Deriving Interface Contracts for Distributed Services

Bernhard Hollunder
Department of Computer Science
Furtwangen University of Applied Sciences
Robert-Gerwig-Platz 1, D-78120 Furtwangen, Germany
Email: hollunder@hs-furtwangen.de

***Abstract*—Software components should be equipped with well-defined interfaces. With *design by contract*, there is a prominent principle for specifying preconditions and postconditions for methods as well as invariants for classes. Although design by contract has been recognized as a powerful vehicle for improving software quality, modern programming languages such as Java and C# did not support it from the beginning. In the meanwhile, several language extensions have been proposed such as Contracts for Java, Java Modeling Language, as well as Code Contracts for .NET. In this paper, we present an approach that brings design by contract to distributed services. To be precise, contracts included in the implementation of a Web service will be automatically extracted and translated into a so-called contract policy, which will be part of the service's WSDL. Our solution also covers the generation of contract-aware proxy objects to enforce the contract policy already on client side. The feasibility of our approach has been demonstrated for .NET/WCF services and for Java based Web Services.**

***Keywords*-Interface Contracts for distributed services; Design by contract; WCF; Web services; WS-Policy.**

I. Introduction

Two decades ago, Bertrand Meyer [1] introduced the *design by contract* principle for the programming language Eiffel. It allows the definition of expressions specifying preconditions and postconditions for methods as well as invariants for classes. These expressions impose constraints on the states of the software system (e.g., class instances, parameter and return values) which must be fulfilled during execution time.

Although the quality of software components can be increased by applying design by contract, widely used programming languages such as Java and C# did not support contracts from the beginning. Recently, several language extensions have been proposed such as *Code Contracts* for .NET [2], *Contracts for Java* [3] as well as *Java Modeling Language* [4] targeting at Java. Common characteristics of these technologies are *i*) specific language constructs for encoding contracts, and *ii*) extended runtime environments for enforcing the specified contracts. Approaches such as Code Contracts also provide support for static code analysis and documentation generation.

In this work, we will show how distributed services such as Web services can profit from the just mentioned language extensions. The solution presented tackles the following problem: Contracts contained in the implementation of a Web service are currently completely ignored when deriving its WSDL interface. As a consequence, constraints such as preconditions are not visible for a Web service consumer.

Key features of our solution for bringing contracts to distributed services are:

- simplicity
- automation
- interoperability
- client side support
- feasibility
- usage of standard technologies.

Simplicity expresses the fact that our solution is transparent for the service developer—no special activities must be performed by her/him. Due to a high degree of *automation*, the constraints (i.e., preconditions, postconditions, and invariants) specified in the Web service implementation are automatically translated into equivalent contract expressions at WSDL interface level.

As these expressions will be represented in a programming language independent format, our approach supports *interoperability* between different Web services frameworks. For example, Code Contracts contained in a WCF service implementation will be translated into a WSDL contract policy, which can be mapped to expressions of the Contracts for Java technology used on service consumer side. This *client side support* is achieved by generating contract-aware proxy objects. The *feasibility* of the approach has been demonstrated by proof of concept implementation including tool support.

In order to represent contract expressions in a Web service's WSDL, we will employ *standard technologies*: *i*) WS-Policy [5] as the most prominent and widely supported policy language for Web services, *ii*) WS-PolicyAttachment [6] for embedding a contract policy into a WSDL description, and *iii*) the Object Constraint Language (OCL) [7] as a standard of the Object Management Group (OMG) for representing constraints in a programming language independent manner.

Before we explain our solution in the following sections, we observe that several multi-purpose as well as domain-specific constraint languages have already been proposed for Web services (see, e.g., [8], [9], [10]). However, these papers have their own specialty and do not address important features of our approach:

- Contract expressions are automatically extracted from the service implementation and mapped to a corresponding contract policy.
- Our approach does not require an additional runtime environment. Instead, it is the responsibility of the underlying contract technology to enforce the specified contracts.
- Usage of well-known specifications and widely supported technologies. Only the notions "contract assertion" and "contract policy" have been coined in this work.

