CodeContracts: Specification & Verification for the working programmer

Francesco Logozzo
joint work with
M. Barnett and M. Fahndrich
Demo!!!
CodeContracts Impact

- API in .NET 4.0
- Externally available ~20 months
  - >50,000 downloads, very active forum
  - 3 book chapters on CodeContracts
  - Many dozens of blog articles
  - Active forum
- Internal and external adoption
- Publications, talks, lectures
  - POPL, ECOOP, OOPSLA, VMCAI, APLAS, SAS, SAC, FoVeOOS, VSTTE ...
"The program does not go wrong"
- What does it mean?
- It does not crash
  - Division by zero
  - Dereference of null (or 0 or nil)
  - No exception is thrown
- ... 
- It meets its specification
  - Specification???
  - What’s that?
Informally: “What the program should do”
   “It should sort all the elements of the array”
Computers do not like English
   (or Italian for what it matters ;-))
Which *formal* language?
   How close to the computer?
   How close to the human?
   How close to the programmer?
   How close to the verification tool?
A plethora of specification lang.

- Implementation
  - Temporal Logic?
  - Z (B) notation?
  - UML?
  - TLA+?
  - Abstract state machines (ASM)?
  - JML? Code Contracts?
  - SAL?
  - ...
  - Many many
  - ...
  - ...
  - ...
The program behavior is included in the behaviors admissible from the specification:

Program is correct 😊
The program behavior is **not** included in the behaviors admissible from the specification:

Program is incorrect 😞
(Some behavior may not meet the specification)
The problem is undecidable
Need to perform abstraction
In the concrete:
  Is the program correct? Yes/No
In the abstract:
  Is the program correct? Yes/No/I do not know
Which abstraction?
  Upper-approximate the program semantics
  Under-approximate the specification semantics
Potato (when lucky)

The **over-approximation** of the program behavior is included in the **under-approximation** of the admissible specification:

Program is correct! 😊
In this case, too many!

The over-approximation of the program behavior is not included in the under-approximation of the admissible specification:

Program may be correct or it may not 😞
To sum up...

Specification

- How do we specify the intent?
  - We should not over-specify

Verification

- How do we check the program is doing the right thing?
  - Runtime (testing)
  - Static time (verification)
Specification with Contracts
Contracts?

- Each program module specifies what it expects from its users:
  - Precondition
- What it ensures to its users:
  - Postcondition
- What holds in a stable state:
  - Object invariants
Contracts: What for?

Document design decisions

```c#
public virtual int Next(
    int minValue,
    int maxValue
)
```

**Parameters**

- `minValue`  
  Type: `System.Int32`  
  The inclusive lower bound of the random number returned.

- `maxValue`  
  Type: `System.Int32`  
  The exclusive upper bound of the random number returned. `maxValue` must be greater than or equal to `minValue`.

**Return Value**  
Type: `System.Int32`  
A 32-bit signed integer greater than or equal to `minValue` and less than `maxValue`; that is, the range of return values includes `minValue` but not `maxValue`. If `minValue` equals `maxValue`, `minValue` is returned.

**Exceptions**

- `ArgumentOutOfRangeException`  
  Condition: `minValue` is greater than `maxValue`.

**Remarks**

Unlike the other overloads of the `Next` method, which return only non-negative values, this method can return a negative random integer.

**Notes to Inheritors:**

Starting with the .NET Framework version 2.0, if you derive a class from `Random` and override the `Sample` method, the distribution provided by the derived class implementation of the `Sample` method is not used in calls to the base class implementation of the `Random.Next(Int32, Int32)` method overload if the difference between the `minValue` and `maxValue` parameters is greater than `Int32.MaxValue`. Instead, the uniform distribution returned by the base `Random` class is used. This behavior improves the overall performance of the `Random` class. To modify this behavior to call the `Sample` method in the derived class, you must also override the `Random.Next(Int32, Int32)` method overload.
Contracts: What for?

