Program Verification via an Intermediate Verification Language

K. Rustan M. Leino
Principal Researcher
Research in Software Engineering (RiSE), Microsoft Research, Redmond
Visiting Professor
Department of Computing, Imperial College London
Static program verification

What is the state-of-art in program verifiers?
How to build a program verifier
Dafny

Put reasoning about programs *first*

Language aimed at reasoning
  Constructs for recording design decisions

Tool support
  Static program verifier enforces design decisions

Integrated development environment
  Tools help in reasoning process
  Verification is not an afterthought

Program safely. With Dafny.
Demo

Queue implemented by a ring buffer

(data: 0)

(start + len) % data.Length

Enqueue at (start + len) % data.Length
Dequeue at start
Separation of concerns

C#

Compiler
Intermediate representation

C#

Verifier
Intermediate verification language

SMT solver
Verification architecture
Meet the family
Verification architecture

- Frama-C
- Krakatoa
- Jessie
- Who
- Hi-Lite Ada
- Pangoline
- CAO
- Why3
- Alt-Ergo
- SMT Lib
- Z3
- Coq
- Isabelle/HOL
- CVC 3
- ...
Boogie language overview

Mathematical features

- type T
- const x...
- function f...
- axiom E

Imperative features

- var y...
- procedure P... ...spec...
- implementation P... { ...body... }
Statement outcomes

Terminate
Go wrong
Block
Diverge
Boogie statements

\( x := E \)
Evaluate \( E \) and change \( x \) to that value

\( a[i] := E \)
Same as \( a := a[i := E] \)

\texttt{havoc } x
Change \( x \) to an arbitrary value

\texttt{assert } E
If \( E \) holds, terminate; otherwise, go wrong

\texttt{assume } E
If \( E \) holds, terminate; otherwise, block

\texttt{call } P()
Act according to specification of \( P \)

\texttt{if}
\texttt{while}
\texttt{break}
\texttt{label:}
\texttt{goto } A, B
Translation basics

**Ada**

```ada
x : Integer;

procedure Update (y : Integer; r : out Integer) is
begin
    if x < y then
        x := y;
    end if;
    r := y;
end Update;

procedure Main is
begin
    Update(5, x);
end Main;
```

**Boogie**

```boogie
var x: int;

procedure Update(y: int)
    returns (r: int)
    modifies x;
{
    if (x < y) {
        x := y;
    }
    r := y;
}

procedure Main()
    modifies x;
{
    call x := Update(5);
}
```
Unstructured control flow

.NET bytecode (MSIL)

.maxstack 2
.locals init ([0] int32 i, [1] bool CS$4$0000)

IL_0000:  nop
IL_0001:  ldc.i4.0
IL_0002:  stloc.0
IL_0003:  br.s    IL_000b
IL_0005:  nop
IL_0006:  ldloc.0
IL_0007:  ldc.i4.1
IL_0008:  add
IL_0009:  stloc.0
IL_000a:  nop
IL_000b:  ldloc.0
IL_000c:  ldarg.0
IL_000d:  clt
IL_000f:  stloc.1
IL_0010:  ldloc.1
IL_0011:  brtrue.s IL_0005
IL_0013:  ret

Boogie

var i: int, CS$4$000: bool;
var $stack0i, $stack1i: int, $stack0b: bool;
IL_0000:
    $stack0i := 0;
i := 0;
goto IL_000b;
IL_0005:
    $stack1i := i;
    $stack0i := $stack0i + $stack1i;
i := $stack0i;
IL_000b:
    $stack0i := i;
    $stack1i := n;
    $stack0b := $stack0i < $stack1i;
    CS$4$000 := $stack0b;
    $stack0b := CS$4$000;
    if ($stack0b) { goto IL_0005; }
IL_0013:
    return;
Reasoning about loops

Java + JML
//@ requires 0 <= n;
void m(int n)
{
    int i = 0;
    //@ loop_invariant i <= n;
    while (i < n) {
        i++;
    }
    //@ assert i == n;
}

Boogie
procedure m(n: int)
    requires 0 <= n;
{
    var i: int;
    i := 0;
    while (i < n)
        invariant i <= n;
    {
        i := i + 1;
    }
    assert i == n;
}
Custom operators: underspecification

**C++**

```c++
void P() {
    int x;
    x = y << z;
    x = y + z;
}
```

**Boogie**

```boogie
const Two^31: int;
axiom Two^31 == 2147483648;

function LeftShift(int, int): int;
axiom (forall a: int ::
    LeftShift(a, 0) == a);

function Add(int, int): int;
axiom (forall a, b: int ::
    -Two^31 <= a+b && a+b < Two^31 ==>
    Add(a,b) == a+b);

procedure P() {
    var x: int;
    x := LeftShift(y, z);
    x := Add(y, z);
}
```
Definedness of expressions

**F#**

```fsharp
let x = y + z in
let w = y / z in
// ...
```

**Boogie**

```boogie
// check for underflow:
assert -Two^31 <= y+z;
// check for overflow:
assert y+z < Two^31;
x := y + z;

// check division by zero:
assert z != 0;
w := Div(y, z);
```
Uninitialized variables

