Towards Reasoning in the presence of code of unknown provenance

- or, trust and risk in an open world -

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

Reasoning with Code of Unknown Provenance

Hoare Rules - Method Call

In this talk, we argue:

- We can do better than that.
- It is important to do better than that.

And if I know nothing about receiver?

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

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
- 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).

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.
- 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).

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

$\mathsf{M} \models \mathsf{Policy}$ iff

$\forall\ \mathsf{M}'.$

$\forall\ \kappa \in \mathit{Arising}(\mathsf{M}' {*} \mathsf{M}):\quad \mathsf{M}' {*} \mathsf{M}, \kappa \models \mathsf{Policy}$

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, $\kappa \vDash$ o **obeys** Spec iff

$\forall$ Policy $\in$ Spec.

$\forall\ \kappa' \in \mathit{Reach}(M, \kappa)$: M, $\kappa' \vDash$ 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* **obeys** ValidPurse) ∧ $MayAccess_{PRE}$ (*bp*,p)]

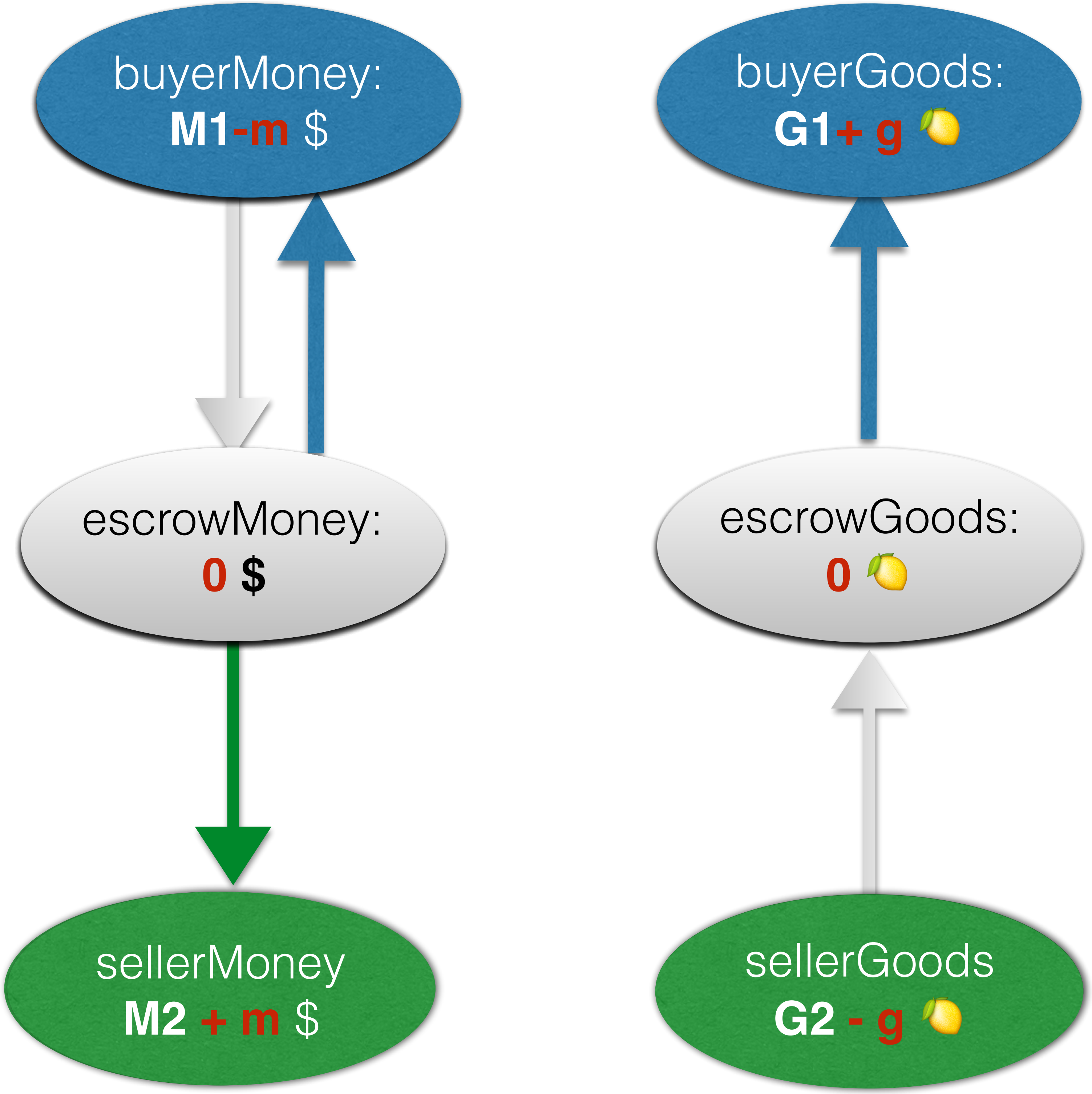
Escrow Agent - Our running example

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 m money, and 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.

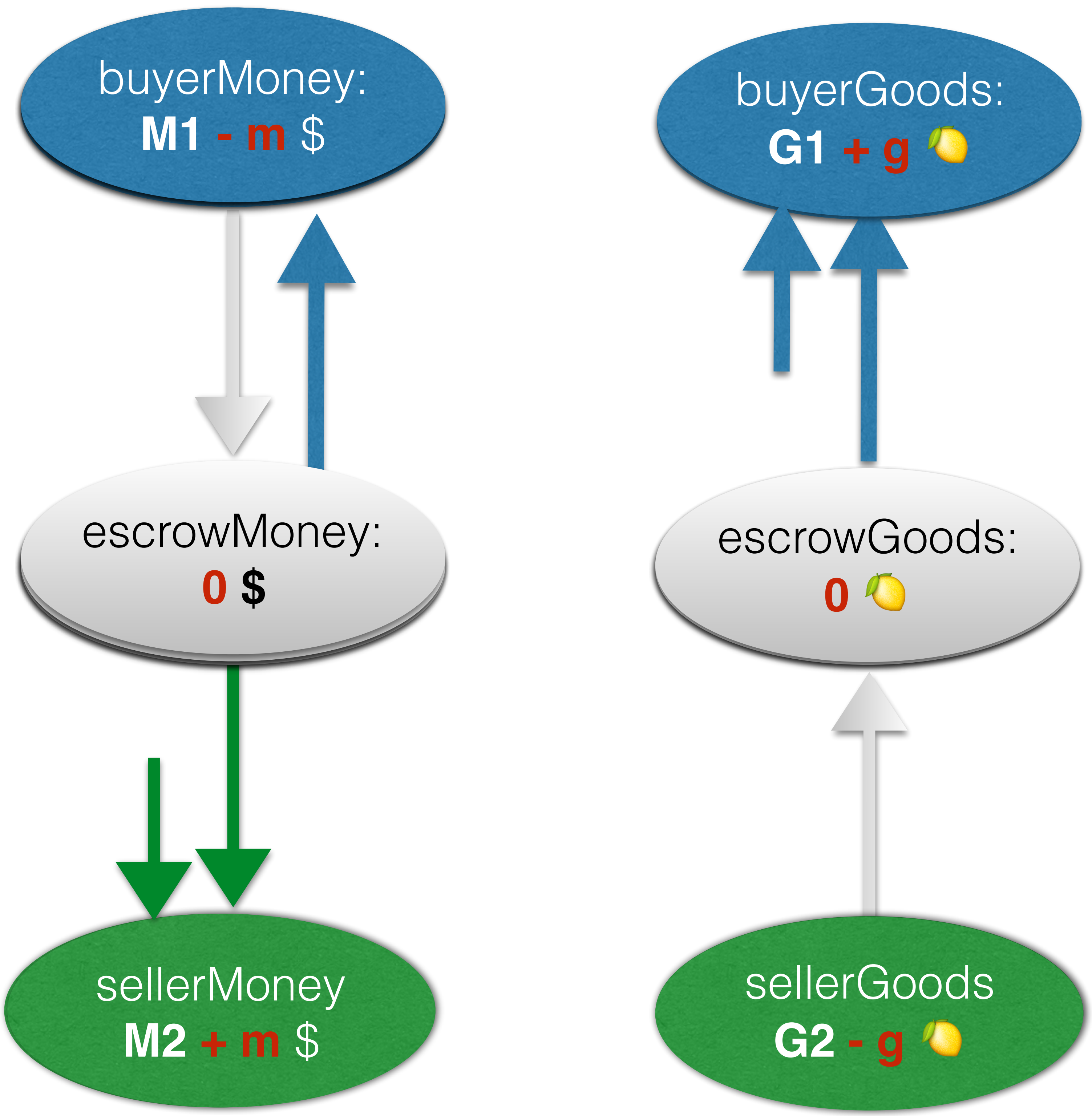
- The **Agent** asks the associated Banks **to supply the escrows purses.**

