Spec#

CSE 6323

Aditya Mangipudi
Niharika Pamu
Srikanth Polisetty

The Verifying Compiler

"A verifying compiler uses
automated reasoning to check the correctness
of the program that it compiles.

Correctness is specified by
types, assertions, .. and other redundant annotations
that accompany the program."
[Hoare, 2004]

Program verifiers: A brief History

- Since the late 60's people are searching for solutions to prove correctness of software
  - *A Program Verifier, 1969*
  - In *Proceedings of the international conference on Reliable software, 1975*
  - GYPSY, 1977
  - Eiffel, 1986
- We find contract definitions in abstraction levels, like Object Constraint Language (OCL)

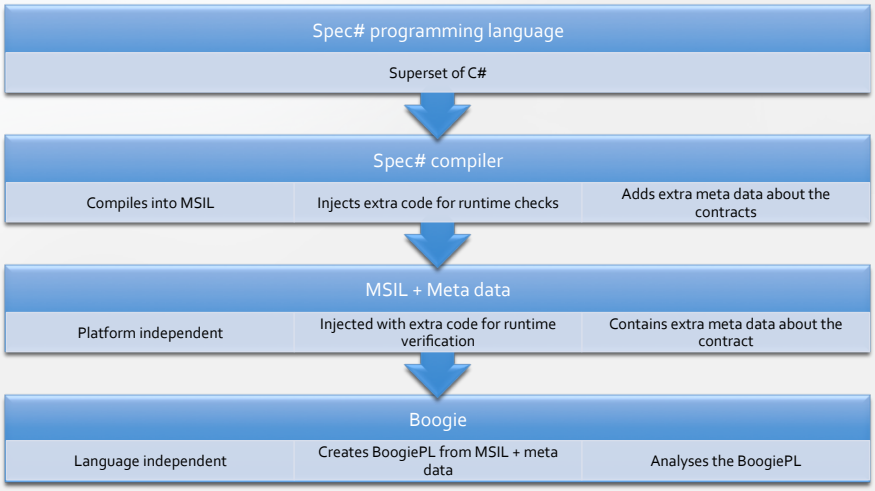
Outline

- Spec# Introduction
- Contracts
- Spec# Programming Language Specifications
- Spec# Tools
- Comparison with other verification languages
- Summary
- Observations

Spec#

- Pronounced as "Spec Sharp".
- Microsoft Research project.
- By Mike Barnett, K. Rustan, M. Leino, Wolfram Schulte.
- Superset of C#.
- Is available for development environment VS 2003,VS 2005 and VS 2008.
- **Its all about contracts.**

Simplified Spec# process



Contracts (I)

- A contract between callers and implementation
  - What the implementation can expect from the caller
  - What the caller can expect from the implementation
- You don't break contracts, that's illegal!

Contracts (II)

```
/// <summary>Divides <i>x</i> by <i>y</i>.</summary>
/// <param name="x">The value to divide.</param>
/// <param name="y">The value to divide by.</param>
/// <exception cref="ArgumentOutOfRangeException">
/// Occurs if <i>y</i> is zero. </exception>
/// <returns>The result of dividing
/// <i>x</i> by <i>y</i>.</returns>
public int Divide(int x, int y)
{
   if (y == 0)
      throw new ArgumentOutOfRangeException("y");

   return x / y;
}
```

- Method signature
- Method body
- Documentation
- What we expect
- What they can expect
- Check pre conditions
- Document pre conditions

Contracts (III)

```
/// <summary>Divides <i>x</i> by <i>y</i>.</summary>
/// <param name="x">The value to divide.</param>
/// <param name="y">The value to divide by.</param>
/// <exception cref="ArgumentOutOfRangeException">
/// Occurs if <i>y</i> is zero. </exception>
/// <returns>The result of dividing
/// <i>x</i> by <i>y</i>.</returns>
public int Divide(int x, int y)
{
   if (y == 0)
      throw new ArgumentOutOfRangeException("y");

   return x / y;
}
```

- What they can expect

Contracts (IV)

```
/// <summary>Divides <i>x</i> by <i>y</i>.</summary>
/// <param name="x">The value to divide.</param>
/// <param name="y">The value to divide by.</param>
/// <exception cref="ArgumentOutOfRangeException">
/// Occurs if <i>y</i> is zero. </exception>
/// <returns>The result of dividing
/// <i>x</i> by <i>y</i>.</returns>
public int Divide(int x, int y)
{
   if (y == 0)
      throw new ArgumentOutOfRangeException("y");

   return x / y;
}
```

