**CodeContracts?**

Specify code with code

```csharp
public virtual int Calculate(object x) {
    Contract.Requires(x != null);
    Contract.Ensures(Contract.Result<int>() >= 0);
}
```

**Advantages**
- **Language agnostic**
  - *No new language/compiler …*
- **Leverage existing tools**
  - *IDE, Compiler …*

**Disadvantages**
- **Lost beauty**
CodeContracts tools

Documentation generator
  MSDN-like documentation generation
  VS plugin – tooltips as you write

Runtime checking
  Postconditions, inheritance …
  Via binary rewriting

Static checking
  Based on abstract interpretation
  This talk!!!
**CodeContracts impact**

API .NET standard since v4

Externally available
- ~100,000 downloads
- Active forum (>7,700 msg)
- Book chapters, blogs …

Internal and External adoption
- Mainly professional
- A few university courses

Publications, talks, tutorials
- Academic, Programmers conferences
Let’s demo!
Why abstract interpretation?

Traditional verification workflow

Verification tool based on
Weakest preconditions
Symbolic execution
Model checking
Fix the code?

Understand the warnings

Add missing specifications
  Pre/Post-conditions, Object/Loop invariants
  Assumptions
    Environment, external code, OS ...

Verifier limits
  Incompleteness....

Fix bugs?
  Tough task verifying a program with bugs...

Tedious and expensive process
Reality is a little bit different

New features, regressions, refactoring …

Help programmer, not drown her

“Verification” is only one facet
Should support correct SW development
Why Abstract interpretation?

Focus on properties of interest
Few programmers interested in ∀∃∀…
Null dereferences a lot more relevant!

Programmer friendly, Tunable, Precise
Easy to explain what’s wrong
Properties known ahead of time
“HowReverse engineered” by some users

Infer, not deduce or search
Loop invariants, contracts, code fixes …
The power of inference

```csharp
public int Max(int[] arr)
{
    var max = arr[0];
    for (var i = 1; i < arr.Length; i++)
    {
        var el = arr[i];
        if (el > max)
            max = el;
    }
    return max;
}
```

Annotations

```csharp
public int Max(int[] arr)
{
    var max = arr[0];
    for (var i = 1; i < arr.Length; i++)
    {
        var el = arr[i];
        if (el > max)
            max = el;
    }
    return max;
}
```

Clousot

1. CodeContracts: Suggested requires: Contract.Requires(arr != null);
2. CodeContracts: Suggested requires: Contract.Requires(0 < arr.Length);
3. CodeContracts: Suggested ensures: Contract.Ensures(Contract.ForAll(0, arr.Length, _k_ => arr[_k_] <= Contract.Result<int>( )));
int BinarySearch(int[] array, int value)
{
    Contract.Requires(array != null);
    var inf = 0;
    var sup = array.Length - 1;

    while (inf <= sup)
    {
        var index = (inf + sup) / 2;
        var mid = array[index];

        if (value == mid) return index;
        if (mid < value) inf = index + 1;
        else sup = index - 1;
    }
    return -1;
}
Scaling up

Real code bases are huge

The promise of automatic proving has been a holy grail for a long time – Even when I was in academia it was possible to reason about toy programs. The main fear that I have is that the current CC engine is not designed to scale enough to be able to handle the huge assembly that we currently have: The snapshot currently on my machine contains 15,596 classes, featuring a total of 191,522 C++ methods. In this light I have the following questions:

Turns out they were ~700K methods
  Overloads, automatically generated

Analysis took 3h on a Xeon
  Output: 116Mb text file
  Cache file: 610Mb

Found new bugs
Scaling up

Real code bases are huge

Should cope with it

Myths:

“I am modular, hence I scale up”
“I analyze in < 1sec, hence I scale up”
Clousot on the huge assembly

No inter-method inference

Quadratic in #methods

Why???
   GC?
   DB?

If the app runs long enough, the GC/DB complexity matters

Intra-method can be costly
   Nested loops, goto ...

\[ y = 14.17x^2 + 228.64x + 434.02 \]
Scaling up: Our experience

Avoid complexity
∀ costly corner case, ∃ user who will hit it

Be incremental
Analysis time should be proportional to changes

Reduce annotation overhead
Avoid boredom of trivial annotations
Save programmer time

Prioritize
Not all the warnings are the same…
Clousot Overview

![Diagram showing the process flow of Clousot with layers for Inference, Checking, and Reporting, leading to various programming languages like C#, VB, F#, X++, and more.]
Clousot Main Loop

Read Bytecode, Contracts
∀ assembly, ∀ module, ∀ type, ∀ method
  Collect the proof obligations
  Analyze the method, discover facts
  Check the facts
  Report outcomes, suggestions, repairs
  Propagate inferred contracts
Examples of Proof Obligations

```java
public int Div(int x, int y)
{
    return x / y;
}
```

```java
public int Abs(int x)
{
    Contract.Ensures(Contract.Result<int>() >= 0);
    return x < 0 ? -x : x;
}
```

1. `y != 0`
2. `x != MinValue || y != -1`
3. `x != MinValue`
4. `result >= 0`
Proof obligations collection

In theory, collect all the proof obligations
  Language: non-null, div-by-0, bounds …
  User supplied: contracts, assert …

In practice, too many language obligations
  Non-null, div-by-0, various overflows, array/buffer overruns, enums, floating point precision ….

