Programming Evolvable Web Services

Martin Treiber, Lukasz Juszczyk, Daniel Schall, Schahram Dustdar
Distributed Systems Group, Vienna University of Technology
Argentinierstraße 8/184-1
A-1040 Vienna, Austria
{treiber,juszczyk,schall,dustdar}@infosys.tuwien.ac.at

ABSTRACT

Web services have emerged as a technology for designing and composing distributed applications. Recent research increasingly addressed the need to adapt such systems based on changing requirements and environmental constraints. From the developers point of view, it is already a daunting task to update the description, implementation, or configuration of individual services that are already deployed in the runtime environment. A major undertaking is update and maintenance of large scale service environments.

In this work, we introduce a programming model enabling the adaptation and evolution of service-oriented systems in a simple and intuitive way. Most existing work focuses on self-adaptation aspects. We present a user-centric approach and a framework supporting both automatic mechanisms for adaptation and foremost a programming model to reduce the burden of reconfiguration, update, and customization of service-based applications. We implemented the programming model on top of *Genesis*, a Java-based Web services framework.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Evolutionary prototyping; H.3.5 [**Online Information Services**]: Web-based Services

General Terms

Web services, service evolution, adaptation

Keywords

Genesis, programming model, evolvable services

1. INTRODUCTION

The continuous evolution of software systems towards more robust, flexible and ultimately self-adapting systems holds different challenges for the implementation of such systems. One way to address these challenges is to build loosely coupled software systems [28] from existing well defined components. Service based applications, service compositions respectively, are examples of such systems, which support modifications (e.g., service replacement) without breaking the service based software system.

While the major emphasis was put on *how to model* services, considerably less attention was paid on the developer's perspective. Significant evolutionary of services during the life cycle [2, 25], like code refactoring or the implementation of new algorithms do not happen automatically, but rather require a human, i.e., the developer, in the loop who conducts these changes [30]. Developers require an answer to the question of *how to implement* services and their changes. Consequently, developers need the support from the service infrastructure to be able to modify services in an efficient manner. Furthermore, an adequate programming abstraction is required that allows developers to change the implementation of services according to changing requirements.

Current approaches offer limited support for developers regarding the modification of services. They treat Web services either as components [5], as resources [15], focus on interface descriptions [10, 16] or describe services with semantic techniques [1, 14]. The support for direct modifications of Web services in the deployment environment is limited, approaches like chain of adapters [18] intercept and transform messages but do not support the developer in modifying deployed Web services.

We propose a programming methodology for Web services which introduces concepts and mechanisms to implement evolvable Web services. We base our approach on an abstract, tree based representation of Web services which is the foundation for the implementation of evolvable Web services. Our prototype runtime environment implementation is build upon the Genesis framework [17], which is able to support the runtime modification of Web services and provides the needed abstractions for our programming methodology.

In the following section, we discuss application scenarios highlighting the need for adaptation mechanisms in Web services-based systems.

1.1 Application Scenarios

The presented application scenarios are motivated by the need to create service-based applications which may be subject to changing requirements. In the first scenario we discuss an example where distributed organizations form virtual organizations (VO) to collaborate on joint tasks/projects. The second case highlights the need to provide customized services based on the service consumers preferences.

- *VO formation and collaboration:* The global scale and distribution of companies have changed the economy and dynamics of businesses. In recent years, companies and individuals have started to form *virtual organizations* (VO) to harvest business opportunities which single partners cannot realize due to missing expertise or resources [29]. VOs are estab-

lished by creating connections between individual partners. Web services and SOA are the ideal technical framework to automate the formation process as well as interactions within VOs. Since VOs form and dissolve for the timespan of a specific collaboration, it is desirable to create tailored services supporting the needed interactions between partners in an easy manner. Thus, developers are required to create a set of services enabling collaborations. On the other hand, tailored services prevent unauthorized access to services or operations that should remain invisible to the collaboration partner.