To the best of our knowledge, the strategy presented in this paper has not been elaborated yet elsewhere.

The paper is structured as follows. Next we will recall the problem description followed by the elaboration of the architecture and an implementation strategy on abstract level. So-called contract policies will be defined in Section IV. Then we will apply our strategy to Code Contracts for WCF services (Section V) and Contracts for Java for JAX Web services (Section VI). Limitations of the approach will be discussed in Section VII. The paper will conclude with a summary and directions for future work.

II. Problem Description

We start with considering a simple Web service that returns the square root for a given number. We apply Code Contracts [2] and Contracts for Java [3], respectively, to formulate the precondition that the input parameter value must be non negative.

The following code fragment shows a realization as a WCF service. According to the Code Contracts programming model, the static method `Requires` (resp. `Ensures`) of the `Contract` class is used to specify a precondition (resp. postcondition).

```
using System.ServiceModel;
using System.Diagnostics.Contract;

[ServiceContract]
public interface IService {
  [OperationContract]
  double squareRoot(double d);
}

public class IServiceImpl : IService {
  public double squareRoot(double d) {
    Contract.Requires(d >= 0);
    return Math.Sqrt(d);
  }
}
```

WCF service with Code Contracts.

The next code fragment shows an implementation of the square root service in a Java environment. In this example, we use Contracts for Java. In contrast to Code Contracts, Contract for Java uses annotations to impose constraints on the parameter values: `@requires` indicates a precondition and `@ensures` a postcondition.

```
import javax.jws.WebMethod;
import javax.jws.WebService;
import com.google.java.contract.Requires;

@WebService()
public class Calculator {
  @WebMethod
  @Requires("d >= 0")
  public double squareRoot(double d) {
    return Math.sqrt(d);
  }
}
```

Java based Web service with Contracts for Java.

Though the preconditions are part of the Web service definition, they will not be part of the service's WSDL interface. This is due to the fact that during the deploying of the service *its preconditions, postconditions, and invariants are completely ignored* and hence are not considered when generating the WSDL. This is not only true for a WCF environment as already pointed out in [11], but also for Java Web services environments such Glassfish/Metro [12] and Axis2 [13].

As contracts defined in the service implementation are not part of the WSDL, they are not visible to the Web service consumer—unless the client side developer consults additional resources such as an up to date documentation of the service. But even if there would exist a valid documentation, the generated client side proxy objects will not be aware of the constraints imposed on the Web service. Thus, if the contracts should already be enforced on client side, the client developer has to manually encode the constraints in the client application or the proxy objects. Obviously, this approach would limit the acceptance of applying contracts to Web services.

Our solution architecture overcomes these limitations by automating the following activities:

- Contracts are extracted from the service implementation and will be transformed into corresponding OCL expressions, which are packaged as WS-Policy assertions (so-called contract assertions).
- A contract policy (i.e., a set of contract assertions) will be included into the service's WSDL.
- Generation of contract-aware proxy objects—proxy objects that are equipped with contract expressions derived from the contract policy.
- Usage of static analysis and runtime checking on both client and server side as provided by the underlying contract technologies.

An important requirement from a Web service development point of view is not only the automation of these activities, but also a seamless integration into widely used

Integrated Development Environments (IDEs) such as Visual Studio, Eclipse, and NetBeans. For example, when deploying a Web service project no additional user interaction should be required to create and attach contract policies.

III. Architecture

The following figure shows the main components of our solution.

