Interface Contracts for WCF Services with Code Contracts

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*—Windows Communication Foundation (WCF) is a widely used technology for the creation and deployment of distributed services such as Web services. Code contracts are another .NET technology allowing the specification of preconditions, postconditions and invariants for .NET interfaces and classes. The embedded constraints are exploited for static analysis, runtime checking, and documentation generation. Basically, WCF services can be equipped with code contracts. However, it turns out that the WSDL interfaces generated for deployed WCF services do not include expressions for code contracts. Hence, the constraints imposed on WCF services are not visible for a service consumer. Though a proper integration of both technologies would bring additional expressive power to WCF and Web services, there does not exist a solution yet. In this paper, we present a novel approach that brings code contracts to WCF. Our solution comprises the integration of code contracts expressions at WSDL level as well as the generation of contracts aware proxies. The feasibility of the approach has been demonstrated by a proof of concept implementation.**

***Keywords*-Code Contracts, Windows Communication Foundations, WCF, Web Services, WS-Policy, WSDL, Contracts Aware Proxies.**

I. Introduction

Code contracts [2] are a specific realization of the *design by contract* concept proposed by Bertrand Meyer. With code contracts, i) methods of .NET types can be enhanced by preconditions and postconditions, and ii) .NET types can be equipped with invariant expressions that each instance of the type has to fulfill. While the application developer specifies code contracts for interfaces and classes, it is the responsibility of the runtime environment for checking the constraints and signaling violations. Furthermore, following tools are available for code contracts:

- Static code analysis;
- Documentation generation;
- Integration into VisualStudio IDE.

From a theoretical point of view, static code checking has its limitations and cannot detect all possible contract violations. Nevertheless, it is a sophisticated instrument to help identifying common programming errors during compile time thus improving code quality at an early stage.

With the Windows Communication Foundation (WCF), service-oriented, distributed .NET applications can be developed and deployed on Windows. WCF provides a runtime environment for hosting services and enables the exposition of .NET types, i.e., Common Language Runtime (CLR) types, as distributed services. WCF employs well-known standards and specifications such as XML [3], WSDL [4], SOAP [5], and WS-Policy [6]. The Web Services Interoperability Technology (WSIT) project [7] demonstrates how to create Web services clients and implementations that interoperate between the Java platform and WCF.

When developing a WCF service, one starts with the definition of an interface (e.g., in C#) that is annotated with the `ServiceContract` attribute. Without this attribute, the interface would not be visible to a WCF client. To realize the service, a class is created that implements the interface. During service deployment, WCF will automatically generate an interface representation in the Web Services Description Language (WSDL) for the service. WSDL is programming language independent and allows the creation of client applications written in other programming languages (e.g., in Java) and running on different platforms. With the help of tools such as `svcutil.exe` and `wsdl2java` so-called proxy classes for specific programming languages can be generated. A proxy object takes a local service invocation and forwards the request to the real service implementation on server side by exchanging so-called SOAP documents.

In order to combine contracts with WCF, one may proceed as follows: The methods in a WCF service implementation class are equipped with code contracts expressions, that impose preconditions, postconditions, as well as object invariants. The C# compiler will not produce any errors and will create executable intermediate code, which can be deployed in a WCF environment. However, the constraints imposed by code contracts are completely ignored when WCF generates the WSDL for the service. As a consequence, a WCF client application cannot profit from the code contracts attached to the service implementation. This behavior has already been observed elsewhere [8]; however, a generic solution has not been elaborated yet.

This is a revisited and substantially augmented version of "Code Contracts for Windows Communication Foundation (WCF)", which appeared in the Proceedings of the Second International Conferences on Advanced Service Computing (Service Computation 2010) [1].

This paper presents a novel approach for deriving interface contracts for WCF Services with code contracts. The strategy is as follows. When deploying a WCF service, the code contracts expressions contained in the service implementation class are extracted. Next, these expressions are encoded as assertions of WS-Policy [6]. The obtained WS-Policy description will be attached to the service's WSDL. On service consumer side, the generation of the proxy classes will be enhanced by including the code contracts expressions, which are extracted from the WS-Policy description of the WSDL.

The approach has the following features:

- It combines standard technologies such as WSDL and WS-Policy to bring code contracts to the interfaces of WCF services.
- The approach is transparent from a WCF service development point of view. The generation of both the interface contracts and the contracts aware proxy objects is completely automated.
- Code contracts will be already checked on client side, including static code analysis. This may save resources during runtime because invalid service requests will not be transmitted to server side.
- The feasibility of the approach has been demonstrated by a proof of concept implementation.

As indicated in [9, page 100], "software contracts play a key role in the design of classes for testability." Thus, with our approach, a WCF developer has a further instrument for improving the quality of distributed .NET components. This is mainly due to the fact that additional constraints are now visible at the interface level in a formalized manner.

The paper is structured as follows. The next section will shortly introduce the underlying technologies. Section III will recapitulate the problem description; the solution proposed will be presented in Section IV. Section V will show how to represent code contracts with WS-Policy and how to attach a WS-Policy description to a WSDL file. Then, in Section VI, an implementation strategy (proof of concept) will be described. Section VII will give more details on interface contracts creation, followed by related work. Section IX concludes the paper.

