Holistic Specifications for Robust Code

Sophia Drossopoulou
Imperial College London and MSR Cambridge

in collaboration with James Noble and Julian Maccay (VU Wellington), Toby Murray (Uni Melbourne) , Mark Miller (Agorics), and Susan Eisenbach, Shupeng Loh and Emil Klasan (Imperial)







Holistic Specifications for Robust Code

with support from

- New Zealand Marsden grants
- Ethereum Foundation
- Facebook
- Agoric
- copious “free time”







Holistic Specification of Code

need new encapsulation features

Ownership types?

… WHAT should these features guarantee?

Encapsulation features are *mechanisms for* robustness, but what are the guarantees?

Today

- Traditional Specifications do not adequately address Robustness
- Holistic Specifications — Summary and by Example
- Holistic Specification Semantics
- Trust and Obeys

Today

- **Traditional Specifications do not adequately address Robustness**
- Holistic Specifications — Summary and Examples
- Holistic Specification Semantics
- Trust and Obeys

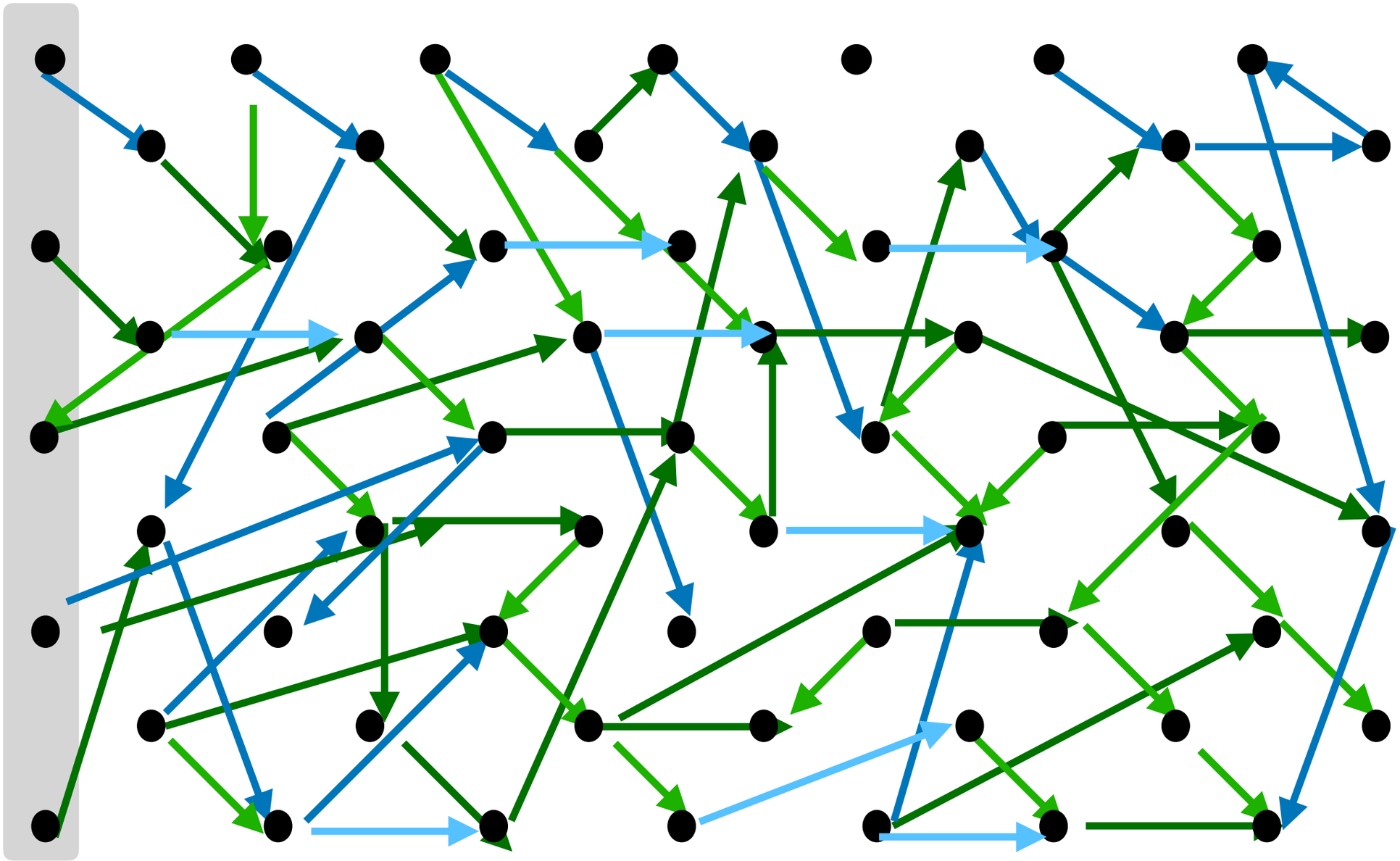
Traditional Specs

- *closed* world
- *sufficient* conditions for some action/effect
- talk of *individual* function

Robustness considerations

- *open* world
- *necessary* conditions for some action/effect
- *emergent* behaviour

traditional



An Example

```
class SafeOne{
    private fld secret;
    private fld treasure;

    mthd take(s){
        if (s==secret)
           { t=treasure;
             treasure = null;
             return t; }   }

    constr SafeOne(s,t){
       secret=s; treasure = t; }

}
```

```
class SafeTwo{
    private fld secret;
    private fld treasure;

    mthd take(s){
        ….    }

    constr SafeTwo(s,t){
       … }

    mthd set(s)
        { this.secret=s;}

}
```

```
class SafeThree{
    private fld secret;
    private fld treasure;

    mthd take(s){
        ….    }

    constr SafeThree(s,t){
       … }

    mthd set(s,sold)
      { if sold==secret then
           { this.secret=s;}}

}
```

traditional

traditional

SpecA:

{ true }
sf.take(s)
{ s=sf.secret -> sf.treasure$_{post}$ = null
∧
s≠sf.secret -> sf.treasure$_{pre}$ = sf.treasure$_{pre}$ }

SpecB: ∀sf:Safe.[sf.secret$_{pre}$ = sf.secret$_{post}$]

SafeOne ⊨ SpecA

SafeTwo ⊨ SpecA

SafeTree ⊨ SpecA

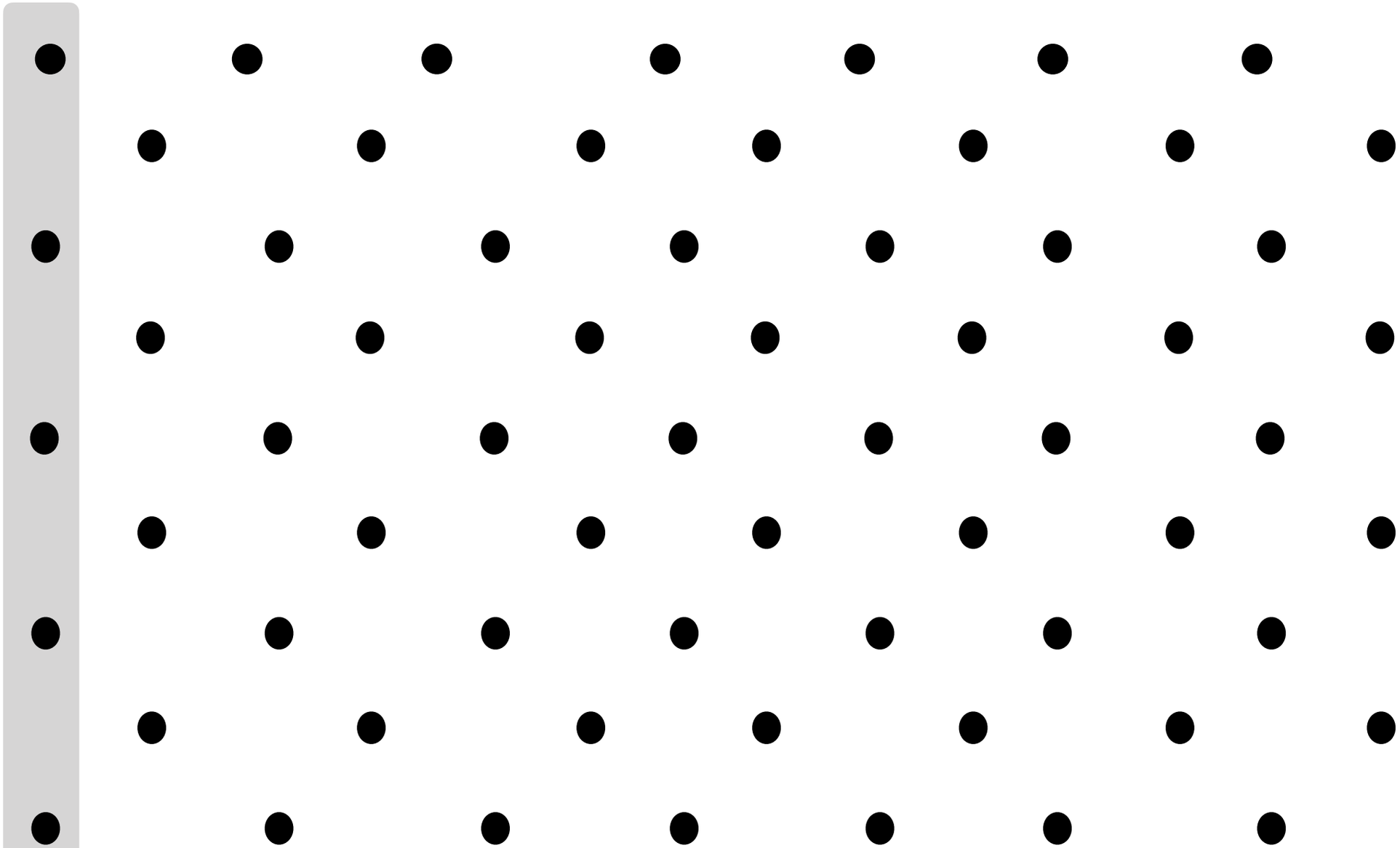
SafeOne ⊨ SpecB

SafeTwo ⊭ SpecB

SafeThree ⊭ SpecB

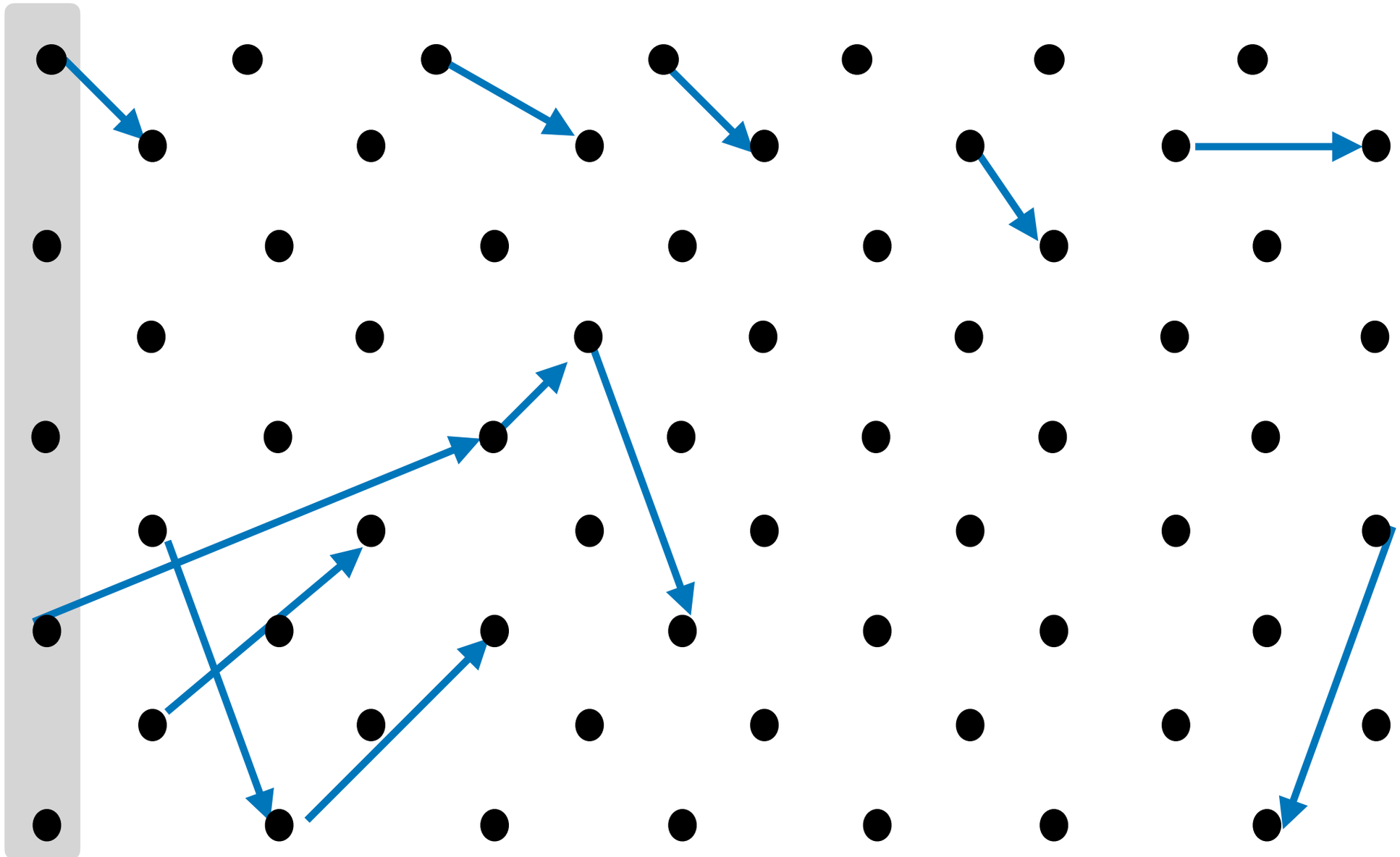
traditional specs do *not* express robustness!

