CDN3: CCRewrite Design

# TODO

# Usage Guideline

Before you start using contracts in your own code, you need to make a few decisions that influence what contract forms to use for argument validation and where. Note that you can make these decisions independently for each managed assembly you produce (each project):

no

yes

**Usage 3**

* Use “if-then-throw” on public surface for checks in release bits and perform manual inheritance.
* Mark these legacy requires with EndContractBlock().
* Use Requires(…) everywhere else.
* Enable Runtime Checking only on debug builds.

**Usage 2**

* Use Requires<Exn>( … ) on public surface methods if you want to throw specific exceptions on failure.
* Use Requires(…) everywhere else.
* Enable Runtime Checking on all builds
* Use options to emit only precondition and only on public surface of assembly.

no

yes

**Usage 1**

* Use Requires(…) everywhere.
* Enable Runtime Checking on Debug builds only

Argument validation in Released code?

Use Contract Tools on Release build?

The easiest use of the contract tools is if you decide that you don’t need to perform argument validation at runtime in release builds (Usage 1). In that case, you use the contract tools during development, but not on the shipped bits. Remember, you can ship a contract reference assembly along with your release bits so clients can get runtime checking of *your* parameter validations on *their* debug builds via call-site requires checking. This is our long-term use strategy.

The second easiest approach if you need argument validation in your release build is to turn on contract checking in all builds (Usage 2). You therefore take advantage of the tools to produce the runtime strings of your conditions and to perform contract inheritance for you. You can choose to produce specific exceptions for your parameter validations, or have the default ContractException. The risk of using the contract tools in your release build is that you depend on tools that have not reached production quality level.

The trickiest combination is when you want argument validation in release builds, but you are not using the contract tool for runtime checking in that build (Usage 3). In that case, you have to continue writing your argument validation the way you already do, namely using if-then-throw statements (we call them legacy-requires). If you want these to be tool discoverable, add other contracts (such as Ensures) after them, or use Contract.EndContractBlock(), if no other contracts are present. Note that since you are not using the runtime checking tools, you are not going to get any inheritance of contracts and you have to manually repeat your legacy-requires on overrides and interface implementations. For interface and abstract methods, you still get the most benefit if you write contract classes with normal requires and ensures forms so that they appear in contract reference assemblies and are visible to static checkers.

## Differences between Requires<Exn> and “if-then-throw”

The difference between these two forms is whether we rely on tools to perform inheritance of contracts or whether programmers manually inherit them as in existing practice. Requires<Exn>(…) is intended to be used with tool support, meaning that they are inherited in all builds and the message strings are generated automatically to include the failing condition. Well-formedness checks of contracts do not allow specifying new Requires<Exn> on overrides or interface implementations.

On the other hand, legacy-requires (if-then-throw) try to capture existing practice for release builds, while providing some extra support for Debug builds using the tools. In Debug builds with contract tool support, legacy-requires are inherited (and added to manually inherited forms). Additionally, on failure of legacy-requires the contract failure hook is called. This provides the ability to detect otherwise silent failures (due to catches) in debug builds.

# Contract Checking Level

The different kinds of contracts can be emitted on demand by providing the rewriter with a desired checking level. The following table maps pre-defined levels to the set of contract checks enabled at that level.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Level  Check | 0 | 1 | 2 | 3 | 4 | 5 |
| Requires |  |  | X | X | X |  |
| Requires<E> |  | X | X | X | X |  |
| Legacy-Requires |  | X | X | X | X |  |
| Ensures |  |  |  | X | X |  |
| Invariant |  |  |  |  | X |  |
| Assert |  |  |  |  | X |  |
| Assume |  |  |  |  | X |  |

Note that there are no conditional compilation symbol differences in all these levels, i.e, whenever the rewriter is run, we assume the original assembly was built with CONTRACTS\_FULL. Thus, it is the job of the rewriter to remove checks that are no desired, such as for example legacy-requires or invariants.

# Runtime Check Code Generation

The basic operation of CCRewrite is to place runtime checks for contracts at appropriate places. The contracts may come from a variety of places, e.g., via inheritance from contract reference assemblies or other code in the assembly being rewritten.

