

# Rationally Reconstructing the Escrow Example

James Noble

Victoria University of Wellington  
kix@ecs.vuw.ac.nz

Sophia Drossopoulou

Imperial College, London  
scd@doc.ic.ac.uk

## Abstract

The Escrow Exchange Contract has been used as a case study of building up complex and trustworthy systems from basic object capabilities, in the context of concurrent and distributed programming. In this short paper we present a Rational Reconstruction of the Escrow Exchange Contract case study, expressed in Grace, concentrating on the most essential issues of trustworthiness, and ignoring issues to do with distribution or more complex protocols. We then use our notation for capability policies to specify the key features of the reconstructed case study.

**Categories and Subject Descriptors** D.3.1 [Programming Languages]: Formal Definitions and Theory; D.4.6 [Security and Protection]: Verification; K.4.4 [Electronic Commerce]: Payment Schemes, Security

## 1. Introduction

Miller, Cutsem, and Tulloh [10]’s Escrow Exchange Contract case study aims to demonstrate how “contracts can be specified elegantly and executed safely, given an appropriate distributed, secure, persistent, and ubiquitous computational fabric”. An escrow exchange contract is a trusted third party that guarantees that two other mutually untrusting services can exchange rights with one another — for example, trading an agreed number of shares in a company for an agreed amount of a virtual currency. For reasons both political and technical, Miller, Cutsem, and Tulloh [10]’s design does not assume a universal clearing service, a government or supranational agency, laws of contract or tort, nor their corresponding components of a software architecture such as a shared database or a trusted transaction service. Rather, the entire mechanism of contracts is built from the bottom up based on object capabilities. The Exchange Escrow Contract case study is important precisely because it is useful to gauge how well object-capability systems in fact support building up larger scale structures of cooperation and trust, even between mutually untrusting counterparties.

One of Miller et al. [10]’s goals for their case study is that “non-experts should be able to write smart contracts understandable by other non-experts.” While the design of the case study is indeed clear and elegant, the code unavoidably tangles a number of cross-cutting concerns (especially concurrency and distribution) with the

```
type Purse = {  
  hashCode -> Number  
  name -> String  
  makePurse -> Purse  
  deposit(amt : Number, src : Purse) -> Done  
}
```

Figure 1. The type Purse

code that manipulates the object-capabilities. To give a little of the chthonic flavour of the code from Miller et al. [10], the lines

```
var makeEscrowPurseP =  
  Q.join(srcPurseP ! makePurse, dstPurseP ! makePurse);  
test whether the srcPurse and dstPurse objects belong to the same  
mint, in part by an asynchronous object identity test on the lambda  
closure object stored in each purse’s makePurse field. In the version  
in this paper, we don’t need this check, but if we did, it would be  
written something like: “srcPurse.mint == dstPurse.mint.”
```

In our previous work [4], we analysed Miller’s Mint and Purse example [8] by expressing it in Joe, a Java subset without reflection and static fields, and discussed the six capability policies that characterise the correct behaviour of the program, as proposed in [8]. We argued that these policies require a novel approach to specification, and showed some first ideas on how to use temporal logic. In our yet unpublished technical report [5], we propose a specification language, and use it to fully specify the six policies from [8]; however, their formalization showed that they allowed several possible interpretations. We also uncovered the need for another four policies and formalized them as well.

**Contributions** In this paper we provide a rational reconstruction [2] of the Escrow case study into Grace [1]. Our version concentrates on the main features of the case study: the fact that a buyer and seller who do not trust each other may safely agree on a (trusted) contract, and expect, when they fulfil their side of the obligation, the contract to go through, or to receive a reimbursement. Moreover each party may back out of a contract that has not yet been completed. We do not directly address issues of synchronization, persistence, or distribution here; rather we assume an effectively object-synchronous programming model (either with actors, fully-synchronized Java-style objects, or a single threaded event loop) where underlying middleware deals with all issues of communications failure, replication, etc. We propose capability policies crucial to the correct operation of the translated case study, and express them in our policy notation [5].