- Amplify runtime checking (debugging)
  - More assertions in the code
  - More stable the code
  - Assertions can be disabled in release builds

- Enable modular static analysis
  - Improved precision
    - Analysis should not assume worst case
  - More scalability
  - A little bit as a type-checker (but more refined!)
But we already have assertions!

- All languages have an assert
  - `assert(exp)` macro in C/C++
  - `assert exp` keyword in Java
  - `Debug.Assert(exp)` static method in .NET

Assert is not visible from the caller!

```csharp
static public int GCD(int x, int y)
{
    Debug.Assert(x > 0);
    Debug.Assert(y > 0);
}
```
Exceptions

Use exceptions for parameter validation:

```csharp
static public int GCD(int x, int y)
{
    if (x < 0)
        throw new ArgumentException("Error");
    ...
}
```

At library surface
To protect from unwanted values
To early detect API misuses
Again, not visible to callers
But we have Debug.Assert!

Cannot (easily) specify a postcondition

```csharp
static public int GCD(int x, int y)
{
    Debug.Assert(x > 0);
    Debug.Assert(y > 0);

    while (true)
        if (x < y) { y %= x; if (y == 0) return x; } 
        else { x %= y; if (x == 0) return y; }
}
```

Debug.Assert(Result > 0) ?
Inheritance

Precondition: Should be weaker
Postcondition: Should be stronger
How do I enforce it?

Object invariants
Valid in steady states
Ex: this.x != null
Should I add it at every method?

Interfaces, abstract methods
Where I put my assert?
Contracts yesterday

- First class citizens in the language
- Provide syntax to express contracts
  - Examples: Eiffel, D, Spec#
- Why not everyone was using it?
  - New language (start from scratch, or almost)
  - New compiler (do you trust it?)

```plaintext
decrement is

-- Decrease counter by one.
require
  item > 0
do
  item := item - 1
ensure
  item = old item - 1
end
```
Contracts yesterday

Inside comments or as code annotation
Ex. JML, Eclipse for non-null ...

```java
//@ public invariant balance >= 0 && balance <= MAX_BALANCE;
//@ assignable balance;
//@ ensures balance == 0;
public BankingExample() { balance = 0; }
//@ requires 0 < amount && amount + balance < MAX_BALANCE;
//@ assignable balance;
//@ ensures balance == \old(balance + amount);
public void credit(int amount) { balance += amount; }
//@ requires 0 < amount && amount <= balance;
//@ assignable balance;
//@ ensures balance == \old(balance) - amount;
public void debit(int amount) { balance -= amount; }
```

Why not everyone is using it?
Persistence?
Need for serialization, parsing...
Separate type checking, name resolution...
Code contracts
Idea: Use the IL as contract representation
Use static methods to a contract library
Language agnostic: same for C#, VB, F# ...

```csharp
Contract.Requires(source != null);
Contract.Requires(!String.IsNullOrEmpty(suffix));

Contract.Ensures(Contract.Result<string>() != null);
Contract.Ensures(!Contract.Result<string>().EndsWith(suffix));

Contract.Assert(trimmed IsNot Nothing)]
Contract.Assert(Not trimmed.EndsWith(".dll"))
```
Plain code for contracts

Static methods to a contract library
  Language agnostic: same for C#, VB, F# ...
  Standard from .NET 4.0

No need for a new compiler/language
  Precondition: Contract.Requires(...)
  Postcondition: Contract.Ensures(...)
  Invariant: Contract.Invariant(...)

What are they?
Preconditions

- **Contract.Requires(exp)**

```csharp
int foo(String s, int y)
{
    Contract.Requires(s != null);
    Contract.Requires(y > 0);
    // ...
}
```

C# expressions
Which is the underlying language specification?

Your programming language!!!

Public Function foo(ByVal s As String, ByVal y As Integer) As Integer
  Contract.Requires(s IsNot Nothing)
  Contract.Requires(y > 0)
  ' ...
End Function

Int32 __gc* foo(String __gc* s, Int32 __gc* y)

Contract:::Requires(s != 0);
Contract:::Requires(y > 0);
Postconditions

Contract.Ensures(exp)

Class Field
{
    int x;

    int Set(int y)
    {
        Contract.Ensures(this.x == y);
        this.x = y;
    }
}
Result value?

