Towards Reasoning in the presence of code of unknown provenance

- or, trust and risk in an open world -

Lorentz Workshop 2015

Sophia Drossopoulou (Imperial),
James Noble (Victoria University Wellington),
Mark S. Miller (Google),
Toby Murray (NICTA)

Reasoning with Code of Unknown Provenance

Hoare Rules - Method Call

When receiver belongs to class C

$$\frac{PRE(m,C)= P \qquad POST(m,C)=Q}{x : C \wedge P[x/this,y/par] \ \{ z= x.m(y) \} \ Q[x/this,y/par,z/res]}$$

What if I know nothing about receiver?

$$\frac{}{true \ \{ z= x.m(y) \} \ true}$$

Trust and Risk in Open Systems - research questions -

- Objects collaborate with other objects of unknown provenance.
- Objects may unknowingly be dealing with malicious objects; they are therefore exposed to risks. Nevertheless, they proceed with the business.
- **No central trusted authority.**
- Therefore,
  - “our” code must be very “robust”,
  - we need means to specify trust and risk.
  - we need means to reason about adherence to such specifications.

Trust and Risk in Open Systems - our contributions -

- To specify trust and risk, we propose
    - **obeys** predicate: an object adheres to a specification,
    - **MayAccess** predicate: an object may read some property
    - **MayAffect** predicate: an object may affect some property
    - specifications talk of necessary conditions
- Predicates **obeys**, **MayAccess** and **MayAffect** are hypothetical and often conditional.
- Hoare logic rules to reason about trust and risk.
- Apply our ideas on the Escrow Exchange (Miller et.al., ESOP'13).
- **Simplifying Assumptions**:
    - We do not consider concurrency and distribution (code in ESOP'13 does).
    - We assume that different arguments to our methods are not aliases (but easy to expand).
    - We do not consider whether two Accounts can trade (but CanTrade in the PLAS paper).

Our findings for the Escrow

- We could write the specification.
- We could prove adherence to specification (by hand).
- The specification is weaker than we, and the Escrow authors, had expected. And no code can satisfy the stronger specification (we think)

Remaining Talk

Terminology: open world, trust and risk

Escrow Agent - Our running example

Hoare Logic

Terminology: open, trust and risk

What do we mean by open system?

We model open systems through dynamic linking of any, unknown, potentially malicious module M'.

Definition

M ⊨ Policy iff

$\forall$ M'.

$\forall \kappa \in$ *Arising*(M'*M): M'*M, κ ⊨ Policy

M' represents the "adversary".

Arising(M'*M) restricts configurations to those reachable though execution of code from M'*M.

What do we mean by trust?

Trust is relative to a configuration (κ), an object reference (o) and a policy-specification (Policy).

trust is hypothetical; no “trust” bit.

Definition

M, $\kappa \models$ o **obeys** Spec   iff

$\forall$ Policy $\in$ Spec.

$\forall$ $\kappa' \in$ *Reach*(M, κ):   M, $\kappa' \models$ Policy[o/this]

Reach(M, κ): intermediate configurations reachable from κ.

What do we mean by risk?

Risks are effects against which we want to guard our objects.

policy Pol_deal_1:
pre: ….

{ res = this.deal(m,g) ; }

post: ….

∀ p. p **obeys** ValidPurse ….

[p.balance = p.balance_{PRE} ∨

∃*bp*. *bp*=… ¬ (*bp* **obeys** ValidPurse) ∧ MayAccess_{PRE} (*bp*,p)]

