Collaborative Verification and Testing

Peter Müller
ETH Zurich

Joint work with
Maria Christakis and Valentin Wüstholz

Static Program Checkers

- Many practical static analyzers and verifiers sacrifice soundness:

  to improve automation

  to improve performance

  to increase precision

Typical Compromises

Assuming write effects

- Modifies clauses in ESC/Java, HAVOC
- Pure methods in Clousot

Unbounded arithmetic

- Clousot, ESC/Java, Spec#

Loop unrolling

- Stratified inlining in Corral (Poirot)
- 1½ iterations in ESC/Java

Heap abstraction

- Method prestates in Clousot
- Pointer arithmetic in points-to analyses

Running Example

```
class Cell {
  int v;

  static int M( Cell c, Cell d )
    requires c != null ∧ d != null;
    requires c.v != 0 ∧ d.v != 0;
    ensures result < 0;
  {
    if ( sign(c.v) == sign(d.v) )
      c.v = (–1) * c.v;

    return c.v * d.v;
  }
}
```

Error due to aliasing

Error due to overflow

Running Example in Clousot

```
class Cell {
  int v;

  static int M( Cell c, Cell d )
    requires c != null ∧ d != null;
    requires c.v != 0 ∧ d.v != 0;
    ensures result < 0;
  {
    if ( sign(c.v) == sign(d.v) )
      c.v = (–1) * c.v;

    return c.v * d.v;
  }
}
```

Unsound Static Checking

Problems

- Errors may be missed
- Unclear what remains to be tested

```
class Cell {
  int v;

  static int M( Cell c, Cell d )
    requires c != null ∧ d != null;
    requires c.v != 0 ∧ d.v != 0;
    ensures result < 0;
  {
    if ( sign(c.v) == sign(d.v) )
      c.v = (–1) * c.v;

    return c.v * d.v;
  }
}
```

Approach

- Make compromises explicit
  - Properties not checked
  - Properties checked under assumptions
- Use information for tool integration

Explicit Assumptions

- At each program point where an unsound assumption *P* is made, we insert a clause

  assumed *P* **as** a;

  where *P* is a predicate and a is a fresh boolean variable (initialized to true)

- The semantics is an assignment

  $a := a \wedge P$

Explicit Assumptions: Examples

Unbounded arithmetic

```
assumed bounded( x * y ) == x * y as a;
v = x * y;
```

Loop unrolling

```
if ( c ) { S }
assumed ¬c as a;
while ( c ) { S }
```

Verification Results

- Each (explicit or implicit) assertion is decorated with a set of verification results

assert V P;

where each element of V is a set of assumption-variables

- The semantics uses an assumption

assume $\left(\bigvee_{A \in V} \left(\bigwedge_{a \in A} a\right)\right) \Rightarrow P$;

assert P;

Verification Results: Example

Assuming write effects

```
method Foo(…)
  modifies M;
{
  assert { verified A } ∀o,f :: (o,f)∈M ∨ o.f == old(o.f);
}
```

Running Example in Clousot Revisited

```
static int M( Cell c, Cell d )
  requires c != null ∧ d != null;
  requires c.v != 0 ∧ d.v != 0;
  ensures { verified { a1, a2, a3 } } result < 0;
{
  assumed c != d as a1;
  if ( sign(c.v) == sign(d.v) ) {
    assumed bounded((–1) * c.v) == (–1) * c.v as a2;
    c.v = (–1) * c.v;
  }
  assumed bounded(c.v * d.v) == c.v * d.v as a3;
  return c.v * d.v;
}
```

