Code Contracts for .NET: Better Contracts Make Better Neighbors

Mike Barnett, Manuel Fähndrich, and Francesco Logozzo
Microsoft Research, One Microsoft Way, Redmond, WA, 98052-6399, USA
{mbarnett, maf, logozzo}@microsoft.com

Abstract

Code Contracts for .NET provides a complete infrastructure and toolset for describing component API specifications. Using a library-based approach, all .NET languages can express method preconditions and postconditions, even for interfaces and abstract types. Exisiting tools provide runtime checking, static checking, automatic documentation generation, and editor enhancements. There is open-source support for manipulating contracts in an easy-to-use object model.

1. Introduction

Components are effective only when it is possible to describe their interfaces in a discoverable manner — by both humans and machines. For humans, it is crucial to have support for programming against a component: knowing what methods to call and how to call them. For machines, it is crucial to be able to verify — either at runtime or statically or a combination of the two — whether a component meets its specification. Almost all existing languages and developer environments (IDEs) provide such support, but only at the level of method names and the types of method parameters. These API (Application Programming Interface) specifications are the boundaries of each component. Such "fences" make for good neighbors: programmers use the editor support when writing their programs and compilers and linkers use the type signatures to prevent errors.

We propose to increase the expressivity of specifications to include more than just type information. Over the past several years we have developed Code Contracts for .NET, a library-based approach for describing method preconditions and postconditions and also object invariants. We have created tools that make use of the contracts, for humans and for machines:

- Runtime checking
- Static verification
- Automatic documentation generation
- Editor enhancements

The tools have been freely available since February 2009 and have been downloaded more than 20,000 times. Part of the infrastructure have already shipped as an integral part of .NET 4.0 and other parts are available for open-source development.

In this paper, we provide an overview of the overall project, the contract "language", and the existing tools. We hope to encourage a community using our shared infrastructure to develop further tools in order to provide better support for components in .NET.

2. Contract Expressivity

Contracts are written as method calls to a set of static methods defined in the class `Contract`, which is defined in the namespace `System.Diagnostics.Contracts`. The namespace and class were introduced in the .NET Base Class Library (BCL) in version 4.0. Support for previous versions of .NET are provided through a separate library comprising the contract class methods.

An example using C# is shown in Figure 1. Preconditions and postconditions are expressed as

```
int Increment(int value, string label) {
  Contract.Requires( value > 0 );
  Contract.Requires( label != null );
  Contract.Ensures( Count ==
                    Contract.OldValue(Count) + value );
  Contract.Ensures( Contract.Result<int>()
                    == Contract.OldValue(Count) );
  ...
```

Figure 1. An Increment specification in C#

calls to the static methods `Contract.Requires` and `Contract.Ensures`. Special dummy methods are used to refer to the method's return value as well as referring to the *old* value of an expression, meaning the value of the expression on method entry. Note that the arguments to the contract methods are just written in

ordinary C# syntax. If the program had been written in VB, then the contract methods would stay the same, but their arguments would be the equivalent VB code.

Such a language-agnostic approach has many advantages:

- Developers need not learn a new language for specifications. Predicates are boolean conditions expressed in the source language.
- No new front-ends or compilers are required. Standard compilers directly translate contracts into .NET intermediate language (MSIL). As a benefit, compilers check the syntax and typing of contract conditions, thus avoiding errors in specifications, such as unresolved names, that would arise if the specifications were written in comments or attributes.
- Standard development environments help writing specifications in the same way they help writing other code, via highlighting, intellisense, completion, etc.
- The semantics of contracts is defined by that the semantics of the generated MSIL. The compiled code acts as a persisted format of specifications consumable by a variety of tools.

The language independence extends from the specification language to the tools themselves as they consume the output of each language's compiler.

