werden. Wird später in der Grammatik erneut auf den Bezeichner verwiesen (zum Beispiel weil die damit identifizierte Variable einen neuen Wert erhalten soll), kann die Prozessinstanz in der Tabelle den Bezeichner nachschlagen und erhält so die Referenz, die auf die zu manipulierende Variable verweist. Diese kann dann an das RRS zur Auflösung weitergegeben werden.

Wird ein Bezeichner nicht mehr benötigt, kann er zusammen mit seiner Referenz aus der Tabelle R entfernt werden. Es ist dabei Aufgabe der Service Composition Engine selbst, dem RRS mitzuteilen, ob dabei die dort gespeicherte Variable gelöscht werden soll, da das RRS keine Informationen darüber hat, ob und wo die Variable später noch verwendet wird.

3.3.1 Auflösen von Referenzen beim Aufruf eines Web-Services

In der grammatikbasierten Service Composition Engine sind die Modelle, die die auszuführenden Prozesse beschreiben, formale Grammatiken. In diesen formalen Grammatiken werden einzelne aufzurufende Services als Nichtterminale repräsentiert. Die Modelle werden von formalen Automaten ausgeführt. Trifft der Automat während der Ausführung auf ein Nichtterminal, muss der damit identifizierte Service aufgerufen werden. Dabei ergibt sich folgender Ablauf, der auch in Abbildung 5 dargestellt ist:

- Der Automat übergibt der Komponente Service Invocation die Adresse des aufzurufenden Web-Service (diese wurde zuvor in der Grammatik definiert) und eine Liste von Bezeichnern, die entweder Eingabe- oder Ausgabewerte markieren (1).

- Die Komponente Service Invocation löst die übergebenen Parameter auf, indem sie die Bezeichner an die Komponente Parameter Resolution übergibt (2). Dieser ist die Tabelle R zugeordnet, anhand derer die Parameter Resolution die Bezeichner auf Referenzen abbilden kann. Diese werden an die Komponente Service Invocation zurückgegeben (3).

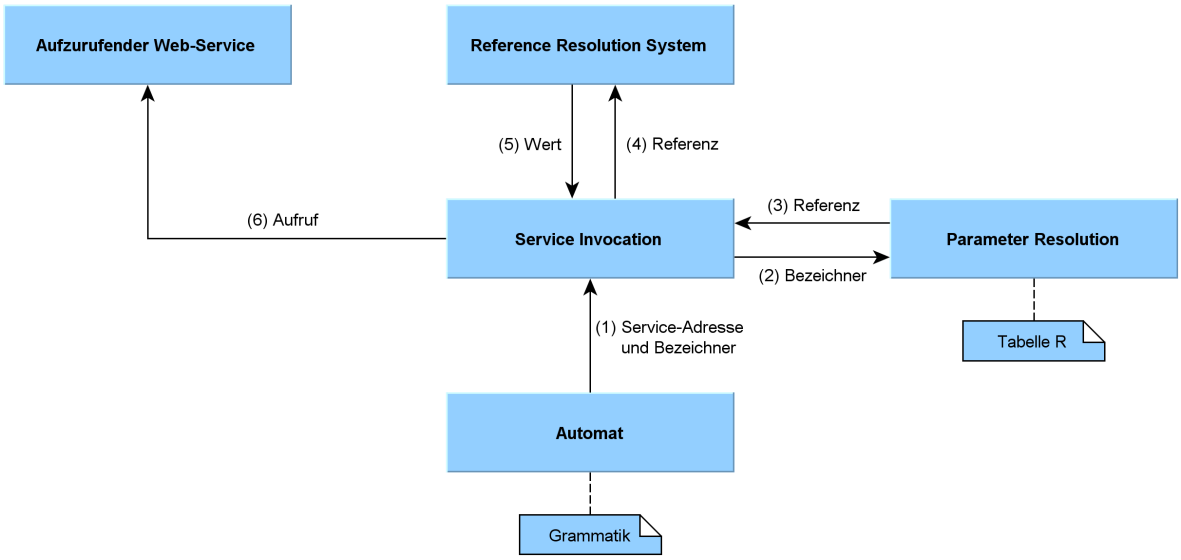

Abbildung 5: Ablaufdiagramm zum Aufruf eines Web-Services durch die grammatikbasierte Service Composition Engine.

- Die Komponente Service Invocation ruft für jeden Eingabewert das RRS auf und übergibt die zugehörige Referenz (4). Von RRS erhält die Service Invocation dann die Werte der Variablen, auf die die Referenzen verwiesen haben (5).

- Nun besitzt die Service Invocation die Adresse des Web-Services und alle nötigen Parameter und kann diesen aufrufen (6).

Besitzt der aufgerufene Web-Service Rückgabewerte, müssen diese zudem durch das RRS rückgespeichert werden. Die entsprechenden Bezeichner hat die Komponente Service Invocation zuvor in Schritt (1) bereits vom Automaten erhalten, sodass diese ebenfalls auf Referenzen abgebildet werden können, die nun an das RRS zusammen mit dem Wert für die Variable übergeben werden.

3.3.2 Zuordnung und Kardinalität

Bisher ungeklärt ist die Frage, woher der Service Composition Engine die Adresse des RRS bekannt ist, das zur Auflösung von Referenzen aufgerufen werden soll. In Abbildung 1 in Kapitel 1.1 wurde die Kardinalität zwischen der Service Composition Engine und dem RRS als n-zu-m-Beziehung dargestellt. Ausschlaggebender ist jedoch die Beziehung zwischen einzelnen Prozessinstanzen und dem RRS. In dem bisher bestehenden System ist einem Geschäftsprozess genau ein RRS zugeordnet. Die Adresse des RRS ist in der Grammatik des Prozesses enthalten. Dies hat den Vorteil, dass diese beliebig angepasst werden kann, ohne

dass der Code der Service Composition Engine verändert werden muss oder ein Redeployment dieser nötig ist.

Der Prozessinstanz ist damit die Adresse des RRS immer bekannt. Ein RRS kann gleichzeitig von mehreren Prozessinstanzen verwendet werden, was kein Problem darstellt, da das RRS selbst die Eindeutigkeit der Referenzen sicherstellt. Die Bezeichner müssen wie zuvor geklärt nur innerhalb der einzelnen Prozesse einzigartig sein.

Das Problem wird komplexer, wenn die Referenzen zwischen Prozessen ausgetauscht werden können. In diesem Fall darf ein Prozess, sobald er ein Datum nicht mehr benötigt, nicht einfach beim RRS dessen Löschung in Auftrag geben, da es möglicherweise von anderen Prozessen noch verwendet werden muss. Dieser Umstand ist im momentanen Gesamtsystem allerdings nicht vorgesehen. Stattdessen wird er als Gegenstand zukünftiger Arbeiten in Kapitel 6 nochmals aufgegriffen.

3.3.3 Gültigkeit von Referenzen

Wird ein neues Datum im RRS abgelegt, wird für dieses eine neue Referenz generiert. Somit ist für jede Variable in einem Datenspeicher zeitweilig mindestens eine Referenz im Umlauf, durch die auf die Variable zugegriffen werden kann. Wird die Referenz kopiert, können allerdings auch mehrere identische Referenzen auf diese Variable verweisen. Wird die Variable nun gelöscht, können die Referenzen nicht mehr aufgelöst werden. Dies ist ebenfalls nicht möglich, wenn beispielsweise der Datenspeicher nicht mehr erreichbar ist. Referenzen können somit ihre Gültigkeit verlieren. Im Rahmen dieser Arbeit wird diese folgendermaßen definiert:

> **Definition (Gültigkeit):**
> Eine Referenz ist gültig, solange sie vom Reference Resolution System aufgelöst werden kann und die Variable, auf die sie verweist, seit der Erstellung der Referenz nicht gelöscht wurde.

Eine Referenz ist also nicht mehr gültig, wenn die Variable, auf die sie verweist, nicht mehr vom RRS aufgefunden werden kann. Dies kann vielfältige Gründe haben – zum Beispiel kann sich das Schema der Referenzen mittlerweile geändert haben, sodass alte Referenzen nicht mehr valide im Bezug auf das im RRS hinterlegte Schema sind, oder das RRS kann keine Verbindung mehr zu einer zuvor benutzten Datenbank herstellen.

Die obige Definition schließt aber auch den Fall ein, dass eine Referenz erstellt, das zugehörige Datum aber gelöscht wird, während die Referenz noch im Umlauf ist. Es entsteht so eine Art Dangling Reference [EY93]. Dieser Fall ist besonders kritisch, wenn nach dem Löschen der Variablen eine neue Variable im selben Datenspeicher abgelegt wird, die zufällig oder systematisch den gleichen Variablennamen erhält. Dann verweist die alte Referenz nämlich auf ein neues Datum, das eventuell einen anderen Wert hat, als erwartet. Dies löst aber keinen Fehler aus, da die Referenz fehlerfrei aufgelöst werden kann. Eine Referenz darf deshalb nicht mehr gültig sein, sobald die zugrunde liegende Variable gelöscht wird. Mögliche Erkennungsstrategien für dieses Problem wären beispielsweise Varianten von Tombstones [LO75] oder das Anreichern von Variablen und Referenzen mit Versionierungsinformationen nach dem Locks-and-Keys-Prinzip [FL80] oder durch Speicherung des Zeitpunkts der Erstellung der Referenz bzw. Löschung der Variablen.

Diese Arbeit beschränkt sich darauf, dass Referenzen nicht zwischen Prozessen ausgetauscht werden. Es werden nur Daten ausgetauscht. Somit ist die Gültigkeit von Referenzen immer gewährleistet. Eine Erweiterung bezüglich dem Austausch von Referenzen müsste jedoch die Gültigkeit von Referenzen berücksichtigen.

