Specification and Verification of Object-Oriented Software

K. Rustan M. Leino

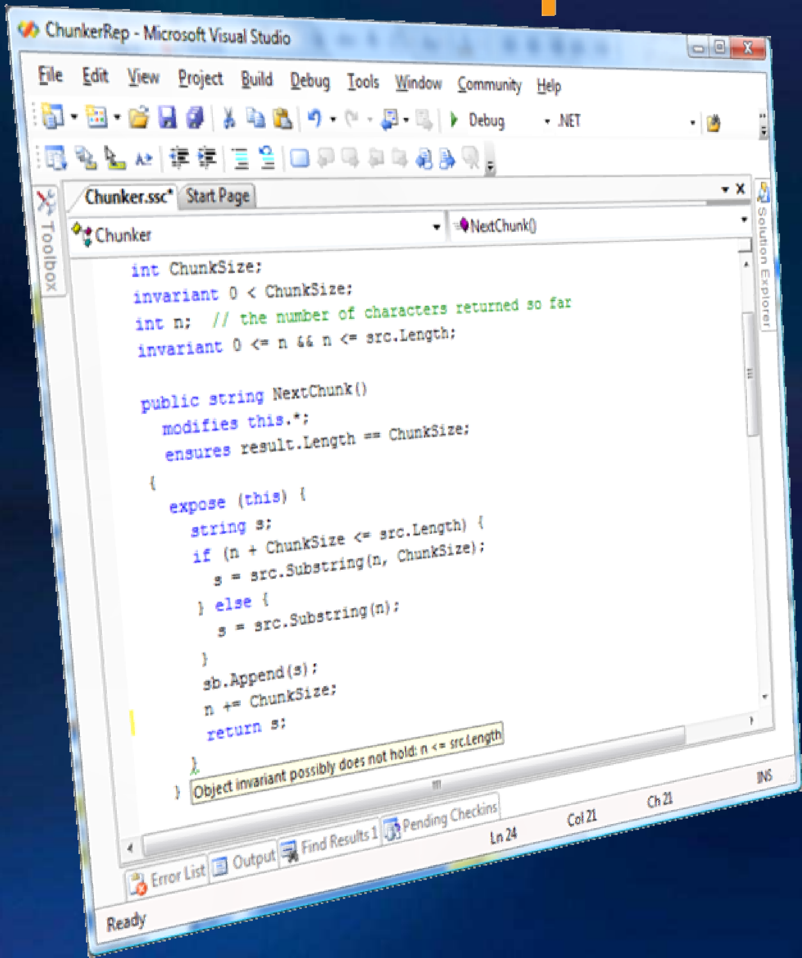
Research in Software Engineering (RiSE)
Microsoft Research, Redmond, WA

part 0
International Summer School Marktoberdorf
Marktoberdorf, Germany
6 August 2008

Contents

- Theory and techniques for building a basic program verifier for a language with references to dynamically allocated objects
- A specification style and encoding thereof