3. Runtime Contract Checking

In a post-build step, the compiled binaries containing the calls to the contract methods are transformed by having each specification injected at the appropriate program points. For instance, method postconditions are moved to the exit points of each method and calls to Contract.Result are replaced with the return value of the method. At the beginning of each method, arguments to Contract.OldValue are evaluated and stored into locals which then replace those method calls within the postcondition-checking code.

In addition, contracts from supertypes and interfaces are inherited by subtypes and interface implementations. This provides the basis for enforcing behavioral subtyping [2], which is required for modular checking.

4. Static Checking

Our static checker is based on abstract interpretation rather than SMT solvers traditionally used for program verification, in order to automate the generation of loop invariants and strongest postconditions.

Although we use modular verification which, in principle, requires specifications at all method boundaries, we use techniques to infer pre- and postconditions whenever possible.

The existing abstract domains provide an analysis that checks for null dereferences, array indexing, arithmetic overflow, in addition to user-defined general properties.

```
Function Increment(ByVal value As Integer) As Integer
  Contract.Requires( value > 0 )
  Contract.Requires( label IsNot Nothing )
  Contract.Ensures( Count
                    = Contract.OldValue(Count) + value )
  Contract.Ensures( Contract.Result(Of Integer)()
                    = Contract.OldValue(Count) )
  ...
```

Figure 3. An Increment specification in Visual Basic

5 Screenshots

Figure 2 shows the contract user interface in Visual Studio. The UI allows the programmer to enable runtime checking and/or static checking. The integration in the IDE manages all the extra build steps transparently.

As an example of the language-agnostic approach, the same specification as in Figure 1 can be written in Visual Basic as shown in Figure 3.

