Design by Contract for Web Services: Architecture, Guidelines, and Mappings

Bernhard Hollunder, Matthias Herrmann, Andreas Hülzenbecher
Department of Computer Science
Furtwangen University of Applied Sciences
Robert-Gerwig-Platz 1, D-78120 Furtwangen, Germany
Email: hollunder@hs-furtwangen.de, matthias.herrmann@hs-furtwangen.de, huelzena@hs-furtwangen.de

***Abstract*—Software components should be equipped with well-defined interfaces. With *design by contract*, there is a well-known 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 Web services. We not only elaborate a generic solution architecture, but also define its components and investigate the foundations such as important guidelines for applying design by contract. Technically, the contract expressions imposed on a Web service implementation will be extracted and mapped into a contract policy, which will be included into the service's WSDL interface. Our solution also covers the generation of contract-aware proxy objects to enforce the contract policy on client side. We demonstrate how our architecture can be applied to .NET/WCF services and JAX Web services.**

***Keywords*-Design by contract; Web services; WS-Policy; Contract policies; WCF; JAX; Contracts for Java; Contracts-aware proxies.**

I. Introduction

Two decades ago, Bertrand Meyer [2] introduced the *design by contract* (DbC) 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 [3], *Contracts for Java* [4] as well as *Java Modeling Language* [5] targeting the Java language. 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 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 interface expressed in the Web Services Description Language (WSDL) [6]. As a consequence, constraints such as preconditions are not visible for a Web service consumer.

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

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

Simplicity expresses the fact that our solution is transparent for the Web service developer—no special activities must be performed by her/him. Due to a high degree of *automation*, the DbC assertions (i.e., preconditions, postconditions, and invariants) specified in the Web service implementation are automatically translated into semantically 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 Windows Communication Foundation (WCF) [7] service implementation will be translated into a WSDL contract policy, which can be mapped to expressions of the Contracts for Java technology deployed 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 [8] as the most prominent and widely supported policy language for Web services, *ii*) WS-PolicyAttachment

This is a revisited and substantially augmented version of "Deriving Interface Contracts for Distributed Services", which appeared in the Proceedings of the Third International Conferences on Advanced Service Computing (Service Computation 2011) [1].

[9] for embedding a contract policy into a WSDL description, and *iii*) the Object Constraint Language (OCL) [10] as a standard of the Object Management Group (OMG) for representing constraints in a programming language independent manner.

Design by contract is a useful instrument to improve service quality by imposing constraints, which must be fulfilled during execution time. As these constraints are typically expressed in a Turing complete language, arbitrary business logic could be encoded. In this paper, we also present guidelines on how to properly apply design by contract by identifying functionality, which should not be part of DbC assertions.

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., [11], [12], [13]). 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 an equivalent 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.

The paper is structured as follows. Next we will introduce the basics of design by contract. In Section III, we will recall the problem description followed by the elaboration of the solution architecture and an implementation strategy on abstract level. Guidelines for applying design by contract will be given in the Sections V and VI. So-called contract policies will be defined in Section VII. Then we will apply our strategy to Code Contracts for WCF services (Section VIII) and Contracts for Java for JAX-WS Web Services [14] (Section IX) followed by an example (Section X). Limitations of the approach will be discussed in Section XI. The paper will conclude with related work, a summary and directions for future work.

II. Design by Contract

Design by contract introduces the so-called assertions to formalize selected aspects of a software system. Assertions impose restrictions, which must be met at certain points during program execution. An assertion can either be a precondition and postcondition of a method or a class invariant. Typically, assertions constrain values of parameters and variables such as range restrictions and `null` values. If an assertion is violated during runtime (i.e., it is evaluated to `false`), this is considered to be a software bug [15].

A. Preconditions and Postconditions

Preconditions and postconditions are a means to sharpen the specification of a method. While the method's signature determines the required parameter types, preconditions and postconditions impose further restrictions on parameter values. Formally, a precondition (resp. postcondition) of a method is a boolean expression that must be true at the moment that the method starts (resp. ends) its execution. In general, such expressions can be quite complex comprising logical (e.g. `and`), arithmetic (e.g. `+`) and relational operators (e.g. `>`) as well as function calls (e.g. `size()`).

