FAKULTÄT FÜR !NFORMATIK

Tool Supported Workflow Integration of RESTful Web Services

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Alexander Bruckner

an der:
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Univ.-Ass. Dipl.-Ing. Martin Vasko

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 14. 05. 2009 Alexander Bruckner

Abstract

The Representational State Transfer (REST) defines a set of architectural constraints for the implementation of distributed hypertext systems, and forms the basis for the implementation of the modern World Wide Web (WWW). Web services, primarily grounded on the principles of REST and base technologies of the WWW, such as HTTP, URI and XML are called RESTful Web services. Due to their uniform interface, formed by the methods of the HTTP protocol specification, the integration of such Web services is fostered by most modern programming languages. Representing a light weight implementation alternative to overly complex WS* Web services RESTful Web services are increasingly gaining relevance in Web 2.0 environments as well as enterprise level applications.

This thesis deals with the issues of RESTful Web service integration in formally described, executable business workflows. Whereas the WS-BPEL specification or the Windows Workflow Foundation framework provide established workflow models for WS* based Web services, comparable solutions for RESTful Web services are rare in scientific research and commercial software products. The conceptual part of the thesis compares the theoretical foundations of the implementation styles REST and WS* and the compliance of real world RESTful Web services with the architectural constraints of REST, in respect of their workflow integratability. A detailed survey of description languages for RESTful Web services forms the basis for the first part of the implementation - a code generation library for WADL (Web Application Description Language) and WF (Windows Workflow Foundation) activity components, facilitating the invocation of RESTful Web services within WF workflows. The second part of the thesis implementation introduces a Web service integration wizard, allowing for dynamic integration of heterogeneous Web services in expressFlow, a web scale visual workflow design application.

Kurzfassung

Der Representational State Transfer (REST) definiert eine Menge architektureller Constraints als Basis für die Implementierung von verteilten Hypertext Systemen und dient als theoretische Grundlage für die Implementierung des modernen World Wide Web (WWW). Web Services, die primär auf den Prinzipien von REST und den Basis Technologien des WWW, wie HTTP, URI and XML basieren, werden RESTful Web Services genannt. Auf Grund ihrer einheitlichen Schnittstelle, gebildet durch die Methoden des etablierten HTTP Protokolls, wird die Integration derartiger Services in den meisten modernen Programmiersprachen unterstützt. Als schlanke Implementierungs Alternative zu den übermäßig komplexen WS* Web Services gewinnen RESTful Web Services sowohl in Web 2.0 Umgebungen als auch professionellen Unternehmensanwendungen zunehmend an Bedeutung.

Diese Diplomarbeit beschäftigt sich mit Möglichkeiten der Integration von RESTful Web Services in formal beschriebenen, ausführbaren Business Prozessen. Während der WS-BPEL Standard oder das Windows Workflow Foundation Framework etablierte Workflow Modelle für WS* basierte Web Services bieten, gibt es im wissenschaftlichen und kommerziellen Umfeld wenige vergleichbare Ansätze für RESTful Web Services. Im konzeptionellen Abschnitt der Arbeit werden die Grundlagen der Implementierungs Stile REST und WS* verglichen und die Einhaltung der architekturellen Prinzipien von REST in realen Anwendungen im Hinblick auf deren Integrierbarkeit in Workflows untersucht. Eine detaillierte Auseinandersetzung mit Beschreibungssprachen für RESTful Services bietet die Grundlage für den ersten Teil der Implementierung - eine Bibliothek zur automatischen Generierung von WADL (Web Application Description Language) Dokumenten und WF (Windows Workflow Foundation) Aktivitäten für den Aufruf von RESTful Web Services in WF Workflows. Ein Web Service Integrations Wizard, zur dynamischen Integration heterogener Web Services in der Web basierten visuellen Workflow Design Applikation expressFlow, bildet den zweiten Teil der Implementierung.

Danksagungen

Ich widme diese Arbeit meiner Familie, die mir durch ihre Unterstützung immer ermöglicht hat, mich auf die Aufgaben des Studiums zu konzentrieren und mir immer den nötigen Rückhalt in wichtigen Entscheidungen geboten hat.

Ich möchte mich auch bei allen StudienkollegInnen und Freunden für die oft sehr produktive, immer spannende Zusammenarbeit in gemeinsamen universitären Projekten bedanken. Das gemeinsame Lösen von Aufgabenstellungen in Gruppen war mit Sicherheit einer der interessantesten Aspekte meiner Studienzeit.

Ein besonderer Dank gilt auch Univ.Prof. Dr. Schahram Dustdar, der in seinen interessanten Vorlesungen zum Gebiet der Service Orientierten Architekturen und Web Service Technologien in der abschließenden Phase meines Studiums den Anstoß zur Themenauswahl dieser Arbeit gegeben hat.

Abschließend möchte ich natürlich Martin Vasko danken, der mir durch viele gute Ratschläge und produktive Treffen während der Implementierung und konzeptionellen Ausarbeitung der Diplomarbeit eine optimale Betreuung geboten hat.

Contents

List of Figures

List of Tables

1. Introduction

There have always been things
which people are good at,
and things computers have been good at,
and little overlap between the two.

Tim Berners-Lee

In 1945, the last year of the second world war, an article by Vannevar Bush, about a photo-electrical mechanic device called Memex, allowing for creating and following links between microfilm documents, denotes the start of the history of the World Wide Web (WWW) [93]. 20 years later Ted Nelson coins the term *Hypertext* in his publication *A File Structure for the Complex, the Changing, and the Indeterminate*, followed two years later by the first Hypertext editing system *FRESS*, by Andy van Damm et al.. In 1969 the first ARPANET (Advanced Research Projects Agency Network) nodes are connected forming the basis for the development of the Internet.

Motivated by the challenges of information distribution in complex scientific projects at CERN, Tim Berners-Lee publishes an article about a distributed hypertext system in 1989. One year later he implements the first web-server/-browser/-editor, called WorldWideWeb (initially without spaces).

In 2000 Roy Fielding, one of the main authors of HTTP and co-founder of the Apache HTTP server project, introduces the *Representational State Transfer* (REST) [30], an architectural style for distributed hypermedia systems, finally segueing from the historical excursion of the WWW into one of the title worlds of this thesis.

Along with the evolvement of the web not only as distribution system for hypermedia, but also as transport medium for distributed business applications, the term *Web* *services* emerged in the early years of 2000, as an implementation technology for Service Oriented Architectures (SOA). Based on open technology standards such as XML, SOAP and WSDL Web services were meant to provide an ideal technology basis for development of loosely coupled interoperable business applications.

But the overly complex technology stack, formed by the core Web services standards and their extensions, referred to as the WS* standards, also detained many developers from writing WS* based applications, fostering an alternative approach for Web services implementation. So-called *RESTful Web services*, based on the theoretical principles of REST and the basic technology standards of the WWW - HTTP, URI and XML, provide a light weight alternative to WS* based Web services. The relevance of these services is underpinned by usage statistics of main stream Web services providers, providing their services through WS* and REST interfaces[1], and recent industry efforts [43, 44, 55].

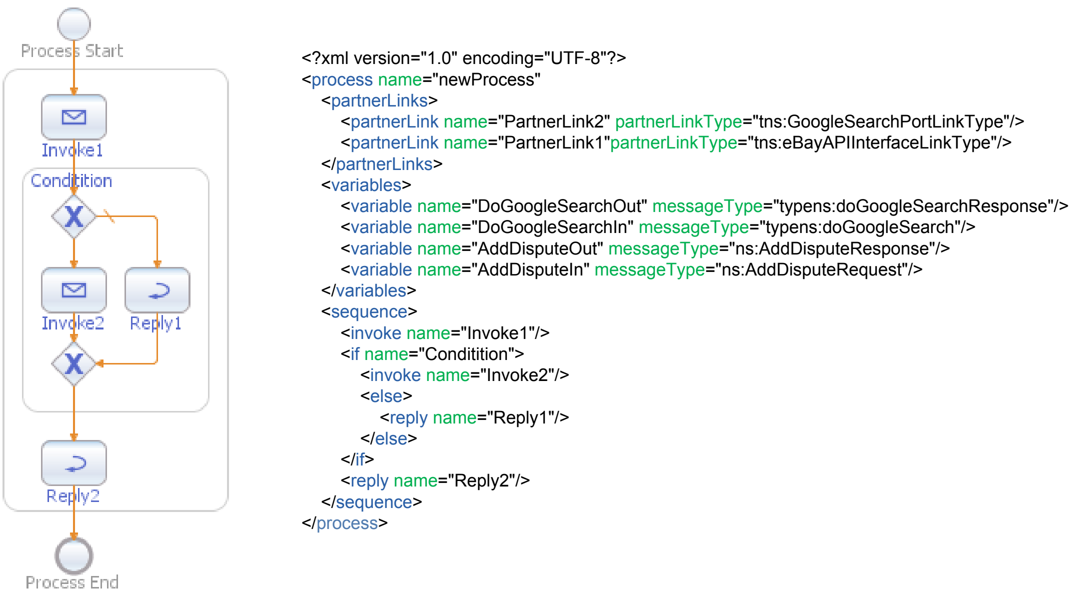
1.1. Problem Definition and Motivation

Web service *orchestration* denotes the aggregation of Web services provided by one ore more parties, into a formally described, executable *workflow*, from the perspective of one controlling party [76]. Such a workflow is commonly specified within an abstract description model (for example WS-BPEL [67], Windows Workflow Foundation [52]), allowing for definition of its execution structure, data flow and involved parties.

A workflow specification is created by means of a visual design tool and executed within a workflow runtime environment. Figure 1.1 depicts the graphical and XML representations of such a specification in a simplified WS-BPEL notation.

While the integration of WS* based Web services is well fostered by existing workflow models and according tools, RESTful Web services are, despite their increasing popularity, often neglected in this context. Motivated by this situation the subsequent issues around the workflow integration of RESTful Web services will be covered in this thesis:

[1] http://www.oreillynet.com/pub/wlg/3005



```xml
<?xml version="1.0" encoding="UTF-8"?>
<process name="newProcess"
   <partnerLinks>
      <partnerLink name="PartnerLink2" partnerLinkType="tns:GoogleSearchPortLinkType"/>
      <partnerLink name="PartnerLink1"partnerLinkType="tns:eBayAPIInterfaceLinkType"/>
   </partnerLinks>
   <variables>
      <variable name="DoGoogleSearchOut" messageType="typens:doGoogleSearchResponse"/>
      <variable name="DoGoogleSearchIn" messageType="typens:doGoogleSearch"/>
      <variable name="AddDisputeOut" messageType="ns:AddDisputeResponse"/>
      <variable name="AddDisputeIn" messageType="ns:AddDisputeRequest"/>
   </variables>
   <sequence>
      <invoke name="Invoke1"/>
      <if name="Conditition">
         <invoke name="Invoke2"/>
         <else>
            <reply name="Reply1"/>
         </else>
      </if>
      <reply name="Reply2"/>
   </sequence>
</process>
```

Figure 1.1.: Sample Workflow in BPEL Notation - Graphical and XML Representation

- What are the prevailing characteristics of real world RESTful service implementations, and how does their (non-)compliance with the architectural constraints of REST affect workflow integration ?
- How do the two implementation models for Web services, REST and WS*, differ conceptionally ?
- Do the elements of established description formats for Web services suffice for formal specification of RESTful Web services, in order to ease their tool supported workflow integration ?
- How can RESTful Web services be invoked from a workflow deployed into a workflow runtime environment, i. e. the Windows Workflow Foundation framework ?
- How can RESTful and WS* based Web services be integrated in a visual workflow design process, abstracting from their underlying technological properties ?

1.2. Organization Of The Thesis

Chapter 2 will introduce the theoretical and technological foundations of a *Service Oriented Architecture* (SOA), and the prevailing implementation styles *REST* and *WS**.

Moreover a survey of common properties of real world RESTful service APIs and their compliance with the architectural constraints of REST is given, followed by a criteria based comparison of REST and WS*.

In chapter 3 related work in the field of this thesis will be presented.

The most widely accepted description formats for RESTful service specification - *WADL*, *WSDL 2.0* and *WSDL 1.1* - will be detailed in chapter 4, forming the theoretical basis for parts of the thesis implementation. An overview of alternative approaches to RESTful service description concludes the chapter.

The implementation of a code generation library for *WADL* documents and *Windows Workflow Foundation* (WF) activity components, allowing for RESTful service invocation within a WF workflow, will be presented in chapter 5.

Chapter 6 introduces the *expressFlow* project, a web scale graphical workflow application, and the implementation of a service integration wizard, allowing for convenient adding of WS* and RESTful Web services to a workflow, based on various description formats such as WADL, WSDL 1.1 or URI.

Chapter 7 provides an evaluation of the implementation and possible feature extensions, concluded by a feature comparison with related projects.

2. REST & WS* - Introduction and Comparison of the Paradigms

This chapter presents the theoretical concepts, forming the basis for the implementation work presented later in this thesis. Furthermore a survey of the prevailing properties of real world RESTful Web service APIs and their compliance with the architectural constraints imposed by REST is given. A comparison of various conceptual aspects of REST and WS* based Web services concludes the chapter.

2.1. Basic Definitions

In this section the underlying theoretical concepts referred to throughout the thesis will be outlined.

2.1.1. Service Oriented Architecture

Service Oriented Architectures (SOAs) have emerged to one of the most prevailing architectural paradigms in enterprise level application development as well as scientific research in the field of distributed computing over the last years. Along with the formation of the World Wide Web not only as distribution system for hypermedia content, but also as a vital transport medium for distributed applications the notion of *Web services*, as an underlying technology for implementation of SOAs, was coined. Though both terms are often mutually used they do not necessarily translate to each other. In short an SOA can be seen as an architectural principle in distributed computing, whereas Web services form a set of underlying implementation technologies for the components of an SOA based application.

In order to provide a better understanding of the technological properties of RESTful and WS* based Web services, introduced in subsequent sections, the notion of *Service*, as the core building block of an SOA, needs to be clarified on a conceptual level. In the reference model for SOA, provided by the Organization for the Advancement of Structured Information Standards (OASIS), a Service is defined as :

> The performance of work (a function) by one for another. It combines the capability to perform work for another, the specification of the work offered for another and the offer to perform work for another.

Shifting the emphasis to a more technical level, Newcomer et al. [61] identify four *primary properties* that need to be observed in a service implementation, independent of the chosen implementation style (RESTful or WS*) :

- *Loosely coupled.* At design time of a service loose coupling denotes the prevention of any affinity to potential service consumers [60]. Clients may use a service without exact knowledge of its internal implementation specifics. They are however required to know how to accomplish valid interaction with a service and are thus dependent on a provided service [69].
- *Well-defined service contracts.* A service contract provides a specification of the capabilities of a service [61] and its technical access properties [60]. It clearly separates the interface from technical implementation details [61]. The formal specification of a service contract is one of the most controversial aspects in debates on RESTful Web service implementations, as discussed more in depth in chapter 4 of the thesis.
- *Meaningful to service requesters.* A service is specified at an appropriate level of abstraction with adherence to the standard vocabulary of its associated business domain while conventions specific to the implementation are kept transparent from service requesters [61].
- *Standards-based.* In consequence of the compliance with established technology standards the number of potential service consumers is increased from the provider's perspective. For a service consumer the variety of applicable services rises. Furthermore the large user community around open technology standards gains momentum due to their *collaborative knowledge*.

An SOA is then defined in [59] as a "*relationship of services and service consumers, both software modules large enough to represent a complete business function*". The classical SOA triangle depicts the traditional roles and their interaction relationships:

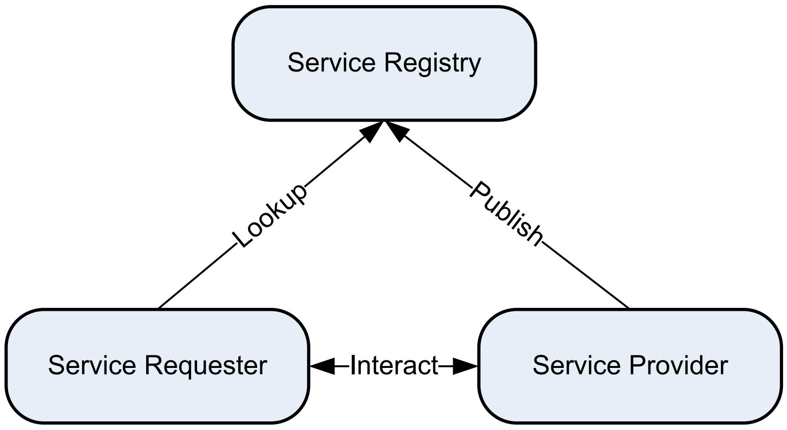


Figure 2.1.: SOA Triangle - Involved Parties and their Relations

A service *provider* provides a service, implementing a particular functionality within an application domain. The service specification, describing the technical interaction specifics and capabilities of a service are published into a service *registry*. A service *requester* looks up the service specification in the registry and interacts with the according service provider in order to achieve the desired application results.

2.1.2. WS* based Web Services

The term WS* based Web services commonly denotes an implementation style for Web services relying on a set of three core standards: *WSDL*, *SOAP* and *UDDI*. Various semantically equivalent terms for such services as for example *Big Web services* [80, 75], *SOAP oriented Web Services* [100] or *WSDL based Web services* [49] can be found in scientific literature and debates on Web service implementation technologies.

According to [25] WS* refers to a set of "*second-generation Web service specifications*" extending the basic Web service framework comprising the three aforementioned basic standards. Details of these specifications, commonly built atop of SOAP [98], will not be covered in this thesis, however many interesting resources on the topic such as [97, 12, 26] can be found in current literature.

Aligned with the abstract principles introduced in 2.1.1 the three core standards relate to the implementation of WS* based Web services as follows:

- *WSDL.* The Web Services Description Language provides an "*XML grammar for specifying properties of a Web Service such as what it does, where it is located and how it is invoked*" [90]. Succeeding the widely accepted version 1.1 [16] WSDL 2.0 [8] has been released as a W3C recommendation in 2007 (see chapter 4 for a detailed discussion of the standards).
- *SOAP.* SOAP is "*a stateless, one-way message exchange paradigm*" [98] defining a standardized XML format for exchange of Web service messages. The XML payload of a SOAP messages comprises the three portions: *Envelope* - for definition of namespaces; *Header* (optional) - allowing for definition of auxiliary information such as security, addressing, payment; *Body* - carrying the main payload of a message, i. e. the name of an invoked service operation and its arguments.
- *UDDI.* UDDI [66], short for *Universal Decription, Discovery and Integration* provides a standardized model for implementation of a Web service registry. The standard comprises XML schema definitions for communication with the registry via SOAP and API specifications describing registry interaction methods [90].

While *WSDL* and *SOAP* have emerged to the prevailing core technologies for the implementation of WS* based Web services, the original vision of UDDI as an open Web service registry has not been realized [61].

2.1.3. RESTful Web Services

REpresentational State Transfer (REST) is the name of an architectural style for distributed hypermedia systems originally introduced by Roy Fielding in his dissertation in 2000 [30]. The set of architectural constraints proposed in the dissertation essentially form the basis for design and implementation of the largest deployed information system today, the World Wide Web (WWW). Web service implementations that rely on the theoretical principles of REST and its core technology standards are referred to as *RESTful Web services* [80], providing a light weight implementation alternative to the previously introduced notion of WS* based Web services (see 2.1.2).

In this section first the main principles of REST will be outlined followed by a summary of the key technology standards for implementation of RESTful Web services.

REST Main Principles

Resource Orientation. Resources constitute the core building blocks a RESTful architecture, described by Fielding as "*the intended conceptual target of a hypertext reference*" [30]. A resource represents an abstract information item of a web based application such as a bookmark in a bookmark service or a particular event in a web based event calendar. The concrete data element being transferred upon interacting with a resource is called the resource *representation*, as for example the XML document describing a particular bookmark resource.

Addressability. Resources are identified and accessible through a unique naming mechanism, namely the URI concept proposed by Berners-Lee et al. [5]. Resource URIs should be meaningful to clients interacting with a service.

Uniform Interface. Interactions between service consumers and resources are mediated through a fixed set of methods as provided by the HTTP protocol [80]. All HTTP methods provided for interacting with a particular resource are specified through the service provider. The associated method semantics for the four basic create, read, update and delete (CRUD) operations adhere to the definitions provided by the HTTP protocol standard [29]: *GET* is solely used for retrieval of information; *POST* is used for creation or update of a resource identified by a server side URI; *PUT* is used for creation or update of a resource identified by a URI determined by the client; *DELETE* is used for deletion of an existing resource.

Statelessness. According to this constraint client-server interaction must be "*stateless in nature, such that each request from client to server must contain all of the information necessary to understand the request*" [30]. A server may not store any information regarding the states of its invoking clients. Hence all information describing a particular state of the client, such as authentication data, have to be transmitted in every request [80].

Layered System. By introducing layers the overall complexity of a system architecture shall be stemmed by allowing for encapsulation of legacy systems and protecting components from unwanted access through clients. Furthermore scalability can be increased through insertion of *load balancing* or *caching* proxy network components.

Underlying Technology Standards

HTTP. The Hypertext Transfer Protocol (HTTP) serves as the primary application protocol for web applications and allows for the transmission of resource representations between network components. The initial version, 1.0, was released in 1996 by Berners-Lee et al. [6]. Derived from the requirements for a modern web architecture stated by Fielding [30] its today widely accepted successor HTTP 1.1 [29] was released in 1999. Revisiting the *uniform interface* constraint of REST the HTTP protocol serves as the uniform *application level protocol* for RESTful Web services. The fixed set of HTTP methods and their interaction semantics associated with the exposed resources essentially define the capabilities of a Web service. Ideally RPC semantics transmitted atop of HTTP, for example by means of a method name request parameter, are not required in truly RESTful implementations (see section 2.2).

URI. Uniform Resource Identifiers (URI) provide a standard mechanism for uniquely identifying and addressing resources in a REST architecture. The URI standard [5] is a superset of two specifications: Uniform Resource Locator (URL) [7] - defining resource identification via its primary access mechanism (e.g. http://address.org/1, ftp://address.org/2/); Uniform Resource Names (URN) [86] - stating requirements for globally unique and persistent naming of resources.

XML. The Extensible Markup Language (XML) [94] was developed by members of the World Wide Web Consortium (W3C) in 1996. Derived from its initial design goals such as *simplicity*, *broad application support* or *easy producible XML documents* XML has emerged to one of the most favored formats for data exchange in Web service applications. XML documents are easy readable for humans and can be automatically processed in most modern programming languages with little effort. Aligned with the *resource orientation* principle of RESTful Web services, an XML formatted document constitutes the concrete *representation* of a resource to be transmitted upon interacting with a service.

JSON. The JavaScript Object Notation (JSON) [21] is a light weight text format allowing for data-exchange in network based applications. Though the format is considered programming language independent it is especially advantageous in JavaScript implementations as JSON structures can easily be accessed in an object oriented manner by means of the JavaScript eval() function. Many main stream Web service providers, such as Google or Yahoo, provide JSON formatted data complementary to XML in their REST APIs. Analogous to XML, JSON serves as a data format for transmission of a resource's *representation*.

2.2. Hybrid Web Services - REST in Real World Scenarios

An important aspect during the implementation of a Web service integration wizard for expressFlow (see section 6.2) was the analysis of real world RESTful Web service APIs, providing the basis for requirements analysis and implementation testing. Many of these Web services, referred to being RESTful throughout the thesis, do not strictly adhere to all of the architectural constraints proposed in Fielding's dissertation [30]. The REST inventor even urges to avoid the term REST in the context of such Web services, which he believes have little in common with the originally proposed REST principles[1].

The formal identification of recurring problems and according solutions in real world software development scenarios, given in this section, is also referred to as "AntiPatterns" [11, 84]. Conceptually related to the notion of a *Software Pattern*, AntiPatterns "*are a method for efficiently mapping a general situation to a specific class of solutions*", they "*provide real-world experience in recognizing recurring problems in the software industry*" and they "*provide a common vocabulary for identifying problems and discussing solutions*" [11].

[1]In his blog Fielding repeatedly expresses his dissatisfaction with the application of his concepts in current implementations: http://roy.gbiv.com/untangled/

Coming back to the identification of recurring misconceptions in real world RESTful Web service implementations, in [80] the term *Hybrid Web Services* is introduced for

> Services often created by programmers who know a lot about real-world web applications, but not much about the theory of REST.

Subsequently the most prevalent properties of such Hybrid Web Services and their implications on the issues of workflow integration dealt with in this thesis will be detailed.

2.2.1. RPC Semantics in the Transmitted HTTP Payload

Problem Definition An essential implication of the uniform interface constraint of REST is the utilization of HTTP as application-level protocol of a Web service. Many implementations ignore this principle and use HTTP rather as an envelope encompassing an operation request specified via a request parameter. The delete photo operation of Flickr's REST API[2] illustrates this issue. In order to delete a photo, an HTTP POST request including a parameter method=flickr.photos.delete is required instead of a simple HTTP DELETE at the photo resource.

