CodeContracts & Clousot

Francesco Logozzo - Microsoft

Mehdi Bouaziz – ENS

CodeContracts?

Specify code with code

```
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

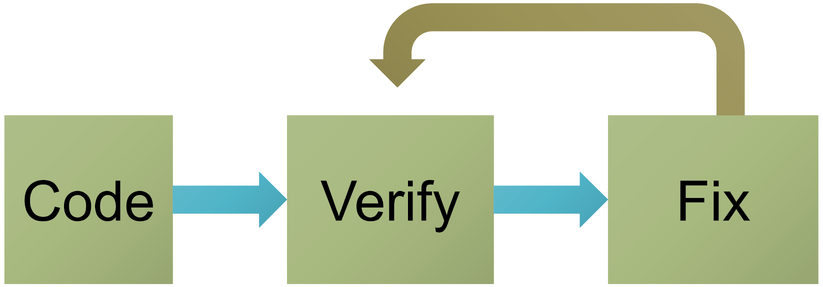
Publications, talks, tutorials

- Academic, Programmers conferences

Let’s demo!

Why abstract interpretation?

Traditional verification workflow



Code

Verify

Fix

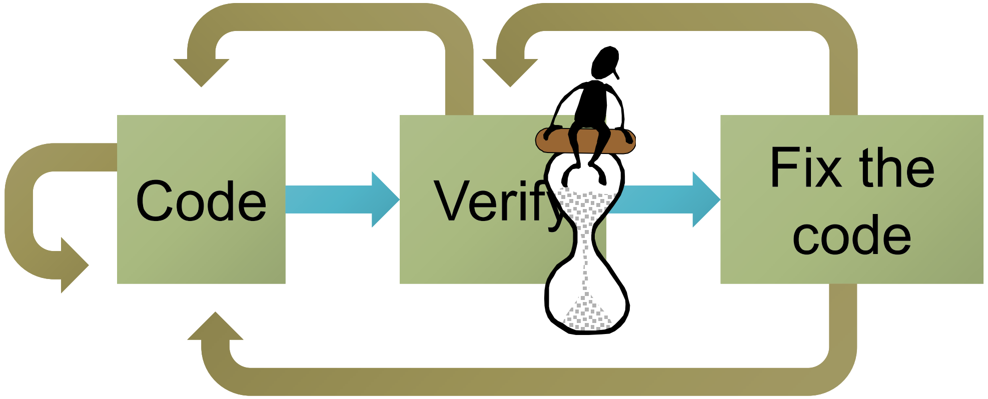
Verification tool based on

- 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
    - *“Reverse engineered” by some users*
- Infer, not deduce or search
  - Loop invariants, contracts, code fixes …

The power of inference

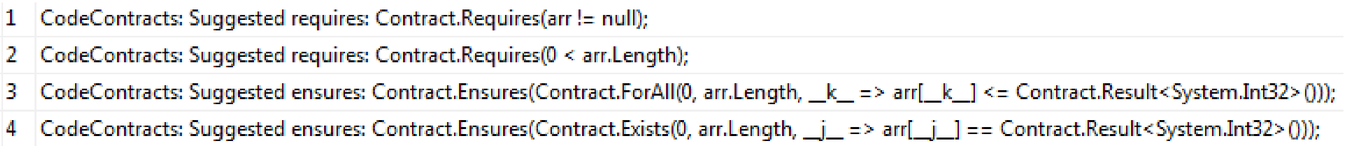
```
public int Max(int[] arr)
{
 Contract.Requires(arr != null);
 Contract.Requires(arr.Length > 0);
 Contract.Ensures(Contract.ForAll(0, arr.Length, j => arr[j] <= Contract.Result<int>()));
 Contract.Ensures(Contract.Exists(0, arr.Length, j => arr[j] == Contract.Result<int>()));

 var max = arr[0];
 for (var i = 1; i < arr.Length; i++)
 {
   Contract.Assert(1 <= i);
   Contract.Assert(Contract.ForAll(0, i, j => arr[j] <= max));
   Contract.Assert(Contract.Exists(0, i, j => arr[j] == max));
   var el = arr[i];
   if (el > max)
     max = el;
   }

