Swapsies on the Internet

First Steps towards Reasoning about Risk and Trust in an Open World

Sophia Drossopoulou[1], James Noble[2], Mark S. Miller[3]

[1]Imperial College London, [2]Victoria University Wellington, [3]Google Inc.

Abstract

Contemporary open systems use components developed by many different parties, linked together dynamically in unforeseen constellations. Code needs to live up to strict *security specifications:* it has to ensure the correct functioning of its objects when they collaborate with external objects which may be malicious.

In this paper we propose specifications that model risk and trust in such open systems. We specify Miller, Van Cutsem, and Tulloh's escrow exchange example, and discuss the meaning of such a specification. We argue informally that the code satisfies its specification.

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

Playground Swapsies Imagine you are a child swapping football stickers in your school playground [40]. Perhaps you've got three Zinedine Zidanes, but you really want a Wiremu Reid? If you hand over the stickers you've got to trade to a gorilla from year ten, how do you know he won't run off with them before he gives you the stickers you need?

Football sticker swapsies illustrates the two complementary forces of *trust* and *risk*. When you show some of the stickers you've got to someone you do not trust, you risk that they might run off with those stickers, or rip them up. Awareness of trust means you can manage the amount of risk you are willing to tolerate. You're quite likely to lend your sticker book to your best friend, but you'd probably be careful to take only stickers you're definitively willing to risk losing when you go to meet the gorilla in year ten.

Internet Swapsies Playground swapsies is just one example of an interaction between *mutually untrusting* parties in an *open system* — other examples include so-called dark Internet markets (like Silk Road and Evolution) and even larger trading systems like eBay when participants choose not to rely on protection systems like PayPal. The systems are composed of different components (i.e. objects) from a range of different providers. There is no central trusted authority, no effective recourse to a universal clearing house, or a government or supranational agency — teachers don't interfere, and the kids don't want them to interfere anyway. This is for both political and technical reasons: a single trusted component becomes a single point of failure for the system if it is compromised, and typically requires a centralised architecture, such as shared databases or trusted transaction services.

Escrow Agents As a motivating example, we use the Escrow Exchange [27], a trusted third party that manages exchanges of different goods (e.g. money and shares) between two counterparties. The escrow exchange is a good example because:

- The escrow does not rely on a central agency to determine a party's trustworthiness, but it can ask one party whether it vouches for the other party's trustworthiness.
- The exchange must be transactional. If some requirement is not met the whole transaction must be aborted; if everything is OK the whole transaction must succeed.
- The escrow cannot guarantee that it will not attempt the operation if one or more of the counterparties are untrustworthy, and thus cannot guarantee absence of risks; its aim is to minimise the risks involved.

The escrow example is important because it gauges how robustly trust and risk are managed. Risk and trust are the focus of our interest, rather than the particulars of exchanges. Supporting the escrow exchange is insufficient to build a whole system, but we consider it necessary: a system that cannot support the escrow exchange will not be fit for most purposes.

Our Contributions We address the following questions:

- How can we specify trust?

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 ACM 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.

PLAS 2015, July 06 2015, Prague, Czech Republic.
Copyright © 2015 ACM 978-1-4503-3661-1/15/07. . . $15.00.
http://dx.doi.org/10.1145/2786558.2786564

- How can we gauge risk?
- How can we express the open nature of systems?
- How can we prove that program code meets the specifications for risk and trust?

To specify trust, we were inspired by Lampson et al.'s "speaks for" and "says" authentication constructs [21] and propose an **obeys** predicate that describes whether the current object trusts another object to satisfy some specification. As in [21], the **obeys** predicate is *relative* because it relates two objects to a specification: objects (or even the same objects) may have different relationships with different specifications. It is also *hypothetical*, in the sense that there is no "trust-bit" to observe in the run-time configuration. Rather, **obeys** states that in all future configurations, one object trusts another to satisfy the specification. Trust is often *conditional*, in the sense that trust can sometimes be established only under the condition that some other object is trusted.

To address risk, we propose further hypothetical predicates $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$, which describe whether an object may access or may modify a certain property.

To address the open nature of the systems, we make the meaning of such assertions parametric with *any* code that may be linked to the current system. Thus, in order for the software to guarantee its specification, it must be written in a very robust manner, so that no further, malicious code can steal its secrets or break its integrity.

To prove program's adherence to risk/trust specifications, we have developed Hoare-style rules dealing with these hypothetical predicates. We do not discuss these in this paper.

Findings We have developed a specification of the escrow system, and we informally argue that the code adheres to it.

We were surprised to find that the specification for the Escrow is weaker than originally anticipated in two significant aspects: the escrow *cannot guarantee* that a reported successful transaction implies a) that the participants were trustworthy, nor that b) the participants are exposed to no risk by an untrustworthy participant (but we were able to characterize the risk to which participants are exposed). We were even more surprised to realize that it is *impossible to write* an escrow which would give guarantees a) and b) — all the more striking given that the third co-author is one of the original developers of the escrow example.

Research Context An important aspect of the escrow example is that it does not *prevent* the exchange from happening with untrusted parties; indeed, as we mentioned above, it cannot do so. Many common approaches to security cannot deal with this example, because they aim to prevent such exchanges, and avoid the risk altogether. Information flow systems detect when information leaks to an untrusted party; confinement systems guarantee that state is not read or leaked to other components; sandboxes ensure that components cannot affect the world outside the sandbox. In the absence of a central universally trusted authority, the escrow cannot characterize parties as high or low, or maintain sandboxes, and thus cannot prevent exchanges between untrustworthy parties. Instead, it can only manage the associated risk. Our approach is to reason explicitly about code's security properties and guarantees, as trusted and untrusted components interact in an open world. We aim to describe how components coöperate to establish trust gradually, and to delineate the risks involved in that coöperation.

Paper Organization Section 2 introduces the Escrow Exchange, and shows why a traditional specification is not descriptive enough and why a naive implementation is not robust enough. Section 3 introduces our constructs for modelling trust and risk; we use them to give revised specifications, and argue that revised code satisfies these specifications. Section 4 sketches our specification language; section 5 discusses related work, and section 6 concludes.

Disclaimers Throughout this paper, we make the simplifying assumptions that no two different arguments to methods are aliases, that the program is executed sequentially, that we can quantify over the entire heap, that objects do not breach their own encapsulation or throw exceptions, that machines on open networks are not mutually suspicious, and that any underlying network is error-free. While these problems are correctly addressed in the code proposed in [27], we do not address them in this work. This allows us to keep the specifications short, and to concentrate on the questions of risk and trust. Aliasing, concurrency, quantification, confinement, network errors, and exceptions can be dealt with using known techniques, but doing so would not shed any further light on the questions addressed here.

2. Purse and Escrow

We will now work our way though a first version of an escrow exchange implementation (developed from [32]), and give a "traditional" specification, where trust is assumed and risk is implicit. In the next section we show how explicit representations of trust and risk let us specify components more precisely, and reason about program code more accurately.

2.1 Purses

The escrow exchange example is based upon a system for modelling resources proposed in [28]. The "mints" and "purses" (or alternatively "banks" and "accounts"), can model anything fungible, including currencies or commodities: a mint models a type of currency or commodity, and a purse models an amount of that currency or commodity. We can model both money and goods by purses, although we distinguish them dynamically by using purses from different mints for money and goods respectively.

Figure 1 shows a traditional specification of the Mints and Purses. Specifications consist of sets of (ghost) fields and policies, and are predicates over classes and objects.

```
specification Purse  {
  field mint    // Mint
  field balance // Number

  policy Pol_deposit_1    //   1st case:
     dest,src:Purse ∧ SameMint(dest,src) ∧ 0 ≤amt≤src.balance
         { res = dest.deposit(amt, src) }
     res   ∧
     dest.balance=dest.balance_pre+amt   ∧ src.balance=src.balance_pre−amt   ∧
     ∀p:_pre Purse\{dest,src} p.balance=p.balance_pre

  policy Pol_deposit_2     // 2nd case:
     prs,src:Purse ∧ ¬( SameMint(dest,src) ∧ 0≤amt≤src.balance  )
         { res = dest.deposit(amt, src) }
     ¬res ∧ ∀p:_pre Purse. p.balance=p.balance_pre

  policy Pol_sprout
     p:Purse
         { res = p.sprout() }
     res:Purse ∧ SameMint(res,p) ∧ res.balance=0 ∧ ∀p:_pre Purse. p.balance=p.balance_pre ∧ res ≠ p
}

predicate SameMint(prs1,prs2) ≡ prs1:Purse ∧ prs2:Purse ∧ prs1.mint=prs2.mint
```

Figure 1. Specification of Purse — First Attempt

Any object which adheres to a specification may be safely assumed to satisfy all the policies in the specification.

