First Steps towards Reasoning about Risk and Trust in the open world: the Escrow example

Sophia Drossopoulou & James Noble

Based on a talk given at iFM 2014 on the 9th September 2014, Bertionoro, Italy

October 2014

Risk and Trust in the open world - in terms of the Escrow example

In the open world, code of unknown provenance is dynamically loaded and linked, without prior static checks.

Thus, trusted objects co-operate with untrusted objects. , and are, unavoidably, exposed to risks.

Through the use of *object capabilities*, code can be written so as to reduce risks to objects.

We want to be able to

- Describe establishing trust.
- Formally specify the risk to objects.
- Reason how code adheres to trust/risk specification.

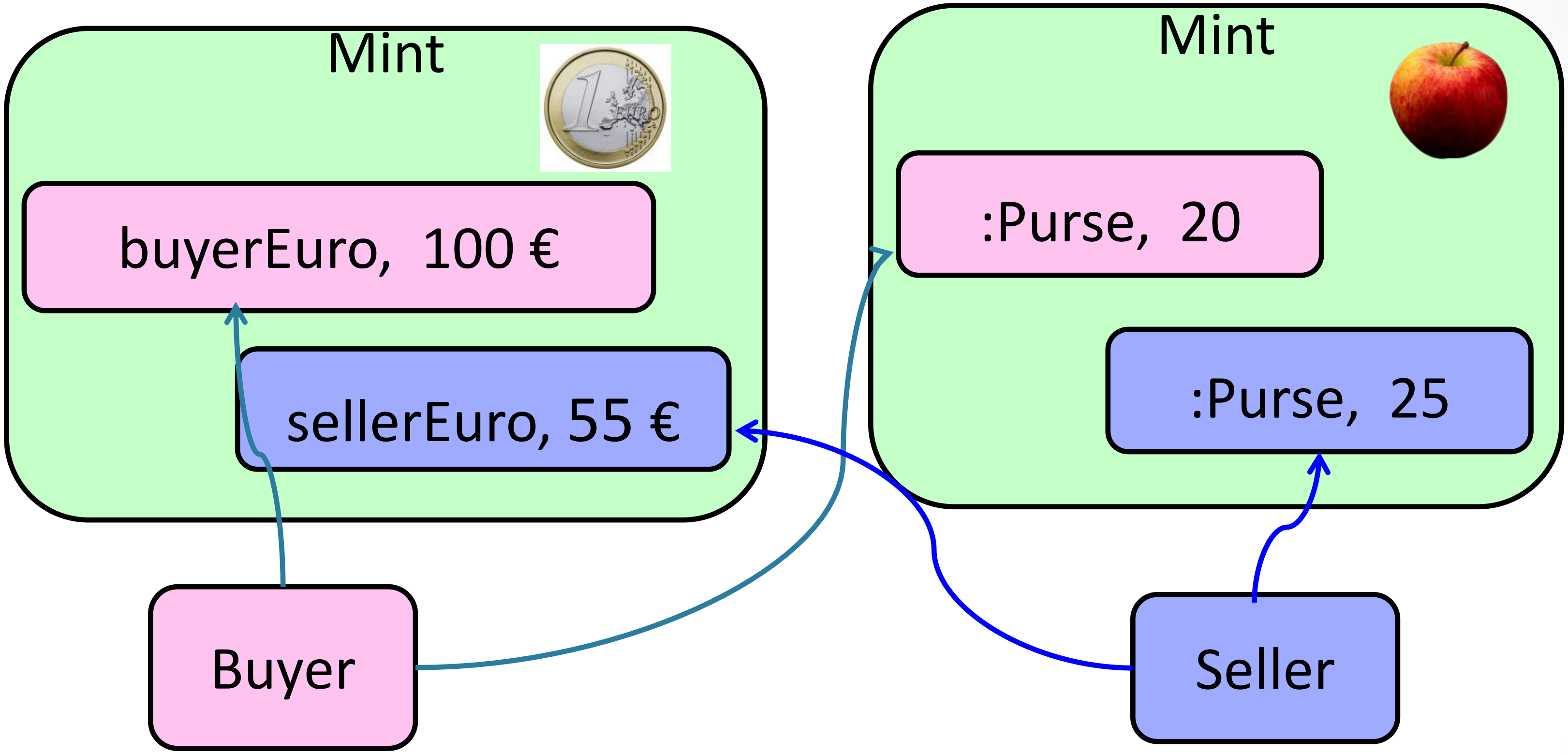
We will demonstrate our ideas in terms of the Escrow example, proposed by Mark Miller et al, ESOP'2013.

The Escrow Example

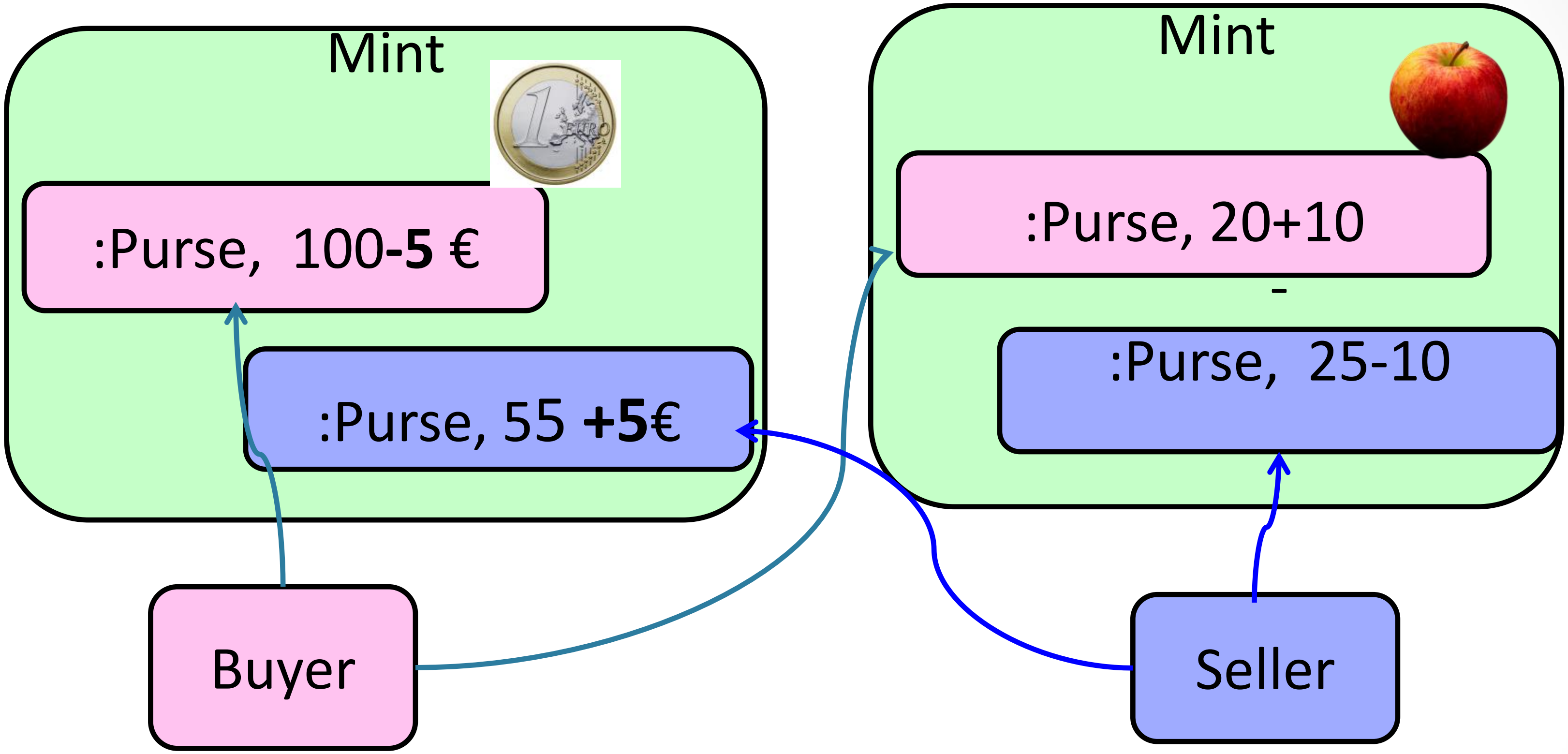
Escrow - buying apples securely

- Setup
  - Buyer has 100 €, and 20 .
  - Seller has 55 €, and 25 .
  - Buyer wants to buy 10 for 5 €.
  - Seller wants to sell 10 for 5 €.
  - Seller and Buyer do not trust each other.
- Questions:
  - How to organize the € and transfer?
  - What are the risks?

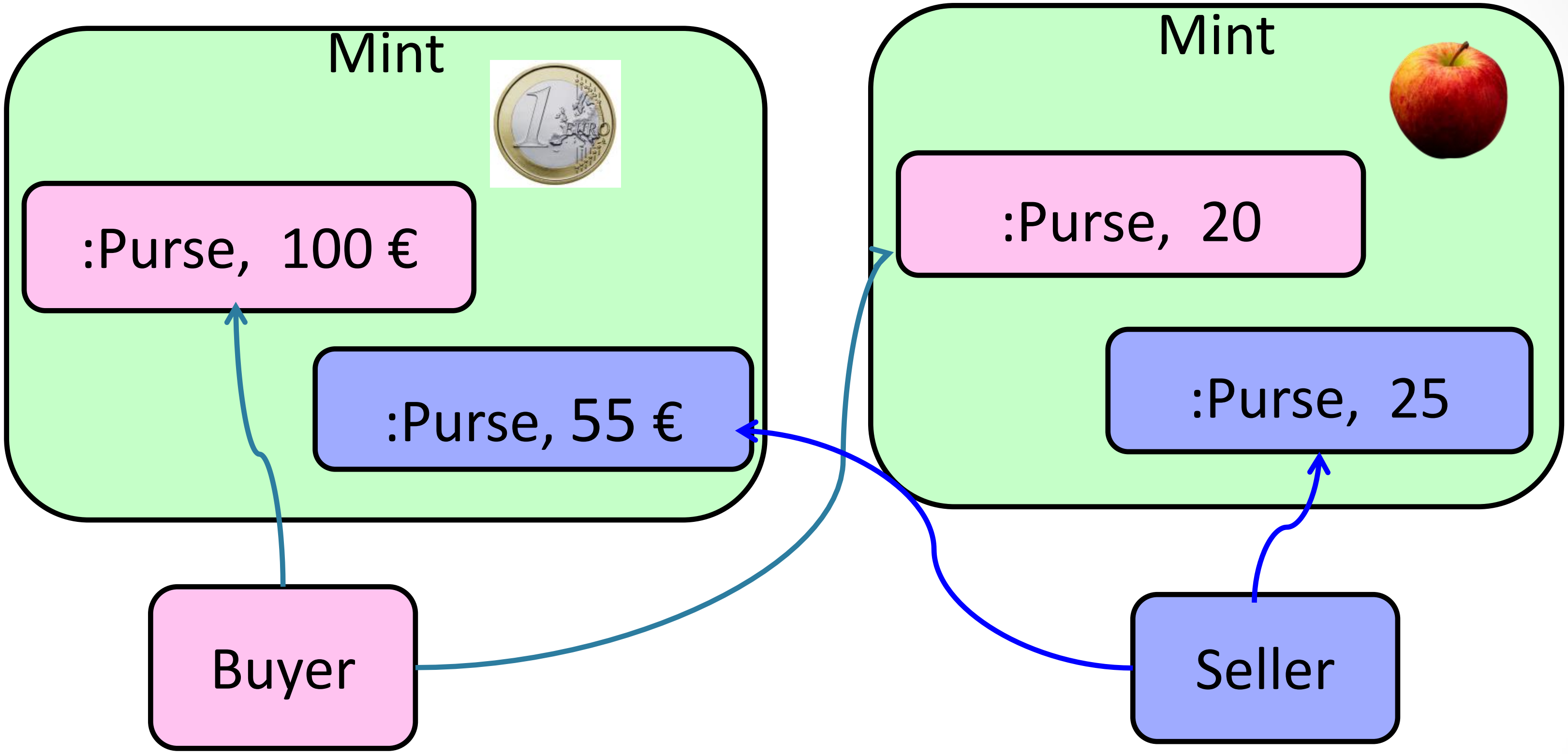
Buying Apples - before



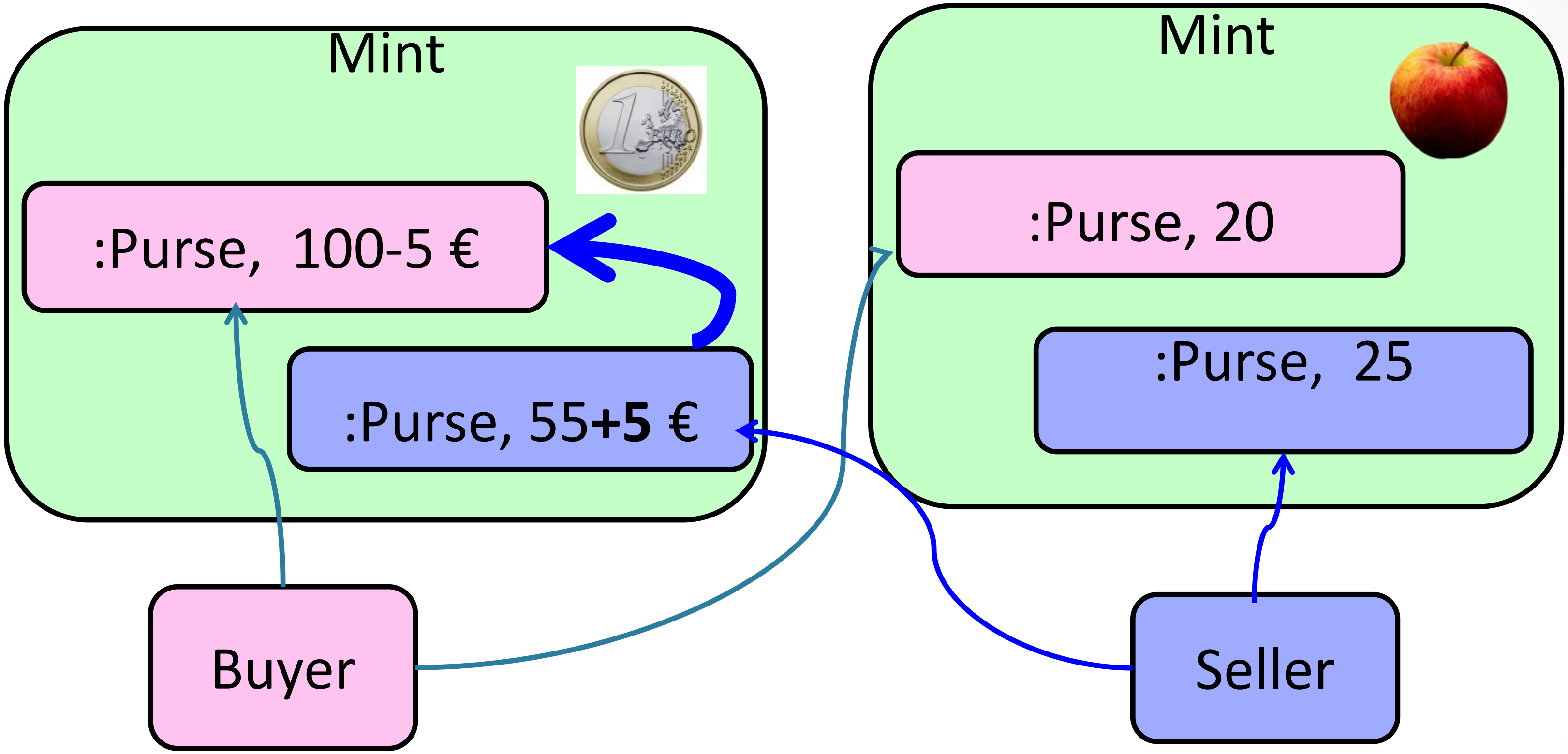
Buying Apples - after



Buying Apples – how?