## Rewriter Methods

Every contract usage is translated to call a particular rewriter method according to the table below:

|  |  |
| --- | --- |
| Requires(cond) | CR.Requires(cond, null, “cond”) |
| Requires(cond, msg) | CR.Requires(cond, msg, “cond”) |
| Requires<E>(cond) | CR.Requires<E>(cond, null, “cond”) |
| Requires<E>(cond, msg) | CR.Requires<E>(cond, msg, “cond”) |
| Ensures(cond) | CR.Ensures(cond, null, “cond”, exn) |
| Ensures(cond,msg) | CR.Ensures(cond, msg, “cond”, exn) |
| EnsuresOnThrow<E>(cond) | CR.EnsuresOnThrow(cond, null, “cond”, exn) |
| EnsuresOnThrow<E>(cond,msg) | CR.EnsuresOnThrow(cond, msg, “cond”, exn) |
| Invariant(cond) | CR.Invariant(cond, null, “cond”) |
| Invariant(cond, msg) | CR.Invariant(cond, msg, “cond”) |
| Assert(cond) | CR.Assert(cond, null, “cond”) |
| Assert(cond, msg) | CR.Assert(cond, msg, “cond”) |
| Assume(cond) | CR.Assume(cond, null, “cond”) |
| Assume(cond, msg) | CR.Assume(cond, msg, “cond”) |

As you can see, the different overloads are all reduced to calls on 7 distinct methods:

class CR {

void Requires(bool cond, string userMessage, string condition);

void Requires<E>(bool cond, string userMessage, string condition);

void Ensures(bool cond, string userMessage, string condition);

void EnsuresOnThrow(bool cond, string userMessage, string condition, Exception exn);

void Invariant(bool cond, string userMessage, string condition);

void Assert(bool cond, string userMessage, string condition);

void Assume(bool cond, string userMessage, string condition);

}

The exception argument to EnsuresOnThrow is the actual caught exception. These runtime contract methods are either generated by CCRewrite in the generated type System.Diagnostics.Contracts.\_\_ContractsRuntime, or they can be provided by the programmer as custom rewriter methods to CCRewrite (see the /rw option).

In the case where CCRewrite synthesizes the methods, they all have the following form:

void Requires(bool cond, string userMessage, string condition) {

if (cond) return;

ReportFailure(ContractFailureKind.Precondition, userMessage, condition, null);

}

That is, the methods return if the condition is true. Otherwise, they call ReportFailure. The other contract methods are similar. EnsuresOnThrow passes the caught exception on to the ReportFailure method.

The method that has special pre-defined failure behavior is Requires<E>:

void Requires<E>(bool cond, string userMessage, string condition)

where E:Exception {

if (cond) return;

var message = RaiseContractFailedEvent(ContractFailureKind.Precondition,

userMessage, condition, null);

if (message == null) return; // handled

// we don’t use default message here to work better with Arg exceptions

if (userMessage != null) {

message = String.Format(“{0} required: {1}”, condition, userMessage);

}

else {

message = String.Format(“{0} required”, condition);

}

Exception e = null;

ConstructorInfo cinfo = typeof(E).GetConstructor(new[]{typeof(string)});

if (cinfo != null) {

e = cinfo.Invoke(message) as Exception;

}

if (e == null) throw new ArgumentException(message);

throw e;

}

The ReportFailure and RaiseContractFailedEvent methods are discussed below.

## ReportFailure Method

The ReportFailure method can be provided as part of the custom rewriter methods passed to CCRewrite. Otherwise the following method is synthesized:

void ReportFailure(ContractFailureKind kind, string userMessage,

string condition, Exception inner)

{

var message = RaiseContractFailedEvent(kind, userMessage, condition, inner);

if (message == null) return; // handled

TriggerFailure(kind, message, userMessage, condition, inner);

}

Failure first calls the RaiseContractFailedEvent method which returns either null if the failure is handled, or the message string to use when calling TriggerFailure.

The two methods called RaiseContractFailedEvent and TriggerFailure can also be provided by the user in the supplied custom rewriter methods. Otherwise, the methods from the Microsoft.Contracts.dll or mscorlib.dll are used.

## TriggerFailure

The default implementation of TriggerFailure in the library is to display an assert dialog box. The rewriter can be instructed via /throwonfailure to synthesize an alternative method that throws a ContractException instead.

## Assert vs. Throw

The default behavior for contract failure is to pop up an assert dialog box as implemented in the TriggerFailure method in the library and the BCL. This applies even to legacy-requires and Requires<E>. Unless the /throwonfailure option is passed to the rewriter, these legacy-requires and Requires<E> preconditions produce runtime checks exactly as an ordinary Requires would, i.e., they call the Requires method in the runtime contract type.