A Purse has two ghost fields: mint, which is expected to point to an object of class Mint, and balance which is expected to be a value of type Number. Note that ghost fields need not appear as such within Purse objects.

A Mint object acts as a token representing a particular currency or type of goods. Mints have no public methods, but can e.g. be tested for identity to help verify transactions between purses.

The specification of Purse consists of three policies: Pol_deposit_1 and Pol_deposit_2 describe the behaviour of the method deposit, and Pol_sprout describes the behaviour of sprout.

We use the standard convention and distinguish the values of terms trm before and after execution of a method through the subscripts trm_{pre} and trm_{post}. Moreover, when omitting the subscript, we mean the value after execution. For example, the assertion dest.balance $=$ dest.balance$_{pre}$ $+$ amt says that the balance of dest after execution of the method will be the sum of amt and the balance of dest before execution of the method. All methods either return a boolean value, or a pointer to an object. We follow the conventions that the result of a method call is assigned to a variable, res. We simulate exceptions and exception handling, by checking the return value, and if it is false, returning from the method. A full model for exceptions will be part of future work.

A Purse object represents a particular purse (account). Money can be transferred between any two purses of the same mint through the method deposit. If the destination and source are purses (dest, src:Purse) from the same mint (predicate SameMint), and the source purse's balance covers the amount to be deposited, then the amount is transferred to the destination purse without modifying any other purses, and the call returns true – c.f. policy Pol_deposit_1. If the two purses are not from the same mint, or the source purse has insufficient funds, the transaction does not take place, all purses remain unaffected, and the call returns false – c.f. policy Pol_deposit_2.

A new purse can be created at any time by asking an existing purse to sprout — this returns a new, empty purse from the same mint – c.f. Pol_sprout. The new purse has a zero balance but can then be filled via deposit.

All three policies have a post-condition of the form

$\forall$p:$_{pre}$ Purse. p.balance=p.balance$_{pre}$

or a variation thereof. This post-condition guarantees that the method call will not modify the balance of any pre-existing purse. This assertion is necessary when reasoning about the effect of calling Purse-methods from the Escrow, and is, essentially, a primitive way of expressing framing. We will study better framing in further work.

To make a secure payment, the payer will typically make a new, empty, temporary purse from one of their existing purses via sprout, and deposit only enough funds for the payment into the temporary purse. The payer then passes the temporary purse to the payee, who then empties it back into their primary purse. This allows two *mutually untrusting* components to transfer funds, provided that they both trust the mint and purse system.

2.2 Specifying Swapsies

Figure 2 is our first attempt at specifying an escrow exchange deal. An Escrow object has fields sellerMoney,

```
specification Escrow {
   fields sellerMoney, sellerGoods, buyerMoney, buyerGoods //  Purse
   fields price, amt   // Number

 policy Pol_deal_1    //   1st case:
    SameMint(buyerMoney,sellerMoney) ∧  SameMint(buyerGoods,sellerGoods) ∧ price, amt:ℕ  ∧
    buyerMoney.balance_pre ≥price ∧  sellerGoods.balance_pre ≥amt
           { res = deal( ) }
    res ∧
    buyerMoney.balance=buyerMoney.balance_pre−price ∧ sellerMoney.balance=sellerMoney.balance_pre+price ∧
    buyerGoods.balance=buyerGoods.balance_pre+amt ∧ sellerGoods.balance=sellerGoods.balance_pre−amt ∧
    ∀p:_pre Purse_pre \ {sellerMoney,sellerGoods, buyerMoney,buyerGoods}: p.balance=p.balance._pre

 policy Pol_deal_2    //   2nd case:
     ¬( SameMint(buyerMoney,sellerMoney) ∧  SameMint(buyerGoods,sellerGoods) ∧
     buyerMoney.balance_pre ≥price ∧  sellerGoods.balance_pre ≥amt  )
           { res = deal( ) }
     ¬res ∧
     ∀p:_pre Purse: p.balance=p.balance._pre
}
```

Figure 2. Specification of `Escrow` :: `deal` – First Attempt

`sellerGoods`, `buyerMoney`, and `buyerGoods`, which represent the money and the goods purses of the buyer and the seller. The fields `amt` and `price` serve to describe the amount of goods to be exchanged, and the price of this exchange. These fields will be supplied by other methods (see [27]) which create the contract.

Even though we expect `sellerMoney`, `sellerGoods`, `buyerMoney`, and `buyerGoods` to be objects representing `Purses`, the `Escrow` cannot guarantee nor check this. The first, superficial reason for this is, that our language is dynamically typed. The deeper reason is that `Escrow` does not necessarily know all the classes which are legitimate implementations of `Purse` objects. As we shall see later, the `Escrow` can successfully co-ordinate objects of many different classes which implement the `Purse` functionality.

The specification consists of two policies: `Pol_deal_1` promises that if the purses come from the same mints, and have sufficient funds (lines 6-7), then the result will be `true` (line 9), the transfer of the monies and the goods will take place (lines 10-11), and all other purses will remain unaffected (line 12). `Pol_deal_2` promises that if the purses do not come from the same mints, or have insufficient funds, then the result will be `false` and all purses will be unaffected.

2.3 Swapping via an Escrow Purse

The `deal` method in Figure 3 shows a first attempt at its implementation. To make the exchange transactional, it uses a pair of private *escrow* purses, one for on each side of the transaction (money and goods). Rather than swapping money and goods between buyer's and seller's purses in one go, the buyer's money and seller's goods are moved first into escrow purses, and then from the escrow purses into the final destinations. In this way, we only complete the second half of the transaction when we are sure enough money and goods are securely in the escrow purses. If the transaction needs to be abandoned halfway through, we can return the buyer's money from the escrow purse without any reference to the seller.

This code from Figure 3 is based on [27] and [32]. First, two escrow purses (`escrowMoney` and `escrowGoods`) are sprouted from the inputs — lines 3–6. The escrow purses are newly created within the method, and so cannot be manipulated by the buyer or seller.

Second, we attempt to escrow the buyer's money by transferring it from the `buyerMoney` purse into the new `escrowMoney` purse — line 8. According to the specification (Fig. 1), if this request returns true, then the money will have been transferred and both purses must be from the same mint. If the request fails we abort the transaction.

Third, we attempt to escrow the seller's goods – line 14, again by depositing them into the escrow purse. If we are unsuccessful, we again abort the transaction, after we have returned that money to the buyer – lines 21 and 22.

At this point (line 24) the `deal` method should have sole access to sufficient money and goods in the escrow purses. The method completes the transaction by transferring the escrowed money and goods into the respective destination purses – lines 26 and 27. Thanks to the escrow purses, these transfers should not fail so this code should meet the Figure 2's specification. If only the truth were that simple.

2.4 The failure of `dealV1`

The `dealV1` method in Figure 3 does not satisfy the `Escrow` specification in Figure 2 — in fact, in an open system, it *cannot*. The critical problems are assumptions about trust: both the code and the specification implicitly trust the purse objects with which they interact. Considering both the `Purse` and `Escrow` specifications: what happens if a purse or escrow is asked to interact with an untrustworthy purse? How

```
1  method dealV1( )
2   {
3     // make temporary money Purse
4     escrowMoney = sellerMoney.sprout
5     // make temporary goods Purse
6     escrowGoods = buyerGoods.sprout
7
8     res = escrowMoney.deposit(price, buyerMoney)
9     if (!res) then
10         // insufficient money in buyerMoney
11         // or different money mints
12         { return false }
13
14    // sufficient money, same mints
15    // price transferred to escrowMoney
16    res = escrowGoods.deposit(amt, sellerGoods)
17    if (!res) then
18         // insufficient goods in sellerGoods
19         // or different goods mints
20         {  // undo the money transaction
21            buyerMoney.deposit(price,escrowMoney)
22            return false }
23
24    // price in escrowMoney, amt in escrowGoods
25    // now complete the transaction
26    buyerMoney.deposit(price, escrowMoney)
27    sellerGoods.deposit(amt, escrowGoods)
28 }
```

Figure 3. First attempt at escrow deal

much risk is involved: just the potentially untrustworthy purse? That purse plus any other purse it knows about, or interacts with (e.g. both are passed into the same method)? Any purse (or indeed any object) anywhere in the system?

