Holistic Specs London July 2019:

0) Terminology

1) Necessary/sufficient for ticket dispenser

2) Necessary/sufficient for Confused Deputy

3) Space and time
   a) $\langle\langle will\ A \rangle\ in\ S \rangle$ vs $\langle will \langle A\ in\ S \rangle\rangle$ and $\langle will\ A\ \underline{using}\ S \rangle$
   b) "robust assertion"
   c) equivalence $\langle\langle will\ A \rangle\ in\ S \rangle$ and $\langle \underline{will}\ A\ \underline{any}\ S \rangle$ when A is robust

4) Language Lemmas
   a) cause of change
   b) what can cause a $\rightsquigarrow$ in the visible state
   c) Only connectivity begets connectivity

5) Proof outlines for some of the properties in 1)

6) Proof outlines for INV-1, INV-2, INV-3 from our OOPSLA19 submission

7) Hoare logic rule using ⟨Will A⟩ in S and its use for the DOM accounts

add

- difference ⟨σ acc σ'⟩
  - vs how it looks in sep. log.
    $\exists f.\ \sigma.f \mapsto \sigma' \vee this = \sigma \wedge \exists x.\ x = \sigma'$
  - vs how in ACL.
    ACL talks about principals not the objects

HSO - 1

Terminology (informal)

$M * M', \sigma \rightsquigarrow \sigma'$ is an "internal step" if $M \oplus M', \sigma \models$ <internal this>

$M \times M', \sigma \rightsquigarrow \sigma'$ is an "external step" if $M \oplus M', \sigma \models$ <external this>

$M \times M', \sigma \rightsquigarrow \sigma'$ is a "full step", as in fig. 6, ie internal or external step.

$M \oplus M', \sigma \rightsquigarrow \sigma'$ is a "visible step", as in Def 24.

To do define $*$ between states.

HS1 -1

Menagerie of Dispensers

One button operation

A `press` operation returns a new ticket number:

```
type Dispenser = interface {
  press -> Number
}

class dispenser -> Dispenser {
  var count : Number := 0
  method press -> Number {
    count := count + 2
    count
  }
}
```

Result must be even

Hoare logic style:

```
d : Dispenser { r = d.press } even(r)
```

Chainmail style:

say that this spec is not cocommended, and HL is nicer

```
forall d : Dispenser. forall o : Object
  [ o.calls {r = d.press} --> Next(even(r)) ]
```

Result must be monotonically in

this is not supported in Chainmail yet

Chainmail v1:

```
forall d : Dispenser [ ~~even(d.count) &&~~ [ Next(d.count == c') --> (c' >= d.count)
] ]
  // requires d.count as ghost field

~~forall~~ c : Number∧d : Dispenser ∧d.count == c { r = d.press } r == c + 2 & d.cou
nt = r ]
  //Hoare tripple ~~in the middle?~~
```

Chainmail v2:

HS1-2

```
forall n, n' : Number, forall d : Dispencer [ n==(d.press) ~~even(n)~~ && [Next(n
'==d.press) --> n' >= n])]
```

~~In current Chainmail we expect d.press to be a ghostfield.~~

Revocable

```
type RevocableDispenser = interface {
press -> Number
revoke
}

class revocableDispenser {
  var count : Number := 0 is ghost //hmm
  var state : Boolean := true is ghost //hmm
  method press {
    if (state) then {
     count := count + 2
     count else {
     error "revoked"
  }
  method revoke {state := false}
  method switch {state := !state}
}
```

Hoare logic version:

```
(d.state = true) && (d.counter = c)  {d.press}  (d.state = true) && (d.counter = c
 + 2)
```

Also need another triple for when d.state=false

Or, stealing syntax from somewhere I've forgotten: Let us stick with one version for Hoare Logics

```
pre  (d.state = true) && (d.counter = c)
prog {d.press}
post (d.state = true) && (d.counter = c + 2)
```

Chainmail version:

→

```
forall d : Dispenser, o : Object (d.state == true) ! Past(o calls d.revoke)
```