**Outline of paper and Structure of our Solution** We first give an outline of our solution. We have a class mint which creates, and keeps in a ledger, objects of type Purse, and a singleton object escrowAgent, which creates and keeps track of objects of class contract.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FTJP’14, July 29 2014, Uppsala, Sweden.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-2866-1/14/07...\$15.00.

http://dx.doi.org/10.1145/2635631.2635850



Execution of a code snippet, code for a module  $M$  takes a configuration  $\kappa$  and returns a value  $v$  and a new configuration  $\kappa'$ . We assume large step semantics of shape  $M, \kappa, \text{code} \rightsquigarrow \kappa', v'$ .

**Paths** We support path expressions  $p$  (i.e., expressions which only involve field reads). For example, `mint`, and `prs.mint` are paths. Paths are interpreted in the context of runtime configurations,

$[\cdot] : \text{Path} \rightarrow \text{RTConf} \rightarrow \text{Value}$

so that  $[\text{p}]_{\kappa} = v$  if  $p$  is a path and  $\emptyset, \kappa, p \rightsquigarrow \kappa, v$ .

**Reached and Arising Snapshots** When defining adherence to policies, it is essential to consider only those snapshots (i.e., configuration and code pairs) which may arise through the execution of the given modules. For example, if we considered *any* well-formed snapshots (well-formed in the sense of the type system), then we would be unable to show, e.g., that balances are always positive as mandated in [9, 10]. Namely, a configuration with a negative balance would be well-formed, but will never actually arise in the execution of the program.

$\text{Reach}(M, \kappa, \text{code})$  [5] is the set of snapshots corresponding to the start of the execution of the body of any constructor or method called in the process of executing code in the context of  $M$  and  $\kappa$ . Note that  $\text{Reach}(M, \kappa, \text{code})$ , corresponds to the *complete* body of a method.

$\text{Arising}(M)$  is the set of snapshots which may be reached during execution of some initial snapshot,  $\kappa_0, \text{code}_0$ .

**Accessible Objects**  $\text{AccAll}(M, \kappa)$  is the set of objects accessible from the frame in  $\kappa$  through *any* path, including confidential fields.

The notation  $z :_{\kappa} c$  indicates that  $z$  is the name of an object which exists in the heap of  $\kappa$  and belongs to class  $c$  — with no requirement that there should be a path from the frame to this object. The notation  $\kappa \in c$  expresses that the currently executing method in  $\kappa$  comes from  $c$ , while  $\kappa \in M$  expresses that the class of the currently executing method is defined in  $M$ .

**Execution Histories** An execution history,  $h$ , is a sequence of snapshots containing all the method calls which arise during one execution, in the order in which these calls were received, and where the snapshots of nested calls follow the snapshot of the nesting call. We use the operator  $\cdot$  to compose histories, for example  $h_1 \cdot (\kappa, \text{code}) \cdot h_2$  is a history. The set  $\text{Histories}(M)$  is the set of all histories which execute code from  $M$ .

## 4. Escrow Agent

The basic mint and purse system allows two untrusting components to make a payment (see section 2 above). But this payment is a one way transaction: a payer pays a payee. Supporting “electronic rights” (in Miller et al. [10]’s redolent phrase) requires contracts, that is, two way exchanges where some currency and some goods (e.g shares) change hands atomically — again in an environment where neither payee nor payer trust each other. The remainder of the Escrow case study uses mints and purses to build such a Escrow Contract system. We now describe our Grace translation / simplification of the escrow system, showing its implementation and the capability policies that are crucial to its correctness.

Given that each side is untrusting of the other, the first challenge is for both sides to have received the same contract object with the same understanding that this object will embody the actual contract. Following Miller et al. [10]’s design we provide a trusted escrow agent object that issues contracts to buyers and sellers.