Classical specifications like Figures 1 and 2 have no notion of the risks involved when an object does not meet its specification. All bets are off: the world ends. Just because we can't write specifications, however, doesn't mean that we can't write programs: unfortunately the code in Figure 3 is in no better shape than the specification. Imagine if `sellerMoney` was a malicious, untrustworthy object. At line 4, the `sprout` call could itself return a malicious object, which would then be stored in `escrowMoney`. Then at line 11, during execution of `escrowMoney.deposit(price, buyerMoney)` the malicious `escrowMoney` purse could steal all the money out of `buyerMoney` purse, and still return `false`. As a result, the buyer would lose all their money, and receive no goods! Even if the buyer was more cautious, and themselves sprouted a special temporary purse with a balance of exactly `price` to pass in as `buyerMoney`, they would still lose all this money without any recompense.

Perhaps there is something else we could do — a `trusted` method on every object, say, that returns `true` if the object is trusted, and `false` otherwise? The problem, of course, is that an object that is untrustworthy is, well, untrustworthy: we cannot expect a `trusted` method ever to return `false`. This leads to our definition of trust: trust is *hypothetical*, and in relation to some specification.

3. Specifying Trust and Risk Explicitly

The key claim of this paper is that we need specifications that let us talk about trust and risk explicitly. In this section, we begin by informally introducing three novel specification language constructs: **obeys** to model trust, and $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$ to model risk. We then revisit the specifications from the previous section using these constructs, showing how they can be used to specify the purse and escrow examples, and we argue informally that a revised `escrow` method can in fact meet revised specifications.

3.1 Modelling Trust: obeys

To model trust, we introduce a special predicate, "**obeys**", of the form o **obeys** $Spec$ which we interpret to mean that the current object trusts o to adhere to the specification $Spec$.

Because we generally can't be sure that an object — especially one supplied from elsewhere in an open system — can actually be trusted to obey a particular specification, our reasoning and specifications tend to be *hypothetical*: analysing the same piece of code under different trust hypotheses — i.e. assuming that particular objects may or may not be trusted to obey particular specifications.

Thus, *if* object `o` can be trusted to obey specification `Spec`, and `Spec` had a policy describing the behaviour of some method `m`, then we may expect the method call `o.m`(...) to behave according to that policy — otherwise, all bets are off. This also leads to chains of hypothetical reasoning; every method request on an object introduces a case-split on whether the object satisfies its specification.

More about the formal treatment of the **obeys** predicate in section 4, definition 6.

3.2 Modelling Risk: $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$

To model risk, we introduce predicates $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$, which express whether an object may read or may affect another object or property. We will write "$\mathcal{M}ay\mathcal{A}ffect$`(o,p)`" to mean that it is possible that some method invocation on o would affect the object or property p. Similarly, we will write "$\mathcal{M}ay\mathcal{A}ccess$`(o,p)`" to mean that it is possible that the code in object o could potentially gain a capability to access to p — that is, a reference to p. In practice, $\mathcal{M}ay\mathcal{A}ccess$`(o,p)` means that p is in the transitive closure of the points-to relation on the heap starting from o.

More about the formal foundation of $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$ in definition 1 in section 4.

3.3 Valid Purse: the Policies of `Purse`

We will now revisit the specifications for `Purse` and `Escrow` and give their policies using the new features introduced in the previous section. Once again, we begin by considering the specification of purses, before going on to the specification and then implementation of the escrow itself.

```
specification ValidPurse(dest){
  field balance // Number

  policy Pol_deposit_1     //   1st case:
      amt∈ ℕ
         { res = dest.deposit(amt, src) }
      res → (
              // TRUST
              src obeys pre ValidPurse ∧ CanTrade(dest,src)pre
              // FUNCTIONAL SPECIFICATION
              ∧ 0≤amt≤src.balancepre ∧
              dest.balance=dest.balancepre+amt ∧ src.balance=src.balancepre−amt  ∧
              //RISK
              ∀p.(p obeys pre ValidPurse ∧ p∉{dest,src} → p.balance=p.balancepre)  ∧
              ∀o:preObject. ∀ p obeys pre ValidPurse.  MayAccess(o,p) → MayAccesspre(o,p)  )

  policy Pol_deposit_2     //   2nd case:
      amt∈ ℕ
         { res = dest.deposit(amt, src) }
       ¬res → (
              // TRUST and FUNCTIONAL SPECIFICATION
              ¬( src  obeys pre  ValidPurse ∧ CanTrade(dest,src)pre ∧ 0≤amt≤src.balancepre) ∧
              // RISK
              ∀p.(p obeys pre ValidPurse → p.balance=p.balancepre)   ∧
              ∀o:preObject. ∀ p obeys pre ValidPurse.  MayAccess(o,p) → MayAccesspre(o,p)  )

  policy Pol_sprout
      true
         { res = dest.sprout() }
      // TRUST
      res obeys  ValidPurse ∧  CanTrade(dest,res)pre ∧
      // FUNCTIONAL SPECIFICATION
      res.balance=0 ∧
      // RISK
      ∀p.(p obeys pre ValidPurse → p.balance=p.balancepre ∧ res ≠ p)  ∧
      ∀o:preObject. ∀ p obeys pre ValidPurse.  MayAccess(o,p) → MayAccesspre(o,p)  )

  policy Pol_can_trade_constant
       true
         { any_code }
       ∀ prs1,prs2 obeys pre ValidPurse. CanTrade(prs1,prs2) ⟷ CanTradepre(prs1,prs2)

  policy Pol_protect_balance
      // RISK
      ∀ o,p:Object. p obeys ValidPurse ∧ MayAffect(o,p.balance) → MayAccess(o,p)
}

abstract predicate CanTrade(prs1,prs2) is transitive, symmetric
```

Figure 4. Specification of `ValidPurse`

Figure 4 revisits the purse specification policies from Figure 1, making the risk and trust explicit.

Note that the specification is parametric with `dest`, the receiver of all method calls described in the policies. This reflects the fact that policies are essentially predicates over objects. Any object which obeys the specification may be safely assumed to satisfy all the policies in that specification. More details and definitions in section 4.

Note also that instead of the concrete predicate `SameMint`, we are using an abstract predicate `CanTrade` which holds when two `Purses` can trade with each other. `CanTrade` must be transitive and symmetric, but does not require that its arguments have the same class or mint: just that `deposit` can transfer currency from one purse to another. This could involve a clearing house, interbank exchange, or could simply boil down to `SameMint`. The point is that an abstract predicate can be satisfied in different ways by different classes.

We now consider the policies in turn. `Pol_deposit_1` and `Pol_deposit_2` taken together distinguish between a successful and an unsuccessful deposit, signalled by returning `true` or `false` respectively. In the first case, i.e. `Pol_deposit_1` where the result is `true`, argument `src` must have been a valid purse ("'src **obeys** ValidPurse") which could trade with the receiver , and `src` must have

sufficient balance. In the second case, i.e. `Pol_deposit_2` where the result is `false`, either `src` was not a valid purse, or would not trade with the receiver, or had insufficient funds.

The last two lines in the postcondition of `Pol_deposit_1` and `Pol_deposit_2` provide framing conditions: In the first case, that all other purses will be unmodified (line 14 in figure 4) , whereas in the second case no purses will be modified (line 24 in figure 4). Moreover, the framing conditions from lines 15, 25 and 36 of figure 4 and not stated in figure 1, require that the methods do not leak access to any `ValidPurse` object. In other words, if after the method call, a pre-existing `o` has access to a `ValidPurse` object `p`, then `o` had access to a `p` already before the call.

The key difference between the `ValidPurse` specification and the earlier `Purse` specification is that `ValidPurse` uses **obeys** clauses to reason about trust explicitly. For the reasons described above, `ValidPurse` cannot make absolute statements about trust, but can support relative, hypothetical statements. Consider the request

`res=dest.deposit(amt, src)`

If the destination purse accepts the deposit, then we would like to deduce that it has been able to retrieve the funds from the source purse, and so assert the *absolute* statement that

`res` $\rightarrow$ `src` **obeys** `ValidPurse`

Unfortunately, the `ValidPurse` specification only applies if the receiver `dest` is trustworthy: we can get only as far as the *conditional* conclusion

`res` $\wedge$ `dest` **obeys** `ValidPurse` $\rightarrow$ `src` **obeys** `ValidPurse`

meaning that, if the `deposit` method returns true, then we can trust `src` if we were willing to trust `dest`. So, if an amount is deposited successfully into a trustworthy destination `ValidPurse`, that purse vouches that the `src` is itself trustworthy. To quote [28]: "*A reported successful deposit can be trusted as much as one trusts the purse one is depositing into*".