State Space



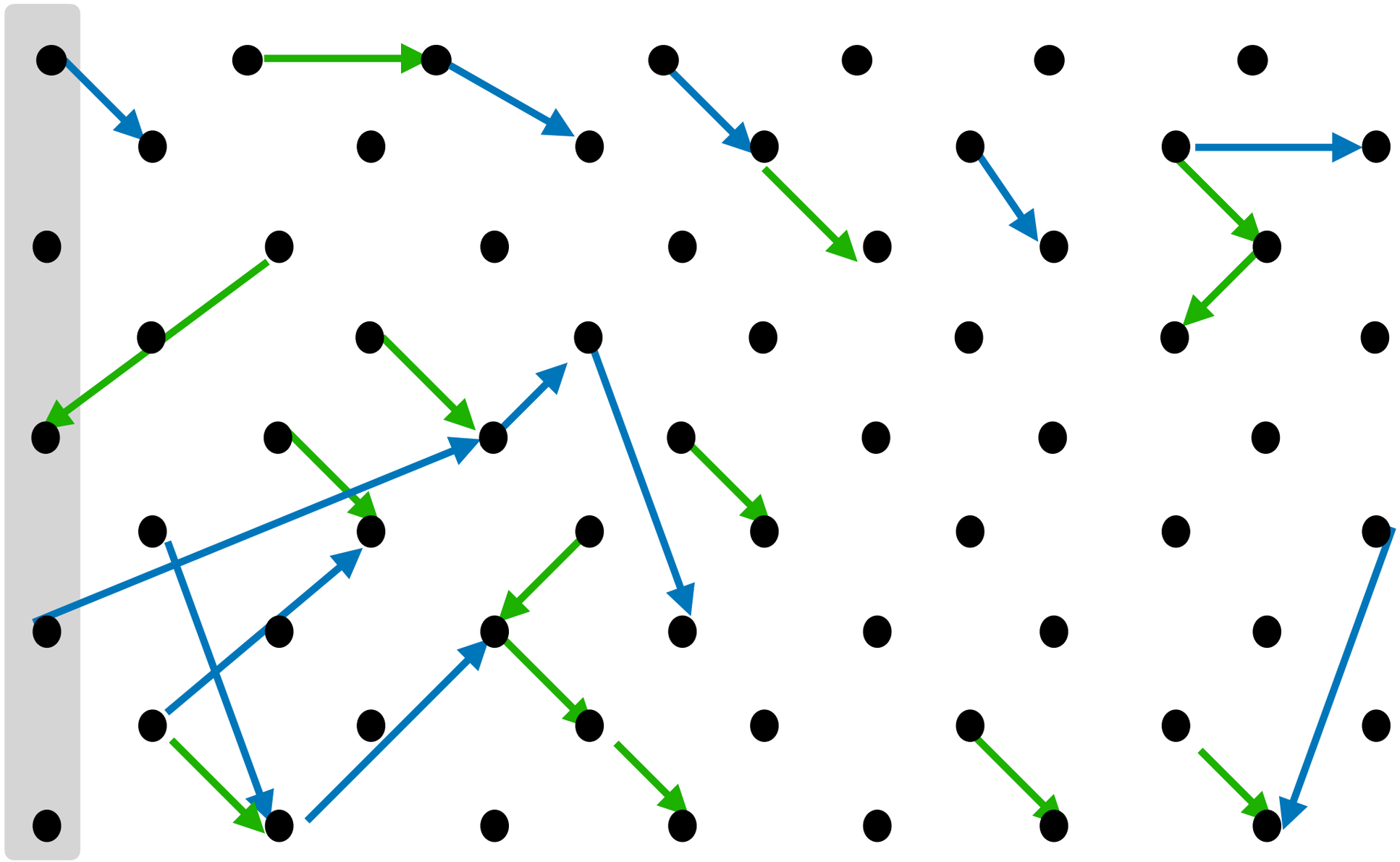
initial configurations

traditional



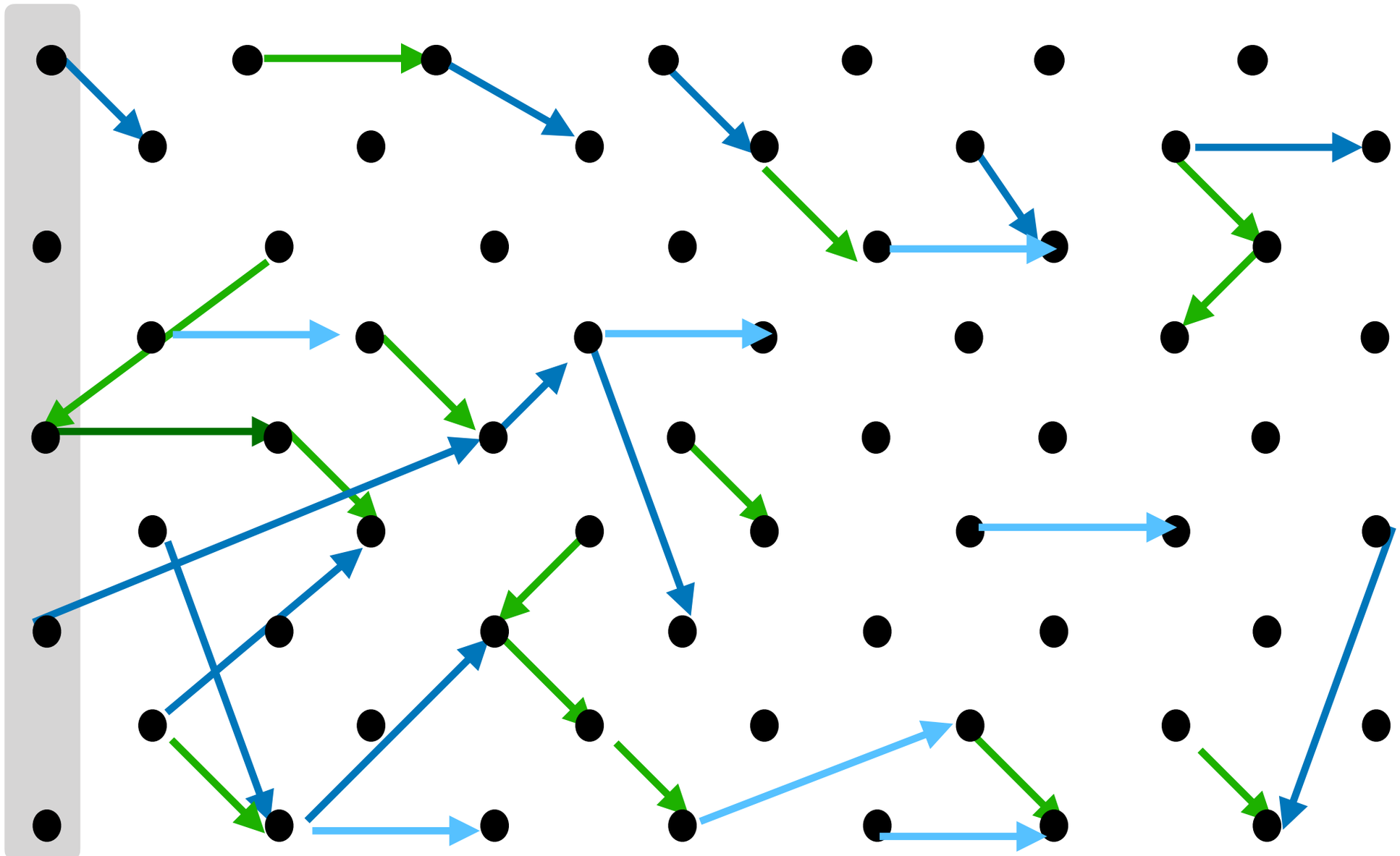
Sufficient Conditions func_1

Traditional



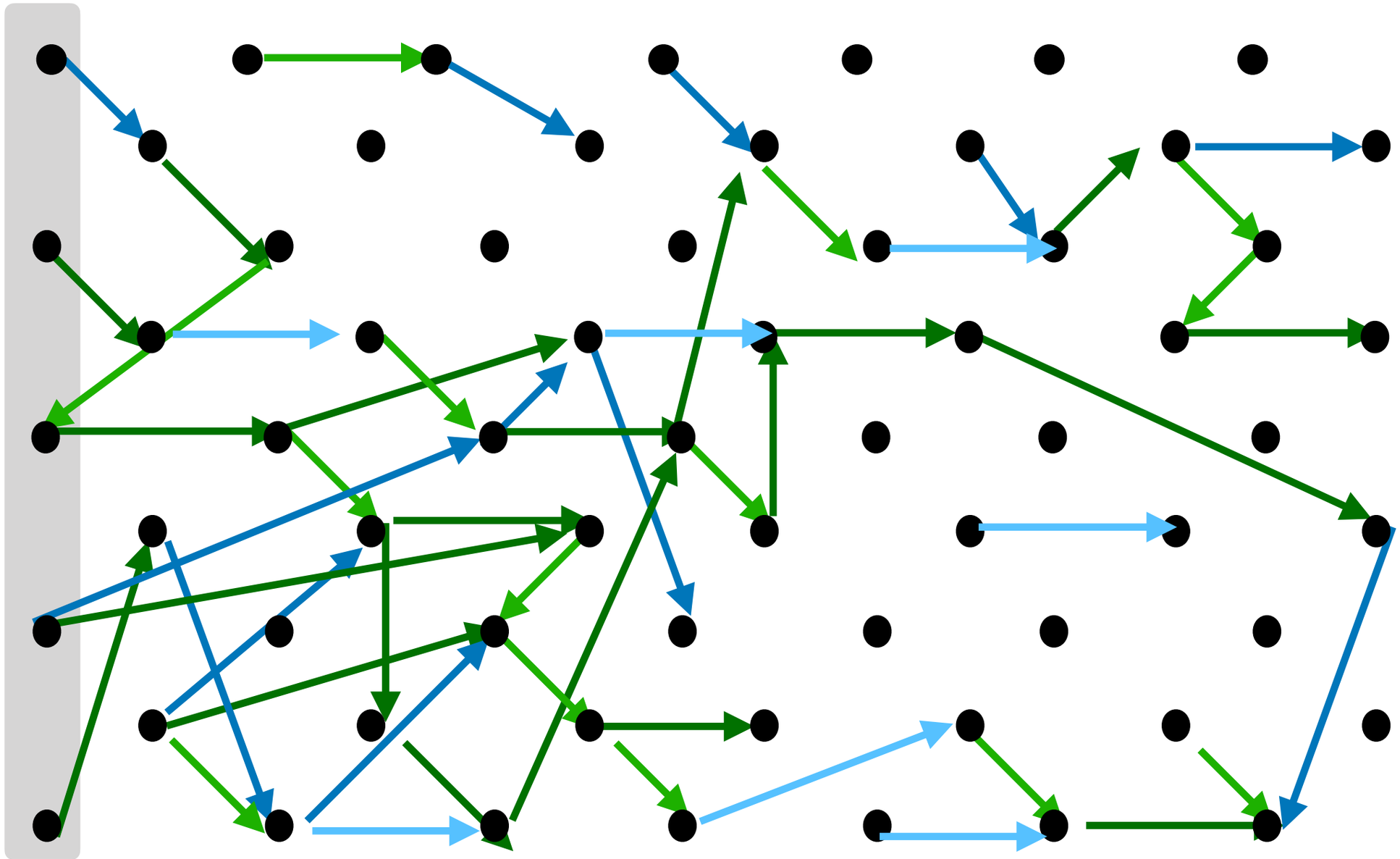
Sufficient Conditions func_1 func_2

Traditional



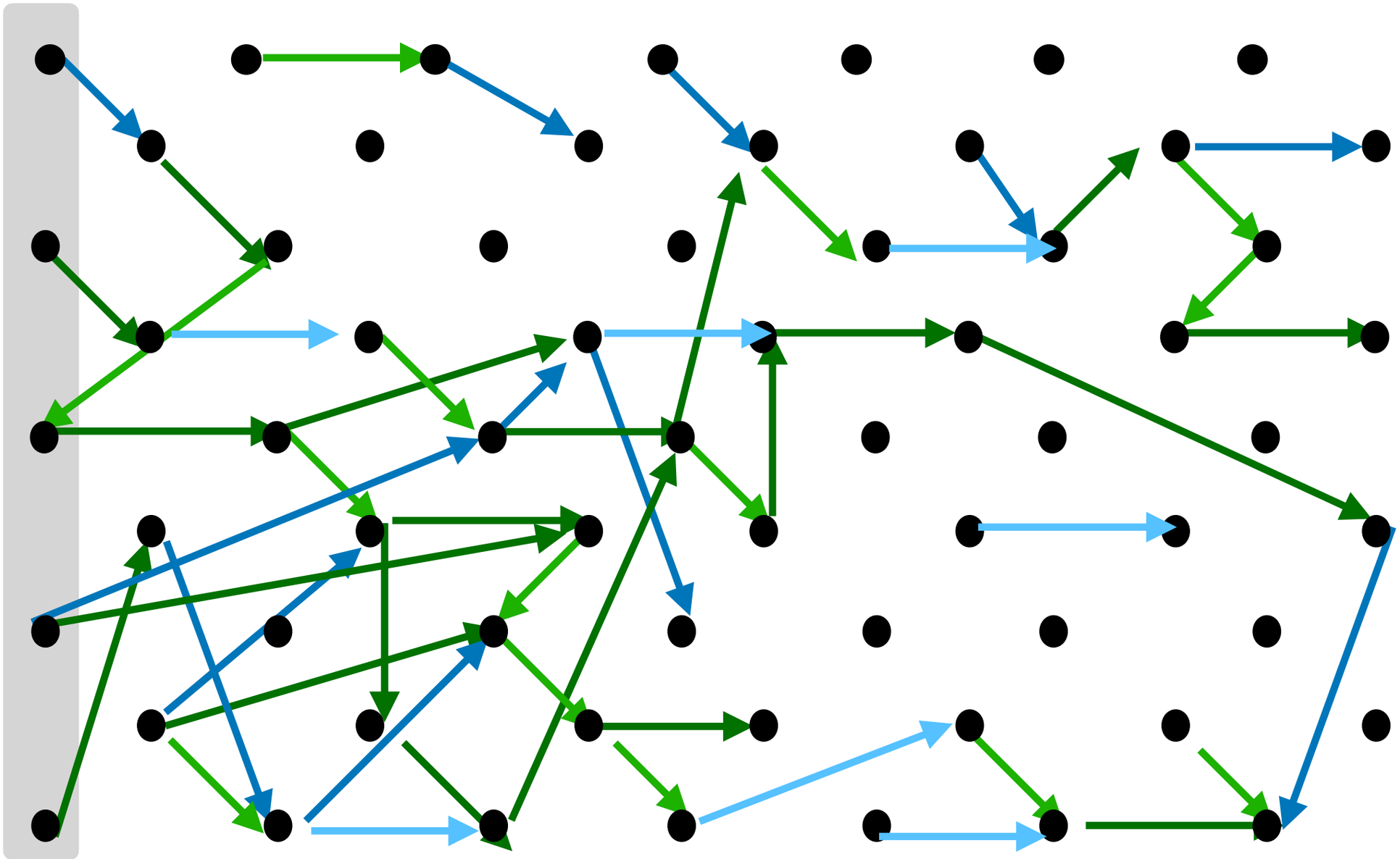
Sufficient Conditions func_1 func_2 func_3

Traditional



Sufficient Conditions func_1 func_2 func_3 func_4

Traditional



Precise, and well-understood, but

- What about the emergent behaviour?
- How many functions' specs fo I need to read to be use of guarantees?
- Which functions may I add to my module?

```
class SafeOne{
    private fld secret;
    private fld treasure;

    mthd take(s){
        if (s==secret)
           { t=treasure;
             treasure = null;
             return t; }   }

    constr SafeOne(s,t){
       secret=s; treasure = t; }

}
```

```
class SafeTwo{
    private fld secret;
    private fld treasure;

    mthd take(s){
        ….    }

    constr SafeTwo(s,t){
       … }

    mthd set(s)
         { this.secret=s;}

}
```

```
class SafeThree{
    private fld secret;
    private fld treasure;

    mthd take(s){
        ….    }

    constr SafeThree(s,t){
       … }

    mthd set(s,sold)
      { if sold==secret then
            { this.secret=s;}}

}
```

SpecA:

$\{ \text{true} \}$ sf.take(s) $\{ s=sf.secret \rightarrow sf.treasure_{post} = null \wedge s \neq sf.secret \rightarrow sf.treasure_{post} = sf.treasure_{pre} \}$

