Software Verification (Autumn 2015)
Lecture 5: Auto-Active Verification

Chris Poskitt

(based on material by Nadia Polikarpova)

Chair of
Software Engineering

ETH zürich

This time last week

|- { x>0 } x := x+1; skip { x>1 }

$$
[\text{comp}]\ \dfrac{[\text{cons}]\ \dfrac{[\text{ass}]\ \dfrac{}{\vdash \{x+1>1\}\ \texttt{x := x + 1}\ \{x>1\}}}{\vdash \{x>0\}\ \texttt{x := x + 1}\ \{x>1\}} \qquad [\text{skip}]\ \dfrac{}{\vdash \{x>1\}\ \texttt{skip}\ \{x>1\}}}{\vdash \{x>0\}\ \texttt{x := x + 1; skip}\ \{x>1\}}
$$

Can we reason about {pre}P{post} mechanically?

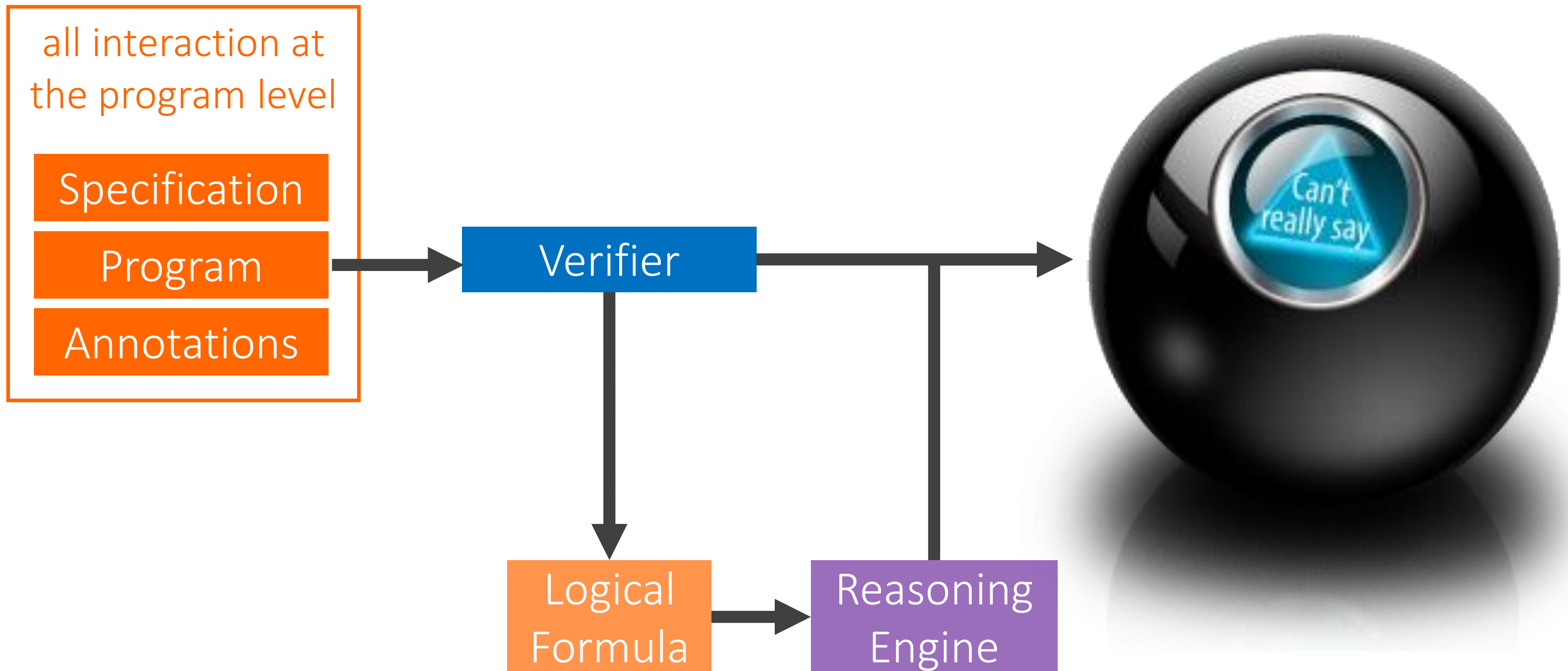
Verification problem undecidable in general

How far can we go? What challenges do we face?