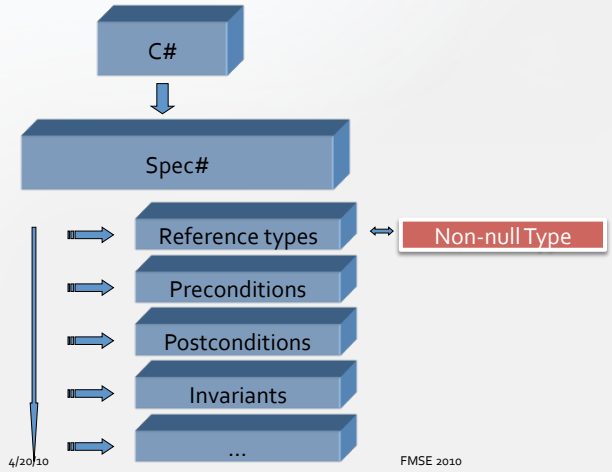
- What we expect
- Check pre conditions
- Document pre conditions

Result!!!

```
/// <summary>Divides <i>x</i> by <i>y</i>.</summary>
/// <param name="x">The value to divide.</param>
/// <param name="y">The value to divide by.</param>
/// <exception cref="ArgumentOutOfRangeException">
/// Occurs if <i>y</i> is zero. </exception>
/// <returns>The result of dividing
/// <i>x</i> by <i>y</i>.</returns>
public int Divide(int x, int y)
{
   if (y == 0)
      throw new ArgumentOutOfRangeException("y");

   return x / y;
}
```

- Method signature
- Method body
- Documentation
- What we expect
- What they can expect
- Check preconditions
- Document pre conditions

Spec# Specifications



Null references Errors

- Assumptions are a serious problem.
  - We trust argument inputs.
- Many errors in modern programs manifest themselves as null-reference errors.