Using Verification Results

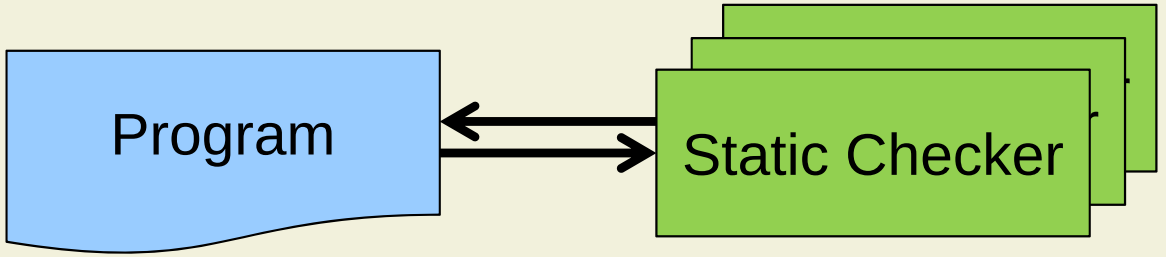
- Verification results capture precisely what has been checked statically

assume $\left(\bigvee_{A \in V} \left(\bigwedge_{a \in A} a\right)\right) \Rightarrow P$;

assert *P*;

- They provide a basis for a soundness proof
- They enable the combination of several, complementary static checkers

