Viper

A Verification Infrastructure for Permission-Based Reasoning

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

24th March 2015, JML Workshop, Leiden
Our Vision

Automatic Program Verification

- Safety (memory accesses, non-null, ...)
- Correctness (functional specs)
- Termination, message replies, ...
Verification using Automatic Provers

- **Automatic first-order logic tools** – major progress in the last decade (SAT, SMT)

- **Intermediate verification languages** - Boogie, Why, ...

- **Back-end**: verifier (verification condition generator)

= **Common infrastructure** for building **front-end verifiers**
Verification using Automatic Provers

- Automatic first-order logic tools – major progress in the last decade (SAT, SMT)

- Intermediate verification languages - Boogie, Why, ...

- Back-ends: verifiers - but also inference engines, slicers, static analysers, ...

= Common infrastructure for building front-end verifiers
Verification using Automatic Provers

Common infrastructure enabled many *success stories* and *tools*

- Microsoft Hypervisor (VCC)
- Device drivers (Corral)
- Spec#, Dafny
- Krakatoa, Jessie
- Frama-C, Why3
- ...
Permission-Based Reasoning

Separation logic and others permission logics:

- *Locally* reason about *shared mutable state*
- Many *successful applications*, including
  - Device driver safety (Microsoft)
  - Belgian Electronic Identity Card
- Many *ongoing developments*
  (esp. fine-grained concurrency)

Not a first-order logic
→ Significantly complicates using existing provers
Permission-Based Reasoning

Consequence: many custom verification engines (usually based on symbolic execution): Smallfoot, VeriFast, jStar, ...

Alternative: Encoding SL into FOL (e.g. Chalice)
Viper: Our Verification Infrastructure

Silver:
- Native support for permissions
- 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)
Modular Static Verification + Shared State

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

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

$\text{foo}(x)$  $\text{bar}(x)$
Permissions

foo(x)  bar(x)
Permission Transfer

foo(x)  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

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

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 **exhale** \( A \) means
- Assert and remove permissions required by \( A \)
- Assert logical constraints in \( A \) (e.g. \( c.f \ == \emptyset \))
- Havoc locations to which all permissions is lost (i.e. forget their values)

Statement **inhale** \( A \) means
- Gain permissions required by \( A \)
- Assume logical constraints in \( A \)
Silver: Assertion Language Basics

Based on *implicit dynamic frames*

*Accessibility predicates* denote permissions

Assertions may be *heap-dependent*

*Fractional permissions*

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

\[ \text{acc}(c.f) \]

\[ \text{acc}(c.f, \frac{1}{2}) \]

\[ \text{acc}(c.f, \frac{1}{2}) \land \text{acc}(c.f, \frac{1}{2}) \]
Demo
Silver: Language Features

Objects and fields, if-then-else, methods (with pre/post specs), loops (with invariants)

No notion of concurrency (encode via **inhale/exhale**)

Simple type system

- Int, Bool, Ref, Perm
- Mathematical sets **Set[T]** and sequences **Seq[T]**
Unbounded data structures via *recursive predicates*

```plaintext
predicate list(x: Ref) {  
    acc(x.val) && acc(x.next)  
    && (x.next != null ==> list(x.next))  
}
```

*fold/unfold* statements exchange predicate instances for their bodies (not automatic due to recursion)

*Heap-dependent, pure abstraction functions*

```plaintext
function elems(x: Ref): Seq[Int]  
  requires list(x)  
{ unfolding list(x) in  
    [x.val] ++ (x.next == null ? [] : elems(x.next))  
}
```
Domains to specify custom mathematical types
- Type-parametric domains
- Domain functions
- Domain axioms

```plaintext
domain Pair[X,Y] {
    function pair(x: X, y: Y): Pair[X,Y]
    function first(p: Pair[X,Y]): X

    axiom forall x: X, y: Y • first(pair(x,y)) == x
}
...
method foo(x: Ref, p: Pair[Int, Int])
    requires acc(x.f)
{ x.f := first(p) }
```
Abstract read permissions

- Alternative to fractional permissions
- No need to commit to concrete fractions, e.g. $\frac{1}{2}$

```
method foo(x: Ref, p: Perm)
    requires 0 < p && acc(x.f, p)
{
    // read x.f
    if (*) {
        var q: Perm
        constraining (q) {
            foo(x, q) // give away q < p
        }
    }
}
```

Allows unbounded splitting and counting
Paired assertions \([A, B]\)

- When inhale, \(A\) is used
- When exhaled, \(B\) is used
- Asymmetry justified elsewhere
  (type system, soundness proof, induction schema, ...)