`Pol_sprout`, the third policy, is basically the same as the earlier version in Figure 1, except that the first postcondition now is slightly weaker, as it only promises that the result is a trusted purse, without guaranteeing which class it belongs to. We also have the additional framing rule about $\mathcal{M}ay\mathcal{A}ccess$.

The fourth policy, `Pol_can_trade_constant`, guarantees that whether two purses can trade with each other can *never* change, no matter what code is run. This is another key ingredient of our approach: we can require that execution of any unknown code linked with our code must preserve some properties.

Finally, the fifth policy, `Pol_protect_balance`, delimits the risk involved with the purses. This policy guarantees that a valid purse `p`'s balance can only be changed ("$\mathcal{M}ay\mathcal{A}ffect$`(o,p.balance)`") by some object `o` that may access that purse ("$\mathcal{M}ay\mathcal{A}ccess$`(o,p)`").

```
1  class Purse {
2    private myMint;
3    Purse(aMint){ myMint = aMint);
4    method sprout { return myMint.newPurse(0) }
5    method deposit(source, amount){
6       return myMint.deposit(self,amount,source)
7    }
8  }
9  class Mint {
10   private ledger = new HashMap;
11
12   method makePurse(balance) {
13      p = Purse(self)
14      ledger.put(p,balance)  }
15   }
16   method deposit(into,amount,from) {
17      if ( (amount >=0)
18               && ledger.contains(from)
19               && ledger.contains(into)
20               && ledger.get(from) > amount) )
21      then {
22        ledger.put(from,ledger.get(from)−amount)
23        ledger.put(into,ledger.get(into)+amount)
24      } else { return false }
25   }
26  }
```

Figure 5. An implementation of Mint and Purse

3.4 Implementing `ValidPurse`

Figure 5 shows a `Purse` class that meets the `ValidPurse` specification, and its associated `Mint` class, to which `Purse` delegates its behavior. (Note `methods` are public, classes, types and specifications are Uppercased and objects lowercased.) A `Mint` is key to the security of all its purses: anyone with access to a `Mint` can create money "out of thin air" by calling `newPurse`, so access to mints must be carefully controlled. This is another reason why the `Escrow` must assess the trustworthiness of the purses without recourse to the mint.

On the other hand, `Purses` can be passed around without affecting the total money held by all the purses in the mint. Each mint has a `ledger` that records the balance of its purses, and a `deposit` method that transfers currency between purses. A `deposit` request will return true only if both purses are listed in the mint's `legder` and the source purse has sufficient funds.

We present this implementation to illustrate two key points about the `ValidPurse` specification. First, this implementation shows the key trust property of the `ValidPurse` specification: that if a request like `dest.deposit`$(0, $`src`$)$ returns true, then the `dest` purse has effectively vouched that the `src` purse can be trusted. In this implementation, purses only trust other purses from the same mint: as all purses are listed in their mint's ledger, a transfer validates the source purse by testing that it is listed in the same ledger as the destination purse. Second, there can be many different families of purses and mints in the system, both from this

```
1  method dealV2(  ) // returns Boolean
2  {
3    //setup and validate Money purses
4    escrowMoney = sellerMoney.sprout
5    res=escrowMoney.deposit(0, sellerMoney)
6    if (!res) then {return false}
7    res = buyerMoney.deposit(0, escrowMoney)
8    if (!res) then {return false}
9    res = escrowMoney.deposit(0, buyerMoney)
10   if (!res) then {return false}
11
12   //setup and validate Goods purses
13   escrowGoods = buyerGoods.sprout
14   res=escrowGoods.deposit(0, buyerGoods)
15   if (!res) then {return false}
16   res = sellerGoods.deposit(0, escrowGoods)
17   if (!res) then {return false}
18   res = escrowGoods.deposit(0, sellerGoods)
19   if (!res) then {return false}
20
21   res = escrowMoney.deposit(price, buyerMoney)
22   if (!res) then {return false}
23   res = escrowGoods.deposit(amt, sellerGoods)
24   if (!res) then {
25     buyerMoney.deposit(price, escrowMoney)
26     return false}
27
28   sellerMoney.deposit(price, escrowMoney)
29   buyerGoods.deposit(amt, escrowGoods)
30
31   return true
32 }
```

Figure 6. Revised `Escrow` method

and other implementations of the `ValidPurse` specification. In an open system, we cannot expect a central authority to know which are trustworthy and which are not.

3.5 Establishing Mutual Trust

The key to successful swapsies — or any other trading — is establishing just enough mutual trust for just long enough for the two parties to be able to complete the transaction. We have argued that a call like:

```
res1=dest.deposit(amt, src)
```

lets us conclude that

$$\texttt{res1} \wedge \texttt{dest}\ \textbf{obeys}\ \texttt{ValidPurse} \rightarrow \texttt{src}\ \textbf{obeys}\ \texttt{ValidPurse}$$

This trust is just one way: from the destination to the source purse. [32] offers a key insight: we can establish mutual trust between two purses by attempting a second deposit in the reverse direction:

```
res2=src.deposit(amt, dest)
```

which gives

$$\texttt{res2} \wedge \texttt{src}\ \textbf{obeys}\ \texttt{ValidPurse} \rightarrow \texttt{dest}\ \textbf{obeys}\ \texttt{ValidPurse}$$

Reasoning conditionally, on a path where $\texttt{res1} \wedge \texttt{res2}$ are true, we'll have established mutual trust:

$$\texttt{dest}\ \textbf{obeys}\ \texttt{ValidPurse} \longleftrightarrow \texttt{src}\ \textbf{obeys}\ \texttt{ValidPurse}$$

As with much of our reasoning, this is both conditional and hypothetical: at a particular code point, when two `deposit` requests have succeeded (or rather, that they have both *reported* success) then we can conclude that either both are trust worthy, or both are untrustworthy: we have only *hypothetical* knowledge of the **obeys** predicate.

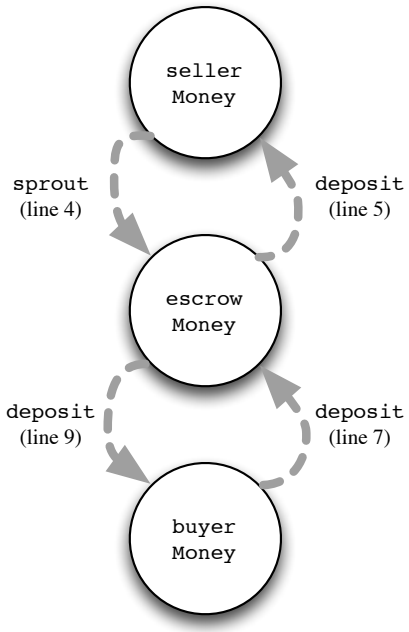

Figure 7. Establishing Mutual Trust

3.6 Escrow with Explicit Mutual Trust

Two way deposit calls are sufficient to establish mutual trust, but come with risks. For example, as part of validating that a buyer's money purse mutually trusts the seller's money purse, we must pass the buyer's purse as argument in a `deposit` call to the seller's money purse, e.g.

```
sellerMoney.deposit(0, buyerMoney)
```

If the seller's purse is not in fact trustworthy, then it can take this opportunity to steal all the money in the buyer's purse before the transaction officially starts, even if the `amt` that is supposed to be deposited is `0`.

We can minimise this risk by careful use of escrow purses. Rather than mutually validating buyers and sellers directly, we can create an escrow purse on the destination side of the transaction (the seller's money and the buyer's goods) and then mutually validate the buyer's and sellers actual purses against the escrow — resulting in a chain of mutual trust between the destination purse and the escrow purse, and the escrow purse and the source purse. This allows us to hypothesise that the source and destination purses are mutually trusting before we start on the transaction proper.

The resulting escrow method is in Figure 6. Line 4 creates an `escrowMoney` purse and then lines 5–10 hypothetically establish mutual trust between the `escrowMoney`, `sellerMoney`, and `buyerMoney` purses. Figure 7 illustrates the trust relationship: After line 4, we have

$$\texttt{sellerMoney}\ \textbf{obeys}\ \texttt{ValidPurse} \rightarrow$$

`escrowMoney` **obeys** `ValidPurse`, [1]

and after line 6, we have

`escrowMoney` **obeys** `ValidPurse` →
`sellerMoney` **obeys** `ValidPurse`.