Motivation: Spec# demo



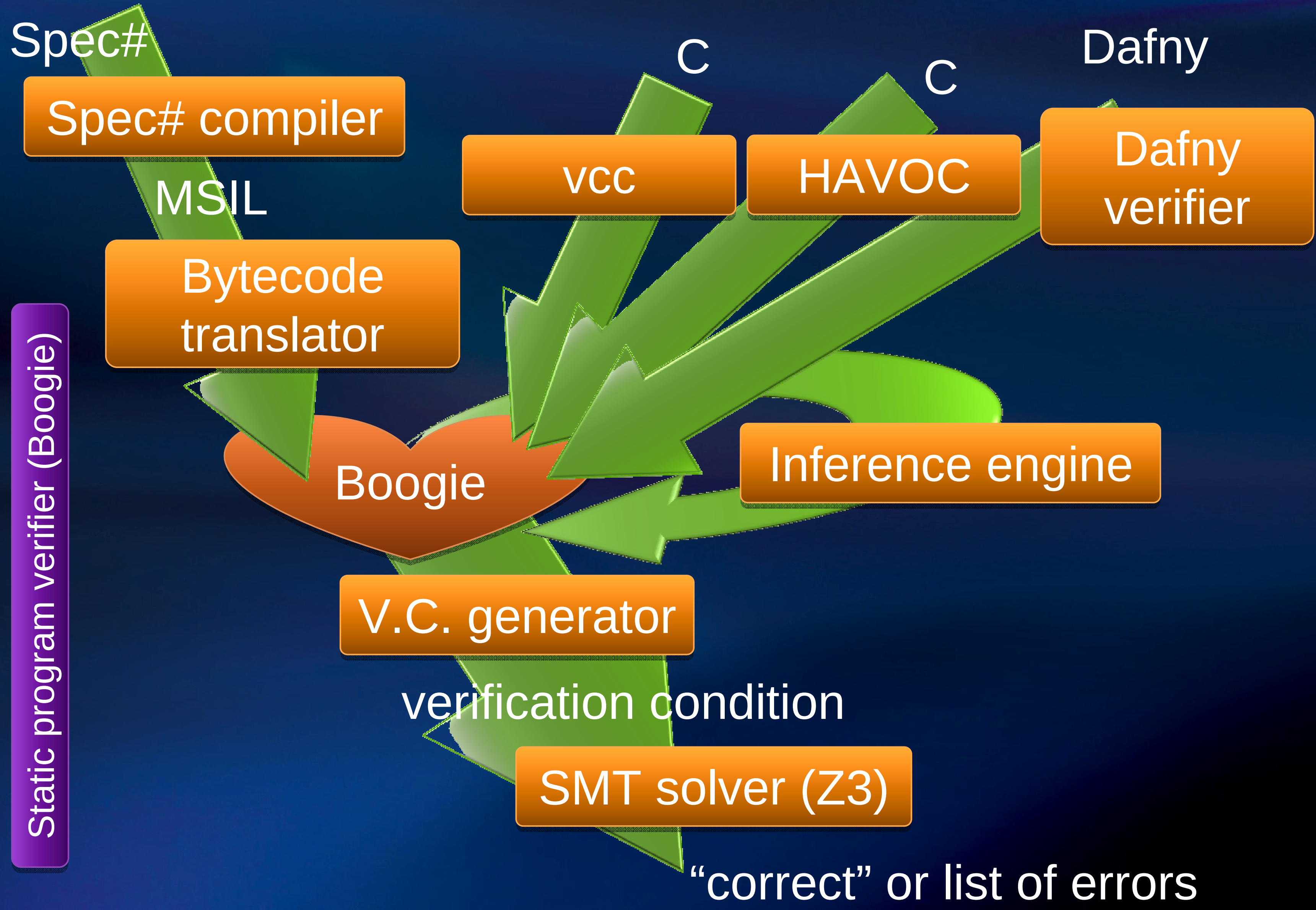
Basic verifier architecture

Source language

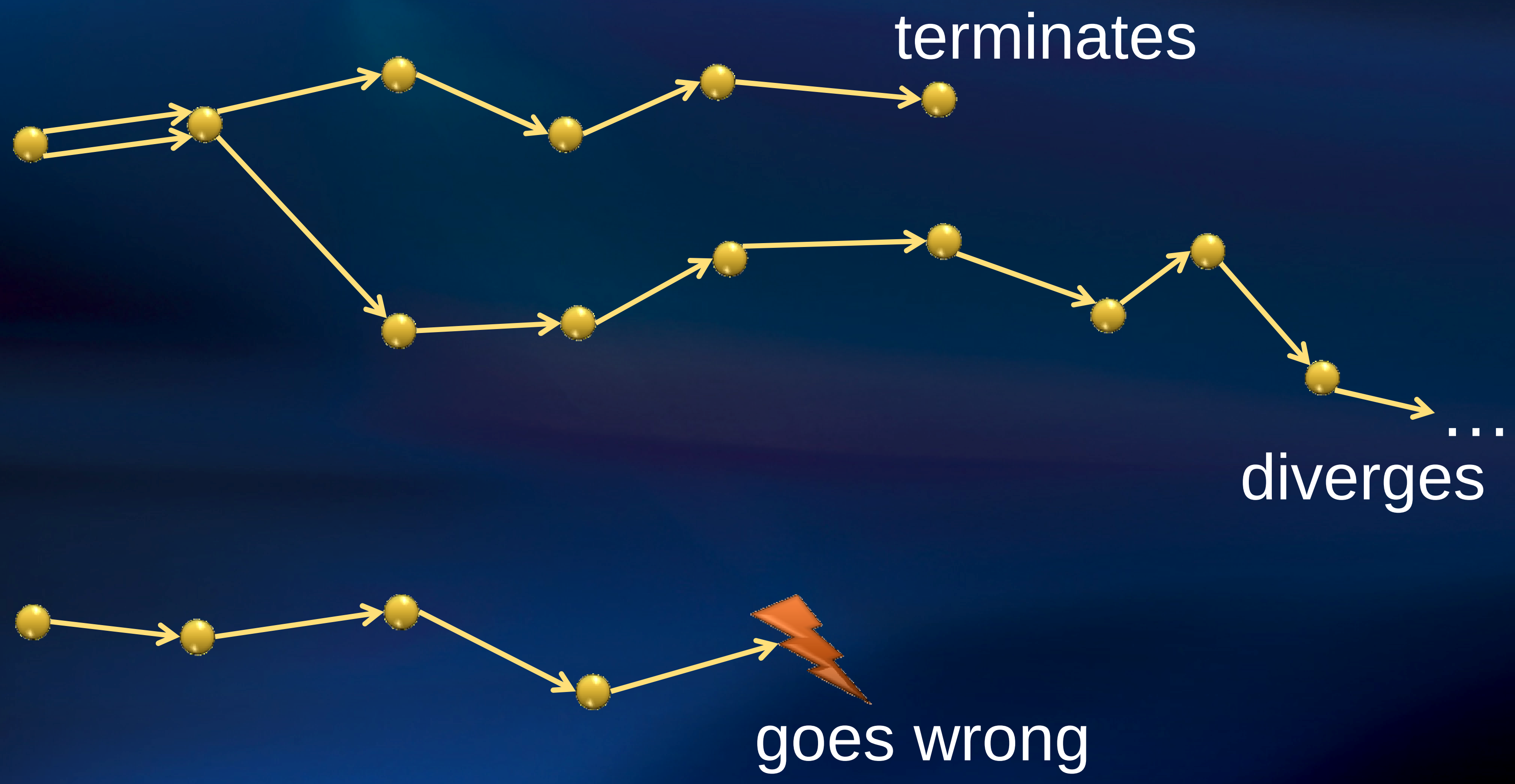
Intermediate verification language

Verification condition
(logical formula)

Verification architecture



Modeling execution traces

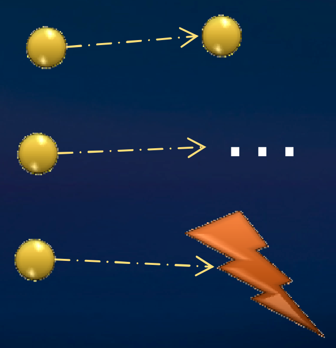


States and execution traces

- State
  - Cartesian product of variables

(x: int, y: int, z: bool)

- Execution trace
  - Nonempty finite sequence of states
  - Infinite sequence of states
  - Nonempty finite sequence of states followed by special error state



Microsoft Research

Commands

- A *command* describes a set of execution traces
- A command is *deterministic* if it describes at most one trace from every initial state
  - Spec#, sequential Java, ML, Haskell

  otherwise, it is *nondeterministic*
  - C, Modula-3, Erlang, Occam
- A command is *total* if it describes at least one trace from every initial state
  - Dijkstra's *Law of the Excluded Miracle*

  otherwise, it is *partial*
  - Juno-2, LIM, Boogie

Microsoft Research

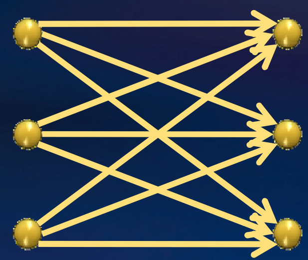
Note, example languages do not necessarily fall squarely into the shown category.

Command language

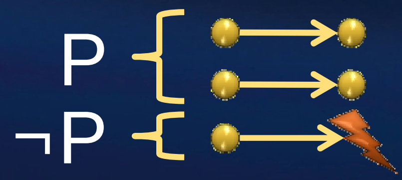
- x := E
  - x := x + 1

  - x := 10

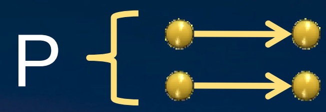
- havoc x



- assert P



- assume P



Solid lines indicate traces whose length is 1

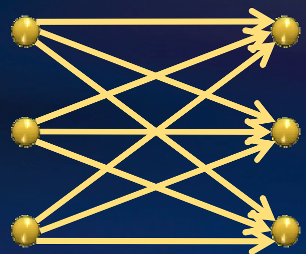
Dotted lines indicate traces whose length may be greater than 1

