Developing Verified Programs with Boogie and Boogaloo

Nadia Polikarpova

Software Verification

October 16, 2013

Overview

What is Boogie?

The Language: how to express your intention?

- Imperative constructs
- Specification constructs

The Tool: how to get it to verify?

- Debugging techniques
- Boogaloo to the rescue

Overview

What is Boogie?

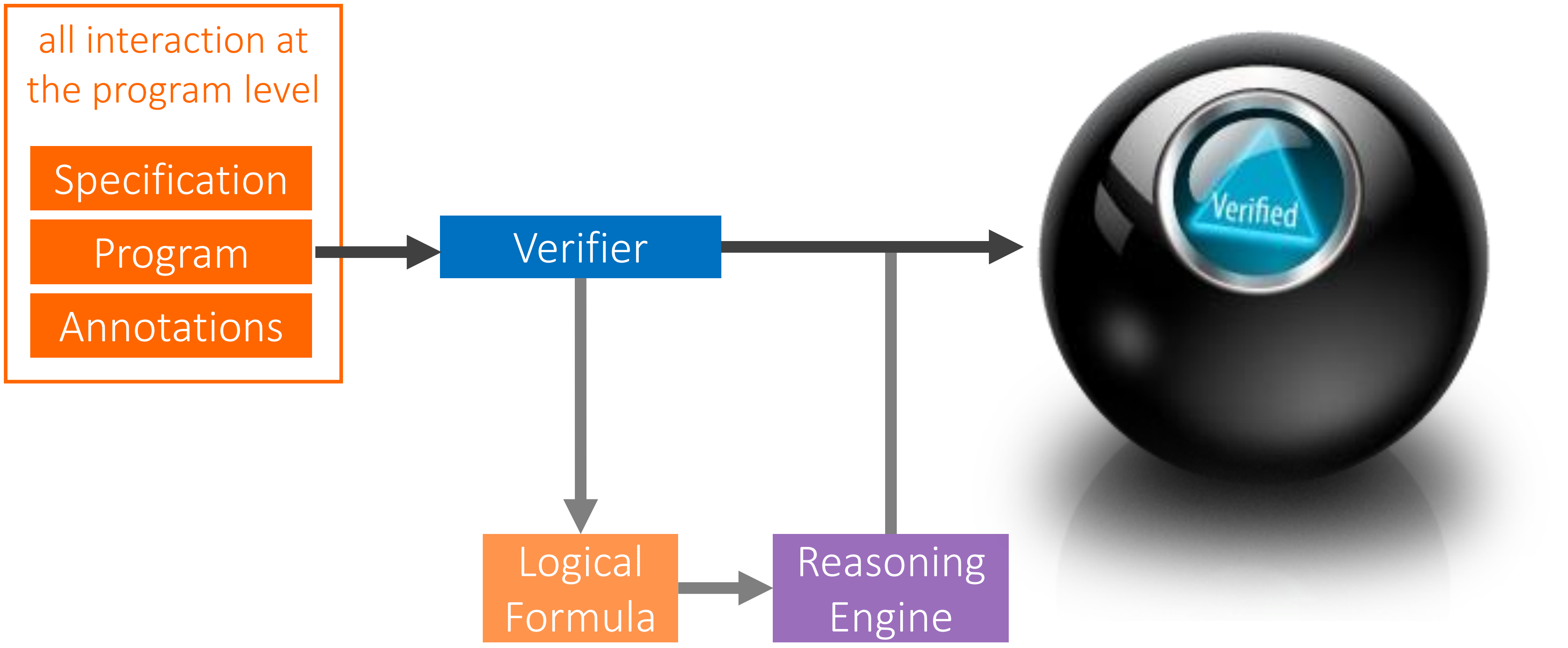
The Language

- Imperative constructs
- Specification constructs

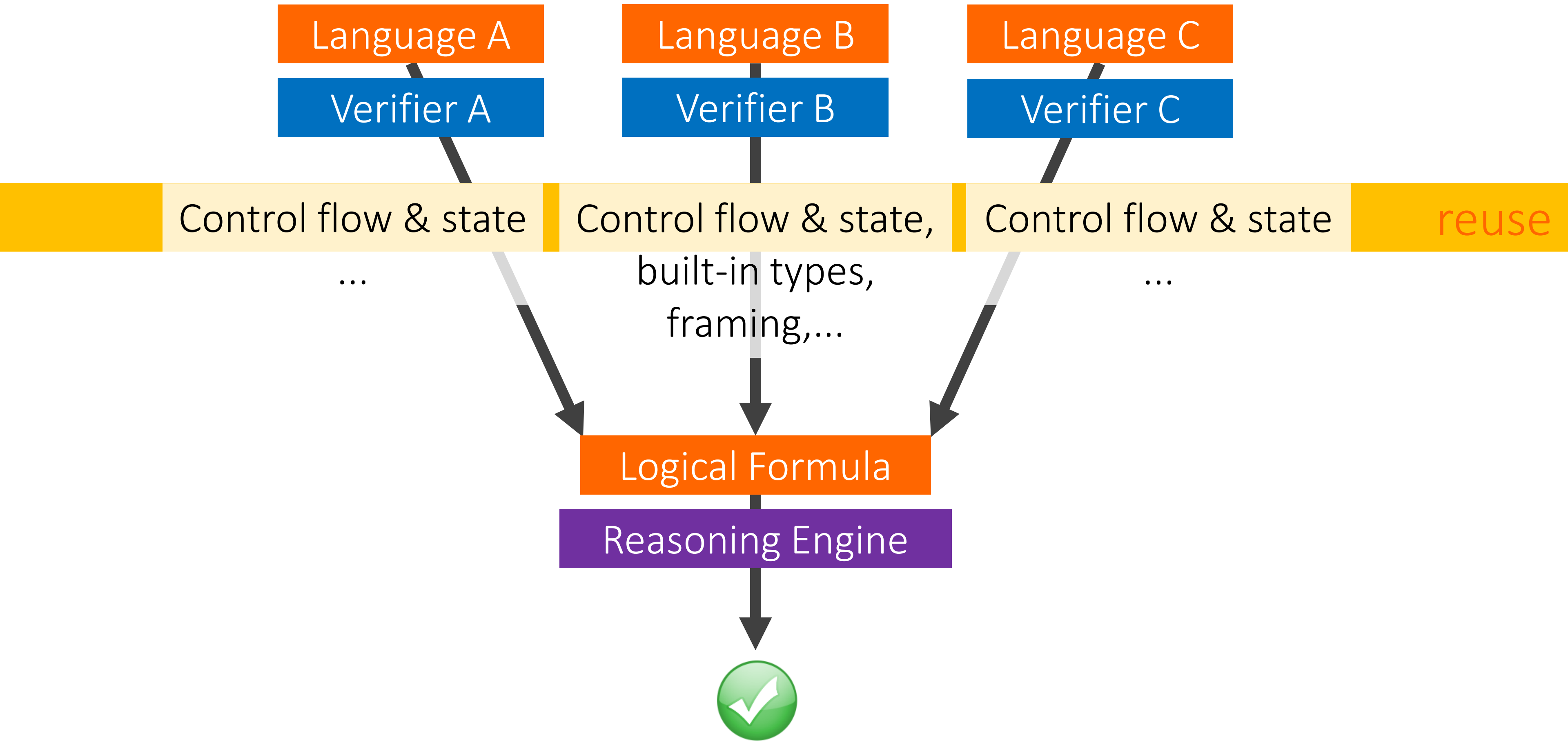
The Tool

- Debugging techniques
- Boogaloo to the rescue

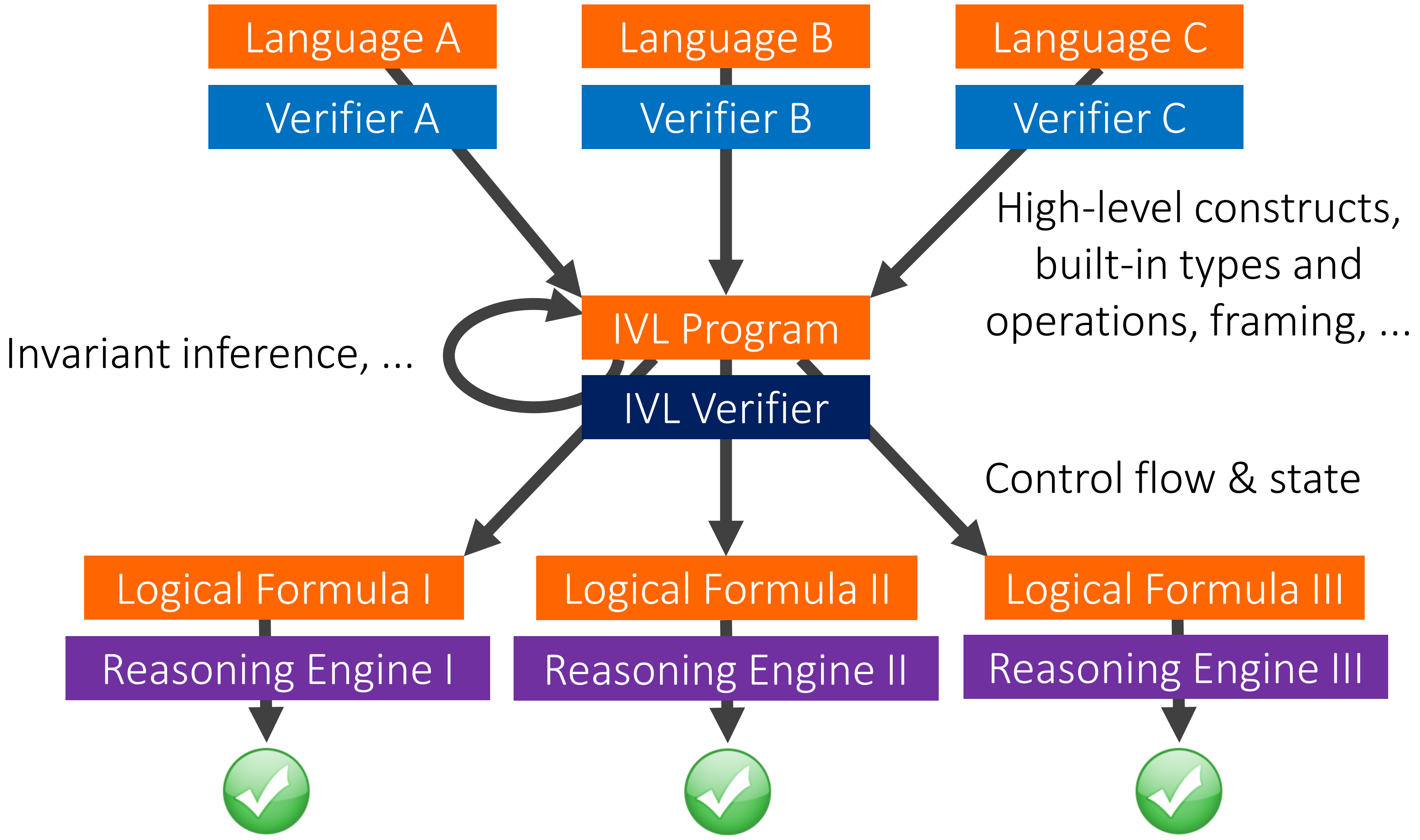
“Auto-active” verification



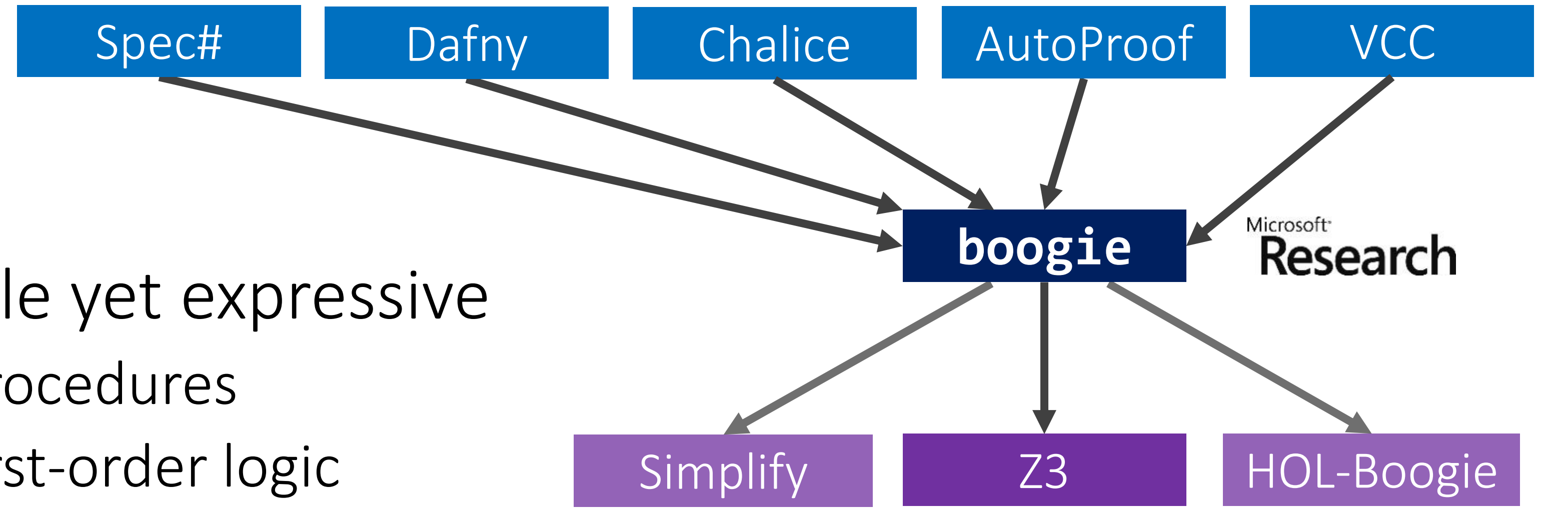
