Lecture 12: Outline

- Static verification vs dynamic verification (of program code)
- A framework for static verification
- The Spec# annotation language and Boogie verification engine
- Other static verification languages/engines

Reminder: the use of formal models

- Requirements formalisation and "debugging"
- Code generation (after making models sufficiently detailed by, e.g., refinement steps)
- Model-based testing (generating test suites out of system models)
- Runtime (dynamic) or static verification of annotated program code. Model pieces are here incorporated into program code as special instructions/annotations

Static Verification vs Dynamic (Runtime) Verification

- Dynamic Verification: testing that a certain assertion/condition holds at a particular point during runtime execution
- Precompiling embedded assertions

  assert $x.length > 0$; ...code...

  into, for example,

  if not $(x.length > 0)$ **then**

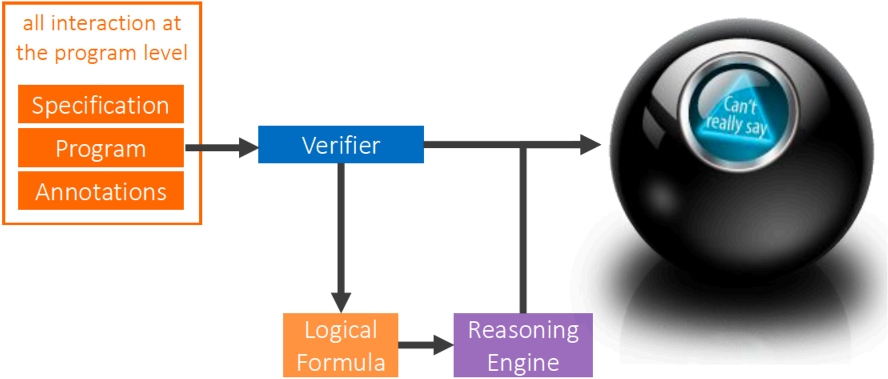
  raise *Assertion_Exception*("*Assertion condition ... violated*!");

  ...code...

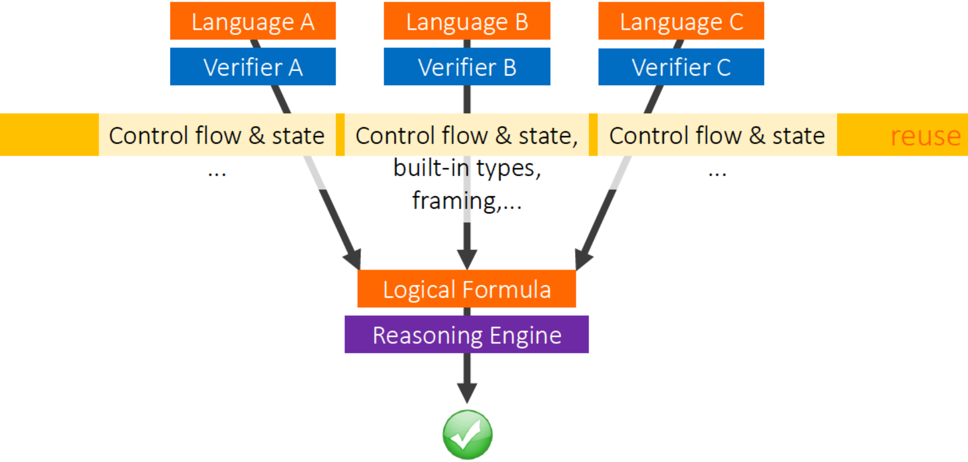
Static verification vs dynamic verification

- Static Verification: Analysing code together with embedded annotations during (pre)compilation time
- Static verification steps:
  - Precompiling of annotations together with code,
  - Generating intermediate formulas / verification conditions to verify,
  - Employing a verification engine to to check conditions and get answers,
  - Incorporating these answers as precompilation results (error messages/ warnings)
- Examples: JML+Esc/Java, Sparc-Ada, Spec#, code contracts, Sing#, Eiffel

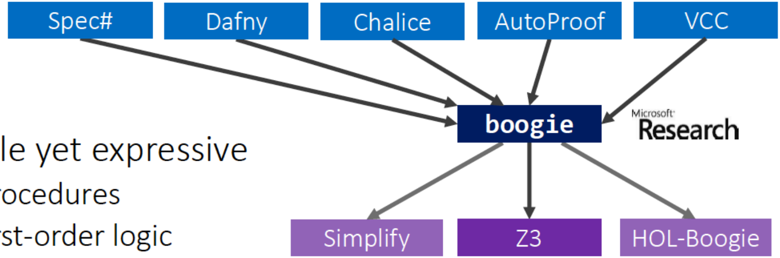
"Auto-active" verification



Verifying imperative programs



The Boogie intermediate verifier/verification language



Simple yet expressive

- procedures
- first-order logic
- integer arithmetic

How do we use Spec#?