3.4 Referenzen-Schema

Zu Beginn dieses Kapitels wurde bereits definiert, dass Referenzen alle für ihre Auflösung nötigen Informationen enthalten. Dies macht es nötig, dass eine geeignete Form gewählt wird, durch die diese Informationen repräsentiert werden. Das in [WG09] vorgeschlagene Schema für Referenzen sieht unter anderem als Möglichkeit zur Auflösung vor, die nötigen Datenbankabfragen (etwa um den Wert einer Variablen zu erhalten oder sie zu löschen) direkt in der Referenz zu speichern. Dies hat mehrere Nachteile:

- Lange und komplexe Queries können die Referenzen, deren Vorteil es unter anderem sein soll, möglichst simpel und klein zu sein, aufblähen. Zudem müssen mehrere verschiedene Queries gespeichert werden (für das Zuweisen von neuen Werten, das Löschen und das Abrufen des Werts von Variablen).

- Die Queries können von anderen Programmen ausgelesen und in veränderter Form wieder in der Referenz gespeichert werden, um damit das RRS (ob gewollt oder ungewollt) dazu zu bringen, schadhafte Operationen auf der Datenbank auszuführen („In-

jection“).

- Die Referenzen sind anfälliger dafür, nach gewisser Zeit ihre Gültigkeit zu verlieren, wenn beispielsweise das Layout der zugehörigen Datenbank verändert wird und die Queries dadurch nicht mehr funktionieren.

Stattdessen wird in dieser Arbeit ein anderer Ansatz verfolgt: Die zur Auflösung nötigen Informationen werden nicht in direkter Form von Queries gespeichert, sondern indem die verschiedenen Informationsfragmente einzeln angegeben werden. Für eine Datenbank können so zum Beispiel der Host, Datenbankname, Benutzername und Passwort mit der Referenz übergeben werden. Allein das RRS kennt nun den internen Aufbau der Datenbank, fügt die gegebenen Informationen dynamisch zur Laufzeit zu einem Query zusammen und führt dieses aus. Das RRS hat somit die alleinige Kontrolle darüber, welche Queries auf der Datenbank ausgeführt werden und kann bei einer Änderung des Datenbankschemas die Queries entsprechend anpassen, ohne dass die Referenzen zwingend geändert werden müssen.

Für die Kopplung des RRS an die XML-Datenbank wird in dieser Arbeit Aufruftechnik verwendet, das heißt, der Datenbankbefehl wird in einem Methodenaufruf direkt an das Datenbanksystem übermittelt. BaseX enthält hierzu bereits einen entsprechenden Client unter BSD-Lizenz[5]. Konzeptionell sind natürlich auch andere Formen wählbar, wie etwa Spracherweiterungen (vergleiche [HR01]) oder eine Anbindung mit Java Database Connectivity [RE00].

Indem lediglich Verbindungsinformationen statt vollständigen Queries angegeben werden, ergeben sich die folgenden Vorteile:

- Die Referenzen werden klein und übersichtlich gehalten und sind gut lesbar. Es müssen nicht mehrere Queries pro Referenz gespeichert werden, eine einmalige Angabe der Verbindungsinformationen genügt.

- Es können keine Datenbankoperationen injiziert werden, da die Queries vom RRS zur Laufzeit erzeugt werden.

- Die Referenzen werden robuster gegenüber Änderungen des Datenbanklayouts, da bei einer solchen nun nicht mehr alle Referenzen, sondern lediglich die Query-Erzeugung

5 „Java client for BaseX“, https://github.com/BaseXdb/basex-examples/blob/master/src/main/java/org/basex/examples/api/BaseXClient.java, aufgerufen am 29.10.2013

im RRS angepasst werden muss.

Auch wenn keine Injektion von unerwünschen Queries mehr möglich ist, bleiben andere Sicherheitsrisiken bestehen, da die Daten aus der Referenz beispielsweise offen das Passwort zur Datenbank beinhalten. Dieses Problem kann durch verschiedene Methoden gelöst werden – so könnten beispielsweise von der Datenbank nur Zugriffe vom RRS zugelassen werden oder aber das RRS verschlüsselt die Passwörter innerhalb der Referenzen. Im Rahmen dieser Bachelorabeit wird diese Problematik jedoch nicht weiter vertieft. Ein Überblick über verbliebene Sicherheitsrisiken, die im Falle einer realen Anwendung vermieden werden müssen, wird im Ausblick in Kapitel 6 gegeben.

Zusätzlich zu den Verbindungsinformationen muss das Schema der Referenzen auch den internen Namen der Variablen enthalten, sodass einzelne Variablen auf demselben Datenspeicher identifiziert werden können. Diese Namen müssen lediglich innerhalb des Datenspeichers einzigartig sein. Bei der Erstellung neuer Variablen wird dies transaktional sichergestellt, damit es nicht zu Namenskollisionen kommt und die Eindeutigkeit der Referenzen bestehen bleibt. Die Verbindungsinformationen zum Datenspeicher und der Name der Variablen legen gemeinsam den Speicherort einer Variablen eindeutig fest.

Das fertige Schema für die Referenzen wird in Listing 4 dargestellt. Der Typ für das Element <storageLocation> wird in ein anderes XML-Schema-Dokument storagetypes.xsd ausgelagert. Dieses enthält alle Definitionen für die verschiedenen Speicherarten, in denen potentiell Variablen abgelegt werden können. In dieser Arbeit wird exakt eine Speicherart verwendet, ein Datenbanksystem auf Basis von BaseX. Die zugehörige storagetypes.xsd ist in Listing 5 dargestellt.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- Include the types of storages that can be used. -->
  <xs:include schemaLocation="storagetypes.xsd" />

  <xs:element name="reference">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="storageLocation" type="storageTypeT" />
        <xs:element name="variableName" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Listing 4: Schema der Referenzen (reference.xsd).

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="storageTypeT">
    <xs:choice>

      <xs:element name="BaseXXMLDatabase">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="hostname" type="xs:string" />
            <xs:element name="port" type="xs:integer" />
            <xs:element name="database" type="xs:string" />
            <xs:element name="username" type="xs:string" />
            <xs:element name="password" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>

