

HOL-Boogie — An Interactive Prover for the Boogie Program-Verifier

Sascha Böhme¹, K. Rustan M. Leino² and Burkhart Wolff³

¹Technische Universität München, ²Microsoft Research, ³Université Paris-Sud

TPHOLs 2008, Montréal

Outline

- 1 Motivation
- 2 Overview of Boogie
- 3 HOL-Boogie
- 4 Applications
- 5 Conclusion

Background

Verifying compiler toolchain developed
by Microsoft Research:

Background

C# with
specification



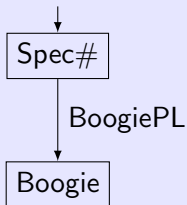
Spec#

Verifying compiler toolchain developed
by Microsoft Research:

- **Spec#**: compiler for annotated C#
(with method contracts, object
invariants)

Background

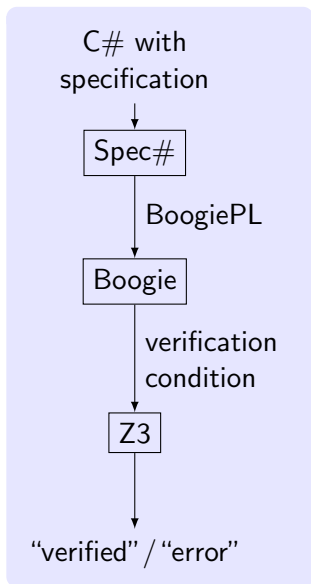
C# with
specification



Verifying compiler toolchain developed
by Microsoft Research:

- **Spec#**: compiler for annotated C# (with method contracts, object invariants)
- **Boogie**: static program verifier

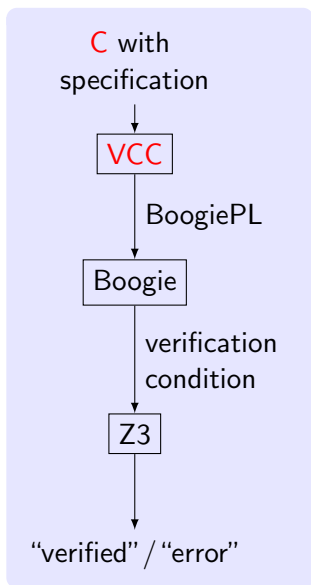
Background



Verifying compiler toolchain developed by Microsoft Research:

- Spec#: compiler for annotated C# (with method contracts, object invariants)
- Boogie: static program verifier
- Z3: automated SMT solver

Background

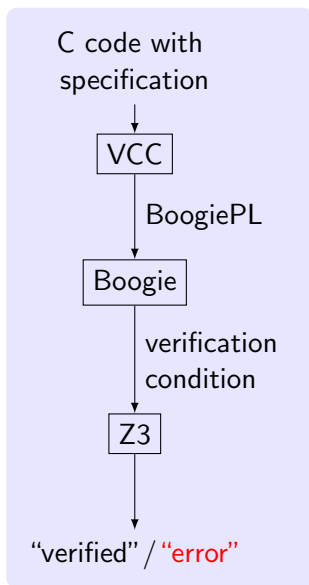


Verifying compiler toolchain developed by Microsoft Research:

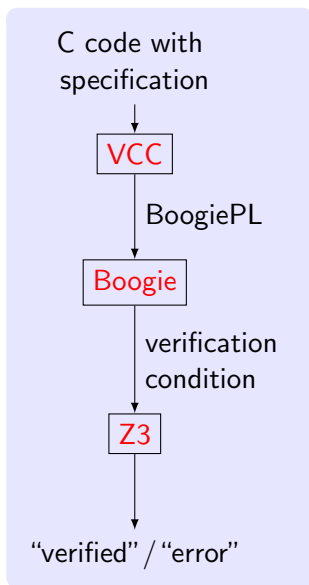
- VCC: Verifying C Compiler
- Boogie: static program verifier
- Z3: automated SMT solver

Motivation

Failures of proof attempts:



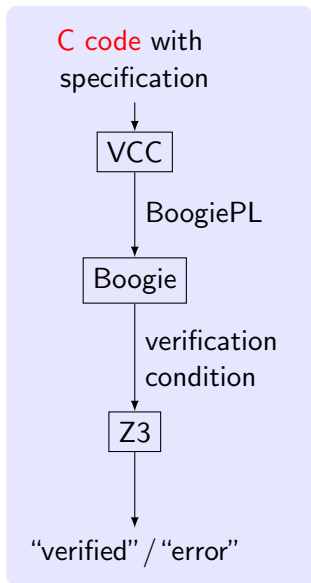
Motivation



Failures of proof attempts:

- caused by the prover?