- In C#/VB/... no name for the returned value
- Use a dummy method

```csharp
public int Fact(int x)
{
    Contract.Ensures(Contract.Result<int>() >= 0);
    ...
}
```
Why `<int>`?  
Why `<bool[]>`?  

```csharp
public int Fact(int x)
{
    Contract.Ensures(Contract.Result<int>() >= 0);
}

public bool[] ArrayFactory(int x)
{
    Contract.Ensures(Contract.Result<bool[]>() != null);
    return new bool[x];
}
```
Class Account
{
    int balance;
    int Add(int k)
    {
        Contract.Ensures(this.balance ==
                        Contract.Old(this.balance) + k);
        this.balance += k;
    }
}
Quantifiers

- Limited form:
  - `Contract.ForAll(0, A.Length, i => A[i] > 0);`
  - `Contract.Exists(0, A.Length, i => A[i] > 0);`
- Exploit higher order functions
Class Account
{
    int balance;

    [ContractInvariantMethod]
    protected void ObjectInvariant()
    {
        Contract.Invariant(balance >= 0);
    }
}
Interfaces

```
[ContractClass(typeof(WithdrawContracts))]  
interface IWithdraw  
{  
    long Balance { get; }  
    void Withdraw(long money);  
}
```
Abstract classes
- Similar to interfaces

Out/ref parameters
- Use dummy method

Legacy code: "if !exp throw exception"
- Use Contract.EndContract()
Advantages

Produced by all the compilers

Free:
- Types
- Intellisense
- Name resolution...

Cross language

Precise semantics

Uniform format understood by our tools
Runtime checking (aka Testing)
Potato & testing

One execution is outside the specification
The program is incorrect!
A sample of the program behavior is included in the behaviors admissible from the specification.

Is the program correct?
Prove the existence of specification violations
  i.e. the existence of bugs
  If a test fails, then there is a bug

Cannot verify the program
  i.e. the existence of no bug!
  Can try only finitely main inputs
    100% code coverage do not imply 100% data coverage

Yet, very useful!!!
Runtime checking of contracts

- C# compiler does not know about contracts
  - Achieved via binary rewriting
    - Handle old, result ...
    - Inherit contracts
    - Stick contracts to interface implementations
public virtual int Add(object value) {
    Contract.Requires(value != null);
    Contract.Ensures(
        Count == Contract.OldValue(Count) + 1);
    Contract.Ensures(
        Contract.Result<int>() == Contract.OldValue(Count));
    if (_size == _items.Length) EnsureCapacity(_size + 1);
    _items[_size] = value;
    return _size++;
}
Static verification
The over-approximation of the program behavior is included in the under-approximation of the admissible specification:

Program is correct! 😊
Static verification

- Any static verification method is incomplete
- Verification is not decidable

```csharp
public static bool NotDecidable()
{
    if (SolvableDiophantineEquation)
        return 1;
    else
        Contract.Assert(false);
}
```
Verification techniques

- Many, many out there...
- Model Checking
- Theorem proving
  - Automatic
    - SMT solvers, resolution based, ...
  - Semi-automatic
  - 2nd order
- All those instances of Abstract interpretation!
Theory of approximations

Semantics are order according to the precision

The more the precise the semantics
The more the properties captured

A static analysis is a semantics
  • Precise enough to capture the properties of interest
  • Rough enough to be computable
Basic Concepts

Concrete domain
- A mathematical structure describing the most precise information on the program
- Usually the program semantics
  - Traces, operational, denotational ...