    </xs:choice>
  </xs:complexType>

</xs:schema>
```

Listing 5: Definition der Speichertypen (storagetypes.xsd).

Auf diese Art und Weise kann nun eine einfache Erweiterung der Speichertypen geschehen, indem in der storagestypes.xsd dem <choice>-Konstrukt weitere Elemente hinzugefügt werden. Soll dieser Speichertyp auch vom RRS aufgelöst werden können, ist ferner die Implementierung eines entsprechenden Konnektors notwendig, worauf in Kapitel 4.2.3 eingegangen wird.

Was die Referenzen nicht enthalten, ist ein Verweis auf das RRS selbst, das zur Auflösung der Referenzen fähig ist. Diese Entscheidung wurde im Hinblick darauf getroffen, dass der Service Composition Engine für einen bestimmten Prozess in dessen Grammatik immer mindestens ein RRS zugeordnet ist und damit stets ein Verweis existiert.

4 Reference Resolution System

Das RRS ist das Kernstück einer referenzbasierten Datenverwaltung. Es ist dafür verantwortlich, Daten abzulegen und in diesem Zuge eine Referenz auf das gespeicherte Datum zu generieren. Durch Auflösung der Referenzen soll anschließend jedes Datum wieder erhalten sowie auch manipuliert werden können, bis es schließlich gelöscht wird und Referenzen, die auf dieses Datum verweisen, damit ihre Gültigkeit verlieren.

Damit besitzt die Schnittstelle des RRS vier zentrale Operationen:

- **Insert**
  Ein Datum in Form von XML wird vom RRS in einem der möglichen Datenspeicher als Variable abgelegt. Eine Referenz, die das Datum eindeutig identifiziert, wird generiert und als Antwort zurückgegeben.

- **Get**
  Eine Referenz wird aufgelöst und der Wert der Variablen, auf die die Referenz verweist, zurückgegeben.

- **Update**
  Eine Referenz wird aufgelöst und der Wert der Variablen, auf die die Referenz verweist, durch einen anderen Wert ersetzt. Dies entspricht etwa einer Zuweisungsoperation.

- **Delete**
  Eine Referenz wird aufgelöst und die Variable, auf die die Referenz verweist, aus dem zugehörigen Datenspeicher gelöscht, wodurch Referenzen auf sie nicht zukünfig nicht mehr aufgelöst werden können.

Das RRS wurde im Rahmen dieser Arbeit als ein SOAP-Web-Service entwickelt. Aufgrund der Tatsache, dass nur die Service Composition Engine das RRS aufruft, wäre für die momentane Verwendung zwar eine einfache, eng an die Service Composition Engine gekoppelte Komponente ausreichend gewesen, allerdings bietet die hier gewählte Architektur große Vorteile hinsichtlich der Erweiterbarkeit: Die Konzeption als Web-Service bietet die Möglichkeit,

dass ein RRS nicht nur von einer Service Composition Engine genutzt wird, sondern von mehreren [GL12]. In einem solchen Szenario wäre mithilfe von Referenzen auch die Möglichkeit existent, Referenzen zwischen verschiedenen Geschäftsprozessen, die von verschiedenen Service Composition Engines ausgeführt werden, auszutauschen. Diese Idee lässt sich erweitern zu Variablen, deren Werte zentral verwaltet oder sogar von externen Instanzen reguliert werden. [WG09] geht auf einen solchen Anwendungsfall genauer ein und beschreibt darin, wie solche außerhalb der Geschäftsprozesse festgelegten Daten (wie zum Beispiel sich ändernde gesetzliche Vorgaben) mithilfe von Referenzen ohne Redundanzen verwaltet werden können. Änderungen an solchen Daten müssten somit nicht durch alle Geschäftsprozess propagiert werden.

In einer solchen Umgebung könnte das RRS aufgrund des Entwurfs als Web-Service dann etwa durch UDDI dynamisch gefunden werden. Dies würde es nicht mehr nötig machen, dass die Adresse des RRS fest in der Grammatik eines Prozesses kodiert ist. Um die auftretende Arbeitslast bei vielen laufenden Geschäftsprozessen auszugleichen, wäre dann auch das Deployment von mehreren RRS im Rahmen dynamischer Skalierbarkeit möglich – dabei wäre es nicht notwendig, dass dasselbe RRS, das eine Referenz generiert, auch zur Auflösung derselben genutzt wird, was sich positiv auf die Robustheit des Gesamtsystems auswirkt.

Implementiert wurde das RRS in Java mit der in Kapitel 2.2 vorgestellten Java API for XML Web Services und der zugehörigen Referenzimplementierung. Im folgenden Kapitel soll wird der Entwurf des Systems vorgestellt. Anschließend geht Kapitel 4.2 darauf ein, wie das System auf einfache Art und Weise ausgebaut werden kann, um weitere Datenspeicher zu unterstützen.

4.1 Entwurf

Aufgrund der oben diskutierten Möglichkeiten zur Ausweitung des Systems spielen beim Entwurf des RRS vor allem zwei Aspekte eine wichtige Rolle: Flexibilität und Erweiterbarkeit.

Das RRS soll ohne Probleme schnell auf anderen Servern bereigestellt bzw. in andere Systeme integriert werden können. Gleichzeitig sollen möglichst auf einfache Art und Weise und ohne ein nötiges Redeployment weitere Datenspeicher hinzugefügt werden können, die zum Halten von Variablen genutzt werden. Die Lösung des Integrationsaspekt ergibt sich bereits aus der Implementierung des RRS als Web-Service, wie zu Beginn von Kapitel 4 angespro-

chen. Die Erweiterbarkeit soll dadurch gewährleistet werden, dass Informationen zu den Datenspeichern nicht fest im Programmcode gespeichert werden, sondern im Deployment-Deskriptor ein Konfigurationsverzeichnis definiert wird. Aus diesem werden dynamisch zur Laufzeit alle verwendeten XML-Schemata ausgelesen, sowie alle XML-Dokumente, die die Verbindungsinformationen zu den genutzten Datenspeichern enthalten.

Abbildung 6 zeigt das grundlegende Entwurfsdiagramm. Die Klasse Interface enthält alle nach außen sichtbaren Funktionen. Soll ein neues Datum extern gespeichert werden, muss für die Wahl eines geeigneten Datenspeichers eine Klasse der Schnittstelle Heuristic implementiert werden, deren Aufgabe es ist, den Datenspeicher auszuwählen. Ein Beispiel hierfür ist die Klasse AlwaysBaseXHeuristic, die die BaseX-Datenbank auswählt, in der im Rahmen dieser Arbeit die Variablen abgelegt werden. Die Klasse ConnectorFactory wird genutzt, um zur Laufzeit Instanzen von Konnektoren zu erzeugen (Klassen, die die Schnittstelle StorageConnector implementieren). Diese realisieren den Zugriff auf die Datenspeicher. In dieser Arbeit ist dies der BaseXXMLDatabaseConnector, der die Aufruffunktionen des BaseXClients zur Übermittlung der Datenbankbefehle nutzt. Konnektoren müssen die folgenden Methoden imlementieren: Initialize (zum Öffnen des Datenspeichers und für andere vorbereitende Maßnahmen), Close (Schließen der Verbindung zum Datenspeicher) sowie Insert, Get, Update und Delete, die verantwortlich dafür sind, die Variablen im Speicher abzulegen, sie zu erfragen, zu modifizieren oder zu löschen (zum Beispiel, indem entsprechende Queries erzeugt und ausgeführt werden).

Bei Anfragen, die an das RRS gestellt werden, arbeitet das System folgendermaßen: Wird eine der Operationen Get, Update oder Delete aufgerufen, wird als Parameter eine Referenz übergeben. Diese enthält bereits alle nötigen Informationen darüber, wo die Variable, auf die verwiesen wird, abgelegt ist (siehe Kapitel 3). Wird dagegen die Operation Insert aufgerufen, muss für die neu anzulegende Variable erst ein Speicherort gefunden werden. Hierzu kann eine Instanz einer Klasse verwendet werden, die von der Schnittstelle Heuristic erbt (wie etwa die in Abbildung 6 dargestellte Klasse AlwaysBaseXHeuristic). In dieser können verschiedene Wege definiert werden, wie der geeignetste Datenspeicher für das zu speichernde Datum aufgefunden werden kann. Näheres zu Heuristiken folgt in Kapitel 4.2. Im Rahmen dieser Bachelorarbeit existiert lediglich eine einzige Heuristik, die immer die einzig genutzte XML-Datenbank auf Basis des BaseX-Datenbanksystems auswählt.

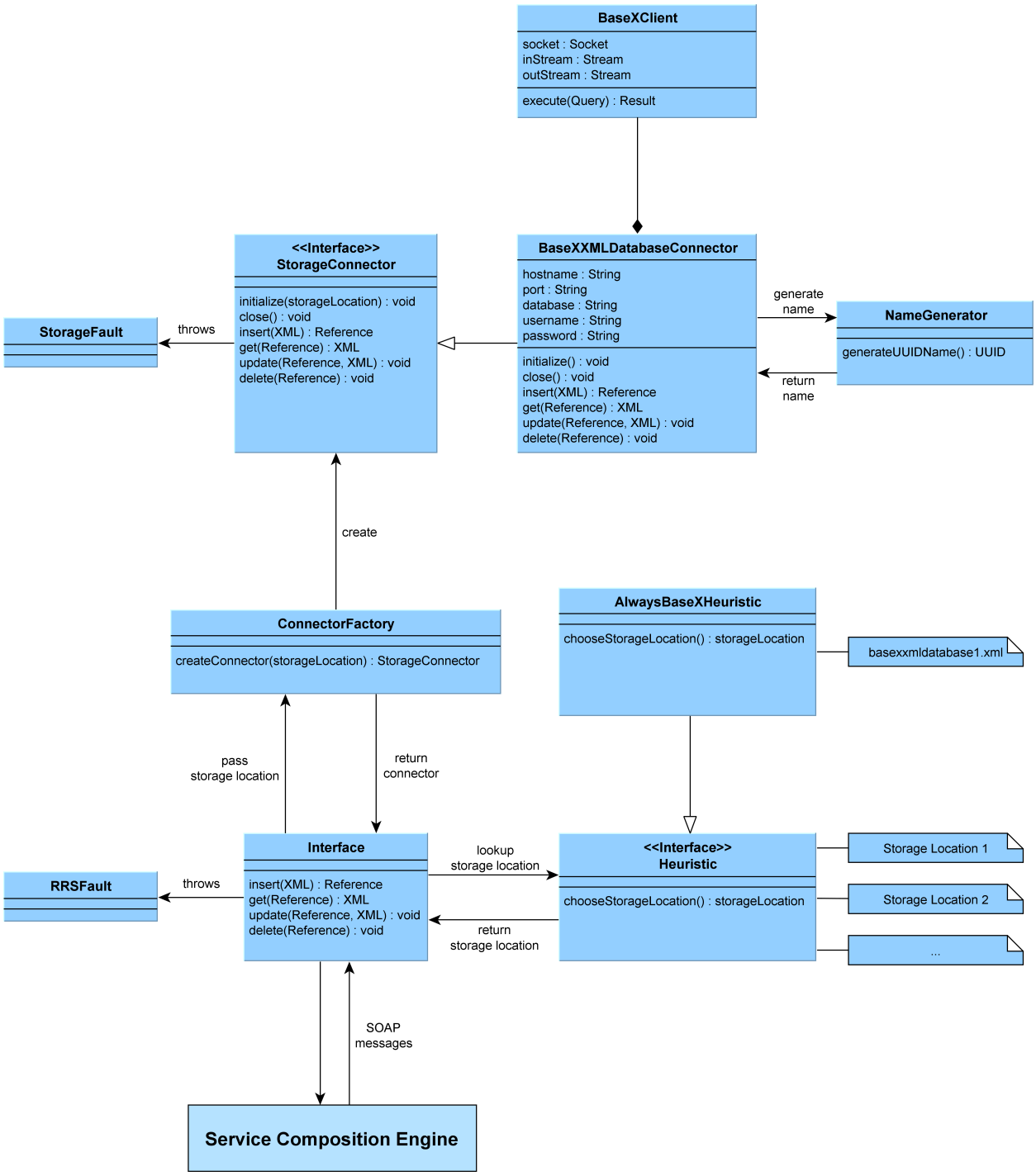

Abbildung 6: Der grundlegende Aufbau des Reference Resolution System.

