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

Spec# (annotated C#)

Spec# Compiler

Boogie PL

VC Generator

Formulas

Automatic Theorem Prover
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#

\[
\text{if } (\text{Guard}) \ S \ \text{else} \ T
\]

BoogiePL

Then Branch

\[\text{assume Guard;} \ S\]

else Branch

\[\text{assume !Guard;} \ T\]
Motivation: Spec# ‘s While Loops to Boogie PL

Spec#

\[
\text{while } (\text{Guard}(x)) \text{ invariant Inv}(x) \{ S(x) \}
\]

BoogiePL

Loop Predecessor

assert Inv(x)

Loop Head

havoc x; assume Inv(x);

assume Guard(x); S(x) assert Inv(x)

Loop Exit

assume !Guard(x);
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 necessary) encoded as axioms.
For any command $S$ and predicate $Q$, which describes the result of executing $S$, we define another predicate, its \textit{weakest precondition}, denoted by $wp(S,Q)$, that represents the set of \textit{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 $E$

$$wp(\text{assert } E, Q) = E \land Q$$

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

$$wp(\text{assume } E, Q) = E \Rightarrow Q$$

$S; T$
- $wp(S; T, E) = wp(S, wp(T, E))$
VCG 1: Examples

- $wp(\text{assert } x>1, Q) = x>1 \land Q$
- $wp(\text{assert } \text{true}, Q) = Q$
- $wp(\text{assume } y=x+1, y=5) = (y=x+1 \implies y=5)$
- $wp(\text{assume } \text{false}, Q) = \text{true}$
- $wp(\text{assert } P; \text{assume } P, Q) = P \land (P \implies Q)$
VCG 1: Assume-Assert Reasoning

\[ wp( \text{assume } P; S; \text{assert } Q, \text{true}) \]

\[ = \ wp( \text{assume } P, wp(S, wp( \text{assert } Q, \text{true}))) \]

\[ = \ wp( \text{assume } P, wp(S, Q)) \]

\[ = \ P \Rightarrow wp(S, Q) \]
VCG 1: Correctness for Procedures (simplified)

Let \( \text{proc } M(\text{par}) \text{ returns (res) requires P, ensures Q} \)
and \( \text{impl } M(\text{par}) \text{ returns (res)} \{ \text{start: S; return; } \} \)

Then
\[
\text{valid (M)} = \wp (\text{assume P; S; assert Q, true}) = P \Rightarrow \wp(S,Q)
\]

We will refine this later.
VCG 2: Acyclic Control Flow

The problem of redundancy

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

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$_A$):

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

• VC (semantics of entire code):

\[
(\forall A : BE_A) \Rightarrow Start_{ok}
\]
VCG 3: State Changes

The \textit{wp} for control flow assumes stateless blocks

\textbf{How do we get rid of assignments?}

\textit{(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
  \[ x := x + 1 \quad \text{with} \quad x_{n+1} = x_n + 1 \]
- Replace \textit{havoc} \( x \) with new incarnations \( x_{n+1} \)
- At join points unify variable incarnations

\textit{2) Eliminate assignments} by replacing

\[ x := E \quad \text{with} \quad \text{assume} \; x = E \]
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
   \[
   \text{assert } P = \text{assert } P; \ \text{assume } P
   \]
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

```plaintext
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 \( x == \text{old}(x)+1; \)
- 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 == \text{old}(ct);  // would disallow the change (despite
                                  // modifies clause)
```
VCG 5:Calls

Given

\[
\text{proc } P(\text{par}) \text{ returns } (\text{res}) \\
\text{requires } \text{Pre}; \text{ modifies } \text{state}; \text{ ensures } \text{Post};
\]

Then

\[
\text{wp}(\text{call } x = P(E), \ R) = \text{wp}( \begin{array}{l}
\text{var } \text{par}, \text{res}; \\
\text{par} := E; \\
\text{assert } \text{Pre}; \\
\text{havoc } \text{state}; \\
\text{assume } \text{Post}; \\
x := \text{res}
\end{array}, \ R)
\]