Our escrow agent is shown in figure 3. Compared with Miller et al. [10]’s escrow agent, this is simpler and more straightforward, because again our code intentionally focuses on the core behaviour of the design, especially regarding the object capabilities, rather than details of middleware infrastructure. Our design

```
def escrowAgent = object { // well known singleton

  class contract.new(name' : String) { ... } // see fig 4

  var terms : String
  var currentContract : Contract
  var waitingForSeller := true

  // called by seller to request a seller-side contract
  method getSellerContract(terms' : String) -> Contract {
    if (!waitingForSeller)
      then { Error.raise "already has seller" }
    terms := terms'
    waitingForSeller := false // now waiting for a buyer
    currentContract := contract.new(terms)
    return currentContract
  }

  // called by buyer to request a buyer-side contract
  method getBuyerContract(terms' : String) -> Contract {
    if (waitingForSeller) then {
      Error.raise "waiting for a seller" }
    if (terms != terms') then {
      Error.raise "terms don't match" }
    def thisContract = currentContract
    waitingForSeller := true
    return thisContract
  }
}
```

**Figure 3.** The core of the Escrow Agent translated into Grace.

also makes a number of other simplifications: we assume Grace objects and classes are single-threaded as if in fully-synchronized Java; and we adopt a very asymmetric protocol where the seller (payee) must always ‘move’ first, followed by the buyer (payer). The `waitingForSeller` variable keeps track of who should move next, the buyer or the seller.

Thus, the seller asks for and is returned a contract; when a matching buyer arrives they will be issued the same contract object.

```
// Alice the seller moves first
def alice = object {
  def alicesContract =
    escrowAgent.getSellerContract("some terms")
  ...
}

// Bob the buyer moves second
def bob = object {
  def bobsContract =
    escrowAgent.getBuyerContract("some terms")
  ...
}
```

Of course, a real implementation would need to keep track of a list of potential buyers, so that a buyer requesting a contract with no matching seller would not deadlock — but these are essentially bookkeeping issues that do not affect the exercise of object capabilities within the system.

Compared with Miller et al. [10] we also rely on a number of end-to-end arguments. For example there is no way for a seller to determine that no buyer has accepted the contract, or for a buyer or seller to explicitly reject a contract when issued. The reason we don’t deal with these issues here is that the contract object itself, and the protocol around its use, seamlessly resolve all these issues in the end-to-end context of the system as a whole.

### 4.1 Capability Policies for the Escrow Agent

We propose some policies which govern the class `escrowAgent`. We believe that the policies below are essential, but giving a complete set of policies is beyond the scope of this paper.

#### 4.1.1 PolA1: getSellerContract returns fresh contracts.

The method `getSellerContract` returns fresh contracts, or throws an exception. We distinguish two versions of the policy:

$$\begin{array}{c}
 \text{Module } M \text{ satisfies policy } \mathbf{PolA1, vrs1} \\
 \text{iff} \\
 \forall \kappa. M, \kappa, \text{escrowAgent.getSellerContract}(\_) \rightsquigarrow \kappa', v \\
 \implies \\
 v \in \text{Error} \vee (v :_{\kappa'} \text{contract} \wedge v \notin \text{dom}(\kappa))
 \end{array}$$

In other words, if we execute `escrowAgent.getSellerContract()`, then we either obtain an exception ( $v \in \text{Error}$ ), or we obtain a fresh contract ( $v \notin \text{dom}(\kappa)$  and  $v :_{\kappa'} \text{contract}$ ).

On the other hand, **PolA1, vrs2** gives sufficient conditions under which the execution will not throw an error, and describes the state after execution of `getSellerContract()`.

$$\begin{array}{c}
 \text{Module } M \text{ satisfies policy } \mathbf{PolA1, vrs2} \\
 \text{iff} \\
 \forall \kappa, \text{txt} : \text{String}. [\text{escrowAgent.waitingForSeller}]_{\kappa} \\
 \wedge M, \kappa, \text{escrowAgent.getSellerContract}(\text{txt}) \rightsquigarrow \kappa', v \\
 \implies \\
 v :_{\kappa'} \text{contract} \wedge v \notin \text{dom}(\kappa) \\
 \wedge [\text{escrowAgent.waitingForSeller}]_{\kappa'} = \text{false} \\
 \wedge [\text{escrowAgent.contract}]_{\kappa'} = v \wedge [v.\text{terms}]_{\kappa'} = \text{txt} \\
 \wedge [\text{escrowAgent.offered}]_{\kappa'} = \text{false}
 \end{array}$$

Thus, if `escrowAgent` is in state where `waitingForSeller` is true, and executes `getSellerContract(txt)`, then `waitingForSeller` resp. `offered` will become false resp. true, and a new contact will be created, containing `txt` as the terms.

