Erschienen in: Andreas Heuer, Frank Leymann und Denny Priebe (Herausgeber). *Tagungsband der 9. GI-Fachtagung „Datenbanksysteme in Büro, Technik und Wissenschaft“* (BTW 2001), Informatik Aktuell, 7.-9. März 2001, Oldenburg. Seiten 322-260. Berlin · Heidelberg · New York [u.a.]: Springer Verlag. ISBN 3-540-41707-9

Komponenten-Middleware

Der nächste Schritt zur Interoperabilität von IT-Systemen

Jochen Rütschlin[1,2], Günter Sauter[1], Jürgen Sellentin[1], Klaudia Hergula[1], Bernhard Mitschang[2]

[1]DaimlerChrysler AG
Forschung und Technologie
Labor IT for Engineering
Abt. Prozesskette Produktentwicklung
(FT3/EK)
Postfach 2360, D-89013 Ulm

[2]Universität Stuttgart
Institut für Parallele und Verteilte
Höchstleistungsrechner (IPVR)
Abt. Anwendungssoftware
Breitwiesenstraße 20-22,
D-70565 Stuttgart

{jochen.ruetschlin, guenter.sauter, juergen.sellentin, klaudia.hergula}
@DaimlerChrysler.com, bernhard.mitschang@informatik.uni-stuttgart.de

Kurzfassung: In diesem Papier stellen wir eine erste Konzeption für eine komponentenbasierte Middleware vor. Dabei verwenden wir neutrale Daten- und Beschreibungsmodelle, um eine Abstraktion bzgl. bestehender Komponentenmodelle zu erlangen. Kernpunkte in unserer Architektur sind die Komponentenschnittstellen, das auf SOAP basierende Kommunikationsprotokoll und ein *Corporate Repository*.

1 Motivation

Zu Beginn der rechnergestützten Informationsverarbeitung waren Anwendungen überwiegend voneinander unabhängig entwickelte, auf einen sehr begrenzten Bereich beschränkte Software-Lösungen. Die so entstandene Problematik der sog. Inselsysteme zog die Anforderung der Interoperabilität über Anwendungsbereiche hinweg nach sich. Dies wurde zum Einen durch die sog. Ein-System-Schnittstellen, zum Anderen durch Middleware-/Integrationslösungen adressiert.

Im Falle der Ein-System-Schnittstelle werden in steigendem Maße Standardsoftwareprodukte wie CATIA® von Dassault Systemes oder SAP/R3® von SAP verwendet. Obwohl diese Produkte ein sehr breites Anwendungsspektrum abdecken, erfordern wettbewerbsrelevante Spezialprobleme dennoch eigene Lösungen. Um ein erneutes Entstehen von Insellösungen durch das Nebeneinander von Standardsoftware und spezifischen Lösungen zu verhindern, wird ein Zusammenbringen dieser beiden Welten benötigt.

Auf der anderen Seite wird bei den Middleware- und Integrationslösungen versucht, bestehende Systeme trotz ihrer Heterogenität miteinander zu koppeln und durch Daten- und Funktionsintegration ein sog. *Single-System-Image* bereitzustellen [HH00, Sa98]. Dabei entsteht zum Einen das Problem, dass die Zusammenführung der Systeme nur über die Integrationsschicht erfolgen kann wodurch keine direkte Interaktion zwischen den Systemen möglich ist. Zum Anderen ist

heute in den meisten Fällen bei diesen Ansätzen eine eindeutige Festlegung der Semantik sowohl von Daten als auch von Funktionen nicht gegeben.

Aus diesen Problemen heraus wollen wir uns die Frage stellen, wie Systemlandschaften in der Zukunft aussehen sollen, um die angesprochenen Schwierigkeiten bei der Systemerweiterung und -integration künftig zu vermeiden. Somit steht im Fokus dieser Arbeit nicht der Integrationsaspekt von bestehenden Altsystemen, sondern vielmehr der Anspruch, durch die Definition einer geeigneten Architektur die Interoperabilität zukünftiger Anwendungen sicher zu stellen. Schon aus der Problemstellung heraus scheint der Einsatz einer Komponentenarchitektur naheliegend. Eine der zentralen Ideen eines Komponentenmodells ist die Bereitstellung einer syntaktisch standardisierten Schnittstelle für einen bestimmten Funktionsumfang mit dem Ziel, die so entstehenden Komponenten einfach auf einer gleichberechtigten horizontalen Ebene miteinander verbinden zu können (im Gegensatz zum vertikalen Integrationsansatz).

