Reasoning about Risk and Trust in an Open Word

Sophia Drossopoulou[1], James Noble[2], Mark S. Miller[3], and Toby Murray[4]

[1] Imperial College London `scd@doc.ic.ac.uk`
[2] Victoria University of Wellington `kjx@ecs.vuw.ac.nz`
[3] Google, Inc. `erights@google.com`
[4] NICTA and UNSW `toby.murray@nicta.com.au`

Abstract. Contemporary open systems use components developed by different parties, linked together dynamically in unforeseen constellations. Code needs to live up to strict *security requirements*, and ensure the correct functioning of its objects even when they collaborate with external, potentially malicious, objects. In this paper we propose special specification predicates that model risk and trust in open systems. We specify Miller, Van Cutsem, and Tulloh's escrow exchange example, and discuss the meaning of such a specification.
We propose a novel Hoare logic, based on four-tuples, including an invariant describing properties preserved by the execution of a statement as well as a postcondition describing the state after execution. We model specification and programing languages based on the Hoare logic, prove soundness, and prove the key steps of the Escrow protocol.

1 Introduction

Traditional systems designs are based on a closed world assumption: drawing a sharp border around a system where the system as a whole can be trusted because every component inside the border is known to be trustworthy, or is *confined* [25] by trustworthy mechanisms. Open systems, on the other hand, have an open world assumption: they must interact with a wide range of component objects with different levels of mutual trust (or distrust) — and whose configuration dynamically changes. Given a method request `x.m(y)`, what can we conclude about the behaviour of this request if we know nothing about the receiver `x`?

In this paper, we lay the foundations for reasoning about the correctness of these kinds of open systems. Building on the object-capability security model [30] we introduce a first-class notion of *trust*, where we write "`o` **obeys** `Spec`" to mean that object `o` can be trusted to obey specification `Spec`. The **obeys** predicate is hypothetical: there is no central authority that can assign trustworthiness (or not) to objects; there is no trust bit that we can test. Rather, "`o` **obeys** `Spec`" is an assumption that may or may not be true, and we will use that assumption to reason by cases. If we trust an object, we can use the object's specification `Spec` to determine the results of a method call on that object. If we don't trust the object, we determine the maximum amount of damage the call could do: the *risk* of calling a method that turns out not to meet its specification.

Risks are effects against which we want to guard our objects: bounds on the potential damage caused by calls to untrusted objects. The key to delineating risks are two further hypothetical predicates: $\mathcal{MayAccess}$ and $\mathcal{MayAffect}$. We write $\mathcal{MayAffect}$(`o,p`) to mean that it is possible that some method invocation on o would affect the object or property p, and $\mathcal{M}ay\mathcal{A}ccess$(o,p) to mean that it is possible that the code in object o could potentially gain a capability to access to p. This first-class notion of risk complements our first-class notion of trust: $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$ let us reason about the potential damage to a system when one or more objects are not trustworthy.

Our complementary notions of trust and risk are set within a very flexible specification language, and supported by an Hoare logic, enabling us to reason whether or not objects can be trusted to meet their specifications, providing sufficient security guarantees while mitigating any risks. Building on our earlier work [15, 17] we formalise and prove correctness, trust, and risk for the Escrow Exchange [31] a trusted third party that manages exchanges of different goods (e.g. money and shares) between untrusting counterparties [44]. We were surprised to find that the specification for the Escrow Exchange 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 a co-author is one of the original developers of the escrow example.

Common approaches to reasoning about programs cannot deal with the escrow exchange example. Most program specification and verification methods have an implicit underlying assumption that components are meant to be trustworthy (i.e. meet their specifications). Our approach first makes that assumption explicit (as **obeys**), lets us reason hypothetically and conditionally about those trust assumptions, even in cases where those assumptions fail (by quantifying risk via $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$).

Paper Organization Section 2 introduces the Escrow Exchange example, shows why a traditional specification is not descriptive enough and why a naive implementation is not robust enough. Section 3 introduces our constructs and Hoare logic for modelling trust and risk, which we apply to a revised implementation of the Escrow to reason formally about its correctness. Section 4 discusses related work, and Section 5 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. 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 further light on the questions we address.

Contribution This paper extends earlier informal work presented at the PLAS workshop [17]. Here we contribute the full formal foundations of the system, defining **obeys**, $\mathcal{M}ay\mathcal{A}ccess$, and $\mathcal{M}ay\mathcal{A}ffect$ in the context of the $\mathcal{F}ocal$ and $\mathcal{C}hainmail$ languages (details in the full technical report [18]). We present a novel Hoare logic based on fourtuples to specify properties preserved during execution: this allows us to model trust and delineate risk even when a method's receiver is unknown. We use our logic to prove formally that the key steps of the escrow example meet the specification.