(either way, need to adapt spec of next to deal with errors one way or another)

switchable

HS1-3

```
forall d : Dispenser, s : Boolean.  d.state = s && Next(d.state == ! s) --> exists
 o : Object. [o.calls d.switch]
forall d : Dispenser, s : Boolean.  d.state = s && Will(d.state == ! s) --> Will(e
xists o : Object. [o.calls d.switch]  )
```

Say that "extra bits" means "not in Chainmail yet"

The version below uses two extra random bits of notation: * `e @ t` - expression at time t * `Tony(t)` - assuming t is a call, the matching return is just done.

```
(d.state == s) @ t && (d.state ==_t !s) @ Tony(t) --> exists t'' . t < t'' < Tony(
t). exists o [o.calls d.switch] @ t''
```

Two button operation

- A press opration presses the button
- A take operation retrieves the ticket

```
type Dispenser = interface {
press
take -> Number
}

class dispenser -> Dispenser {
  var count : Number := 0
  var pressed : Boolean := false
  method press { pressed := true  }
  method take {
    if (pressed) then { count := count + 2
                        pressed := false
                        count }
                 else { error "nothing will come of nothing" }
  }
}
```

push button, non accumulating

```
d.pressed {r = take} r : Ticket && d.pressed = false
!d.pressed { r = take} r = error
d.pressed && Next(~d.pressed) --> exists o. <o calls d.take>
```

push button, non accumulating

assume additional `var presses := 0 is ghost`

HSI-4

I propose that we use n or n' for numbers, and d or d' for Pispensers

```
forall d' : Number;  d' == d.presses && d' > 0 {r = take} r : Ticket && d.presses
== d' - 1
forall d' : Number;  d' == d.presses && {press} d.presses == d' + 1
d.presses <= 0 { r = take} r = error
forall d' : Number; d.presses && Next(d.presses != d) --> exists o. (<o calls d.ta
ke> || <o calls d.press>)
```

small matters of specifying

- coloured tickets
- price
- delay / timeout
- pin number

HS1-5

DOM Membrane

the trick here is the example from the paper is (almnost) all that's needed:

```
forall S : Set, nd : Node
  [ <will<changes< nd.property>> in S >> -->
  exists  o : Object[
    o ∈ S && !(o : Node) && !(o: Wrapper) &&
      [ exists nd' : Node < o access nd' >  ||
      exists w:Wrapper. exists k:Number.
        (⟨o access w⟩ ∧ nd.parent(k)=w.node.parent(w.height)) ]]
```

that's fine for a one-way wrapper; turns out it would requires a two-way membrane if e.g. the DOM got a notify messaage.

Here's a DOM with membrane

```
type Node  = interface {
   property -> String
   property:=(_ : String)
   parent -> Node
   click
   callback( l : Listener )
}

type Listener = interface {
   clicked(n : Node)
}

def root = object {
  method property { "Root" }
  method property:=(_) { }
  method parent {self}
  method callback( _ ) { }
}

class node(parent' : Node, property' : String) {
  method parent { parent' }
  var property is public := property'
  method callback( l : Listener ) {
    l.clicked( self )
  }
}
```

```
method usingWrappers(unknwn){
  def n1 = node(root,"fixed")
  def n2 = node(n1,"robust")
  def n3 = node(n2,"const")
  def n4 = node(n3,"fluid")
  def n5 ≝ node(n4,"variable")
  def n6 = node(n5,"ethereal")

  def w = n5 //BUG

  def w  = wrapper(n5,1)
  //w.parent.parent.parent.property:= "hacked"

  w.callback( object {
      method clicked(w) { w.parent.parent.parent.property:= "hacked" }
  } )

  assert {n2.property == "robust"}
}

usingWrappers( object { method untrusted( w ) { } } )



class wrapper(node, depth) -> Node {
  //method parent { node.parent }  //BUG
  method parent {
     if (depth > 0) then {wrapper(node.parent, depth - 1)}
        else { error "Hack attempt detected" } }
  method property { node.property }
  method property:=(p) { node.property:= p }
  //method callback( l : Listener ) { node.callback( l ) } // BUG
  //method callback( l : Listener ) { l.clicked( self ) } //SEMI-CHEATING
  method callback( l : Listener ) { node.callback( repparw(l, depth) ) }
}

class repparw( listener, depth ) -> Listener {
   method clicked (node) { listener.clicked( wrapper(node, depth) ) }
}



method assert(block) {
  if (!block.apply) then {Exception.raise "Assertion Failed!"}
}
```