Non-Null Types

- Spec# tries to eradicate all null reference 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;
```

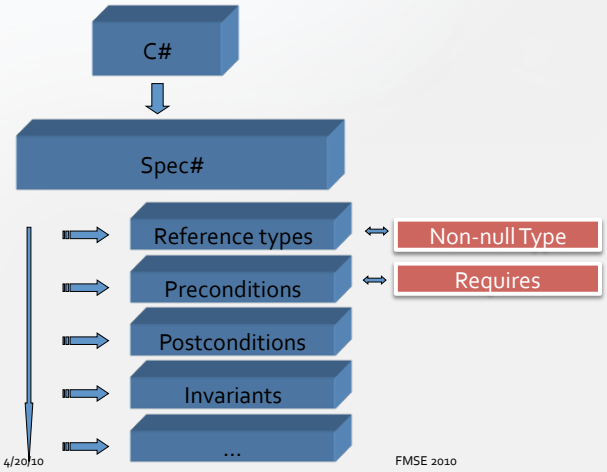
declares an array called xs which cannot be null

Non-Null by Default

	Without /nn	/nn
Possibly-null T	T	T?
Non-null T	T!	T

From Visual Studio, right-click Properties on the project, then Configuration Properties, and set ReferenceTypesAreNonNullByDefault to true

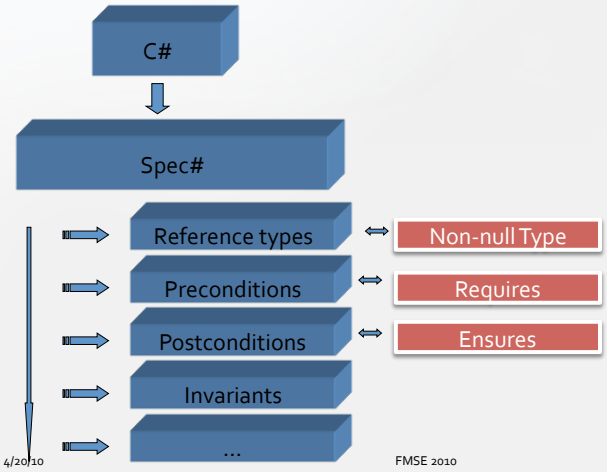
Spec# Specifications



Preconditions

- Describes the circumstances in which the method can be invoked.
- It is the callers responsibility to make sure that the preconditions are met.

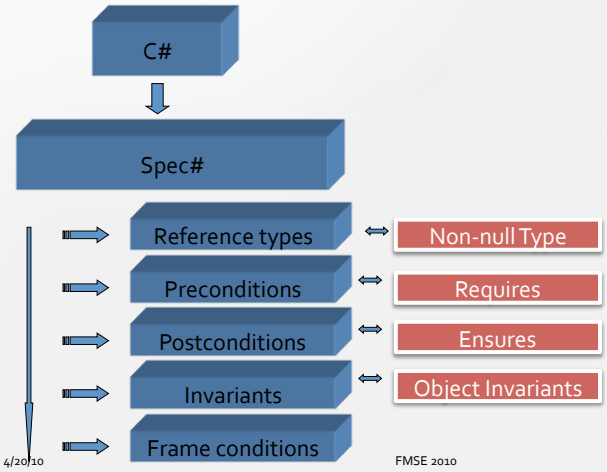
Spec# Specifications



Postconditions

- Describes the states in which the method is allowed to return.
- It is a two state predicate: It relates to the method's pre and post states.
- To refer to pre state: use **old(…)**

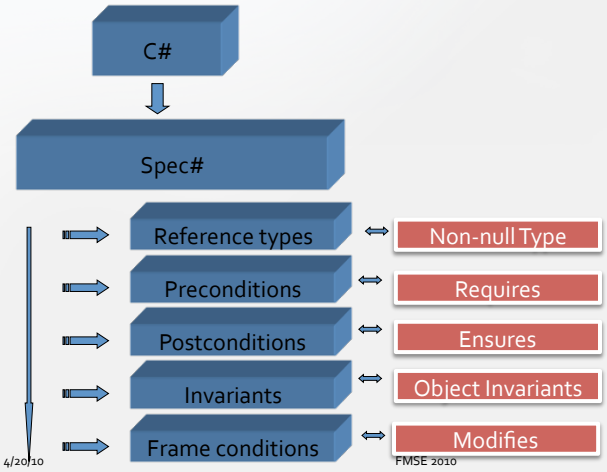
Spec# Specifications



Object Invariants

- Specifications for the steady state of an object
- Each object's data fields must satisfy the invariant at all stable times.

Spec# Specifications



Frame Conditions

- Used to restrict which pieces of the program state a method implementation is allowed to modify.
- Ex:

```
class C {
        int x, y;
        void M() modifies x; { . . . }
}
```

- Here modifies specifies that only x can be modified.

Swap Example:

```
static void Swap(int[]a, inti, intj)
    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;
}
```

Preconditions

Frame conditions

Post conditions

SPEC# QUICKIES

Immutable types

- Immutable types cannot be changed.
- Their member values are frozen.

```
[Immutable]
public class Foo
{
   // ...
}
```

Purity

- Methods are pure if they have no 'side-effect'
  - Great for optimization
  - Purity of the method is checked

```
Public class  XYZ
{
    int inc;
    int dec;
    [Pure]
    public int Getx()
    {
            return (inc – dec) ;
    }
}
```


Assume (Keyword)

- With the assume keyword you can say 'ignore' negative checks at compile time.
- Assume generates a run time check but is taken on faith by the program verifier.
- It trades static checking for dynamic checking.

Inline Assertions

- Assert statement can be used in code to indicate a condition that is expected to hold at that program point.
- It's a bit redundant.

