Towards a Verifying Compiler: The Spec# Approach

Wolfram Schulte

Microsoft Research

Formal Methods 2006

Joint work with **Rustan Leino, Mike Barnett**, Manuel Fähndrich, Herman Venter, Rob DeLine, Wolfram Schulte (all MSR), and *Peter Müller* (ETH), *Bart Jacobs* (KU Leuven) and Bor-Yuh Evan Chung (Berkley)

The Verifying Compiler

"A verifying compiler uses
automated .. reasoning to check the correctness
of the program that it compiles.

Correctness is specified by
types, assertions, .. and other redundant annotations
that accompany the program."
[Hoare, 2004]

Spec# Approach for a Verifying Compiler

- As *source language* we use C#
- As *specifications* we use method contracts, invariants, and also class, field and type annotations
- As *program logic* we use Dijkstra's weakest preconditions
- For *automatic verification* we use type checking, verification condition generation (VCG) and automatic theorem proving (ATP)

Spec#: Research Challenge

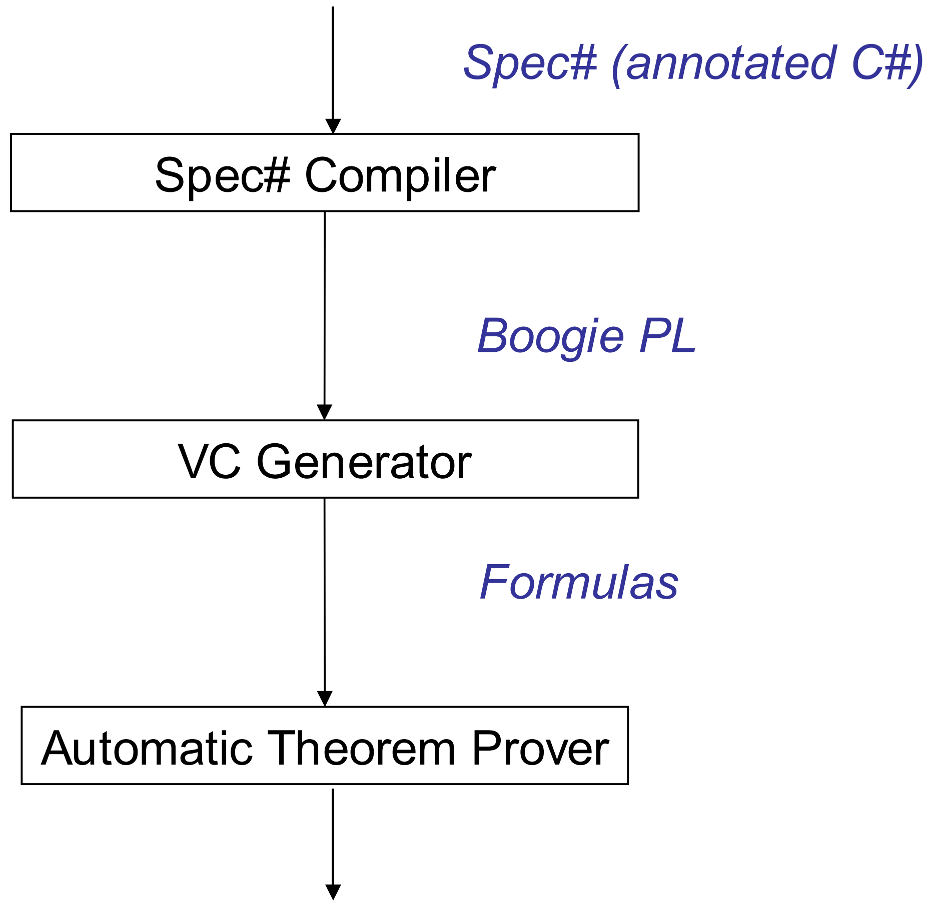
How to *verify object oriented programs and in particular object invariants*

in the presence of

- Callbacks
- Aliasing
- Inheritance
- Multi-threading

Demo (Spec#)