Komponentenmodelle gibt es schon seit geraumer Zeit. Aber gerade die drei am weitesten verbreiteten Modelle CORBA (1991), DCOM (1996) und EJBs (1997) weisen einige Mängel auf, insbesondere bei der semantischen Beschreibung und der strikten Trennung zwischen Beschreibungs- und Ausführungsmodell. Hinzu kommt noch eine mehr oder weniger ausgeprägte Unverträglichkeit zwischen den Modellen, so dass wir versuchen, ein Komponentenmodell zu entwerfen, in dem diese Mängel nicht auftreten. Dazu müssen wir zum Einen die Frage nach einer geeigneten Komponentenschnittstelle, zum Anderen nach einem geeigneten Kommunikationsprotokoll zwischen den Komponenten klären.

In diesem Beitrag wollen wir aus der oben beschriebenen Problematik in Kapitel 2 die Anforderungen an unser Komponentenmodell ableiten und in Kapitel 3 kurz auf bereits bestehende Konzepte und Standards eingehen. Die Idee unseres Lösungsansatzes stellen wir in Kapitel 4 dar. Kapitel 5 schließt den Beitrag mit einer Zusammenfassung und dem Ausblick.

2 Anforderungen

In der Einleitung haben wir erkannt, dass die Architektur künftiger IT-Systeme auf einer Art Bausteinprinzip basieren sollte. Diese recht allgemeine Forderung wollen wir im Folgenden konkretisieren. In Anlehnung an die übliche Begriffsbildung wollen wir dabei den Begriff der Komponente verwenden. Da es bisher keine allgemeingültige Definition dieses Begriffes gibt, stellen die folgenden Anforderungen gleichzeitig unsere Definition von Komponente dar. Prinzipiell wollen wir bei der Diskussion zwei Bereiche unterscheiden: Die Beschreibung von Schnittstellen einzelner Bausteine/Komponenten sowie die Infrastruktur (sog. Middleware) zur Kommunikation zwischen diesen.

2.1 Beschreibung von Schnittstellen

Die Beschreibung der Funktionalität einer Komponente sollte im Wesentlichen über die Spezifikation ihrer Schnittstelle erfolgen (sog. *Black-Box*-Ansatz). Hierbei reicht es aber nicht aus, nur die Syntax von Funktionsaufrufen festzulegen. In

diesem Fall bliebe das Verhalten (und damit auch die eigentliche Funktionalität) des Bausteines unbekannt. Vielmehr ist eine Beschreibung von Syntax und Semantik der enthaltenen Funktionalität nötig. Und genau hier entsteht eines der wesentlichen Probleme. Es gibt bereits einige Sprachen zur Definition der Syntax von Daten oder Funktionen (z. B. SQL [ANSI99] und EXPRESS [ISO94] für Daten oder die IDL von CORBA [OMG99b] für Signaturen). Allerdings fehlen technologie-unabhängige Standards oder Beschreibungsmodelle zur Spezifikation der Semantik von Daten und Signaturen. Unserer Meinung nach lassen sich praxistaugliche Verfahren zur (maschinenlesbaren und -interpretierbaren) Beschreibung der Semantik momentan nicht realisieren. Diese Ansätze lösen in der Praxis nicht einmal einfache Probleme [Sa98]; es sei hier nur das Synonym/Homonym Problem genannt. Insofern bleibt hier als Mindestanforderung nur die umgangssprachliche und damit fehleranfällige Beschreibung. Eine Verbesserung bei der Erfüllung der Anforderungen würde erreicht durch die Einführung von Klassifikationskriterien oder Standards.[1] Wichtig ist jedoch in beiden Fällen, dass sowohl die Syntax als auch die Semantik vom Ausführungsmodell neutral beschrieben werden kann, damit die Spezifikationen auch nach Einführung einer neuen Technologie ihre Gültigkeit behalten.

