

The Spec# Programming System

Tomáš Matoušek

<http://tm.matfyz.cz>

CHARLES UNIVERSITY IN PRAGUE
Faculty of Mathematics and Physics



Outline

- Spec# System Architecture
- Spec# Programming Language
- Boogie static verifier
- Conclusion



System Architecture

- Spec# language
 - C# extended with specifications
 - predecessors: AsmL, JML (Java Modeling Language), Eiffel
 - compiled into assemblies (IL + metadata)
 - specifications serialized to language independent metadata
- assemblies consumed by correctness tools
 - Boogie – static verification by theorem proving
 - SpecExplorer – model-based testing
- add-ons
 - Spec# Class Library
 - specified Base Class Library (BCL)
 - VS.NET integration module



Out-of-band Specifications

- specifications for a code not written in Spec#
- compiled to Spec# repository
 - consulted if a missing specification is required
- specifications repository for BCL
 - Spec# companion project
 - semi-automatic extraction



Spec# Language

- extends C# with
 - non-null types
 - checked exceptions
 - method contracts (pre-, postconditions)
 - object and class contracts (invariants)
 - ...more experimental features ...
- backward compatibility
 - Spec# without specifications (almost) equivalent to C#
- interoperability with existing .NET classes
 - spec-aware method can use spec-unaware ones and vice-versa
 - but soundness not guaranteed then



Specifications in General

- specification rules enforced
 - statically by the compiler or Boogie
 - dynamically by injected code
 - depends on a particular rule
- expressions defining contracts
 - should be pure (i.e. have no side effects)
 - public contracts should not refer to private members
 - enforced by the compiler



Non-null Reference Types

- reference types
 - possibly null (no syntax change)
 - non-null (an exclamation mark is added)
- a problem with partially constructed objects
 - non-nullity rule violated before field assignment in constructor

```
class Student : Person
{
    private School! school;

    public Student(string! name, School! school)
    : base(name)
    {
        /* the school is null here */
        this.school = school;
    }
}
```



Solution: Initializers

- retrofits C#
- initializers can be called before the base constructor
- initializers required for every non-null field
 - a field have to be initialized before it can be read
- corrected example:

```
class Student : Person
{
    private School! school;

    public Student(string! name, School! school)
    : this.school = school, base(name)
    {

    }
}
```