  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

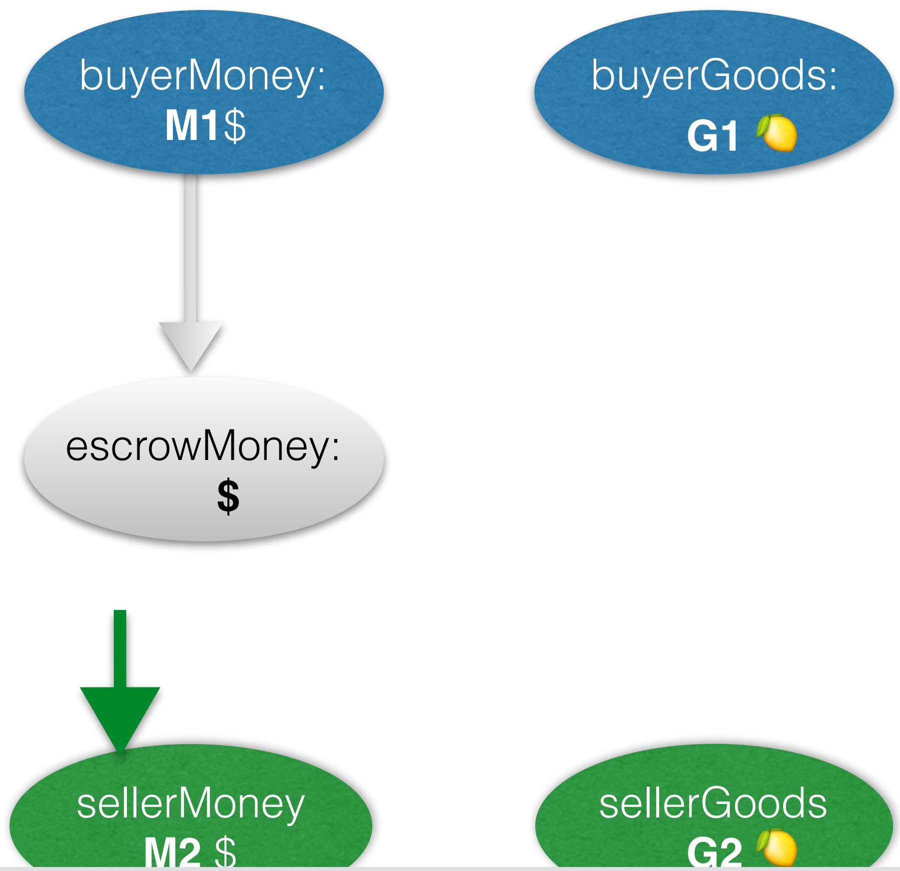
// sufficient money and goods

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

true { escrowMoney. deposit() } **true**

How much damage can it make?

