

# Status Report for Week 1

Phillip Casey & Greg Glazer  
06 April 2009

## Accomplishments for the Week

- Downloaded and Installed Visual Studio
- Downloaded and Installed Spec#
- Began some initial research to try and answer the questions:
  - What is Spec#?
  - What is Boogie?
  - How does it relate to Program Verification?

## Goals for Next Week

- Create an outline of the Spec# components we want to research and understand
- Create a presentation outline
- Begin to work on Spec# examples/samples

## Additions to Annotated Bibliography

**Barnett, Mike, and DeLine, Robert, F"ahndrich, and Leino, Rustan, and Schulte, Wolfram. Verification of object-oriented programs with invariants. JOT 3(6), 2004, 27-56.**

Defines a programming methodology for using object invariants. Central to the methodology, is that an object publicly express the invariant state while, at the same time, hiding the implementation and specification details of the state conditions. The paper starts with a basic idea and expands it, through several examples, to a full specification. Finally, several examples are explored and discussed in detail. The interesting part of this methodology is that it allows modular verification of *interesting* object oriented programs. The methodology is a central part of Boogie.

**Leino, Rustan and Monahan, Rosemary. Program Verification Using the Spec# Programming System. Available from <http://research.microsoft.com/en-us/projects/specsharp/etaps-specsharp-tutorial.ppt>; accessed 3 April 2009.**

Provides a very high level overview of the Spec# Language. Covers concepts at a high level, intended for demonstrative and introductory purposes: non-null types, assert statements, assume statements, design by contract, static verification, invariants, aggregates, and pure methods. Contains several examples, (Insertion Sort and Greatest Common Divisor).

**Ouimet, Martin. Formal Software Verification: Model Checking and Theorem Proving. Available from <http://esl.mit.edu/publications/ESL-TIK-00214.pdf>; accessed 25 March 2009.**

Provides an introduction to Model Checking and Theorem Proving from the context of building more reliable software. Discusses both model checking and theorem proving; stating that model checking is suited for fairly finite state structured programs and that theorem proving is more suited for more complex types; while, at the same time, discussing the complimentary aspects of the two. Goes onto discuss the human usage side of such tools and the need for ease of use and the learning curve involved.

## **Notes from the Week**

### ***New Learning's***

#### **Boogie**

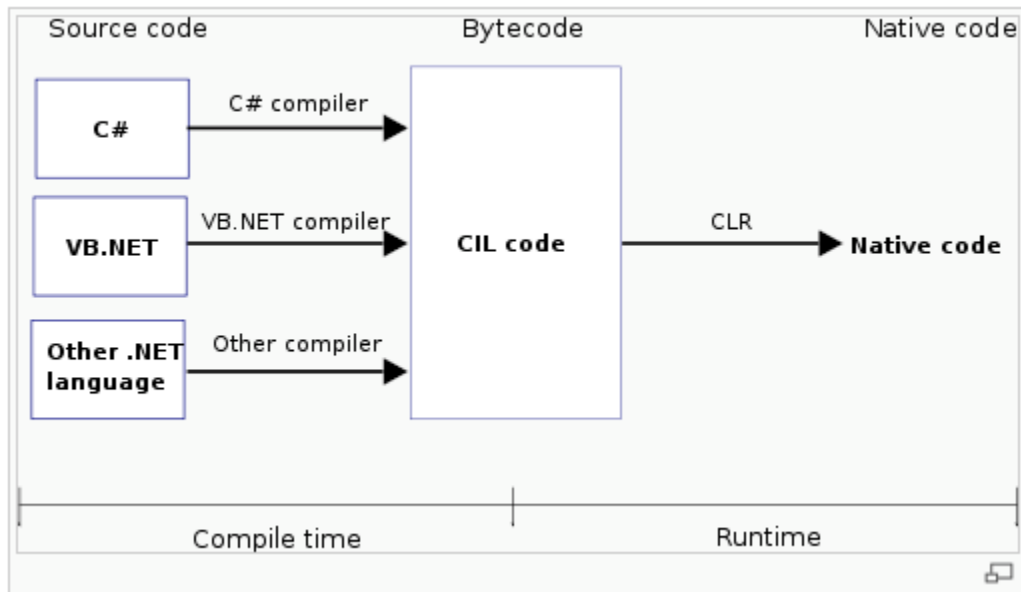
Boogie is an intermediate verification language and is intended as an architectural layer upon which to build program verifiers for specific languages. Boogie is also the name of a tool which accepts, as input, the Boogie language and then generates verification conditions that are passed to an SMT solver

Boogie and Spec# were developed/created hand-in-hand and, because of this, the name Boogie is sometimes used to describe Spec# related items.