Pascal

```pascal
var r: integer;
if B then
  r := z;
(* ... *)
if C then begin
  d := r
end
```

Boogie

```boogie
var r: int;
var r$defined: bool;

if (B) {
  r, r$defined := z, true;
}
// ...
if (C) {
  assert r$defined;
  d := r;
}
```
Loop termination

**Eiffel**

```
from
  Init
until
  B
invariant
  Inv
variant
  VF
loop
  Body
end
```

**Boogie**

```
Init;
while (!B)
  invariant Inv;
  // check boundedness:
  invariant 0 <= VF;
  {
    tmp := VF;
    Body;
    // check decrement:
    assert VF < tmp;
  }
```
Modeling memory

C#  

```csharp
class C {
    C next;
    void M(C c) {
        C x = next;
        c.next = c;
    }
}
```

Boogie  

```boogie

type Ref;
const null: Ref;

type Field;
const unique C.next: Field;

var Heap: [Ref,Field]Ref;
   // Ref * Field --> Ref

procedure C.M(this: Ref, c: Ref)
    requires this != null;
    modifies Heap;
    {
        var x: Ref;

        assert this != null;
        x := Heap[this, C.next];

        assert c != null;
        Heap[c, C.next] := y;
    }
```
More about memory models

Encoding a good memory model requires more effort

Boogie provides many useful features
  Polymorphic map types
  Partial commands (assume statements)
  Free pre- and postconditions
  where clauses
Demo

RingBuffer translated
Verification-condition generation

0. passive features: `assert`, `assume`, `;`
1. control flow: `goto` (no loops)
2. state changes: `:=`, `havoc`
3. loops
Weakest preconditions

The *weakest precondition* of a statement $S$ with respect to a predicate $Q$ on the post-state of $S$, denoted $wp(S,Q)$, is the set of pre-states from which execution:

- does not go wrong, and
- if it terminates, terminates in $Q$
VC generation: passive features

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

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

\[
wp( S; T, Q) = wp( S, wp( T, Q))
\]
VC generation: acyclic control flow

For each block A, introduce a variable $A_{ok}$ with the meaning: $A_{ok}$ is true iff every program execution starting in the current state from block A does not go wrong.

The verification condition for the program:

A: $S$; goto B or C

... is:

$$(A_{ok} \equiv \text{wp}(S, B_{ok} \land C_{ok})) \land$$

...$

$\Rightarrow$

$A_{ok}$
VC generation: state changes

Replace definitions and uses of variables by definitions and uses of different *incarnations* of the variables

\[
\{ x \mapsto x_0, \ y \mapsto y_0 \} \quad x := E(x,y)
\]

\[
\quad x_1 := E(x_0,y_0) \quad \{ x \mapsto x_1, \ y \mapsto y_0 \}
\]

\[
\{ x \mapsto x_0, \ y \mapsto y_0 \} \quad \text{havoc } x
\]

\[
\quad \text{skip} \; \{ x \mapsto x_1, \ y \mapsto y_0 \}
\]
VC generation: state changes (cont.)

Given:
\[
\{x \rightarrow x_0, y \rightarrow y_0\} S \\
{\{x \rightarrow x_0, y \rightarrow y_0\} T}
\]

then we have:
\[
{\{x \rightarrow x_0, y \rightarrow y_0\} \text{ if } E(x,y) \text{ then } S \text{ else } T \text{ end}}
\]

\[
\text{if } E(x_0,y_0) \text{ then} \\
S' ; \ x_3 := x_1
\]

\[
\text{else} \\
T' ; \ x_3 := x_2
\]

end
\[
{\{x \rightarrow x_3, y \rightarrow y_0\}}
\]
VC generation: state changes (cont.)

Replace every assignment

\[ x := E \]

with

\[ \text{assume } x = E \]
VC generation: loops

loop head:

assert LoopInv(x);

loop body:

assume Guard(x);
x := ...

after loop:

assume ¬Guard(x);
VC generation: loops

assert \( P \)

assert \( P \)

assert \( P \); assume \( P \)

assert LoopInv( x );
assume LoopInv( x );

assert \( \neg \text{Guard}(x) \);

assume \( \text{Guard}(x) \);
\( x := \ldots \)
VC generation: loops

assert \text{LoopInv}(x) ;

loop head:
assert \text{LoopInv}(x) ;
assume \text{LoopInv}(x) ;

after loop:
assume \neg\text{Guard}(x) ;

loop body:
assume \text{Guard}(x) ;
x := ...
assert \text{LoopInv}(x) ;
VC generation: loops

assert LoopInv(x);

loop head:

havoc x;
assume LoopInv(x);

loop body:
assume Guard(x);
x := ...;
assert LoopInv(x);

after loop:
assume ¬Guard(x);

loop target
VC generation: loops

assert LoopInv(x);

loop head: havoc x;
assume LoopInv(x);

after loop: assume Guard(x);

loop body: x := ...
assert LoopInv(x);
assume false;
Demo

/traceverify
Take-home messages

To build a verifier, use an intermediate verification language (IVL)
   An IVL is a thinking tool
   An IVL helps you separate concerns
   IVL lets you reuse and share infrastructure

Try Dafny and Boogie in your browser at rise4fun.com
Watch Verification Corner on YouTube