Command language

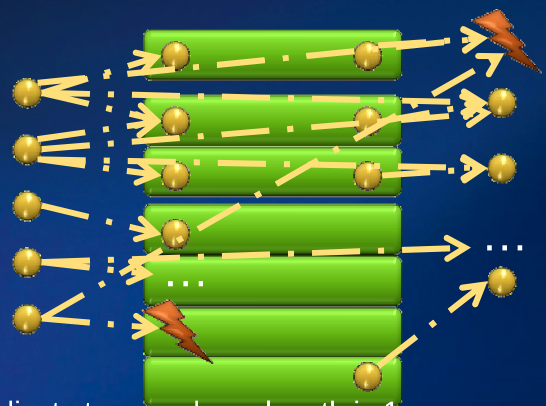
- x := E
  - x := x + 1

  - x := 10

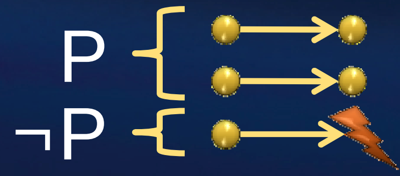
- havoc x



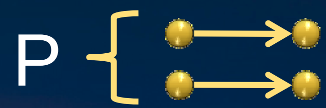
- S ; T



- assert P



- assume P

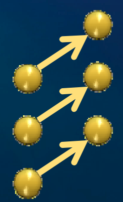


Solid lines indicate traces whose length is 1

Dotted lines indicate traces whose length may be greater than 1

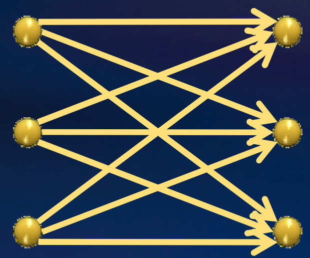
Command language

- x := E
  - x := x + 1



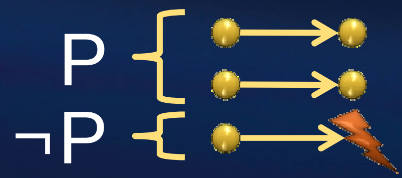
  - x := 10

- havoc x

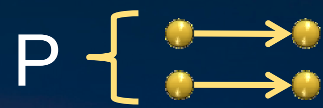


- S ; T

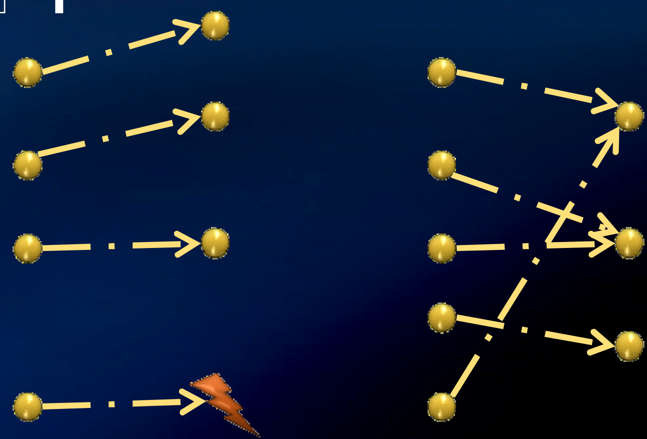
- assert P



- assume P



- S □ T



Solid lines indicate traces whose length is 1

Dotted lines indicate traces whose length may be greater than 1

Specification and Verification of Object-Oriented Software

K. Rustan M. Leino

Research in Software Engineering (RiSE)
Microsoft Research, Redmond, WA

part 1
International Summer School Marktoberdorf
Marktoberdorf, Germany
7 August 2008

Basic verifier architecture

Source language

Intermediate verification language

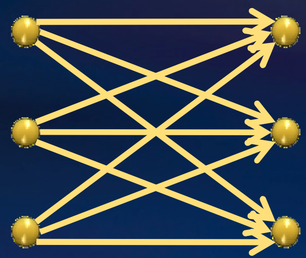
Verification condition
(logical formula)

Command language

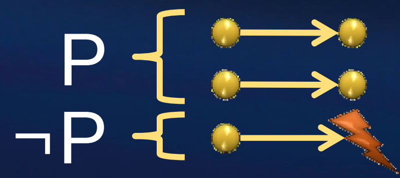
- x := E
  - x := x + 1

  - x := 10

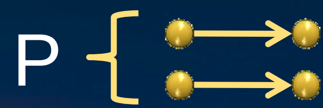
- havoc x



- assert P



- assume P



Solid lines indicate traces whose length is 1

Dotted lines indicate traces whose length may be greater than 1

Command language

- S ; T

Solid lines indicate traces whose length is 1

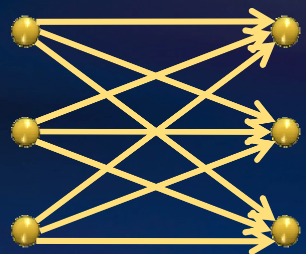
Dotted lines indicate traces whose length may be greater than 1

Command language

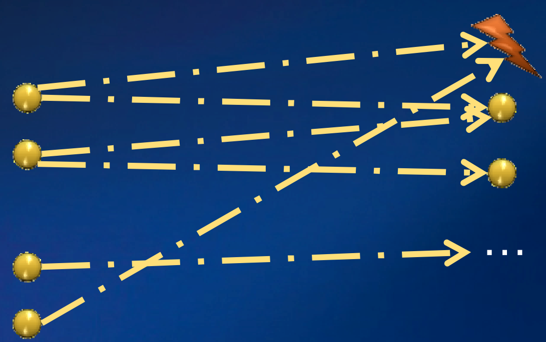
- x := E
  - x := x + 1

  - x := 10

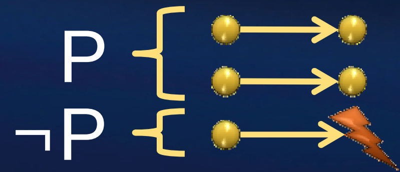
- havoc x



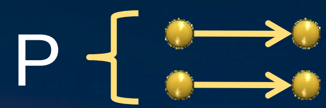
- S ; T



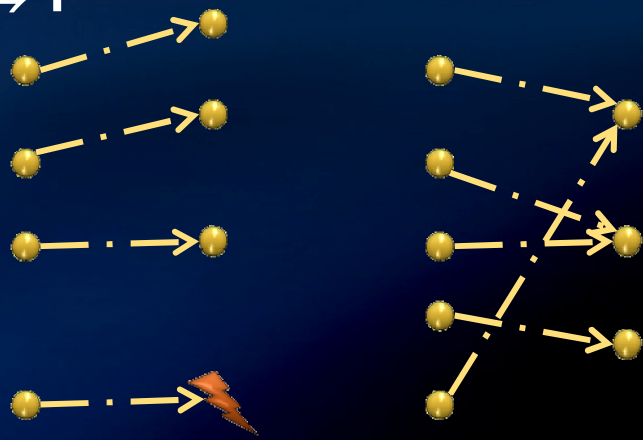
- assert P



- assume P



- S ⇒T



Solid lines indicate traces whose length is 1

Dotted lines indicate traces whose length may be greater than 1

Reasoning about execution traces

- Hoare triple { P } S { Q } says that
  every terminating execution trace of S that starts in a state satisfying P
  - does not go wrong, and
  - terminates in a state satisfying Q
- Given P and Q, what is the largest S' satisfying {P} S' {Q} ?
  - to check {P} S {Q}, check $S \subseteq S'$