Abstract domain
- A mathematical structure describing the property of interests
- Ex.: range of a variable
Example: Rule of signs

Abstract semantics is over signs

$$a[k] \rho = \text{sign}(k)$$
$$a[x] \rho = \rho(x)$$
$$a[e_1 + e_2] \rho = a[e_1] \rho \pm a[e_2] \rho$$
$$a[e_1 \ast e_2] \rho = a[e_1] \rho \ast a[e_2] \rho$$

<table>
<thead>
<tr>
<th>$\pm$</th>
<th>$\bot$</th>
<th>neg</th>
<th>zero</th>
<th>pos</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>neg</td>
<td>$\bot$</td>
<td>neg</td>
<td>neg</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>zero</td>
<td>$\bot$</td>
<td>neg</td>
<td>zero</td>
<td>$\bot$</td>
<td>$T$</td>
</tr>
<tr>
<td>pos</td>
<td>$\bot$</td>
<td>$T$</td>
<td>pos</td>
<td>Pos</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$\bot$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>
Example: Rule of signs

(12345565 * 13456) + (-9873 * -1344678)

Sign of the result?

Do the computation: 179 397 928 534
Then take the sign : pos

Do the abstract computation:

(pos * pos) ± (neg * neg)

= pos ± pos

= pos
CodeContracts
Static checker
aka Clousot
Algorithm: High level

For each assembly A, class C, method M

1. Extract the proof obligations
   • What should I prove?
2. Run the analyses
   • Discover facts on the program
3. Use the facts to prove the proof obligations
   • If not, do something else...
Proof obligations

Two kinds: Implicit and explicit

Implicit
- NonNull checking
- Bounds checking
- Arithmetic: Divisions by zero, overflows,
  ...

Explicit
- Assertions
- When calling a method, its precondition
- When returning from a method, its postcondition
The string $s$ should not be null
- Otherwise, exception at run time

Generate the proof obligation: $s \neq \text{null}$
public int[] RandomArray(int len)
{
    var random = new Random(len);
    var arr = new int[len];
    for (var i=0; i < len; i++)
    {
        arr[i] = random.Next();
    }
    return arr;
}
public int Div(int x, int y)
{
    return x / y;
}

public int Abs(int x)
{
    if (x < 0)
        return -x;
    return x;
}
```csharp
public string Concat(string p, string q)
{
    Contract.Requires(p != null);
    Contract.Requires(q != null);

    var concat = p + q;

    Contract.Assert(concat != null);
    Contract.Assert(concat.Length > 0);

    return concat;
}
```
Preconditions

```csharp
public string Concat(string p, string q)
{
    Contract.Requires(p != null);
    Contract.Requires(q != null);
    // …
}

public string MyConcat()
{
    return Concat("Ciao", null);
}
```

Concat “believes” (assumes) those preconditions

- “Ciao” != null
- null != null
public double Abs(double x)
{
    Contract.Ensures(
        Contract.Result<double>(() >= 0);
    
    if (x < 0)
    {
        return -x;
    }
    return x;
}

public double Sqrt(double z)
{
    return Math.Sqrt(Abs(z));
}
Algorithm: High level

For each assembly A, class C, method M

1. Extract the proof obligations
   • What should I prove?
2. Run the analyses
   • Discover facts on the program
3. Use the facts to prove the proof obligations
   • If not, do something else...
Inferred facts

- Heap structure
- Null/Not-Null
- Numerical values
  - Ranges, relations, floating points ...
- Enum values
- Array/Collection contents
- ...

Ranges, relations, floating points ...

...
public string TrimSuffix(string s, string suffix)
{
    Contract.Requires(s != null);
    Contract.Requires(suffix != null);

    string res = s;

    while (res.EndsWith(suffix))
    {
        int len = res.Length - suffix.Length;
        res = res.Substring(0, len);
    }

    Contract.Assert(res != null);

    return res;
}
Associate to each variable an element of

Null  T  NotNull

⊥  ⊥
public string TrimSuffixWPO(string s, string suffix)
{
    Contract.Requires(s != null);
    Contract.Requires(suffix != null);

    string res = s;

    Contract.Assert(res != null);
    while (res.EndsWith(suffix))
    {
        Contract.Assert(res != null);
        Contract.Assert(suffix != null);
        int len = res.Length - suffix.Length;
        Contract.Assert(res != null);
        res = res.Substring(0, len);
        Contract.Assert(res != null);  // Postcondition of res
    }

