Program Verification Using the Spec# Programming System

ETAPS Tutorial

K. Rustan M. Leino, Microsoft Research, Redmond

Rosemary Monahan, NUIM Maynooth & LERO

29 March 2008

Introducing Spec#

Spec#: An Overview

Installing Spec#

Using Spec#

Spec#: An Overview

The Spec# Programming System provides language and tool support for assertion checking in 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: a compiler that 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: a component (named Boogie) that generates logical verification conditions from a Spec# program. Internally, it uses an automatic theorem prover that analyzes the verification conditions to prove the correctness of the program or find 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.

This Tutorial

- Goal: Support the exploration of Spec# both for yourself and for your students.
- Tutorial Structure:
  - Getting started with Spec#
    - Overview and Installation
  - Programming in the small.
    - Preconditions, Postconditions, Loop invariants
  - Programming in the large:
    - Object invariants, Ownership

} Before Break (Getting started with Spec#; Programming in the small)

} After Break (Programming in the large)

Installing Spec#

- Download & install the latest version (April 2008) of Spec# from http://research.microsoft.com/specsharp/
  - Installation includes the compiler, VS plug-in, Boogie, Z3
  - Required: .NET
  - Recommended: Visual Studio
  - 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

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.

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

Using Boogie at the Command line

- Open the Spec# command prompt (Start -> Programs -> Microsoft Spec# Language-> Tools -> Spec# Command Prompt).
- C:\temp> ssc /t:library /debug Program.ssc

  compiles a Spec# program called Program.ssc stored in C:\temp. This generates a file called Program.dll which is also stored in C:\temp. Leave out /t:library to compile into an .exe executable.
- C:\temp> boogie Program.dll (or Program.exe) verifies the compiled file using the SMT solver Z3.
- The /trace option gives you more feedback on the verification process i.e. C:\temp> boogie Program.dll /trace
- Further switches for boogie can be seen by typing boogie /help

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# adds 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 Example

```
public class Program
{
  public static void Main(string[] args)
  {
   foreach (string arg in args) // Possible null dereference
   {
        Console.WriteLine(arg); // Possible null dereference
   }
   Console.ReadLine();
  }
}
```

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

```
public class Program
{
  public static void Main(string![]! args)
  {
        foreach (string arg in args)
        {
              Console.WriteLine(arg);
        }
        Console.ReadLine();
  }
}
```

args != null

args[i] != null

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

Initializing Non-Null Fields

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

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

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 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.

Assert

Assert Statements

```
public class Program
{
  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 Statements

```
public class Program
{
  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.

Using Spec# by C# plus Annotations

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

By enabling contracts from the Contracts pane of a C# project's properties, compiling a program will first run the C# compiler and then immediately run the Spec# compiler, which looks inside comments that begin with a ^

Design by Contract

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 ++

Spec# Method Contract

```
static int min(int x, int y)
  requires 0<=x && 0<= y ;
  ensures  x<y ? result == x: result == y;
{
        int m;
        if (x < y)
                m = x;
        else
                m = y;
        return m;
}
```