SpecB: $\forall sf:Safe.[\ sf.secret_{pre} = sf.secret_{post}\ ]$

SafeOne ⊨ SpecA
SafeTwo ⊨ SpecA
SafeThree ⊨ SpecA

SafeOne ⊨ SpecB
SafeTwo ⊭ SpecB
SafeThree ⊭ SpecB

SafeOne ⊨ SpecC
SafeTwo ⊭ SpecC
SafeThree ⊨ SpecC

holistic

Want to assert that unless you know the password,
you cannot take treasure, no matter how many functions you use!

```
class SafeOne{
    private fld secret;
    private fld treasure;

    mthd take(s){
        if (s==secret)
           { t=treasure;
             treasure = null;
             return t; }   }

    constr SafeOne(s,t){
       secret=s; treasure = t; }

}
```

```
class SafeTwo{
    private fld secret;
    private fld treasure;

    mthd take(s){
        ….    }

    constr SafeTwo(s,t){
       … }

    mthd set(s)
         { this.secret=s;}

}
```

```
class SafeThree{
    private fld secret;
    private fld treasure;

    mthd take(s){
        ….    }

    constr SafeThree(s,t){
       … }

    mthd set(s,sold)
      { if sold==secret then
            { this.secret=s;}}

}
```

SpecA:

$\{ \text{true} \}$ sf.take(s) $\{ s=sf.secret \rightarrow sf.treasure_{post} = null \wedge s \neq sf.secret \rightarrow sf.treasure_{post} = sf.treasure_{pre} \}$

SafeOne ⊨ SpecA
SafeTwo ⊨ SpecA
SafeThree ⊨ SpecA

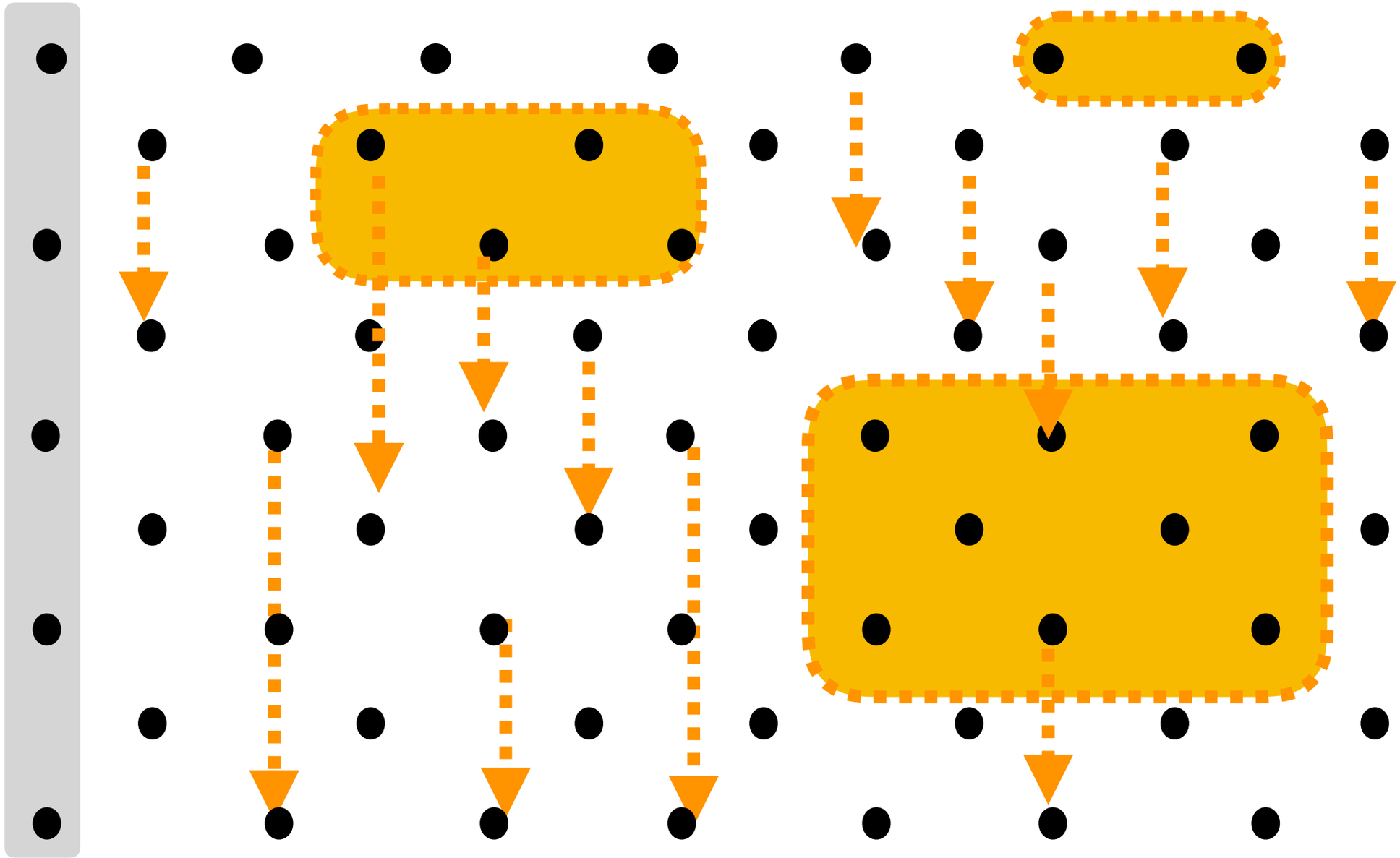
SpecB: $\forall sf{:}Safe.[\ sf.secret_{pre} = sf.secret_{post}\ ]$

SafeOne ⊨ SpecB
SafeTwo ⊭ SpecB
SafeThree ⊭ SpecB

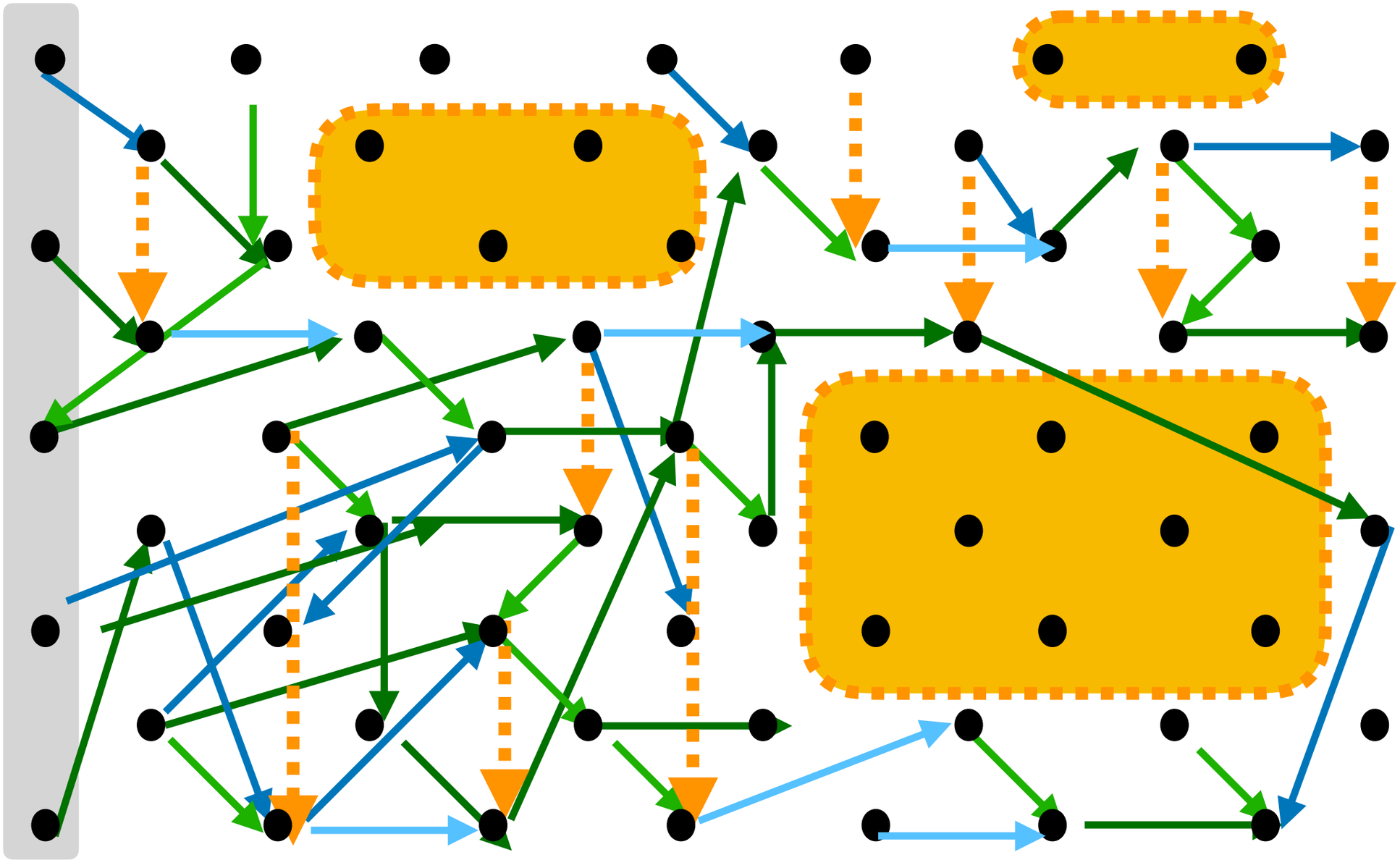
SpecC: $\forall sf{:}Safe.[\ \langle \textbf{Will}\langle \textbf{Changes}\langle sf.treasure\rangle\rangle \longrightarrow \exists\ o.[\ \textbf{External}\langle o\rangle \wedge \langle o\ \textbf{Access}\ sf.secret\rangle\ ]$

SafeOne ⊨ SpecC
SafeTwo ⊭ SpecC
SafeThree ⊨ SpecC

holistic



traditional + holistic



Traditional Specs

- *closed* world
- *sufficient* conditions for some action/effect
- talk of *individual* function

Robustness considerations

- *open* world
- *necessary* conditions for some action/effect
- *emergent* behaviour

Today

- Traditional Specifications do not adequately address Robustness
- **Holistic Specifications — Summary Examples**
- Holistic Specification Semantics

Holistic Assertions — summary

e ::= this | x | e.fld | …

A ::= e>e | e=e | …

| A → A | A ∧ A | ∃x. A | …

| **Access**(e,e') permission

| **Changes**(e) authority

| **Will**(A) | **Was**(A) | **Prev**(A) | **Nect**(A) time

| A **in** S | **External**(e) space

| x.**Calls**(y,m,z1,..zn) control

| x **obeys** A trust

Holistic Assertions — examples

- ERC20
- [DAO]
- DOM attenuation
- [Bank & Account]
- Escrow

Example1: ERC20

a popular standard for initial coin offerings;

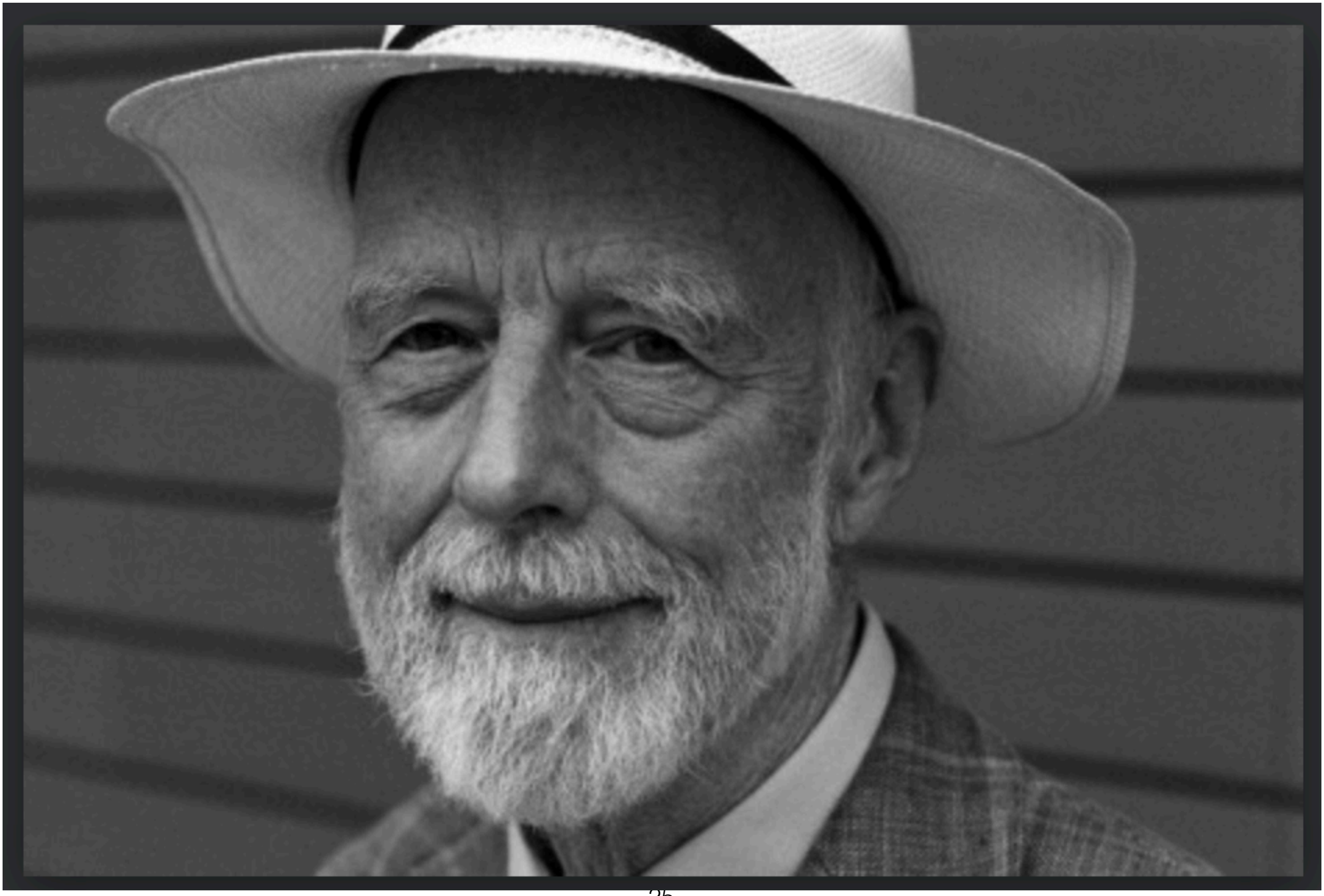
clients can buy and transfer tokens, and can designate other clients to transfer on their behalf.

In particular, a client may call

- `transfer`: transfer some of her tokens to another clients,
- `approve`: authorise another client to transfer some of her tokens on her behalf.
- `transferFrom`: cause another client's tokens to be transferred

Moreover, ERC20 keeps for each client

- `balance` the number of tokens she owns



ERC20 classical spec - transfer

For any ERC20 contract e, and different clients c1, c2.

c1's balance is larger than m.

