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

Summary, so far

- 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

Microsoft Research

Today's lecture

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

Microsoft Research

demo

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:

```
method M() …
{
  acquire a;
  acquire b;
  …
}
```

```
method N() …
{
  acquire b;
  acquire a;
  …
}
```

Preventing 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

- 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, $\underline{<<}$) with a bottom element ⊥
- << is the strict version of $\underline{<<}$
- 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

Microsoft Research

Example revisited

```
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;
  …
}
```

- With these preconditions, both methods verify
- The conjunction of the preconditions is false, so the methods can never be invoked at the same time

Setting the wait order

- Recall, the wait level of an object o is stored in the ghost field o.mu
- Initially, the .mu field is ⊥
- The .mu field is set by the share statement:

```
share o between L and H;
```

  picks some wait level strictly between L and H, and sets o.mu to that level
- Provided L << H and neither denotes an extreme element, such a wait level exists, since the order is dense
- `share o;` means

```
share o between maxlock and ;
```

Microsoft Research

demo

OwickiGriesCounterD

Deadlock prevention

Microsoft Research

demo

DiningPhilosophers

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 $(\forall \ell \in \text{Held} \bullet \ell.\text{mu} \ll X)$, so uttering `maxlock` has the effect of reading many .mu fields
- We either need `rd(maxlock)`, or

Microsoft Research

Deadlocks when joining

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

```
method N() …
{
  acquire a;
  …
  release a;
}
```

- Include threads in wait order

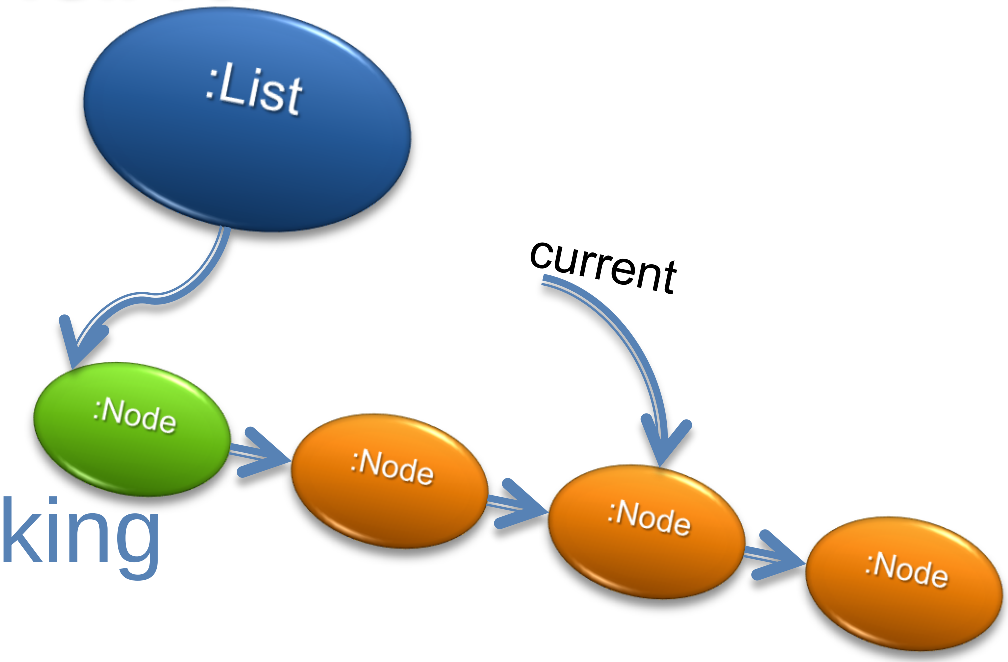
Microsoft Research

Thread levels

- `fork tk := o.M() between L and H;` picks a level θ between L and H, and then sets tk.mu to θ
- The precondition of o.M() is checked, substituting θ as the value of any occurrence of `maxlock`
- `maxlock << X` now means $(\forall \ell \in \text{Held} \bullet \ \ell.\text{mu} << X) \ \wedge \theta << X$ where θ is the one for the current thread
- `join tk;` requires `maxlock` << tk.mu
- without between clause, θ is picked as just barely above `maxlock` of the forking thread