- *Provisioning of custom services:* The Web and services have undergone fundamental changes. Users demand for personalization and context-awareness when using services. Services may be adapted based on the users' context information by offering extended features. Also, personalization plays an increasing role as the number of available services increases. A simple example is a Web portal where personalized views are created based on user preferences. Web service providers, for example hosting services in cloud environments, may want to create services offering a set of features (operations) targeting a specific consumer and/or community. These custom services can be created by selecting and aggregating operations from existing services.

Both scenarios demand for flexibility and adaptability of services. A system satisfying the needs of the presented application cases is not designed, developed, and deployed in a top-down manner, but rather changes and evolves over time. While research in the semantic Web services community focuses on automatic adaptation (e.g., mediation of messages, goal driven compositions, etc.) of services, our approach focuses on the user perspective. The fundamental question we attempt to address in this work is: how can developers efficiently adapt systems while maintaining the system's availability?

1.2 Adaptation in Service-oriented Systems

In the following, we discuss various possibilities for adapting service-oriented architectures (SOA). Generally, we distinguish between *(i) manual adaptation* performed by developers and *(ii) automatic (self-)adaptation*. The latter has recently received considerable attention from the research community while the former is not sufficiently supported by existing WS toolkits and frameworks. Figure 1 provides an overview of adaptation in SOA.

The development process can be viewed from various starting points. The *Provider* may start to implement services based on market demands and innovations of competitors. We make no assumption about the role of the provider with regards to development aspects of services. From the Web services point of view, functional capabilities of services can be described through well-defined interfaces [10], whereas non-functional characteristics, for example service metering, costs, and Quality-of-Service (QoS) attributes [25], can modeled using policy frameworks.

Providers typically offer a set of services to a number of *Consumers*. As mentioned in our motivating application scenarios, consumers may have different preferences and requirements, thereby demanding for customizations of services. The provider is responsible for hosting services in an appropriate *Infrastructure* to satisfy consumer demands in terms of functional capabilities and QoS.

Autonomic adaptation (see self-referential arrow) may need to be performed to satisfy QoS guarantees, service availability, to name a few and has received considerable attention (e.g., see [22]). For example, such self-adaptation actions may be triggered by notifying services about infrastructure events.

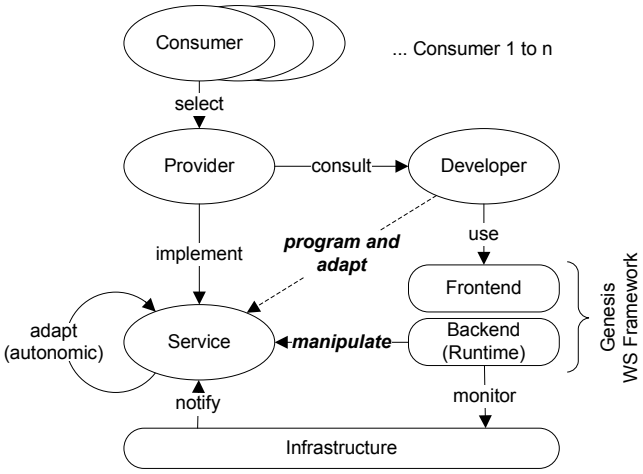

Figure 1: Overview adaptive SOA: the proposed programming model comprises programming and adaptation performed by developers and manipulation of services.

Here we address adaptations performed by the *Developer*. The developer is in charge of programming services and implementing adaptations (dotted arrow pointing from Developer to Service in Figure 1). A set of tools are needed for this purpose, which are depicted as *Frontend* and *Backend*, both provided by the *Genesis WS Framework* [17]. The developer performs the logical action 'program and adapt' using the frontend comprising tools, APIs, user shell, etc. The backend is deployed in the infrastructure — potentially multiple backend instances to achieve scalability. Also, the infrastructure is monitored by the backend to receive information about deployed resources or load conditions which can be propagated back to the frontend to assist the developer when adapting services. Automatic adaptations could potentially be triggered by the backend, although this is not within the scope of this work.

1.3 Contributions

In this work, we highlight the following novel key contributions:

1. A user-centric approach for programming and adapting Web services using the *Genesis WS Framework*. The proposed framework offers a script-based Web service programming environment.

