Spec#: Contracts and Specifications for C#
Topics

- What is Spec#
- Similarities with SPARK
- Dealing with an Object Oriented Language
- Dealing with Abstraction
Spec#

- Created by Microsoft
- Developed to explore specifications and tools for verifications
- Attempt at cost efficient method to develop and maintain software
- Consists of:
  - The Programming language
  - The compiler
  - The verifier: Boogie
Boogie

- Verifier for object-oriented languages
- Developed side by side with Spec#
- Design-time feedback
  - Specification violations detected immediately
Boogie

- Spec# compiler generates CIL
- Boogie converts CIL to BoogiePL
Method Contracts

- **SPARK**
  ```
  Total : Integer := 0
  
  procedure Add(X: in Integer)
   with Global => (In_Out Total),
   Pre => X > 0,
   Post => Total = Total'Old + X;
  ```

- **Spec#**
  ```
  int Total = 0;

  void Add(int X)
   modifies this.0,Total;
   requires X > 0;
   ensures Total = old(Total) + X;
  
  /* method body */
  ```

- **Pre ➔ requires**
- **Post ➔ ensures**
- **Spec# does not specify input or output**
Loop Invariants

- **SPARK**

```plaintext
procedure Divide(M, N : in Natural;
    Q, R : out Natural);
  is begin
    R := M;
    Q := 0;
    while R > N loop
      pragma Loop_Invariant (M = Q * N + R);
      Q := Q + 1;
      R := R - N;
    end loop;
  end Divide;
```

- **Spec#**

```plaintext
void Divide(int M, int N, out int Q, out int R)
{
    R = M;
    Q = 0;
    while( R > N )
    {
        invariant M = Q * N + R;
        Q = Q + 1;
        R = R - N;
    }
```
Loop Invariants

**SPARK**

```plaintext
def Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
begin
  for J in A'First .. A'Last loop
    if A (Result) < A (J) then
      Result := J;
    end if;
  pragma Loop_Invariant
    ((for all K in A'First .. J =>
      A(K) <= A(Result)) and
     (for all K in A'First .. Result - 1 =>
       A(K) < A(Result)));
  end loop;
return Result;
ext Max_Index;
```

**Spec#**

```plaintext
int Max_Index(int[] A)
{
  int Result = 0;
  for( int J = 0; J < A.Length; J++ )
    invariant forall{int i in (0 : J-1);
  {
      {
        Result = i;
      }
  }
return Result;
}
```
Inline Assertions

- SPARK
  - `pragma Assert X > 0`
  - `pragma Assert_And_Cut (X > 0)`

- Spec#
  - `assert X > 0;`
Variable Modifications

- **modifies** keyword specifies variable modifications
- **this.** - All members of the class and all members of its peers may be modifies
- **this.** – All members of the class may be modified
  - Compiler places **this.** at the end of every modifier clause, regardless if one is declared
  - **this.o** tells the compiler not to do this
Variable Modifications

```csharp
int Total = 0;

void Add(int X)
    modifies this.0,Total;
    requires X > 0;
    ensures Total = old(Total) + X;
{
    /* method body */
}
```
Variable Modifications

- Loops inherit the modifies clause of the containing method
- Can result in verifier being unable to determine certain conditions.

```csharp
void ContrivedModifications()
    requires 8 <= Dx;
    modifies X, Y;
{
    Y = 125;
    while (X < 27) {
        X += Dx;
    }
    assert 8 <= Dx;
    assert Y == 125; // error reported here
}
```
Object Oriented Verification

- We need to ensure objects are valid whenever they are used
- Need to ensure specification is not violated by changes to objects
- Need to take into account object ownership
Valid/Mutable State

- Objects are either in a:
  - Valid state
    - Referred to as Valid Objects
    - All the object’s invariants hold
    - The **components** of the object cannot be changed
  - Mutable state
    - Its invariants do no necessarily hold
    - Its **components** can be freely updated and altered
  - **expose** is used to put an object into a mutable state
Valid/Mutable State

- Valid objects with no owner or a mutable owner are called *consistent*
  - There are no possible restrictions on the object
- Valid objects with a valid owner are called *committed*
Object Invariants

- A class can have a set of invariants that must always hold true
- These invariants are checked to hold:
  - After every constructor
  - After every operation on any relevant variables

```csharp
public class Rectangle
{
    int width;
    int height;

    invariant width >= 0 && height >= 0;
    invariant (width == 0) <=> (height == 0);
}
```
Object Invariants

- Sometimes invariants must be temporarily violated

```csharp
void SetRectangleEmpty()
{
    width = 0;
    height = 0;
}
```

- `expose(x) { /*code*/ }`
  - When the object is in a mutable state, its invariants will not be checked until after the block exits.

```csharp
public class Rectangle
{
    int width;
    int height;

    invariant width >=0 && height >= 0;
    invariant (width == 0) <=> (height = 0);
}
```
Aggregate Objects

- How do we determine if a complex class has been modified?
- Aggregate Objects!
  - Objects that are defined by their member variables
  - Representing member variables are called *components*
  - Components are designated by the [Rep] tag

```csharp
class Band
{
    int gigs;
    [Rep] Guitar gt;
}
```
Aggregate Objects

- **Components** are owned by their containing class
  - Owner must be mutable in order for any operations to be put upon a component.

```csharp
class Band
{
    int gigs;
    [Rep] Guitar gt;

    public void Play()
    {
        expose (this)
        {
            gigs++;
            gt.Strum();
        }
    }
}
```
Peers

- Objects may be defined other objects but do not own them
- Peers
  - Defined by [Peer] tag
  - Can be modified by containing class
  - Is not necessarily only acted upon by containing class
  - Is owned by the owner of the containing class

```csharp
public class Iterator
{
    [Peer] internal Collection? c;
    int i = 0;
    invariant 0 <= i;
}
```
Peer Consistency

• In order to ensure the validity of all objects being modified or used:
  • Spec# adds a pre and post condition to every method
    • All in-coming parameters and their peers must be consistent
    • All in-coming parameters and their peers must not be bound by the invariants of their owner
  • This means that we can safely act on these objects without risking breaking any invariants
Virtual Methods

- Methods can be overridden in C# by subclasses
- How we deal with contracts of overridden functions?
- Overriding function inherit the contracts
  - They can still add to the contract
  - Contracts of the super class cannot be removed
Virtual Methods

```csharp
public class Subtracter {
    public override int Subtract(int first, int second)
        modifies this.0;
    ensured result = first - second;
    { return first - second; }
}

public class NonNegativeSubtracter : Subtracter {
    public override int Subtract(int first, int second)
        requires first >= 0 && second >= 0;
    { return Base.Subtract(first, second); }
}

public class ProperSubtracter : NonNegativeSubtracter {
    public override int Subtract(int first, int second)
        requires first >= second;
    { return Base.Subtract(first, second); }
}
```
Pure Spec# Functions

- Pure functions do not modify the state of the program in any way.
- Designated by the [Pure] tag
- Property getters are implicitly pure

```csharp
private int value;

[Pure]
public int GetValue() { return value; }

public int Value
{
    get { return value; } // This is implicitly [Pure]}
```
Conclusion

- Spec# offers many features SPARK does
- Spec# handles object orientation
- Spec# handles inheritance
Questions?
References


