ifm 2010 integrated Formal Methods

NUI MAYNOOTH
Ollscoil na hÉireann Má Nuad

Verification of C# programs using Spec# and Boogie 2

IFM Tutorial 2010

Rosemary Monahan
National University of Ireland, Maynooth

With thanks to K. Rustan M. Leino, Microsoft Research and Peter Müller, ETH Zurich for work on previous tutorials

Tutorial Overview

Goal:

Learn to use Spec# to verify programs

Aim:

Specifications should be detailed enough that programs can be verified statically

Ideal Background:

Basic familiarity with object oriented programming concepts and the syntax of Java-like languages

Tutorial Overview

Structure:

- Spec# overview and installation
- Programming in the small:
  - Preconditions, Postconditions, Loop invariants
- Programming in the large:
  - Object invariants, Ownership,
- A look behind the scenes:
  - Boogie 2: An intermediate verification language
- Overview of other verification language research

Introducing Spec#

Spec#: An Overview

Installing Spec#

Using Spec#

The Spec# Programming System

The Spec# Programming System provides language and tool support for static verification of object oriented programs.

The Spec# programming language:

- an extension of C# 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
- records the contracts as metadata for consumption by downstream tools

The Spec# static program verifier (Boogie):

- generates logical verification conditions from a Spec# program
- uses an automatic theorem prover (Z3) to analyse the verification conditions proving the correctness of the program or finding errors in it

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.

Installing Spec#

Download the latest version of Spec# from
http://specsharp.codeplex.com/

- The Spec# installation requires Visual Studio.
- Installation includes the compiler, VS plug-in, Boogie 2, Z3
- Optional: Simplify
- Programs may also be written in any editor and saved as Spec# files (i.e. with a .ssc extension).
- Visual Studio projects provide immediate feedback when an error is detected

Spec#

Spec#

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

- 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.

Structure of .NET programs

- Programs are split into source files (.ssc).
- Source files are collected into projects (.sscproj).
- Each project is compiled into one assembly (.dll .exe) and each project can use its own language and compiler.
- Projects are collected into solutions (.sln).
- Typical situation: 1 solution with 1 project and many source files.
- Note that the compiler does not compile individual source files, but compiles projects. This means that there need not be a 1:1 correspondence between classes and files.

Using the Visual Studio IDE

- Open Visual Studio
- Set up a new Project (File -> new -> project)
- Open a Spec# project console application.(Spec# projects -> Console application)

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

- Build the solution (Build -> Build Solution) F6
- Execute the program (Debug -> Start) F5
- Tip: adding Console.Read(); to the end of your program requires that the user presses a key before the screen disappears.

Interactive mode (in VS 2008)

- To run the program verifier as you program, set the RunProgramVerifierWhileEditing to True
  - Find this in the project properties option of the project menu. Click on Configuration Properties, then Build and under Misc.
  - This means that you get verification errors underlined in green as you type. Anything underlined in red is a compilation error.
- To run the verifier when debugging (F6), set RunProgramVerifier to True
  - Under the Misc heading as above.

<Counter.ssc> <MinFct.ssc>

Using your favourite Editor

- Type up your Spec# program e.g.

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

- Save it with a .ssc extension e.g. Program.ssc

Using Boogie at the Command line

- Open a command prompt
  - Go to the directory where you have specsharp installed or add it to your path (e.g. C:\Program Files\specsharp\)
- Compile Program.ssc stored in on C:\temp using

  C:\temp> ssc /t:library /debug Program.ssc

  This generates files called Program.dll and program.pdb which are stored in C:\temp.
- C:\temp> ssc Program.ssc compiles Program.ssc into a .exe executable.
- C:\temp> sscboogie Program.dll (or Program.exe) verifies the compiled file using the SMT solver Z3.

Using Boogie at the Command line

- To create Boogie PL programs use
  sscboogie Program.dll /print:Program.bpl
- To get more feedback on the verification process use
  sscboogie Program.dll /trace

- Further switches can be seen by typing sscboogie /help
  or ssc /help
- To execute the program type Program.exe

Rosemary.Monahan@NUIM.ie

The Language

- The Spec# language is a superset of C#, an object-oriented language targeted for the .NET platform
  - C# features include single inheritance whose classes can implement multiple interfaces, object references, dynamically dispatched methods, and exceptions
  - Spec# extends C# with contracts allowing programmers to document their design decisions in their code (with support for non-null types, checked exceptions and throws clauses, method contracts and object invariants).

Non-Null Types

!

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

- 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

```
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

```
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

```
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);
    }
}
```


Rosemary.Monahan@NUIM.ie

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;
    }

 }

}
```

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

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


Rosemary.Monahan@NUIM.ie

Non-Null by Default

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

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

When we compile a Spec# program at the command line we can use the switch /nn to make non-null types the default:

ssc /t:library /debug /nn Program.ssc

Initializing Non-Null Fields

```
class C {
    T! x;
    public C(T! y) {
        x = y;
    }
    public C(int k) {
        x = new T(k);
    }
    ...
```

Delayed Constructors

- In C#, if the constructor body does not explicitly call a constructor, a call **base()** is inserted by the compiler at the beginning of the body (immediately following the field initialisers).

- Before the **base()** constructor has been called, we say that the object ***this*** is delayed.

Initializing Non-Null Fields

```
class C {
    T! x;
    public C(int k) {
        x = new T(k);
        x.M();
    }
```

Delayed receiver is not compatible with non-delayed method

Delayed Constructors

- The default in Spec# is that ***this*** is delayed throughout the constructor body
- This means that we cannot assume non-null fields to be non-null (and we cannot assume that object invariants hold) until the constructor call terminates.
- Hence, the object under construction can only be used as the target object in field assignments.
  - This is why we get an error when we call **x.M()** in our previous example.

Initializing Non-Null Fields

```
using Microsoft.Contracts;
class C {
    T! x;
    [NotDelayed]
    public C(int k) {
        x = new T(k);
        base();
        x.M();
    }
```

Allows fields of the receiver to be read

Spec# allows base calls anywhere in a constructor.

In non-delayed constructors, all non-null fields (e.g. x) must be initialized before calling base

Non-Null and Delayed References

- Declaring and checking non-null types in an object-oriented language. Manuel Fähndrich and K. Rustan M. Leino. In *OOPSLA 2003*, ACM.
- Establishing object invariants with delayed types. Manuel Fähndrich and Songtao Xia. In *OOPSLA 2007*, ACM.
- Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs. K. Rustan M. Leino and Peter Müller. On specsharp.codeplex.com

Assert

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