Preconditions ensure that methods are really invoked according to their specifications. Hence, a violation of a precondition can be viewed as a software bug in the invoking client code. In contrast, a method implementation can be considered incorrect, if its postcondition is violated. This is due to the fact that the implementation does not conform to its specification.

B. Class Invariants

A class invariant is a constraint that should be true for any instance of the class during its complete lifetime. In particular, an invariant guarantees that only those instances of a class are exchanged between method invoker and its implementation that conform to the invariant constraints. Analogously to preconditions and postconditions, a class invariant is a boolean expression, which is evaluated during program execution. If an invariant fails, an invalid program state is detected.

III. Problem Description

We start with considering a simple Web service that returns the square root for a given number. We apply Code Contracts [3] and Contracts for Java [4], 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` of the `Contract` class is used to specify a precondition while a postcondition is indicated by the method `Ensures`.

```
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);
  }
}
```

Listing 1. 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);
  }
}
```

Listing 2. 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 deployment 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 [16], but also for Java Web services environments such Glassfish/Metro [17] and Axis2 [18].

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 implementation. 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.
- The OCL expressions will be 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.

IV. Solution Architecture

In this section we introduce the components of the proposed architecture (see Figure 1). This architecture has been designed in such a way that is can be instantiated in several ways supporting both .NET/WCF as well as Java environments.

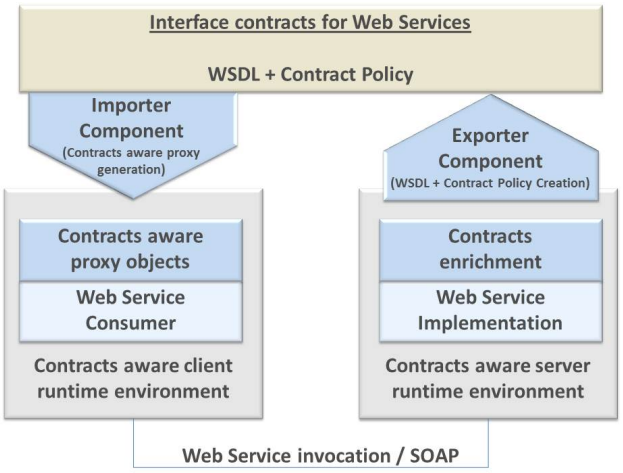

Figure 1. Solution architecture.

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 III.

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.

We adapt the standard deployment of the Web service such that a contract policy is created and attached to the WSDL. The *exporter component* performs the following tasks:

1) Extraction of contract expressions by inspecting the Web service implementation.
2) Validation of contract expressions.

3) Construction of contract assertions and contract policies.
4) Creation of the service's WSDL and attachment of the contract policy.
5) Upload of the WSDL on a Web server.

Note that the generated contract policy is 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) Mapping 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. Due to the usage of OCL as "neutral" constraint language, syntactic differences between the underlying programming languages will be compensated.

V. GUIDELINES

As mentioned in Section II, the DbC principle is a useful instrument to improve service quality by imposing constraints, which must be fulfilled during execution time. As the constraints are typically expressed in a Turing complete language, DbC can be misused for specifying arbitrary business logic contradicting the idea of interface contracts. This section is concerned with the question, which functionality should (not) be realized as DbC assertions. The conformance to these guidelines can be viewed as quality checking for DbC usage.

In the following, we take a closer look to the following areas:

1) Side effect free operations
2) Validation of external input data
3) Exception handling with assertions
4) Visibility of member variables and data types
5) Subtyping.

A. Side Effect Free Operations

In [15], the nature of assertions is described to be *applicative*. The term emphasizes that assertions should behave like mathematical functions and hence should not produce any side effects. As a consequence, read access to resources such as member variables is feasible, however their modification is disallowed. Strictly speaking, the term *applicative* not only excludes modifications of the object the assertions applies to, but also the invocation of operations that change the state of runtime objects such as logging or console entities.

It should be noted that in DbC technologies such as *Eiffel* [19], *Code Contracts* for .NET [3], *Contracts for Java* [4] and *Java Modeling Language* [5] assertions may invoke arbitrary functions of the underlying programming language. In their current versions, these technologies do not check the applicative nature of DbC assertions. In other words, a DbC designer does not get hints when invoking functions that directly or indirectly produce side effects.

B. Validation of External Input Data

Bertrand Meyer recommends that assertions must not be used for input validating (cf. [15]). Preconditions and postconditions are "between" the method caller and the method provider within a software component. In this sense, both caller and provider are part of the same software and do not represent an external system. Although a Web service consumer is typically part of a different software component, there is a deep logical dependency between service consumer and service provider component, which means that both components belong to the same system.

