An Introduction to Automated Program Verification with Permission Logics

Uri Juhasz, Ioannis Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, Alex Summers (and several students)

15th May 2015, Systems Group, ETH Zurich
Initial Example (Pseudo Java)

class Cell {
    int v

    void add(Cell c)
    { v = v + c.v }
}

void client()
{
    Cell c1 = new Cell()
    c1.v = 1
    Cell c2 = new Cell()
    c2.v = 2

    c1.add(c2)

    assert c1.v == 3
    assert c2.v == 2
}

Goal: Check assertions \textit{statically}

Challenges:
- Whole-code analysis is \textit{expensive}
- Dynamic dispatch (inheritance; open-world assumption)
class Cell {
    int v

    void add(Cell c)
    {
    // missing implementation
    }
}

void client() {
    Cell c1 = new Cell()
    c1.v = 1
    Cell c2 = new Cell()
    c2.v = 2

    c1.add(c2)

    assert c1.v == 3
    assert c2.v == 2
}
Specifications

class Cell {
    int v

    void add(Cell c)
        requires c != null
        ensures v == old(v) + old(c.v)
        { v = v + c.v }
}

void client() {
    Cell c1 = new Cell()
    c1.v = 1
    Cell c2 = new Cell()
    c2.v = 2
    c1.add(c2)

    assert c1.v == 3
    assert c2.v == 2
}
Reasoning with Specifications

```java
class Cell {
    int v

    void add(Cell c) {
        requires c != null
        ensures v == old(v) + old(c.v)
        { }  // missing code
    }
}
```

```java
void client() {
    Cell c1 = new Cell()
c1.v = 1
    Cell c2 = new Cell()
c2.v = 2
c1.add(c2)
    assert c1.v == 3
    assert c2.v == 2
}
```
An Incorrect Implementation

class Cell {
    int v

    void add(Cell c) {
        requires c != null
        ensures v == old(v) + old(c.v)
    {
        v = v + c.v
        c.v = 0
    }
}

void client() {
    Cell c1 = new Cell()
    c1.v = 1
    Cell c2 = new Cell()
    c2.v = 2
    c1.add(c2)
    assert c1.v == 3
    assert c2.v == 2
}
class Cell {
    int v

    void add(Cell c) {
        requires c != null
        ensures v == old(v) + old(c.v)
        ensures c.v == old(c.v)
        
        v = v + c.v
        c.v = 0
    }
}

void client() {
    Cell c1 = new Cell()
    c1.v = 1
    Cell c2 = new Cell()
    c2.v = 2

    c1.add(c2)

    assert c1.v == 3
    assert c2.v == 2
}
class Cell {
    int v
    void add(Cell c) {
        requires c != null
        ensures v == old(v) + old(c.v)
        ensures c.v == old(c.v)
        { v = v + c.v }
    }
}

void client() {
    Cell c1 = new Cell()
    c1.v = 1
    Cell c2 = new Cell()
    c2.v = 2
    c1.add(c2)
    assert c1.v == 3
    assert c2.v == 2
}
class Cell {
    int v

    void add(Cell c) {
        requires c != null
        ensures v == old(v) + old(c.v)
        ensures c.v == old(c.v)
        { v = v + c.v }
    }
}

void client() {
    Cell c1 = new Cell()
    c1.v = 1
    Cell c2 := new Cell()
    c2.v = 2
    c1.add(c1)
    assert c1.v == 3
    assert c2.v == 2
}
Reason about Shared State (Including Data Races) and Control Aliasing
Modular Static Verification + Shared State

\[ \text{foo}(x) \quad \text{bar}(x) \]
Modular Static Verification + Shared State

foo(x)   bar(x)
Modular Static Verification + Shared State

foo(x)  bar(x)
foo(x)  bar(x)
Permission Transfer

foo(x) \quad bar(x)
Permission Transfer

foo(x)  bar(x)

?
Fractional Permissions

\[ \text{foo}(x) \quad \text{bar}(x) \]
Splitting Fractional Permissions

foo(x)  bar(x)
Merging Fractional Permissions

$\text{foo}(x)$  $\text{bar}(x)$
Accessibility predicates denote permissions

Assertions may be heap-dependent

Fractional permissions

Conjunction sums up permissions (similar to * in separation logic)

Write permission is exclusive (similar to * in separation logic)
Demo
Permission Transfer Reloaded

Idea of *permission transfer* generalises
  - Fork-join (transfer between threads)
  - Locks (transfer to/from lock invariant)
  - Message passing (pass permissions)

Common operations
  - **Gain** permissions
  - **Lose** permissions
Silver: Inhale and Exhale Statements

Statement **inhale A** means
- Gain permissions required by A (e.g. \( \text{acc}(x.f) \))
- Assume logical constraints in A (e.g. \( x.f \neq \emptyset \))