```
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);
          }
    }
  }
}
```

Assume Statements

- The statement **assume** E; is like **assert** E; at run-time, but the static program verifier checks the assert whereas it blindly assumes the assume.

Design by Contract

Code Examples on

http://ifm2010.loria.fr/satellite.html

See subfolder Part1

Design by Contract

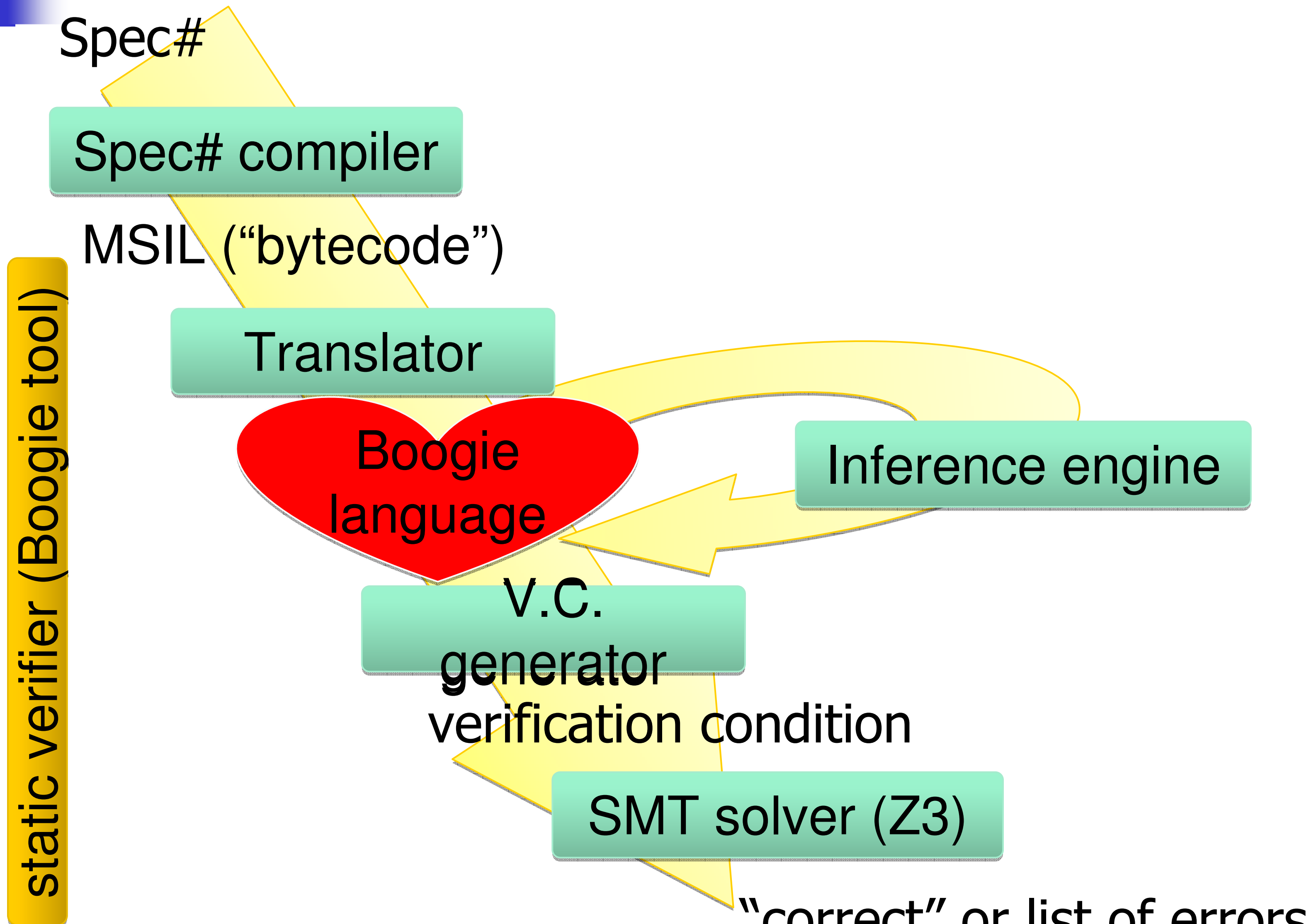
- Every public method has a precondition and a postcondition
- The precondition expresses the constraints under which the method will function properly
- The postcondition expresses what will happen when a method executes properly
- Pre and postconditions are checked
- Preconditions and postconditions are side effect free boolean-valued expressions - i.e. they evaluate to true/false and can't use ++

Static Verification

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



Rosemary.Monahan@NUIM.ie

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

```
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

```
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

Modifies Clauses

```
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.

Swap Example:

```
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

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

<Rectangle.ssc>

- We can use a postcondition to exclude some modifications (from the default **this**.*)
  - Like MoveToOrigin in Rectangle.ssc

- We can use a **modifies** clause to allow certain modifications
  - like Transpose in Rectangle.ssc
  - x++; x--; in a method => must have a modifies clause

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

```
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

```
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;
}
```

Quantifiers in Spec#

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};

Quantifiers in Spec#

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};
```

<Search.ssc>

Loop Invariants

```
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

<SqArray1.ssc>

Rosemary.Monahan@NUIM.ie

Inferring Loop Invariants

```
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

Comprehensions in Spec#

Examples:

- **sum** {**int** k **in** (0: a.Length); a[k]};
- **product** {**int** k **in** (1..n); k};
- **min** {**int** k **in** (0: a.Length); a[k]};
- **max** {**int** k **in** (0: a.Length); a[k]};
- **count** {**int** k **in** (0: n); a[k] % 2 == 0};

Intervals:

- The half-open interval {**int** i **in** (0: n)} means i satisfies 0 <= i < n
- The closed (inclusive) interval {**int** k **in** (0..n)} means i satisfies 0 <= i <= n

Invariants: Products

```
public static int Product(int[]! a)
  ensures result == product{int i in (0: a.Length); a[i]};
{
  int ans = 1;
  for (int n = 0; n < a.Length; n++)
    invariant n <= a.Length;
    invariant ans == product{int i in (0: n); a[i]};
  {
        ans *= a[n];
  }

  return ans;
}
```

<Product.ssc>

Quantifiers in Spec#

We may also use **filters**:

- sum {int k in (0: a.Length)**, 5<=k**; a[k]};
- product {int k in (0..100)**, k % 2 == 0;** k};

Note that the following two expressions are equivalent:

- sum {int k in (0: a.Length)**, 5<=k**; a[k]};
- sum {int k in (5: a.Length); a[k]};

Using Filters

```
public static int SumEvens(int[]! a)
 ensures result == sum{int i in (0: a.Length); a[i] % 2 == 0; a[i]};
 {
   int s = 0;
   for (int n = 0; n < a.Length; n++)
     invariant n <= a.Length;
     invariant s == sum{int i in (0:n), a[i] % 2 == 0; a[i]};

    {
        if (a[n] % 2 == 0)
        {
                s += a[n];
        }
    }
   return s;
 }
```

Filters the even values
From the quantified range

Segment Sum Example:

```
public static int SeqSum(int[] a, int i, int j)
 {
    int s = 0;
     for (int n = i; n < j; n++)
    {
           s += a[n];
    }
    return s;
}
```

<SegSum.ssc>

Using Quantifiers in Spec#

A method that sums the elements in a segment of an array `a` i.e. `a[i] + a[i+1] + … + a[j-1]` may have the following contract:

```
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]};
```

Post condition

Precondition

Non-null type

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:Rolling your own!

```
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;
}
```

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

Writing Invariants

Some more examples …

Invariant variations: Sum0

```
public static int Sum0(int[]! a)
ensures result == sum{int i in (0 : a.Length); a[i ]};
{   int s = 0;
    for (int n = 0; n < a.Length; n++)
    invariant n <= a.Length && s == sum{int i in (0: n); a[i]};
    {
        s += a[n];
    }
    return s;
}
```

This loop invariant focuses on what has been summed so far.

Invariant variations: Sum1

```
public static int Sum1(int[]! a)
ensures result == sum{int i in (0 : a.Length); a[i ]};
{   int s = 0;
    for (int n = 0; n < a.Length; n++)
    invariant n <= a.Length &&
          s + sum{int i in (n: a.Length); a[i]}
                          == sum{int i in (0: a.Length); a[i]}
    {
          s += a[n];
    }
    return s;
}
```

This loop invariant focuses on what is yet to be summed.

Invariant variations: Sum2

```
public static int Sum2(int[]! a)
ensures result == sum{int i in (0: a.Length); a[i]};
{   int s = 0;
    for (int n = a.Length;0 <= --n; )
    invariant 0 <= n && n <= a.Length &&
                    s == sum{int i in (n: a.Length); a[i]};
    {
        s += a[n];
    }
    return s;
}
```

This loop invariant that focuses on what has been summed so far

Invariant variations:Sum3

```
public static int Sum3(int[]! a)
ensures result == sum{int i in (0 : a.Length); a[i ]};
{   int s = 0;
    for (int n = a.Length; 0<= --n)
    invariant 0 <= n && n<= a.Length &&
          s + sum{int i in (0: n); a[i]}
                           == sum{int i in (0: a.Length); a[i]}
    {
          s += a[n];
    }
    return s;
}
```

This loop invariant focuses on what has been summed so far

The *count* Quantifier

```
public int Counting(int[]! a)
    ensures result == count{int i in (0: a.Length); a[i] == 0};
 {
   int s = 0;
   for (int n = 0; n < a.Length; n++)
     invariant n <= a.Length;
     invariant s == count{int i in (0: n); a[i] == 0};
   {
     if (a[n]== 0) s = s + 1;
   }
   return s;
 }
}
```

Counts the number of 0's in an int []! a;

The *min* Quantifier

```
public int Minimum()
 ensures result == min{int i in (0: a.Length); a[i]};
{
  int m = System.Int32.MaxValue;
  for (int n = 0; n < a.Length; n++)
   invariant n <= a.Length;
   invariant m == min{int i in (0: n); a[i]};
  {
      if (a[n] < m)
            m = a[n];
   }
  }
  return m;
}
```

Calculates the minimum value in an int []! a;

The *max* Quantifier