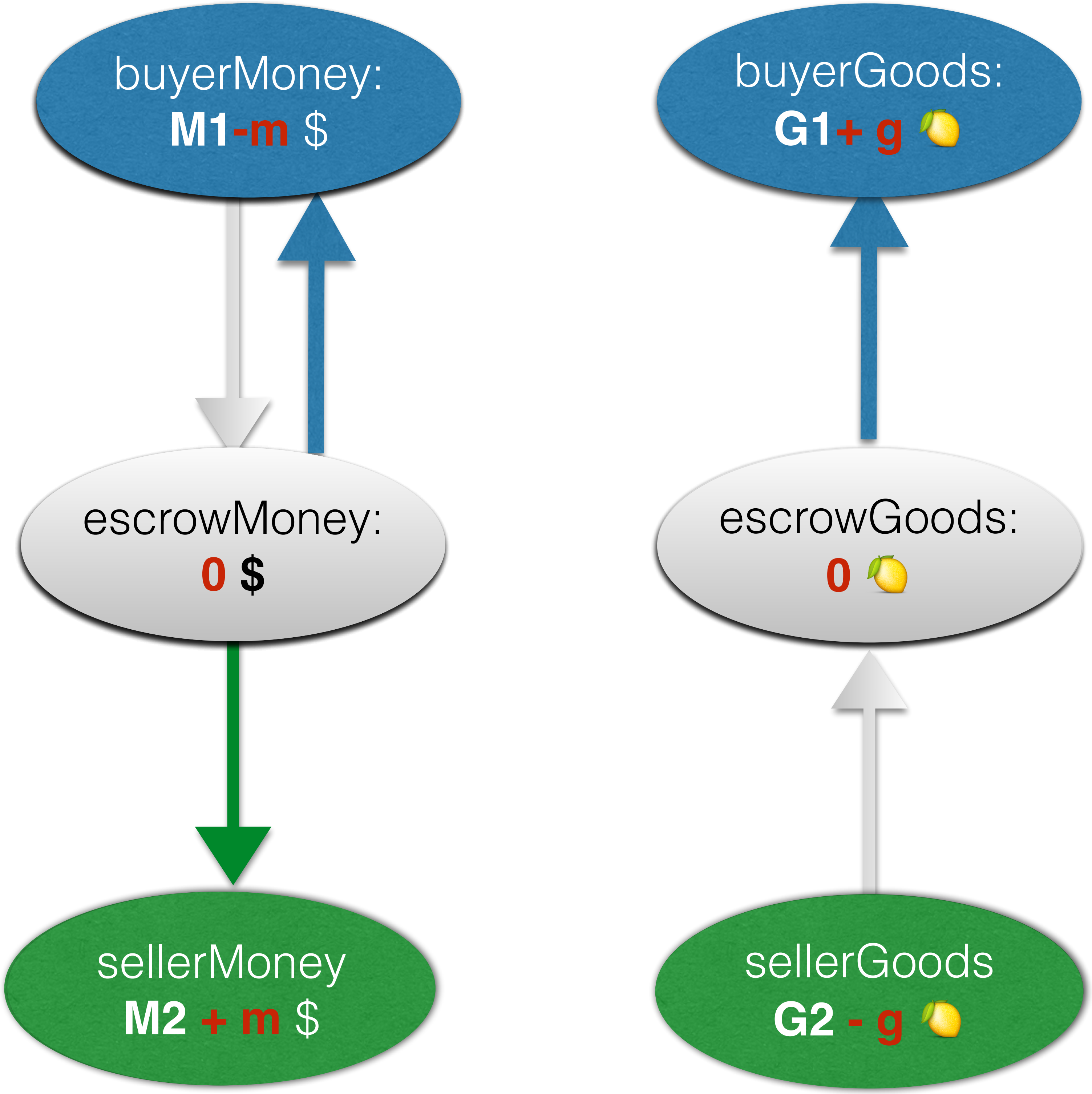
Escrow Agent - Our running example

Escrow Agent - Remit

(proposed by Miller, van Cutsem, Tulloh, ESOP 2013)

- **Buyer** and **Seller** want to exchange **g** goods for **m** money.
- **Buyer** does not trust **Seller**; **Seller** does not trust **Buyer**.
- Escrow Agent to make the exchange.
- If insufficient money or goods, then no exchange.
- Escrow Agent does not trust **Buyer** nor **Seller**, nor any **Banks**.
- Escrow Agent to mitigate risk to **Buyer** and **Seller.**

Escrow Agent - First Attempt



1. pay **m** to escrowMoney
   from buyerMoney

2. if no success then exit

// sufficient money

3. pay **g** to escrowGoods
   from sellerGoods

4. if no success then
   pay **m** to buyerMoney
   from escrowMoney
   exit

// sufficient money and goods

5. pay **g** to buyerGoods
   from escrowGoods

6. pay **m** to sellerMoney
   from escrowMoney

Exchange of **g** goods for **m** money

The Escrow purses

- intermediate store of $\mathbf{m}$ money, and $\mathbf{g}$ goods
- allow exchange to be undone, if insufficient goods or money
- Agent interrogates the escrow purses, to determine whether deposits were successful.
- Therefore, the correctness of process depends on the integrity of the escrow purses.
- But … where do escrow purses come from?

Where do Escrow Purses come from?

- The Agent has them before the transaction.

  No! This would require the Agent to know about all possible purses. Remember, no central authority.

- Seller and Buyer supply the escrows purses.

  No! It would require Seller and Buyer to have agreed before the transaction. Remember: Seller and Buyer do not trust each other.