 return max;
}
```

```
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

Clousot

```
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;
}
```



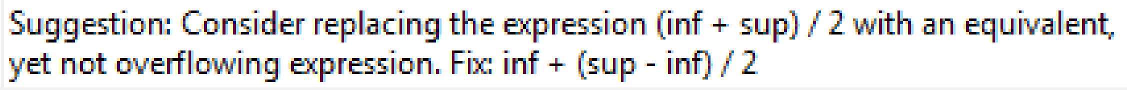
```
1  CodeContracts: Suggested requires: Contract.Requires(arr != null);
2  CodeContracts: Suggested requires: Contract.Requires(0 < arr.Length);
3  CodeContracts: Suggested ensures: Contract.Ensures(Contract.ForAll(0, arr.Length, __k__ => arr[__k__] <= Contract.Result<System.Int32>()));
4  CodeContracts: Suggested ensures: Contract.Ensures(Contract.Exists(0, arr.Length, __j__ => arr[__j__] == Contract.Result<System.Int32>()));
```

Code Repairs

```
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;
}
```

Exploit the inferred loop invariant

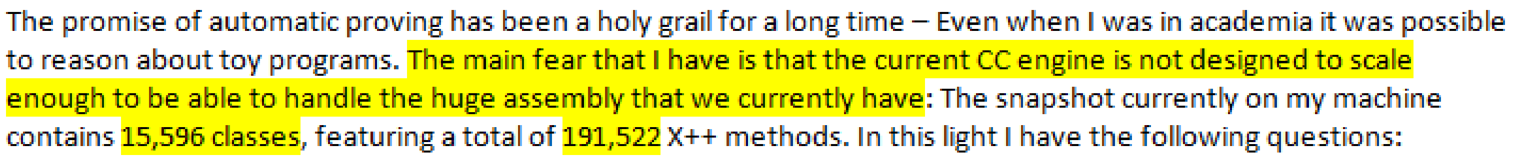
Clousot



Suggestion: Consider replacing the expression (inf + sup) / 2 with an equivalent, yet not overflowing expression. Fix: inf + (sup - inf) / 2

Scaling up

- Real code bases are huge

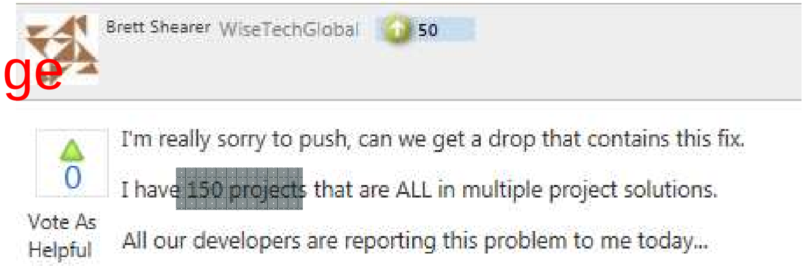


The promise of automatic proving has been a holy grail for a long time – Even when I was in academia it was possible to reason about toy programs. The main fear that I have is that the current CC engine is not designed to scale enough to be able to handle the huge assembly that we currently have: The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



I'm really sorry to push, can we get a drop that contains this fix.

I have 150 projects that are ALL in multiple project solutions.

All our developers are reporting this problem to me today...