II. Foundations

This section will give a brief overview on the required technologies. We start with introducing code contracts, followed by WCF and WS-Policy.

A. Code Contracts

With code contracts [2] additional expressivity is brought to .NET interfaces and classes by means of preconditions, postconditions, and object invariants. A method can be equipped with preconditions and postconditions. A precondition is a contract on the state of the system when a method is invoked and typically imposes constraints on parameter values. Only if the precondition is satisfied, the method is really executed; otherwise an exception is thrown. In contrast, a postcondition is evaluated when the method terminates, prior to exiting the method.

Code contracts provide a `Contract` class in the namespace `System.Diagnostics.Contracts`. Static methods of `Contract` are used to express preconditions and postconditions. To give an example, consider a method `squareRoot` that should not accept negative numbers. This could be encoded as follows:

```
using System.Diagnostics.Contracts;

class MyService {
  double squareRoot(double d) {
    Contract.Requires(d >= 0);
    Contract.Ensures(Contract.Result<int>() >= 0);
    return Math.Sqrt(d);
  }
}
```

Defining a precondition and a postcondition for `squareRoot`.

The `Contract.Requires` statement defines a precondition by means of a boolean expression. To specify the postcondition that the return value of `squareRoot` is also non-negative, we apply the `Contract.Ensures` method. With help of the expression `Contract.Result<int>` the return value of the method can be referred to.

Object invariants of code contracts are conditions that should hold on each instance of a class whenever that object is visible to a client. During runtime checking, invariants are checked at the end of each public method. In order to specify an invariant for a class, an extra method is introduced that is annotated with the attribute `ContractInvariantMethod`. Within this method, the conditions are defined with the method `Contract.Invariant`.

To give an example, consider the type `CustomerData` with members name, first name, identifier, and address. For sake of simplicity, the following excerpt does not show the complete class definition, but focuses on the specification of an invariant. The invariant ensures that any instance of `CustomerData` must have a name with a certain length, a non-negative identifier, as well as a real address instance.

```
using System.Diagnostics.Contracts;

class CustomerData {
  string name;
  string firstName;
  int identifier;
  Address address;

  [ContractInvariantMethod]
  void ObjectInvariant(){
    Contract.Invariant(
      name.length() >= 2 && identifier > 0 &&
      address != null);
  }
}
```

Definition of an invariant.

The expressions contained in code contracts 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. Code contracts also provide the universal quantifier `Contract.ForAll` and the existential quantifier `Contract.Exists`.

Both quantifiers expect a collection and a predicate, i.e., a unary method that returns a boolean. Universal quantification yields true if the predicate returns true on all the elements in the collection. Analogously, existential quantification checks whether the predicate is fulfilled for at least one element in the collection.

The above `squareRoot` example shows how preconditions and postconditions can be specified for *classes*. As a method in an *interface* is described only by its signature and does not have a body, `Contract.Requires` and `Contract.Ensures` statements cannot be part of an interface definition. Code contracts foresee a simple trick to encode constraints for interface methods: the required constraints are specified in a separate class, which is associated with the interface.

Suppose a class `AContract` should implement code contracts for an interface `IA`. Then `IA` is annotated with the attribute `[ContractClass(typeof(AContract))]`, and the class `AContract` is equipped with `[ContractClassFor(typeof(IA))]`. Now the code contracts of `AContract` apply to the interface `IA`.

Note that most methods of the `Contract` class are conditionally compiled. It can be configured via symbols to which degree code contracts should be applied during compilation. Code contracts can be completely turned on, which means that a full checking is performed, and off, i.e., all `Contract` methods are ignored. It is also possible to check only selected code contracts constraints such as preconditions (for details see [2]).

B. Windows Communication Foundation

According to [10], "WCF is a software development kit for developing and deploying services on Windows." Services are autonomous, distributed and have well-defined interfaces.

An important feature of a WCF service is its location transparency: a consumer always uses a local proxy object – regardless of the location (local vs. remote) of the service implementation. The proxy object has the same interface as the service and forwards a call to the service implementation by exchanging SOAP documents. As the messages are independent of transport protocols, WCF services may communicate over different protocols such as HTTP, TCP, IPC and Web services.

The following listing shows the `squareRoot` functionality from above as a WCF service.

```
using System.ServiceModel;

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

public class IServiceImpl : IService {
  public double squareRoot(double d) {
    return Math.Sqrt(d);
  }
}
```

`squareRoot` as a WCF service.

The `ServiceContract` attribute maps the interface to a technology-neutral service contract in WSDL. To be part of the service contract, a method must be explicitly annotated with `OperationContract`. In order to implement the service, a class is created that inherits the interface as shown in the example.

Besides service contracts WCF also provides so-called data contracts. Data contracts are types, which can be passed to and from the service. There are built-in types such as `int` and `string`. Custom types can be declared as data contracts with help of the `DataContract` attribute. The `CustomerData` from above can be published as a data contract as follows:

```
using System.ServiceModel;

[DataContract]
class CustomerData {
  [DataMember]
  string name;
  [DataMember]
  string firstName;
  [DataMember]
  int identifier;
  [DataMember]
  Address address;
}
```

`CustomerData` as data contract.

In order to successfully deploy a WCF service, the WCF runtime environment requires the definition of at least one *endpoint*. An endpoint consists of

- an *address*,
- a *binding* defining a particular communication pattern,
- a *contract* that defines the exposed services.

