A Foundation for Verifying Concurrent Programs

K. Rustan M. Leino
RiSE, Microsoft Research, Redmond

joint work with Peter Müller and Jan Smans

Lecture 1
2 September 2009
FOSAD 2009, Bertinoro, Italy
Permissions guide what memory locations are allowed to be accessed.

Activation records and monitors can hold permissions.

Permissions can be transferred between activation records and monitors.

Locks grant mutually exclusive access to monitors.
Today’s lecture

- More examples
- Preventing deadlocks
- Using abstraction
- Building a program verifier
OwickiGriesCounter

Summary, and ghost variables
Deadlocks

A deadlock is the situation where a nonempty set (cycle) of threads each waits for a resource (e.g., lock) that is held by another thread in the set.

Example:

```java
method M() ...
{
    acquire a;
    acquire b;
    ...
}

method N() ...
{
    acquire b;
    acquire a;
    ...
}
```
A deadlock is the situation where a nonempty set (cycle) of threads each waits for a resource (e.g., lock) that is held by another thread in the set.

- Deadlocks are prevented by making sure no such cycle can ever occur.
  - The program partially order locks.
  - The program must acquire locks in strict ascending order.
Wait order

- Wait order is a dense partial order (Mu, <=) with a bottom element ⊥
- <= is the strict version of <
- The wait level of an object o is stored in a mutable ghost field o.mu
- Accessing o.mu requires appropriate permissions, as for other fields
- The syntax `maxlock <= X` means (∀ ℓ∈Held • ℓ.mu <= X) where
  Held denotes the set of locks held by the current thread
With these preconditions, both methods verify

The conjunction of the preconditions is false, so the methods can never be invoked at the same time

```
method M()
    requires rd(a.mu)
    requires rd(b.mu)
    requires a.mu << b.mu
{
    acquire a;
    acquire b;
    ...
}

method N()
    requires rd(a.mu)
    requires rd(b.mu)
    requires b.mu << a.mu
{
    acquire b;
    acquire a;
    ...
}
```
Recall, the wait level of an object \( o \) is stored in the ghost field \( o.mu \).

Initially, the \( .mu \) field is \( \perp \).

The \( .mu \) field is set by the share statement:

\[
\text{share } o \text{ between } L \text{ and } H; \\
\]

picks some wait level strictly between \( L \) and \( H \), and sets \( o.mu \) to that level.

Provided \( L \ll H \) and neither denotes an extreme element, such a wait level exists, since the order is dense.

\[
\text{share } o; \\
\]

means

\[
\text{share } o \text{ between maxlock and } ; \\
\]
Deadlock prevention
Dining Philosophers

Specifying wait levels
Changing the wait order

When is:

reorder o between L and H; allowed?

When o.mu is writable!

... and the thread holds o

Recall, maxlock << X means

(∀l∈Held • l.mu << X), so uttering maxlock has the effect of reading many .mu fields

We either need rd(maxlock), or
Deadlocks when joining

method M() ...
{
  fork tk := N();
  acquire a;
  join tk;
  ...
}

method N() ...
{
  acquire a;
  ...
  release a;
}
Thread levels

fork tk := o.M() between L and H;
picks a level $\theta$ between L and H, and then sets tk.mu to $\theta$

The precondition of o.M() is checked, substituting $\theta$ as the value of any occurrence of `maxlock`

$maxlock << X$ now means

$(\forall l \in \text{Held} \bullet l.mu << X) \land \theta << X$

where $\theta$ is the one for the current thread

`join tk;` requires $maxlock << tk.mu$

without between clause, $\theta$ is picked as just barely above $maxlock$ of the forking thread
HandOverHand

Fine-grained locking
method Update(p: Node)
    requires acc(p.data, 40)
    ...
    {
        acquire p;
        while (p.next != null) ... {
            var nx := p.next;
            acquire nx;
            nx.data := nx.data + 1;
            release nx; // Note: This seems to overlook the acquire after the acquire nx.
            release p;
            p := nx;
        }
        release p;
    }

invariant
    acc(data, 60) && ... &&
    (next != null ==> acc(next.data, 40) && data <= next.data);
method Update(p: Node)
    requires acc(p.data,40)
    ...
    {
        acquire p;
        while (p.next != null) ... {
            var nx := p.next;
            acquire nx;
            nx.data := nx.data + 1;
            release p;
            p := nx;
        }
    }

invariant
    acc(data,60) && ... &&
    (next != null ==> acc(next.data,40) &&
    data <= next.data);
method Update(p: Node)
    requires acc(p.data, 40)
    ...
    {
        acquire p;
        while (p.next != null) ...
        {
            var nx := p.next;
            acquire nx;
            nx.data := nx.data + 1;
            release p;
            p := nx;
        }
    }

invariant
    acc(data, 60) && ...
    &&
        (next != null ==> acc(next.data, 40) && data <= next.data);
Hand-over-hand locking: the idea