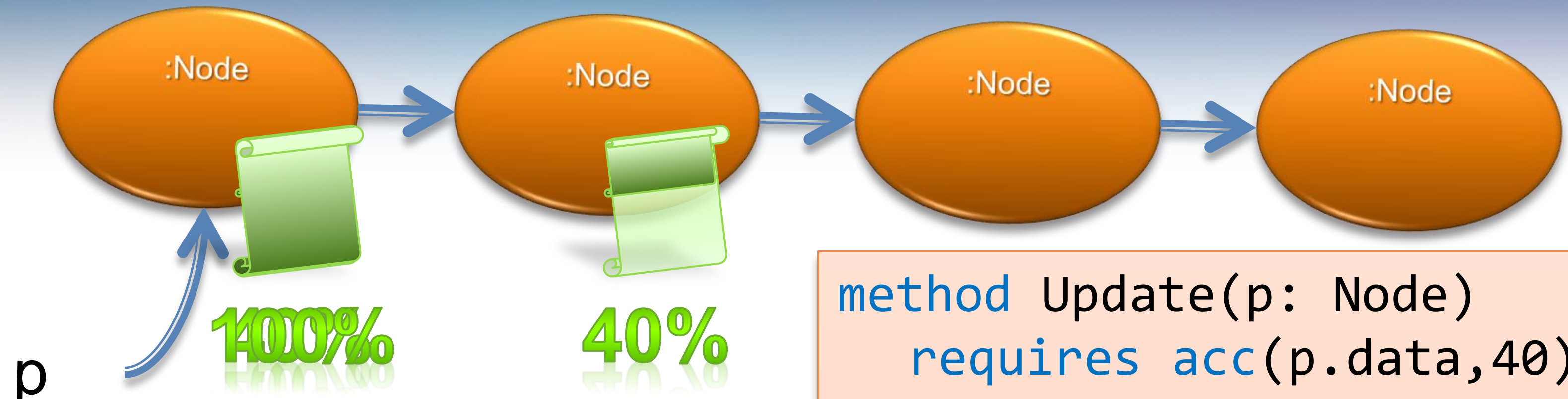
demo

HandOverHand



Fine-grained locking

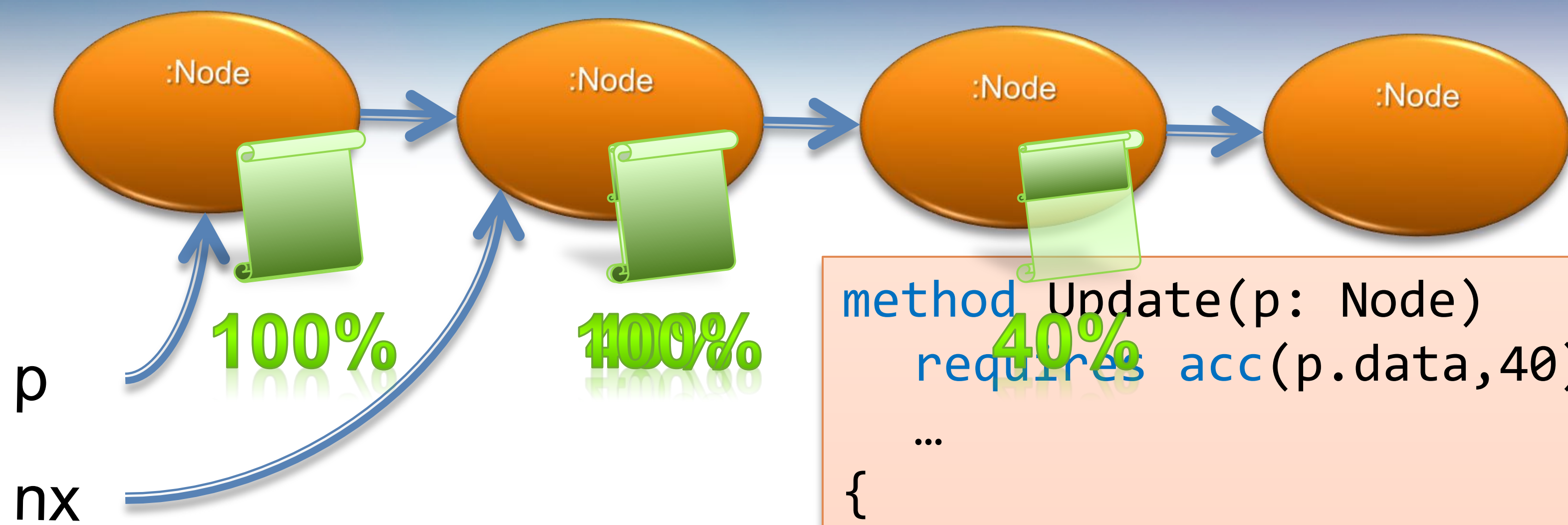
Hand-over-hand locking: the idea




```
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;
   }
   release p;
}
```

```
invariant
   acc(data,60) && … &&
   (next != null ==>
        acc(next.data,40) &&
        data <= next.data);
```