Reasoning about execution traces

- Hoare triple $\{ P \}\ S\ \{ Q \}$ says that every terminating execution trace of S that starts in a state satisfying P
  - does not go wrong, and
  - terminates in a state satisfying Q
- Given S and Q, what is the weakest P' satisfying {P'} S {Q} ?
  - P' is called the *weakest precondition* of S with respect to Q, written wp(S, Q)
  - to check {P} S {Q}, check $P \Rightarrow P'$

Microsoft Research

Reasoning about execution traces

- Hoare triple { P } S { Q } says that every terminating execution trace of S that starts in a state satisfying P
  - does not go wrong, and
  - terminates in a state satisfying Q
- Given P and S, what is the strongest Q' satisfying {P} S {Q'} ?
  - to check {P} S {Q}, check Q' ⇒ Q

not well defined

For example, what is the strongest Q' satisfying

{ true } assert false { Q' } ? (there isn't one)

Microsoft Research

Checking correctness with sp

{ x < 10 } x := x + 1; assert P(x); x := x + 1 { true }

- sp(x < 10, x := x + 1) =
  x ≤ 10
- need to check the assert:
  x ≤ 10 ⇒ P(x)
- sp(x ≤ 10, assert P(x)) =
  x ≤ 10 ∧ P(x)
- sp(x ≤ 10 ∧ P(x), x := x + 1) =
  x ≤ 11 ∧ P(x-1)
- check: x ≤ 11 ∧ P(x-1) ⇒ true

Microsoft Research

Checking correctness with wp

{ x < 10 } x := x + 1; assert P(x); x := x + 1 { true }

= wp(x := x + 1, true)

true

= wp(assert P(x), true)

P(x)

= wp(x := x + 1, P(x))

P(x+1)

check: x < 10 $\Rightarrow$ P(x+1)

Microsoft Research

Advanced: wp, wlp, sp, Galois

- sp treats assert as it treats assume
- wlp is like wp but treats assert as assume
- wlp and sp form a Galois connection:

  $[\underline{sp_S}(P) \Rightarrow Q] \quad \Leftrightarrow \quad [P \Rightarrow \underline{wlp_S}(Q)]$

  lower adjoint — upper adjoint

- one adjoint uniquely determines the other
- an upper adjoint is universally conjunctive
- wp is not univerally conjunctive ($wp_{\text{assert false}}(\text{true}) \neq \text{true}$)
- so, wp has no lower adjunct
- that is, there is no function f such that

  $[f(P) \Rightarrow Q] \quad \Leftrightarrow \quad [P \Rightarrow wp_S(Q)]$

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))
- wp(S $\Rightarrow$ T, Q) = wp(S, Q) $\wedge$ wp(T, Q)

Command correctness

- A command S is correct iff

    wp(S, true)

  is valid

Microsoft Research

Structured if statement

if E then S else T end =

assume E; S

$\Rightarrow$

assume ¬E; T

Microsoft Research

Dijkstra's guarded command

if E → S | F → T fi =

assert E ∨ F;

(

assume E; S

⇒

assume F; T

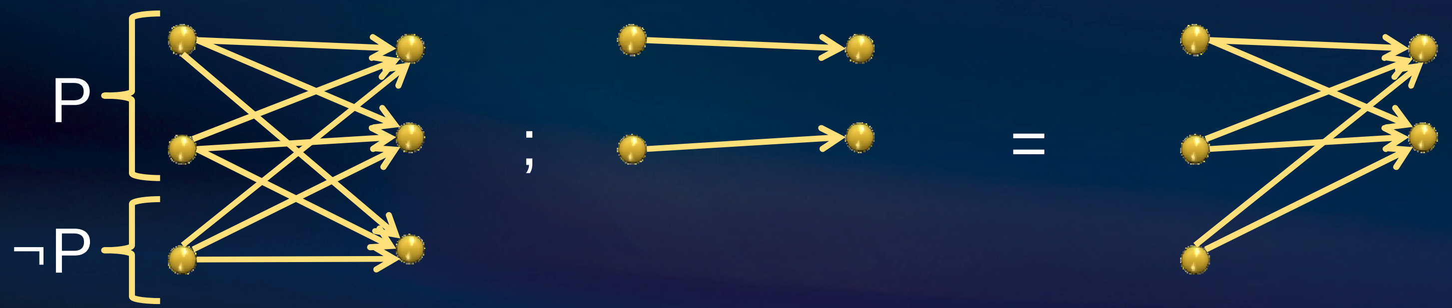
)

Microsoft Research

Picking any good value

assign x such that P =

havoc x; assume P



Example:

assign x such that x*x = y

- What if we want assign to be total?

assert (∃x • P); havoc x; assume P

Definedness of expressions

- x := a / b   // possible div-by-0

```
assert b ≠ 0;
x := a / b
```

- x := a + b   // possible overflow

```
assert -2^31 ≤ a + b;
assert a + b < 2^31;
x := a + b
```

- x := a + b   // use modular arith.

```
x := PlusWrap(a, b)
```

Complex data values: Arrays

- An *array* is a map from indices to values
- array update is map update:
  - a[j] := E
    means
    a := a[j → E]
- apply/select/get/rd and update/store/set/wr follow the familiar properties:
  - $(\forall a,j,k,x \bullet \; j = k \Rightarrow a[j \to x][k] = x)$
  - $(\forall a,j,k,x \bullet \; j \neq k \Rightarrow a[j \to x][k] = a[k])$

Microsoft Research

While loop with loop invariant

```
while E
    invariant J
do
    S
end
```

= ?

Homework

Microsoft Research

Specification and Verification of Object-Oriented Software

K. Rustan M. Leino

Research in Software Engineering (RiSE)
Microsoft Research, Redmond, WA

part 2
International Summer School Marktoberdorf
Marktoberdorf, Germany
8 August 2008

While loop with loop invariant

while E invariant J do S end

= assert J; ← check that the loop invariant holds initially

havoc x; assume J; } "fast forward" to an arbitrary iteration of the loop

(assume E; S; assert J; assume false

 □ assume ¬E

)

↑ check that the loop invariant is maintained by the loop body

where x denotes the assignment targets of S

Microsoft Research

wp of while

- wp(while E invariant J do S end, Q) =
  $J \wedge$
  $(\forall x \bullet\ J \wedge E \ \Rightarrow\ wp(S, J)\ ) \wedge$
  $(\forall x \bullet\ J \wedge \neg E\ \Rightarrow Q\ )$
- assert J;
  havoc x; assume J;
  (assume E; S; assert J; assume false
  □   assume ¬E
)

wp calculation for while

wp(havoc x; assume J; assume E; S; assert J; assume false, Q)

= wp(havoc x; assume J; assume E; S; assert J, false ⇒ Q)

= wp(havoc x; assume J; assume E; S; assert J, true)

= wp(havoc x; assume J; assume E; S, J ∧ true)

= wp(havoc x; assume J; assume E; S, J)

= wp(havoc x; assume J; assume E, wp(S, J))

= wp(havoc x, assume J, E ⇒ wp(S, J))

= wp(havoc x, J ⇒ (E ⇒ wp(S, J)))

= wp(havoc x, J ∧ E ⇒ wp(S, J))

= (∀ x • J ∧ E ⇒ wp(S, J))

Loop termination

```
while E
    invariant J
    decreases B
do
    S
end
```

= ?