Method `Update(p: Node)`
- **Requires** `acc(p.data, 40)`

```
method Update(p: Node)
  requires acc(p.data, 40)
  ...
  {
    acquire p;
    while (p.next != null) ... {
      var nx := p.next;
      acquire nx;
      nx.data := nx.data + 1;
      release p;
      p := nx;
    }
  }
```

**Invariant**
- `acc(data, 60) && ... && (next != null ==> acc(next.data, 40) && data <= next.data);`
What permissions to include in method Play’s precondition?
Predicates

Named container of permissions

class C
{
    predicate P {...}
    ...
}

fold P;

unfold P;
Predicates
Boogie

Intermediate verification language

Verification engine

Spec#
Dafny
Chalice
C with VCC specifications
C with HAVOC specifications

Boogie

Simplify
Z3
SMT Lib
Isabelle/HOL

Microsoft Research
Boogie language

- First-order mathematical declarations
  - type
  - const
  - function
  - axiom

- Imperative declarations
  - var
  - procedure
  - implementation
Boogie statements

- \( x := E \)
- \texttt{havoc x}
- \texttt{assert E}
- \texttt{assume E}
- ...

Useful idiom:

- \texttt{havoc x; assume \( P(x) \)};
- “set \( x \) to a value such that \( P(x) \) holds”
Weakest preconditions

For any command $S$ and post-state predicate $Q$, $\text{wp}(S,Q)$ is the pre-state predicate that characterizes those initial states from which every terminating trace of $S$:

- does not go wrong, and
- terminates in a state satisfying $Q$

\[
\begin{align*}
\text{wp}( x := E, Q ) &= Q[ E / x ] \\
\text{wp}( \text{havoc} x, Q ) &= (\forall x \bullet Q ) \\
\text{wp}( \text{assert} P, Q ) &= P \land Q \\
\text{wp}( \text{assume} P, Q ) &= P \Rightarrow Q \\
\text{wp}( S ; T, Q ) &= \text{wp}( S, \text{wp}( T, Q ))
\end{align*}
\]
var Heap: Ref × FieldName → Value;
var Mask: Ref × FieldName → Permission;

x := o.f; ⇔

assert o ≠ null;
assert Mask[o, f] > 0;
x := Heap[o, f];

o.f := x ⇔

assert o ≠ null;
assert Mask[o, f] == 100;
Heap[o, f] := x;
**Semantics (defined by translation into Boogie)**

\[ o := \text{new } C \equiv \ldots o.\mu := \bot \ldots \]

**share o between L and H \equiv**
- \text{assert CanWrite}(o,\mu) \land o.\mu = \bot;
- \text{assert } L \ll H;
- \text{havoc } \mu; \text{ assume } L \ll \mu \ll H;
- o.\mu := \mu;
- \text{Exhale } \text{MonitorInv}(o);

**acquire o \equiv**
- \text{assert CanRead}(o,\mu);
- \text{assert maxlock } \ll o.\mu;
- Held := Held \cup \{o\};
- \text{Inhale } \text{MonitorInv}(o);

**release o \equiv**
- \text{assert } o \in \text{Held};
- \text{Exhale } \text{MonitorInv}(o);
- Held := Held – \{o\};
Exhale and Inhale

- Defined by structural induction
- For expression P without permission predicates

- \( \text{Exhale} \ P \equiv \text{assert} \ P \)
- \( \text{Inhale} \ P \equiv \text{assume} \ P \)

- \( \text{Exhale acc}(o.f, p) \equiv \)
  - \( \text{assert} \ \text{Mask}[o,f] \geq p; \)
  - \( \text{Mask}[o,f] := \text{Mask}[o,f] - p; \)

- \( \text{Inhale acc}(o.f, p) \equiv \)
  - if (\( \text{Mask}[o,f] == 0 \)) { \text{havoc} \ \text{Heap}[o,f]; } 
  - \( \text{Mask}[o,f] := \text{Mask}[o,f] + p; \)
Boogie encoding
Try it for yourself

- Chalice (and Boogie) available as open source: http://boogie.codeplex.com

- Spec# also available as open source under academic license: http://specsharp.codeplex.com