2. A simple and intuitive programming model assisting developers in adaptation actions such as service migration, replication, or even refactoring of multiple distributed services at the same time.

3. Extensibility and flexibility in managing services through *behavior modules*.

The rest of this paper is organized as follows: in section 2 we introduce our programming model and discuss the mechanisms that are required to program evolvable Web services. We show how developers can be program and adapt services using a script-based programming approach. In the following section 3, an architectural overview is given. Afterwards, we discuss benefits and limitations of our approach in 4 and related work in 5. Finally, we conclude the paper in 6 and give an outlook on future work.

2. PROGRAMMING MODEL

The proposed programming model differs from the usual methodology for Web service development. In order to make Web services adaptable, in the sense of altering the service's interface and behavior at runtime, we regard it as useful to encapsulate its implementation into serializeable and composable building blocks. Today, the most common way of developing Web services is to create data/-code objects representing the services, with class methods implementing the service's operations. The main drawback of this approach is the tight binding of operations to services (which means, methods to objects) making it impossible to perform adaptations on a structural level without recompiling and redeploying the whole service. This poses a hard limitation for flexibility and adaptability.

Our approach, derived from the programming model of the Genesis testbed generator framework [17], splits the Web service model, according to the structure in WSDL documents [10], into 4 main layers of element types, namely Host, Service, Operation, and MessageType (see Figure 2).

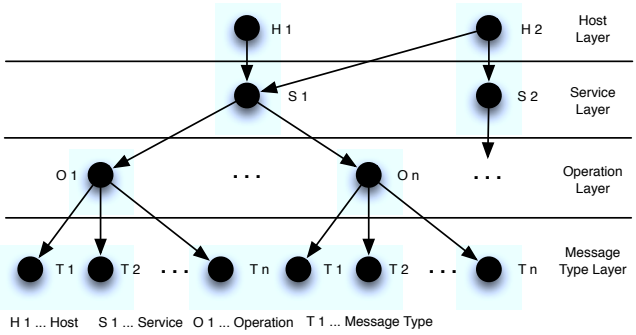

Figure 2: Programming abstraction for Web services.

The Host type references a remote host and provides access to its deployed services, in order to de-/re-/install these. The Service type represents the general specification of a Web service, with all its global dependencies and properties. Operations encapsulate the behavior of the service and MessageTypes define the schemas of the exchanged messages, which also includes headers.

Our methodology comprises the usage of an API at the front-end in order to control a set of back-end hosts. The main features of the API deal with creating and manipulating Web service models and for deploying them on the back-end hosts. At the back-end, a runtime environment handles the transformation of models into service instances and supports the basic CRUD (Create, Read, Update, Delete) operations for manipulating them. Furthermore, our approach supports the usage of pluggable behavior modules which augment the Web services with arbitrary functionality, e.g., access to data bases or registration at UDDI brokers. Table 1 summarizes the main effects of CRUD operations on the model.

2.1 Script-based Web Service Programming

Our framework provides a Java-based API for accessing the model, which can be also integrated into scripting environments such as the Bean Scripting Framework (BSF)[1] or Groovy[2]. In this paper, we are using sample code snippets written in Jython[3], a BSF-based implementation of the Python language, to demonstrate the simple usage of our programming model. The following sample shows the creation and deployment of a simple Web service:

[1] `http://jakarta.apache.org/bsf/`
[2] `http://groovy.codehaus.org`
[3] `http://www.jython.org`

Model	Modification
Host	**Create:** Bootstrapping of remote host (requires Cloud-like host manipulations) **Read:** Retrieve model of deployed Web services (e.g., for migration) **Update:** • Update of host-global behavior modules • Setting host-global properties **Delete:** Host shutdown
Service	**Create:** Service deployment for creation / replication / migration **Read:** Read service configuration and metadata **Update:** • of service behavior modules • of service properties (e.g. URL) **Delete:** Service undeployment
Operation	**Create:** Addition of operation **Update:** • of operation code • of operation properties (e.g. binding) **Delete:** Removal of operation
Message-Type	**Create:** Addition of headers and/or message types **Update:** • of request/response types and headers • of header processing code **Delete:** Removal of headers and/or message types
Behavior Module	**Create:** Deployment of pluggable behavior modules for extending Web services **Update:** • Replacement of modules • Steering of pluggable functions via parameter manipulation **Delete:** Undeployment of modules

