K. Rustan M. Leino, Peter Müller

Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs

Presented by Paolo Antonucci

04/05/13

About this paper

This paper is more similar to a practical tutorial than a research paper.

“*It is specifically not a goal of this tutorial to justify the Spec# methodology, only to explain how it is used.*”

Focus on the language, in the status it was when the paper was published.

What is Spec#?

Spec# is a research language that extends C# 2.0 with new constructs and features.

In particular it features program specifications that are enforced both statically and dynamically.

A verifier runs at compile time and attempts to prove the program correct.

Verification is modular.

The Spec# methodology

Spec# imposes a methodology (programming discipline). Following this will lead to well specified and easily verifiable programs.

Unfortunately it is very easy to fall outside the boundaries.

It is much easier to verify programs if they are designed according to this methodology from the start.

Outline

- Introduction
- Quick tour into Spec#
- Demo
- Final discussion

We are here!

Class invariants

```
public class Exam {

    private int ExGrade = 100;
    private DateTime ExDate;

    invariant ValidGrade(ExGrade);

    [Pure] static bool ValidGrade(int grade)
    ensures result == ((grade % 25 == 0)
        && 100 <= grade && grade <= 600);
    {
        return (grade % 25 == 0)
            && 100 <= grade && grade <= 600;
    }

}
```

Class invariant: must always hold when the object is consistent

Pure function

Pure functions promise to return values with no side effect. This makes it possible to use them in program specifications.

Class invariants

Sometimes it is necessary to temporarily break an object invariant.

This is done by “exposing” the object with the expose construct.

```
    expose (this) {
          // Break the invariant
          // Restore the invariant
    }
```

The verifier will attempt to prove that at the end of the expose block the invariant is restored.

Method contracts

```
public void setGrade(int grade)
requires ValidGrade(grade);
ensures ExGrade == grade;
modifies this.ExGrade;
{
    ExGrade = grade;
}
```

Methods are only allowed to modify fields declared here

Contracts are checked both statically and dynamically.

Methods are also checked to maintain class invariants.

The `modifies` clause can be omitted: in this case it is interpreted as `modifies this.*`.

Redefined methods inherit contracts from the superclass.

Assertions and assumptions

```
public void setGrade(int grade)
requires ValidGrade(grade);
ensures ExGrade == grade;
modifies this.ExGrade;
{
    ExGrade = grade;
    assert 400 <= ExGrade;
}
```

Inline assertions are checked by the program verifier. They are redundant in principle.

```
public void setGrade(int grade)
requires ValidGrade(grade);
ensures ExGrade == grade;
modifies this.ExGrade;
{
    ExGrade = grade;
    assume 400 <= ExGrade;
}
```

Assumptions are taken on faith by the program verifier. They trade static checking for flexibility.

<u>BOTH</u> are checked at runtime.

Non-null types

All variables of non-primitive types can be declared either as non-null or as possibly null.

An exclamation mark (bang!) declares a non-null type.
A question mark (uh?) declares a possibly-null type.

Dereferencing a possibly-null pointer is only allowed if static dataflow analysis can prove that it cannot be null at that point.

```
public static void pippo(string! foo, string? bar) {
     Console.WriteLine(foo.Length);
     if (bar == null)
          return;
     Console.WriteLine(bar.Length);
}
```

Demo

DEMO

Conclusion – how it feels

Verification warnings feel like a natural extension of classic compiler warnings.

The verifier seems to be reasonably “smart”.

Nevertheless, Spec# is indeed complex. Not suitable for the average programmer.

This complexity feels usually elegant, but sometimes some innocuous features can be surprisingly awkward.

Conclusion – current status

Undoubtedly still a research language.

- Almost no documentation
- Development tools somewhat buggy
- Sometimes unexpected behavior
  (e.g. cannot prove anything about doubles)

In the future some features could make it to C#. ☺

- This was already the case for Code Contracts

Sadly, development doesn't seem to be active. ☹

References

- Spec# home page
  http://research.microsoft.com/en-us/projects/specsharp/

- Spec# CodePlex repository
  http://specsharp.codeplex.com/

- Try Spec# online
  http://rise4fun.com/SpecSharp

- Many research papers related to Spec#, more references in the Spec# home page and in this paper

Question time

Questions?

Bonus slide
Object ownership – recall

Spec# supports object topology/ownership.

An object can, for example, own other objects used for the internal representation of its data.

This is encoded with the `[Rep]` attribute in the code.

```
public class StudentCurriculum {

    [Rep] Thesis? MasterThesis;

}
```

Object owners can also be set manually and specified in contracts.

Bonus slide
Object ownership – recall

As an informal rule, an object can only be modified with the “permission” of its owner.

This makes it possible for object invariants to rely on [Rep] owned objects.

Bonus slide
Object ownership and invariants – recall

Sometimes it is necessarily to temporarily break an object invariant.

This is done by “exposing” the object with the expose construct.

```
expose (this) {
     // Break the invariant
     // Restore the invariant
}
```

Bonus slide
Object ownership and invariants – recall

As object invariants can refer to owned [Rep] objects, any change to an object can potentially break the invariant of its owner.

For this reason, non-pure methods can only be called on an object after exposing its owner.

```
public void setThesisTitle(string title) {
     expose (this) { // This is necessary!
          MasterThesis.setTitle(title);
     }
}
```

This was just a quick recall, there is much more about this.