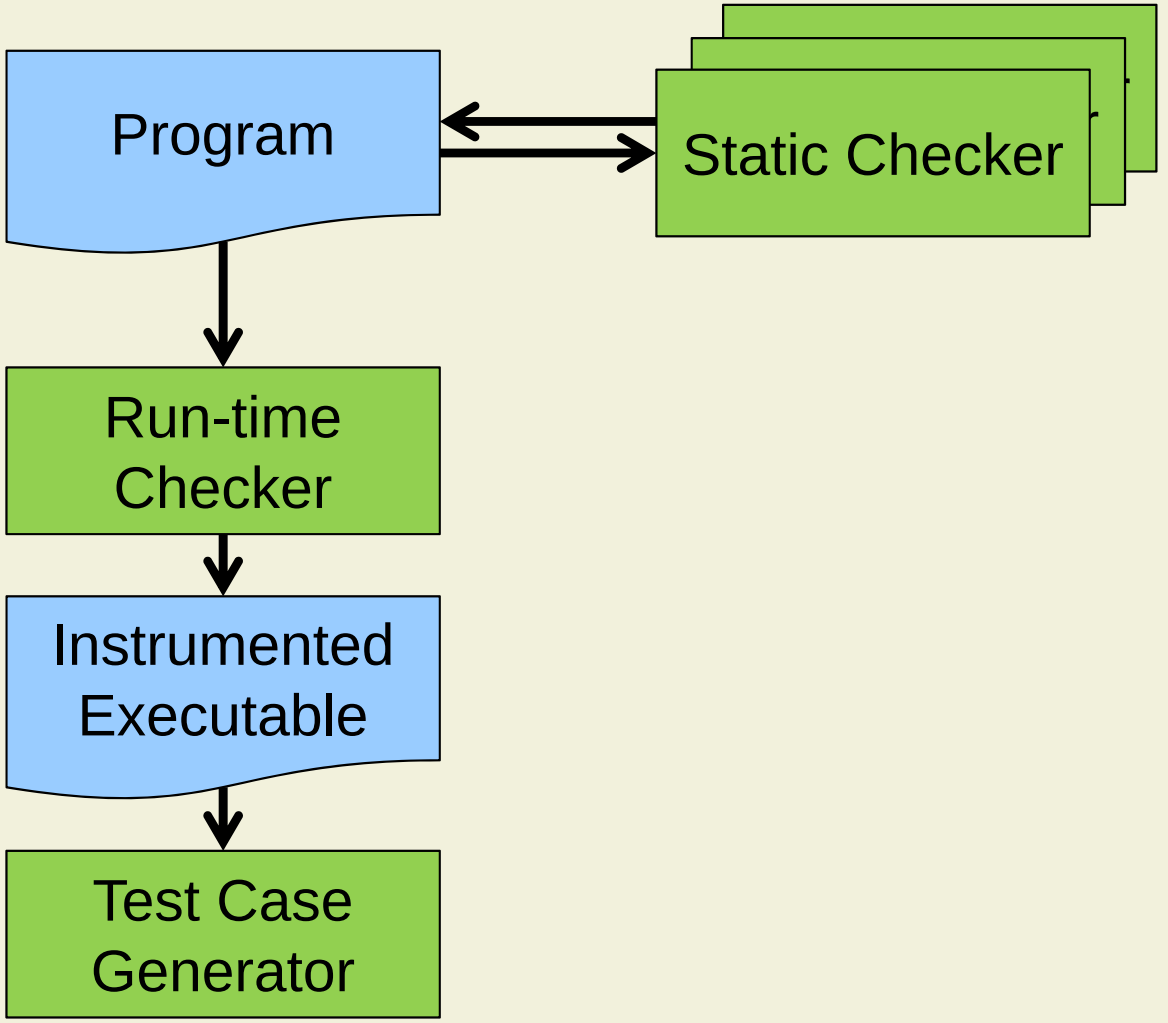
Collaborative Verification



assume $\left(\bigvee_{A \in V} \left(\bigwedge_{a \in A} a\right)\right) \Rightarrow P$;

assert *P*;

Collaborative Verification and Testing



Running Example in Pex

```
class Cell {
  int v;

  static int M( Cell c, Cell d )
    requires c != null ∧ d != null;
    requires c.v != 0 ∧ d.v != 0;
    ensures result < 0;
  {
    if ( sign(c.v) == sign(d.v) )
      c.v = (–1) * c.v;

    return c.v * d.v;
  }
}
```

Error due to aliasing

Error due to overflow

Reminder: Verification Results

```
static int M( Cell c, Cell d )
  requires c != null ∧ d != null;
  requires c.v != 0 ∧ d.v != 0;
  ensures { verified { a1, a2, a3 } } result < 0;
{
  assumed c != d as a1;
  if ( sign(c.v) == sign(d.v) ) {
    assumed bounded((–1) * c.v) == (–1) * c.v as a2;
    c.v = (–1) * c.v;
  }
  assumed bounded(c.v * d.v) == c.v * d.v as a3;
  return c.v * d.v;
}
```

Instrumentation for Pex

```
static int M( Cell c, Cell d )
{
  assume c != null ∧ d != null;
  assume c.v != 0 ∧ d.v != 0;
  a1 = ( c != d );
  if ( a1 ∧ c.v * d.v < 0 ) {
    assert new BigInt(c.v) * new BigInt(d.v) == new BigInt((–1) * c.v) );
  }
  a3 = ( new BigInt(c.v) * new BigInt(d.v) < 0 );
  assume a1 ∧ a2 ∧ a3 ⇒ c.v * d.v < 0;
  assert c.v * d.v < 0;
  return c.v * d.v;
}
```