2.2 Kommunikationsinfrastruktur

Neben der Beschreibung einzelner Komponenten gilt es auch, die Interaktion zwischen diesen zu regeln. Im Wesentlichen erfordert dies die Definition eines Kommunikationsmodells mit zentralen Diensten. Letztere umfassen Funktionen, die aufgrund ihrer Koordinationsaufgaben zentral angeboten werden müssen. Dies sind beispielsweise Dienste wie die globale Verwaltung von Transaktionen, Sicherheitsfunktionen sowie ein *Corporate Repository* mit Metadaten zu allen verfügbaren Komponenten. Das Kommunikationsmodell zusammen mit den zentralen Diensten bezeichnen wir dann als Middleware-Lösung.

Bereits in der Einleitung haben wir erkannt, dass die Beschränkung auf eine Kommunikationstechnologie nicht ausreicht. So entstehen z. B. stetig neue Kommunikationstechniken. Um so bedeutender ist es, dass die gewählte Middleware-Lösung flexibel und erweiterbar bzgl. der zugrundeliegenden Technologien ist und damit einen gewissen Grad an Offenheit bietet. Durch diese Flexibilität kann auf einfache Systemanforderungen auch mit einfachen und schlanken Umsetzungen reagiert werden. Die Middleware-Lösung sollte insbesondere auch unabhängig von einer Programmiersprache oder einem Betriebssystem sein. Wir benötigen also ein relativ abstraktes Kommunikationsmodell (Beschreibungsmodell), das auf verschiedene Kommunikationstechnologien (Ausführungsmodell) abgebildet werden kann. Diese Abbildung muss natürlich auch die Informationen aus dem Beschreibungsmodell für einzelne Komponenten berücksichtigen (Abschnitt 2.1). In diesem Sinne kann man die zentralen Dienste auch als ausgezeichnete Komponenten betrachten.

[1] z. B. in Anlehnung an die CR-Klassifikation bei Fachliteratur [ACM98] oder das STEP Application Protocol 214 für Daten der Automobilindustrie [ISO97].

3 Diskussion bestehender Ansätze

Prinzipiell sind die im letzten Kapitel dargestellten Anforderungen (jede für sich) nicht neu. Insofern gibt es Lösungsansätze, die bereits einige unserer Anforderungen abdecken. Es stellt sich jedoch immer die Frage, wie sich die Gesamtheit aller Anforderungen abdecken lässt. In diesem Sinne wollen wir existierende Arbeiten betrachten und die Möglichkeiten und Grenzen ihres Einsatzes diskutieren.

3.1 Kommerzielle Komponentenmodelle

Betrachtet man den Markt für kommerziell verfügbare Software, so fallen im wesentlichen drei Konzepte/Produkte auf, die mit dem Anspruch eines Komponentenmodells antreten: CORBA (Component Object Request Broker Architecture, [OMG99b]), (D)COM (Distributed Common Object Model, [MS00]) und EJB (Enterprise JavaBeans, [Sun00]). Alle drei Ansätze bieten ein mehr oder weniger ausgeprägtes Beschreibungsmodell sowie grundlegende Schnittstellen für Basisfunktionalität von Komponenten. Auf den ersten Blick erfüllen sie damit einen Großteil unserer Anforderungen. Im Detail gibt es jedoch wesentliche Defizite [Se00]. So ist das Beschreibungsmodell jeweils zu stark mit dem Ausführungsmodell verknüpft. Es kann daher nicht die geforderte Abstraktion erreichen, um als globales Beschreibungsmodell zu dienen wie die folgende, exemplarische Zusammenstellung zeigt:

- EJBs werden direkt in der Programmiersprache Java beschrieben.
- Die Spezifikation der CORBA Interface Definition Language (IDL) und der CORBA Components [OMG99a] basiert zu stark auf dem verteilten Ausführungsmodell sowie den CORBA Services. So muss z. B. während der Modellierung mit IDL bereits das Schlüsselwort *oneway* verwendet werden, um für die Laufzeit einen eingeschränkten *Quality of Service* zu wählen. Positiv zu bewerten ist zwar der Ansatz, in Form sog. CORBA Facilities standardisierte Funktionalität für einzelne Anwendungsbereiche anzubieten. Deren Beschreibung ist aber viel zu sehr auf die CORBA-Technologie bezogen.
- Microsofts DCOM ist in das Betriebssystems Windows eingebettet. Insofern wird die Anforderung der Interoperabilität überhaupt nicht adressiert.