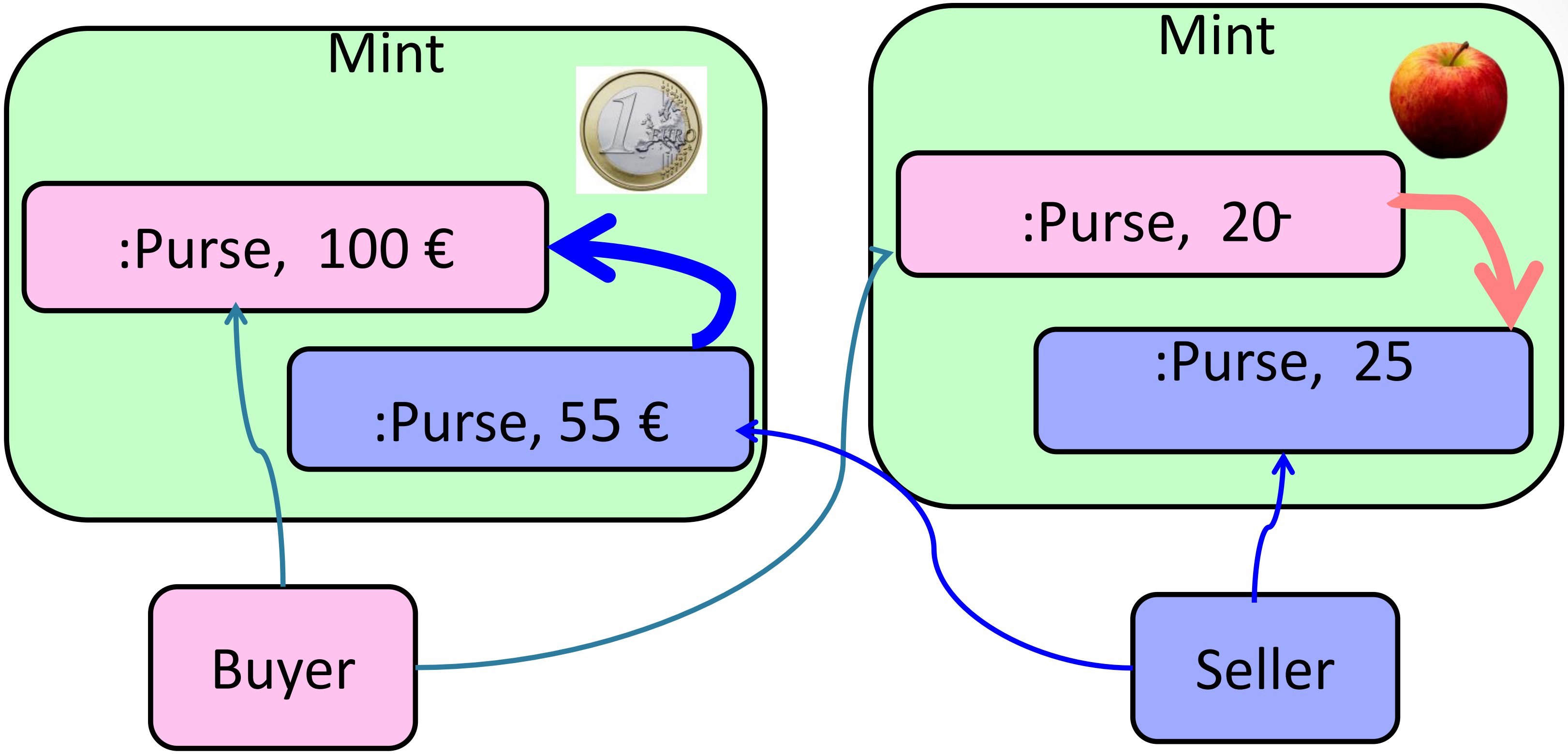


Buying Apples 1st attempt: pass purses



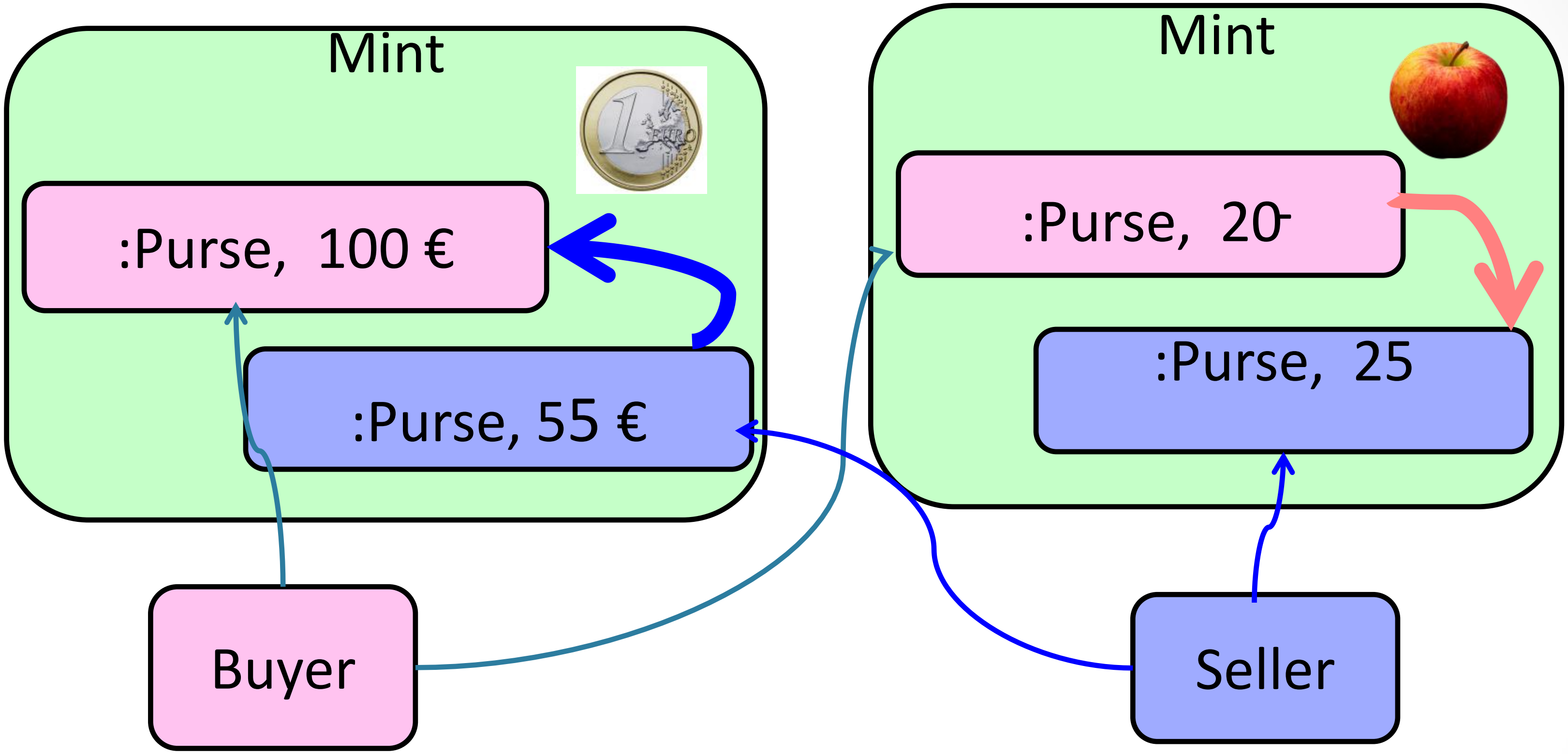

```
sellerEuros.transfer(5,buyerEuros);
```

Buying Apples 1st attempt: pass purses




```
sellerEuros.transfer(5,buyerEuros);
buyerApples.transfer(10,sellerApples);
```

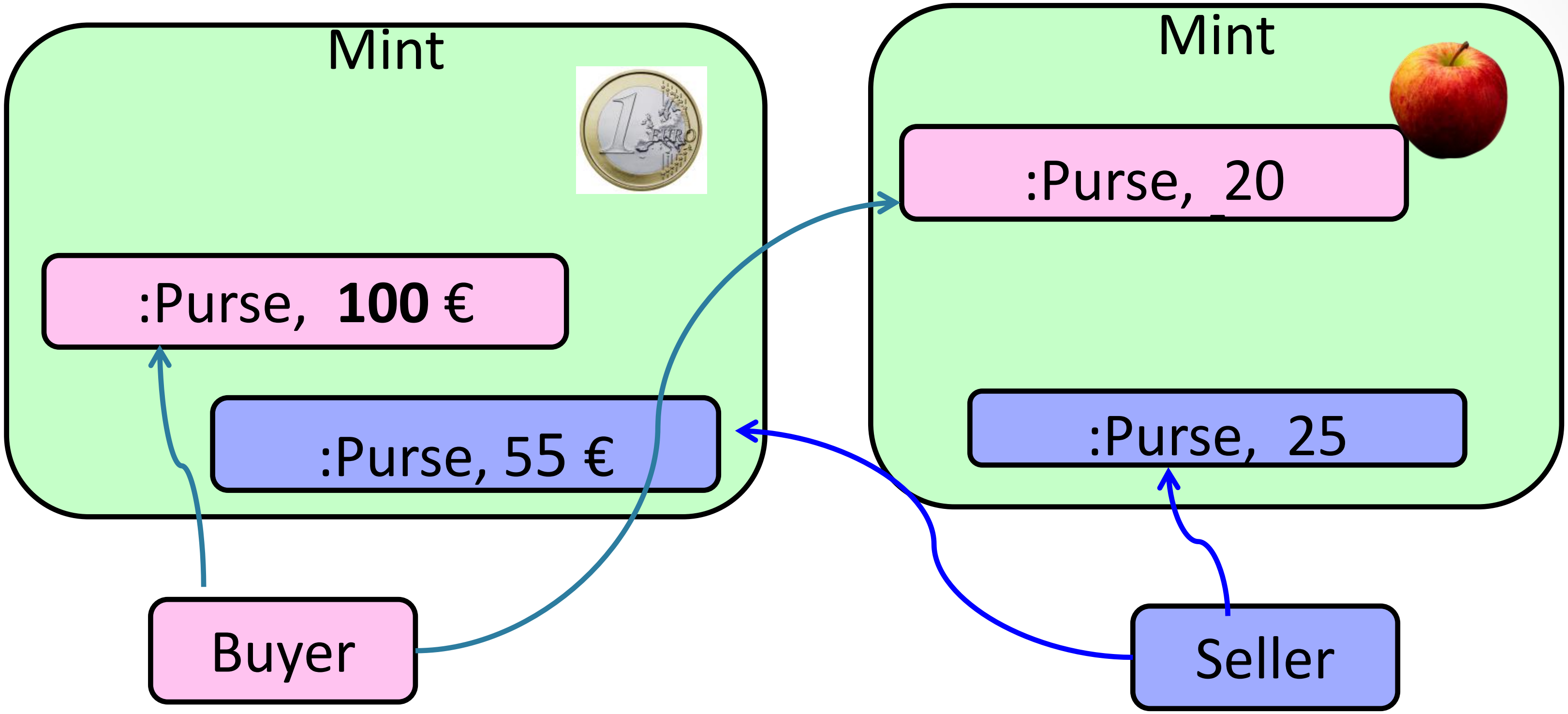
Buying Apples 1st attempt pass purses - **Risk**



