Code Contracts User Manual

Mike Barnett

Programming Languages and Methods

Microsoft Research

# Code Contracts Library Overview

Contracts allow you to express preconditions, postconditions & object invariants in your code for runtime checking, static analysis, and documentation. This document covers how to author them in your code.

**Microsoft Internal Use Only**

The location of the contract library depends on your environment. The current situation is:

You are writing code within System.Core.dll inside of Orcas. Then it is defined as a set of internal classes within that assembly.

You are writing code with the mscorlib that will ship as part of CLR v4. Then it is defined as a set of public classes within mscorlib.

Otherwise, the library is packaged in a separate assembly: Microsoft.Contracts.dll.

All of the contract methods are static methods defined in the CodeContract class contained in the System.Diagnostics.Contracts namespace. Several members of the class are conditionally compiled, meaning the compiler only calls these methods when a special symbol, the *full-contract symbol*, is defined. That symbol is CONTRACTS\_FULL, which you can define using a /d: option to most compilers. Also, the precondition methods (Section 2.1) are defined when a special symbol, the *precondition symbol*, is defined. That symbol is CONTRACTS\_PRECONDITIONS.

The expectation is that debug and checked builds of products will define the full-contract symbol (as well as DEBUG) and use a binary rewriter to produce a binary with runtime checking for preconditions, postconditions, and object invariants. For at least the .NET Framework v4, the expectation is retail and “free” builds are expected to define neither symbol, and will not use a binary rewriter.

# Contract Methods

## Preconditions

Preconditions are expressed using CodeContract.Requires(). They are contracts on the state of a method when it is invoked. They generally are used for parameter validation. All members mentioned in preconditions must be at least as accessible as the method itself.

CodeContract.Requires( x != null );

This version is conditionally defined on either the full-contract symbol or the precondition symbol. When either of those symbols are defined during compilation, then the precondition is present in the IL of that build.

The runtime behavior of a precondition depends on several factors. If the binary rewriter is not used, (which implies that only the precondition symbol and not the full-contract symbol was defined so that only preconditions are being checked and not any postconditions and invariants), then if the condition does not hold Debug.Assert(false) is called and then Environment.FailFast(). When the condition does hold, it does nothing. When the binary rewriter is used and the rewriter methods option is not used, then the behavior is the same. When the rewriter methods option is used, then the behavior depends on the method supplied to the rewriter (Section 5.1).

To make sure that a precondition is defined in all build configurations, there is an unconditionally defined version:

CodeContract.RequiresAlways( x != null );

### Backward Compatibility

For backward compatibility, there are cases where you want to specify the exact exception that should be thrown or where existing code is not allowed to be modified. In that case, an if-then statement is recognized as a precondition:

if ( x == null ) throw new …;

CodeContract.EndContractBlock(); // All previous ‘if’ checks are preconditions

Note that the polarity of the precondition has to be reversed. This form of precondition is highly restricted: it must be written as above, i.e., there are no else-clauses and the body of the then-clause must be a single throw statement. If the only contracts in a method are of this form, then the EndContractBlock marker (Section ) must be used.

The if-test is subject to both the purity (Section ) and visibility rules (Section ), but the throw-expression is subject only to the purity rules. However, the type of the exception thrown must be as visible as the method in which the contract occurs.

## Postconditions

Postconditions are contracts on the state of a method when it terminates. Their runtime behavior is determined in the same way as preconditions. Postconditions are conditionally defined on the full-contract symbol.

Unlike preconditions, members with less visibility may be mentioned in a postcondition. A client may not be able to understand or make use of some of the information expressed by a postcondition using private state, but it doesn’t affect the client’s ability to use the API correctly. For preconditions, it would negatively affect a client: they would be unable to guarantee that they are not violating a precondition.

### Normal postconditions

Normal postconditions are expressed using CodeContract.Ensures(). They express a condition that must hold on normal termination of the method.

CodeContract.Ensures( this.F > 0 );

### Exceptional postconditions

Postconditions that should hold on exceptional return paths from a method are specified using CodeContract.EnsuresOnThrow.

CodeContract.EnsuresOnThrow<T>( this.F > 0 );

The argument is the condition that must hold whenever an exception is thrown that is a subtype of T. There are many exception types that would be difficult to use in an exceptional postcondition. For instance, using the type Exception for T would require the method to guarantee the condition no matter what type of exception is thrown, even if it is a stack overflow or other impossible-to-control exception. It is recommended to use exceptional postconditions only for those exceptions that a caller should expect as part of the API, e.g., a socket-closed exception on a method in an API for sending messages across a socket.

