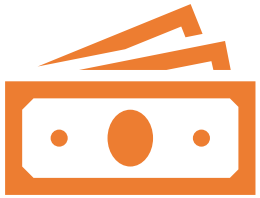
Automatic Formal Verification of Flint Smart Contracts

Ioannis Gabrielides

- Smart contract programming language
- Ethereum Blockchain
- Execute code via transactions
- Handle digital money: Ether / Wei

Why formal verification?



Smart contracts handle money

Smart contracts are immutable

Unit testing is not good enough

+

Formal Verification

(Boogie)

Functional Specifications

- Assert function

```
assert (a * b != 23447)
```

- Pre-conditions – New

```
pre (i == 2)
```

- Post conditions – New

```
post (returns 3)
```

Contract Invariants

```
invariant (a >= 0)
```

- Assumed at start of every function
- Must hold by end of every function
- Inspired by class invariants by Bertrand Meyer

Holistic Specifications

```
will (i == 10)
```

- Will Predicate
  - S. Drossopoulou, S. Eisenbach and J. Noble
  - Over multiple transactions
  - Eventually holds

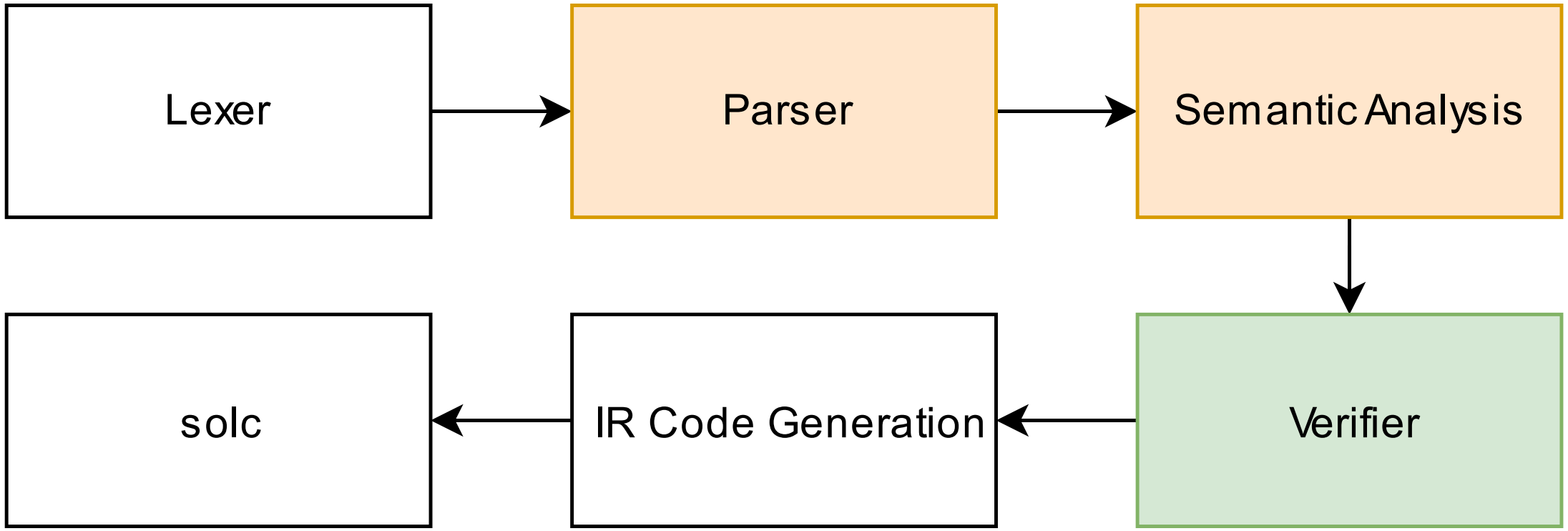
Other specification predicates

- **`prev`**
- **`dictContains`**
- **`arrayContains`**
- **`returns`**
- **`returning`**
- **`arrayEach`**