Escrow Agent - Second Attempt

Escrow Agent - Second Attempt 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: { res=this.deposit(prs, amt) }

res=true implies trust, enough funds, and transfer of amt

policy Pol_deposit_2: { res=this.deposit(prs, amt) }

res=false implies no trust or not enough funds, and no transfer

policy Pol_sprout: { res=this.sprout( ) }

res is a new Purse of same trustworthiness

policy Pol_protect_balance:

balance cannot be affected, unless you hold the purse itself

ValidPurse - deposit_1

policy Pol_deposit_1:

pre: amt : Number ∧ amt ≧ 0

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

post:

res = true →

// FUNCTIONAL

$\text{prs.balance}_{\text{pre}}$ - amt ≧ 0 ∧

prs.balance = $\text{prs.balance}_{\text{pre}}$ - amt ∧

this.balance = $\text{this.balance}_{\text{pre}}$ + amt ∧

// TRUST

prs **obeys** ValidPurse ∧

// RISK

Note: conditional trust

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

ValidPurse - protect_balance

balance cannot be affected, unless you hold the purse itself

policy Pol_protect_balance:

$\forall$ p, o.

(p **obeys** ValidPurse $\wedge$ o **:**Object. $\rightarrow$

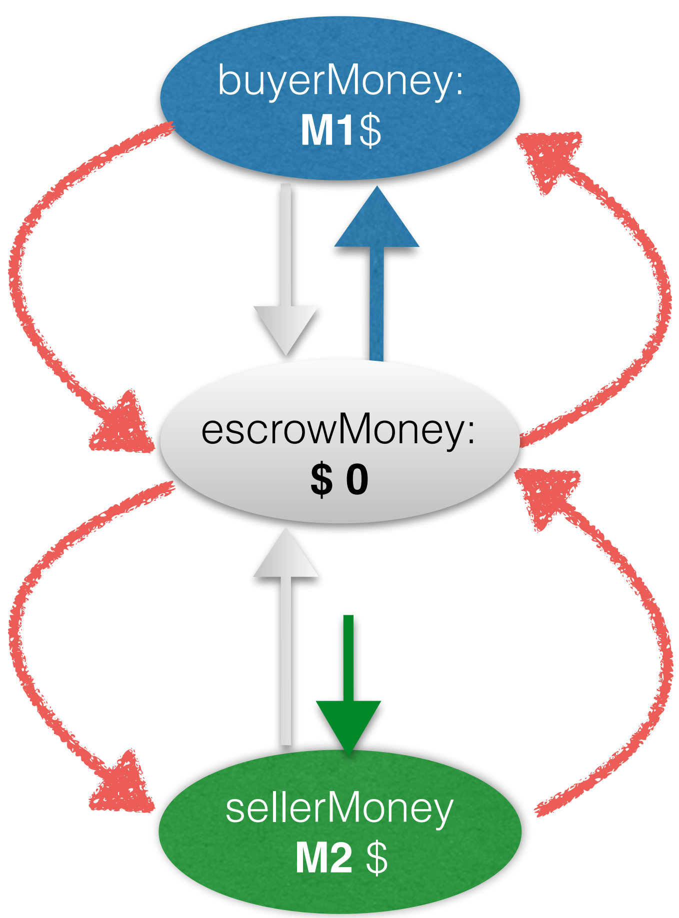
[MayAffect(o,p.balance) $\rightarrow$ MayAccess(o,p)])