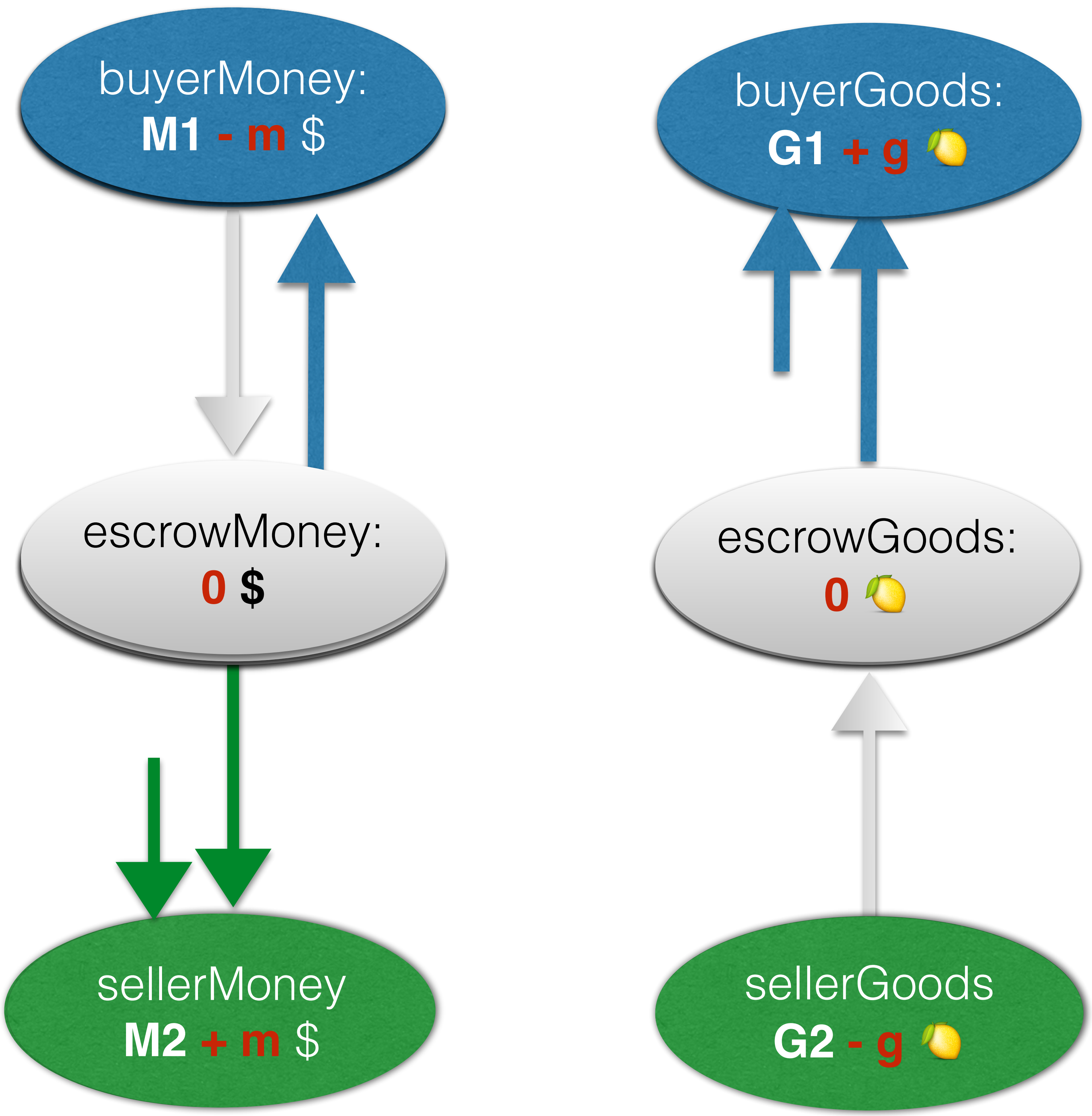
- The Agent asks the associated Banks to supply the escrows purses.

  No! It would require the Agent to know about all possible banks. Remember, no central authority.

- The Agent asks sellerMoney to make one, and buyerGoods to make another one.

  Yes!