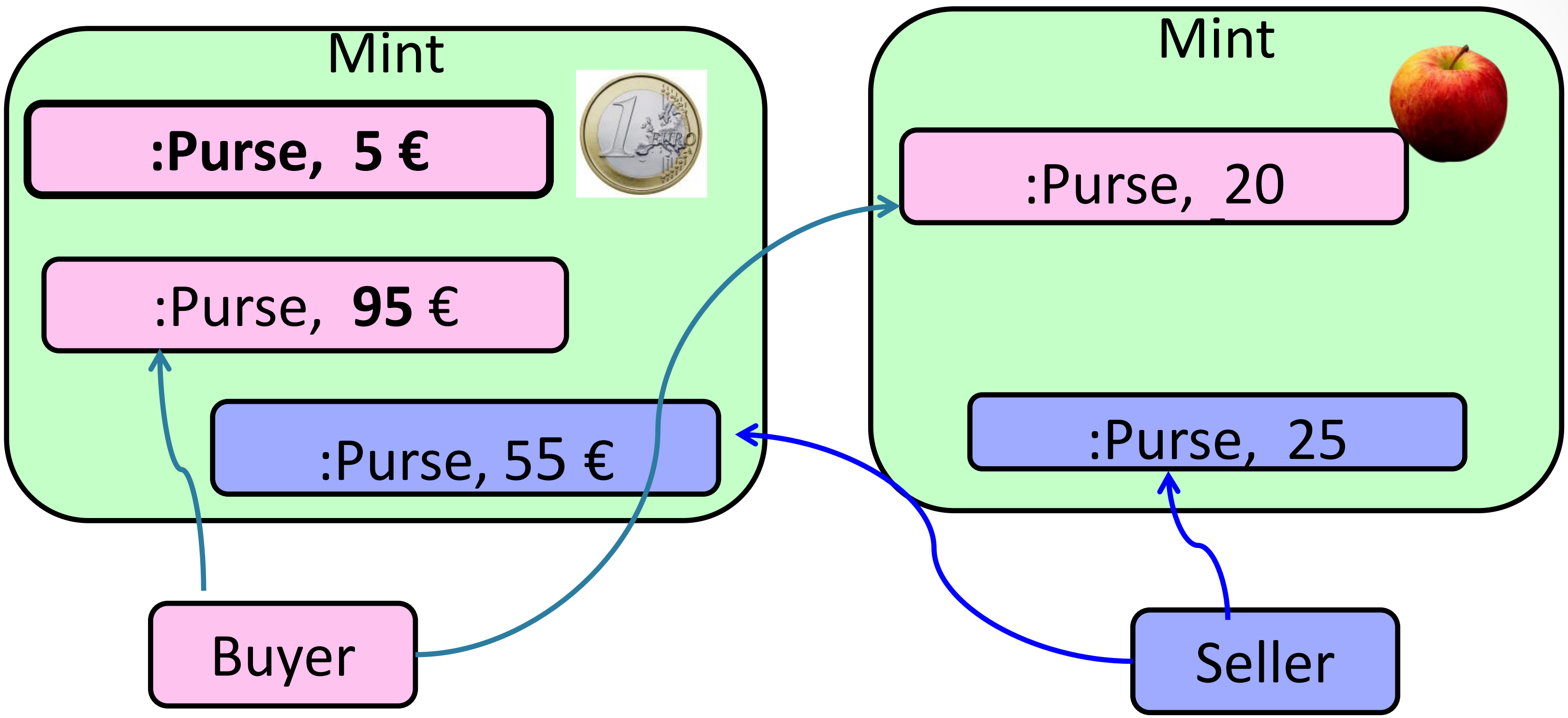
Risk to Buyer: - 100 €, 0

Risk to Seller: 0€, -15

Buying Apples 2nd Attempt: tmp purses

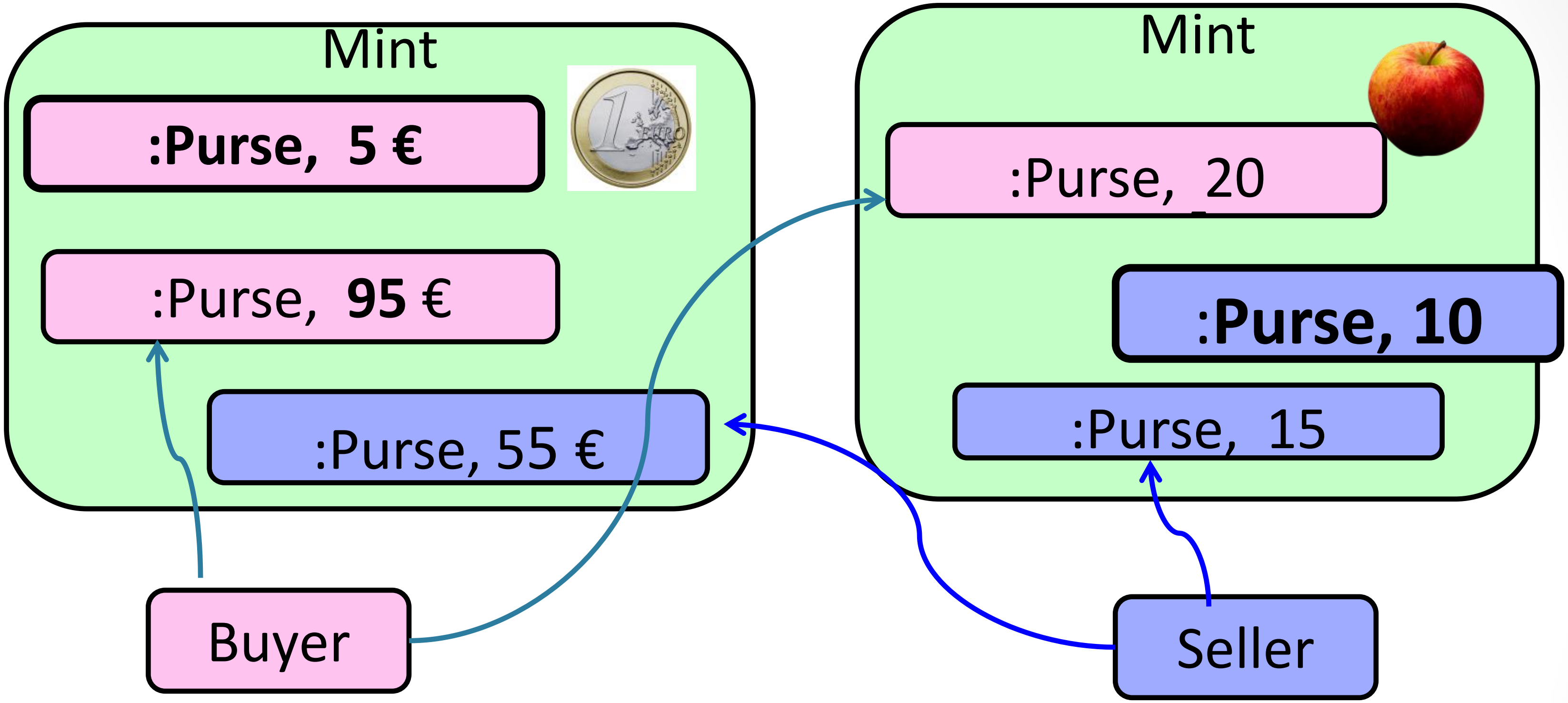


Buying Apples – 2nd Attempt: tmp purses



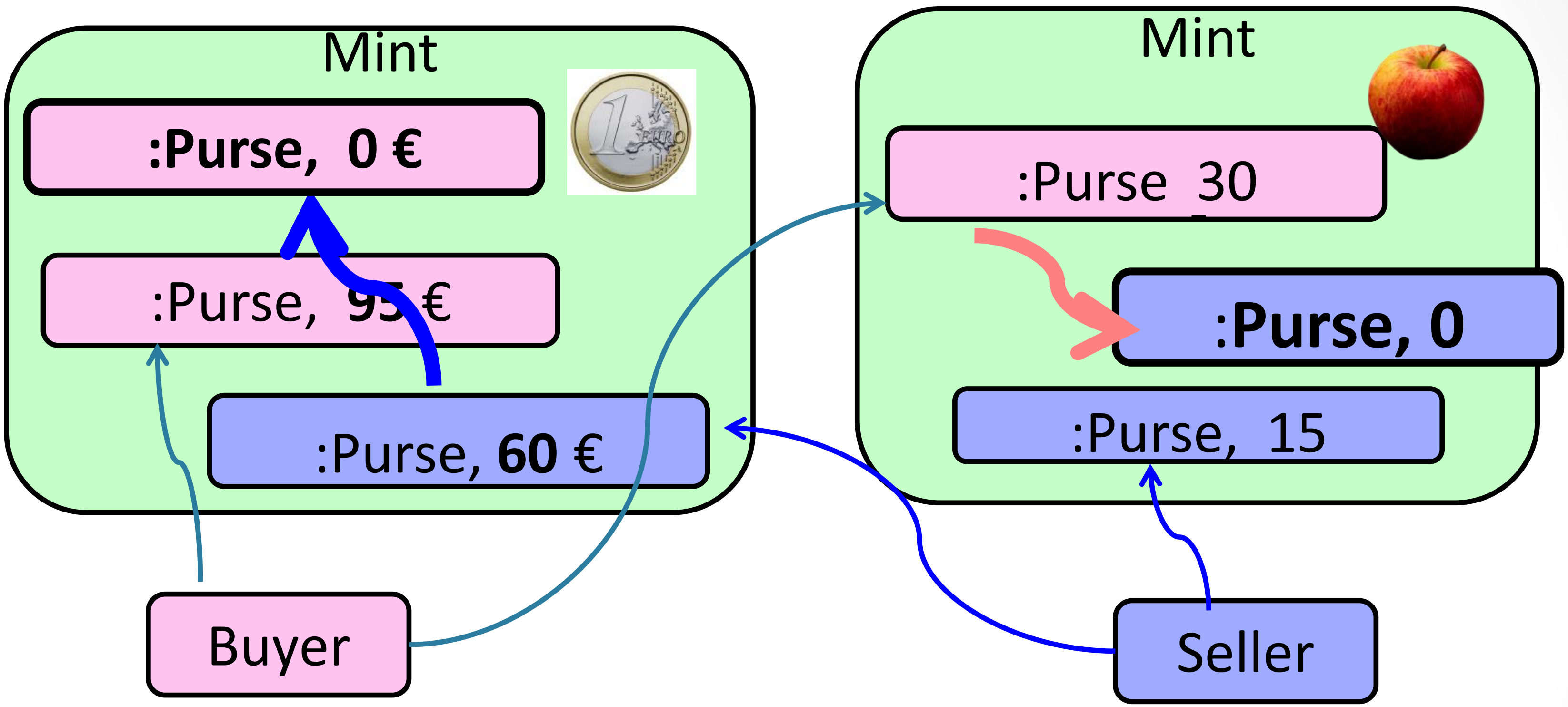

```
buyerEurosTmp = buyerEuros.makePurse();
buyerEurosTmp.transfer(5,buyerEuros);
```

Buying Apples – 2nd Attempt: tmp purses



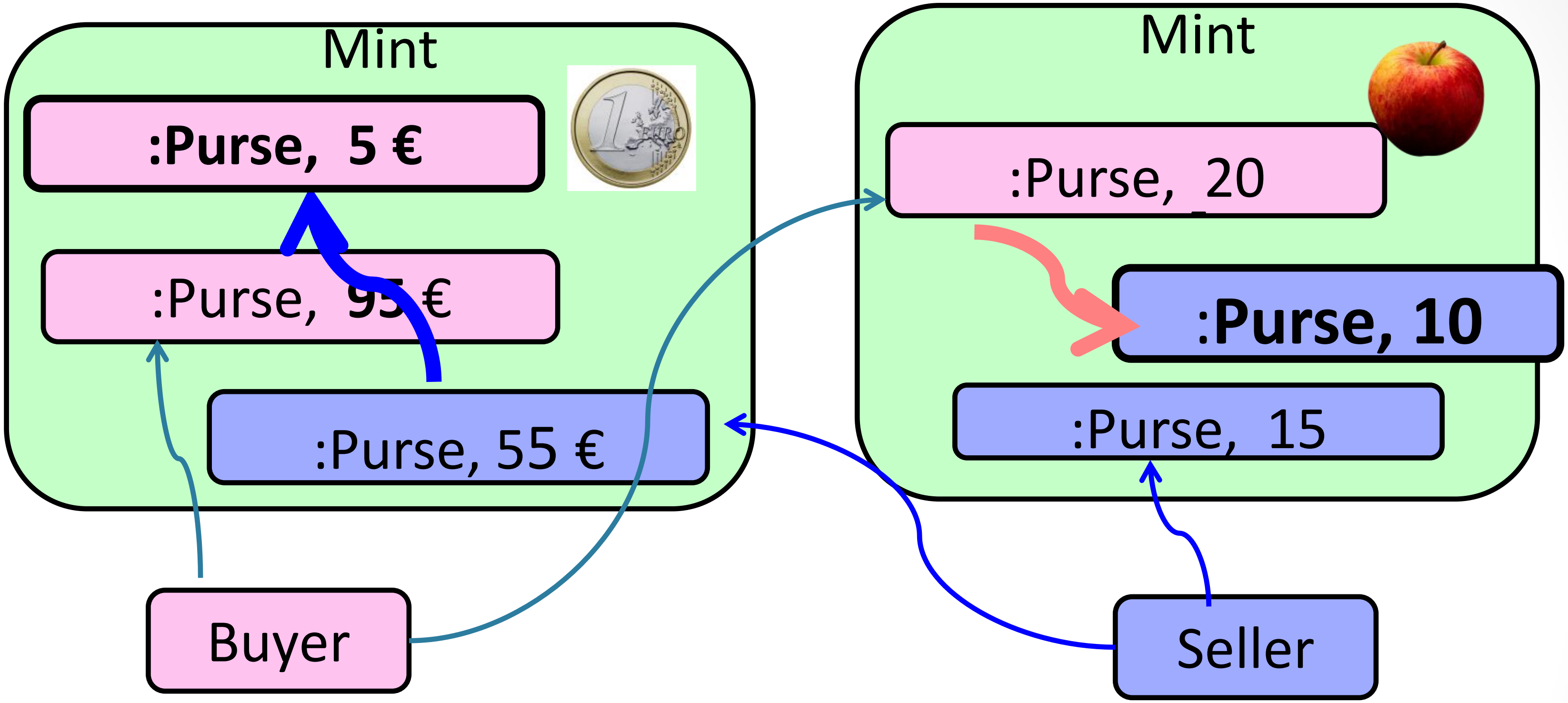

```
buyerEurosTmp = buyerEuros.makePurse();
buyerEurosTmp.transfer(5,buyerEuros);
sellerApplesTmp = sellerApples.makePurse();
sellerApplesTmp.transfer(10,sellerApples);
```