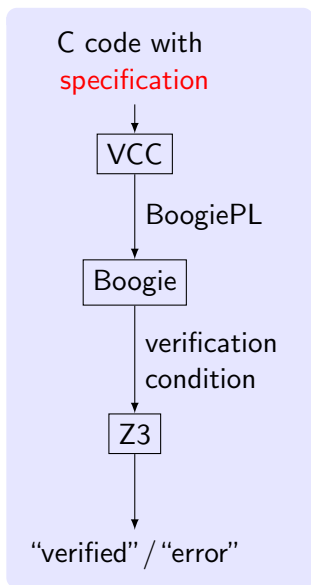
Motivation



Failures of proof attempts:

- caused by the prover?
- the program?

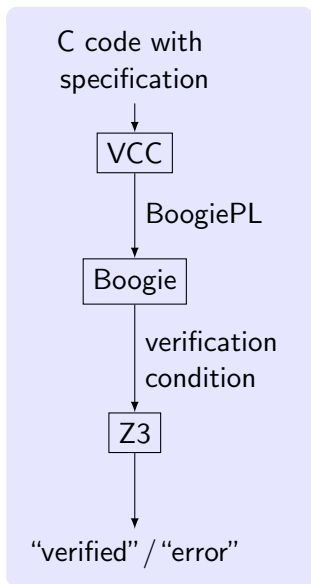
Motivation



Failures of proof attempts:

- caused by the prover?
- the program?
- the specification?

Motivation



Failures of proof attempts:

- caused by the prover?
- the program?
- the specification?

In particular with:

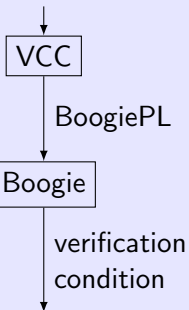
- recursive data structures
- recursive algorithms
- complex invariants

Apparently:

- reaching the limits of the automated tools

Motivation

C code with
specification

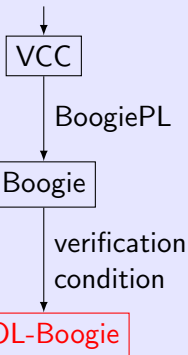


Solution:

- interactive proofs
- automated solvers as backends

Motivation

C code with
specification



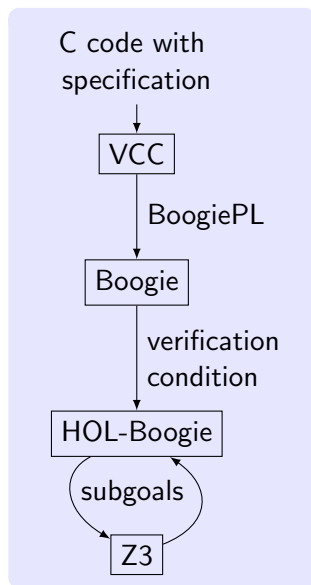
Solution:

- interactive proofs
- automated solvers as backends

HOL-Boogie:

- based on Isabelle/HOL

Motivation



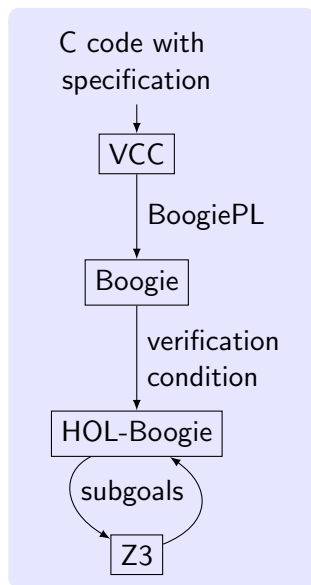
Solution:

- interactive proofs
- automated solvers as backends

HOL-Boogie:

- based on Isabelle/HOL
- combines automated and interactive tools

Motivation



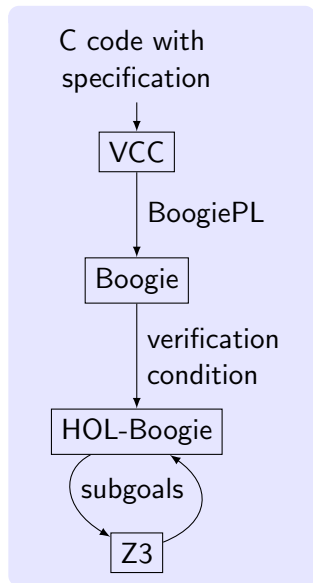
Solution:

- interactive proofs
- automated solvers as backends

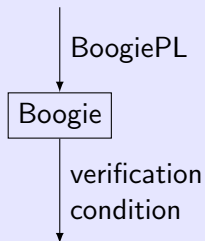
HOL-Boogie:

- based on Isabelle/HOL
- combines automated and interactive tools
- includes features to make interactive proofs manageable

Overview of Boogie



Overview of Boogie



BoogiePL example

```
function max(a:int, b:int) returns (m:int);
```

BoogiePL example

```
function max(a:int, b:int) returns (m:int);
```

```
axiom ( $\forall a:int, b:int :: \max(a,b) \geq a \wedge \max(a,b) \geq b$ );
```

BoogiePL example

function max(a:int, b:int) **returns** (m:int);

axiom ($\forall a:\text{int}, b:\text{int} :: \text{max}(a,b) \geq a \wedge \text{max}(a,b) \geq b$);

procedure Max(array:[int]int, length:int) **returns** (m:int);

requires ($0 < \text{length}$);

ensures ($\forall k:\text{int} :: 0 \leq k \wedge k < \text{length} \longrightarrow \text{array}[k] \leq m$);

BoogiePL example

```
function max(a:int, b:int) returns (m:int);  
axiom ( $\forall a$ :int,  $b$ :int :: max(a,b)  $\geq$  a  $\wedge$  max(a,b)  $\geq$  b);  
procedure Max(array:[int]int, length:int) returns (m:int);  
  requires (0 < length);  
  ensures ( $\forall k$ :int :: 0  $\leq$  k  $\wedge$  k < length  $\longrightarrow$  array[k]  $\leq$  m);  
implementation Max(array:[int]int, length:int) returns (m:int)  
{ var p:int;   m := 0;   p := 0;  
  while (p < length)  
    invariant ( $\forall k$ :int :: 0  $\leq$  k  $\wedge$  k < p  $\longrightarrow$  array[k]  $\leq$  m);  
    {  
      m := max(m, array[p]);  
      p := p + 1;  
    }  
}
```

BoogiePL example

```
function max(a:int, b:int) returns (m:int);  
axiom ( $\forall a$ :int,  $b$ :int :: max(a,b)  $\geq$  a  $\wedge$  max(a,b)  $\geq$  b);  
procedure Max(array:[int]int, length:int) returns (m:int);  
  requires (0 < length);  
  ensures ( $\forall k$ :int :: 0  $\leq$  k  $\wedge$  k < length  $\longrightarrow$  array[k]  $\leq$  m);  
implementation Max(array:[int]int, length:int) returns (m:int)  
{ var p:int;   m := 0;   p := 0;  
  while (p < length)  
    invariant ( $\forall k$ :int :: 0  $\leq$  k  $\wedge$  k < p  $\longrightarrow$  array[k]  $\leq$  m);  
    {  
      m := max(m, array[p]);  
      p := p + 1;  
    }  
}
```

BoogiePL example

```
function max(a:int, b:int) returns (m:int);  
axiom ( $\forall a:\text{int}, b:\text{int} :: \text{max}(a,b) \geq a \wedge \text{max}(a,b) \geq b$ );  
procedure Max(array:[int]int, length:int) returns (m:int);  
  requires ( $0 < \text{length}$ );  
  ensures ( $\forall k:\text{int} :: 0 \leq k \wedge k < \text{length} \longrightarrow \text{array}[k] \leq m$ );  
implementation Max(array:[int]int, length:int) returns (m:int)  
{ var p:int;   m := 0;   p := 0;  
  while (p < length)  
    invariant ( $\forall k:\text{int} :: 0 \leq k \wedge k < p \longrightarrow \text{array}[k] \leq m$ );  
    {  
      m := max(m, array[p]);  
      p := p + 1;  
    }  
}
```