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

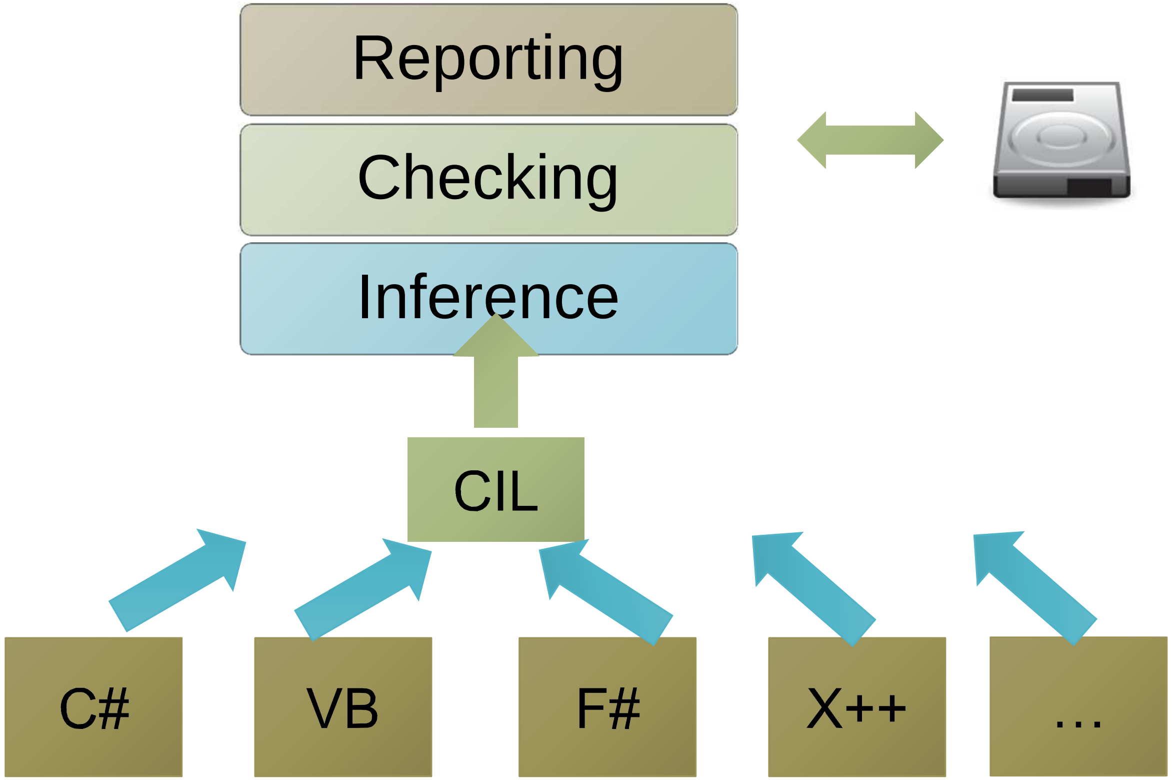
Intra-method can be costly

- Nested loops, goto …

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



Reporting

Checking

Inference

CIL

C#

VB

F#

X++

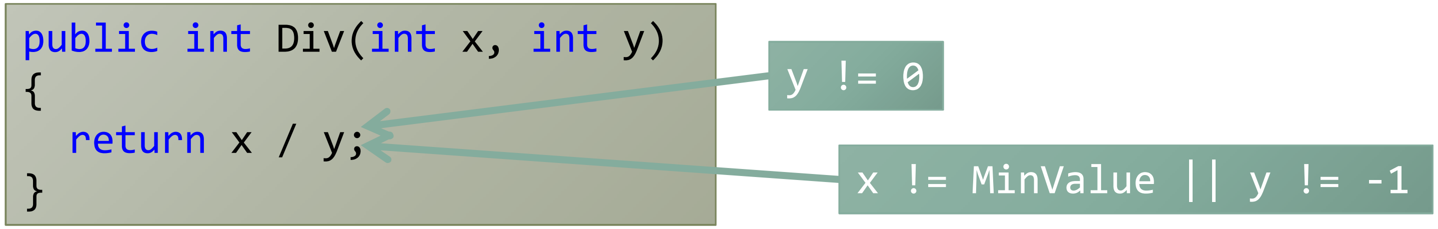
…

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

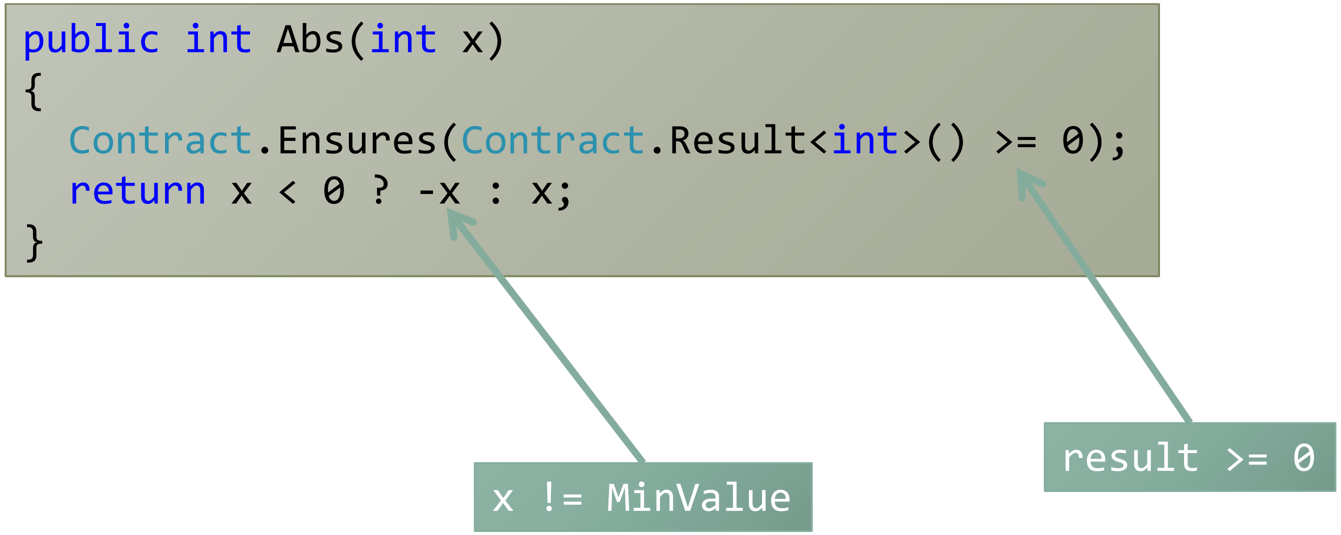
Examples of Proof Obligations



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

y != 0

x != MinValue || y != -1



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