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

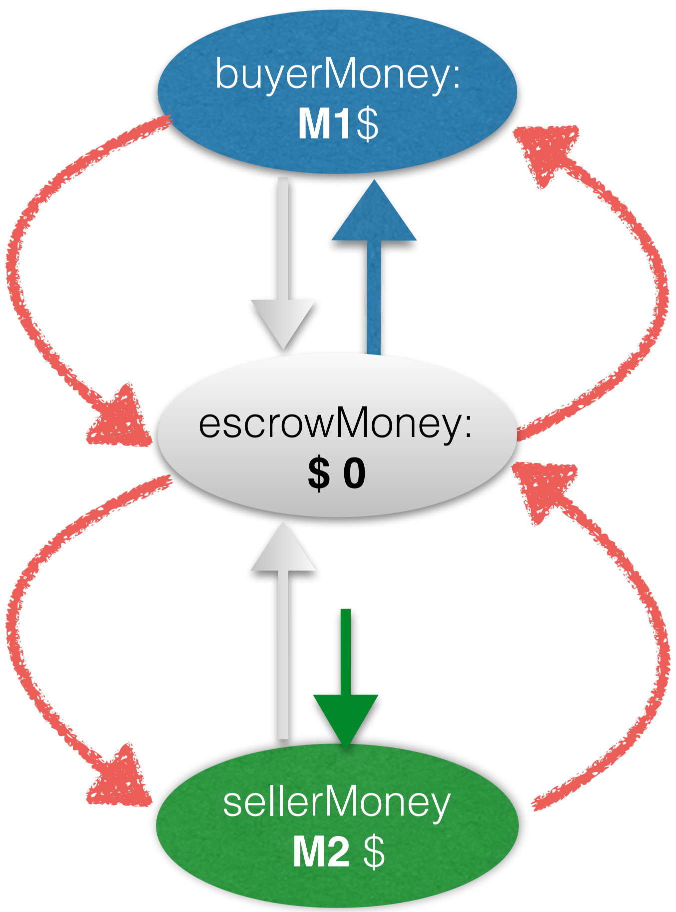
if !res then exit

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

if !res then exit

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

EscrowAgent - the risk while establishing trust



escrowMoney = sellerMoney.sprout()

// $\forall$p. p **obeys**$_{PRE}$ ValidPurse $\rightarrow$
// [p.balance$_{PRE}$=p.balance $\vee$
// MayAccess$_{PRE}$(sellerMoney,p) $\wedge$ $\neg$(sellerMoney **obeys** ValidPurse)

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

// ....

if !res then exit // ....

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

//$\forall$p. p **obeys**$_{PRE}$ ValidPurse $\rightarrow$
// [p.balance$_{PRE}$=p.balance $\vee$
// MayAccess$_{PRE}$(sellerMoney,p) $\wedge$ $\neg$(sellerMoney **obeys** ValidPurse) $\vee$
// MayAccess$_{PRE}$(buyerMoney,p) $\wedge$ $\neg$(buyerMoney **obeys** ValidPurse)]

if !res then exit // ....

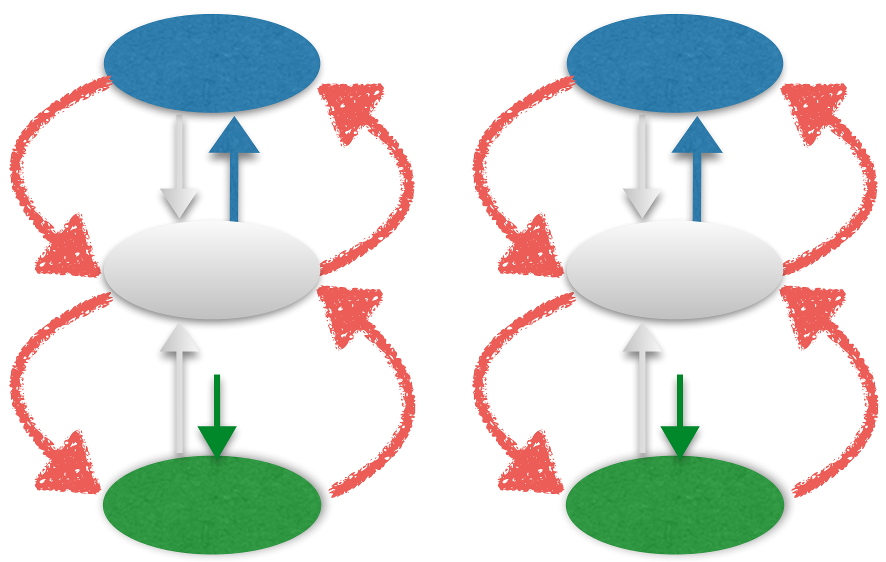
res= escrowMoney. deposit (buyerMoney,**o**)

