Code Contracts in C#

Hans-Wolfgang Loidl
<H.W.Loidl@hw.ac.uk>

**School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh**

Semester 1 — 2018/19

HERIOT WATT UNIVERSITY

Motivation

Debugging programs is time-consuming and therefore expensive:

- **As a tester you need to make sure to *cover all control-flow paths***
- **You should test the program with a wide range of input data**
- **You need to test the behaviour with *corner cases*, e.g. empty string**

Systematic testing can help with that (see the guest lecture on "Systematic Testing" by Murray Crease from ScottLogic later in the course)
But still, you can't be sure to cover all possible cases.
For that, methods of *program verification* are superior.

HERIOT WATT UNIVERSITY

Motivation

Debugging programs is time-consuming and therefore expensive:

- **As a tester you need to make sure to *cover all control-flow paths***
- **You should test the program with a wide range of input data**
- **You need to test the behaviour with *corner cases*, e.g. empty string**

***Systematic testing* can help with that (see the guest lecture on "Systematic Testing" by Murray Crease from ScottLogic later in the course)**

But still, you can't be sure to cover all possible cases.

For that, methods of *program verification* are superior.

HERIOT WATT UNIVERSITY

Motivation

Debugging programs is time-consuming and therefore expensive:

- As a tester you need to make sure to *cover all control-flow paths*
- You should test the program with a wide range of input data
- You need to test the behaviour with *corner cases*, e.g. empty string

Systematic testing can help with that (see the guest lecture on "Systematic Testing" by Murray Crease from ScottLogic later in the course)
But still, you can't be sure to cover all possible cases.
For that, methods of *program verification* are superior.

HERIOT WATT UNIVERSITY

Motivation

Debugging programs is time-consuming and therefore expensive:

- As a tester you need to make sure to *cover all control-flow paths*
- You should test the program with a wide range of input data
- You need to test the behaviour with *corner cases*, e.g. empty string

Systematic testing can help with that (see the guest lecture on "Systematic Testing" by Murray Crease from ScottLogic later in the course)
But still, you can't be sure to cover all possible cases.
For that, methods of *program verification* are superior.

HERIOT WATT UNIVERSITY

Code Contracts

Code Contracts are a set of libraries, together with tools to establish properties of your program.
Code Contracts allow you to express

- *preconditions*
- *postconditions*
- *object invariants*

in your code for

- *runtime checking*
- *static analysis*
- *documentation*

HERIOT WATT UNIVERSITY

A Simple Example

```
1  class Rational {
2    int numerator;  int denominator;
3
4    public Rational (int numerator, int denominator) {
5      Contract.Requires( denominator != 0 ); // pre-
           condition!
6      this.numerator = numerator;
7      this.denominator = denominator;
8    }
9
10   public int Denominator {
11     get {
12       Contract.Ensures( Contract.Result<int>() != 0 );
             // post-condition
13       return this.denominator;
14     }
15   }
```

A Simple Example (cont'd)

```
1  [ContractInvariantMethod]
2  void ObjectInvariant () {                    // invariant
3    Contract.Invariant ( this.denominator != 0 );
4  }
```

With *static checking* enabled, the IDE will report an error for the call below:

```
Rational badRat = new Rational(3,0);
```

[0] See RationalsWithCodeContracts.cs

Discussion

- The code defines a class for *rational numbers.*
- Semantic side-condition: *the denominator must be non-zero.*
- We express this as a *precondition* in the constructor using `Contract.Requires` (line 5).
- Every Rational object must have a non-zero denominator.
- We express this by an ObjectInvariant method tagged with a `[ContractInvariantMethod]` attribute.
- It uses the method call `Contract.Invariant` to check for a non-zero denominator (line 3).
- `Contract.Ensures` expresss a *postcondition*: the getter Denominator always returns non-zero.

HERIOT WATT UNIVERSITY

Preconditions

Definition (Precondition)

Preconditions **are** ***contracts*** **on the state of the world when a method is invoked.**

- ***Preconditions*** **are expressed using** `Contract.Requires(...)`**.**
- **They generally are used to specify valid parameter values.**
- **All members mentioned in preconditions must be at least as accessible as the method itself**
- **Otherwise, the precondition cannot be understood (and thus satisfied) by all callers of a method.**
- **The condition should also be side-effect free**

HERIOT WATT UNIVERSITY

Precondition Example

```
1  Contract.Requires( x != null );
```

- The above precondition expresses that parameter `x` must be non-null.
- If your code must throw a particular exception on failure of a particular precondition, you can use the generic overloaded form below.

```
1  Contract.Requires<ArgumentNullException>( x != null,
       "x" );
```

HERIOT WATT UNIVERSITY

Postconditions

Definition (Postcondition)