Honest Deputy

James thinks the answer to the compiler as confused deputy problem is relatively straghtforward. Given a spec for the compiler, something like:

```
forall s : Object, calls compiler.compile(inName,outName)  ->
  Next[ FileContents( outName ) == Compile( FileContents( intName )) ]
```

and for a billable service

```
forall s : Object, s calls billable(_) ->
   Next( FileContents ( BILLING ) == Prev(FileContents(BILLING)) ++ ThisBill )
```

or the Hoare logic verions, being a bit more picky:

```
     ExistingFile(inName) && ValidFileName(outName)
 { compiler.compile(inName,outName) }
     FileContents( outName ) == Compile( FileContents( intName ))

   forall c = FileContents(BILLING)
 { compiler.compile(inName,outName) }
   FileContents (BILLING) == c ++ ThisBill
```

the point is that composing those specs together must lead to an unstatisfiable specifiaction, because if you call the compiler with outName=BILLING, then the spec requires both billing data and compiled file contents to be in the BILLING file. To make it satisfiable, you need to add in a precondition e.g. that inFile != Billing...

We could also bound the authority of the deputy - this will manage the risk, but doesn't stop the classical confusion.

```
forall d : Deputy; forall f : File;
   d accesses f ->
     (f = BILLING) ||
       (Was( exists o : Object. o calls d(_) && o accesses f))
```

HS-3.1

Space and time

a) Difference $\langle\langle \text{will } A \rangle \text{ in } s \rangle$ and $\langle \text{will } \langle A \text{ in } s \rangle \rangle$

let us take

$\langle \text{will } A \text{ in } s \rangle$ // we do not have

and also

$$\exists \sigma', \sigma_2 \cdot \sigma = \sigma'|_s * \sigma_2 \wedge$$
$$\sigma|_s \rightsquigarrow \sigma' \wedge$$
$$\sigma' * \sigma_2 \models A$$

vs $\langle\langle \underline{\text{will}}\ A \rangle\ \underline{\text{in}}\ s \rangle$ with means.

$$\exists \sigma', \sigma_2 \quad \sigma \models |_s * \sigma_2 \wedge$$
$$\sigma|_s \rightsquigarrow \sigma' \wedge$$
$$\sigma_2 \models A$$

I think that the two versions are equivalent if A is "robust" and $|_s$ is accordingly to new def. We need the version above (perhaps) for soundness of Hoare logic rule HS6-5

Original Def

Define $\sigma|_S$

$$(\varphi, h)|_S = (\varphi, h|_S)$$

"New Def"

$$(\varphi; h)|_S = (\varphi, h|_{S \cup \{v \mid M_S M'_S \sigma \vdash \langle \text{interp } v \rangle\}})$$

Lemma LS // do we need it? HS-3.?

For all assertions A, B and C:

$$\forall \sigma,\sigma' \left[\sigma \overset{*}{\rightsquigarrow} \sigma' \wedge \sigma \models A \wedge \sigma' \models B \rightarrow \sigma \models C \right]$$

(only if B robust, and $|_S$ as re-defined)

$\longleftrightarrow$

$$\forall S,\sigma,\sigma' \left[\sigma|_S \overset{*}{\rightsquigarrow} \sigma' \wedge \sigma|_S \models A \wedge \sigma' \models B \rightarrow \sigma|_S \models C \right]$$

Proof easy, again two versions of ope seqs ie $M,\sigma \overset{*}{\rightsquigarrow} \sigma'$ and $M;M',\sigma \rightsquigarrow \sigma'$

Lemma LS' // not sure this is needed.

For all assertions A, B and C:

- $\forall \sigma,\sigma' \left[\sigma \overset{*}{\rightsquigarrow} \sigma' \wedge \sigma \models A_1 \wedge A_2 \wedge \sigma' \models B \rightarrow \sigma \models C \right]$

and

- ?? A_1

$\longrightarrow$

- $\forall S, \forall \sigma. \left[\sigma|_S \overset{*}{\rightsquigarrow} \sigma' \wedge \sigma_1|_S \models A_1 \wedge \sigma \models A_2 \wedge \sigma' \models B \rightarrow \sigma|_S \models C \right]$

This lemma needs tweaking & thinking.