Escrow Agent code - v1



Exchange of **g** goods for **m** money

1a. escrowMoney
= sellerMoney.sprout()

1b.res= escrowMoney.
deposit (buyerMoney,**m**)

2. if **!**res then exit

// sufficient money

3a. escrowGoods =
buyerGoods.sprout()

3b. res = escrowGoods.
deposit (buyerGoods,**g**)

4. if **!**res then
buyerMoney.deposit
(escrowMoney,**m**)
exit

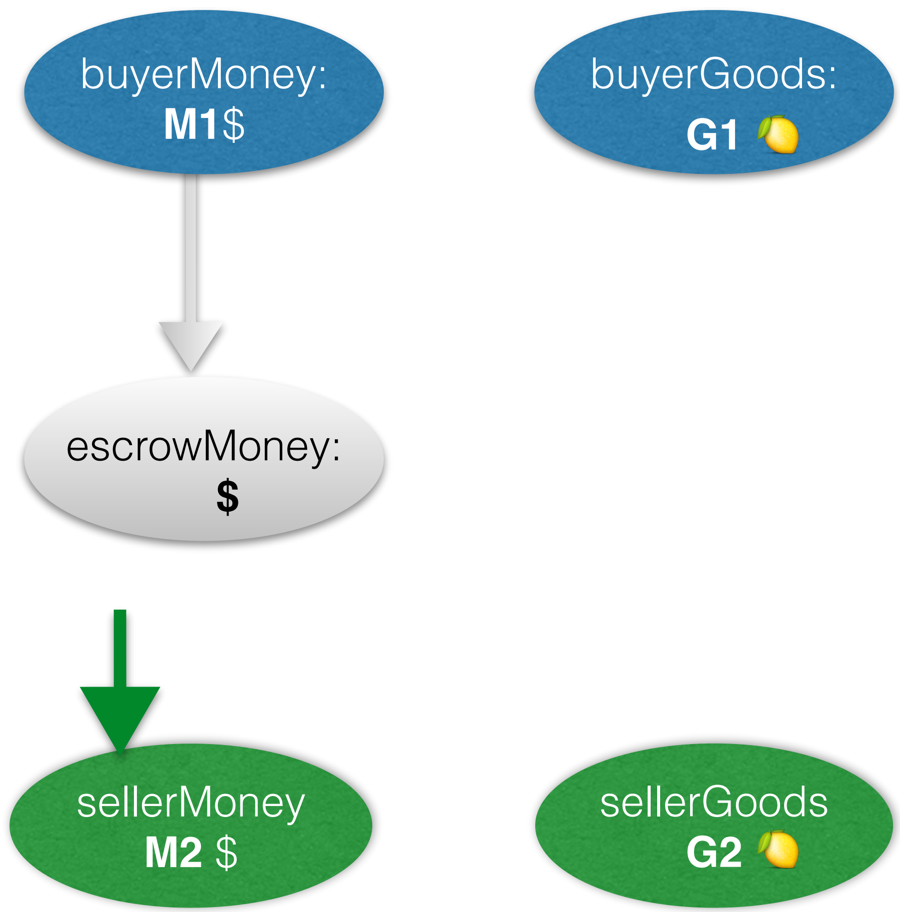
// sufficient money and goods

5. buyerGoods.
deposit(escrowGoods,**g**)

6. sellerMoney.
deposit (escrowMoney,**m**)

Risk and Trust
Has Escrow Agent version1 fulfilled its remit?

- Buyer and Seller want to exchange g goods for m money.
- Buyer does not trust Seller; Seller does not trust Buyer.
- Escrow Agent to make the exchange.
- If insufficient money or goods, then no exchange.
- Escrow Agent does not trust Buyer nor Seller, nor any Banks.
- Escrow Agent to mitigate risk to Buyer and Seller.



1a. escrowMoney = sellerMoney.sprout()

1b.res= escrowMoney. deposit (buyerMoney,**m**)

2. if **!res** then exit

More Risks

- Similar risk scenarios exist for malicious escrowGoods.
- Similar risk scenarios exist for when escrowMoney is sprouted from buyerMoney, or when escrowGoods is sprouted from sellerMoney.

Escrow Agent - Second Attempt

Escrow Agent - v2 - summary

- Extend Purse's remit to ascertain trust and limit risk.
- Add introductory phase to Escrow Agent code, which assesses trustworthiness of Purses.

