Viper
A Verification Infrastructure for Permission-Based Reasoning

Quantified Permissions
Dynamic-Frames-Style Specifications in Permission Logics
Frame Problem

\[
\{P\} \land \{R\} \not\rightarrow \{Q\} \land \{R\}
\]
Framing Methodologies

Dynamic Frames (no permissions)

\[
\{ P \} \cap \text{reads}(R) = \emptyset
\]

\[
\frac{\{ P \} \cap \text{reads}(R) = \emptyset}{\{ P \wedge R \} \cap \text{reads}(R) = \emptyset}
\]
Framing Methodologies

\[
\{ P \} C \{ Q \} \quad \text{modifies}(C) \cap \text{reads}(R) = \emptyset \\
\{ P \wedge R \} C \{ Q \wedge R \}
\]

Dynamic Frames (no permissions)

\[ \{ P \} C \{ Q \} \quad \text{modifies}(C) \cap \text{reads}(R) = \emptyset \]
\[ \{ P \wedge R \} C \{ Q \wedge R \} \]

Separation Logic (permissions)

\[
\{ P \} C \{ Q \} \\
\{ P \ast R \} C \{ Q \ast R \}
\]
method mutate()
  requires acc(this.val)
  ensures acc(this.val)

method client(x, y)
  requires acc(x.val) * acc(y.val)
  \{
  var tmp := y.val
  x.mutate()
  assert tmp == y.val
  \}
method mutate()
  requires acc(this.val)
  ensures acc(this.val)

method client(x, y)
  requires acc(x.val) * acc(y.val)
  
  var tmp := y.val
  x.mutate()
  assert tmp == y.val

{P} C {Q}

{P} C {Q* R}
Permissions

method mutate()
  requires acc(this.val)
  ensures acc(this.val)

method client(x, y)
  requires acc(x.val) * acc(y.val)
  
  var tmp := y.val
  x.mutate()
  assert tmp == y.val

\[
\{ P \} \mathcal{C} \{ Q \} \\
\{ P* R \} \mathcal{C} \{ Q* R \}
\]
**Permissions**

```plaintext
method mutate()
  requires acc(this.val)
  ensures acc(this.val)

method client(x, y)
  requires acc(x.val) * acc(y.val)
  { 
    var tmp := y.val 
    x.mutate() 
    assert tmp == y.val 
  }
```

\[
\{ P \} \ C \ \{ Q \} \\
\{ P \land R \} \ C \ \{ Q \land R \} \\
\{ P \ast R \} \ C \ \{ Q \ast R \}
\]
method mutate()
  requires acc(this.val)
  ensures acc(this.val)

method client(x, y)
  requires acc(x.val) \times acc(y.val)
  \{ 
  var tmp := y.val
  x.mutate()
  assert tmp == y.val
  \}

\{ P \} C \{ Q \}
\{ P R \} C \{ Q R \}
method mutate()
  requires acc(this.val)
  ensures acc(this.val)

method client(x, y)
  requires acc(x.val) * acc(y.val)
  \{ 
  var tmp := y.val
  x.mutate()
  assert tmp == y.val
  \}

\{P\} C \{Q\}
\{P \land R\} C \{Q \land R\}
\{P \land R\} C \{Q \land R\}
Common Tool Infrastructures

No Permissions

Prog. language, spec. language and methodology

Front end

Intermediate verification language

Verification condition generator

SMT solver
Common Tool Infrastructures

No Permissions

- Prog. language, spec. language and methodology
- Front end
- Intermediate verification language
- Verification condition generator
- SMT solver
Common Tool Infrastructures

No Permissions

- Front end
- Intermediate verification language
- Verification condition generator
- SMT solver

Permissions

- Custom verifier
- SMT solver

Prog. language, spec. language and methodology
Common Tool Infrastructures

No Permissions

- Prog. language, spec. language and methodology
- Front end
- Intermediate verification language
- Verification condition generator
- SMT solver

Permissions

- Prog. language, spec. language and methodology
- Custom verifier
- SMT solver
Common Tool Infrastructures

No Permissions

- Front end
- Intermediate verification language
- Verification condition generator
- SMT solver

abstraction gap

Permissions

- Custom verifier
- Prog. language, spec. language and methodology
- SMT solver
Insufficient Tool Support for Permission Logics

Verification efforts do not benefit fully from advances in theory

Theory does not receive feedback from applications

Facilitate the

1. development of tools
2. prototyping of encodings

for permission-based verification
Viper

Front end

Intermediate verification language

Back-end tools

SMT solver

http://www.iconsonland.com/
Permission Transfer

\{ P \} C \{ Q \}

\{ P \ast R \} C \{ Q \ast R \}

callee

caller
Viper Features: Inhale and Exhale

exhale P
- assert value constraints
- check and remove permissions
- havoc newly-inaccessible locations

inhale Q
- obtain permissions
- assume value constraints
Demo
Recursive Predicates