In Spec# terms, Boogie is the Spec# static program verifier. Boogie translates compiled Spec# programs (.NET bytecode) into Boogie; the Boogie tool accepts compiled Spec# programs (extensions .dll and .exe). Additionally, although not specific to Spec#, Boogie accepts Boogie programs (extension .bpl) as input.

Spec# uses the Boogie methodology, which is an “ownership-based discipline for handling object invariants”. In laymen’s terms it appears that this means that object owns its invariants. This methodology is described by the Verification of Object-Oriented Programs with Invariants paper.

## .NET Bytecode



(citation: diagram comes from Wikipedia)

## Intermediate Verification Language

Define

### SMT Solver

Satisfiability Modulo Theories. SMT instance is a first-order-logic formula and SMT is the problem of determining whether such a formula is satisfiable. Z3 is the default Boogie SMT Solver.

### Z3

Automated satisfiability checker with many built in theories (citation: <http://research.microsoft.com/en-us/um/redmond/projects/z3/intro.html>)

Z3 can produce full models and is guaranteed to return a correct answer for programs containing only existential formulas. For programs with the universal quantifier, then Z3 produces a potential model which is not guaranteed to be complete.

Timeline:

- 25 June 2007. Z3 0.1 submitted to SMT – COMP 07
- 13 September 2007. Z3 1.0 released
- 22 October 2007. Z3 1.1 released
- 7 December 2007. Z3 1.2 released
- 25 June 2008. Z3 1.3 released

Those funny guys! Historical connection: Konrad Zuse was the creator. Z3 was the world's first working, fully programmable computing machine (not electronic).

## Object Invariant

An invariant used to constrain objects (classes). Class Methods should preserve the invariant. The invariant constrains the state stored within the context of the object. Think of it as a set of invariant properties that are not compromised during the lifecycle of the object (throughout all permutations of the state of the object).

An object invariant defines what it means for an object's data to be in a consistent state. The correctness and verifiability of object oriented programs is performed through these object invariants.

## Notes from Journal of Object Technology

Analogy of a type checker: ensure that variables don't take on a forbidden value. Detail management techniques leave the enforcement details to a mechanical tool.

Popular (flawed) view: an object's invariant is simply a shorthand for a post-condition on every constructor and a pre and post condition on every method. Faulty premise: an object's invariant should hold whenever the object is publicly visible. This approach allows an object's invariant to be violated during a call, so long as it is re-established upon exit. Solution: express invariants explicitly with pre-and post conditions. Problem introduced here is that an object's invariants are a condition of the internal representation of the object; the details should be no concern to the client, but the client needs to ensure the object meets the pre-condition.

Goal: allow client to be aware if invariant holds without exposing the implementation details of the invariant. To achieve this: introduce a public view of whether an invariant holds or not; "hiding" the details of the invariant specification.

An object is Valid if  $o.st = Valid$  and Invalid if  $o.st = Invalid$ . Only way to modify state is with the pack and unpack methods. Pack, changes from Invalid to Valid. Unpack, changes from Valid to Invalid.

```
pack o    ≡  assert o ≠ null ∧ o.st = Invalid ;
           assert Invt(o) ;
           o.st := Valid
unpack o  ≡  assert o ≠ null ∧ o.st = Valid ;
           o.st := Invalid
```

Restrict field updates to invalid objects—mark the object invalid, modify the field, mark it valid.

```
public method M()  
  requires st = Valid ;  
  modifies x, y ;  
  {  
    assert  $y - x \geq 0$  ;  
    unpack this ;  
     $x := x + 3 ; y := 4 * y ;$   
    pack this ;  
  }
```

Further discussion goes on to specify composition, object ownership, and hiding of cross object invariant implementations while preserving the publicly viewable validity state. Not including all the specific details here, they build upon the core concept described above and is specified in detail in the paper.