# Code Contracts for Windows Communication Foundation (WCF)

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*—Code contracts allow the specification of preconditions, postconditions and invariants for .NET interfaces and classes. Code contracts not only perform constraint checking at runtime, but also provide tools for static code analysis and documentation generation. WCF is another .NET technology supporting the creation and deployment of distributed services such as Web services. Currently, WCF services cannot be equipped with code contracts. Though a combination 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 combines standard technologies such as WSDL and WS-Policy. 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**

## I. INTRODUCTION

Code contracts [1] 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 [2], WSDL [3], SOAP [4], and WS-Policy [5]. The Web Services Interoperability Technology (WSIT) project [6] 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 a `ServiceContract` attribute. To implement 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). WSDL is programming language independent and makes it possible to create client applications written in other programming languages (e.g., 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 bring code contracts to WCF, one may proceed as follows: The methods in a WCF service implementation class are extended with code contracts expressions, i.e., preconditions, postconditions, and object invariants. In fact, the compiler will not produce any errors and will create executable intermediate code. However, the code contracts constraints are completely ignored when WCF generates the WSDL description 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 [7]; however, a generic solution has not been elaborated yet.

This paper presents a novel approach that combines WCF with code contracts. The strategy is as follows. When deploying a WCF service, the code contracts contained in the service implementation class are extracted. Next, code contracts constraints are represented in a programming language independent manner with WS-Policy [5]. The WS-Policy description will be attached to the service's WSDL. On service consumer side, the generation of the proxy classes is enhanced by including the code contracts expressions, which are extracted from the WSDL/WS-Policy file.

The approach has the following features:

- It combines standard technologies such as WSDL and WS-Policy to bring code contracts to WCF.
- The approach is transparent from a WCF service de-

 ISBN: 978-1-61208-105-2 

velopment point of view. There are no special activities required.
- Code contracts are 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.

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, the client side proxy generation will be addressed. An implementation strategy (proof of concept) will be given in Section VII. The paper will conclude with a summary and directions for future work.

## 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 [1] 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`. 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.Contract;

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

Definition of a precondition for `squareRoot`.

The `Contract.Requires` statement defines a precondition. There is an analogous method `Contract.Ensures` that can be used to specify postconditions.

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`.

The above sample shows how preconditions can be expressed for *classes*. As a method in an *interface* is described only by its signature and cannot have a body, code contracts foresee a simple trick to encode constraints for interface methods. The required constraints are specified in another 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 `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 (full checking) and off (all `Contract` methods are ignored); it is also possible to check only selected code contracts constraints such as preconditions.

### B. Windows Communication Foundation

According to [8], "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.

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, 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 WSDL file one will find a couple of *policy* entries. These entries add further information to the service such as security requirements.

With the help of the WS-Policy specification [5], policies can be expressed in an interoperable manner. 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.

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

WS-Policy attachment.

In the example, a WS-Policy description is attached to the `squareRoot` service via the `PolicyReference` element. The policy states that the body of the SOAP request must be encrypted. Note that the policy is part of the WSDL interface of the service. Hence, if a client does not encrypt the message body, the server would reject the request.

## 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.Contract;

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

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

WCF service with code contracts.

We define a WCF service interface as usual. The code contracts for the service are encoded in the implementation class of the service.

This WCF service implementation can be successfully compiled and deployed. However, the generated WSDL description does not include any information about code contracts. In other words, code contracts 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 are not considered when generating the proxy classes.
2) Clients of the WCF service are not aware of any code contracts. Hence, code contracts support such as static analysis and runtime checking is not available on client side.

Next we will elaborate a concept that resolves these deficits.

## IV. Code Contracts and WCF: the Concept

We observe that a WCF service implementation class can use the methods of the `Contract` class according to the code contracts programming model (see Section II-A). When deploying the service, the following additional activities will be performed:

- 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).
- The resulting WS-Policy description is included into the WSDL interface of the WCF service.