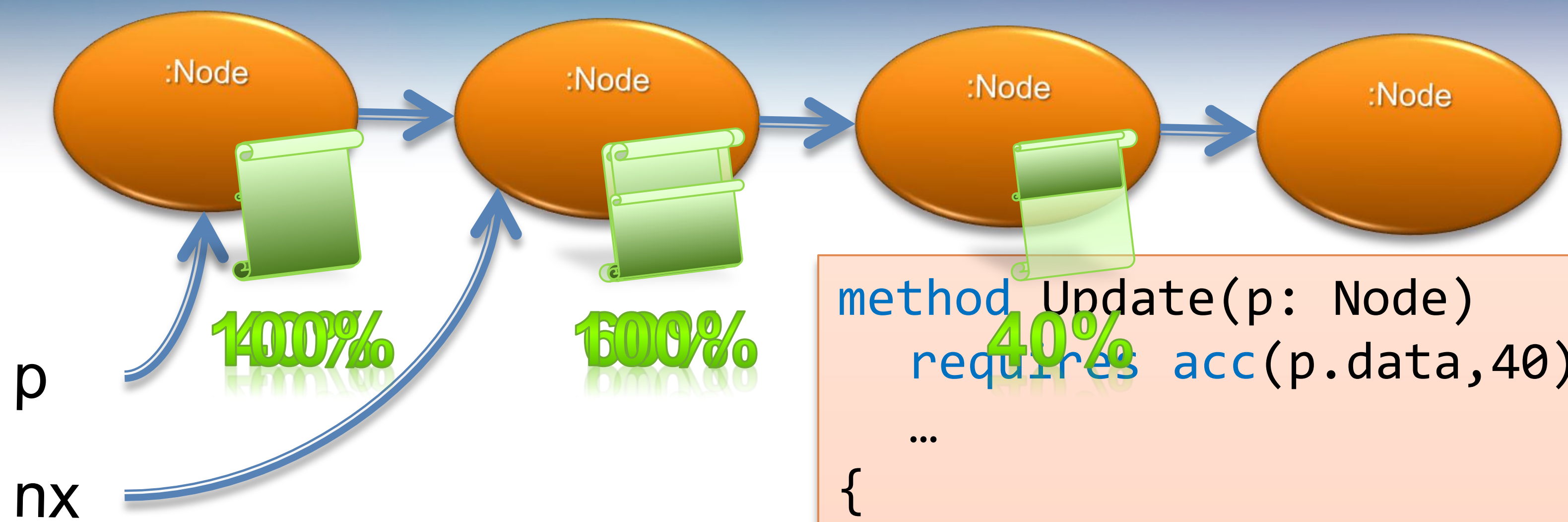
Hand-over-hand locking: the idea




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

```
invariant
   acc(data,60) && … &&
   (next != null ==>
      acc(next.data,40) &&
      data <= next.data);
```