x != MinValue

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

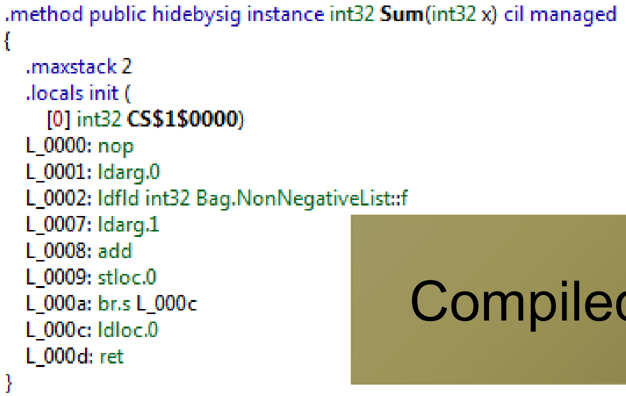
Goal: Discover facts on the program

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

```
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$1$0000)
   L_0000: nop
   L_0001: ldarg.0
   L_0002: ldfld int32 Bag.NonNegativeList::f
   L_0007: ldarg.1
   L_0008: add
   L_0009: stloc.0
   L_000a: br.s L_000c
   L_000c: ldloc.0
   L_000d: ret
}
```

Compiled

s0 = ldarg this
s0 = ldfld Bag.NonNegativeList.f s0
s1 = ldarg x
s0 = s0 Add s1
nop
ret s0