BoogiePL example

```
function max(a:int, b:int) returns (m:int);  
axiom ( $\forall a:\text{int}, b:\text{int} :: \text{max}(a,b) \geq a \wedge \text{max}(a,b) \geq b$ );  
procedure Max(array:[int]int, length:int) returns (m:int);  
  requires ( $0 < \text{length}$ );  
  ensures ( $\forall k:\text{int} :: 0 \leq k \wedge k < \text{length} \longrightarrow \text{array}[k] \leq m$ );  
implementation Max(array:[int]int, length:int) returns (m:int)  
{ var p:int;   m := 0;   p := 0;  
  while (p < length)  
    invariant ( $\forall k:\text{int} :: 0 \leq k \wedge k < p \longrightarrow \text{array}[k] \leq m$ );  
    {  
      m := max(m, array[p]);  
      p := p + 1;  
    }  
}
```

Internal transformation

```
requires (0 < length);  
m := 0;  p := 0;  
while (p < length)  
  invariant ( $\forall k:\text{int} :: 0 \leq k \wedge k < p \longrightarrow \text{array}[k] \leq m$ );
```

Internal transformation

```
requires ( $0 < \text{length}$ );  
 $m := 0$ ;  $p := 0$ ;  
while ( $p < \text{length}$ )  
  invariant ( $\forall k:\text{int} :: 0 \leq k \wedge k < p \longrightarrow \text{array}[k] \leq m$ );
```

Init:

```
assume  $0 < \text{length}$ ;  
assert ( $\forall k:\text{int} :: 0 \leq k \wedge k < 0 \longrightarrow \text{array}[k] \leq 0$ );  
goto LoopHead;
```

LoopHead:

```
assume  $0 \leq p_0$ ;  
assume ( $\forall k:\text{int} :: 0 \leq k \wedge k < p_0 \longrightarrow \text{array}[k] \leq m_0$ );  
goto LoopDone, LoopBody;
```

Internal transformation

```
requires (0 < length);  
m := 0;  p := 0;  
while (p < length)  
  invariant ( $\forall k:\text{int} :: 0 \leq k \wedge k < p \longrightarrow \text{array}[k] \leq m$ );
```

Init:

```
assume 0 < length;  
assert ( $\forall k:\text{int} :: 0 \leq k \wedge k < 0 \longrightarrow \text{array}[k] \leq 0$ );  
goto LoopHead;
```

LoopHead:

```
assume 0  $\leq$  p0;  
assume ( $\forall k:\text{int} :: 0 \leq k \wedge k < p_0 \longrightarrow \text{array}[k] \leq m_0$ );  
goto LoopDone, LoopBody;
```

VC generation

Init:

assume $0 < \text{length}$;

assert $(\forall k:\text{int} :: 0 \leq k \wedge k < 0 \longrightarrow \text{array}[k] \leq 0)$;

goto LoopHead;

VC generation

Init:

assume $0 < \text{length}$;

assert $(\forall k:\text{int} :: 0 \leq k \wedge k < 0 \longrightarrow \text{array}[k] \leq 0)$;

goto LoopHead;

$A_1 \equiv \text{block}_{(11,5)} \text{ True} \longrightarrow$

$0 < \text{length} \longrightarrow$

$\text{assert}_{(13,5)} (\forall k : \text{int} :: 0 \leq k \wedge k < 0 \longrightarrow \text{array}[k] \leq 0) \wedge$

$\text{LoopHead}_{\text{correct}} \longrightarrow$

$\text{Init}_{\text{correct}}$

VC generation

Init:

```
assume 0 < length;  
assert ( $\forall k:\text{int} :: 0 \leq k \wedge k < 0 \longrightarrow \text{array}[k] \leq 0$ );  
goto LoopHead;
```