Endpoints are typically defined in an XML configuration file (external to the service implementation), but can also be created programmatically.

During deployment, WCF generates a WSDL interface description for the service. A WSDL description has an interchangeable, XML-based format and comprises different parts, each addressing a specific topic such as the abstract interface and data types, the mapping onto a specific communication protocol such as HTTP, and the location of a specific WCF service implementation.

There are tools that transform WSDL descriptions into a programming language specific representation. Such a representation comprises classes for the proxy objects used by client applications. WCF delivers the tool `svcutil.exe`, which generates proxy classes for, e.g., C# together with a configuration file containing endpoint definitions. Basically, a proxy object constructs a SOAP message, which is sent to server side. A SOAP message consists of a body, containing the payload of the message (including the current parameter values of the request), and an optional header, containing additional information such as addressing or security data.

C. WS-Policy

When taking a closer look to a generated WSDL file one will find a couple of policy entries. These entries add further information to the service such as security requirements, reliable messaging, and arbitrary constraints. For example, it can be formally described that the parameter values of a request are to be encrypted during transmission.

WS-Policy is a widely used specification [6] to formulate policies in an interoperable manner. Almost all application servers including WCF support WS-Policy. In general, WS-Policy is a framework for defining policies, which comprise so-called (WS-Policy) assertions. A single assertion may represent a domain-specific capability, constraint or requirement and has an XML representation.

The following XML fragment shows how to associate a WS-Policy description to a service definition.

```
<definitions name="Service">
  <Policy wsu:Id="SamplePolicy">
    <ExactlyOne>
      <All>
        <IncludeTimestamp/>
        <EncryptedParts>
          <Body/>
        </EncryptedParts>
      </All>
    </ExactlyOne>
  </Policy>
  ...
  <binding name="IService" type="IService">
    <wsp:PolicyReference URI="#SamplePolicy"/>
    <operation name="squareRoot"> ... </operation>
  </binding>
  ...
</definitions>
```

WS-Policy attachment to a WSDL description.

In the example, a WS-Policy description is embedded into the WSDL of the `squareRoot` service. To be precise, via the `PolicyReference` element (for details see [6], [11]) a policy can be attached to a service. The top-level `Policy` element of a policy description has the child element `ExactlyOne`, which contains a set of so-called policy alternatives. Each alternative is surrounded by the `All` operator. The (single) alternative in the sample policy contains two assertions: `IncludeTimestamp` and `EncryptedParts`. This policy requires that the caller of the service has to

- include a time stamp into the SOAP message
- encrypt the body of the SOAP message.

Note that an attached policy description is part of the WSDL interface of the service and must be taken into account by the service invoker. In the example, the WCF server side runtime environment would immediately reject the request (without performing `squareRoot`), if a client does neither include a time stamp nor encrypt the message body.

This example also demonstrates the declarative approach of WS-Policy. Additional, non-functional requirements can be formally described with corresponding assertions in a policy. Policies are external to the service implementation and can be simply combined. The WCF runtime environment has the responsibility to obey the policy.

WS-Policy itself does not come with concrete assertions. Instead, related specifications such as WS-SecurityPolicy [12] and WS-ReliableMessaging [13] apply WS-Policy to introduce specific assertions (e.g., `IncludeTimestamp` and `EncryptedParts` from the example above) covering specific domains. The respective specifications do not only define the syntax, but also the meaning of the assertions and their impact on the Web services runtime behavior.

WS-Policy has been designed in such a way that further, custom-designed assertions can be introduced. Our approach exploits this features to encode contracts expressions and to attach them to the service's WSDL.