The above specification is a complete description of the behaviour of `getSellerContract`, but exposes its implementation's details, some of which are only relevant to the working of other methods, e.g., setting the field `text` is necessary for the correct working of `getBuyerContract`. The well-known approach to alleviate such exposures is the introduction of ghost variables [6, 7]; in section 4.1.4 we will show instead specifications based on the possible sequences of method calls — in line with [3], we believe that this is a more abstract style.

Note that both versions of **PolA1** could have been expressed in a Hoare Logic variant, e.g. in [6]. They both are concerned with *sufficient* conditions for the creation of contracts. The next policy, **PolA2**, will be concerned with *necessary* conditions. Using the terminology we introduced in [4], the former is a *rely* property, while the latter is a *deny* property, i.e., a new contract cannot be created *unless* the method `getSellerContract` is called.

#### 4.1.2 PolA2: Contracts created only through getSellerContract

The policy **PolA2** guarantees that any newly created contract must have been the result of calling `getSellerContract` on `escrowAgent`.

$$\begin{array}{c}
 \text{Module } M \text{ satisfies policy } \mathbf{PolA2} \\
 \text{iff} \\
 \forall M'. \forall c. \kappa \notin \text{escrowAgent}, \text{contract} \\
 \wedge M * M', \kappa, \text{code} \rightsquigarrow \kappa', v \\
 \wedge c :_{\kappa'} \text{contract} \wedge c \notin \text{dom}(\kappa).
 \end{array}$$

$$\begin{array}{c}
 \implies \\
 \exists \kappa'', v', \kappa''' \\
 c :_{\kappa'''} \text{contract} \wedge c \notin \text{dom}(\kappa'') \\
 \wedge (\kappa'', \text{escrowAgent.getSellerContract}(\_)) \in \\
 \text{Reach}(M * M', \kappa, \text{code}) \\
 \wedge M * M', \kappa'', \text{escrowAgent.getSellerContract}(\_) \rightsquigarrow \\
 \kappa''', [c]_{\kappa''}.
 \end{array}$$

In other words, if a snapshot  $(\kappa, \text{code})$  which does not execute code from `escrow` (expressed through  $\kappa \notin \text{escrowAgent}$ ) leads to the creation of a contract  $c$  (expressed through  $M * M', \kappa, \text{code} \rightsquigarrow \kappa', v$  and  $c :_{\kappa'} \text{contract}, c \notin \text{dom}(\kappa)$ ), then the execution of that snapshot must have gone through an intermediate snapshot which executed the method `getSellerContract` (expressed through requirement  $(\kappa'', \text{escrowAgent.getSeller...} \in \text{Reach}(M * M', \kappa, \text{code}))$  and which indeed, created that contract (expressed through  $M * M', \kappa'', \dots \rightsquigarrow \kappa''', [c]_{\kappa''}$  and  $c :_{\kappa'''} \text{contract} \wedge c \notin \text{dom}(\kappa'')$ ).

This policy forbids any other way of creating contracts than through `escrowAgent`. Thus, if class `contract` was public, i.e., not encapsulated within the `escrowAgent`, it would still satisfy **PolA1**, but would not satisfy **PolA2**.

In this policy, we quantify over all possible legal extensions of the code, i.e. over all modules  $M'$ . If we did not, then the version where `contract` was public would satisfy the policy, provided that  $M$  did not contain any offending code. This demonstrates what we call the *open* nature of policies [4]. Moreover, we restrict our requirement to snapshots outside `escrowAgent` or `contract` ( $\kappa \notin \text{escrowAgent}, \text{contract}$ ); this is necessary in order to exclude calls within those modules. Without this premise, the policy would not be satisfiable. Finally, note that we cannot replace in the above the use of class `contract` by the type `Contract`. Grace's types are primarily structural, and it takes a little more effort to rule out something that is structurally a `Contract`, even through it was not created by the class `contract`.