In contrast, validation of data coming from external systems should not be performed in DbC assertions. The logic for checking the quality of those data should be implemented in separate, standalone (importer) components by means of typical validation constructs such as `if/else`.

C. Exception Handling with Assertions

This aspect addresses the question how to proceed if an assertion is violated during runtime. Modern programming languages support well-known exception handling strategies based on `try/catch` blocks to locate abnormal situations and to start appropriate compensation actions.

Although current DbC technologies allow exception handling for dealing with failed assertions, a DbC designer should not intermix both techniques. Otherwise, an exception thrown by the DbC runtime environment should be handled by the surrounding application logic. This would not only have a negative impact on the overall code structure, but would also violate the first guideline "side effect free operations".

D. Visibility of Member Variables and Data Types

Web Services consumers and providers can be viewed as two parts of a common software system. However, both components are typically deployed and executed in separated runtime environments. As indicated in Figure 1, the Web service consumer sees an abstraction of the service implementation and its contained DbC assertions by means of the WSDL interface description. Thus, implementation details are not passed to the client.

For example, suppose that an assertion specified in the Web service implementation accesses a private member variable. As private member variables are not contained in a WSDL, this information is missing on consumer side. Obviously, it would not be possible to check this assertion in components, which invoke the Web service. The same

situation arises when specific data types and classes are embedded into DbC assertions, which are not available in the client environment. Thus, DbC assertions should only contain variables, data types, and methods that are meaningful and available also on Web service consumer side.

E. Subtyping

As most DbC technologies allow inheritance of preconditions and postconditions, it has to be considered how to handle such circumstances. Liskov and Wing [20] analyzed problems, which may occur in such settings. They argue that preconditions should not be strengthened by preconditions defined in derived classes. In contrast, postconditions and class invariants should not be weakened in the same way. This principle is also called *behavioral subtyping*.

Current DbC technologies apply a pragmatic approach to ensure this principle. They simply build disjunctions of the inherited preconditions. Analogously, postconditions and invariants are connected by a conjunction, which means that derived DbC assertions never become weaker (see, e.g., [21]). Of course, from a DbC developer point of view one would expect more support exceeding this basic syntactic manipulation.

VI. Automated Recognition of Guidelines

The automated detection of the described guidelines would be a useful feature of a DbC infrastructure. In the following, we elaborate to what extent such an approach would be feasible.

A. Side Effect Free Operations

Basically, the automated recognition of side effects is possible. We have to distinguish several cases. For example, suppose an assertion invokes a function that does not return any value (i.e., the return type is `void`). The only reason for calling this function is due to its side effects. The same is true in situations where return values are not processed in DbC assertions.

Another indicator for side effects are assignments to member variables such as `this.x = 5;`. Such statements can be easily identified by inspecting the code structure. It should be noted that some programming languages provide build-in support for declaring methods that have only limited access to resources. For example, in C++ member function can be marked with the keyword `const`, indicating that the function will not change the state of its enclosing objects [22].

In Code Contracts, the property `Pure` can be used to mark methods as side effect free. As described in [3], the current version behaves as follows: If a method not marked as `Pure` is used within an assertion, a warning is generated.

When it comes to automated recognition of side effects of entities such as logging or console, different strategies are conceivable. One strategy would be to disallow such method calls at all, even though they are not part of the "real" business logic. Such a restrictive approach would be inline with the applicative nature of DbC assertions. One could also imagine a more liberal strategy, which allows the usage of well-defined operations (e.g., `System.out.println`, `BufferedWriter`, etc. in a Java environment). Such method calls may facilitate debugging and testing both of the software system and the attached DbC assertions.

B. Validation of External Input Data

As described in the previous section, data received from external systems should not be validated by means of DbC assertions. To check this guideline, it must be figured out, whether a specific data source should be considered external. Due to the fact that DbC technologies support function calls within assertions, data can be fetched from arbitrary sources as shown in the following listing.