III. Problem Description

Suppose we want to create a WCF service with code contracts. A straightforward approach to combine both technologies would be as follows:

```
using System.ServiceModel;
using System.Diagnostics.Contracts;

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

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

WCF service with code contracts.

We define a WCF service interface as usual according to the WCF programming model. In addition to the previous implementation of Section II-B, the `squareRoot` service is equipped with a precondition and a postcondition.

We first note that this WCF service implementation will be successfully compiled and deployed. However, we also observe that the WSDL description created during the deployment phase does not include any information about code contracts contained in the service's implementation. In other words, code contracts expressions are completely ignored and are not part of the WSDL interface.

There are two important consequences to stress here:

1) Code contracts imposed on the service implementation will not be considered when generating the proxy classes.
2) Clients of the WCF service are not aware of any code contracts expressions. Hence, code contracts support such as static analysis and runtime checking is not available on client side.

Thus, if a client invokes the `squareRoot` service passing a negative number as argument, the proxy object will forward the request to the server. The server itself will delegate the request to the service implementation. During the execution of the service, the code contracts runtime environment will eventually detect the violation of the precondition. The execution will be aborted and an exception will be returned to the client.

In the following, we will elaborate a concept that brings code contracts to the client side, thus enabling constraint checking already before passing the request via a network protocol to the server.

IV. Code Contracts and WCF: The Concept

The overall concept of our solution for combining WCF and code contracts is illustrated in Figure 1.

One starts with implementing a WCF service according to both the WCF and the code contracts programming models.The service may use methods of the `Contract` class such as `Requires` and `Ensures` to specify preconditions and postconditions. For WCF data contracts, invariants can be defined.

During service deploying, the WCF infrastructure generates a WSDL for the service. Our approach will perform the following additional activities:

1) The code contracts expressions are extracted from the WCF service implementation class and are translated into corresponding WS-Policy assertions (so-called code contracts assertions).
2) A `PolicyReference` element is included into the WSDL of the service according to the WS-PolicyAttachment specification. The reference points to the code contracts assertions from the previous step.

The right part of Figure 1 shows the strategy to create proxy objects that are equipped with code contracts, so-called contracts aware proxies. In a first step, we derive the standard proxies from the WSDL by applying `svcutil.exe` provided by WCF. Then, these proxies will be enhanced in the following way:

1) Extraction of the code contracts assertions attached to the service's WSDL.
2) Creation of corresponding preconditions as well as postconditions and their integration into the proxy classes. Classes derived for data contracts are extended by invariant methods.

Before we will discuss each of these steps in more detail, we make some observations. First of all, the standard programming models both for WCF and code contracts can be applied when implementing a service. The enhanced deployment infrastructure has the responsibility to perform the above mentioned activities. By automating these activities, our approach is transparent from a developer point of view.

Secondly, code contracts imposed on WCF services are also available for client technologies other than .NET. In fact, Web services technologies such as JAX-WS [14] extended by Java-based contract technologies can also profit from the code contracts assertions attached to the WCF service's WSDL. In Section VII-B, we will elaborate this feature in more detail.

Finally, we observe that our solution exploits and applies widely used technologies and specifications such as WS-Policy and WS-PolicyAttachment, which are supported by WCF and almost all Java-based Web services infrastructures. Thus, no proprietary frameworks must be installed to realize our approach.

V. Code Contracts Assertions for WS-Policy

To formally represent code contracts expressions with WS-Policy, we introduce a WS-Policy assertion type, which is called `CodeContractsAssertion`.

The XML schema is defined as follows. Note that we omit, for sake of simplicity, some attributes such as `targetNamespace`.

```
<xsd:schema ...>
  <xsd:element name = "CodeContractsAssertion"/>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name = "requires"
                   type = "xsd:string"
                   maxOccurs = "unbounded"/>
      <xsd:element name = "ensures"
                   type = "xsd:string"
                   maxOccurs = "unbounded"/>
      <xsd:element name = "invariant"
                   type = "xsd:string"
                   maxOccurs = "unbounded"/>
    </xsd:sequence>
    <xsd:attribute name = "context"
                   type = "xs:anyURI"
                   use  = "required"/>
    <xsd:attribute name = "name"
                   type = "xs:anyURI"/>
  </xsd:complexType>
</xsd:schema>
```

XML schema for `CodeContractsAssertion`.

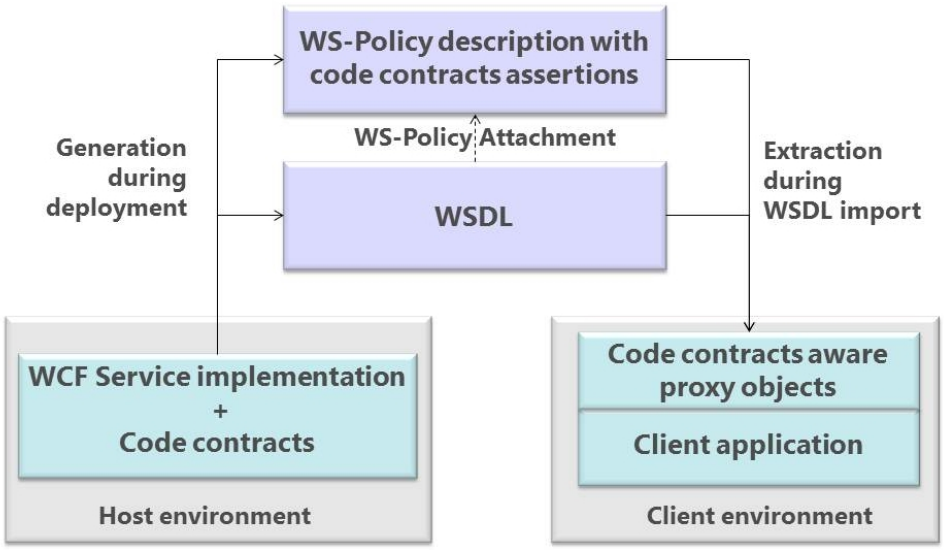

Figure 1. Combining code contracts and WCF.

A `CodeContractsAssertion` assertion has two attributes: a mandatory `context` and an optional `name`. The `context` attribute specifies the service to which the constraint applies. To be precise, the value of the context attribute is the (uniquely defined) name of a service as specified in the `binding` section of the WSDL. In case of an invariant, the context refers to a type defined in the `types` section. The `name` attribute can be applied to attach additional information to an assertion, which is not is processed.

The body of a `CodeContractsAssertion` consists of a set of `requires`, `ensures`, and `invariant` elements. The values of these elements have the type `xsd:string` and should be valid code contracts expressions. An expression contained in a `requires` (resp. `ensures`) element represents a precondition (resp. postcondition) and typically refers to parameter names of the service. Note that these names are also part of the WSDL and can therefore be resolved properly.

An `invariant` expression applies to instances of data types used as service parameters. Such an expression may impose restrictions on the public members of the type, which are visible to a WCF client application.

Note that a code contracts expression contained in a service implementation class may impose restrictions on members, which are not visible – and hence are not meaningful – at WSDL interface level. Section VII will discuss this issue in more depth.

The created `CodeContractAssertions` are packaged into a WS-Policy description, which is attached via a `PolicyReference` to the service definition. The following WS-Policy description is produced for the WCF `squareRoot` service from Section III.