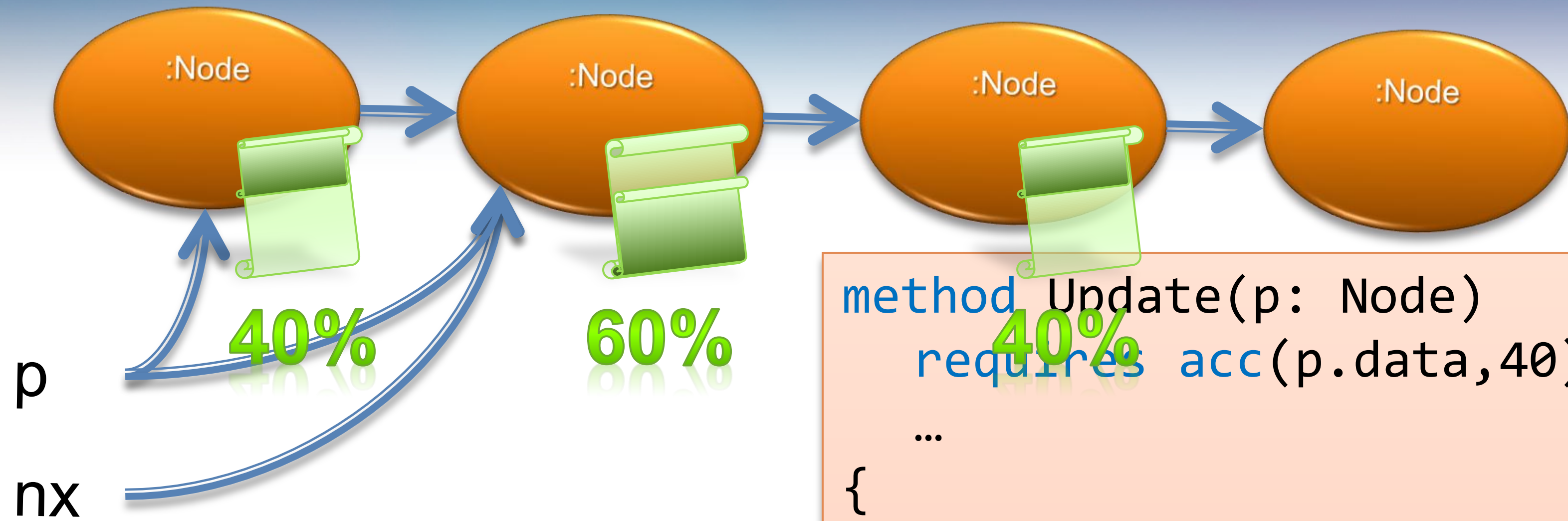
Hand-over-hand locking: the idea




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

```
invariant
   acc(data,60) && … &&
   (next != null ==>
       acc(next.data,40) &&
       data <= next.data);
```

