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

Mint

buyerEuro, 100 €

sellerEuro, 55 €

Buyer

Mint

:Purse, 20

:Seller

:Purse, 25
Buying Apples - after

Mint

: Purse, 100-5€

: Purse, 55+5€

Buyer

Mint

: Purse, 20+10

: Purse, 25-10

Seller
Buying Apples – how?

Mint

Buyer

::Purse, 100 €
::Purse, 55 €

Seller

::Purse, 20
::Purse, 25

Buyer gives 100 € to Mint.
Seller gives 20 € to Mint.
Mint gives 55 € to Buyer.
Mint gives 25 € to Seller.

Note: This diagram illustrates the exchange of money between a buyer and a seller in a market transaction.
Buying Apples 1\textsuperscript{st} attempt: pass purses

\begin{verbatim}
sellerEuros.transfer(5, buyerEuros);
\end{verbatim}
Buying Apples 1\textsuperscript{st} attempt: pass purses

\begin{verbatim}
sellerEuros.transfer(5,buyerEuros);
buyerApples.transfer(10,sellerApples);
\end{verbatim}
Buying Apples  1\textsuperscript{st} attempt pass purses - **Risk**

Risk to Buyer:  - 100 \(€\), 0
Risk to Seller:  0\(€\), -15
Buying Apples 2\textsuperscript{nd} Attempt: tmp purses
Buying Apples – 2\textsuperscript{nd} 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 2\textsuperscript{nd} Attempt: tmp purses - Risk

Risk to Buyer:  - 5 €,  0
Risk to Seller:  0€,  -10
Buying Apples 2\textsuperscript{nd} Attempt: tmp purses - Risk

Risk to Buyer: $-5\,\text{€, 0}$
Risk to Seller: $0\,\text{€, -10}$

... what if Buyer/Seller use tmp-Purses to steal from other Purses, or Mint itself?
Buying Apples – 3rd Attempt Escrow

Mint

Buyer

Purse, 95 €
Purse, 55 €

Seller

Purse, 20
Purse, 15

EscrowAgent
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.
The currency of a mint is the sum of balances of its purses.

\[
aMint_1\text{.currency} = 5 + 15 + 8
\]
Mint & Purse – objects– vrs1

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

```
aMint_1
```

```
aMint_2
```

```
aPurse_1
balance: 5
```

```
aPurse_2
balance: 15
```

```
aPurse_3
balance: 8
```

```
aPurse_4
balance: 12
```

```
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.

The currency of a mint is a *model* field of the mint.
```java
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 IllegalArgumentException(); };
        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
```

```
aPurse_1

APurse_2

APurse_3

5

15

8
```

Database
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
  - aPurse_1: 5
  - aPurse_2: 15
  - aPurse_3: 8
  - `aMint_1.currency = 5 + 15 + 8`

- aMint_2
  - aPurse_4: 12
  - `aMint_2.currency = 12`

The *currency* of a mint, and the *balance of a purse* are *model fields*.
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 IllegalArgumentException(); }
        database.put(from, database.get(from)-amnt);
        database.put(into, database.get(into) +amnt);
    }
}
Mint & Purse – objects – vrs3

```
Mint_1.currency = 5 + 15 + 8
Mint_2.currency = 12
```

Diagram showing the relationships between Mint_1, Purse_1, Purse_2, Purse_3, Purse_4, and their corresponding currencies.
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

```java
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 >= 0 && prs.balance - amnt >= 0;

        POST this.balance_{new} = this.balance_{old} + amount &&
        prs.balance_{new} = this.balance_{new} - amount;
    }
```
Classical spec. does \textit{not} imply policies!

Code below satisfies classical specification, but breaks policies.