```
assert x<0;
```


Exceptions

- **Checked Exceptions:**
  - Admissible Failures:
    when method is unable to complete for reasons outside o
    f its boundaries such as network
    errors or socket problems.
  - Signaled using an ICheckedException.
- **Unchecked Exceptions:**
  - Program Errors
  - Client Failures: conditions for getting to the method are not met

Exceptions Contd…

Failure	Reason
Client Failure	When is method is invoked under an illegal condition(Precondition is not satisfied)
Provider Failure	When procedure is unable to complete the task it is supposed to perform.

Provider Failure	Reason
Admissible Failure	Method unable to complete : a) Either at all b) After some effort
Observed Program Error	Either an intrinsic error in the program or a global failure (e.g. Out of memory error)

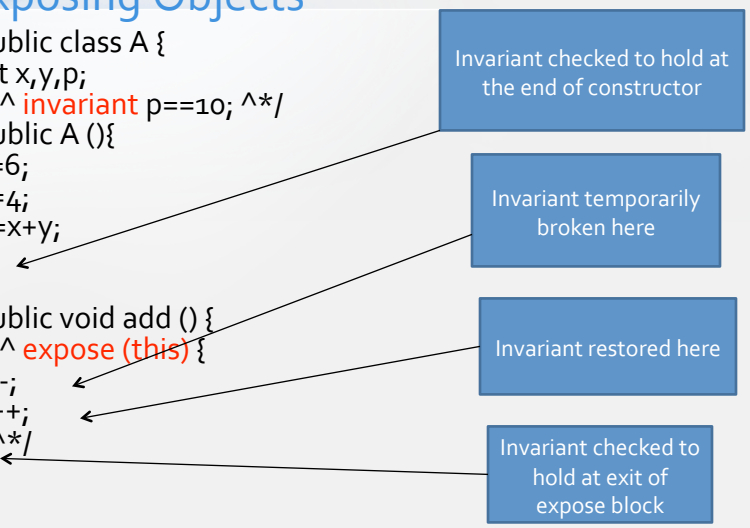
Exceptional Postconditions

- Limit which exceptions can be thrown by the method and for each such exception, describe the resulting state.

```
Void ReadToken(ArrayList a)
 throws EndOfFileException ensures a.Count == old(a.Count);
 { .....}
```

Exposing Objects

```
public class A {
int x,y,p;
/*^ invariant p==10; ^*/
public A (){
x=6;
y=4;
p=x+y;
}

public void add () {
/*^ expose (this) {
x--;
y++;
} ^*/
}
```



Inheritance of Specifications (I)

- Method's contracts of the Super Class are inherited by the Sub Classes
  - Thus Preconditions, Post Conditions and Invariants are inherited as well.
- A Method Contract of the Sub Class can add more post conditions by declaring additional **ensures** clauses.
- **Preconditions** and the **modifies** clauses are not allowed to be added in the **method's** overrides and preconditions on the overridden method cannot be even weakened in the subclass.

Inheritance of Specifications (II)

- Multiple inheritance is supported in Spec# through Interfaces,
  interface I { **void** M(**int** x) **requires** *x* <= 10; }
  interface J { **void** M(**int** x) **requires** *x* >= 10; }
- Spec# does not allow *C* to provide one shared implementation for *I .M* and *J .M* .
- class *C* needs to give explicit interface method implementations for *M* :