Hand-over-hand locking: the idea

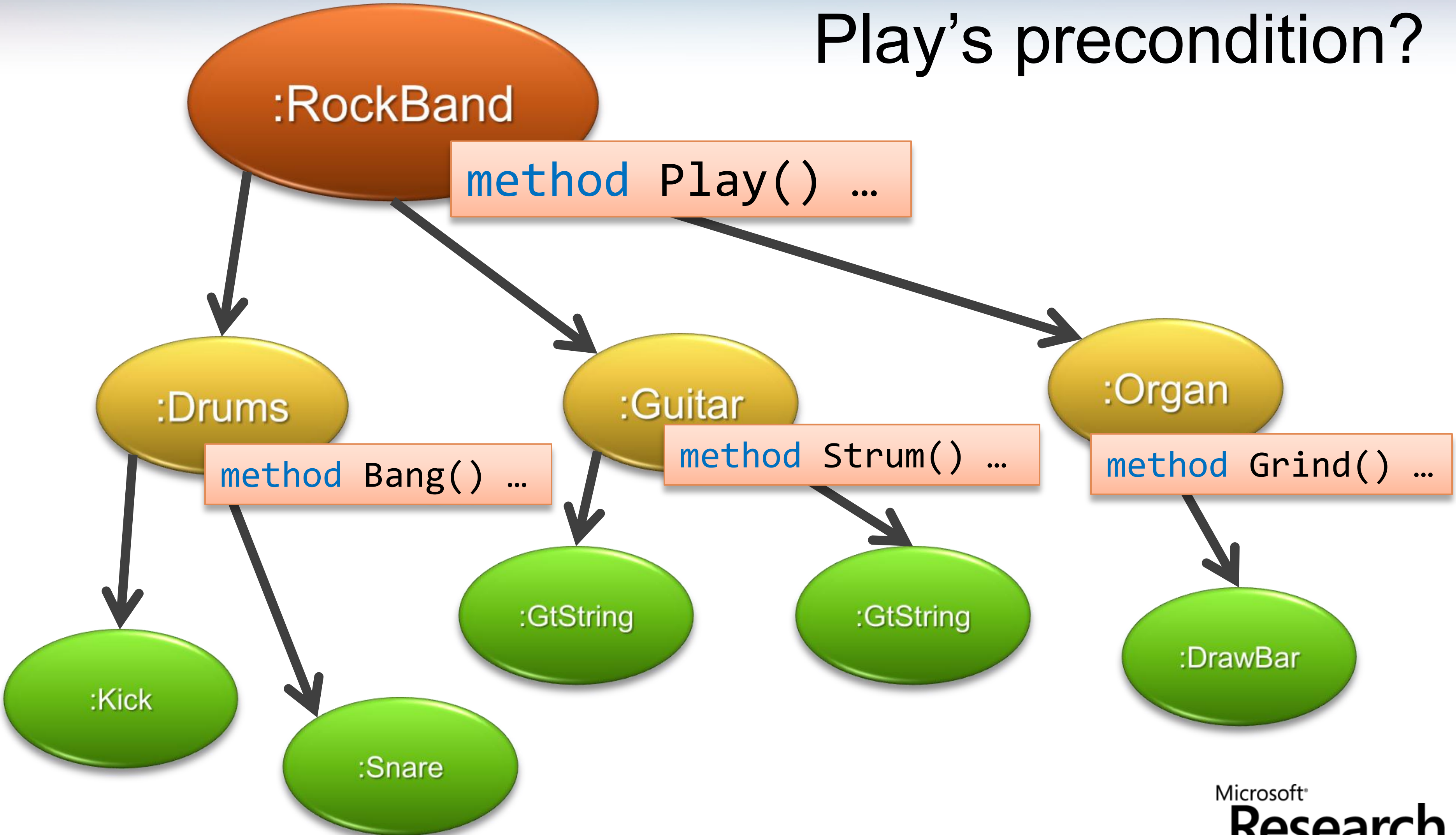



```
method Update(p: Node)
   requires acc(p.data,40)
   …
{
   acquire p;
   while (p.next != null) … {
       var nx := p.next;
       acquire nx;
       nx.data := nx.data + 1;
       release p;
       p := nx;
   }
   release p;
}
```

```
invariant
   acc(data,60) && … &&
   (next != null ==>
       acc(next.data,40) &&
       data <= next.data);
```

Abstraction

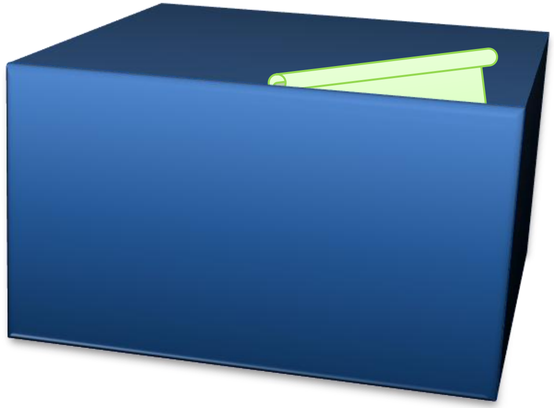
- What permissions to include in method Play's precondition?