HS- 14.11

Language Lemma

Lemma L1

$M, \sigma \rightsquigarrow \sigma'$; // that is in the "full-step"

and $\sigma(o, f) \neq \sigma'(o, f)$

$\rightarrow$

$Class'(o)_{\sigma} = Class(thisN)\sigma$

Proof direct from oper semantics

Lemma L0

$M, \sigma \rightsquigarrow \sigma' \quad \wedge \quad \sigma \vdash \langle external\ o \rangle$

$\rightarrow$

$\sigma' \vdash \langle external\ o \rangle$

Lemma L0'

// not deep

- $M, H, \sigma \rightsquigarrow \sigma' \rightarrow M, H+H', \sigma \overset{*}{\rightsquigarrow} \sigma'$

lemma L2

If a. $M_g M', \sigma \rightsquigarrow \sigma'$ // ie in visible states

and

b. $\sigma(0, f) \neq \sigma'(0, f)$

and

c. $M_g M', \sigma \models$ 'internal(o)

and

d. All calls from internal functions in M (other) to internal functions only, ie no external callbacks.

Then

e. the transition $\sigma \rightsquigarrow \sigma'$ is due to a call of some internal function.

TODOs

T1) How do we ensure d? By open semantics or by invariant. How do we formalise it?

T2 formalise e.

T3 How do we generalise/weaken d?

HS-4.3

Only Connectivity begets Connectivity

lemma L4

$$\forall \sigma,\sigma' \; [\sigma \overset{*}{\rightsquigarrow} \sigma' \wedge \sigma' \not\models \langle o \; \underline{acc} \; o' \rangle \wedge o \neq o' \rightarrow \exists o''. \; [\sigma \models (\langle o'' \; \underline{acc} \; o' \rangle \wedge o' \neq o'')]]$$

lemma L4'

// L4' → L4

$$\forall S. \forall \sigma,\sigma' \; [\sigma|_S \overset{*}{\rightsquigarrow} \sigma' \wedge \sigma \models o : \text{Object} \wedge \sigma' \not\models \langle o \; \underline{acc} \; o' \rangle \wedge o \neq o' \rightarrow \exists o'', \; \sigma|_S \models (\langle o'' \; acc \; o' \rangle \wedge o'' = o)]$$

Proof & Comments

Note that these lemmas should hold both for $M \mathbin{g} M', \sigma \overset{*}{\rightsquigarrow} \sigma'$ // visible steps and for $M, \sigma \rightsquigarrow \sigma'$ // full steps

(advanced:)
Can we find a lemma that translates properties of $M \mathbin{g} M', \sigma \overset{*}{\rightsquigarrow} \sigma'$ to $M \mathbin{g} M', \sigma \overset{*}{\rightsquigarrow} \sigma'$ and opposite?

Lemma LX

For all states σ, σ'
all sets S, T

If a. $S \subseteq T$

b. $\sigma|_S \overset{*}{\rightsquigarrow} \sigma'$

then there exist a σ''

$$\sigma|_T \overset{*}{\rightsquigarrow} \sigma' * \sigma''$$

Note: We must define

$$\sigma * \sigma'$$

Proof outlines for INV-1, INV-2, INV-3 from OOPSLA-19

Lemma LG

for all $M, M', \sigma, \sigma', a, o$

a) $M = M_{BA1}$ or $M = M_{BA2}$

b) $M * M', \sigma \rightsquigarrow \sigma' \wedge \sigma(this) = M$
// ie an internal step

c) $\sigma' \models a : Account \wedge \sigma \models \langle external\ o \rangle \wedge \sigma' \not\models \langle o\ \underline{acc}\ a \rangle$

$\longrightarrow$

$\sigma \models \langle o\ \underline{acc}\ a \rangle$

That is, M_{BA1} and M_{BA2} do not leak accounts.

//

// generalise so that
$\sigma \models a : Account \vee \sigma \models a : Bank$.