Implication on Workflow Integration The specification of a desired service operation by means of a method request parameter essentially leads to an intermingling of two types of parameters within a request. While the method parameter has a fixed value that may not be altered in order to avoid invalid service requests, a *dynamic* parameter as for example the input text parameter for a translation service may be assigned an arbitrary value during workflow execution. The lack of a formal differentiation between these two parameter types basically imposes two consequences on successful visual service integration. One is the increase of complexity due to a higher number of parameters presented, which could possibly overstrain a user. The other is an increased probability of adding faulty invoke elements to a workflow, due to binding fixed parameters to invalid values. To overcome this issue particular parameters could be marked by means of a special attribute (fixed={true|false}) in a formal interface specification such as WADL or WSDL 2.0(see 4). WADL provides a `default` attribute

[2]http://www.flickr.com/services/api/flickr.photos.delete.html

for a `param` declaration being utilized in expressFlow for default parameter values in request URIs.

2.2.2. Negligence of HTTP Method Semantics

Problem Definition In the HTTP specification methods are classified by two criterias [27]. *Safe* methods such as GET and HEAD are intended solely for retrieving information from a resource, while *unsafe* methods such as POST, PUT and DELETE might cause changes on a resource's state. A method is called *idempotent* if its repeated execution ($N > 1$) has the same effect on a resource as a single execution. Though not calling itself RESTful the delicious HTTP API[3] provides a good exemplification of how this constraint can be ignored completely. Essentially all operations provided by the service, including such causing a change to a resource state such as *posts/delete* are called via HTTP GET.

Implication on Workflow Integration Due to a negligence of the proposed HTTP method semantics in existing service APIs automatic analysis facilities for integration of provided services in workflow tools are confined. This can best be explained by means of an example. Consider a user adding a service invoke element associated with a non-idempotent HTTP method to a workflow. Due to the definition of idempotence the state of the resource might change in consequence of its invocation. If during execution of a workflow this resource is invoked repeatedly, for example within a loop, its state might differ between two invocations possible leading to unexpected workflow results. A static workflow analysis tool might reveal this flaw, however analysis would fail if the non-idempotent POST would be masked by an idempotent GET.

2.2.3. Ignored HTTP Header Facilities

Problem Definition Many service implementations tend to define custom query parameters in order to specify request properties for which standardized HTTP header fields exist. The desired response format of a request for example can be specified

[3] http://delicious.com/help/api

in the HTTP *Accept* header field. Authentication information can be passed in the *Authorization* header (the field name is rather unfortunate), rather than within the query string. A sample request URI taken from the ebay item search API illustrates this issue: http://open.api.ebay.com.shopping?callname=FindItems&responseencoding=XML&appid= Alexande1234567. Authentication information is passed in the *appid* parameter, the *responseencoding* parameter specifies XML as the desired result format.

Implication on Workflow Integration Essentially the flexibility gained by deviating from provided header mechanisms imposes a rise of complexity of workflow integration of a RESTful service. Comparably to the aforementioned RPC issue (see section 2.2.1) the encoding of authentication and format parameters within the query string causes more complex query strings that might become unreadable and error-prone. Furthermore the associated semantics of such declared parameters (e. g. parameter1 specifies the result format, parameter2 provides authentication information) cannot be automatically discovered by a workflow tool. Hence a user would have to provide necessary request properties, such as the desired result format of a service request, manually to a workflow run time in order to gain working results. A valuable application of specifying sensible authentication information in a standardized way could be the ability to keep such information transparent in the visual representation of a workflow, comparable to the masking of user passwords in many applications.

2.2.4. One Endpoint Catches All

Problem Definition In the URI syntax specification [5] a "/" character is proposed for denoting sub path components of a hierarchical resource URI. A violation of this principle commonly seen in existing implementations is that essentially all resources provided by the service are accessible through a *single URI*. Particular resources are differentiated by means of query parameters appended to the URI. The REST API[4] of upcoming, a social event calendar application provided by Yahoo elucidates the issue: All service resources are provided through a single base URI, http://upcoming.yahooapis.com/services/rest , supplemented by the desired information category such as method=event.getInfo or method=venue.getList encoded within a query string parameter.

[4] http://upcoming.yahoo.com/help/w/Example_REST_URLs

Implication on Workflow Integration Dynamic resource URIs possibly possibly created by means of a PUT request during execution of a workflow cannot be discovered automatically. A workflow designer would have to rely on a static URI scheme being defined in the API specification resulting in a tight coupling of the workflow application and its included services. In order to create for example a new event in the aforementioned upcoming service one could issue a PUT onto /events/category/eventId. In a truly RESTful implementation the created event resource would subsequently be addressable by means of its creation URI. However in the upcoming service the resource is accessible via a fixed base URI supplemented by the resource identification encoded within a query string parameter, as for example method=event.getInfo&event_id=20849.

2.3. REST & WS* - Criteria Based Comparison

In this section a comparison of the two Web service implementation paradigms REST and WS* is drawn, based on a set of four criteria, reflecting the primary conceptual distinctions between both styles.

2.3.1. Architectural Constraints vs. Set of Standards

As previously intimated in this chapter, an inherent disparity in the conceptual definition of the two paradigms REST and WS* exists. Roy Fielding states in his thesis that "*An architecture's properties are created by the application of constraints*" [30]. Whereas REST is a highly constrained architectural style, based on a clearly defined set of abstract principles, WS* services lack such a theoretical basis almost completely. Constraints for WS* style Web services mainly stem from a set of standard specifications, collectively denoted as the WS* standards stack [50]. Commonly these standards are designed and maintained through standards bodies such as the World Wide Web Consortium (W3C) and the Organization for the Advancement of Structured Information Standards (OASIS), or industry consortiums comprising leading software vendors.

Critics of the WS* stack state, that many of its specifications are bloated, ambiguous and too complex in order to efficiently yield interoparable components. REST on the other hand fosters the implementation of loosely coupled architectures, independent of concrete implementation technologies and standard interpretations. While this argumentation might approve for the programmatic integration of RESTful services it is only partly applicable on the issues of workflow integration. A glimpse at real world RESTful implementations (see section 2.2) reveals a large diversity in the interpretation of the REST constraints, highly impeding their tool supported integration in workflow environments. The standardized interface descriptions and data exchange formats of WS* based services, fostering the implementation of according software tools, seem to be superior to the theories of REST in this regard. Chapter 4 of this thesis will examine how this conflict can be mitigated by applying standardized description languages to RESTful services.

2.3.2. Service Design Process

Based on the two different SOA implementation paradigms the set of decisions required for designing a RESTful service varies significantly from the design process of a WS* based service. In [80] the primary steps traversed during design of a RESTful service are described as follows:

1. *Resource Definition* - Based on the given data set exposed by the service, the core elements of every RESTful service, its *resources* have to be identified. According to [80] these can be divided into three categories: *Predefined one-off resources, required by all remaining resources* (typically the base URI of a service), *Exposed data objects* (for example a bookmark), *Processing results of an algorithm applied to a set of resources* (for example the text returned from a translation service).

2. *Resource Naming / URI Design* - In order to formally identify the specified resources and expose them to potential service consumers an URI scheme has to be defined. Among the best practice recommendations for a meaningful URI design are: Hierarchical structures are denoted through path variables (/smith/-bookmarks/bm1); Flat hierarchies can be expressed through matrix parameters separated by a ";" character (bm1/tags/tag1;tag2); Algorithmic input parameters are expressed through query string parameters (?method=a&param1=val1).

3. *Interaction Semantics Definition* - Aligned with the method semantics specified in the HTTP protocol specification the allowed HTTP methods and their intended function have to be determined for the previously created set of resources (GET

on resource A retrieves the resource representation, PUT on URI B creates a new resource at this location).

4. *Resource Relationships* - Aligned with the *Hypermedia as the Engine of Application State* principle of REST, resources returned upon interacting with a service are interconnected through hyper links. As stated in [75] these links could specify the intended state transitions traversed by a client interacting with a service.

5. *Resource Representation* - Finally the data format returned upon requesting a resource representation and accepted for creation of new resources has to be stipulated. In most data-centric implementations this is either XML or JSON.

Additionally a number of secondary properties, such as HTTP response codes and the (optional) authentication mechanism have to be specified. However as these questions are not strictly bounded to RESTful service design they will be omitted in this comparison.

Pautasso et al. [75] identify three primary decisions required for WS* based service design, essentially centered around the definition of its WSDL interface description:

1. *Data Modeling* - Commonly within the types section of a WSDL document the structure of exchanged XML data has to be defined. In a *contract-last* approach, where a WSDL document is automatically generated from an existing implementation this structure is derived from the data type definitions of the underlying programming language.

2. *Message Exchange Pattern Definition* - Essentially the message exchange pattern constitutes whether a service operation yields a response message, i. e. *request-response* or is defined as *request-only* operation. The latter type allows a client for immediate continuation of its execution after issuing a service request.

3. *Operations Enumeration* - The exposed operations represent the capabilities provided by the service provider. Operation names must depict the provided business functionality in a meaningful way to potential service consumers. Related operations may be grouped in *portTypes* by various criteria as for example security requirements, functional domain or organizational factors.

2.3.3. Application Scenarios

A typical criteria used in comparisons of REST and WS* is the intended area of application. Whereas the extensive set of WS* standards addresses many of the issues required in complex enterprise scenarios the light weight nature of RESTful services seems to align well with the ad hoc integration facilities of RESTful services in Web 2.0 applications. A closer look at the current real world service landscape reveals that this argumentation is only partly appropriate. While the ideas behind many of the specifications defined in the WS* stack [12], such as WS-Security, WS-ReliableMessaging or WS-AtomicTransaction, are assuredly valuable in business related scenarios, also applications stemming from the Web 2.0 world and consequently their associated RESTful APIs increasingly find their way to the enterprise [14]. The relevance of RESTful integration in business surroundings is also underpinned by recent software products, such as *Lotus Mashups* [44] or *IBM WebSphere sMash* [43] (see section 3.4).

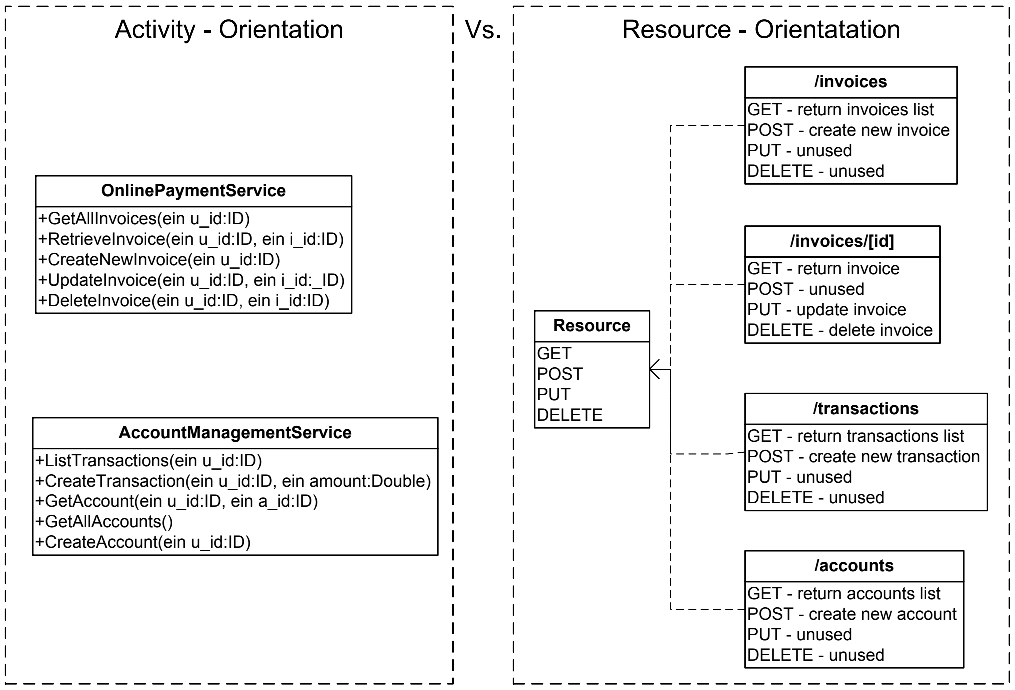

Activity - Orientation
Vs.
Resource - Orientatation
OnlinePaymentService
+GetAllInvoices(ein u_id:ID)
+RetrieveInvoice(ein u_id:ID, ein i_id:ID)
+CreateNewInvoice(ein u_id:ID)
+UpdateInvoice(ein u_id:ID, ein i_id:_ID)
+DeleteInvoice(ein u_id:ID, ein i_id:ID)
AccountManagementService
+ListTransactions(ein u_id:ID)
+CreateTransaction(ein u_id:ID, ein amount:Double)
+GetAccount(ein u_id:ID, ein a_id:ID)
+GetAllAccounts()
+CreateAccount(ein u_id:ID)
Resource
GET
POST
PUT
DELETE
/invoices
GET - return invoices list
POST - create new invoice
PUT - unused
DELETE - unused
/invoices/[id]
GET - return invoice
POST - unused
PUT - update invoice
DELETE - delete invoice
/transactions
GET - return transactions list
POST - create new transaction
PUT - unused
DELETE - unused
/accounts
GET - return accounts list
POST - create new account
PUT - unused
DELETE - unused

Figure 2.2.: Activity vs. Resource Abstraction

Another aspect mentionable in this context is the intended type of application to be solved by a Web service implementation. Snell compares *resource-oriented* (REST) to *activity-oriented* (WS*) services [85]. Whereas RESTful services, centered around the core abstraction of a *resource*, seem to fit better with *data-oriented* applications, the notion of WS* services is focused on *actions* that can be carried out by a service provider. Figure 2.2 depicts these varying concepts in an UML like notation based on [70]. However this rationale holds only partially true, as real world implementations taken from the programmableWeb directory [58], such as *Moneybird*[5] or *Iron Money*[6] demonstrate. Though providing their functionalities through a REST interface these services provide typical functions commonly allocated to the WS* domain.

2.3.4. Orchestration Models

The orchestration models for RESTful and WS* based Web services are often referred to as orthogonal concepts in terms of their underlying behavioral model. Whereas the traditional emphasis in combining RESTful Web services lies on their aggregation in a shared *namespace*, in order to form new content, WS* based Web services are commonly meant to be composable in a *time axis* [77], by means of a formally defined executable workflow. Subsequently recently introduced mashup concepts in academic and industrial research will be contrasted to traditional workflow models as provided by the Windows Workflow Foundation framework and WS-BPEL.

Web Service Aggregation in Namespace - Mashup Concepts

Web 2.0 Mashups. The notion of web 2.0 mashups commonly denotes the aggregation of web content supplied either by RESTful Web services APIs or standardized feed formats (RSS,Atom), in order to form a new application. The aggregation process may either be located at the *client-* (web browser) or the *server side* (web server). Existing client side mashup solutions are mostly implemented as browser extensions [45, 48], extending the browser functionality by custom definable macro commands or enhancing displayed web pages by interfering in the page rendering process. Server side mashup implementations are fostered through web application frameworks or tailored products allowing for non programmatic creation of mashup sites [44].

[5]An online billing service available at `http://www.moneybird.nl/help/api`

[6]A finance application providing personal account management available at `https://ironmoney.com/api/documentation/`

Service Mashups. Benslimane et al. [4] introduce the notion of *Service Mashups* as a novel concept for end user service compositions. New challenges arise from the utilization of semantic web technologies (RSS, RDFa, SAWSDL[7]) for Web service mashup creation. The authors underpin the necessity of semantic *annotations* for Web services, essentially yielded through four dimensions of heterogeneity: "*data (input/output), functional (behavioral), nonfunctional(quality of service, policy) and execution (runtime, infrastructure, exceptions)*". Due to the increasing popularity of RESTful services and Web APIs compared to traditional WS* based implementations, new solutions for semantic description of such services such as hRESTS [37] or SA-REST (Semantic Annotations of RESTful services) [83] will take on greater significance in the future, according to [4].

Workflow Oriented Web service Orchestration

WS-BPEL. The *Web Services Business Process Execution Language* (WS-BPEL) is a standardized XML language allowing for the formal specification of abstract and executable business processes [67]. WS* based Web services may be invoked from within a BPEL process, after linking their WSDL interface description to the process definition by means of a `<partnerLink>` element. Particular input-/output-elements of invoked service operations, being defined in the WSDL description, are accessible within the process scope and may be stored within process variables by means of the `<assign>` / `<copy>` construct. Structured activity elements allow for the specification of repetitive (`<while>`) or conditional (`<if>`) execution of simple process activities. BPEL documents are commonly created by means of a visual design tool (for example [78]), abstracting from the complex XML structure. A process is deployed into a BPEL engine (for example [33]), that interprets the BPEL description and its associated WSDL documents, and executes the process on demand.

Windows Workflow Foundation. The Windows Workflow Foundation (WF) provides an implementation and execution framework [52] for business workflows comparable to WS-BPEL. The WF workflow component model comprises a set of simple and structured activities in order to define the execution structure of a workflow and allows for the integration of WS* based Web services. A workflow specification may either be expressed solely by means of the declarative *Extensible Application Markup*

[7] http://www.w3.org/2002/ws/sawsdl/

Language (XAML) or as a combination of an XAML and an associated code behind file (for example written in C#, Visual Basic). Notably is the extensibility of the WF component model, being utilized for the integration of RESTful Web services in WF workflows, detailed in section 5.3.1 of this thesis.

3. Related Work

In this chapter related work in the field of this thesis will be presented. The introduced approaches address the issues dealt with in the thesis implementation either directly (3.1, 3.3) or provide alternative strategies for mashup creation, partly stemming from the field of theoretical computer science (3.2). A glimpse at current industry implementations concludes the chapter.

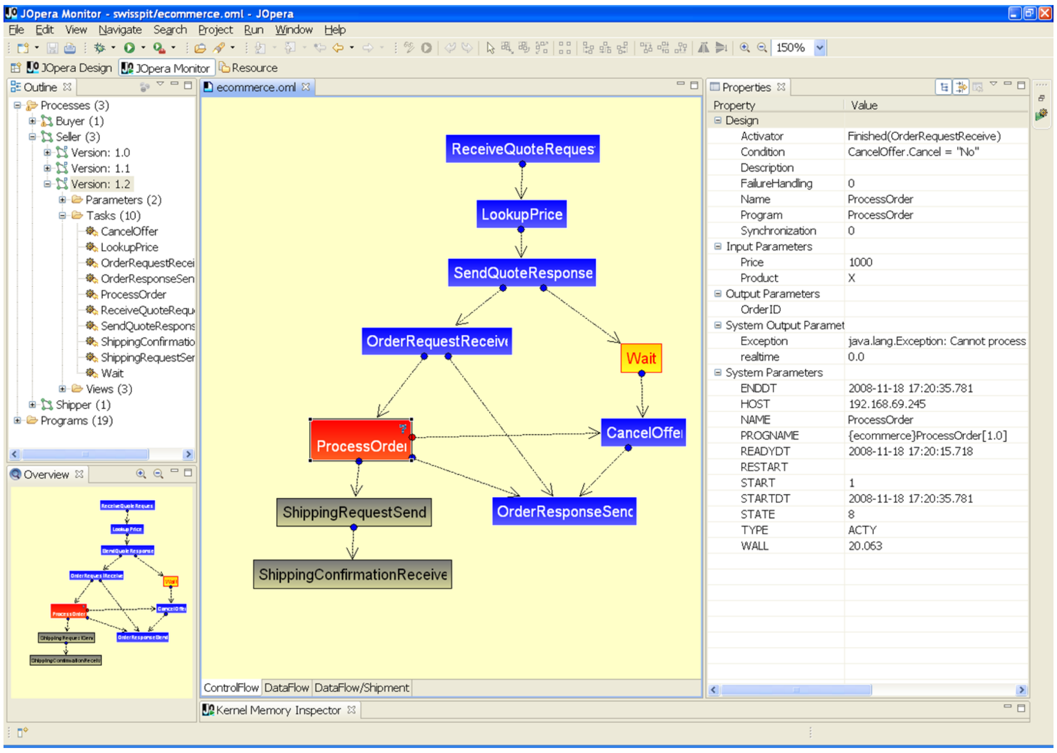
3.1. Workflow Design And Runtime Integration

Workflow design and runtime integration addresses issues of dynamic integration of RESTful services in a graphical workflow design tool (3.1.1) and their invocation during execution of a formally described workflow in a workflow runtime.

3.1.1. JOpera for Eclipse

Pautasso et al. propose JOpera for Eclipse , an Eclipse based workflow application comprising a visual workflow designer and an integrated run time allowing for immediate execution of a workflow. The implementation relies on the *JOpera Visual Composition Language* [73], describing a process as a set of directed graphs. Nodes of a graph represent tasks of a workflow and associated data parameters. Edges represent control flow or data flow dependencies between tasks. The JOpera process model provides two abstract representations of a process: the *data flow* representation describes the input- and output-parameters of tasks/service invocations within a workflow and their relationships through data flow bindings; the *control flow* representation depicts the tasks of a workflow and its execution structure.

Particularly interesting in the field this thesis is the facility for orchestration of RESTful and WS* based Web services within a JOpera process. The abstraction of service implementation specifics is achieved through so called *adapters* linking a service components with its parent process. The *HTTP adapter* for integration of RESTful Web services allows for specifying the core properties of a RESTful service invocation, i. e. request URI, HTTP header fields and message body elements. Dynamic request elements (for example process parameters) may be denoted through template expressions.



Property	Value
Design	
Activator	Finished(OrderRequestReceive)
Condition	CancelOffer.Cancel = "No"
Description	
FailureHandling	0
Name	ProcessOrder
Program	ProcessOrder
Synchronization	0
Input Parameters	
Price	1000
Product	X
Output Parameters	
OrderID	
System Output Paramet	
Exception	java.lang.Exception: Cannot process
realtime	0.0
System Parameters	
ENDDT	2008-11-18 17:20:35.781
HOST	192.168.69.245
NAME	ProcessOrder
PROGNAME	{ecommerce}ProcessOrder[1.0]
READYDT	2008-11-18 17:20:15.718
RESTART	
START	1
STARTDT	2008-11-18 17:20:35.781
STATE	8
TYPE	ACTY
WALL	20.063

Figure 3.1.: JOpera Visual Design Surface - Control Flow Representation

While JOpera introduces some promising features for heterogeneous service composition such as the flexible integration mechanism for RESTful Web services and the integrated workflow run time for immediate testing of generated workflows the Eclipse based architecture induces some flaws as well. The viable service adapter components providing support for various kinds of service resources are hidden behind the overly complex standard Eclipse GUI structure making its usage rather unpalatable especially to non experts. Furthermore the necessity for local installation on a client computer may detain a user from using the application. Also currently the integration of formally described (for example by a WADL description) RESTful Web services is not supported.

3.1.2. Apache ODE

Apache ODE (Orchestration Director Engine) [33] is an open source BPEL engine with support for RESTful service integration in BPEL processes. The application is implemented in Java and can be deployed either as a JBI [89] service assembly (using for example Apache ServiceMix [34] as container) or as Apache Axis2 Web service [32].

Figure 3.2 depicts the application architecture, comprising three core components: the *BPEL Compiler* converts BPEL artifacts into an executable representation; execution tasks such as process instantiation, concurrency management and persistence of execution state are implemented in the *BPEL runtime* component; the *integration layer* ties ODE to its application container.