```
<definitions name="Service">
  <Policy wsu:Id="CCPolicy">
    <ExactlyOne>
      <All>
        <CodeContractsAssertion
          name="squareRootAssertion"
          context=
            "IService.squareRoot(System.Double)">
          <requires>
            d >= 0
          </requires>
          <ensures>
             Contract.Result<int>() >= 0
          </ensures>
        </CodeContractsAssertion>
      </All>
    </ExactlyOne>
  </Policy>
  ...
  <binding name="IService" type="IService">
    <wsp:PolicyReference URI="#CCPolicy"/>
    <operation name="squareRoot"> ... </operation>
  </binding>
  ...
</definitions>
```

Code contracts policy for the `squareRoot` service.

In the following section, we will describe how to technically realize our concept.

VI. Proof of Concept

We start with elaborating how to create and attach policies for code contracts during the WCF deployment process. Thereafter, we will focus on the activities at the service consumer side, which especially covers the generation of contracts aware proxies.

A. Code Contracts Extraction

Given a WCF service implementation, we need some mechanism to obtain its preconditions, postconditions and invariants. API functions have been published to access code contracts expressions. These functions are part of the *Common Compiler Infrastructure* project [15]. We adapted the proposed visitor pattern to obtain the methods' code contracts expressions and created a function `getCodeContractsForAssembly` that computes for a given assembly a code contracts dictionary. This dictionary is organized as follows:

- The *key* is the full qualified name of the method.
- The *value* is a list of strings each representing a code contracts expression. Each expression has the prefix `requires:`, `ensures:`, or `invariant:` to indicate its type.

The function `getCodeContractsForAssembly` can be implemented in straightforward manner by using types defined directly or indirectly in the namespace `Microsoft.Cci`.

B. Creation of WS-Policy Code Contracts Assertions

In this step, we create an XML representation for the code contracts expressions according to XML schema for `CodeContractsAssertion` as defined in the previous section.

We have realized a function `createCodeContractsAssertions` that takes a filled dictionary from the previous step. It iterates on the keys and performs the following actions:

- For each key, a `CodeContractsAssertion` is created. The value of its context attribute will be the key's name.
- For each expression contained in a key's value, an XML element is embedded into the `CodeContractsAssertion`. Depending on the well-defined prefix, the XML element will be either `requires`, `ensures`, or `invariant`.

The result of this step is a complete list of `CodeContractsAssertion`s for the code contracts expressions in the service's implementation. How to embed a set of `CodeContractsAssertion`s as a WS-Policy description into a WSDL file is described next.

C. WS-Policy Creation and Attachment

In WCF, additional policies can be attached to a WSDL file via custom bindings [16]. We define a custom binding that uses the `PolicyExporter` mechanism also provided by WCF. To achieve this, we implement two classes:

- `ExporterBindingElementConfigurationSection`
- `CCPolicyExporter`.

The former class is derived from the abstract WCF class `BindingElementExtensionElement`. The inherited method `CreateBindingElement` is implemented in such a way that an instance of the `CCPolicyExporter` class is created. `CCPolicyExporter` has `BindingElement` as super class and implements the `ExportPolicy` method, which contains the specific logic for creating code contracts policies. Figure 2 visualizes the class layout.

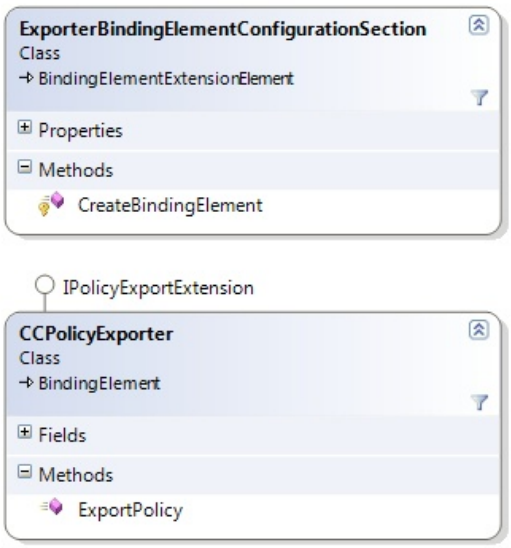

Figure 2. Class diagram for WS-Policy creation.

We have implemented the `ExportPolicy` method in the following way. In a first step, `getCodeContractsForAssembly` is invoked to obtain the filled dictionary. Next, `createCodeContractsAssertions` produces an XML representation for the code contracts expressions, which is then embedded into a valid WS-Policy description. Finally, a `policyReference` element pointing to this policy is embedded into the WSDL. Thus, we end up with an enriched WSDL description as shown at the end of the previous section.

To publish the service, a so-called endpoint must be configured (for details see, e.g., [10]). We have to change the standard configuration file of the WCF service such that the custom binding will be used:

1) In the definition of the service endpoint, the attribute `binding` is changed to `customBinding` and the attribute `bindingConfiguration` is set to `exporterBinding`.
2) In the `bindings` section, the `customBinding` element declares `exporterBinding`.
3) The element `bindingElementExtensions` is introduced in the `extensions` section. Its `add` element specifies the assembly in which the `ExporterBindingElementConfigurationSection` class is implemented.

During service deployment, WCF now uses the custom binding. As a result, the generated WSDL file will contain the code contracts policy.

D. Importing Code Contracts Policies

In order to invoke a service, a WCF client application requires a definition of a service endpoint. Typically, this is declared in a configuration file, similar to the one used on server side. In the `metadata` section of this file we included the definition of a so-called policy importer. By default, custom policies attached to a WSDL will not be evaluated when importing a WSDL.

In order to process code contracts policies, the `policyImporters` element refers to the class `CCPolicyImporter` of Figure 3.

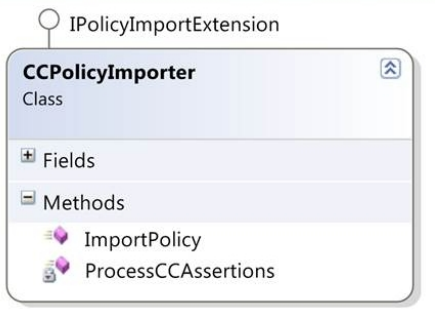

Figure 3. Class diagram for accessing WS-Policy descriptions.

We have realized this class in the following way. It implements the WCF interface `IPolicyImporterExtension`, which declares the `ImportPolicy` method. When processing a WSDL interface description, the runtime environment invokes this method and passes the attached policy description. `CCPolicyImporter` implements `ImportPolicy` in such a way that a code contracts dictionary (similar to the one on server side as described in Section VI-A) is constructed. To achieve this, the private method `ProcessCCAssertions` iterates on the code contracts assertions of the policy and adds corresponding entries to the dictionary. This dictionary will be used to enhance the proxy classes, which is shown next.