De-Stack

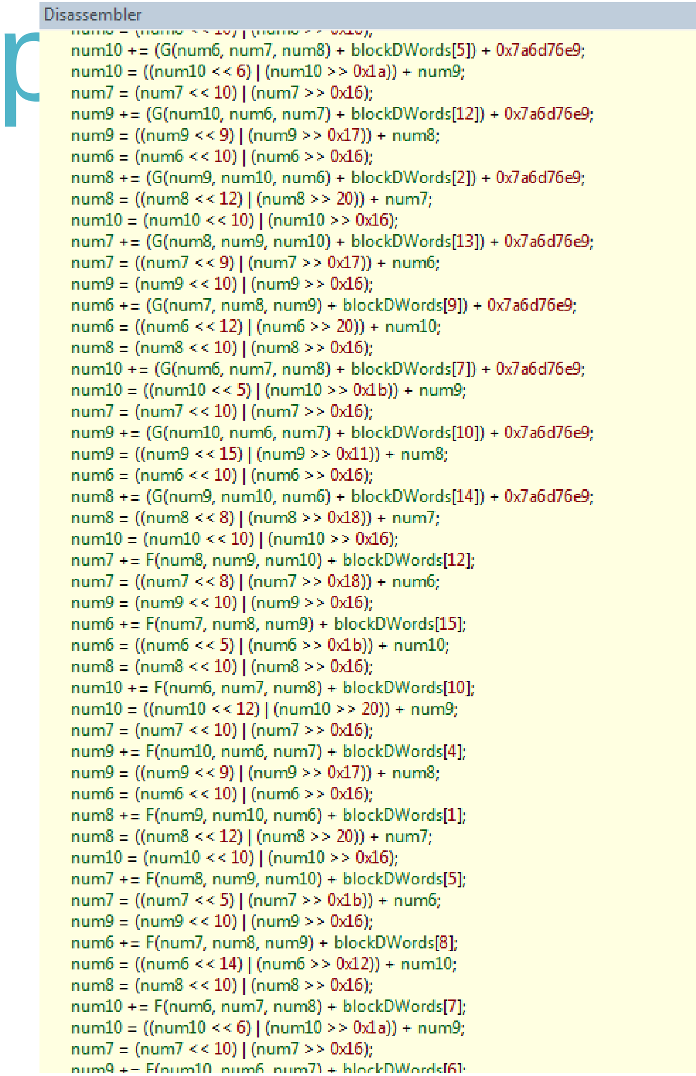
sv11 (13) = ldarg this
sv13 (15) = ldfld Bag.NonNegativeList.f sv11 (13)
sv8 (10) = ldarg x
sv22 (24) = **sv13 (15)** Add sv8 (10)
ret sv22 (24)

De-Heap

sv11 (13) = ldarg this
sv13 (15) = ldfld Bag.NonNegativeList.f sv11 (13)
sv8 (10) = ldarg x
sv22 (24) = sv13 (15) Add sv8 (10)
ret **(sv13 (15) Add sv8 (10))**

Exp. recovery

Exp co



```
num10 += (G(num6, num7, num8) + blockDWords[5]) + 0x7a6d76e9;
num10 = ((num10 << 6) | (num10 >> 0x1a)) + num9;
num7 = (num7 << 10) | (num7 >> 0x16);
num9 += (G(num10, num6, num7) + blockDWords[12]) + 0x7a6d76e9;
num9 = ((num9 << 9) | (num9 >> 0x17)) + num8;
num6 = (num6 << 10) | (num6 >> 0x16);
num8 += (G(num9, num10, num6) + blockDWords[2]) + 0x7a6d76e9;
num8 = ((num8 << 12) | (num8 >> 20)) + num7;
num10 = (num10 << 10) | (num10 >> 0x16);
num7 += (G(num8, num9, num10) + blockDWords[13]) + 0x7a6d76e9;
num7 = ((num7 << 9) | (num7 >> 0x17)) + num6;
num9 = (num9 << 10) | (num9 >> 0x16);
num6 += (G(num7, num8, num9) + blockDWords[9]) + 0x7a6d76e9;
num6 = ((num6 << 12) | (num6 >> 20)) + num10;
num8 = (num8 << 10) | (num8 >> 0x16);
num10 += (G(num6, num7, num8) + blockDWords[7]) + 0x7a6d76e9;
num10 = ((num10 << 5) | (num10 >> 0x1b)) + num9;
num7 = (num7 << 10) | (num7 >> 0x16);
num9 += (G(num10, num6, num7) + blockDWords[10]) + 0x7a6d76e9;
num9 = ((num9 << 15) | (num9 >> 0x11)) + num8;
num6 = (num6 << 10) | (num6 >> 0x16);
num8 += (G(num9, num10, num6) + blockDWords[14]) + 0x7a6d76e9;
num8 = ((num8 << 8) | (num8 >> 0x18)) + num7;
num10 = (num10 << 10) | (num10 >> 0x16);
num7 += F(num8, num9, num10) + blockDWords[12];
num7 = ((num7 << 8) | (num7 >> 0x18)) + num6;
num9 = (num9 << 10) | (num9 >> 0x16);
num6 += F(num7, num8, num9) + blockDWords[15];
num6 = ((num6 << 5) | (num6 >> 0x1b)) + num10;
num8 = (num8 << 10) | (num8 >> 0x16);
num10 += F(num6, num7, num8) + blockDWords[10];
num10 = ((num10 << 12) | (num10 >> 20)) + num9;
num7 = (num7 << 10) | (num7 >> 0x16);
num9 += F(num10, num6, num7) + blockDWords[4];
num9 = ((num9 << 9) | (num9 >> 0x17)) + num8;
num6 = (num6 << 10) | (num6 >> 0x16);
num8 += F(num9, num10, num6) + blockDWords[1];
num8 = ((num8 << 12) | (num8 >> 20)) + num7;
num10 = (num10 << 10) | (num10 >> 0x16);
num7 += F(num8, num9, num10) + blockDWords[5];
num7 = ((num7 << 5) | (num7 >> 0x1b)) + num6;
num9 = (num9 << 10) | (num9 >> 0x16);
num6 += F(num7, num8, num9) + blockDWords[8];
num6 = ((num6 << 14) | (num6 >> 0x12)) + num10;
num8 = (num8 << 10) | (num8 >> 0x16);
num10 += F(num6, num7, num8) + blockDWords[7];
num10 = ((num10 << 6) | (num10 >> 0x1a)) + num9;
num7 = (num7 << 10) | (num7 >> 0x16);
num9 += F(num10, num6, num7) + blockDWords[6];
```