Postconditions **are contracts on the state of a method when it terminates.**

- **The condition is checked just prior to exiting a method.**
- **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.**

HERIOT WATT UNIVERSITY

Postcondition Example

```
1       Contract.Ensures( this.F > 0 );
```

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

```
1       Contract.EnsuresOnThrow<T>( this.F > 0 );
```

- **Postconditions that should hold when particular exceptions escape from a method, are specified using `Contract.EnsuresOnThrow`.**
- **The condition must hold whenever an exception is thrown that is a subtype of T.**

HERIOT WATT UNIVERSITY

Postcondition Example

```
1       Contract.Ensures( this.F > 0 );
```

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

```
1       Contract.EnsuresOnThrow<T>( this.F > 0 );
```

- Postconditions that should hold when particular exceptions escape from a method, are specified using `Contract.EnsuresOnThrow`.
- The condition must hold whenever an exception is thrown that is a subtype of T.

HERIOT WATT UNIVERSITY

Special Methods within Postconditions

- *Method Return Values:* **Within postconditions the method's return value can be referred as** `Contract.Result<T>()` **(**`T` **... return type)**
- *Pre-state Values (OldValue):* **Within a postcondition, an old expression refers to the value of an expression from the pre-state, using** `Contract.OldValue<T>(e)` **(**`T` **... type of** `e`**).**
- *Out Parameters:* **Because contracts appear before the body of the method, most compilers do not allow references to out parameters in postconditions. To get around this issue, the library provides the method** `Contract.ValueAtReturn<T>(out T t)` **which will not require that the parameter is already defined.**

HERIOT WATT UNIVERSITY

Object Invariants

Definition (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.**
- **All object's invariants must be in one private nullary instance method containing only `Contract.Invariant`**
- **These methods are identified by being marked with the attribute `[ContractInvariantMethod]`**

```
1  [ContractInvariantMethod]
2  private void ObjectInvariant () {
3    Contract.Invariant(this.y >= 0);
4    Contract.Invariant(this.x > this.y);
5    ...
6  }
```

Assertions

Definition (Assertions)

Assertions **are** ***contracts*** **on the state of the world at an arbitrary point in the program.**

- **Assertions are specified using `Contract.Assert`.**
- **They are used to state a condition that must hold at that program point.**

```
1 Contract.Assert(this.privateField > 0);
2 Contract.Assert(this.x == 3, "Why␣isn't␣the␣value␣of␣
      x␣3?");
```

HERIOT WATT UNIVERSITY

Assume

Definition (Assumptions)

Assumptions **are properties that are expected to be true.**

- **Assumptions are specified using `Contract.Assume`.**
- **These are properties that are expected to be true without checking them.**

```
1 Contract.Assume(this.privateField > 0);
2 Contract.Assume(this.x == 3, "Static checker assumed
      this");
```

Quantifiers

- **Code contracts support a limited form of quantifiers:**
  - **the mathematical $\forall$ quantifier as `Contract.ForAll`**
  - **the mathematical $\exists$ quantifier as `Contract.Exists`**
- **The usage of these is limited to what can be computed efficiently.**

HERIOT WATT UNIVERSITY

Quantifier Examples

```
1 public int Foo<T>(IEnumerable<T> xs){
2   Contract.Requires(
3     Contract.ForAll(xs , x => x != null) );
4   foreach (var x in xs)
5     ...
6 }
```

This example of a contract says that all elements contained in the parameter xs must be non-null.

```
1 public int[] Bar(){
2   ...
3   Contract.Ensures(
4     Contract.ForAll(0, Contract.Result<int[]>().
        Length,
5                     index => Contract.Result<int[]>()
                        [index] > 0));
```

This method has a postcondition that all returned values in the array must be positive (could use LINQ's `Enumerable.All`