$A_1 \equiv \text{block}_{(11,5)} \text{ True} \longrightarrow$ **Positional labels**
 $0 < \text{length} \longrightarrow$
 $\text{assert}_{(13,5)} (\forall k : \text{int} :: 0 \leq k \wedge k < 0 \longrightarrow \text{array}[k] \leq 0) \wedge$
 $\text{LoopHead}_{\text{correct}} \longrightarrow$
 $\text{Init}_{\text{correct}}$

VC generation

Init:

```
assume 0 < length;  
assert ( $\forall k:\text{int} :: 0 \leq k \wedge k < 0 \longrightarrow \text{array}[k] \leq 0$ );  
goto LoopHead;
```

$A_1 \equiv \text{block}_{(11,5)} \text{ True} \longrightarrow$
 $0 < \text{length} \longrightarrow$
 $\text{assert}_{(13,5)} (\forall k : \text{int} :: 0 \leq k \wedge k < 0 \longrightarrow \text{array}[k] \leq 0) \wedge$
 $\text{LoopHead}_{\text{correct}} \longrightarrow$
 $\text{Init}_{\text{correct}}$

Structural labels

VC generation

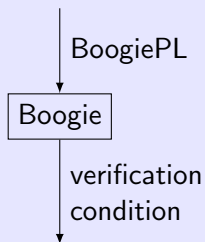
Init:

```
assume 0 < length;  
assert ( $\forall k:\text{int} :: 0 \leq k \wedge k < 0 \longrightarrow \text{array}[k] \leq 0$ );  
goto LoopHead;
```

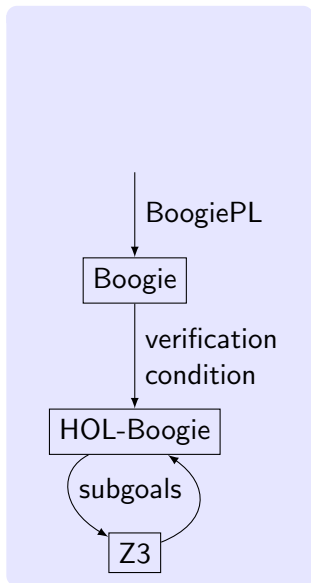
$$A_1 \equiv \text{block}_{(11,5)} \text{ True} \longrightarrow$$
$$0 < \text{length} \longrightarrow$$
$$\text{assert}_{(13,5)} (\forall k : \text{int} :: 0 \leq k \wedge k < 0 \longrightarrow \text{array}[k] \leq 0) \wedge$$
$$\text{LoopHead}_{\text{correct}} \longrightarrow$$
$$\text{Init}_{\text{correct}}$$

complete VC: $A_1 \wedge \dots \wedge A_n \longrightarrow \text{Init}_{\text{correct}}$

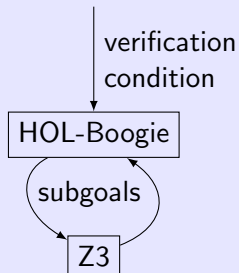
HOL-Boogie



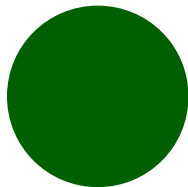
HOL-Boogie



HOL-Boogie



The HOL-Boogie approach



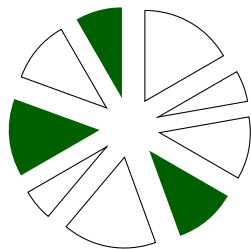
- take a VC generated by Boogie

The HOL-Boogie approach



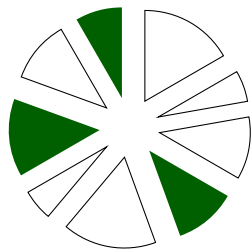
- take a VC generated by Boogie
- split the VC into subgoals for each assertion

The HOL-Boogie approach



- take a VC generated by Boogie
- split the VC into subgoals for each assertion
- apply an SMT solver (Z3) or some Isabelle tactics to every subgoal
- remaining subgoals: interactive proofs in Isabelle/HOL

The HOL-Boogie approach



- take a VC generated by Boogie
- split the VC into subgoals for each assertion
- apply an SMT solver (Z3) or some Isabelle tactics to every subgoal
- remaining subgoals: interactive proofs in Isabelle/HOL