Spec# Tool Architecture



Goal of these Lectures

Enable participants to

- Understand and verify Spec# programs
- Understand and verify Boogie PL programs
- Build your own verifier [reusing Boogie]

Lectures

1. Verification Condition Generation
2. Logic of Object-oriented Programs
3. Invariants and Ownership
4. Abstraction
5. Multithreaded Programs

From Boogie PL
To Formulas

From Spec#
To BoogiePL

Lecture 1

Verification Condition Generation for Boogie PL

Unstructured Code
Theories
Theorem Provers

Boogie PL

Source language
(eg. Spec#)

Translate source language features using particular programming methodology

Intermediate language for automatic verification of imperative code

BoogiePL

Translate Boogie PL code using particular VC generation

Formulas

Boogie PL: Parts

Boogie PL source contains

- *a first order theory* to encode the background semantics of the source language and the program, described by

  constants, functions and axioms

- *an imperative part* used to encode the traces of the source program, described by:

  procedures, pre and postconditions,
  mutable variables, and unstructured code

Limits of Boogie PL

Boogie PL does not contain

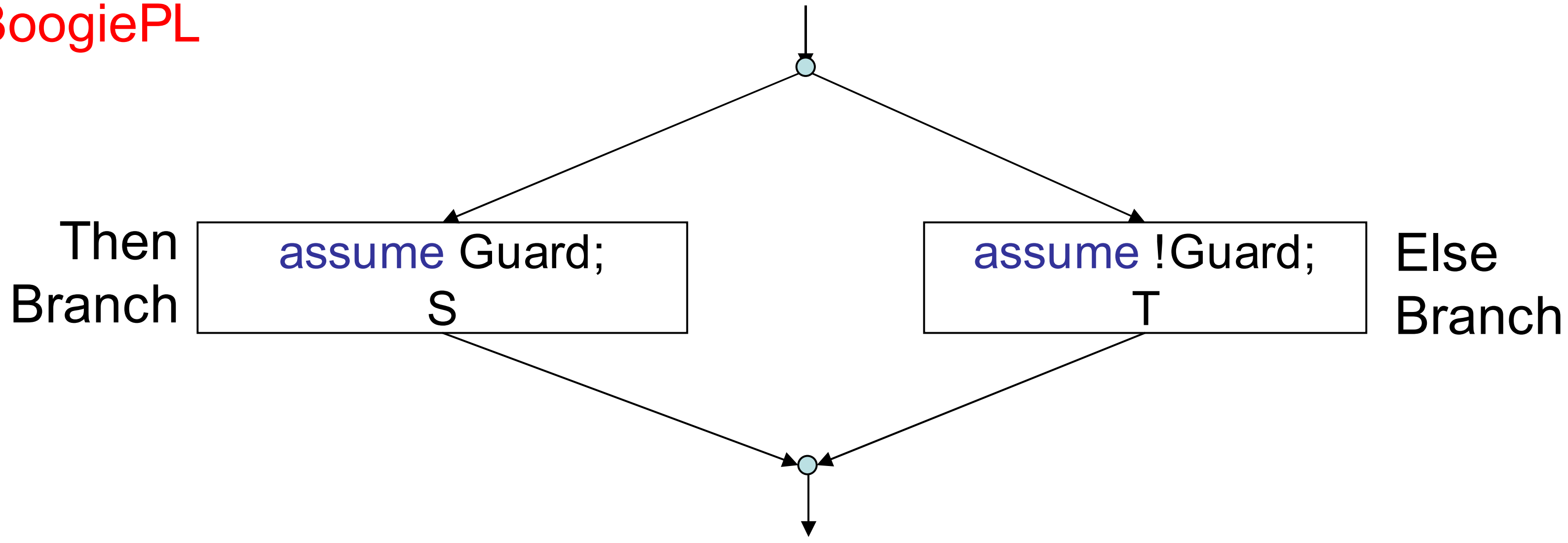
- structured control flow
- structured types
- a heap
- expressions with side effects
- visibility
- subtyping
- dynamic dispatch