Detection of common errors

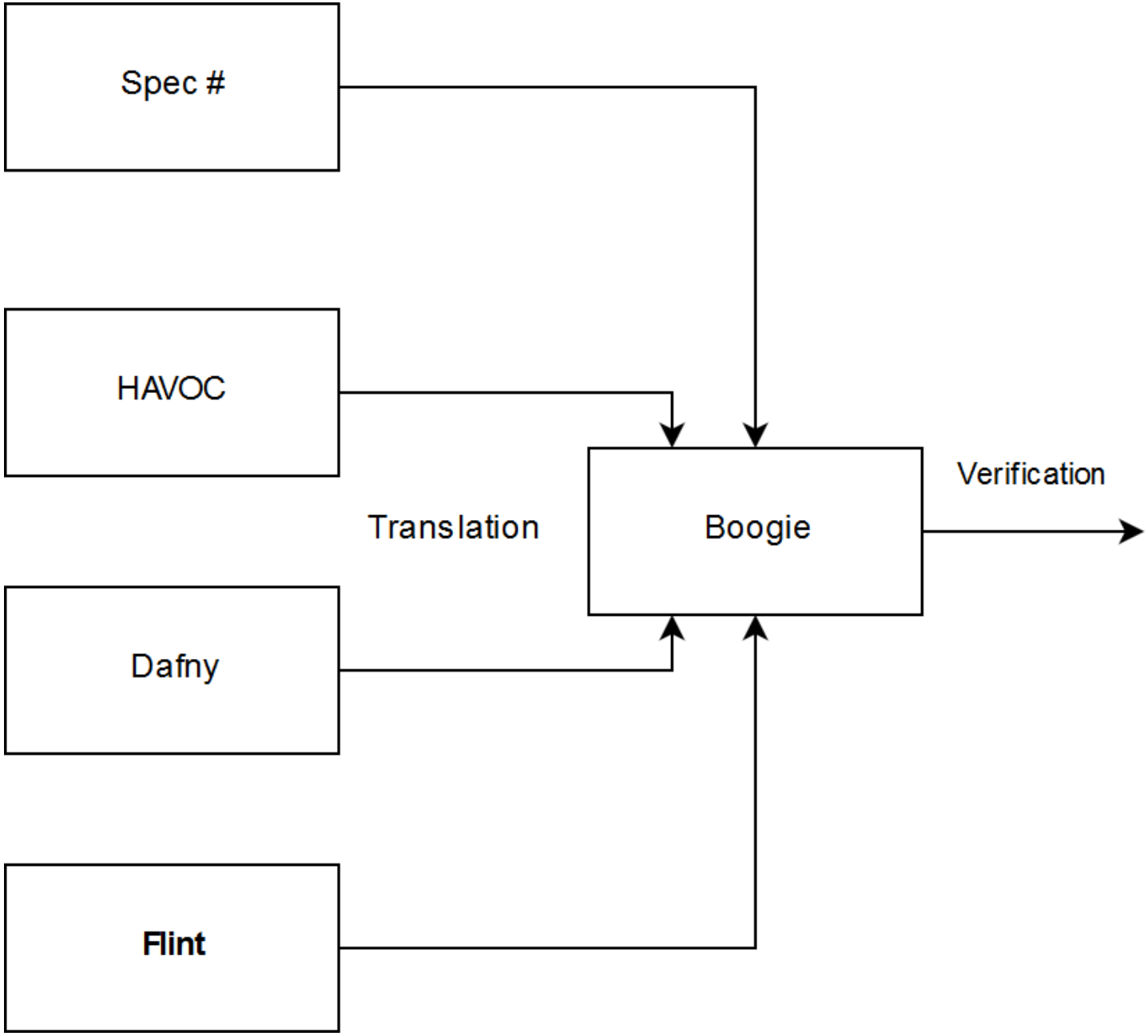
- Array out-of-bounds access
- Unreachable code
- Division by zero
- Inconsistent pre-conditions

Compiler Architecture



Boogie

- Intermediate Verification Language
- Automatic Formal Verification System



```
procedure factorial_Factorial(n: int) returns (result: int)
{
  var factorial_Factorial_result: int;
  if (n < 2) {
    result := 1;
    return;
  }

  call factorial_Factorial_result := factorial_Factorial(n - 1);
  result := n * factorial_Factorial_result;
}
```

Translating Flint code to Boogie

```
contract Factorial {
  var value: Int = 0
}

Factorial :: (any) {
  public init() {}

  func factorial(n: Int) -> Int {
    if (n < 2) { return 1 }
    return n * factorial(n: n - 1)
  }
```

```
var value: int;

procedure init_Factorial()
  modifies value;
{
  value := 0;
}

procedure factorial_Factorial(n: int) returns (result: int)
{
  var factorial_Factorial_result: int;
  if (n < 2) {
    result := 1;
    return;
  }

  call factorial_Factorial_result := factorial_Factorial(n - 1);
  result := n * factorial_Factorial_result;
}
```

Pre and Post-Conditions, Assertions

```
func factor(a: Int, b: Int)
  pre (1 < a && a < 500)
  pre (1 < b && b < 25000)
{
  assert ((a * b) != 23449)
}
```

```
procedure factor(a: int, b: int)
  requires (1 < a && a < 500);
  requires (1 < b && b < 25000);
{
  assert (!(a * b) == 23449);
}
```

Contract Invariants

```
invariant (a >= 0)
```

```
procedure f()
  requires (a >= 0);
  ensures(a >= 0);
```