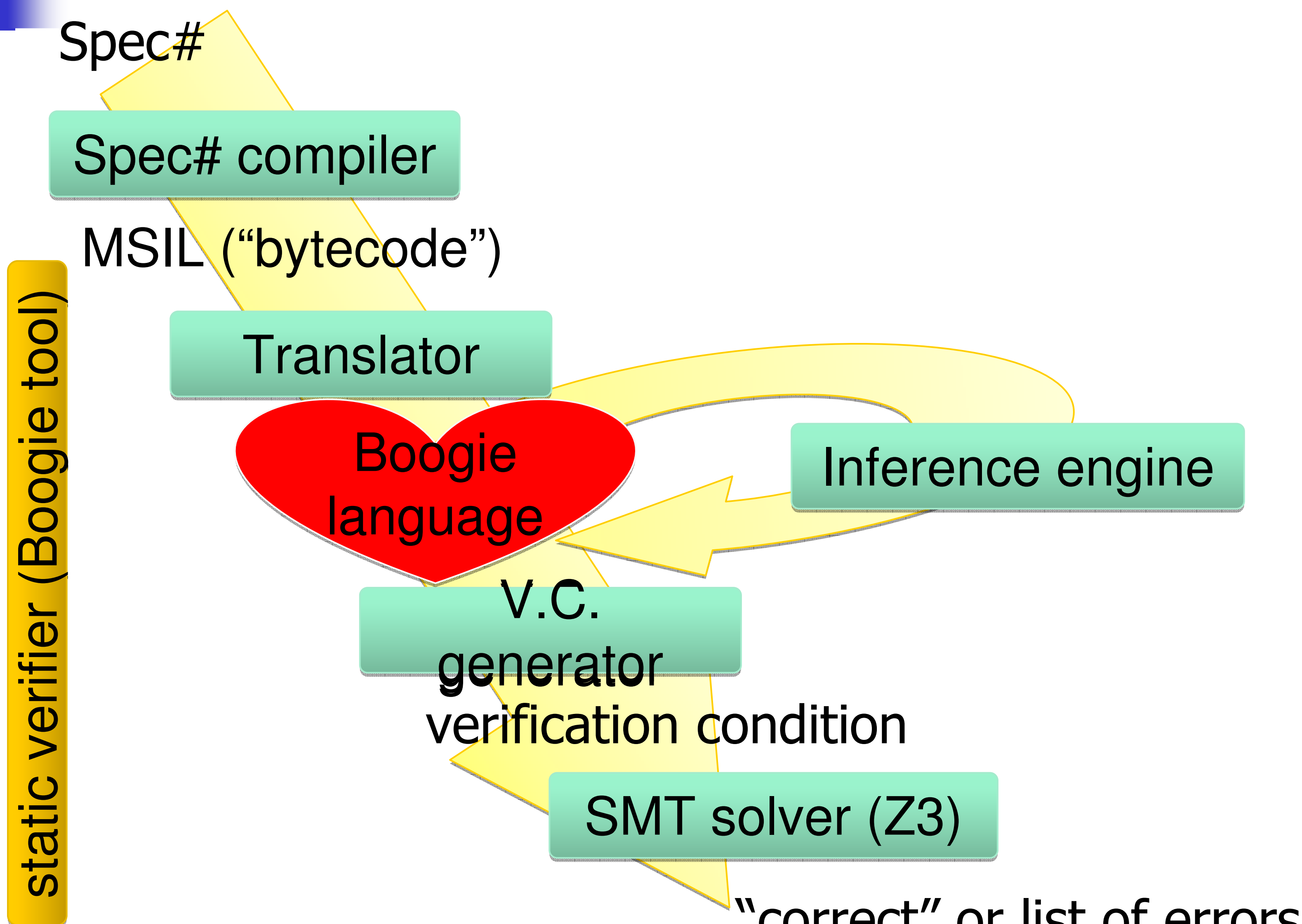
requires annotations denote preconditions

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 (yet)

Spec# verifier architecture



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;
}
```

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

Loop Invariants

Examples:
Squaring/cubing by addition – no need for quantifiers

Summing
Binary Search
Sorting
Coincidence count
gcd/lcm
Factorial

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 += x;
    x += 2;
  }
  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};
```

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;
      }
  }
```

Strengthening 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;
       }
 }
```

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:Summing Arrays

```
public static int SumValues(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;
    invariant s == sum{int i in (0: n); a[i]};
  {
        s += a[n];
  }

  return s;
}
```

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;
}
```

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;
}
```

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, 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;
}
```

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 **i <= n && n <= j;**

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

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;
}
```

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;

Another Use of Comprehension Operators

- Example expressions:
  - **min**{ x, y }
  - **max**{ a, b, c }

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 …

```
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;
}
```

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

Binary Search

```
public static int BinarySearch(int[]! a, int key)
  requires forall{int i in (0: a.Length), int j in (i: a.Length); a[i] <= a[j]};
  ensures 0 <= result ==> a[result] == key;
  ensures result < 0 ==> forall{int i in (0: a.Length); a[i] != key};
{
        int low = 0;
        int high = a.Length - 1;

        while (low <= high)
          invariant high+1 <= a.Length;
          invariant 0 <= low;
          invariant forall{int i in (0: low); a[i] != key};
          invariant forall{int i in (high+1: a.Length); a[i] != key};
```