{ c1 calls `e.transfer(c2,m)` }

c1's balance decreases by m, and c2's balance increases by m.

e:ERC20 ∧ this = c1≠c2 ∧ e.balance(c1) >m

{ `e.transfer(c2,m)` }

$e.balance(c1) = e.balance(c1)_{pre} - m \wedge e.balance(c2) = e.balance(c2)_{pre} + m$

sufficient condition

effect

holistic spec - reduce balance

∀ e:ERC20. ∀ c1: Client. ∀ m: Nat.

```
[  e.balance(c1) = Prev(e.balance(c1))-m
```

⟶

(∃c2,c3: Client.

```
   Prev ( c1.Calls(e,transfer,c2,m) ) )
                  ∨
   Prev ( e.Authorized(c1,c2,m)  ∧
  c2.Calls(e,transferFrom,c1,c3,m)) )   ]
```

effect

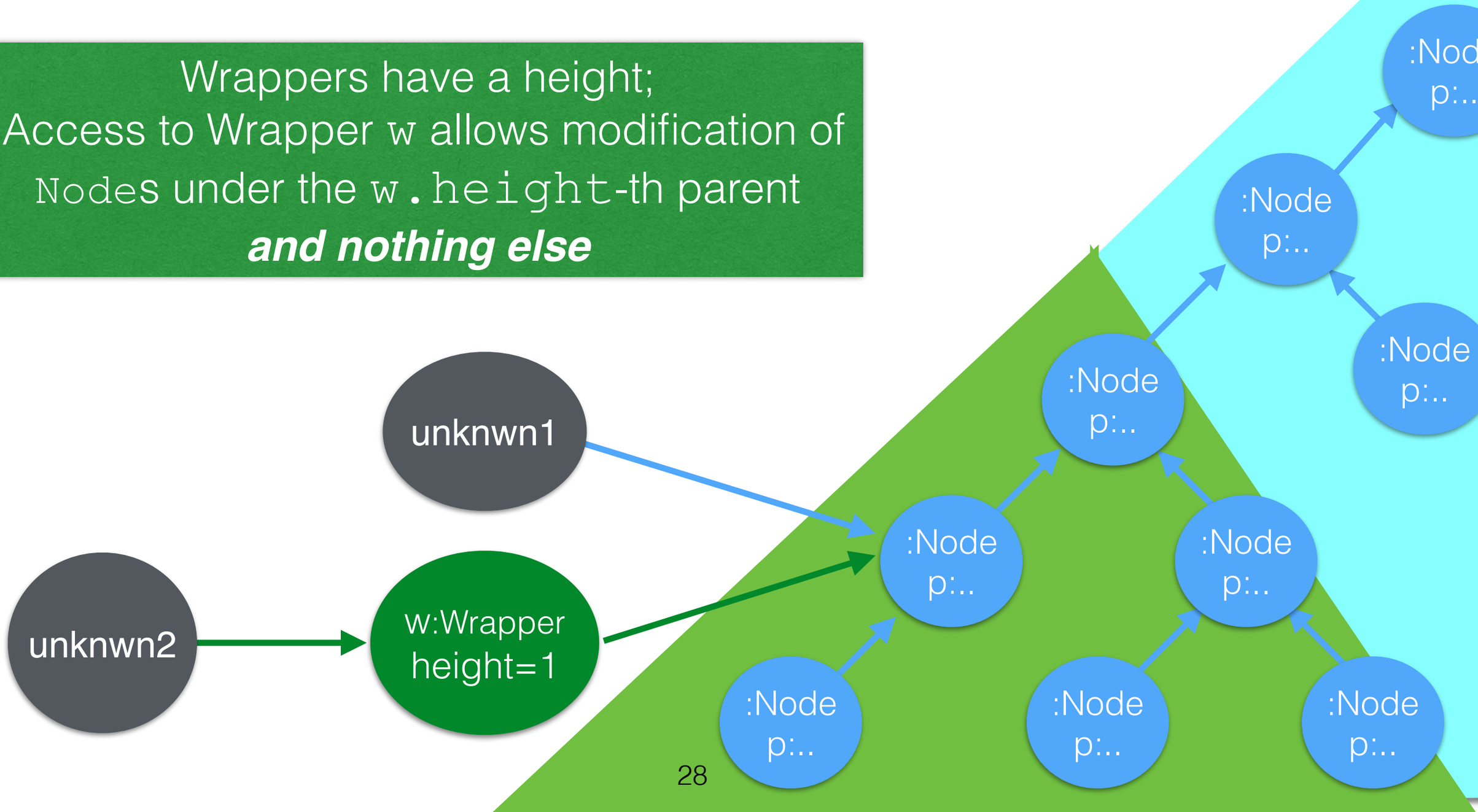
necessary condition

A client's balance decreases *only* if that client, or somebody authorised by that client, made a payment.

Example 3: DOM attenuation

Access to any `Node` gives
access to ***complete*** tree

Wrappers have a height;
Access to Wrapper `w` allows modification of
`Node`s under the `w.height`-th parent
and nothing else



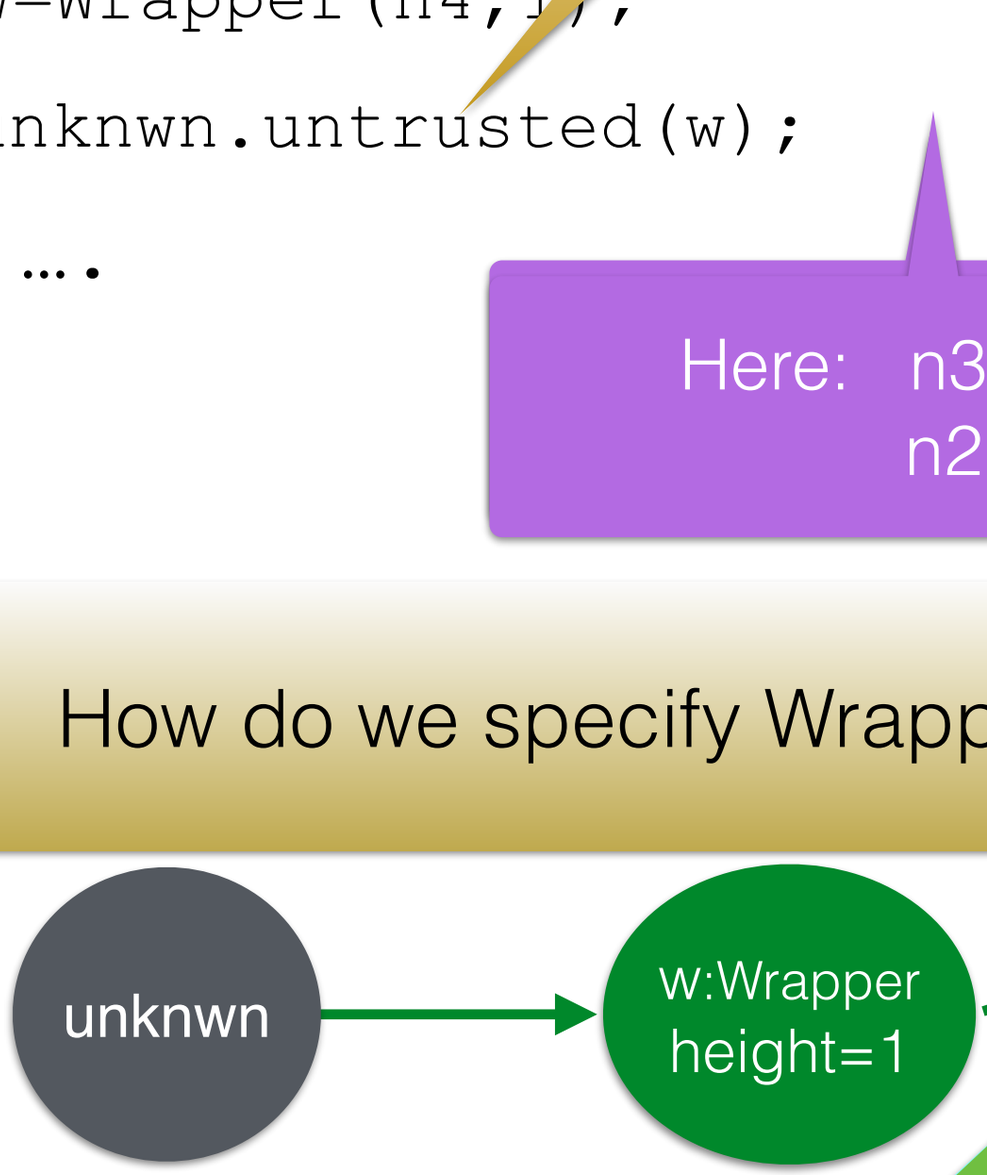
DOM attenuation

```
function mm(unknwn)
  n1:=Node(…); n2:=                    :=Node(n3,…);
  n2.p:="robust"; 3.p:="volatile";
  w=Wrapper(n4,1);
  unknwn.untrusted(w);
  … •
```

open world

Here: n3.p = ????
n2.p = "robust"

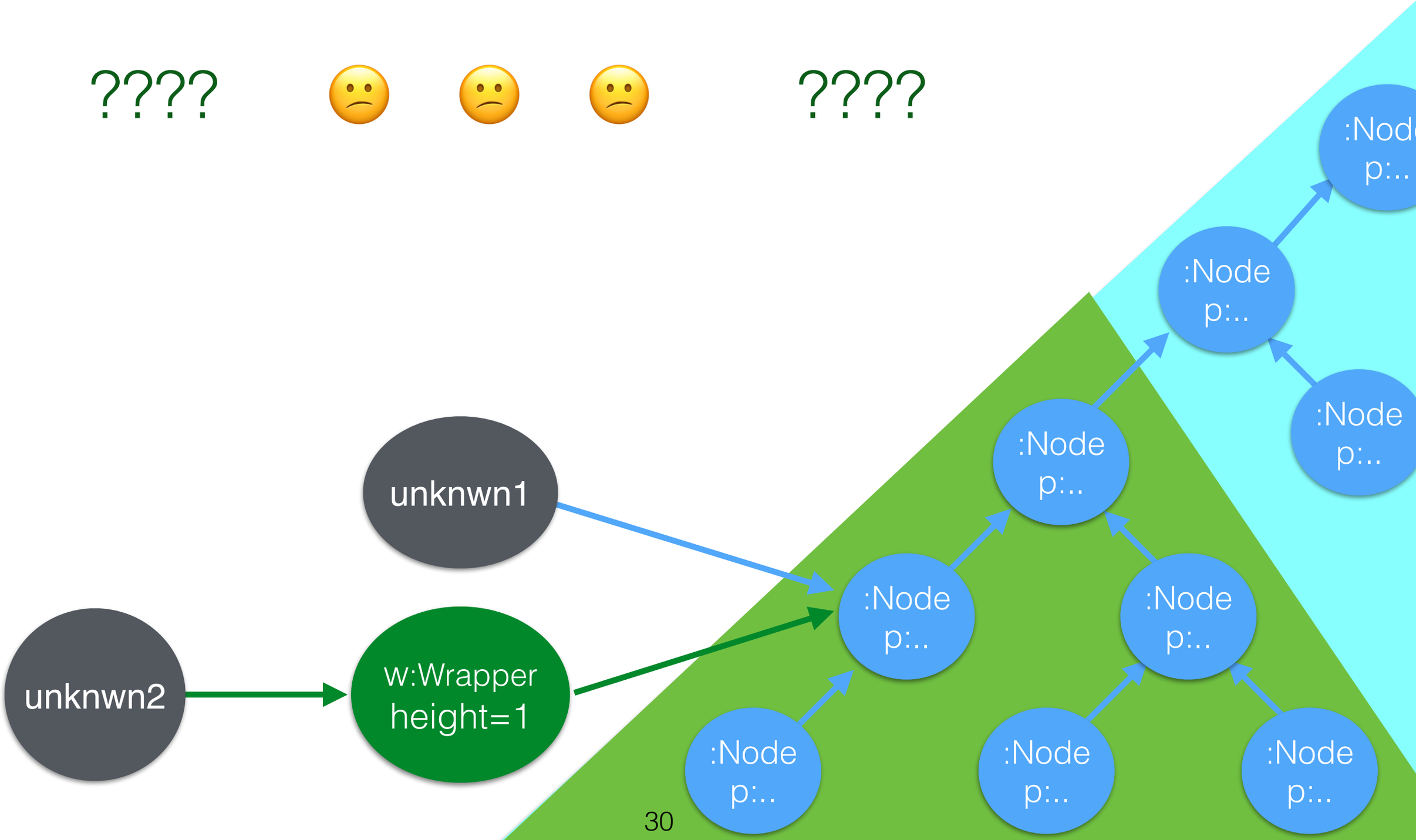
How do we specify Wrapper?



classical

Access to Wrapper `w` allows modification of `Node`s under the `w.height`-th parent and nothing else

???? 😕 😕 😕 ????



holistic

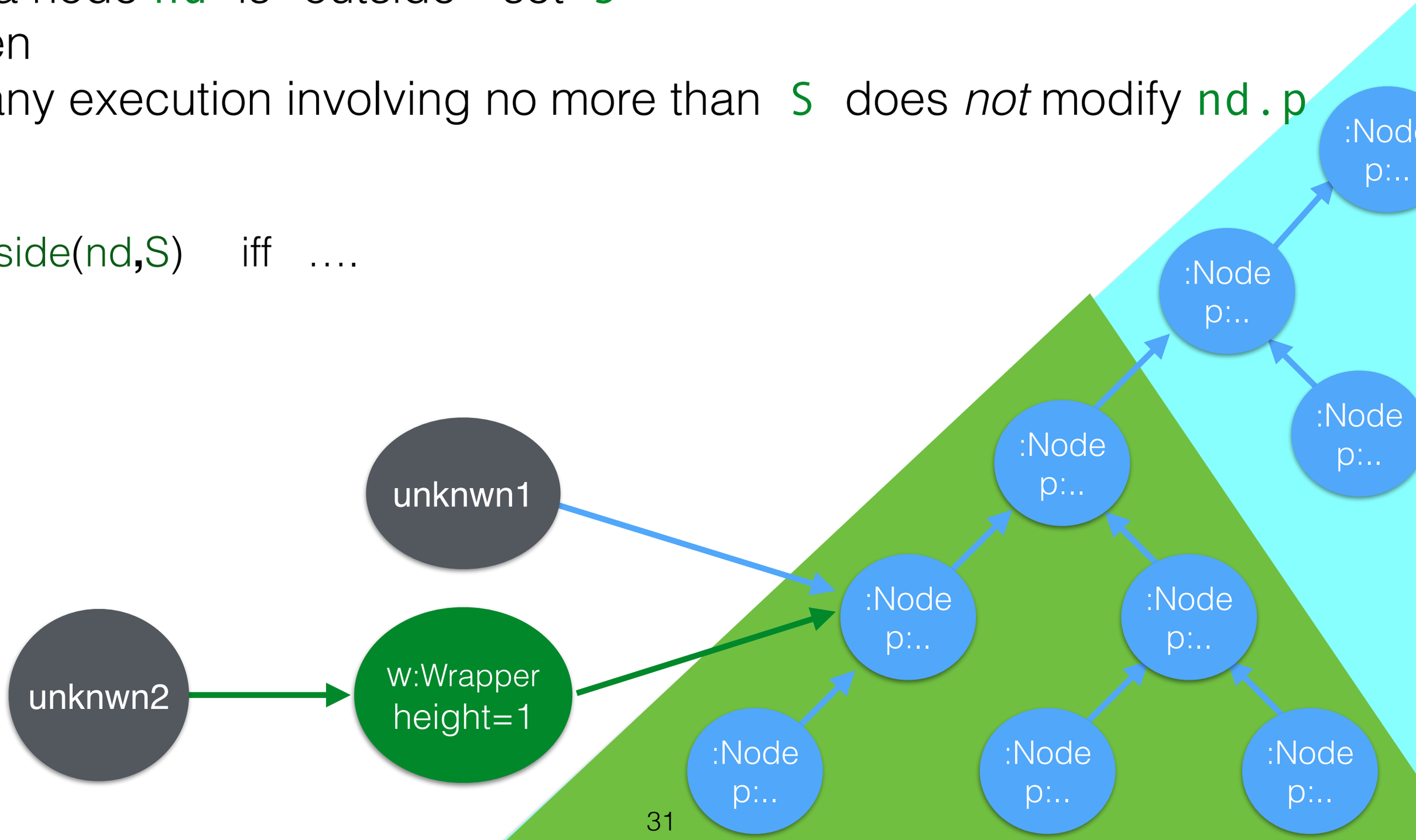
If

a node `nd` is "outside" set `S`

then

any execution involving no more than `S` does *not* modify `nd.p`

Outside(nd,S) iff ….



holistic

∀S:Set. **∀**nd:Node.