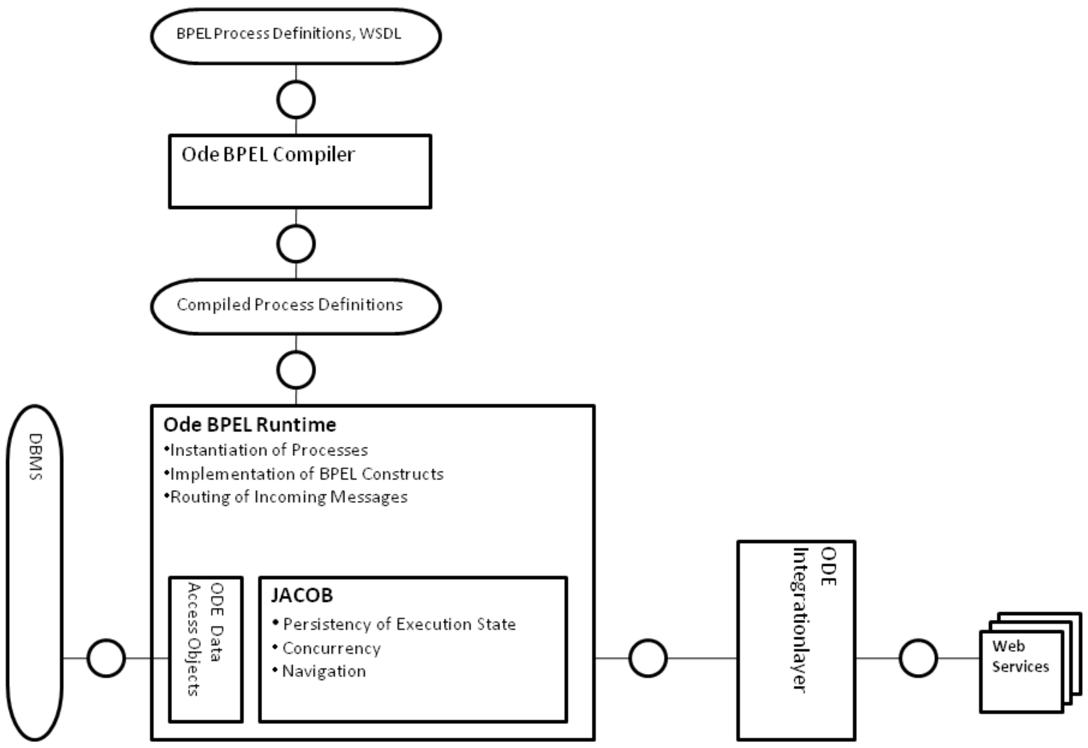

BPEL Process Definitions, WSDL
Ode BPEL Compiler
Compiled Process Definitions
DBMS
Ode BPEL Runtime
•Instantiation of Processes
•Implementation of BPEL Constructs
•Routing of Incoming Messages
ODE Data Access Objects
JACOB
• Persistency of Execution State
• Concurrency
• Navigation
ODE Integrationlayer
Web Services

Figure 3.2.: ODE Architectural Overview

ODE fully implements the WS-BPEL 2.0 standard [67], RESTful services are integrable via the WSDL 1.1 HTTP binding. In order to overcome the limitations of WSDL 1.1 regarding the description of RESTful Web services (see 4.2) 4 extensions of its HTTP binding mechanism are proposed:

- The *HTTP verb* is pushed down from the binding- to the operation-level, by introducing an additional element `<odex:binding verb="method name">` as child of the binding *operation* element.
- The *request uri* is specified in the *address* element of a *port* definition. The *urlReplacement* mechanism allowing for definition of dynamic URI components through template expressions is extended in order to allow for partitioning dynamic request elements into request-URI and -body.
- A `<header>` element as child of a binding level operation definition allows for mapping operation *part* names to HTTP request-/response-header fields (e.g. *Authorization*).
- In order to enable fault handling a *fault* element is required within the abstract (*portType*) and concrete (*binding*) parts of an operation definition. The ODE runtime will then raise an exception if an HTTP error status code is returned from an invoked service. The exception name is inferred from the XML message payload of the error response.

Furthermore an extension for *RESTful BPEL* in ODE is proposed, in order to allow for a more native integration of RESTful services (see section 3.3.2), however no implementation exists yet.

3.2. Requirements Driven Mashup Creation

Requirements driven mashup creation denotes a strategy for composing services essentially relying on concepts stemming from the field of theoretical computer science, such as probabilistic methods, and automated planing techniques. The work presented in this section is less focused on the technical issues of integrating various service styles but on the automatic creation of mashups based on user provided criteria.

3.2.1. MashupAdvisor

Elmeleegy et al. propose *MashupAdvisor* [24], an AI based mashup creation tool, aiding the mashup design process through automated recommendations based on the given user input and a repository of existing services and mashups matching this input. A typical design process aided by MashupAdvisor comprises three basic steps that are iteratively repeated in order to create a desired mashup. First a user defines an *input*, identified by a name, e. g. *email*. In basis of the input definition the recommendation process is initiated, yielding a list of recommended *outputs* that match the given input. An output can be an existing service, such as *WhitePages* service, or an existing mashup stored in the mashup repository. Finally the user chooses an output from the list, upon which the best plan for linking the selected output with the given input is computed. The resulting partial mashup can be extended by repeating these steps iteratively.

The focus in MashupAdvisor is on the automatic composition of services and data feeds based on statistical and semantic analysis of their formal input-/output-properties. Though the integration of RESTful services is theoretically supported the technical issues of dynamic service integration and editing of existing mashups are not addressed. The authors propose the extension of existing editing tools such as Yahoo Pipes [99], Google Mashup Editor [38] or Microsoft PopFly [51] (see section 3.4) through the recommendation facilities provided by MashupAdvisor instead.

3.2.2. Wishful Search / MARIO

Wishful Search / MARIO (Mashup Automation with Runtime Orchestration and Invocation) [79], introduced by Riabov et al., allows for automatic service composition through iterative specification of desired *mashup composition goals*. The composition model relies on three abstract concepts:

- *Objects, Tags, Taxonomies* - resources within a mashup such as services or feeds are represented as *objects* described by a set of *tags* together forming a *taxonomy*. Tags may be hierarchically related - for example *GoogleTranslate::Translation*.
- *Queries* - a tag query is the formal specification of a desired composition result - for example {*Translation, English, German*}.
- *Operators* - operators represent the building blocks of a mashup, i. e. the formal representation of a particular service or a news feed. An operator description comprises the specification of service variables (for example *language*) and the

describing tags partitioned into *input-* (for example *InEnglish*) and output-tags (for example *InGerman*).

The mashup creation process is then formally described as composition problem comprising these three concepts that has to be satisfied by applying a planning algorithm. For a detailed discussion of the highly formal description of this process the interested reader may refer to [79].

The tool implementation includes a web based user interface that presents a user with an input field for interactive specification of composition goals, a graphical representation of the proposed mashup and an output field displaying the generated mashup results. As in MashupAdvisor (see 3.2.1) issues specific to concrete service implementations (RESTful or WS* based) are not further covered by the approach.

3.3. Native Integration Of REST Semantics In BPEL

In this section two approaches for *RESTful BPEL*, i. e. the native integration of REST semantics in WS-BPEL [67] based process definitions, are introduced. Cesare Pautasso, also involved in the implementation of JOpera (see section 3.1.1), proposes a set of elements extending the WS-BPEL standard vocabulary in order to express the invocation of RESTful services in BPEL based processes. The second presented approach describes an extension of the aforementioned Apache ODE (see section 3.1.2) project.

3.3.1. BPEL for REST

In order to overcome the limitations of WS-BPEL [67] regarding the integration of RESTful Web services Pautasso proposes a set of standard extensions for native integration of REST semantics in WS-BPEL based processes [72]. Among the mentioned flaws forming the rationale for the proposed extensions is the tight coupling of WS-BPEL and WSDL 1.1, providing only rudimentary support for RESTful service description. WSDL 2.0 on the other hand solves the description problem theoretically but is currently only rarely deployed in real world service implementations and formally not supported by the WS-BPEL standard specification. Furthermore the author argues, that REST semantics should not be hidden behind a WSDL description due to the different interaction models of REST and WS* based services.

The proposed extensions are grouped into three categories:

- *Resource Invocation* - In order to specify the interaction with a REST resource the `<invoke>` element of WS-BPEL is replaced by four elements derived from the four basic HTTP CRUD method names: `<get>`, `<post>`, `<put>`, `<delete>`. The request URI and the optional request-/response-payload are declared by means of the attributes `url`, `request` and `response`. HTTP Header fields may be declared within a `<head>` child element, a `<catch>` element allows for handling error code conditions.
- *Resource Publishing* - Resources may be published dynamically upon reaching a <resource> element during execution of a process. Its child elements <onGet>, <onPost>, <onPut> and <onDelete> specify request handlers, corresponding to the request HTTP method, in which arbitrary BPEL structured activities may be declared. The result data sent back to a client invoking a published resource is declared within a <respond> element.
- *Minor Extensions* - All published resources are discarded upon reaching the BPEL <exit/> activity. The strong typing imposed by the `messageType` variable attribute of BPEL is relaxed, in favour of optional element and type elements referencing XML schema type declarations of invoked resources.

3.3.2. RESTful BPEL in Apache ODE

Within the Apache ODE project (see section 3.1.2) an extension for RESTful BPEL has been proposed [35]. The WS-BPEL extension introduces a variant of the <invoke> activity, including two attributes specific to REST interaction: `resource` - allows for identification of an invoked resource via a BPEL variable containing the request URI; `method` - defines the HTTP method of the request. Dynamic URI strings may be constructed by means of two additional XPath functions, `combine-url` (combines base- and relative-URI components URI) and `compose-url` (allows for building an URI from template expressions). Moreover a RESTful variant of the WS-BPEL <receive> activity is proposed, allowing for exposing dynamic process resources.

3.4. Industry Projects

The integration of RESTful Web services into custom applications is also addressed by numerous industry projects, some of which will be briefly summarized in this section. Among the tools allowing for a workflow oriented aggregation of REST resources and data feeds are *Yahoo Pipes* [99] and *Microsoft Popfly* [51]. Yahoo Pipes comprises an impressive JavaScript based workflow editor, allowing for integration and manipulation of arbitrary data feeds and other XML based data sources, and an integrated runtime, immediately presenting a user with the results of a created workflow. The Silverlight[1] based Microsoft Popfly provides a comprehensive web front end with 3D elements, however its integration facilities for external data sources do not seem to be very elaborated yet (the application is currently officially considered beta). Very promising, though not based on the notion of workflow like integration of services, are *Mozilla Ubiquity* [48] and *Intel Mash Maker* [45], allowing for client side creation of data mashups through extending the browser functionality. Ubiquity allows for defining custom natural language like instructions, such as *translate this page*, linking together arbitrary web resources, such as a news site and a translation service. Mash Maker extends existing web sites by so called widgets that hook into currently visited web sites. Such a widget could then display the geographic location of a user's Facebook friends within the actual browser tab by interconnecting the Facebook API with the Google Maps service. *Lotus Mashups* [44] allows for creation of personalized sites, assembling various web resources, to be shared within an enterprise environment for example.

[1] http://silverlight.net/

4. RESTful Web Service Description

In order to foster the integration facilities for RESTful Web services in workflow applications, a machine processable service specification entails numerous advantages over the textual service description provided in many RESTful Web service API documentations. From a user's point of view the initial barrier for adding a RESTful Web service in a visual workflow design application is lowered. Due to an abstraction of technical service specifics, such as the request-/response-serialization rules or the message payload format, the focus within a workflow design process remains on the functional integration of a Web service rather than technical issues. As demonstrated in the implementation of a Web service integration wizard, detailed in chapter 6 of this thesis, the formal enumeration of Web service parameters also allows for data binding between respective Web service components of a workflow.

This chapter provides an investigation of current description standards, allowing for formal specification of RESTful Web services. The concepts introduced in detail are :

- WADL (Web Application Description Language) - A light weight description language, aligned with the resource orientation principle of REST and intended solely for the specification of RESTful Web services.
- WSDL (Web Services Description Language) 1.1 - The de facto description standard for WS* based Web services, including rudimentary support for specification of RESTful Web service characteristics.
- WSDL 2.0 - The rarely deployed follow-up specification to WSDL 1.1, comprising many elements required for RESTful Web service description.

The introduction of the respective language concepts is complemented by a description of two real world Web service operations, taken from the social bookmark system *Bibsonomy* [91]. The complete description documents, based on the textual description provided in the API documentation, have been created in order to gain a deeper understanding of the language specifications, in the course of writing the thesis. To complete the picture, the chapter concludes with a summary of various alternative attempts, dealing with particular issues of RESTful Web service description.

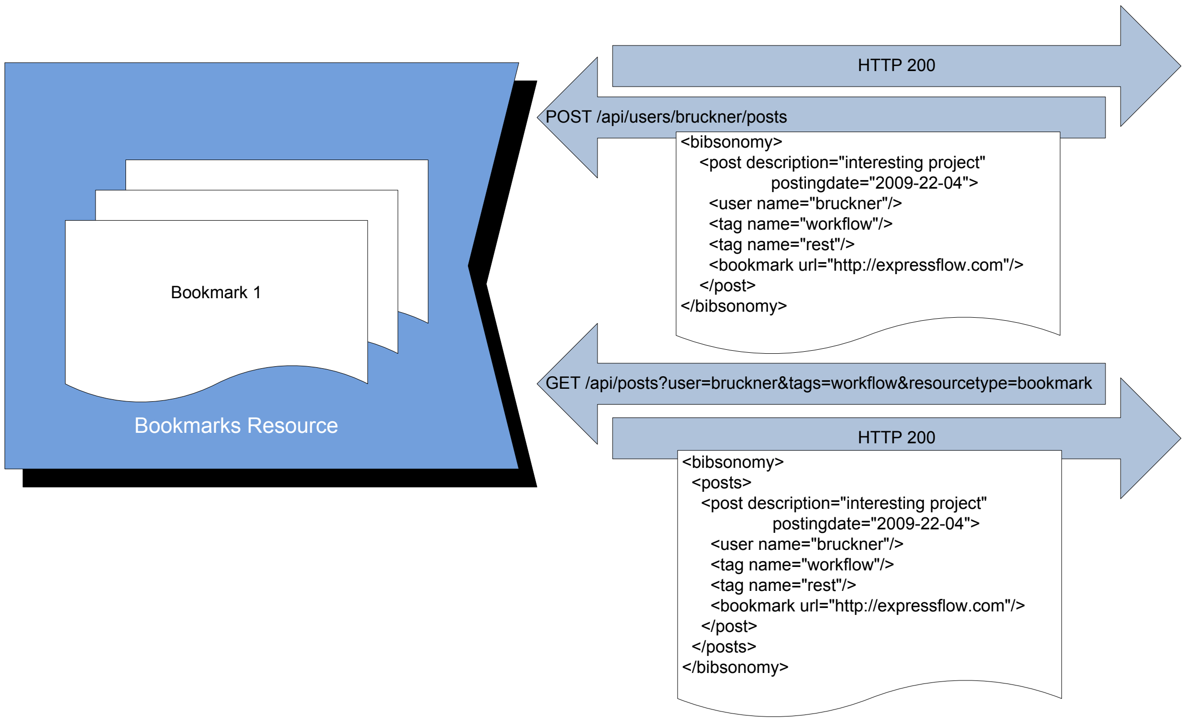
4.1. Describing a RESTful Web Service in WADL

WADL (Web Application Description Language) [42], is an XML description language for web resources, first introduced by Marc Hadley[40] in a blog entry in 2005. Hadley is Senior Staff Engineer at Sun Microsystems and is working among others on JSR 311, the Java API for development of RESTful Web services, better known as JAX-RS [56].

WADL provides an XML vocabulary for hierarchical description of resources and their request-/response-behavior. The original motivation for its specification was to provide a machine processable description format for RESTful Web services comparable to WSDL. Though still not being wide spread in mainstream web APIs today, WADL is one of the most promising approaches to solve the REST description problem. It facilitates many of the attributes indispensable to complete RESTful Web service description and is well supported by tools compared to other description approaches [65]. In the reference implementation for JAX-RS, Jersey [55], WADL is integrated by automatic generation of WADL descriptions for deployed resources [54]. *Rest Describe & Compile* is a WADL based open source project by Google Software Engineer Thomas Steiner [87]. Parts of the library implementation, detailed in section 5, are based on its WADL generation algorithm.

4.1.1. WADL Exemplified - The Bibsonomy Example

Accompanied by a description of two Web service operations provided by the Bibsonomy API [91], the potentialities of WADL for detailed request-/response-specification of a RESTful Web service are elucidated. Bibsonomy is a web scale social bookmarking service, allowing for saving web URIs and literature lists on a central server. Bookmarks may be enhanced by descriptive tags and published among the user community. The functionality of the chosen operations, *Get Posts by a User* and *Create Post* is depicted in figure 4.1. A *Get Posts* request is invoked via HTTP GET , passing the *user* name, associated *tags* and the resource *type* (bookmark or bibtex). The list of resources matching the request are returned within an XML document. In order to save a bookmark by means of the *Create Post* operation, an HTTP POST request is required, passing the XML formatted resource (bookmark or bibtex) in the request body.


HTTP 200
POST /api/users/bruckner/posts
Bookmark 1
Bookmarks Resource
GET /api/posts?user=bruckner&tags=workflow&resourcetype=bookmark
HTTP 200

```
<bibsonomy>
   <post description="interesting project"
              postingdate="2009-22-04">
     <user name="bruckner"/>
     <tag name="workflow"/>
     <tag name="rest"/>
     <bookmark url="http://expressflow.com"/>
   </post>
</bibsonomy>
```

```
<bibsonomy>
  <posts>
    <post description="interesting project"
               postingdate="2009-22-04">
      <user name="bruckner"/>
      <tag name="workflow"/>
      <tag name="rest"/>
      <bookmark url="http://expressflow.com"/>
    </post>
  </posts>
</bibsonomy>
```

Figure 4.1.: Bibsonomy API - Get Posts and Create Post

The subsequently used language constructs are excerpted from the full Web service description, to be found in appendix A of the thesis. Namespace declarations and prefixes have been omitted for better readability.

Schema Import

```
<application>
  <grammars>
    <include href="http://www.bibsonomy.org/help/doc/xmlschema.xsd"/>
  </grammars>
```

Listing 4.1: WADL Application and Grammar Definition

Succeeding an `application` element, forming the root of each WADL specification, an XML schema document, describing the exchanged data is referenced via `include`. WADL does not restrict the format of this definition, although it is very common in many API documentations. This schema definition for exchanged XML data can be compared to the declaration of message types in WSDL. Elements defined within a schema specification can subsequently be referenced throughout a WADL description (see section 4.1.1 for an example).

Resource Definition

```
<resources base="http://www.bibsonomy.org/">
  <resource path="api">
    <resource path="posts">
  </resource>
  <resource path="users/{username}/posts"/>
</resources>
```

Listing 4.2: Resource Definition

The `resources` element forms the root of a hierarchically organized set of resources. Its `base` attribute usually marks a server address preceding all subsequent resource URIs. Subsequent URI parts are defined in the `path` attribute of a `resource` element. The resource node with `path`="api" followed by a child `resource` with `path`="posts" is concatenated to `api/posts` in the final URI. This could represent the file system structure of a web server as well as a URI template declared in a WCF REST service [31]. Values of `path` may also contain slashes, so that the `api/posts` path could equivalently be declared within a single `resource` element. The hierarchical structure allows for efficient and easy processable declaration of arbitrary sets of resources. Its generation is described in section 2.5.1 of the WADL specification. The according resource URI for the Get Posts operation is http://www.bibsonomy.org/api/posts.

The URI path for a Create Post operation is defined within a single resource element. Notable is the usage of an embedded `template parameter` [47] `username` enclosed by curly brackets "{}". The actual value of the parameter is inserted at runtime upon performing a request on the resource. The hierarchical resource description as demonstrated for `Get Posts` is now replaced by a flat definition.

Request Definition

```
<!-- Get Posts Request-->
<method name="GET">
  <request>
    <param name="user" type="xsd:string" style="query"/>
    <param name="tags" type="xsd:string" style="query"/>
    <param name="resourcetype" type="xsd:string"
    style="query"/>
  </request>
</method>
<!--Create Post Request-->
<method name="POST">
  <request>
    <representation mediaType="application/xml"
      element="bib:post"/>
  </request>
</method>
```

Listing 4.3: Request Definition

`Method` denotes the HTTP method to be used on performing a request on a resource followed by the parameter definition through `param` elements. Used attributes are `name` (required), `style` (optional) and `type` (optional). `Style` distinguishes different parameter styles allowing for the definition of a `query` string parameter or a HTTP `header` field. Amazon's Simple Storage Service, S3 utilizes the `Authentication` header to pass authentication information. The `Accept` header could indicate the expected data format for a request.

Concatenated to the resource definition the complete query string for the Get Posts operation is now http://www.bibsonomy.org/api/posts?user=<user>&tags=<tags> &resourcetype=<resourcetype>.

A Create Post operation expects XML formatted input, indicated by the `mediaType` attribute value of its `representation` definition. The `element` attribute value references an element `post`, declared in the included XML schema definition(see section 4.1.1).

Response

```
<!--Get Posts Response-->
<response>
  <representation mediaType="application/xml"
    element="bibsonomy">
    <param name="postdate" style="plain"
        path="/bibsonomy/posts/post/@postingdate"/>
   </representation>
    <fault status="400" mediaType="application/xml"
      element="bibsonomy">
      <param name="statusName" style="plain"
        path="/bibsonomy/@stat"/>
    </fault>
</response>
<!--Create Post Response-->
<response>
  <representation mediaType="appication/xml"
    status="201"/>
</response>
```

Listing 4.4: Response definition

Enclosed by a `response` element the resource `representation` returned on performing the request is defined. The `mediaType` attribute indicates an XML document with a root `element` *bibsonomy*. `Param` denotes a particular element within the result. Its `path` attribute contains an XPath expression yielding an element's value. This explicit element specification eases the selection of desired result elements in visual workflow design, preventing a user from manually typing XPath expressions.

A `fault` element indicates possible invocation errors where the value of the `status` attribute defines the HTTP error code. A `param` element denotes a single result node within the XML payload identical to the definition of result nodes in the `representation`. For a *Create Post* response a generic XML document (`mediaType="application/xml"`) with an associated HTTP status code of 201 is specified.

4.1.2. Strengths and Limitations

WADL allows for detailed specification of the interaction behavior of almost arbitrary web resources. The description format closely follows the semantics of REST while being highly flexible and easy processable. In respect of a workflow integration of web resources described through WADL the exact specification of input- and output-parameters including their primitive types and the declaration of particular XML result elements via XPath expressions are among the most viable benefits of the description language.

In his release statement of the current version Hadley mentions two issues he considers important in future versions. One is the declaration of authentication mechanisms, which is currently completely disregarded. Due to a high diversity of techniques in actual REST APIs going beyond easy describable basic HTTP authentication the author believes that more research on this topic will be necessary. The second issue mentioned is an improved support for specific data formats, as for example JSON. Considering the integration of XML schema definitions and the specification of single output elements via XPath expressions are especially relevant to successful service integration, these functionalities would assuredly be desirable for JSON too, being the standard data format in many main stream service implementations today.

Though generative URIs not fully known until runtime are supported by denoting parameter names with curly braces [1], dynamic URIs being generated at runtime can be viewed as a flaw in the static resource description concept of WADL. A resource URI created during execution of a workflow through a HTTP PUT for instance cannot be known in advance to be included in a WADL description. Hence further references to such a resource within the workflow would have to be specified manually during workflow design. A promising approach addressing this issue is the dynamic WADL mechanism introduced in the latest release of Jersey [41].

Another issue associated with the description of fixed resources is yielded by the concept of virtual URI names, utilized for example in the Amazon Simple Storage Service, S3[2]. In S3 a virtual URI, requiring a unique base URI component (http://bruckner.s3.amazonaws.com) unambiguously defines a bucket providing a data storage location. In

[1] refer to section 2.5.1 in the WADL specification [42] for examples

[2] https://s3.amazonaws.com/

order to provide a WADL specification of the service, a custom description would have to be generated for each individual bucket.

4.1.3. Workflow Integration Support

Having now exemplified WADL's opportunities for exact description of RESTful Web services, its benefits for workflow integration remain to be clarified. In debates [3] on the necessity of description languages for REST considerations on improved workflow integration capabilities facilitated by a RESTful service description are rather rare. A consensus among REST advocates is to retain the light weight character of REST in favour of a set of description standards comparable to the WS* stack. The header of W3C's mailing list on the topic[4] states that REST description shall in contradiction to WSDL not be aimed at the description of Web services unlike Marc Hadley's original description of WADL in his introductory web log entry [40]:

> WADL is designed to provide a simple alternative to WSDL for use with XML/HTTP Web applications.

However it is clear that the formal service description provided by WADL can be a viable foundation for its initial workflow integration. Though currently no standardized workflow description language allowing for native integration of WADL exists we believe that especially for resource interactions requiring a more complex description than a GET request, a POST request carrying an XML message payload for example, a formal description is indispensable. In our implementation (5) we show how Windows Workflow Foundation (WF)- components are automatically generated based on arbitrary WADL descriptions. Though this injective integration process omits the initial WADL description in the final workflow, as service attributes are translated into C# source code, it eases workflow design considerably. Without a service description the integration of RESTful services in WF would require programming skills, as currently no support for direct REST integration comparable to WS* is provided.

In the service integration wizard of the expressFlow, described in section 6.2 WADL plays a vital role as well. The resource definitions of a WADL document are parsed dynamically in order to be presented in a selection list of available request URIs. The ex-

[3] http://www.infoq.com/news/2007/06/rest-description-language

[4] http://lists.w3.org/Archives/Public/public-web-http-desc/

act specification of input parameters and their according type definitions contained in a WADL description allow for the data binding specification between various services invoked within a workflow. Input parameters are presented equivalently to WSDL message type elements abstracting from the different description standards for RESTful and WS* based Web services. An output element of a WSDL based Amazon item search request could be bound to the input text parameter of a Google Translate service request described in a WADL document for example. The parameter type definitions of a WADL resource description may be utilized for design time validation of parameter compatibility in future versions of the wizard implementation. In custom WF activities for RESTful service invocation, detailed in section 5.3, parameter type validation is facilitated through the strongly typed *Dependency Properties* (see section 5.3.1).