Binary Search (cont.)

```
    {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key) {
                low = mid + 1;
        } else if (key < midVal) {
                high = mid - 1;
        } else {
                return mid; // key found
        }
    }
    return -(low + 1);  // key not found.
}
```

Insertion Sort

```
public void sortArray(int[]! a)
  modifies a[*];
  ensures forall{int j in (0: a.Length), int i in (0: j); a[i] <= a[j]};
{
    for (int k = 0; k < a.Length; k++)
           invariant 0 <= k && k <= a.Length;
           invariant forall{int j in (0: k), int i in (0: j); a[i] <= a[j]};
    {

           // Inner loop – see next slide

    }
}
```

Insertion Sort

```
for (int t = k; t >0 && a[t-1] > a[t]; t--)
  invariant 0<=t;
  invariant forall{int j in (1: t), int i in (0: j); a[i] <= a[j]};
  invariant forall{int j in (t: k+1), int i in (t: j); a[i] <= a[j]};
  invariant forall{int j in (t+1: k+1), int i in (0: t); a[i] <= a[j]};
  //set an upper bound for t (t<a.Length works too).
  invariant t <=k;
{
        int temp;
        temp = a[t];
        a[t] = a[t-1];
        a[t-1] = temp;
}
```

Greatest Common Divisor (slow)

```
static int GCD(int a, int b)
  requires a > 0 && b > 0;
  ensures result > 0 && a % result == 0 && b % result == 0;
  ensures forall{int k in (1..a+b), a % k ==0 && b % k == 0; k <= result};
  {
    int i = 1; int res = 1;
    while (i < a+b)
      invariant i <= a+b;
      invariant res > 0 && a % res == 0 && b % res == 0;
      invariant forall{int k in (1..i), a % k == 0 && b % k == 0; k <= res};
    {
      i++;
      if (a % i == 0 && b % i == 0) {
        res = i;
      }
    }
    return res;
 }
```

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

- Spec# Wiki http://channel9.msdn.com/wiki/default.aspx/SpecSharp.HomePage

Class Contracts

Pre- & Post are not Enough

```
class C {
  private int a, z;
  public void M( )
   requires a != 0;
  { z = 100 / a; }
}
```

```
class C {
  private int a, z;
  invariant a != 0;
  public void M( )
  { z = 100 / a; }
}
```

- Contracts break abstraction
- We need invariants

Pre- & Post are not Enough

```
class C {
  private int a, z;
  public void M( )
    requires a != 0;
  { z = 100 / a; }
}
```

```
class C {
  private int a, z;
  invariant a != 0;
  public void M( )
  { z = 100 / a; }
}
```

- Contracts break abstraction
- We need invariants

When used in a class the keyword *invariant*, indicates an object invariant

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

Object Invariants

```
class Counter{
   int c;
   invariant 0 <= c;

   public Counter()
   {    c= 0;
   } //invariant established & checked after construction

   public void Inc ()
     modifies c;
     ensures c == old(c)+1;
   {    c++; //invariant checked after every increment
   }
```

Breaking Object Invariants

```
public void BadInc () //Not Allowed – may break the Invariant
   modifies c;
   ensures c == old(c)+1;
{  c--; //Error here
   c+=2; //invariant checked after every increment
}
```

Establishing Object Invariants

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

  public Counter()
  {
        c= 0; // OK to break inv here(in constructor)
        even = true;
  } //invariant established & checked after construction