E. Enhanced Proxy Generation

The tool `svcutil.exe` does not process custom policies. Hence, the standard proxy classes generated do not contain any code contracts constraints.

In our proof of concept we have realized the following approach. First, we apply `svcutil.exe` to create the standard proxy classes. In a second step, the following activities are performed:

1) Create an additional source file that will contain all constraints found in the code contracts policy. This file is called contract file.
2) Link the contract file to the proxy class. Note that we do not want to modify the proxy class generated by `svcutil.exe`, because this would result in a strong dependency to the concrete code structure of the proxy.

Before we discuss the structure of the contract file, we observe that the generated proxy has a public interface (the proxy interface) that describes the supported services. There is also a public partial class (the proxy class) that implements the proxy interface. A client application instantiates the proxy class and invokes a provided service.

In the proof of concept, we apply the following strategy to bring the code contracts to the proxy. We create a new interface (the contract interface), that contains those methods that must be equipped with preconditions and postconditions. We also introduce a further class (the contract class) implementing the contract interface. The inherited methods are implemented in the contract class in such a way that they contain the required `Contract.Requires` and `Contract.Ensures` statements. The contract class will be annotated with `ContractClassFor` to indicate that the constraints apply to the methods of the contract interface. The details for linking a contract class to an interface can be found at the end of Section II-A.

Next, we extend the partial proxy class in the contract file by inheriting the proxy interface. As the contract class is linked to the proxy interface, the preconditions and postconditions are also applicable to the proxy class.

For an invariant expression contained in the code contract policy we proceed as follows. The partial proxy class to which the invariant applies will be extended in the contract file by a new method that is annotated with `ContractInvariantMethod`. This method contains the required `Contract.Invariant` statements. This completes the generation and linkage of the contract file with the proxy generated by `svcutil.exe`.

F. Data Contracts

In WCF, data contracts are types that can be passed to and from the service. In addition to built-in types such as `int` and `string`, user defined data contracts can be introduced be annotating a class with the `DataContract` attribute. WCF will serialize all fields marked with `DataMember`. To impose object invariants on data contracts, methods annotated with `ContractInvariantMethod` will be introduced in the class context. The `Contract.Invariant` statements contained in these methods will then be managed by code contracts runtime.

As an example consider a data contract `CustomerData` (cf. Section II-A) with members such as name and address and an object invariant method that imposes restrictions on possible values. Suppose a WCF service `createCustomer` takes an instance of `CustomerData`. Because `CustomerData` is part of the service's signature, it has a representation as `complexType` in the WSDL. Therefore, `svcutil.exe` will generate a corresponding partial C# class `CustomerData`, which is used by the service consumer to construct instances. This class provides public setters

and getters for the members, but contains only a default constructor to create "empty" instances.

In order to invoke the `createCustomer` service, a client may proceed as follows:

1) Create an empty instance of `CustomerData`.
2) Set the specific values of the members with the public setters.
3) Pass the instance to the service.

Unfortunately, the code contracts runtime environment on client side will report an error after the first step. This is due to the fact that the empty members will (probably) define an invalid state of the instance, which is recognized by the object invariant.

To overcome this problem, one needs on client side a public constructor that takes all relevant customer data and constructs a properly initialized instance, which conforms to the object invariant. However, such a constructor is not generated by the standard `svcutil.exe` tool.

Therefore, part of the enhanced proxy generation is also the introduction of a suitable public constructor for a data contract class. This constructor will be part of the partial proxy class introduced in the contract file.

Observe that on WCF service provider side this is not an issue, though. When introducing a data contract, specific constructors can be implemented by the creator of the WCF service. These constructors are available for general usage on WCF provider side.

G. Exception Handling

There are two separated code contracts runtime environments: one on WCF service consumer side and one on WCF service provider side.

As described in Section 7 of [2], code contracts support several runtime behavior alternatives. By default, a contract violation yields an "assert on contract failure". Thereafter, a user interaction is required to continue or abort program execution. While this behavior may be acceptable on client side during the development and testing phase, an analogous behavior would not be helpful on WCF provider side. Each time a violation occurs, the WCF service process requires a user interaction, which means that the server process must be observed the whole time. In general, this is not acceptable, not even during development and testing.

To remedy this problem, we disable "assert on contract failure" in the WCF service project. As a consequence, a contract violation now leads to the creation of an exception, which will be handled by the WCF runtime environment. By default, WCF returns a `FaultException` to the client indicating that something went wrong without giving detailed information. In order to embed the real reason into the exception (e.g., a "Precondition failed: d >= 0" message) the `IncludeExceptionDetailInFaults` parameter of the `ServiceBehavior` attribute in the WCF service implementation class is set to true as shown in the following listing:

```
using System.ServiceModel;
using System.Diagnostics.Contracts;

[ServiceBehavior
  (IncludeExceptionDetailInFaults = true)]
public class IServiceImpl : IService {
  public double squareRoot(double d) {
    Contract.Requires(d >= 0);
    Contract.Ensures(Contract.Result<int>() >= 0);
    return Math.Sqrt(d);
  }
}
```

Exception creation for a WCF service with code contracts.

On client side, standard exception handling can be applied to inspect the exception's reason.

H. Development Model

To sum up, the development model that brings code contracts to WCF services is as follows:

1) Creation of a WCF service and an assembly with VisualStudio as usual, e.g., as *WCF Service Library* project.
2) Definition of a service endpoint that includes a modified configuration file with a custom binding as described in Section VI-C.
3) Deployment of the WCF service by launching the project. The published WSDL will contain a code contracts policy.
4) Creation of a WCF client project with VisualStudio as usual.
5) Invocation of the `ClientConnectorTool`, which is part of the proof of concept. This tool has a graphical user interface (see Figure 4) and generates for a specified WSDL contracts aware proxies, which will be included into a selected client project and assembly, respectively.
6) Usage of the code contracts infrastructure on client side.

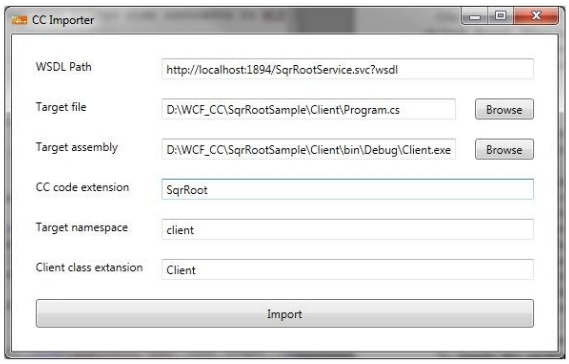

Figure 4. Tool for generating contracts aware proxies.

It should be noted that the code contracts processing is transparent to the developer – with the exception that the code contracts runtime environment and tools are now available on client side.

Our approach has the following advantages for the client developer. First, a static analysis of the code contracts can be performed, which helps detecting invalid invocations of WCF services during compile time. Second, during runtime a validation of the constraints will already be performed on client side. As a consequence, invalid service calls are not transmitted to the service implementation thus saving resources such as bandwidth and server consumption.

VII. Creating Code Contracts Policies: A Closer Look

Before discussing interoperability issues in VII-B, we will observe that in general only a subset of the code contracts expressions of the service implementation should be mapped to code contracts policies.

A. Limitations

When specifying code contracts for a WCF implementation class, one may impose constraints on private members, which are not visible at the WSDL interface level. As a consequence, it is not helpful to map these constraints to contracts policies. Therefore, the generated contract policies should only contain constraints, which are meaningful to service consumers and hence can be validated on client side. To be precise, the code contracts policy should constrain the parameter and return values of WCF services as well as the public members of data contracts types. Constraints in the service implementation, which are not mapped to the code contracts policy, will be checked by the contracts environment on server side.

As mentioned in Section II-B, 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. For example, suppose that a precondition checks whether a parameter value of type `int` is a prime number. This can be achieved by invoking a custom predicate `isPrimeNumber` in the `Contract.Requires` statement. In order to check this precondition during service invocation, the implementation of the predicate must be available on client side. Thus, when importing code contracts policies on client side, only those predicates should be included into the contracts aware proxies, which are known on client side. Otherwise, the compiler will report an error on client side.

B. Interoperability

So far we have assumed that both on client and server side there is a .NET environment supporting the code contracts technology. Due to the interoperability of the Web services technology, Java based technologies may invoke WCF services (see [14], Section 12). In our current approach, the code contracts expressions are implemented in a .NET language and must adhere to the specific syntax of C# or VB.

In order to use code contracts policies in a Java-based client environment, the expressions must be translated into equivalent Java expressions. As argued in [17], it is of advantage to represent expressions in contracts policies in a neutral, programming language independent format. In fact, the Object Constraint Language (OCL) of the Object Management Group [18] is a standardized language for formalizing constraints. In [17], the mappings from C# and Java, respectively, to OCL and vice versa are elaborated in more detail.

VIII. Related Work

There are two research areas, which are related to our approach:

- Design by contract technologies;
- Constraints for policy languages.

While the former category comprises the realization of the *design by contract* principle for programming languages, the latter covers formalisms for specifying constraints for Web services.