Obwohl die drei Ansätze jeweils nicht unsere Anforderungen erfüllen, sollten wir trotzdem bemüht sein, die zugrundeliegenden Ausführungsmodelle unterstützen zu können. Ein großer Teil der existierenden Software basiert auf diesen Technologien. Deswegen sollte es bidirektionale Abbildungen von unserem (quasi globalen) Beschreibungsmodell auf CORBA, EJBs und DCOM geben.

3.2 Forschungsansätze

Vergleichbar mit unserer Zielsetzung wurde von Wiederhold et al. das sog. Megaprogramming entwickelt [WWC92]. Die Funktionen einzelner Komponenten werden dabei als Anweisungen einer Mega-Programmiersprache angesehen, mit

denen ein Mega-Programm geschrieben wird. Zur Ausführung gibt es eine eigene Laufzeitumgebung, welche die einzelnen Funktionsaufrufe an konkrete CORBA- oder DCOM-Komponenten leitet [MBSW99]. Insofern erfolgt auch hier eine klare Trennung zwischen Beschreibungs- und Ausführungsmodell. Entgegen unseren Ideen ist dieser Ansatz aber stark vertikal orientiert, d. h. übergeordnete Mega-Programme benutzen quasi untergeordnete Mega-Module. Wir möchten hingegen ein symmetrisches Modell definieren, bei dem sich gleichwertige Komponenten gegenseitig aufrufen können (vertikal und horizontal).

Natürlich gibt es weitere Arbeiten im Bereich Komponentenmodelle, die wir hier nicht alle aufzählen können. Die bisher betrachteten Ansätze haben jedoch keine gesamtheitliche Lösung unserer Anforderungen unterstützt.

4 Lösungsansatz

In diesem Abschnitt stellen wir einen Lösungsansatz für die in Kapitel 2 beschriebenen Komponenten und die Infrastruktur vor. Dabei verfolgen wir die grundlegende Idee, unseren Ansatz in ein Beschreibungs- und Ausführungsmodell aufzuspalten. Der Vorteil hierbei ist, dass mittels des Beschreibungsmodells von der tatsächlichen Implementierung (Ausführungsmodell) abstrahiert wird.

4.1 Schnittstellen der Komponenten

Bei den Komponenten konzentrieren wir uns auf das Beschreibungsmodell für die Schnittstellen, um alle Implementierungsmöglichkeiten offen zu halten. Ziel ist die Erarbeitung einer abstrakten Beschreibung der Komponentenschnittstelle, mit welcher jede Komponente in derselben Form beschrieben werden kann, unabhängig von ihrer Realisierung. Dies bedeutet, dass wir eine unabhängige (standardisierte) Beschreibungssprache benötigen. Da existierende standardisierte Sprachen meistens jedoch an ein bestimmtes Ausführungsmodell gekoppelt sind (wie z. B. IDL an CORBA), benötigen wir eine weiter abstrahierende Sprache, die im Folgenden erarbeitet wird.

Bei der Definition einer solchen Beschreibungssprache soll darauf geachtet werden, dass sie mit standardisierten Mitteln festgelegt wird, um proprietäre Lösungen zu vermeiden. Des Weiteren muss die Abbildung auf das Ausführungsmodell beschrieben werden können. Zur Erfüllung dieser Kriterien scheint der Einsatz von XML (*Extensible Markup Language*, [W3C98]) eine elegante Lösung darzustellen. XML ermöglicht nicht nur die standardisierte Definition einer eigenen Beschreibungssprache. Viel wichtiger ist die Entwicklung weiterer, verwandter Standards, welche die Anwendung von XML erweitern und erleichtern.