Homework

Microsoft Research

Example: Mutual exclusion

- monitor m { var x; invariant x ≤ y; }
- acquire m
- release m

Homework

Procedures

- A *procedure* is a user-defined command
- procedure M(x, y, z) returns (r, s, t)
  requires P
  modifies g, h
  ensures Q

Procedure example

- procedure Inc(n) returns (b)
  requires $0 \leq n$
  modifies g
  ensures $g = \text{old}(g) + n \;\wedge\; b = (g \text{ even})$

Procedure calls

- procedure M(x, y, z) returns (r, s, t)
  requires P modifies g, h ensures Q
- call a, b, c := M(E, F, G)

```
=  x' := E;  y' := F;  z' := G;
   assert P';
   g0 := g;  h0 := h;
   havoc g, h, r', s', t';
   assume Q';
   a := r';  b := s';  c := t'
```

where
•x', y', z', r', s', t', g0, h0 are fresh variables
•P' is P with x',y',z' for x,y,z
•Q' is Q with x',y',z',r',s',t',g0,h0 for x,y,z,r,s,t, old(g), old(h)

Microsoft Research

Procedure implementations

- procedure M(x, y, z) returns (r, s, t)
  requires P modifies g, h ensures Q
- implementation M(x, y, z) returns (r, s, t) is S

  correct if:

```
assume P;
g0 := g;  h0 := h;
S;
assert Q'
```

  is correct

where
•g0, h0 are fresh variables
•Q' is Q with g0,h0 for old(g), old(h)

syntactically check that S assigns only to g,h

Microsoft Research

Translating a source language

Microsoft Research

Translation functions

- The meaning of source statement S is given by Tr[[S]]
  - Tr : source-statement → command
- When defined, the meaning of a source expression E is given by Tr[[E]]
  - Tr : source-expression → expression
- In a context permitted to read set of locations R, source expression E is defined when Df_R[[E]] holds
  - Df_R : source-expression → boolean expression
  - If R is the universal set, drop the subscript R

Example translations

- Tr[[x := E]] =

```
assert Df[[ E ]];
x := Tr[[ E ]]
```

Example translations

- Tr[[x := E]] = assert Df[[E]]; x := Tr[[E]]
- Df_R[[E / F]] =

  Df_R[[E]] $\wedge$ Df_R[[F]] $\wedge$ Tr[[F]] $\neq$ 0
- Df_R[[E.x]] =

  Df_R[[E]] $\wedge$ Tr[[E]] $\neq$ null $\wedge$
  (Tr[[E]], x) $\in$ R
- Df_R[[E && F]] =

  Df_R[[E]] $\wedge$ (Tr[[E]] $\Rightarrow$ Df_R[[F]])

Object features

- class C { var x: int; var y: C; … }

- Idea: c.x is modeled as Heap[c, x]
- Details:
  - var Heap
  - const x
  - const y

Microsoft Research

Object features, with types

- class C { var x: int; var y: C; … }
- Idea: c.x is modeled as Heap[c, x]
- Details:
  - type Ref
  - type Field
  - var Heap: Ref × Field → ?
  - const x: Field
  - const y: Field

Object features, with types

- class C { var x: int; var y: C; … }
- Idea: c.x is modeled as Heap[c, x]
- Details:
  - type Ref;
  - type Field α;
  - var Heap: ∀α. Ref × Field α → α;
  - const x: Field int;
  - const y: Field Ref;
- Heap[c, x] has type int

Microsoft Research

Object features

- class C { var x: int; var y: C; … }
- Translation into Boogie:
  - type Ref;
  - type Field α;
  - type HeapType = ⟨α⟩[Ref, Field α] α;
  - var Heap: HeapType;
  - const unique C.x: Field int;
  - const unique C.y: Field Ref;

Accessing the heap

- introduce:

  const null: Ref;

- Df_R[[E.x]] =

  Df_R[[E]] ∧ Tr[[E]] ≠ null ∧
  (Tr[[E]], x) ∈ R

- Tr[[E.x := F]] =

  assert Df[[E]] ∧ Df[[F]] ∧ Tr[[E]] ≠ null;
  Heap[Tr[[E]], x] := Tr[[F]]

Microsoft Research

Object creation

- introduce:

  const unique alloc: Field bool;

- Tr[[c := new C]] =

  havoc c;
  assume c ≠ null ∧ ¬Heap[c, alloc];
  Heap[c, alloc] := true

Microsoft Research

Object creation, advanced

- introduce:

  const unique alloc: Field boo

- Tr[[c := new C]] =

  havoc c;
  assume c ≠ null Heap[ ];
  assume dtype(c) = C;
  assume Heap[c, x] = 0 ∧ Heap[c, y] = null;
  Heap[c, alloc] := tr

dynamic type information

initial field values

Microsoft Research

Fresh

- $Df_R[[\ fresh(S)\ ]] =$

  $Df_R[[\ S\ ]]$
- $Tr[[\ fresh(S)\ ]] =$

  $(\forall o \bullet\ o \in Tr[[\ S\ ]] \Rightarrow$

  $o = null \lor \neg old(Heap)[o, alloc])$

Microsoft Research

Properties of the heap

- introduce:

  axiom (∀ h: HeapType, o: Ref, f: Field Ref •
  o ≠ null ∧ h[o, alloc]
  ⇒
  h[o, f] = null ∨ h[h[o,f], alloc]);

Properties of the heap

- introduce:

  function IsHeap(HeapType) returns (bool);

- introduce:

  axiom (∀ h: HeapType, o: Ref, f: Field Ref •
  IsHeap(h) ∧ o ≠ null ∧ h[o, alloc]
  ⇒
  h[o, f] = null ∨ h[h[o,f], alloc]);

- introduce: assume IsHeap(Heap)
  after each Heap update; for example:
  Tr[[E.x := F]] =
  assert …; Heap[…] := …;
  assume IsHeap(Heap)

Microsoft Research

Demo

- Example0.dfy

Microsoft Research

Microsoft Research

Specification and Verification of Object-Oriented Software

K. Rustan M. Leino

Research in Software Engineering (RiSE)
Microsoft Research, Redmond, WA

part 3
International Summer School Marktoberdorf
Marktoberdorf, Germany
9 August 2008

Methods, basics

- method M(x: X) returns (y: Y)
  { Stmt }

- procedure M(this: Ref, x: Ref) returns (y: Ref);
  requires this ≠ null;
- implementation M(this: Ref, x: Ref)
  returns (y: Ref)
  { Tr[[Stmt]] }