```java
final class Mint {
}

final class Purse {
    private final Mint mint;
    private long balance;

    public Purse(Mint mint, long balance) {
        if (balance<0) {
            throw ...  
        }
        this.mint = mint;
        this.balance = balance;
    }

    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;  }
}
```

allows \textit{mint} to be set externally; thus may affect \textit{currency of a mint} \textbf{(breaks Pol 2)}

allows \textit{balance} to be set externally; thus may transfer money without access to second \textit{Purse}, or may affect \textit{currency of a mint} \textbf{(breaks Pol 1, and Pol 2)}
Classical spec. does not imply policies - 2

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

```java
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 $Q$

$$M \models Q$$

iff

$$\forall M'. \ \forall (\kappa, _) \in Arising(M*M'). \ \ M*M', \ \kappa \models Q$$

• $Arising(M)$ is the set of all configuration, code pairs which may arise through execution of the initial configuration with $M$. 
Open policy, increases number of configurations considered

\[ M \models Q \iff \forall M'. \forall (\kappa, _) \in Arising(M*M'). M*M', \kappa \models Q \]
Open policy, increases number of configurations considered

\[ M \models Q \iff \forall M'. \forall (\kappa, _) \in \text{Arising}(M \times M'). \quad M \times M', \kappa \models Q \]

Only reachable configurations, i.e. decrease number of configurations considered.
Our Policy Specification Language - 2

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

\[ M \models \{ Q \} \text{code} \{ R \} \iff \forall M'. \forall (\kappa, _) \in Arising(M*M'). M*M', \kappa \models Q \land M*M', \text{code}, \kappa \sim \kappa', \nu. \Rightarrow M*M', \kappa' \models R. \]

- $M*M'$, $\text{code}$, $\kappa \sim \kappa'$, $\nu$ is the large steps semantics.
Our Policy Specification Language - 2

Open policy

\[ M \models \{ Q \} \text{ code } \{ R \} \]

iff

\[ \forall M'. \forall (\kappa, _) \in Arising(M*M'). M*M', \kappa \models Q \land M*M', \text{ code}, \kappa \sim \kappa', \forall. \]

\[ \Rightarrow \]

\[ M*M', \kappa' \models R. \]

Only reachable configurations considered.
**Pol_1:** With two purses of the same mint, one can transfer money between them.

\[
\text{Pol}_1 \equiv \\
\{ \ p1 \text{ is PurseSpec}, \\
p2 \text{ is PurseSpec}, \\
p1.\text{balance} \geq \text{amt}, \\
p1.\text{mint} = p2.\text{mint} \ \} \\
p1.\text{transfer}(\text{amt}, p2 ) \\
\{ \ p1.\text{balance} = p1.\text{balance}_{\text{old}} - \text{amt}, \\
p2.\text{balance} = p1.\text{balance}_{\text{old}} + \text{amt}, \\
"nothing else changed" \ \}
\]
**Pol_1**: With two purses of the same mint, one can transfer money between them.

\[
\text{Pol}_1 \equiv \\
\{ \text{p1 is PurseSpec,} \\
\text{p2 is PurseSpec,} \\
\text{p1.balance} \geq \text{amt,} \\
\text{p1.mint} = \text{p2.mint} \} \\
\text{p1.transfer(amt, p2)} \\
\{ \text{p1.balance} = \text{p1.balance}_{\text{old}} - \text{amt,} \\
\text{p2.balance} = \text{p1.balance}_{\text{old}} + \text{amt,} \\
\text{“nothing else changed”} \}
\]

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.

\[
\text{Pol}_2 \equiv \\
\forall m : \text{MintSpec}. \forall o : \text{Object}. \\
\text{MayAffect}(o, m.\text{currency}) \\
\implies \\
\text{MayAccess}(o, m)
\]

This is an execution invariant.

Again, we are using model fields.

Note predicates `MayAccess`, and `MayAffect`. 
The meaning of MayAffect

- Take a runtime configuration $\kappa$, module $M$, a variable $x$, and a pure expression $e$

$$\begin{align*}
M, \kappa \models \text{MayAffect}(x, e) \\
\text{iff} \\
\exists m. \ M, x.m(\ldots), \kappa \sim \kappa', \_ \land \left[ e \right]_{\kappa} \neq \left[ e_{\text{pure}} \right]_{\kappa'}
\end{align*}$$

In the above, we are using notation as follows

- Large step semantics $M, \text{expr}, \kappa \sim \kappa', \nu$
- The value of a pure expression $e_{\text{pure}}$ in context of $\kappa$ is $\left[ e_{\text{pure}} \right]_{\kappa}$
The meaning of \textbf{MayAccess}