4.2. Describing a RESTful Web Service in WSDL 1.1

The Web Service Description Language (WSDL) provides a standard XML format for syntactical description of Web services. It was developed by Microsoft, IBM and Ariba and submitted to the World Wide Web Consortium (W3C) in 2001. Despite the fact that its by far most popular version 1.1 is still not considered an official W3C standard but a note mainly released for discussion purposes (as stated in [16]) it has evolved to a major foundation in SOA implementations, with a broad tooling and programming language support.

In this section the capabilities of WSDL 1.1 for description of RESTful Web services will be detailed preceded by a brief summary of its basic document structure. In the subsequent section the revised facilities for RESTful service description introduced in WSDL 2.0 will be outlined. The introduction of the according language elements is accompanied by excerpts of the example Bibsonomy service description.

4.2.1. WSDL 1.1 Document Structure

WSDL 1.1 defines five basic elements, conceptually grouped into an *abstract* (`types, message, portType`) and a *concrete* (`binding, service`) part:

- `<types>` : Encloses the type description of parts (parameters) of the exchanged messages. Typically XML schema is used as description format, as it is also marked as *preferred* in the W3C recommendation. While it provides a very flexible and powerful mechanism for description of almost arbitrary data structures, its parsing can be cumbersome as the experience during the implementation of a WSDL parsing component for expressFlow showed. Other type definition systems may be included via extensibility elements.
- `<message>` : Here the signature of input-/output-messages is defined via message `part` elements. A part type can either be declared directly as simple XML schema type (xsd:int, xsd:string,..) via a `type` attribute or reference an `element` definition within the `types` section of the document. While the specification provides a high degree of flexibility in this part, it also increases the complexity of automatic parsing and mapping the provided information to other description formats such as WADL (see section 6.2.4).
- `<portType>` : `PortTypes` define sets of abstract operation definitions, i. e. the named methods provided by a service and their associated input-/output-messages. Fault messages can also be declared here, commonly being translated to runtime exception definitions by WSDL tool kits.
- `<binding>` : `Binding` defines the format and transport protocol of exchanged data for a certain `portType`. Mostly SOAP over HTTP is used, defined via a child element `<soap:binding>` with a `transport` attribute value of `http://schemas.xmlsoap.org/soap/http`. For RESTful Web service description the HTTP binding [5], examined in section 4.2.2, is particularly important.
- `<service>` : A `service` aggregates a `binding` and a communication endpoint into a `port` definition.

A common practice is to depict the separation of abstract and concrete elements in the file structure of a WSDL description. The WSDL document is partitioned into two files, where the abstract document is imported in the concrete document.

The overall structure of WSDL seems to be overly complex at first sight due to its structure aimed at decoupling and high reuseability of its components. The separation of abstract operation definitions in a *portType* and its concrete protocol bindings aims at

[5] `http://www.w3.org/TR/wsdl#_http`

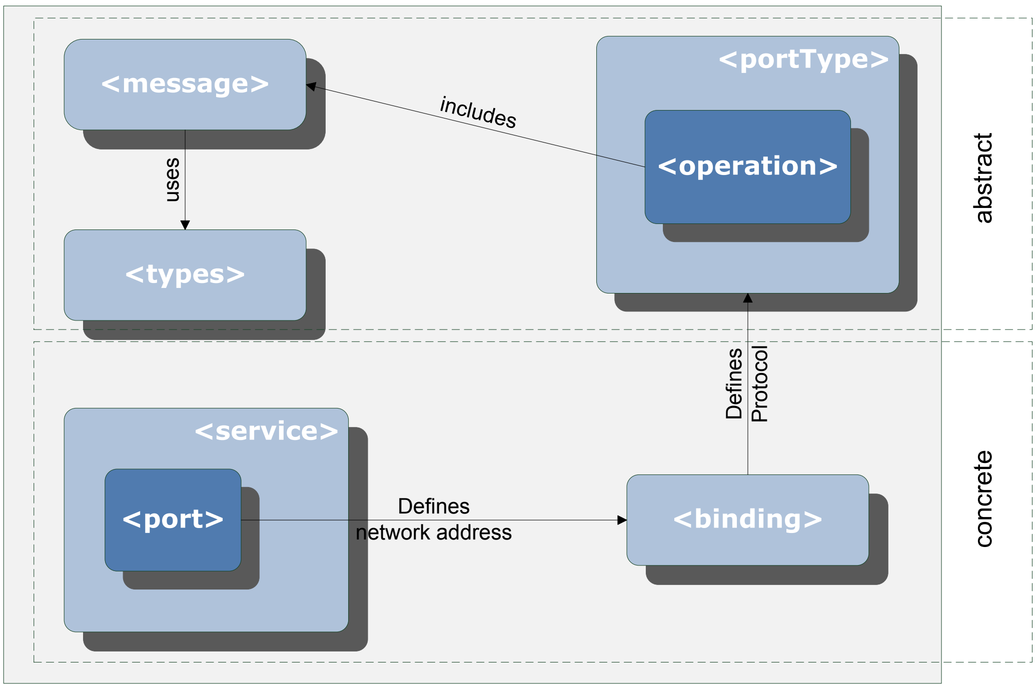


Figure 4.2.: WSDL 1.1 Basic Elements

providing a means for defining multiple message transport protocols as for example HTTP and SMTP. The decoupling of messages from their associated operations allows for their reuse among multiple operations of a service interface. In fact this results in highly complex service descriptions being completely unreadable to human users and causing difficulties in WSDL tool implementations. During implementation of a WSDL processing component for this thesis the 4 mega bytes ebay WSDL document induced message buffer overflows at various points of the application regularly.

4.2.2. WSDL 1.1 HTTP Binding Exemplified

In this section the capabilities for RESTful service description provided by the WSDL 1.1 HTTP binding specification will be clarified on the basis of the Bibsonomy example, introduced in section 4.1.1.

Types Definition

```
<types>
  <xs:schema targetNamespace="http://example.org/Bibsonomy/WSDL2.0">
    <xs:include schemaLocation=
    "http://www.bibsonomy.org/help/doc/xmlschema.xsd"/>
  </xs:schema>
</types>
```

Listing 4.5: Extract Bibsonomy Types Definition

Type definitions for messages exchanged during interaction with the service are enclosed by the `types` element. In order to allow for the specification of XML message payloads as for example a bookmark element formatted in XML, the Bibsonomy XML schema definition is imported via an XML schema include statement. This results in a more compact description compared to an inline definition of message types.

Message Definition

```
<message name="GetAllPostsRequest">
  <part name="user" type="xsd:string"/>
  <part name="tags" type="xsd:string"/>
  <part name="resourcetype" type="xsd:string"/>
</message>

<message name="GetAllPostsResponse">
  <part name="posts" type="tns:posts"/>
</message>
```

Listing 4.6: Bibsonomy Message Definition

A *Get Posts* request requires three input parameters represented by according `message part` elements defined in the *GetAllPostsRequest* message. The message definition does not contain declarations specific to the HTTP request, as for example the parameter syntax, keeping the WSDL description generic in this part. All HTTP-/REST-specific definitions are included within the `binding` and `service` definitions.

In the *GetAllPostsResponse* message, returning an XML formatted lists of posts matching the request, a single `part` element references the *posts* XML element defined in the imported XML schema document.

The definition of associated `portType` definitions can be found in the complete WSDL 1.1 document in Appendix B.

Binding and Service Definition

```
<service name="BibsonomyPostsService">
  <port name="BibsonomyGetPostsPort"
    binding="tns:BibsonomyGetPostsBinding">
    <http:address location="http://bibsonomy.org"/>
  </port>
  ...
</service>

<binding name="BibsonomyGetPostsBinding" type="tns:PostsPort">
  <http:binding verb="GET"/>
  <operation name="GetAllPosts">
    <http:operation location="posts"/>
    <input>
      <http:urlEncoded/>
    </input>
    <output>
      <mime:content type="text/xml"/>
    </output>
  </operation>
</binding>
```

Listing 4.7: Bibsonomy Binding and Service Definition

The location attribute of the `http:address` element contained in a `port` definition defines the base component of a service URI, comparable to `resources base` in a WADL description. Its value is prepended to the `location` value of an `http:operation` element in the `binding`. The complete request URI minus input parameters for a `GetPosts` request is now http://bibsonomy.org/posts.

In order to complete the request definition finally its input parameter syntax has to specified. The WSDL HTTP binding provides two alternative mechanisms for this task:

- `http:urlEncoded` specifies the encoding of all message parts either into the query string or into the body of a POST message. This definition is mutually exclusive, meaning that parts of a single message cannot be separated into query string and body parts - see section 4.2.3 for a more detailed discussion. The syntax of a final URI follows the standard name1=value1&name2=value2 convention. A "?" character prefixes the parameter enumeration by default.

- `http:urlReplacement` allows for the definition of URI patterns within a HTTP request. Message parts are defined using their names as defined in the message definition enclosed by parenthesis "()". The search patterns are included within the relative URI part, defined in the location attribute of `http:operation`, as for example location="api/posts/(user)". This limits the flexibility of the mechanism in respect of POST / PUT support, where dynamic request elements would have to be included in the request *body* as well. However adhering to the URI template principle [47] the mechanism supports meaningful resource URIs (http://endpoint-.uri/posts/johnsmith/) and provides syntactical alternatives to the syntax convention of `urlEncoded` requests.

In the example `http:urlEncoded` is used enclosed by an `input` element within the `operation` definition of the `binding`. The resulting request URI is http://www.bibsonomy.org/api/posts?user=<user>&tags=<tags>&resourcetype=<resourcetype>.

The output format is defined as plain XML via the `type` attribute <`mime:content`>. For messages containing more than one output part the format of each part has to be declared separately using a <`mime:content`> element with a `part` attribute referencing the message part name.

The appropriate HTTP method is defined in the <`http:binding`> element via the `verb` attribute. Though its value is formally not restricted to GET and POST [6], the specification title ("HTTP GET & POST Binding") and the limitations mentioned in the subsequent section show that the HTTP binding mechanism is in fact limited to only a small subset of possible HTTP GET and POST based resource interaction scenarios.

4.2.3. WSDL 1.1 HTTP Binding Limitations

1. *Supported HTTP methods* - The limitation of supported HTTP methods to GET and POST is one of the major drawbacks of the HTTP binding concept in WSDL 1.1. In order to support the uniform interface principle of REST at least PUT and

[6]http://www.w3.org/TR/wsdl#_http:binding

DELETE would have to be supported as well, however requiring more flexible mechanisms for request syntax specification.

2. *Missing Support for HTTP Header Specification* - Another vital REST service characteristic omitted in the WSDL 1.1 HTTP binding is the HTTP header. Standardized HTTP header fields, allowing for example passing authentication information (*Authorization*) or the specification of a desired response format (*Accept*) cannot be specified.

3. *Missing Fault Declaration Support* - The specification considers fault declaration only for operations with a SOAP binding. In the HTTP binding a fault declaration concept allowing for mapping HTTP status codes and optional XML payloads to invocation error conditions is completely omitted. In the HTTP binding extensions of Apache ODE, an open source BPEL engine implementation with basic support for RESTful service invocation, a solution mainly based on the XML content of an error response [33] is proposed. However processing of the HTTP status code is also neglected there.

4. *Binding Level Method Specification* - The HTTP method for an operation is defined at the top level of a binding definition, permitting only one method per binding. This leads to considerable overhead in cases where multiple HTTP methods are allowed for one single resource. For each method associated with this resource a separate binding would have to be defined. A preferable alternative would be the integration of the HTTP method into the operation definition of a binding, as demonstrated in the following example:

```
<binding name="BibsonomyPostsBinding" type="tns:PostsPort">
  <operation name="CreatePost">
    <http:binding verb="POST"/>
    <http:operation location="posts"/>
  </operation>
  <operation name="DeletePost">
    <http:binding verb="DELETE"/>
    <http:operation location="posts/(postID)"/>
  </operation>
</binding>
```

Listing 4.8: Defining the HTTP Method at Operation Level

4.3. Describing a RESTful Web Service in WSDL 2.0

4.3.1. Document Structure - Changes from WSDL 1.1

Subsequently the major changes in the basic document structure of WSDL 2.0 from WSDL 1.1 will be elucidated preceding a detailed view on its improved HTTP binding mechanism. A good starting point into the complete specification is the primer [8], offering a less formal and better exemplified standard description compared to the core and adjuncts part. However the creation of a complete WSDL 2.0 description for the aforementioned Bibsonomy example a deeper look into the adjuncts part [13] of the specification was required as well.

- The *root element* name of a WSDL 2.0 document is `<description>`. It includes a mandatory `targetNamespace` attribute, comparable to an XML schema targetNamespace. All definitions contained in a WSDL 2.0 description, such as interface- , binding- and service- definitions are associated with this namespace. The import- and inheritance- facilities, permitting linkage of multiple WSDL documents and referencing of abstract interface definitions constitute the necessity of a targetNamespace declaration. Its recommended value is a URI identifying the address of the document.

- A simplification compared to WSDL 1.1 is the omittance of the `<message>` element. Data types of exchanged messages are still described via XML schema but can now be referenced directly from within an operation declaration. Thus the reuse effect of type definitions is retained while gaining a more compact description.

- The `<portType>` element is renamed to `<interface>`. The interface mechanism allows for inheritance via an `extends` attribute. The declaration of operation faults has been shifted one level up to the interface level, allowing for reuse of fault definitions across the operations of an interface. Combined with the inheritance mechanism this can for example be utilised for describing an interface containing common fault declarations to be extended by subsequent interface definitions within the description, as illustrated in the following example taken from the WSDL 2.0 Primer [8]:

```
<interface name="creditCardFaults">
  <fault name="cancelledCreditCard"
    element="cc:CancelledCreditCard"/>
  <fault name="expiredCreditCard"
    element="cc:ExpiredCreditCard"/>
</interface>

<interface name="reservation" extends="tns:creditCardFaults">
  <operation name="makeReservation">...</operation>

  <outfault ref="tns:cancelledCreditCard" messageLabel="Out"/>
  <outfault ref="tns:expiredCreditCard" messageLabel="Out"/>
</interface>
```

Listing 4.9: Utilization of the inheritance concept for abstract fault declarations

- A `<binding>` element may be implicitly declared *reusable* by omitting a reference to a particular interface. A reusable binding is then associated with an interface in the `<service>` element, grouping a binding and an according interface into an `<endpoint>`. The latter replaces the `<port>` element of WSDL 1.1.

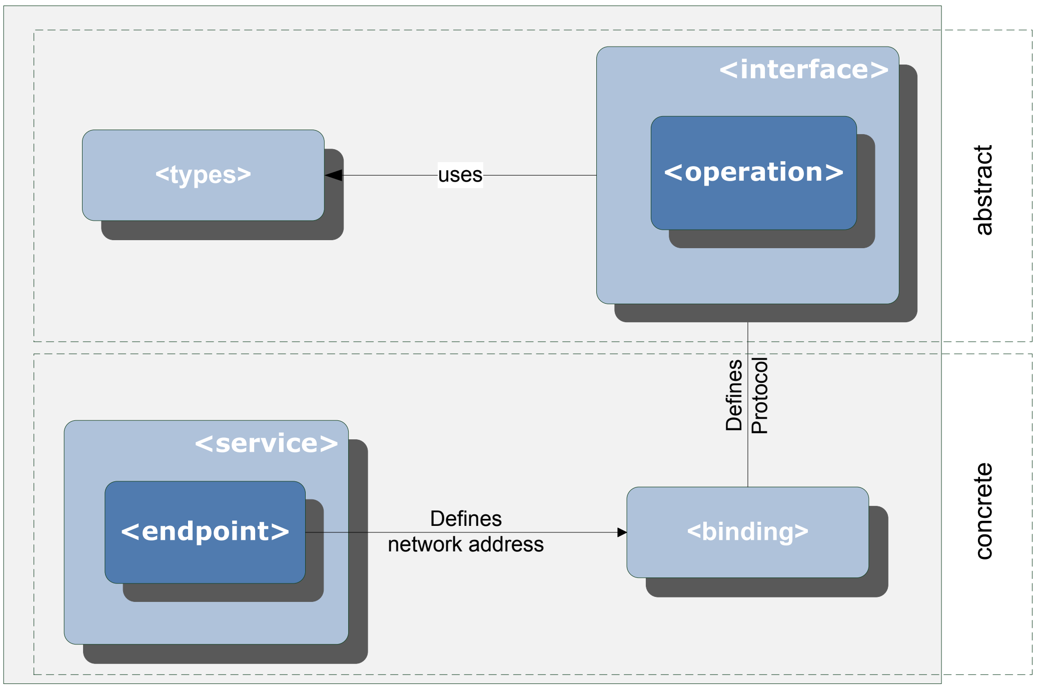


Figure 4.3.: WSDL 2.0 Basic Elements

4.3.2. WSDL 2.0 HTTP Binding Exemplified

As in preceding sections the introduction of WSDL 2.0 and its HTTP binding mechanism is accompanied by excerpts from the complete service description, to be found in Appendix C of this thesis.

Types Definition

```
<types>
  <xs:schema targetNamespace="http://example.org/Bibsonomy/WSDL2.0">
    <xs:include
  schemaLocation="http://www.bibsonomy.org/help/doc/xmlschema.xsd"/>

    <xs:element name="userPosts" type="tGetPostsIn"/>

    <xs:complexType name="tGetPostsIn">
      <xs:sequence>
        <xs:element name="user" type="xsd:string"/>
        <xs:element name="tags" type="xsd:string"/>
        <xs:element name="resourcetype" type="xsd:string"/>
      </xs:sequence>
    </xs:complexType>
</types>
```

Listing 4.10: WSDL 2.0 Types Definition

The `types` definition mechanism in WSDL 2.0 has not changed from WSDL 1.1 syntactically, but is now the core location for message type declaration. Due to an omission of the message construct input and output message details are fully declared within the types section. The schema import mechanism, being now explicitly described in the WSDL 2.0 specification, prescribes the declaration of a schema with a `targetNamespace` declaration, for import of schemas missing this attribute. Hence a `schema` element is defined inside the types declaration, where the referenced Bibsonomy schema is integrated via an `include` element. The use of `include` instead of `import` makes an element declared in the imported schema available for reference using its *q-name*[7].

[7]A very detailed yet understandable explanation is given in the WSDL 2.0 Primer: http://www.w3.org/TR/wsdl20-primer/#more-types-schema-import

The *userPosts* element specifies the input message type for the *getPosts* operation. According to the specification a message type has to be declared as single element with an arbitrary[8] substructure. The type attribute references a `complexType` with a `sequence` enumerating the query parameters of the operation. HTTP serialization rules, defining whether input message parts are contained in the query string, or the message body for instance, are specified in the bindings section (see section 4.3.2).

Interface and Operation Definition

```
<interface name="PostsInterface">

  <fault name="unauthorizedRequest"/>
  <fault name="badRequest"/>

  <operation name="GetAllPosts"
    pattern="http://www.w3.org/ns/wsdl/in-out"
    style="http://www.w3.org/ns/wsdl/style/iri"
    wsdlx:safe="true">

    <input messageLabel="in"
      element="tns:userPosts"/>
    <output messageLabel="out"
      element="tns:posts"/>
    <outfault ref="tns:badRequest" messageLabel="out"/>
    <outfault ref="tns:unauthorizedRequest" messageLabel="out"/>

  </operation>
</interface>
```

Listing 4.11: WSDL 2.0 Interface and Operation Definition

The `interface` element provides an abstract specification of the operations provided by a service. Apart from its name change it differentiates from the `portTypes` concept of WSDL 1.1 by introducing interface inheritance. Via the `extends` attribute an interface may be derived from other interfaces.

The *GetAllPosts* operation introduces new operation attributes, two of which are particularly important for services with a HTTP binding. The first one is the `style` attribute

[8]Though the type is claimed to be arbitrary in the Primer, the element structure partly depends on the style attribute value of its referencing operation (see 4.3.2)

indicating an *Internationalized Resource Identifier(IRI)* style, basically defining the following restrictions[9] for the XML schema representation of operation parameters:

- The element referenced in an operation must be a single schema element (i. e. userPosts).
- This element is defined as a complex type including a sequence. No other structure elements such as xs:choice are allowed.
- Elements of this sequence are local elements with a simple type (for example xs:string, xs:int).

Due to these declaration restrictions the serialization of input parameters into an according request URI string is ensured. A sequence of simple elements can be mapped to a query string unambiguously, as discussed more detailed in section 4.3.2.

The second operation attribute important to RESTful service description is `wsdlx:safe`. An operation denoted as safe can essentially be characterized through not entailing any obligations to a client invoking this operation. For a REST architecture this implies an HTTP request not causing a change to a resource's state. Most likely this will be a GET request, as in our example, though the HTTP method name is not restricted by the concept [46].

In the `pattern` attribute of an operation element a message exchange pattern is explicitly set. WSDL 2.0 provides eight predefined patterns and allows for the definition of new patterns, identified by a unique URI.

A conceptual improvement from WSDL 1.1 is the dissociation of `fault` declarations from operations, providing their reuse throughout multiple operations of an interface. Operation faults are declared abstract and referenced within `infault` or `outfault` elements of an operation declaration. The interface binding (see section 4.3.2) describes protocol specifics of a fault, such as the status code of an HTTP response message.

[9]The trivial requirements as defined in the specification are left out for brevity. Refer to the adjuncts for a complete list: (`http://www.w3.org/TR/2007/REC-wsdl20-adjuncts-20070626/#_operation_iri_style`)

HTTP Binding and Endpoint Definition

```
<binding name="PostsBinding"
  type="http://www.w3.org/ns/wsdl/http"
  interface="tns:PostsInterface"
  whttp:methodDefault="GET"
  whttp:queryParameterSeparatorDefault="&">

  <fault ref="tns:badRequest" whttp:code="400"/>
  <fault ref="tns:unauthorizedRequest" whttp:code="401"/>

  <operation ref="tns:GetAllPosts"
    whttp:inputSerialization="application/x-www-form-urlencoded"
    whttp:location="/api/posts/{userPosts}"/>

  <operation ref="tns:CreatePost"
    whttp:inputSerialization="multipart/form-data"
    whttp:method="POST" whttp:location="/api/users/{user/}/posts"/>

</binding>

<service name="PostsService"
    interface="tns:PostsInterface">

    <endpoint name="PostsEndpoint"
      binding="tns:PostsBinding"
      address="http://www.bibsonomy.org"
      whttp:authenticationScheme="basic"
      whttp:authenticationRealm="BibSonomyWebService"/>
</service>
```

Listing 4.12: WSDL 2.0 HTTP Binding

Apart from the *style*- and *safe*-attributes of the operation definition the service description does not contain REST specific elements so far. All definitions specific to RESTful interaction description go into the HTTP *binding* and its associated *endpoint* definition. As subsequently demonstrated, WSDL 2.0 is syntactically sufficient for technical description of all characteristics of the Bibsonomy API, including the authentication specification.

The binding type is set in the `type` attribute of the `binding` element. Its attributes `methodDefault` and `queryParameterSeparatorDefault` specify default properties affecting all operations of an interface associated with the binding. These default rules may be overridden in particular operation definitions, as in the *CreatePost* operation, where POST is defined as HTTP method instead of GET.

A `fault` element specifies according HTTP status codes for the abstract error definitions introduced in the interface definition.

The definition concept for *input serialization* of HTTP requests has improved significantly from WSDL 1.1. It allows among others for partitioning input message elements into URI string and body components. The language elements required for exact specification of request URI and message body format are:

- The optional `style` attribute of an interface level operation definition, putting syntactical restrictions on the XML schema representation of an input message. This implicitly affects the request format, as for example a value of *style/iri* is required for serialization of input message elements into query string parameters.
- `QueryParameterSeparator` defines the separation character for query parameter enumerations of a query string or an *application/x-www-form-urlencoded* message body. Most commonly a "&" character is used.
- The `inputSerialization` attribute, indicating the serialization data format of a request. The attribute value *multipart/form-data*, used in the *CreatePost* operation definition, indicates an *XML* formatted input message body.
- The `location` attribute of an operation allows for a detailed syntactical description of the request URI. Dynamic URI components are denoted by curly braces "{}" and are replaced with instance data at run time. A viable improvement from WSDL 1.1 is the optional partition of request parameters into the query string and the message body: If the name of a template parameter is prepended with a "/", for example {user/}, all succeeding parameter values are serialized into the message body[10].

All remaining service properties are specified within the `endpoint` element. The `address` element defines a base address component prepended to all operation locations. The attributes `authenticationScheme` and `authenticationRealm` are used to declare a basic HTTP authentication mechanism.