  ...
```

Breaking the Invariant

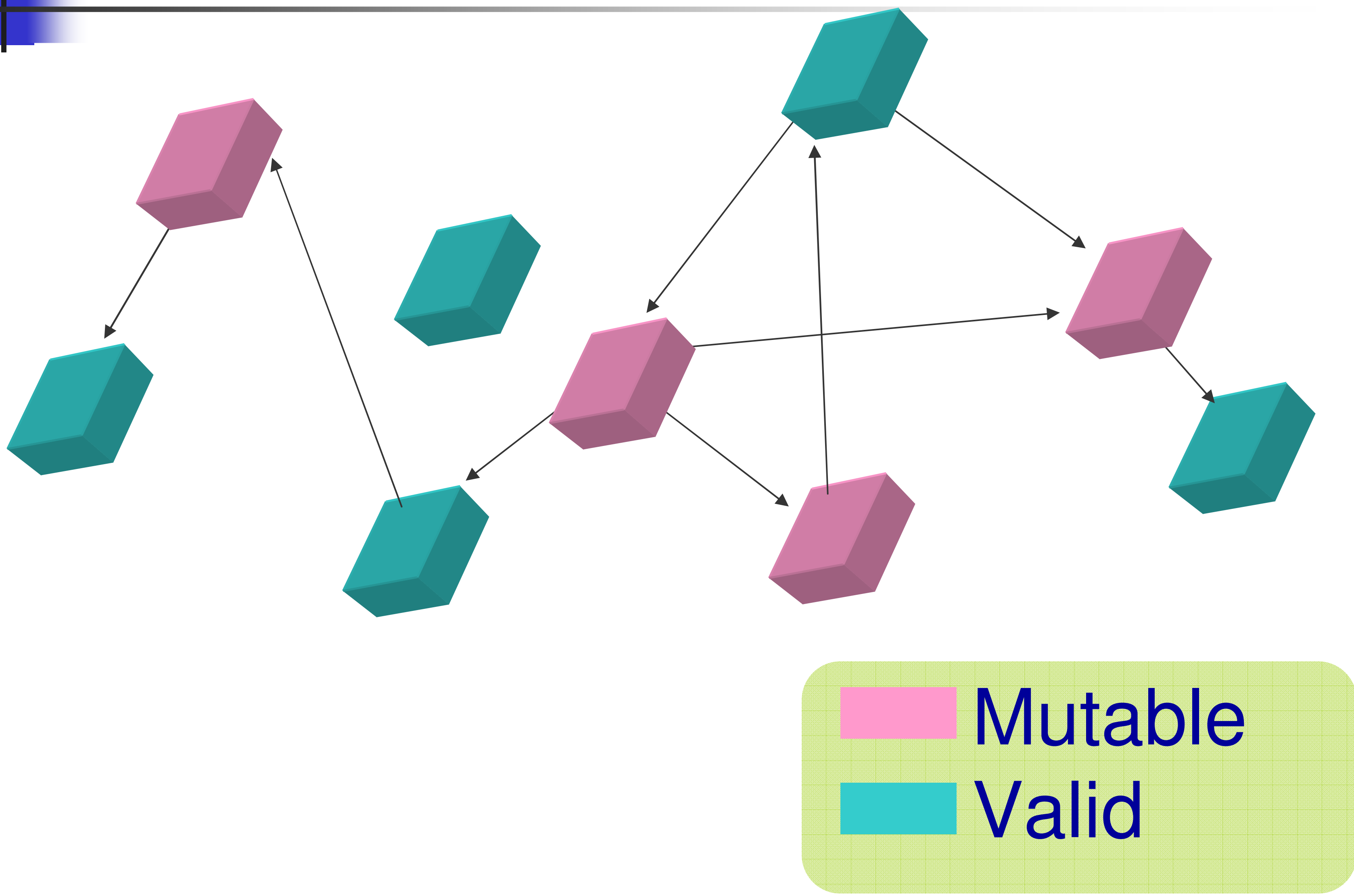
```
class Counter{
    int c;
    bool even;
    invariant 0 <= c;
    invariant even <==> c % 2 == 0;
    …

    public void Inc ()
    modifies c;
    ensures c == old(c)+1;
    {    c++; //invariant doesn't hold after c++;
         even = !even ;
    }
    …
```

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

The Heap (the Object Store)



Summary for simple objects

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

- invariant … this.f …;
- x.f = E;

Check:
x.mutable
or
assignment maintains
invariant

o.mutable ≡ ¬ o.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

Summary of Example