#### 4.1.3 PolA3: Contracts may only be obtained through calls of getSellerContract or getBuyerContract

**PolA3** guarantees that the only way to obtain contracts encapsulated in a module  $M$  is through the call of the functions `getSellerContract` or `getBuyerContract`. Thus, this policy forbids the addition, of, say, a public method `getContract` accessor that returns a contract, and which would make it possible to leak a contract to a malicious third party.

$$\begin{array}{c}
 \text{Module } M \text{ satisfies policy } \mathbf{PolA3} \\
 \text{iff} \\
 \forall M'. \forall (\kappa, \text{code}) \in \text{Arising}(M * M'). \forall c :_{\kappa'} \text{contract}. \\
 M * M', \kappa, \text{code} \rightsquigarrow \kappa', v \\
 \wedge \kappa, \kappa' \in M' \wedge [c]_{\kappa'} \in \text{AccAll}(M * M', \kappa') \\
 \implies \\
 [c]_{\kappa'} \in \text{AccAll}(M * M', \kappa) \vee \\
 [ \exists \text{code}', \kappa'', \kappa'''. M * M', \kappa'', \text{code}' \in \\
 \text{Reach}(M * M', \kappa, \text{code}) \\
 \wedge \kappa'', \text{code}' \rightsquigarrow \kappa''', [c]_{\kappa'} \\
 \wedge (\text{code} = \text{escrowAgent.getSellerContract}(\_) \\
 \vee \text{code} = \text{escrowAgent.getBuyerContract}(\_)) ].
 \end{array}$$

In other words, if execution of code external to  $M$  ( $\kappa \in M'$ ) leads to a configuration which has access to contract object  $c$  ( $[c]_{\kappa'} \in \text{AccAll}(M * M', \kappa')$  and  $c :_{\kappa'} \text{contract}$ ), then either  $c$  was

already accessible to  $M'$  ( $\lceil c \rceil_{\kappa'} \in AccAll(M * M', \kappa)$ ), or  $c$  was returned through execution of `escrowAgent.getSellerContract(-)`, or `escrowAgent.getBuyerContract(-)`.

#### 4.1.4 Pol\_A4: Interplay of `getBuyerContract` and `getSellerContract`

**Pol\_A4, vrs1** guarantees that a call of `getSellerContract(txt)` followed by a call to `getBuyerContract(txt)` will return the same object, provided that it is not preceded by an unmatched call of `getSellerContract(-)`, and provided that there is no intermediate call on the `escrowAgent` between the two calls.

We first define the predicate  $Matched(-)$  on histories, which requires that either there are no calls on `escrowAgent`, or that any `getSellerContract(-)` are followed by a corresponding call of `getBuyerContract(-)`:

$Matched(h)$  iff  $(-, escrowAgent.getSellerContract(txt)) \notin h$ , or there exists  $h_1, h_2, h_3$  such that  
 $h = h_1 \cdot (-, escrowAgent.getSellerContract(txt)) \cdot h_2 \cdot (-, escrowAgent.getBuyerContract(txt)) \cdot h_3$ ,  
and  $Matched(h_1)$ , and  $(escrowAgent.get.....(-), -) \notin h_2$ , and  $Matched(h_3)$ .

We now express the policy:

Module  $M$  satisfies policy **Pol\_A4**  
iff  
 $\forall M', h_1, h_2, h_3, txt : String. \forall h \in Histories(M * M').$   
 $h = h_1 \cdot (\kappa_1, escrowAgent.getSellerContract(txt)) \cdot h_2 \cdot$   
 $(\kappa_2, escrowAgent.getBuyerContract(txt)) \cdot h_3$   
 $\wedge Matched(h_1)$   
 $\wedge (escrowAgent.get...Contract(-), -) \notin h_2$   
 $\implies$   
 $\exists \kappa_3, \kappa_4, v.$   
 $M * M', \kappa_1, escrowAgent.getSellerContract(txt) \rightsquigarrow \kappa_3, v$   
 $\wedge M * M', \kappa_2, escrowAgent.getBuyerContract(txt) \rightsquigarrow$   
 $\kappa_4, v.$

The policy from above, together with **Pol\_A1, vrs1** will guarantee that the returned value is a contract, which is new in  $\kappa_1$ .