```
public int MaxEven()
ensures result == max{int i in (0: a.Length), a[i] % 2== 0;a[i]};
 {
  int m = System.Int32.MinValue;
  for (int n = 0; n < a.Length; n++)
   invariant n <= a.Length;
   invariant m == max{int i in (0: n), a[i] % 2== 0; a[i]};
  {
      if (a[n] % 2== 0 && a[n] > m)
           m = a[n];
  }
  return m;
 }
```

Calculates the maximum even value in an int []! a;

How to help the verifier …

Recommendations when using comprehensions:
• Write specifications in a form that is as close to the code as possible.
• When writing loop invariants, write them in a form that is as close as possible to the postcondition

In our *SegSum* example where we summed the array elements a[i] … a[j-1], we could have written the postcondition in either of two forms:

```
ensures result == sum{int k in (i: j); a[k]};
ensures result ==
        sum{int k in (0: a.Length), i <= k && k < j; a[k]};
```

How to help the verifier …

Recommendation: When writing loop invariants, write them in a form that is as close as possible to the postcondition.

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

OR

```
ensures result ==
        sum{int k in (0: a.Length), i <= k && k < j; a[k]};
invariant 0 <= n && n <= a.Length;
invariant s == sum{int k in (0: n), i <= k && k < j; a[k]};
```

Some Additional Examples

<InsertionSort.ssc>

<BinarySearch.ssc>

<Rev.ssc>

…

Insertion Sort

```
public static void sortArray( int[]! a )
modifies a[*];
ensures forall{int j in (1:a.Length);(a[j-1] <= a[j])};
{
    int k;  int t;
    if (a.Length > 0){
       k=1;
       while(k < a.Length)
        invariant 1 <= k && k <= a.Length;
        invariant forall {int j in (1:k), int i in (0:j);(a[i] <= a[j])};
          {
                    //see next slide for nested loop
          }
}
```

<InsertionSort.ssc>

Nested loop of Insertion Sort

```
for( t = k;  t>0 && a[t-1]>a[t];  t-- )
     invariant 0<=t && t<=k && k < a.Length;
     invariant forall{int j in (1:k+1),
                       int i in (0:j);  j==t || a[i] <= a[j] };
   {
          int temp;
          temp = a[t];
          a[t] = a[t-1];
          a[t-1] = temp;
   }
   k++;
```

Some more difficult examples…

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

- A method of programming. Edsger W. Dijkstra and W. H. J. Feijen

Class Contracts

Code Examples on

http://ifm2010.loria.fr/satellite.html

See subfolder Part2

Object Invariants

- Specifying the rules for using methods is achieved through contracts, which spell out what is expected of the caller (preconditions) and what the caller can expect in return from the implementation (postconditions).
- To specify the design of an implementation, we use an assertion involving the data in the class called an *object invariant*.
- Each object's data fields must satisfy the invariant at all stable times
- <RockBand.ssc>

Invariants Example:RockBand1

```
public class RockBand
{       int shows;
        int ads;

        invariant shows <= ads;
        public void Play()
        {
                ads++;
                shows++;
        }
}
```

<RockBand1.ssc>

Broken Invariant:RockBand2

```
public class RockBand
{       int shows;
        int ads;

        invariant shows <= ads;
        public void Play()
        {
                shows++;
                ads++;
        }
}
```

Object Invariants:RockBand2

public class RockBand

RockBand2.ssc(13,5): Error: Assignment to field RockBand.shows of non-exposed target object may break invariant: shows <= ads

Spec# program verifier finished with 4 verified, 1 error

```
        {
            shows++;
            ads++;
        }
}
```

Expose Blocks:RockBand3

```
public class RockBand
{       int shows;
        int ads;
        invariant shows <= ads;
        public void Play()
        {
                expose(this)
                {       shows++;
                        ads++;
                }
        }
}
```

Method Reentrancy:RockBand4

```
public class RockBand
{       int shows;
        int ads;
        invariant shows <= ads;
        public void Play()
        {
                expose(this)
                {       shows++;
                        Play();
                        ads++;
                }
        }
}
```

Method Reentrancy:RockBand4

Verifying RockBand.Play ...
RockBand4.ssc(20,3): Error:

The call to RockBand.Play()
requires target object to be peer consistent

```
        {       expose(this)
                {       shows++;
                        Play();
                        ads++;
                }
        }
}
```

Method Reentrancy:RockBand5

```
public class RockBand
{       int shows;
        int ads;
        invariant shows <= ads;
        public void Play()
        {
                expose(this)
                {       shows++;
                        ads++;
                }
                Play();
        }
}
```

<RockBand6.ssc>

Establishing the Invariant

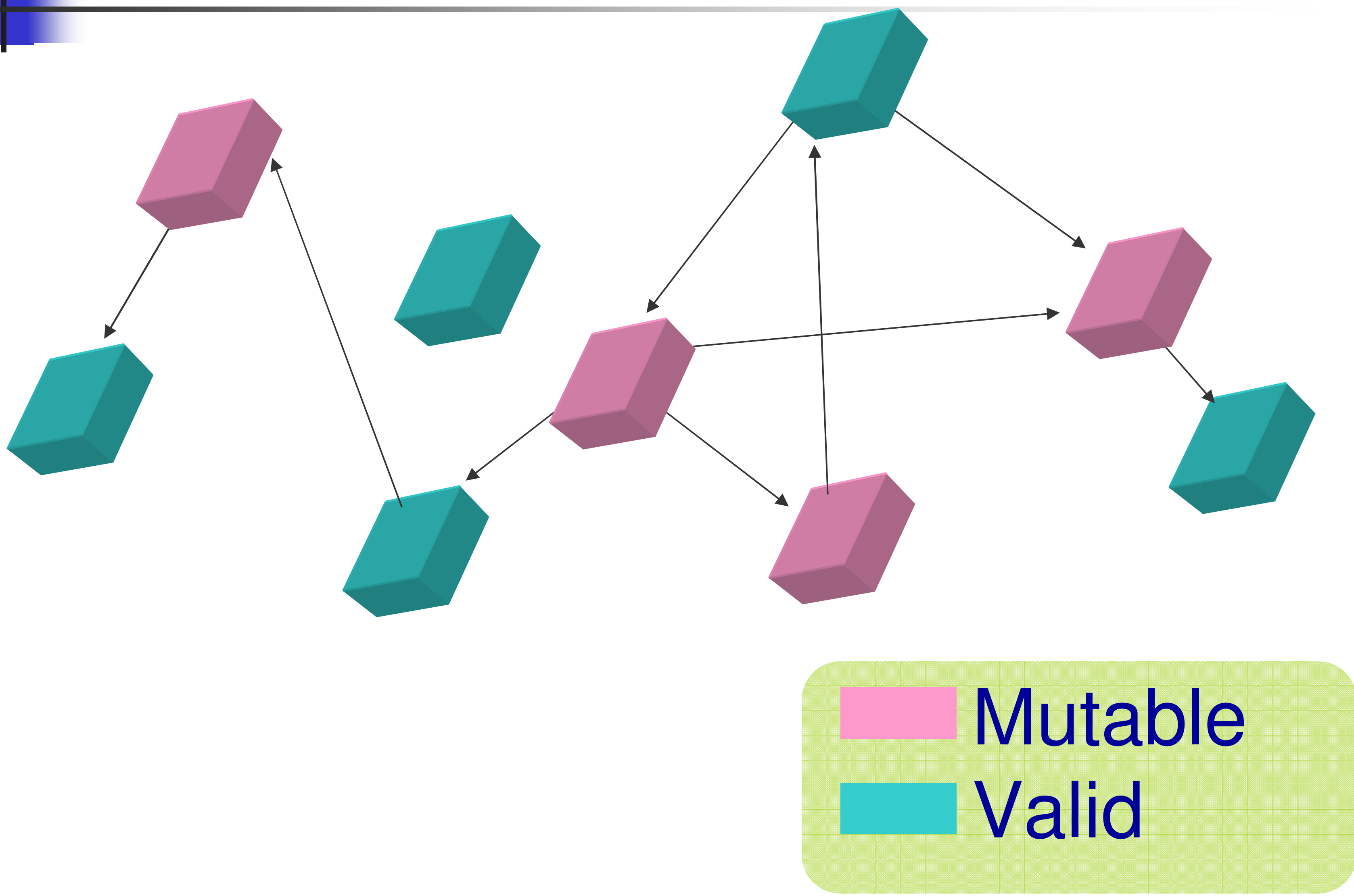
```
public class RockBand
{       int shows;
        int ads;
        invariant shows <= ads;
        public RockBand()
        {       shows = 0
                ads = shows *100;
        }
        ...
}
```

Object states

- Mutable
  - Object invariant might be violated
  - Field updates are allowed
- Valid
  - Object invariant holds
  - Field updates allowed only if they maintain the invariant

Rosemary.Monahan@NUIM.ie

The Heap (the Object Store)



Rosemary.Monahan@NUIM.ie

Summary for simple objects

$(\forall o \bullet\ o.\text{mutable} \vee \text{Inv}(o))$

- invariant … this.f …;
- x.f = E;

Check:
x is mutable
or
the assignment maintains
the invariant

$o.\text{mutable} \equiv \neg\, o.\text{valid}$

To Mutable and Back: Expose Blocks