## Legacy Requires

When /throwonfailure is used, legacy requires trigger their original failure behavior. However, the rewriter still inserts calls to RaiseContractFailedEvent and treats the failure as handled when that method returns null (thus skipping over the original throw code).

This provides a uniform treatment of all contracts when rewritten:

1. The ContractFailedEvent is triggered
2. If not handled, runtime failure behavior follows
   1. By default, an assertion dialog
   2. With /throwonfailure a ContractException or programmer specified exception is thrown.

## Recursion Guards

Contract instrumentation may lead to new non-terminating call paths among methods. To guard against that, the rewriter emits a ThreadStatic Boolean field (insideContractEvaluation) in the runtime contract type (System.Diagnostics.Contracts.\_\_ContractsRuntime) and guards against evaluating contracts when the field is true.

Requires sections in methods are emitted as follows:

if (!insideContractEvaluation) {

insideContractEvaluation = true;

try {

<all requires checks>

}

finally {

insideContractEvaluation = false;

}

}

Ensures checks are emitted similarly inside such a guarded block. For EnsuresOnThrow, each individual exceptional post condition appears in its own catch block. Therefore, the above guard appears for each exceptional post condition individually.

# Eliminating Microsoft.Contracts.dll dependencies

Some clients want to eliminate all dependencies on Microsoft.Contracts.dll after rewriting. In order to do so, such clients need to provide custom rewriter methods for all contract forms as described in the previous section so no calls to either RaiseFailureEvent or ShowFailure are generated by the rewriter.

Furthermore, the custom rewriter class must contain replacements for Contract.Forall and Contract.Exist methods. The rewriter will retarget uses of such methods to the provided methods.

The rewriter also takes care of eliminating all original methods marked with the ContractInvariantMethod attribute.

Finally, the rewriter will remove all references to attributes in Microsoft.Contracts.dll.

# Call-Site Requires Instrumentation

When using contracts with libraries where not all libraries are instrumented with the same contract checking level as the one being rewritten, it becomes important to perform Requires checks at call-sites, rather than relying on the called method being instrumented.

A call-site calling a method T.M should be instrumented, unless we are sure that the call goes to a method that is itself instrumented. To determine if a method is properly instrumented with the appropriate level of requires checking, one of the following has to be true:

* The target method has no requires
* The call is effective non-virtual (meaning either the method is non-virtual or the call site) and
  + The called method is defined in the current assembly, or
  + The called method is defined in an assembly with a compatible runtime instrumentation level
  + The target method only contains legacy requires
* The call is to an interface method, and
  + The interface is internal to the current assembly (thus all implementations have to be internal and are instrumented, ignoring friend access)
* The call is virtual to a non-interface method
  + The containing type of the virtual method is internal to the current assembly (thus all implementations are internal to the current assembly and instrumented, ignoring friend access)

In any other case, the method might be implemented by another assembly which has less runtime instrumentation.

It is worth considering performing all virtual method Requires at call-sites.

## Call-Site Call Wrappers

When a call-site must be instrumented, we create a wrapper method that performs the check and then calls the original method target. There are four cases to distinguish:

|  |  |  |
| --- | --- | --- |
| Non-virtual call to public method T.M<X> | Replace with non-virtual call to static wrapper method WrapperT.NVM<X> | Static wrapper method WrapperT.NVM<X> can be shared in assembly. Contains non-virtual call. |
| Virtual call to public method T.M<X> | Replace with non-virtual call to static wrapper method Wrapper.VM<X>. | Wrapper method WrapperT.VM<X> can be shared in assembly. Contains virtual call. |
| Non-virtual call to protected method M<X>.  (Note: must occur on “this”) | Replace with non-virtual call to instance wrapper method WNVM<X> | Private wrapper method WNVM must be inserted in same type containing call. Wrapper performs a non-virtual call on “this”. |
| Virtual call to protected method M<X>.  (Note: must occur on “this”) | Replace with non-virtual call to instance wrapper method WVM<X>. | Private wrapper WVM must be inserted in same type containing call. Wrapper performs a virtual call to original method on “this”. |
| Constrained<S> Virtual call to public method T.M<X> | Replace with non-virtual call to static wrapper method WrapperT.VM<X,S> | Wrapper method WrapperT.VM<X,S> contains a constrained<S> callvirt to original method. |

The wrapper types WrapperT generated for public methods are assembly internal types. We should organize them by using the same namespace/nested type structure as their original, but in the System.Diagnostics.Contracts.Wrappers namespace.