Um eine Verbindung zum Datenspeicher herzustellen, existieren die sogenannten Konnektoren. Sie leiten ab von der Schnittstelle StorageConnector und implementieren für einen bestimmten Typ Datenspeicher den Zugriff auf diesen sowie das Speichern, Manupulieren und Löschen von Variablen. Der richtige Konnektor wird von der Klasse StorageConnectorFactory zur Laufzeit mithilfe der Verbindungsinformationen ausgewählt und die entsprechende

Klasse dynamisch instanziiert. Die Instanz wird an das Interface zurückgegeben, das nun direkt auf die Operationen des Konnektors zugreifen kann und durch diese eine Verbindung zum Datenspeicher besitzt.

Nachdem der Konnektor erzeugt wurde, gibt die Klasse Interface bei diesem die nötige Operation in Auftrag und leitet die hierfür nötigen Daten (beispielsweise die Referenz im Falle der Operationen Get, Update und Delete) per Parameter weiter, damit der Konnektor in Folge die entsprechende Operation durchführen kann. Die Antwort (zum Beispiel der Wert der Variablen im Falle der Operation Get) wird anschließend per SOAP-Nachricht zum Web-Service-Requester zurückgegeben. Im Sonderfall der Methode Insert muss außerdem noch ein Name für die Variable generiert werden, der mit hoher Wahrscheinlichkeit noch nicht in der Datenbank vertreten ist – hierfür ist der Namensgenerator zuständig. Im Rahmen dieser Arbeit werden hierfür pseudo-zufällige Universally Unique Identifier (kurz: UUID) mit einer Größe von 128 Bit als Namen vergeben[6], um die Wahrscheinlichkeit für eine Namenskollision möglichst gering zu halten. Wird zufällig ein Name gewählt, der bereits vergeben ist, lässt der hier verwendete Connector den Namensgenerator einen weiteren Namen erzeugen.

Falls es bei der Verarbeitung der Anfrage zu Fehlern kommt, enthält die zurückgegebene SOAP-Nachricht eine SOAP-Fault, die Auskunft darüber gibt, warum die Anfrage nicht ausgeführt werden konnte. Die Faults sind als Java Exceptions implementiert und werden von JAX-WS entsprechend in Faults konvertiert. Diese Fehler existieren auf zwei Ebenen: RRS-Faults werden erzeugt, wenn grundsätzliche Fehler auf Ebene der Klassen Interface und ConnectorFactory sowie in den Heuristiken auftreten (etwa wenn eine nicht valide Referenz übergeben wird oder kein passender Konnektor erzeugt werden kann). Tritt ein Fehler während Operationen auf dem Datenspeicher auf, wird stattdessen eine StorageFault erzeugt.

4.2 Erweiterbarkeit und Konfiguration

In Anbetracht der zu Beginn von Kapitel 4 erörterten Möglichkeiten zur Ausweitung des Einsatzes des RRS ist eine Erweiterbarkeit des Systems ein wesentliches Kriterium für den Entwurf. Um dieser Anforderung gerecht zu werden, bietet die zu dieser Arbeit gehörende Implementierung verschiedene Möglichkeiten zur Erweiterung:

6 „UUID (Java 2 Platform SE 5.0)“, http://docs.oracle.com/javase/1.5.0/docs/api/java/util/UUID.html#randomUUID(), aufgerufen am 04.11.2013

- Die Definition eines Konfigurationsverzeichnisses, das sowohl die XML-Schemata der Referenzen und Speicherarten als auch Verbindungsinformationen zu allen bestehenden Datenspeichern enthält, erlaubt das dynamische Hinzufügen und Entfernen von Speicherorten.

- Die Schnittstelle Heuristic dient als Vorlage für die Implementierung von Heuristiken, deren Aufgabe es ist, automatisiert einen geeigneten Speicherort für ein abzulegendes Datum zu finden.

- Konnektoren implementieren die Verbindung des RRS zu den Datenspeichern und die Ausführung der Insert-, Get-, Update- und Delete-Operationen auf den Speichern. Eine andere Art von Datenspeicher kann schnell und flexibel durch Implementierung eines zugehörigen Konnektors hinzugefügt werden.

- Ein Namensgenerator kann verschiedene Methoden zur Generierung geeigneter, möglichst noch nicht in der Datenbank vorhandener Namen implementieren.

4.2.1 Verwaltung der Datenspeicher durch das Konfigurationsverzeichnis

In Kapitel 4.1 wurde bereits die Existenz eines zentralen Konfigurationsverzeichnisses erwähnt. Dieses Verzeichnis wird im Deployment-Deskriptor festgelegt und enthält alle zur Validierung nötigen Schemata (das Schema für Referenzen, die Speicherarten und die Verbindungsinformationen) sowie die Verbindungsinformationen über alle verfügbaren Datenspeicher, die in Form von XML-Dokumenten vorliegen.

Das Verzeichnis kann dynamisch zur Laufzeit durchsucht werden, wodurch es insbesondere möglich wird, Datenspeicher während der Laufzeit hinzuzufügen oder zu entfernen. Die enthaltenen XML-Instanzen, die jeweils die Verbindungsinformationen zu einem Datenspeicher enthalten, müssen dem in Listing 6 dargestellten Schema folgen.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- Include the types of storages that can be used. -->
  <xs:include schemaLocation="storagetypes.xsd" />

  <xs:element name="storageInformation">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="storageLocation" type="storageTypeT" />
        <xs:element name="storageProperties">
          <xs:complexType>
            <xs:all>
              <!-- Possible properties of storages. -->
              <xs:element name="maximumDataSize" type="xs:integer" />
              <xs:element name="setupDate" type="xs:date" />
              <xs:element name="lastUsedOn" type="xs:data" />
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Listing 6: Schema der XML-Dokumente, die Informationen zu einem bestimmten Datenspeicher enthalten (storageinformation.xsd).

Das Wurzelelement <storageInformation> gliedert sich in zwei Teile: Zum einen enthält es die Verbindungsinformationen zum Datenspeicher (Element <storageLocation>), die in der storagetypes.xsd definiert wurden (siehe Listing 5 in Kapitel 3.4); zum anderen besteht die Möglichkeit, diese rein technischen, funktionalen Daten mit nichtfunktionalen Eigenschaften anzureichern. Diese werden innerhalb des Elements <storageProperties> aufgelistet. Die mometane Implementierung nutzt diese nichtfunktionalen Eigenschaften noch nicht – daher ist ihr Vorkommen optional. Allerdings können Heuristiken definiert werden, die einen geeigneten Datenspeicher auf Basis dieser Eigenschaften auswählen. Denkbar ist beispielsweise auch das Einbinden von WS-Policy-Richtlinien an dieser Stelle.

4.2.2 Auswahl geeigneter Datenspeicher mit Heuristiken

Der Begriff Heuristik findet außer in der Informatik auch in zahlreichen anderen Anwendungsgebieten Gebrauch. [PE84] beispielsweise definiert mit Heuristiken Strategien, die verfügbare und zugängliche Informationen nutzen, um das Lösen von Problemen sowohl durch Maschinen als auch durch Menschen zu kontrollieren. In der künstlichen Intelligenz werden besonders bei Problemen mit großem Suchraum Heuristiken genutzt, um mit begrenzter Information möglichst schnell gute Entscheidungen zu treffen [HN68]. Im Kontext des RRS soll der Begriff jedoch deutlich enger gefasst werden:

Definition (Heuristik):

Eine Heuristik beschreibt eine Sammlung verwandter Vorgehensweisen, die mit möglicherweise unterschiedlichen Mengen an Information über die zu speichernden Daten den dafür im Sinne einer bestimmten Strategie passendsten verfügbaren Datenspeicher auswählen.

Im RRS sind Heuristiken dafür vorgesehen, von der Operation Insert verwendet zu werden. Immer dann, wenn eine neue Variable in einem Datenspeicher abgelegt werden soll, muss entschieden werden, welcher der verfügbaren Speicher hierzu verwendet wird. Heuristiken sollten hierzu die Schnittstelle Heuristic implementieren. Eine Heuristik kann dann verschiedene Methoden implementieren, um eine Wahl zu treffen.

