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
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;
    }
}
Running Example in Clousot

class Cell {
  int v;

  static int M( Cell c, Cell d )
  requires c != null \land d != null;
  requires c.v != 0 \land 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

Approach

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

```java
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;
    }
}
```
Explicit Assumptions

- At each program point where an unsound assumption $P$ is made, we insert a clause
  
  \[
  \text{assumed } P \text{ as } a;
  \]

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

- The semantics is an assignment
  
  \[
  a := a \land P
  \]
Explicit Assumptions: Examples

**Unbounded arithmetic**

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

**Loop unrolling**

```plaintext
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:

\[
\text{assert } \bigvee P;
\]

where each element of \( V \) is a set of assumption-variables.

- The semantics uses an assumption:

\[
\text{assume } \left( \bigvee_{A \in V} \left( \bigwedge_{a \in A} a \right) \right) \Rightarrow P; \\
\text{assert } P;
\]
Verification Results: Example

method Foo(…)
modifies M;
{
assert { verified A } \forall o,f :: (o,f) \in M \lor o.f == old(o.f);
}
static int M( Cell \textit{c}, Cell \textit{d} )
  \textbf{requires} \textit{c} \neq \textbf{null} \land \textit{d} \neq \textbf{null};
\textbf{requires} \textit{c}.v \neq 0 \land \textit{d}.v \neq 0;
\textbf{ensures} \{ \text{verified} \{ a_1, a_2, a_3 \} \} \ \text{result} < 0;
\{
  \textbf{assumed} \ \textit{c} \neq \textit{d} \ \textbf{as} \ a_1;
  \textbf{if} ( \text{sign}(\textit{c}.v) == \text{sign}(\textit{d}.v) ) \{ \\
    \textbf{assumed} \ \textit{bounded}((-1) \times \textit{c}.v) == (-1) \times \textit{c}.v \ \textbf{as} \ a_2; \\
    \textit{c}.v = (-1) \times \textit{c}.v;
  \}
  \textbf{assumed} \ \textit{bounded}(\textit{c}.v \times \textit{d}.v) == \textit{c}.v \times \textit{d}.v \ \textbf{as} \ a_3; \\
  \textbf{return} \ \textit{c}.v \times \textit{d}.v;
\}
Using Verification Results

- Verification results capture precisely what has been checked statically

\[
\text{assume } \left( \bigvee_{A \in V} \left( \bigwedge_{a \in A} a \right) \right) \Rightarrow P; \\
\text{assert } P;
\]

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

Program

Static Checker

assume \( \bigvee_{A \in V} \left( \bigwedge_{a \in A} a \right) \) \( \Rightarrow P \);
assert \( P \);
Collaborative Verification and Testing

- Program
- Run-time Checker
- Instrumented Executable
- Test Case Generator
- Static Checker
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;
    }
}
static int M( Cell c, Cell d )
  requires c != null \&\& d != null;
  requires c.v != 0 \&\& d.v != 0;
  ensures \{ verified \{ a_1, a_2, a_3 \} \} result < 0;
{
  assumed c != d as a_1;
  if ( sign(c.v) == sign(d.v) ) {
    assumed bounded((-1) * c.v) == (-1) * c.v as a_2;
    c.v = (-1) * c.v;
  }
  assumed bounded(c.v * d.v) == c.v * d.v as a_3;
  return c.v * d.v;
}
static int M( Cell c, Cell d )
{
    assume c != null \&\& d != null;
    assume c.v != 0 \&\& d.v != 0;
    a1 = ( c != d );
    if ( sign(c.v) == sign(d.v) ) {
        a2 = ( new BigInt(-1) * new BigInt(c.v) == new BigInt((-1) * c.v) );
        c.v = (-1) * c.v;
    }
    a3 = ( new BigInt(c.v) * new BigInt(d.v) == new BigInt(c.v * d.v) );
    assume a1 \&\& a2 \&\& a3 \Rightarrow c.v * d.v < 0;
    assert c.v * d.v < 0;
    return c.v * d.v;
}
Small Test Suites

- Fully verified methods need not be tested
## Small Test Suites: Dafny Experiment

<table>
<thead>
<tr>
<th>Verification</th>
<th>Tested Methods</th>
<th>Test Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sound</td>
<td>Equals, ContainsElement, ReverseInPlace</td>
<td>66%</td>
</tr>
<tr>
<td>Unbounded integer arithmetic</td>
<td>Length, Equals, ContainsElement, ReverseInPlace</td>
<td>58%</td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>Equals, ContainsElement, ReverseInPlace</td>
<td>65%</td>
</tr>
<tr>
<td>Unbounded integer arithmetic and loop unrolling</td>
<td>Length, Equals, ContainsElement, ReverseInPlace</td>
<td>58%</td>
</tr>
</tbody>
</table>

- 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**