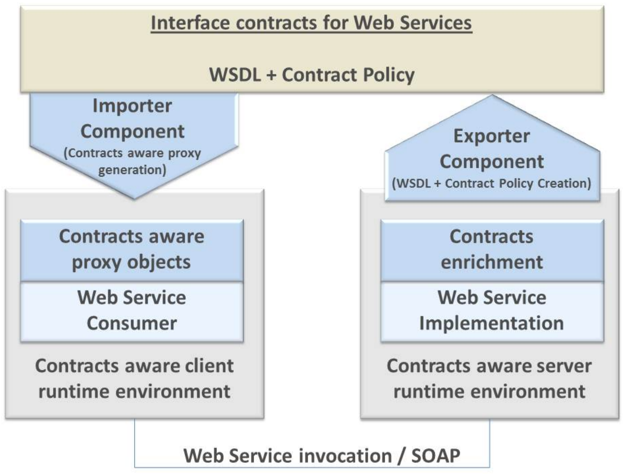
Interface contracts for Web Services
WSDL + Contract Policy
Importer Component
(Contracts aware proxy generation)
Exporter Component
(WSDL + Contract Policy Creation)
Contracts aware proxy objects
Web Service Consumer
Contracts aware client runtime environment
Contracts enrichment
Web Service Implementation
Contracts aware server runtime environment
Web Service invocation / SOAP

Figure 1. Solution architecture.

In short, our approach adopts the *code first strategy* for developing Web services. One starts with implementing the Web service's functionality in some programming language such as C# or Java. We assume that some contract technology is used to enhance the service under development by preconditions, postconditions, and invariants. In Figure 1, this activity is indicated by *contract enrichment*. At this point, one ends up with a contract-aware Web service such as the sample square root service at the beginning of Section II. In order to properly evaluate the contracts during service execution, a contract-aware runtime environment is required. Such an environment is part of the employed contract technology.

The standard deployment of the Web service will be adapted such that a contract policy is created and attached to the WSDL. The *exporter component* performs the following actions:

1) Extraction of contract expressions by inspecting the Web service implementation.
2) Construction of contract assertions and contract policies.
3) Creation of the service's WSDL and attachment of the contract policy.
4) Upload of the WSDL on a Web server.

Note that the contract policy will be part of the service's WSDL and is therefore accessible for the service consumer. Both the WSDL and the contract policy is used by the *importer component* to generate and enhance the proxy objects on service consumer side. The importer component fulfills the following tasks:

1) Generation of the "standard" proxy objects.
2) Translation of the contract assertions contained in the contract policy into equivalent expressions of the contract technology used on service consumer side.
3) Enhancement of the proxy objects with the contract expressions created in the previous step.

Note that service consumer and service provider may use different contract technologies.

IV. Contract Policies

A. Contract Assertions

This section defines contract assertions and contract policies. An important feature is their representation in some neutral, programming language independent format. We apply the well-known WS-Policy standard for the following reasons: WS-Policy is supported by almost all Web services frameworks and is the standard formalism for enriching WSDL interfaces. With WS-PolicyAttachment [6], the principles for including policies into WSDL descriptions are specified.

WS-Policy defines the structure of so-called assertions and their compositions, but does not define the "content" of assertions. To represent preconditions, postconditions, and invariants, we need some adequate language. We decided to use the Object Constraint Language (OCL) due to its high degree of standardization and support by existing OCL libraries such as the Kent OCL Library, the Dresden OCL Toolkit, and the Open Source Library for OCL (OSLO).

To formally represent constraints with WS-Policy, we introduce so-called *contract assertions*. The XML schema as follows:

```
<xsd:schema ...>
  <xsd:element name = "ContractAssertion"/>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name = "Precondition"
                   type = "xsd:string"
                   maxOccurs = "unbounded"/>
      <xsd:element name = "Postcondition"
                   type = "xsd:string"
                   maxOccurs = "unbounded"/>
      <xsd:element name = "Invariant"
                   type = "xsd:string"
                   maxOccurs = "unbounded"/>
    </xsd:sequence>
    <xsd:attribute name = "Name"
                   type = "xs:string"/>
    <xsd:attribute name = "Context"
                   type = "xs:anyURI"
                   use  = "required"/>
  </xsd:complexType>
</xsd:schema>
```

XML schema for contract assertions.