Error due to aliasing

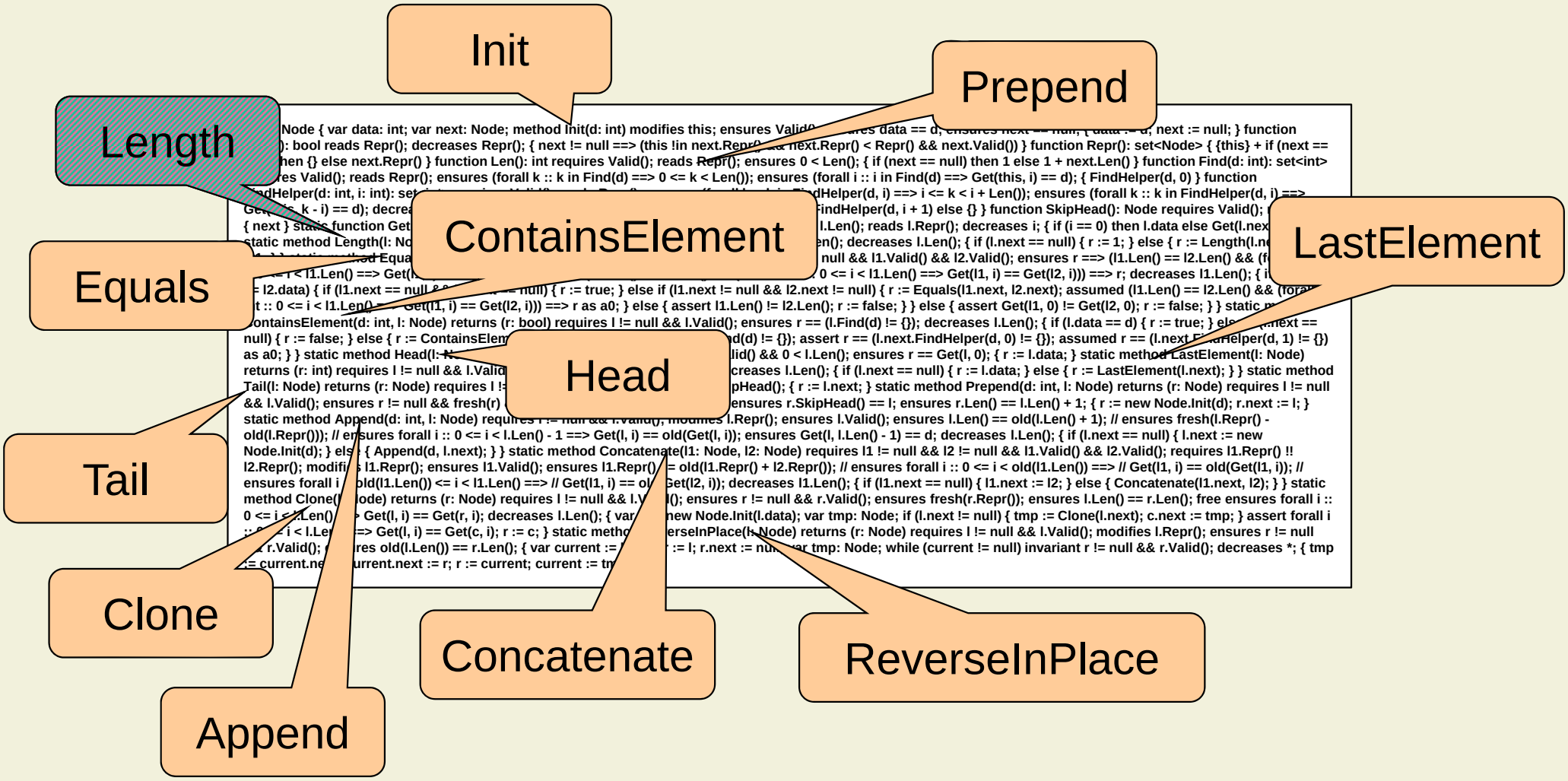
Assumption introduces more branches

Error due to overflow

Assumption directs search to cases not verified statically

Small Test Suites

- Fully verified methods need not be tested



Small Test Suites: Dafny Experiment

Verification	Tested Methods	Test Reduction
Sound	Equals, ContainsElement, ReverseInPlace	66%
Unbounded integer arithmetic	Length, Equals, ContainsElement, ReverseInPlace	58%
Loop unrolling	Equals, ContainsElement, ReverseInPlace	65%
Unbounded integer arithmetic and loop unrolling	Length, Equals, ContainsElement, ReverseInPlace	58%

- Verification time limited to two hours

Conclusion

- Verification results with explicit assumptions
  - Make compromises of static checkers explicit
  - Give definite answers about program correctness
  - Enable tool integration
- Testing with verification results
  - Finds more errors than testing alone
  - Leads to smaller, more targeted test suites
- Collaborative verification and testing allows engineers to balance static checking and testing