Bei der Definition unserer Beschreibungssprache greifen wir auf Ergebnisse von [HH00] zurück. In dieser Arbeit wurde eine Sprache entwickelt, mit welcher Funktionsaufrufe von Clients auf die APIs mehrerer heterogener Server abgebildet werden können. Wir haben uns zunächst für diesen Lösungsansatz entschieden, da er in der Lage ist, die Heterogenitäten der Server zu überwinden. Da man auch bei den Komponenten von sehr unterschiedlichen Realisierungen ausgehen

muss, können wir demnach mittels der Abbildungssprache von den daraus resultierenden Heterogenitäten abstrahieren. In dieser Abbildungssprache wird die Beschreibung der einzelnen, zu integrierenden Systeme und ihrer Schnittstellen von der Beschreibung der eigentlichen Funktionsabbildung getrennt. Bezüglich des genauen Aufbaus der Sprache sowie erläuternden Beispielen verweisen wir den interessierten Leser auf [HH99]. Diese Systembeschreibung kann für unsere Komponentenschnittstellen übernommen werden. Auf diese Weise können wir zumindest die grundlegende Schnittstelle zum Aufruf von Funktionen der Geschäftslogik einer Komponente beschreiben.

Offensichtlich muss diese Sprache für unsere Anforderungen erweitert und angepasst werden, da neben reinen Funktionsaufrufen u. a. auch die GUI oder Transaktionen beschrieben werden sollen [Rü00]. Grundsätzlich stellt sich die Frage, ob diese Art von Schnittstellen nicht auf Funktionsaufrufe heruntergebrochen werden können. So beinhaltet eine Transaktionsschnittstelle z. B. Operationen wie *commit* und *abort*, die wiederum als Funktionen dargestellt werden können.

Wurde die Sprache gemäß den Anforderungen von Abschnitt 2.1 weiterentwickelt, so erhält man ein Beschreibungsmodell, das die Funktionalität einer Komponente vollkommen unabhängig von der zugrundeliegenden Technologie beschreibt. Um die Verbindung zwischen Beschreibungs- und Ausführungsmodell zu gewährleisten, muss außerdem eine Abbildung zum eingesetzten Ausführungsmodell erfolgen. Hier kann man auf XML-verwandte Standards wie XSLT (*Extensible Stylesheet Language Transformations*, [W3C99]) zurückgreifen, um eine entsprechende Transformation zu definieren und auszuführen.

4.2 Infrastruktur

Gemäß der in der Einleitung zu diesem Kapitel postulierten Unterscheidung zwischen einem Beschreibungs- und einem Ausführungsmodell, geben wir zunächst die abstrakte Beschreibung unserer Middleware-Architektur und gehen in dem darauffolgenden Abschnitt auf eine mögliche Realisierung der Architektur ein. Dabei soll der Fokus nicht so sehr auf die Realisierung der Basiskomponenten gelegt werden, sondern vielmehr auf das Kommunikationsprotokoll für die Interaktion zwischen den Komponenten eingegangen werden.

Architekturbeschreibung. Abbildung 1 zeigt den schematischen Aufbau der vorgeschlagenen Middleware-Lösung. Sie besteht aus fünf zentralen Basiskomponenten, wobei wir das *Component Registry*, *User Directory* und die *Authorization Database* unter dem Begriff des *Corporate Repository* zusammenfassen.

Das *Component Registry* (CR) enthält die Metadaten der Komponenten. Es ist vergleichbar mit dem *RMI registry* von Java oder dem *Naming Service* von CORBA. Nimmt eine Komponente ihren Dienst auf, so meldet sie sich bei dem *Registry* an. Dabei übermittelt sie ihre syntaktischen und semantischen Schnittstellenbeschreibungen dem CR.

Im *User Directory* werden die Benutzerkennungen zusammen mit den dazugehörigen Authentifizierungsinformationen gespeichert. Dabei können einem

Benutzer verschiedene Authentifizierungsinformationen zugeordnet sein, je nachdem von welchem Client-Typ (Endgerät) aus er sich anmeldet (z. B. vom PC mittels Zertifikat, vom PDA mit Passwort und vom WAP-Handy mit PIN).

Die *Authorization Database* enthält die Autorisierungsinformationen der Anwendungen. Im Normalfall werden diese Daten aus administrativen und Performanzgründen in der Anwendung selbst verwaltet. Diese Komponente soll es dennoch kleineren Anwendungen (auf welche diese Restriktionen nicht zutreffen oder die keine eigene Datenhaltung erlauben) ermöglichen, ihre Autorisierungsinformationen in dieser dedizierten Datenbank abzulegen.