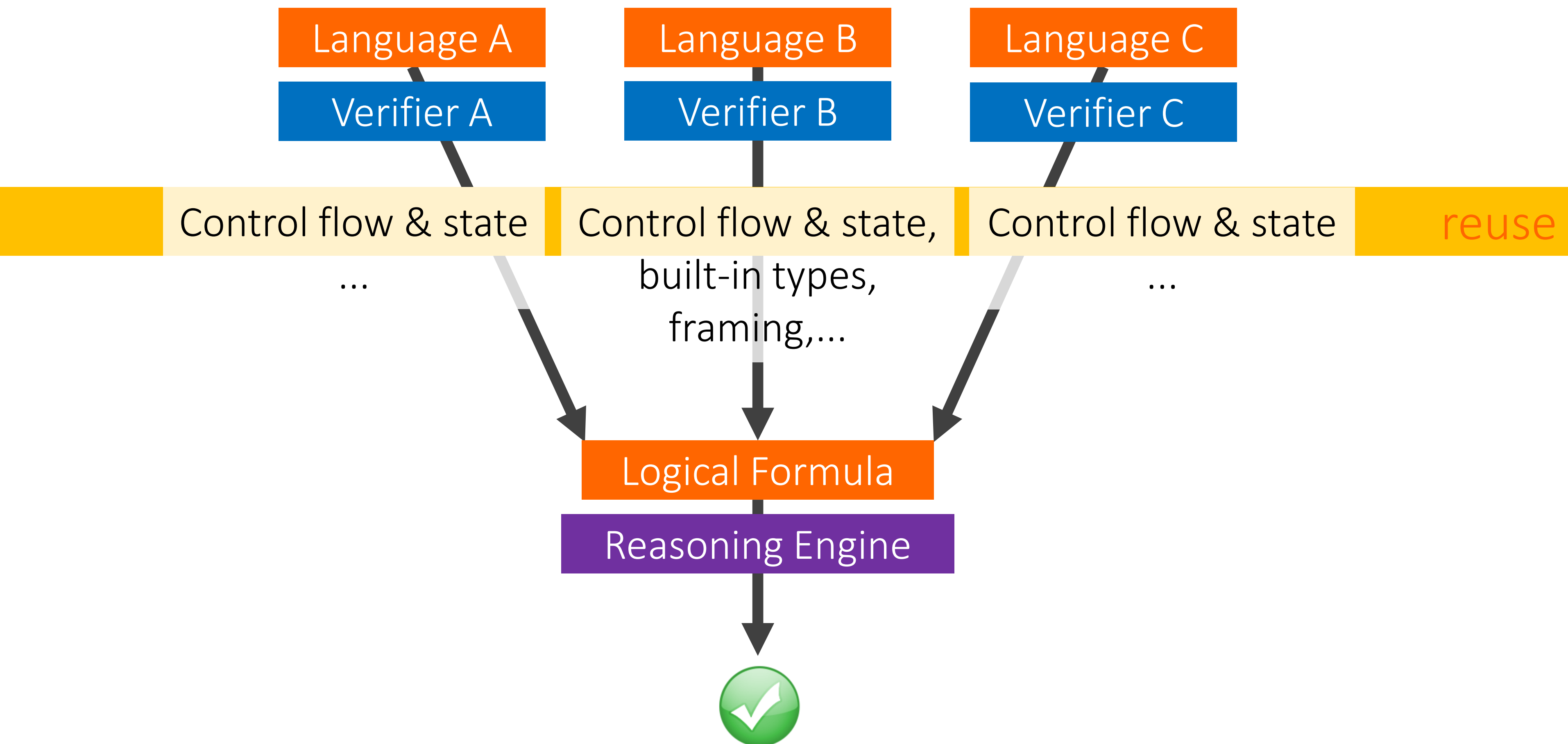
- Determining loop invariants
- Weak or missing assertions
- Undecidable assertion logics
- …