- For a configuration $\kappa$, and variables $x$ and $y$

\[
M, \kappa \models \text{MayAccess}(x,y) \iff \exists f_1, \ldots, f_n. \left\lfloor x.f_1.f_2. \ldots f_n \right\rfloor_{\kappa} = \left\lfloor y \right\rfloor_{\kappa}
\]

- For a configuration $\kappa$, and variable $x$

\[
\text{MayAccess}(x)_{\kappa} = \{ o \mid \exists f_1, \ldots, f_n. \left\lfloor x.f_1.f_2. \ldots f_n \right\rfloor_{\kappa} = o \}
\]
**Pol_3**: The mint can only inflate its own currency.

\[
\text{Pol}_3 \equiv \forall m : \text{MintSpec.} \\
\{ \text{true} \}
\]

any

\[\{ \text{m.currency} \geq \text{m.currency}_{\text{old}} \}\]
Pol_4: No one can affect the balance of a purse they do not have.

Pol_4 \equiv
\forall p : \text{PurseSpec}. \forall o : \text{Object}.
\text{MayAffect}(o, p.\text{balance})
\Rightarrow
\text{MayAccess}(o, p)
**Pol_4:** No one can affect the balance of a purse they do not have.

\[ \forall p : \text{PurseSpec}. \forall o : \text{Object.}\]
\[ \text{MayAffect}(o, p.\text{balance}) \]
\[ \Rightarrow \]
\[ \text{MayAccess}(o, p) \]

Note use of model fields, and predicates \text{MayAccess}, and \text{MayAffect}. 
The meaning of PublicAccess

- For a configuration $\kappa$, and variables $x$ and $y$

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

if $x_\kappa = y_\kappa$

$\lor$ exists field $f$.

$\exists$ public methods $m_1, \ldots, m_n$.

$$x.m_1(\ldots).m_2(\ldots). \ldots m_n(\ldots)_\kappa = y_\kappa$$
**Pol_4, revisited:** No one can affect the balance of a purse they 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.

\[
\text{Pol}_4.b \equiv \forall p : \text{PurseSpec}. \forall o : \text{Object}. \\
p \in \text{ThisModule}, o \notin \text{ThisModule}. \\
\text{MayAffect}(o, p.\text{balance}) \implies \\
\exists o' \notin \text{ThisModule}. \\
\text{PublicAccess}(o', p)
\]

*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 \(o' \in M\).
Open policy, increases number of configurations considered

\[ M \models Q \quad \text{iff} \quad \forall M'. \; \forall (\kappa, \_ ) \in Arising(M*M'),
\]

\[ M*M', \; \kappa \models Q[ThisModule/M] \]
We now give meaning to variable ThisModule

\[ M \models Q \iff \forall M'. \forall (\kappa, _) \in \text{Arising}(M \ast M'). M \ast M', \kappa \models Q[\text{ThisModule}/M] \]
**Pol_5**: Balances are always non-negative.

\[ Pol_5 \equiv \forall p : \text{PurseSpec}. \ p.\text{balance} \geq 0 \]
**Pol_6:** *A reported successful deposit can be trusted as much as one trusts the purse one is depositing into.*

We introduce the notation

\[ p \text{ is } \text{PurseSpec} \]

To express that \( p \) adheres to specification \text{PurseSpec}.
Pol_6: *A reported successful deposit can be trusted as much as one trusts the purse one is depositing into.*

\[
\text{Pol}_6, a \equiv \\
\{ \text{true} \} \\
\text{res} = p.\text{transfer}(\text{amt}, p') \\
\{ \text{res} \land p \text{ is PurseSpec} \\
\implies \\
p' \text{ is PurseSpec} \\
\land p.\text{mint} = p'.\text{mint} \\
\land p'.\text{balance}_{\text{old}} \geq \text{amt} \\
\land p'.\text{balance} = p'.\text{balance}_{\text{old}} - \text{amt} \\
\land p.\text{balance} = p.\text{balance}_{\text{old}} + \text{amt} \}
\]
**Pol_6:** A reported successful deposit can be trusted as much as one trusts the purse one is depositing into.