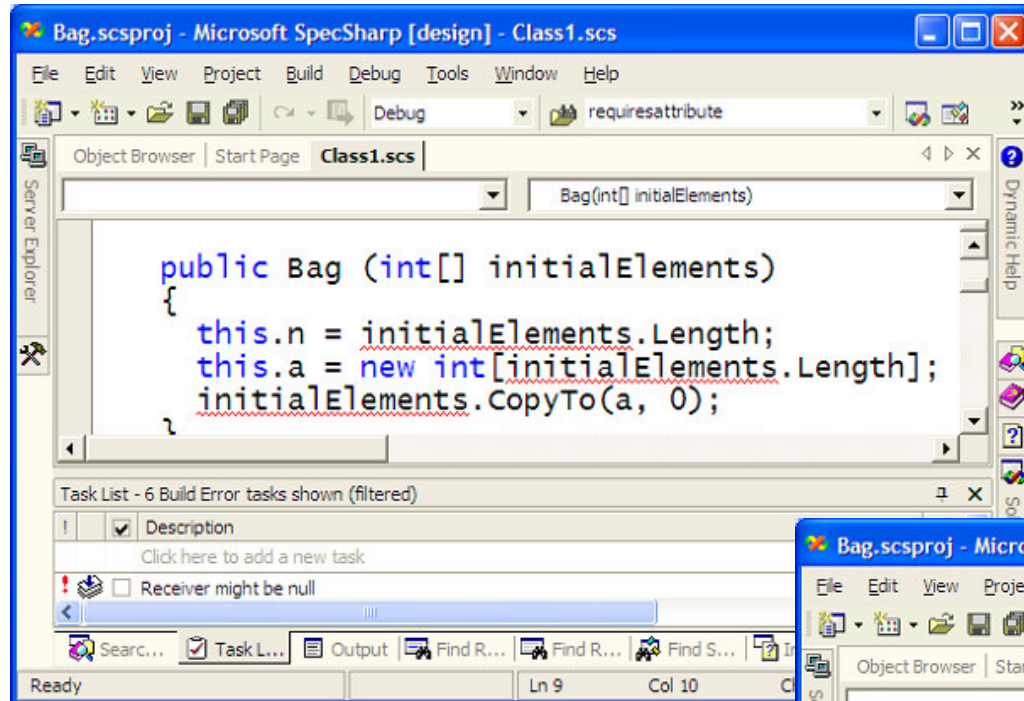


Non-null Type Usage

- non-null type can be used for
 - instance fields
 - formal arguments
 - return values
 - local variables
- disallowed for
 - static fields
 - to avoid problems with type loading
 - array elements
 - initialization problems

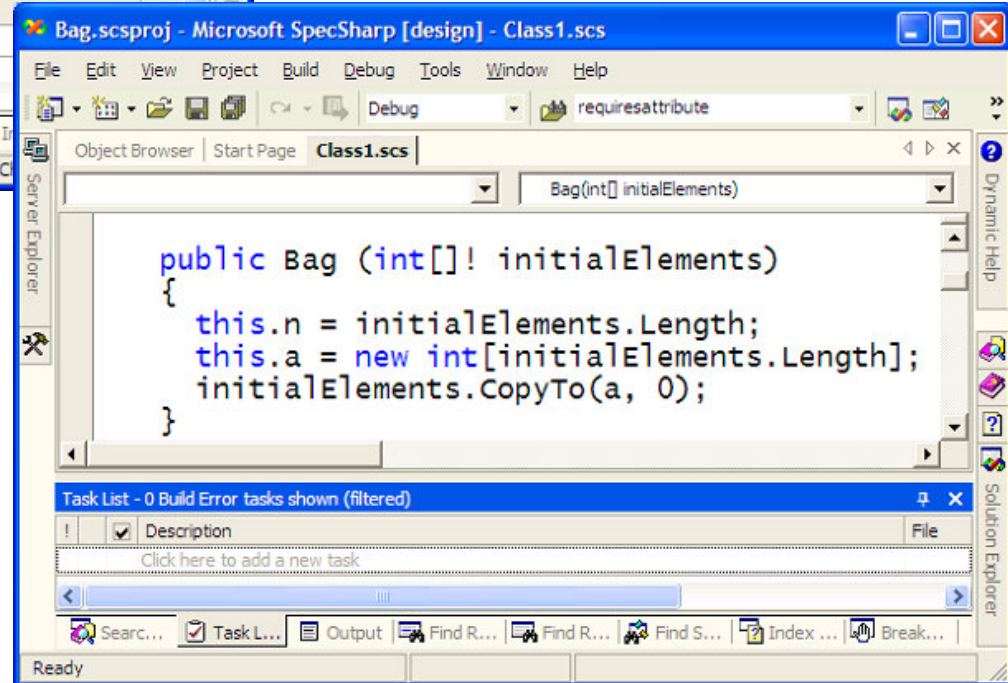


Demo: Checking Non-null Types



incorrect

correct



Checked Exceptions

- Java
 - each exception is checked except for subclasses of
 - *Error* class
 - *RuntimeException* class
- C#
 - no checked exceptions in C#
 - no method throw sets
- Spec#
 - to be compatible with C#, an exception is unchecked by default
 - checked exceptions implement *ICheckedException* interface



Checked Exceptions (cont.)