```
public class RockBand
{   int shows;
    int ads;
    invariant shows <= ads;
    public void Play()
      modifies shows, ads;
      ensures ads== old(ads)+1 && shows ==old(shows)+1
    {      expose(this) {
                shows++;
                ads ++;
           }
    }
}
```

changes this from valid to mutable

can update ads and shows because this.mutable

changes this from mutable to valid

To Mutable and Back: Expose Blocks

```
class Counter{
    int c;
    bool even;
    invariant 0 <= c;
    invariant even <==> c % 2 == 0;
    …
    public void Inc ()
      modifies c;
      ensures c == old(c)+1;
    {     expose(this) {
              c ++;
              even = !even ;
          }
    }
}
```

changes this from valid to mutable

can update c and even, because this.mutable

changes this from mutable to valid

Invariants: Summary

```
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

Aggregate Objects and Ownership

Sub-Object Example

```
public class Guitar {
        public int solos;

        public Guitar()
         ensures solos == 0;
         {
         }
        public void Strum()
         modifies solos;
         ensures solos == old(solos) + 1;
         {
                solos++;
        }
}
```

Aggregate-Object Example

```
public class RockBand {
  int songs;
  Guitar gt;
  invariant songs == gt.solos;

  public void Play()
{
        gt.Strum();
        songs++;

  }
```

```
public RockBand()
{
   songs = 0;
   gt = new Guitar();
}
```

Aggregate-Object Example

```
public class RockBand {
  int songs;
  Guitar gt;
  invariant songs == gt.solos;

  public void Play()
{
          gt.Strum();
          songs++;

  }
```

```
public RockBand()
{
    songs = 0;
    gt = new Guitar();
}
```

RockBand[Rep].ssc(7,22): error CS2696: Expression is not admissible: it is **not visibility-based**, and first access 'gt' is **non-rep** thus further field access is not admitted.

Aggregate-Objects

- In Spec#, fields that reference a sub-object of the aggregate object are declared with the [Rep] attribute, where “rep” stands for “representation”.
- This makes it possible for the program text to distinguish between component references and other object references that a class may have.
- To keep track of which objects are components of which aggregates, Spec# uses the notion of object ownership.
- An aggregate object owns its component objects.

Visibility Based Invariants

- Visibility-based invariants allow the invariant to dereference fields that are not declared with [Rep]
  - Visibility-based invariants are useful to specify invariants of object structures that are not aggregates.
- A visibility-based invariant may dereference a field only if the declaration of the invariant is visible where the field is declared.
  - This allows the static verifier to check for every field update that all objects whose visibility-based invariants depend on that field are exposed.

Rosemary.Monahan@NUIM.ie

Aggregate-Object

```
public class RockBand {
  int songs;
  [Rep] Guitar ! gt;
  invariant songs == gt.solos;

  public void Play()
{
        gt.Strum();
        songs++;

  }
```

```
public void Client() {
  RockBand b = newRockBand();
  b.Play();
  b.Play();
 }
```

To fix the error we annotate Guitar with **[Rep]** making the RockBand b the owner of b.gt. (We also make it non null.)

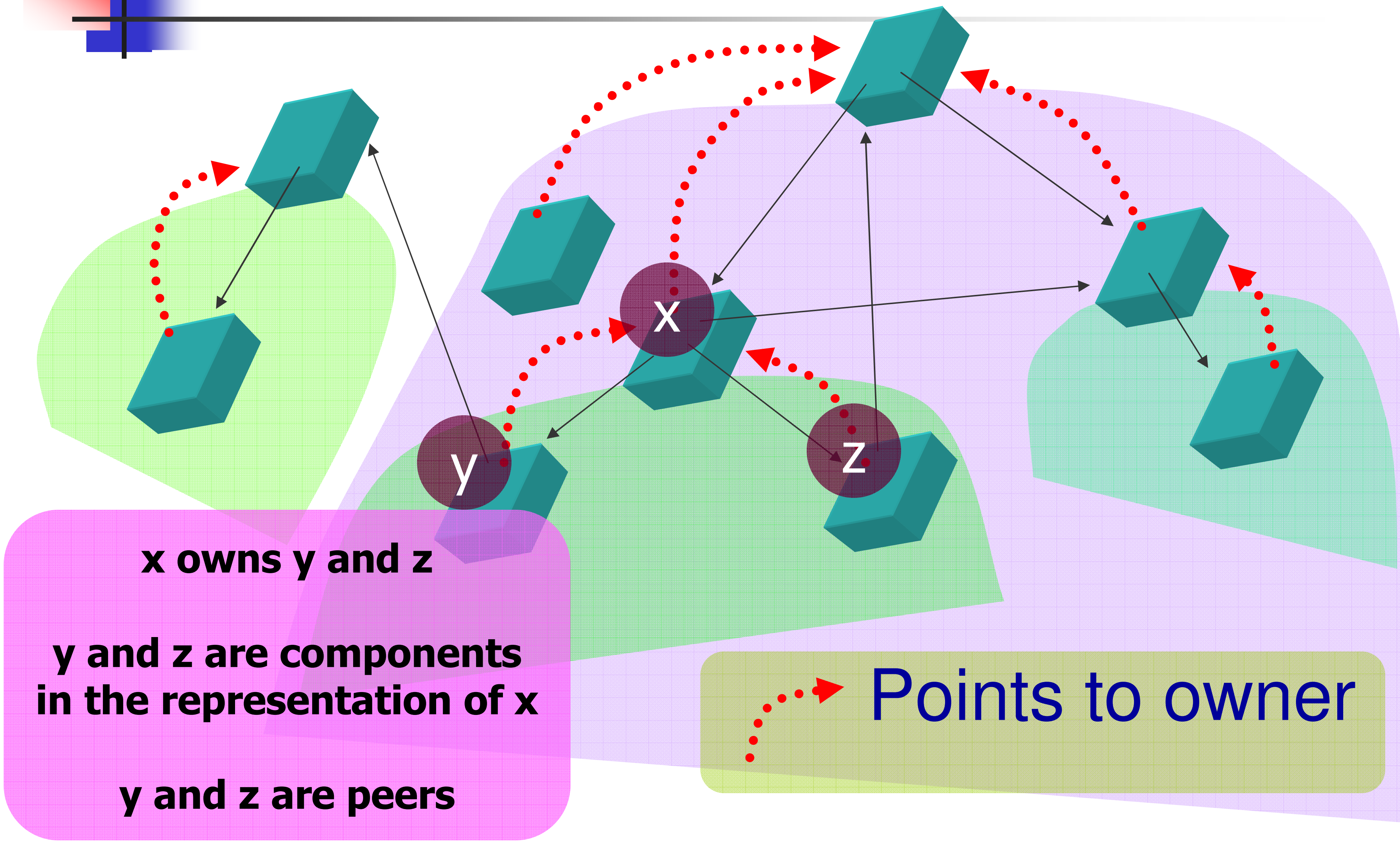
Aggregate-Object

```
public class RockBand {
  int songs;
  [Rep] Guitar ! gt;
  invariant songs == gt.solos;

  public void Play()
{
        gt.Strum();
        songs++;

  }
```

```
public void Client() {
  RockBand b = new RockBand();
  b.Play();
  b.Play();
 }
```

Error: The call to Guitar.Strum() requires target object to be **peer consistent** (owner must not be valid)

Peers?