Remark: \text{par} and \text{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) par₀]; return;)
  in (start₀k ≡ wp(S’, (∀ b ∈ succs(start) : b₀k)) …
   end₀k ≡ wp(S’’, true)
   ⇒ start₀k)
```

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.

\[
\begin{align*}
x := a[i] & \equiv x := \text{select}(a, i) \\
a[i] := E & \equiv a := \text{store}(a, i, E)
\end{align*}
\]

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

\[
(\forall m, i, j, v \\
\quad i \neq j \Rightarrow \text{select}(\text{store}(m, i, v), i) = v \\
\quad \land \text{select}(\text{store}(m, i, v), j) = \text{select}(m, j))
\]
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:

\[ BP_{Univ} \land BP_{Prog} \implies valid(P) \]
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;

loop  : assert x >= 0;

goto loop;

body  : assume x > 0;
        x := x - 1;
        goto body, end;

end   : assume !(x > 0);
        assert x == 0;
        return;
### Create assume

<table>
<thead>
<tr>
<th>start</th>
<th>: assume x &gt; 100;</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>: assert x &gt;= 0;</td>
</tr>
<tr>
<td></td>
<td>assume x&gt;=0;</td>
</tr>
<tr>
<td>body</td>
<td>: assume x &gt; 0;</td>
</tr>
<tr>
<td></td>
<td>x := x - 1;</td>
</tr>
<tr>
<td></td>
<td>goto loop;</td>
</tr>
<tr>
<td>end</td>
<td>: assume !(x &gt; 0);</td>
</tr>
<tr>
<td></td>
<td>assert x == 0;</td>
</tr>
<tr>
<td></td>
<td>goto loop;</td>
</tr>
<tr>
<td></td>
<td>return;</td>
</tr>
</tbody>
</table>
Move loop invariant into Loop-Pre-Header and after Loop Body

\begin{align*}
\text{start} & : \text{assume } x > 100; \\
& \quad \text{assert } x \geq 0; \quad \text{goto loop}; \\
\text{loop} & : \\
& \quad \text{assume } x \geq 0; \quad \text{goto body, end}; \\
\text{body} & : \text{assume } x > 0; \\
& \quad x := x - 1; \\
& \quad \text{assert } x \geq 0; \quad \text{goto loop}; \\
\text{end} & : \text{assume } !(x > 0); \\
& \quad \text{assert } x == 0; \quad \text{return}; \\
\end{align*}
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

<table>
<thead>
<tr>
<th>start</th>
<th>assume x &gt; 100;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>assert x &gt;= 0;</td>
</tr>
<tr>
<td></td>
<td>goto loop;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loop</th>
<th>skip</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>assume x1 &gt;= 0;</td>
</tr>
<tr>
<td></td>
<td>goto body, end;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>body</th>
<th>assume x1 &gt; 0;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x2 := x1 - 1;</td>
</tr>
<tr>
<td></td>
<td>assert x2 &gt;= 0;</td>
</tr>
<tr>
<td></td>
<td>return;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>end</th>
<th>assume !(x1 &gt; 0);</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>assert x1 == 0;</td>
</tr>
<tr>
<td></td>
<td>return;</td>
</tr>
</tbody>
</table>
Passify Assignments

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 \textit{wp}

\[
\begin{align*}
\text{start} & \equiv x > 100 \Rightarrow \\
& \quad \quad x \geq 0 \land \quad \quad \text{loop} \\
\text{loop} & \equiv \quad \\
& \quad \quad x1 \geq 0 \Rightarrow \quad \quad \text{body} \land \text{end} \\
\text{body} & \equiv x1 > 0 \Rightarrow \\
& \quad \quad x2 = x1 - 1 \Rightarrow \\
& \quad \quad \quad \quad x2 \geq 0 \land \quad \quad \text{true} \\
\text{end} & \equiv !(x1 > 0) \Rightarrow \\
& \quad \quad x1 = 0 \Rightarrow \quad \quad \text{true} \\
\Rightarrow \text{start}
\end{align*}
\]