```
private int userNumberInput() {
  try {
    return Integer.parseInt(new BufferedReader(new
    InputStreamReader(System.in)).readLine());
  } catch (Exception e) { return -1; }
}
@Requires({ "userNumberInput() != -1" })
public int add(int a, int b) {
  return a + b;
}
```

Listing 3. Example for validating external input.

This code fragment applies *Contracts for Java* for specifying a precondition, which depends on data read from an input stream.

In this case, it is obvious that the precondition does not have any semantic relationship to the `add` method and should therefore be avoided. However, in general there are situations, which are more complicated. For example, consider a method that processes data taken from files or network sockets. Depending on its functionality, the validation of the received data may be part of the method's contract (and hence should be specified in a precondition) or may define some separate processing, which is only required in a specific context. Thus, an automatic compliance checker for this guideline is conceivable only in limited settings.

C. Exception Handling with Assertions

In contrast to the previous guideline, this rule can be recognized automatically. This can be achieved by analyzing the syntactic structure of the DbC assertions.

D. Visibility of Member Variables and Data Types

In general, due to the modifiers for visibility of member variables such as `private`, it could be derived automatically whether a member variable used in a DbC assertion is really meaningful to a local client.

Now suppose a client application, which invokes a Web service. As mentioned before, such a client sees the data

types and its embedded members, which are published in the WSDL of the Web service implementation. Hence, it can be checked whether a DbC assertion contains a member or a data type, which is not occurring in the WSDL. Hence, the exporter component can fully check this guideline.

E. Subtyping

Finally, the fifth guideline demands the conformance to the behavioral subtyping principle. In the previous section we observed that most DbC technologies have chosen a pragmatic approach by simply joining preconditions, postconditions, and invariants in derived classes (see, e.g., [23], [4]). However, this simple rewriting does not really solve the specification error.

In contrast, *Code Contracts* comes with a different strategy. As mentioned in [3], this technology does not allow adding any preconditions in derived types. For postconditions and invariants the behavior is similar to that of the corresponding Java technologies.

It should be noted, that a full validation of the subtyping principle is general not possible. Given two expressions (e.g., preconditions), it is in general not decidable for Turing complete DbC languages whether the one expression entails the other.

F. DbC Infrastructure Extensions

We have investigated to what extent an automated recognition of the proposed guidelines is possible. We have seen that most of the guidelines can be checked by inspecting the syntactic structure of the source code. So far, we have not implemented such a "code inspector". The focus of our work is the elaboration of an overall solution architecture, which identifies important components for, e.g., the extraction of DbC expressions and the generation of contracts-aware proxy objects.

A conformance checker for the proposed guidelines is not part of this work. In fact, we believe that such a functionality should be provided by concrete DbC technologies. The designers of DbC implementations such as [3] and [4] can use this information to improve their approaches. Basically, we assume that a DbC compiler should produce warnings, if guidelines are violated.

VII. Contract Policies

Having investigated important guidelines for DbC assertions, we now take a closer look to the exporter component. As shown in Figure 1, this component creates the interface contract for Web services, which is represented by a WSDL description together with a contract policy. In this section, we start with defining the building blocks of contract policies, followed by a very short introduction to the Object Constraint Language (OCL). In the final subsection, examples are given.

A. Contract Assertions

We now define contract assertions and contract policies, which allow the representation of constraints 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 [9], the principles for including policies into WSDL descriptions are specified.

WS-Policy defines the structure of the so-called assertions and their compositions, but does not define their "content". To represent preconditions, postconditions, and invariants, we need some language for formulating such expressions. We decided to use the Object Constraint Language (OCL) because of its high degree of standardization and support by existing OCL libraries such as the Dresden OCL Toolkit [24].

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>
```

Listing 4. 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 a contract assertion consists of a set of 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) [25] 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" [26]. 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., `+`, `*`) and relational operators (e.g., `=`, `<`) for the types `Integer` and `Real`.
- Operations such as `concat`, `size`, and `substring` for the `String` type.
- The collection types `Set` and `Sequence`.
- The construct `Tuple` to compose several values.

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.
- `reject(expr)`: returns a subcollection containing all objects for which `expr` is false.