```
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

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 subclass
is mentioned in the
the object invariant
of the superclass

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

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;

    LuxuryCar()
    {
        cruiseControlSettings = -1;
    }
    [Additive] override public void SetSpeed(int kmph)
      //requires 0<= kmph; not allowed in an override
      ensures cruiseControlSettings == 50 && speed == cruiseControlSettings;
    {
        additive expose (this) {
            cruiseControlSettings = 50;
            speed = cruiseControlSettings;
        }
    }
}
```

Aggregates

Rich object structures need specification and verification support

simple invariants

aggregate objects

subclasses

additive invariants

visibility-based invariants

observer invariants

static class invariants

…

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

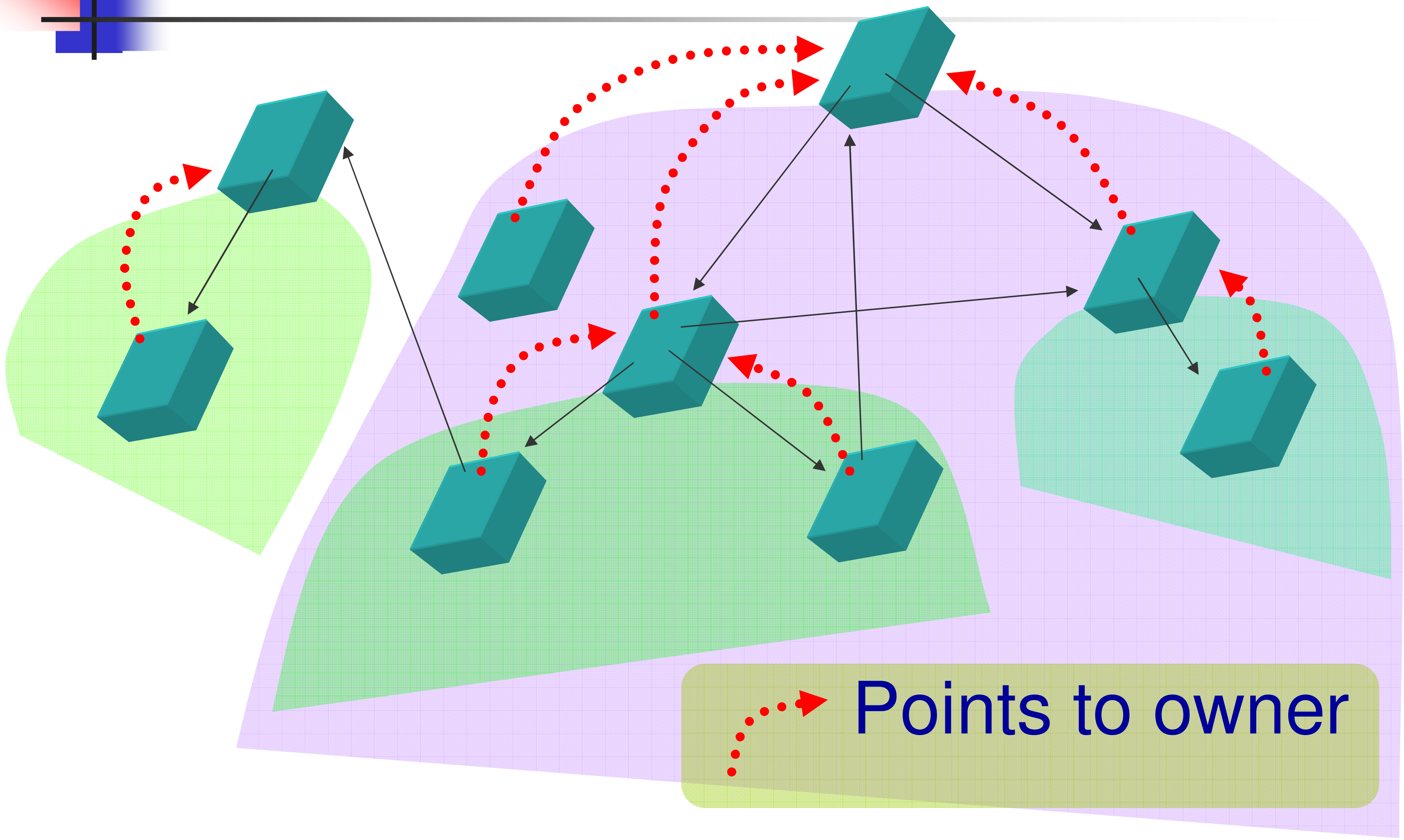
```
public class Car {
  int speed;
  invariant 0 <= speed;
  [Rep] Radio! r;