Buying Apples – 2nd Attempt: tmp purses



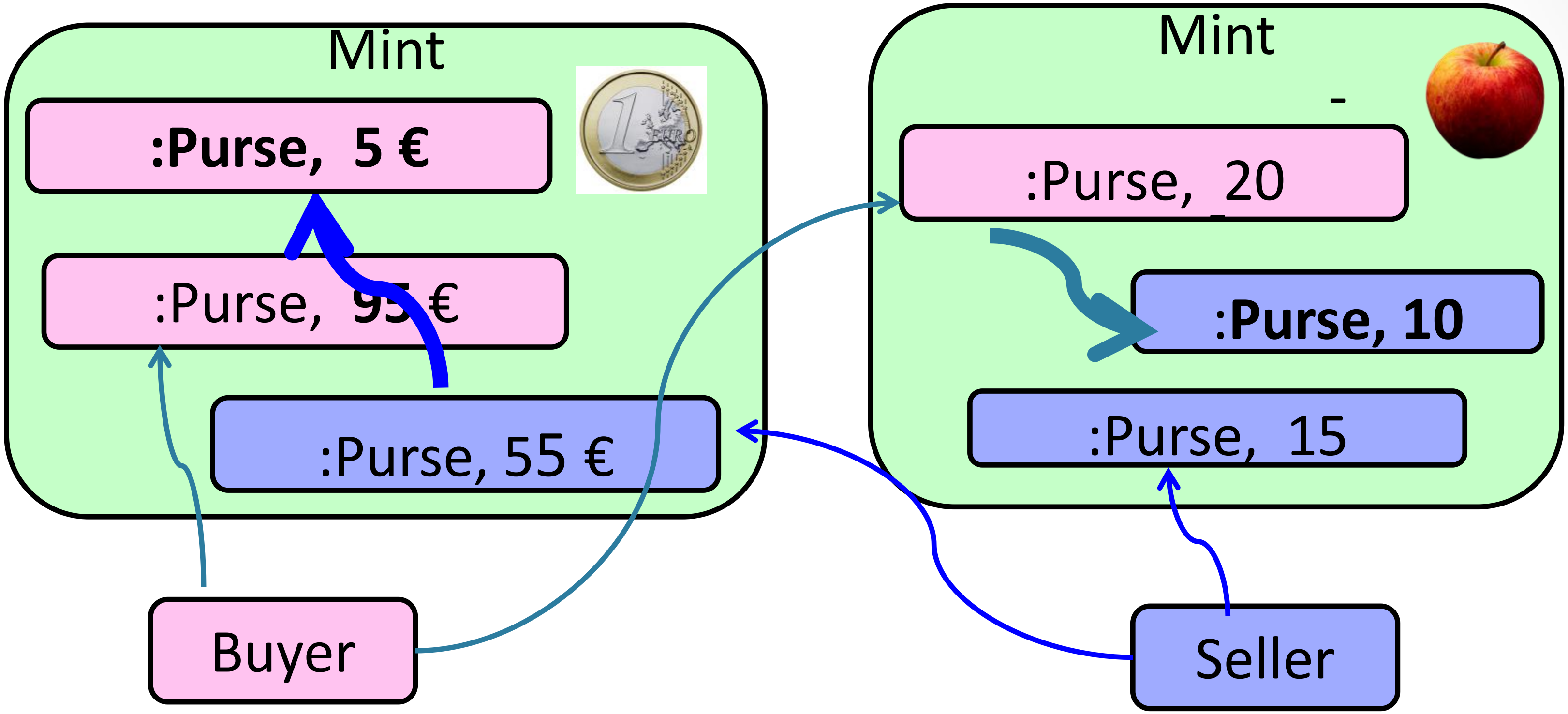

```
buyerEurosTmp = buyerEuros.makePurse();
buyerEurosTmp.transfer(5,buyerEuros);
SellerApplesTmp = sellerApples.makePurse();
sellerApplesTmp.transfer(10,sellerApples);
sellerEuros.transfer(5,buyerEurosTmp);
buyerApples.transfer(10,sellerApplesTmp);
```

Buying Apples – 2nd Attempt: tmp purses




```
buyerEurosTmp = buyerEuros.makePurse();
buyerEurosTmp.transfer(5,buyerEuros);
sellerEurosTmp = sellerApples.makePurse();
sellerEurosTmp.transfer(10,sellerApples);
sellerEuros.transfer(5,buyerEurosTmp);
buyerApples.transfer(10,sellerEurosTmp);
```

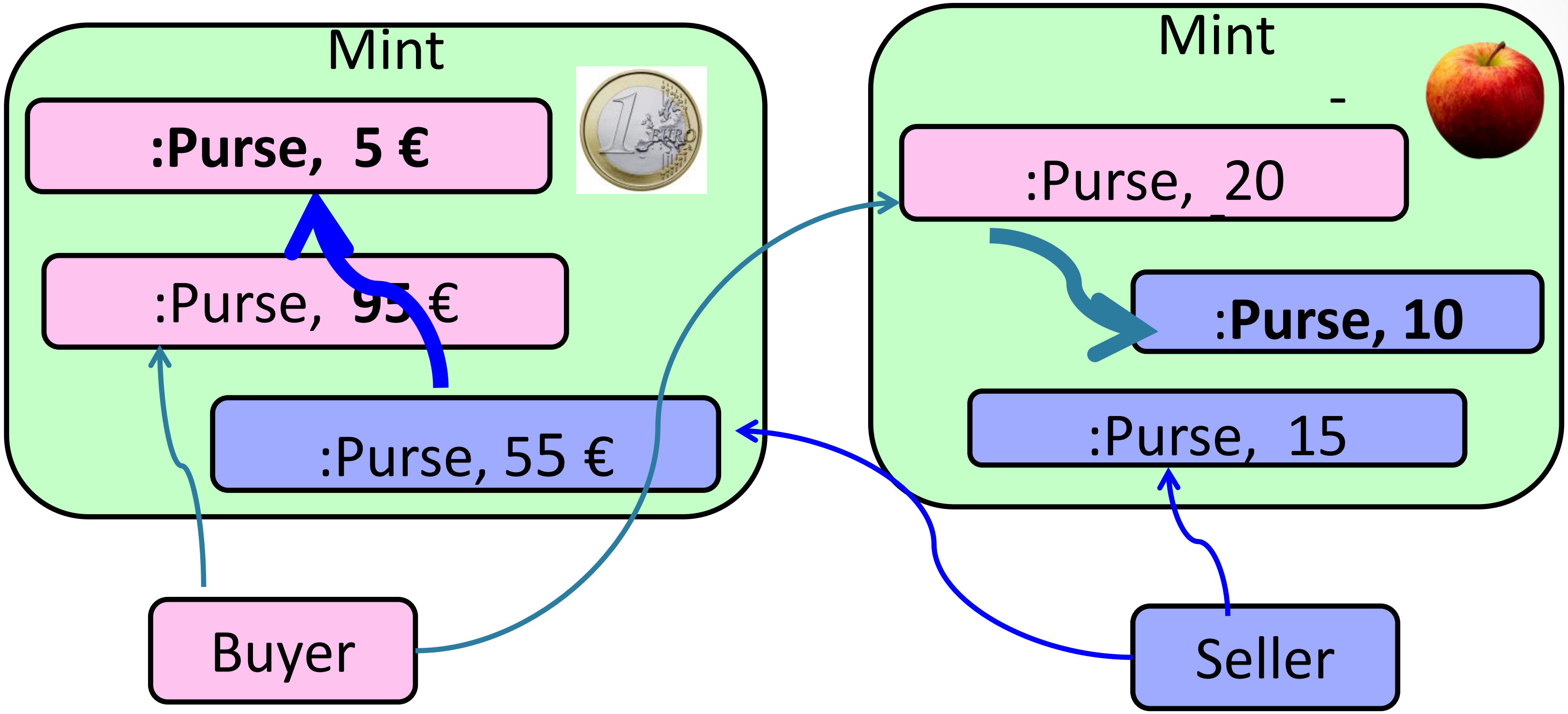
Buying Apples 2nd Attempt: tmp purses - Risk



Risk to Buyer: - 5 €, 0

Risk to Seller: 0€, -10

Buying Apples 2nd Attempt: tmp purses - Risk

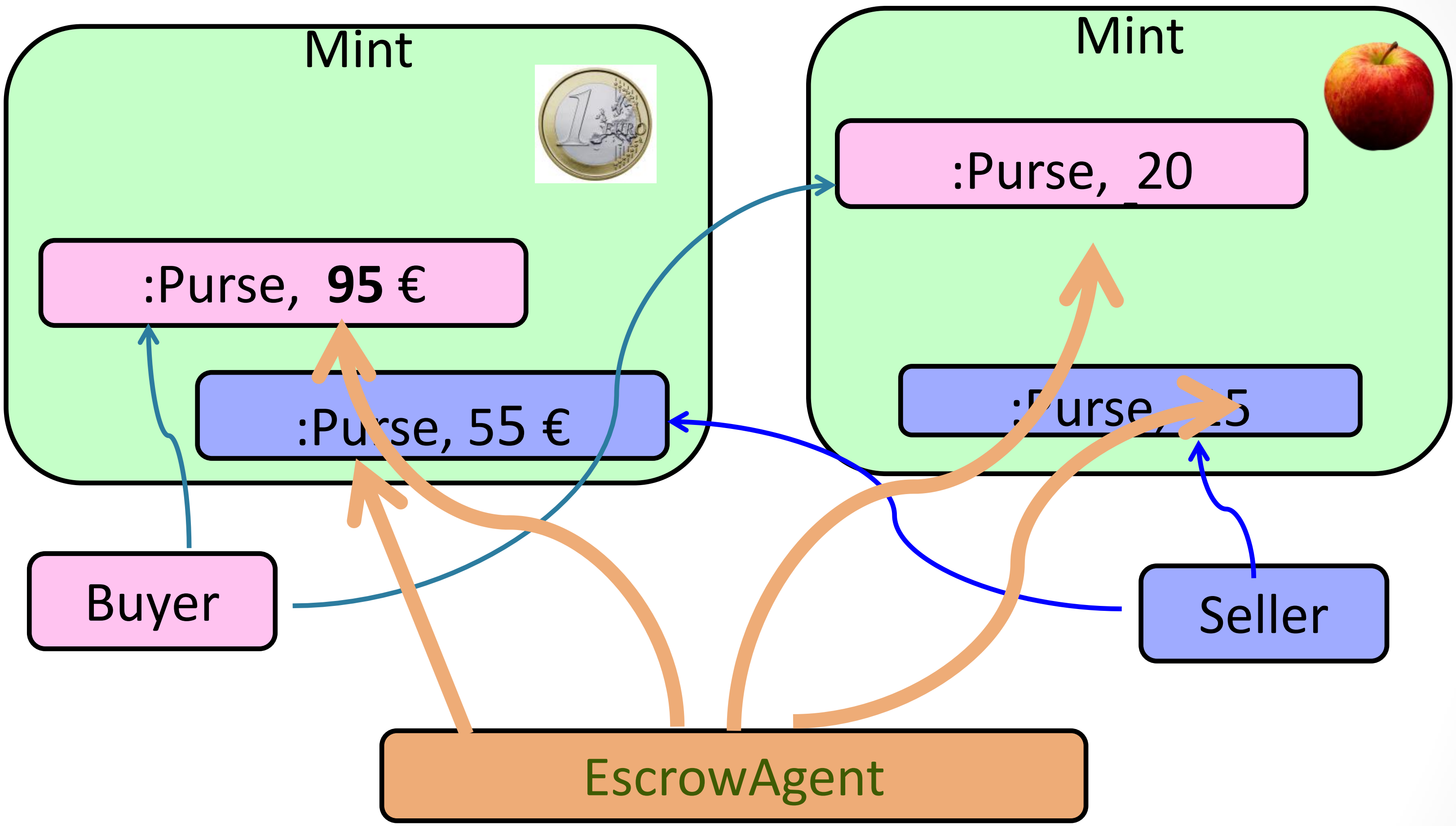


Risk to Buyer: - 5 €, 0

Risk to Seller: 0€, -10

… what if Buyer/Seller use tmp-Purses to steal from other Purses, or Mint itself?

Buying Apples – 3rd Attempt Escrow



The risks of using potentially untrustworthy objects

Challenges – our contributions

- Develop code of Escrow, so as to minimize the risk to which it exposes its clients, cf Miller et al, ESOP 2014
- Specify the Escrow's behaviour when Buyer and Seller are trustworthy, cf Hoare Logics, JML, jStar, C-sharp, etc.
- Write Escrow without Escmascript features
- Develop Specification Language
- Specify the Bank and Mint.
- Specify the Escrow's behaviour when Buyer is trustworthy and Seller in not; and the opposite.
- Develop proof methodology
- Prove that Escrow code indeed satisfies the specification.

Electronic Money, Mints and Purses
-
or,
Banks and Accounts

Mints and Purses

The electronic money as proposed in [MillerEtAl,FinCrypto’00]

- Mints with electronic money,
- Purses held within mints,
- Transfers of funds between purses.
- A purse’s balance “guarded” by the purse.
- The *currency* of a mint is the sum of balances of its purses.
- The currency “guarded" by the mint (no devaluation).

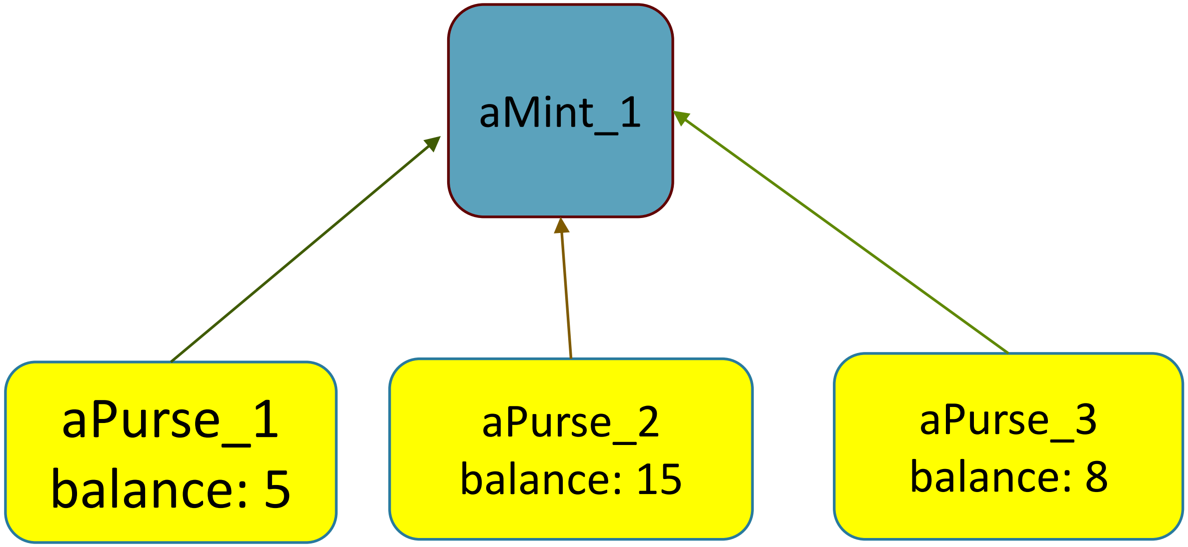
Mint & Purse code – vrs1

```
public final class Mint {       }
public final class Purse {
   private final  mint;
   private long balance;
    public Purse(mint, balance) {
       if (balance<0) { throw …   };
       this.mint = mint; this.balance = balance; }
    public Purse sprout( ) {
       p = new Purse;
       p.mint = this.mint;  p.balance = 0;
       return p; }
    public transfer(prs, amnt) {
        if ( mint!=prs.mint || amnt>prs.balance
                              || amnt+balance<0 )
           { throw …   };
         prs.balance -= amnt;  balance += amnt; }
}
```

The `final`, `private` field annotations are dynamically checked.