Motivation: Spec#'s *Conditional* to Boogie PL

Spec#

```
if (Guard) S else T
```

BoogiePL

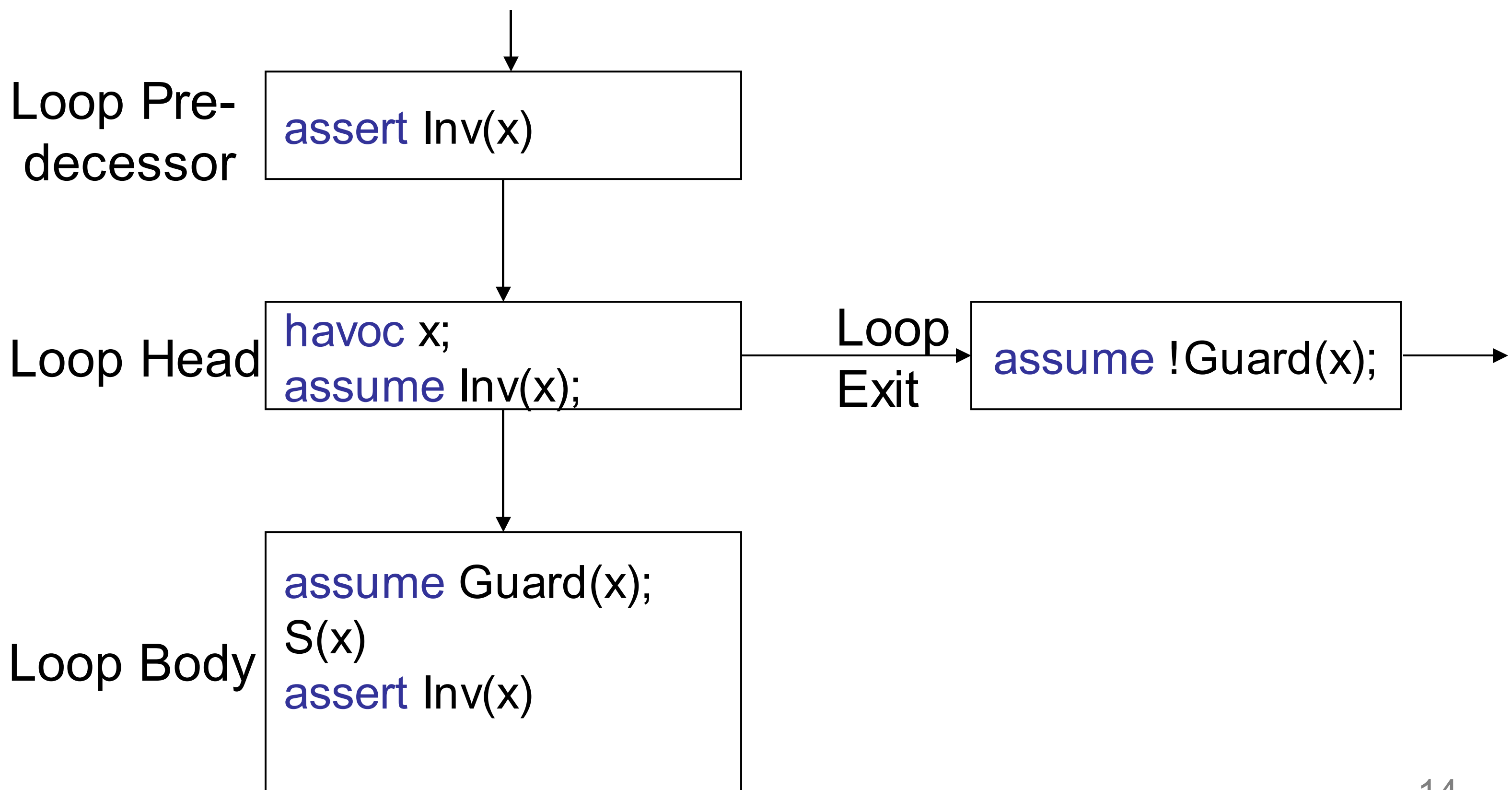


Motivation: Spec# ‘s *While Loops* to Boogie PL

Spec#

```
while (Guard(x)) invariant Inv(x) { S(x) }
```

BoogiePL



Boogie PL: Code

Code is unstructured

Code	::= VarDecl* Block+
Block	::= Label: Cmd goto Label+; | return;
Cmd	::= Passive | Assign | Call
Passive	::= assert E | assume E | Cmd ; Cmd
Assign	::= id := E | havoc id
Call	::= call id := P(E)

Variables are (weakly) typed

VarDecl ::= var id : Type

Type ::= int | bool | Array |…

Array ::= [Type+] Type

Remark: Types disappear during VCG; they are (if ncessary) encoded as axioms.

Boogie PL: Meaning of Code

For any command S and predicate Q, which describes the result of executing S, we define another predicate, its *weakest precondition,* denoted by wp(S,Q), that represents the set of *all* states such that execution of S begun in any of those states

- does not go wrong, and
- if it terminates, terminates in Q

Verification Condition Generation

2. Passive commands: assert, assume, ;
3. Acyclic control flow: goto (no loops)
4. State changes: :=, havoc
5. Loops
6. Procedure calls

VCG 1: Passive Commands

assert E

- Programmer claims that the condition E holds
- Verifier checks E

wp(assert E, Q) = E $\wedge$ Q

assume E

- Programmer cares only about executions where E holds
- Verifier uses E as an assumption henceforth

wp(assume E, Q) = E $\Rightarrow$ Q

S; T

– wp(S; T, E) = wp(S, wp(T, E))