 ISBN: 978-1-61208-105-2 

In order to exploit code contracts contained in WSDL on client side, we will enhance the generated proxy classes. This is achieved by two activities:

- Extraction of the code contracts expressions contained in the WSDL description.
- Creation of corresponding `Contract` method calls and integration into the proxy classes.

Before we will discuss each of these steps in more detail, we give some remarks. From a service development point of view, the approach is transparent. One can apply the standard programming models both for WCF and code contracts. The enhanced deployment infrastructure has the responsibility to realize the above mentioned activities. Secondly, code contracts imposed on WCF services are also available for client technologies other than .NET. Finally, due to enhanced proxy generation, code contracts tool support is available for .NET clients. Again, this enhancement is transparent for the (client) developer.

## 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 = "name"
                   type = "xs:anyURI"/>
    <xsd:attribute name = "context"
                   type = "xs:anyURI"
                   use  = "required"/>
  </xsd:complexType>
</xsd:schema>
```

XML schema for `CodeContractsAssertion`.

A `CodeContractsAssertion` has two attributes: `name` and `context`. 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 the service as specified in the `binding` section of the WSDL.

The body of `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. The expressions contained in the `requires` and `ensures` elements typically refer to parameter names of the service, which are also part of the WSDL. 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.

Observe that code contracts expressions should only be imposed on parameters that are visible at WCF service interface level, and hence are meaningful to the client developer.

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 service `squareRoot` from the previous section.