Idea: automate as much as possible, with users indirectly providing guidance through program-level annotations

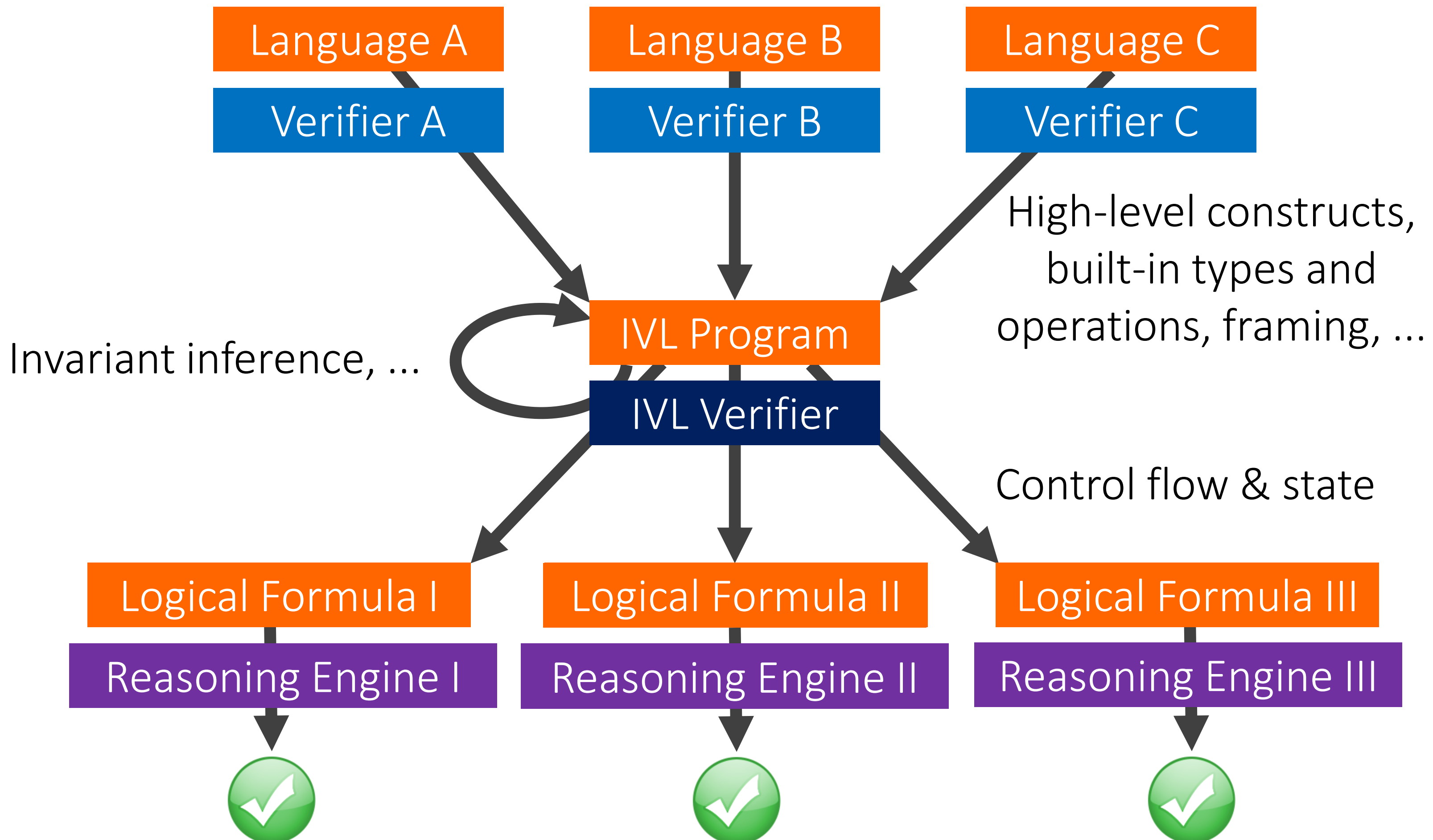
“Auto-active” verification



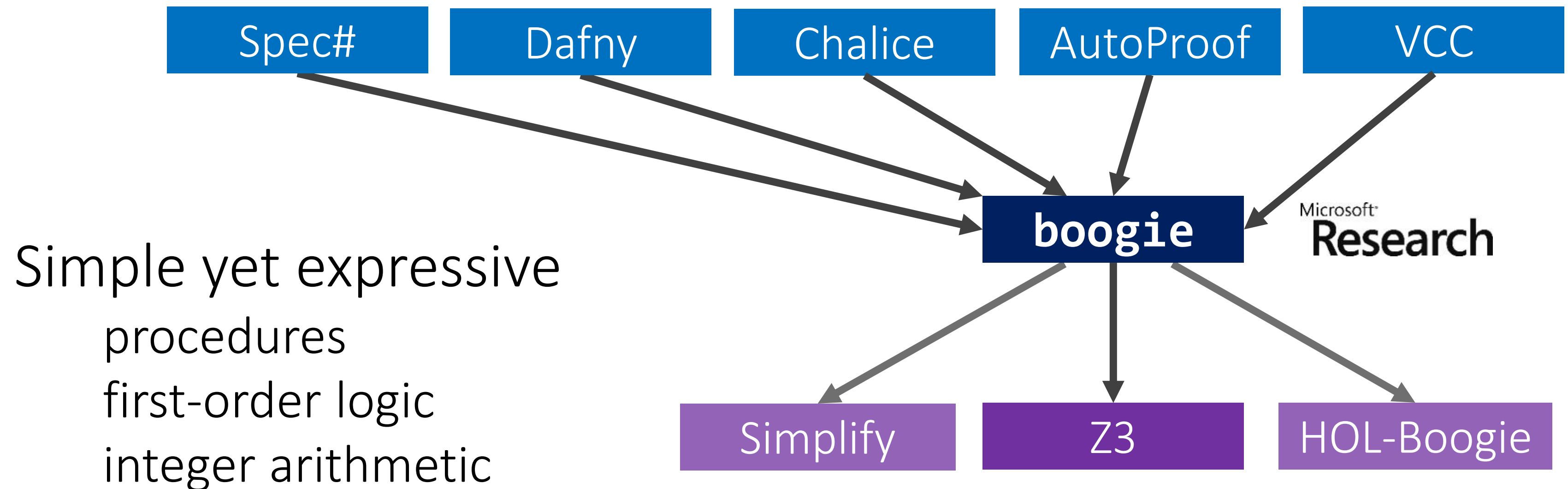
Verifying imperative programs



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

Alternatives: Why3 [http://why3.lri.fr/]
Viper [http://www.pm.inf.ethz.ch/]

Overview

The Boogie Language : how to express your intention?

- Imperative constructs
- Specification constructs

The Boogie Tool : how to get it to verify?

- Debugging techniques
- Boogaloo to the rescue

The AutoProof Verifier

Overview

The Boogie Language
- Imperative constructs
- Specification constructs

The Boogie Tool
- Debugging techniques
- Boogaloo to the rescue

The AutoProof Verifier

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?