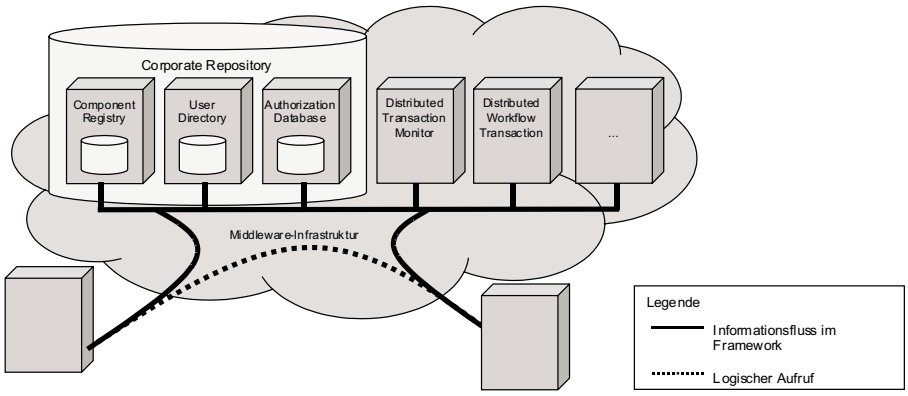

Abb. 1: Aufbau der Komponentenarchitektur (Middleware-Infrastruktur).

Zusätzlich zu dem *Corporate Repository* sind optional weitere Dienste denkbar wie z. B. für verteilte Transaktionen (zwischen den Komponenten) oder zur Koordination komponentenübergreifender Workflows (in diesem Aufsatz gehen wir nicht weiter auf sie ein).

Die zentralen Dienste stellen wieder Komponenten dar. Sie weisen somit – als Gegenstück – die gleichen Schnittstellen auf und unterscheiden sich von normalen Komponenten nur durch die Art der Dienste, die sie den anderen anbieten. Dadurch können wir auch für die Basiskomponenten den Beschreibungsansatz aus Abschnitt 4.1 verwenden und erhalten damit eine homogene Systemsicht.

Die Interaktion zwischen den Komponenten erfolgt über ein noch näher zu spezifizierendes Transportprotokoll (vgl. nächster Abschnitt). Als Grundfunktion kann der Austausch von Nachrichten bzw. der Funktionsaufruf betrachtet werden. Solch ein Aufruf soll in unserer Architektur über die semantische Beschreibung erfolgen („Es wird die Funktionalität *get_part* benötigt“). Dabei wendet sich eine Komponente mit ihrem Funktionsaufruf an das CR, welches mit der semantischen Beschreibung alle momentan verfügbaren Komponenten der gewünschten Funktionalität ermittelt und die Anfrage an diese weiterleitet.

Umsetzung der Architektur. Unsere bisherigen Betrachtungen zu einem konkreten Ausführungsmodell beschränken sich auf das *Corporate Repository* und das Kommunikationsprotokoll. Wie in Abb. 1 beschrieben gliedert sich das *Corporate Repository* in drei Subkomponenten. Für Benutzerinformationen (*User Directory*/Authentifizierung) bietet es sich wegen seiner großen Verbreitung in Unternehmen an, einen Verzeichnisdienst mit LDAP zu verwenden. Berücksichtigt man den Zugriffsmatrix-Charakter von Autorisierungsinformationen, so liegt es nahe, für die Autorisierung eine relationale Datenbank zu verwenden (*Authorization Database*). Für die Realisierung des *Component Registry* kommen unterschiedliche Standardvarianten in Frage (Dateien, Datenbanken, etc.), die allerdings nicht Gegenstand dieses Papiers sein sollen.

Als Kommunikationsprotokoll könnte man sich prinzipiell alle *Remote Procedure Call* (synchron) oder *Message Queuing* Technologien (asynchron) vorstellen. In unserem Ansatz wollen wir das von Microsoft (und IBM) ins Leben gerufene *Simple Object Access Protocol*, kurz SOAP [SOAP00], als Kommunikationsprotokoll verwenden. Dabei handelt es sich um einen in XML eingebetteten Funktionsaufruf, der oft – quasi als spezielle Ausprägung – in Zusammenhang gebracht wird mit einer Einbettung in HTTP. Als Vorteile dieses Ansatzes sehen wir die Möglichkeit einer direkten Verwendung unserer bereits in XML vorliegenden Metainformationen, die Unabhängigkeit von bestehenden Systemen und Komponentenarchitekturen (Abstraktion) sowie die Einfachheit und Erweiterbarkeit des Protokolls. Ein Nebeneffekt von SOAP über HTTP ist die Umgehung der Firewall-Problematik (nur HTTP-Port ist freigeschaltet).

```
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Header>
      <t:Transaction xmlns:t="some-URI" xsi:type="xsd:int"
        SOAP-ENV:mustUnderstand="1">
         0815
      </t:Transaction>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <m:SetPartName xmlns:m="Some-URI">
         <no> A1638204661</no>
         <value>Leuchteneinheit rechts</value>
      </m:SetPartName>
   </SOAP-ENV:Body>
```