Table 1: Modifications on model types and behavior modules.

```
% create host reference
h = Host("http://example.net:8080/services")

% create service definition with doc/lit binding
s = Service("SampleService")
s.setDocumentStyle()
s.setLiteralUse()

% create dummy method in Jython
def sign(par):
    signatureStr = "Not implemented yet"
    return signatureStr

% bind method to service operation
o = Operation("SignData")
o.setBehavior(sign)

% create XSD-based message types
t = MessageType("types.xsd", "dataTypeName")

% attach message types to operation
o.addInputType("param", t)
o.setOutputType("string") % type for sign() response

% attach operation to service
s.addOperation(o)

% attach service to host and deploy
s.deployAt(h)
```

By executing this script code, the Web service is deployed at `http://example.net:8080/services/SampleService`. The functionality of the service is defined by binding a native Jython method (`sign()`) to a Web service operation, in order to encapsulate the operation's behavior for remote execution.

Deployed Web services can be adapted by importing their model from a host, performing changes to it, and redeploying it again. The next snippet demonstrates this feature and shows how to extend services with behavior modules.

```
% import service model from remote host
h = Host("http://example.net:8080/services")
s = h.getService("SampleService") % get by name
o = s.getOperation("SignData")

% create new header message type
ct = MessageType("types.xsd", "credentials")
ct.setHeader(true)

o.addInputType("creds",ct)

% uses "auth" and "gpg" plugins
def newSign(par,creds):
    auth.check(creds) % exception on failure
    signatureStr = gpg.sign(par)
    return signatureStr

% replace operation behavior
o.setBehavior(newSign)

% install plugin at host using global alias "auth"
h.usePlugin("auth.jar")
% install plugin at service using local alias "gpg"
s.usePlugin("gpg.jar")

% redeployment at remote host
s.update()
```

Apart from simple adaptations, which change a service's interface and/or behavior, changes can be also performed at a higher level, by combining API methods into composite ones, for instance to migrate or replicate services to other hosts (see Figure 3).

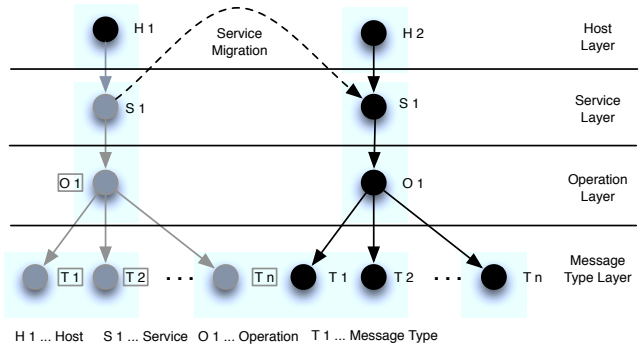

Figure 3: Migration of service S1 from host H1 to host H2

During deployment at back-end hosts, models of Web services are translated into running instances of these. For this purpose our system serializes service models, including the code blocks of the operations and all referenced pluggable modules, and transfers them to the designated hosts. The translation process itself comprises the analysis of the Web service model and the generation of Java code which implements the intented behavior. We do not present the details of the translation process in this paper, but refer interested readers to [17].

```
def replicateService(serviceName,fromHost,toHost)
    % import model from source host
    s = fromHost.getService(serviceName)
    if s is None:
        raise Exception("Unknown service")
    % deploy at new one
    s.deployAt(toHost)
    return s

def migrateService(serviceName,fromHost,toHost)
    s = replicateService(serviceName,fromHost,toHost)
    % remove after successful deployment
    s.undeployFrom(fromHost)

def migrateHost(fromHost,toHost)
    % iterate through deployed plugins and services
    for p in fromHost.getPluginModules():
        toHost.usePlugin(p)
    for s in fromHost.getServices():
        migrateService(s.getName(),fromHost,toHost)
```

2.2 Extending Services with Behavior Modules

For our programming model particular priority has been put on simplicity, allowing developers to set up Web services quickly and to perform adaptations in a convenient manner.