Verifying imperative programs



Intermediate Verification Language



The Boogie IVL



Simple yet expressive

- procedures
- first-order logic
- integer arithmetic

Great for teaching verification!

- skills transferable to other auto-active tools

Alternative: Why [http://why3.lri.fr/]

Getting started with Boogie

boogie Microsoft Research

Try online [rise4fun.com/Boogie]

Download [boogie.codeplex.com]

User manual [Leino: This is Boogie 2]

Hello, world?

Overview

What is Boogie?

The Language

- Imperative constructs
- Specification constructs

The Tool

- Debugging techniques
- Boogaloo to the rescue

Types

definition

usage

Booleans: `bool`

Mathematical integers: `int`

User-defined: **type** Name t_1, ..., t_n;

```
type ref; // references
type Person;
type Field t; // fields with values of type t
Field int
Field ref
```

Maps: [dom_1,...,dom_n]range

```
[int]int // array of int
[Person]bool // set of persons
[ref]ref // "next" field of a linked list
<t>[ref, Field t]t // generic heap
```

Synonyms: **type** Name t_1, ..., t_n = *type*;

```
type Array t = [int]t;
type HeapType = <t>[ref, Field t]t;
```

Imperative constructs

Regular procedural programming language

[Absolute Value & Fibonacci]

... and non-determinism

great to simplify and over-approximate behavior

```
havoc x; // assign an arbitrary value to x
```

```
if (*) { // choose one of the branches non-deterministically
    statements
} else {
    statements
}
```

```
while (*) { // loop some number of iterations
    statements
}
```

Specification statements: `assert`

`assert` **`e`**: executions in which **`e`** evaluates to **`false`** at this point are bad

- expressions in Boogie are pure, no procedure calls

Uses

- explaining semantics of other specification constructs
- encoding requirements embedded in the source language

```
assert lo <= i && i < hi; // bounds check
result := array[i];
```

```
assert this != null; // O-O void target check
call M(this);
```

- debugging verification (see later)

[Absolute Value]