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 III, extended by a postcondition. The XML fragment in Listing 5 shows a formulation as a contract assertion. The identifier `d` in the precondition refers to the parameter name of the service as specified in the WSDL.

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

Listing 5. Contract assertion for square root service.

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>
```

Listing 6. 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(zip = 78120)`.

VIII. Code Contracts and WCF

We now instantiate the solution architecture presented in Section IV. We start with investigating Code Contracts for WCF (this section); 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 IV, 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., [16]) 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 [10], [26].

The following table gives some examples for 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)`

Table 1. Mapping of Code Contracts expressions to OCL.

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. Collections and Functions

OCL provides a limited set of collection types and functions. Table 2 shows how important collection types and functions of C# will be mapped to OCL.

C#	OCL
`System.Collections.Generic. HashSet(Of T)`	`Set`
`System.Array`	`Sequence`
`System.Collections.Generic.List( Of T)`	`Sequence`
`!System.Linq.Enumerable.Any()`	`Collection. isEmpty()`
`System.Collections.ICollection. Count()`	`Collection. size()`
`System.Collections.Generic.List. Add(object)`	`Sequence .append( object)`
`int % int`	`Integer.mod (int)`
`System.Math.Max(double, double)`	`Double.max( double)`
`System.String.ToUpper()`	`String. toUpper()`
`System.String.Concat(String)`	`String. concat( String)`
`System.String.Substring(intStart, intOffset)`	`String. substring (intStart - 1, intStart + intOffset)`

Table 2. Mapping types and functions from C# to OCL.

It should be noted that there are some widely used types, which are not supported by OCL. An example is a type representing dates. Such a type is part of Java (e.g., `java.util.Date`) and C# (e.g., `DateTime` in the .NET system namespace). In such a case we propose the following strategy. We define a set of "virtual" OCL types (e.g., `OCL_Date`) together with the mapping rules between the corresponding types in the programming languages such as C# and Java. Thus, we can easily extend OCL by additional types, which are typically used in DbC assertions and which should be available at WSDL interface level. Of course, the implementations of the mappings to OCL must be adapted accordingly.

D. 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 `IPolicyImporterExtension` 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.

IX. Contracts for Java for JAX Web services

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

A. Exporting Contract Policies

In Contracts for Java [4], the preconditions, postconditions, and invariants are expressed with the annotations `Requires`, `Ensures`, and `Invariant`, respectively. An example has been given in Section III.

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 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 VIII-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)`
`old(param)`	`@pre param`
`result`	`return`

Table 3. Mapping of selected Contracts for Java expressions to OCL.

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. Collections and Functions

As for C#, we will also give mappings from Java collection types and functions to OCL (see Table 4).

Java type	**OCL type**
`java.util.Set`	`Set`
`java.util.Array`	`Sequence`
`java.util.List`	`Sequence`
`java.util.Collection.isEmpty()`	`Collection.isEmpty()`
`java.util.Collection.size()`	`Collection.size()`
`java.util.List.add(object)`	`Sequence.append(object)`
`int % int`	`Integer.mod(int)`
`java.lang.Math.max(double, double)`	`Double.max(double)`
`java.lang.String.toUpperCase()`	`String.toUpper()`
`java.lang.String.concat(String)`	`String.concat(String)`
`java.lang.String.substring(intStart, intEnd)`	`String.substring(intStart - 1, intEnd)`

Table 4. Data type mappings from Java to OCL.

D. 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 description.
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 provide their specific 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 for the contract assertions contained in the WS-Policy.

The second step 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.

X. Example

As an example, consider a weather data Web service as provided by the National Weather Service (NWS, `http://www.weather.gov`). We take a closer look to the `NDFDgenByDay` service, which is described as follows: *"Returns National Weather Service digital weather forecast data. Supports latitudes and longitudes for the continental United States, Hawaii, Guam, and Puerto Rico only. Allowable values for the input variable "format" are "24 hourly" and "12 hourly". The input variable "startDate" is a date string representing the first day (Local) of data to be returned. The input variable "numDays" is the integer number of days for which the user wants data."*

Many of the Web services provided by NWS require a latitude and a longitude both specifying the geographical point of interest. Depending on the service, additional parameters such as the start date and length of the forecast. If called successfully, such a service will return a string, which represents the weather forecast encoded in the Digital Weather Markup Language (DWML).

The following listing shows the service's interface with WCF/C#.

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

[ServiceContract]
public interface WeatherService {
 [OperationContract]
 string NDFDgenByDay(
         decimal latitude, decimal longitude,
         System.DateTime startDate,
         int numDays,
         string format);
}
```

Listing 7. WCF/C# interface of `NDFDgenByDay`.