[10]http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/#more-bindings-http

4.3.3. Description Capabilities and Workflow Integration

Improved Description Concepts. As compared to the specification facilities for RESTful Web services provided by the HTTP binding concept of WSDL 1.1, the range of supported REST characteristics has widened significantly in WSDL 2.0. Among the notably conceptual improvements are the operation attributes `style`, allowing for a formal mapping of the XML schema type definitions to the request serialization of an HTTP request, and the `safe` attribute. The explicit designation of safe operations through the latter attribute could be utilized in static workflow analysis, in order to reveal design flaws caused be repetitive *unsafe* requests onto a particular resource during execution of a workflow (refer to section 2.2 for a detailed discussion of this issue).

WSDL 2.0 is also superior to WADL regarding the specification capabilities for HTTP *authentication*. An endpoint definition may be complemented by a specification of the required authentication mechanism, being either HTTP *basic* or *digest* authentication [36].

Moreover the greater flexibility of the serialization rules for request input parameters can be considered as a major advantage of the WSDL 2.0 specification. Comparable to the parameter definition concept of WADL (see section 4.1.1) request parameters may be serialized into parts of the request URI as well as the message body.

Workflow Integration. Though approaches for native integration of REST semantics into WS-BPEL based workflow definitions have been recently proposed [71, 68] WSDL 2.0 should not be neglected in considerations of workflow integration for RESTful Web services. Due to its conceptual resemblances to WSDL 1.1, being already satisfactory supported in existing WS-BPEL based workflow applications WSDL 2.0 might emerge to a viable concept in future solutions. However the relevance of the specification in this context will mainly depend on the future acceptance of WSDL 2.0 in RESTful and WS* based Web service implementations, and its implication on future versions of the WS-BPEL standard specification. Among related workflow solutions previously introduced in this thesis (see section 3.1.2) currently only Apache ODE [33] provides basic support for WSDL 2.0 endpoint references.

4.4. Alternative Approaches Summarized

To complete an overview of the REST description language landscape this section provides a glimpse at alternative approaches for machine processable REST service descriptions. Many of them are more or less ad hoc concepts solving only small subsets of the REST description problem while omitting software implementations completely. The introduced concepts represent a selection of David Orchard's list of REST description languages [63].

4.4.1. WDL

David Orchard has been the Web standards lead at BEA systems until its acquisition by Oracle in 2008 and committed to the HTTP binding specification of WSDL 2.0. He proposes the *Web Description Language* (WDL). The main concept behind WDL is the description of a Web application as a set of resources, instead of interfaces associated with bindings and endpoint definitions as known from WSDL 2.0. Hence the proposal tries to avoid specification overhead by focusing on description of RESTful services only. Essentially a WDL description is a set of resources, each of which may include 4 attributes:

- location
- authenticationType
- authenticationRealm
- httpVersion

The definition of operations, included within a resource definition, reminds of the WSDL 2.0 HTTP binding concept. URI path components are denoted by template expressions and replaced through schema instance data at runtime, instead of specifying each parameter separately. Input and output elements may include specifications for HTTP header components and HTTP status codes. An output element may declare its type by referencing a schema type definition. An example WDL description of Yahoo's news search can be found at the specification site [64].

4.4.2. SMEX-D

In May 2005 Tim Bray, Director of Web Technologies at Sun Microsystems, proposed SMEX-D [10], an XML language for description of SOAP and REST based Web services. SMEX-D stands for Simple Message Exchange Descriptor, according to the message exchange behaviour being a basic characteristic of every Web service. The main elements of a SMEX-D instance are <request> and <response>. A request may have three different formats: a list of name value pairs, a Non SOAP XML (NSX) message and a SOAP message. A response may be empty in case of a one way operation, or contain either an NSX or a SOAP formatted response. <Pair> elements contained within a request may have an explicitly declared primitive XML schema type or contain an enumeration of allowed values included in <enum> elements. Their request encoding is only informally regulated by following the common encoding mechanisms for HTTP GET and POST. Request response messages declared as NSX or SOAP may reference XML schema descriptions. However an exact specification of response elements is omitted, making the description rather unfeasible to workflow integration purposes. A SMEX-D example description of Amazon's ItemSearch is provided at [9].

4.4.3. NSDL

Norm's Service Description Language, NSDL, was introduced by Norman Walsh, Principal Technologist at Mark Logic Corporation in his blog [95] in 2005. The initial motivation for developing the language was the authors struggle with WSDL during his development of a sample Web service called WITW (Where in the World). NSDL aims at describing services accessible via HTTP GET and POST, omitting many of the concepts contained in WSDL as for example abstract port type definitions or protocol bindings. A <service> element defines the intended HTTP method as well as a URI and encloses subsequent <request> and <response> definitions. A request declaration allows for enumeration of input parameters associated with an XML schema type. Support for POST requests containing an HTTP body is ensured by an optional <body> element to be included in a request definition. It may contain XML content, parameters are bound through template expressions. The designation of result elements of interest is facilitated by XPath select statements contained in the <result> and <fault> children of a response declaration. Along with the specification Walsh provides three Perl modules NSDL::Request, NSDL::Response and NSDL::UA (User Authentication), allowing for transparent programmatic access to service operations.

4.4.4. RSWS

Richard Salz, among other internet standard activities contributor to HTTP 1.0 and HTTP 1.1, proposed a concept called Really Simple Web Service Descriptions, RSWS [81], separating a service description essentially in three parts:

- The message *schema* definition
- An *interface* definition containing provided operations
- A *location* definition, describing the actual service location

The container element for these parts is <description>, containing a name attribute defining a URI. The three child elements <schema>, <interface> and <location> may be shared among descriptions through reference via their id attribute. In order to support arbitrary schema languages the schemaType may be declared in a schema definition. An <operation> contained within an interface definition includes subsequent <input>, <output> elements, implicitly constituting a request-response message exchange pattern. Transport specific information, such as the service URI, the protocol (for example soap over http) and encryption requirements go into the <provides> child element of a location definition.

5. Automatic WADL and WF Code Generation

This chapter details the implementation of a WADL- and Windows Workflow Foundation [52] code-generation library, forming the basis for URI based service integration in the expressFlow Service Integration Wizard (see section 6.2). The original implementation[1], emerging from an internship at the Distributed Systems Group of the Vienna University of Technology, provided both functionalities through a WS* Web service interface. The focus of this chapter is on WADL and WF Code generation components, all service oriented interface aspects will be omitted subsequently.

Parts of the implementation result from a porting of Thomas Steiner's "Rest Describe & Compile" project [28] into C#. The Java source code of the application, implemented with the Google Web Toolkit [39], is released under the Apache License 2.0. Due to the strong similarity of the two languages, regarding the language syntax as well as library functions, the C# port fully depicts the original functionality. Minor modifications have been made to the regular expressions used in the type estimation of request parameters.

Section 5.1 details the internal architecture of the library including a description of the contained C# namespaces and an UML class diagram depicting the main implementation classes and their relations. In sections 5.2 and 5.3 core features and according implementation issues of the WADL and WWF Activity code generation components are described. A summary of the underlying technology stack in section 5.4 concludes the chapter.

[1]The complete source code and documentation is available from the Google Code project site [28]

5.1. Internal Architecture

Figure 5.1 shows an UML class diagram of the classes involved in WADL processing and code generation. Each supported WADL language element is encapsulated in a corresponding class named after its element name. The `Analyzer` class provides processing of a provided URI and HTTP method into a corresponding WADL description. `TypeEstimator` implements type guessing based on parameter names and their values.

In the `WadlXml` class a mapping between WADL specification elements and implementation constructs is defined, in order to convert the memory WADL representation into a corresponding XML string. This process is inverted in the `WadlParser` class, where a given WADL document is parsed into a memory WADL representation. Finally the `CodeGenerator` class implements the generation of Windows Workflow Foundation Activities from a WADL description.

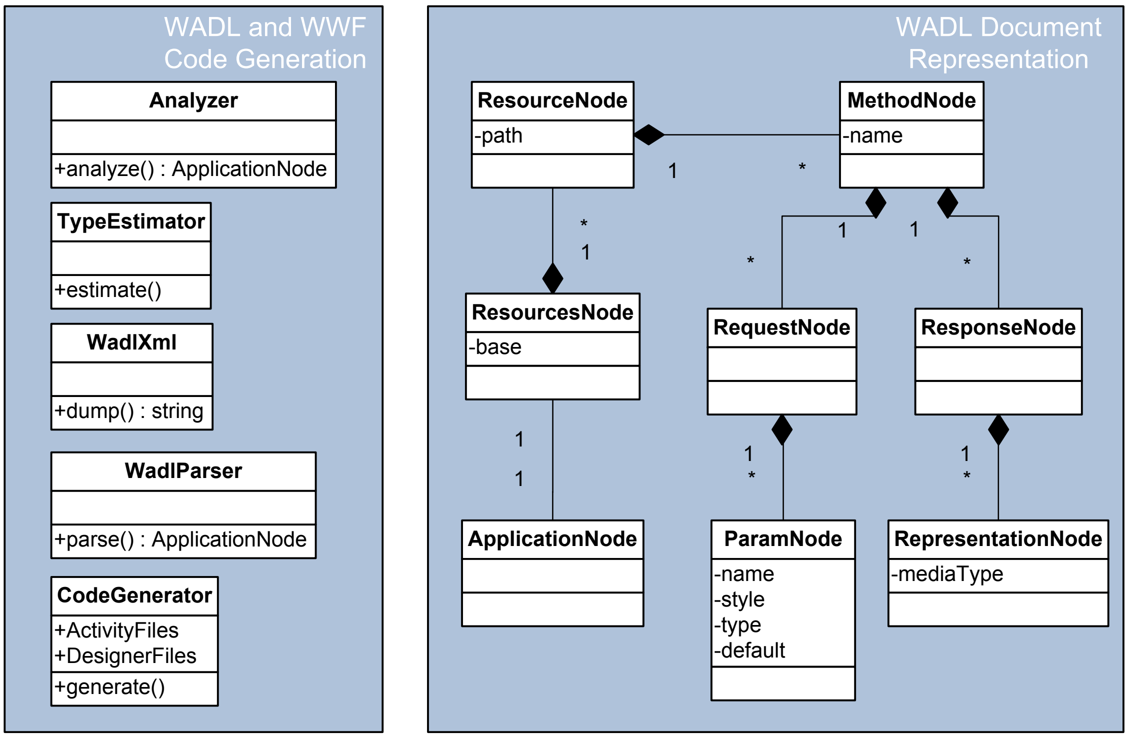


Figure 5.1.: WADL and WF Code Generation - UML Class Diagram

The implementation is internally split into three main components, a `WADL` namespace implementing WADL creation and parsing, a `CodeGeneration` namespace providing

Namespace	Provided Functionality
RestDescribe.CodeGeneration	Creation of Workflow Components from a provided WADL file. Mapping of XML schema to C# types.
RestDescribe.Util	Estimation of query paramter types. URI parsing. Declaration of custom exceptions.
RestDescribe.Wadl	WADL language element mapping. URI anaylsis. WADL document parsing.

Table 5.1.: WADL Library - Namespace Structure

C# code generation and a `Util` namespace, providing helper functions to the other components. Table 5.1 summarizes these namespaces and their according functionalities.

5.2. WADL Generation

In order to initiate WADL generation first a `RestDescribe.Wadl.Analyzer` has to be instantiated, passing the request URI to the constructor. Its `analyze` method, taking the intended HTTP method as input parameter, launches the creation process. Starting with a `RestDescribe.Wadl.ApplicationNode` the nodes of a WADL representation matching the given URI and HTTP method are instantiated. Each node is associated to its successor, with the ApplicationNode forming the root of this structure. URI parsing is implemented in the RestDescribe.Util.Uri utility class, providing the following URI components: *base* (http://endpoint.org), *path* (/some/resource/a/) and *queryParameters* (?method=a&value1=b&value2=c). A standard query string syntax is presumed, with a "/" as path separator, a parameter list prepending the resource path with a leading "?" and a "&" as parameter separator.

Each element contained in the queryParameters list is associated to an XML schema type using the `RestDescribe.Util.TypeEstimator` class. The estimation mechanism mainly based on a regular expression matching of parameter-names and -values is introduced in section 5.2.2.

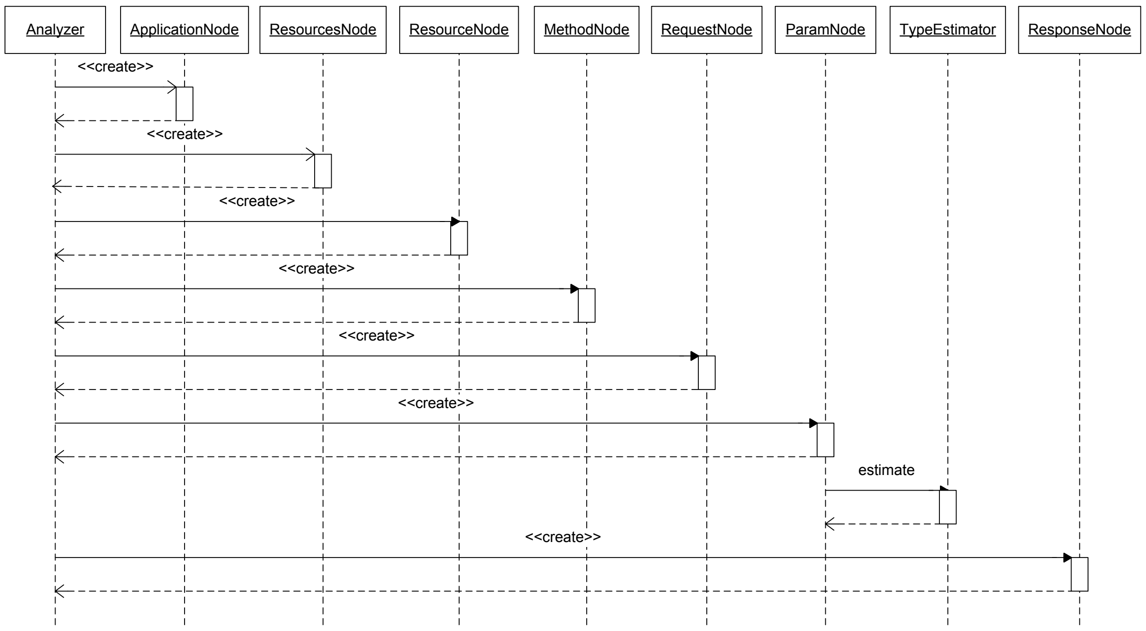


Figure 5.2.: WADL Generation - UML Sequence Diagram

5.2.1. Result Format

In order to determine the expected data format of an invocation response, a sample request to the provided Web service address is issued during WADL generation. Depending on the first character of the returned payload, being either "<" or "{", XML and JSON formats may be differentiated. The format specification is integrated into the generated WADL description by adding a `RestDescribe.Wadl.ResponseNode` with a subsequent `RestDescribe.Wadl.RepresentationNode` containing a `mediaType` attribute defining the response format. While the mechanism widens the number of support Web service APIs, it is on the other hand confined to operations not causing a change on a resource's state (for example data retrieval via HTTP GET).

The result format specified in a WADL document is then kept transparent in generated Workflow Activities (see section 5.3) through conversion of JSON formatted results into XML formated data at runtime. WF Activities provide a Web service invocation result through a uniform XML interface in two ways:

- A special `WantedNodes` property returning a System.XML.XmlNodeList containing nodes of interest. The desired element *name* can be set dynamically in the WF design surface via a String Property `WantedElement`.

- A System.XML.XmlDocument containing the complete invocation result, allowing for arbitrary programmatic processing of result elements, for example in workflow code components.

5.2.2. Type Estimation

The definition of service parameter types can be vital especially for integration purposes, allowing for static type checking at design time. However a service description solely based on its request URI does not include any explicit information about its parameter types. To overcome this issue a type estimation mechanism based on a fall through hierarchy of regular expressions is proposed in the original implementation. Following the decision path each *parameter-value* and *-name* is matched against a regular expression associated with a certain xml schema type. To express levels of certainty the three valued estimateQuality attribute is added to a type estimation result. In the final WADL document estimation quality information is included via a <doc> element of a <param> definition. Figure 5.2.2 shows an excerpt of the type estimation mechanism, refer to [88] for the full diagram.

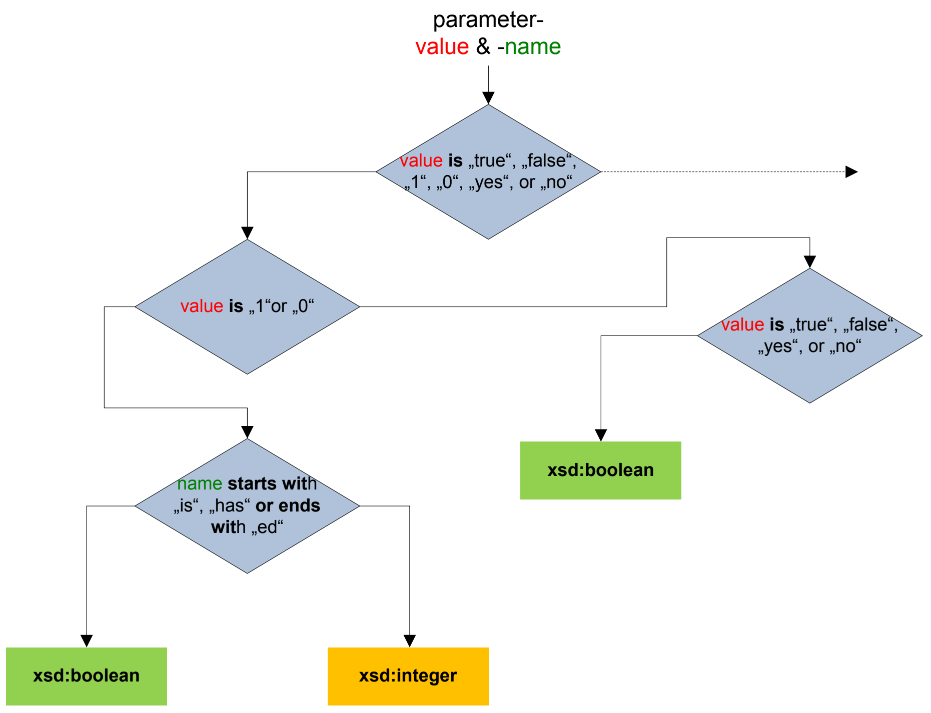

parameter-
value & -name
value is „true“, „false“, „1“, „0“, „yes“, or „no“
value is „1“or „0“
value is „true“, „false“, „yes“, or „no“
xsd:boolean
name starts with „is“, „has“ or ends with „ed“
xsd:boolean
xsd:integer

Figure 5.3.: WADL Generation - Type Estimation Decision Structure

5.3. WF Code Generation

In order to generate Windows Workflow Foundation Activities from a WADL file, an instance of type `RestDescribe.Wadl.WadlParser` is created first. The `parse` method, taking a string parameter representing a WADL document initiates the parsing process. In case of success a `RestDescribe.Wadl.ApplicationNode` instance is returned that can be passed to the constructor of a `RestDescribe.CodeGeneration.CodeGenerator`. The WADL document tree is processed in a top down manner. Upon reaching a request node the `createActivity` method is called where the actual code generation takes place.

Due to expressing the source code of a generated activity through a `System.CodeDom` [18] graph complex string expressions can be avoided. The CodeDom concept allows for abstract representation of a syntax tree first that can be put out in a desired .NET programming language. Theoretically C#, VB.NET, C++, J#, and JScript code can be generated from a CodeDom representation. In Microsoft Visual Studio 2008, used as IDE and test environment during implementation, C# and VB.NET based workflow projects are supported.

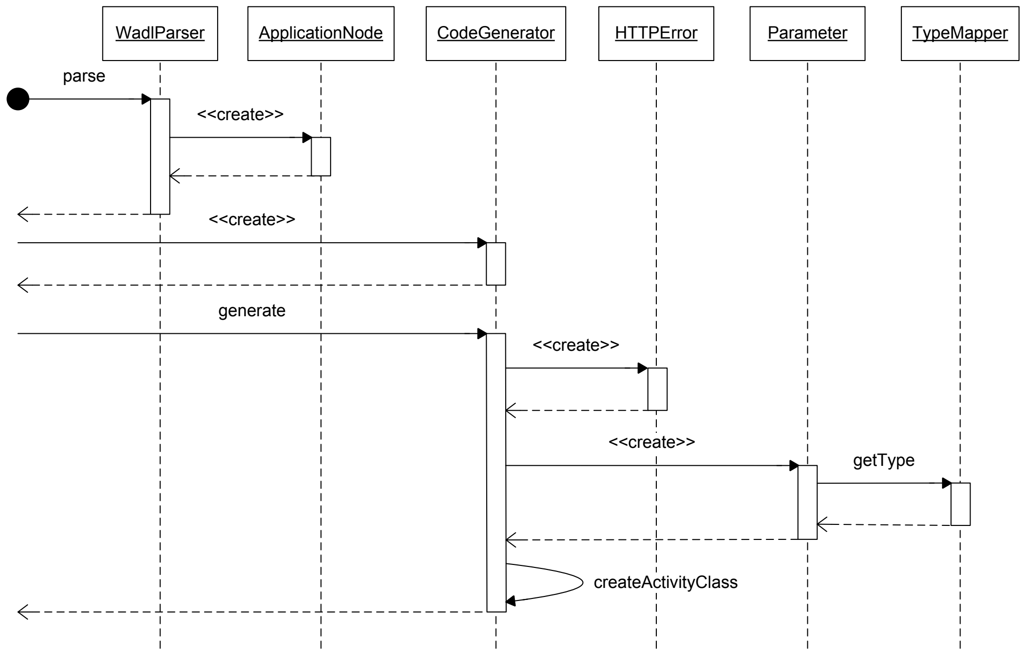


Figure 5.4.: WF Code Generation - UML Sequence Diagram

5.3.1. RESTful Invocation through Custom WF Activities

A Custom Workflow Activity essentially is a software component to be integrated within a WF workflow, usually dealing with a particular issue not addressed by the standard workflow components, contained in the component library. The minimal steps required for implementation of such a component include:

1. Creation of a .NET Class Library- or alternatively a Sequential Workflow-Project serving as a container for custom activities.
2. Creation of a class derived from either `System.Workflow.ComponentModel.Activity` in case of a simple activity, or `System.Workflow.ComponentModel.CompositeActivity` for a complex activity encompassing several basic activities. The source code of the class is split into two files, following the partial class concept of C#[2]
3. Overriding the `Execute` method, containing the functionality to be carried out upon reaching the component during execution of a workflow. Through a comprehensive .NET library support for HTTP conversation and XML processing the implementation of a RESTful service invocation is comparatively easy.
4. After compilation of the project containing the activity within Visual Studio the activity becomes available in the workflow designer toolbox, to be dragged onto the designer surface analogous to other pre-defined components.

JSON Handling

For processing of JSON formatted invocation results *Json.NET* [62] by James Newton-King is used. The .NET library available through Microsoft's open source project site `CodePlex`[3] allows among others for serialization of .NET objects to and from JSON and as a vital feature to the thesis implementation conversion of JSON formatted data to and from XML. The subsequent listing depcits the necessary steps for run time conversion of a JSON result to an XML document:

[2]Refer to the C# Programming Guide, available at [20], for an explanation of partial classes in C#
[3]`http://www.codeplex.com/`

```
StreamReader reader =
  new StreamReader(response.GetResponseStream());
JsonTextReader jr = new JsonTextReader(reader);
JObject joResult = JObject.Load(jr);
JProperty jpRoot = new JProperty("root", joResult);
JObject joContainer = new JObject();
joContainer.Add(jpRoot);
XmlNode node =
  JavaScriptConvert.DeserializeXmlNode(joContainer.ToString());
```

Listing 5.1: Conversion of JSON to XML by means of the Json.NET api

The library elements forming the basis for the conversion are `JObject`, representing a JSON object (the top level container for JSON formatted data, enclosed by curly braces "{","}") and `JProperty` the container for a JSON key-value pair. Unlike a hierarchical organized XML document a typical JSON structure does not have a root element. Hence in order to convert JSON to XML such an element has to be created first. This is done in line 4, where a JProperty (jpRoot) is declared with a key name "root" and a value representing the actual JSON result. The property is added to a JObject instance (joContainer) that can then be converted to an according XML document by means of the `JavaScriptConvert` class.

