The Boogie Intermediate Language
What is BoogiePL?

• A simplified C-like language that’s structured for verification tasks
• Has constructs that allow specification of assumptions and axioms, as well as assertions that can be true or false depending on program state
• Has a front-end (confusingly also named Boogie) to the z3 solver which allows verifying assertions or producing counterexamples
• Has a framework for easily parsing, analyzing and generating BoogiePL programs
C -> BoogiePL -> Proof

∀ assertion ∈ prog.bpl: assertion = true ∨ counter = example

python smackgen.py prog.c
Boogie.exe prog.bpl /printModel:1 /useArrayTheory /errorLimit:100
Disclaimer

• All of these tools are *experimental*

• You can and will encounter bugs, unfriendly output and compatibility issues while installing & working with them.

• Part of the homework assignments is to overcome these challenges
  • Please consult the manual before contacting course staff 😊
Example: Boogie Procedure

```boogie
procedure polynum(x: int) returns (y: int);

implementation polynum(x: int) returns (y: int) {
    var a0: int, a1: int, a2: int;
    a0 := 3;
    a1 := 7;
    a2 := 0;
    y := a0 + a1*x + a2*x*x;
    assert y > 0;
    return;
}
```

Integers are mathematical integers (unbound)

Variable names can have stuff like ‘$’ and ‘#’, especially in smack generated code.

Assert statements apply to the program location they are specified in
Branching in Boogie

```c
if (volume > 50) {
    sad_neighbors = true;
}
```

```c
procedure karaoke(volume:int) returns (sad_neighbors:bool);

implementation karaoke(volume:int) returns (sad_neighbors:bool) {
    goto L1,L2;
L1:
    assume volume > 50;
    sad_neighbors := true;
    goto L3;
L2:
    assume !(volume > 50);
    goto L3;
L3:
    assert sad_neighbors <=> volume > 50;
    return;
}
```

Contains all C-like logical expressions (&&, ||, etc.) and some more..

Will it verify?
procedure karaoke(volume: int) returns (sad_neighbors: bool);

implementation karaoke(volume: int) returns (sad_neighbors: bool) {
    sad_neighbors := false;
goto L1,L2;
L1:
    assume volume > 50;
    sad_neighbors := true;
goto L3;
L2:
    assume !(volume > 50);
goto L3;
L3:
    assert sad_neighbors <=> volume > 50;
return;
}
Assume Statements

• assume statements can be used independently of branches, to direct the solver at any program point

```plaintext
procedure polynum(x: int) returns (y: int);

implementation polynum(x: int) returns (y: int) {
    var a0:int, a1:int, a2:int;
    a0 := 3;
    a1 := 7;
    a2 := 0;
    assume x > 0;
    y := a0 + a1*x + a2*x*x;
    assert y > 0;
    return;
}
```
Assume Statements

• Important: if an assumption *contradicts* what is known at that point, the path will be considered unfeasible, and the solver will not try and prove the assertions on that path!
  • This is why we don’t lose information on branches – if a branch contradicts what is known at the point of branching, it will be unfeasible and not be considered
    • much like if (0) { ... }

```plaintext
procedure contradiction() returns ();

implementation contradiction() returns () {
    var a: int;
    a := 0;
    assume a > 0;
    assert false;
    return;
}
```

> Boogie.exe contradiction.bpl
Boogie program verifier finished with 1 verified, 0 errors
Loops in Boogie

```boogie
procedure karaoke(volume:int) returns (sad_neighbors:bool);
implementation karaoke(volume:int) returns (sad_neighbors:bool) {
    var new_volume: int;
    new_volume := volume;
    sad_neighbors := false;
    LOOP_HEAD:
        goto L1,L2;
    L1:
        assume new_volume > 50;
        sad_neighbors := true;
        new_volume := new_volume - 1;
        goto LOOP_HEAD;
    L2:
        assume !(new_volume > 50);
        goto L3;
    L3:
        assert new_volume <= 50;
        return;
}
```

Inputs are immutable.

In smack this will look slightly more complicated.
Functions in Boogie

• Functions can also be recursive!
Globals in Boogie

```boogie
var call_police : bool;

procedure karaoke(volume:int) returns ();
modifies call_police;

implementation karaoke(volume:int) returns () {
    goto L1,L2;
    L1:
        assume volume > 50;
        call_police := true;
        goto L3;
    L2:
        assume !(volume > 50);
        call_police := false;
        goto L3;
    L3:
        assert call_police <=> volume > 50;
        return;
}
```

Specifies all modified globals
Axioms in Boogie

• We can convey further properties to the solver

```plaintext
const zero: int;
const unique null: int;

axiom zero == 0;

function foo(x: int) : int;

axiom foo(0) == 0;

function goo(x: int, y: int) : int;

axiom (forall a: int, b: int :: a == b ==> foo(a) == foo(b));
```
Memory in Boogie

```boogie
var Mem: [int]int;

procedure setmem(ind1: int, ind2: int) returns ();
modifies Mem;

implementation setmem(ind1: int, ind2: int) returns () {
    Mem[ind1] := 1;
    Mem[ind2] := 2;
    assert Mem[ind1] == 1 && Mem[ind2] == 2;
    return;
}
```

This is how smack models memory and heap.
More Boogie

• Very rich language, has support for unique values, type systems, structs, etc.

• See more at:
Translating C to Boogie

• Get the **smack** tool from [https://github.com/smackers/smack](https://github.com/smackers/smack)
  • Has a 1-liner install for Ubuntu
  • My VM will also be provided (but may disagree with your hardware 😞)

• Actual translation after install
  • `smackgen.py test_smack.c`
Next week (after the lecture)

- How to manipulate and analyze BoogiePL programs with the Boogie framework