However, this came at the cost of sacrificing the ability to create complex Web services due to the encaplusation of Web service operations to single code blocks, e.g., to Jython methods like in the previous samples. To overcome this limitation to a certain degree, we are using pluggable behavior modules which can provide arbitrary functionality and which can be accessed by the operations.

Our framework supports developers by providing an abstract Java class for the modules, which takes care of binding them to the runtime environment via alias names. Modules can bei either registered at the host level, for globally visibility, or at the service level, for restricted visibility to particular services. We have developed a set of behavior modules which are frequently needed in the SOA domain, e.g., a service invoker for calling remote services and a simple workflow engine for executing nested BPEL[4] processes.

In the next sample a UDDI plugin is invoked in two special operations (`onDeploy()` and `onUndeploy()` which are executed automatically by the Genesis framework when services are installed or removed) in order to register the service automatically at a UDDI broker.

```
s = % ... get Service

% define deployment and undeployment hooks
def register():
    uddi.register(this)

d = Operation("onDeploy")
d.setBehavior(register)

def deregister():
    uddi.deregister(this)

ud = Operation("onUndeploy")
d.setBehavior(deregister)

s.addOperation(d)
s.addOperation(ud)

% attach plugin to service using local alias
s.usePlugin("uddi.jar")

% redeploy with hooks
s.update()
```

[4] `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`

3. ARCHITECTURE AND FRAMEWORK

Existing service execution environments have limited support for the run time modification of services. Traditionally, a runtime modification of a service would require the re-deployment of a service leading to downtimes. Our prototype is based on the Genesis framework [12, 17] allowing for the direct modification of services that are deployed. Genesis provides a Java-based framework for specifying characteristics of Web services and for generating instances of these services on-the-fly on a distributed backend. Via a plug-in facility, the service environment can be enhanced and updated (e.g., policies, implementation, and configuration) and, furthermore, can be controlled remotely.

The architecture is presented in Figure 3 consisting of four essential layers:

(i) SBA Developer Tools offering user interfaces for the developer, *(ii) Genesis Frontend* to access core API features, *(iii) Genesis Backend* responsible for hosting and monitoring services, and *(iv) SOA Environment* consisting of a set of service, clients, and other infrastructure elements such as a Web service registry to dynamically discover services, legacy services, and workflow engines, for example, BPEL execution engines.

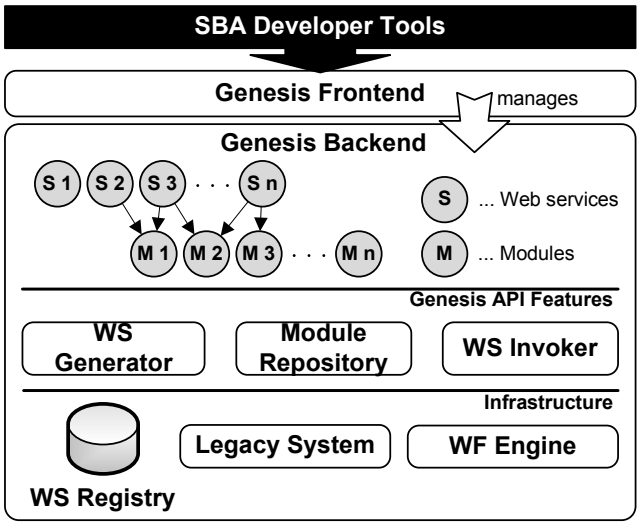

Figure 4: Architectural overview.

- *Genesis Frontend:* Provides means for defining, updating, and controlling SOA environments. Generated service instances are deployed and registered in the backend. The developer interacts with the frontend using a set of SBA developer tools. As mentioned previously, changes in the environment may including updates in service selection policies, features by enhancing message types, and upgrades in the services' implementation (modules).