Microsoft Research

Method pre/post specifications

- method M(x: X) returns (y: Y)
  requires P; ensures Q;

- procedure M(this: Ref, x: Ref) returns (y: Ref);
  requires Df[[P]] $\wedge$ Tr[[P]];
  ensures Df[[Q]] $\wedge$ Tr[[Q]];

Microsoft Research

Method modifies clauses

- method M(x: X) returns (y: Y)
  modifies S;
- procedure M(this: Ref, x: Ref) returns (y: Ref);
  requires Df[[S]];
  modifies Heap;
  ensures (∀⟨α⟩ o: Ref, f: Field α •
  o ≠ null ∧ old(Heap)[o,alloc] ⇒
  Heap[o,f] = old(Heap)[o,f] ∨
  (o,f) ∈ old(Tr[[S]])
  );

Microsoft Research

Method modifies clauses: example

- method M(x: X) returns (y: Y)
  modifies this.*, x.s, this.p.t;
- procedure M(this: Ref, x: Ref) returns (y: Ref);
  requires Df[[S]];
  modifies Heap;
  ensures (∀⟨α⟩ o: Ref, f: Field α •
  o ≠ null ∧ old(Heap)[o,alloc] ⇒
  Heap[o,f] = old(Heap)[o,f] ∨
  o = this ∨
  (o = x ∧ f = s) ∨
  (o = old(Heap)[this,p] ∧ f = t));

Methods, boilerplate

- method M(x: X) returns (y: Y)
- procedure M(this: Ref, x: Ref) returns (y: Ref);

```
    requires IsHeap(Heap);
    requires this ≠ null ∧ Heap[this, alloc];
    requires x = null ∨ Heap[x, alloc];
    ensures IsHeap(Heap);
    ensures y = null ∨ Heap[y, alloc];
    ensures (∀o: Ref •
            old(Heap)[o,alloc] ⇒ Heap[o,alloc]);
```

"Free-conditions"

- The source language offers no way to violate these conditions
- procedure M(this: Ref, x: Ref) returns (y: Ref);

  free requires IsHeap(Heap);
  free requires this ≠ null ∧ Heap[this, alloc];
  free requires x = null ∨ Heap[x, alloc];

  free ensures IsHeap(Heap);
  free ensures y = null ∨ Heap[y, alloc];
  free ensures (∀o: Ref •
  old(Heap)[o,alloc] ⇒ Heap[o,alloc]);

Methods, putting it all together

- method M(x: X) returns (y: Y)
  requires P; modifies S; ensures Q;
  { Stmt }
- procedure M(this: Ref, x: Ref) returns (y: Ref);

```
free requires IsHeap(Heap);
free requires this ≠ null ∧ Heap[this, alloc];
free requires x = null ∨ Heap[x, alloc];
requires Df[[ P ]] ∧ Tr[[ P ]];
requires Df[[ S ]];
modifies Heap;
ensures Df[[ Q ]] ∧ Tr[[ Q ]];
ensures (∀⟨α⟩ o: Ref, f: Field α •
                o ≠ null ∧ old(Heap)[o,alloc] ⇒
                    Heap[o,f] = old(Heap)[o,f]  ∨
                    (o,f) ∈ old( Tr[[ S ]] ));
free ensures IsHeap(Heap);
free ensures y = null ∨ Heap[y, alloc];
free ensures (∀o: Ref •  old(Heap)[o,alloc] ⇒ Heap[o,alloc]);
```

Spec# Chunker.NextChunk translation

```
procedure Chunker.NextChunk(this: ref where $IsNotNull(this, Chunker)) returns ($result: ref where $IsNotNull($result, System.String));
  // in-parameter:  target object
  free requires $Heap[this, $allocated];
  requires ($Heap[this, $ownerFrame] == $PeerGroupPlaceholder || !($Heap[$Heap[this, $ownerRef], $inv] <: $Heap[this, $ownerFrame]) ||
     $Heap[$Heap[this, $ownerRef], $localinv] == $BaseClass($Heap[this, $ownerFrame])) && (forall $pc: ref :: $pc != null && $Heap[$pc, $allocated]
     && $Heap[$pc, $ownerRef] == $Heap[this, $ownerRef] && $Heap[$pc, $ownerFrame] == $Heap[this, $ownerFrame] ==> $Heap[$pc, $inv] ==
     $typeof($pc) && $Heap[$pc, $localinv] == $typeof($pc));
  // out-parameter:  return value
  free ensures $Heap[$result, $allocated];
  ensures ($Heap[$result, $ownerFrame] == $PeerGroupPlaceholder || !($Heap[$Heap[$result, $ownerRef], $inv] <: $Heap[$result, $ownerFrame]) ||
     $Heap[$Heap[$result, $ownerRef], $localinv] == $BaseClass($Heap[$result, $ownerFrame])) && (forall $pc: ref :: $pc != null && $Heap[$pc,
     $allocated] && $Heap[$pc, $ownerRef] == $Heap[$result, $ownerRef] && $Heap[$pc, $ownerFrame] == $Heap[$result, $ownerFrame] ==>
     $Heap[$pc, $inv] == $typeof($pc) && $Heap[$pc, $localinv] == $typeof($pc));
  // user-declared postconditions
  ensures $StringLength($result) <= $Heap[this, Chunker.ChunkSize];
  // frame condition
  modifies $Heap;
  free ensures (forall $o: ref, $f: name :: { $Heap[$o, $f] } $f != $inv && $f != $localinv && $f != $FirstConsistentOwner && (!IsStaticField($f) ||
     !IsDirectlyModifiableField($f)) && $o != null && old($Heap)[$o, $allocated] && (old($Heap)[$o, $ownerFrame] == $PeerGroupPlaceholder ||
     !(old($Heap)[old($Heap)[$o, $ownerRef], $inv] <: old($Heap)[$o, $ownerFrame]) || old($Heap)[old($Heap)[$o, $ownerRef], $localinv] ==
     $BaseClass(old($Heap)[$o, $ownerFrame])) && old($o != this || !(Chunker <: DeclType($f)) || !$IncludedInModifiesStar($f)) && old($o != this || $f
     != $exposeVersion) ==> old($Heap)[$o, $f] == $Heap[$o, $f]);
  // boilerplate
  free requires $BeingConstructed == null;
  free ensures (forall $o: ref :: { $Heap[$o, $localinv] } { $Heap[$o, $inv] } $o != null && !old($Heap)[$o, $allocated] && $Heap[$o, $allocated] ==>
     $Heap[$o, $inv] == $typeof($o) && $Heap[$o, $localinv] == $typeof($o));
  free ensures (forall $o: ref :: { $Heap[$o, $FirstConsistentOwner] } old($Heap)[old($Heap)[$o, $FirstConsistentOwner], $exposeVersion] ==
     $Heap[old($Heap)[$o, $FirstConsistentOwner], $exposeVersion] ==> old($Heap)[$o, $FirstConsistentOwner] == $Heap[$o,
     $FirstConsistentOwner]);
  free ensures (forall $o: ref :: { $Heap[$o, $localinv] } { $Heap[$o, $inv] } old($Heap)[$o, $allocated] ==> old($Heap)[$o, $inv] == $Heap[$o, $inv] &&
     old($Heap)[$o, $localinv] == $Heap[$o, $localinv]);
  free ensures (forall $o: ref :: { $Heap[$o, $allocated] } old($Heap)[$o, $allocated] ==> $Heap[$o, $allocated]) && (forall $ot: ref :: { $Heap[$ot,
     $ownerFrame] } { $Heap[$ot, $ownerRef] } old($Heap)[$ot, $allocated] && old($Heap)[$ot, $ownerFrame] != $PeerGroupPlaceholder ==>
     old($Heap)[$ot, $ownerRef] == $Heap[$ot, $ownerRef] && old($Heap)[$ot, $ownerFrame] == $Heap[$ot, $ownerFrame]) &&
     old($Heap)[$BeingConstructed, $NonNullFieldsAreInitialized] == $Heap[$BeingConstructed, $NonNullFieldsAreInitialized];
```