Specification statements: `assume`

`assume` `e`: executions in which `e` evaluates to **`false`** at this point are impossible

```
havoc x; assume x*x == 169; // assign such that
```

```
assume true; // skip
```

```
assume false; // this branch is dead
```

Uses

- explaining semantics of other specification constructs
- encoding properties guaranteed by the source language

```
havoc Heap; assume NoDangling(Heap); // managed language
```

- debugging verification (see later)

Assumptions are dangerous! [Absolute Value]

Loop invariants

```
before_statements;
while (c)
    invariant inv;
{
    body;
}
after_statements;
```

=

```
before_statements;
assert inv;

havoc all_vars;
assume inv && c;
body;
assert inv;

havoc all_vars;
assume inv && !c;
after_statements;
```

The only thing the verifier know about a loop

simple invariants can be inferred

[Fibonacci]

Procedure contracts

```
procedure P(ins) returns (outs)
    free requires pre';
    requires pre;
    modifies vars; // global
    ensures post;
    free ensures post';
{ body; }
```

=

```
assume pre && pre';
body;
assert post;
```

```
call outs := P (ins);
```

=

```
assert pre;
havoc outs, vars;
assume post && post';
```

The only thing the verifier knows about a call

this is called modular verification

[Abs and Fibonacci]

Enhancing specifications

How do we express more complex specifications?

e.g. `ComputeFib` actually computes Fibonacci numbers

Uninterpreted functions

```
function fib(n: int): int;
```

Define their meaning using axioms

```
axiom fib(0) == 0 && fib(1) == 1;
axiom (forall n: int :: n >= 2 ==> fib(n) == fib(n-2) + fib(n-1));
```

[Fibonacci]

Overview

What is Boogie?