Types

Basic types: **bool**, **int**, **real**

definition

usage

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: `<`t_1, ..., t_n`>[`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

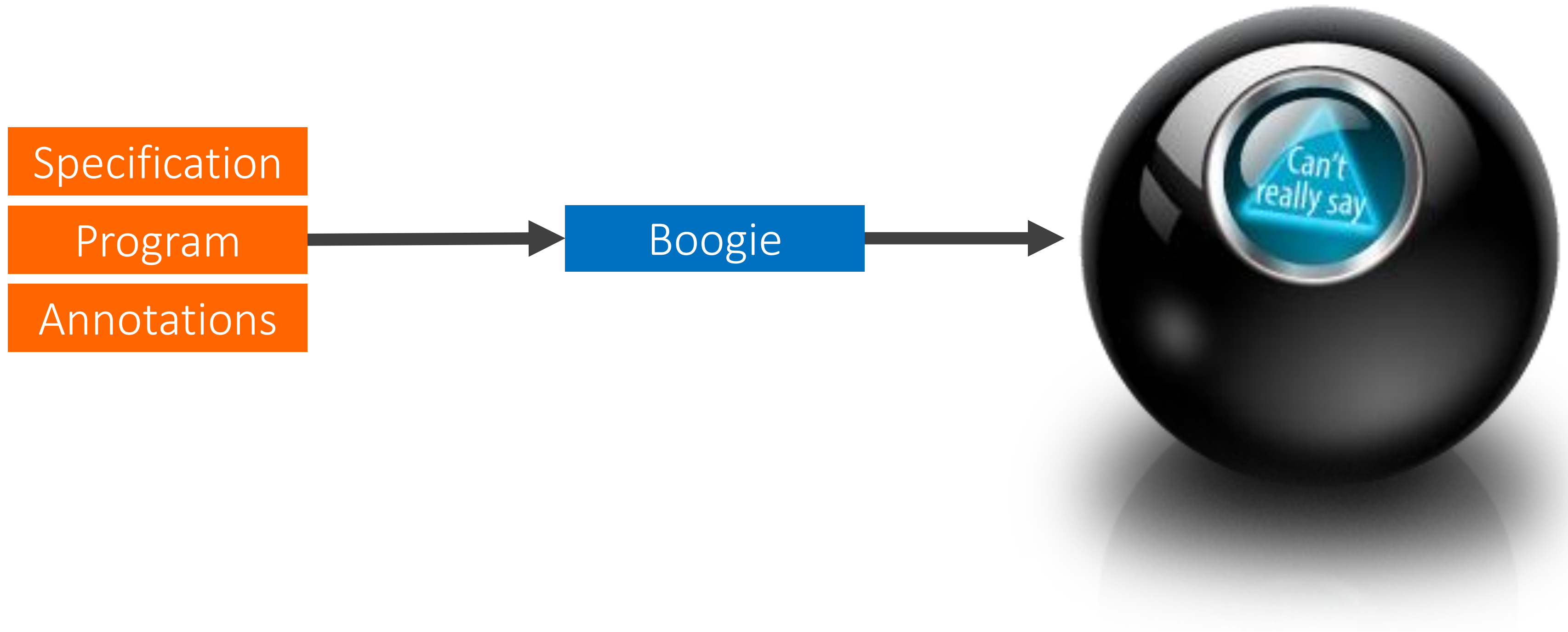
```
axiom fib(0) == 0 && fib(1) == 1;
axiom (forall n: int :: n >= 2 ==> fib(n) == fib(n-2) + fib(n-1));
```

[Fibonacci]

Overview

- The Boogie Language
  - Imperative constructs
  - Specification constructs
- **The Boogie Tool**
  - Debugging techniques
  - Boogaloo to the rescue
- The AutoProof Verifier