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

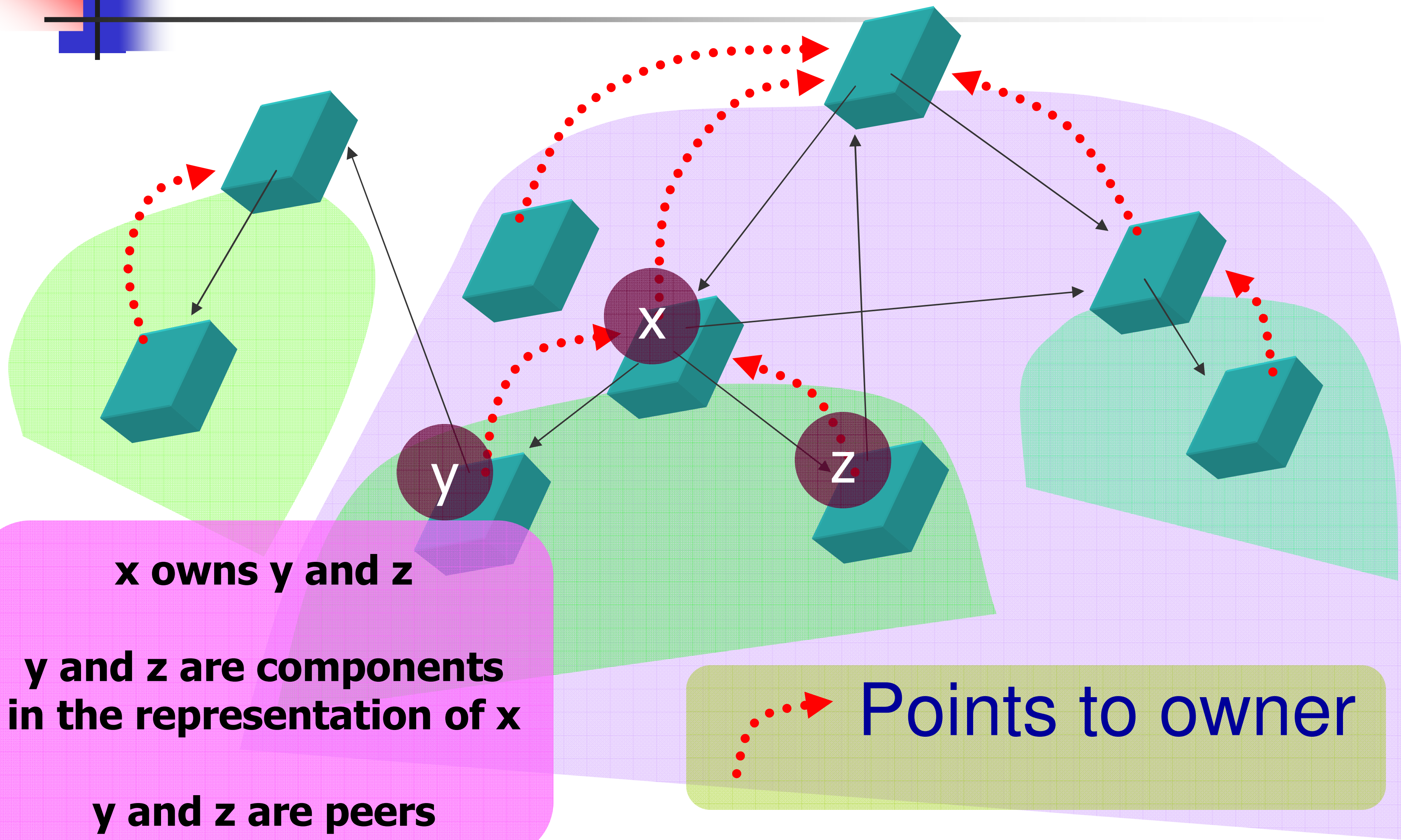
```
public void SetSpeed(int kmph)
    requires 0 <= kmph;
    modifies this.*;
  {
    expose (this) {
      speed = kmph;
      if (r.IsOn()) {
        r.soundBoosterSetting =
                            2 * kmph;
      }
    }
  }
```