Quantifier Examples

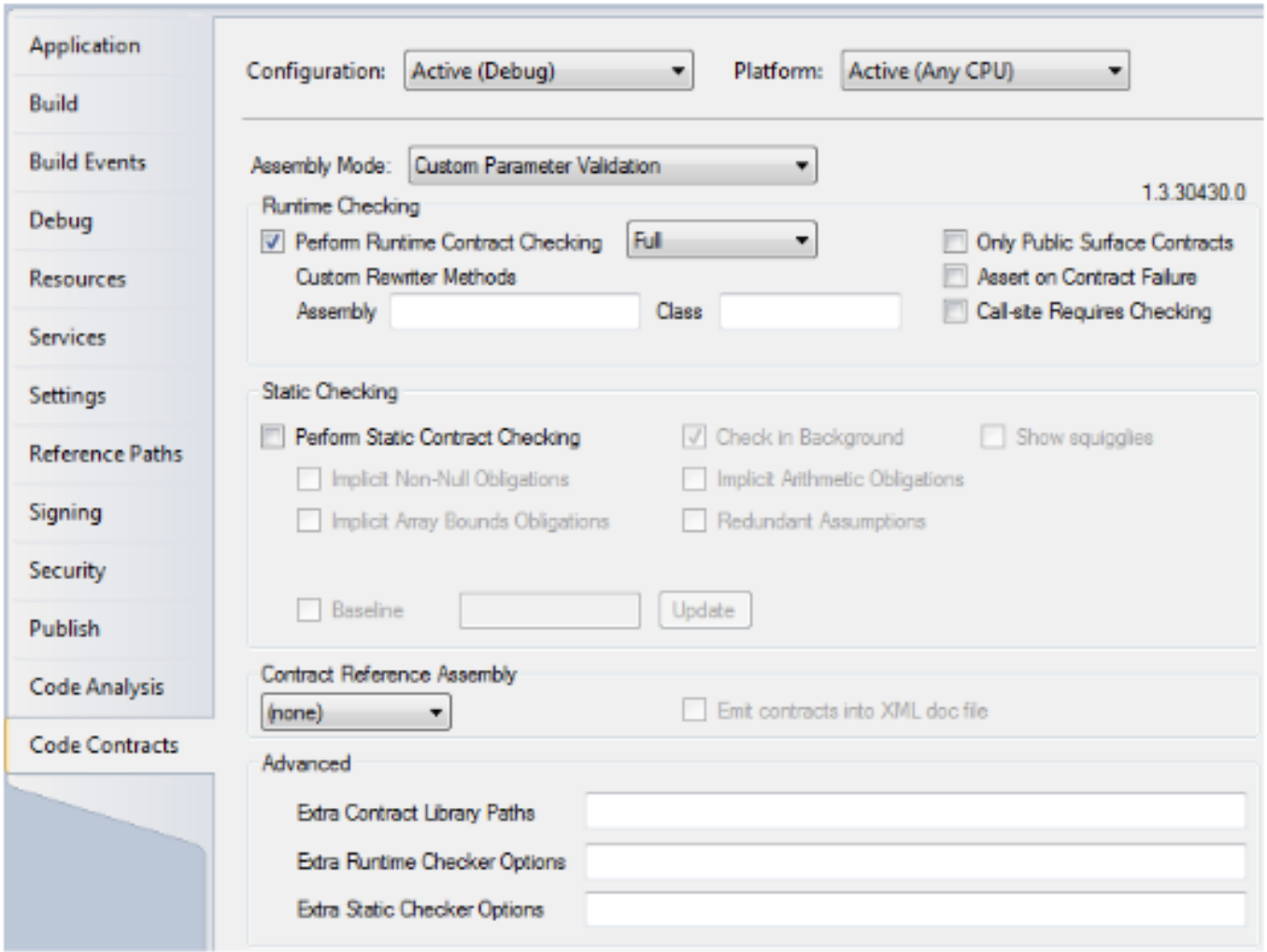
```
1 public int Foo<T>(IEnumerable<T> xs){
2   Contract.Requires(
3     Contract.ForAll(xs , x => x != null) );
4   foreach (var x in xs)
5     ...
6 }
```

This example of a contract says that all elements contained in the parameter xs must be non-null.

```
1 public int[] Bar(){
2   ...
3   Contract.Ensures(
4     Contract.ForAll(0, Contract.Result<int[]>().
        Length,
5                     index => Contract.Result<int[]>()
                        [index] > 0));
```

This method has a postcondition that all returned values in the array must be positive (could use LINQ's `Enumerable.All`

Enabling Code Contracts in Visual Studio



Another Example: index lookup

Our early `Get` example of a function for returning the n-th element of a data-structure with an indexer only works if the index is less than the array length, now specified as a *pre-condition* using a code contract:

```
1 static int Get(int[] arr, int n) {
2   Contract.Requires(n < arr.Length);
3   return arr[n];
4 }
```

If we define data structures in the main code like this:

```
1 static void Main() {
2   int[] arr = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
3   int good_n = 3;   /* OK index */
4   int bad_n = 15;   /* illegal index */
```

the following call will generate an error in the IDE (if static checking is enabled): `Get(arr, bad_n)`

Summary

- *Code Contracts* are a systematic way to *test properties* of your program, either at compile-time or at run-time.
- If enabled, the properties will be checked at *run-time*, raising an *exception* if it's wrong.
- If enabled, some properties can be checked at *compile-time*, giving an *error message in the IDE*.
- If none of the above is enabled, there is no performance penalty for having code contracts in your code.
- You should use in particular pre- and post-conditions for methods to specify correct behaviour.