If any of these `deposit` requests fail, we abort.

Lines 13–19 do exactly the same, but for goods purses rather than money purses. Finally, lines 21–31 carry out the escrow exchange itself, in exactly the same manner as lines 8–27 of the first escrow implementation in Figure 3.

3.7 Specifying the Mutual Trust Escrow

Figure 8 shows a specification for the revised escrow deal method from Figure 6. Whereas our original specification in Figure 2 consisted of two cases based on the value of the result, our revised `ValidEscrow` specification distinguishes four cases, based on the value of the result, *as well as* the trustworthiness of the participants. We use these auxiliary definitions:

```
GoodPrs= { p | p obeys_pre ValidPurse }
PPrs= { sellerMoney, sellerGoods, buyerMoney,
    buyerGoods }
OthrPrs=GoodPrs \ PPrs
BadPPrs=PPrs \ GoodPrs
```

The set `PPrs` contains the four "participant purses" passed as arguments. `BadPPrs` contains the untrustworthy participant purses. `GoodPrs` are all trustworthy purses in the system that do conform to the `ValidPurse` specification, and `OthrPrs` are the trustworthy purses that do *not* participate in this particular deal.

We now discuss the four cases of the policy

1^{st} case: The result is `true` and all participant purses are trustworthy. Then, the goods and money purses were from the same mints respectively, and there was sufficient money in the buyer's purse and sufficient goods in the sellers purse. In this case, everything is fine, so we can play swapsies: `price` will have been transferred from the buyer's to the seller's money purse, and `amt` will have been transferred from the seller's to the buyer's goods purse. No risk arises: no other purses' balance will change (whether passed in to the method or not).

2^{nd} case: The result is `false` and all participant purses are trustworthy. Then one or more of the functional correctness conditions are not satisfied: purses' mints did not match appropriately, or input purses did not have sufficient balance. Again, no risk arises to any purses.

3^{rd} case: The result is `false` and some participant purse is untrustworthy. In this case, no trustworthy purses' balances have been changed — unless they were already accessible by an untrustworthy purse passed in to the method.

4^{th} case: The result is `true` and some participant purse is untrustworthy — actually at least two matching participant purses are untrustworthy. That is, the pair of matching purses have coöperated to suborn the escrow *and we cannot tell*. Therefore, either both money purses are untrustworthy, (as per line 41), or both goods purses are untrustworthy, (as per line 42), or all four are bad.

The risk is that an uninvolved trustworthy purse's balance can be changed if it was previously accessible from a bad purse.

The first and second case correspond to the traditional `Escrow` specification in Figure 2, because traditional specifications assume all objects are trustworthy.

Discussion The 3^{rd} and 4^{th} case represent more of a risk than we would like: ideally (as in the 2^{nd} case) we'd hope nothing should have changed. But an escrow method cannot undo a system that is already suborned — if one of the participant purses is already benefiting from a security breach, passing that purse in to this method gives it an opportunity to exercise that breach. On the other hand, the risk is contained: this method cannot make things worse.

The 4^{th} case does not prevent trustworthy participant purses from being modified, to cater e.g., for the possibility that the two money purses are trustworthy, while the two goods purses are not, in which case the money transaction will take place as expected, while all bets are off about the goods transaction. We can give the stronger guarantee for the 3^{rd} case, because by the time the escrow starts making non-0 transactions it has established that the purses in each pair are either both trustworthy or both not trustworthy.

Most importantly, and perhaps surprisingly, the return value of the method, `res`, does *not* indicate whether the participants were trustworthy or not. Namely, a `true` result may be obtained in the 1^{st} case (all purses trustworthy) as well as the 4^{th} (some purses are untrustworthy). The return value indicates *only* whether the escrow attempted to complete the transaction (returning `true`) or abort (returning `false`). This came indeed as a surprise to us, as well as the original developers of the `deal` method. As with much of our reasoning around trust, this leads to yet more conditional reasoning, which must be interpreted hypothetically.

Nevertheless, the return value does communicate a valuable guarantee to an honest participant, whose money and goods purses are both trustworthy: If `deal` returns `true`, then the swap has taken place. Furthermore if it returns `false`, the swap has not taken place and with *no* more risk to the honest purses than those that existed before the call.

This `ValidEscrow` specification also gives a guarantee to other purse objects, who do *not* necessarily take part in the deal. Namely, if the participants had no prior access to these purses, then even if those participants were dishonest, the purses' balance can never be affected.

Reasoning We have 16 pages of hand-developed proof that the code from Figure 6 adheres to the specification from Figure 8. Some of the arguments used in that proof are discussed at the end of section 4.

[1] We don't need the `sellerMoney` purse to validate the `escrowMoney` purse explicitly because the `sprout` method specification says sprouted purses can be trusted as much as their parent purses.

```
specification ValidEscrow(e) {
   fields sellerMoney, sellerGoods, buyerMoney, buyerGoods //  Purse−s
   fields price, amt   // ℕ

 policy Pol_deal_1    //   1st case:
   price,amt∈ ℕ ∧ price,amt>0
       { e.deal(  ) }
    res ∧ BadPPrs=∅  → (
      // FUNCTIONAL SPECIFICATION
      buyerMoney.balance=buyerMoney.balance_pre−price ∧sellerMoney.balance=sellerMoney.balance_pre+price∧
      buyerGoods.balance=buyerGoods.balance_pre+amt ∧ sellerGoods.balance=sellerGoods.balance_pre−amt ∧
      // RISK
      ∀p:_preOthrPrs. p.balance=p.balance._pre  ∧
      ∀o:_preObject,p:_preGoodPrs. MayAccess(o,p) → MayAccess(o,p)_pre  )

 policy Pol_deal_2    //   2nd case:
    price,amt∈ ℕ ∧ price,amt>0
        { e.deal(  ) }
    ¬res ∧ BadPPrs=∅  → (
       // FUNCTIONAL SPECIFICATION
       ¬( CanTrade(buyerMoney,sellerMoney) ∧  CanTrade(buyerGoods,sellerGoods) ∧
         buyerMoney.balance_pre ≥ price ∧ sellerGoods.balance_pre ≥ amt ) ∧
      // RISK
      ∀p:_preGoodPrs. p.balance=p.balance._pre   ∧
      ∀o:_preObject,p:_preGoodPrs.  MayAccess(o,p) → MayAccess(o,p)_pre  )

 policy Pol_deal_3    //   3rd case:
    price,amt∈ ℕ ∧ price,amt>0
        { e.deal(  ) }
    ¬res ∧ BadPPrs≠∅  →  (
       //RISK
       ∀p:_preGoodPrs. ( p.balance=p.balance._pre ∨ ∃ bp∈BadPPrs_pre.MayAccess(bp,p)_pre ∧
       ∀o:_preObject,p:_preGoodPrs.MayAccess(o,p)→(MayAccess(o,p)_pre∨∃b∈BadPPrs_pre.MayAccess(b,p)_pre))


 policy Pol_deal_4    //   4th case:
    price,amt∈ ℕ ∧ price,amt>0
        { e.deal(  ) }
    res ∧ BadPPrs≠ ∅    →  (
      // TRUST
      buyerMoney obeys PurseSpec ⟷ sellerMoney obeys PurseSpec  ∧
      buyerGoods obeys PurseSpec ⟷ sellerGoods obeys PurseSpec  ∧
      //RISK
      ∀p:_preOthrPrs. ( p.balance=p.balance._pre ∨ ∃ bp∈ BadPPrs_pre.MayAccess(bp,p)_pre   ∧
       ∀o:_preObject,p:_preGoodPrs.MayAccess(o,p)→
           (MayAccess(o,p)_pre∨∃b∈BadPPrs_pre.MayAccess(b,p)_pre))
}
```

Figure 8. `ValidEscrow` specification

Stength of Specification The risk specification is weaker than we would have liked. We currently guarantee that a preëxisiting purse's balance will remain unaffected, unless a rogue purse had access to it before the call. A guarantee that the balance is unaffected, unless a rogue purse could affect the balance before the call would be stronger, and more useful. Unfortunately, the current specification of `ValidPurse` is too weak for this guarantee to be implementable. Namely, imagine a "gullible" object, which has access to some third, "victim" purse and which, when given a purse which the victim can trade with, will transfer moneys from the victim. Then, a rogue purse could pass the escrow purse with `0` balance to the gullible, and thus remove moneys from the victim. We expect to tackle this problem by applying notions of encapsulation.