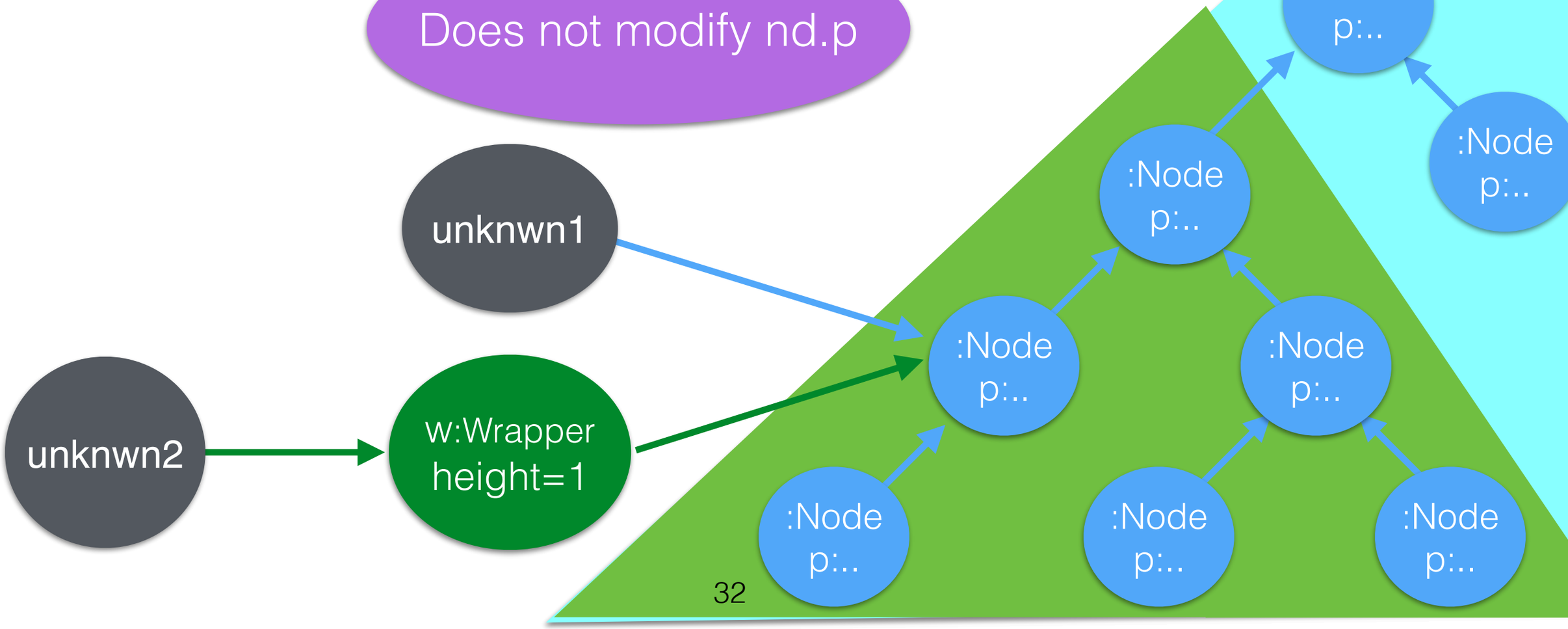
[Outside(nd,S) → ¬ (**Will**(**Changes**(nd.p)) **in** S)]

nd outside S

Outsid

Execution involves no more than S

Does not modify nd.p

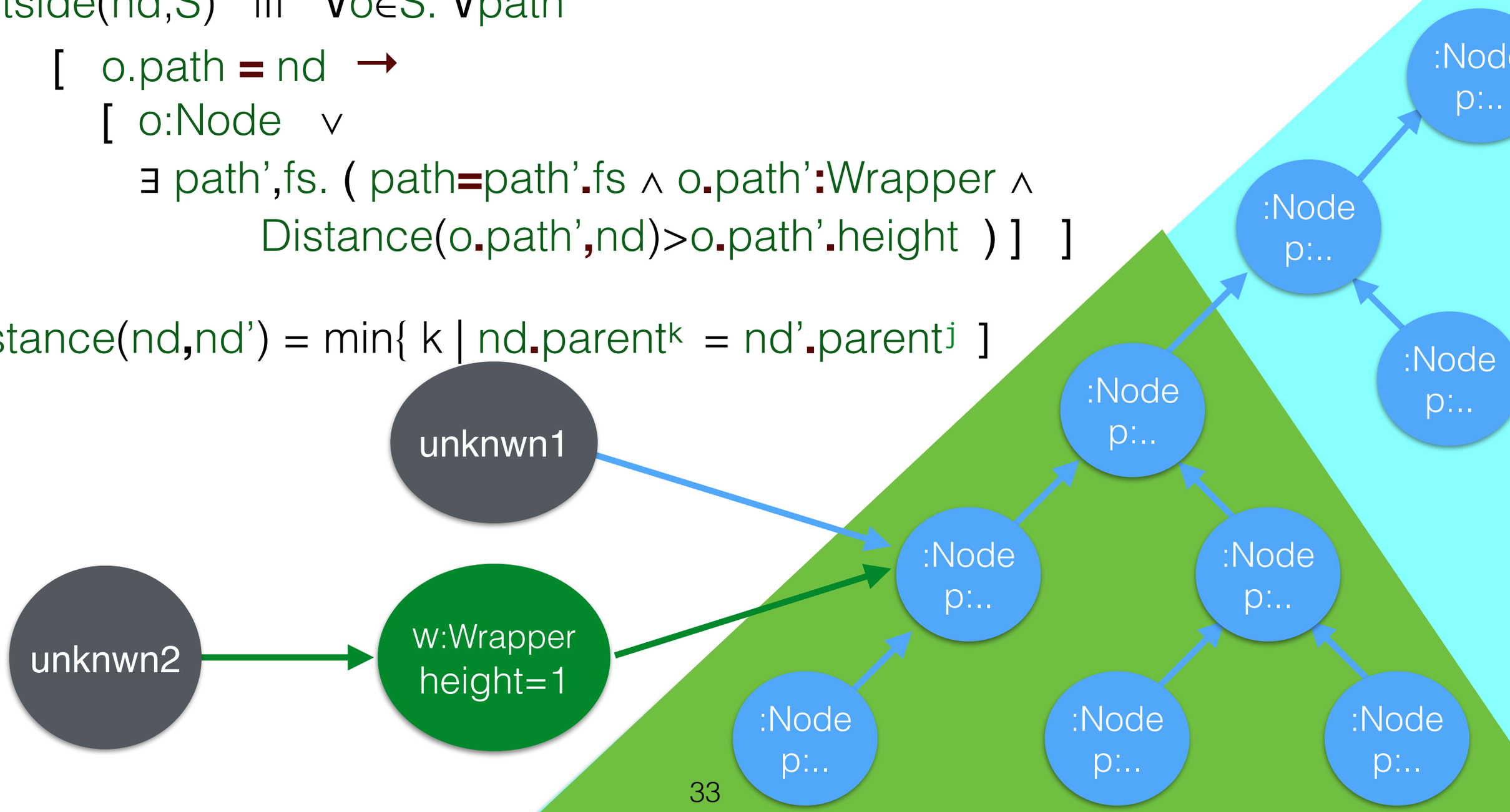


holistic

∀S:Set. **∀**nd:Node.
[Outside(nd,S) → ¬ (**Will**(**Changes**(nd.p)) **in** S)]

Outside(nd,S) iff **∀**o∈S. **∀**path
[o.path **=** nd →
[o:Node ∨
∃ path',fs. (path**=**path'**.**fs ∧ o**.**path'**:**Wrapper ∧
Distance(o**.**path'**,**nd)>o**.**path'**.**height)]]

Distance(nd**,**nd') = min{ k | nd**.**parent^{k} = nd'**.**parent^{j}]



Outside(**RedNode**,**YellowSet**)

Outside(nd,S) iff ∀o∈S. ∀path

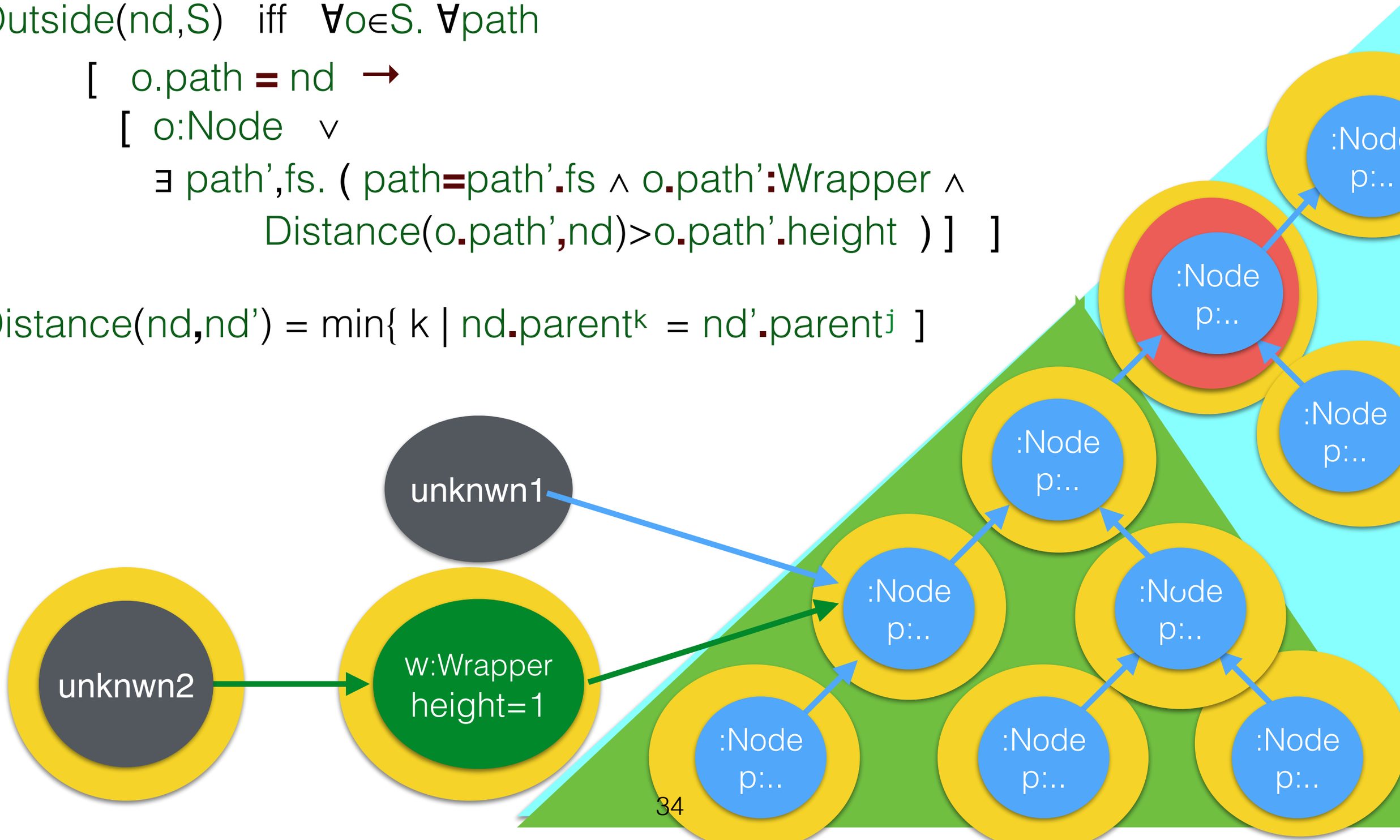
[o.path = nd →

[o:Node ∨

∃ path’,fs. (path=path’.fs ∧ o.path’:Wrapper ∧

Distance(o.path’,nd)>o.path’.height)]]

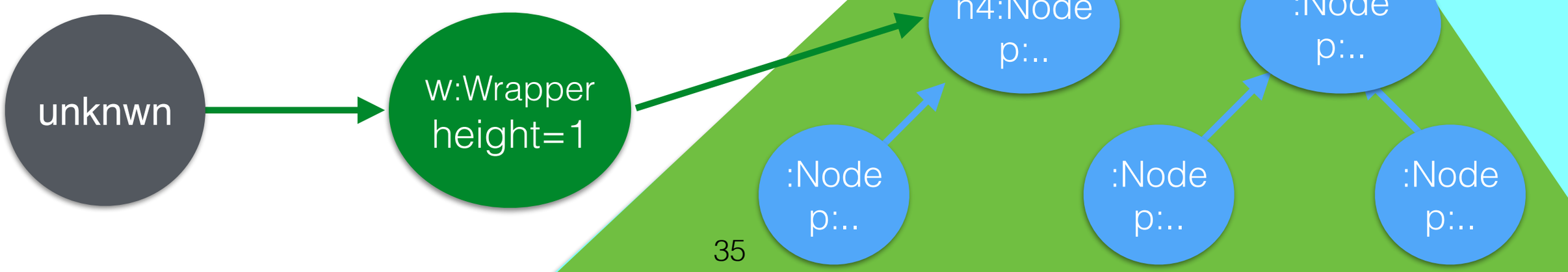
Distance(nd,nd’) = min{ k | nd.parent^{k} = nd’.parent^{j}]



using holistic spec

```
function mm(unknwn){
  n1:=Node(…); n2:=Node(n1,…);n3:=Node(n2,…);n4:=Node(n3,…);
  n2.p:="robust"; n3.p:="vola
  w=Wrapper(n4,1);
  unknwn.untrusted(w);
    ….
```

With holistic spec we can sho
that despite the call to unknown object,
at this point:
n2.p = "robust"



Today

- Traditional Specifications do not adequately address Robustness
- Holistic Specifications — Summary and by Example
- **Holistic Specification Semantics**

Chainmail

Swapsies on the Internet

First Steps towards Reasoning about Risk and Trust in an Open World

Sophia Drossopoulou[1], James Noble[2], Mark S. Miller[3]
[1]Imperial College London, [2]Victoria University Wellington, [3]Google Inc.