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

Ownership domains



Ownership domains

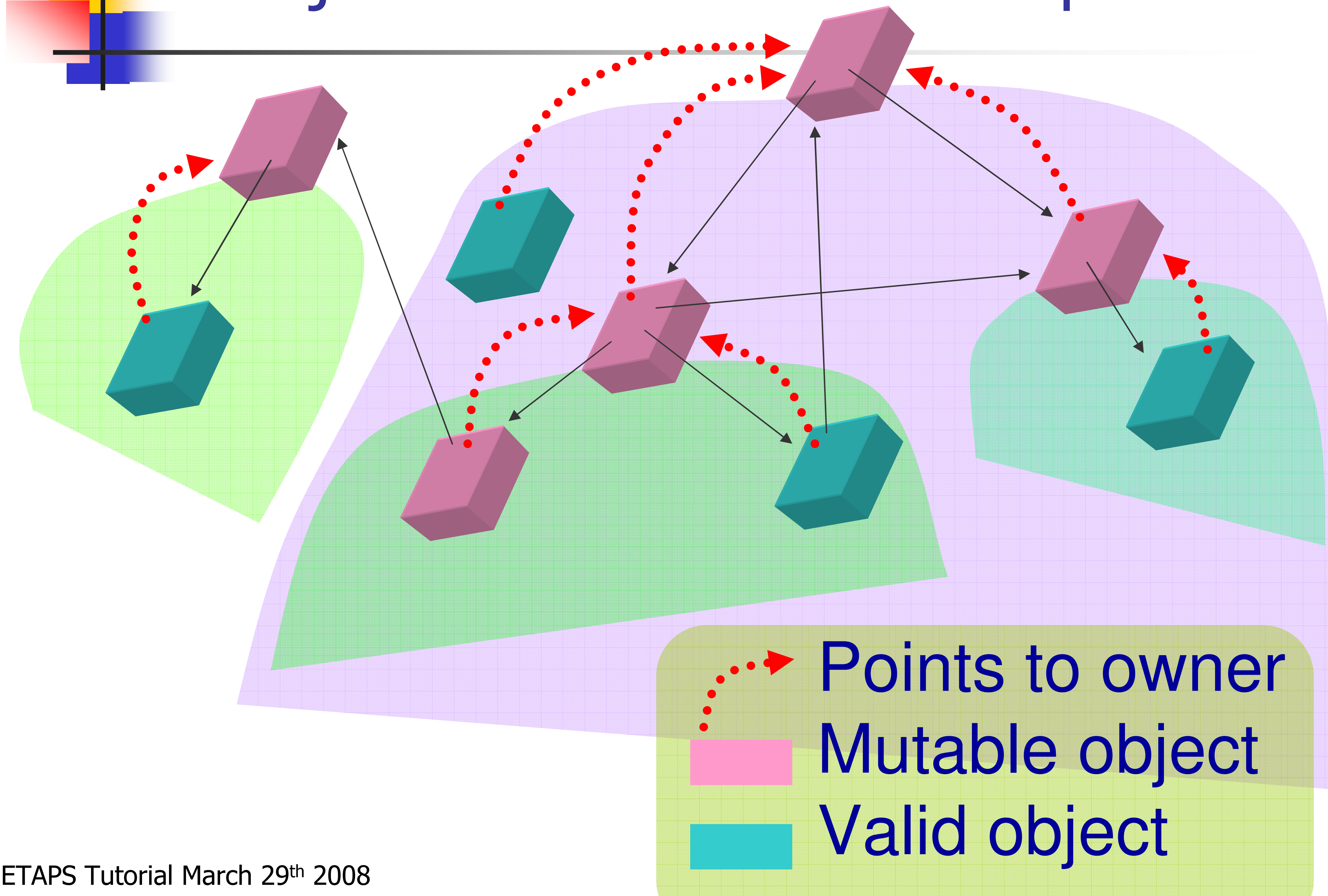


x owns y and z

y and z are components in the representation of x

y and z are peers

An object is as valid as its components



Visibility Based Invariants

```
public class Car {
  int speed;
  invariant 0 <= speed;
  [Peer] 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;
      }
    }
  }
```

Using [Peer] and expose together would give a visibility based error

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?

```
pu…                                   …t kmph)

invariant 0 <= speed;                   {
[Rep] Radio! r;                           expose (this) {
                                            speed = kmph;
public Car() {                              if (r.IsOn()) {
  speed = 0;                                  r.soundBoosterSetting =
  r = new Radio();                                          2 * kmph;
}                                           }
                                          }
                                        }
```

Making radio [Rep] makes Radio peer valid

Need the expose block to make it peer consistent.

Rep

t kmph)

pu

Why ever use Rep?

invari

[Rep]

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

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

```
if (r.IsOn()) {
  r.soundBoosterSetting =
                    2 * kmph;
}
```

Making radio [Rep] makes Radio peer valid

Need the expose block to make it peer consistent.

}

Representation (rep) fields

```
class Seat { public void Move(int pos)
        requires this.Consistent; … }
class Car {
  [Rep] Seat s;
  public void Adjust(Profile p)

     requires this.Consistent ∧ p.Consistent;

{
        expose (this) {
                s.Move(p.SeatPosition);
        }
}
```

o.Consistent ≡ o.owner.mutable ∧ o.valid

Peer fields and peer validity

```
class Seat { public void Move(int pos) requires this.PeerConsistent; … }
class Car {
  rep Seat s;
  public void Adjust(Profile p)
    requires this.PeerConsistent ∧
                          p.PeerConsistent;
    {
              expose (this) {
                 s.Move(p.SeatPosition);
              }
    }
```

```
peer Seat s;
public void Adjust(Position p)
    requires this.PeerConsistent ∧