### Special Methods within Postconditions

There are several special methods that may be used only within postconditions.

#### Method Return Values

Within a postcondition the method’s return value can be referred to as CodeContract.Result<T>(), where T is replaced with the return type of the method. When the compiler is unable to infer the type it must be explicitly given. For instance, the C# compiler is unable to infer types for methods that do not take any arguments.

CodeContract.Ensures(0 < CodeContract.Result<int>());

Methods with a return type of void cannot refer to CodeContract.Result<T>() within their postconditions.

#### Prestate Values (OldVaule)

Within a postcondition, an *old expression* refers to the value of an expression from the pre-state. It uses the method CodeContract.OldValue<T>(e), where T is the type of e. The generic type argument may be omitted whenever the compiler is able to infer its type. For instance, the C# compiler is able to infer the type since it takes an argument. There are several restrictions on what can occur in e and also in which contexts an old expression may appear. An old expression may appear only within a postcondition. An old expression cannot contain another old expression. A general rule is that an old expression must refer to a value that existed in the method's pre-state. Here are several instances of that rule.

1. The method's return value cannot be referred to in an old expression:

CodeContract.OldValue(CodeContract.Result<int>() + x) // ERROR

1. Out parameters cannot be referred to within an old expression.
2. An old expression cannot depend on the bound variable of  a quantifier (Section 2.7) **if** the range of the quantifier depends on the return value of the method:

CodeContract.ForAll(0,CodeContract.Result<int>(),  
            i => CodeContract.OldValue(xs[i]) > 3 ); // ERROR

1. An old expression cannot refer to the parameter of the anonymous delegate within a CodeContract.ForAll or CodeContract.Exists unless it is used as an indexer or argument to a method call:

CodeContract.ForAll(0,xs.Length,  
                i => CodeContract.OldValue(xs[i]) > 3 ); // OK

CodeContract.ForAll(0,xs.Length,  
                i => CodeContract.OldValue(i) > 3 ); // ERROR

1. An old expression cannot occur in the body of an anonymous delegate if the value of the old expression depends on any of the parameters of the anonymous delegate (unless the anonymous delegate is an argument to the methods CodeContract.ForAll or CodeContract.Exists):

Foo( ... (T t) => CodeContract.OldValue(... t ...) ... ); // ERROR

#### Out Parameters

Because the contracts appear before the body of the method, most compilers do not allow references to out parameters in postconditions. To get around this, use the method CodeContract.ValueAtReturn<T>(out T t) which will make the compiler think that the parameter is defined.

public void OutParam(out int x){  
  CodeContract.Ensures(CodeContract.ValueAtReturn(out x) == 3);  
  x = 3;  
}

As with OldValue, the generic type argument may be omitted whenever the compiler is able to infer its type. The binary rewriter will replace the method call with the argument to the call. The method may appear only within postconditions. The argument to the method must be an out parameter; no expressions are allowed.

**Note:** Currently the rewriter does not check to make sure that out parameters are initialized properly disregarding their mention in the postcondition. Thus, in the above example, if the line after the contract had used the value of x instead of assigning to it (and CodeContract.Ensures is defined), a compiler would not issue the error that it should. However, on a build where the full-contract symbol is *not* defined, then the compiler will issue an error.

## Object Invariants

Object invariants are conditions that should hold on each instance of a class whenever that object is visible to a client. They express the conditions under which the object is in a “good” state.

instance

Individual invariants within the invariant method are specified using CodeContract.Invariant:

CodeContract.Invariant( this.x > this.y );

Invariants are conditionally defined on the full-contract symbol. Their runtime behavior is determined in the same way as preconditions and postconditions.

During runtime checking, invariants are checked at the end of each public method. If an invariant mentions a public method in the same class, then the invariant check that would normally happen at the end of that public method is disabled and checked only at the end of the outermost method call to that class. This also happens if the class is re-entered because of a call to a method on another class.

## Assert

Assertions are specified using CodeContract.Assert. They are used to state a condition that must hold at that program point.

CodeContract.Assert( this.private\_field > 0 );  
CodeContract.Assert( this.x == 3, “Why isn’t the value of x 3?” );

The members mentioned in an assertion may have less accessibility than the method in which they occur. Assertions are conditionally defined and so exist in the build only when the full-contract symbol or the DEBUG symbol is defined. There is also another overload that takes a string as a second parameter and uses that as the argument to the exception constructor when the condition fails to hold.

At runtime failing assertions just call Debug.Assert while a successful assertion does nothing.