With the help of the Code Contract technology we formalize some of the constraints expressed in the documentation. Listing 8 focusses on selected preconditions and postconditions of the service.

```
public class WeatherServiceImpl : WeatherService {
 public string NDFDgenByDay(
         decimal latitude,
         decimal longitude,
         System.DateTime startDate,
         int numDays,
         string format) {
  Contract.Requires(latitude > 120 && ...);
  Contract.Requires(longitude < -175 && ...);
  Contract.Requires(numDays > 0 && numDays < 8);
  Contract.Requires(format.Equals("12 hourly") ||
                    format.Equals("24 hourly"));
  Contract.Ensures(
       Contract.Result<string>().Length > 0);

  // here follows the implementation
  // of the core functionality
  return ... ;
 }
}
```

Listing 8. Excerpt of a NDFDgenByDay implementation.

Given this description of the service, our approach automatically derives a representation of the DbC constraints. In particular, it creates a contract assertion with an OCL encoding of the constraints (see Listing 9).

```
<ContractAssertion context="NDFDgenByDay">
  <Precondition>
    latitude > 120 && ...
  </Precondition>
  <Precondition>
    longitude < -175 && ...
  </Precondition>
  <Precondition>
    numDays > 0 && numDays < 8
  </Precondition>
  <Postcondition>
    result.size() > 0
  </Postcondition>
</ContractAssertion>
```

Listing 9. Contract assertion for the weather service.

As described above, this contract assertion is packaged into a contract policy, which will be attached to the weather service's WSDL. This interface description is used to generate the contract-aware clients for .NET/WCF (cf. Section VIII-D) and JAX (cf. Section IX-D).

XI. Limitations and Open Issues

We have already mentioned that contract languages have a higher expressivity than OCL. They in particular allow the usage of user-defined predicates implemented, e.g., in Java or C#. As OCL is 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.

Although WS-Policy [8] and WS-PolicyAttachment [9] 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 slightly 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—at least during the test phase of the software application.

XII. Related Work

There are several approaches that increase the expressivity of WSDL towards the specification of constraints. An overview of different constraint validation approaches is given in Froihofer [27] together with an evaluation of different implementation strategies ranging from in-place injection to wrapper- and compiler-based approaches. In particular, the impact on performance was investigated. In [28], Web services are enhanced by using DbC expressions to add behavorial information. In contrast to our work, the DbC assertions are not extracted from the Web service's implementation.

With WS-PolicyConstraints ([11], [29]) there is a domain independent, multi-purpose assertion language for formalizing capabilities and constraints as WS-Policy assertions. Basically, this language could be used to define code contract constraints. However, since WS-PolicyConstraints does not have an implementation, we use OCL expressions within contract assertions.

There are some results originating from the service monitoring area that are related to our approach. A communality is the usage of formal languages for specifying additional requirements for services not expressible within WSDL. Languages such as WS-Col (cf. [30]), WS-Policy4MASC (cf. [31]) and annotated BPEL (cf. [32]) are optimized for monitoring messages and support, for example, message filtering, logging and correlation. However, this is not the target of our approach. Instead, our focus lies on the usage of the standardized language OCL for the specification of constraints.

The formalization of security constraints for Web services is a hot topic since the early days of Web services. WS-Security [33] and WS-SecurityPolicy [12] are two well-known and widely used specifications for imposing constraints addressing encryption and attachment of signatures for Web services. These approaches do not compete with our approach, but can be applied in addition.

XIII. Conclusions

In this paper, we have elaborated a solution architecture that brings design by contract to Web services. Through our approach, important constraints (i.e., preconditions and postconditions for methods as well as class invariants) are no longer ignored, but will be included into the service's WSDL interface description. As a consequence, service consumers not only see the parameter types required to successfully invoke the Web service, but also restrictions imposed on the types such as range restrictions.