Da im Rahmen dieser Arbeit dem RRS nur ein einziger Datenspeicher zur Verfügung steht, wählt die implementierte Heuristik (AlwaysBaseXHeuristic) immer die BaseX-Datenbank aus, die im XML-Dokument basexxmldatabase1.xml deklariert wird. Soll das RRS zukünftig in einer größeren Umgebung verwendet werden, in der es gegebenenfalls von mehreren unterschiedlichen Prozessen mit unterschiedlich großen zu speichernden Datenmengen verwendet wird, wird die Wahl des geeignetsten Datenspeichers unter Umständen deutlich komplexer, sodass die Definition weiterer Strategien nötig sein kann. Hierzu kann es sinnvoll sein, gemäß Kapitel 4.2.1 für jeden Speicher eine Menge an nichtfunktionalen Eigenschaften zur Verfügung zu stellen, die den Heuristiken als Information zur Verfügung stehen. Einige mögliche sich daraus ergebende Strategien sollen hier beispielhaft aufgeführt werden:

- Es wird immer der Datenspeicher ausgewählt, der am längsten nicht verwendet wurde, um die Variablen möglichst gleichmäßig auf verfügbare Datenbanken zu verteilen.
- Es wird stets ein Datenspeicher zufällig ausgewählt, um durch Randomisierung zu vermeiden, dass periodische Muster bei den Aufrufen von Insert zur Überfüllung einzelner Datenspeicher führen.
- Für große Variablen werden vorrangig Speicher ausgewählt, die solche Datenmengen effizienter verarbeiten können.
- Bestimmte Datenspeicher werden generell präferiert, während andere nur als Ersatzlösung zur Verfügung stehen, falls primäre Datenbanken ausfallen.

4.2.3 Hinzufügen von Konnektoren

In dieser Arbeit wird zur Speicherung der Variablen das Datenbanksystem BaseX verwendet. Die Vorteile von BaseX als ein generell für diese Verwendung gut geeigneter Datenspeicher wurden bereits in Kapitel 2.3 erläutert. Weitere BaseX-Datenbanken können wie in Kapitel 4.2.1 beschrieben einfach durch das Hinzufügen neuer XML-Dokumente, die dem in der storageinformation.xsd definierten XML-Schema folgen, in das Konfigurationsverzeichnis an das RRS angebunden werden.

Wird das Gesamtsystem jedoch ausgeweitet und in spezifischeren Szenarien verwendet, kann es zu Sonderfällen kommen, wenn beispielsweise für hochperformante Anfragen Key-Value-Stores verwendet werden sollen [DH07] oder aufgrund rechtlicher Einschränkungen ein Auslagern der Daten in beliebige externe Datenbanken nicht möglich ist. In diesen Fällen ist es nötig, dass andere Arten von Datenspeichern unterstützt werden. Dies können andere Datenbanksysteme mit abweichender Kopplung sein oder aber es werden gar keine Datenbanken zum Speichern der Variablen verwendet, sondern das lokale Dateisystem. Ist das der Fall, müssen für das Öffnen des neuen Datenspeichers und die Übertragung der Queries neue Routinen implementiert werden, im besten Fall ohne die Funktionalität der bestehenden Methoden zu ändern. Zu diesem Zweck werden die sogenannten Konnektoren verwendet. Dies sind Klassen, die die Schnittstelle StorageConnector implementieren. Sie sind verantwortlich für das Herstellen und Trennen der Verbindung und das Ausführen aller Befehle, die zum Speichern, Abrufen, Manipulieren und Löschen von Variablen benötigt werden.

Soll das System eine neue Art Datenspeicher unterstützen, muss zusätzlich zur Erweiterung der Typen in der storagetypes.xsd lediglich ein neuer Konnektor implementiert werden, der diese Operationen unterstützt. Die ConnectorFactory instantiiert den neuen Konnektor dann auf Basis der zum Datenspeicher zugehörigen XML-Instanz automatisch, wodurch keine weitere Integration notwendig ist.

4.2.4 Erweiterung des Namensgenerators

In dieser Arbeit werden als Namen für Variablen ausschließlich UUIDs verwendet[7]. Diese werden von einem Namensgenerator erzeugt und in Form von Strings zurückgegeben.

7 „UUID (Java 2 Platform SE 5.0)“, http://docs.oracle.com/javase/1.5.0/docs/api/java/util/UUID.html#randomUUID(), aufgerufen am 04.11.2013

Ist diese Art der Namensvergabe für einen bestimmten Datenspeicher nicht möglich (etwa weil diese zu lang sind oder dadurch unerwünschte Zeichen im Bezeichner vorkommen), müssen andere Namen generiert werden. Hierzu kann der Namensgenerator um die entsprechenden Methoden erweitert werden. Diese können anschließend vom Konnektor benutzt werden.

5 Deployment und Validierung

In diesem Kapitel soll das zu dieser Arbeit gehörende, implementierte RRS (das dem in Kapitel 4 vorgestellten Entwurf folgt) auf einem Server bereitgestellt werden; ebenso soll eine Datenbank auf Basis des Datenbanksystems BaseX bereitgestellt werden, die an das RRS angebunden wird. Anschließend soll die Funktionsweise der implementierten Methoden validiert werden.

Kapitel 5.1 stellt vorerst die Systemspezifikation des Systems vor, auf dem die Tests durchgeführt wurden. Kapitel 5.2 widmet sich anschließend der Validierung. Hierzu wird das System anhand verschiedener Testfälle geprüft. In Kapitel 5.3 werden einige Statistiken im Hinblick auf die Performanz des Web-Services erläutert.

5.1 Testumgebung

Alle folgenden Prüfungen wurden auf einem Computer mit Windows 7 Home Premium als Betriebssystem in der 64-Bit-Version durchgeführt. Das System enthält einen Zweikern-Prozessor mit einem Takt von jeweils 2,27 Gigahertz. Es verfügt über einen Arbeitsspeicher von 4 Gigabyte.

Das BaseX-Datenbanksystem wurde direkt auf dieses System aufgespielt. Um eine einfache Konfiguration zu gewährleisten, existiert lediglich ein Benutzer und eine Datenbank, in der die Variablen abgelegt werden. Das RRS wurde auf einem Tomcat-Server der Version 7.0 bereitgestellt[8]. Im Deployment-Deskriptor des RRS wurde entsprechend der Konfigurationspfad zu allen XML-Schemata und der XML-Datei mit den Verbindungsinformationen zur verwendeten Datenbank gesetzt. Ebenfalls im Deployment-Deskriptor angegeben wurden die URL und die Implementierungsklasse. Der Tomcat-Server selbst läuft ebenfalls auf dem oben beschriebenen System.

Um den Web-Service aufzurufen, wird SoapUI verwendet[9]. SoapUI ist eine quelloffene Software, mit der das funktionale Prüfen von Web-Services möglich ist. Sie bietet sowohl einfache Möglichkeiten zum Testen von Methoden durch den Versand einzelner SOAP-Nachrichten als auch komplexere Prüfmöglichkeiten wie das Definieren von Testsuites, die kompliziер-

8 „Apache Tomcat“, http://tomcat.apache.org, abgerufen am 13.11.2013
9 „SoapUI – The Home of Functional Testing“, http://www.soapui.org, abgerufen am 13.11.2013

tere Szenarien abdecken können.

5.2 Validierung

Um die korrekte Funktion des RRS sicherzustellen, wurde eine Reihe von Testfällen in Form von SOAP-Nachrichten entworfen, die mithilfe von SoapUI an das RRS geschickt wurden. Das Ergebnis der Operationen wurde anschließend mit dem erwarteten Ergebnis verglichen. Im Folgenden werden einige der Ergebnisse auszugsweise detailliert vorgestellt. Daraufhin folgt eine kurzgefasste Liste aller Ergebnisse.

5.2.1 Testfall A: Insert komplexer XML-Daten und anschließendes Get

Dieser Testfall beschäftigt sich mit der Insert- und der Get-Operation. Es sollen XML-Daten in der Datenbank gespeichert werden, die gängige Konstrukte (Attribute, Kommentare, Escape-Sequenzen) sowie Sonderzeichen enthalten. Als Ergebnis wird erwartet, dass die Daten erfolgreich in der Datenbank gespeichert werden können und bei einem Get in derselben Form zurückgeliefert werden.

Die in Listing 7 dargestellte SOAP-Nachricht wurde hierfür an das RRS übermittelt:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:core="http://core/">
 <soapenv:Header/>
 <soapenv:Body>
  <core:insert>
   <initialValue>
    <![CDATA[<Element1 Attrib1="5">Test</Element1>
    <Element2 Attrib2="Test">42.2</Element2>
    <!-- This is a comment. -->
    <Element3>1&lt;2 &amp; 2lt;3 -&gt; 1&lt;3</Element3>
    <Element4>!"§$%/()=?´*+~#'^°;:|@€</Element4>]]>
   </initialValue>
  </core:insert>
 </soapenv:Body>
</soapenv:Envelope>
```