\[ \text{Pol}_6, a \equiv \{ \text{true} \} \]

\[
\begin{align*}
\text{res} &= p.\text{transfer}(\text{amt}, p') \\
\{ \text{res} \land p \text{ is PurseSpec} \} &\implies \text{p'} \text{ is PurseSpec} \\
&\land p.\text{mint} = p'.\text{mint} \\
&\land p'.\text{balance}_{\text{old}} >= \text{amt} \\
&\land p'.\text{balance} = p'.\text{balance}_{\text{old}} - \text{amt} \\
&\land p.\text{balance} = p.\text{balance}_{\text{old}} + \text{amt} \}
\]
Pol_6: *A reported successful deposit can be trusted as much as one trusts the purse one is depositing into.*

\[
\text{Pol}_6, b \equiv \\
\{ \text{true} \} \\
\{ \text{res} = p.\text{sprout}() \} \\
\{ \text{res} \land p \text{ is PurseSpec} \} \\
\Rightarrow \\
\text{res is PurseSpec} \\
\land p.\text{mint} == \text{res.mint} \\
\land \text{res.balance} = 0 \\
\land \text{“all else is unmodified”} \} \\
\]
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

```java
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 sellerApples from same mint
// buyerEuros has more than price euros
// sellerApples has more than amount apples
```
The Escrow specification - 1\textsuperscript{st} case

```csharp
public bool deal(
    buyerEuros, buyerApples sellerEuros, sellerApples,
    amount, price )

POST:
[ res=true ∧
( buyerEuros, buyerApples,
  sellerEuros, sellerApples is PurseSpec )
```
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 ∧
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

```java
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

```csharp
public bool deal(
    buyerEuros, buyerApples sellerEuros, sellerApples, 
    amount, price )

POST:
[  res=true ∧
( buyerEuros, buyerApples, 
    sellerEuros, sellerApples is PurseSpec )
  ∧
  buyerEuros.mint == sellerEuros.mint ∧
  buyerApples.mint == sellerApples.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 + amnt ∧
  buyerApples.balance == buyerApples.balance_pre - amnt ∧
```
The Escrow specification - 1\textsuperscript{st} case

\textbf{public bool} deal(
    buyerEuros, buyerApples sellerEuros, sellerApples, amount, price )

\textbf{POST:}
[ res=true \land
  ( buyerEuros, buyerApples, sellerEuros, sellerApples \textbf{is} PurseSpec )
    \land
  buyerEuros.mint == sellerEuros.mint \land
  buyerApples.mint == sellerApples.mint \land
  buyerEuros.balance\textsuperscript{pre} \geq price \land
  sellerApples.balance\textsuperscript{pre} \geq amnt \land
  buyerEuros.balance == buyerEuros.balance\textsuperscript{pre} - price \land
  sellerEuros.balance == sellerEuros.balance\textsuperscript{pre} + price \land
  buyerApples.balance == buyerApples.balance\textsuperscript{pre} + amnt \land
  buyerApples.balance == buyerApples.balance\textsuperscript{pre} - amnt \land
  \forall p: \textsuperscript{pre} PurseSpec. p.balance == p.balance\textsuperscript{pre} ]

\lor

... 2\textsuperscript{nd} 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 ...
  \[ \text{res}=false \land \\
  ( \text{buyerEuros, buyerApples,} \\
    \text{sellerEuros, sellerApples is PurseSpec} ) \\
  \land \\
  \text{buyerEuros.mint} == \text{sellerEuros.mint} \land \\
  \text{buyerApples.mint} == \text{sellerApples.mint} \land \\
  ( \text{buyerEuros.balance}_{\text{pre}} < \text{price} \lor \\
    \text{sellerApples.balance}_{\text{pre}} < \text{amnt} ) \land \\
  \forall p:_{\text{pre} \text{PurseSpec.} p.\text{balance} == p.\text{balance}_{\text{pre}} } \]

V

... 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

```java
public boolean 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

```java
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.balance_{pre} ]
    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 - 3rd case, vrs 2

```java
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.balance_{pre}
   ∨ MayAffect(sellerEuros,p)_{pre} ]
```
The Escrow specification - 3rd case, vrs 2