- The programmer writes each class containing methods and their specification together in a Spec# source file (similar to Eiffel, similar to Java + JML)
- Invariants that constrain the data fields of objects may also be included
- We then run the verifier
- The verifier is run like the compiler—either from the IDE or the command line.
  - In either case, this involves just pushing a button, waiting, and then getting a list of compilation/verification error messages, if they exist.
  - Interaction with the verifier is done by modifying the source file.

Spec# at Microsoft

- http://research.microsoft.com/en-us/projects/specsharp/
- Spec# is a formal language for API contracts (influenced by JML, AsmL, and Eiffel), which extends C# with constructs for non-null types, preconditions, postconditions, and object invariants.
- Spec# is an extension of the object-oriented language C#. It extends the type system to include non-null types and checked exceptions. It provides method contracts in the form of pre- and postconditions as well as object invariants.

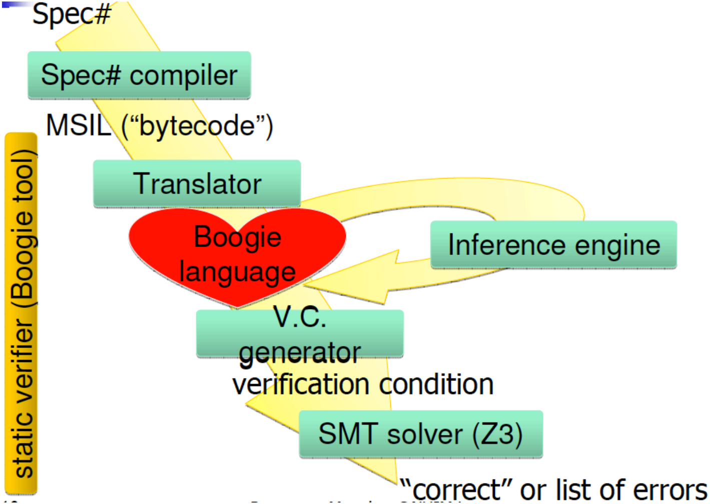
Spec# at Microsoft

- The Spec# compiler. Integrated into the Microsoft Visual Studio development environment for the .NET platform. Recently, incorporated as a part of the Code Contracts library;
- The compiler statically enforces non-null types, emits run-time checks for method contracts and invariants, and records the contracts as metadata for consumption by downstream tools;
- The Spec# static program verifier – Boogie. Generates logical verification conditions from a Spec# program. Internally, it uses an automatic theorem prover that analyses the verification conditions to prove the correctness of the program or find errors in it.

Static Verification

- Static verification checks all executions
- Spec# characteristics
  - Sound modular verification
  - Focus on automation of verification rather than full functional correctness of specifications
  - No termination verification
  - No verification of temporal properties
  - No arithmetic overflow checks

Spec# verifier architecture



SMT Solver Z3

- Z3 is a high-performance theorem prover being developed at Microsoft Research
- Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifiers
- Z3 is integrated with a number of program analysis, testing, and verification tools from Microsoft Research. These include: VCC, Spec#, Boogie, Pex, Yogi, Vigilante, SLAM, F7, F*, SAGE, VS3, FORMULA, and HAVOC

"Hello World" program in Spec#?

```
using System;
using Microsoft.Contracts;
public class Program
{
  public static void Main(string![]! args)
  {
    Console.WriteLine("Spec# says hello!");
     Console.Read();
  }
}
```

Non-Null Types

- Many errors in modern programs manifest themselves as null-dereference errors
- Spec# tries to eradicate all null dereference errors
- In C#, each reference type T includes the value null
- In Spec#, type T! contains only references to objects of type T (not null)

```
int []! xs;
```

declares an array called xs which cannot be null

Non-Null Types (cont.)

- If you decide that it's the caller's responsibility to make sure the argument is not null, Spec# allows you to record this decision concisely using an exclamation point
- Spec# will also enforce the decision at call sites returning Error: null is not a valid argument if a null value is passed to a method that requires a non null parameter

Non-Null Example

```
using System;
using Microsoft.Contracts;
class NonNull
{
 public static void Clear(int[] xs)
 {
    for (int i = 0; i < xs.Length; i++)
    {
          xs[i] = 0;
    }

 }

}
```

Where is the *possible null dereference*?

