Managed Contract Usage Scenarios

Mike Barnett (mbarnett) and Manuel Fähndrich (maf)

1 July 2008

# Introduction

This document describes how developers and testers inside and outside of Microsoft leverage contracts and use the associated tools in the simplest and most streamlined fashion.

# Assemblies

The managed contract scenarios provide four flavors of target assemblies: traditional, rewritten, declarative, and reference. This section describes these assemblies, how they are built, their function, and the trade-offs with respect to contracts.

## Traditional Assembly

A traditional assembly is an assembly built without the contract rewriter, i.e., the direct result of a compilation with either C# or VB. Contract use is limited without the contract rewriter in that only pre-conditions (requires) can be part of such assemblies. Furthermore, the form of the requires influences what kind of runtime information is available when a runtime pre-condition failure occurs.

Traditional assemblies are typically built for RETAIL flavors where running the contract rewriter as part of the build is deemed too risky.

As a result of not running the contract rewriter, the symbol FEATURE\_FULL\_CONTRACTS, which enables post-conditions and other contracts *must not be defined* in such builds.

### Legacy requires

Legacy pre-conditions of the form

if (!cond) throw …;

are clearly emitted into the final assembly. Furthermore, this form provides full control over the runtime behavior when such conditions are not met. In order for legacy requires to be recognized by tools in other builds, they must be demarcated by calls to other Contract methods (other Requires, Ensures, or the special EndContract call).

### RequiresInRetail

Contracts of the form

Contract.RequiresInRetail(cond);

are emitted in all build flavors as this method is not conditionally defined. The runtime behavior is limited to invoking Debug.Assert with the message “pre-condition failure”, followed by a failfast. At a runtime failure, no indication beyond “pre-condition failure” is provided, as the failed condition is not available without rewriting.

An overload of the form

Contract.RequiresInRetail(cond, “message”);

allows programmers to provide a specific message on failure. The message should be a literal string rather than a complicated expression, as it will be evaluated even when the pre-condition is satisfied.

### FEATURE\_RUNTIME\_REQUIRES

The conditional symbol FEATURE\_RUNTIME\_REQUIRES causes ordinary pre-conditions of the forms

Contract.Requires(cond);

Contract.Requires(cond, “message”);

to be emitted into the target assembly. The runtime behavior is limited to invoking Debug.Assert with the message “pre-condition failure” or the provided message argument, followed by a failfast. . If provided, the explicit message should be a literal string rather than a complicated expression, as it will be evaluated even when the pre-condition is satisfied.

### DEBUG

The conditional symbol DEBUG enables calls to

Contract.Assert(cond);

Contract.Assert(cond, “message”);

Contract.Assume(cond);

Contract.Assume(cond, “message”);

to be emitted into the target assembly. The runtime behavior is limited to invoking Debug.Assert with the message “pre-condition failure” or the provided message argument, followed by a failfast. If provided, the explicit message should be a literal string rather than a complicated expression, as it will be evaluated even when the condition is satisfied.

## Rewritten Assembly

A rewritten assembly is built by first running a normal compile using any combination of the following conditional symbols:

* FEATURE\_FULL\_CONTRACTS: enables all contract forms to be emitted and runtime checked after rewriting.
* FEATURE\_RUNTIME\_REQUIRES: enables “requires” to be emitted and runtime checked after rewriting
* DEBUG: enables “asserts” and “assumes” to be emitted and runtime checked after rewriting.

After compilation, the contract rewriter must be used on the assembly to produce the final executable assembly. Rewritten assemblies are typically produced for testing builds (debug/checked), but may also be used for retail builds where the use of the contract rewriter is deemed acceptable. The advantages of using rewriting are:

* Post-conditions and invariants are executable
* All contracts are properly inherited (no duplication/inconsistency on overridden methods)
* Contracts can be attached to abstract and interface methods
* Contracts are checked for well-formedness
* Runtime failure of contracts is completely customizable and the failed condition is available in string form at runtime.

The behavior and options of the contract rewriter are detailed in Section 3.

## Declarative Assembly

A declarative assembly is obtained by compiling with the conditional symbol FEATURE\_FULL\_CONTRACTS. Declarative assemblies contain all contracts and all method implementations. They are used only as an intermediate form, and never need to be shipped. The two uses of declarative assemblies are:

1. Declarative assemblies are the input to the static checker which aims to validate the implementation against the contracts
2. Declarative assemblies can be run through asmmeta2 to produce a contract reference assembly containing the publicly visible metadata as well as the contracts of publicly visible types and methods.

## Contract Reference Assembly

A contract reference assembly is obtained by running asmmeta2 on a declarative assembly to strip out non-publicly visible metadata and all method bodies. The remaining contents of a contract reference assembly is the publicly visible types and methods along with their contracts.

Contract reference assemblies are used by downstream tools:

* The static checker uses them to determine contracts on cross-assembly calls
* The contract rewriter uses contract reference assemblies to perform contract inheritance across assemblies
* The document generation tools can produce API documentation from contract reference assemblies that includes pre- and post-conditions.

# Contract Rewriter

The contract rewriter is an IL to IL transformer. It is responsible for instrumenting method bodies in such a way that they perform contract checking at runtime, while respecting inherited method contracts and object invariants. Besides the assembly to be rewritten, the contract rewriter can be provided with a reference to a class (let’s call it RC, but any name is valid), containing runtime implementations for the following methods:

// Runtime behavior for pre-conditions failure

void Requires(bool cond, string message);

// Runtime behavior for post-conditions failure

void Ensures(bool cond, string message);

// Runtime behavior for invariant failure

void Invariant(bool cond, string message);

// Runtime behavior for assert failure

void Assert(bool cond, string message);

// Runtime behavior for assume failure

void Assume(bool cond, string message);

If these methods are not provided explicitly, the contract rewriter synthesizes them automatically with the default behavior of calling Debug.Assert, followed by a failfast.