\[
\begin{align*}
&\forall x: \text{Nat} \cdot P(x), \\
&\forall x: \text{Nat} \cdot \\
&\quad (\forall y: \text{Nat} \cdot y < x \implies P(y)) \implies P(x)
\end{align*}
\]
Magic Wands
Magic Wands Primer

Boolean implication \( A \Rightarrow B \)

Modus Ponens \( A \land (A \Rightarrow B) \models B \)

Separating implication \( A \rightarrow* B \)

Modus Ponens \( A \ast (A \rightarrow* B) \models B \)

\( A \rightarrow* B \) can be understood as an exchange promise

“If \( A \) and \( A \rightarrow* B \) are given up, then \( B \) is guaranteed to hold”
Semantics of the magic wand:

\[ h \models A \rightarrow \star B \iff \forall h' \perp h \cdot (h' \models A \Rightarrow h \cup h' \models B) \]

*Quantification over state*; typically not supported in automated verifiers

Used in proofs by hand (e.g. data structure modifications, barrier synchronization)
Permission Bookkeeping and Recursive Predicates

```java
predicate list(xs: Ref) {
  acc(xs.next) && (xs.next != null ==> list(xs.next))
}
```

```java
method rec(xs: Ref) {
  requires list(xs)
  ensures list(xs)

  { 
    unfold list(xs)
    rec(xs.next)
    // Ignoring base case
    unfold list(xs)
  }
}
```
Permission Bookkeeping and Recursive Predicates

Predicate

```java
predicate list(xs: Ref) {
    acc(xs.next) && (xs.next !null ==> list(xs.next))
}
```

Method

```java
method rec(xs: Ref)
    requires list(xs)
    ensures list(xs)
{
    unfold list(xs)
    rec(xs.next)
    // Ignoring base case
    fold list(xs)
}
```

Bookkeeping *implicitly* done by the *call stack*
Permission Bookkeeping and Recursive Predicates

\[
\text{predicate } \text{list}(\text{xs: Ref}) \{ \\
\quad \text{acc(\text{xs.next})} \land \text{(\text{xs.next} \neq \text{null} \Rightarrow \text{list(\text{xs.next})})} \\
\}\]

\[
\text{method } \text{it}(\text{xs: Ref}) \\
\text{requires } \text{list}(\text{xs}) \\
\text{ensures } \text{list}(\text{xs}) \\
\{ \\
\quad \text{var } \text{cur := xs} \\
\quad \text{while (*)} \\
\quad \quad \{ \\
\quad\quad \text{unfold } \text{list(\text{cur})} \\
\quad\quad \text{cur := cur.next} \\
\quad\quad \} \\
\}\]

unfold
unfold
unfold
iteration
Permission Bookkeeping and Recursive Predicates

predicate list(xs: Ref) {
    acc(xs.next) && (xs.next != null ==> list(xs.next))
}

method it(xs: Ref)
    requires list(xs)
    ensures list(xs)
{
    var cur := xs
    while (*)
        inv ???
    {
        unfold list(xs)
        cur := cur.next
    }
    ???
}
predicate list(xs: Ref) {
    acc(xs.next) && (xs.next != null ==> list(xs.next))
}

method it(xs: Ref) {
    requires list(xs)
    ensures list(xs)
    {
        var cur := xs
        while (*)
            inv list(cur) --* list(xs)
        {
            unfold list(xs)
            cur := cur.next
            // Update wand
        }
        // Get list(xs) from wand
    }
    // Get list(xs) from wand
}
Magic Wands in Silver

Specifying *partial data structures* is just **one** application

We support arbitrary* wands

**Main contribution:** *automatic footprint computation*

- Recall $A \ast (A \rightarrow \ast B) \models B$
- Footprint = permission delta between $A$ and $B$

**Examples**

- $\text{true} \rightarrow \ast \text{acc}(x.f)$
- $\text{acc}(x.f) \rightarrow \ast \text{acc}(x.f)$
- $\text{acc}(x.f, 1/3) \rightarrow \ast \text{acc}(x.f, 1/1)$
- $\text{acc}(x.f) \rightarrow \ast \text{acc}(y.f)$
- $\text{acc}(x.f) \rightarrow \ast \text{acc}(y.f)$
Demo
http://bitbucket.org/viperproject/

Chalice

Scala

Java (U Twente)

OpenCL (U Twente)

generate

Silver AST

infer additional specifications

Static Analysis

verified by

Carbon

encodes in

Boogie (Microsoft)

queries

queries

Z3 (Microsoft)

Scala

Silicon

queries