Non-Null Example (cont.)

```
using System;
using Microsoft.Contracts;
class NonNull
{
 public static void Clear(int[] xs)
 {
    for (int i= 0; i < xs.Length; i++) //Warning: Possible null dereference?
    {
          xs[i] = 0;   //Warning: Possible null dereference?
    }

 }

}
```

Non-Null Example (cont.)

```
using System;
using Microsoft.Contracts;
class NonNull
{
 public static void Clear(int[] ! xs)
 {
    for (int i = 0; i < xs.Length; i++)  // No Warning due to !
    {
          xs[i] = 0;   // No Warning due to !

    }

 }

}
```

Non-Null Example (cont.)

```
using System;
using Microsoft.Contracts;
class NonNull
{
 public static void Clear(int[] ! xs)
 {
    for (int i = 0; i < xs.Length; i++)
    {
          xs[i] = 0;
    }

 }

}
```

```
class ClientCode
{
   static void Main()
   {
        int[] xs = null;
         NonNull.Clear(xs);
   }
}
```

Non-Null Example (cont.)

```
using System;
using Microsoft.Contracts;
class NonNull
{
 public static void Clear(int[] ! xs)
 {
   for (int i = 0; i < xs.Length; i++)
   {
         xs[i] = 0;
   }

 }

}
```

"Null cannot be used where a non-null value is expected"

```
class ClientCode
{
   static void Main()
   {
         int[] xs = null;
          NonNull Clear(xs);
   }
}
```

Difference: assertions and assumptions!

Assert Statements

```
public class Assert
{
 public static void Main(string![]! args)
 {
  foreach (string arg in args)
  {    if (arg.StartsWith("Hello"))
       {        assert 5 <= arg.Length; // runtime check
                char ch = arg[2];
                Console.WriteLine(ch);
       }
  }
 }
}
```

<Assert.ssc>

Assert Statements (cont.)

```
public class Assert
{
  public static void Main(string![]! args)
  {
    foreach (string arg in args)
    {     if (arg.StartsWith("Hello"))
          {       assert 5 < arg.Length; // runtime error
                  char ch = arg[2];
                  Console.WriteLine(ch);
          }
    }
  }
}
```

Design by Contract

- Every public method has a precondition and a postcondition
- The precondition expresses the constraints under which the method will function properly
- The postcondition expresses what will happen when a method executes properly
- Pre- and postconditions are checked
- Preconditions and postconditions are side-effect free boolean-valued expressions - i.e. they evaluate to true/false and cant use ++

The Swap Contract

```
static void Swap(int[] a, int i, int j)
requires

modifies

ensures
{
    int temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

The Swap Contract(cont.)

```
static void Swap(int[]! a, int i, int j)
requires 0 <= i && i < a.Length;
requires 0 <= j && j < a.Length;
modifies a[i], a[j];
ensures a[i] == old(a[j]);
ensures a[j] == old(a[i]);
{
   int temp;
   temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
```

The Swap Contract (cont.)

```
static void Swap(int[]! a, int i, int j)
requires 0 <= i && i < a.Length;
requires 0 <= j && j < a.Length;
modifies a[i], a[j];
ensures a[i] == old(a[j]);
ensures a[j] == old(a[i]);
{
    int temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

requires annotations denote preconditions

Requires Clause

```
static void Swap(int[]! a, int i, int j)
requires 0 <= i && i < a.Length;
requires 0 <= j && j < a.Length;
modifies a[i], a[j];
ensures a[i] == old(a[j]);
ensures a[j] == old(a[i]);
{
   int temp;
   temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
```

frame conditions limit
the parts of the program state
that the method is allowed to modify.

Referring to Old Values

```
static void Swap(int[]! a, int i, int j)
requires 0 <= i && i < a.Length;
requires 0 <= j && j < a.Length;
modifies a[i], a[j];
ensures a[i] == old(a[j]);
ensures a[j] == old(a[i]);
{
    int temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

old(*a[j]*) denotes the value of *a[j]* on entry to the method

Referring to the Result