Listing 7: SOAP-Nachricht zur Prüfung des Verhaltens bei der Insert-Operation.

Die Antwortnachricht soll die Referenz zu diesem Datum enthalten. Listing 8 stellt die Nachricht dar.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <ns2:insertResponse xmlns:ns2="http://core/">
   <return>
    <[CDATA[<reference><storageLocation><BaseXXMLDatabase><hostname>
    localhost</hostname><port>1984</port><database>testdb</database>
    <username>testuser</username><password>password</password>
    </BaseXXMLDatabase></storageLocation><variableName>
    cc73ed87-1264-4f0f-934a-2e8b02d3e5dd</variableName></reference>]]>
   </return>
  </ns2:insertResponse>
 </S:Body>
</S:Envelope>
```

Listing 8: SOAP-Nachricht als Antwort auf die Insert-Operation.

Mit der erhaltenen Referenz sollen die Daten nun wieder abgerufen werden. Die Listings 9 und 10 zeigen die Anfrage- und die Antwortnachricht. Die XML-Daten wurden korrekt in der Datenbank abgelegt und zurückgeliefert.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:core="http://core/">
 <soapenv:Header/>
 <soapenv:Body>
  <core:get>
   <reference>
    <![CDATA[<reference><storageLocation><BaseXXMLDatabase><hostname>
    localhost</hostname><port>1984</port><database>testdb</database>
    <username>testuser</username><password>password</password>
    </BaseXXMLDatabase></storageLocation><variableName>
    cc73ed87-1264-4f0f-934a-2e8b02d3e5dd</variableName></reference>]]>
   </reference>
  </core:get>
 </soapenv:Body>
</soapenv:Envelope>
```