How to Break the Bank: Semantics of Capability Policies

Sophia Drossopoulou[1] and James Noble[2]

[1] Imperial College London s.drossopoulou@imperial.ac.uk
[2] Victoria University of Wellington kjx@ecs.vuw.ac.nz

Holistic Specifications for Robust Programs

Sophia Drossopoulou[1[0000−1111−2222−3333]], James Noble[2[0000−1111−2222−3333]], Julian Mackay[2[0000−1111−2222−3333]], and Susan Eisenbach[1[0000−1111−2222−3333]]

[1] Imperial College London {scd,susan}@springer.com
[2] Victoria University of Wellington {julian.mackay,kjx}@ecs.vuw.ac.nz

Abstract Functional specifications describe what program components *can* do: the *sufficient* conditions to invoke a component's operations. They allow us to reason about the use of components in the *closed world* setting, where the component interacts with known client code, and where the client code must establish the appropriate pre-conditions before calling into the component.
Sufficient conditions are not enough to reason about the use of components in the *open world* setting, where the component interacts with external code, possibly of unknown provenance, and where the component itself may evolve over time. In this open world setting, we must also consider the *necessary* conditions,

L_{oo} syntax

ClassDescr ::= class *ClassId* { (*FieldDecl*)* (*MethDecl*)* (*GhosDecl*)* }
FieldDecl ::= field `f`
MethDecl ::= method `m(` `x`* `)` { *Stmts* }
Stmts ::= *Stmt* | *Stmt* ; *Stmts*
Stmt ::= `x.f := x` | `x := x.f` | `x := x.m(` `x`* `)` | `x :=` new `C(` `x`* `)` | return `x`
GhostDecl ::= ghost `f(` `x`* `)` { `e` }
`e` ::= true | false | null | `x` | `e=e` | if `e` then `e` else `e` | `e.f(` `e`* `)`
`x, f, m` ::= `Identifier`

Giving meaning to holistic Assertions

We define in a“conventional” way (omit from slides):

module M : Ident —-> ClassDef ∪ PredicateDef ∪ FunctionDef
configuration σ : Heap × Stack × Code
execution M, σ ⇝ σ’

Define module concatenation * so that

M*M’ undefined, iff dom(M)∩dom(M’) ≠∅

otherwise

(M*M’)(id) = M(id) if M’(id) undefined, else M’(id)

Lemma We will define

- M*M’ = M’*M, if defined
- (M1*M2)*M3 = M1*(M2*M3)
- M, σ ⇝ σ’ M ⊨ A*M’ defined ⟶ M*M’, σ ⇝ σ’

Giving meaning to holistic Assertions

We define in a“conventional” way (omit from slides):

module M : Ident —-> ClassDef ∪ PredicateDef ∪ FunctionDef
configuration σ : Heap × Stack × Code
execution M, σ ⇝ σ’

Define module concatenation * so that
    M*M’ undefined, iff dom(M)∩dom(M’) ≠∅
otherwise
    (M*M’)(id) = M(id) if M’(id) undefined, else M’(id)

We will define M, σ ⊨ A
Initial(σ) and Arising(M)
M ⊨ A

Holistic Assertions — summary

e ::= this | x | e.fld | …

A ::= e>e | e=e | …

| A → A | A ∧ A | ∃x. A | …

| **Access**(e,e') permission

| **Changes**(e) authority

| **Will**(A) | **Was**(A) time

| A **in** S | **External**(e) space

| x.**Call**(y,m,z1,..zn) control

| x **obeys** A trust

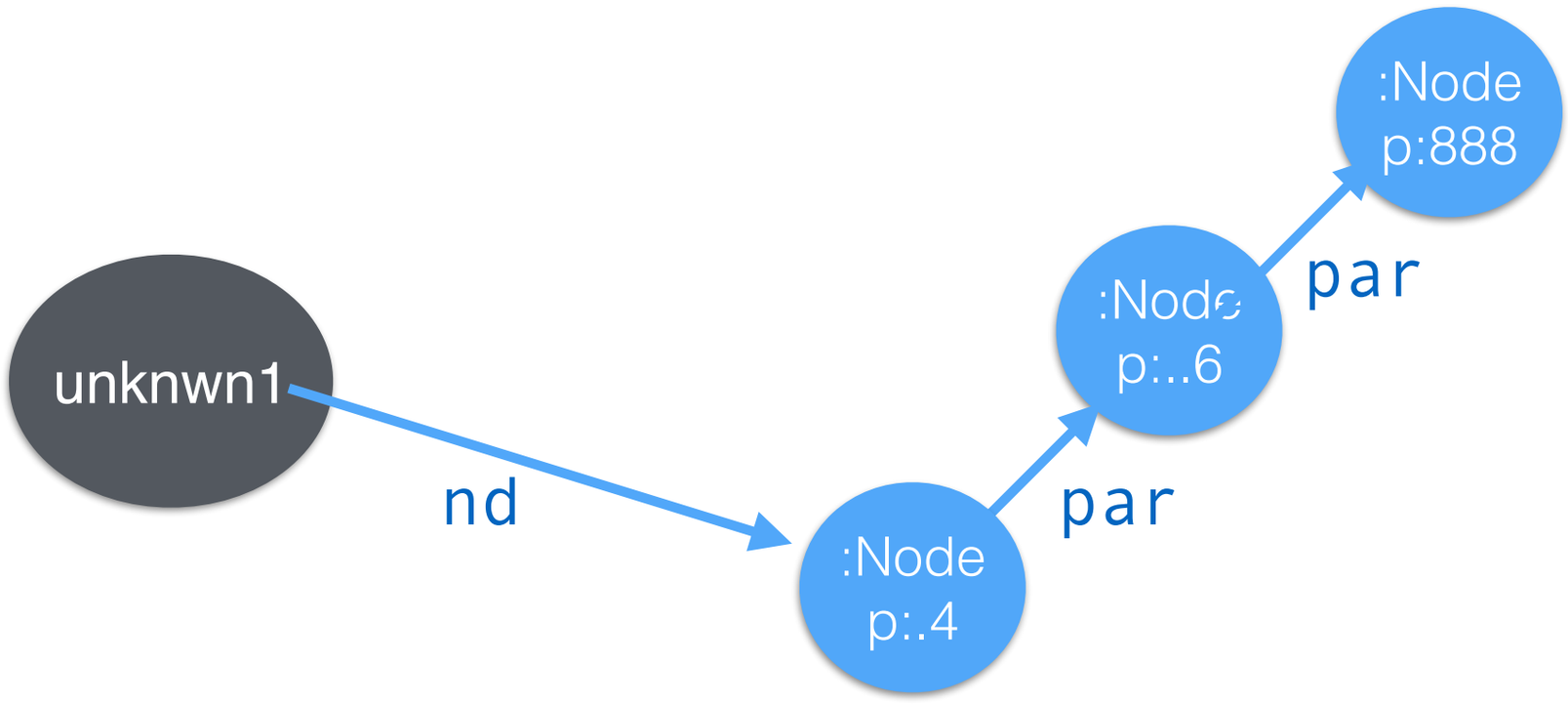
Semantics of Holistic Assertions
- Challenges -

- Time
- Space
- Internal/External - module dependent
- What can we observe?
- Ghost fields - module dependent and recursive

Semantics of Expressions

e ::= this | x | e.fld | func(e1,…en) | …

Define $\lfloor e \rfloor_{M,\sigma}$ as expected … more or less



Eg, $\lfloor unknwn1.nd.par.par.p \rfloor_{M,\sigma} = 888$

Semantics of holistic Assertions

"Conventional part"

$A ::= e \mid e{>}e \mid \ldots \mid A \rightarrow A \mid \exists x.A \mid \ldots$

We define $M, \sigma \models A$

$M,\sigma \models e{>}e'$ iff $\lfloor e \rfloor_{M,\sigma} > \lfloor e' \rfloor_{M,\sigma}$

$M,\sigma \models A \rightarrow A'$ iff $M,\sigma \models A$ implies $M,\sigma \models A'$

$M,\sigma \models \exists x.A$ iff $M,\sigma[z \mapsto \iota\,] \models A[x \mapsto z\,]$
for some $\iota \in dom(\sigma.heap)$, and z free in A

Semantics of holistic Assertions

"Unconventional part"

A ::= **Access**(x,x') | **Changes**(e) | **Will**(A) | A **in** S | x.**Calls**(y,m,z_1,..z_n)

$M,\sigma \models$ **Access**(x,x') iff $\lfloor x \rfloor_{M,\sigma} = \lfloor x' \rfloor_{M,\sigma}$ ∨

$\lfloor x.fld \rfloor_{M,\sigma} = \lfloor x' \rfloor_{M,\sigma}$ for some field fld ∨

$\lfloor this \rfloor_{M,\sigma} = \lfloor x \rfloor_{M,\sigma} \wedge \lfloor y \rfloor_{M,\sigma} = \lfloor x' \rfloor_{M,\sigma}$

∧ y is formal parameter of current function

$M,\sigma \models$ **Changes**(e) iff $M, \sigma \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{M,\sigma} \neq \lfloor e \rfloor_{M,\sigma'}$

$M,\sigma \models$ **Will**(A) iff $\exists \sigma'.[\, M,\sigma \rightsquigarrow^* \sigma' \wedge M, \sigma' \models A \,]$

$M,\sigma \models$ A **in** S iff $M,\sigma@_{Os} \models$ A where $Os = \lfloor S \rfloor_{M,\sigma}$

$M,\sigma \models$ x.**Calls**(y,m,z_1,..z_n) iff $\lfloor this \rfloor_{M,\sigma} = \lfloor x \rfloor_{M,\sigma} \wedge$ σ.code=y'.m(z_1'..z_n') ∧ ..

Semantics of holistic Assertions

- the full truth -

$M,\sigma \models$ **Access**(e,e') iff … as before …

$M,\sigma \models$ **Changes**(e) iff $M, \sigma \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{M,\sigma} \neq \lfloor e[z \mapsto y] \rfloor_{M,\sigma'[y \mapsto \sigma(z)]}$

where $\{z\}=Free(e) \wedge y$ fresh in e, σ,σ'

$M,\sigma \models$ **Will**(A) iff $\exists \sigma',\sigma'',\phi.[\ \sigma = \sigma'.\phi \wedge M,\phi \rightsquigarrow^* \sigma' \wedge M, \sigma'[y \mapsto \sigma(z)] \models A[z \mapsto y]\ ]$

where $\{z\}=Free(A) \wedge y$ fresh in A, σ,σ'

$M,\sigma \models$ A **In** S iff $M,\sigma@_{Os} \models$ A where $Os = \lfloor S \rfloor_{M,\sigma}$

$M,\sigma \models$ x.**Calls**(y,m,z_1,..z_n) iff … as before …

Arising Configurations

A runtime configuration is *initial* iff
1) The heap contains only one object, of class Object
2) The stack consists of just one frame, where this points to that object.

The code can be arbitrary

$$\text{Initial}(\sigma) \quad \text{iff} \quad \sigma.\text{heap}=(1\mapsto(\text{Object},\ldots)\,) \wedge \quad \sigma.\text{stack}=(\text{this}\mapsto 1).[\,]$$

A runtime configuration σ *arises* from a module M if there is some initial configuration σ_0 whose execution M in reaches σ in a finite number of steps.

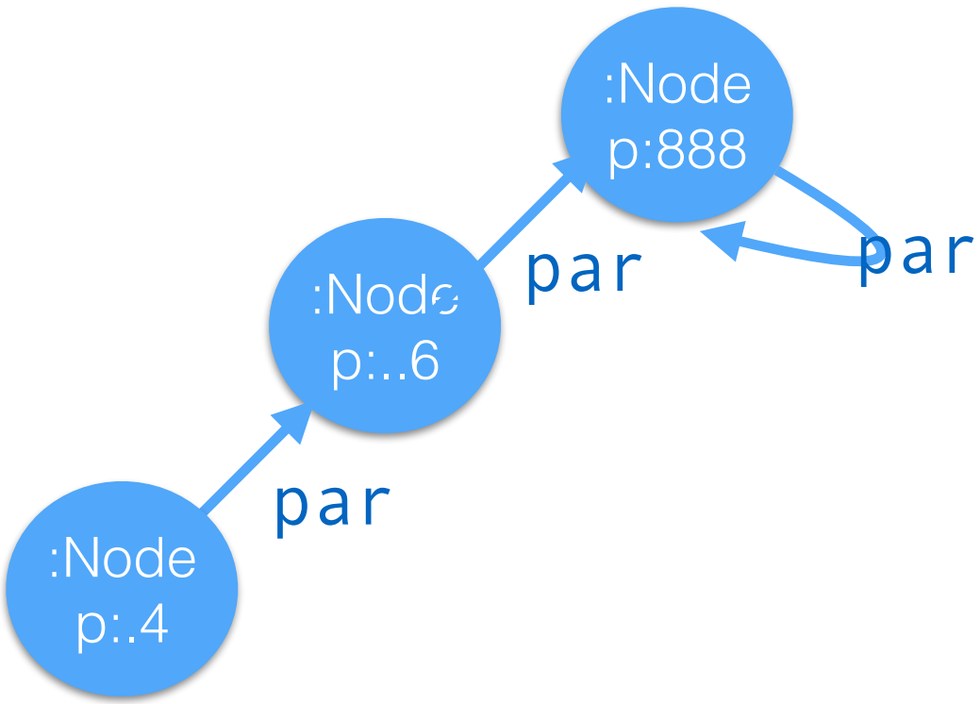
$$\text{Arising}(M) = \{\ \sigma \mid \exists\, \sigma_0.\ \text{Initial}(\sigma_0) \wedge \ M, \sigma_0 \rightsquigarrow^* \sigma \ \}$$

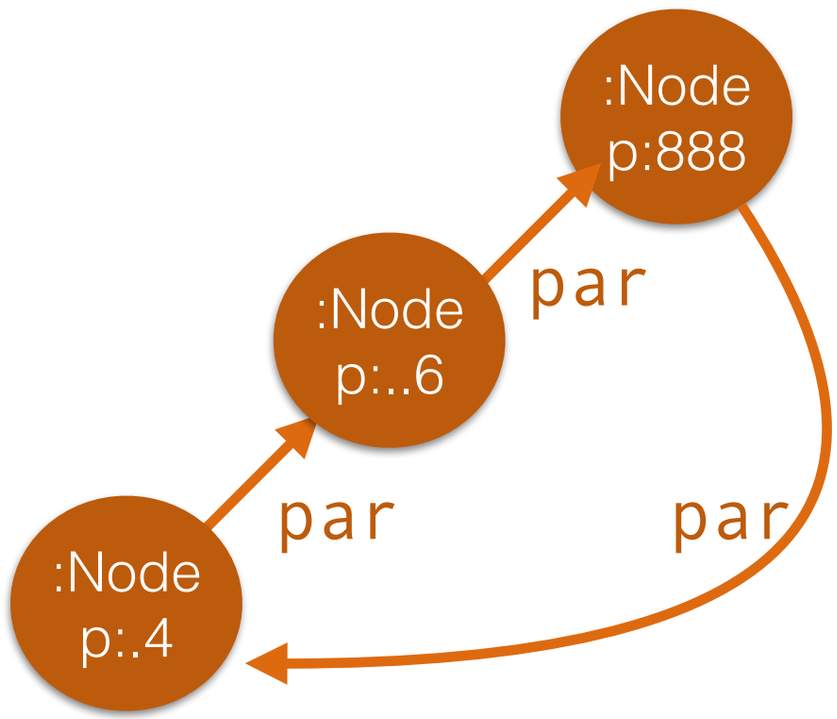
Arising expresses "defensiveness"

Assume a Tree-module, M_{tree}.

`blue configuration` *arises* from M_{tree}*M' for some module M'

`brown configuration` *does not arise* from M_{tree}*M' for *any* module M'





Giving meaning to Assertions

$M \models A$ iff $\forall M'.\forall \sigma \in \text{Arising}(M ; M').\ M ; M', \sigma \models A$

A module M satisfies an assertion A if all runtime configurations σ which arise from execution of code from M ; M' (for any module M'), satisfy A.

open world

- quantify over external modules M'
- M' is external; intermediate states observable
- M is internal; intermediate states are not observable

ghost fields - module dependent

```
module M1:

class Bank {
}

class Account {
   fld bal
   …
   ghost balance()
          this.bal
   }
```

The denotation of expressions

$$\lfloor e \rfloor_{M,\sigma}$$

involves looking up the definition of ghost functions in the module — ghost functions have no side-effects

```
module M2:

class Bank {
    fld ledger

    ghost balance(acc)
        this.ledger.balance(acc)

}

class Account {
   fld bal
   fld myBank
   …
   ghost balance()
        this.balance
}

class Ledger{
    …
    ghost balance(acc)
        ….
}
```

ghost fields - recursive

Denotation of expressions, $\lfloor e \rfloor_{M,\sigma}$ looks definition of ghost functions

A ::= e | … e ::= e | e.ghostfunc(…) | e=e | ….