We now consider the *deny*-counterpart of that policy, which says that calls of `getSellerContract(-)` only succeed if the previous calls on `escrowAgent` are matched correctly: We omit the complementary policy for the calls of `getBuyerContract(-)`.

Module  $M$  satisfies policy **Pol\_A4, vrs2**  
iff  
 $\forall M', h_1, h_2, \kappa_1, \kappa_2, v. \forall h \in Histories(M * M').$   
 $h = h_1 \cdot (\kappa_1, escrowAgent.getSellerContract(txt)) \cdot h_2$   
 $\wedge M * M', \kappa_1, escrowAgent.getSellerContract(txt) \rightsquigarrow \kappa_3, v$   
 $\wedge v :_{\kappa_3} contract$   
 $\implies$   
 $Matched(h_1)$

## 5. Escrow Contract

Figure 4 implements our version of the actual escrow contract object. Again, this uses a protocol where we expect the seller to move first and populate the contract, and then the buyer to move and accept the contract. The seller must pass two purses into the offer method — the `sellersGoods` purse that contains the goods to be sold, and `sellersMoney` a (presumably empty) temporary purse

```
class contract.new(name' : String) {
  var offered := false
  var sellersGoods : Purse
  var amount : Number
  var price : Number
  var sellersMoney : Purse

  method offer(sellersGoods' : Purse,
              amount' : Number,
              price' : Number,
              sellersMoney' : Purse) {
    sellersGoods := sellersGoods'
    amount := amount'
    price := price'
    sellersMoney := sellersMoney'
    offered := true
  }

  method bid(buyersGoods : Purse,
            amount' : Number,
            price' : Number,
            buyersMoney : Purse) -> Done {
    if (!offered) then { Error.raise "Not offered" }
    if ( (amount != amount') || (price != price') ) then
      { Error.raise "Bid/Offer mismatch" }
    if ( (amount < 0) || (price < 0) ) then
      { Error.raise "Bid/Offer fraud" }

    // make temporary escrow purses
    def moneyEscrow : Purse = buyersMoney.makePurse
    def goodsEscrow : Purse = sellersGoods.makePurse

    // make extra temporary purses for check
    def moneyTmp : Purse = sellersMoney.makePurse
    def goodsTmp : Purse = buyersGoods.makePurse

    // check purses are from the same mints
    moneyTmp.deposit( 0, moneyEscrow )
    moneyEscrow.deposit( 0, moneyTmp )
    goodsTmp.deposit( 0, goodsEscrow )
    goodsEscrow.deposit( 0, goodsTmp )

    // here we go — uncaught exceptions from deposit end the bid
    moneyEscrow.deposit( price, buyersMoney )

    try { goodsEscrow.deposit( amount, sellersGoods ) }
    catch { _ -> buyersMoney.deposit( price, moneyEscrow );
           Error.raise "TXN FAILURE" }

    sellersMoney.deposit( price, moneyEscrow )
    buyersGoods.deposit( amount, goodsEscrow )
  }
}
```

**Figure 4.** The Escrow Contract translated into Grace.

that will receive the money. Similarly the buyer supplies an empty `buyersGoods` purse and a (temporary) `buyersMoney` purse that contains the payment when they call the bid method. The transaction succeeds if the bid method completes.

```
// Alice the seller
def mDst = mint.newPurse("Alice's mDst", 0)
def gSrc = goods.newPurse("Alice's gSrc", 7)
alicesContract.offer(gSrc, 7, 10, mDst)

// Bob the buyer
def mSrc = mint.newPurse("Bob's mSrc", 10)
def gDst = goods.newPurse("Bob's's gDst", 0)
bobsContract.bid(gDst, 7, 10, mSrc)
```

The seller can cancel the contract at any time before settlement by simply emptying their `sellersGoods` purse: this will cause the overall transaction to fail when the buyer makes a bid. Similarly

the buyer can cancel by not calling bid. The only subtlety is the use of the moneyEscrow and goodsEscrow purses towards the end of the bid method — the goods and money are first moved into these escrow purses so that an unscrupulous party cannot remove the goods or money while the transaction is in progress. From our capability policy perspective, this complexity is essential, not accidental, because it captures the heart of the escrow behaviour.