Expectation (and experience):

- SMT solvers and Isabelle tactics discharge most subgoals
- remaining subgoals are usually the critical ones (incorrect or complicated)

Integration of SMT solvers

Currently:

- oracle mechanism (i.e. without verification by Isabelle)
- proof reconstruction is work in progress
- supported solvers: Z3 and any SMT solver understanding the standardized SMT-LIB format

Integration of SMT solvers

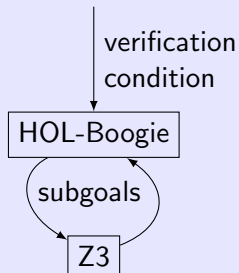
Currently:

- oracle mechanism (i.e. without verification by Isabelle)
- proof reconstruction is work in progress
- supported solvers: Z3 and any SMT solver understanding the standardized SMT-LIB format

Challenges:

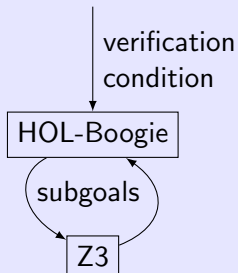
- preserve semantics
- pass through solver instrumentation from Boogie:
 - triggers: instances for quantifier instantiation
- efficiency

Applications of HOL-Boogie



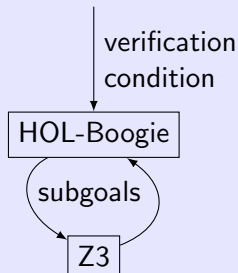
Applications of HOL-Boogie

- debug annotations



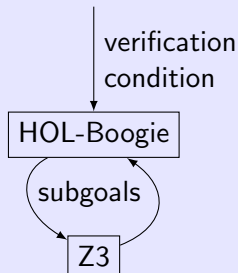
Applications of HOL-Boogie

- debug annotations
- verify programs not manageable by automated tools

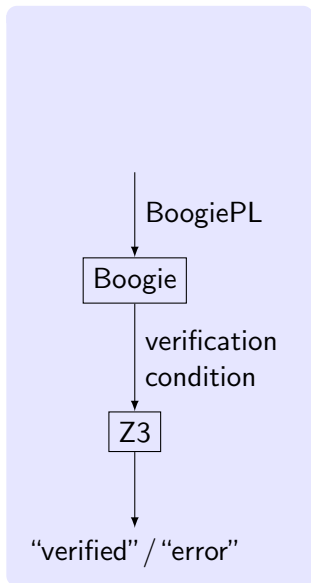


Applications of HOL-Boogie

- debug annotations
- verify programs not manageable by automated tools
- verify consistency of background theories



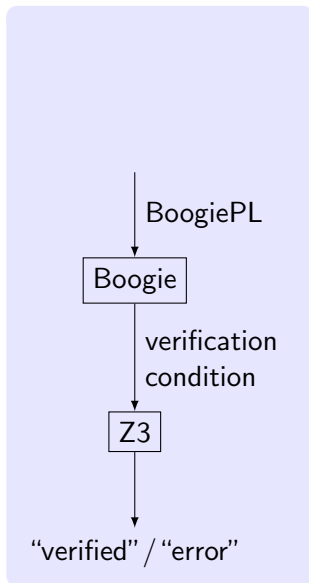
Back to automated tools



Back to automated tools

Specifications get easily wrong:

- incorrect function contracts
- missing loop invariants
- ...

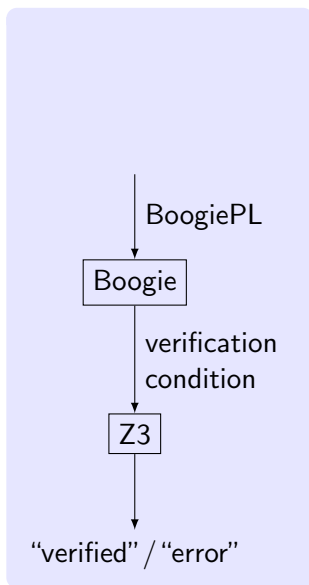


Back to automated tools

Specifications get easily wrong:

- incorrect function contracts
- missing loop invariants
- ...

How does Boogie respond?