A `ContractAssertion` has two attributes: a mandatory *context* and an optional *name* for an identifier. The context attribute specifies the Web service to which the constraint applies. To be precise, the value of the context attribute is the name of the operation as specified in the `portType` section of the WSDL. In case of an invariant, the context attribute refers to the type defined in the `types` section.

The body of an contract assertion consists of a set OCL expressions. Depending on the surrounding element type the expression represents a precondition, a postcondition, or an invariant. The expressions may refer to the parameter and return values of an operation as well as to the attributes of a type.

B. OCL Expressions

OCL is a formal language for specifying particular aspects of an application system is a declarative manner. Typically, OCL is used in combination with the Unified Modeling Language (UML) to further constrain UML models. In OCL, “a constraint is a restriction on one or more values of (part of) an object-oriented model or system” [14]. In our context, OCL expressions will be used to specify constraints for Web services.

We use the following features of OCL in contract assertions:

- The basic types `Boolean`, `Integer`, `Real`, and `String`.
- Operations such as `and`, `or`, and `implies` for the `Boolean` type.
- Arithmetic (e.g., +, $\star$) and relational operators (e.g., =, <) for the types `Integer` and `Real`.
- Operations such as `concat`, `size`, and `substring` for the `String` type.

In order to impose restrictions on collections of objects, OCL defines operations for collection types. Well-known operations are:

- `size()`: returns the number of elements in a collection to which the method applies.
- `count(object)`: returns the number of occurrences of `object` in a collection.
- `includes(object)`: yields true if `object` is an element in a collection.
- `forAll(expr)`: yields true if `expr` is true for all elements in the collection.
- `select(expr)`: returns a subcollection containing all objects for which `expr` is true.

These operations may be used to constrain admissible values for collections occurring in the service’s WSDL.

Before we give some examples, we introduce the keywords `@pre` and `result`, which can be used in postconditions. To impose restrictions on the return value of a service, the latter keyword can be used. In a postcondition, the parameters may have different values at invocation and termination, respectively, of the service. To access the original value upon completion of the operation, the parameter must be equipped with the prefix `@pre`.

C. Examples

The first example considers the square root service from Section II, extended by a postcondition. The following XML fragment shows a formulation as a contract assertion:

```
<ContractAssertion context="SquareRootService">
  <Precondition>
    d >= 0
  </Precondition>
  <Postcondition>
    return >= 0
  </Postcondition>
</ContractAssertion>
```

Contract assertion for square root service.

The identifier `d` in the precondition refers to the parameter name of the service as specified in the WSDL.

The next example illustrates two features: *i*) the definition of an invariant and *ii*) the usage of a path notation to navigate to members and associated data values. Consider the type `CustomerData` with members name, first name and address. If address is represented by another complex data type with members such as street, zip and city, we can apply the path expression `customer.address.zip` to access the value of the zip attribute for a particular customer instance.

Whenever an instance of `CustomerData` is exchanged between service provider and consumer, consistency checks can be performed as shown in the following figure:

```
<ContractAssertion context="CustomerDataService">
  <Invariant>
    this.name.size() > 0
  </Invariant>
  <Invariant>
    this.age >= 0
  </Invariant>
  <Invariant>
    this.address.zip.size() >= 0
  </Invariant>
</ContractAssertion>
```

An invariant constraint.

To demonstrate the usage of constraints on collections we slightly extend the example. Instead of passing a single `customerData` instance, assume that the service now requires a collection of those instances. Further assume that the parameter name is `cds`. In order to state that the collection must contain at least one instance, we can apply the expression `cds->size() >= 1`. With the help of the `forAll` operator one can for instance impose the constraint that the zip attribute must have a certain value: `cds->forAll(age = 78120)`.

V. Code Contracts and WCF

Having defined the solution architecture in Section III and contract policies in the previous section, we will now instantiate our approach. This section investigates Code Contracts for WCF and the following section applies Contracts for Java to JAX Web services.