`MDTransform` in mscorlib.dll
9000 straight line instructions

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!“

```
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;
}
```

-∞ -∞ =

NaN

Myth (popular in types)

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

```
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

```
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;
}
```

{0}	<= max, ∃= max	{i}	Top	{arr.Length}?

Compact for:

$\forall$ 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, ϕ $\rangle$

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 $\mathbb{B}_{pc}$, necessary condition at pc

- If $\mathbb{B}_{pc}$ does not hold at pc, program will crash later
- $\mathbb{B}_{entry}$ is necessary precondition

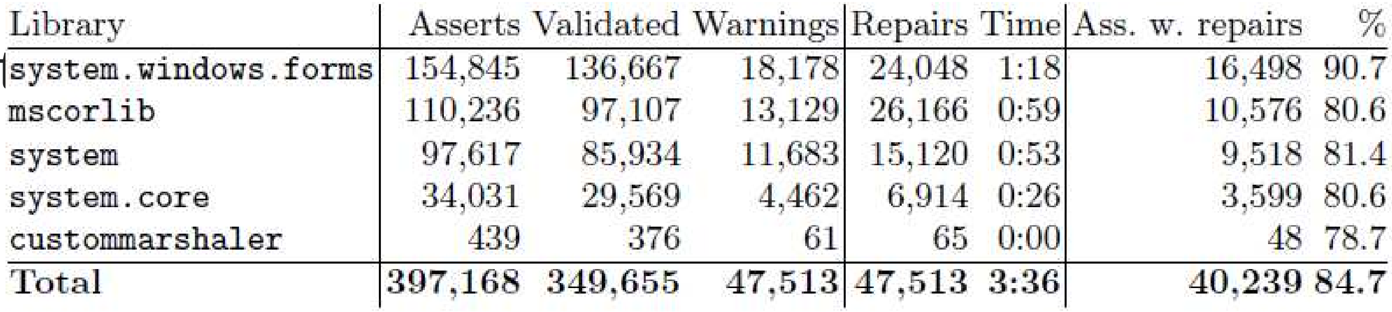
Leverage them to code repairs

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

Un-annotat



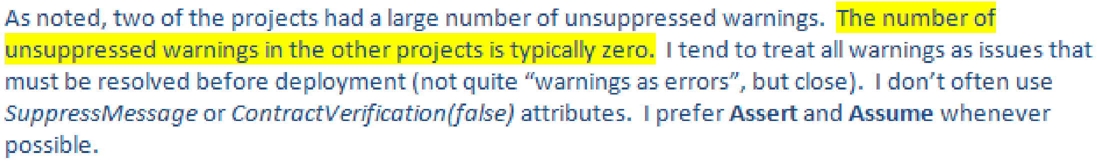
Library	Asserts	Validated	Warnings	Repairs	Time	Ass. w. repairs	%
`system.windows.forms`	154,845	136,667	18,178	24,048	1:18	16,498	90.7
`mscorlib`	110,236	97,107	13,129	26,166	0:59	10,576	80.6
`system`	97,617	85,934	11,683	15,120	0:53	9,518	81.4
`system.core`	34,031	29,569	4,462	6,914	0:26	3,599	80.6
`custommarshaler`	439	376	61	65	0:00	48	78.7
Total	**397,168**	**349,655**	**47,513**	**47,513**	**3:36**	**40,239**	**84.7**

Suggest a repair >4/5 of times

If applied, precision raises 88%→9

Precision: % of validated assertic

Annotated libraries: usually ~100%