```
static int F( int p )
ensures 100 < p ==> result == p - 10;
ensures p <= 100 ==> result == 91;
{
    if ( 100 < p )
        return p - 10;
    else
        return F( F(p+11) );
}
```

`result` denotes the value returned by the method

Spec# Constructs so far

- ==> short-circuiting implication
- <==> if and only if
- **result** denotes method return value
- **old**(E) denotes E evaluated in method's pre-state
- **requires** E; declares precondition
- **ensures** E; declares postcondition
- **modifies** w; declares what a method is allowed to modify
- **assert** E; in-line assertion

Modifies Clauses

- **modifies** w   where w is a list of:
  - p.x   field x of p
  - p.*   all fields of p
  - p.**   all fields of all peers of p
  - **this**.*   default modifies clause, if **this**-dot-something is not mentioned in modifies clause
  - **this**.0   disables the “**this**.*” default
  - a[i]   element i of array a
  - a[*]   all elements of array a

Modifies Clauses (cont.)

- We can use a postcondition to exclude some modifications (from the default **this**.*)
- We can use a **modifies** clause to allow certain modifications
- x++, x-- in a method $\Rightarrow$ must have a **modifies** clause

Loop Invariants

- Statically verifying (calculating whether some condition is true at a certain point) is relatively easy for assignments, if statements, calls etc.
- The tough part – calculating what is true after a loop that may involve a significant number of iterating statements
- The problem can be often solved by submitting loop invariants – hints what is supposed to be true before and after each loop iteration
- Loop invariants also often formulate what intermediate results are achieved after each step

Loop Invariants: Computing Square by Addition

```
public int Square(int n)
  requires 0 <= n;
  ensures result == n*n;
 {
  int r = 0;
  int x = 1;
  for (int i = 0; i < n; i++)
   invariant i <= n;
   invariant r == i*i;
   invariant x == 2*i + 1;
  {
   r = r + x;
   x = x + 2;
  }
  return r;
 }
```

Square(3)

- r = 0 and x = 1 and i = 0
- r = 1 and x = 3 and i = 1
- r = 4 and x = 5 and i = 2
- r = 9 and x = 7 and i = 3

Loop Invariants (cont.)

- The pre-compiler makes the loop invariants into assertions to be checked
- Moreover, the verifier uses the invariant information to verify postconditions (**ensures** or **assert** statements occuring after the loop body)
- Formally:
  $loop\_invariants \wedge \neg loop\_condition \Rightarrow loop\_postcondition$

Loop Invariants: Integer Square Root

```
public static int ISqrt(int x)
requires 0 <= x;
ensures result*result <= x && x < (result+1)*(result+1);
{
        int r = 0;
        while ((r+1)*(r+1) <= x)
          invariant r*r <= x;
        {
                r++;
        }
         return r;
}
```

<Isqrt.ssc>

Loop Invariants: Integer Square Root (cont.)

```
public static int ISqrt1(int x)
requires 0 <= x;
ensures result*result <= x && x < (result+1)*(result+1);
{
        int r = 0; int s = 1;
        while (s<=x)
          invariant r*r <= x;
          invariant s == (r+1)*(r+1);
        {
                r++;
                s = (r+1)*(r+1);
        }
          return r;
}
```

Examples:

- **forall** {**int** k **in** (0: a.Length); a[k] > 0};
- **exists** {**int** k **in** (0: a.Length); a[k] > 0};
- **exists unique** {**int** k **in** (0: a.Length); a[k] > 0};

```
void Square(int[]! a)
  modifies a[*];
  ensures forall{int i in (0: a.Length); a[i] == i*i};
```

Loop Invariants (cont.)

```
void Square(int[]! a)
  modifies a[*];
  ensures forall{int i in (0: a.Length); a[i] == i*i};
 {
     int x = 0; int y = 1;
     for (int n = 0; n < a.Length; n++)
      invariant 0 <= n && n <= a.Length;
      invariant forall{int i in (0: n); a[i] == i*i};
     {      a[n] = x;
            x += y;
            y += 2;
     }
 }
```

<SqArray.ssc>

Error Message from Boogie

Spec# program verifier version 2, Copyright (c) 2003- 2010, Microsoft.

Error: After loop iteration: Loop invariant might not hold: forall{int i in (0: n); a[i] == i*i}

Spec# program verifier finished with 1 verified, 1 error

Loop Invariants (cont.)

```
void Square(int[]! a)
  modifies a[*];
  ensures forall{int i in (0: a.Length); a[i] == i*i};
 {
      int x = 0; int y = 1;
      for (int n = 0; n < a.Length; n++)
       invariant 0 <= n && n <= a.Length;
       invariant forall{int i in (0: n); a[i] == i*i};
       invariant x == n*n && y == 2*n + 1;
      {       a[n] = x;
              x += y;
              y += 2;
      }
  }
```

Inferred by /infer:p

Inferred by default