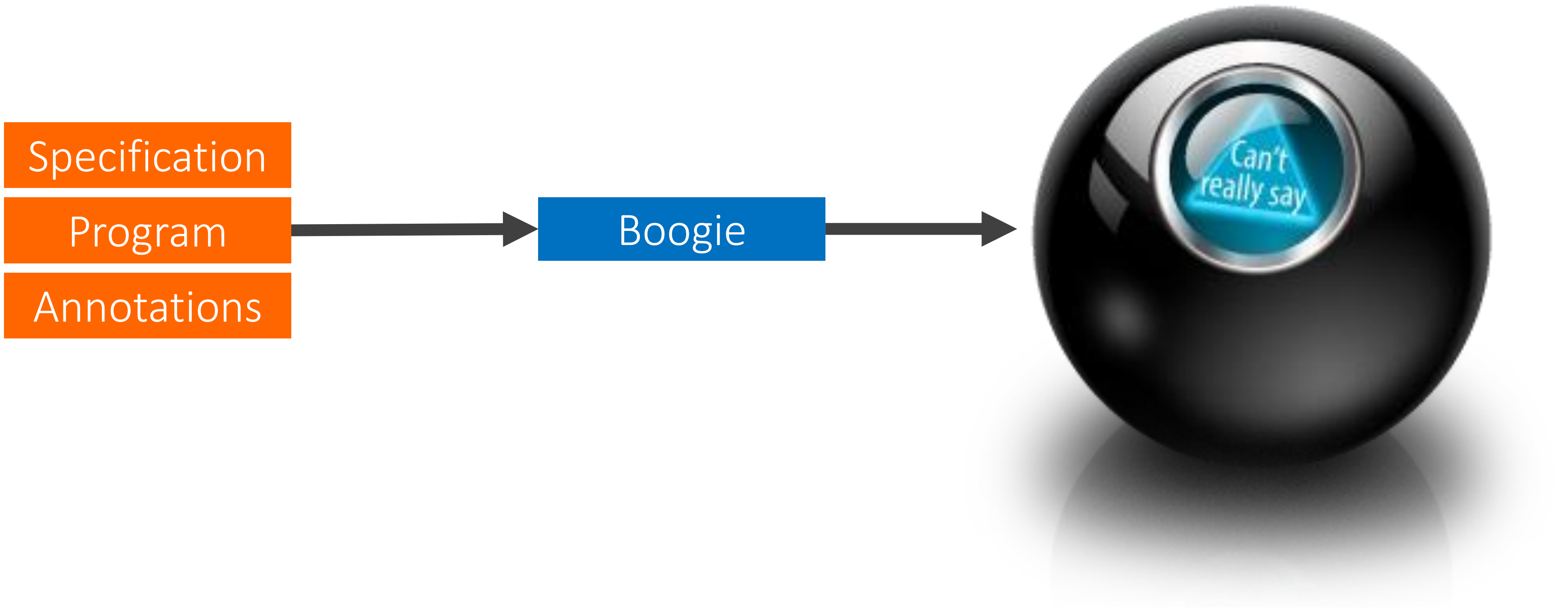
The Language

- Imperative constructs
- Specification constructs

The Tool

- Debugging techniques
- Boogaloo to the rescue

What went wrong?



Debugging techniques

Proceed in small steps [Swap]

  use **`assert`** statements to figure out what Boogie knows

Divide and conquer the paths

  use **`assume`** statements to focus on a subset of executions

Prove a lemma [Non-negative Fibonacci]

  write ghost code to help Boogie reason

Look at a concrete failing test case [Array Max]

  Boogaloo to the rescue!

Getting started with Boogaloo

Try online [cloudstudio.ethz.ch/comcom/#Boogaloo]

Download [bitbucket.org/nadiapolikarpova/boogaloo]

User manual
[bitbucket.org/nadiapolikarpova/boogaloo/wiki/User_Manual]

Features

Print directives

```
assume {: print “hello, world”, x + y } true;
```

[Array Max, print the loop counter]

Bound on loop iterations

```
--loop-max=N
```

```
-l=N
```

N = 1000 by default

[Array Max, comment out loop counter increment]

Conclusions

Boogie is an Intermediate Verification Language (IVL)

- IVLs help develop verifiers

The Boogie language consists of:

- imperative constructs ≈ Pascal
- specification constructs (**`assert`**, **`assume`**, **`requires`**, **`ensures`**, **`invariant`**)
- math-like part (functions + first-order axioms)

There are several techniques to debug a failed verification attempt

Boogaloo helps by generating concrete test cases

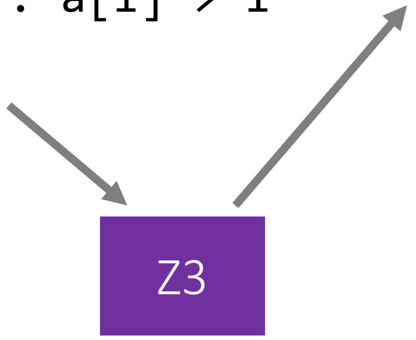
Backup slides

How it works: an Example

```
procedure Test(a: [int]int, x: int)
    requires (forall i: int :: a[i] > i);
{
    if (x == 1000) {
        assert a[x] > 1001;
    }
}
```

Path constraints

```
forall i: int :: a[i] > i
!(x == 1000)
a[x] > 1001
```



Valid executions

```
1: Test(a = [1000 -> 1001],
            x = 1000) failed
2: Test(a = [1000 -> 1002],
            x = 1000) passed
3: Test(a = [], x = 0) passed
```

Evaluation

verification

Program (LOC)	Correct		Buggy	
	N	T (sec)	N	T (sec)
Fibonacci (40)	20	6.4	0	0.0
TuringFactorial (37)	21	0.2	3	0.0
ArrayMax (33)	46	0.4	0	0.0
ArraySum (34)	46	0.3	1	0.0
BinarySearch (49)	46	0.0	0	0.1
DutchFlag (96)	20	3.8	1	0.0
Invert (37)	20	13.3	2	0.0
BubbleSort (74)	10	6.5	2	0.1
QuickSort (89)	10	2.0	2	0.1
QuickSortPartial (79)	10	16.7	2	0.1
ListTraversal (49)	20	2.5	2	0.0
ListInsert (52)	7	164.5	1	0.0
Split (22)	-	0.0		
SendMoreMoney (36)	-	0.3		
Primes (31)	8	0.2		
NQueens (37)	15	1.2		

fast

partial implementation

declarative