HS6-2

Lemma L6'

for all $M, M', \sigma, \sigma', a, o, S$.

a) $M = M_{BA1}$ or $M = M_{BA2}$

b) $M \underset{g}{\circ} M', \sigma|_S \rightsquigarrow \sigma'$

c) $\sigma \models a : \text{Account} \wedge \sigma|_S \models \langle \text{ext } o \rangle$

d) $\sigma' \models \langle o \underline{\text{acc}} a \rangle$

$\rightarrow$

$\exists o'.\ \sigma|_S \models (\langle o' \underline{\text{acc}}\ a \rangle \wedge \sigma \models \langle \text{ext } o' \rangle)$

Proof using L6 and L5 or L5'

This is a proof of INV-3

Lemma L6''

for all $M, M', \sigma, \sigma', a, o'$:

a) $M = M_{BA1}$ or $M = M_{BA2}$

b) $\sigma \in \text{Arising}(M \underset{g}{\circ} M')$

c) $M \underset{g}{\circ} M'; \sigma \models (\langle \text{external } o \rangle \wedge \langle \text{internal } o' \rangle \wedge \langle o \text{ acc } o' \rangle)$

$\rightarrow$

$M \underset{g}{\circ} M', \sigma \models o' : \text{Account} \vee o' : \text{Bank}$

TODO: Write the lemma as a policy

lemma L7

adherence to INV-1

for all $M, M', a, o, \sigma, \sigma', k$

If

a) $M = M_{BA1}$ or $M = M_{BA2}$

b) $M \circ_g M', \sigma \sim \sigma'$

c) $M \circ_g M', \sigma \models (a : Account \wedge a.balance = k)$

d) $M \circ_g M', \sigma \models a.balance \neq k$

Then $\exists x, j$

$\sigma.cnt = a.deposit(x, j)$ ✓

$\sigma.cnt = x.deposit(a, j)$

Proof

By L2, the step in b) was an internal function. By looking at internal functions in M_{BA1}, we see that the only one that is externally callable (using L6"), we deduce that the only externally callable function will modify the balance is transfer

NOTE assume we had a classical spec of transfer. THINK what about deposit(-,-,-) in M_{BA2}: Account class. Ah! Just change to method 2. Do we add internal annotations

Lemma L8

adherance to INV-2

For any $\sigma_1, \sigma_1', \sigma_1'', \sigma_1''', \sigma_2, a, M, M', k$

If

a) $M = M_{BAL}$ or $N = M_{BAL2}$

b) $\sigma_1 * \sigma_2 \models [a : Acc \wedge a.bal = k]$

c) $\wedge \sigma_1 \overset{*}{\rightsquigarrow} \sigma_1' \rightsquigarrow \sigma_1''$

d) $\sigma_1' \models a.bal = k \quad \sigma_1'' \models a.bal \neq k$

Then

α) $\exists o'.\ \sigma_1 \models \langle ext\ o \rangle \wedge \langle o\ acc\ a \rangle$

Proof:

From c) and by ... we obtain, that ex. o'

1) $\sigma_1, \sigma_1' \models \langle ext\ o' \rangle \wedge \langle o'\ \underline{calls}\ a.dep(\ldots) \rangle \vee \langle o'\ call\ -.dep(a..) \rangle$

From 1) and language lemma 2. we obtain

2) $\sigma_1' \models \langle o'\ acc\ a \rangle$

from 2) and lemm L6' (or a variation.)

3) $\exists o'.\ \sigma_1 \models \langle ext\ o \rangle \wedge \langle o\ \underline{acc}\ a \rangle$

which is exactly α.

?? There was no need to use robust spec, the new mean of $\models$ nor the lemmas?

Proving preservation of DOM-properties

$$\forall s.\ [\langle\langle \underline{\text{will}}\ A\rangle\ \underline{\text{in}}\ s\rangle \rightarrow A \vee \langle B\ \underline{\text{in}}\ s\rangle]$$

$$\frac{A' \rightarrow \langle \neg B\ \underline{\text{in}}\ \{z \mid x\ acc^*\ z \vee y\ acc^*\ z\}\rangle}{\neg A \wedge A'\ \{x.m(y)\}\ \neg A}$$

Now, we can argue in DOM.

```
meth(link1, link2) {
    nd1 = --
    nd2 =
    link2.take(w, link1)        use rule here.
```