- each thrown checked exception have to be in a throw set
 - guaranteed by Spec# compiler
 - a problem may arise

```
public void Cheater()  
{  
    Exception e = new CheckedException();  
    throw e;  
}
```

- if the compiler can't determine a real type of thrown object
 - a run-time check is injected

```
public void Cheater()  
{  
    Exception e = new CheckedException();  
    if (!(e is ICheckedException)) throw e; else { /* some error */ }  
}
```



Exception Categories

- domain failures
 - a method is called under false preconditions
 - unchecked
- range failures
 - observable program errors
 - e.g. an array index out of bounds
 - unchecked
 - admissible failures
 - e.g. a file not exists
 - checked – a part of a method contract



Method Contracts

- preconditions
 - define conditions under which a method is called
- postconditions
 - define conditions valid when a method returns
- exceptional postconditions
 - define conditions valid when a method throws a checked exception



Preconditions

```
class ArrayList
{
    public int Count { get { ... } }
    public bool IsReadOnly { get { ... } }
    public bool IsFixedSize { get { ... } }

    public virtual void Insert(int index, object item)
        requires 0 <= index && index <= Count otherwise ArgumentException;
        requires !IsReadOnly && !IsFixedSize;
    { ... }
}
```

- syntax
 - (**requires** *condition* [**otherwise** *exception*];)*
 - *condition* can contain parameters and public members only
 - *exception* defaults to *RequiresException*
 - unchecked (a domain failure)



Postconditions

```
class ArrayList
{
    public int Count { get { ... } }

    public virtual void Insert(int index, object item)
        ensures Count == old(Count) + 1;
        ensures item == this[index];
        ensures Forall{int i in 0:index; old(this[i]) == this[i] };
        ensures Forall{int i in index:old(Count); old(this[i]) == this[i+1]};
    { ... }
}
```

- throws *EnsureException* if a run-time check fails
- special constructs:
 - *Forall* quantifier
 - intervals (lower-bound-included : upper-bound-excluded)
 - *old(x)* denotes a value of *x* on entry to the method
 - at run-time the expression *x* is evaluated on the method entry and the result is stored into a local variable



Exceptional Postconditions

- contracts added to throw sets
- define a state of an object when the exception is thrown

```
void ReadToken(ArrayList! a)
  throws EndOfFileException ensures a.Count == old(a.Count);
{ ... }
```



Invariants

- constraint field values
- instance (object contracts) or static (class contracts)

```
class C
{
    int[]! a, b;
    invariant a.Length == b.Length; // object invariant
}
```

- problem:
 - no atomic operation creating both arrays at once
- solution:
 - invariants holds in *steady* states
 - invariants temporary broken when an object is *exposed*



Exposing Object

- a block statement defining exposure

```
class C
{
    int[]! a, b;
    invariant a.Length == b.Length;

    void SetLength(int length)
        requires length > 0;
    {
        ...
        expose(this) { a = new int[length]; b = new int[length]; }
        ...
    }
}
```

- implicit exposure
 - an object is exposed through construction
 - all public methods (can be disabled by a custom attribute)



Exposure and Inheritance

- class frame
 - a set of members and invariants declared directly in the class

`expose (obj)`

- exposes the class frame of the static type of *obj* if it has not been exposed yet

`expose (obj upto T)`

- exposes class frames which has not been exposed yet starting with the static type of *obj* up to the class *T* including

- at least one frame has to be exposed by the operation
 - otherwise an exception is thrown



Fields Referred by Invariants

- simple approach
 - only fields of declaring class or its superclass

```
class C
{
    int[]! a, b;
    invariant a.Length == b.Length;
}
```

- more complex approach
 - fields of an arbitrary reachable object

```
class List
{
    Item! head;
    invariant head.Value == null && "the list is linked properly";
}
```



Checking Simple Invariants

- a special field *inv*
 - one for each object
 - contains the most-derived class which invariant holds
 - invariants of superclasses hold as well
 - invariants of subclasses can be broken
- let T be the static type of obj and S be the immediate superclass of T then

```
expose(obj) { /* code */ }
```

is

```
assert obj!=null && obj.inv == T;  
obj.inv = S;  
/* code */  
assert obj!=null && obj.inv == S;  
assert InvariantT(obj)  
obj.inv = T;
```