```
class C : I , J
{
      void I .M(int x ) { . . . }
      void J .M(int x ) { . . . }
}
```


Spec# Tools

- Compiler
- Runtime Library
- Static Verifier

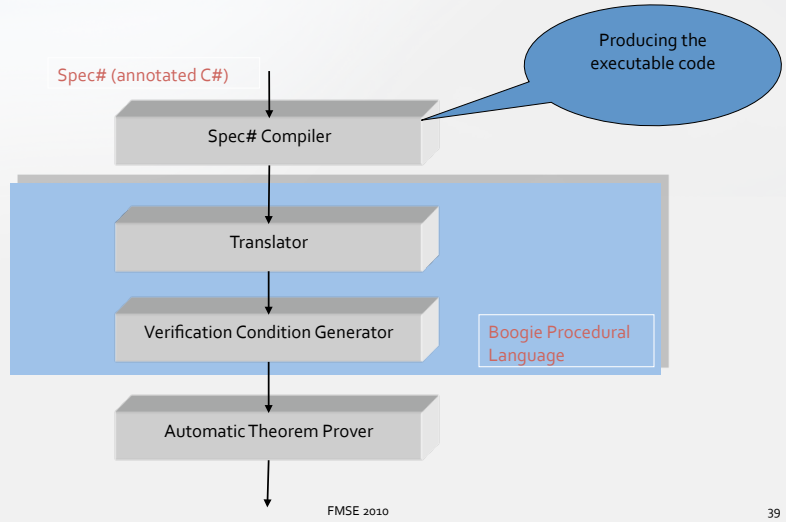
Spec# compiler

- Emits run-time checks for:
  - Contracts
  - Invariants
- Records the contracts as metadata for consumption by downstream tools.

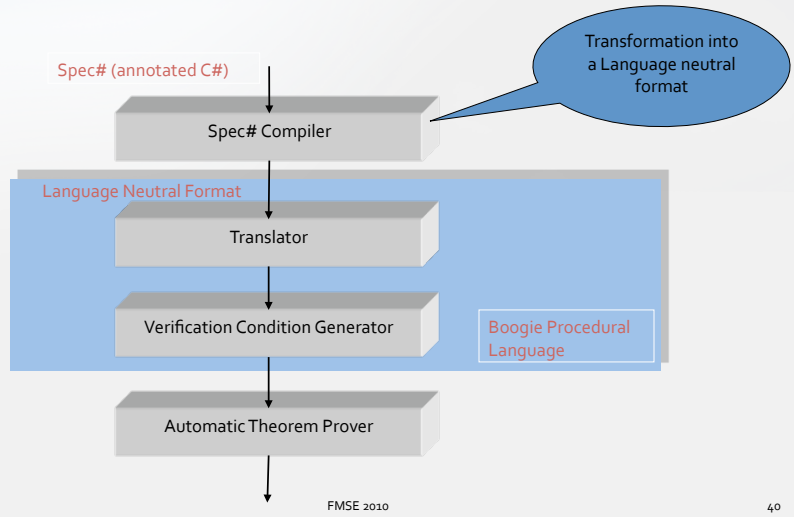
Spec# static program verifier

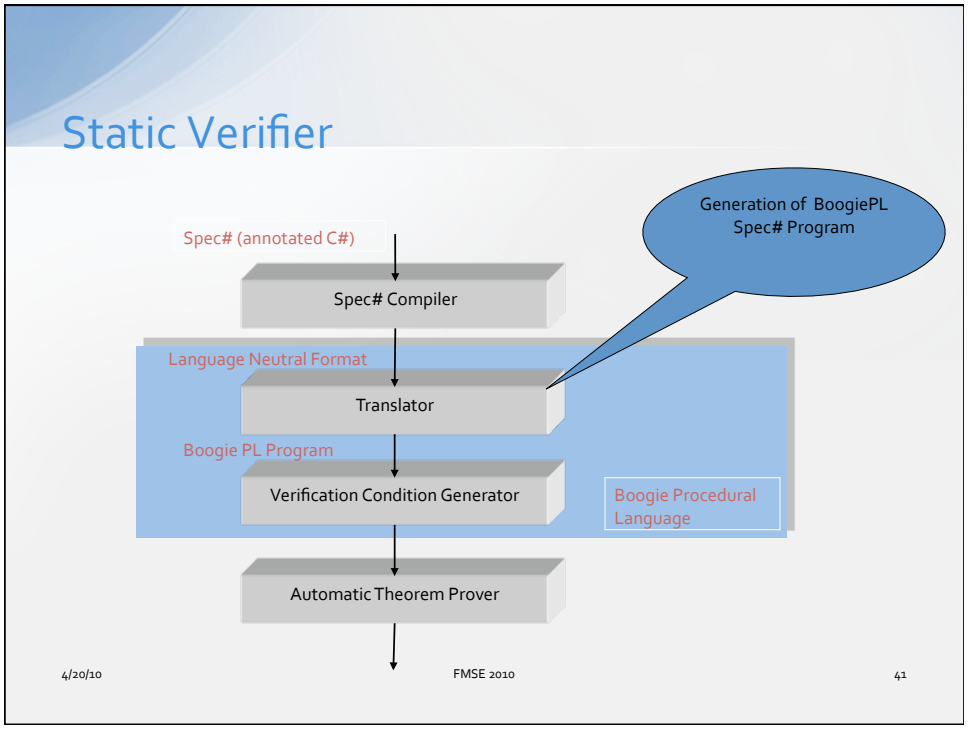
- It's called Boogie.
- A component that generates logical verification conditions from a Spec# program.
- Read MSIL + Meta Data to create BoogiePL
- Language independent.

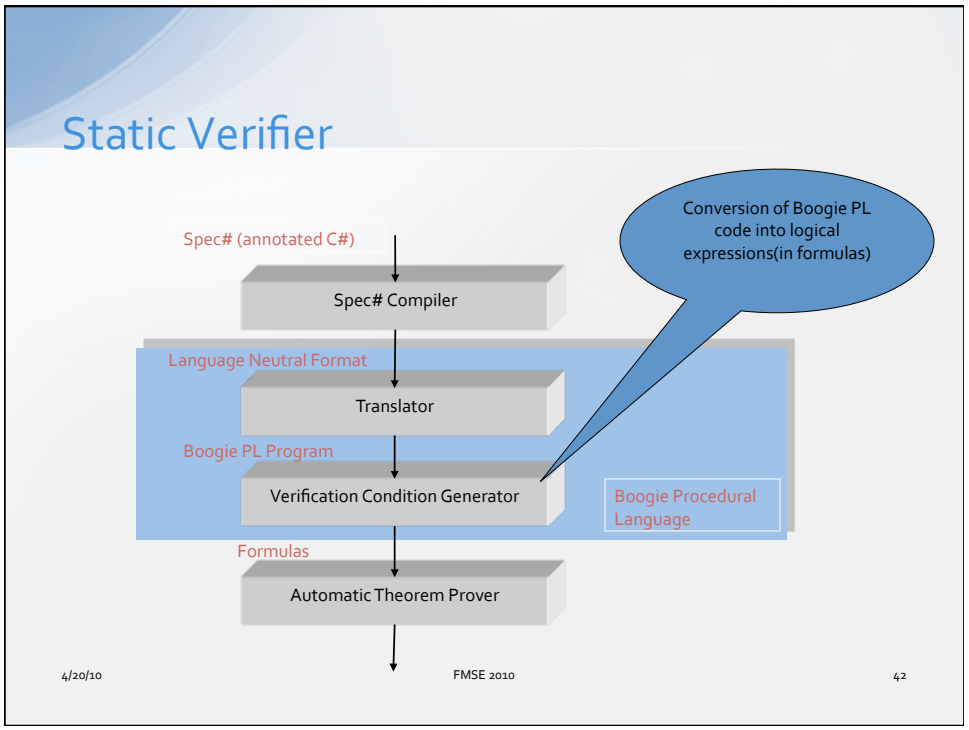
Static Verifier



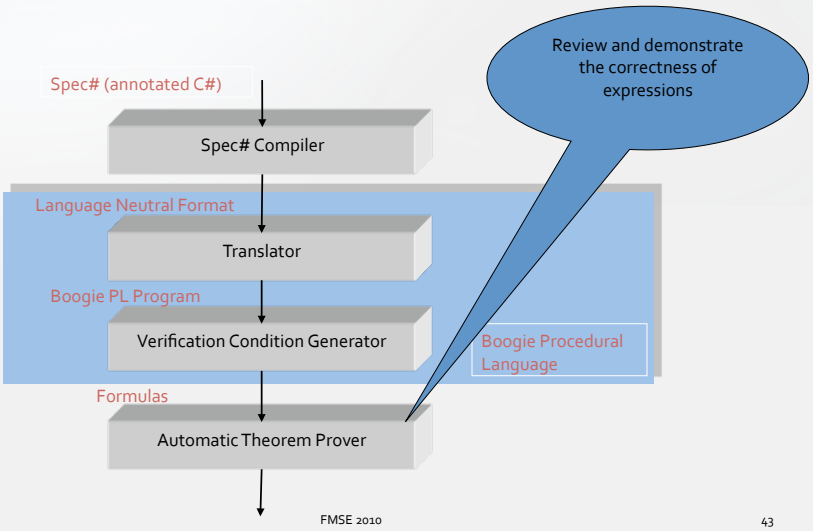
Static Verifier




Static Verifier
Generation of BoogiePL Spec# Program
Spec# (annotated C#)
Spec# Compiler
Language Neutral Format
Translator
Boogie PL Program
Verification Condition Generator
Boogie Procedural Language
Automatic Theorem Prover
4/20/10
FMSE 2010
41


Static Verifier
Conversion of Boogie PL code into logical expressions(in formulas)
Spec# (annotated C#)
Spec# Compiler
Language Neutral Format
Translator
Boogie PL Program
Verification Condition Generator
Boogie Procedural Language
Formulas
Automatic Theorem Prover
4/20/10
FMSE 2010
42

Static Verifier



From Spec#…