- *Genesis API:* Provides the features to generate, deploy, and invoke Web services. As shown in Figure 3, `WS Generator` takes the user input (specification of a service) and generates the corresponding service artifacts. Behavioral models that can be (re)used by various services residing in the backend are stored and managed by the `Module Repository`. The `WS Invoker` provides the facility for creating requests to interact with services. Furthermore, the invoker plugin contains a reference to a logger plugin to log service invocations and other events triggered by Genesis.

- *Genesis Backend:* The backend comprises services that are deployed on a distributed set of hosts. Multiple backend instances can be used to manage large scale SOA environments. Backend instances communicate with each other to synchronize service updates and registrations.

4. DISCUSSION

The adaptation of services requires careful considerations. In the following two subsections, we discuss the strengths and the limitations of our proposed programming model and its current implementation.

4.1 Strengths

The flexibility of our model has several positive implications for the service developer who benefits from the following features:

- *Simplicity and intuitivity:* Web services can be created in a simple and intuitive manner. For example, the developer is able to modify services on the host layer (e.g., replication, migration) with the same programming primitives like on the service layer when changing the operations of a service.

- *Modularity:* As a unit of reuse [27], behavior modules allow developers to encapsulate arbitrary functionality and reuse these modules with different services in the Genesis runtime environment.

- *Run time service adaptation:* The modification of services at run time is supported by the Genesis based prototype infrastructure. The Genesis environment manages the (re-) deployment of services and the modifications in a transparent way for the developer. Our prototype implementation supports developers by hiding the complexity of service modifications and keeps the available services in sync with the abstract programming model.

- *Consistency between Model and Service:* The gap between the abstract model that describes a service and its implementation is very narrow. When manipulating the service model, the developer directly changes the implementation of the service and vice versa. Thus, we lay the foundation for automated service modifications which can be caused other than by the developer.

4.2 Current Limitations

Using a flexible programming model, we encounter a set of challenges that are of importance and are not fully addressed in our current version of the programming model. Limitations at this stage include:

- *Support for Stateful Services:* An issue that has strong impact on service adaptations is state [21]. Services which have internal state that influences the execution result of a service[5] must be treated in a manner that does not corrupt the state of the service during the adaptation process. A straightforward approach is to allow service modifications only in ground states when no transaction is active and the internal state of a service can be persisted and then later recovered to restart the service. This obviously limits applicable modifications, because (i) adaptations only can take place during a certain time interval and (ii) modifications that change the service

[5]S-Cube Knowledge Model: `http://www.s-cube-network.eu/km/terms/s/stateful-service`

semantics (e.g., the removal of an operation) are not applicable if the service is not in a ground state. In the present implementation, we do not support the complex manipulations of stateful services which require knowledge of meta information like active transactions.

- *Dependency Model:* Dependencies of services manifest themselves in different dimensions [31]. We can roughly distinguish between internal and external service dependencies. Examples of internal dependencies are the use of behavior modules, external dependencies can be observed as links to databases or libraries. These dependencies need to be taken into account, before a service is adapted. An approach to model dependencies is the use of manifest files that simply lists required resources of Web services that must be available for a service to function properly. In our programming model, we encounter these types of dependencies on all four layers. For example, the migration of an operation from one service to another might require the availability of a certain behavior module at the target service. Currently, we do not provide active support to manage dependencies on the infrastructure level.

- *Security Model:* Security concerns are no addressed with our current prototype implementation. Basically, each user that has access to the Genesis framework can modify each service at any given moment.

- *User Model:* Related to the security model are considerations about the user model. Similar to the security model, we do not support a dedicated user model in the current version.

- *Eventing Model - Event Propagation:* The causes that trigger adaptations can originate from different sources. For example, the observation that a service is operating near its pre-defined maximum throughput of 50 requests per second, might trigger a duplication of the service to another host to handle the load in order to fulfill the requirements of customers. Manually triggered adaptions, include for example internal code changes due to optimization of algorithms are triggered by the developer. The needed infrastructure is not yet implemented subject to future work.

5. RELATED WORK

Most existing works on the adaptation of services [11] addresses the adaptability on the level of service composition. Approaches like [26] or [8, 9] solve service composition adaptation with the help aspect oriented programming [19]. Similar in spirit, but focusing on syntactic replacement strategies, the work presented in [22] also adapts services on the composition level. With a finer granularity, the authors in [24] present an aspect oriented approach to implement adaptive services.

