

The Spec# Programming System: An Overview

Mike Barnett, L. Rustan, M. Leino, and Wolfram Schulte

Presented by
Sherrie Campbell



What is Spec#?

- Extension (superset) of C# in the .NET environment
- Adds the capability to add specifications
 - Pre-conditions
 - Post-conditions
 - Non-null types
 - Abstraction



What are uses of Spec#?

- Aid in designing and creating software that is
 - correct
 - maintainable
- Captures programmers assumptions
 - Assumptions captured into the code
 - So they are not lost
 - Assumptions can be checked to assure that assumptions are met now and in the future

For Example

```
public class Program
{
    static bool Test(int[]! array)
    forall {int i in (0 : array.Length);array[i]>0};
    {
        return true;
    }

    static void Main(string![]! args)
    {
        Test(new int[]!{1});
    }
}
```

From Spec# Mailing List Digest



Static Checking

- Static type checking.
 - Done by the compiler, extending C# type checking
 - non-null types
 - exception throws clauses
 - To compile a program the static check must be clean.
(For compatibility with C# static checker can just give warnings)

From Spec# Mailing List Digest



Dynamic checking

- The compiler produces run-time checks for
 - pre- and postconditions
 - Object invariants – which are checked at the end of constructors and at the end of expose blocks.
- Checks are done every time code runs but compiler can be instructed to omit some checks

From Spec# Mailing List Digest

Static program verification

- This part is optional, program verifier can be used
 - at design time
 - compile time
 - from the command line
- “The static program verifier attempts to mathematically verify the correctness of your program, checking that the program lives up to the preconditions of various operations. These checks include the correctness conditions prescribed by the language (e.g., that array indices are within bounds and type casts succeed), user-supplied correctness conditions (e.g., pre- and postconditions, including preconditions specified in libraries that your program calls), and correctness conditions that come from the Spec# programming methodology (e.g., that an object is exposed at the time its fields are modified)”

From Spec# Mailing List Digest



Spec#

- Small extension of popular language (C#)
- Enables specifying and reasoning about object invariants
- Set of Tools that enforce the methodology for both static and dynamic checking
- Programmers can gradually start adopting

Non-Null Types

- Ensures that variables that must not be null have values.
- Identified by a “!” After the datatype.
 - `int! a;`
 - `String! NucControlLocation;`
- Protects against accessing variables before they are initialized.

Non-Null Types

C# - way of initializing class variables

```
Class Student : Person {  
    Transcript! t;  
    public Student (string name, EnrollmentInfo! Ei) : base (name) {  
        t = new Transcript(ei);  
    }  
}
```

Spec# way of initializing to ensure null data cannot be referenced

Note: Important to Initialize non-null values before calling base constructor

```
Class Student : Person {  
    Transcript! t;  
    public Student (string name, EnrollmentInfo! Ei) :  
        t = new Transcript(ei), base (name) {  
    }  
}
```



Method Contracts

- Specify conditions that must be true for the method to execute
 - Exceptions
 - Preconditions
 - Postconditions
 - Exceptional Postconditions
 - Frame Conditions
 - Inheritance of specifications



Method Contracts - Exceptions

- Checked Exceptions
 - Admissible Failures – when method is unable to complete for reasons outside of its boundaries such as network errors or socket problems
 - Signaled using an `IOException`
- Unchecked Exceptions
 - Program Errors
 - Client Failures – conditions for getting to the method are not met



Method Contracts - Preconditions

```
class ArrayList {  
    void Insert (int index, object value)  
        requires  $0 \leq \textit{index} \ \&\& \ \textit{index} \leq \textit{Count}$   
            otherwise ArgumentOutOfRangeException;  
        requires !IsReadOnly && !IsFixedSize  
            otherwise NotSupportedException;  
    { ... }
```

The otherwise clauses defines what is executed on Contract Condition errors these are unchecked exceptions

Method Contracts - Postconditions

- Post conditions can be checked programmatically by compiler so that dynamic checking during runtime will not be necessary.
- Postconditions for Insert
 - ensures** *Count* == **old** (*Count*) + 1;
 - ensures** *value* == **this** [*index*];
 - ensures** *Forall*{**int** *i* in 0 : *index*; **old** (**this**[*i*] == **this** [*i*] } ;
 - ensures** *Forall*{**int** *i* in *index*: **old** (*count*); **this**[*i*] == **this** [*i*+1] } ;

Method Contracts – Exceptional Postconditions

- Methods must declare which exceptions they will throw and they may only throw checked exceptions
- Can then add a post condition that runs only if the exception is thrown.
- **void** *ReadToken* (*ArrayList a*)
 throws *EndOfFileException* **ensures** *a.Count == old(a.Count)*;
 { ... }

Method Contracts – Frame conditions

- Specify explicitly what can be changed during a method
`class C {`
 `int x, y;`
 `void M() modifies x; { . . . }`
- Wild cards may be used to indentify if internal class variables are going to change
- Frame conditions check with the verifier but are not checked at run time due to run time costs.



Method Contracts - Inheritance

- A method inherits the checks of the methods it overrides
- Additional postconditions can be added
- Can add additional exceptional postconditions only for exceptions already in the throws set
- Multiple inheritance
 - When an interface inherits from a class and the class from a superclass.
 - Specifications are combined
 - Combines exceptional postconditions but throws sets of all the inherited specifications must be the same

Class Contracts

- Object invariants – specifications for the steady state of an object

```
class AttendanceRecord {
```

```
    Student[]! Students;
```

```
    int []! absent;
```

```
    invariant students.Length == absent.Length ;
```

- To change the variable and not violate the invariant you use expose

```
    expose (o) {
```

```
        S;
```

```
    }
```



Architecture

- Components
 - Compiler
 - Keeps specifications with the compiled code
 - Runtime library
 - Checks are added via inline code for methods
 - New methods for classes with invariants
 - Boogie verifier
 - Static Verification done by iterating through code and producing an intermediate file that is run through a theorem prover



In Use Today

- In the Spec# Mailing List 6-10 clarifications are posted each month on how to implement portions of spec#
- Created as part of a research group at Microsoft so it cannot be used for for-profit programs
- <http://research.microsoft.com/specsharp/>
Spec# home page has many publications including benchmarks to compare it with other verifiers and theorem provers