```java
public boolean deal(
    buyerEuros, buyerApples, sellerEuros, sellerApples,
    amount, price )

POST:
... 1st case ... ∨ ... 2nd case ...
    ∨
[ res=false ∧
    buyerEuros is PurseSpec ∧
    ( ¬ (sellerEuros is PurseSpec ) ∨
        sellerEuros.mint ≠ buyerEuros.mint )
    ∧
    ∨ 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 \( g.duped(buyerMoneyTmp) \).

```java
class Gullible {
    ...
    method duped(p'){
        p'.transfer(300000,prs);
    }
}
```
The Escrow specification - 3\textsuperscript{rd} case, vrs 3

```csharp
public bool deal(
    buyerEuros, buyerApples, sellerEuros, sellerApples, amount, price )

POST:
... 1\textsuperscript{st} case ... \lor ... 2\textsuperscript{nd} case ...
\lor
[ res=false \land
  buyerEuros is PurseSpec \land
  ( \neg (sellerEuros is PurseSpec ) \lor
    sellerEuros.mint \neq buyerEuros.mint )
\land
  \forall_{p: \text{PurseSpec}} p.balance == p.balance_{\text{pre}}
  \lor PubAccess(sellerEuros,p)_{\text{pre}} ]
```

This specification is satisfied by Escrow.deal, :-), provided that PurseSpec also satisfies
\[ \text{Pol}_7 \equiv \forall_{p,p'} : \text{PurseSpec} \ p \neq p' \Rightarrow \neg \text{MayAccess}(p, p') \]
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 :-{.
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 )
  ∧
  ∨ ( ∃∉ 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

\[
\{ \text{true} \} \\
\x\m(y) \\
\{ \forall z, z' : \text{pre \ Object.} \\
\text{MayAccess}(z, z')_{\text{pre}} \Rightarrow \\
\text{MayAccess}(z, z')_{\text{pre}} \\
\lor \text{MayAccess}(x, z')_{\text{pre}} \\
\lor \text{MayAccess}(y, z')_{\text{pre}} \\
\}
\]

These restrictions on \text{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 \textit{not} expressed by the Hoare triple above. We need to find a way of expressing this.
Access Propagation - Rule 2

\[
\{ \text{true} \} = \{ x.m(y) \}
\]

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

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.
The method Escrow.deal
1. Establishes that the Buyer’s and the Seller’s Euro Purses have the same trustworthiness. Aborts, if unsuccessful.
2. Establishes that the Buyer’s and the Seller’s 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

buyerEuro, 100 €
sellerEuro, 55 €

buyerEuroTmp: 0 €
sellerEuroTmp: 0 €

Buyer, Seller

same 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();
{ sellerMoney is PurseSpec -> sellerMoneyTmp is PurseSpec

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
  sellerMoneyTmp = sellerMoney.sprout();
  { sellerMoney is PurseSpec -> sellerMoney
    ∧ sellerMoney is PurseSpecSpec ->
      (sellerMoney.balance == sellerMoney.balance_pre ∧
       sellerMoneyTmp.balance == 0)
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
    sellerMoneyTmp = sellerMoney.sprout();
    {
        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}
    }
\{ 
\text{sellerMoney is PurseSpec} \rightarrow \text{sellerMoneyTmp is PurseSpec} \\
\land \text{sellerMoney is PurseSpec} \rightarrow \\
\quad \left( \text{sellerMoney.balance} == \text{sellerMoney.balance\_pre} \land \\
\quad \text{sellerMoneyTmp.balance} == 0 \right) \\
\land \forall p: \_\text{old PurseSpec}. \\
\quad p.\text{balance} == p.\text{balance}_{\text{PRE}} \lor \text{MayAccess(sellerMoney,p)}_{\text{PRE}} \\
\land \forall o: \text{Object}. \\
\quad \text{MayAccess(o) } \subseteq \text{MayAccess(o)}_{\text{PRE}} \cup \text{MayAccess(sellerMoney)}_{\text{PRE}} \} \\