As noted, two of the projects had a large number of unsuppressed warnings. The number of unsuppressed warnings in the other projects is typically zero. I tend to treat all warnings as issues that must be resolved before deployment (not quite “warnings as errors”, but close). I don’t often use *SuppressMessage* or *ContractVerification(false)* attributes. I prefer **Assert** and **Assume** whenever possible.

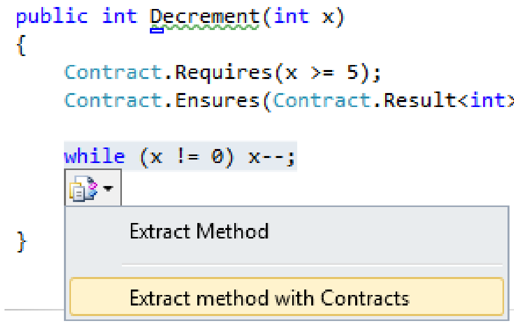
And for the other Tops?

- Make buckets
  - Related warnings go together
- Rank them
  - Give each warning a score
    - *f(Outcome, warning kind, semantic info)*
- Enable suppression via attribute
  - Particular warning, family of warnings
  - Preconditions at-call, object invariants
  - Inherited postconditions
  - …

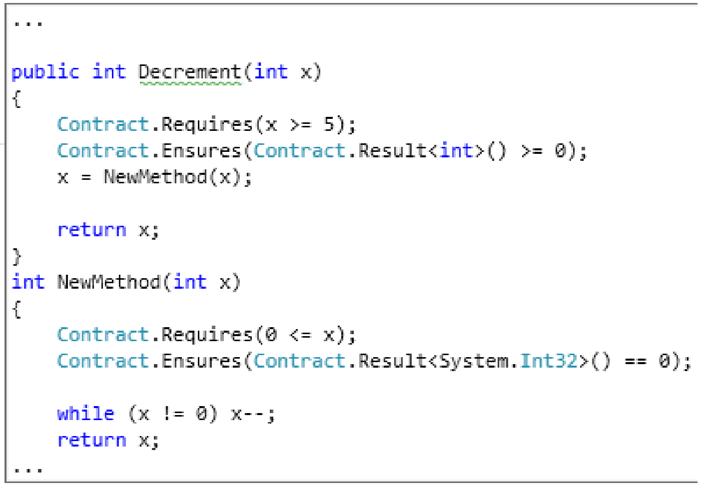
More?

Integrate in Roslyn CTP

- Design time warnings, fixes, semantic refactoring, deep program understanding



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

    while (x != 0) x--;

}
```



```
...

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

    return x;
}
int NewMethod(int x)
{
    Contract.Requires(0 <= x);
    Contract.Ensures(Contract.Result<System.Int32>() == 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!

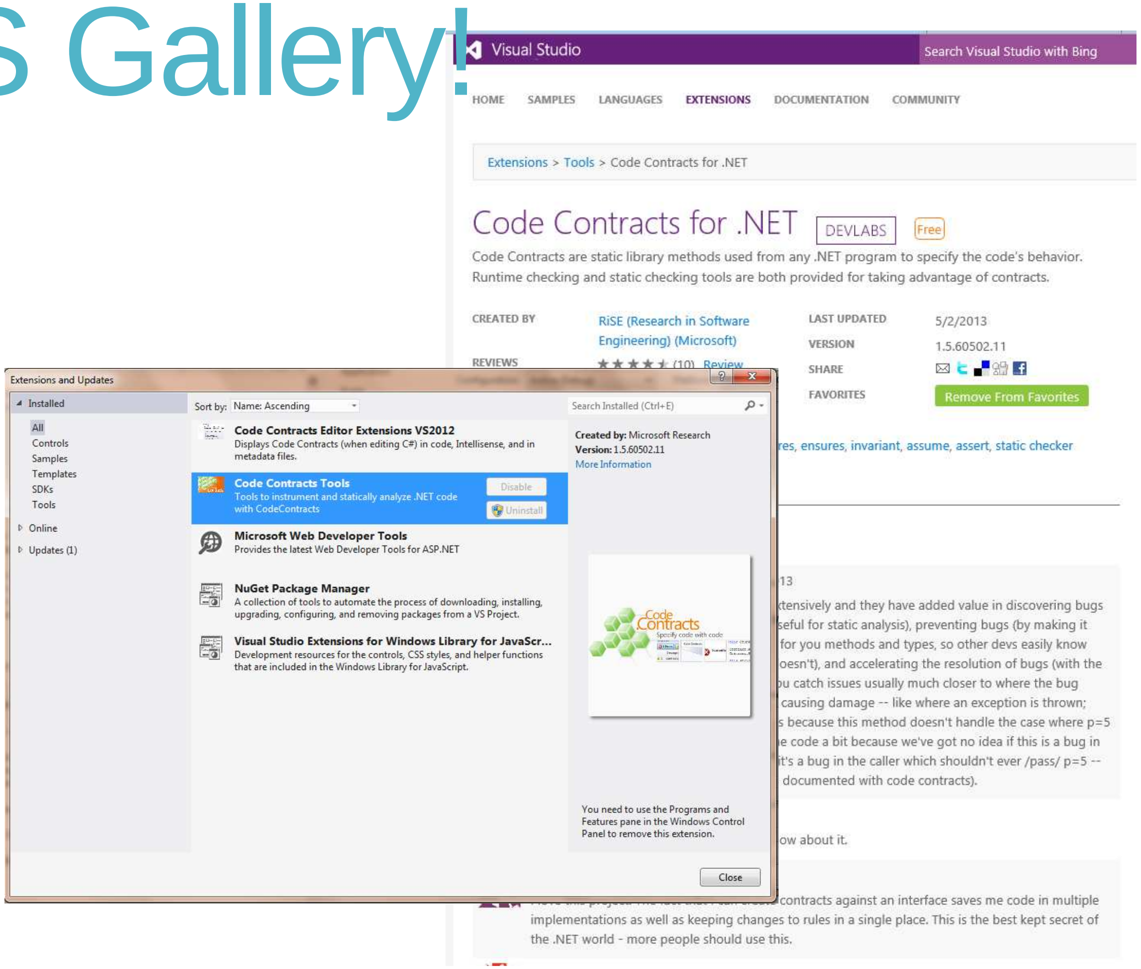
VS 2012 Integration

Runtime checking

Documentation generation

Post-build static analysis

Scale via team shared SQL DB



Code Contracts for .NET

Code Contracts are static library methods used from any .NET program to specify the code's behavior. Runtime checking and static checking tools are both provided for taking advantage of contracts.

CREATED BY RiSE (Research in Software Engineering) (Microsoft)

REVIEWS

LAST UPDATED 5/2/2013