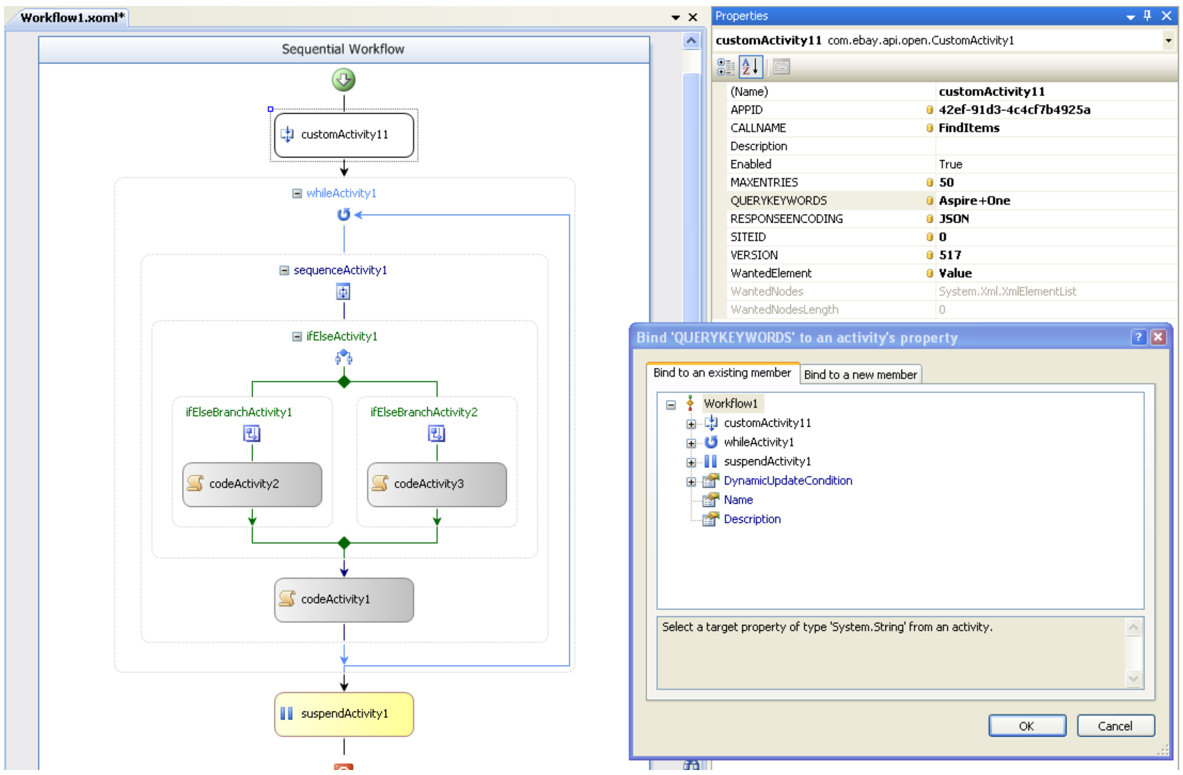
Dependency Properties

In order to make workflow design more palatable to non-programmers the parameters of a custom activity are provided through so called *Dependency Properties*. Different from a standard property, where the property value is stored within an instance variable of an object, the value of a Dependency Property is stored in a central repository maintained by the property system. As a consequence of adhering to the mechanism the property becomes available in the workflow design surface and workflow runtime.

Three types of Dependency Properties are differentiated[4]: *metadata* properties being immutable at workflow runtime, *attached* properties allowing for access from another activity within the workflow and *instance* properties. In the generated custom activities the latter type is used. Essentially it allows for *Activity Binding*, i.e. the binding of an activity property value to data not known until workflow runtime. This can either be the value of a property of another activity within the workflow, a field, a property or

[4] http://msdn.microsoft.com/en-us/library/ms734499.aspx

a method. For affiliation of multiple service invocation activities within a workflow the binding of one activity's output to an input property of another activity is perhaps the most important application of the concept.



customActivity11 com.ebay.api.open.CustomActivity1

(Name)	customActivity11
APPID	42ef-91d3-4c4cf7b4925a
CALLNAME	FindItems
Description	
Enabled	True
MAXENTRIES	50
QUERYKEYWORDS	Aspire+One
RESPONSEENCODING	JSON
SITEID	0
VERSION	517
WantedElement	Value
WantedNodes	System.Xml.XmlElementList
WantedNodesLength	0

Select a target property of type 'System.String' from an activity.

Figure 5.5.: Visual Studio Workflow Design Surface - Property Binding

A major benefit of using Dependency Properties in combination with the Visual Studio Workflow Designer is the built in support for checking type compatibilities of activity bindings at design time of a workflow. One drawback of this combination is the inability for dynamically referencing array elements using an index variable in a bind expression. For example an expression like XmlResultElements[i], where i represents a counter variable, is not valid. Hence the dynamic selection of a particular element from an XML node list returned by a service invocation is not possible in a convenient manner. To overcome this confinement a code expression for programmatic selection of a desired XML element would have to be inserted in the workflow.

The C# source code for a Dependency Property definition can be found in the example custom activity for the invocation of the *Google Translate* service in Appendix D.

5.4. Underlying Technology Stack

To complete the first part of the implementation description this section takes a glimpse at the underlying technology stack. For a description of *Rest Describe & Compile* omitted subsequently the reader may refer to the extensive project documentation [88, 87].

All functional components, i. e. the WADL processing and WF code generation classes, as well as sample workflow projects are implemented in C#. For XML parsing the `System.XML` namespace of the .NET Framework 3.5[19] class library is used instead of `System.XML.Linq`, due to its closer syntactical similarity to the Google Web Toolkit XML API[39].

The *Windows Workflow Foundation* [17], being in its most recent version a part of .NET Framework 3.5, is used as the underlying workflow framework. It provides an extendable programming model, which we considered as a major advantage over BPEL based open source workflow engines, being overly complex and not aiming at extension of its workflow component model. Furthermore due to its tight integration with Visual Studio 2008 including according workflow project templates and an integrated workflow engine rapid testing of the implementation was well supported. The testing of the generated Custom Activities was simplified as well through their immediate availability within the integrated visual workflow designer just after their compilation. However we also faced the boundaries of the closed source technology during investigation of possible extension points of the design surface [22], influencing the technological decisions for the Service Integration Wizard introduced in the following chapter.

6. Dynamic Service Integration in expressFlow

This chapter details the integration of a wizard based approach for dynamic service integration in the *expressFlow* visual workflow designer[1]. ExpressFlow is a web based workflow application, developed at the Distributed Systems Group of the Vienna University of Technology. It provides a visual design surface for the *Service Mashup Abstraction* introduced in [92] as well as comprehensive support for collaborative workflow design. Collaborative aspects are facilitated through the web based architecture as well as a tightly integrated user-/role model allowing for sharing of workflow information among registered users.

6.1. ExpressFlow Architecture

6.1.1. Application Overview

Figure 6.1 depicts the high level architecture of expressFlow. The application is essentially split up into two main components, a Flash client implemented with the Adobe Flex Framework [3] (refer to section 6.1.2 for the detailed architectural description) and an ASP.NET [53] server application. The client application is delivered as SWF file to the user workstation after successful authentication at the application website. It comprises the workflow design surface including the Service Wizard introduced in section 6.2 and the process- and role-administration interface. Due to large parts of the application, including resource intensive XML parsing components for WSDL and

[1]A prototype of the application is deployed at the project website, `http://expressflow.com`, and can be tested after signing up for a test account

WADL processing, being carried out at the client side performance and scalability is increased.

However as one of the aims of expressFlow is support for collaborative workflow design a server component for persistent storage and exchange of application data between clients is required. The ASP.NET web application, detailed in section 6.1.3, fulfills two roles: One is the login portal from which the Flash client is launched after successful user authentication. Particularly interesting is its second function, a WebORB.NET based proxy to the underlying data base and the WADL processing library, introduced in section 5.2.

Client-Server communication is carried out as asynchronous remote procedure calls by means of the Flex Remote Object API and *WebORB .NET* [57]. Transmitted data is encoded into the *Action Message Format, AMF 3* [1], a compact binary format allowing for serialization of ActionScript objects, first introduced by Adobe in Flash Player 6 in 2001.

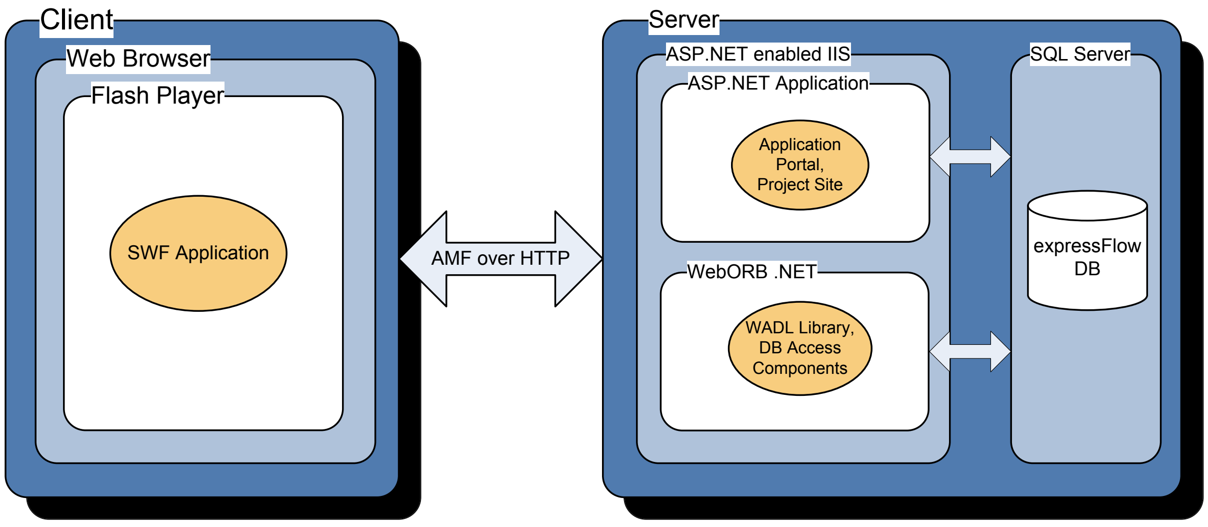


Figure 6.1.: expressFlow General Architecture

6.1.2. Flash Client - Internal Architecture

The internal architecture of the Flex application partly adheres to the principles of *Cairngorm* [96]. Cairngorm is an open-source architectural framework providing concepts for organizing and partitioning code and packages and assigning functionality

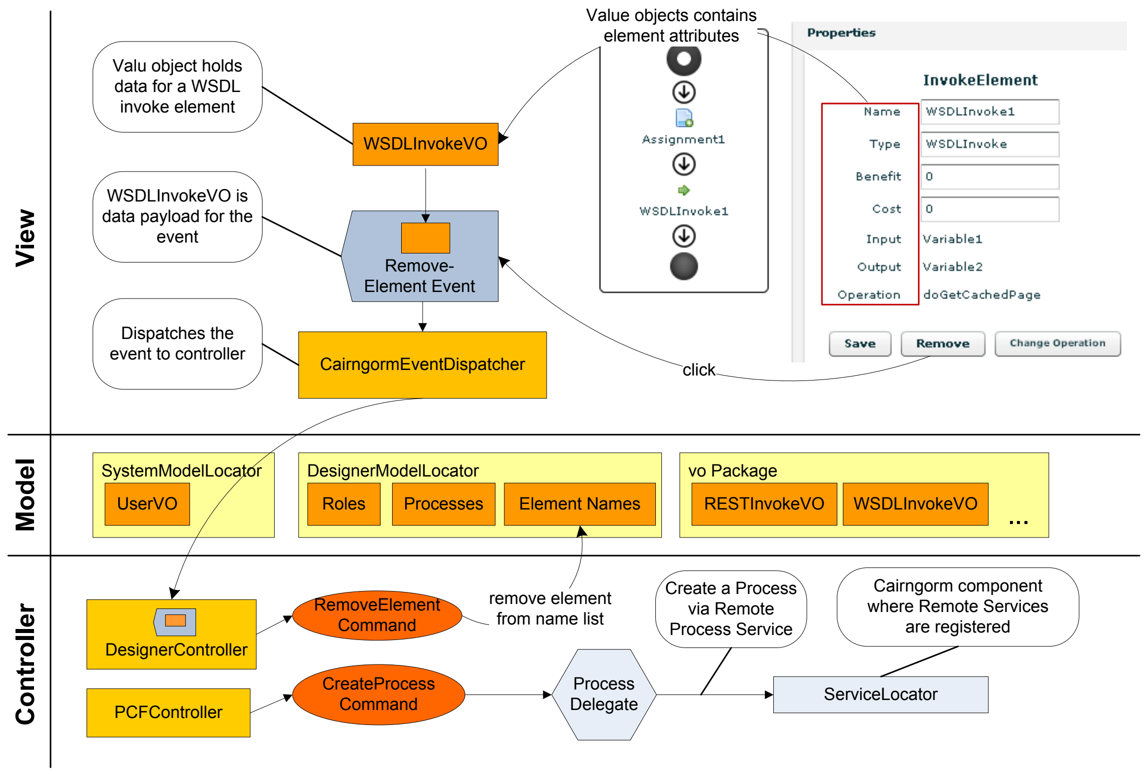


Figure 6.2.: expressFlow Internal Client Architecture

and roles to components of a Flex project. Basically a Model View Controller architecture is proposed aiming at a strict separation of concerns; a concept that is well fostered in the Flex framework through its technical separation of GUI logic being written in *MXML*[2] while business logic is written in *ActionScript*. Among the main aims of the framework are the enhancement of code maintainability and easier feature extendibility while allowing for parallel development of GUI and business logic.

Figure 6.2 depicts the implementation of the main Cairngorm concepts in expressFlow. The graphic illustrates how the data attributes of a workflow element (`InvokeElement`) initially retrieved from the server via de-serialization of the process XML representation are represented in a value object (`WSDLInvoke`) on the client side. A click on the `Remove` button dispatches an event (`RemoveElementEvent`) to the corresponding controller responsible for updating the model layer (`DesignerModelLocator`). Server communication is kept solely at the controller level, implemented as a chain between the Controller (`PCFontroller`), the desired command (`CreateProcessCom-`

[2]An XML language describing the GUI layout of a Flex application

mand), a delegate (ProcessDelegate) where the actual RPC call is issued and the registry of Remote Services (ServiceLocator). In order to gain a more complete picture of the client architecture the reader may refer to [92]. The paper provides a slightly different architectural view focusing on the class model and the design issues raised by the application architecture involving a highly decoupled RIA client and a server application enabling collaborative aspects. Table 6.1 summarizes the essential packages of the Flex application and their provided functionality.

Package	Provided Functionality
business	Includes delegate classes providing a local proxy to server functions.
commands	Provides non-GUI functions: Creation of a new process; Retrieval of user name and roles of a logged in user. Server functions are accessed through the delegate classes.
components.servicewizard	GUI logic for service integration wizard
elements	Implementation of workflow components available in the design surface toolbox. Each contained element provides (de)serialization methods from and to its XML representation (see section 6.2.5).
events	Definition of event classes derived from the CairngormEvent base class. Each event corresponds to a command.
interfaces	Interfaces implemented in workflow elements and structures.
util	Common helper classes. Among others deserialization of process XML representation. WSDL parsing used in the service wizard.
vo	Contains value object classes, i.e. container classes holding the data of application entities (user, process, workflow element, ...).

Table 6.1.: expressFlow - ActionScript Package Structure

6.1.3. Server Implementation

The server component of expressFlow serves as an application portal for the Flash client and enables persistent storage of workflow data created at the client. The user / role model enabling all collaborative aspects, such as participative workflow design, workflow change notifications and shared service descriptions (WSDL and WADL documents) is modelled in a relational data base[3]. The underlying SQL data model is mapped to implementation classes by means of *Linq to SQL*, an object relational mapping library introduced by Microsoft as part of the .NET Framework 3.5.

Client access to the data base server is accomplished by proxy classes to be invoked via RPC by means of WebORB.Net [57]. Furthermore the framework provides access to the following business components:

- A *WADL generation library*, previously introduced in chapter 5, required for integration of a RESTful service described by a request URI. The URI string is passed to the proxy returning an according WADL description that can be processed at the client. Generated WADL documents are stored in the data base for future use and can also be declared public in order to share them with other users.

- An *external data proxy* component allows to overcome the *same origin policy* limitation at the client. Basically the security mechanism, originally introduced in Netscape Navigator 2.0, prevents a script (such as ActionScript or JavaScript) executed at the client originating from one origin to load data from a different origin[4]. The concept has mainly emerged as a consequence of a security vulnerability commonly known as *cross site scripting*[5]. Loading external data at the client is required in order to gain information on the response data structure of a service request. Typically in a URI based service integration scenario this information is completely missing. However in order to improve service integration capabilities a means for visual selection of result elements is desirable. The client therefore issues a sample request to the service by means of the data proxy at workflow design time and presents the list of returned (XML) nodes to the user. A more detailed discussion of the mechanism is given in section 6.2.

[3]For easier integration with the .NET based server application Microsoft SQLServer is used as DBMS.

[4]See `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript` for an exemplified explanation

[5]`http://www.cert.org/advisories/CA-2000-02.html`

6.2. expressFlow Web Service Integration Wizard

In order to improve the service integration capabilities of expressFlow a wizard based client extension has been implemented as part of this thesis. The wizard replaces the workflow elements *RESTInvoke* and *WSDLInvoke* originally intended for integration of Web services in the visual designer. The motivation for developing a solution for integration of heterogeneous Web services mainly arised from the flaws of popular workflow design tools regarding this task. In the current version of the Visual Studio integrated Workflow Designer, providing a convenient design surface for WF workflows, the term REST[6] is omitted completely. Service invocations are solely restricted to WS* services, for which a wizard is provided.

Another motivating design tool was the BPEL designer integrated in the Netbeans IDE [78]. The designer fully implements the WS-BPEL standard and fosters rapid testing of designed processes through a tight integration with the Glassfish application server [15]. However the integration of RESTful Web services is not solved by the tool, probably also due to the lack of support for RESTful Web service invocation in the current version of WS-BPEL. Another flaw making the workflow design error prone and rather unpalatable for non experts is the strict mapping of the WS-BPEL standard in the design process. Consequently a user designing a process must be familiar with the standard elements in order to gain valid workflows. Adding for example a service invocation element to the workflow does not trigger the addition of corresponding *Assign* elements for input parameter binding. Hence the user must know the required elements from the standard and add them manually.

Resulting from the mentioned flaws in existing solutions the main aims for the service wizard implementation are as follows:

1. Allowing for integration of heterogeneous Web services, i. e. WS* based and RESTful Web services. RESTful Web services may be described by a WADL document or a request URI. A request URI is processed into a corresponding WADL document at workflow design time.
2. Abstraction from the technical interface characteristics in the designer surface. Only one *ServiceInvoke* element is available in the toolbox. The user is lead in a few steps through the integration process. Apart from a differentiation regarding

[6]The issue could possibly be addressed in the upcoming .NET version as [82] suggests.

the service description language further technical terms are kept transparent to the user.

3. Affiliation of WSDL and RESTful services through abstraction of input-/output-element syntax (see section 6.2.4).

4. Dynamic result element selection for RESTful services through sample requests at workflow design time. An element format description is not required in advance. This fosters a *light weight* integration approach where a service is solely described by its request URI at first.

5. Design of valid workflows - Automatically add *Assign Elements* for input parameter binding for a WSDL Invoke Element. Maintain default parameter values for REST URIs while allowing for binding parameters to workflow variables where necessary. Provide automatic deletion of automatically added Assign Elements upon changing the service operation / the request URI.

6. Reuse of existing implementations and seamless integration: Allowing for XML serialization of workflows containing a Service Invoke element in order to store a process at the expressFlow server; Extension of the Service Mashup Abstraction [92] through according Service Invoke Elements; Reuse of the .NET based WADL processing library in the Flex Client.

Figure 6.3 depicts the basic steps traversed using the wizard:

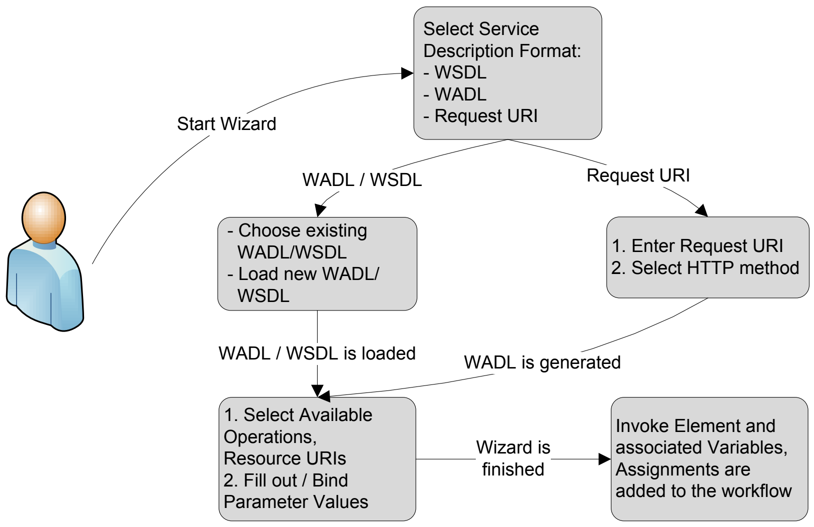


Figure 6.3.: expressFlow Service Integration Wizard - Application Flow

6.2.1. WADL based Service Integration

WADL based service integration provides adjacent to URI based integration a second means of integration RESTful services into expressFlow workflows. The idea is to combine the capabilities of WADL for detailed resource description including input parameters and output format, both of which are essential for tool supported workflow design, with the lightweight nature of RESTful Web services. Through the enumeration of request parameters and their according data types contained in the resource description of a WADL document the user can be presented with a parameter binding form at design time. Though not being integrated in the current version, static checking of type compatibilities yet at design time would certainly be a further benefit of the approach.

Furthermore describing RESTful services through a WADL document also provides a mechanism for building up a repository of service resources at the server. A user can reuse his WADL documents and publish them for reuse among other users. A means for editing existing WADL documents in order to include additional resource URIs or complement existing resource descriptions by a result format specification would be an interesting enhancement for future versions.

The encapsulation of a service specification within a WADL document also facilitates a compacter workflow XML representation. Within an Invoke Element for RESTful service invocation the corresponding WADL description can simply be referenced (see section 6.2.5), thus preserving all required service information while avoiding redundancy and overly complex workflow descriptions. Aspects of the concept can be compared to the *Partner Link* mechanism [76] of WS-BPEL, where a WSDL document containing the required operation descriptions of a service is referenced within a BPEL process description.

Subsequently the implementation aspects of WADL integration in the Flex application will be detailed.

Implementation

All GUI components related to WADL integration in the service wizard are part of the `components.servicewizard` package. For transitions between wizard steps the convenient *View States*[7] concept of Flex is used. It allows for declaration of states of a GUI component, such as 'Select a WADL file' followed by 'Select a Resource URI', in the MXML code. Used in combination with Adobe Flex Builder[8] different GUI states can be previewed in a design view during implementation. Existing WADL documents are presented in an `mx:DataGrid` element. The corresponding data is retrieved via asynchronous RPC by means of the WebApplication.WebORB.ProcessService class located at the server. External WADL documents may also be retrieved from *external web resources* using the server side WebApplication.WebORB.ExternalDataProxy to overcome the same origin policy limitation (see section 6.1.3). Furthermore files can be loaded from the *local file system* using the `FileReference` API class contained in ActionScript3. Each successfully loaded WADL document is uploaded to the server in order to be available for future workflow design.

All WADL XML parsing tasks are implemented by means of the ActionScript 3 `XML` class. The class implements the ECMAScript for XML (E4X) specification [23] and provides an efficient mechanism for processing a WADL document. Remarkable is the speed at which the XML structure is parsed. A wizard user is presented with the processing result without noticeable delay at the client.

The serialization from and to a workflow XML representation is implemented in the `RESTInvokeElement` class, all node values descend from its associated value object `RESTInvokeVO`. Again the task is alleviated by the E4X compliant ActionScript API, allowing for native declaration of XML nodes as the following code excerpt demonstrates:

```
var xmlData:XML = <Invoke name={content.name} type={content.type}
method={method}>
<WADL>{wadlSrc}</WADL>
</Invoke>;
```

Listing 6.1: Native declaration of a new XML Element in ActionScript 3

The data attributes of a WADL based Invoke Element are encapsulated in an according value object, represented by the `content` instance variable in listing 6.1. The value object implementation is also responsible for issuing a sample request at the given request URI. The returned XML payload is processed into an array of elements available for visual selection in the workflow designer. A major advantage of implementing

[7] `http://livedocs.adobe.com/flex/3/html/using_states_1.html`
[8] An Eclipse based IDE for Flex applications by Adobe

this task in ActionScript is the pass by reference behaviour[2] for non primitive types. Hence a request can be issued at instantiation time of a value object without blocking the design process while awaiting the response. Usually the response elements of a value object become available for parameter binding in a subsequent workflow element in time.