Microsoft Research

Predicates

- Named container of permissions
-
```
class C
{
  predicate P {…}
  …
}
```



-
```
fold P;
```
-
```
unfold P;
```

Microsoft Research

Microsoft Research

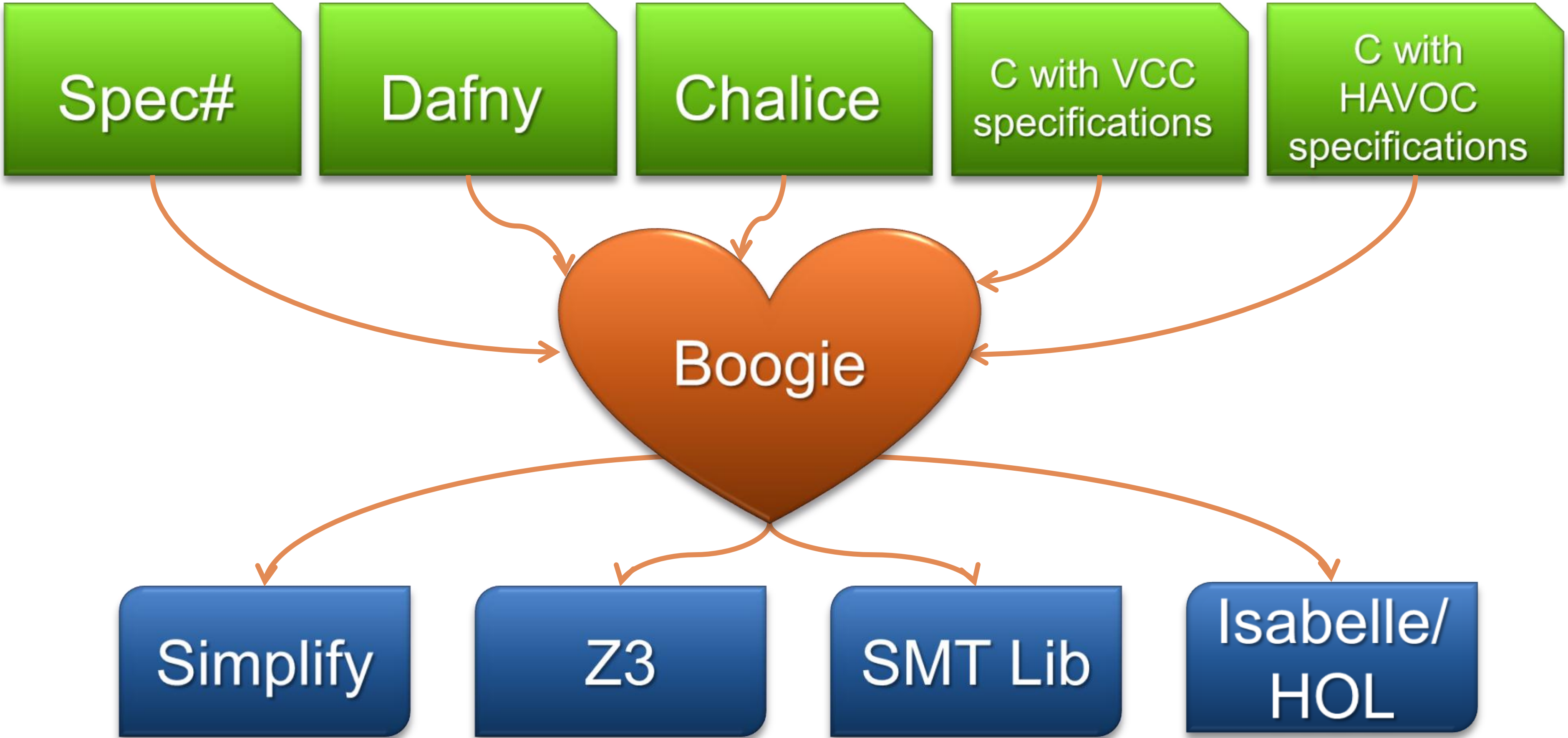
demo

RockBand

Predicates

Boogie

- Intermediate verification language
- Verification engine



Microsoft Research

Boogie language

- First-order mathematical declarations
  - **type**
  - **const**
  - **function**
  - **axiom**
- Imperative declarations
  - **var**
  - **procedure**
  - **implementation**

Boogie statements

- x := E
- **havoc** x
- **assert** E
- **assume** E
- …

- Useful idiom:
  - **havoc** x; **assume** P(x);
  - “set x to a value such that P(x) holds”

Microsoft Research

Weakest preconditions

For any command S and post-state predicate Q, 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

- wp(x := E, Q) = Q[E / x]
- wp(**havoc** x, Q) = $(\forall x \bullet Q)$
- wp(**assert** P, Q) = $P \wedge Q$