Much relevant work has been done in the area of Model Driven Development (MDD) of Web services. In [3] an approach for semi-automatic generation of Web service artifacts is presented. These artifacts include workflow definitions as BPEL, Web service interfaces as WSDL, and security constraints in WS-Policy. A similar approach is described in [4], where service templates and executable specifications are generated for simplifying the development of Web services. In [32] a framework is presented, which uses UML (Unified Modeling Language) specifications for Enterprise Distributed Object Computing (EDOC), which are translated into Web service skeletons and, optionally, BPEL processes.

Other approaches focus on mediation aspects of service adaption. The work presented [6] introduces the concept of adaptor, Kaminski et al. [18] who propose a chain of adaptors, and Kongdenfha et al. [20] who identify mismatch pattern between service protocols tackle the problem of service adaption problem from a behavior perspective. The work of [23] uses a semi-automated approach to identify and to resolve mismatches between service interfaces and protocols, in oder to generate service adapters. The ITACA toolbox [7] creates behavior models from abstract BPEL descriptions in order to generate adaptation contract specifications. A formal approach is taken by Dumas et al. [13] who introduce an algebra, complemented by a visual notation, to reconcile behavioral mismatches between services.

While having the same goal in creating adaptable services, our approach addresses the general problem of service adaptation on a different level. We introduce a hierarchical abstraction to implement services and organize the service internals. Using basic create, read, update and delete operations on the service abstraction, we provide the means for developers to modify services as well as the foundation for future automated adaptations.

6. CONCLUSION

We presented a programming model that supports the developer in creating adaptive services. Our proposed programming model structures services in a hierarchical model which abstracts from the actual implementation. We further separate implementation concerns since we encapsulate the functionality of services in behavior modules. This strong separation provides the developer of services to program adaptations with elementary operations (create, read, update and delete). For example the migration of a service operation from one service to another service is a sequence of read and update operations.

We implemented a prototype based on the Genesis framework [12, 17] that provides the required functionality to adapt services during run time. Our current prototype supports elementary operations on the proposed programming model and manages the run time service adaptations. Thus, developers of services do not need keep infrastructure issues in mind when modifying services.

In future work, we will address several issues. First of all, we are going to extend our programming model. We intend to include the support for complex events to trigger adaption activities. To address the issue of potentially creating adaptions which lead to inconsistent service states (e.g., a service operation change during a transaction), we aim at providing constraints that govern the adaption of services.

On a higher level of abstraction, we will investigate the use of policy driven adaptions such as load balancing or refactoring of services. Furthermore, we will investigate the use of an environment resource description model in order to be able to express service dependencies from external resources.

On the infrastructure level, we are going to extend the Genesis framework to support the aforementioned adaptation mechanisms and integrate rule engines like Drools [6] to provide the ability to manage complex adaptations. And finally, the deployment of the Genesis framework into cloud environments will be investigated to address scalability issues of the Genesis infrastructure.

Acknowledgment

The research leading to these results has received funding from the European Community Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (SCube) and 216256 (COIN).

[6] `http://www.jboss.org/drools/`

7. REFERENCES

[1] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M.-T. Schmidt, A. Sheth, and K. Verma. Web Services Semantics – WSDL-S, 2005.

[2] V. Andrikopoulos, S. Benbernou, and M. Papazoglou. Managing the evolution of service specifications. *Advanced Information Systems Engineering*, pages 359–374, 2008.

[3] R. Anzböck and S. Dustdar. Semi-automatic generation of web services and bpel processes - a model-driven approach. In *Business Process Management*, pages 64–79, 2005.

[4] K. Baïna, B. Benatallah, F. Casati, and F. Toumani. Model-driven web service development. In *CAiSE*, pages 290–306, 2004.