    Contract.Assert(res != null);

    return res;
}
public string TrimSuffixWPO(string s, string suffix)
{
    Contract.Requires(s != null);
    Contract.Requires(suffix != null);

    string res = s;
    Contract.Assert(res != null);
    while (res.EndsWith(suffix))
    {
        Contract.Assert(res != null);
        Contract.Assert(suffix != null);
        int len = res.Length - suffix.Length;
        Contract.Assert(res != null);
        res = res.Substring(0, len);
        Contract.Assert(res != null); // Postcondition of res
    }
    Contract.Assert(res != null);

    return res;
}
What we did?

- We over-approximated the semantics
- We kept the concrete specification
Example: Numerical analysis

```csharp
static public int GCD(int x, int y)
{
    Contract.Requires(x > 0);
    Contract.Requires(y > 0);

    Contract.Ensures(Contract.Result<int>() > 0);

    while (true)
    {
        if (x < y)
        {
            y %= x;
            if (y == 0)
                return x;
        }
        else
        {
            x %= y;
            if (x == 0)
                return y;
        }
    }
}
```

We need numerical reasoning
We need to infer loop invariants
Numerical Analysis

\[ 0 \leq \text{index} < \text{array.Length} \]

- **Intervals**
  - \( O(n) \)
  - \( a \leq x \leq b \)
  - No 😞

- **Pentagons**
  - \( O(n) \)
  - \( a \leq x \leq b \) & \( x < y \)
  - Yes 😊

- **Octagons**
  - \( O(n^3) \)
  - \( \pm x \pm y \leq a \)
  - Yes 😊

- **Polyhedra**
  - \( O(2^n) \)
  - \( \Sigma a_i x_i \leq b \)
  - Yes 😊
Approximate each variable with a range 
\[ [a, b] \text{ where } a, b \in \mathbb{Z} \cup \{ +\infty, -\infty \} \]

More complex in reality because of
- Overflows
- Different Int types (16, 32, 64 bits, signed/unsigned)

Idea: Replace a value, a set of values with an interval
static public int GCD(int x, int y)
{
    Contract.Requires(x > 0);
    Contract.Requires(y > 0);
    Contract.Ensures(Contract.Result<int>() > 0);

    while (true)
    {
        if (x < y)
        {
            y %= x;
            if (y == 0)
            {
                return x;
            }
        }
        else
        {
            x %= y;
            if (x == 0)
            {
                return y;
            }
        }
    }
}
public void AllToZero(int[] a)
{
    for (int i = 0; i < a.Length; i++)
    {
        Contract.Assert(i >= 0);
        Contract.Assert(i < a.Length);

        a[i] = 0;
    }
}
Intervals keep only numerical information
No symbolic information
  Ex. \( i < a.\text{Length} \)
No relations
Intervals are an example of a non-relational domain
  Non-null is non-relational too
Capture properties in the form of

\[ x \in [a, b] \land x < y \]

- \( x, y \) variables
- \( a, b \) constants

Elements are pairs of maps

\((\text{Var} \rightarrow \text{Intv}) \times (\text{Var} \rightarrow \wp(\text{Var}))\)

Information is propagated

“reduction”
Subpolyhedra

Needed for more complex examples

\[ \sum a_i x_i \leq k \iff \sum a_i x_i = \beta \land \beta \leq k \]

Introduce a slack variable \( \beta \)

Reduced product of

Intervals

Scalable, fast...

Linear Equalities

Precise join, fast ...