} unpack obj from T

} pack obj to T



Custom Attributes on Contracts

- each contract clause can be annotated by custom attrs.
- can be also used by third party tools

```
int BinarySearch(object[]! a, object item, int low, int high)
    requires 0 <= low && low <= high && high <= a.Length;
    [Conditional("DEBUG")]
    requires IsSorted(a);
{ ... }
```

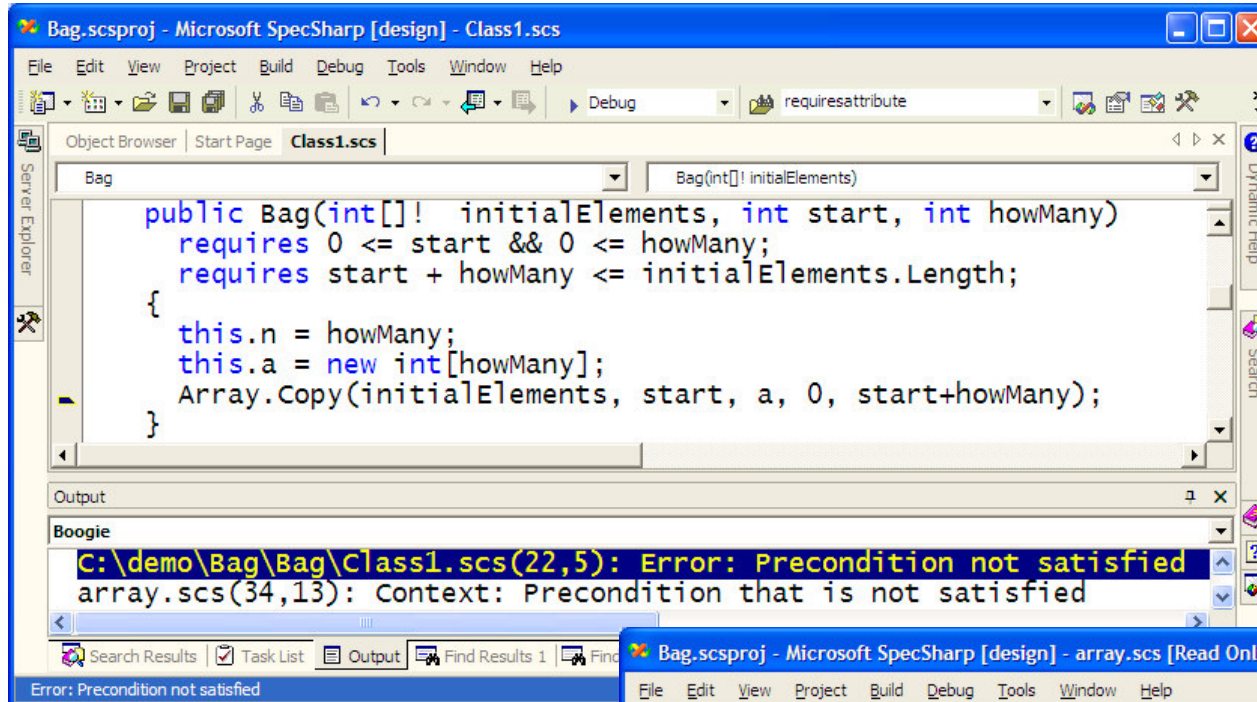


Boogie

- a static verifier of contracts
- based on theorem proving
 - translation: IL and metadata to BoogiePL
 - inference engine working on BoogiePL
 - theorem prover
 - current version: the Simplify theorem prover
 - future: the Zapato theorem prover
 - results mapped back to source program
 - user can interact with prover (add contracts etc.)