Rosemary.Monahan@NUIM.ie

Peer Consistent?

Object states Reminder

- Mutable
  - Object invariant might be violated
  - Field updates are allowed
- Valid
  - Object invariant holds
  - Field updates allowed only if they maintain the invariant

We now sub-divide valid objects into consistent objects and committed objects.

Peer Consistent ?

A valid object is consistent if

- it has no owner object or
- its owner is mutable.

This is the typical state in which we apply methods to the object, for there is no owner that currently places any constraints on the object.

A valid object is committed it does have an owner and that owner is in the valid state. Intuitively, this means that any operation on the object must first consult with the owner.

Rosemary.Monahan@NUIM.ie

Valid objects sub-divided

- A default precondition of a method is that the receiver be consistent (the methods receiver must be mutable)
- Therefore to operate on components of the receiver, the method body must change the receiver into the mutable state
- We achieve this using an expose statement

What was the Error?

```
public class RockBand {
  int songs;
  [Rep] Guitar ! gt;
  invariant songs == gt.solos;

  public void Play()
{
          gt.Strum();
          songs++;
```

```
public void Client() {
  RockBand b = new RockBand();
  b.Play();
  b.Play();
```

Error: The call to Guitar.Strum() requires target object to be **peer consistent** (owner must not be valid)

How do we change an object from valid to mutable?

Aggregate-Object

```
public class RockBand {
  int songs;
  [Rep] Guitar ! gt;
  invariant songs == gt.solos;

  public void Play()
{
          expose(this)
          {    gt.Strum();
               songs++;
          }

  }
```

```
public void Client() {
RockBand b = new RockBand();
 b.Play();
 b.Play();
}
```

Ownership Based Invariants

```
public class RockBand {
  int songs;
  [Rep] Guitar ! gt;
  invariant songs == gt.solos;

  public void Play()
{
          expose(this)
          {    gt.Strum();
               songs++;
```

```
public void Client() {
RockBand b = new RockBand();
 b.Play();
 b.Play();
}
```

<Rockband[Rep].ssc>

Note the **Ownership based invariant** ... This invariant also requires that gt is a [Rep] object as it dereferences gt.

A Note on Modifies clauses

- In our example when the Guitar gt is annotated as [Rep], the method Play() does not need to specify modifies gt*
- This is a private implementation detail so the client doesn’t need to see it

Subtyping and Inheritance

Inheritance
[Additive] and Additive Expose
Overriding methods – inheriting contracts

Base Class

```
public class Car
{
    protected int speed;
    invariant 0 <= speed;

    protected Car()
    {     speed = 0;
    }

public void SetSpeed(int kmph)
  requires 0 <= kmph;
  ensures speed == kmph;
{
    expose (this) {
        speed = kmph;
    }
}

}
```

Inheriting Class: Additive Invariants

```
public class LuxuryCar:Car
{
    int cruiseControlSettings;
    invariant cruiseControlSettings == -1 || speed == cruiseControlSettings;

    LuxuryCar()
    {
        cruiseControlSettings = -1;
    }


}
```

The speed attribute of the superclass is mentioned in the the object invariant of the subclass

Change required in the Base Class

```
public class Car{

    [Additive] protected int speed;
    invariant 0 <= speed;

    protected Car()
    {     speed = 0;
    }

    ...
```

The [Additive] annotation is needed as speed is mentioned in the object invariant of LuxuryCar

Additive Expose

```
[Additive] public void SetSpeed(int kmph)
    requires 0<= kmph;
    ensures speed == kmph;
{
        additive expose (this) {
         speed = kmph;
        }
}
```

An additive expose is needed as the SetSpeed method is inherited and so must expose LuxuryCar if called on a LuxuryCar Object

Specification Inheritance

Spec# verifies a call to a virtual method M against the specification of M in the *static* type of the receiver and enforces that all overrides of M in subclasses live up to that specification

An overriding method inherits the precondition, postcondition, and **modifies** clause from the methods it overrides.

It may declare additional postconditions, but not additional preconditions or **modifies** clauses because a stronger precondition or a more permissive **modifies** clause would come as a surprise to a caller of the superclass method

Virtual Methods

```
public class Car{

    [Additive] protected int speed;
    invariant 0 <= speed;

    protected Car()
    {       speed = 0;
    }

    [Additive] virtual public void SetSpeed(int kmph)
      requires 0 <= kmph;
      ensures speed == kmph;
    {
            additive expose (this) {
               speed = kmph;
            }
    }
}
```

Overriding Methods

```
public class LuxuryCar:Car{
protected int cruiseControlSettings;
invariant cruiseControlSettings == -1 || speed == cruiseControlSettings;

…

    [Additive] override public void SetSpeed(int kmph)
     //requires 0<= kmph is not allowed
     ensures cruiseControlSettings == kmph && speed == cruiseControlSettings;
    {
            additive expose(this){
                    cruiseControlSettings = kmph;
                            additive expose((Car)this){
                            speed =cruiseControlSettings;}
            }               }
}
```

Class Frames

Class Frame: refers to a particular class declaration, not its subclasses or its superclasses. Each frame can declare its own invaraints which constrain the fields declared in that frame.

```
[Additive] override public void SetSpeed(int kmph)
    ensures cruiseControlSettings == kmph && speed == cruiseControlSettings;
  {
        additive expose(this){
                cruiseControlSettings = kmph;
                        additive expose((Car)this){
                        speed =cruiseControlSettings;}
        }               }
}
```

Class Frames ctd.

- We refine the notions of mutable and valid to apply individually to each class frame of an object.
- We say an object is consistent or committed only when all its class frames are valid. i.e. they apply to the object as a whole.

- The expose statement changes one class frame of an object from valid to mutable.
- The class frame to be changed is indicated by the static type of the (expression denoting the) given object. E.g. expose (this) and expose ((Car)this).

Peers

[Peer]

- It is appropriate that one object owns another if the other is part of the private implementation of the first as with [Rep] objects.
- Sometimes, one object holds a reference to another for some other reason ... [Peer]

Example:

A linked-list node n holds a reference to the next node in the list, n.next. However, n.next is usually not thought of as an implementation detail or component of n. Rather, n and n.next have a more equal relationship, and both nodes may be part of the same enclosing aggregate object.

[Peer] Example

```
class Node {

   public string key;
   public int val;

   [Peer] public Node next;
   public Node(string key, int val) {
            this.key = key;
            this.val = val;
   }
}
```

<Dictionary.ssc>

Rep vs. Peer Guidelines

- Use [Rep] where possible as it strengthens encapsulation and simplifies verification.
- Use [Rep] when
  - the field references an object whose type or mere existence is an implementation detail of the enclosing class
- Use [Peer] when
  - two objects are part of the same aggregate or
  - the objects are part of a recursive data structure or
  - the field references an object that can also be accessed by clients of the enclosing class
    - e.g. a collection is not an implementation detail of its iterator and clients of the iterator may also access the collection directly.