Mint & Purse – objects– vrs1

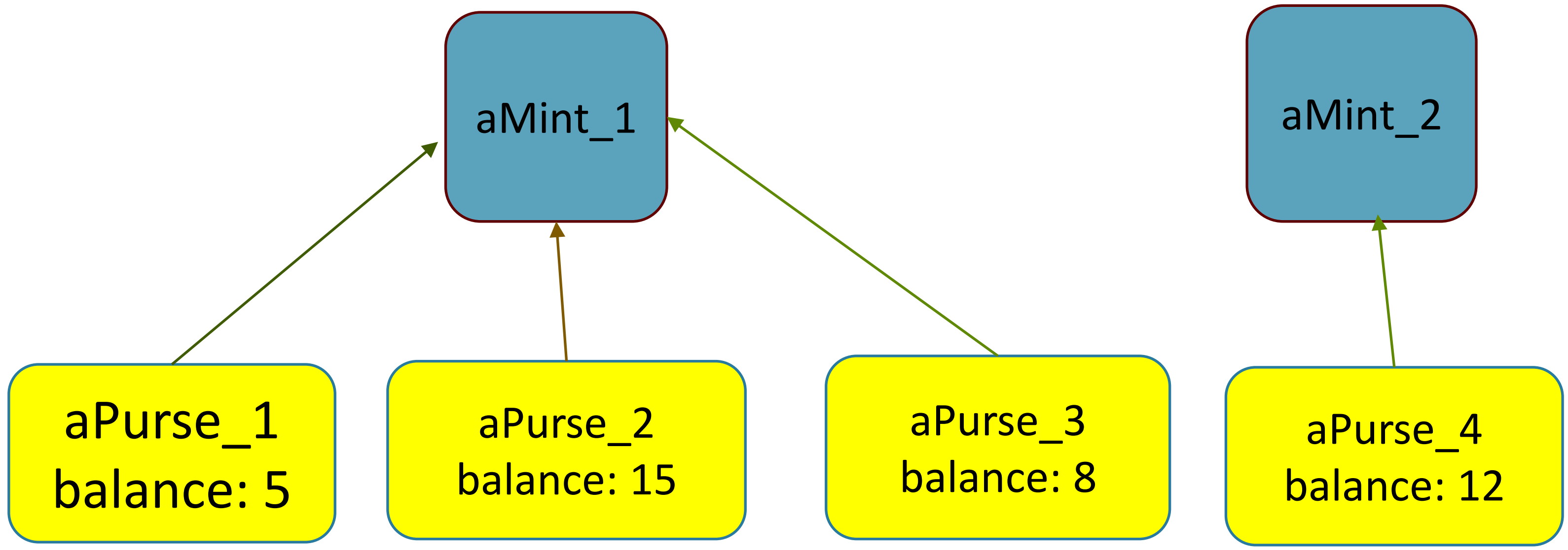
The *currency* of a mint is the sum of balances of its purses



aMint_1.currency = 5 +15 + 8

Mint & Purse – objects– vrs1

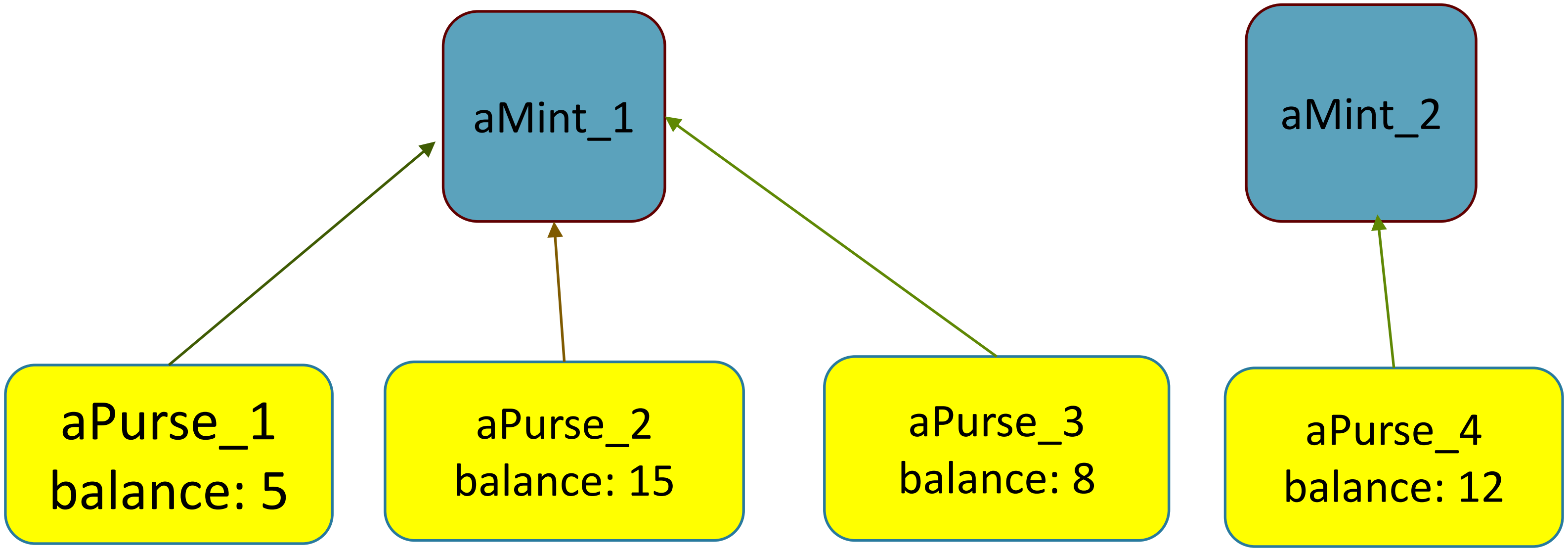
The *currency* of a mint is the sum of balances of its purses



aMint_1.currency = 5 +15 + 8
aMint_2.currency = 12

Mint & Purse – objects– vrs1

The *currency* of a mint is the sum of balances of its purses



aMint_1.currency = 5 +15 + 8
aMint_2.currency = 12

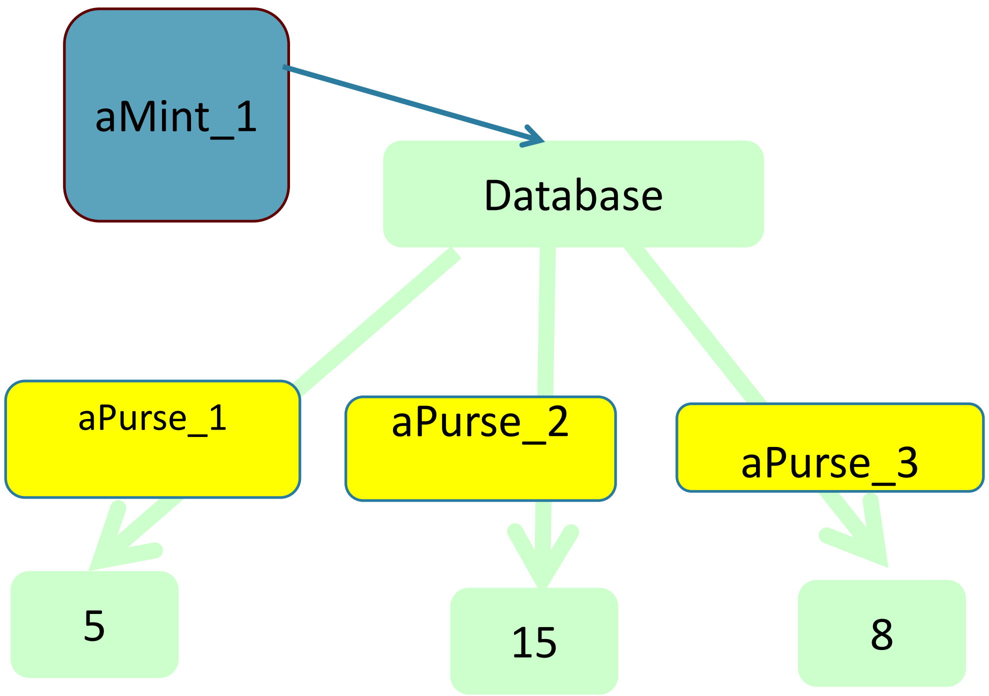
The *currency* of a mint is a *model* field of the mint.

Mint & Purse – Java code – vrs2

```
public final class Purse {   }
public final class Mint {
      private final HashMap<Purse,long> database
                                = new HashMap <>();
      public Purse makePurse(balance) {
            Purse p = new Purse( );
            database.put(p,balance);
            return p; }
      public transfer
                  (from, into, long amnt){
            if((amount<0) || (!database.contains(from))
            || (database.get(from) < amnt)
                  || (!database.contains(into)) )
             { throw new IlleglArgtException(); };
             database.put(from, database.get(from)-amnt);
            database.put(into, database.get(into) +amnt);
      }
}
```

Mint & Purse – objects– vrs2

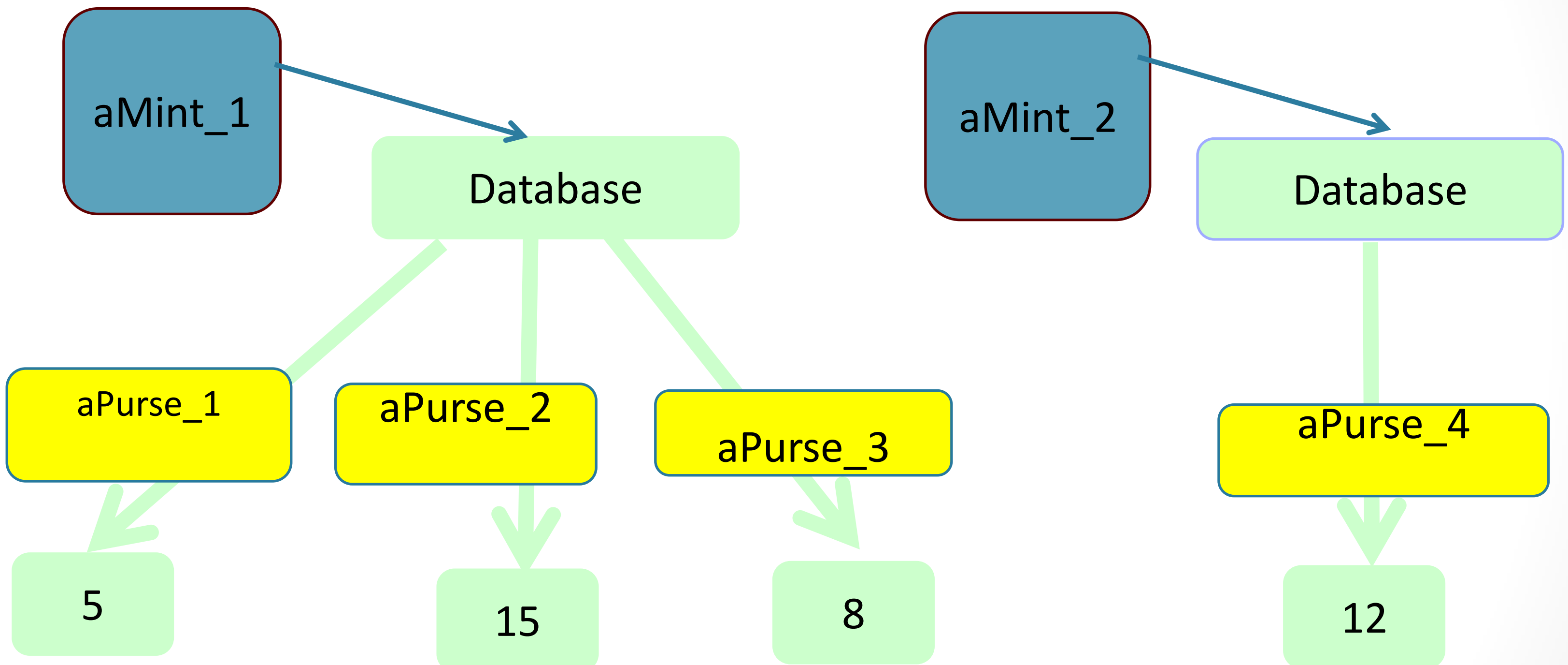
The *currency* of a mint is the sum of balances of its purses



aMint_1.currency = 5 +15 + 8
aMint_2.currency = 12

Mint & Purse – objects– vrs2

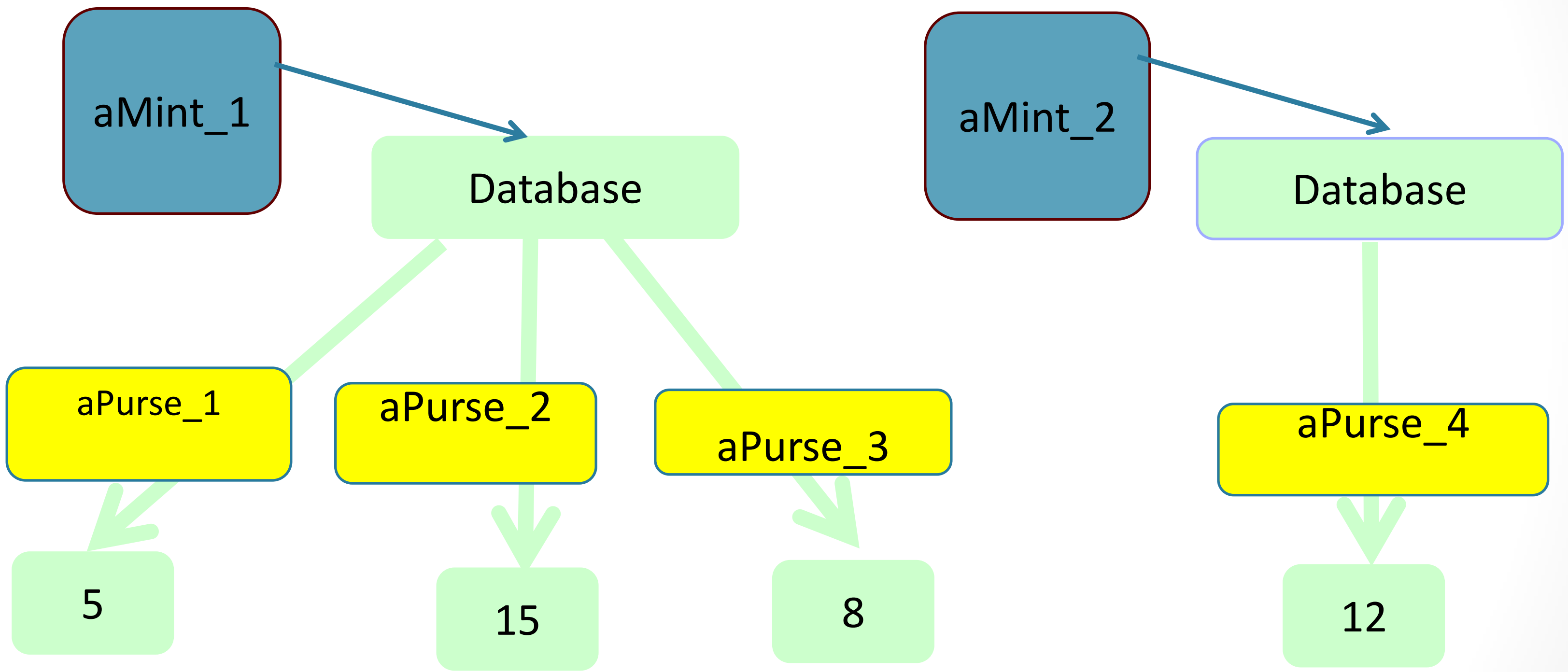
The *currency* of a mint is the sum of balances of its purses