Recently, several language extensions for Java have been proposed towards the specification of preconditions and postconditions for methods as well as invariants. Two well-known frameworks are *Contracts for Java* [19] and *Java Modeling Language* [20]. Typically, annotations such as `@Requires` and `@Ensures` are introduced by the frameworks to impose additional constraints. These approaches are targeting at the core Java programming language and do not address the impact on Web services environments such as JAX-WS. As in the case of WCF, these annotations are completely ignored when generating a service's WSDL. As argued in [17], our concept is rather generic and can also be applied to Java environments.

The formalization of non-functional requirements for Web services is a hot topic since the early days of Web services. Based on WS-Policy, WS-SecurityPolicy [12] is a well-known specification for imposing security constraints for Web services such as message integrity and confidentiality. There are proposals for defining domain-independent assertion languages such as WS-PolicyConstraints [21] and WS-Policy4MASC [22], which can in principle be used to encode code contracts expressions. However, these and related approaches do not address how to map constraints embedded in the service implementation to these formalisms.

IX. Conclusion

In this paper, we have elaborated a concept that brings code contracts to WCF. To be precise, we have shown how to i) derive interface contracts for WCF services and ii) create contracts aware proxy objects. As a consequence, WCF application developers can now profit from the additional

expressive power of code contracts including runtime and tool support. It has been stressed elsewhere that there did not exist a solution to the problem.

Our approach exploits well-known standards such as WSDL, WS-Policy, and WS-PolicyAttachment. We have described how to represent code contracts expressions by means of WS-Policy assertions. This representation will be used to generate an enhanced client proxy infrastructure, thus allowing the evaluation of the WCF service's code contracts already on client side. The developer of a WCF client application now explicitly sees important constraints imposed on the service implementation thus reducing the number of service invocations with invalid parameter values.

As noted in the section on related work, there are design by contracts approaches for Java. An interesting direction for future work is concerned with the question how interface contracts can be mapped to the Java environment. A solution may map code contracts expressions into a programming-language independent representation (e.g., in OCL). Afterwards, the OCL constraints will be translated into the specific syntax of the design by contracts technology used on service consumer side. Hence, a Java client application can also profit from the code contracts embedded in the WCF service implementation [17].

Currently, we are elaborating a tool that facilitates the development of Web services with Quality of Service (QoS) attributes such as security, performance and robustness [23]. As code contracts can be viewed as a sophisticated instrument to produce more robust and fault-tolerant software components, an interesting question is how to embed design by contracts technologies into the more general setting of tool support for arbitrary (QoS) attributes.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 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.

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

[3] Extensible Markup Language (XML) 1.1. http://www.w3.org/TR/xml11/, last access Jan. 2012.

[4] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl/, last access Jan. 2012.

[5] SOAP Version 1.2. http://www.w3.org/TR/soap/, last access Jan. 2012.

[6] Web Services Policy 1.5 - Framework. http://www.w3.org/TR/ws-policy/, last access Jan. 2012.

[7] Web Services Interoperability Technology (WSIT). https://wsit.dev.java.net, last access Jan. 2012.

[8] Writing rock solid code with Code Contracts. http://blog.hexadecimal.se/2009/3/9, last access Dec. 2011.

[9] D. Esposito and A. Saltarello, *Microsoft .NET: Architecting Applications for the Enterprise*. Microsoft Press, 2009.

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

[11] Web Services Policy 1.5 - Attachment. http://www.w3.org/TR/ws-policy-attach/, last access Jan. 2012.

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

[13] WS-ReliableMessaging 1.2. http://docs.oasis-open.org/ws-rx/wsrm/v1.2/, last access Jan. 2012

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

[15] Common Compiler Infrastructure: Code Model and AST API. http://cciast.codeplex.com/, last access Jan. 2012.

[16] J. Smith, *Inside Windows Communication Foundation*. Microsoft Press, 2007.

[17] B. Hollunder, "Deriving interface contracts for distributed services," in *Proceedings of the Third International Conferences on Advanced Service Computing (Service Computation 2011)*. Xpert Publishing Services, 2011.

[18] OMG, "Object constraint language specification, version 2.2," http://www.omg.org/spec/OCL/2.2, last access Jan. 2012.

[19] N. M. Le, "Contracts for java: A practical framework for contract programming," http://code.google.com/p/cofoja/, last access Jan. 2012.

[20] Java Modeling Language. http://www.jmlspecs.org/, last access Jan. 2012.

[21] 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.

[22] 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.

[23] B. Hollunder, A. Al-Moayed, and A. Wahl, "A tool chain for constructing QoS-aware web services," in *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*. IGI Global, 2011.