<collection.ssc>

Back to Aggregates…

```
public class Radio {
  public int soundBoosterSetting;
  invariant 0 <= soundBoosterSetting;

  public bool IsOn()
  {
    int[] a = new int[soundBoosterSetting];
    bool on = true;
    // ... compute something using "a", setting "on" appropriately
    return on;
  }
}
```

Peer

```
public class Car {
  int speed;
  invariant 0 <= speed;
  [Peer] public Radio! r;

  public Car() {
    speed = 0;
    r = new Radio();
  }
```

```
public void SetSpeed(int kmph)
    requires 0 <= kmph;
    modifies this.*, r.*;
  {
    speed = kmph;
    if (r.IsOn()) {
      r.soundBoosterSetting =
                          2 * kmph;
    }
  }
```

[Peer] there is only one owner- the owner of the car and radio

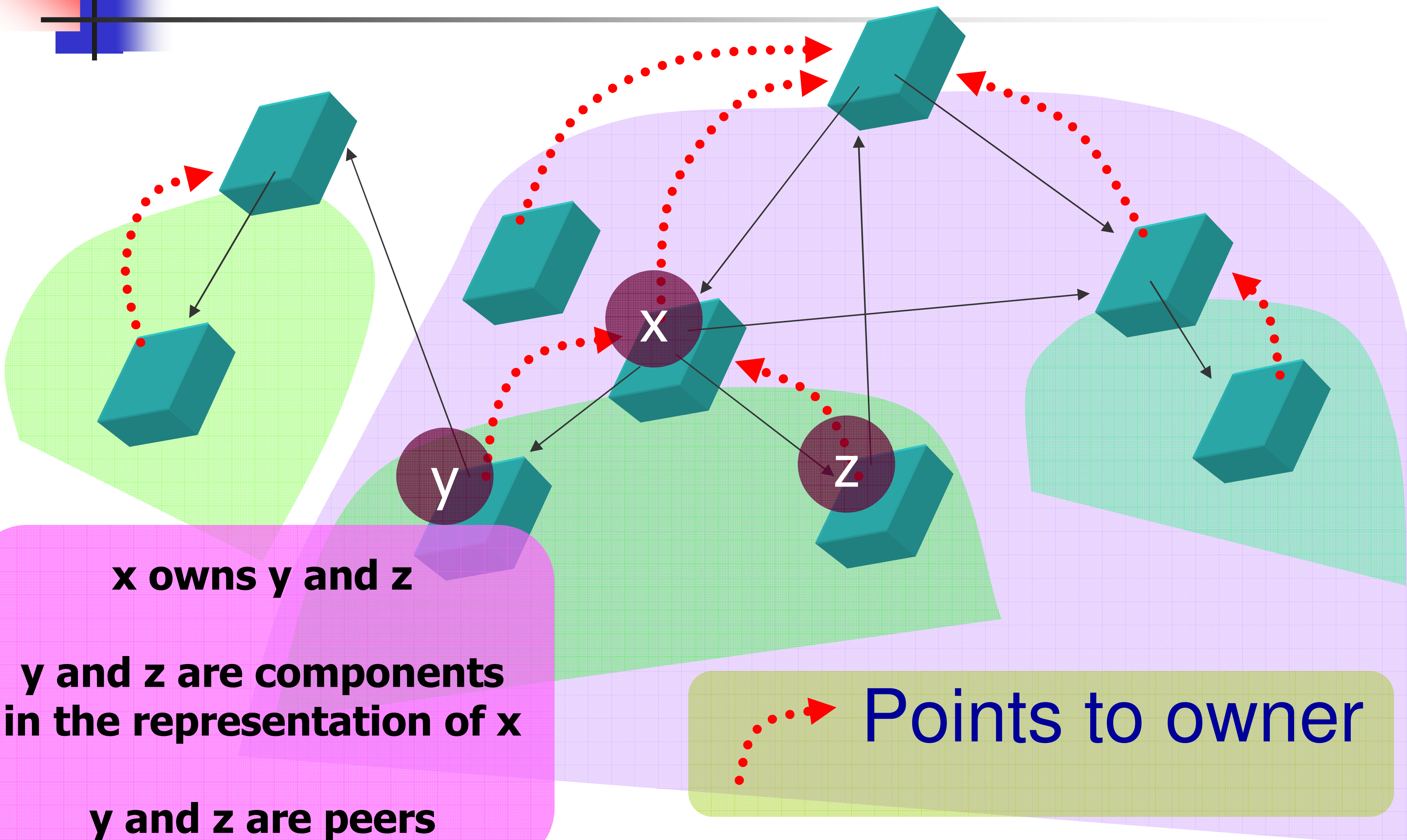
Rep

```
public class Car {
  int speed;
  invariant 0 <= speed;
  [Rep] Radio! r;

  public Car() {
    speed = 0;
    r = new Radio();
  }
```

```
public void SetSpeed(int kmph)
    requires 0 <= kmph;
    modifies this.*;
  {
    expose (this) {
      speed = kmph;
      if (r.IsOn()) {
        r.soundBoosterSetting =
                             2 * kmph;
      }
    }
  }
}
```

[Rep] there is an owner of car and an owner of radio

Ownership domains



x owns y and z

y and z are components in the representation of x

y and z are peers

Points to owner

Rosemary.Monahan@NUIM.ie

Rep

```
public class Car {
  int speed;
  invariant 0 <= speed;
  [Rep] Radio! r;

  public Car() {
    speed = 0;
    r = new Radio();
  }
```

```
public void SetSpeed(int kmph)
    requires 0 <= kmph;
    modifies this.*;
  {
    expose (this) {
      speed = kmph;
      if (r.IsOn()) {
        r.soundBoosterSetting =
                              2 * kmph;
```

Making radio [Rep] makes Radio peer valid

Need the expose block to make it peer consistent.

Rep

Why ever use Rep?

We gain Information Hiding, e.g. if we add an invariant to Car with reference to radio components we get a warning via a visibility based error

```
r = new Radio();
}
```

Making radio [Rep] makes Radio peer valid

Need the expose block to make it peer consistent.

Ownership of Array Elements

[ElementsRep] attribute on an array expresses that every non-null element of the array is owned by the enclosing object.

Example: <DrawingEngine.ssc>

- With the [ElementsRep] attribute on the array called steps and expose(this) in the code, steps[i] is peer consistent.
- DrawingEngine methods are then allowed to modify the elements of steps[] because they are components of DrawingEngine

Ownership of Array Elements

- Spec# also provides an attribute [ElementsPeer], which expresses that the array elements are peers of the object containing the [ElementsPeer] field.

Rosemary.Monahan@NUIM.ie

Ownership for Generics

Ownership for generics is very generics, similar to arrays, with two differences.

- We specify the owner individually for each generic type argument.
  - This is done by passing the number of the type argument to the attributes [ElementsRep] and [ElementsPeer]

    E.g. [ElementsPeer(0)] Dictionary<K,V> dict;

    adds implicit checks and assumptions to all operations on dict that values of type K are peers of this.
- There are no automatic owner assignment when objects are passed to operations of generic classes.

Using Collections

```
public class Car {
  [Rep] [ElementsPeer]
  List<Part!>! spares =
    new List<Part!>();


  public void AddPart() {
    expose (this) {
      Part p = new Part();
      Owner.AssignSame(p, Owner.ElementProxy(spares));
      spares.Add(p);
    }
  }
```

```
public void UsePart()
  modifies this.**;
 {
  if (spares.Count != 0) {
    Part p = spares[0];
    p.M();
   }
  }
}
```

[Rep] locks

```
public class Car {
  int speed;
  invariant 0 <= speed;
  [Rep] public Radio! r;
  invariant r.soundBoosterSetting == 2 * speed;

  [Rep] bool[]! locks;
  invariant locks.Length == 4;
```

Capture Rep objects

```
public Car([Captured] bool[]! initialLocks)
  requires initialLocks.Length == 4;
{
  speed = 0;
  r = new Radio();
  locks = initialLocks;
}
```

We can't take ownership of initialLocks as someone else might own it so we need to capture it

Modifies clause expanded

```
public void SetSpeed(int kmph)
   requires 0 <= kmph;
   modifies this.*, r.*, locks[*];
  {
   expose (this) {
     if (kmph > 0) {
       locks[0] = true;
     }
     speed = kmph;
     r.soundBoosterSetting = 2 * kmph;
   }
  }
}
```

Peer

```
public class Car {
  int speed;
  invariant 0 <= speed;
  [Rep] public Radio! r;
  invariant r.soundBoosterSetting == 2 * speed;

  [Peer] bool[]! locks;
  invariant locks.Length == 4;
```

[Captured] and [Peer]

[Captured]

```
public Car(bool[]! initialLocks)
  requires initialLocks.Length == 4;
  ensures Owner.Same(this, initialLocks);
{
  speed = 0;
  r = new Radio();
  Owner.AssignSame(this, initialLocks);
  locks = initialLocks;
}
```

Set the owner manually

The constructor has the [Captured] attribute, indicating that the constructor assigns the owner of the object being constructed.

