Microsoft Research
FacultySummit

# *Teaching Using Spec# in Europe*

## *An Experience Report from University Teaching and Various Verification Tutorials*

**Dr Rosemary Monahan**

**Lecturer**
**National University of Ireland, Maynooth**

FUTURE WORLD 2011 2031

# Structure of Presentation

- Third Level Education in Europe
- Teaching Software Verification at NUIM/in Europe
- The Spec# Programming System
- Teaching Software Verification using Spec#
- Conference Tutorials
- Software Verification Developments at NUIM

# Third Level Education in Europe

- **Bologna Declaration 1999**: a pledge by 29 European countries to reform the structures of their higher education systems in a convergent way. Objectives include:
  - **the introduction of undergraduate and postgraduate levels** in all countries, with first degrees no shorter than 3 years and relevant to the labour market;
  - **ECTS-compatible credit systems** also covering lifelong learning activities;
- **Bachelors and Masters Degree**
  - Typically achieved in **5 years** (3+2, 4+1, 60 ECTS per year)
- **Doctoral Degree**
  - Typically achieved in **4 years** full time research (may include some course work)

# Teaching Software Verification at NUIM

## BSc Computer Science Degree Students: 5 ECTS in Year 3

**Learning Outcomes:**

- Explain the limitations of testing as a means to ensure correctness.
- Evaluate the role of verification in software engineering.
- Create mathematically precise specifications and designs using logic-based specification languages.
- Prove the correctness of programs with respect to a specification using Hoare logic.
- Analyse the properties of formal specifications and designs.
- Use tools to verify properties of specifications and designs.

**Teaching & Learning Methods:**

24 Lecture hours, 24 Laboratory hours, 32 Tutorial and Independent Study hours.

**Timetable:** Over 12 weeks from September – December each year

# Teaching Software Verification at NUIM

## MSc in Computer Science Degree Students: 7.5 ECTS in Year 1

**Learning Outcomes:**

- Reason about why software projects fail.
- Identify the role of rigor in the software process in improving the chances of success.
- Create mathematically precise specifications and designs.
- Analyse the properties of formal specifications and designs, performing proofs of correctness.
- Analyse the correctness of object-oriented programs (e.g. static analysis, simulation, model checking).
- Describe how tools that assist in this analysis are designed and implemented.

**Teaching & Learning Methods:**
One week of full-time lectures followed by one week for the completion of marked assignments

**Timetable:** During 2 weeks each November

# Typical Student Background

- Object Oriented Programming
- First Order Logic
- Compilers
- Algorithms and Data Structures
- Testing
- Software Metrics
- Hoare Logic

# Problems Encountered

- Limited tool support
- Limited automation of tool support
- Tools change rapidly and don't build on previous versions
- Consistency of tool support
- Mathematics upfront intimidates (some) students
- Feedback to students is limited
- Correcting homework is time consuming and labour intensive
- Few motivated tutors
- Student effort drained on the wrong focus points

# Teaching Software Verification in Europe

- http://resources.cost-ic0701.org/teaching-materials

# The Spec# Programming System

**The Spec# programming language** is an extension of C# 2.0 with non-null types, checked exceptions and throws clauses, method contracts and object invariants.