Back to automated tools

Boogie error message

max.bpl(12,3): Error: A postcondition might not hold at this return statement.

max.bpl(6,3): Related location: This is the postcondition that might not hold.

Execution trace:

max.bpl(11,5): Init

max.bpl(12,3): LoopHead

max.bpl(12,3): LoopDone

Back to automated tools

Boogie error message

max.bpl(12,3): Error: A postcondition might not hold at this return statement.

max.bpl(6,3): Related location: This is the postcondition that might not hold.

Execution trace:

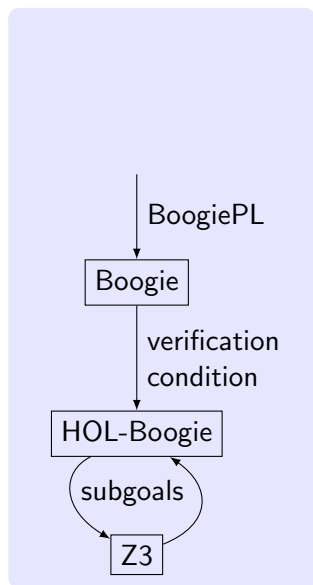
max.bpl(11,5): Init

max.bpl(12,3): LoopHead

max.bpl(12,3): LoopDone

How to find the **cause** for the error?

HOL-Boogie: Debugging annotations



HOL-Boogie: Debugging annotations

- complete “execution path” available

$$0 < \text{length} \longrightarrow$$

$$(\forall i. 0 \leq i \wedge i < 0 \longrightarrow \text{array}[i] \leq 0) \longrightarrow$$

$$0 \leq p_0 \longrightarrow$$

$$(\forall i. 0 \leq i \wedge i < p_0 \longrightarrow \text{array}[i] \leq m_0) \longrightarrow$$

$$p_0 \geq \text{length} \longrightarrow$$

$$(\forall i. 0 \leq i \wedge i < \text{length} \longrightarrow \text{array}[i] \leq m_0)$$

HOL-Boogie: Debugging annotations

- complete “execution path” available
- “execution path” and subgoals are **related to the source code**

block_(11,5) True \longrightarrow

$0 < \text{length} \longrightarrow$

$(\forall i. 0 \leq i \wedge i < 0 \longrightarrow \text{array}[i] \leq 0) \longrightarrow$

block_(12,3) True \longrightarrow

$0 \leq p_0 \longrightarrow$

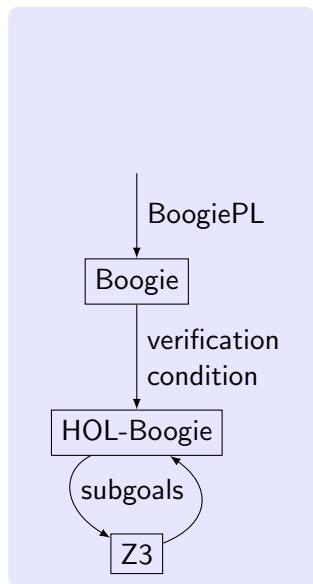
$(\forall i. 0 \leq i \wedge i < p_0 \longrightarrow \text{array}[i] \leq m_0) \longrightarrow$

block_(12,3) True \longrightarrow

$p_0 \geq \text{length} \longrightarrow$

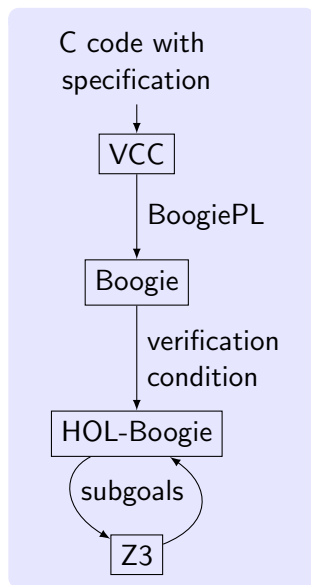
assert_(6,3) $(\forall i. 0 \leq i \wedge i < \text{length} \longrightarrow \text{array}[i] \leq m_0)$