Manual Loop Invariants

```
public void SetSpeed(int kmph)
    requires 0 <= kmph;
    modifies this.*, locks[*];
  {       expose (this) {
              if (kmph > 0)
              {
                      bool[] prevLocks = locks;
                      for (int i = 0; i < 4; i++)
                      invariant locks == prevLocks && locks.Length == 4;
                      {         locks[i] = true;
                      }
               }
              speed = kmph;
             r.soundBoosterSetting = 2 * kmph;
          }
  }
```

Manual Loop invariant to satisfy the modifies clause

Pure Methods

Pure Methods

- If you want to call a method in a specification, then the method called must be *pure*
- This means it has no effect on the state of objects allocated at the time the method is called
- Pure methods must be annotated with [Pure], possibly in conjunction with:
  - [Pure][Reads(ReadsAttribute.Reads.Everything)] methods may read anything
  - [Pure][Reads(ReadsAttribute.Reads.Owned)] (same as just [Pure]) methods can only read the state of the receiver object and its (transitive) representation objects
  - [Pure][Reads(ReadsAttribute.Reads.Nothing)] methods do not read any mutable part of the heap.
- Property getters are [Pure] by default

Using *Pure* Methods

- Declare the pure method within the class definition e.g.

```
[Pure] public bool Even(int x)
  ensures result == (x % 2 == 0);
{
      return x % 2 == 0;
}
```

Using *Pure* Methods

```
public int SumEven()
  ensures result ==
        sum{int i in (0: a.Length), Even(a[i]); a[i]};
 {
  int s = 0;
  for (int n = 0; n < a.Length; n++)
    invariant n <= a.Length;
    invariant s == sum{int i in (0: n) , Even(a[i]); a[i]};
  {    if (Even(a[i]))
        s += a[n];
  }
  return s;
}
```

Pure method calls

Boogie 2

An intermediate language designed to accommodate the encoding of verification conditions for imperative, object-oriented programs.

Code Examples on
http://ifm2010.loria.fr/satellite.html
See subfolder Part3

Rosemary.Monahan@NUIM.ie

Microsoft Research Boogie

CodePlex
Open Source Community

Search a

Home | Downloads | Documentation | Discussions | Issue Tracker | Source Code | People | License

View All Comments | Print View | Page Info | Change History (all pages)

Home

Microsoft Research Boogie

(aka *The World's Best Program Verification System*)
Boogie is a program verification system that produces verification conditions for programs written in an intermediate language (also named Boogie). The intermediate language is easy to target from source languages such as Spec#, C#, or even C.

- Frequently asked questions
- External Dependencies
- How to install the binaries
- How to install and build the sources
- How to contribute
- MSR Boogie site

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

Last edited Aug 7 2009 at 11:50 PM by mikebarnett, version 16

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

CURRENT Bo
DATE Tu
STATUS St
RATING N 17
MORE Vi

Activity

Page Views
Visits
Downloads
Application Ru

Related Pro

- Commor Metadat
- VCC
- Spec#

Microsoft® Research

Search Microsoft Research

Videos | Projects | Publications | People | Downloads

Home | Our Research | Collaboration | Careers

Worldwide Labs | Research Areas | Research Groups

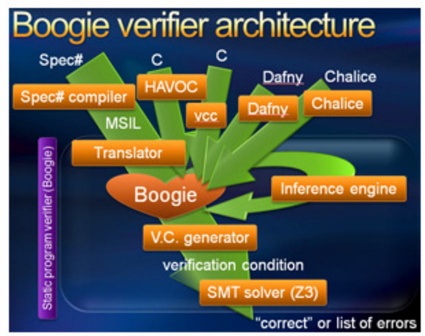
Project Tuva Enhanced Video Player
Watch the Feynman Lectures

> Projects > Boogie: An Intermediate Verification Language

Boogie: An Intermediate Verification Language

Boogie is an intermediate verification language, intended as a layer on which to build program verifiers for other languages. Several program verifiers have been built in this way, including one for the object-oriented .NET language Spec#, the HAVOC and vcc verifiers for C, the Dafny language and verifier, and the concurrent language Chalice. A previous version of the language was called BoogiePL; the new version of the language is Boogie 2.

Boogie is also the name of a **tool**. The tool accepts the Boogie language as input, optionally infers some invariants in the given Boogie program, and then generates verification conditions that are passed to an SMT solver. The default SMT solver is Z3.



In their geneses, **Boogie** and **Spec#** were developed hand in hand. For this reason, the name Boogie has been used to describe Spec#-related things. In particular, the Spec# static program verifier, which translates compiled Spec# programs (.NET bytecode) into Boogie, has been called Boogie. In fact, this **bytecode translator** is still part of the Boogie executable, so the Boogie tool accepts compiled Spec# programs (extensions .dll and .exe) in addition to Boogie programs (extension .bpl). Finally, Spec# uses an ownership-based discipline for handling object invariants

Boogie File Generation

To create Boogie PL programs use

sscboogie Program.dll /print:Program.bpl

- <MyString.ssc>
- <MyString.bpl>

Rosemary.Monahan@NUIM.ie

Boogie Declarations

Mathematical constructs to define a logical basis for the terms used in the program:

- type, const, function, and axiom.

Imperative constructs to define the behaviour of the program:

- var (global variables), procedure (declarations), and (procedure) implementation.

Boogie Statements

- x := E
- a[i] := E
- havoc x
- assert E
- assume E
- ;
- call P()
- if
- while
- break
- label:
- goto A, B

Rosemary.Monahan@NUIM.ie

Some Examples

- <Find.bpl>
- <monapoli_sum_max.bpl>
- <monapoli_search.bpl>
- <DutchFlag.bpl>
- <Bubble.bpl>

Some Research Languages that use Boogie as an Intermediate Language

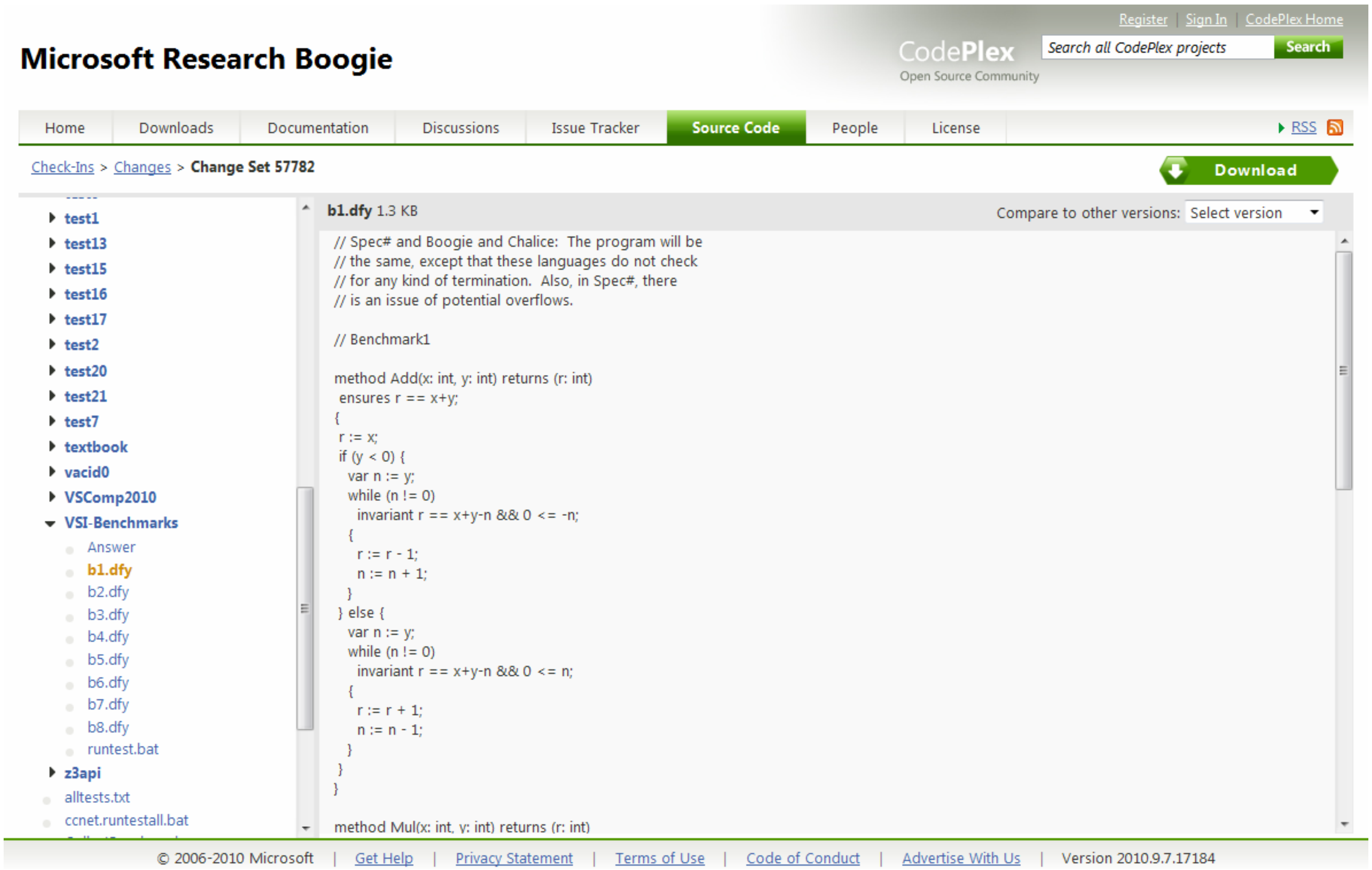
Chalice: specification and verification of concurrent programs using shared memory and mutual exclusion via locks.