```
static int Abs(int x)
   ensures 0 <= x ==> result == x;
   ensures x < 0 ==> result == -x;
{ if (x < 0) x = -x; return x; }
```

...to BoogiePL

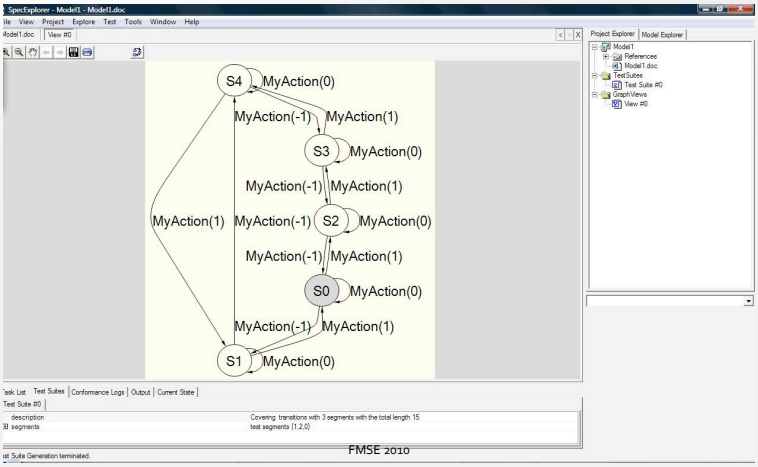
```
procedure Abs(x$in: int) returns ($result: int);
   ensures 0 <= x$in ==> $result == x$in;
   ensures x$in < 0 ==> $result == -x$in;
{ var x1, x2: int, b: bool;