Escrow Agent - Second Attempt

ValidPurse specification

ValidPurse specification v2- overview

specification ValidPurse{

policy Pol_deposit_1:

successful report, implies trust and enough funds, and transfer

policy Pol_deposit_2:

unsuccessful report, implies no trust or not enough funds, and no transfer

policy Pol_sprout:

sprout() creates a Purse of same trustworthiness

policy Pol_protect_balance:

balance cannot be affected, unless you hold the purse itself

ValidPurse - deposit_1

successful report, implies trust and enough funds; transfer

policy Pol_deposit_1:

pre: amt : Number $\wedge$ amt $\geqq$ 0

{ res=this.deposit(prs, amt) }

post:

res = true $\rightarrow$

// FUNCTIONAL

$prs.balance_{pre}$ - amt $\geqq$ 0 $\wedge$

prs.balance = $prs.balance_{pre}$ - amt $\wedge$

this.balance = $this.balance_{pre}$ + amt $\wedge$

// TRUST

prs **obeys** ValidPurse $\wedge$

// RISK

Note: conditional trust

[MayAccess(o,p) $\rightarrow$ $MayAccess_{pre}$(o,p)])

ValidPurse - deposit_**2**

unsuccessful report, implies **no** trust or **insufficient** trust , and no transfer

policy Pol_deposit_2:

pre: amt : Number $\wedge$ amt $\geqq$ 0

{ res=this.deposit(prs, amt) }

post:

res = **false** $\rightarrow$

// FUNCTIONAL & TRUST

prs.balance$_{pre}$ - amt **<** 0 $\vee$ **NOT**(prs **obeys** ValidPurse)

// RISK

$\forall$ p, o.

(p **obeys**$_{pre}$ ValidPurse $\wedge$ o **:**$_{pre}$ Object. $\rightarrow$

p.balance$_{pre}$ = p.balance $\wedge$

[MayAccess(o,p) $\rightarrow$ MayAccess$_{pre}$(o,p)]

ValidPurse - sprout

sprout creates a Purse of same trustworthiness

policy Pol_sprout:

pre: true

{ res=this.spout() }

post

// FUNCTIONAL & TRUST

res **obeys** ValidPurse $\land$ res.balance=0

// RISK

$\forall$ p, o.

(p $\textbf{obeys}_{pre}$ ValidPurse $\land$ o $\mathbf{:}_{pre}$ Object. $\rightarrow$

(p≠prs,this $\rightarrow$ $p.balance_{pre} = p.balance$) $\land$

[MayAccess(o,p.balance) $\rightarrow$ $MayAccess_{pre}$(o,p.balance)]

ValidPurse - protect_balance

balance cannot be affected, unless you hold the purse itself

policy Pol_protect_balance:

∀ p, o.

(p **obeys** ValidPurse ∧ o **:**Object. →

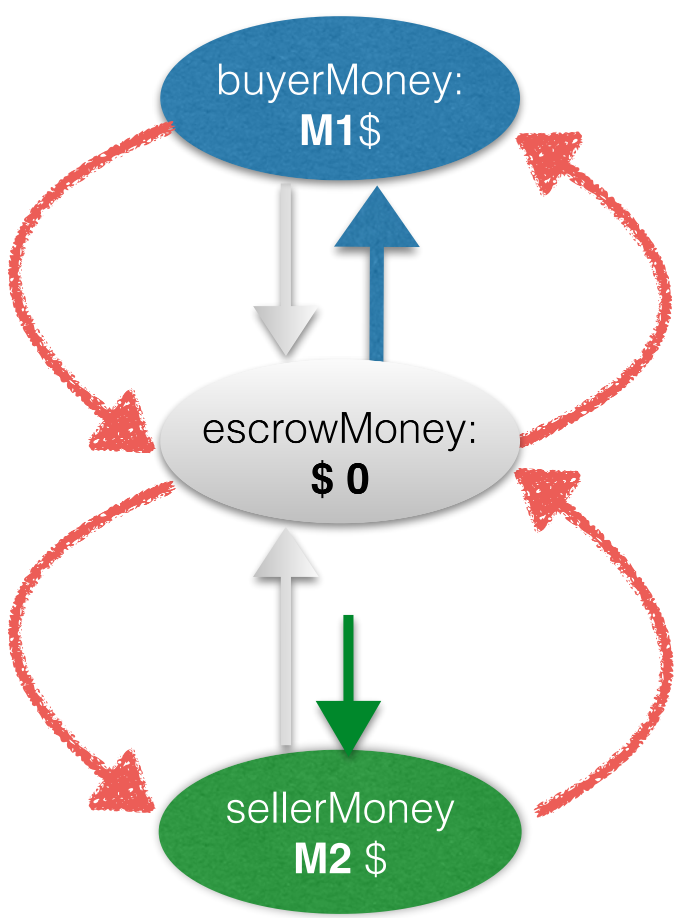
[MayAffect(o,p.balance) → MayAccess(o,p)])