4. Formal Definitions — a Sketch

In this section, we sketch the most salient features of the formal underpinnings of our system; we leave the full exposition to further work.

Underlying Programming and Specification Language We assume a small object oriented language, $\mathcal{FOCaL}$ Therefore (Featherweight Object Capability Language, not to be confused with FOCAL [24]) which supports classes, fields and methods. $\mathcal{FOCaL}$ is memory-safe: it does not al-

low addresses to be forged, or non-existent methods or fields to be called, read or written. $\mathcal{FOCAL}$ is dynamically typed: it does not check that the arguments to a method call or a field write are of the appropriate type either statically or dynamically: in this sense, $\mathcal{FOCAL}$ is inspired by JavaScript, E, and Dart's unchecked mode.

$\mathcal{FOCAL}$ supports modules, M, which are mappings from class identifiers to class definitions. The module linking operator $*$ combines these definitions, provided that the modules' mappings have separate domains, and performs no other checks. This reflects the open world setting, where objects of different provenance interoperate without a central authority. For example, if the mint and purse module (Figure 7) is M_{mp} and the escrow module (Figure 6) is M_e, then $M_{mp} * M_e$ is defined but $M_e * M_e$ is not.

$\mathcal{FOCAL}$ dynamically enforces `private` fields and methods. Accessing or calling private fields or methods is only allowed from method bodies of the same class; if not, the exception `error` is thrown. We model private fields as they are simpler than nested lexical scopes.

The operational semantics of $\mathcal{FOCAL}$ has the shape

$$\mathtt{M},\ \kappa,\ \mathtt{code} \rightsquigarrow \kappa',\ r,$$

where `M` is a module containing all class declarations used, κ, κ' are runtime configurations, `code` is some code in the syntax of $\mathcal{FOCAL}$, and *r* is a result. Results are addresses, or the exception `error`.

Paths are written as p. They start with the receiver **this**, or the formal parameter **x**, followed by a sequence of field identifiers (`f`). We define $\lfloor p \rfloor_\kappa$, the lookup of a path p in a context κ in the expected way, where we read the receiver or argument from the frame, and follow the values of the fields in the heap. Therefore, if execution of a path is defined, then looking up that path in the same configuration will return the same value, i.e., $\mathtt{M},\ \kappa,\ p \rightsquigarrow \kappa,\ \mathtt{v}$ implies $\lfloor p \rfloor_\kappa = \mathtt{v}$.

We assume an underlying specification language with assertions indicated by *P*. Validity of these assertions is expressed through the judgment $\mathtt{M}, \kappa \models P$. We also expect support for the assertion p:`ClassId` which expresses that path p is pointing to an object of class `ClassId`. When writing method specifications, we need to compare properties of the state before with properties of the state after method call. For this, we use annotations pre, or post and write two-state assertions, whose validity has the form $\mathtt{M}, \kappa, \kappa' \models P$.

If $\lfloor \mathtt{x.balance} \rfloor_\kappa = 4$, and $\lfloor \mathtt{x.balance} \rfloor_{\kappa'} = 14$, then $\mathtt{M_{mp}}, \kappa \models \mathtt{x.balance} < 10$, and $\mathtt{M_{mp}}, \kappa, \kappa' \models \mathtt{x.balance} = \mathtt{x.balance}_{pre} + 10$. Also, if $\mathtt{M_{mp}}, \kappa_1 \models x : \mathtt{Purse}$, and execution of $y = x.\mathtt{sprout}()$ leads to configuration κ_2, then we expect that $\mathtt{M_{mp}}, \kappa_2 \models y : \mathtt{Purse}$.

Hypothetical Access and Affect We expand the specification language with the special predicates *MayAffect*, and *MayAccess* which we use to model risk.

Definition 1 (*MayAffect* and *MayAccess*)**.**
We expand the definition of assertions with the predicates MayAffect(_, _), and MayAccess(_, _). We define their validity as follows:

- $\mathtt{M}, \kappa \models \mathit{MayAccess}(\mathtt{p}, \mathtt{p}')$ *iff*
  $\exists$ *fields* $\mathtt{f}_1 ... \mathtt{f}_n.\ \lfloor \mathtt{p.f}_1 ... \mathtt{f}_n \rfloor_\kappa = \lfloor \mathtt{p}' \rfloor_\kappa$
- $\mathtt{M}, \kappa \models \mathit{MayAffect}(\mathtt{p}, \mathtt{p}')$ *iff*
  $\exists$ *public method* $\mathtt{m}$, *paths* $, \bar{\mathtt{p}}$, *and configuration* κ' :
  $M, \kappa, \mathtt{p.m}(\bar{\mathtt{p}}) \rightsquigarrow \_, \kappa'$ *and* $\lfloor \mathtt{p}' \rfloor_\kappa \neq \lfloor \mathtt{p}' \rfloor_{\kappa'}$.

We expand validity of *MayAccess* and *MayAffect* to two state assertions, *e.g.,* $\mathtt{M}, \kappa, \kappa' \models \mathit{MayAccess}(\mathtt{p}, \mathtt{p}')_t$ iff $\mathtt{M}, \kappa'' \models \mathit{MayAccess}(\mathtt{p}, \mathtt{p}')$, where $\kappa'' = \kappa$ if *t*=pre, else $\kappa'' = \kappa'$.

Arising Runtime Configurations To give meaning to our policies, it is essential to examine only those runtime contexts (i.e. configuration and code pairs) which may arise through the execution of the given modules.

We therefore define *Arising*(`M`) as the set of runtime contexts which may be reached during execution of some initial context (κ_0,$\mathtt{code}_0$). A context is initial if its heap contains only objects of class `Object`, its stack contains only one frame, and the code contains exclusively method calls of methods defined in M. The set $\mathcal{R}each(\mathtt{M}, \kappa, \mathtt{e})$ collects all contexts at the start of any method call during execution of $\kappa, \mathtt{e}$ — as in visible states.

Definition 2.

$$\begin{array}{lcl} \mathit{Arising} & : & \mathit{Program} \longrightarrow \mathcal{P}(\mathit{Configuration} \times \mathit{Expr}) \\ \mathit{Arising}(\mathtt{M}) & = & \bigcup_{(\kappa,\mathtt{code}) \in \mathcal{I}nit(\mathtt{M})} \mathcal{R}each(\mathtt{M}, \kappa, \mathtt{code}) \end{array}$$

Policies and Specifications Policies have one of the three following forms: a) invariants of the form P, which require that some property holds at all visible states of a program; b) Hoare-logic-like triples, $P\,\{\mathtt{code}\}\,P'$ where P must be a one-state assertion, and which require that execution of `code` in any state which satisfies P will lead to a state which satisfied P'; c) $P\,\{\mathtt{any\_code}\}\,P'$ which, like two state invariants require that execution of *any* code in a state which satisfies P will lead to a state which satisfies P'.

Definition 3 (Policies)**.**

$$\mathit{Policy} \quad ::= \quad P \mid P\,\{\mathtt{code}\}\,P \mid \ P\,\{\mathtt{any\_code}\}\,P$$

Weak adherence$M, \kappa \models_{weak}$ *Policy*, ensures that the requirements of *Policy* are satisfied if κ arises from M.

Definition 4 (Weak Adherence to Policies)**.**

- $M, \kappa \models_{weak} P$ *iff*
  $(\kappa, \_) \in \mathit{Arising}(M) \ \rightarrow \ M, \kappa \models P$
- $M, \kappa \models_{weak} P\,\{\mathtt{code}\}\,P'$ *iff*
  $(\kappa, \mathtt{code}) \in \mathit{Arising}(M) \ \wedge \ M, \kappa \models P$
  $\wedge \ \ M, \kappa, \mathtt{code} \rightsquigarrow \mathtt{res}, \kappa'$
  $\rightarrow \quad M, \kappa, \kappa' \models P'$
- $M, \kappa \models_{weak} P\,\{\mathtt{any\_code}\}\,P'$ *iff*
  $(\kappa, \mathtt{code}) \in \mathit{Arising}(M) \ \wedge \ M, \kappa \models P$
  $\wedge \ \ M, \kappa, \mathtt{code} \rightsquigarrow \mathtt{res}, \kappa'$
  $\rightarrow \quad M, \kappa, \kappa' \models P'$

For example, taking any κ, and taking M_{mp} as before, we obtain that $M_{mp}, \kappa \models_{weak} \mathtt{Pol\_protect\_bal}$. Taking

an M'_{mp} similar to M_{mp} but where instead of the ledger, the `Mint` kept a map from `Purses` to codes, and another map from codes to balances, and where the latter map was public, we would also obtain that $M'_{mp} \models_{weak}$ `Pol_protect_bal`. This is so because none of the methods offered by M'_{mp} modify a `Purse` without access to it. However, clearly M'_{mp} is not robust enough. We have not yet taken the open nature of systems into account.