 entry:  x1 := x$in; b := x < 0;   goto t, f;
 t:          assume b; x := -x;      goto end;
 f:          assume !b;              goto end;
 end:      $result := x;             return; }
```


Theorem Provers

- HP Simplify
  - Developed as part of the Extended Static Checking project to check Java and Modula-3 programs
- Microsoft Z3
  - Utilizes the Satisfiability Modulo Theories (SMT)

Tool support: Spec Explorer (I)

- Spec Explorer Model-Based Test Development



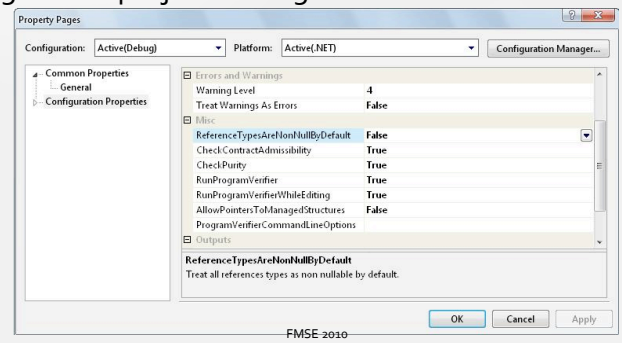
Spec Explorer (II)

•**Spec Explorer** is a software development tool for model-based specification and testing.

•Discrepancies between actual and expected results are called conformance failures and may indicate : Implementation Bugs, Modeling errors, Specification errors, Design errors.

Configuring Spec# in VS 2008

- Download & Install Spec# for VS 2008.
  http://research.microsoft.com/en-us/downloads/8826adb9-8398-40d6-a22d-951923fe2647/default.aspx
- Create a new Project user Spec#.
- Configure the project settings as shown.



Comparison: JML & Spec#

JML	Spec#
requires	requires (precondition)
assignable	modifies (frame condition)
ensures	ensures (normal postcondition)
signals, exceptional_behavior	otherwise, throw
diverges, accessible, callable	doesn't support so far

Spec# & other verification languages

- The design space of Spec# is somewhat less constrained than JML, since JML does not seek to alter the underlying programming language (which, for example, has let Spec# introduce field initializers and expose blocks).
- AsmL has many of the same aspirations as Spec#: to be an accessible, widely-used specification language tailored for object-oriented .NET systems.
  However, AsmL is oriented toward supporting model-based development with its facilities for model programs, test-case generation, and meta-level state exploration .
- Compared to ESC/Java2, Spec# saves good amount of annotation overhead.

Summary (I)

- Spec# extends c#
  - Integrates well with new keywords & syntax.
  - The learning curve for c# programmers is very low.
- Make it easier to record detailed design decisions
  - Help prevent and detect bugs
  - Reduce cost of software lifecycle
- Reduces defensive checking with help from Non Null types. So removal of safety checking code.
- Code with specification can be optimized!
  - (e.g. int with invariant >= 0 can become a uint)

Summary (II)

- Verifier can be customized to various levels of verification ranging from warnings to errors.
- The Static Verifier helps in detecting errors while coding.
- The compile, Boogie are integrated into Visual Studio providing a comprehensive set of tools.

Observations

- Spec# works better if you have small concrete methods.
- Preconditions and Postconditions can contain errors.
- Subclasses cannot weaken the preconditions.
- Can verify all linear assertions but non-linear expressions are not completely handled so some explicit invariants need to be declared. Gives timeouts in such undecidable cases.

Vision of the Spec# project team

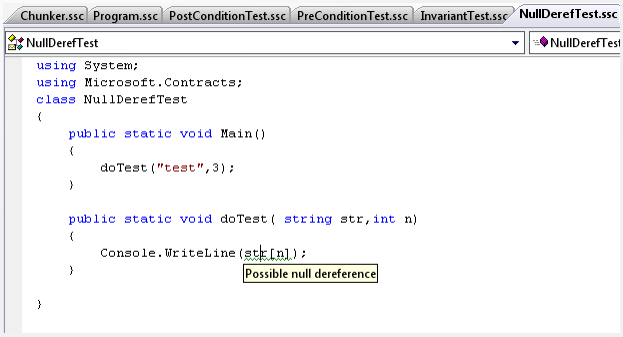
"Program development would be improved if more assumptions were recorded and enforced."

- Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte in Manuscript KRML 136, 12 October 2004

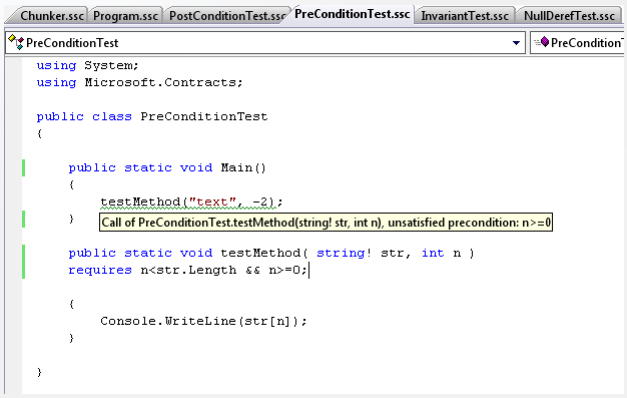
Questions

DEMO

Null Dereference Error



Preconditions Test



Postconditions Test

Chunker.ssc | Program.ssc | PostConditionTest.ssc | PreConditionTest.ssc | InvariantTest.ssc | NullDerefTest.ssc

PostConditionTest | Main()