```
<definitions name="Service1">
  <Policy wsu:Id="CCPolicy">
    <ExactlyOne>
      <All>
        <CodeContractsAssertion
          name="squareRootAssertion"
          context=
            "IService.squareRoot(System.Double)">
          <requires>d >= 0</requires>
        </CodeContractsAssertion>
      </All>
    </ExactlyOne>
  </Policy>
  ...
  <binding name="IService" type="IService">
    <wsp:PolicyReference URI="#CCPolicy"/>
    <operation name="squareRoot"> ... </operation>
  </binding>
</definitions>
```

Code contracts policy.

Before we will describe in Section VII how to create and attach policies for code contracts during the deployment process, we first take a look at the service consumer side.

## VI. Code Contracts on Client Side

On client side, a WSDL description is compiled into proxy classes of a concrete programming language. The tool `svcutil.exe`, provided by WCF, takes a URL of a WSDL description and creates C# proxy classes. To be precise, a C# interface is generated that defines the available services, and a C# class that implements the interface. This class is instantiated by the client application to invoke a WCF service.

The standard version of `svcutil.exe` does not take into account custom WS-Policy descriptions such as code contracts policies. Hence, the generated proxy classes do not contain any code contracts expressions.

In order to include code contracts into proxy classes, one can proceed as follows. One can either modify the generated client proxy classes by incorporating the required `Contract` methods calls. For object invariants new methods will be added. Alternatively, an additional class can be created that

 ISBN: 978-1-61208-105-2 

contains only the code contracts expressions. This class will be linked via the `ContractClassFor` attribute to the proxy interface.

From a client developer point of view, the enhanced proxy classes bring the following advantages. First, a static analysis of the code contracts can be performed, which helps detecting invalid invocations of the WCF service 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. Proof of Concept

### A. Code Contracts Extraction

Given a WCF service implementation, we need some mechanism to obtain its preconditions, postconditions and invariants. Recently, API functions have been published to access code contracts expressions. These functions are part of the *Common Compiler Infrastructure* project [9]. 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; the *key* is the full qualified name of the method and the *value* is a list of strings each representing a code contracts expression. Each expression starts either with `pre:`, `post:`, or `inv:` to indicate its type.

The function makes use of 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 WS-Policy. The XML schema for `CodeContractsAssertion` has been described in Section V.

This transformation is realized as follows: It takes the code contracts dictionary from the previous step and iterates over the keys (i.e., methods with code contracts). For each key, a corresponding `CodeContractsAssertion` is created. A single `CodeContractsAssertion` may contain several expressions. As each expression string starts with `pre:`, `post:`, or `inv:`, it is clear which of the elements `requires`, `ensures` and `invariant` are to be created in the assertion.

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. 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 `CCPolicyExporter` is created. `CCPolicyExporter` has `BindingElement` as super class and implements the `ExportPolicy` method, which contains the specific logic for creating code contracts policies.

The following figure visualizes the class layout.

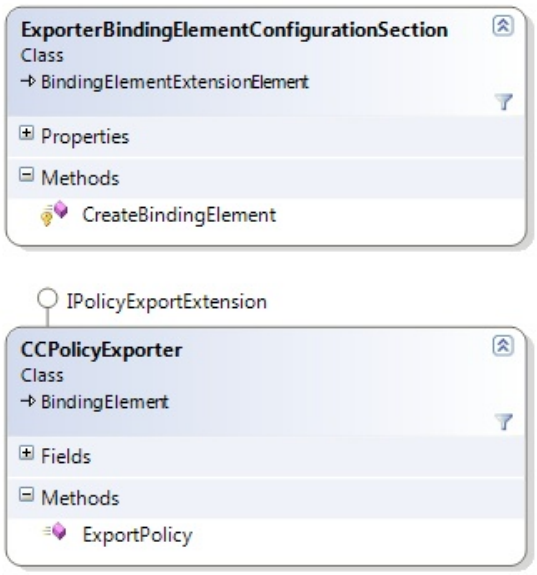


Figure 1. Class diagram for WS-Policy creation.

In our case, the `ExportPolicy` method creates the `CodeContractsAssertion`s as described in the previous step. The result of this activity is an enriched WSDL description as shown in Section V.

To use the custom binding, the configuration file of the WCF service must be adapted as follows:

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 element `customBinding` 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 deployment of the service, WCF now uses the custom binding. As a result, the generated WSDL file will include the code contracts policy.

 ISBN: 978-1-61208-105-2 

### 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 our case, we extend the endpoint definition by a `policyImporters` element that refers to the class `CCPolicyImporter`.

We have realized this class in the following way. It implements the WCF interface `IPolicyImporterExtension`, which declares the `ImportPolicy` method. `CCPolicyImporter` implements this method in such a way that code contracts policies referenced in the WSDL are imported. During the import, a code contracts dictionary (similar to the one on server side as described in Section VII-A) is constructed. 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 contains a contract class for the proxy interface;
2) Link the generated contract class to the proxy interface.

The contract class will contain all constraints that are found in the code contracts policy. In the proof of concept, we construct the contract class as follows. Via the reflection interface we iterate on the methods of the proxy interface. For each method contained in the code contracts dictionary we create a method body with the corresponding `Contract.Requires`, `Contract.Ensures`, and `Contract.Invariant` statements. Otherwise, if the code contracts policy does not contain any constraints for the method at hand, an empty method body is generated, which means that no additional constraint is imposed to the method.

Next, we link the generated contract class to the proxy interface. This is achieved by equipping the contract class with the `ContractClassFor(typeof(...))` attribute. Finally, the proxy interface generated by `svcutil.exe` will be extended by an analogous `ContractClass(typeof(...))` attribute. This completes the generation and linkage of the code contract class with the proxy interface.