In order to model open systems, we require that after linking *any* module with the module at hand, the policy will be satisfied. As stated in [30], "A programmer should be able to prove that his programs have various properties and do not malfunction, solely on the basis of what he can see from his private bailiwick." For example, to express that $\mathtt{M}_e$ satisfies `EscrowSpec` we need to allow any possible implementation of `Purse` as well as any other code to be linked, and still ensure that the policies from figure 8 are satisfied.

Definition 5 (Strong Adherence to Policies)**.**

- $M \models \mathit{Policy}$ *iff*
  $\forall M', \kappa.$
  $M * M'$ *is defined* $\wedge\ (\kappa, \_) \in \mathcal{A}\mathit{rising}(M * M')$
  $\rightarrow\ M * M', \kappa \models_{weak} \mathit{Policy}$

Therefore, $M \models \mathit{Policy}$ not only ensures that execution of M will guarantee *Policy*, but also, that the code of M is so robust, that any further other module linked with M cannot break *Policy*. With this definition, we obtain that $M_{mp} \models$ `Pol_protect_bal`, but $M'_{mp} \not\models$ `Pol_protect_bal` – as expected .

Policy Specifications and obeys assertion A policy specification consists of a name, a parameter, and a set of policies.

Definition 6 (Policy Specifications)**.**

$\mathit{PolicySpec}\ ::=\ \mathtt{specification}\ \mathit{PolSpecId}(\mathit{PId})$
$\{\ \mathit{Policy}^*\ \}$

An object o satisfies a *PolSpecId* in a configuration κ, if it satisfies all the *Policy*s from *PolSpecId*. A class satisfies a *PolSpecId* if all objects of that class are guaranteed to satisfy the policies from that specification.

Definition 7 (Adherence to Policy Specifications)**.** *Assuming a PolSpecId defined by*
`specification` *PolSpecId* (*PId*)$\{\ \mathit{Policy}_1, ...\mathit{Policy}_n\ \}$, *then*

- $M, \kappa \models o$ ***obeys*** *PolSpecId iff*
  $\forall i \in \{1..n\}.\forall M'.$
  $M * M'$ *defined* $\rightarrow\ M * M', \kappa \models_{weak} \mathit{Policy}_i[o/\mathit{PId}]$
- $M \models$ *ClassId* ***obeys*** *PolSpecId iff*
  $\forall i \in \{1..n\}.\forall M'.$
  $M * M'$ *defined* $\rightarrow$
  $M * M', \kappa \models_{weak} o : \mathit{ClassId} \rightarrow \mathit{Policy}_i[o/\mathit{PId}]$

Thus, we want to have $\mathtt{M}_{mp} \models$ `Mint` **obeys** `ValidPurse`, and $\mathtt{M}_e \models$ `Escrow` **obeys** `ValidEscrow`.

Validity of `Escrow :: deal`***, Reasoning about Accessibility*** Reasoning about accessibility is central to arguing that `Escrow :: deal` satisfies its risk specification. The argument hinges, essentially, on the observation that if any pre-existing purse's money were to be affected by the call of `Escrow :: deal`, then one of the parties must have had access to that purse at some time during execution of the call (policy `Pol_protect_balance`). However, since neither the escrow, nor `ValidPurses` grant to any objects access to any pre-existing purses, the only way for a malicious participant to have access to a pre-existing purse, is if it already had the access before the call of `Escrow :: deal`.

We can reason about accessibility based on one of the underlying properties of object-capability systems, that "*only connectivity begets connectivity*" [26]. Namely, object references can only be created through object creation, and cannot be forged. Thus the only way one object o can get access to another object o' is if o creates o'; or if o' is passed to o as a method argument in a method call; or if both o and o' are passed as arguments in a method call; or if o' is returned from a method call to o. In all previous assertions, the term "o passed" is a shorthand for "some object o'' with access to o is passed", and similar for o'.

This is expressed through the following Hoare-logic connectivity rule:

ConnRule_MethCall:

$$\frac{\{\ \mathit{true}\ \}}{\mathtt{x.m(y)}}$$

$\{\ \forall z, z' :_{pre} \mathit{Object}.$
$(\ \ \mathcal{M}\mathit{ayAccess}_{post}(z, z') \wedge \neg \mathcal{M}\mathit{ayAccess}_{pre}(z, z') \rightarrow$
$[(\mathcal{M}\mathit{ayAccess}(\mathtt{x}, z)_{pre} \vee \mathcal{M}\mathit{ayAccess}(\mathtt{y}, z)_{pre}) \wedge$
$(\mathcal{M}\mathit{ayAccess}(\mathtt{x}, z')_{pre} \vee \mathcal{M}\mathit{ayAccess}(\mathtt{y}, z')_{pre})]\ \ )\ \}$

We have similar rules for field and variable assignment, which we omit here. The rules are independent of the trustworthiness of `x` and `y`. Soundness follows from memory safety of the underlying programming language. A stronger version of **ConnRule_MethCall**, to be used for concurrency, guarantees that accessibility restrictions hold throughout the method's execution and not just at the post-state.

5. Related Work

Object Capabilities and Sandboxes. *Capabilities* as a means to support the development of concurrent and distributed system were developed in the 60's by Dennis and Van Horn [10], and were adapted to the programming languages setting in the 70's [30]. *Object capabilities* were first introduced [26] in the early 2000s, and many recent studies manage or verify safety or correctness of object capability programs. Google's Caja [29] applies sandboxes, proxies, and wrappers to limit components' access to *ambient* authority. Sandboxing has been validated formally: Maffeis et al. [23] develop a model of JavaScript, demonstrate that it obeys two principles of object capability systems and show how untrusted applications can be prevented from interfering with the rest of the system.

JavaScript analyses. More practically, Karim et al. apply static analysis on Mozilla's JavaScript Jetpack extension framework [20], including pointer analyses. Bhargavan et al. [5] extend language-based sandboxing techniques to support "defensive" components that can execute successfully in otherwise untrusted environments. Politz et al. [34] use a JavaScript type checker to check properties such as *"multiple widgets on the same page cannot communicate."* Lerner et al. extend this system to ensure browser extensions observe *"private mode"* browsing conventions, such as that *"no private browsing history retained"* [22]. Dimoulas et al. [11] generalise the language and type checker based approach to enforce explicit policies, that describe which components may access, or may influence the use of, particular capabilities. Alternatively, Taly et al. [39] model JavaScript APIs in Datalog, and then carry out a Datalog search for an "attacker" from the set of all valid API calls. The problem posed by the Escrow example is that it establishes a two-way dependency between trusted and untrusted systems — precisely the kind of dependencies these techniques prevent.

Concurrent Reasoning Deny-Guarantee [12] distinguishes between assertions guaranteed by a thread, and actions denied to all other threads. Deny properties correspond to our requirements that certain properties be preserved by all code linked to the current module. Compared with our work, deny-guarantee assumes coöperation: composition is legal only if threads adhere to their deny properties. In our work, a module has to be robust and ensure that these properties cannot be affected by other code.

Relational models of trust. Artz and Gil [4] survey various types of trust in computer science generally, although trust has also been studied in specific settings, ranging from peer-to-peer systems [2] and cloud computing [18] to mobile ad-hoc networks [9], the internet of things [17], online dating [33], and as a component of a wider socio-technical system [8, 41]. Considering trust (and risk) in systems design, Cahill et al.'s overview of the SECURE project [6] gives a good introduction to both theoretical and practical issues of risk and trust, including a qualitative analysis of an e-purse example. This project builds on Carbone's trust model [7] which offers a core semantic model of trust based on intervals to capture both trust and uncertainty in that trust. Earlier Abdul-Rahman proposed using separate relations for trust and recommendation in distributed systems [1], more recently Huang and Nicol preset a first-order formalisation that makes the same distinction [19]. Solhaug and Stølen [38] consider how risk and trust are related to uncertainties over actual outcomes versus knowledge of outcomes. Compared with our work, these approaches produce models of trust relationships between high-level system components (typically treating risk as uncertainty in trust) but do not link those relations to the system's code.