VCG 1: Examples

- wp(assert $x>1$, Q)

  $= x>1 \wedge Q$
- wp(assert true, Q)

  = Q
- wp(assume $y=x+1$, $y=5$)

  $= (y=x+1 \Rightarrow y=5)$
- wp(assume false, Q)

  = true
- wp(assert P; assume P, Q)

  $= P \wedge (P \Rightarrow Q)$

VCG 1: Assume-Assert Reasoning

$$
\begin{aligned}
& wp(\text{ assume } P;\ S;\ \text{assert } Q,\ \text{true}) \\
= \;& wp(\text{ assume } P,\ wp\ (S,\ wp(\text{ assert } Q,\ \text{true}))) \\
= \;& \qquad wp(\text{ assume } P,\ wp\ (S,\ Q)) \\
= \;& P \Rightarrow wp(S,Q)
\end{aligned}
$$

VCG 1: Correctness for Procedures (simplified)

Let proc M(par) returns (res) requires P, ensures Q

and impl M(par) returns (res) { start: S; return; }

Then

valid (M) = wp (assume P; S; assert Q, true) =

$P \Rightarrow$ wp(S,Q)

We will refine this later.

VCG 2: Acyclic Control Flow

The problem of redundancy

$$wp(l_0{:}S_0;\ \text{goto}\ l_1,..l_n,\ Q) =$$
$$wp(S_0,\ wp(l_1{:}S_1,\ Q) \wedge \ldots \wedge wp(l_n{:}S_n,\ Q))$$

How can we get a linear (in size of the passive program) formula?

VCG 2: Acyclic Control Flow

- For each block A = L: S goto $L_{B1},..,L_{Bn}$ introduce a variable *A_{ok}, which holds when all executions starting at A are okay.*

- Introduce a Block Equation for each block A (BE_A):

$$A_{ok} \equiv \mathrm{wp}(S, (\forall B \in Succ(A) : B_{ok}))$$

- VC (semantics of entire code):

$$(\forall A : BE_A) \Rightarrow Start_{ok}$$

VCG 3: State Changes

The wp for control flow assumes stateless blocks

How do we get rid of assignments?