HERIOT WATT UNIVERSITY

Appendix: Basics of Program Logics

A *Hoare-style logic* is a formalism to reason about the correctness of programs.
In Hoare-style logics we write: $\{P\}\texttt{e}\{Q\}$
This should be read as "if the property P holds before the execution of the program e, then the property Q holds after executing e"
Example: Specification of an exponential function

$$\{0 \leq y \ \wedge \ x = X \wedge \ y = Y\} \ \texttt{exp}(x, y) \ \{r = X^Y\}$$

Note: X, Y **are *auxiliary variables* and must not appear in** e

HERIOT WATT UNIVERSITY

A Simple while-language

Language:

$$
\begin{array}{rcl}
e & ::= & \texttt{skip} \\
 & | & x := t \\
 & | & e_1 ; e_2 \\
 & | & \texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2 \\
 & | & \texttt{while } b \texttt{ do } e \\
 & | & \texttt{call}
\end{array}
$$

A judgement has this form (for now!)

$$\vdash \{P\}\ e\ \{Q\}$$

A judgement is valid if the following holds

$$\forall z\ s\ t.\ s \overset{e}{\rightsquigarrow} t \Rightarrow P\ z\ s \Rightarrow Q\ z\ t$$

HERIOT WATT UNIVERSITY

A Simple while-language

Language:

$$
\begin{array}{rcl}
e & ::= & \texttt{skip} \\
 & | & x := t \\
 & | & e_1 ; e_2 \\
 & | & \texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2 \\
 & | & \texttt{while } b \texttt{ do } e \\
 & | & \texttt{call}
\end{array}
$$

A judgement has this form (for now!)

$$\vdash \{P\}\ e\ \{Q\}$$

A judgement is valid if the following holds

$$\forall z\ s\ t.\ s \overset{e}{\rightsquigarrow} t \Rightarrow\ P\ z\ s \Rightarrow\ Q\ z\ t$$

HERIOT WATT UNIVERSITY

A Simple Hoare-style Logic

$$\frac{}{\vdash \{P\}\ \texttt{skip}\ \{P\}}\ (\textsc{skip}) \qquad \frac{}{\vdash \{\lambda z\ s.\ P\ z\ s[t/x]\}\ x := t\ \{P\}}\ (\textsc{assign})$$

$$\frac{\vdash \{P\}\ e_1\ \{R\} \quad \{R\}\ e_2\ \{Q\}}{\vdash \{P\}\ e_1;e_2\ \{Q\}}\ (\textsc{comp})$$

$$\frac{\vdash \{\lambda z\ s.\ P\ z\ s\ \wedge\ b\ s\}\ e_1\ \{Q\} \quad \vdash \{\lambda z\ s.\ P\ z\ s\ \wedge\ \neg(b\ s)\}\ e_2\ \{Q\}}{\vdash \{P\}\ \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2\{Q\}}\ (\textsc{if})$$

$$\frac{\vdash \{\lambda z\ s.\ P\ z\ s\ \wedge\ b\ s\}\ e\ \{P\}}{\vdash \{P\}\ \texttt{while}\ b\ \texttt{do}\ e\{\lambda z\ s.\ P\ z\ s\ \wedge\ \neg(b\ s)\}}\ (\textsc{while})$$

$$\frac{\vdash \{P\}\ body\ \{Q\}}{\vdash \{P\}\ \texttt{CALL}\ \{Q\}}\ (\textsc{call})$$

HERIOT WATT UNIVERSITY

A Simple Hoare-style Logic (structural rules)

The consequence rule allows us to weaken the pre-condition and to strengthen the post-condition:

$$\frac{\forall s\ t.\ (\forall z.\ P'\ z\ s \Rightarrow P\ z\ s) \quad \vdash \{P'\}\ e\ \{Q'\} \quad \forall s\ t.\ (\forall z.\ Q\ z\ s \Rightarrow Q'\ z\ s)}{\vdash \{P\}\ e\ \{Q\}}\ (\textsc{conseq})$$

Recursive Functions

In order to deal with recursive functions, we need to collect the knowledge about the behaviour of the functions.

We extend the judgement with a context Γ, mapping expressions to Hoare-Triples:

$$\Gamma \vdash \{P\}\ e\ \{Q\}$$

where Γ has the form $\{\ldots, (P', e', Q'), \ldots\}$.

HERIOT WATT UNIVERSITY

Recursive Functions

Now, the call rule for recursive, parameter-less functions looks like this:

$$\frac{\Gamma \cup \{(P, \texttt{CALL}, Q)\} \vdash \{P\}\ body\ \{Q\}}{\Gamma \vdash \{P\}\ \texttt{CALL}\ \{Q\}} \qquad (\textsc{call})$$

We collect the knowledge about the (one) function in the context, and prove the body.

***Note*: This is a rule for partial correctness: for total correctness we need some form of measure.**

HERIOT WATT UNIVERSITY