Challenge: Have a precise Join
public void Init(int N)
{
    Contract.Requires(N > 0);

    int[] a = new int[N];
    int i = 0;

    while (i < N)
    {
        a[i] = 222;
        i = i + 1;
    }

    Contract.Assert(\( \forall k \in [0, N). a[k] == 222 \));
}

Challenge 1:
Effective handling of disjunction

If \( i == 0 \) then
- a not initialized
else if \( i > 0 \)
- \( a[0] == \ldots a[i] == 222 \)
else
impossible

Challenge 2:
No overapproximation (can be unsound)
(no hole, all the elements are initialized)
Our idea

- Precise and very very fast!
- Basis: Array segments

\[ \begin{align*}
0 & \leq i, 0 \leq k \\
0 & \leq i, 0 \leq k \\
0 & \leq i, 0 \leq k \\
i == k \\
i < N, k < N
\end{align*} \]
Example

Contract.Requires(N > 0);
int[] a = new int[N];

int i = 0;

assume i ≥ N
assume i < N

a[i] = 222;

j = i+1;

i -> _
j -> i
N -> N
Segment unification

Join

Can be empty segments! (Disjunction)
```
Contract.Requires(N > 0);
int[] a = new int[N];
```

```
int i = 0;
```

```
assume i ≥ N
```

```
assume i == N && N > 0
```

```
We visited all the elements in [0, N)
```

```
j = i+1;
```

```
Remove doubts:
```

```
0 222 i ? 0 N ?
```

```
And so on up to a fixpoint ...
```

```
0 222 i ? 0 N
```

```
i -> _
j -> i
N -> N
```
Algorithm: High level

For each assembly A, class C, method M

1. Extract the proof obligations
   • What should I prove?
2. Run the analyses
   • Discover facts on the program
3. Use the facts to prove the proof obligations
   • If not, do something else...
We inferred many facts on the program
We use those to prove assertions
Algorithm:
   For each assertion $a$ at program point $p$
      For each set of facts $F$
         Check if $F(p)$ implies $a$:
            True: It is always ok
            False: It is always not ok
            Bottom: The assertion is never reached
            Top: We do not know
Why do we get Top?

- The analysis is not precise enough
  - Abstract domain not precise
    - Widening loses too many constraints
  - Algorithmic properties
  - Implementation bug
  - Incompleteness

- Some contract is missing
  - Precondition or Postcondition
  - Object-invariant

- The assertion is sometimes wrong
First analyze with “cheap” domains

- If check != Top
  
  Done!

- If check == Top
  
  Try a more precise domain

On the average great performance gains

- Persist analysis options in different runs
Disjunctions

(So far) Join approximates disjunction

- Compact representation

Sometimes not enough:

```csharp
public int Simple(bool b)
{
    int z;
    if (b)
        z = 12;
    else
        z = -12;
    return 1 / z;
}

public string Simple2(bool b)
{
    Contract.Ensure(
        Contract.Result<string>() == null
        || Contract.Result<string>().Length > 0);
    if (b)
        return null;
    else
        return "Ciao!";
}
```
The solution in Clousot

- Backward analysis
- The failing assertion is pushed back to the program

```csharp
public int Simple(bool b)
{
    int z;
    if (b)
        z = 12;
    else
        z = -12;
    return 1 / z;
}
```

- \( z : [12, 12] \)
- \( z \neq 0 \)
- \( z : [-12, -12] \)
- \( z \neq 0 \)
- \( z : (-\infty, +\infty) \)
- \( z \neq 0 \)
public string Simple2(bool b)
{
    Contract.Ensures(
        Contract.Result<string>() == null
        || Contract.Result<string>().Length > 0);

    if (b)
    {
        return null;
    }
    else
    {
        return "Ciao!";
    }
}
public void NonNull()
{
    string foo = null;
    for (int i = 0; i < 5; i++)
    {
        foo += "foo";
    }
    Contract.Assert(foo != null);
}
Contract
Inference
Programmers write all the contracts
  Even loop invariants?

Reality
  Should infer “evident” contracts
public int ZeroValues(int[] a)
{
    int count = 0;
    for (int i = 0; i < a.Length; i++)
    {
        if (a[i] == 0)
            count++;
    }
    return count;
}
The problem

- An inferred precondition should
  - Remove bad runs
  - Keep all the good runs
- Several algorithms in Clousot
  - Precision/cost tradeoff
Propagate a failing assertion the entry
If it respects the visibility rules,
then it is a precondition
Otherwise
Try to suggest as object invariant
  E.g. assertion on private fields on public method
Try to suggest as an assumption
  The programmer is making some implicit assumption
Have a method \( m \)

