CDN1: Contract Extractor Design

# Overview

The contract extractor is a library responsible for finding and extracting contracts from C#, VB, and F# compiled code in order to make the contracts available to tools such as documentation generation, runtime check instrumentation, and static analysis.

Besides identifying the sequence of IL instructions inside a method body that represent contracts, the extractor also performs a number of checks on the code that guarantee that the code sequence can be extracted without affecting the normal body, or the functioning of the contract code. We call these well-formedness checks.

# Method Contracts

A method contract consists of slightly more than just the list of requires and ensures appearing in a method. In addition to these lists, there may be initialization code that needs to be made part of the contracts, such as a closure objects initialization, or local initialization for a helper local that is used inside a contracts (such as @this sometimes used in interface contracts).

Such initializations that may be necessary to properly evaluate contracts are extracted into a contract initialization block. When contracts are inherited or copied into call-site-wrappers, or during static analysis, these contract initialization blocks must be copied along/analyzed in order for the semantics of method contracts to be well defined.

# Extraction

The basic extraction scheme assumes that the method body consists of three distinct code sections of the following schematic form:

<method preamble section>

<method contract section>

<ordinary method body>

The boundary between the method contract section and the ordinary method body is identified by the last call to either Contract.Requires, Contract.Ensures, Contract.EnsuresOnThrow or Contract.EndContractBlock. If no such call exists, then the method contract section is empty and no contracts are extracted from the method.

The more difficult part is identifying the boundary between the method preamble section and the start of the method contracts. The preamble section contains code such as nops for stepping during debugging, closure object creation, and initialization code in constructors. The next sections describe what kinds of preamble code the extractor recognizes and what it does with them.

The result of extraction of method contracts is

* a list of requires
* a list of normal ensures
* a list of exceptional ensures
* a header section containing such things as initial nops for debug/stepping that evaluates prior to contracts
* a contract initialization section containing parts of the preamble that need to be evaluated prior to any contract evaluation and copied to places where the contract is inherted/copied to
* a residual preamble block containing anything that is not in the header or contract initialization
* A post-preamble contract block that contains residual initialization that needs to be performed prior to postcondition evaluation but cannot be performed in the contract initialization section

The idea is that the rewriter can then rearrange the blocks to produce:

<header section>

<initial nops>

<contract initialization section>

<closure initialization>

<delegate local initialization>

<contract local initialization>

<contract requires>

<residual preamble section>

<field initializers C#>

<base/this .ctor calls>

<field initializers VB>

<ordinary method body>

<post preamble section>

<contract ensures>

Splitting the preamble section into the contract initialization section and the residual preamble is explained below.

## Preambles

The preamble section in general looks as follows:

<initial nops> // unless ctor

<Nulling of cached closure locals>

<closure object allocation>

<closure object field initialization of parameters except this>

<field initializers> // C# constructors

<base class ctor call> // if ctor

<initial nops>

<closure object field initialization of this parameter>

<secondary closure object allocation> // C# constructors

<secondary closure field initialization> // C# constructors

<field initializers VB

### Nop Preambles

Compilers may emit nop instructions in debug builds to improve the stepping experience inside a debugger, e.g., to stop on the opening curly brace of a C# method. In order to maintain that experience, such initial nops are not part of the contract initialization section, but should be maintained at the beginning of the method. Thus, they go into the **header section**.

### Closure Initialization

If a method uses anonymous delegates, compilers generate nested private classes to hold method state that the closure methods need to access. These closure objects are usually initialized on entry to the method. Such closure initialization code consists of the object construction followed by a number of field-writes into the closure object. Any object construction whose object type is marked as compiler generated is recognized as a closure initialization.

Such code sequences must be executed prior to contracts and need to be copied wherever contracts are copied to, since contracts may refer to the closure object. Closure initializations are thus made part of the **contract initialization section**.

### Delegate Local Initialization

If a method contains anonymous delegates that don’t require a closure object, compilers may emit locals to hold and cache the constructed delegate object. Sometimes, compilers will emit null initializations of such delegate local variables at the beginning of a method. Instructions to initialize such locals need to go into the **contract initialization section**, as the contracts may test/initialize these delegate locals.

### Object Instance Construction

Constructors of classes contain code to call the base class constructor or another constructor of the same class. They may also contain code to initialize fields generated from field initializers. In C#, field initializers appear before the base/this constructor call. In VB, field initializers appear after the base/this constructor call.

Any code appearing before and including the base/this constructor call is made part of the preamble. In addition, any field assignments on this or constructor calls on field struct addresses of this are also included as part of the preamble block in constructors. All field initialization and base/this .ctor calls are part of the **residual preamble section**.

A complicating factor in constructors with closures is that the C# compiler may split the closure into two separate closures, one containing only parameters (except this), and a second one, containing locals, potentially this, and a pointer to the closure of parameters. The closure for parameters only is initialized prior to the base ctor call, whereas the other closure is initialized after the base ctor call (in order to access this). As a result, for constructors, the code that is generated needs to look as follows:

<header section>

<initial nops>

<field inits>

<parameter closure initialization>

<pre base ctor contract initialization section>

<parameter closure initialization>

<delegate local initialization>

<contract local initialization>

<contract requires>

<residual preamble section>

<field initializers C#>

<base/this .ctor calls>

<remaining closure initialization>

<field initializers VB>

<post base ctor contract initialization section>

<remaining closure initialization>

<delegate local initialization>

<contract local initialization>

<ordinary method body>

### Struct Constructors

Struct or Structure constructors do not contain field initializer code, but they may contain this constructor calls. Any code before and including the this-constructor call is considered part of the preamble section. Additionally, VB emits an initobj this on structure constructors that don’t call any other constructors. This instruction is also considered part of the preamble section.

These code sequences are part of the **residual preamble section**.

### Contract Locals

For interface contract methods, the contract section may start with a local initialization of the form:

I @this = (I)this;

This is necessary, if the contract wants to refer to other interface methods. Since interface contract methods use explicit implementations, one cannot refer to them from “this”, and a cast is needed. To simplify this, we allow a local binding as above. This binding is considered part of the **contract initialization section**.

## Contract Section

The contract section consists of a sequence of sections of code delimited either by calls to Contract.Requires, Contract.Ensures, or Contract.EnsuresOnThrow. Additionally, legacy requires of the form if-then-throw are recognized as contracts if they appear prior to any calls of the above form or a Contract.EndContractBlock call.

### Legacy-Requires

Any code of the form

if (cond) { throw new …; }

is treated as a legacy-requires of the form, where the precondition is !cond.