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

unfold list(this)
// access this.data
// and this.next
fold list(this)
```
Recursive Predicates: Limitations

1. Extending
Recursive Predicates: Limitations

1. Extending
Recursive Predicates: Limitations

1. Extending
2. Sharing
Recursive Predicates: Limitations

1. Extending

2. Sharing
Recursive Predicates: Limitations

1. Extending
2. Sharing
3. Traversing
Unbounded Data Structures

**Unidirectional**

Recursive predicates are often a suitable specification mechanism

**Multidirectional**

**Random Access**

**Unstructured**
Unidirectional

Multidirectional

Random Access

Unstructured

need for an alternative specification mechanism
Quantified Permissions

forall n in nodes ::
  acc(n.next) && acc(n.prev)
forall n in nodes :: acc(n.next) && acc(n.prev)

forall i in [0..5] :: acc(arr[i])
forall n in nodes ::
  acc(n.next) && acc(n.prev)

forall i in [0..5] ::
  acc(arr[i])

forall i in [0..5] ::
  i % 2 == 1 ==> acc(arr[i])
Quantified Permissions

forall n in nodes ::
    acc(n.next) && acc(n.prev)

forall i in [0..5] ::
    acc(arr[i])
forall i in [0..5] ::
    i % 2 == 1 ==> acc(arr[i])

forall n in nodes ::
    acc(n.succs) && acc(n.marked)

**Multidirectional**

**Random Access**

**Unstructured**
**Quantified Permissions**

```
forall n in nodes ::
  acc(n.next) && acc(n.prev)
```

```
forall i in [0..5] ::
  acc(arr[i])
```

```
forall i in [0..5] ::
  i % 2 == 1 ==> acc(arr[i])
```

```
forall n in nodes ::
  acc(n.succs) && acc(n.marked) &&
  (n.marked ==> forall m in n.succs :: m.marked)
```
List Tail Sharing Revisited

**predicate** list(nodes: Set[Ref]) {
  forall n ∈ nodes ::
  acc(n.data) &&
  acc(n.next) &&
  (n.next != null ==> n.next ∈ nodes)
}

list(nodes) &&
v ∈ nodes &&
w.next ∈ nodes
General Receiver Expressions

inhale $\forall x \in S :: \text{acc}(e(x).f)$

exhale $\forall y \in R :: \text{acc}(y.f)$
General Receiver Expressions: Challenge

inhale $\forall x \in S :: \text{acc}(e(x).f)$

$\{x_1, x_2, x_3, x_4, \ldots, x_n\}$

$e(x).f$

exhale $\forall y \in R :: \text{acc}(y.f)$
General Receiver Expressions: Challenge

inhale $\forall x \in S :: \text{acc}(e(x).f)$

$\{x_1, x_2, x_3, x_4, \ldots, x_n\}$

$e(x).f$

$\exists x \in S :: e(x) = y?$

$\forall y \in R :: \text{acc}(y.f)$

$\{y_1, y_2, y_3, \ldots, y_m\}$
General Receiver Expressions: Injectivity

1. Require $e(x)$ to be injective
(naturally satisfied by e.g. arrays and graphs)

inhale $\forall x \in S :: \text{acc}(e(x).f)$

$\{x_1, x_2, x_3, x_4, \ldots, x_n\}$

$e(x).f$

exhale $\forall y \in R :: \text{acc}(y.f)$

$\{y_1, y_2, y_3, \ldots, y_m\}$
General Receiver Expressions: Inverse Functions

inhale $\forall x \in S :: \text{acc}(e(x).f)$

1. Require $e(x)$ to be injective

2. Axiomatise inverse function $e^{-1}(x)$ to SMT solver

exhale $\forall y \in R :: \text{acc}(y.f)$
**General Receiver Expressions: Challenge**

**Inhale** \( \forall x \in S :: \text{acc}(e(x).f) \)

\( \{x_1, x_2, x_3, x_4, \ldots, x_n\} \)

**Exhale** \( \forall y \in R :: \text{acc}(y.f) \)

\( \{y_1, y_2, y_3, \ldots, y_m\} \)

\( e^{-1}(y) \in S? \)

\( y.f \in L? \)

\( \text{acc}(y.f)? \)
Demo
Dynamic Frames vs. Permissions

Permission Logics: disjointness per default

Concurrency

Dynamic Frames: sharing per default

Arbitrary data structures

poorly supported in tools
Permission Logics: disjointness per default

Dynamic Frames: sharing per default

Concurrencies

Arbitrary data structures

Viper’s quantified permissions

specify and maintain disjointness explicitly

Dynamic Frames vs. Permissions
Viper: Currently

- Chalice
- Java
- OpenCL
- Python

Intermediate verification language

- Verification condition generator (verifier)
- Symbolic execution (verifier)
- Abstract interpretation (inference)

Boogie

SMT solver

Viper
Viper: Next

- Chalice
- Java
- OpenCL
- Python
- Fine-Grained Concurrency

Intermediate verification language

- Verification condition generator (verifier)
- Symbolic execution (verifier)
- Abstract interpretation (inference)

Boogie

SMT solver
SMT solver

Intermediate verification language

Verification condition generator (verifier)
Symbolic execution (verifier)
Abstract interpretation (inference)

Boogie → SMT solver

Chalice
Java
OpenCL
Python

Viper

http://viper.ethz.ch

C₁ || C₂ +