Listing 9: SOAP-Nachricht zur Prüfung des Verhaltens bei der Get-Operation.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <ns2:getResponse xmlns:ns2="http://core/">
   <return>
    <![CDATA[<Element1 Attrib1="5">Test</Element1>
    <Element2 Attrib2="Test">42.2</Element2>
    <!-- This is a comment. -->
    <Element3>1&lt;2 &amp; 2lt;3 -&gt; 1&lt;3</Element3>
    <Element4>!"§$%/()=?´*+~#'^°;:|@€</Element4>]]>
   </return>
  </ns2:getResponse>
 </S:Body>
</S:Envelope>
```

Listing 10: SOAP-Nachricht als Antwort auf die Get-Operation.

5.2.2 Testfall B: Update für nicht existierende Variable

Dieser Testfall zeigt das Ergebnis einer Update-Operation, deren übergebene Referenz einen Variablennamen trägt, der nicht im Datenspeicher vorkommt. Da es sich hierbei um einen Fehler auf Ebene des Datenspeichers handelt, wird nach Kapitel 4.1 als Ergebnis eine StorageFault erwartet. Die Listings 11 und 12 zeigen die Anfrage und die erhaltene Antwortnach-

richt mit dem erwünschten Ergebnis.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:core="http://core/">
 <soapenv:Header/>
 <soapenv:Body>
  <core:update>
   <reference>
    <![CDATA[<reference><storageLocation><BaseXXMLDatabase><hostname>
    localhost</hostname><port>1984</port><database>testdb</database>
    <username>testuser</username><password>password</password>
    </BaseXXMLDatabase></storageLocation><variableName>NonExistent
    </variableName></reference>]]>
   </reference>
   <newValue>NewValue</newValue>
  </core:update>
 </soapenv:Body>
</soapenv:Envelope>
```

Listing 11: SOAP-Nachricht zur Prüfung des Verhaltens bei der Update-Operation, wenn die Variable nicht existiert.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
   <faultcode>S:Server</faultcode>
   <faultstring>Reference could not be resolved.</faultstring>
   <detail>
    <ns2:StorageFault xmlns:ns2="http://core/">
     <message>Reference could not be resolved.</message>
    </ns2:StorageFault>
   </detail>
  </S:Fault>
 </S:Body>
</S:Envelope>
```

Listing 12: SOAP-Nachricht als Antwort auf die fehlerhafte Update-Anfrage.

5.2.3 Testfall C: Delete mit nicht valider Referenz

Im Folgenden wird die Operation Delete aufgerufen, allerdings mit einer Referenz, die nicht dem definierten Schema innerhalb der reference.xsd entspricht (es fehlen der Name der Variable sowie das umschließende <StorageLocation>-Element und das Passwort). Der Aufruf soll daher eine RRSFault erzeugen. Listing 13 zeigt die SOAP-Nachricht des Aufrufs, Listing 14 die erwartete und erhaltene RRSFault.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:core="http://core/">
 <soapenv:Header/>
 <soapenv:Body>
  <core:delete>
   <reference>
    <![CDATA[<reference><BaseXXMLDatabase><hostname>localhost</hostname>
    <port>1984</port><database>testdb</database><username>testuser
    </username></BaseXXMLDatabase></reference>]]>
   </reference>
  </core:delete>
 </soapenv:Body>
</soapenv:Envelope>
```

Listing 13: Aufruf der Delete-Operation als SOAP-Nachricht mit nicht valider Referenz.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
   <faultcode>S:Server</faultcode>
   <faultstring>No valid reference was passed.</faultstring>
   <detail>
    <ns2:RRSFault xmlns:ns2="http://core/">
     <message>No valid reference was passed.</message>
    </ns2:RRSFault>
   </detail>
  </S:Fault>
 </S:Body>
</S:Envelope>
```

Listing 14: SOAP-Nachricht als Antwort auf die fehlerhafte Delete-Anfrage.

5.2.4 Übersicht über weitere Testfälle

Tabelle 1 liefert eine Übersicht über weitere geprüfte Testfälle, die allesamt das erwartete Ergebnis lieferten.

Anfrage	**Antwort**
Insert einer leeren Zeichenkette	Referenz
Insert wohlgeformter XML-Daten	Referenz
Insert nicht wohlgeformter XML-Daten	StorageFault
Datenbank ist offline oder nicht erreichbar	StorageFault
Get mit einer validen, gültigen Referenz	Wert
Get/Update/Delete mit einer invaliden Referenz	RRSFault
Get/Update/Delete einer nicht existierenden Variable	StorageFault
Update mit leerem String als neuem Wert	OK
Update mit wohlgeformten XML-Daten als neuem Wert	OK
Update mit nicht wohlgeformten XML-Daten als neuem Wert	StorageFault
Delete mit einer validen, gültigen Referenz	OK
Konfigurationsverzeichnis falsch gesetzt	RRSFault
Datenspeicher-Typ wird nicht unterstützt	RRSFault
Get/Update/Delete ohne Referenz	RRSFault

Tabelle 1: Übersicht über Testfälle bei der Prüfung des Reference Resolution Systems.

5.3 Performanz

Um die Performanzeigenschaften des Systems unter verschiedenen Bedingungen zu testen, wurde die Datenbank vorerst mit einer vorgegebenen Anzahl an Variablen gefüllt. Um den Vorgang zu beschleunigen und nicht viele Male den Web-Serivce aufrufen zu müssen, wurden diese mit XQuery-Befehlen eingefügt, wie beispielhaft in Listing 15 dargestellt. Der dort gezeigte XQuery-Befehl fügt 100.000 Variablen in die Datenbank ein; diese haben eine Form wie in Listing 16 gezeigt. Der Vorgang wurde für eine verschiedene Anzahl Variablen wiederholt.

```
insert node (for $x in (1 to 100000) return <Variable id='test-variable-
00{$x}-rrs'><TestXML><Name>test-variable-00{$x}-
rrs</Name><Identifier>{$x}</Identifier><Text-Content>Text Content</Text-
Content><ElementWithAttributes attrib1='{$x}' attrib2='{$x}'>Element with
Attributes</ElementWithAttributes></TestXML></Variable>) into
/VariableDatabase
```

Listing 15: Beispielhafter XQuery-Befehl, um 100.000 Variablen in die Datenbank einzufügen.

```
<Variable id='test-variable-001-rrs'>
  <TestXML>
    <Name>test-variable-001-rrs</Name>
    <Identifier>1</Identifier>
    <Text-Content>Text Content</Text-Content>
    <ElementWithAttributes attrib1='1' attrib2='1'>Element with
        Attributes</ElementWithAttributes>
  </TestXML>
</Variable>
```

Listing 16: Beispiel für eine durch den XQuery-Befehl erzeugte Variable.

Gemessen wurden die durchschnittlichen, minimalen und maximalen Antwortzeiten für die Anfragen. In allen Fällen wurde darauf geachtet, dass die Anfragen in korrekter Form gestellt werden und die Datenbank funktionsfähig ist, um die Performanzmessung der grundlegenden Funktionalität nicht durch Sonderfälle oder SOAP-Faults zu verfälschen. Ferner wurde mithilfe der Java-Funktion System.nanoTime() der Anteil der Zeit gemessen, den die Ausführung der Befehle auf der Datenbank einnimmt. Aus diesem Wert kann rekonstruiert werden, wie viel Zeit die datenbankunabhängigen Berechnungen des RRS selbst in Anspruch nehmen. Abbildung 7 zeigt einen Screenshot der Konfiguration der Prüfung in SoapUI. In den Tabellen 2 bis 5 werden die Ergebnisse der Messungen dargestellt.

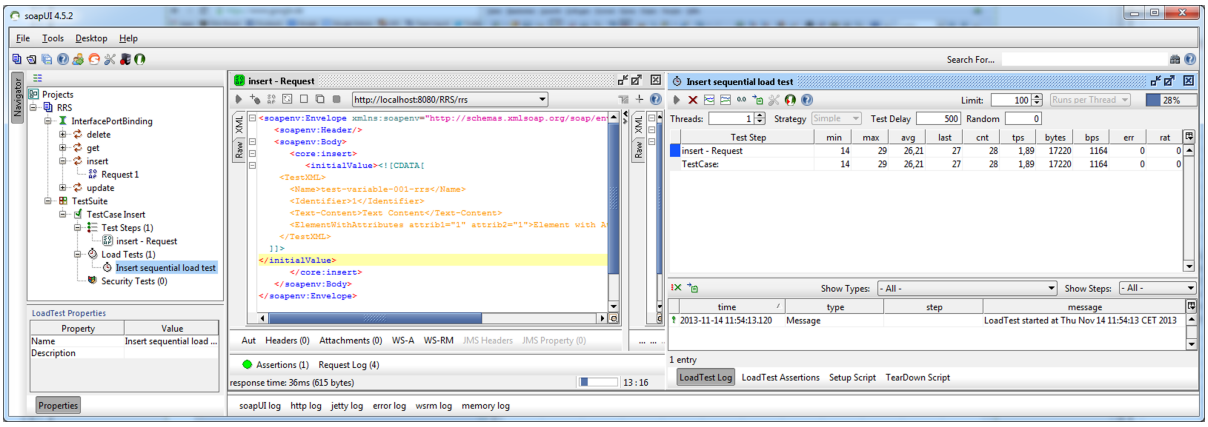

Abbildung 7: Konfiguration von und Performanzprüfung mit SoapUI am Beispiel der Insert-Operation.

Insert				
Initiale Anzahl Variablen in der Datenbank	**Minimale Antwortzeit**	**Maximale Antwortzeit**	**Durchschnittliche Antwortzeit**	**Zeit für RRS-interne Berechnungen**
10	16 ms	51 ms	23 ms	14 ms
100	17 ms	64 ms	24 ms	13 ms
1.000	19 ms	41 ms	26 ms	18 ms
10.000	29 ms	58 ms	43 ms	15 ms
100.000	168 ms	276 ms	185 ms	15 ms
1.000.000	1502 ms	1790 ms	1531 ms	18 ms

Tabelle 2: Messung der Antwortzeiten der Insert-Operation.

Get				
Anzahl Variablen in der Datenbank	**Minimale Antwortzeit**	**Maximale Antwortzeit**	**Durchschnittliche Antwortzeit**	**Zeit für RRS-interne Berechnungen**
10	19 ms	60 ms	25 ms	20 ms
100	19 ms	42 ms	25 ms	21 ms
1.000	22 ms	55 ms	30 ms	24 ms
10.000	45 ms	72 ms	58 ms	23 ms
100.000	320 ms	390 ms	338 ms	21 ms
1.000.000	2962 ms	3252 ms	3038 ms	22 ms

Tabelle 3: Messung der Antwortzeiten der Get-Operation.

Update				
Anzahl Variablen in der Datenbank	**Minimale Antwortzeit**	**Maximale Antwortzeit**	**Durchschnittliche Antwortzeit**	**Zeit für RRS-interne Berechnungen**
10	21 ms	68 ms	30 ms	22 ms
100	22 ms	64 ms	32 ms	22 ms
1.000	26 ms	77 ms	38 ms	21 ms
10.000	50 ms	95 ms	66 ms	27 ms
100.000	318 ms	410 ms	346 ms	26 ms
1.000.000	2930 ms	3436 ms	3012 ms	24 ms

Tabelle 4: Messung der Antwortzeiten der Update-Operation.

Delete				
Initiale Anzahl Variablen in der Datenbank	**Minimale Antwortzeit**	**Maximale Antwortzeit**	**Durchschnittliche Antwortzeit**	**Zeit für RRS-interne Berechnungen**
10	26 ms	39 ms	32 ms	26 ms
100	29 ms	37 ms	34 ms	24 ms
1.000	32 ms	69 ms	38 ms	23 ms
10.000	64 ms	92 ms	77 ms	24 ms
100.000	328 ms	376 ms	342 ms	23 ms
1.000.000	2998 ms	3353 ms	3067 ms	24 ms

Tabelle 5: Messung der Antwortzeiten der Delete-Operation.

Für eine geringe Anzahl (bis etwa 10^4 Variablen in der Datenbank) sind kaum Einbußen an der Antwortzeit zu erkennen. Die Antwortzeiten liegen in der Größenordnung gängiger Antwortzeiten im Web, wodurch sie in einem realen Umfeld nur wenig ins Gewicht fallen. Erst wenn die Anzahl der in der Datenbank gespeicherten Variablen gegen die Größenordnung 10^6 geht, machen sich deutlich längere Antwortzeiten bemerkbar. Greifen in diesem Fall in der Realität viele Prozessinstanzen auf das RRS zu, würde dies die Performanz des Gesamtsystems erheblich beeinträchtigen. Daher muss in der Praxis rechtzeitig auf diesen Umstand reagiert werden, etwa durch das Hinzufügen neuer Datenbanken.

Betrachtet man die Menge an Zeit, die das RRS mit Berechnungen verbringt und nicht auf die Datenbank wartet, fällt auf, dass diese nahezu konstant bleibt. Die Schwankungen weisen keinerlei Tendenz auf und können daher als Messungenauigkeiten angesehen werden. Dies zeigt, dass für eine große Anzahl an Variablen der Engpass die angebundene Datenbank ist, nicht die Berechnungen innerhalb des RRS. Nur wenn die Datenbankoperationen schnell abgeschlossen sind, nehmen die Berechnungen innerhalb des RRS (Parsing, Instantiierung, Kommunikation) einen signifikanten Anteil ein.

Da ein wesentlicher Anwendungsfall für eine referenzbasierte Datenverwaltung das Speichern großer Datenmengen ist, wurden außerdem ähnliche Prüfungen mit großen Variablen durchgeführt. Die Listings 17 und 18 zeigen den dafür verwendeten XQuery-Befehl und die resultierenden Variablen. Die Größe dieser Variablen beträgt etwa Faktor 1.000 im Vergleich zu der vorherigen kleinen Variablen. Bei 100.000 eingefügten Variablen hat die resultierende Datenbank dabei eine Größe von etwa 22 Gigabyte.