We have developed a simple tool `ccsvcutil.exe` that wraps `svcutil.exe` as described. Thus, a client developer uses `ccsvcutil.exe` to generate the client proxy infrastructure. It should be noted that the code contracts processing is transparent for the client developer – with the exception that the code contracts runtime environment and tools are now available on client side.

### F. Object Invariants

In WCF, so-called *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 that are marked with `DataMember`. To impose object invariants on data contracts one may introduce a method annotated with `ContractInvariantMethod` that contains `Contract.Invariant` statements (cf. Section II-A).

As an example consider a data contract `AddressData` with members such as street, zip and city and an object invariant method that, for example, controls the zip format. Suppose a WCF service `ChangeAddress` takes an instance of `AddressData` together with a customer id as parameters. Because `AddressData` is part of the service's signature, it has a representation as `complexType` in the WSDL. Therefore, `svcutil.exe` will generate a corresponding C# class `AddressData`, which is used by the service consumer to construct address instances. We note that this class contains only a default constructor to create "empty" instances; their members can be accessed via public getters and setters.

In order to invoke the `ChangeAddress` service, a client may proceed as follows: i) create an empty instance of `AddressData`, ii) set the specific values of the members with the public setters, and iii) pass the instance together with the customer id to the service. Unfortunately, the code contracts infrastructure on client side will report an error after the first step. This is due to the fact that the empty zip member contains an invalid value, which is recognized by the object invariant.

To overcome this problem, one needs on client side a public constructor that takes all relevant address 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. Thus, we propose that the code contracts aware version `ccsvcutil.exe` should generate for each user defined data contract a corresponding public constructor.

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 [1], 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

 ISBN: 978-1-61208-105-2 

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.

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

### H. Service Provider Side 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 custom binding as described in Section VII-C.
3) Deployment of the WCF service by launching the project.
4) Creation of a WCF client project as usual.
5) Invocation of `ccsvcutil.exe` to generate the enhanced proxy classes.
6) Usage of the code contracts infrastructure on client side.

## VIII. Summary and Future Work

In this paper we have elaborated a concept that combines WCF with code contracts. As a consequence, WCF application developers – both on server and client side – can now profit from the additional expressive power of code contracts including runtime and tool support. It has been stressed elsewhere that there does not exist a generic solution yet.

Our novel approach exploits well-known standards such as WSDL and WS-Policy. We have described how to transform code contracts expressions contained in the WCF service into a programming language independent representation. This representation will be used to generate an enhanced client proxy infrastructure, thus allowing to evaluate the WCF service's code contracts already on client side.

We see several areas for future work. One direction is concerned with a precise definition of "WCF code contracts expressions." When defining code contracts for WCF services, only those variables should be referred that are visible to the service consumer. While service parameters are public and hence meaningful for a service consumer, it is not useful for the client when members of the service implementation class are included into the created code contracts assertions. Therefore, rules should be defined that i) characterize valid expressions (similar to the ones presented in Section 5 on contract extraction in [10]) and ii) translate the code contracts statements into corresponding WS-Policy assertions embedded into the service's WSDL description.

Additional tool support for WCF code contracts is another topic. We have shown how a custom binding can be defined such that code contracts expressions are exported to (resp. imported from) the WSDL. For a WCF developer, it would be helpful to have a specific "WCF code contracts" project type for VisualStudio that automatically introduces the required elements in the WCF configuration files.

This work is concerned with making code contracts available for a WCF client environment. Another interesting question is how a WCF service consumer developed with an alternative technology such as Java (see e.g., [11]) can process the code contracts expressions.

## 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] Microsoft Corporation, "Code contracts user manual," 2009.

[2] Extensible Markup Language (XML) 1.1. http://www.w3.org/TR/xml11/.

[3] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl/.

[4] SOAP Version 1.2. http://www.w3.org/TR/soap/.

[5] Web Services Policy 1.5 - Framework. http://www.w3.org/TR/ws-policy/.

[6] Web Services Interoperability Technology (WSIT). https://wsit.dev.java.net.

[7] Writing rock solid code with Code Contracts. http://blog.hexadecimal.se/2009/3/9, last access on 08/24/2010.

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

[9] Common Compiler Infrastructure: Code Model and AST API. http://cciast.codeplex.com/, last access on 08/24/2010.

[10] M. Barnett, M. Fahndrich, and F. Logozzo, "Embedded contract languages," in *ACM SAC - OOPS*. Association for Computing Machinery, 2010.

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

 ISBN: 978-1-61208-105-2 