**The Spec# 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

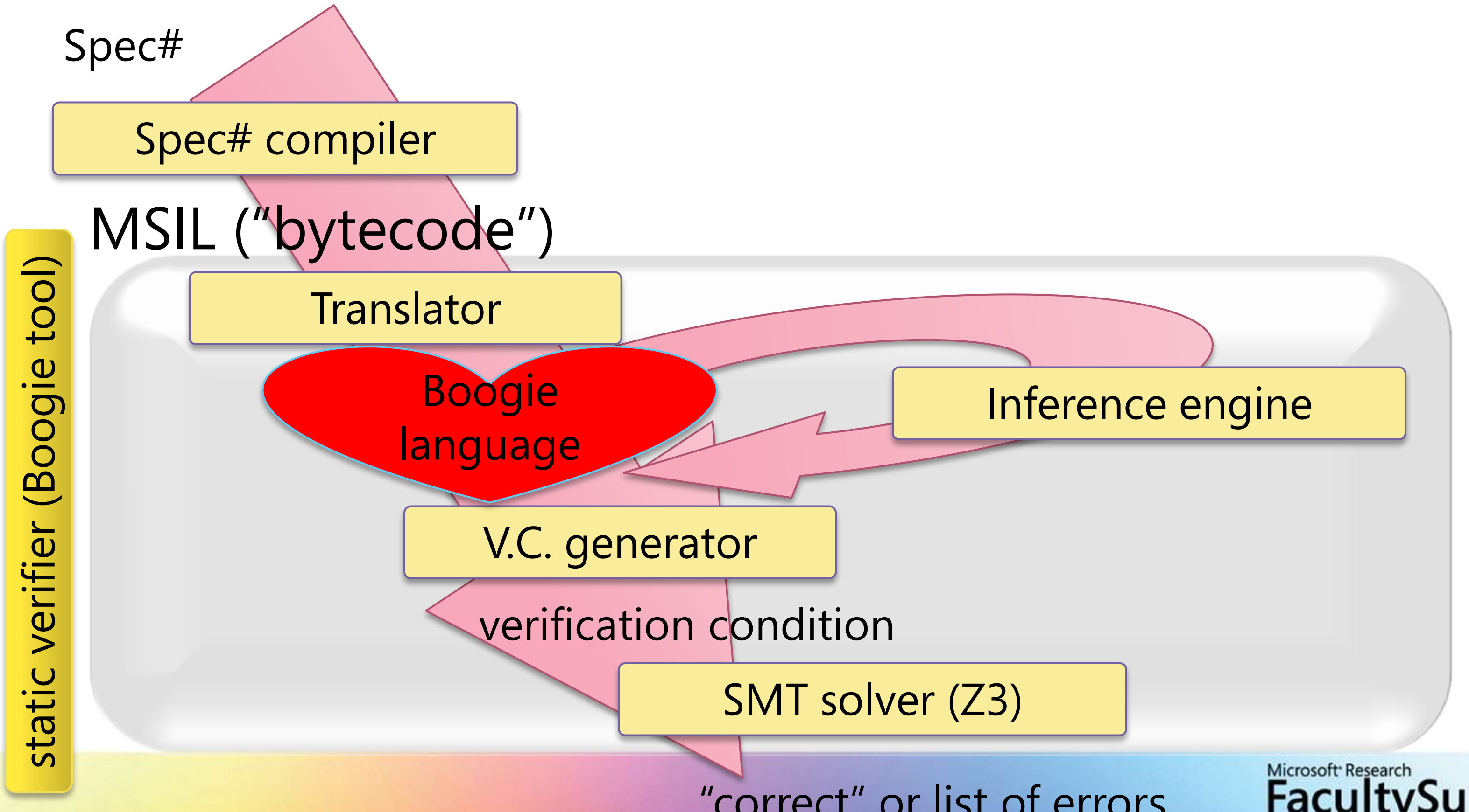
**The Spec# static program verifier (SscBoogie) :**

- generates logical verification conditions from a Spec# program
- uses an automatic reasoning engine (Z3) to analyse the verification conditions proving the correctness of the program or finding 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



# 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#** codeplex.specsharp.com



# Spec#

Spec# ("speck-sharp") is an object-oriented .NET programming language with design-by-contract features for method pre- and postconditions and object invar

- Frequently asked questions
- External Dependencies
- How to install the binaries
- How to install and build the sources
- How to contribute
- Spec# @ MSR
- Spec# tutorial

This project is sponsored by the Research in Software Engineering Group (RiSE) based in the Microsoft Research Redmond Laboratory.

Last edited Aug 14 2009 at 2:28 AM by rustanleino, version 13

---

Want to leave feedback?
Please use Discussions or Reviews instead.

# Using Spec#

## Option 1: Visual Studio (VS)

- Using Visual Studio in interactive mode means that we can get immediate feedback from the verifier
- .ssc file must be part of a VS project

## Option 2: Spec# at the command line

- Use your favorite editor
- .ssc file need not be part of a VS project

## Option 3: RISE4fun.com

- Interact online - No tool installation necessary


Microsoft Research

> **114,692** answers!

No installation needed:
Run it from an Internet café or from your phone

# 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;
}
```

http://rise4fun.com/Sp0

# The Swap Contract

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

**Requires clauses denote preconditions**

http://rise4fun.com/Sp0

# The Swap Contract

```
static void Swap(int[]! a, int i, int j)
  modifies a[i], a[j];
{
        int temp;
        temp = a[i];
        a[i] = a[j];
        a[j] = temp;
}
```

Modifies clauses limit the part of the program state that a method is allowed to modify

http://rise4fun.com/SpecSharp/0

# The Swap Contract

```
static void Swap(int[]! a, int i, int 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;
}
```

**Ensures clauses denote postconditions**

http://rise4fun.com/SpecSharp/0

# The Swap Contract

```
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;
}
```

http://rise4fun.com/SpecSharp/s16cX

# Mc Carthys Contract

```
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) );
      }
```

http://rise4fun.com/SpecSharp/o

# Collaboration with Microsoft Research

- K. R. M. Leino and R. Monahan, Automatic verification of textbook programs that use comprehensions. In Formal Techniques for Java-like Programs, ECOOP Workshop (FTfJP'07: July 2007, Germany)

- K. R. M. Leino and R. Monahan, Reasoning about Comprehensions with First-Order SMT Solvers, (SAC'09: March 2009, Hawaii, U.S.A.)

# Loops in Spec#

```
public static int SegSum(int[]! a, i int i, int j)
requires 0 <= i  &&  i <= j  &&  j <= a.Length;
ensures result == sum{int k in (i: j); a[k]};
{
      int s = 0;
      for (int n = i; n < j; n++)
       {
             s += a[n];
       }
       return s;
 }
```

# Loops in Spec#