Let the user chose and focus
Clousot Main Loop

- **Read** Bytecode, Contracts
- ∀assembly, ∀module, ∀type, ∀method
  - Collect the proof obligations
- **Analyze** the method, discover **facts**
- Check the facts
- Report outcomes, suggestions, repairs
- Propagate inferred contracts
Static Analysis

Goal: Discover facts on the program

Challenges:

Precise analysis of IL

Compilation lose structure

Which properties are interesting?

Which abstract domains should we use?

How we make them practical enough?

Performance

Usability

E.g. No templates
precise IL analysis

```csharp
private int f;
int Sum(int x) { return this.f + x; }
```

```
.method public hidebysig instance int32 Sum(int32 x) cil managed
{
  .maxstack 2
  .locals init (  
      [0] int32 CS$1500000
  )
  L_0000: nop
  L_0001: idarg.0
  L_0002: ldfld int32 Bag.NonNegativeList.f
  L_0007: idarg.1
  L_0008: add
  L_0009: stloc.0
  L_000a: br.s L_000c
  L_000c: ldloc.0
  L_000d: ret

  sv11 (13) = ldarg this
  sv13 (15) = ldfld Bag.NonNegativeList.f sv11 (13)
  sv8 (10) = ldarg x
  sv22 (24) = sv13 (15) Add sv8 (10)
  ret sv22 (24)
}
```
Which Abstract Domains?

Which properties?
Exploratory study inspecting BCL sources
   Existing parameter validation
      Mainly Non-null, range checking, types
      Types no more issue with Generics introduction
Well studied problems
   Plenty of numerical abstract domains
      Intervals, Octagons, Octahedra, Polyhedra ...
Problem solved??
Myth

“For NaN checking only one bit is required!”

```csharp
public double Sub(double x, double y)
{
    Contract.Requires(!Double.IsNaN(x));
    Contract.Requires(!Double.IsNaN(y));
    Contract.Ensures(!Double.IsNaN(Contract.Result<double>()));

    return x - y;
}
```

\[ -\infty - -\infty = \text{NaN} \]
Myth (popular in types)

"I should prove x != null, so I can simply use a non-null type system"

```java
public void NonNull()
{
    string foo = null;

    for (int i = 0; i < 5; i++)
    {
        foo += "foo";
    }

    Contract.Assert(foo != null);
}
```

Need numerical information to prove it!
Numerical domains in Clousot

Numerical information needed everywhere
  Ranges, enums, ∀/∃, contracts, code repairs …

Core of Clousot

Several new numerical abstract domains
  DisIntervals, Pentagons, SubPolyhedra …

  Infinite height, no finite abstraction

  Combined by reduced product

Incremental application

Validated by experience
∀/∃ abstract domain

Instance of FunArray (POPL’11)
Discover collection segments & contents

```csharp
public int Max(int[] arr)
{
    var max = arr[0];
    for (var i = 1; i < arr.Length; i++)
    {
        var el = arr[i];
        if (el > max) max = el;
    }
    return max;
}
```

Compact for:
∀ j. 0 ≤ j < i: arr[j] ≤ max ∧
∃ k. 0 ≤ k < i: a[k] = max ∧
i ≤ arr.Length ∧
1 ≤ i
Other abstract domains

Heap, un-interpreted functions
  Optimistic parameter aliasing hypotheses

Non-Null
  A reference is null, non-null, non-null-if-boxed

Enum
  Precise tracking of enum variables (ints at IL)

Intervals of floats, actual float types
  To prove NaN, comparisons

Array purity

...
Clousot Main Loop

Read Bytecode, Contracts
∀assembly, ∀module, ∀type, ∀method
  Collect the proof obligations
  Analyze the method, discover facts
Check the facts
Report outcomes, suggestions, repairs
Propagate inferred contracts
Checking

For each proof obligation \( \langle pc, \phi \rangle \)
   Check if Facts@pc \( \models \phi \)

Four possible outcomes
   True, correct
   False, definite error
   Bottom, assertion unreached
   Top, we do not know

In the first 3 cases we are happy
Why Top?