VERSION 1.5.60502.11

SHARE

FAVORITES

Code Contracts Editor Extensions VS2012
Displays Code Contracts (when editing C#) in code, Intellisense, and in metadata files.

Code Contracts Tools
Tools to instrument and statically analyze .NET code with CodeContracts

Microsoft Web Developer Tools
Provides the latest Web Developer Tools for ASP.NET

NuGet Package Manager
A collection of tools to automate the process of downloading, installing, upgrading, configuring, and removing packages from a VS Project.

Visual Studio Extensions for Windows Library for JavaScr...
Development resources for the controls, CSS styles, and helper functions that are included in the Windows Library for JavaScript.

Created by: Microsoft Research
Version: 1.5.60502.11
More Information

You need to use the Programs and Features pane in the Windows Control Panel to remove this extension.

13
xtensively and they have added value in discovering bugs
seful for static analysis), preventing bugs (by making it
for you methods and types, so other devs easily know
oesn't), and accelerating the resolution of bugs (with the
u catch issues usually much closer to where the bug
causing damage -- like where an exception is thrown;
s because this method doesn't handle the case where p=5
e code a bit because we've got no idea if this is a bug in
it's a bug in the caller which shouldn't ever /pass/ p=5 --
documented with code contracts).

ow about it.

contracts against an interface saves me code in multiple implementations as well as keeping changes to rules in a single place. This is the best kept secret of the .NET world - more people should use this.