2: \text{res=sellerMoneyTmp.transfer(0,sellerMoney);} \\

\{ 
\text{sellerMoney is PurseSpec} \rightarrow \text{sellerMoneyTmp is PurseSpec} \\
\land \left( \text{sellerMoneyTmp is PurseSpec} \land \text{res} \right) \rightarrow \text{sellerMoney is PurseSpec} \\
\land \text{sellerMoney is PurseSpec} \rightarrow \\
\quad \left( \text{sellerMoney.balance} == \text{sellerMoney.balance\_pre} \land \\
\quad \text{sellerMoneyTmp.balance} == 0 \right) \\
\land \forall p: \_\text{old PurseSpec}. \\
\quad \left( p.\text{balance} == p.\text{balance}_{\text{PRE}} \lor \text{MayAccess(sellerMoney)}_{\text{PRE}} \right) \\
\land \forall o: \text{Object}. \\
\quad \text{MayAccess(o) } \subseteq \text{MayAccess(o)}_{\text{PRE}} \cup \text{MayAccess(sellerMoney)}_{\text{PRE}} \}
{ sellerMoney is PurseSpec -> sellerMoneyTmp is PurseSpec
\land ( sellerMoneyTmp is PurseSpec && res ) -> sellerMoney is PurseSpec
\land sellerMoney is PurseSpec ->
  ( sellerMoney.balance == sellerMoney.balance_{pre} \land
    sellerMoneyTmp.balance == 0 ) \land
\land \forall p:\text{PRE PurseSpec}.
  ( p.balance == p.balance_{pre} \lor \text{MayAccess(sellerMoney,p)}_{\text{PRE}} )
\land \forall o:\text{Object}.
  \text{MayAccess(o)} \subseteq \text{MayAccess(o)}_{\text{PRE}} \cup \text{MayAccess(sellerMoney)}_{\text{PRE}} }$

3: if not(res) then {
\forall p:\text{PRE PurseSpec}.
  p.balance == p.balance_{pre} \lor \text{MayAccess(sellerMoney,p)}_{\text{pre}}
\forall o:\text{Object}.
  \text{MayAccess(o)} \subseteq \text{MayAccess(o)}_{\text{pre}} \cup \text{MayAccess(sellerMoney)}_{\text{pre}}$

// this fulfils the spec of deal!

  \text{return res;}
}

Several steps later ...

\[
\{ 
\text{sellerMoney is PurseSpec} \leftrightarrow \text{sellerMoneyTmp is PurseSpec} \\
\land \text{buyerMoney is PurseSpec} \leftrightarrow \text{buyerMoneyTmp is PurseSpec} \\
\land \forall p: \text{PurseSpec}. \left( p.\text{balance} = p.\text{balance}_{\text{PRE}} \lor \text{MayAccess(sellerMoney}, p)_{\text{PRE}} \lor \text{MayAccess(buyerMoney}, p)_{\text{PRE}} \right) \\
\land \forall o: \text{Object}. \\
\text{MayAccess(o) \subseteq \text{MayAccess(o)}_{\text{PRE}}} \\
\lor \text{MayAccess(sellerMoney)}_{\text{PRE}} \lor \text{MayAccess(buyerMoney)}_{\text{PRE}} \}
\]

8: \text{res=sellerMoneyTmp.transfer(0,buyerMoneyTmp);} \\
\text{if (not res) return false;}

\[
\{ 
\text{sellerMoney is PurseSpec} \leftrightarrow \text{sellerMoneyTmp is PurseSpec} \\
\land \text{buyerMoney is PurseSpec} \leftrightarrow \text{buyerMoneyTmp is PurseSpec} \\
\land \text{sellerMoneyTmp is PurseSpec} \rightarrow \text{buyerMoneyTmp is PurseSpec} \\
\land \forall p: \text{PurseSpec}. \left( p.\text{balance} = p.\text{balance}_{\text{PRE}} \lor \text{MayAccess(sellerMoney}, p)_{\text{PRE}} \lor \text{MayAccess(buyerMoney}, p)_{\text{PRE}} \right) \\
\land \forall o: \text{Object}. \\
\text{MayAccess(o) \subseteq \text{MayAccess(o)}_{\text{PRE}}} \\
\lor \text{MayAccess(sellerMoney)}_{\text{PRE}} \lor \text{MayAccess(buyerMoney)}_{\text{PRE}} \}
\]
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