Note - necessary, rather than sufficient condition

Escrow Agent - Second Attempt

code

EscrowAgent - establishing trust



escrowMoney = sellerMoney.sprout()

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

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

// **res=true** ∧ escrowMoney **obeys** ValidPurse
// → buyerMoney **obeys** ValidPurse

if !res then exit // sellerMoney **obeys** ValidPurse →
// ¬(buyerMoney **obeys** ValidPurse)

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

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

// **res=true** ∧ buyerMoney **obeys** ValidPurse
// → escrowMoney **obeys** ValidPurse

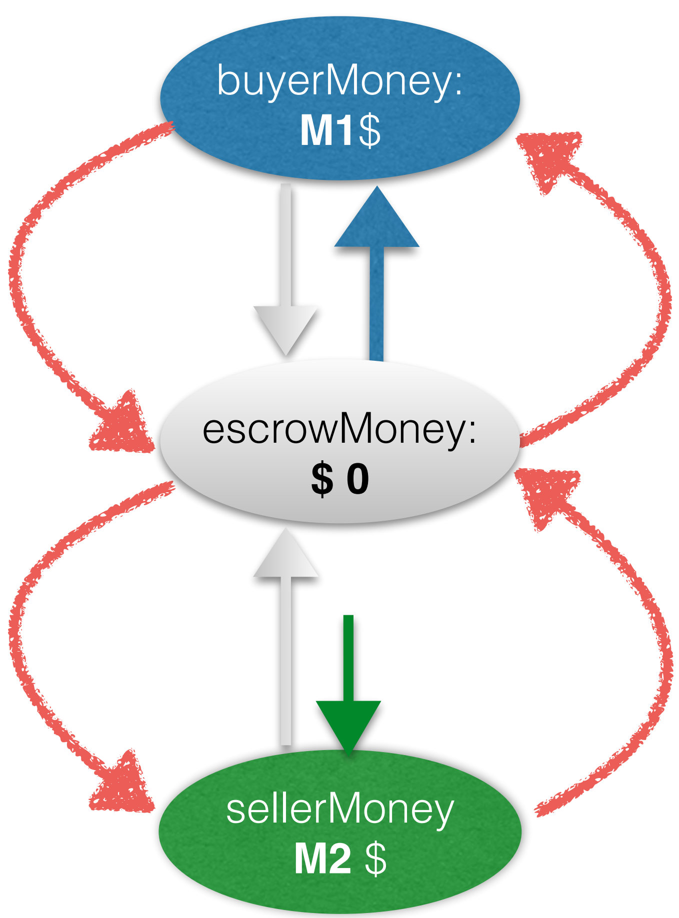
if !res then exit

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

if !res then exit

// buyerMoney **obeys** ValidPurse ↔ seller **obeys** ValidPurse

EscrowAgent - the risk while establishing trust



escrowMoney = sellerMoney.sprout()

// ∀p. p **obeys**$_{PRE}$ ValidPurse →
// [p.balance$_{PRE}$=p.balance ∨
// MayAccess$_{PRE}$(sellerMoney,p) ∧ ¬(sellerMoney **obeys** ValidPurse)

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

// ....

if !res then exit // ....

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

//∀p. p **obeys**$_{PRE}$ ValidPurse →
// [p.balance$_{PRE}$=p.balance ∨
// MayAccess$_{PRE}$(sellerMoney,p) ∧ ¬(sellerMoney **obeys** ValidPurse) ∨
// MayAccess$_{PRE}$(buyerMoney,p) ∧ ¬(buyerMoney **obeys** ValidPurse)]

if !res then exit // ....

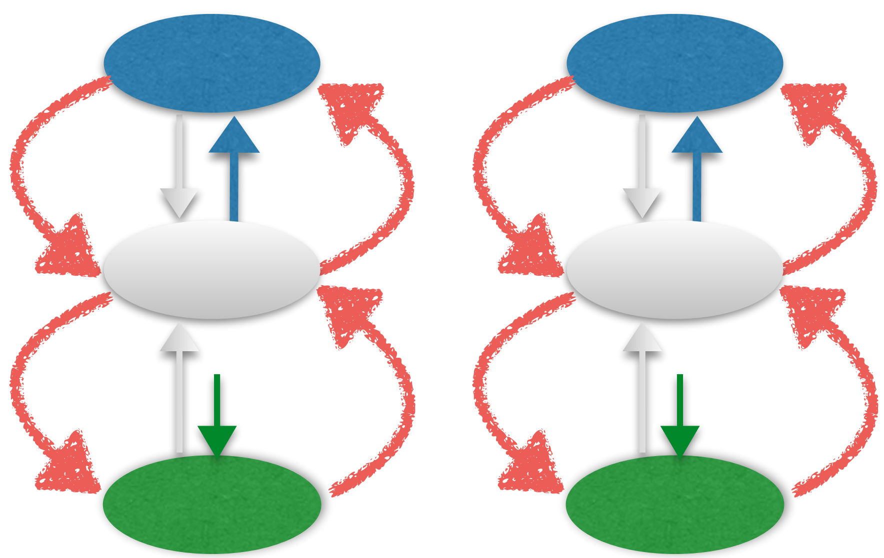
res= escrowMoney. deposit (buyerMoney,**0**)