## Assume

Assumptions are specified using CodeContract.Assume.

CodeContract.Assume( this.private\_field > 0 );  
CodeContract.Assume( this.x == 3, “Static checker couldn’t prove this” );

The members mentioned in an assumption may have less accessibility than the method in which they occur.

For runtime checking their effect is exactly the same as an assertion. However, for static verification an assumption is something that the verification will just add to the facts that it knows about the program at that program point. You should use this for conditions that you believe to hold at a program point, but that the static verification is unable to prove due to its limitations. It is probably best to start with just assertions and then change them as needed while verifying the code.

Assumptions are conditionally defined and so exist in the build only when the full-contract symbol or the DEBUG symbol is defined.

At runtime failing assumptions call Debug.Assert(false). A successful assumption does nothing.

## EndContractBlock

When a method’s contracts contain only preconditions in the if-then-throw form (Section 2.1.1), this method is used to mark the end of the contract section. All if-then-throw statements before EndContractBlock are assumed to be preconditions. It is also defined conditionally on either the full-contract symbol or the precondition symbol.

if ( x == null )

throw new ArgumentNullException(“x”);

if ( y < 0 )

throw new ArgumentOutOfRangeException(…);

Contract.EndContractBlock( );

## Quantifiers

Limited support is available for quantifiers within contracts. We support only those forms which we are able to translate into quantifiers as understood by the static verification. Currently, this means that all quantifications must be over an integer range. Also, the “body” of each quantification must be an expression, i.e., not contain any loops or assignment statements.

### ForAll

Universal quantifications are written using CodeContract.ForAll. There are two overloads, both of which are static methods. The first form takes two parameters: a collection and a *predicate*. A predicate is a unary method that returns a boolean. The predicate is applied to each element in the collection. If it returns false on any element, ForAll stops iterating and returns false. If the predicate returns true on all of the elements in the collection, then ForAll returns true. Here's an example of a contract that says that all elements in the array must not be null:

public int[] Foo<T>(T[] xs){  
    CodeContract.Requires(CodeContract.ForAll(xs,(T x) => x != null) );

Note that this overload of ForAll enumerates a collection. It is generic; it is parameterized by the type of elements in the collection. (That is, the first argument is of type IEnumerable<T> and the predicate must take an argument of type T.) The other overload takes three parameters: a lower bound, an upper bound, and a *predicate*. It iterates over a range within a collection, and is **not** generic: the predicate must take an integer as its argument. For example, the following method has a precondition that all values in the array must be positive:

public int[] Foo(String[] xs){  
 CodeContract.Requires(  
 CodeContract.ForAll(0, xs.Length, index => xs[index] != null));

### Exists

Existential quantifications are written using CodeContract.Exists. There are the same two overloads with the same parameters as CodeContract.ForAll.

## Interface Contracts

Since the C# compiler will not let you put method bodies within an interface, to write an interface contract you must create a separate helper class to hold them. The interface and the *contract class* are linked via a pair of attributes (Section ).

[ContractClass(typeof(IFoo\_Contract))]  
interface IFoo {  
  int Count { get; }  
  void Put(int value);  
}  
[ContractClassFor(typeof(IFoo))]  
sealed class IFoo\_Contract : IFoo {  
  int Count {  
    get {  
      CodeContract.Ensures( 0 <= CodeContract.Result<int>() );  
      return CodeContract.Result<int>(); // dummy return  
    }

  }  
  void Put(int value){  
    CodeContract.Requires( 0 <= value );  
  }  
}

Note how you can use CodeContract.Result in the body of the method to avoid having to produce some dummy return value (admittedly not hard for a return type of int...). Of course, you can return any type-correct value (e.g., null for reference types, or default(T) in code that is generic for a type parameter T). The contract class **must** implement the interface.

# Contract Attributes

## ContractClass and ContractClassFor

For interfaces and abstract types, contracts are written in a separate class (Section 2.8). This attribute is added to the interface (or abstract type) to point to that separate class.

[ContractClass(typeof(ContractForJ))]  
interface J { … }

This attribute is used to provide the backwards link from a class that holds the contracts for an interface or abstract type to the interface or abstract type (Section 2.8).

[ContractClassFor(typeof(J))]  
class ContractForJ : J { … }

## Mutability

This is an experimental attribute and not fully supported. (Okay, it isn’t supported at all yet.) It indicates how “constant” a value of a particular type is. It can be applied to classes and structures. (Not interfaces?) The values it can take on are defined in the enumeration Mutability.

public enum Mutability {

Unspecified,

Immutable, // read-only after construction, except for lazy initialization and caches  
 // Do we need a “deeply immutable” value?