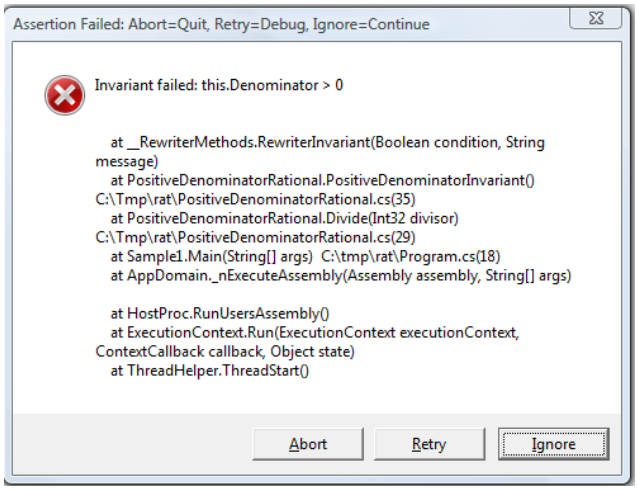

Figure 4. Example of Runtime Contract Failure

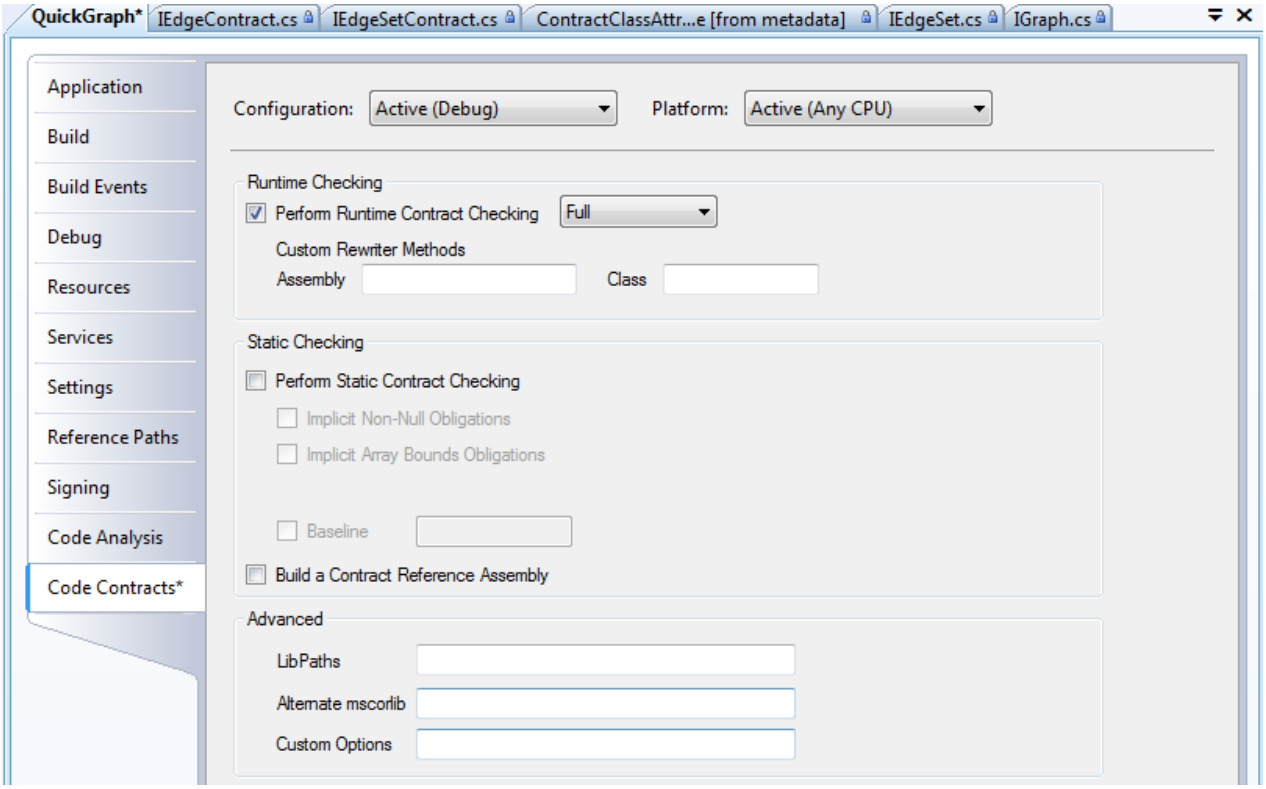

Figure 2. The Code Contact User Interface

Figure 4 shows the default contract failure behavior when runtime checking of contracts is enabled. In this case, an invariant was violated.

Figure 5 shows the output produced by a run of the static contract verifier.

References

[1] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *SIGPLAN Notices*, 40(6):213–223, 2005.

[2] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[3] Michael C. Two, Charlie Poole, Jamie Cansdale, Gary Feldman, James W. Newkirk, Alexei A. Vorontsov and Philip A. Craig. NUnit. http://www.nunit.org/.

[4] Microsoft. Visual Studio Team System, Team Edition for Testers. http://msdn2.microsoft.com/en-us/vsts2008/products/bb933754.aspx.

[5] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153, Prato, Italy, April 2008. Springer.

Error List

0 Errors | 12 Warnings | 0 Messages

	Description	File	Line	Column	Project
⚠ 1	invariant unproven	Rational.cs	22	7	Rational
⚠ 2	location related to previous warning	CodeContracts.Samples.Rati			Rational
⚠ 3	invariant unproven	Rational.cs	22	7	Rational
⚠ 4	location related to previous warning	CodeContracts.Samples.Rati			Rational
⚠ 5	requires unproven	Rational.cs	14	7	Rational
⚠ 6	location related to previous warning	Rational.cs	37	7	Rational
⚠ 7	ensures unproven	Rational.cs	54	7	Rational
⚠ 8	location related to previous warning	Rational.cs	60	5	Rational
⚠ 9	requires unproven	Rational.cs	14	7	Rational
⚠ 10	location related to previous warning	PositiveDenominatorRationa	11	5	Rational
⚠ 11	invariant unproven	PositiveDenominatorRationa	35	7	Rational
⚠ 12	location related to previous warning	PositiveDenominatorRationa	30	5	Rational

Figure 5. Example of Static Checker Output