// ....

if !res then exit

//∀p. p **obeys**$_{PRE}$ ValidPurse →
// [p.balance$_{PRE}$=p.balance ∨
// MayAccess$_{PRE}$(sellerMoney,p) ∧ ¬(sellerMoney **obeys** ValidPurse) ∨
// MayAccess$_{PRE}$(buyerMoney,p) ∧ ¬(buyerMoney **obeys** ValidPurse)]

EscrowAgent the full code



1st phase:
trustworthiness buyerMoney and sellerMoney
— as in previous slide

2nd phase:
trustworthiness buyerGood and sellerGood
—- similar to previous slide

3rd phase:
Do the transaction
—- as a couple of slides ago

Escrow Agent - Second Attempt

The specification

EscrowAgent specification - overview

Escrow Agent has fields buyerMoney, sellerMoney, buyerGoods, sellerGoods.

For the method call

this.deal(m,g)

we have four cases:

policy Pol_deal_1:

no malicious participants, sufficient money/goods, deal proceeds
res==true

policy Pol_deal_2:

no malicious participants, **insufficient** money/goods, deal does **not** proceed
res==false

policy Pol_deal_3:

buyer *or* seller malicious, deal does **not** proceed
res==false

policy Pol_deal_4:

buyer *and* seller malicious, deal **does** proceed
res==true

Pol_deal_1

no malicious participants, sufficient money/goods, deal proceeds
res==true

policy Pol_deal_1:

pre: m,g: Number

{ res = this.deal(m,g) ; }

post: res $\wedge$ MalPartPurses = $\varnothing$

$\Rightarrow$

$buyerMoney.balance_{PRE} \geqq \mathbf{m} \wedge sellerMoney.balance_{PRE} \geqq \mathbf{g} \wedge$
$buyerMoney = buyerMoney_{PRE} - \mathbf{m} \wedge sellerMoney = sellerMoney_{PRE} + \mathbf{m} \wedge$
$buyerGoods = buyerGoods_{PRE} + \mathbf{g} \wedge sellerGoods = sellerGoods_{PRE} - \mathbf{g}$

$\forall$ p. p $\in_{PRE}$ OtherPurses. p.balance = $p.balance_{PRE}$ $\wedge$
$\forall$ o,p. o $:_{PRE}$ Object. p $\mathbf{obeys}_{PRE}$ ValidPurse .
MayAccess(o,p.balance) $\Rightarrow$ MayAccess(o,p.balance) $_{PRE}$

MalPartPurses = { sellerPurse, buyerPurse, sellerGoods, buyerGoods} $\cap$ { p | $\neg$ p **obeys** ValidPurse }
OtherPurses = { p | p **obeys** ValidPurse } \ { sellerPurse, buyerPurse, sellerGoods, buyerGoods}

Pol_deal_4

seller AND buyer malicious
deal proceeds
res==true

policy Pol_deal_4:

pre: m,g: Number

{ res = this.deal(m,g) ; }

post: res $\wedge$ MalPartPurses $\neq \emptyset$

$\Rightarrow$

sellerMoney **obeys**$_{PRE}$ ValidPurse $\Leftrightarrow$ buyerMoney **obeys**$_{PRE}$ ValidPurse $\wedge$

$\wedge$

sellerGoods **obeys**$_{PRE}$ ValidPurse $\Leftrightarrow$ buyerGoods **obeys**$_{PRE}$ ValidPurse

$\wedge$

$\forall$ p. p $\in_{PRE}$ GoodPurses. [p.balance = p.balance$_{PRE}$

$\vee$ $\exists$ *bp* $\in_{PRE}$ MalPartPurses. MayAccess$_{PRE}$ (bp,p)]

$\wedge$

$\forall$ o,p. o :$_{PRE}$ Object. p $\in_{PRE}$ GoodPurses.