A final implementation aspect worth mentioning is the mechanism for determination of available output elements as contained in Service Invoke Elements within the actual Element's scope. The scope of a RESTInvoke element is processed bottom up starting at the actual element. All output elements found in the scope are stored in a drop down list that can be presented to the user for binding of output values from other elements to the input parameters of the current element. The according algorithm is sketched below in pseudo code:

```
function GETRESULTPARTSINSCOPE(actualElement)
    p = actualElement.parent
    result = new Array()
    for all p.children do
        if child != actualElement and child is ServiceInvokeElement then
            if child.valueObject.type == RESTInvokeVO then
                for all RESTResultElements in valueObject.resultParts do
                    add RESTResultElement to result
            if child.valueObject.type == WSDLInvokeVO then
                for all SchemaDisplayElements in valueObject.resultParts do
                    add SchemaDisplayElement to result
    return result.concat (parent.GetResultPartsInScope(acutalElement)
```

6.2.2. Request URI based Service Integration

Providing a means for service integration solely based on a given request URI is from our point of view the most vital feature for real world RESTful service integration. Though WADL or WSDL 2.0 based service descriptions theoretically provide a far more comprehensive specification of a service's characteristics, as discussed in chapter 4, their availability in real world service APIs is rather sparse. This may partly result from the fact that description languages for RESTful services are either not standardized (WADL) or poorly supported by according tools (WSDL 2.0). On the other hand REST advocates believe that having a description atop of a RESTful service contradicts the inherent light-weight nature of REST. Debates on the topic often seem to be ideologically grounded[9], however the lack of machine processable interface descriptions in real world REST APIs is a fact.

In the expressFlow service wizard implementation we propose a combination of the minimal service description represented through a request URI and the comprehensive specification capabilities provided by WADL. This is achieved by generating a WADL description from the provided URI string at design time. The WADL document is generated at the server by means of the WADL library introduced in chapter 5. Supplementary to the URI string only the HTTP method has to be specified as input to the generation process. On success the generated document is stored at the server, while at the client the user is presented with a form allowing for resource selection and parameter binding identical to step two of the WADL based service integration (see section 6.2.1).

The intermediate step for generation of a service description offers a range of advantages over direct service integration solely based on the request string:

- Generated service descriptions can be stored in a repository to be reused and shared among other users. Through storing WADL documents instead of URI strings service descriptions can subsequently be used in the WADL based integration approach previously described (6.2.1).
- WADL documents can be supplemented with additional service characteristics and/or corrected once stored at the server. Such additions could for example be a description of the response format for a request, further resource definitions or documentation properties for existing elements.
- A determination of the request parameters and their according data types forms the basis for a convenient parameter value binding mechanism at the client. The user is presented with a list of parameters that can be bound to literal values or workflow variables. The specification of data types contained in the WADL description can be utilized for static type compatibility checking.
- The WADL generation library issues a request to the provided URI in order to guess the associated response format (XML, JSON), an information that is most

[9]See the following resources for discussions on the exigence of description languages for REST: `http://www.infoq.com/news/2007/06/rest-description-language`, `http://www.artima.com/forums/flat.jsp?forum=276&thread=207471`

likely not contained in the request URI. Though the specification of the response format is not necessarily required at workflow design time, it has to be specified to the runtime environment of a workflow. A sample request could also be utilized for server side determination of the result elements of a request, currently implemented at the client.

Implementation

The implementation for request URI based service integration supplements the previously described WADL integration component by an additional input form allowing for input of the URI string and HTTP method. All steps following the retrieval of a generated service description are implemented as previously described in section 6.2.1.

6.2.3. WSDL based Service Integration

A third Web service description format supported in the expressFlow wizard is WSDL. Though not directly addressing the issues of RESTful service integration in workflows it entails some interesting challenges to the wizard implementation. As already mentioned in the introduction of this section a major flaw of many visual workflow design tools supporting WSDL based service integration is the requirement to be familiar with the standard terminology in order to achieve valid results. Among the usually required tasks succeeding the addition of a service by importing a WSDL description are:

- Creation of input-/output-variables associated with the added service invoke element.
- Selection of a desired service operation provided by the service description.
- Binding of operation input-/output-elements to literal-values/workflow-variables. In BPEL based workflows this step requires the addition of an *Assign* element first enclosing a set of *Copy* elements representing the actual variable assignment.

Given these tedious tasks a main aim of the wizard implementation is increasing the usability of workflow design by divesting the user of as many manual input steps as possible while still gaining valid workflow results. Furthermore through abstraction of interface description semantics for WADL and WSDL the implementation aims at providing a convenient integration of heterogeneous services within a workflow. Subsequently the main objectives and their according implementation issues will be detailed.

Keep Unneeded Service Characteristics Transparent

Displaying all elements of the complex description standard, such as binding information or message type definition according to the hierarchical structure of the WSDL document would very likely overstrain a user. Hence the wizard reduces the amount of information presented to the user to a minimum required during workflow design, i. e. the contained operations and their associated input parameters. Operations are presented in a selection list, the display of associated parameters is updated dynamically upon changing the operation selection. A challenging aspect during the implementation was the correct parsing of XML schema message type definitions. The parsing component implemented in ActionScript has been successfully tested with real world service descriptions taken from main stream APIs such as Google, Amazon or Ebay.

Automate Addition of Obligatory Associated Elements

In order to gain valid workflow design results the input parameters of a *WSDL Invoke Element* have to be bound to values preceding their execution within the workflow. In contrast to a *REST Invoke Element* where all parameter values are probably encoded in the initially provided request string already, for WSDL based services this step is always required. The wizard automates this task by adding an *Assignment Element* preceding the newly created Invoke Element automatically. It encloses a set of *Copy Elements* containing the user defined bindings in the final wizard step. A Copy Element is implemented in a self contained implementation class equivalently to other workflow elements and becomes associated to its enclosing Assignment by means of an array instance variable of an Assignment. This programmatic association is important for (de)serialization of the process description, as the Mashup Language does not permit a syntactical linkage of the elements. Furthermore a newly created WSDL Invoke Element is supplemented with according In-/Output *Variable* elements.

Simplify Parameter Binding

Parameter binding is the essential mechanism allowing for data transfer between different services of a workflow. The idea is to present the user with a drop down list populated with available output variable parts in the element scope (see section 6.2.1 for a description of the mechanism). In order to allow for binding to literal values the box is declared editable. Through integration of the binding step into the wizard the creation of valid Invoke Elements is fostered, superseding complicated subsequent binding steps.

Provide Editing Capabilities

The application permits editing of all created components. Copy Elements may be modified by editing their *From* and *To* attribute values directly, allowing for maximum flexibility. The editing of a Service Invoke element constitutes a more elaborate task, as its semantically associated Assignment elements are formally not linked to the Invoke element. In the Mashup Abstraction [92] such an association is not provided, in order to allow for the reuse of an Assignment in multiple Invoke Elements. Hence the wizard is not capable of restoring its state before creation of the Invoke element including the defined variable binding expressions being declared within an Assignment element.

Alternatively the application presents a user with the option to automatically delete previously created Assignment elements, as upon changing a WSDL operation or a REST resource URI the signature of a request may be altered and consequently the associated Assignment may become invalid. Once a user requests a change of an operation / a resource URI the wizard tries to determine the associated Assignment element based on three conditions:

1. The number of nested Copy Elements matches the number of input message elements.
2. The input variable name of the Copy Element matches the input variable name of the edited Invoke Element.
3. The input message part name matches the message part name component of the copy expression.

If a matching Assignment element is found and the user confirms its deletion it will be removed from the workflow immediately. The wizard is then launched with the previously selected service description and the operation/resource URI.

6.2.4. WADL / WSDL - Specification Mapping

In order to allow for the integration of WS* and RESTful services in an equal measure an abstraction of their characteristics as provided through their interface descriptions is defined. From a usability point of view the abstraction allows for keeping technical specifics of the different service paradigms transparent from the user during the design process. The required steps and their associated GUI representation are basically equivalent for both types of services.

Moreover the specification mapping fosters the syntactical linkage of heterogeneous service sources in a workflow description language and subsequently a workflow runtime environment. By mapping the request-/response-parameters defined in WADL to the input-/output message parts/elements of WSDL the syntax of a corresponding *Copy* Element for describing the data flow between services can be kept simple. A demanding aspect of mapping WSDL message parts arises from the flexibility of WSDL 1.1 regarding the definition of message structures[10]. The structure of a complex message can essentially be declared in three ways:

1. The message definition includes *one* message part, referencing a complex XML schema type via its `type` attribute. In the complex type several simple message elements (xsd:int, xsd:string,..) are declared, as the subsequent excerpt from the Amazon Item Search definition illustrates:

```
<message name="KeywordSearchRequest">
 <part name="KeywordSearchRequest" type="typens:KeywordType"/>
</message>
<xsd:complexType name="KeywordType">
 <xsd:all>
  <xsd:element name="keyword" type="xsd:string"/>
  <xsd:element name="page" type="xsd:string"/>
  <xsd:element name="mode" type="xsd:string"/>
   ...
```

Listing 6.2: Amazon Item Search - message definition

[10] http://www.w3.org/TR/wsdl#_messages

2. The elements of a message structure are declared subsequent to the message definition via *multiple* part definitions, each of which represents a simple XML schema type. The following example from a Google Search WSDL document depicts the mechanism:

```
<message name="doGoogleSearch">
  <part name="key"  type="xsd:string"/>
  <part name="q"  type="xsd:string"/>
  <part name="start"  type="xsd:int"/>
  ...
```

Listing 6.3: Google Search - Message Definition

3. In the message part declaration an XML schema element is referenced. Besides an additional required step for resolving the name of the type definition associated with the element the parsing implementation is equivalent to the aforementioned processing of complex type definitions.

In the WSDL description of the Google Search service the specification concepts are mixed up even within a single operation. However given the abstraction scheme the range of supported service descriptions mostly depends on the capabilities of the WSDL parsing component.

Figure 6.4 illustrates the mapping of the specification elements between WADL and WSDL 1.1. A WADL request definition is decomposed into its *Resource* (identified by a unique URI path) associated with an HTTP method and a list of query parameters. The WSDL equivalent is an *Operation*, declared within a *Port Type* definition. Its request parameters are either declared as *Message Parts* with an inline type definition or via a complex *Xml Schema Types* definition. The response declaration is analogous to the request specification.

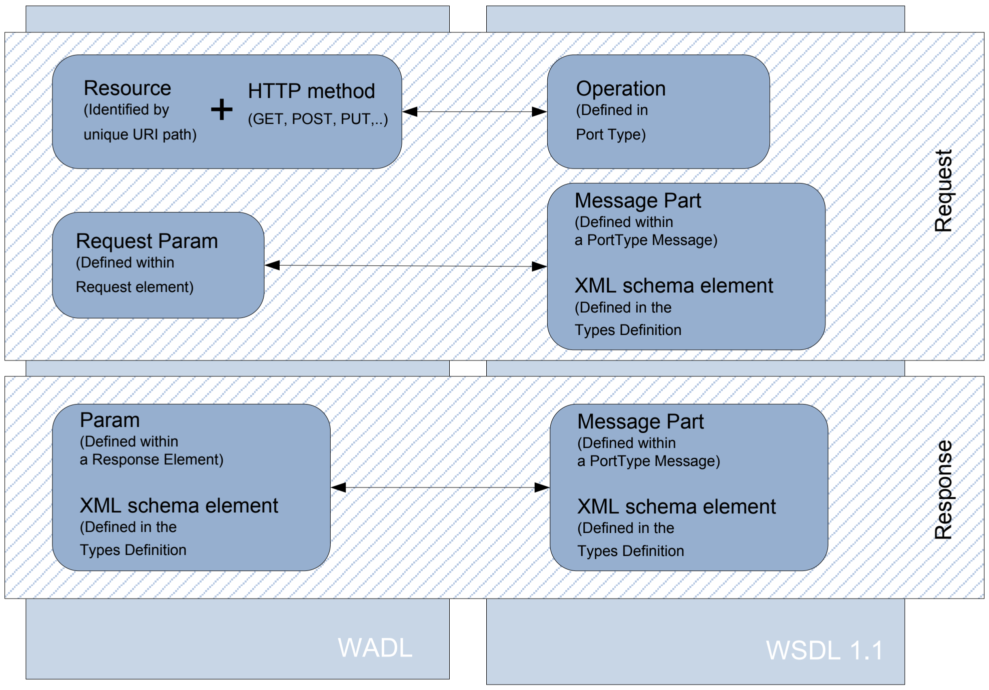


Figure 6.4.: WADL, WSDL 1.1 - Specification Mapping

6.2.5. Service Mashup Abstraction Extensions

Vasko et al. propose a *Service Mashup Abstraction* [92] providing the underlying abstract specification for XML serialization of a workflow created by means of the expressFlow designer. The abstraction is based on a meta-model identifying three main domains: Collaboration, Orchestration and Service Integration. A transformation scheme provides a rule set for transformation of a WS-BPEL based orchestration description into the Service Mashup Abstraction. Essential items of the scheme for the service wizard implementation are the notions of `Invoke`, `Assignment` and `Copy`. Subsequently the complements and modifications from the original specification in order to support the elements created by the service wizard will be detailed.

1. An *Invoke* Element declares the invocation of an external Web service within a workflow. Besides the standard activity attributes (name, type, benefit, cost) it is linked to the *Input-*/*Output*-Variable elements automatically created for each Invoke element. In order to express the service type (REST, WS*) the child el-

ements, *Resource* (RESTful invocation) and *WSDL* (WS* invocation) are defined supplementary to the original specification:

```
<Invoke standardAttributes input="Variable1" output="Variable2">
  [ <Resource uri={uri} wadlID={wadlId}/> |
    <WSDL id={wsdlId} operation="KeywordSearchRequest"/> ]
</Invoke>
```

2. A *Variable* Element is automatically created upon finishing the service wizard. It has a unique name and provides the type information for input-/output messages of a service:

```
<Variable name="Variable1" type={TypeDefinitionReference}/>
```

3. An *Assignment* Element, encloses a set of *Copy* Elements, used to assign literal- or variable-values to a variable. It is automatically created for each WSDL Invoke Element and for a RESTful Invoke Element with at least one defined parameter binding expression. Due to the encapsulation of multiple Copy Elements within one Assignment a more compact visual representation of a workflow is facilitated:

```
<Assignment standardAttributes>
  <Copy name="Copy" type="Activity"  copy_from="Mac Book"
    copy_to="$Variable1.keyword" previous="null" next="null"/>
  <Copy name="Copy" type="Activity" copy_from="price"
    copy_to="$Variable1.sort" previous="null" next="null"/>
</Assignment>
```

7. Evaluation and Future Work

7.1. Runtime Integration

The integration of RESTful service invocations into the *Windows Workflow Foundation* framework, through automatic generation of custom components, entails numerous advantages. From a user's point of view such a component encapsulates the functionality of a service and abstracts from the technical specifics required for valid service interaction. A custom service component can be provided in the palette of the workflow design application as an ordinary component equivalently to other workflow building blocks.

The publishing of component parameters through *Dependency Properties* (see section 5.3.1) fosters the service integration in two ways. On the one hand parameters can be bound dynamically by a user, as dependency properties are presented in a special toolbox integrated in the designer surface. On the other hand multiple services may be affiliated through binding their input-/output-parameter values at runtime.

Moreover the component integration approach meets with the requirements imposed by the large diversity of real world RESTful service APIs. As pointed out in section 2.2, the originally proposed architectural constraints [30] of REST are often neglected, leading to individual solutions awkward to integrate in workflow environments. Specifics of particular service APIs, such as a custom authentication mechanism, can be tackled through programmatic extension of created components. However as this step requires programming skills it needs to be carried out be an expert user in advance allowing for subsequent convenient use of the component in a visual design process.

WADL descriptions, generated by the WADL library component, form the basis for the subsequent generation of workflow components. The detailed WADL specification of a RESTful service, allows for the translation of complex REST interaction semantics, including for example requests with an XML payload in the message body into the C# source code specification of a custom workflow component.

7.1.1. Future Work

Automatically generated workflow components and WADL descriptions for real world service APIs have been successfully tested during the implementation phase. Resulting from these tests a number of requirements, to be addressed by future versions, have emerged:

- The result element structure of a service request, i. e. the concrete XML/JSON elements and optionally associated data type definitions, can be analyzed through a sample service request in the WADL generation phase. Currently only the result format, being either XML or JSON is automatically specified.
- A means for formal differentiation of *fixed* parameters (for example a method name) and *dynamic* parameters (for example a search string) in request definitions shall be provided. Fixed parameters could then be excluded from the binding mechanism in the workflow design surface.
- Further HTTP methods have to be supported in custom workflow components in addition to GET and POST.
- Basic support for standardized authentication mechanisms (for example HTTP basic authentication) shall be integrated into custom WF code components. As previously mentioned in this thesis (see section 4.1.2) WADL currently does not permit the specification of authentication elements, due to a high diversity of concepts implemented in real world scenarios. The future integration of Web service authentication in WF components could be achieved either based on the facilities provided in future WADL versions or by means of ancillary input parameters for the code generation process (`authentication_method`, `user_name`, `password`).

7.2. Visual Workflow Design

The service integration wizard for expressFlow, introduced in section 6.2, proves to be a viable solution to the issues of dynamic integration of heterogeneous services in visual workflow design. By hiding as many technical details as possible from the user, the entry barrier to workflow design is lowered compared to other approaches that require a detailed understanding of the involved technology standards (for example WS-BPEL, WSDL).

Due to allowing addition of formally described services, possibly required for more complex service invocations, as well as ad hoc style integration by simply providing a request URI, the initial *flexibility* of the design process is increased.

Service descriptions may be loaded from external or local resources and are stored in a repository located at the server. Hence such descriptions can be shared among other users, fostering the collaborative aspects of workflow design.

By encapsulating the interface specifications of integrated services in according service descriptions that are only referenced within a process definition, a clear separation of concerns is accomplished. This could be utilized in the transformation of expressFlow workflow definitions into another workflow format, as for example WS-BPEL, or a Windows Workflow Foundation process.

7.2.1. Future Work

By shifting many of the tedious tasks required for heterogeneous service integration in a workflow from the user into the implementation many issues of the task have already been solved. However some features stated in the original vision and emerged during the implementation evaluation remain to be solved in future versions:

- The type definitions provided by WSDL and WADL service descriptions may be utilized for checking static type compatibilities of bound parameters yet at design time. A mapping of compatible types and a workflow validation step are required for this purpose.
- Editing facilities for submitted/generated WADL/WSDL documents may be added, in order to enhance, adopt or view interface descriptions located at the server.

- The implementation of XML schema type parsing for WSDL documents is not fully implemented in the current version. Though the parsing component has been successfully tested with real world WSDL documents taken from Amazon, Ebay and Google APIs, it deserves some further work.
- Future versions shall allow for integration of WS* and RESTful services based on a WSDL 2.0 service description.

7.3. Feature Comparison

Table 7.1 depicts a feature comparison between the thesis implementation and related projects, introduced in chapter 3. The projects included in the comparison - *JOpera* [74] and *Apache ODE* [33] - were chosen due to their functional overlapping with the thesis implementation.

Feature	**expressFlow**	**WF**	**JOpera**	**ODE**
Description Format:				
WSDL 1.1	✓	✓	✓	✓
WSDL 2.0	✗	✗	✗	✓
WADL	✓	✗	✗	✗
URI	✓	✗	✓	✗
Runtime Invocation:				
WS*	✗	✓	✓	✓
REST	✗	(✓)	✓	✓
Design Process:				
Single User	✓	✓	✓	✗
Collaborative	✓	✗	✗	✗
Other:				
Web Scale Design	✓	✗	✗	✗

Table 7.1.: Implementation Feature Comparison

As the comparison reveals WSDL 1.1 is supported by all projects, while WSDL 2.0- and WADL-based service integration are only facilitated by the expressFlow service wizard (WADL) and Apache ODE (WSDL 2.0). URI-based service integration is confined to the tools with a graphical design surface including REST support (expressFlow, JOpera), as this integration type requires manual configuration steps opposed to WSDL/WADL based integration.

ExpressFlow falls behind the compared projects in terms of runtime service invocation, as a run time environment for designed workflows is currently not integrated. RESTful service invocation in the Windows Workflow Foundation (WF) is supported indirectly through import of custom activity components (see section 5.3.1).

Collaborative and web scale workflow design are unique features of expressFlow, and are not considered in the implementation architectures of the compared projects.

A. Bibsonomy WADL Description

```
<?xml version="1.0" encoding="utf-8"?>
<application
xmlns="http://research.sun.com/wadl/2006/10"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:bib="http://bibsonomy.org"
xsi:schemaLocation="http://research.sun.com/wadl/2006/10 wadl.xsd">

  <grammars>
    <include href="http://www.bibsonomy.org/help/doc/xmlschema.xsd"/>
  </grammars>

  <resources base="http://www.bibsonomy.org/">

    <resource path="api">
      <resource path="posts">

        <!-- Get Posts Request-->
        <method name="GET">
          <request>
            <param name="user" type="xsd:string" style="query"/>
            <param name="tags" type="xsd:string" style="query"/>
            <param name="resourcetype" type="xsd:string"
            style="query"/>
          </request>
          <!--Get Posts Response-->
          <response>
            <representation mediaType="application/xml"
            element="bibsonomy">
              <param name="postdate" style="plain"
              path="/bibsonomy/posts/post/@postingdate"/>
              <param name="user" style="plain"
              path="/bibsonomy/posts/post/user"/>
              <param name="bookmarkurl" style="plain"
              path="/bibsonomy/posts/post/bookmark/@url"/>
            </representation>
            <fault status="400" mediaType="application/xml"
            element="bibsonomy">
              <param name="statusName" style="plain"
              path="/bibsonomy/@stat"/>
              <param name="error" style="plain"
              path="/bibsonomy/error"/>
            </fault>
            <fault status="401" mediaType="application/xml"
            element="bibsonomy">
              <param name="statusName" style="plain"
              path="/bibsonomy/@stat"/>
              <param name="error" style="plain"
              path="/bibsonomy/error"/>
            </fault>
          </response>
        </method>
      </resource>

      <resource path="users/{username}/posts">

        <!--Create Post Request-->
        <method name="POST">
          <request>
            <representation mediaType="application/xml"
            element="bib:post"/>
          </request>
          <!--Create Post Response-->
          <response>
            <representation mediaType="appication/xml"
              status="201"/>
            <fault status="400" mediaType="application/xml"
            element="bibsonomy">
              <param name="statusName" style="plain"
              path="/bibsonomy/@stat"/>
              <param name="error" style="plain"
              path="/bibsonomy/error"/>
            </fault>
            <fault status="401" mediaType="application/xml"
            element="bibsonomy">
              <param name="statusName" style="plain"
              path="/bibsonomy/@stat"/>
              <param name="error" style="plain"
              path="/bibsonomy/error"/>
            </fault>
          </response>
        </method>
      </resource>
    </resource>
```

```
  </resources>
</application>
```

Listing A.1: WADL description of Bibsonomy Get All Posts and Create Post Operations

B. Bibsonomy WSDL 1.1 Description

```
<?xml version="1.0" ?>
<definitions targetNamespace="http://example.org/Bibsonomy/WSDL1.1"
                 xmlns:tns="http://example.org/Bibsonomy/WSDL1.1"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
                 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
                 xmlns="http://schemas.xmlsoap.org/wsdl/">

  <!-- message types - import bibsonomy schema -->
  <types>
    <xs:schema targetNamespace="http://example.org/Bibsonomy/WSDL2.0">
      <xs:include schemaLocation="http://www.bibsonomy.org/help/doc/
      xmlschema.xsd"/>
    </xs:schema>
  </types>

  <!--input- / output-messages -->
  <message name="GetAllPostsRequest">
    <part name="user" type="xsd:string"/>
    <part name="tags" type="xsd:string"/>
    <part name="resourcetype" type="xsd:string"/>
  </message>

  <message name="GetAllPostsResponse">
    <part name="posts" type="tns:posts"/>
  </message>

  <message name="CreatePostRequest">
    <part name="post" type="tns:post"/>
  </message>

  <message name="CreatePostResponse">
    <part name="response" type="xsd:string"/>
  </message>

  <!-- port types for Get Posts and Create Post -->
```

```
<portType name="PostsPort">
  <operation name="GetAllPosts">
    <input message="tns:GetAllPostsRequest"/>
    <output message="tns:GetAllPostsResponse"/>
  </operation>
  <operation name="CreatePost">
    <input message="tns:CreatePostRequest"/>
    <output message="tns:CreatePostResponse"/>
  </operation>
</portType>

<!--service aggregates bindings and defines an address endpoint-->
<service name="BibsonomyPostsService">
  <port name="BibsonomyGetPostsPort"
  binding="tns:BibsonomyGetPostsBinding">
    <http:address location="http://bibsonomy.org"/>
  </port>
  <port name="BibsonomyCreatePostPort"
  binding="tns:BibsonomyCreatePostBinding">
    <http:address location="http://bibsonomy.org"/>
  </port>
</service>

<!-- bindings define HTTP method and request/response format -->
<binding name="BibsonomyGetPostsBinding" type="tns:PostsPort">
  <http:binding verb="GET"/>
  <operation name="GetAllPosts">
    <http:operation location="posts"/>
    <input>
      <http:urlEncoded/>
    </input>
    <output>
      <mime:content type="text/xml"/>
    </output>
  </operation>
</binding>

<binding name="BibsonomyCreatePostBinding" type="tns:PostsPort">
  <http:binding verb="POST"/>
  <operation name="CreatePost">
    <http:operation location="api/users/johnsmith/posts"/>
    <input>
      <mime:content type="text/xml"/>
    </input>
    <output>
      <mime:content type="text/xml"/>
```

```
         </output>
      </operation>
   </binding>
</definitions>
```