[5] M. Beisiegel, H. Blohm, D. Booz, M. Edwards, O. Hurley, S. Ielceanu, A. Miller, A. K. an Ashok Malhotra, J. Marino, M. Nally, E. Newcomer, S. Patil, G. Pavlik, M. Raepple, M. Rowley, K. Tam, S. Vorthmann, P. Walker, and L. Waterman. SCA Service Component Architecture SCA Service Component Architecture - Assembly Model Specification, 2007.

[6] B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani. Developing adapters for web services integration. *Advanced Information Systems Engineering*, pages 415–429, 2005.

[7] J. Camara, J. A. Martin, G. Salaun, J. Cubo, M. Ouederni, C. Canal, and E. Pimentel. Itaca: An integrated toolbox for the automatic composition and adaptation of web services. *Software Engineering, International Conference on*, 0:627–630, 2009.

[8] A. Charfi and M. Mezini. Aspect-oriented web service composition with ao4bpel. *Web Services*, pages 168–182, 2004.

[9] A. Charfi and M. Mezini. Ao4bpel: An aspect-oriented extension to bpel. *World Wide Web Journal: Recent Advances on Web Services (special issue)*, 10(3):309–344, September 2007.

[10] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1, 2001.

[11] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3-4):313–341, 2008.

[12] T. U. V. Distributed Systems Group, DSG. Genesis. Prototype web page, 2009. `http://www.infosys.tuwien.ac.at/prototype/Genesis`.

[13] M. Dumas, M. Spork, and K. W. 0002. Adapt or perish: Algebra and visual notation for service interface adaptation. In S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2006.

[14] D. Fensel, H. Lausen, J. de Bruijn, M. Stollberg, D. Roman, A. Polleres, and J. Domingue. Wsml a language for wsmo. *Enabling Semantic Web Services*, pages 83–99, 2007.

[15] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. Chair-Taylor, Richard N.

[16] M. J. Hadley. Web Application Description Language (WADL), 2006.

[17] L. Juszczyk, H.-L. Truong, and S. Dustdar. Genesis - a framework for automatic generation and steering of testbeds of complexweb services. In *ICECCS '08: Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems*, pages 131–140, Washington, DC, USA, 2008. IEEE Computer Society.

[18] P. Kaminski, M. Litoiu, and H. Müller. A design technique for evolving web services. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 23, New York, NY, USA, 2006. ACM.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[20] W. Kongdenfha, H. Motahari-Nezhad, B. Benatallah, F. Casati, and R. Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *Services Computing, IEEE Transactions on*, 2(2):94–107, April-June 2009.

[21] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The smart way to migrate replicated stateful services. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 103–115, New York, NY, USA, 2006. ACM.

[22] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In J. Huai, R. Chen, H.-W. Hon, Y. Liu, W.-Y. Ma, A. Tomkins, and X. Zhang, editors, *WWW*, pages 815–824. ACM, 2008.

[23] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 993–1002, New York, NY, USA, 2007. ACM.

[24] G. Ortiz and A. Prado. Adapting web services for multiple devices: A model-driven, aspect-oriented approach. In *Services - I, 2009 World Conference on*, pages 754–761, July 2009.

[25] M. P. Papazoglou. The challenges of service evolution. In *CAiSE '08: Proceedings of the 20th international conference on Advanced Information Systems Engineering*, pages 1–15, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] K. Ponnalagu, N. Narendra, J. Krishnamurthy, and R. Ramkumar. Aspect-oriented approach for non-functional adaptation of composite web services. In *Services, 2007 IEEE Congress on*, pages 284–291, July 2007.

[27] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. 2743:327–339, 2003.

[28] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[29] F. Skopik, D. Schall, and S. Dustdar. Innovative Human Interaction Services Specification (COIN FP7-216256). Technical report, 2009.

[30] M. Treiber, H.-L. Truong, and S. Dustdar. Semf - service evolution management framework. *Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference*, pages 329–336, Sept. 2008.

[31] M. Treiber, H.-L. Truong, and S. Dustdar. On analyzing evolutionary changes of web services. pages 284–297, 2009.

[32] X. Yu, J. Hu, Y. Zhang, T. Zhang, L. Wang, J. Zhao, and X. Li. A model driven development framework for enterprise web services. In *EDOC*, pages 75–84, 2006.