Dafny: an object-based language where specifications are written in the style of dynamic frames.

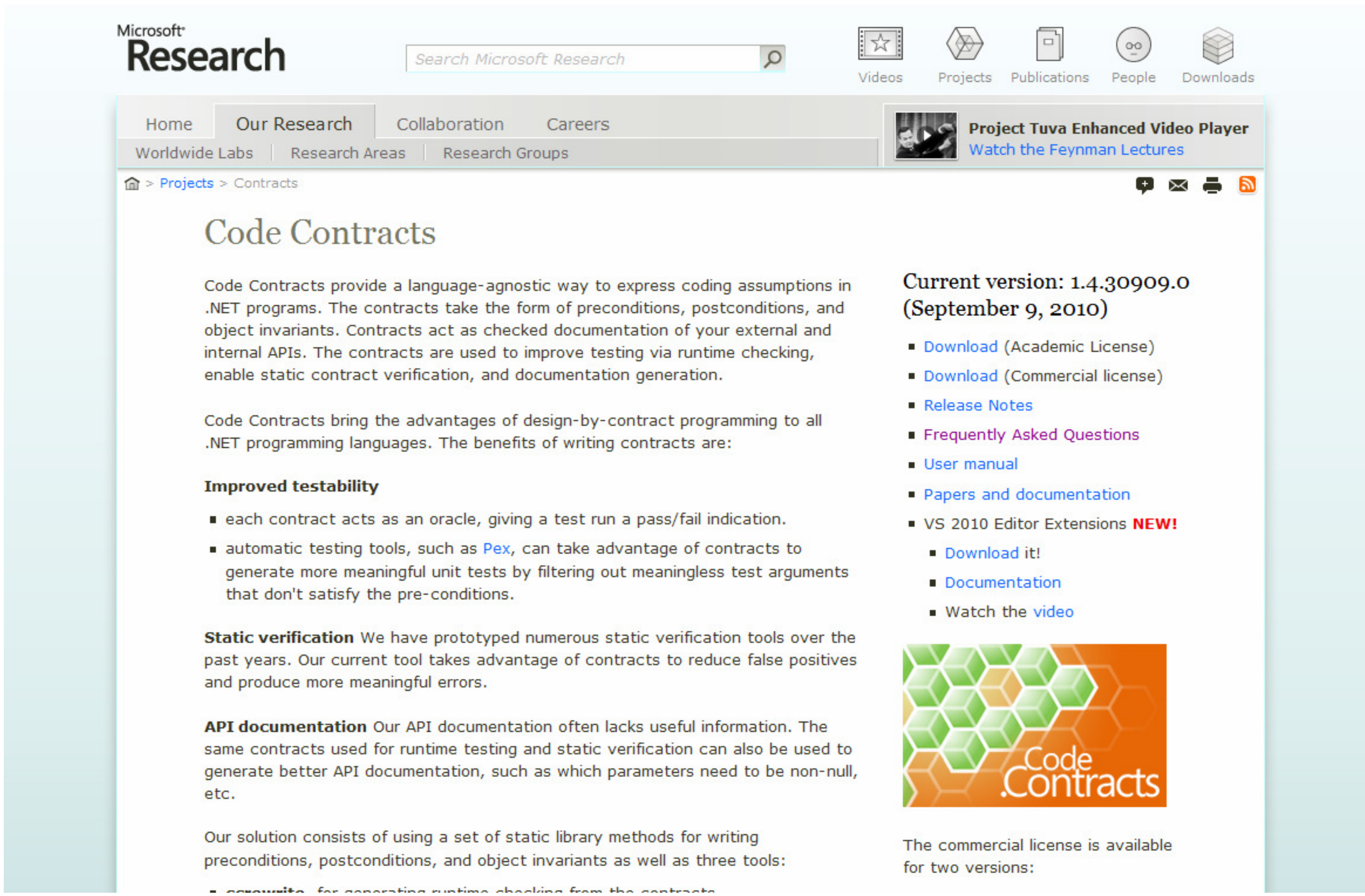
Java BML, Eiffel, C (Havoc), C (VCC), Region Logic, …

Rosemary.Monahan@NUIM.ie

VSTTE 2010: Benchmarks in Dafny



Microsoft Code Contracts



Microsoft Research

Search Microsoft Research

Videos Projects Publications People Downloads

Home | Our Research | Collaboration | Careers

Worldwide Labs | Research Areas | Research Groups

Project Tuva Enhanced Video Player
Watch the Feynman Lectures

> Projects > Contracts

Code Contracts

Code Contracts provide a language-agnostic way to express coding assumptions in .NET programs. The contracts take the form of preconditions, postconditions, and object invariants. Contracts act as checked documentation of your external and internal APIs. The contracts are used to improve testing via runtime checking, enable static contract verification, and documentation generation.

Code Contracts bring the advantages of design-by-contract programming to all .NET programming languages. The benefits of writing contracts are:

Improved testability

- each contract acts as an oracle, giving a test run a pass/fail indication.
- automatic testing tools, such as Pex, can take advantage of contracts to generate more meaningful unit tests by filtering out meaningless test arguments that don't satisfy the pre-conditions.

Static verification We have prototyped numerous static verification tools over the past years. Our current tool takes advantage of contracts to reduce false positives and produce more meaningful errors.

API documentation Our API documentation often lacks useful information. The same contracts used for runtime testing and static verification can also be used to generate better API documentation, such as which parameters need to be non-null, etc.

Our solution consists of using a set of static library methods for writing preconditions, postconditions, and object invariants as well as three tools:

Current version: 1.4.30909.0 (September 9, 2010)

- Download (Academic License)
- Download (Commercial license)
- Release Notes
- Frequently Asked Questions
- User manual
- Papers and documentation
- VS 2010 Editor Extensions **NEW!**
  - Download it!
  - Documentation
  - Watch the video

Code Contracts

The commercial license is available for two versions:

Conclusions

The main contributions of the Spec# programming system are:

- a contract extension to the C# language
- a sound programming methodology that permits specification and reasoning about object invariants even in the presence of callbacks
- tools that enforce the methodology, ranging from easily usable dynamic checking to high-assurance automatic static verification
- inspired other tools such as Dafny, Chalice and Code Contracts

References and Resources

- Spec# websites http://specsharp.codeplex.com/ and http://research.microsoft.com/specsharp/

  - The Spec# programming system: An overview. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. In *CASSIS 2004*, LNCS vol. 3362, Springer, 2004.
  - Boogie: A Modular Reusable Verifier for Object-Oriented Programs. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. In *FMCO 2005*, LNCS vol. 4111, Springer, 2006.
  - Automatic verification of textbook programs that use comprehensions. K. Rustan M. Leino and Rosemary Monahan. In *Formal Techniques for Java-like Programs*, ECOOP Workshop (FTfJP'07: July 2007, Berlin, Germany), 2007.
  - The Spec# programming system: An overview. In FM 2005 Tutorial given by Bart Jacobs, K.U.Leuven, Belgium.

Tutorials and Examples

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

Spec# examples and course notes available by emailing Rosemary.Monahan@NUIM.ie and at http://www.cs.nuim.ie/~rosemary/

Further resources and papers at http://research.microsoft.com/specsharp/