Listing B.1: WSDL 1.1 Description of Bibsonomy Get All Posts and Create Post Operations

C. Bibsonomy WSDL 2.0 Description

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/ns/wsdl"
  targetNamespace="http://example.org/Bibsonomy/WSDL2.0"
  xmlns:tns="http://example.org/Bibsonomy/WSDL2.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdlx="http://www.w3.org/ns/wsdl-extensions">

  <types>
    <xs:schema targetNamespace="http://example.org/Bibsonomy/WSDL2.0">
      <xs:include schemaLocation="http://www.bibsonomy.org/help/doc/
      xmlschema.xsd"/>

      <xs:element name="userPosts" type="tGetPostsIn"/>

      <xs:complexType name="tGetPostsIn">
        <xs:sequence>
          <xs:element name="user" type="xsd:string"/>
          <xs:element name="tags" type="xsd:string"/>
          <xs:element name="resourcetype" type="xsd:string"/>
        </xs:sequence>
      </xs:complexType>

      <xs:element name="newPost" type="tNewPost"/>

      <xs:complexType name="tNewPost">
        <xs:sequence>
          <xs:element name="user" type="xsd:string"/>
          <xs:element name="post" type="tns:post"/>
        </xs:sequence>
      </xs:complexType>

    </xs:schema>
  </types>

  <interface name="PostsInterface">
    <fault name="unauthorizedRequest"/>
    <fault name="badRequest"/>

    <operation name="GetAllPosts"
      pattern="http://www.w3.org/ns/wsdl/in-out"
      style="http://www.w3.org/ns/wsdl/style/iri"
      wsdlx:safe="true">

      <input messageLabel="in"
        element="tns:userPosts"/>
      <output messageLabel="out"
        element="tns:posts"/>
      <outfault ref="tns:badRequest" messageLabel="out"/>
      <outfault ref="tns:unauthorizedRequest" messageLabel="out"/>

    </operation>

    <operation name="CreatePost"
      pattern="http://www.w3.org/ns/wsdl/in-out"
      style="http://www.w3.org/ns/wsdl/style/multipart">

      <input messageLabel="in"
        element="tns:newPost"/>
      <output messageLabel="out"
        element="#any"/>
      <outfault ref="tns:badRequest" messageLabel="out"/>
      <outfault ref="tns:unauthorizedRequest" messageLabel="out"/>

    </operation>

  </interface>

  <binding name="PostsBinding"
    type="http://www.w3.org/ns/wsdl/http"
    interface="tns:PostsInterface"
    whttp:methodDefault="GET"
    whttp:queryParameterSeparatorDefault="&">

    <fault ref="tns:badRequest" whttp:code="400"/>
    <fault ref="tns:unauthorizedRequest" whttp:code="401"/>

    <operation ref="tns:GetAllPosts"
      whttp:inputSerialization="application/x-www-form-urlencoded"
      whttp:location="/api/posts/{userPosts}"/>

    <operation ref="tns:CreatePost"
```

```
      whttp:inputSerialization="multipart/form-data"
      whttp:method="POST" whttp:location="/api/users/{user/}/posts"/>

  </binding>

  <service name="PostsService"
    interface="tns:PostsInterface">

    <endpoint name="PostsEndpoint"
      binding="tns:PostsBinding"
      address="http://www.bibsonomy.org"
      whttp:authenticationScheme="basic"
      whttp:authenticationRealm="BibSonomyWebService"/>

  </service>

</description>
```

Listing C.1: WSDL 2.0 Description of Bibsonomy Get All Posts and Create Post Operations

D. Custom Workflow Activity for Google Translate

```
namespace com.googleapis.ajax
{
  using System;
  using System.Workflow;
  using System.Workflow.ComponentModel;
  using System.IO;
  using System.Net;
  using System.Text;
  using System.Workflow.Activities;
  using System.Xml;
  using System.Web;
  using Newtonsoft.Json;
  using Newtonsoft.Json.Converters;
  using Newtonsoft.Json.Linq;

  public partial class CustomActivity1 : Activity
  {
    private static XmlDocument xmlDoc = new XmlDocument();

    public static DependencyProperty WantedElementProperty =
    DependencyProperty.Register("WantedElement", typeof(String),
      typeof(CustomActivity1));

    public static DependencyProperty QProperty =
    DependencyProperty.Register("Q", typeof(String),
      typeof(CustomActivity1),
      new PropertyMetadata("maus"));

    public CustomActivity1()
    {
      this.InitializeComponent();
    }

    public static XmlDocument XmlDoc
    {
      get
      {
        return xmlDoc;
      }
    }

    [System.ComponentModel.DescriptionAttribute("WantedElement")]
    [System.ComponentModel.CategoryAttribute("WantedElement Category")]
    [System.ComponentModel.BrowsableAttribute(true)]
    [System.ComponentModel.DesignerSerializationVisibilityAttribute(
        System.ComponentModel.DesignerSerializationVisibility.Visible)]
    public virtual string WantedElement
    {
      get
      {
      return ((String)(base.GetValue(CustomActivity1.
      WantedElementProperty)));
      }
      set
      {
      base.SetValue(CustomActivity1.WantedElementProperty, value);
      }
    }

    public virtual XmlNodeList WantedNodes
    {
      get
      {
      return xmlDoc.GetElementsByTagName(this.WantedElement);
      }
    }

    public virtual int WantedNodesLength
    {
      get
      {
      return this.WantedNodes.Count;
      }
    }

    //for brevity the dependency property definitions
    //for the query parameters V, Q and LANGPAIR are
    //omitted. their definition is analogous to the
    //subsequent definition of the Q parameter

    [System.ComponentModel.DescriptionAttribute("Q")]
    [System.ComponentModel.CategoryAttribute("Q Category")]
```

```
    [System.ComponentModel.BrowsableAttribute(true)]
    [System.ComponentModel.DesignerSerializationVisibilityAttribute(
        System.ComponentModel.DesignerSerializationVisibility.
        Visible)]
    public virtual String Q
    {
      get
      {
        return ((String)(base.GetValue(CustomActivity1.QProperty)));
      }
      set
      {
        base.SetValue(CustomActivity1.QProperty, value);
      }
    }

    protected override ActivityExecutionStatus Execute(
      ActivityExecutionContext context)
    {
      HttpWebRequest request = ((HttpWebRequest)(WebRequest.Create(
        "http://ajax.googleapis.com/ajax/services/
        language/translate?v="+HttpUtility.UrlEncode(
        Convert.ToString(V),new UTF8Encoding())+"&q="+
        HttpUtility.UrlEncode(Convert.ToString(Q),new
        UTF8Encoding())+"&langpair="+HttpUtility.UrlEncode(
          Convert.ToString(LANGPAIR),new UTF8Encoding())+"&")));
      HttpWebResponse response = ((HttpWebResponse)(
      request.GetResponse()));
      StreamReader reader = new StreamReader(response.
      GetResponseStream());
      JsonTextReader jr = new JsonTextReader(reader);
      JObject jo = JObject.Load(jr);
      JProperty jp = new JProperty("root", jo);
      JObject jo1 = new JObject();
      jo1.Add(jp);
      XmlNode node = JavaScriptConvert.DeserializeXmlNode(
      jo1.ToString());
      xmlDoc.LoadXml(node.OuterXml);
      return ActivityExecutionStatus.Closed;
    }
  }
}
```

Listing D.1: Custom Activity for Invocation of Google Translate

E. ExpressFlow Sample Workflow

```
<Process efid="129a8ff0-d5eb-49d8-bb10-eee036f54b33"
name="amazon_ebay" type="Activity" benefit="0" cost="0"
creator="alex">
  <Variables>
    <Variable name="Var1" type="var1Type">
    <Variable name="Var2" type="var2Type"/>
    <Variable name="Var3" type="var3Type"/>
    <Variable name="Var4" type="var4Type"/>
  </Variables>
  <Assignment name="Assignment1" type="Activity">
    <Copy name="Copy" type="Activity" benefit="0" cost="0"
    copy_from="Mac Book" copy_to="$Var1.keyword"/>
    <Copy name="Copy" type="Activity" copy_from="12"
    copy_to="$Var1.page"/>
    <Copy name="Copy" type="Activity" copy_from="mixed"
    copy_to="$Var1.mode"/>
    <Copy name="Copy" type="Activity" copy_from="no"
    copy_to="$Var1.tag"/>
    <Copy name="Copy" type="Activity" copy_from="search"
    copy_to="$Var1.type"/>
    <Copy name="Copy" type="Activity" copy_from="what"
    copy_to="$Var1.devtag"/>
    <Copy name="Copy" type="Activity" copy_from="price"
    copy_to="$Var1.sort"/>
    <Copy name="Copy" type="Activity" copy_from="no"
    copy_to="$Var1.variations"/>
    <Copy name="Copy" type="Activity" copy_from="de"
    copy_to="$Var1.locale"/>
  </Assignment>
  <Invoke name="WSDLInvoke1" type="WSDLInvoke" benefit="0" cost="0"
  input="Var1" output="Var2">
    <WSDL id="369596e1-101e-4dea-bb45-dac2562d212d"
    operation="KeywordSearchRequest"/>
  </Invoke>
  <While name="While1" type="Activity" benefit="0" cost="0"
  previous="WSDLInvoke1" next="null" expression="i">
  <Var2.output.length" role="null" user="null">
    <Assignment name="Assignment" type="Activity">
```

```
      <Copy name="Copy" type="Activity" benefit="0" cost="0"
      copy_from="$Var2.ListName[i]" copy_to="$Var3.QueryKeywords"
      previous="null" next="null"/>
    </Assignment>
    <Invoke name="RESTInvoke1" type="RESTInvoke"
    input="Var3" output="Var4">
      <Resource uri="http://open.api.ebay.com/shopping?
      callname=FindItems&responseencoding=XML
      &appid=<appid>&siteid=0&version=517
      &QueryKeywords={QueryKeywords}&MaxEntries=50"
      wadlID="15910a54e48"/>
    </Invoke>
  </While>
</Process>
```

Listing E.1: XML Listing of an expressFlow designed workflow containing two types of service invoke elements

F. Bibliography

[1] Adobe. Action Message Format - AMF 3, Specification. http://download.macromedia.com/pub/labs/amf/amf3_spec_121207.pdf, 2006.

[2] Adobe. ActionScript 3, Language Specification. http://livedocs.adobe.com/flex/201/html/03_Language_and_Syntax_160_19.html, 2007.

[3] Adobe. Adobe Flex 3. http://www.adobe.com/de/products/flex/, 2009.

[4] Djamal Benslimane, Schahram Dustdar, and Amit Sheth. Service Mashups, The New Generation of Web Applications. IEEE Internet Computing, 2008.

[5] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform Resource Identifiers (URI): Generic Syntax. http://www.ietf.org/rfc/rfc2396.txt, 1998.

[6] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk Nielsen. Hypertext Transfer Protocol - HTTP/1.0. http://www.ietf.org/rfc/rfc1945.txt, 1996. last accessed April 2009.

[7] Tim Berners-Lee, Larry Masinter, and Mark McCahill. Uniform Resource Locators (URL). http://www.w3.org/Addressing/rfc1738.txt, 1994. last accessed April 2009.

[8] David Booth and Canyang Kevin Liu. Web Services Description Language (WSDL) Version 2.0, Part 0: Primer. http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/, 2007.

[9] Tim Bray. SMEX-D Description of Amazon ItemSearch. http://www.tbray.org/ongoing/When/200x/2005/05/03/Amazon.smex, 2005. last accessed, February 2009.

[10] Tim Bray. SMEX-D, Simple Message Exchange Descriptor. http://www.tbray.org/ongoing/When/200x/2005/05/03/SMEX-D, 2005. last accessed, February 2009.

[11] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. AntiPatterns - Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, Inc., 1998.

[12] Michele Leroux Bustamante. Making Sense of all these Crazy Web Service Standards. Information Queue, 2007.

[13] Roberto Chinnici, Hugo Haas, Amelia A. Lewis, Jean-Jacques Moreau, David Orchard, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0, Part 2: Adjuncts. http://www.w3.org/TR/2007/REC-wsdl20-adjuncts-20070626/, 2007.

[14] Michael Chui, Andy Miller, and Roger P. Roberts. Six ways to make Web 2.0 work. McKinsey Consulting, 2009.

[15] Java.Net Developer Community. GlassFish - Open Source Application Server. https://glassfish.dev.java.net/, 2009.

[16] World Wide Web Consortium. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001. last accessed, January 2009.

[17] Microsoft Corporation. Windows Workflow Foundation Overview. http://msdn.microsoft.com/en-us/library/ms734631.aspx, 2007. last accessed, February 2009.

[18] Microsoft Corporation. Dynamic Source Code Generation and Compilation. .NET Framework Developer's Guide, 2008.

[19] Microsoft Corporation. .NET Framework 3.5. http://msdn.microsoft.com/en-us/library/w0x726c2.aspx, 2008. last accessed, February 2009.

[20] Microsoft Corporation. C# Programming Guide, Partial Class Definitions. http://msdn.microsoft.com/en-us/library/wa80x488(VS.80).aspx, 2009. last accessed, February 2009.

[21] Douglas Crockford. JavaScript Object Notation (JSON). http://tools.ietf.org/html/rfc4627, 2006. last accessed April 2009.

[22] Vihang Dalal. Windows Workflow Foundation: Everything About Re-Hosting the Workflow Designer. MSDN Library, 2006.

[23] ECMA. Standard ECMA-357, ECMAScript for XML (E4X) Specification. http://www.ecma-international.org/publications/standards/Ecma-357.htm, 2005.

[24] Hazem Elmeleegy, Anca Ivan, Rama Akkiraju, and Richard Goodwin. MashupAdvisor: A Recommendation Tool for Mashup Development. IEEE International Conference on Web Services, Beijing, China, 2008.

[25] Thomas Erl. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, 2005.

[26] Thomas Erl. SOA Specifications. http://www.soaspecs.com/, 2009.

[27] Roy Fielding et al. HTTP 1.1 Method Definitions. http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html, 1999.

[28] Thomas Steiner et al. Rest Describe & Compile, Project Website. http://code.google.com/p/rest-api-code-gen/, 2009. last accessed, February 2009.

[29] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html, 1999. last accessed April 2009.

[30] Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD Thesis, University of California, Irvine, CA, 2000.

[31] Jon Flanders. An Introduction To RESTful Services With WCF. MSDN Magazine, 2009.

[32] Apache Foundation. Apache Axis 2. http://ws.apache.org/axis2/. last accessed April 2009.

[33] Apache Foundation. Apache ODE (Orchestration Director Engine). http://ode.apache.org. last accessed April 2009.

[34] Apache Foundation. Apache ServiceMix. http://servicemix.apache.org/home.html. last accessed April 2009.

[35] Apache Foundation. RESTful BPEL in Apache ODE. http://ode.apache.org/restful-bpel-part-i.html,http://ode.apache.org/restful-bpel-part-ii.html, 2009. last accessed, March 2009.

[36] John Franks, Phillip Hallam-Baker, Jeff Hostetler, Scot Lawrence, Paul Leach, Ari Luotonen, and Lawrence Stewart. HTTP Authentication: Basic and Digest Access Authentication. IETF RFC 2617, 1999.

[37] Karthik Gomadam, Amit P. Sheth, Jacek Kopecký, and Tomas Vitvar. hRESTS: HTML Microformat for Describing RESTful Web Services and APIs. Technical Report, Wright State University, 2008.

[38] Google. Google Mashup Editor. http://code.google.com/intl/de-DE/gme/. last accessed April 2009.

[39] Google. Google Web Toolkit API Reference. http://google-web-toolkit.googlecode.com/svn/javadoc/1.5/index.html, 2009.

[40] Marc Hadley. Introducing WADL. http://weblogs.java.net/blog/mhadley/archive/2005/05/introducing_wad.html, 2005. last accessed, January 2009.

[41] Marc Hadley. WADLing in Jersey. http://weblogs.java.net/blog/mhadley/archive/2007/12/wadling_in_jers.html, 2007. last accessed, January 2009.

[42] Marc Hadley. Web Application Specification Language (WADL), Specification and Tools. https://wadl.dev.java.net/, 2009. last accessed, January 2009.

[43] IBM. IBM WebSphere sMash. http://www-01.ibm.com/software/webservers/smash/. last accessed April 2009.

[44] IBM. Lotus Mashups. http://www-01.ibm.com/software/lotus/products/mashups/. last accessed April 2009.

[45] Intel. Intel Mash Maker. http://mashmaker.intel.com/web/. last accessed April 2009.

[46] Ian Jacobs and Norman Walsh. Architecture of the World Wide Web, Volume One. W3C Recommendation, 2004.

[47] Marc Hadley Joe Gregorio and mark Nottingham. URI Template Internet Draft. http://bitworking.org/projects/URI-Templates/draft-gregorio-uritemplate-00.html, 2006. last accessed, January 2009.

[48] Mozilla Labs. Mozilla Ubiquity. http://labs.mozilla.com/2008/08/introducing-ubiquity/. last accessed April 2009.

[49] MSDN Library. Web Service Endpoints Based on WSDL Files. http://msdn.microsoft.com/en-us/library/bb385701.aspx, 2008. last accessed, April 2009.

[50] MSDN Library. Web Services Specifications. http://msdn.microsoft.com/en-us/webservices/aa740689.aspx, 2009. last accessed April 2009.

[51] Microsoft. Microsoft PopFly. http://www.popfly.com/. last accessed April 2009.

[52] Microsoft. Windows Workflow Foundation (WF). http://msdn.microsoft.com/en-us/library/ms735967.aspx. last accessed, February 2009.

[53] Microsoft. ASP.Net. http://www.asp.net/, 2009.

[54] Sun Microsystems. Jersey and WADL. http://wikis.sun.com/display/Jersey/WADL, 2008. last accessed, January 2009.

[55] Sun Microsystems. Jersey. https://jersey.dev.java.net/, 2009. last accessed, January 2009.

[56] Sun Microsystems. JSR311. https://jsr311.dev.java.net/, 2009. last accessed, January 2009.

[57] Midnightcoders. WebORB for .NET. http://www.themidnightcoders.com/products/weborb-for-net/overview.html, 2009.

[58] John Musser, Andres Ferrate, Raymond Yee, and Kevin Farnham. Programmable Web - Mashups, APIs, and the Web as Platform. http://www.programmableweb.com/. last accessed April 2009.

[59] Yefim V. Natis. Service-Oriented Architecture Scenario. Gartner Consulting, 2003.

[60] YefimV. Natis and Roy W. Schulte. Introduction to Service-Oriented Architecture. Gartner Consulting, 2003.

[61] Eric Newcomer and Greg Lomow. Understanding SOA with Web Services. Addison Wesley Professional, 2004.

[62] James Newton-King. Json.NET. http://www.codeplex.com/Json, 2008.

[63] David Orchard. List of REST Description Languages. http://www.pacificspirit.com/Authoring/REST/, 2005.

[64] David Orchard. Web Description Language, WDL. http://www.pacificspirit.com/Authoring/WDL/, 2005. last accessed, February 2009.

[65] David B. Orchard. List of REST Description Languages. http://www.pacificspirit.com/Authoring/REST/, 2005. last accessed, January 2009.

[66] OASIS Organization for the Advancement of Structured Information Standards. UDDI Version 3.0.2. http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm, 2004. last accessed April 2009.

[67] OASIS Organization for the Advancement of Structured Information Standards. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007. last accessed April 2009.

[68] Hagen Overdick. Towards Resource-Oriented BPEL. 2nd ECOWS Workshop on Emerging Web Services Technology, 2007.

[69] Mike P. Papazoglou. Service -Oriented Computing: Concepts, Characteristics and Directions. Proceedings of the Fourth International Conference on Web Information Systems Engineering, WISE, 2003.

[70] Michael Parkin. If Web Services are the Answer, What's the Question? IMG Seminar, Manchester, 2007.

[71] Cesare Pautasso. BPEL for REST. In BPM '08: Proceedings of the 6th International Conference on Business Process Management, pages 278–293, Berlin, Heidelberg, 2008. Springer-Verlag.

[72] Cesare Pautasso. RESTful Web service composition with BPEL for REST. Data and Knowledge Engineering, 2009.

[73] Cesare Pautasso and Gustavo Alonso. JOpera: A Toolkit for Efficient Visual Composition of Web Services. International Journal of Electronic Commerce, Vol. 9, No. 2, pp. 107–141, 2004.

[74] Cesare Pautasso, Biörn Biörnstad, Thomas Heinis, Francesco Lelli, Andreas Bur, Lucia Rusconi, and Adnan Al Hariri. JOpera for Eclipse. http://jopera.org, 2009.

[75] Cesare Pautasso, Olaf Zimmermann, and Frank Leyman. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. WWW 2008, Beijing, China, 2008.

[76] Chris Peltz. Web Services Orchestration and Choreography. IEEE Computer, vol.36, no.10, pp. 46-52, 2003.

[77] Ganesh Prasad. Namespace-Time: Why REST and SOAP Composition Models are so Different. http://wisdomofganesh.blogspot.com/2008/01/namespace-time-why-rest-and-soap.html, 2008.

[78] Netbeans Project. Netbeans IDE 6.5, SOA Features. http://www.netbeans.org/features/soa/index.html, 2009.

[79] Anton V. Riabov, Eric Bouillet, Mark D. Feblowitz, Zhen Liu, and Anand Ranganathan. Wishful Search: Interactive Composition of Data Mashups. WWW 2008, Beijing, China, 2008.

[80] Leonard Richardson and Sam Ruby. RESTful Web Services. O'Reilly, 2007.

[81] Richard Salz. Really Simple Web Service Descriptions. http://webservices.xml.com/pub/a/ws/2003/10/14/salz.html, 2003. last accessed, February 2009.

[82] Holger Schwichtenberg. Microsoft schreibt Workflow Foundation komplett neu. heise online, 2008. last accessed, February 2009.

[83] Amit P. Sheth, Karthik Gomadam, and Jon Lathem. SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups. IEEE Internet Computing, Vol. 11, No. 6., pp. 91-94., 2007.

[84] Connie U. Smith and Lloyd G. Williams. Software Performance AntiPatterns. WOSP '00: Proceedings of the 2nd international workshop on Software and performance, Ottawa, Ontario, Canada, 2000.

[85] James Snell. Resource-Oriented vs. Activity-Oriented Web Services. IBM developerWorks, 2004. http://www.ibm.com/developerworks/webservices/library/ws-restvsoap/.

[86] Karen Sollins and Larry Masinter. Functional Requirements for Uniform Resource Names. http://www.w3.org/Addressing/rfc1737.txt, 1994. last accessed April 2009.

[87] Thomas Steiner. Rest Describe & Compile. http://code.google.com/p/rest-api-code-gen/, 2007. last accessed, January 2009.

[88] Thomas Steiner. REST description languages. http://docs.google.com/View?docid=dgdcn6h3_38fz2vn5, 2007.

[89] Ron Ten-Hove and Peter Walker. JSR 208: Java Business Integration (JBI). http://www.jcp.org/en/jsr/detail?id=208, 2005. last accessed April 2009.

[90] Aphrodite Tsalgatidou and Thomi Pilioura. An Overview of Standards and Related Technology in Web Services. Distributed and Parallel Databases, Volume 12 , Issue 2-3, p. 135-162, 2002.

[91] Knowledge University of Kassel and Data Engineering Group. BibSonomy. http://www.bibsonomy.org/, 2009. last accessed, January 2009.

[92] Martin Vasko and Schahram Dustdar. Towards collaborative Service Mashup design - Actual challenges in collaborative web-scale workflow design. Technical Report, 2009.

[93] World Wide Web Consortium (W3C). A Little History of the World Wide Web. http://www.w3.org/History.html, 2000.

[94] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.1 (Second Edition). http://www.w3.org/TR/2006/REC-xml11-20060816/, 2006. last accessed April 2009.

[95] Norman Walsh. Norm's Service Description Language, NSDL. http://norman.walsh.name/2005/03/12/nsdl, 2005. last accessed, February 2009.

[96] Steven Webster and Leon Tanner. Developing Flex RIAs with Cairngorm microarchitecture. Flex Developer Centers, 2008.

[97] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall PTR, 2005.

[98] W3C World Wide Web Consortium. SOAP Version 1.2, Part 0: Primer. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/, 2007. last accessed March 2009.

[99] Yahoo. Yahoo Pipes. http://pipes.yahoo.com/pipes/. last accessed April 2009.

[100] Michael zur Muehlen, Jeffrey V. Nickersona, and Keith D. Swensonb. Developing Web Services Choreography Standards - The Case of REST vs. SOAP. Decision Support Systems 37, Elsevier, North Holland, 2004.