HOL-Boogie: Verifying C programs



HOL-Boogie: Verifying C programs

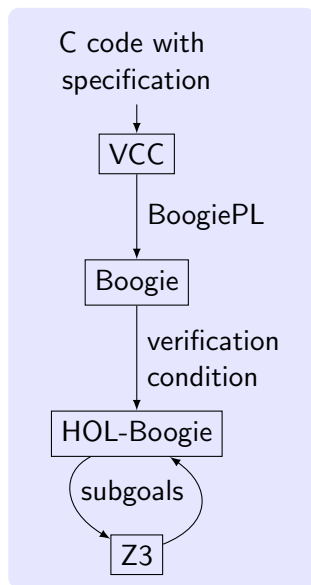
Same approach, but with specifics of C:



HOL-Boogie: Verifying C programs

Same approach, but with specifics of C:

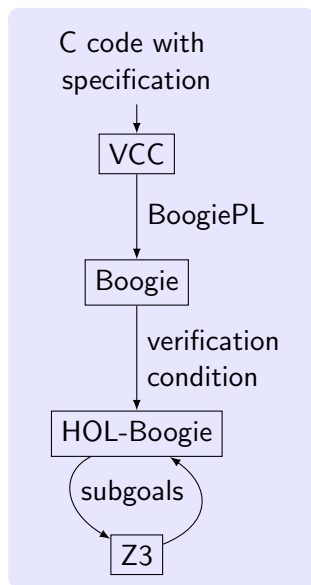
- explicit memory and ghost memory



HOL-Boogie: Verifying C programs

Same approach, but with specifics of C:

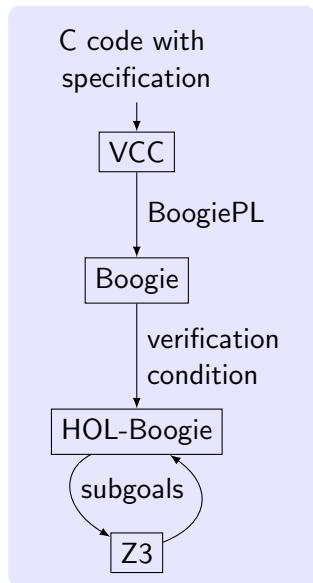
- explicit memory and ghost memory
- explicit control flow



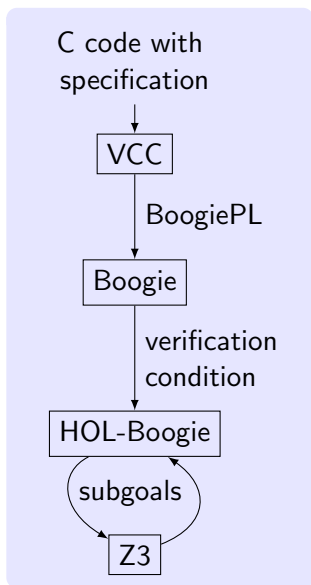
HOL-Boogie: Verifying C programs

Same approach, but with specifics of C:

- explicit memory and ghost memory
- explicit control flow
- explicit checks for arithmetic overflows, null-pointer dereferencing, ...



HOL-Boogie: Verifying C programs



Same approach, but with specifics of C:

- explicit memory and ghost memory
- explicit control flow
- explicit checks for arithmetic overflows, null-pointer dereferencing, ...
- large background theory (around 900 axioms):
 - x64 bitvector model
 - C memory model

Conclusion

Conclusion

HOL-Boogie:

- interactive backend for Boogie
- uses automated solvers, especially SMT solvers (Z3)

Conclusion

HOL-Boogie:

- interactive backend for Boogie
- uses automated solvers, especially SMT solvers (Z3)

Advantages:

- HOL-Boogie overcomes incompleteness of automated tools
- HOL-Boogie helps to debug and verify complex specifications
- labels make interactive proofs manageable

Conclusion

HOL-Boogie:

- interactive backend for Boogie
- uses automated solvers, especially SMT solvers (Z3)

Advantages:

- HOL-Boogie overcomes incompleteness of automated tools
- HOL-Boogie helps to debug and verify complex specifications
- labels make interactive proofs manageable

Future work:

- larger case studies: linked lists, quicksort, red-black-trees
- improved user interface: specialized tactics, structural labels
- improved SMT solver binding