(3) Establish dynamic single assignment form (DSA), i.e. there is at most one definition for each variable on each path

- Replace defs/uses with new incarnations

  x := x+1 *with* $x_{n+1} = x_n + 1$

- Replace havoc x with new incarnations x_{n+1}
- At join points unify variable incarnations

2) Eliminate assignments by replacing

x := E *with* assume x = E

VCG 4: Loops

Loops introduce back edges in control flow graph. But technique can only deal with acyclic graphs.

How do we get rid of back edges?

We showed the result of this transformation earlier in the slide entitled: Spec# ‘s*While Loops* to Boogie PL

In detail:

8. Duplicate loop invariant P by using
   assert P = assert P; assume P
9. Check loop invariant at loop entry and exit
10. Delete back edges after “havoc”-ing loop targets

Boogie PL: Procedures

- Declaration

```
proc Find(xs: [int] int, ct: int, x: int) returns (result: int);
```

- Implementation

```
impl Find(xs: [int] int, ct: int, x: int) returns (result: int)
    {…}
```

- Call

```
call r := Find(bits, 100, true)
```

Remark: In Boogie PL the keywords are procedure and implementation

Boogie PL: Procedure Specifications

Caller obligations described by

- Precondition

Implementation obligation described by

- Postcondition

```
proc Find(xs: [int] int, ct: int, x: int) returns (result: int);
    requires ct≥0;
    ensures result ≥ 0 ⇒ result < ct ∧ xs[result]=x;
    ensures result < 0 ⇒ !(∃ i:int :: 0≤i ∧ i<ct ∧ xs[i] == x);
```

A specification spells out the entire contract.

A Bogus Implementation?

```
var xs: [int] int;
var ct: int;

proc Find(x: int) returns (result: int);
    requires ct≥0;
    ensures result ≥ 0 ⇒ result < ct ∧ xs[result]=x;
    ensures result < 0 ⇒ ! (∃ i:int :: 0≤i ∧ i<ct ∧ xs[i] == x);

impl Find(x: int) returns (result: int)
    { start:       ct := 0; result := -1; return; }
```

More about Postconditions

Postconditions

- often relate pre-state and post-state
  - ensures x == old(x)+1;
- must say which variables x might change
  - modifies x;

  variables not mentioned are not allowed to change

```
proc Find(x: int) returns (result: int);
  …
  modifies ct;               // would allow the previous implementation
  ensures ct == old(ct); // would disallow the change (despite
                             // modifies clause)
```

VCG 5: Calls

Given

```
proc P(par) returns (res)
  requires Pre; modifies state; ensures Post;
```

Then

```
wp(call x =  P(E), R)
   =
   wp(  {var par, res;
         par := E;
         assert Pre;
         havoc state;
         assume Post;
         x :=  res          }, R)
```

Remark: par and res are assumed to be fresh locals in the method body’s scope

VCG 5: Bodies

Given

proc P(par) returns (res)
  requires Pre; modifies state; ensures Post;
impl P(par) returns (res)
  {var …; start: S goto … end: return;}

Then

valid (P) =
  let (start: S’ goto … end: S’’; return;) = Passify(MakeAcyclic
    (start: assume Pre; S goto …
     end: assert Post[old(par) par_0]; return;)
  in (start$_{ok}$ ≡ wp(*S’*, (∀ b ∈ succs(start) : b$_{ok}$)) …
    end$_{ok}$ ≡ wp(S’’, true)
    ⇒ start$_{ok}$)

Remark: assumes that all normal terminations of P terminate at “end”.