[MayAccess(o,p) $\Rightarrow$ MayAccess(o,p) $_{PRE}$

$\vee$ $\exists$ *bp* $\in_{PRE}$ MalPartPurses. MayAccess$_{PRE}$ (bp,p)]

Pol_deal_1 - 4 discussion

- not as strong as originally expected,
- deal code can never know whether in case 1 or case 4,
- but strong enough for “good participant”
- We want to make specification stronger. Rather than currently

$$\forall p.\ p\ \mathbf{obeys}_{PRE} ValidPurse.\ \ [\ p.balance = p.balance_{PRE}$$
$$\vee\ \ \exists\ bp \in_{PRE} MalPartPurses.\ MayAccess_{PRE}\ (bp,p)\ \ ]$$

- we would like

$$\forall p.\ p\ \mathbf{obeys}_{PRE} ValidPurse.\ \ [\ p.balance = p.balance_{PRE}$$
$$\vee\ \ \exists\ bp \in_{PRE} MalPartPurses.\ \mathbf{MayAffect}_{PRE}\ (bp,p.balance)\ \ ]$$

Hoare Logic

Hoare Tuples

- Hoare tuples of form P { **code** } Q⋈Q'

- P a one-state assertion, Q, Q' two-state assertions.

- P { **code** } Q⋈Q' promises that if the initial configuration satisfies P, then the final configuration will satisfy Q,
  and all intermediate configurations will satisfy Q'.

- M ⊨ P { **code** } Q⋈Q' iff
  ∀ M'. ∀ κ ∈ *Arising*(M'*M):
  (M'*M, κ ⊨ P ∧ M'*M, κ ⇝ κ'
  → M'*M, κ, κ' ⊨ Q)
  and
  (∀ κ'' ∈ *Reach*(M'*M, code, κ): M'*M, κ, κ'' ⊨ Q')

Hoare Rules - Structural (some)

$$\frac{P\ \{code\}\ Q \bowtie Q' \qquad P' \rightarrow P \qquad Q \rightarrow Q'' \qquad Q' \rightarrow Q'''}{P'\ \{code\}\ Q'' \bowtie Q'''}$$

$$\frac{P\ \{code\}\ Q \bowtie Q'}{P \wedge P'\ \{code\}\ Q \wedge Q' \bowtie Q'}$$

$$\frac{P\ \{code\}\ Q \bowtie Q' \qquad Spec = \forall x.P(x) \wedge \ldots}{P\ \{code\}\ Q \bowtie Q' \wedge \forall x.x\ \textbf{obeys}\ Spec \rightarrow P(x)}$$

Hoare Rules - Method Call

when receiver is trusted to obey Spec

$$\frac{PRE(m,Spec) = P \qquad POST(m,Spec) = Q}{x\ \textbf{obeys}\ Spec \land P[x/this,y/par]\ \{\ z = x.m(y)\ \}\ Q[x/this,y/par,z/res] \bowtie true}$$

even if received not trusted

$$\frac{}{\begin{array}{l} true\ \{\ z = x.m(y)\ \}\ true\ \bowtie\ \forall u,v.\ \mathit{MayAccess}(u,v) \rightarrow \\ \quad (\ \mathit{MayAccess}(u,v)_{pre} \lor \\ \quad\quad (\ \mathit{MayAccess}(x,u)_{pre} \lor \mathit{MayAccess}(y,u)_{pre}\ ) \land \\ \quad\quad (\ \mathit{MayAccess}(x,v)_{pre} \lor \mathit{MayAccess}(y,v)_{pre})\ \ ) \end{array}}$$

Hoare Rules - Framing

$$\frac{P\ \{code\}\ Q \bowtie Q' \qquad P \wedge Q' \rightarrow \mathit{Footprint}(code)\ \text{disjoint}\ \mathit{Footprint}(P')}{P \wedge P'\ \{code\}\ Q \wedge P' \bowtie Q' \wedge P'}$$

$$\frac{P\ \{code\}\ true \bowtie \forall u.\ \mathit{MayAffect}(u,P') \rightarrow Q'(u) \qquad P\ \{code\}\ true \bowtie \forall u.\ \neg Q'(u)}{P \wedge P'\ \{code\}\ true \bowtie P'}$$

Conclusions

- We introduced **MayAccess**, **MayAffect**, and **obeys**.
- These are hypothetical and conditional predicates.
- Hoare tuples extended by properties preserved. New Hoare rules.
- The concept of encapsulation needs to percolate to specification level.
- More work for concurrency, distribution, expressivity, framing, examples, encapsulation.