Microsoft Research

Dafny: an object-based language

- Program ::= Class*
- Class ::= class C { Field* Method* Function* }
- S, T ::=
  - var x;
  - x := E;
  - x := new C;
  - E.f := F;
  - assert E;
  - S T
  - if (E) { S } else { T }
  - while (E) invariant J; { S }
  - call a,b,c := E.M(F, G);

Specifying programs using *dynamic frames* in Dafny

```
class Chunker {
   var src: String;
   var n: int;
   method Init(source: String)
      requires source ≠ null;
      modifies {this};
    {
      this.src := source;
      this.n := 0;
    }
```

For simplicity, in Dafny, modifies clauses are done at the object granularity, not the (object,field) granularity.
In Spec#: this.*
In Dafny: {this}

In Spec#: c = new Chunker(source);
In Dafny: c := new Chunker;
call c.Init(source);

Dynamic frames: [Kassios]

Microsoft Research

Dafny Chunker example (cont.)

```
method NextChunk() returns (s: String)
   modifies {this};
   ensures s ≠ null;
{
   if (this.n + 5 ≤ s.Length) {
      call s := this.src.Substring(this.n, this.n + 5);
   } else {
      call s := this.src.Substring(this.n, s.Length);
   }
   this.n := this.n + s.Length;
}
```

correctness relies on:

this.src ≠ null ∧

0 ≤ this.n ≤ this.src.Length

Microsoft Research

Dafny demo

- Chunker0.dfy

Microsoft Research

Functions

- function Valid() returns (bool) {
  this.src ≠ null ∧
  0 ≤ this.n ∧ this.n ≤ this.src.Length
  }
- method Init(…) …
  ensures this.Valid();
- method NextChunk(…) …
  requires this.Valid();
  ensures this.Valid();

Encoding Dafny functions

- function F( ) returns (T) { E }

- function #F(HeapType, Ref) returns (T);
- Tr[[o.F( )]] =
  #F(Heap, o)
- axiom (∀ h: HeapType, this: Ref •
  #F(h, this) = Tr[[E]]);

Well-definedness of functions

- function F( ) returns (int) { F( ) + 1 }

- function #F(HeapType, Ref) returns (int);
- axiom (∀ h: HeapType, this: Ref •
  #F(h, this) = #F(h, this) + 1);

Bad!

Microsoft Research

Function reads

- function F(p: T) retu
  reads R;
  { E }
- procedure CheckW
  (this: Ref, p: T) returns (result: U)
  { assert Df_R[[E]]; }
- Df_R[[O.M(E)]] =
  Df_R[[O]] $\wedge$ Tr[[O]] $\neq$ null $\wedge$ Df_R[[E]] $\wedge$
  S[Tr[[O]] / this, Tr[[E]] / p] $\subset$ R

can allow $\subseteq$ if M returns bool and occurs in a positive position in the definition of a bool function

where M has reads clause S

Dafny demo

- Chunker1.dfy

Standard specifications

```
class C {
    var footprint: set<object>;
    function Valid() returns (bool)
        reads this.footprint;
    method Init()
        modifies {this};
        ensures this.Valid();
    method Mutate()
        requires this.Valid();
        modifies this.footprint;
        ensures this.Valid();
```

Microsoft Research

Reads clause of Valid

```
class C {
    var footprint: set<object>;
    function Valid() returns (bool)
        reads this.footprint;
    {  this ∈ this.footprint ∧ this.x < this.p.y ∧ …  }
    …
```

Reads clause of Valid

```
class C {
      var footprint: set<object>;
      function Valid() returns (bool)
         reads {this}, footprint;
      {  this ∈ this.footprint ∧
         this.p ∈ this.footprint ∧
         this.x < this.p.y ∧ …
      }
      …
```

A client

```
method Client0() {
    var c := new C;
    call c.Init();
    call c.Mutate();
}
```

Error: unsatisfied modifies clause

Evolving footprint of Init

- method Init()
  modifies {this};
  ensures Valid();
  ensures fresh(footprint – {this});

Microsoft Research

Another client

```
method Client1() {
    var c := new C;  call c.Init();
    call c.Mutate();
    call c.Mutate();
}
```

Error: unsatisfied modifies clause

Microsoft Research

Evolving footprint of Mutate

- method Mutate()

```
    requires Valid();
    modifies footprint;
    ensures Valid();
    ensures fresh(footprint – old(footprint));
```

Microsoft Research

Standard specifications, revisited

```
class C {
    var footprint: set<object>;
    function Valid() returns (bool)
        reads {this}, footprint;
    { this ∈ footprint ∧ … }
    method Init()
        modifies {this};
        ensures Valid();
        ensures fresh(footprint – {this});
    method Mutate()
        requires Valid();
        modifies footprint;
        ensures Valid();
        ensures fresh(footprint – old(footprint));
```

Aggregate objects

```
class RockBand {
   var footprint: set<object>;
   var g: Guitar;
   function Valid() returns (bool)
            reads {this}, footprint;
   {  this ∈ footprint ∧
      g ≠ null ∧ g ∈ footprint ∧
      g.footprint ⊆ footprint ∧
      ¬(this ∈ g.footprint) ∧
      g.Valid() ∧
      …
   }
```