aMint_1.currency = 5 +15 + 8
aMint_2.currency = 12

Mint & Purse – objects– vrs2

The *currency* of a mint is the sum of balances of its purses



aMint_1.currency = 5 +15 + 8

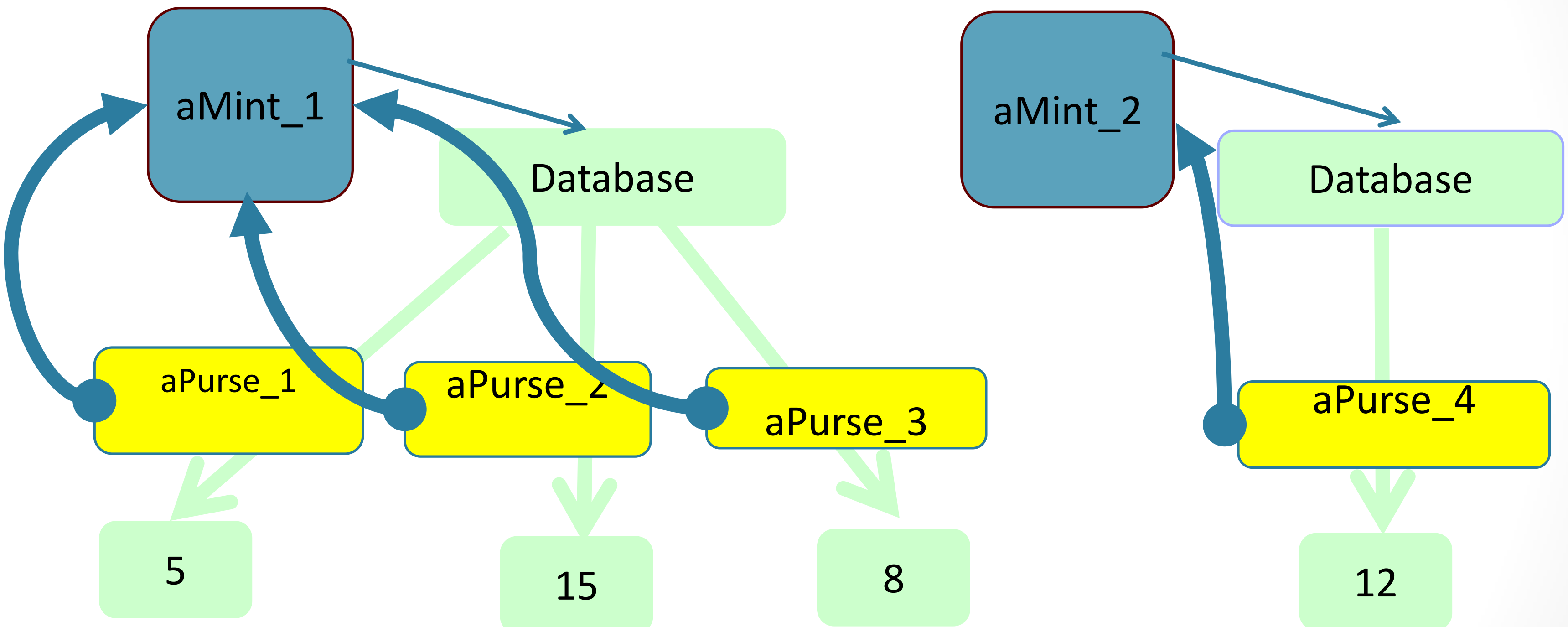
aMint_2.currency = 12

The *currency* of a mint,
and the *balance of a purse* are *model* fields.

Mint & Purse code – vrs3

```
public final class Purse {
     private final Mint mint;
     public deposit(amt, from){mint.transfer(from,this.amt) ; }
     public sprout( ){ return mint.makePurse(0); }
}
public final class Mint {
      private final HashMap<Purse,long> database
                                  = new HashMap <>();
      public makePurse(balance) {
             Purse p = new Purse( );
             database.put(p,balance);
             return p; }
      public transfer(from, into, long amnt){
             if((amount<0) || (!database.contains(from))
             || (database.get(from) < amnt)
                    || (!database.contains(into)) )
              { throw new IlleglArgtException(); };
              database.put(from, database.get(from)-amnt);
             database.put(into, database.get(into) +amnt); }
}
```

Mint & Purse – objects– vrs3



aMint_1.currency = 5 +15 + 8

aMint_2.currency = 12

Capability Policies Mints & Purses

- **Pol_1** With two purses of the same mint, one can transfer money between them.
- **Pol_2** Only someone with the mint of a given currency can violate conservation of that currency.
- **Pol_3** The mint can only inflate its own currency.
- **Pol_4** No one can affect the balance of a purse they don't have.
- **Pol_5** Balances are always non-negative integers.
- ***Pol_6** A reported successful deposit can be trusted as much as one trusts the purse one is depositing into*.

Capability Policies go beyond classical specifications

We claim that capability policies go beyond classical specifications. Because capability policies are:

- *Open* – They apply to a module and all its possible extensions.
- *Pervasive* – they apply across any two consecutive point of execution
- They sometimes talk about *necessary* rather than *sufficient* conditions.
- They *sometimes* talk about trust.

Mint example – “classical” specification

```
public final class Mint {      }
public final class Purse {
   private final Mint mint;
   private long balance;
   INV balance >= 0;

   public Purse(Mint mint, long balance)
   PRE balance >= 0;

   public Purse sprout( ) { … }
   public void transfer(long amnt, Purse prs)
   PRE prs.mint=this.mint &&
    this.balance+amnt>= && prs.balance-amnt >=0;
    POST this.balance_new=this.balance_old+amount &&
         prs.balance_new = this.balance_new-amount
}
```

Classical spec. does *not* imply policies!

allows `mint` to be set externally; thus may affect `currency` of a `mint` without access to it (**breaks** Pol 2)

Code below sa ... ut breaks policies.

allows `balance` to be set externally; thus may transfer money without access to second `Purse`, or may affect `currency` of a `mint` (**breaks** Pol 1, and Pol 2)

```
final class
final class Purse {
   private final Mint mi
   private long balance;
   public Purse(Mint mi
      if (balance<0) {
      this.mint = mint;
   public Purse sprout( ) {
      Purse p = new Purse;
      p.mint = prs.mint;  p.balance = 0;
      return p; }
   void transfer(long amnt, Purse prs) {
      if ( mint!=prs.mint || amnt>prs.balance
         || amnt+balance<0 ) { throw ...   };
      prs.balance -= amnt;  balance += amnt; }
}
```

Classical spec. does *not* imply policies - 2

Code below satisfies classical specification, but breaks policies.
Nor does it prevent:

```
final class Mint {        }
final class Purse {
   final private Mint mint;
   private long balance;
   public Purse(Mint mint, long balance) {
                   …
    Purse(Purse prs) {
                   …
    void transfer(Purse prs, long amnt) {
                   …
    void subvert( )         // BREAKS POLICY_2
          { new Purse(mint, 200000,45); }
}
```

Our Policy Specification Language - 1

- Take a module M
- Take condition Q
- Policies may have the form Q
- We say that module M adheres to a policy Q

$$M \vDash Q$$

iff

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

$$M{*}M',\ \kappa \vDash Q$$

- Arising(M) is the set of all configuration, code pairs which may arise through execution of the initial configuration with M.

Our Policy Specification Language - 1

Open policy,
increases number of
configurations considered

$$M \vDash Q$$

iff

$$\forall M'.\ \forall\ (\kappa,\ \_) \in \mathit{Arising}(M * M').$$
$$M * M',\ \kappa \vDash Q$$

Our Policy Specification Language - 1

Open policy,
increases number of
configurations conisdered

$$M \vDash Q$$

iff

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

$$M*M',\ \kappa \vDash Q$$

Only reachable configurations,
i.e. *decrease* number of
configurations considered.

Our Policy Specification Language -2

- Take a module M and some code code
- Take conditions Q and R
- Policies have the form Q **or** **{ Q } code { R }**
- We define adherence to a policy

$$M \models \{\ Q\ \}\ \text{code}\ \{\ R\ \}$$

$$\text{iff}$$

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

$$M{*}M',\ \kappa \models Q \quad \wedge \quad M{*}M',\ \text{code},\ \kappa \leadsto \kappa',\ v.$$

$$\Rightarrow$$

$$M{*}M',\ \kappa' \models R.$$

- $M{*}M',\ \text{code},\ \kappa \leadsto \kappa',\ v$ is the large steps semantics.

Our Policy Specification Language -2

Open policy

$$M \models \{\ Q\ \}\ \text{code}\ \{\ R\ \}$$

iff

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

$$M * M',\ \kappa \models Q \quad \wedge \quad M * M',\ \text{code},\ \kappa \leadsto \kappa',\ v.$$

$$\Rightarrow$$

$$M * M',\ \kappa' \models R.$$

Only reachable configurations, considered.

Pol_1: With two purses of the same mint, one can transfer money between them.

Pol_1 $\equiv$

{ p1 is PurseSpec,
p2 is PurseSpec,
p1.balance >= amt,
p1.mint = p2.mint }

p1.transfer(amt, p2)

{ p1.balance = p1.balance$_{old}$ – amt,
p2.balance = p1.balance$_{old}$ + amt,
"nothing else changed" }

Pol_1: With two purses of the same mint, one can transfer money between them.

Pol_1 $\equiv$

{ p1 is PurseSpec,
p2 is PurseSpec,
p1.balance >= amt,
p1.mint = p2.mint }

p1.transfer(amt, p2)

{ p1.balance = p1.balance$_{old}$ – amt,
p2.balance = p1.balance$_{old}$ + amt,
“nothing else changed” }

Note the use of model fields in the spec.

Pol_2: Only someone with the mint of a given currency can violate conservation of that currency.

Pol_2 $\equiv$

$\forall m : \text{MintSpec}.\ \forall o : \text{Object}.$

$\text{MayAffect}(o, m.\text{currency})$

$\Rightarrow$

$\text{MayAccess}(o, m)$

This is an execution invariant.

Again, we are using model fields.

Note predicates MayAccess, and MayAffect.

The meaning of MayAffect

- Take a runtime configuration κ, module M, a variable x, and a pure expression e

$$M, \kappa \vDash \mathrm{MayAffect}(x,e)$$

iff

$$\exists\, m. \quad M,\ x.m(\ldots),\ \kappa \rightsquigarrow \kappa',\ \_ \quad \wedge \quad \lceil e \rfloor_{\kappa} \neq \lceil e_{pure} \rfloor_{\kappa'}$$

In the above, we are using notation as follows

- Large step semantics $M,\ expr,\ \kappa \rightsquigarrow \kappa',\ v$
- The value of a pure expression e_{pure} in context of κ is $\lceil e_{pure} \rfloor_{\kappa}$

The meaning of MayAccess

- For a configuration κ, and variables x and y

$$M, \kappa \vDash MayAccess(x,y)$$

iff

$$\exists\, f1,\ldots fn.\quad \lceil x.f1.f2.\ \ldots\ fn \rfloor_{\kappa} = \lceil y \rfloor_{\kappa}$$

- For a configuration κ, and variable x

$$MayAccess(x)_{\kappa} = \{\ o \mid \exists\, f1,\ldots fn.\quad \lceil x.f1.f2.\ \ldots\ fn \rfloor_{\kappa} = o\ \}$$

Pol_3: The mint can only inflate its own currency.

Pol_3 $\equiv$

$\forall m$: MintSpec.

{ true }

any

{ m.currency >= $m.currency_{old}$ }

Pol_4: No one can affect the balance of a purse they do not have.

Pol_4 $\equiv$

$\forall p : PurseSpec.\ \forall o : Object.$

$MayAffect(o, p.balance)$

$\Rightarrow$

$MayAccess(o, p)$

Pol_4: No one can affect the balance of a purse they do not have.

$$\text{Pol\_4} \equiv$$

$$\forall p : \text{PurseSpec}.\ \forall o : \text{Object}.$$

$$\text{MayAffect}(o, p.\text{balance})$$

$$\Rightarrow$$

$$\text{MayAccess}(o, p)$$

Note use of model fields, and predicates MayAccess, and MayAffect.

The meaning of PublicAccess

- For a configuration κ, and variables x and y

$$M, \kappa \models PublicAccess(x,y)$$

iff

$$\lceil x \rfloor_\kappa = \lceil y \rfloor_\kappa$$

$$\lor \quad \exists \text{ field } f. \quad \lceil x.f \rfloor_\kappa = \lceil y \rfloor_\kappa$$

$$\exists \textit{ public} \text{ methods } m_1, \ldots m_n. \quad \lceil x.m_1(\ldots).m_2(\ldots). \ldots m_n(\ldots) \rfloor_\kappa = \lceil y \rfloor_\kappa$$

Pol_4, revisited: No one can affect the balance of a purse th do not have.

We had defined

$$\text{Pol\_4} \equiv \forall p : \text{PurseSpec}.\ \forall o : \text{Object}.\ \text{MayAffect}(o, p.\text{balance}) \Rightarrow \text{MayAccess}(o, p)$$

Pol_4 is perhaps too weak. What about?

$$\text{Pol\_4.a} \equiv \forall p : \text{PurseSpec}.\ \forall o : \text{Object}.\ \text{MayAffect}(o, p.\text{balance}) \Rightarrow \text{PublicAccess}(o, p)$$

Too strong. Cannot be satisfied.

Pol_4, re-revisited: No one can affect the balance of a purse they do not have.

Pol_4.b $\equiv$ $\forall p : PurseSpec.\ \forall o : Object.$
$p \in ThisModule,\ o \notin ThisModule.$
$MayAffect(o,\ p.balance)$
$\Rightarrow$
$\exists o' \notin ThisModule.$
$PublicAccess(o',\ p)$

ThisModule stands for the module which is expected to satisfy the policy.

$o \in M$ says that o "belongs" to M, ie that the class of has been defined in module M,
or that o is owned (or was created) by object $o' \in M$.