We would like to stress two important features of our approach. First, our solution is based on well-known and widely used standards such as WS-Policy and the Object Constraint Language. Hence, no proprietary frameworks are required to implement the solution architecture. Second, in order to show the feasibility of our approach, we have instantiated our architecture with two different DbC technologies and Web services frameworks.

As mentioned in Sections V and VI, the proposed guidelines can be viewed as quality checker. This means that only those expressions should be contained in DbC assertions, which are inline with the design by contract principle.

Acknowledgments

We would like to thank Ahmed Al-Moayed, Varun Sud, 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. Hollunder, "Deriving interface contracts for distributed services," in *The Third International Conferences on Advanced Service Computing*, 2011, p. 7.

[2] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40 –51, oct 1992.

[3] Microsoft Corporation, "Code contracts user manual," http://research.microsoft.com/en-us/projects/contracts/-userdoc.pdf, last access on 06/10/2012.

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

[5] Java Modeling Language. http://www.jmlspecs.org/, last access on 06/10/2012.

[6] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl/, last access on 06/10/2012.

[7] J. Löwy, *Programming WCF Services*. O'Reilly, 2007.

[8] Web Services Policy 1.5 - Framework. http://www.w3.org/-TR/ws-policy/, last access on 06/10/2012.

[9] Web Services Policy 1.5 - Attachment. http://www.w3.org/-TR/ws-policy-attach/, last access on 06/10/2012.

[10] Object Constraint Language Specification, Version 2.2, http://www.omg.org/spec/OCL/2.2, last access on 06/10/2012.

[11] 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.

[12] WS-SecurityPolicy 1.3. http://docs.oasis-open.org/ws-sx/wssecuritypolicy/v1.3, last access on 06/10/2012.

[13] 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.

[14] E. Hewitt, *Java SOA Cookbook*. O'Reilly, 2009.

[15] B. Meyer, *Object-Oriented Software Construction (Book/CD-ROM) (2nd Edition)*, 2nd ed. Prentice Hall, 3 2000. [Online]. Available: http://amazon.com/o/ASIN/0136291554/

[16] 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.

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

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

[19] B. Meyer, *Eiffel : The Language (Prentice Hall Object-Oriented Series)*, 1st ed. Prentice Hall, 10 1991. [Online]. Available: http://amazon.com/o/ASIN/0132479257/

[20] B. H. Liskov and J. M. Wing, "Behavioral subtyping using invariants and constraints," Tech. Rep., 1999.

[21] Y. Feldman, O. Barzilay, and S. Tyszberowicz, "Jose: Aspects for design by contract80-89," in *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, sept. 2006, pp. 80 –89.

[22] B. Stroustrup, *C++ Programming Language, The (3rd Edition)*, 3rd ed. Addison-Wesley Professional, 6 1997.

[23] G. T. Leavens, "Jml's rich, inherited specifications for behavioral subtypes," in *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*. Springer-Verlag, 2006.

[24] Dresden OCL: OCL support for your modeling language, http://www.dresden-ocl.org/, last access on 06/10/2012.

[25] Unified Modeling Language (UML), Infrastructure, http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF last access on 06/10/2012.

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

[27] L. Froihofer, G. Glos, J. Osrael, and K. M. Goeschka, "Overview and evaluation of constraint validation approaches in Java," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007.

[28] M. L. Reiko Heckel, "Towards contract-based testing of web services," *Electronic Notes Theoretical Computer Science*, vol. 82, pp. 145–5156, 2005.

[29] "WS-PolicyConstraints: A domain-independent web services policy assertion language," Anderson, Anne H., 2005.

[30] P. P. Luciano Baresi, Sam Guinea, "WS-Policy for service monitoring," in *Technologies for E-Services*. Springer, 2006.

[31] A. Erradi, P. Maheshwari, and V. Tosic, "WS-Policy based monitoring of composite web services," in *Proceedings of European Conference on Web Services*. IEEE Computer Society, 2007.

[32] S. G. Luciano Baresi, Carlo Ghezzi, "Smart monitors for composed services," in *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC 2004)*, 2004, pp. 193–202.

[33] WS-Security. http://www.oasis-open.org/committees/wss, last access on 06/10/2012.