A. Exporting Contract Policies

In WCF, additional WS-Policy descriptions can be attached to a WSDL via a so-called custom binding. Such a binding uses the `PolicyExporter` mechanism also provided by WCF. To export a contract policy as described in Section III, a class derived from `BindingElement` must be implemented. The inherited method `ExportPolicy` contains the specific logic for creating contract policies. Details for defining custom bindings and applying the WCF exporter mechanism are described elsewhere (e.g., [11]) and hence are not elaborated here.

B. Creating Contract Assertions

Code Contracts expressions are mapped to corresponding contract assertions. Thereby we distinguish between the creation of *i*) the embedding context and *ii*) OCL expressions for preconditions, postconditions, and invariants.

In Code Contracts, a precondition (resp. postcondition) is specified by a `Contract.Requires` statement (resp. `Contract.Ensures`). Thus, for each `Requires` and `Ensures` statement contained in the Web service implementation, a corresponding element (i.e., `Precondition` or `Postcondition`) will be generated. The context attribute of the contract assertion is the Web service to which the constraint applies.

According to the Code Contracts programming model, a class invariant is realized by a method that is annotated with the attribute `ContractInvariantMethod`. For such a method, the element `Invariant` will be created; its context is the type that contains the method.

Let us now consider the mapping from Code Contracts expressions to corresponding ones of OCL. We first observe that Code Contracts expressions may not only be composed of standard operators (such as boolean, arithmetic and relational operators), but can also invoke pure methods, i.e., methods that are side-effect free and hence do not update any pre-existing state. While the standard operators can be mapped to OCL in a straightforward manner, user defined functions (e.g., prime number predicate) typically do not have counterparts in OCL and hence will not be translated to OCL. For a complete enumeration of available OCL functions see [7], [14].

Due to lack of space we cannot discuss details of the mapping. The following table focuses on selected features:

Code Contracts	OCL
`0 <= x && x <= 10`	`0 <= x and x <= 10`
`x != null`	`not x.isType (OclVoid)`
`Contract.OldValue(param)`	`@pre param`
`Contract.Result<T>()`	`return`
`Contract.ForAll (cds, cd => cd.age >= 0)`	`cds.forAll (age >= 0)`

In the first two examples `x` denotes a name of an operation parameter. They illustrate that there are minor differences regarding the concrete syntax of operators in both languages. The third example shows the construction how to access the value of a parameter at method invocation. While Code Contracts provide a `Result` method to impose restrictions on the return value of an operation, OCL introduces the keyword `return`. In the final example, `cds` represents a collection; the expressions impose restrictions which must be fulfilled by all instances contained in the collection.

C. Importing Contract Assertions

As shown in Figure 1, the role of the importer component is to construct contract-aware proxy objects. WCF comes with the tool `svcutil.exe` that takes a WSDL description and produces the classes for the proxy objects. Note that `svcutil.exe` does not process custom policies, which means that the proxy objects do not contain contract assertions.

WCF provides a mechanism for evaluating custom policies by creating a class that implements the `IPolicy-ImporterExtension` interface. In our approach, we create such a class that realizes the specific logic for parsing contract assertions and for generating corresponding Code Contracts assertions. As the standard proxy class is a partial class, the created Code Contracts assertions can be simply included by creating a new file.

VI. Contracts for Java for JAX Web services

In this section, we consider a contract technology for Java. These principles of this description can be carried over to other Java based contracts technologies.

A. Exporting Contract Policies

In Contracts for Java [3], the preconditions, postconditions, and invariants are expressed with the annotations `Requires`, `Ensures`, and `Invariant`, respectively. An example has been given in Section II.

The reflection API of Java SE allows the inspection of meta-data. In order to access the annotations of methods we apply these API functions. Given a method (which can be obtained by applying `getMethods()` on a class or an interface), one can invoke the method `getAnnotations()` to get its annotations. Such an annotation object represents the contract expression to be transformed into an OCL expression.