Our Policy Specification Language - 3

Open **policy,**
increases number of
configurations considered

$$M \vDash Q$$

iff

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

$$M{*}M',\ \kappa \vDash Q[\mathrm{ThisModule}/M]$$

Our Policy Specification Language - 3

We now give meaning to variable `ThisModule`

$$\texttt{M} \vDash Q$$

iff

$$\forall \texttt{M}'.\ \forall\ (\kappa,\ \_) \in \mathit{Arising}(\texttt{M}*\texttt{M}').$$

$$\texttt{M}*\texttt{M}',\ \kappa \vDash Q[\texttt{ThisModule}/\texttt{M}]$$

Pol_5: Balances are always non-negative.

Pol_5 $\equiv$

$$\forall p : PurseSpec.\ \ p.balance >= 0$$

•

Pol_6: *A reported successful deposit can be trusted as much as one trusts the purse one is depositing into*.

We introduce the notation

`p` **is** `PurseSpec`

To express that `p` adheres to specification `PurseSpec`.

Pol_6: *A reported successful deposit can be trusted as much as one trusts the purse one is depositing into.*

Pol_6,a ≣

$\{$ true $\}$

```
res = p.transfer(amt,p')
```

$\{$ `res` $\land$ `p` is `PurseSpec`

$\Rightarrow$

`p'` is `PurseSpec`

$\land$ `p.mint` == `p'.mint`

$\land$ $\texttt{p'.balance}_{old}$ >= `amt`

$\land$ `p'.balance` = $\texttt{p'.balance}_{old}$ − `amt`

$\land$ `p.balance` = $\texttt{p.balance}_{old}$ + `amt` $\}$

Pol_6: *A reported successful deposit can be* ***trusted*** *as much as one* ***trusts*** *the purse one is depositing into.*

Pol_6,a ≣

```
{  true  }
      res = p.transfer(amt,p')
{  res ∧ p is PurseSpec
       ⇒
   p' is PurseSpec
   ∧ p.mint == p'.mint
   ∧ p'.balance_old >= amt
   ∧ p'.balance  = p'.balance_old - amt
   ∧ p.balance   = p.balance_old + amt   }
```

Pol_6: *A reported successful deposit can be trusted as much as one trusts the purse one is depositing into.*

Pol_6,b ≣

```
{ true }
    res = p.sprout()
{ res ∧ p is PurseSpec
      ⇒
  res is PurseSpec
  ∧ p.mint == res.mint
  ∧ res.balance = 0
  ∧ "all else is unmodified" }
```

Capability Policies characteristics

- *Open*
- *Pervasive*
- *Hypothetical* actions (MayAffect)
- *Necessary* rather than *sufficient* conditions (MayAffect requires MayAccess)
- *Establishing trust*
- *Provenance* of effects *(who* caused the balance change*)*

We do not claim that the proposed specifications are the final word for the precise meanings for these policies.

But we have proposed a language with which to *explore* the meanings of the mint policies.

And we used the Mint policies to prove the Escrow policies.

Dynamic Types and Trust

Trust? Back to the Escrow

- Seller wants to sell amt apples, for price Euros.
- Buyer wants to buy amt apples, for price Euros.
- Buyer trusts his Purses, but does not trust Seller's purses.
- Seller trusts his Purses, but does not trust Buyer's purses.
- Buyer and the Seller trust the Escrow.
- Escrow does not trust either Seller or Buyer.

Trust?
The Escrow Example

- The Escrow needs to cater for the following:
    - Can the Seller’s purses be trusted?
    - Can the Buyer’s purses be trusted?
    - Might the Seller withdraw goods during the transaction?
    - Might the Buyer withdraw money during transaction?
    - Could a malicious Seller harm the Buyer?
    - Could a malicious Buyer harm the Seller?

The Escrow – 1st case

```
public bool deal(
     buyerEuros, buyerApples,   // buyer’s Purses
     sellerEuros, sellerApples, // seller’s Purses
     amount                     // amount apples
     price                      // Euro-price of goods
   )
   // transfer amnt and price,
   // provided that
   //    buyerEuros, sellerEuros are PurseSpec’s
   //    buyerApples, sellerApples are PurseSpec’s
   //    buyerEuros and sellerEuros from same mint
   //    buyerApples, and sellerAuros from same mint
   //    buyerEuros has more than price euros
   //    sellerApples has more than amount apples
```

The Escrow specification - 1st case

```
public bool deal(
    buyerEuros, buyerApples sellerEuros, sellerApples,
    amount, price )
POST:
[  res=true  ∧
 ( buyerEuros, buyerApples,
   sellerEuros, sellerApples is PurseSpec )
```

The Escrow specification - 1st case

```
public bool deal(
    buyerEuros, buyerApples sellerEuros, sellerApples,
    amount, price )
POST:
[  res=true  ∧
 ( buyerEuros, buyerApples,
   sellerEuros, sellerApples is PurseSpec )
        ∧
   buyerEuros.mint == sellerEuros.mint  ∧
   buyerApples.mint == sellerAuros.mint ∧
```

The Escrow specification - 1st case

```
public bool deal(
    buyerEuros, buyerApples sellerEuros, sellerApples,
    amount, price )
POST:
[  res=true  ∧
 ( buyerEuros, buyerApples,
   sellerEuros, sellerApples is PurseSpec )
         ∧
   buyerEuros.mint == sellerEuros.mint  ∧
   buyerApples.mint == sellerAuros.mint ∧
   buyerEuros.balance_pre >= price ∧
   sellerApples.balance_pre >= amnt ∧
```

The Escrow specification - 1st case

```
public bool deal(
    buyerEuros, buyerApples sellerEuros, sellerApples,
    amount, price )
POST:
[  res=true  ∧
 ( buyerEuros, buyerApples,
   sellerEuros, sellerApples is PurseSpec )
        ∧
   buyerEuros.mint == sellerEuros.mint  ∧
   buyerApples.mint == sellerAuros.mint ∧
   buyerEuros.balance_pre >= price ∧
   sellerApples.balance_pre >= amnt ∧
   buyerEuros.balance == buyerEuros.balance_pre - price ∧
   sellerEuros.balance == sellerEuros.balance_pre + price ∧
   buyerApples.balance == buyerApples.balance_pre + amt ∧
   buyerApples.balance == buyerApples.balance_pre - amt ∧
```

The Escrow specification - 1st case

```
public bool deal(
     buyerEuros, buyerApples sellerEuros, sellerApples,
     amount, price )
POST:
[  res=true  ∧
  ( buyerEuros, buyerApples,
    sellerEuros, sellerApples is PurseSpec )
          ∧
    buyerEuros.mint == sellerEuros.mint  ∧
    buyerApples.mint == sellerAuros.mint ∧
    buyerEuros.balance_pre >= price ∧
    sellerApples.balance_pre >= amnt ∧
    buyerEuros.balance == buyerEuros.balance_pre – price ∧
    sellerEuros.balance == sellerEuros.balance_pre + price ∧
    buyerApples.balance == buyerApples.balance_pre + amt ∧
    buyerApples.balance == buyerApples.balance_pre – amt ∧
    ∀p:_pre PurseSpec. p.balance == p.balance_pre  ]
```

The Escrow specification - 1st case

```
public bool deal(
    buyerEuros, buyerApples sellerEuros, sellerApples,
    amount, price )
POST:
[  res=true  ∧
 ( buyerEuros, buyerApples,
   sellerEuros, sellerApples is PurseSpec )
         ∧
   buyerEuros.mint == sellerEuros.mint  ∧
   buyerApples.mint == sellerAuros.mint ∧
   buyerEuros.balance_pre >= price ∧
   sellerApples.balance_pre >= amnt ∧
   buyerEuros.balance == buyerEuros.balance_pre – price ∧
   sellerEuros.balance == sellerEuros.balance_pre + price ∧
   buyerApples.balance == buyerApples.balance_pre + amt ∧
   buyerApples.balance == buyerApples.balance_pre – amt ∧
   ∀p:_pre PurseSpec. p.balance == p.balance_pre  ]

        V

   … 2nd case …
```

The Escrow – 2nd case

```
public bool deal(
     buyerEuros, buyerApples,   // buyer's Purses
     sellerEuros, sellerApples, // seller's Purses
     amount                     // amount apples
     price                      // Euro-price of goods
   )
   // leave everything unaffected,
   // if
   //    buyerEuros, sellerEuros are PurseSpec's
   //    buyerApples, sellerApples are PurseSpec's
   //    buyerEuros and sellerEuros from same mint
   //    buyerApples, and sellerApples from same mint
   // but
   //    buyerEuros not got enough euros, or
   //    sellerApples has not got enough apples
```

The Escrow specification - 2nd case

```
public bool deal(
    buyerEuros, buyerApples, sellerEuros, sellerApples,
    amount, price )
POST:
… 1st case …
      ∨
[  res=false  ∧
 ( buyerEuros, buyerApples,
   sellerEuros, sellerApples is PurseSpec )
        ∧
   buyerEuros.mint == sellerEuros.mint  ∧
   buyerApples.mint == sellerAuros.mint ∧
   ( buyerEuros.balance_pre < price ∨
     sellerApples.balance_pre < amnt ) ∧
   ∀p:_pre PurseSpec. p.balance == p.balance_pre  ]
       ∨
  … 3rd case …
```

The Escrow – 3rd case

```
public bool deal(
     buyerEuros, buyerApples,   // buyer's Purses
     sellerEuros, sellerApples, // seller's Purses
     amount                     // amount apples
     price                      // Euro-price of goods
   )
   // leave everything unaffected,
   // if
   //    buyerEuros is PurseSpec
   //    NOT( sellerApples is PurseSpec)
   //      or buyerEuros and sellerEuros not same mint
```

The Escrow specification - 3rd case

```
public bool deal(
    buyerEuros, buyerApples, sellerEuros, sellerApples,
    amount, price )
POST:
… 1st case …   ∨ … 2nd  case …
         ∨
[  res=false ∧
   buyerEuros is PurseSpec ∧
   (  ¬ (sellerEuros is PurseSpec )∨
      sellerEuros.mint ≠ buyerEuros.mint )
```

The Escrow specification - 3rd case, vrs 1

```
public bool deal(
    buyerEuros, buyerApples, sellerEuros, sellerApples,
    amount, price )
POST:
… 1st case …   ∨ … 2nd  case …
         ∨
[  res=false  ∧
   buyerEuros is PurseSpec ∧
   (  ¬ (sellerEuros is PurseSpec )∨
      sellerEuros.mint ≠ buyerEuros.mint )
    ∧
   ∀p:pre PurseSpec. p.balance == p.balancepre  ]
```

Too strong!

Namely, what if seller had access to `prs`, a `PurseSpec` object with `prs.mint=buyerMoney.mint`, and calls

`prs.transfer(300000,buyerMoneyTmp);`

The Escrow specification - 3^{rd} case, vrs 2

```
public bool deal(
    buyerEuros, buyerApples, sellerEuros, sellerApples,
    amount, price )
POST:
… 1st case …   V … 2nd  case …
          V
[  res=false  ∧
   buyerEuros is PurseSpec ∧
   (  ¬ (sellerEuros is PurseSpec )V
      sellerEuros.mint ≠ buyerEuros.mint )
    ∧
   ∀p:pre PurseSpec. p.balance == p.balance_pre
         V  MayAffect(sellerEuros,p)_pre ]
```

The Escrow specification - 3rd case, vrs 2

```
public bool deal(
     buyerEuros, buyerApples, sellerEuros, sellerApples,
     amount, price )
POST:
… 1st case …    V … 2nd  case …
           V
[  res=false  ∧
   buyerEuros is PurseSpec ∧
   (  ¬ (sellerEuros is PurseSpec )V
      sellerEuros.mint ≠ buyerEuros.mint )
    ∧
   ∀p:pre PurseSpec. p.balance == p.balance_pre
         V  MayAffect(sellerEuros,p)_pre ]
```

Too strong!

Namely, what if seller had access to `g`, an object of class `Gullible`, with `g.prs=buyerEuros`, and calls `g.duped(buyerMoneyTmp)`.

```
class Gullible {
     … prs …
     method duped(p'){  p'.transfer(300000,prs); }
}
```

The Escrow specification - 3rd case, vrs 3

```
public bool deal(
    buyerEuros, buyerApples, sellerEuros, sellerApples,
    amount, price )
POST:
… 1st case …    V … 2nd case …
          V
[  res=false ∧
   buyerEuros is PurseSpec ∧
   (  ¬ (sellerEuros is PurseSpec )V
      sellerEuros.mint ≠ buyerEuros.mint )
  ∧
  ∀p:PRE PurseSpec. p.balance == p.balance_pre
        V  PubAccess(sellerEuros,p)_pre ]
```

This specification is satisfied by `Escrow.deal`, :-), provided that `PurseSpec` also satisfies

Pol_7 ≡ ∀p,p' : PurseSpec. p≠p' ⇒ ¬ MayAccess(p, p')

Pol_7 is satisfied by class `Purse`, vrs1 and vrs 2 :-),

But, is too low level! :-(, and is not satisfied by class `Purse`, vrs 3 :-(.

The Escrow specification - 3rd case, vrs 4

```
public bool deal(
    buyerEuros, buyerApples, sellerEuros, sellerApples,
    amount, price )
POST:
… 1st case …    V … 2nd  case …
          V
[  res=false  ∧
   buyerEuros is PurseSpec ∧
   (  ¬ (sellerEuros is PurseSpec )V
      sellerEuros.mint ≠ buyerEuros.mint )
    ∧
   ∀p:PRE PurseSpec. p.balance == p.balance_pre
         V  ( ∃∉ Module(sellerEuros)∪ Purse*Bank∧
                PubAccess(o,p)_pre  ) ]
```

This specification is satisfied by `Escrow.deal`, :-),
provided that `PurseSpec` also includes Pol_4.b.

Remember, Pol_4.b ≡ ∀p : PurseSpec. ∀o : Object. p∈ ThisModule, o ∉ThisModule.
MayAffect(o, p.balance ⇒∃o' ∉ThisModule. PublicAccess(o', p)

Pol_4.b is satisfied by class `Purse`, vrs1 and vrs 2 **and** vrs 3 : -).

The Escrow specification - 4^{th}, 5^{th}, … cases

are similar

Calculating Trust and Risk

The Access Propagation Rules

Method calls may increase the MayAccess – ibility.

Or, in Mark Miller's terms: Connectivity begets Connectivity.

Access Propagation - Rule 1

$\{\quad \text{true} \quad\}$

```
        x.m(y)
```

$$\{\quad \forall z,z' :_{pre} \texttt{Object}.\ \ \texttt{MayAccess}(z,z')_{pre} \Rightarrow \texttt{MayAccess}(z,z')_{pre} \lor \texttt{MayAccess}(x,z')_{pre} \lor \texttt{MayAccess}(y,z')_{pre} \quad\}$$

These restrictions on `MayAccess` not only apply for the snapshot after execution of `x.m(y)`, but also for any snapshot reached during execution of x.m(y), including within nested method calls.