Logical models of trust. A detailed study of how web-users decide whether to trust appears in [16]. Starting with [21], various different logics have been used to measure trust in different kinds of systems. Murray and Lowe [31] model object capability programs in CSP, and use a model checker to ensure program executions do not leak authority. Carbone et al. [37] use linear temporal logic to model specific trust relationships in service oriented architectures. Ries et al. [36] evaluate trust under uncertainty by evaluating Boolean expressions in terms of real values for average rating, certainty, and initial expectation. Aldini [3] describes a temporal logic for trust that supports model checking to verify some trust properties. Primiero and Taddeo [35] have developed a modal type theory that treats trust as a second-order relation over base relations between counterparties. Merro and Sibilio [25] developed a trust model for a process calculus based on labelled transition systems. Compared with our proposal, these approaches use process calculi or other abstract logical models of systems, rather than engaging directly with the system's code.

Formal Verification of Object Capability Programs. Drossopoulou and Noble [13, 32] have analysed Miller's Mint and Purse example [26] by expressing it in Joe-E and in Grace [32], and discussed the six capability policies as proposed in [26]. In [15], they proposed a complex specification language, and used it to fully specify the six policies from [26]; their formalisation showed that several possible interpretations were possible. They also uncovered the need for another four policies and formalised them as well. Most recently, [14] they have shown how different implementations of the underlying Mint and Purse systems coexist with different policies. In contrast, this work proposes $\mathcal{FOCaL}$, which is untyped and modelled on JavaScript rather than Java; a much simpler specification language; the **obeys** predicate to model trust; clearer definitions of accessibility predicates to model risk; a full specification of the `Escrow`; and sketches the reasoning steps using the proposed predicates.

6. Conclusions and Further Work

In this paper we addressed the questions of specification of risk, trust, and reasoning about such specifications. To answer these questions, we proposed:

- *Hypothetical* predicates to describe who may access or modify an object or a property,
- *Obeys* predicates to describe whether an object can be trusted to satisfy some specification,
- *Open Assertions* whose validity must be guaranteed in spite of the execution of *any* code,
- *Open Policies* whose validity holds in the presence of any code linked to the current code.
- *Conditional* reasoning steps.

During this work, we considered encapsulation as a low-level mechanism, which should be transparent to specifications. To our surprise we have observed multiple times that unless encapsulation percolates to the specification level, specifications end up being too weak and wordy.

In further work we will re-consider encapsulation, and we will complete the formal model and prove soundness of the rules for reasoning. We will also extend our approach to deal with concurrency and distribution.

Acknowledgments

We gratefully thank Toby Murray for crucial feedback, and the Programming Language group at VUW, the SLURP group at Imperial College, the PLAS reviewers, the IFIP WG2.3 and WG2.16 members for valuable discussions. This work is supported in part by the Royal Society of New Zealand Marsden Fund, a James Cook Fellowship, and the EU project Upscale.

References

[1] A. Abdul-Rahman and S. Halles. A distributed trust model. In *New Security Paradigms Wkshp.*, 1988. Langdale, Cumbria.

[2] K. Aberer and Z. Despotovic. Managing trust in a peer-2-peer information system. In *CKIM*, 2001.

[3] A. Aldini. A calculus for trust and reputation systems. In *IFIPTM*, 2014.

[4] D. Artz and Y. Gil. A survey of trust in computer science and the semantic web. *Journal of Web Semantics*, 2007.

[5] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security*, 2013.

[6] Cahill et al. . Using trust for secure collaboration in uncertain environments. *Pervasive Computing*, July 2003.

[7] M. Carbone, M. Nielsen, and V. Sassone. A formal model for trust in dynamic networks. In *SEFM*, 2003.

[8] J.-H. Cho and K. S. Shan. Building trust-based sustainable networks. *IEEE Tech. and Soc.*, Summer, 2013.

[9] J.-H. Cho, A. Swami, and I.-R. Chen. A survey on trust management for mobile ad hoc networks. *IEEE Comms. Surv. & Tuts.*, 13(4), 2011.

[10] J. B. Dennis and E. C. V. Horn. Programming Semantics for Multiprogrammed Computations. *Comm. ACM*, 9(3), 1966.

[11] C. Dimoulas, S. Moore, A. Askarov, and S. Chong. Declarative policies for capability control. In *Computer Security Foundations Symposium*, 2014.

[12] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*. Springer, 2009.

[13] S. Drossopoulou and J. Noble. The need for capability policies. In *FTfJP*, 2013.

[14] S. Drossopoulou and J. Noble. How to break the bank: Semantics of capability policies. In *iFM*, 2014.

[15] S. Drossopoulou and J. Noble. Towards capability policy specification and verification, May 2014. `ecs.victoria-.ac.nz/Main/TechnicalReportSeries`.

[16] Y. Gil and D. Artz. Towards Content Trust of Web Resources. *IWeb Semantics: Science, Services and Agents on the World Wide Web*, 2007.

[17] L. Gu, J. Wang, and B. Sun. Trust management mechsnism for internet of things. *China Communications*, Feb. 2014.

[18] S. M. Habib and M. M. Sebastian Ries and. Towards a trust management system for cloud computing. In *TrustCom*, 2011.

[19] J. Huang and D. M. Nicol. A formal-semantics-based calculus of trust. *IEEE INTERNET COMPUTING*, 2010.

[20] R. Karim, M. Dhawan, V. Ganapathy, and C.-C. Shan. An Analysis of the Mozilla Jetpack Extension FrameworK. In *ECOOP*, Springer, 2012.

[21] B. Lampson, M. Abadi, M. Burrows, and E. Wobbler. Authentication in Distributed Systems: Theory and Practice. *ACM TOCS*, (4):265–310, 1992.

[22] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *European Symposium on Research in Computer Security (ESORICS)*, Sept. 2013.

[23] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc of IEEE Security and Privacy*, 2010.

[24] R. Merrill. focal: new conversational language. DEC, 1969. `homepage.cs.uiowa.edu/~jones/pdp8/focal/-focal69.html`.

[25] M. Merro and E. Sibilio. A calculus of trustworthy ad hoc networks. *Formal Aspects of Computing*, page 25, 2013.

[26] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, Maryland, 2006.

[27] M. S. Miller, T. V. Cutsem, and B. Tulloh. Distributed electronic rights in JavaScript. In *ESOP*, 2013.

[28] M. S. Miller, C. Morningstar, and B. Frantz. Capability-based financial instruments: From object to capabilities. In *Financial Cryptography*. Springer, 2000.

[29] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Safe active content in sanitized JavaScript. `code.google.com/p/google-caja/`.

[30] J. H. Morris Jr. Protection in programming languages. *CACM*, 16(1), 1973.

[31] T. Murray and G. Lowe. Analysing the information flow properties of object-capability patterns. In *FAST*, LNCS, 2010.

[32] J. Noble and S. Drossopoulou. Rationally reconstructing the escrow example. In *FTfJP*, 2014.

[33] G. Norcie, E. D. Cristofaro, and V. Bellotti. Bootstrapping trust in online dating: Social verification of online dating profiles. In *Financial Cryptography and Data Security*, 2013.

[34] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. Adsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security*, 2011.

[35] G. Primiero and M. Taddeo. A modal type theory for formalizing trusted communications. *J. Applied Logic*, 10, 2012.

[36] S. Ries, S. M. Habib, M. M. Sebastian Ries and, and V. Varadharajan. Certainlogic: A logic for modeling trust and uncertainty. In *TRUST*, 2011. LNCS 6740.

[37] Roberto Carbone et al. Towards formal validation of trust and security in the internet of services. In *Future Internet Assembly*, 2001. LNCS 6656.

[38] Solhaug and Stølen. Uncertainty, subjectivity, trust and risk: How it all fits together. In *STM*, 2011.

[39] A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *SOSP*, 2011.

[40] The Swapsies. Got Got Need. In *5: A February Records Anniversary Compilation*. February Records, 2015.

[41] M. Walterbusch, B. Martens, and F. Teuteberg. Exploring trust in cloud computing: A multi- method approach. In *ECIS*, page 145, 2013.