Will

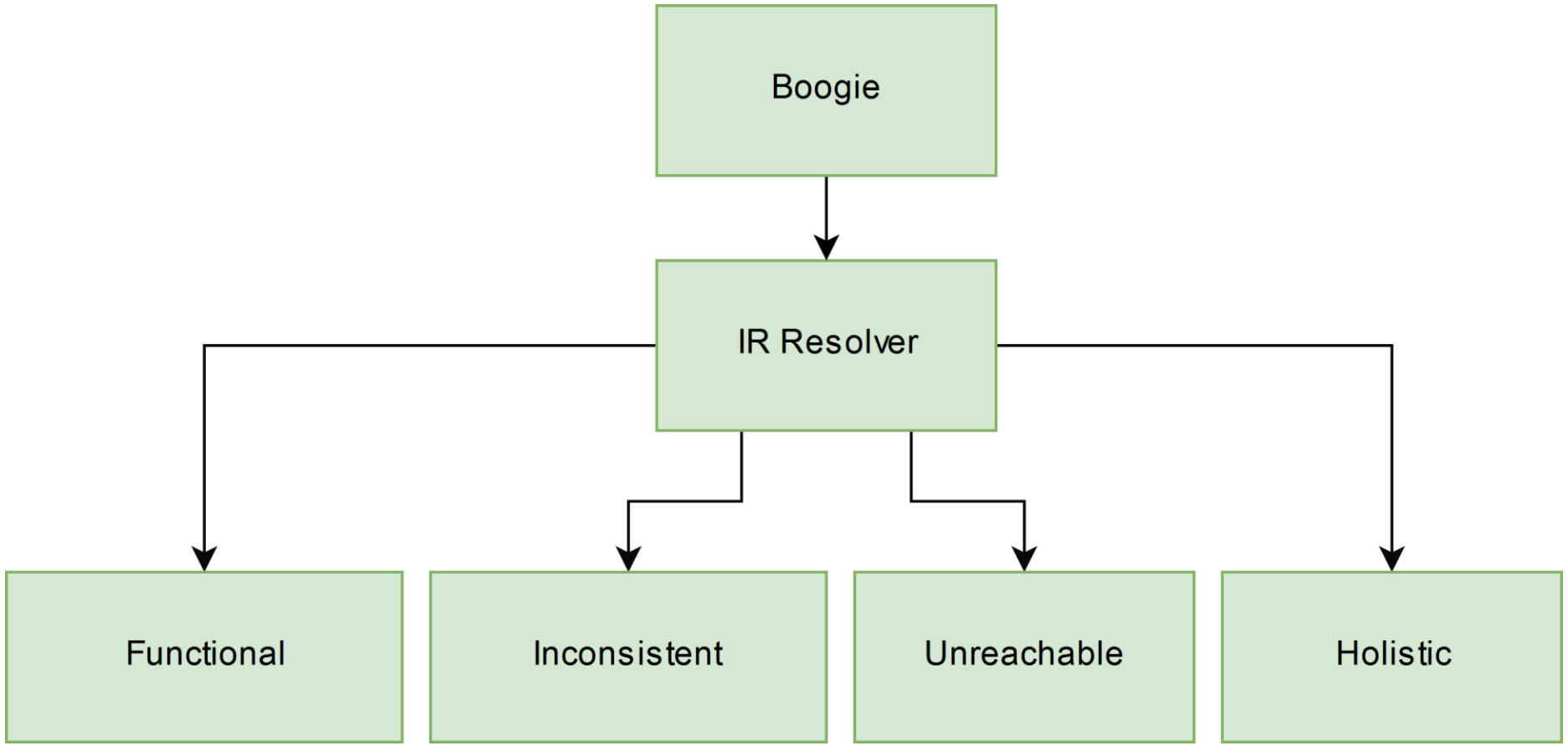
- Translate the Flint code into Boogie as normal
- Encode the will operator into a Boogie representation
- Emulate multiple transactions
- Using symbolic execution

Verification

```
Boogie program verifier version 2.3.0.61016, Copyright (c) 2003-2014,
    Microsoft.
program.bpl(417,1): Error BP5002: A precondition for this call might not
    hold.
program.bpl(345,1): Related location: This is the precondition that might
    not hold.
Execution trace:
    program.bpl(406,1): anon0
    program.bpl(415,1): anon6_Then
program.bpl(534,1): Error BP5001: This assertion might not hold.
Execution trace:
    program.bpl(508,7): anon0
    program.bpl(512,1): anon6_LoopHead
    program.bpl(512,1): anon6_LoopDone
    program.bpl(534,1): anon5

Boogie program verifier finished with 12 verified, 2 errors
```

Verifier Architecture



Challenges

- Designing the specification language
- Designing the translation from Flint to Boogie
- Implementing the translation in an extensible way

To Wrap Up

- Introduced Specification syntax
- Automatic verification
- Detect common bugs

- Translation not completely semantic preserving
- Informal proof of correctness, of Boogie translation
- Limited functional and holistic predicates

Questions?