Mutable,

HasInitializationPhase, // read-only after some point  
 // Do we need a value for mutable types with read-only wrapper subclasses?

}

## ContractInvariantMethod

This attribute is used to mark the method that represents the object invariant. The method marked with this attribute must be a nullary method that returns void and contains nothing other than calls to CodeContract.Invariant (Section 2.3).

## Pure

Methods are allowed in contracts only if they are *pure*. See Section 4.2 for an explanation of when you need to use this attribute.

## RuntimeContracts

This assembly-level attribute is added to assemblies by the binary rewriter to flag that an assembly has already been rewritten.

## ContractPublicPropertyName

This attribute is used on a field to allow it to be used in a method contract where the method is more visible than the field, e.g., a private field and a public method. This is used to be exempt from the Visibility rules (Section 4.3).

[ContractPublicPropertyName(“PublicProperty”)]  
private int \_internal;  
public int PublicProperty { get { … } }

The argument to the attribute is the string name of a corresponding property. The type of the field should be assignable to the type of the corresponding property, but that is currently not checked. However, an error will result if or property can be found that has the indicated name.

## ContractVerification

This attribute is used to instruct downstream tools whether to assume the correctness of the assembly, type or member it decorates without performing any verification or not. You can use [ContractVerification(false)] to explicitly mark an assembly, type or member as one to not have verification performed on it. The most specific element found (member, type, parent type, then assembly) takes precedence. (That is a useful feature for turning off verification for an assembly but enabling it for one particular type in the assembly.) When it is applied to a type, then it applies to all members of the type, including nested types. When it is applied to an assembly, then it applies to all types and members of the assembly. When it is applied to a property, then it applies to both the getter and setter.

# Usage Guidelines

There are many context conditions for contracts, in addition to the restrictions already mentioned. Not all of the following rules are currently being checked by Foxtrot, but in the future they will be.

## Contract Ordering

Method contracts should be written with the different elements ordered as follows:

|  |  |
| --- | --- |
| If-then-throw | Backward-compatible public preconditions |
| Requires | All public preconditions |
| Ensures | All public (normal) postconditions |
| EnsuresOnThrow or Throws | All public exceptional postconditions |
| EndContractBlock | If you were using if-then-throw and didn’t have any postconditions, use EndContractBlock to indicate all previous if checks are preconditions. |
| Assert | All private preconditions |

## Purity

All methods called within a contract must be *pure*: that is, they must not update any pre-existing state. (A pure method is allowed to modify objects that have been created after entry into the pure method.) Foxtrot currently assumes the following things are pure:

* Methods marked [Pure] (If a type is marked [Pure], then that applies to all of its methods.) The pure attribute is defined in the contract library. (Section 3.4)
* Property getters
* Operators (static methods whose names start with “op\_”, have one or two parameters and a non-void return type)
* Any method whose fully qualified name begins with “System.CodeContracts.”, “System.String.”, “System.IO.Path”, or “System.Type.”

In the future, there will be a purity checker that will enforce these assumptions.

## Visibility

All members mentioned in a contract must be at least as visible as the method in which they appear. For instance, a private field cannot be mentioned in a precondition for a public method: clients wouldn’t be able to validate such a contract before they call the method. However, if the field is marked with the ContractPublicNameProperty attribute (Section3.7), then it is exempt from these rules.

## Special Method Usage

Special methods such as CodeContract.Result<T> can only appear in Ensures contracts and type T must agree with the method return type.

# Runtime Checking

To enable the contracts at runtime, the original assembly must be rewritten. This performs several tasks: postconditions are moved to the end of the method body, method return values are substituted for occurrences of CodeContract.Result<T>() and pre-state values are substituted for occurrences of CodeContract.OldValueOf<T>(). In addition, contract inheritance is enforced.

## Runtime Contract Behavior

The runtime behavior of a contract can be configured at the time the binary rewriter is run. The default behavior is that the binary rewriter defines a class within the rewritten assembly containing the runtime methods for preconditions, postconditions, and invariants. Their default behavior is to call Debug.Assert(false) followed by Environment.FailFast() if the condition is false and to do nothing if the condition is true. (Note that this induces an external reference to System.dll if the assembly being rewritten did not already.)

Runtime behavior of assertions and assumptions are already defined in the original Contract class. However, their behavior can be modified with the rewriter.