BoogiePL: Arrays and Background

Boogie 's *array operations are just a short hand notation*, i.e.

x := a[i]	≡ x := select(a, i)
a[i] := E	≡ a := store(a, i, E)

select and store are defined as (untyped) *axioms* in Boogie's background predicate

$$
\begin{aligned}
&(\forall m,i,j,v \\
&\qquad i \neq j \Rightarrow \quad select(store(m, i, v), i) = v \\
&\qquad\qquad \wedge\ select(store(m, i, v), j) = select(m, j))
\end{aligned}
$$

Boogie PL: Final VCG

- Boogie PL has a universal background predicate BP_{Univ}
- Each Boogie PL program has a local theory BP_{Prog}
- The generated VC for each procedure implementation P is:

$$\boxed{BP_{Univ} \wedge BP_{Prog} \Rightarrow \text{valid}(P)}$$

Background: Automatic Theorem Provers

Usable ATPs have to support first order logic

- examples: Simplify, Zap, SMT solvers

They are build on Nelson-Oppen cooperating decision procedures and have decision procedures for

- congruence closure
- linear arithmetic
- partial orders
- quantifiers

Their key features are

- automatic: no user interaction
- refutation based: searches for counterexamples
- heuristics tuned for program checking
- Labels and time limit

Summary

Boogie PL is a simple intermediate language.

Boogie supports

- Modular verification using contracts
- Linear (in size of the code) VC generation
- A standard background as well as a program specific one

Appendix: VCG Example

```
start   : assume x > 100;
                                        goto loop;
loop    : assert  x >= 0;
                                        goto body, end;
body    : assume x > 0;
          x := x - 1;
                                        goto loop;
end     : assume !(x > 0);
          assert x == 0;                return;
```

Create assume

```
start   : assume x > 100;
                                        goto loop;
loop    : assert  x >= 0;
          assume x>=0;                  goto body, end;
body    : assume x > 0;
          x := x - 1;
                                        goto loop;
end     : assume !(x > 0);
          assert x == 0;                return;
```

Move loop invariant into Loop-Pre-Header and after Loop Body

```
start   : assume x > 100;
          assert  x >= 0;                         goto loop;
loop    :
          assume x>=0;                            goto body, end;
body    : assume x > 0;
          x := x - 1;
          assert  x >= 0;                         goto loop;
end     : assume !(x > 0);
          assert x == 0;                          return;
```

Cut back jumps: assume havoc on variables assigned inside the loop;block loop body

```
start  : assume x > 100;
         assert  x >= 0;                  goto loop;
loop   : havoc x;
         assume x>=0;                     goto body, end;
body   : assume x > 0;
         x := x - 1;
         assert  x >= 0;                  return;
end    : assume !(x > 0);
         assert x == 0;                   return;
```

Create Dynamic Single Assignment Form

```
start  : assume x > 100;
         assert  x >= 0;                          goto loop;
loop   : skip
         assume x1>=0;                            goto body, end;
body   : assume x1 > 0;
         x2 := x1 - 1;
         assert  x2 >= 0;                         return;
end    : assume !(x1 > 0);
         assert x1 == 0;                          return;
```

Passify Assigments

```
start  : assume x > 100;
         assert  x >= 0;                    goto loop;
loop   : skip
         assume x1>=0;                      goto body, end;
body   : assume x1 > 0;
         assume x2 == x1 - 1;
         assert  x2 >= 0;                   return;
end    : assume !(x1 > 0);
         assert x1 == 0;                    return;
```

Apply Block Translation and wp

start	$\equiv$ x > 100 $\Rightarrow$	
	x >= 0 $\wedge$	loop
loop	$\equiv$	
	x1 >= 0 $\Rightarrow$	body $\wedge$ end
body	$\equiv$ x1 > 0 $\Rightarrow$	
	x2 = x1 - 1 $\Rightarrow$	
	x2 >= 0 $\wedge$	true
end	$\equiv$!(x1 > 0) $\Rightarrow$	
	x1 = 0 $\Rightarrow$	true

$\Rightarrow$ start