This is *not* expressed by the Hoare triple above.

We need to find a way of expressing this.

Access Propagation - Rule 2

$$\{\ \text{true}\ \}$$

`x.m(y)`

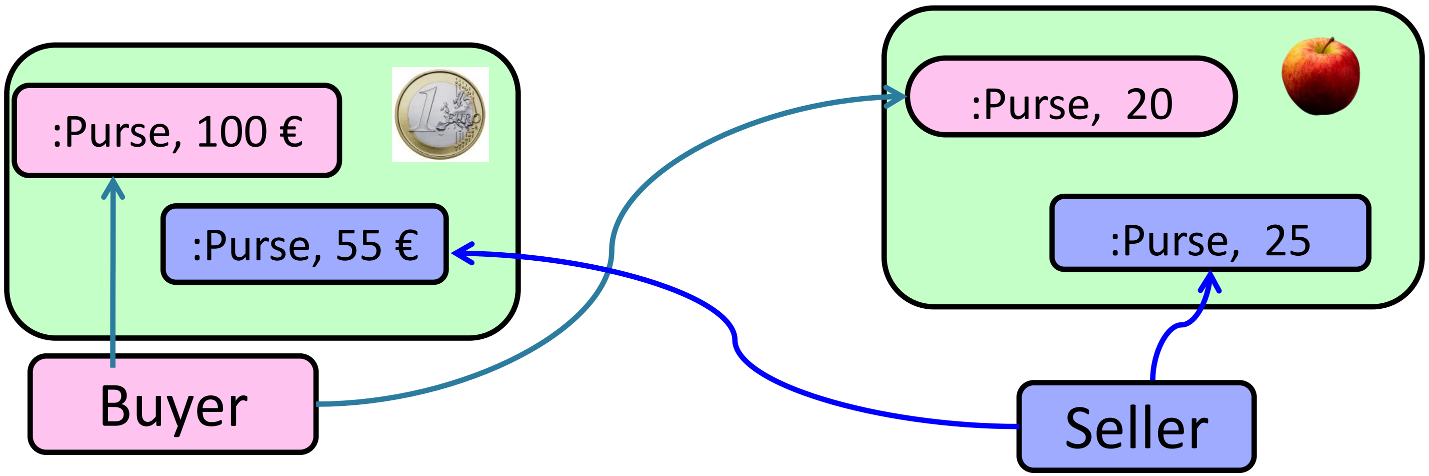
$$\{ \forall z :_{pre} \text{Object}.\ \mathtt{MayAccess}(z)_{pre} \cap (\mathtt{MayAccess}(x)_{pre} \cup \mathtt{MayAccess}(y)_{pre}) = \emptyset \Rightarrow \mathtt{MayAccess}(z) = \mathtt{MayAccess}(z)_{pre} \}$$

As for Access Propagation Rule 1, the restrictions on `MayAccess` apply for any snapshot reached during execution of x.m(y), including within nested method calls.

Reasoning about Escrow code's adherence to policy

We will outline how to demonstrate that Escrow.deal adheres to its specification version 3.

`Escrow.deal` – outline

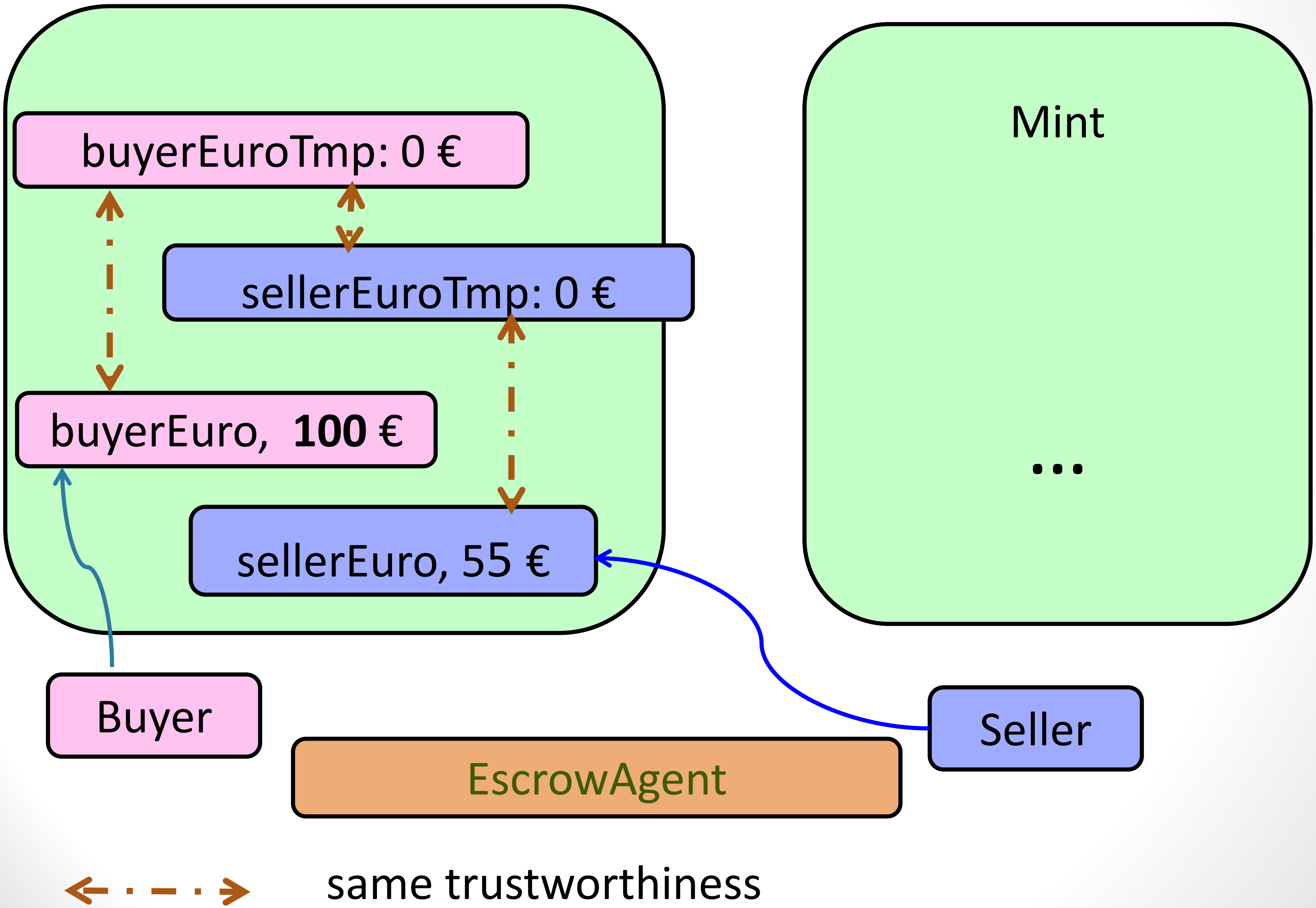


The method `Escrow.deal`

1. Establishes that the Buyer's and the Selle'r Euro Purses have the same trustworthiness. Aborts, if unsuccessful.
2. Establishes that the Buyer's and the Selle'r Apple Purses have the same trustworthiness. Aborts, if unsuccessful.
3. Transfers `price` from Seller into a temporary Purse. Aborts, if unsuccessful.
4. Takes `amount` from Buyer and puts into temporary Purse. If unsuccessful, reimburses Seller with `price`, and aborts. If successful, then transfers `price` from temporary Purse to Seller, and `amount` from temporary Purse to Buyer.

For (1) and (2) it uses the method deposit from PurseSpec. This exposes Seller to Buyer and opposite. The challenge is how to restrict the risk to Seller and Buyer. We shall discuss (1) and its verification.

(1) Seller and Buyer Trustworthiness



```
void deal( sellerMoney, sellerGoods, // seller's money and goods
        buyerMoney, buyerGoods,      // buyer's money and goods
        price, amnt                  // price and amount
){

// Create sellerMoneyTmp purse, of same credibility as sellerMoney

1: sellerMoneyTmp = sellerMoney.sprout();
```

Pol_6,a

{ sellerMoney is PurseSpec -> sellerMoneyTmp is PurseSpec

```
void deal( sellerMoney, sellerGoods, // seller's money and goods
        buyerMoney, buyerGoods,      // buyer's money and goods
        price, amnt                  // price and amount
){

// Create sellerMoneyTmp purse, of same credibility as sellerMoney
1:
sellerMoneyTmp = sellerMoney.sprout();
{    sellerMoney is PurseSpec ->  sellerMoney
 ∧ sellerMoney is PurseSpec ->
      (    sellerMoney.balance == sellerMoney.balance_pre  ∧
           sellerMoneyTmp.balance == 0)
```

Pol_6,a

```
void deal( sellerMoney, sellerGoods, // seller’s money and goods
        buyerMoney, buyerGoods,      // buyer’s money and goods
        price, amnt                  // price and amount
){

// Create sellerMoneyTmp purse, of same credibility as sellerMoney
1:
sellerMoneyTmp = sellerMoney.sprout();
```

{ sellerMoney is PurseSpec -> sellerMoneyT

∧ sellerMoney is PurseSpec ->

(sellerMoney.balance == sellerMoney.balance_pre ∧

sellerMoneyTmp.balance == 0)

∧ ∀p:_old PurseSpec.

(p.balance == p.balance$_{PRE}$ ∨ MayAccess(sellerMoney,p)$_{PRE}$)

∧ ∀o: Object.

MayAccess(o) ⊆ MayAccess(o)$_{PRE}$ ∪ MayAccess(sellerMoney)$_{PRE}$ }

Pol_4 & Access Prop Rule -1

Access Prop - Rule 1

{ sellerMoney is PurseSpec -> sellerMoneyTmp is PurseSpec

∧ sellerMoney is PurseSpec ->

(sellerMoney.balance == sellerMoney.balance_pre ∧

sellerMoneyTmp.balance == 0)

∧ ∀p:_old PurseSpec.

p.balance == $p.balance_{PRE}$ ∨ MayAccess(sellerMoney,p) $_{PRE}$

∧ ∀o: Object.

MayAccess(o) ⊆ MayAccess(o) $_{PRE}$ ∪ MayAccess(sellerMoney) $_{PRE}$ }

2: res=sellerMoneyTmp.transfer(0,sellerMoney);

{ sellerMoney is PurseSpec -> sellerMoneyTmp is PurseSpec

∧ **(sellerMoneyTmp is PurseSpec && res) -> sellerMoney is PurseSpec**

∧ sellerMoney is PurseSpec ->

(sellerMoney.balance == sellerMoney.balance_pre ∧

sellerMoneyTmp.balance == 0)

∧ ∀p:_old PurseSpec.

(p.balance == $p.balance_{\mathbf{PRE}}$ ∨ MayAccess(sellerMoney) $_{\mathbf{PRE}}$)

∧ ∀o: Object.

MayAccess(o) ⊆ MayAccess(o) $_{\mathbf{PRE}}$ ∪ MayAccess(sellerMoney) $_{\mathbf{PRE}}$ }

{ sellerMoney is PurseSpec -> sellerMoneyTmp is PurseSpec

∧ **(sellerMoneyTmp is PurseSpec && res) -> sellerMoney is PurseSpec**

∧ sellerMoney is PurseSpec ->

(sellerMoney.balance == sellerMoney.balance_pre ∧

sellerMoneyTmp.balance == 0) ∧

∧ $\forall p:_{PRE}$ PurseSpec.

(p.balance == $p.balance_{PRE}$ ∨ $MayAccess(sellerMoney,p)_{PRE}$)

∧ ∀o: Object.

$MayAccess(o) \subseteq MayAccess(o)_{PRE} \cup MayAccess(sellerMoney)_{PRE}$ }

3: if not(res) then {

$\forall p:_{PRE}$ PurseSpec.

p.balance == p.balance_pre || MayAccess(sellerMoney,p)_pre

∀o: Object.

MayAccess(o) ⊆ MayAccess(o)_pre ∪ MayAccess(sellerMoney)_pre

// this fulfils the spec of deal!

return res;

}

Several steps later …

$\{\quad sellerMoney \text{ is } PurseSpec \text{ <-> } sellerMoneyTmp \text{ is } PurseSpec$

$\wedge\ buyerMoney \text{ is } PurseSpec \text{ <-> } buyerMoneyTmp \text{ is } PurseSpec$

$\wedge\ \forall p:_{PRE} PurseSpec.\ (\ p.balance == p.balance_{PRE}$

$\qquad \vee\ MayAccess(sellerMoney,p)_{PRE}\ \vee\ MayAccess(buyerMoney,p)_{PRE}\ )$

$\wedge\ \forall o: Object.$

$\quad MayAccess(o) \subseteq \quad MayAccess(o)_{PRE}$

$\qquad\qquad \cup\ MayAccess(sellerMoney)_{PRE} \cup MayAccess(buyerMoney)_{PRE}\ \}$

```
8:  res=sellerMoneyTmp.transfer(0,buyerMoneyTmp);
    if (not res) return false;
```

$\{\quad sellerMoney \text{ is } PurseSpec \text{ <-> } sellerMoneyTmp \text{ is } PurseSpec$

$\wedge\ buyerMoney \text{ is } PurseSpec \text{ <-> } buyerMoneyTmp \text{ is } PurseSpec$

$\wedge\ \mathbf{sellerMoneyTmp\ is\ PurseSpec\ \text{->}\ buyerMoneyTmp\ is\ PurseSpec}$

$\wedge\ \forall p:_{PRE} PurseSpec.\ (\ p.balance == p.balance_{PRE}$

$\qquad \vee\ MayAccess(sellerMoney,p)_{PRE}\ \vee\ MayAccess(buyerMoney,p)_{PRE}\ )$

$\wedge\ \forall o: Object.$

$\quad MayAccess(o) \subseteq \quad MayAccess(o)_{PRE}$

$\qquad\qquad \cup\ MayAccess(sellerMoney)_{PRE} \cup MayAccess(buyerMoney)_{PRE}\ \}$

Conclusions

- We argued that capability policies are open, hypothetical, and necessary.
- We proposed a capability policy specification language.
- We used it to formally specify the policy for mints and purses.
- We have proven adherence of code to these policies – not these slides.
- We have specified the trust/risk policy of the Escrow.
- We have shown adherence of the Escrow code to the policies using the specification for Purses (more in separate document available on demand).

Further Work

- Revisit, Rethink everything.
  - Revisit Formal System
  - Find a natural expression of module and encapsulation
  - Prove Escrow.deal adherence to specification version 4
  - More Case Studies
- Expand Inference Rules
- Tool Development

Thank you