## 5.1 Capability Policies for Contracts

### 5.1.1 Pol\_C1: Effects and conditions of exchange

The key policy for the escrow contract is that once it receives a bid matching the offer, it performs the transfer of goods and moneys, provided the corresponding purses come from the same mint. Here we write the deny counterpart of this policy, which says that if, as the result of a bid, the buyer's goods balance changes, then the bid must have been preceded by the creation of pairs of purses in corresponding mints, and of an offer which matched the bid:

$$\begin{array}{c}
\text{Module } M \text{ satisfies policy } \mathbf{Pol\_C1} \\
\text{iff} \\
\forall M', h_1, h_2, c, \kappa, \kappa', v. \forall \text{amt}, \text{prc} : \mathbb{N}. \\
\quad \forall h \in \mathcal{H}istories(M * M'). \\
c :_{\kappa} \text{contract} \wedge p_g :_{\kappa} \text{Purse} \wedge p_m :_{\kappa} \text{Purse} \\
\wedge h = h_1 \cdot (\kappa, c.\text{bid}(p_g, \text{amt}, \text{prc}, p_m)) \cdot h_4 \\
\wedge M * M', \kappa, c.\text{bid}(p_g, \text{amt}, \text{prc}, p_m) \rightsquigarrow \kappa', v \\
\wedge \lceil p_g.\text{balance} \rceil_{\kappa'} \neq \lceil p_g.\text{balance} \rceil_{\kappa} \\
\implies \\
( \lceil p_g.\text{balance} \rceil_{\kappa'} = \lceil p_g.\text{balance} \rceil_{\kappa} + \text{amt} \\
\wedge \lceil p_m.\text{balance} \rceil_{\kappa'} = \lceil p_m.\text{balance} \rceil_{\kappa} - \text{prc} \\
\wedge \lceil p_g.\text{balance} \rceil_{\kappa} \geq \text{prc} \\
\wedge \\
[ \exists p'_g, p'_m, \text{mnt}_m, \text{mnt}_g, h_3, h_4, \\
\kappa'', \kappa_1, \kappa'_1, \kappa_2, \kappa'_2, \kappa_3, \kappa'_3, \kappa_4, \kappa'_4. \\
h_1 = h_3 \cdot (\kappa'', c.\text{offer}(p'_g, \text{amt}, \text{prc}, p'_m)) \cdot h_4 \\
\wedge \text{mnt}_m :_{\kappa_2} \text{Mint} \wedge \text{mnt}_g :_{\kappa_4} \text{Mint} \\
\wedge (\kappa_1, \text{mnt}_m.\text{makePurse}(-)), (\kappa_3, \text{mnt}_g.\text{makePurse}(-)) \in h_1 \\
\wedge (\kappa_2, \text{mnt}_m.\text{makePurse}(-)), (\kappa_4, \text{mnt}_g.\text{makePurse}(-)) \in h_3 \\
\wedge M * M', \kappa_1, \text{mnt}_m.\text{makePurse}(-) \rightsquigarrow p_m, \kappa'_1 \\
\wedge M * M', \kappa_2, \text{mnt}_m.\text{makePurse}(-) \rightsquigarrow p'_m, \kappa'_2 \\
\wedge M * M', \kappa_3, \text{mnt}_g.\text{makePurse}(-) \rightsquigarrow p_g, \kappa'_3 \\
\wedge M * M', \kappa_4, \text{mnt}_g.\text{makePurse}(-) \rightsquigarrow p'_g, \kappa'_4 \\
\wedge p_m :_{\kappa_1} \text{Purse} \wedge p'_m :_{\kappa_2} \text{Purse} \\
\wedge p_g :_{\kappa_3} \text{Purse} \wedge p'_g :_{\kappa_4} \text{Purse} \\
\wedge \lceil p'_g.\text{balance} \rceil_{\kappa'} = \lceil p'_g.\text{balance} \rceil_{\kappa} - \text{amt} \\
\wedge \lceil p_m.\text{balance} \rceil_{\kappa'} = \lceil p_m.\text{balance} \rceil_{\kappa} + \text{prc} \\
\wedge ( (\kappa, c.\text{bid}(p'_g, \text{amt}', \text{prc}', p'_m)) \in h_3 \rightarrow \text{amt} \neq \\
\text{amt}' \dots ) \\
\wedge \lceil p'_g.\text{balance} \rceil_{\kappa} \geq \text{amt} \quad ] )
\end{array}$$