Statement **exhale A** means
- Assert and remove permissions required by A
- Assert logical constraints in A
- Havoc locations to which all permissions were removed (i.e. forget their values)
Concurrency Examples
Fork-Join Concurrency (Pseudo-Java)

```java
class Cell {
    int v

    void add(Cell c) {
        v = v + c.v
    }
}

void client() {
    Cell c1 = new Cell()
    c1.v = 1
    Cell c2 = new Cell()
    c2.v = 2

    Token tk = fork c1.add(c2)
    // ...
    join tk

    assert c1.v == 3
    assert c2.v == 2
}
```
Locks and @GuardedBy Annotations (Pseudo-Java)

class SharedPair {
    @GuardedBy(“this”)
    int x, y

class SharedPair {
    @GuardedBy("this", "x < y")
    int x, y
}

synchronized (this) {
    assert x < y
    x = x + dx
    y = y + dy
    assert x < y
}
class SharedPair {
    @GuardedBy("this", "x < y")
    int x, y

    void inc(int dx, int dy) {
        assert dx <= dy
        synchronized(this) {
            assert x < y
            x = x + dx
            y = y + dy
            assert x < y
        }
    }
}
Viper: Our Verification Infrastructure

Silver:
- Intermediate Verification Language
- Few (but expressive) constructs
- Designed with verification and inference in mind

Back-ends: Two verifiers; plans to develop inference, slicer

Front-ends (proof of concept):
- Chalice (concurrency research)
- Scala (very small subset)
- Java (VerCors, U Twente)
- OpenCL (VerCors, U Twente)
Viper: Our Verification Infrastructure

Silver AST

Chalice
Scala
Java (U Twente)
OpenCL (U Twente)

generate

verifies by

Carbon
Scala
Boogie (Microsoft)

encodes in

queries

queries

Carbon
Boogie (Microsoft)

encodes in

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

queries

verifies by

query
Verification Condition Generation vs. Symbolic Execution

Query prover once with full information (Carbon)

Program read by Verifier calculates WPs given to Prover

one weakest precondition per method

Query prover often with limited information (Silicon)

Program read by Verifier maintains Symbolic State σ

symbolically execute every path through each method

σ₁, σ₂, σ₃, σ₄, σ₅ used by Prover

query prover at every step if next statement is executable
Outlook

Information hiding, abstraction and inheritance

Unbounded (recursive) data structures

Obligations – the dual to permissions

Specification inference

Encoding of high-level features
  – Immutable data (vs. permissions)
  – Lazy evaluation (vs. permissions)
  – Closures/higher order functions
  – Actor-based concurrency
  – Fine-grained locking, lock-free algorithms
Silver AST

Chalice

Scala

Java (U Twente)

OpenCL (U Twente)

generate

verified by

Carbon

Silicon

Boogie (Microsoft)

Z3 (Microsoft)

encodes in

queries

queries

infer additional specifications

Static Analysis

(Boogie

(Silicon (Microsoft))
You shouldn’t even be here!
Fork-Join Concurrency (Pseudo-Java)

```java
class Cell {
    int v

    void add(Cell c) {
        v = v + c.v
    }
}

void client() {
    Cell c1 = new Cell()
    c1.v = 1
    Cell c2 = new Cell()
    c2.v = 2

    Token tk = fork c1.add(c2)
    // ...
    join tk

    assert c1.v == 3
    assert c2.v == 2
}
```
Fork-Join Concurrency (Silver)

field v: Int

method add(this: Ref, c: Ref)
    requires acc(this.v) && acc(c.v, ½)
    ensures acc(this.v) && acc(c.v, ½)
    ensures this.v == old(this.v) + old(c.v)
{
    this.v := this.v + c.v
}
Locks and @GuardedBy Annotations (Pseudo-Java)

class SharedPair {
    @GuardedBy("this")
    int x, y
Locks and @GuardedBy Annotations (Silver)

field x: Int
field y: Int

define inv(this) acc(this.x) && acc(this.y)
Lock Invariants (Pseudo-Java)

class SharedPair {
    @GuardedBy("this", "x < y")
    int x, y

    void inc(int dx, int dy) {
        synchronized (this) {
            assert x < y
            x = x + dx
            y = y + dy
            assert x < y
        }
    }
}

Lock Invariants (Pseudo-Java)

class SharedPair {
    @GuardedBy("this", "x < y")
    int x, y

    void inc(int dx, int dy) {
        assert dx <= dy

        synchronized(this) {
            assert x < y

            x = x = dx
            y = y = dy

            assert x < y
        }
    }
}
Lock Invariants (Silver)

```plaintext
field x: Int
field y: Int

define inv(this)  acc(this.x) && acc(this.y)
&& this.x <= this.y
```