Demo

- RockBand0.dfy

Microsoft Research

Specification and Verification of Object-Oriented Software

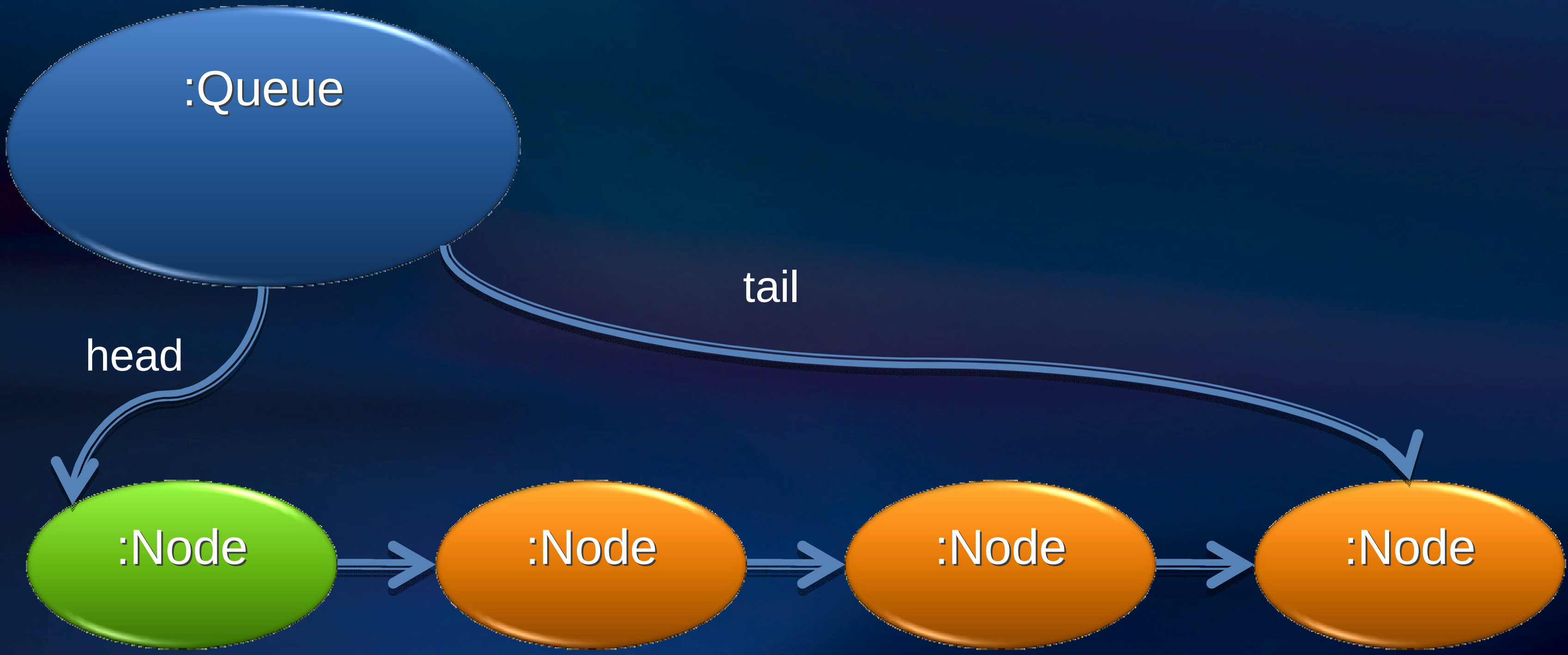
K. Rustan M. Leino

Research in Software Engineering (RiSE)
Microsoft Research, Redmond, WA

part 4
International Summer School Marktoberdorf
Marktoberdorf, Germany
11 August 2008

Example: Queue

- Demo: Queue.dfy



Parallel field update

- foreach (x in S) { x.f := E; }

Homework

Microsoft Research

Capturing a parameter

```
method Init() {
  this.g := new Guitar;
}
method InitFromGuitar(gt: Guitar) {
  this.g := gt;
}
```

Capturing a parameter

```
method InitFromGuitar(gt: Guitar)
  requires gt ≠ null ∧ gt.Valid();
  requires this ∉ gt.footprint;
  modifies {this};
  ensures Valid();
  ensures fresh(footprint – {this} – gt.footprint);
{
  this.g := gt;
  this.footprint := … ;
}
```

Does gt.Valid() hold
after InitFromGuitar?

Microsoft Research

A caller

```
method Client() {
  var kim := new Guitar;  call kim.Init();
  var r := new RockBand;
  call r.InitFromGuitar(kim);
  call kim.Strum();          ← allowed
  call r.Play();             ← error
}
```

Demo

- RockBand0.dfy

Microsoft Research

Borrowing a parameter

```
method Session(org: Organ) {
  … call g.Strum(); call org.Grind(); …
}
```

Microsoft Research

Borrowing a parameter

```
method Session(org: Organ)
  requires Valid() ∧ org ≠ null ∧ org.Valid();
  modifies footprint, org.footprint;
  ensures Valid ∧ org.Valid();
  ensures fresh(footprint – old(footprint));
  ensures fresh(org.footprint – old(org.footprint));
```

Microsoft Research

A client

```
method Client() {
  var r := new RockBand;  call r.Init();
  var b3 := new Organ;  call b3.Init();
  call r.Session(b3);
  call r.Play();
  call b3.Grind();
}
```

Microsoft Research

Demo

- RockBand1.dfy

Microsoft Research

Borrowing a parameter, variation

```
method Session(org: Organ)
  …
  ensures fresh(fo[illegible](footprint));
  ensures fre[illegible](org.footprint [illegible]rg.footprint));
  ensures fresh(footprint + org.footprint
                  – old(footprint) – old(org.footprint));
  ensures footprint !! org.footprint;
  requires footprint !! org.footprint;
```

Microsoft Research

Demo

- RockBand1.dfy, variation

Microsoft Research

Hiding a definition

- function F(p: T) returns (U) reads R;
- axiom (∀ h0: HeapType, h1: HeapType, this: C, p: T •

  $IsHeap(h0) \wedge IsHeap(h1) \wedge$

  $(\forall\ o,f \bullet (o,f) \in R \Rightarrow h0[o,f] = h1[o,f])$

  $\Rightarrow$

  $\#F(h0,this,p) = \#F(h1,this,p));$

Example: BinaryTree

- IntSet.dfy

Example: List

- List.dfy (see pre-lecture notes for Reverse)

Microsoft Research

Specifications in Spec#

- non-null types
- Valid() implicit (declared via invariant)
- [Rep] for components of aggregates
- [Captured] (“borrowed” is default)
- modifies this.* implicit
- modifies p.* implicit for “committed” p

Microsoft Research

Combining access and value

- Implicit dynamic frames [Smans et al.]
- Separation logic [Reynolds, O'Hearn, Parkinson, …]

Microsoft Research

Summary

- Design semantics in terms of an intermediate language!
  - can support different logics: first-order, higher-order, separation, etc.
- Research problem: how to specify programs
- Trade-offs in specification styles:
  - economic (non-verbose) specifications
  - flexibility, expressibility
  - automation
- Links:
  - http://research.microsoft.com/~leino
  - http://research.microsoft.com/specsharp

Microsoft Research