```
insert node (for $x in (1 to 10) return <Variable id='test-variable-00{$x}-
rrs'>{(for $y in (1 to 1000) return <TestXML><Name>test-variable-00{$x}-
rrs</Name><Identifier>{$x}-{$y}</Identifier><Text-Content>Text
Content</Text-Content><ElementWithAttributes attrib1='{$x}'
attrib2='{$y}'>Element with Attributes</ElementWithAttributes></TestXML>)}
</Variable>) into /VariableDatabase
```

Listing 17: Beispielhafter XQuery-Befehl um zehn große Variablen in die Datenbank einzufügen.

```
<Variable id='test-variable-001-rrs'>
  <TestXML>
    <Name>test-variable-001-rrs</Name>
    <Identifier>1-1</Identifier>
    <Text-Content>Text Content</Text-Content>
    <ElementWithAttributes attrib1='1' attrib2='1'>Element with
        Attributes</ElementWithAttributes>
  </TestXML>

   .
   .
   .

  <TestXML>
    <Name>test-variable-001-rrs</Name>
    <Identifier>1-1000</Identifier>
    <Text-Content>Text Content</Text-Content>
    <ElementWithAttributes attrib1='1' attrib2='1000'>Element with
        Attributes</ElementWithAttributes>
  </TestXML>
</Variable>
```

Listing 18: Beispiel für eine durch den XQuery-Befehl erzeugte große Variable.

Die Tabellen 6 und 7 zeigen die Ergebnisse der Tests für die Insert- und die Get-Operation.

Insert, große Variablen				
Initiale Anzahl Variablen in der Datenbank	**Minimale Antwortzeit**	**Maximale Antwortzeit**	**Durchschnittliche Antwortzeit**	**Zeit für RRS-interne Berechnungen**
10	105 ms	342 ms	134 ms	24 ms
100	108 ms	387 ms	130 ms	20 ms
1.000	118 ms	290 ms	142 ms	22 ms
10.000	229 ms	411 ms	258 ms	23 ms
100.000	1288 ms	1562 ms	1343 ms	26 ms

Tabelle 6: Messung der Antwortzeiten der Insert-Operation für große Variablen.

Get, große Variablen				
Anzahl Variablen in der Datenbank	**Minimale Antwortzeit**	**Maximale Antwortzeit**	**Durchschnittliche Antwortzeit**	**Zeit für RRS-interne Berechnungen**
10	73 ms	185 ms	92 ms	34 ms
100	62 ms	104 ms	85 ms	35 ms
1.000	68 ms	129 ms	99 ms	38 ms
10.000	242 ms	396 ms	269 ms	39 ms
100.000	1753 ms	2088 ms	1830 ms	39 ms

Tabelle 7: Messung der Antwortzeiten der Get-Operation für große Variablen.

Eine Erhöhung der Antwortzeit auf über eine Sekunde tritt hier bereits bei weniger als

100.000 Variablen auf. Für eine Zahl von Variablen um 10^4 ist die Performanz allerdings noch immer unter 300 Millisekunden. Interessant ist die Tatsache, dass das Einfügen bei noch wenigen gespeicherten Variablen in diesem Fall länger dauert als das Abrufen der Werte. Erst wenn viele Variablen in der Datenbank gespeichert sind, bewegt sich das Verhältnis wieder in Richtung der Werte, die für kleine Variablen gemessen wurden. Die Zeit für RRS-interne Berechnungen ist geringfügig gestiegen, allerdings um einen Faktor kleiner als zwei. Zu beachten ist jedoch, dass die SOAP-Nachrichten auf derselben Maschine erzeugt und empfangen wurden. Wird das RRS über das Netzwerk aufgerufen, sind bei größeren Variablen entsprechend längere Zeiten für die Übertragung der Daten einzurechnen.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Konzept für die Datenverwaltung mit Referenzen in Service Composition Engines vorgestellt. Der Ansatz benutzt externe Datenbanken oder andere Datenspeicher, um von den Geschäftsprozessen verwendete Variablen auszulagern und stattdessen per Referenz auf sie zuzugreifen. Zentral dabei war die Einführung eines RRS, das die Verwaltung der Variablen in den Datenspeichern übernimmt und als Schnittstelle zwischen der Service Composition Engine und den Datenspeichern fungiert.

Es wurde ein Konzept vorgestellt, mit dem das RRS in eine bestehende Service Composition Engine, die auf der Verwendung formaler Grammatiken und Automaten zur Definition und Ausführung von Prozessen basiert, eingebunden werden kann. Hierzu wurde ein Referenzbegriff eingeführt, der im Rahmen dieser Arbeit verwendet wurde, und das Schema für solche Referenzen definiert. Die Arbeit hat die Vorteile einer referenzbasierten Datenverwaltung diskutiert und dargelegt, wie die Service Composition Engine mit den Referenzen umgeht und diese mithilfe von Bezeichnern identifiziert und hält.

Anschließend wurde ein Entwurf des RRS als Web-Service vorgestellt und dessen Aufbau und Funktionsweise diskutiert. Es wurden Möglichkeiten gesammelt, mit denen zukünftig eine Ausweitung der Funktionalität des Systems möglich ist. Schließlich wurde anhand mehrerer Testfälle die Korrektheit der zu dieser Arbeit gehörenden Implementierung des RRS gezeigt. Die Arbeit hat Ergebnisse von Performanzprüfungen vorgestellt, diskutiert und ins Verhältnis zueinander gesetzt.

In zukünftigen Arbeiten zu diesem Thema müssen insbesondere zwei Aspekte diskutiert werden: Der Sicherheitsaspekt solcher Systeme und die Ausweitung der referenzbasierten Datenverwaltung auf die Services selbst.

Sicherheitslücken bestehen vor allem bei den Referenzen, da diese momentan die Speicherung aller Verbindungsinformationen in Klartext vorsehen und so insbesondere dort gespeicherte Passwörter einfach ausgelesen werden können. Somit muss eine Maßnahme entwickelt werden, die verhindert, dass nicht autorisierte Instanzen Referenzen auslesen und durch die dort enthaltenen Informationen Zugriff auf den vollständigen Datenspeicher erhalten können.

Damit einher geht möglicherweise der Entwurf eines Systems von Rechten, die an verschiedene Prozesse oder Benutzer vergeben werden können. [WG09] stellt hierzu beispielsweise eine Trennung der Schnittstelle des RRS in ein Retrieval Interface und ein Management Interface vor, sodass nicht alle Prozesse Variablen einfügen, modifizieren oder löschen können. Dieser Ansatz muss jedoch noch mit anderen Möglichkeiten verglichen und in der Praxis erprobt werden.

Soll die Benutzung von und der Zugriff auf Daten mit Referenzen nicht auf die Service Composition Engine beschränkt sein, müssen Möglichkeiten entwickelt werden, wie die aufgerufenen Services diese handhaben können. Die Übertragung der Referenzen selbst stellt kein Problem dar, allerdings müssen die Services mit ihnen umgehen können und hierzu umgeschrieben werden. [WG09] schlägt hierfür die Entwicklung sogenannter Service-Wrapper vor, bleibt dabei jedoch auf konzeptioneller Ebene, sodass eine Realisierung des Ansatzes als Gegenstand weiterer Arbeiten dienen kann.

Weiter vertieft werden kann ferner auch das Abrufen und Modifizieren großer Variablen. Bei großen Datenmengen ist es möglicherweise in vielen Fällen ausreichend, nur Teile der abgelegten XML-Instanzen zu ändern oder aus dem RRS zu laden. In einem solchen Fall wäre es ineffizient, die gesamte Variable zu erfragen, da dies ein erneutes Parsing der gesamten XML-Daten der Variablen nötig macht. Überdies müsste sie die gesamte Variable über das Netzwerk übertragen werden. Stattdessen wäre es sinnvoller, nur Teile von Variablen abrufen oder verändern zu können.

Ebenfalls ungeklärt ist, wie Daten auf Referenzbasis zwischen verschiedenen Prozessen ausgetauscht werden können. Das momentane System sieht diesen Umstand nicht vor, sodass eine Prozessinstanz, sobald eine bestimmte Variable nicht mehr benötigt wird, zusammen mit dem Bezeichner auch die Variable aus dem RRS löschen kann. Wenn allerdings die Referenzen zwischen den Prozessen weitergegeben werden können, muss ein Verfahren entwickelt werden, das bestimmt, wann Variablen aus den Datenspeichern entfernt werden können und wann dies nicht erlaubt ist, weil etwa andere Prozesse die Daten noch verwenden. Ein möglicher Ansatz zur Bestimmung des Löschzeitpunkts der Daten wäre etwa die Verwendung verteilter Garbage Collection wie etwa durch Reference Counting [PS95].

Literaturverzeichnis

[BC08] Bharathi, S.; Chervenak, A.; Deelman, E.; Mehta, G.; Su, M.; Vahi, K.: „Characterization of Scientific Workflows“, Third Workshop on Workflows in Support of Large-Scale Science, 2008, S. 1-10

[BE99] Böhnlein, M.; vom Ende, A. U.: „XML – Extensible Markup Language”, Wirtschaftsinformatik, Volume 41, Springer Fachmedien Wiesbaden, 1999, S. 274-276

[DH07] DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W.: „Dynamo: Amazon's Highly Available Key-value Store“, SOSP, Volume 7, 2007, S. 205-220

[EM06] Eckstein, R.; Mordani, R.: „Introducing JAX-WS 2.0 With the Java SE 6 Platform, Part 1”, http://www.oracle.com/technetwork/articles/javase/jax-ws-2-141894.html, aufgerufen am 22.10.2013

[EY93] Eyre-Todd, R. A.: „The Detection of Dangling References in C++ Programs“, ACM Letters on Programming Languages and Systems, Volume 2, 1993, S. 127-134

[FK99] Florescu, D.; Kossman, D.: „Storing and Querying XML Data using an RDMBS”, Data Engineering Bulletin, Volume 22, IEEE, 1999, S. 27-34

[FL80] Fischer, C. N.; LeBlanc, R. J.: „The Implementation of Run-Time Diagnostics in Pascal“, IEEE Transactions on Software Engineering, Volume SE-6, 1980, S. 313-319

[FU01] Fujimoto, R. M.: „Parallel Simulation: Parallel and Distributed Simulation Systems“, Proceedings of the 33rd conference on Winter simulation, 2001, S. 147-157

[GD05] Graham, S.; David, D.; Simeonov, S.; Daniels, G.; Brittenham, P.; Nakamura, Y.; Fremantle, P.; König, D.; Zentner, C.: „Building Web Services with Java”, Sams Publishing, 2005

[GH07] Grün, C.; Holupirek, A.; Scholl, M. H.: „Visually Exploring and Querying XML with BaseX”, LNI, Volume 103, Gesellschaft für Informatik, 2007, S. 629-632

[GL12] Görlach, K.; Leymann, F.: „Dynamic Service Provisioning for the Cloud“, Proceedings of the 9th IEEE International Conference on Service Computing, 2012, S. 555-561

[GL13] Görlach, K.; Leymann, F.; Claus, V.: „Unified Execution of Service Compositions“, Processdings of the 6th IEEE International Conference on Service Oriented Computing & Applications, SOCA 2013 (to appear), http://www.iaas.uni-stuttgart.de/institut/mitarbeiter/goerlach/UnifiedExecutionOfServiceCompositions.pdf

[GR81] Gray, J. N.: „The Transaction Concept: Virtues and Limitations”, Proceedings of the 7th International Conference on Very Large Databases, 1981, S.144-154

[GR10] Grün, C.: „Storing and Querying Large XML Instances”, Dissertation, Universität Konstanz, 2010

[HA05] Havey, M.: „Essential Business Process Modeling“, O'Reilly Media, 2005

[HE01] Herrmann, D.: „C++ für Naturwissenschaftler“, Prentice Hall, 2001

[HN68] Hart, P. E.; Nilsson, N. J.; Raphael, B.: „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“, IEEE Transactions on Systems Science and Cybernetics, Volume 4, 1968, S. 100-107

[HR01] Härder, T.; Rahm, E.: „Datenbanksysteme – Konzepte und Techniken der Implementierung“, Springer-Verlag, 2001

[JE07] Jordan, D.; Evdemon, J.: „Web Services Business Process Execution Language Version 2.0“, OASIS Standard, 2007

[JO08] Josuttis, N.: „SOA in der Praxis – System-Design für verteilte Geschäftsprozesse”, dpunkt.verlag, 2008

[KM99] Kanne, C.; Moerkotte, G.: „Efficient Storage of XML data”, Universität Mannheim Technical Report, 1999

[KO05] Koch, C.: „A New Blueprint for the Enterprise”, CIO Magazine, March Issue, 2005

[LO75] Lomet, D. B.: „Scheme for Invalidating References to Freed Storage“, IBM Journal of Research and Development, Volume 19, 1975, S. 26-35

[NE02] Newcomer, E.: „Understanding Web Services – XML, WSDL, SOAP, and UDDI”, Pearson Education, 2002

[PA06] Pesic, M.; van der Aalst, W. M. P.: „A Declarative Approach for Flexible Business Processes Management“, Business Process Management Workshops, 2006

[PE84] Pearl, J.: „Heuristics: Intelligent Search Strategies for Computer Problem Solving“, Addison Wesley Longman Publishing Co, 1984

[PS95] Plainfossé, D.; Shapiro, M.: „A Survey of Distributed Garbage Collection Techniques“, Lecture Notes in Computer Science, Volume 986, Springer-Verlag Berlin Heidelberg, 1995, S. 211-249

[PZ08] Pautasso, C.; Zimmerman, O.; Leymann, F.: „RESTful Web Services vs 'Big' Web Services: Making the Right Architectural Decision”, Proceedings of the 17th international conference on World Wide Web, 2008, S. 805-814

[RE00] Reese, G.: „Database Programming with JDBC and Java“, O'Reilly & Associates, 2000

[ST10] Stroustrup, B.: „Die C++ Programmiersprache“, Addison-Wesley, 2010

[UO01] UN/CEFACT; OASIS: „ebXML Business Process Specification Schema Version 1.01”, http://www.ebxml.org/specs/ebBPSS.pdf, abgerufen am 09.11.2013

[VS08] Vrhovnik, M.; Schwarz, H.; Radeschiitz, S.; Mitschang, B.: „An Overview of SQL Support in Workflow Producs“, IEEE 24th International Conference on Data Engineering, 2008, S. 1287-1296

[WC05] Weerawarana, S.; Curbera, F.; Leymann, F.; Storey, T.; Ferguson, D. F.: „Web Services Platform Architecture”, Pearson Education, 2005

[WG09] Wieland, M.; Görlach, K.; Leymann, F.: „Towards Reference Passing in Web Service and Workflow-based Applications“, IEEE International Enterprise Distributed Object Computing Conference, 2009, S. 109-118

[WW04] World Wide Web Consortium: „Web Services Glossary”, W3C Working Group Note 11 2004, 2004

[WW06] World Wide Web Consortium: „Web Services Addressing 1.0 – Core“, W3C Recommendation 9 May 2006, 2006

[WW08] World Wide Web Consortium: „Extensible Markup Language (XML) 1.0 (Fifth Edition)”, W3C Recommendation 26 November 2008, 2008

[YA12] The YAWL Foundation: „YAWL – User Manual, Version 2.3“, 2012

Erklärung

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

Ebersbach, den 27. November 2013 _____