Abb. 2: Beispiel eines in SOAP gekapselten Funktionsaufrufes.

Wenden wir uns dem Aufbau von SOAP zu. Grundkonstrukt ist der *Envelope*, der die SOAP-Nachricht darstellt. Er besteht analog zur HTML-Spezifikation aus *Header* und *Body* (Abb. 2). Der optionale *Header* dient der Übermittlung ergänzender Informationen. Solche Ergänzungen sind beispielsweise Authentifi-

zierungs- oder (wie in Abb. 2 gezeigt) Transaktionsinformationen. Des Weiteren bietet SOAP die Attribute *mustUnderstand* und *actor* für Header-Elemente an (für Details siehe [SOAP00]). Der *Body* enthält den eigentlichen Funktionsaufruf, wobei die Attribute des Funktionsaufrufes durch ein den Funktionsnamen repräsentierendes Tag geklammert werden (*SetPartName*). Der Aufbau der Funktionssignatur (also die Serialisierungsregeln einer SOAP-Nachricht) wird durch die hinter dem globalen XML-Attribut *encodingStyle* (Element Envelope) stehenden XML-Schema-Definition beschrieben. D. h., wir können die Typdefinitionen wie sie bereits bei der Funktionsbeschreibung in dem Ansatz von [HH00] verwendet wurden, (ggf. mit kleineren Anpassungen) direkt in SOAP nutzen.

Betrachtet man den Verlauf einer SOAP-Nachricht, so wird eigentlich kein Funktionsaufruf im herkömmlichen Sinne, sondern nur eine Einweg-Übertragung vom Sender zum Empfänger unterstützt. In Kombination mit der *Request/Response*-Funktionalität von HTTP und einem extra Funktionsaufruf als Resultat zurück an den Sender, kann ein synchroner Aufruf realisiert werden [SOAP00]. Durch Einsatz des *Corporate Repository* sind auch asynchrone Funktionsaufrufe möglich. Es kann nämlich durch seine Vermittlerfunktion beim Nachschlagen der angeforderten Funktionalität wie eine Art *Message Broker* fungieren.

5 Zusammenfassung und Ausblick

In diesem Papier stellen wir eine erste Konzeption für eine komponentenbasierte Middleware vor. Für eine solche Lösung benötigt man einerseits Mechanismen zur Beschreibung der Komponenten, andererseits eine geeignete Kommunikationsinfrastruktur. Konkret haben wir für die Beschreibung einen XML-basierten Ansatz gewählt, bei dem die syntaktischen und semantischen Schnittstelleninformationen in einem *Corporate Repository* abgelegt werden. Für die Kommunikation verwenden wir das bidirektionale *Request/Response*-Protokoll SOAP. Diesen eher statischen Charakterzügen des Ansatzes steht der dynamische Aspekt des Nachrichten-*En-/Decoding* gegenüber: plattformspezifische Funktionsaufrufe werden innerhalb der Komponente durch ein spezielles Modul in das plattformunabhängige SOAP-Format kodiert bzw. in umgekehrter Richtung dekodiert.

Als Vorteile dieses Ansatzes sehen wir die Verwendung von XML als Metasprache, durch deren Flexibilität einfache Anpassungen an unsere (oder auch neue) Anforderungen möglich sind. Dies kann u.U. bei einer Verwendung von XSLT beim *En-/Decoding* zum Tragen kommen. Durch den Einsatz von SOAP bleibt man in einer homogenen Beschreibungswelt (XML für die Komponentenbeschreibung als auch für die Kommunikation) und nutzt mit diesem Protokoll ein abstraktes/anpassbares Kommunikationsmodell. Insgesamt werden damit bereits vorhandene Technologien benutzt mit dem weiteren Vorteil, dass auf die dafür vorhandenen Werkzeuge zurückgegriffen werden kann.