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

Overview

The Boogie Language

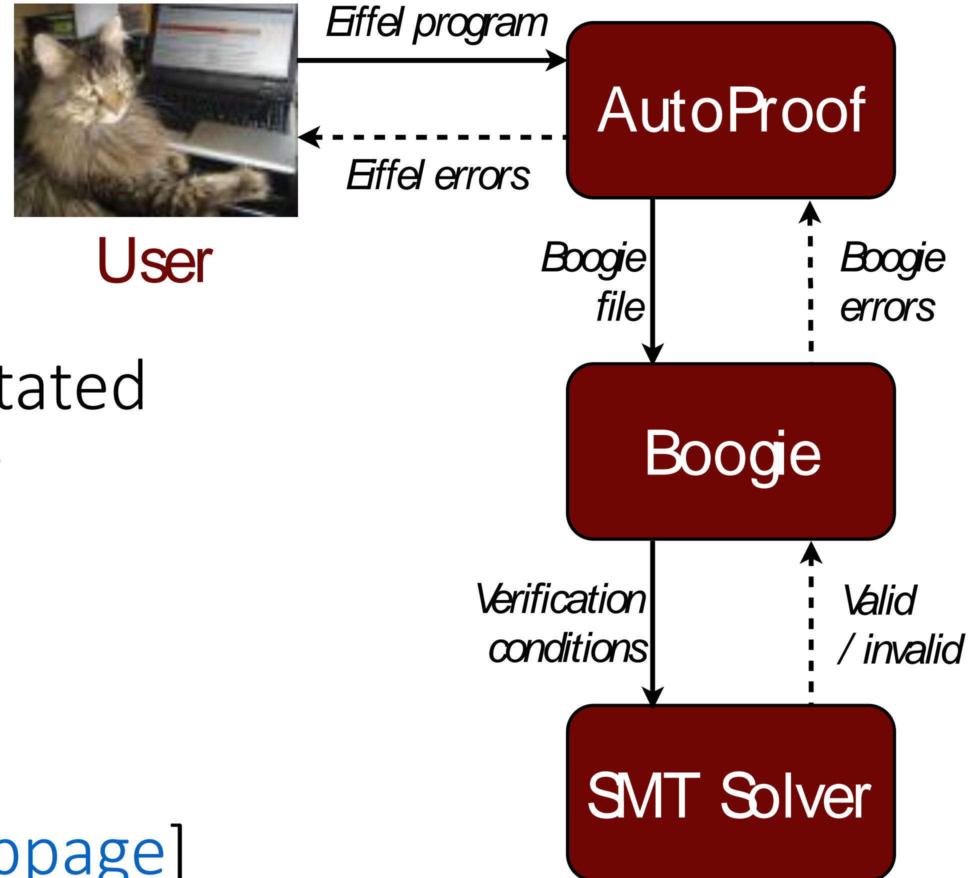
- Imperative constructs
- Specification constructs

The Boogie Tool

- Debugging techniques
- Boogaloo to the rescue

The AutoProof Verifier

AutoProof: a Boogie-based verifier for Eiffel



Translates contract-annotated Eiffel programs to Boogie

Try online [via Comcom]

Manual, tutorial, examples [AutoProof webpage]

How the translation works [Slides]

```
1  -- Simple bank account class.
2  -- Try to fix it and make the verification go through.
3
4  class
5      ACCOUNT
6
7  feature -- Access
8
9      balance: INTEGER
10             -- Balance of account.
11
12 feature -- Element change
13
14     deposit (amount: INTEGER)
15             -- Deposit `amount' on account.
16         require
17             amount_not_negative: amount >= 0
18         do
19             balance := balance + amount
20         ensure
```

Run...

Feature	Line	Result
ACCOUNT (invariant admissibility)		Successfully verified.
ANY.default_create (creator, inherited by ACCOUNT)		Successfully verified.
ACCOUNT.deposit		Successfully verified.
ACCOUNT.withdraw	31	Postcondition balance_decreased may be violated.
ACCOUNT.transfer	49	Postcondition balance_decreased may be violated.

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

AutoProof is one of several auto-active verifiers, based on translating annotated programs to Boogie