// ....

if !res then exit

//$\forall$p. p **obeys**$_{PRE}$ ValidPurse $\rightarrow$
// [p.balance$_{PRE}$=p.balance $\vee$
// MayAccess$_{PRE}$(sellerMoney,p) $\wedge$ $\neg$(sellerMoney **obeys** ValidPurse) $\vee$
// MayAccess$_{PRE}$(buyerMoney,p) $\wedge$ $\neg$(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

For res := this.deal(m,g) we have four cases:

policy Pol_deal_1:

res ∧ buyer and seller "are good" ⇒

p

"Surprises":

- deal method not as "risk-free" as expected
- res=true does not imply successful transaction, nor that participants were good.

p

no Purse affected unless malicious participant had access before

policy Pol_deal_4:

res ∧ buyer and seller "are bad" ⇒

no Purse affected unless malicious participant had access before

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'.

- $M \models P$ { code } Q⋈Q' iff

  $\forall M'.\ \forall \kappa \in \mathit{Arising}(M'{*}M)$:

  $(\ M'{*}M, \kappa \models P \quad \wedge \quad M'{*}M, \kappa \rightsquigarrow \kappa'$

  $\rightarrow \quad M'{*}M, \kappa, \kappa' \models Q\ )$

  and

  $(\ \forall \kappa'' \in \mathit{Reach}(M'{*}M, \text{code}, \kappa):\ \ M'{*}M, \kappa, \kappa'' \models Q'\ )$

Hoare Rules - Structural (some)

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

$$\frac{\begin{array}{l} P\ \{code\}\ Q\ \bowtie\ Q' \\ Spec = \text{spec}\{\ Pol\_1, \ldots Pol\_i, \ldots\ Pol\_n\ \} \end{array}}{P\ \{code\}\ Q\ \bowtie\ Q' \wedge \forall x.x\ \textbf{obeys}\ Spec \rightarrow Pol\_i[x/this]}$$

$$\frac{\begin{array}{c} P\ \{code\}\ Q\ \bowtie\ Q' \\ P' \rightarrow P \qquad Q \rightarrow Q'' \qquad Q' \rightarrow Q''' \end{array}}{P'\ \{code\}\ Q''\ \bowtie\ Q'''}$$

$P' \rightarrow P$ iff $\kappa \models P'$ implies $\kappa \models P$

$Q \rightarrow Q''$ iff $\kappa, \kappa' \models Q$ implies $\kappa, \kappa' \models Q''$

$$\frac{\begin{array}{l} P\ \{code\}\ Q\ \bowtie\ Q'' \\ P', Q' \rightarrow P \end{array}}{P'\ \{code\}\ Q' \rightarrow Q\ \bowtie\ Q'''}$$

$P', Q' \rightarrow P$ iff

$\kappa \models P' \wedge \kappa, \kappa' \models Q'$ implies $\kappa \models P$

Hoare Rules - Method Call

when receiver is trusted to obey Spec

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

and regardless of whether receiver is trusted

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

Hoare Rules - Framing

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

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

Summary

- 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. More case studies. Ongoing design: refinement of the predicates, and new predicates.

Conclusions

In this talk, we argued:

- We can reason in the presence of "untrusted"/"unknown" code
- It is important to do that.

We need to specify

- what *will* happen,
- as well as what *will not* happen