             p.PeerConsistent;
    {
        s.Move(p.SeatPosition);
    }
```

$$o.Consistent \equiv o.owner.mutable \wedge o.valid$$

$$o.PeerConsistent \equiv o.owner.mutable \wedge (\forall p \bullet p.owner = o.owner \Rightarrow p.valid)$$

[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;
}
```

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

Modifies clauses

- In our example when the Radio r is annotated as rep, the method setSpeed does not need to specify modifies r.*
- This is a private implementation detail so the client doesn't need to see it
- Expert level!!! Option on switches – 1,5 and 6

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();
    }
  }
}
```

Pure Methods

Pure Methods

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

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

Error as we are not allowed to use the method IsOn in the specification as it may cause side effects.

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

Pure methods

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

  [Pure] public bool IsOn()
  {
    …
    return on;
  }
}
```

Using *Pure* Methods

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

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

- Declare the class attributes e.g.

```
[SpecPublic] int[]! a = new int[100];
```

- Specify and implement a method that uses the pure method

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

Expert comment …

- RecursionTermination
- ResultNotNewlyAllocated
- NoReferenceComparisons

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

  (see Verification of object-oriented programs with invariants. Mike Barnett, Rob DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. JOT 3(6), 2004 and Object invariants in dynamic contexts. K. Rustan M. Leino and Peter Müller. In ECOOP 2004, LNCS vol. 3086, Springer, 2004 and Class-local invariants. K. Rustan M. Leino and Angela Wallenburg, ISEC 2008. IEEE.)
- tools that enforce the methodology, ranging from easily usable dynamic checking to high-assurance automatic static verification

References and Resources

- Spec# website 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.
- Spec# wiki http://channel9.msdn.com/wiki/default.aspx/SpecSharp.HomePage
- Spec# examples http://www.cs.nuim.ie/~rosemary/