In the above,  $p_m$  and  $p_g$  stand for the buyer's money and goods' purse, while  $p'_m$  and  $p'_g$  stand for the seller's money and goods' purse. The policy expresses that if a call of bid modified the balance in the buyer's goods purse, then the modification was exactly by amt, while the buyer's monte purse will have been debited the price. Moreover, there exist further purses,  $p'_m$  and  $p'_g$  which were created by the same mints as  $p'_m$  and  $p'_g$  respectively (lines 14-25). Furthermore,  $p'_g$  is debited by amt, while  $p'_m$  is credited by prc (lines 26-27). Also, there was no bid of sufficient money between the offer at  $\kappa''$  and the bid at  $\kappa$  (lines 28-29) and there were enough goods in the seller's goods' purse (line 30).

### 5.1.2 Pol\_C2: Bids do not leak references

Finally, this policy guarantees that execution of a bid does not leak references to third parties — satisfying the requirement that the escrow system mediate between two mutually untrusting objects.

$$\begin{array}{c}
\text{Module } M \text{ satisfies policy } \mathbf{Pol\_C2} \\
\text{iff} \\
\forall M', \kappa, c, o, o'. \\
(\kappa, c.\text{bid}(p_g, \text{amt}, \text{prc}, p_m)) \in \mathcal{A}rising(M * M'). \\
\wedge c :_{\kappa} \text{contract} \\
\wedge M * M', \kappa, c.\text{bid}(p_g, \text{amt}, \text{prc}, p_m) \rightsquigarrow \kappa', v \\
\wedge o \in \mathcal{A}ccAll(o', \kappa') \\
\implies \\
o \in \mathcal{A}ccAll(o', \kappa)
\end{array}$$

In other words, if after execution of  $c.\text{bid}(p_g, \text{amt}, \text{prc}, p_m)$ , the object  $o'$  has access to the object  $o$ , then it already had access to it before execution of the call.

## 6. Conclusions and Future Work

In this paper we have presented a rational reconstruction of Miller, Cutsem, and Tulloh's contract exchange escrow case study. We have translated the code from JavaScript to Grace, and have removed code corresponding to the crosscutting concerns of distribution, asynchrony, genericity, and symmetry, leaving the core of the program that records, exchanges, and exercises object capabilities.

We have proposed some of the capability policies that must be maintained to demonstrate that the escrow design meets its specifications, and clarified them through formal specifications.

In further work, we want to complete the definition of execution observations, complete the policies, consider whether they are minimal (can any of these be inferred from the others), and prove that the Grace code adheres to these policies.

**Acknowledgments** We thank the anonymous referees for their comments. This work is partially supported by the Royal Society of New Zealand Marsden Fund, and by the EU FP7 project Upscale.

## References

- [1] Andrew P Black, Kim B Bruce, Michael Homer, and James Noble. Grace: the absence of (inessential) difficulty. In *Onward!* ACM, 2012.
- [2] Simon Blackburn. *The Oxford Dictionary of Philosophy*. 2008.
- [3] F. S. de Boer, S. de Gouw, and J. Vinju. Prototyping a tool environment for run-time assertion checking in JML with communication histories. In *FT/JFP, ACM DL*, 2010.
- [4] Sophia Drossopoulou and James Noble. The need for capability policies. In *FT/JFP*, 2013.
- [5] Sophia Drossopoulou and James Noble. Towards capability policy specification and verification. available from authors' website, May 2014. <http://ecs.victoria.ac.nz/Main/TechnicalReportSeries>.
- [6] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Iowa State Univ. [www.jmlspecs.org](http://www.jmlspecs.org), February 2007.
- [7] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.
- [8] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, Maryland, 2006.
- [9] Mark Samuel Miller, Chip Morningstar, and Bill Frantz. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. Springer, 2000.
- [10] Mark Samuel Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in JavaScript. In *ESOP*, 2013.