Boogie Demo



Microsoft SpecSharp [design] - Class1.scs

```
public Bag(int[]! initialElements, int start, int howMany)
  requires 0 <= start && 0 <= howMany;
  requires start + howMany <= initialElements.Length;
{
  this.n = howMany;
  this.a = new int[howMany];
  Array.Copy(initialElements, start, a, 0, start+howMany);
}
```

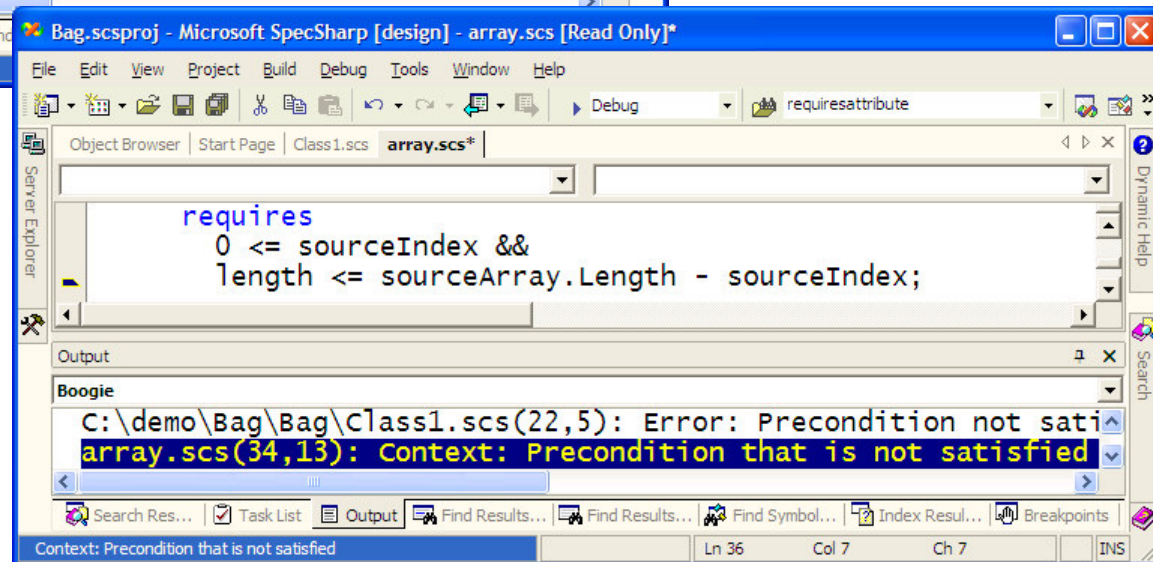
Output

Boogie

```
C:\demo\Bag\Bag\Class1.scs(22,5): Error: Precondition not satisfied
array.scs(34,13): Context: Precondition that is not satisfied
```

Search Results | Task List | Output | Find Results 1 | Find Results 2

Error: Precondition not satisfied



Microsoft SpecSharp [design] - array.scs [Read Only]*

```
requires
  0 <= sourceIndex &&
  length <= sourceArray.Length - sourceIndex;
```

Output

Boogie

```
C:\demo\Bag\Bag\Class1.scs(22,5): Error: Precondition not sati
array.scs(34,13): Context: Precondition that is not satisfied
```

Search Res... | Task List | Output | Find Results... | Find Results... | Find Symbol... | Index Resul... | Breakpoints

Context: Precondition that is not satisfied

Ln 36 Col 7 Ch 7 INS

Conclusion

- seems to be working
 - but not released yet (maybe at the end of this year)
- writing specifications is simple
 - simple rules
 - integration to VS.NET
 - can be adopted even by non-experts
- no additional effort to write basic preconditions
 - anyway checks are written in the code nowadays
 - non-null types make a lot of work themselves



References

- M. Barnett, K. R. M. Leino, and W. Schulte.
The Spec# Programming System: An Overview.
Microsoft Research, May 2004.
- K. R. M. Leino, W. Schulte.
Exception safety for C#.
Microsoft Research, April 2004.
- M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, W. Schulte.
Verification of object-oriented programs with invariants,
in *Journal of Object Technology*, vol. 0, no. 0, 2004, pages 1–30.
- K. R. M. Leino, Peter Müller
Object Invariants in Dynamic Contexts,
in *ECOOP 2004*, June 2004.