By using the /rewriterMethods option to the binary rewriter, arbitrary behaviors can be provided. An assembly and class are specified using the option. The class may contain a definition for any of ten methods: two each for assertions, assumptions, preconditions, postconditions, and invariants. They must all be static methods that return void. They must be named “Assert”, “Assume”, “Requires”, “Ensures”, and “Invariant”, respectively. The first of each pair must take a single boolean argument and the second of each pair must take two arguments, the first of type boolean and the second of type string. If the appropriate method is not found, then the default behavior is used for that method.

Here’s a sample implementation that throws exceptions instead of the default behavior.

namespace MyNameSpace {

public class RequiresException : Exception { … }

public class EnsuresException : Exception { … }

public class InvariantException : Exception { … }

public class AssertException : Exception { … }

public class AssumeException : Exception { … }

public static class RuntimeFailureMethods {

    [DebuggerStepThrough]

    public static void Requires(bool cond, string msg)

    {

if (!cond) throw new RequiresException(msg);

    }

    [DebuggerStepThrough]

    public static void Ensures(bool cond, string msg)

    {

if (!cond) throw new EnsuresException(msg);

    }

    [DebuggerStepThrough]

    public static void Assert(bool cond, string msg)

    {

if (!cond) throw new AssertException(msg);

    }

    [DebuggerStepThrough]

    public static void Assume(bool cond, string msg)

    {

if (!cond) throw new AssumeException(msg);

    }

    [DebuggerStepThrough]

    public static void Invariant(bool cond, string msg)

    {

if (!cond) throw new InvariantException(msg);

    }

}

}

Of course, when this option is used, the specified assembly must be able to be found by the binary rewriter and then deployed with the rewritten assembly when it is executed.

## Binary Rewriter command line options

To rewrite an assembly, use the command-line tool Foxtrot.exe with the /rewrite option.

Usage: Foxtrot [/verify[+|-]] [/rewrite[+|-]] [/debug[+|-]] [/shortBranches[+|-]] [/out:<string>] [/contracts:<string>]\* [/libpaths:<string>]\* [/passthrough[+|-]] [/verbose[+|-]] [/warn:<uint>] [/targetplatform:<string>] [/allowRewritten[+|-]] [/pureProps[+|-]] [/rewriterMethods:<string>] <assembly> @<file>

|  |  |
| --- | --- |
| [/verify[+|-]] | Statically verifies that the given assembly has no contract violations. Default value:'-' (Short form: /v) |
| [/rewrite[+|-]] | Rewrites the given assembly to insert runtime contract checks. Default value:'-' (Short form: /r) |
| [/debug[+|-]] | Use debug information. Default value:'+' (Short form: /d) |
| [/shortBranches[+|-]] | Preserve short branches in the rewritten assembly. Default value:'+' (Short form: /s) |
| [/out:<string>] | Output path for the rewritten assembly. Default value:'same' (Short form: /o) |
| [/contracts:<string>]\* | Out-of-band contract assemblies. (Short form: /c) |
| [/libpaths:<string>]\* | Additional paths to search for referenced assemblies. (Short form: /l) |
| [/verbose[+|-]] | Print out statistics on each method processed. Default value:'-' |
| [/warn:<uint>] | Set warning level (0-4) Default value:'1' (Short form: /w) |
| [/targetpltform:<string>] | Path to alternate set of framework assemblies. Default value: '' (Short form: /tp) |
| [/allowRewritten[+|-]] | Silently do nothing if the input assembly has already been rewritten. Default value: '-' (Short form: /ar) |
| [/pureProps[+|-]] | Determines whether property getters (and operators) are assumed to be pure or not. Default value: '+' (Short form: /pp) |
| [/rewriterMethods:<string>] | Alternative methods to use for checking contracts at runtime. Syntax: /rw:<assembly name>,<class name>. (Short form: /rw) |
| <assembly> | Assembly to process. |
| @<file> | Read response file for more options. |

## Troubleshooting

When using the binary rewriter, you must use it only after all of the assemblies have been built. In other words, you cannot run it as a post-build step on (the project that builds) assembly A if you are also going to run it as a post-build step on assembly B if B references A. That is because while the rewriter is working on assembly B, if it needs to inherit any contracts from assembly A, then the rewriter must first extract those contracts from A. It cannot extract contracts from a rewritten assembly and so you will get an error message. **Solution:** Either run the rewriter after all of the projects have been built, or else use the /out option to name the rewritten assemblies with a different name and then after all of the assemblies have been built, rename/move/copy the rewritten assemblies to stomp over the original output from the C# compiler.

# Static Verification

Static verification is available, but described in a separate document.