For each program point

Abstract state \( a \)

Approximate the concrete states at that point

Take the abstract state at the exit point of the method

```
public int HalfSum(int x, int y)
{
    Contract.Requires(x >= 0);
    Contract.Requires(y >= 0);

    return x + (y - x) / 2;
}
```

CodeContracts: Suggested postcondition: Contract.Ensures(Contract.Result<System.Int32>() >= 0);
Filter locals

Take into account inheritance rules

Remove redundant information

Ex. $x \geq 0, y == 0, x + y \geq 0$

Avoid suggesting existing precondition

...
Message prioritization
We should report it to the programmer

It can be:
- a real bug?
- a false positive?

In general impossible to tell
- Undecidability of the analysis

Should sort all the messages
- The one most likely to be bugs at the top
1. Warning partitioning

- Partition warnings in classes
  - Contract violation
  - Non-null
  - Arrays
  - Overflows
  - ...

- Assign a *fixed* reward $R$ to each class
  - $R \in [\mathbb{C} \rightarrow \mathbb{N}]$

- The highest the reward the more important
2. Scale rewards with outcome

- **False** = 1.0 * R(c)
  - Important
  - Always wrong...
- **Bottom** = 0.75 * R(c)
  - Unreached, we wanted it?
- **Top** = 0.50 * R(c)
  - So many ...
- **True** = 0 * R(c)
  - Don’t care
3. Scale rewards with info

- Proof obligation $p$ contains
  - Variables from parameters
  - Variables result of a method call
  - ...

- The scale the reward

```csharp
public foo(int z)
{
    // ...
    Contract.Assert(z + x > 0);
    var f = Add();
    Contract.Assert(f != null);
    // ...
}
```
Caching
Caching?

- At design time, few changes between two builds
- Avoid re-analysis by caching

Algorithm
- Construct the CFG for the method
  - Includes contracts explicit/inferred
- Hash the CFG
- If in the DB, just report the same output
- Otherwise, re-analyze the method
  - Save it in the cache
Floating points...
Computers & numbers

- Computers crunch numbers
- But computer numbers are not mathematical ones!
- Plethora of Int*
  - Int8, Int16, Int32, Int64, BigInt
  - UInt8, UInt16, UInt32, UInt64
- And even more fun:
  - Floating points
Contract.Assert(Double.Epsilon != Double.Epsilon * Double.Epsilon);

Contract.Assert(1.0d == 1.0d + Double.Epsilon);

Contract.Assert(1000000000d == 1000000000d + Double.Epsilon);
\[(x + k) - (x - k) \equiv 2k\ ?\]

double x, y, z, r;

x = 1.0000000019e+38d;
y = x + 1.0e21d;
z = x - 1.0e21d;
r = y - z;

Contract.Assert(r == 2.0e21d);

Assert is false!
Contract.Assert(0.0 == 0.0);

Contract.Assert(Double.NegativeInfinity == Double.NegativeInfinity);

Contract.Assert(Double.NaN == Double.NaN);

Assert is true!

Assert is false!!!!!!!

1/0 is NaN
Sqrt(-1) is NaN
\[ K \neq 0 \implies (x + k) - (x - k) \neq 0 \]

double x, y, z, r;

x = 1.000000019e+38d;
y = x + 1.0e21d;
z = x - 1.0e21d;
r = y - z;

Contract.Assert(r == 0.0d);

Assert is true!
public class Point
{
    public double X, Y;

    Point(double x, double y)
    {
        this.X = x;
        this.Y = y;
    }
}

public CreatePoint(double x)
{
    var p = new Point(x, x);

    Contract.Assert(x == p.X);
}
A double is a synonym for Float64

A Float64 is represented
- in RAM with 64 bits
- in the CPU with 80 bits!!!
Conclusions...
Specify code with code
- No change to the build environment
- Part of .NET v4
Documentation generation
Runtime checking
Static checking
- Based on abstract interpretation
- Predicatable, tunable, scalable, automatic!!!!

Try it today!!!!!