- wp(**assume** P, Q) = $P \Rightarrow Q$
- wp(S ; T, Q) = wp(S, wp(T, Q))

Modeling Chalice's memory and permissions in Boogie

- **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 := **new** C ≡ … o.mu := ⊥ …

share o **between** L **and** H ≡

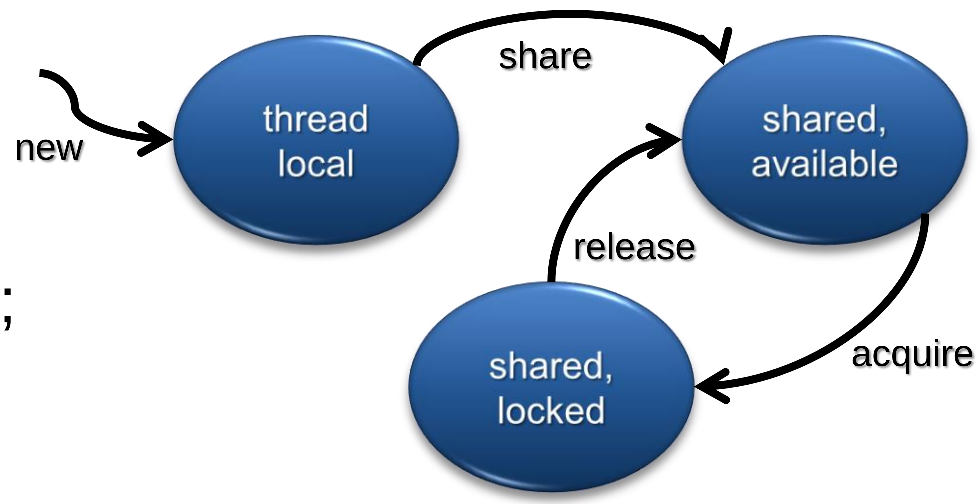
assert CanWrite(o,mu) ∧ o.mu = ⊥;
assert L << H;
havoc μ; **assume** L << μ << H;
o.mu := μ;
Exhale MonitorInv(o);

acquire o ≡

assert CanRead(o,mu);
assert maxlock << o.mu;
Held := Held ∪ {o};
Inhale MonitorInv(o);

release o ≡

assert o ∈ Held;
Exhale MonitorInv(o);
Held := Held – {o};



Exhale and Inhale

- Defined by structural induction
- For expression P without permission predicates
  - *Exhale* P ≡ **assert** P
  - *Inhale* P ≡ **assume** P
- *Exhale* acc(o.f, p) ≡
  assert Mask[o,f] ≥ p;
  Mask[o,f] := Mask[o,f] – p;
- *Inhale* acc(o.f, p) ≡
  if (Mask[o,f] == 0) { **havoc** Heap[o,f]; }
  Mask[o,f] := Mask[o,f] + p;

demo

Inc

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