```
using System;
using Microsoft.Contracts;


//Post Condition with assume keyword

class PostConditionTest
{
    private int x = 10;

    public static void Main(){

        new PostConditionTest().testMethod();

    }
    public void testMethod()


    ensures x > 0;
    {
        x = -1;
    }
}
```


Object Invariants Test

Chunker.ssc | Program.ssc | PostConditionTest.ssc | PreConditionTest.ssc | InvariantTest.ssc | NullDerefTest.ssc

InvariantTest | InvariantTest()

```
using System;
using Microsoft.Contracts;
class InvariantTest
{

    int c;
    invariant 0<= c;

     static void Main()
    {
        InvariantTest p = new InvariantTest();
        p.Inc();
    }

    public InvariantTest()
    {
        c = 0;
    }
    public void Inc()
    modifies c;
    ensures c == old(c) + 1;
    {
        c++;
    }


}
```

References

- Using Spec Explorer for Model-Based Test Development - http://dotnet.sys-con.com/node/163765
- http://research.microsoft.com/en-us/projects/specsharp/
- http://research.microsoft.com/en-us/projects/specsharp/krml135.pdf
- http://tmd.havit.cz/Papers/SpecSharp.pdf
- http://blogs.msdn.com/sriram/archive/2005/06/18/Digging-Into-SpecSharp.aspx
- http://channel9.msdn.com/Wiki/SpecSharp/DelayedTypesIntroduction/
- http://en.wikipedia.org/wiki/C_Sharp_
- http://www.slideshare.net/pjvdsande/specsharp-presentation
- http://blogs.msdn.com/sriram/archive/2005/06/18/Digging-Into-SpecSharp.aspx