Before we consider in more detail this transformation, we discuss how to create and embed contract policies into WSDL descriptions. A Web services framework provides API functions for these tasks; these functions are not standardized, though. As a consequence, we need to apply the specific mechanisms provided by the underlying Web services frameworks.

Basically, the developer has to create a WS-Policy with the assigned assertions. To include the policy file into the service's WSDL, one can use the annotation `@Policy`, which takes the name of the WS-Policy file and embeds it into the WSDL. Other frameworks create an "empty" default policy, which can be afterwards replaced by the full policy file. During deployment, the updated policy will be embedded into the service's WSDL.

B. Creating Contracts Assertions

In Contracts for Java, the expressions contained in the `@requires`, `@ensures`, and `@invariant` annotations are either simple conditions (e.g., `d >= 0`) or complex terms with operators such as `&&` and `||`. As in Code Contracts, the expressions may refer to parameter values and may contain side-effect free methods with return type `boolean`. Similar to the mapping of Code Contracts expressions, these methods will not be mapped to contract assertions (see Section V-B).

The following table gives some hints how to map expressions from Contracts for Java to OCL.

Contracts for Java	OCL
`0 <= x && x <= 10`	`0 <= x and x <= 10`
`x != null`	`not x.isType (OclVoid)`
`Contract.OldValue(param)`	`old(param)`
`Contract.Result<T>()`	`result`

Note that Contracts for Java currently does not provide special support for collections (such as a `ForAll` operator). Thus, a special predicate needs to be defined by the "contract developer".

C. Importing Contract Assertions

To obtain the (standard) proxy objects, tools such as `WSDL2Java` are provided by Java Web services frameworks. Given a WSDL file, such a tool generates Java classes for the proxy objects. In order to bring the contract constraints to the proxy class, we apply the following strategy:

1) Import of the contract policy contained in the WSDL.
2) Enhancement of the proxy classes by Contracts for Java expressions obtained from the contract policy.

There is no standardized API to perform these tasks. However, Java based Web services infrastructures have their own mechanisms. A well-known approach for accessing the assertions contained in a WS-Policy is the usage of specific importer functionality. To achieve this, one can implement and register a customized policy importer, which in our case generates `@requires`, `@ensures`, and `@invariant` annotations as required for the contract assertions contained in the WS-Policy.

The second steps interleaves the generated expressions with the standard proxy classes. A minimal invasive approach is as follows: Instead of directly enhancing the methods in the proxy class, we create a new interface which contains the required Contracts for Java expressions. The proxy objects must only slightly be extended by adding an "implements" relationship to the interface created. This extension can be easily achieved during a simple post-processing activity after `WSDL2Java` has been called.

VII. Limitations and Open Issues

We have already mentioned that contract languages are more expressive that OCL. They in particular allow the usage of user-defined predicates implemented, e.g., in Java or C#. As OCL it not a full-fledged programming language, not every predicate can be mapped to OCL. In other words, only a subset of the constraints will be available at interface level. At first sight, this seems to be a significant limitation. However, the role of preconditions and postconditions is usually restricted to perform (simple) plausibility checks on parameter and return values. OCL has been designed in this direction and hence supports such kinds of functions.

When specifying contracts for a class, one may impose constraints on private members, which are not visible at interface level. As a consequence, it is not helpful to map these constraints to contract policies for WSDL. Thus, the generated contract policies should only impose constraints which are meaningful to service consumers. To be precise, the generated contract assertions should only constrain the parameter and return values of Web services as well as the public members of complex data types contained in the `types` section of a WSDL.

Although WS-Policy [5] and WS-PolicyAttachment [6] are widely used standards, there is no common API to export and import WS-Policy descriptions. As mentioned before, Web services infrastructures have their specific mechanisms and interfaces how to attach and access policies. Thus, the solutions presented in this paper must be adapted if another Web services framework should be used. For instance, the exporter and importer classes for processing contract policies must be derived from different interfaces; also the deployment of these classes must be adapted.