```
public static int SegSum(int[]! a, int i, int j)
requires 0 <= i  &&  i <= j  &&  j <= a.Length;
ensures result == sum{int k in (i: j); a[k]};
{
    int s = 0;
    for (int n = i; n < j; n++)
    {
        s += a[n];
    }
     return s;
 }
```

When we try to verify this program using Spec# we get an Error:
*Array index possibly below lower bound as the verifier needs more information*

# Adding Loop Invariants

**Postcondition:**
ensures result == sum{int k in (i: j); a[k]};

**Loop Initialisation**: n == i

**Loop Guard:** n < j

**Loop invariant:**
invariant **s** == sum{int k in (i: **n**); a[k]};
invariant **i <= n && n <= j;**

**Introduce the loop variable & provide its range.**

# Adding Loop Invariants

```
public static int SegSum(int[]! a, int i, int j)
requires 0 <=i  &&  i <= j  &&  j <= a.Length;
ensures result == sum{int k in (i:j); a[k]};
{
          int s = 0;
          for (int n = i; n < j; n++)
           invariant i<= n && n <= j;
           invariant s == sum{int k in (i:n); a[k]}
          {
               s += a[n];
          }
          return s;
}
```

**Verifier Output:**

*Spec# Program Verifier finished with 3 verified, 0 errors*

# Variant Functions

```
public static int SegSum(int[]! a, int i, int j)
requires 0 <= i  &&  i <= j  &&  j <= a.Length;
ensures result == sum{int k in (i: j); a[k]};
            {
                        int s = 0; int n=i;
                        while (n < j)
                          invariant i < = n && n < =j;
                          invariant s == sum{int k in (i:n); a[k]};
                          invariant 0 < = j - n;
                        {
                                    int vf = j - n;  //variant function
                                    s + = a[n];
                                    n++;
                                    assert j - n < vf;
                        }
                        return s;
            }
```

http://rise4fun.com/SpecSharp/q

*We can use assert statements to determine information about the variant functions.*

# Object Invariants

```
class Counter{
        int c;
        bool even;
        invariant 0 <= c;
        invariant even <==> c % 2 == 0;

        public Counter()
        {       c= 0;
                even = true;
        }

        public void Inc ()
          modifies c;
          ensures c == old(c)+1;
        {       expose (this) {
                        c++;
                        even = !even ;
                }
        }
}
```

The invariant may be broken in the constructor

The invariant must be established & checked after construction

The object invariant may be broken within an expose block

# Using Spec# in Teaching since 2007

- **CS357 Software Verification  (5 ECTS, BSc Level)**
  - Focus on programming in the small (writing contracts for methods and classes)
  - Using the Spec# Programming System to verify code
- **CS603 Rigorous Software Process (7.5 ECTS, MSc Level)**
  - Focus on programming in the large (Inheritance, Ownership, Aggregation)
  - Details of how the implementation is statically checked against the specification

**Interfaces Used:**
Command Line
Visual Studio
www.rise4fun.com

# Other Courses using Spec#

## NUIM International Summer School

**Practical Program Analysis (4 course for undergraduate students)**

This course covers the practical elements of analysing and verifying object-oriented programs, which forms the basis of the research being carried out by the Principles of Programming research group at NUIM.

## Universite Henri Poincare 1, Nancy, France

**Erasmus Teaching Exchange ( 4 hour introduction to Spec#)**

# Sample Project Work

- University Marks and Standards
- Poker hand comparisons
- Verification Benchmarks (VSTTE 2008)
- Verification Competition Problems (VSTTE 2010)
- Verify-this Benchmark Collection
  - http://verifythis.cost-ic0701.org
- VACID-0: Verification of Ample Correctness of Invariants of Data-structures (VSTTE 2010)

# Student Feedback

- Tool usage motivates students to verify their software
- Students are motivated to learn how the tools work
- Students are using Spec# outside of their course work
- Main difficulty is in understanding error messages
- Interest in internships and projects concerning program verification has increased
- Interest in PhD work concerning verification related topics

# Conference Tutorials

- ETAPS 2008
- ECOOP 2009
- iFM 2010
- SBMF 2010
- Cost Action IC0701 PhD Training School 2011

# Participant Feedback

Researchers and Lecturers

- Interested in using Spec# in teaching
- Visual Studio environment
- www.Rise4fun.com
- Supports typical classroom examples
- “makes it look so easy”

# Software Verification Developments at NUIM

- PhD Scholarships funded involving Software Verification
- Student Internships
- A renewed interest from students in Software Verification
- Erasmus Mundus MSc in Dependable Software Systems

# Spec# Teaching Resources

**Spec# Tutorial Paper:**

Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs. K. Rustan M. Leino and Peter Müller at http://specsharp.codeplex.com/

**Spec# Tutorial and Exercises:**

Spec# examples and course notes available by emailing rosemary (rosemary@cs.nuim.ie) or at http://limerick.cost-ic0701.org/

# Questions?

Microsoft® Research
FacultySummit



2011 FUTURE/WORLD 2031

Microsoft® Research
FacultySummit

2031
FUTURE WORLD
2011