The contract rewriter then performs the following steps:

* Pre-conditions are inherited from base methods and interface methods. Only the original method may have an explicit pre-condition. All overriding and implementing methods cannot have their own pre-condition and simply inherit the base pre-condition.
* Pre-condition visibility is checked so that pre-conditions can only refer to caller visible fields and methods.
* Post-conditions are inherited from base methods and interface methods and joined with any explicit additional post-conditions on the method. Post-conditions are allowed to refer to non-caller visible fields and methods.
* Calls of the form Contract.Requires(cond) are rewritten into calls to RC.Requires(cond, “cond”);
* Calls of the form Contract.Requires(cond, message) are rewritten into calls to RC.Requires(cond, message)
* Any uses of Contract.Old(exp) in post-conditions are turned into locals that are initialized on entry to a method to exp. The local is used in the post-condition instead of the Contract.Old(exp) expression.
* Any uses of Contract.Result in post-conditions are turned into references to the actual method result.
* Calls of the form Contract.Ensures(cond) are rewritten into calls to RC.Ensures(cond, “cond”) at all exit points of the method.
* Calls of the form Contract.Ensures(cond, message) are rewritten into calls to RC.Ensures(cond, message) at all exit points of the method.
* Invariants are inherited from base classes so that a class invariant is the conjunction of its own invariant and that of all base classes. The rewriter inserts calls to RC.Invariant(cond, “cond”) on all public method exits.
* The contract rewriter performs additional well-formedness checks on contracts:
  + All methods called from contracts are annotated with [Pure]
  + Legacy requires do not contain local bindings, returns, or other arbitrary code unrelated to the pre-condition evaluation
  + Contract.Old is only used inside of post-conditions.
  + Contract.Result is only used inside normal post-conditions.

## Rewriter Inputs

Besides the assembly A to be rewritten, the contract rewriter also needs contract reference assemblies for assemblies being referenced from A that contain contracts on interfaces and abstract classes that are derived in A. This leads to an interesting problem, namely, given that we use conditional compilation to include some or all contract forms in A, how can we do the same for the contracts in the reference assembly? The contract reference assemblies contain all the contracts, but depending on how A was compiled, we might want only RequiresInRetail to be inherited, but not post conditions or object invariants.

This suggests that we need to pass the same defines we use for compiling A to the rewrite step of A, so that we can determine what to include.

## Client side contract checks

In a scenario where a client A is built against retail libraries that contain no requires, but for which contract reference assemblies are available, the contract rewriter inserts the pre-conditions of library methods at call-sites from within client A. When the client is ready to ship, no pre-condition checks are performed in the final application. Note that this does not apply to pre-conditions expressed using RequiresInRetail or legacy if-then-throw forms, as these are always present.

The contract rewriter determines for a referenced assembly X, whether it is instrumented or not to decide whether to insert call-site pre-condition checking in a client calling X. If it is not instrumented, then the contract reference assembly for X is used to insert requires into A at call sites into X. Again, the rewriter must determine based on the compiler define’s used to build A, which contracts should be inserted.

# Static Checking

Static checking is performed on a declarative assembly A (Section 2.3) and involves contract reference assemblies for all assemblies referenced directly from A. Static checking is always done involving all available contracts, so there are no issues with respect to particular builds and what contract are actually defined.

# Usage Scenarios

Figure 1 shows a general development where a number of assemblies are being developed together (and built together), making use of a number of libraries that are not being built.

Libraries

Assemblies under development  
and test

**Figure 1. General Development Scenario**

Static or runtime contract checking is performed on the assemblies under development and test only, not on the library assemblies. Thus, we assume that the library assemblies contain no runtime contract checking (except possibly RequiresInRetail or legacy requires), but that we do have contract references assemblies available for the library assemblies that provide all contracts. The library contracts are important to have available, as they influence how to check the assemblies under test:

* Pre-conditions (other than RequiresInRetail or legacy pre-conditions) for call from assemblies under test to library methods must be evaluated and checked at call-sites.
* Inherited invariants, requires, and ensures from the libraries must be properly inserted on overriding types and methods in the assembly under test.

The next section describes how this main scenario plays out under different development and build environments.

## Visual Studio Usage

Under Visual Studio, managed contracts provide an extra property pane on C# and VB projects that makes it easy to enable runtime and static checking on a per project basis and configuration dependent. The distinction between assemblies under test and libraries is made as follows: In a particular configuration, the set of assemblies being built that have runtime contract checking enabled is the set of assemblies under test. The remaining projects and external assemblies are considered library code.

Using the MSBuild hooks, we have added extra targets for building contract reference assemblies, rewriting assemblies under test, and performing static checking, so that normal VS builds perform the appropriate actions as automatically as possible.

## Command Line Usage (msbuild)

The VS integration works through project files and msbuild, so the same mechanism works from the command line using msbuild as well.

# Third Party Software Deployment with Contracts

How does a third party distribute its software component to enable other people to make use of contracts written for that component’s interface? The VS/msbuild usage described above allows programmers to build X.Contracts.dll for each assembly X built in VB or C#. This contract reference assembly can be shipped along with the retail version of X.dll. The rewriter and static checking tools automatically use such contract reference assemblies when they are located in the same place as the retail assembly X. If not, a search path can be provided.