Finally, we observe that the exception handling must be changed, if contract policies are used. This is due to the fact that the contract runtime environment has the responsibility to check the constraints. If, e.g., a precondition is violated, an exception defined by the contract framework will be raised, that contains a description of the violation (e.g., that the value of a particular parameter is invalid). This must be respected by the client developer.

VIII. Conclusion and Future Work

In this work, we have demonstrated how contract technologies designed for programming languages can be leveraged for the interface creation for distributed services. Constraints imposed on the implementation will now be part of the WSDL interface and hence visible to the Web service consumer. This approach is a further step to improve the quality of distributed software components. This concept is in particular useful when applying the *code-first* approach for developing Web services, since additional properties of the service implementation (e.g., preconditions) will be automatically mapped to contract policies.

The developer of a Web service client application explicitly sees important constraints imposed on the service implementation and hence can consider this information. In addition, constraint violations can be detected already on client side, thus reducing network traffic and server consumption.

Due to the usage of the standardized, "neutral" language OCL for expressing constraints, the interoperability between different contract technologies and Web services infrastructures is given. For instance, preconditions encoded on server side with Contracts for Java will be translated to corresponding Code Contracts statements in a .NET consumer application.

As the contracts can be generated automatically, no additional effort is required for the service developer. As shown in our proof of concept, current tool chains can be enhanced such that the creation of the contract polices are completely transparent for the developer. Similar tool support is possible for service consumer side.

In this paper, we have focused on Contracts for Java as a contracts technology for Java. However, there are other well-known alternative technologies. A closer look to these approaches and their usage for generating contract policies is part of future work.

Acknowledgments

I would like to thank Ahmed Al-Moayed and the anonymous reviewers for giving helpful comments on an earlier version. This work has been partly supported by the German Ministry of Education and Research (BMBF) under research contract 17N0709.

References

[1] B. Meyer, "Applying "Design by Contract"," *Computer*, vol. 25, pp. 40–51, October 1992.

[2] Microsoft Corporation, "Code contracts user manual," http://research.microsoft.com/en-us/projects/contracts/user-doc.pdf, last access on 08/15/2011.

[3] N. M. Le, "Contracts for java: A practical framework for contract programming," http://code.google.com/p/cofoja/, last access on 08/15/2011.

[4] Java Modeling Language. http://www.jmlspecs.org/, last access on 08/15/2011.

[5] Web Services Policy 1.5 - Framework. http://www.w3.org/-TR/ws-policy/, last access on 08/15/2011.

[6] Web Services Policy 1.5 - Attachment. http://www.w3.org/-TR/ws-policy-attach/, last access on 08/15/2011.

[7] OMG, "Object constraint language specification, version 2.2," http://www.omg.org/spec/OCL/2.2, last access on 08/15/2011.

[8] A. H. Anderson, "Domain-independent, composable web services policy assertions," in *POLICY '06: Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 149–152.

[9] WS-SecurityPolicy 1.3. http://docs.oasis-open.org/ws-sx/wssecuritypolicy/v1.3, last access on 08/15/2011.

[10] A. Erradi, V. Tosic, and P. Maheshwari, "MASC - .NET-based middleware for adaptive composite web services," in *IEEE International Conference on Web Services (ICWS'07)*. IEEE Computer Society, 2007.

[11] B. Hollunder, "Code contracts for windows communication foundation (WCF)," in *Proceedings of the Second International Conferences on Advanced Service Computing (Service Computation 2010)*. Xpert Publishing Services, 2010.

[12] A. Goncalves, *Beginning Java EE 6 Platform with GlassFish 3*. Apress, 2009.

[13] D. Jayasinghe and A. Afkham, *Apache Axis2 Web Services*. Packt Publishing, 2011.

[14] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, 2003.