Als nächsten Schritt wollen wir prüfen, ob der UDDI-Ansatz (*Universal Description, Discovery and Integration*, http://www.uddi.org/) – dem ja ein ähnlicher Gedanke zugrunde liegt, wie bei unserer Architektur – ergänzend oder auch alternativ zu unserem *Corporate Registry* verwendet werden kann.

Referenzen

ACM98: The ACM Computing Classfication System [1998 Version]. http://www.acm.org/class/1998/

ANSI99: American National Standards Institute, Inc. (1999). Database Languages – SQL – Part 2. Foundation (SQL/Foundation). ANSI/ISO/IEC9075-2-1999. In: Amercian National Standard for Information Technology, approved Dec-1999.

HH99: K. Hergula, T. Härder (1999) *Ein Abbildungsbeschreibung zur Funktionsintegration in heterogenen Anwendungssystemen*. Proc. 4. Workshop „Föderierte Datenbanken". TU-Berlin, 1999, S. 71-88.

HH00: K. Hergula, T. Härder (2000) *A Middleware-Approach for Combining Heterogeneous Data Sources – Integration of Generic Query and Predefined Function Access*. In: Proc. of the 1st Int. Conf. on Web Information Systems Engineering (WISE). Seiten 22-29.

ISO94: ISO 10303 (1994). Industrial automation systems and integration – Product data representation and exchange – Part 11: „Description methods: The EXPRESS language reference manual", International Standard.

ISO97: ISO CD 10303 (1997). Industrial Automation Systems and Integration – Product Data Representation and Exchange – Part 214: „Core Data for Automative Mechanical Design Processes", Committee Draft.

MBSW99: L. Melloul, D. Beringer, N. Sample und G. Wiederhold (1999). *CPAM, A Protocol for Software Composition*. In: M. Jarke und A. Oberweis (Hrsg). Proc. of the 11th CAiSE, Heidelberg, Juni 1999. Nr. 1626 in LNCS, S. 317-332. Springer.

MS00: Mircosoft Corporation (2000). Microsoft COM Technologies – DCOM. http://www.microsoft.com/com/tech/DCOM.asp

OMG99a: The Object Management Group – OMG (Februar 1999). CORBA Components – Joint Revised Submission. OMG TC document 99-02-05. http://cgi.omg.org/cgi-bin/doc?orbos/99-02-05

OMG99b: The Object Management Group – OMG (Oktober 1999). The Common Object Request Broker: Architecture and Specification, Revision 2.3.1. http://cgi.omg.org/cgi-bin/doc?formal/99-10-07

Rü00: J. Rütschlin (2000) *The Requirements for a Component-based Architecture*. In: R. Anderl, C. Frick, A. Katzenbach und J. Rix (Hrsg.). Proc. of the ProSTEP Science Days 2000 „SMART Engineering", 13./14. Sep. 2000 bei DaimlerChrysler, Stuttgart. S. 252-260. Darmstadt: ProSTEP e.V.

Sa98: G. Sauter (1998) Interoperabilität von Datenbanksystemen bei struktureller Heterogenität, Dissertation, infix Verlag.

Se00: J. Sellentin (2000). Datenversorgung komponentenbasierter Informationssysteme. Springer Verlag.

SOAP00: Simple Object Access Protocol (SOAP). W3C Note, 08. Mai 2000. http://www.w3.org/TR/SOAP/

Sun00: Sun Microsystems (2000). The Enterprise JavaBeans Technology. http://java.sun.com/products/ejb/

W3C98: World Wide Web Consortium (1998). Extensible Markup Language (XML) 1.0. W3C Recommendation. http://www.w3.org/TR/REC-xml

W3C99: World Wide Web Consortium (1999). XSL Transformations (XSLT) Version 1.0. W3C Recommendation. http://www.w3.org/TR/xslt

WWC92: G. Wiederhold, P. Wegner und S.1 Ceri (1992). Towards Megaprogramming: A Paradigm for Component-Based Programming. Communications of the ACM, 35 (11):89-99, November 1992.