The analysis is not precise enough
  Abstract domain not precise
    *Re-analyze with more precise abstract domain*

Algorithmic properties
Implementation bug
Incompleteness

Some contract is missing
  *Pre/Postcondition, Assumption, Object-invariant*

The assertion is sometimes wrong (bug!)
Can we repair the code?
Dealing with Top

Every static analysis has to deal with Tops a.k.a. warnings

Just report warnings: overkilling

Explain warnings: better
Still expensive, programmer should find a fix
Ex. no inter-method inference:

Checked 2 147 956 assertions: 1 816 023 correct 331 904 unknown 29 false
Inspecting 1 warning/sec, 24/24: 230 days

Suggest code repairs: even better
But, there still we be warnings: rank & filter
Clousot Main Loop

Read Bytecode, Contracts
∀assembly, ∀module, ∀type, ∀method
  Collect the proof obligations
  Analyze the method, discover facts
  Check the facts
Report outcomes, suggestions, repairs
Propagate inferred contracts
Precondition inference

What is a precondition?

\{P\} \ C \ \{Q\}

So we have a solution?

\{wp[\ C]Q\} \ C \ \{Q\}

WP rule out good runs

Loops are a problem
Loop invariant ⇒ No “weakest” precondition
Inference of sufficient preconditions

public static void WPex(int[] a)
{
    for (var i = 0; i <= a.Length; i++)
    {
        a[i] = 11;
        if (NonDet()) return;
    }
}
Necessary conditions

Our approach: Infer necessary conditions

Requirements
   - No new run is introduced
   - No good run is eliminated
   - Therefore, only bad runs are eliminated

Analyses infer $B_{pc}$, necessary condition at pc
   - If $B_{pc}$ does not hold at pc, program will crash later
   - $B_{entry}$ is necessary precondition

Leverage them to code repairs
Verified Code Repairs

**Semantically justified** program repair

- Contracts
  - Pre/post-conditions, object invariants inference
- Bad initialization
- Guards
- Buffer overrun
- Arithmetic overflow

... ...

**Inferred** by static analysis

**Extracted** by abstract states
Some data

<table>
<thead>
<tr>
<th>Library</th>
<th>Asserts Validated</th>
<th>Warnings</th>
<th>Repairs</th>
<th>Time</th>
<th>Ass. w. repairs</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>system.windows.forms</td>
<td>154,845</td>
<td>136,667</td>
<td>18,178</td>
<td>24,048</td>
<td>1:18</td>
<td>16,498</td>
</tr>
<tr>
<td>mscorlib</td>
<td>110,236</td>
<td>97,107</td>
<td>13,129</td>
<td>26,166</td>
<td>0:59</td>
<td>10,576</td>
</tr>
<tr>
<td>system</td>
<td>97,617</td>
<td>85,934</td>
<td>11,683</td>
<td>15,120</td>
<td>0:53</td>
<td>9,518</td>
</tr>
<tr>
<td>system.core</td>
<td>34,031</td>
<td>29,569</td>
<td>4,462</td>
<td>6,914</td>
<td>0:26</td>
<td>3,599</td>
</tr>
<tr>
<td>custommarshaler</td>
<td>439</td>
<td>376</td>
<td>61</td>
<td>65</td>
<td>0:00</td>
<td>48</td>
</tr>
<tr>
<td>Total</td>
<td>397,168</td>
<td>349,655</td>
<td>47,513</td>
<td>47,513</td>
<td>3:36</td>
<td>40,239</td>
</tr>
</tbody>
</table>

Suggest a repair \(\geq 4/5\) of times

If applied, precision raises 88\% \(\rightarrow 9\%\)

Precision: \% of validated assertions

Annotated libraries: usually \(\sim 100\%\)
And for the other Tops?

Make buckets
  Related warnings go together

Rank them
  Give each warning a score
    \( f(\text{Outcome}, \text{warning kind}, \text{semantic info}) \)

Enable suppression via attribute
  Particular warning, family of warnings
  Preconditions at-call, object invariants
  Inherited postconditions
  …
More?

Integrate in Roslyn CTP
Design time warnings, fixes, semantic refactoring, deep program understanding

```csharp
public int Decrement(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensures(Contract.Result<int>() >= 0);

    while (x != 0) x--;
}
```

```csharp
public int Decrement(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensures(Contract.Result<int>() >= 0);
    x = NewMethod(x);

    return x;
}
```

```csharp
int NewMethod(int x)
{
    Contract.Requires(0 <= x);
    Contract.Ensures(Contract.Result<int>(System.Int32.Zero) == 0);

    while (x != 0) x--;
    return x;
}
```
Conclusions

“Verification” only a part of the verified software goal

Other facets
  Scalable & incremental
  Programmer support & aid
    Inference
    Automatic code repairs
    IDE support
      Refactoring, focus verification efforts

Try Clousot today!
Available in VS Gallery!

VS 2012 Integration
Runtime checking
Documentation generation
Post-build static analysis
Scale via team shared SQL DB