We define

* $M, \sigma \models e$ if $\lfloor e \rfloor_{M,\sigma} = \text{true}$
* $\lfloor e1=e2 \rfloor_{M,\sigma}$ true if $\lfloor e1 \rfloor_{M,\sigma} = \lfloor e2 \rfloor_{M,\sigma}$
* $\lfloor e1=e2 \rfloor_{M,\sigma}$ undefined if $\lfloor e1=e2 \rfloor_{M,\sigma}$ or $\lfloor e1=e2 \rfloor_{M,\sigma}$

allows infinite recursion in functions, and keeps the logic of assertions classical.
Eg assuming x is cyclic in σ1, we have

…,σ1 ⊭ x.acyclic …,σ1 ⊨ ¬(x.acyclic).

But this comes with some surprises. Eg

…,σ1 ⊭ x.acyclic iff …,σ1 ⊭ x.acyclic=false

Summary of our Proposal

A ::= e>e | e=e | f(e1,..en) | …

| A → A | A ∧ A | ∃x. A | …

| **Access**(x,y) permission

| **Changes**(e) authority

| **Will**(A) | **Was**(A) | **Next**(A) | **Prev**(A) time

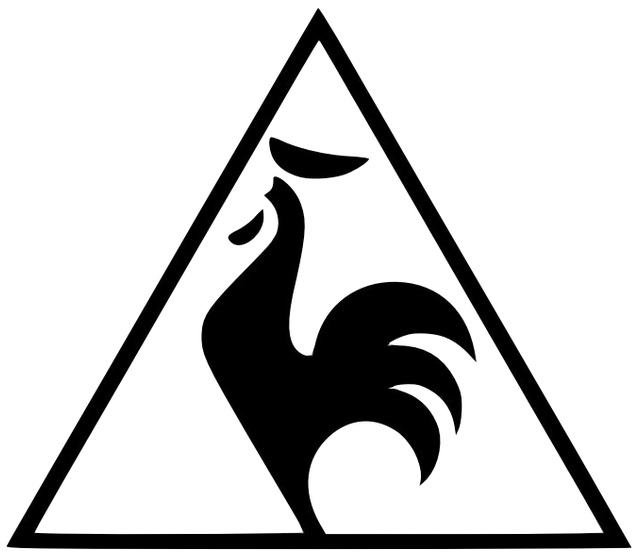
| A **in** S space

| x.**Calls**(y,m,z1,..zn) call

M ; M', σ ⊨ A

Arising(M ; M')

M ⊨ A



Properties of Linking	
(1) $A \wedge \neg A \equiv$ `false` (2) $A \vee \neg A \equiv$ `true` (3) $A \vee A' \equiv A' \wedge A$ (4) $A \wedge A' \equiv A' \wedge A$ (5) $(A \vee A') \vee A'' \equiv A \vee (A' \vee A'')$	(1) `moduleLinking_associative` (2) `moduleLinking_commutative_1` (3) `moduleLinking_commutative_2` (4) `linking_preserves_reduction`
(1) $A \wedge \neg A \equiv$ `false` (2) $A \vee \neg A \equiv$ `true` (3) $A \vee A' \equiv A' \wedge A$ (4) $A \wedge A' \equiv A' \wedge A$ (5) $(A \vee A') \vee A'' \equiv A \vee (A' \vee A'')$ (6) $(A \vee A') \wedge A'' \equiv (A \vee A'') \wedge (A' \vee A'')$ (7) $(A \wedge A') \vee A'' \equiv (A \wedge A'') \vee (A' \wedge A'')$ (8) $\neg(A \wedge A') \equiv (\neg A \vee \neg A')$ (9) $\neg(A \vee A') \equiv (\neg A \wedge \neg A')$ (10) $\neg(\exists x.A) \equiv \forall x.(\neg A)$ (11) $\neg(\exists S.A) \equiv \forall S.(\neg A)$ (12) $\neg(\forall x.A) \equiv \exists x.(\neg A)$ (13) $\neg(\forall S.A) \equiv \exists S.(\neg A)$	(1) `sat_and_nsat_equiv_false` (2) - (3) `and_commutative` (4) `or_commutative` (5) `or_associative` (6) `and_distributive` (7) `or_distributive` (8) `neg_distributive_and` (9) `neg_distributive_or` (10) `not_ex_x_all_not` (11) `not_ex_`Σ`_all_not` (12) `not_all_x_ex_not` (13) `not_all_`Σ`_ex_not`

- Traditional Specifications do not adequately address Robustness
- Holistic Specifications — Summary and Examples
- Holistic Specification Semantics
- **Trust and Obeys**

- **Trust and Obeys**

Swapsies on the Internet

First Steps towards Reasoning about Risk and Trust in an Open World

Sophia Drossopoulou[1], James Noble[2], Mark S. Miller[3]

[1]Imperial College London, [2]Victoria University Wellington, [3]Google Inc.

Abstract

Contemporary open systems use components developed by many different parties, linked together dynamically in unforeseen constellations. Code needs to live up to strict *security specifications:* it has to ensure the correct functioning of its objects when they collaborate with external objects which

careful to take only stickers you're definitively willing to risk losing when you go to meet the gorilla in year ten.

Internet Swapsies Playground swapsies is just one example of an interaction between *mutually untrusting* parties in an *open system* — other examples include so-called dark Internet markets (like Silk Road and Evolution) and even larger trading systems like eBay when participants choose not to

Bank/Account - 2

classical | robustness

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_3**: The bank can only inflate its own currency
- **Pol_4**: No one can affect the balance of an account they do not have.
- **Pol_5**: Balances are always non-negative.
- **Pol_6**: A reported successful deposit can be trusted as much as one trusts the account one is depositing to.

[Miller et al, Financial Crypto 2000]

Exchange money and goods
buyer and seller do not trust one another

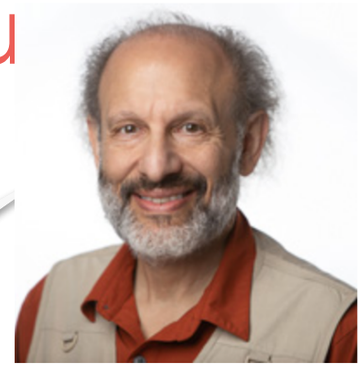
Distributed Electronic Rights in JavaScript

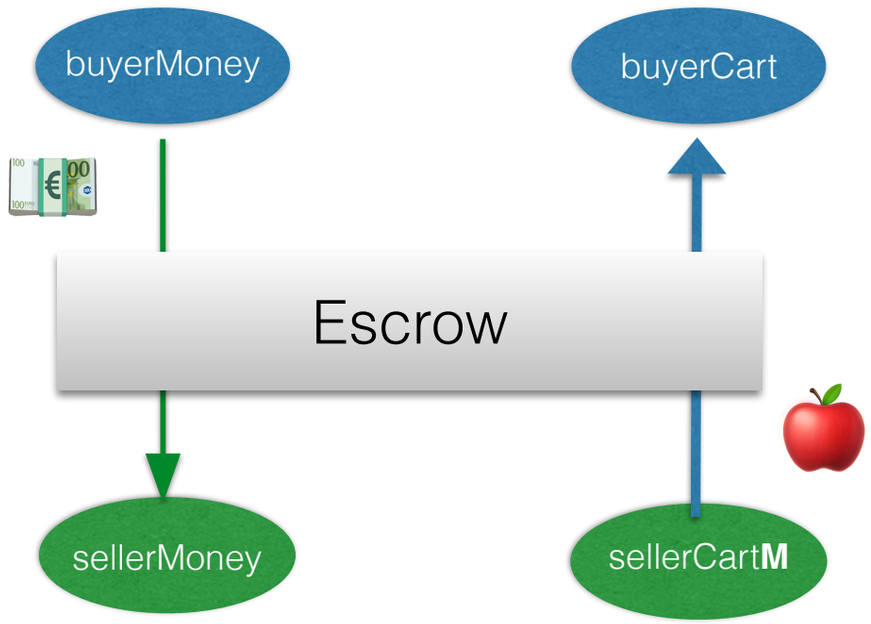
Mark S. Miller[1], Tom Van Cutsem[2], and Bill Tulloh

[1] Google, Inc.
[2] Vrije Universiteit Brussel

Abstract. Contracts enable mutually suspicious parties to cooperate safely through the exchange of rights. Smart contracts are programs whose behavior enforces the terms of the contract. This paper shows how such contracts can be specified elegantly and executed safely, given an appropriate distributed, secure, persistent, and ubiquitous computational fabric. JavaScript provides the ubiquity but must be significantly extended to deal with the other aspects. The first part of this paper is a progress report on our efforts to turn JavaScript into this fabric

Exchanging money for goods without mutual tru





That *is* a surprise!

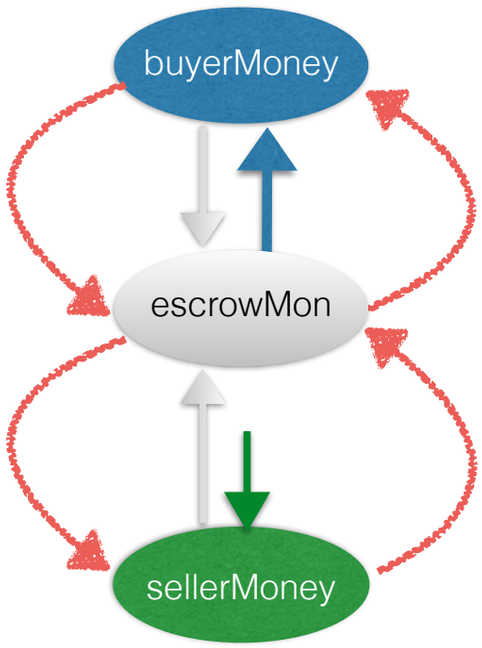
```
cla                                    y,
                                       ney:Purse<Money>,
                                       rt,
                                       erCart: Purse<Goods>)
        : bool
}
```

¬res → "no transfer" ∧
(¬ buyerMoney **obeys** ValidPurse ∧ sellerMoney **obeys** ValidPurse ∨
buyerMoney **obeys** ValidPurse ∧ ¬ sellerMoney **obeys** ValidPurse ..)

res → "transfer happened" ∧ buyerMoney **obeys** ValidPurse
∧ sellerMoney **obeys** ValidPurse
∧ …

∨ ¬(buyerMoney **obeys** ValidPurse) ∧ ¬ sellerMoney **obeys** ValidPurse

Exchanging money for goods without mutual trust



escrowMoney = sellerMoney.sprout()

// sellerMoney **obeys** ValidPurse → escrowMoney **obeys** ValidPurse

res= escrowMoney. deposit (buyerMoney,**0**)

if **!res** then **return** res

// sellerMoney **obeys** ValidPurse → buyerMoney **obeys** ValidPurse

res= buyerMoney. deposit(escrowMoney,**0**)

if **!res** then **return** res

// buyerMoney **obeys** ValidPurse → escrowMoney **obeys** ValidPurse

res= escrowMoney. deposit (buyerMoney,**0**)

if **!res** then **return** res

// buyerMoney **obeys** ValidPurse → seller **obeys** ValidPurse

// sellerMoney **obeys** ValidPurse ⟷ buyerMoney **obeys** ValidPurse)

Summary

- We argue that robustness characterisation requires a mix of classical and holistic specifications.
- We have made a proposal for holistic specifications and applied to examples from contracts and from object capabilities literature.

What was challenging/ fun

- two module execution a
- adaptations for time, W
- dealing with space <.. i
- ghostfields, (treat cycle
- keeping assertions clas
- Language properties,
  eg only connectivity be
- Coq model for language specs
- Proof logic for obeys

Still to do ..

- Proof logic using holistic assertions (in progress)
- Proof logic to establish holistic assertions (in progress)
  challenge: avoid explicit naming configurations
- Semantics of obeys

Thank you

Classical Specification vs Holistic Specification

- fine-grained
- per function

- ADT as a whole
- emergent behaviour

Which is "stronger"?
"Closed" ADT with classical spec implies holistic spec.
(closed: no functions can be added, all functions have classical specs, ghost state has known representation)

Why do we need holistic specs?
* "closed ADT" is sometimes too strong a requirement.
* Holistic aspect is cross-cutting (eg no payment without authorization)
* Allows reasoning in open world (eg DOM wrappers)

ERC20 classical spec - transfer - 2

What if c1's balance not large enough?

e:ERC20 ∧ this = c ∧ e.balance(c1) < m

{ `e.transfer(c2,m)` }

∀ c. e,balance(c) = $e,balance(c)_{pre}$

ERC20 classical spec
- what about the other functions?

ERC20 classical spec

Is that robust?

a “super-cleint,” authorised on all?

a function that takes 0.5% from each account?

sufficient
for chan e of balance

I am worried about who/what can reduce my balance

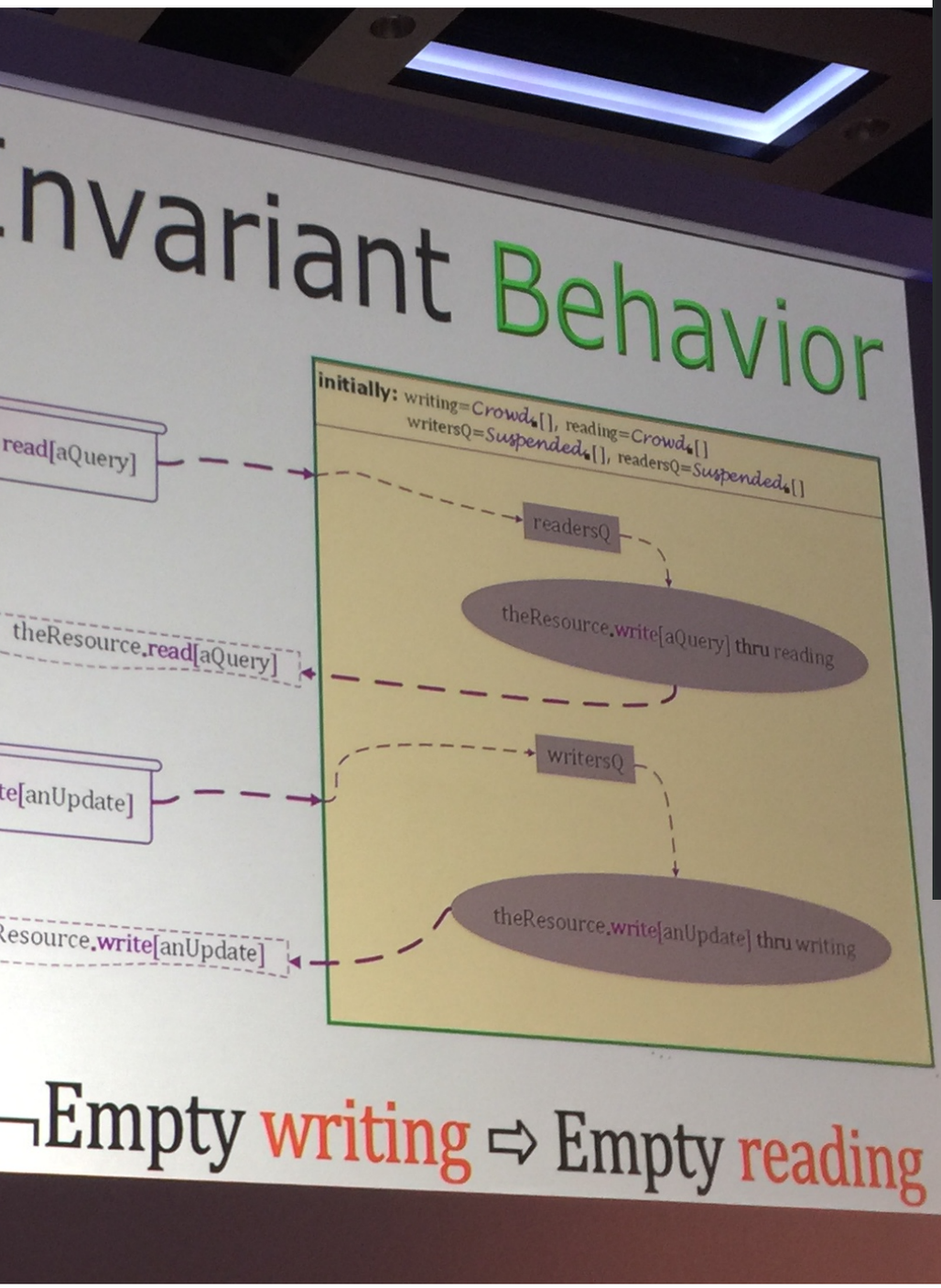
can authority increase?

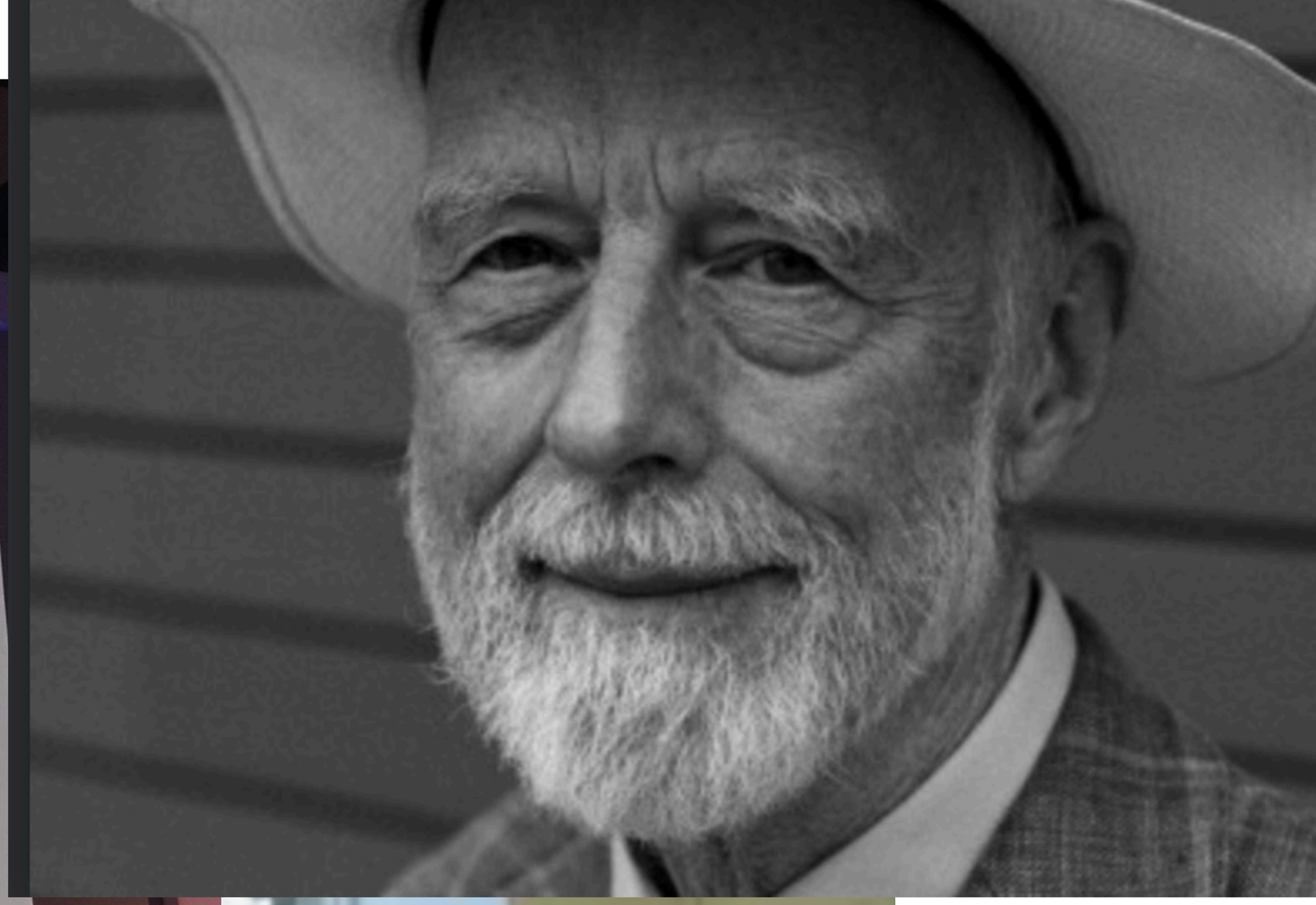
```
e.balance(c1) ≧ m
    { e.transferFrom                m) }
e.balance(c1) = e.bala                pre -m ∧
e.balance(c2) = e.balanc              re +m ∧ e.Authc
```

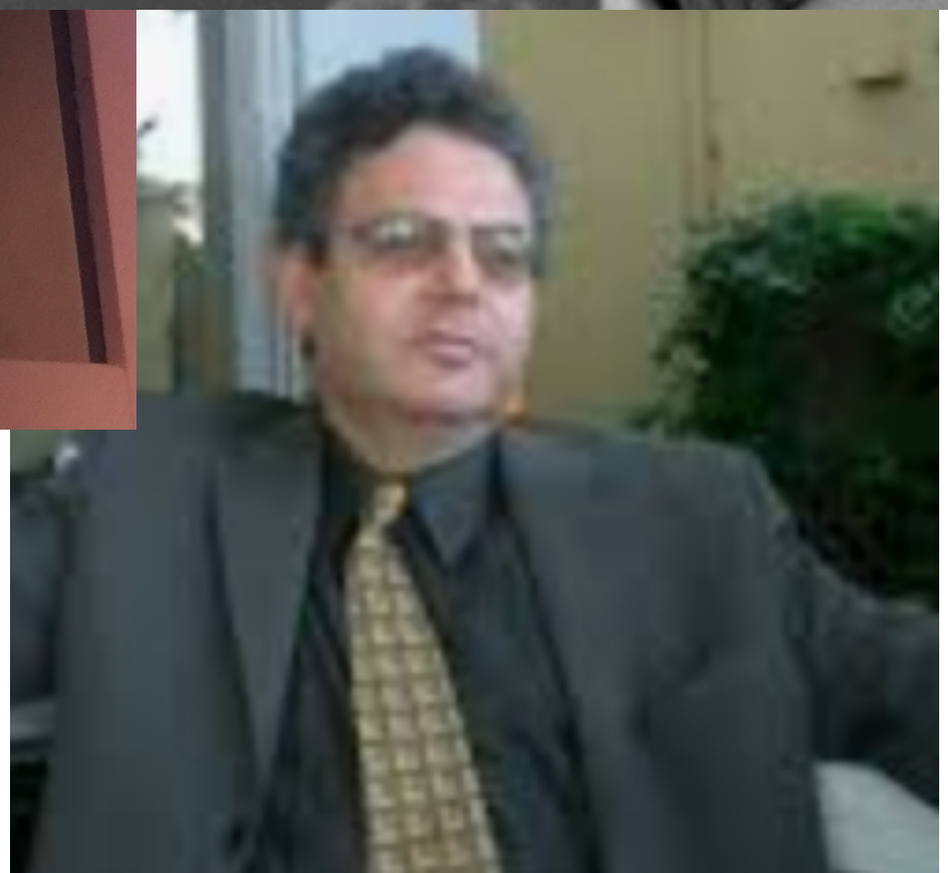
```
RC20 ∧ thi = c1 ∧ e.balance(c1) < m
        { .transf
∀ c. e.balance(c)
```

```
zed(c
ance(c) pre ∧
c2,m) →e.Authorized(c1,c2,m)]
```

```
totalSupply() }
… { e.allowanceOf(c2) } ….
… { e.balanceOf(c) } ….
```


nvariant Behavior
read[aQuery]
initially: writing=Crowd[], reading=Crowd[]
writersQ=Suspended[], readersQ=Suspended[]
readersQ
theResource.read[aQuery]
theResource.write[aQuery] thru reading
te[anUpdate]
writersQ
Resource.write[anUpdate]
theResource.write[anUpdate] thru writing
Empty writing ⇨ Empty reading





holistic specs - invariants ++

- state
- **time**
- **space**
- **control**
- **permission**
- **authority**
- **(when/where do they hold)**

holistic spec - authority

effect

`e.Authorized(c1,c2,m)` ≙

Prev(`c1.`**Calls**`(e,approve,c2,m)`)

∨

Prev(`e.Authorized(c1,c2,m+m')` ∧ `c2.`**Calls**`(e,transferFrom,c1,_,m')`)

∨

Prev(`e.Authorized(c1,c2,m)` ∧ ¬ `c2.`**Calls**`(e,transferFrom,c1,_,_)`)

necessary conditions

`c2` is authorised by `c1` for `m` iff

in previous step `c1` informed `e` that it authorised `c2` for `m`

or

in previous step `c2` was authorised for `m+m'` and spent `m'` for `c1`

or

in previous step `c2` was authorised for `m` and did not spend `c1`

classical vs holistic

e:ERC20 ∧ e.balance(cl) >m ∧ e.balance(cl') = m' ∧ cl≠cl'
{ e.transfer(cl',m) ∧ Caller=cl}
e.balance(cl) = e.balance(cl)$_{pre}$ -m ∧ e.balance(cl')$_{pre}$ = m'+m

e:ERC20 ∧ e.balance(cl) >m ∧ e.balance(cl') = m' ∧ cl≠cl'
∧ Authorized(e,cl,cl'')
{ e.transferFrom(cl',m) ∧ Caller=cl''}
e.balance(cl) = e.balance(cl)$_{pre}$ -m ∧ e.balance(cl')$_{pre}$ = m'+m

e:ERC20 ∧ e.balance(cl) >m ∧ e.balance(cl') = m'
{ e.allow(cl') ∧ Caller=cl }
Authorized(e,cl,cl'')

… … another 7 specs … ….

Holistic

- *necessary* conditions for some action/effect
- *explicit* about emergent behaviour

Classical

- per function; :*sufficient* conditions for some action/ effect
- *explicit* about individual function, and *implicit* about emergent behaviour

∀ e:ERC20. ∀ cl: Client.
[e.balance(cl) = **Prev**(e.balance(cl)) - m)
→
(∃cl',cl'': Client.
Prev (cl.**Calls**(e.transfer(cl',m)))
∨
Prev (Authorized(e,cl,cl'') ∧ cl''.**Calls**(e.transferFrom(cl,cl',m)))]

Authorized(cl,cl') ≜ ∃m: Nat. **Prev**(cl.**Calls**(e.approve(cl',m)))

Example2: DAO simplified

DAO, a “hub that disperses funds”; (https://www.ethereum.org/dao).
… clients may contribute and retrieve funds :

- `payIn(m)` pays into DAO m on behalf of client
- `repay()` withdraws all moneys from DAO

Vulnerability: Through a buggy version of `repay()`, a client could re-enter the call and deplete all funds of the DAO.

classical spec

Assuming DAO keeps a directory of contributions, and require:

R1: directory is compatible with the amount of ether kept in the DAO, and

R2: that withdraw reduces the ether by that amount.

R1: ∀d:DAO. d.ether = $\sum_{\texttt{cl} \in \texttt{dom(d.directory)}}$ `d.directory(cl)`

R2: d:DAO ∧ n:Nat ∧ d.directory(cl)=n>0 ∧ this=cl

{ `d.repay()` }

d.directory(cl)=0 ∧ d.**Calls**(cl,send,`n`)

d:DAO ∧ n:Nat ∧ d.directory(cl)=0 ∧ this=cl

{ `d.repay()` }

"nothing changes"

This spec avoids the vulnerability, 😅

provided the attack goes through the function repay. 😢

To avoid the vulnerability in general, we need to inspect the specification of *all* the functions in the DAO.

DAO - interface has *nineteen* functions. 🥵

holistic

∀ cl:External. **∀**d:DAO. **∀**.n:Nat.
[cl.**Calls**(`d.repay()`) ∧ d.Balance(cl) = n
→
d.ether≧`n` ∧ **Will**(d.**Calls**(cl.send(`n`))]

This specification avoids the vulnerability, regardless of which function introduces it:
The DAO will always be able to repay all its customers.

d.Balance(cl) ≡ 0 if cl.**Calls**(d,initialize())
m+m' if **Was**(d.Balance(cl),m) ∧ cl.**Calls**(d.payIn(m'))
0 if **Was**(cl.**Calls**(d.repayIn())
Was(d.Balance(cl)) otherwise

classical vs holistic

∀d:DAO. d.ether = ∑ cl ∈ dom(d.directory) d.directory(cl)

d:DAO ∧ n:Nat ∧ d.directory(cl)=n>0 ∧ this=cl
{ d.repay() }
d.directory(cl)=0 ∧ d.Calls(cl.send(n))

d:DAO ∧ n:Nat ∧ d.directory(cl)=0 ∧ this=cl
{ d.repay() }
"nothing changes"

… … specs for another 19 functions ….

Classical

- per function; *sufficient* conditions for some action/effect
- *explicit* about individual function, and *implicit* about emergent behaviour

Holistic

- *necessary* conditions for some action/effect
- *explicit* about emergent behaviour

∀ cl:External. ∀d:DAO. ∀.n:Nat.
[cl.**Calls**(d.repay()) ∧ d.Balance(cl) = n
→
d.ether≧n ∧ **Will**(d.**Calls**(cl.send(n))]

d.Balance(cl) ≡ m if cl.**Calls**(d,initialize,m)
m+m' if **Prev**(d.Balance(cl),m)
∧ **Prev**(cl.**Calls**(d.payIn(m'))
0 if **Prev**(cl.**Calls**(d.repayIn())

Bank and Account

- Banks and Accounts
- Accounts hold money
- Money can be transferred between Accounts
- A banks' currency = sum of balances of accounts held by bank

[Miller et al, Financial Crypto 2000]

Bank Account - 2

classical | robustness

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_3**: The bank can only inflate its own currency
- **Pol_4**: No one can affect the balance of an account they do not have.
- **Pol_5**: Balances are always non-negative.
- **Pol_6**:….

[Miller et al, Financial Crypto 2000]

Pol_4 — holistic

- **Pol_4**: No-one can affect the balance of an account they do not have

a:Account ∧ **Will** (**Changes**(a.balance)) **in** S)

→

necessary condition

∃o∈S. [External(o) ∧ Access(o,a)]

This says: If some execution starts now and involves at most the objects from S, and modifies a.balance at some future time, then at least one of the objects in S is external to the module, and can access a directly now.

Bank Account - 2

classical | robustness

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_3**: The bank can only inflate its own currency
- **Pol_4**: No one can affect the balance of an account they do not have.
- **Pol_5**: Balances are always non-negative.
- **Pol_6**: **??????**

[Miller et al, Financial Crypto 2000]