2 Escrow Exchange

Figure 1 shows a first attempt to implement an escrow exchange, also shown in previous work [31, 36]. We model both money and goods by `Purses` (a resource model proposed in E [32]). The call `dst.deposit(amt, src)` will either transfer `amt` resources from the `src` purse to the `dst` purse and return true, or do nothing and return false. A new, empty purse can be created at any time by asking an existing purse to `sprout` — the new purse has a zero balance but can then be filled via `deposit`.

```
 1 method deal_version1( ) {
 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 goods transaction
21            buyerMoney.deposit(price,escrowMoney)
22            return false }
23
24    // price in escrowMoney; amt in escrowGoods.
25    // now complete the transaction
26    sellerMoney.deposit(price, escrowMoney)
27    buyerGoods.deposit(amt, escrowGoods)
28    return true
29 }
```

Fig. 1. First attempt at Escrow Exchange deal method

The goal of the escrow is to exchange `amt` goods for `price` money, between the purses of a seller and buyer. To make the exchange transactional, we use two private *escrow* purses, one for on each side of the transaction (money and goods). Lines 3–6 of Figure 1 show how we first set up the escrow purses, by sprouting two new purses (`escrowMoney` and `escrowGoods`) from their respective input purses.

It is important that the escrow purses are newly created within the method, and cannot have been manipulated or retained by the buyer or seller, which is why the escrow asks `sellerMoney` to make one, and `buyerGoods` to make the other. The requirements of an open system means that the escrow method cannot have the escrow purses before the transaction, because the escrow cannot know the right kind of purses to create, and there is no central trusted authority that could provide them. Buyers and sellers cannot provide escrows purses directly, precisely because we must assume they don't trust each other: if they did, they wouldn't need to use an escrow.

Second, we attempt to escrow the buyer's money by transferring it from the `buyerMoney` purse into the new `escrowMoney` purse — line 8. If this `deposit` request returns true, then the money will have been transferred. If the deposit fails we abort the transaction. Third, we attempt to escrow the seller's goods — line 16, again by depositing them into the other escrow purse. If we are unsuccessful, we again abort the transaction, after we have returned the escrowed money to the buyer — lines 21 and 22. At this point (line 26) 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, and indeed, if `deal_version1` is called in good faith it will carry out the transaction correctly. Unfortunately, we cannot assume good faith in a mutually untrusting open system.

2.1 The failure of `deal_version1`

The method in Figure 1 does not behave correctly in an open system. The critical problems are assumptions about trust: both the code and the specification implicitly trust the purse objects with which they interact.

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`. At line 8, `escrowMoney.deposit(price, buyerMoney)` would let the malicious `escrowMoney` purse steal all the money out of `buyerMoney` purse, and still return `false`. As a result, the seller would lose all their money, and receive no goods! Even if the seller was more cautious, and themselves sprouted a special temporary purse with a balance of exactly `price` to pass in as `sellerMoney`, 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 of expected behaviour.

2.2 Modelling Trust and Risk: obeys , $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$

The key claim of this paper is that, to reason about the behaviour of systems in an open world, we need specifications that let us talk about trust and risk explicitly. In the rest of this section, we informally introduce 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, show

how they can be used to specify the purse and escrow examples, and argue a revised `deal_version2` method can meet that specification. Section 3 formalises these ideas.

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 are 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.

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` including both public and private references.

2.3 Valid Purse: Specifying `Purse`

Using **obeys**, $\mathcal{M}ay\mathcal{A}ccess$, and $\mathcal{M}ay\mathcal{A}ffect$, we write the `ValidPurse` specification in Figure 2 that makes trust and risk explicit.

`ValidPurse` consists of five policies. `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 can 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. To quote Miller et al. [32]: "*A reported successful deposit can be trusted as much as one trusts the purse one is depositing into*".

The last two lines in the postcondition of `Pol_deposit_1` and `Pol_deposit_2` provide framing conditions. In the first case, the transaction will happen, but all other purses will be unmodified (line 14 in figure 2) , whereas in the second case no purses will be modified (line 24 in figure 2). Another framing condition, appears on lines 15, 25 and 36 of figure 2, and requires 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` already had access to a `p` *before* the call.

`Pol_sprout` promises that the result is a trusted purse that can trade with the receiver, no other valid purse's balance is affected, and references have not been leaked.

`Pol_can_trade_constant` guarantees that whether or not two purses can trade with each other can *never* change, no matter what code is run. This is another key

```
specification ValidPurse {
  field balance // Number

  policy Pol_deposit_1      //    1st case:
      amt∈ ℕ
          { res = this.deposit(amt, src) }
      res → (
          // TRUST
          src obeys_pre ValidPurse ∧ CanTrade(this,src)_pre
          // FUNCTIONAL SPECIFICATION
          ∧ 0≤amt≤src.balance_pre ∧
          this.balance=this.balance_pre+amt ∧
                    src.balance=src.balance_pre−amt  ∧
          //RISK
          ∀p.(p obeys_pre ValidPurse ∧ p∉ {this,src} →
                    p.balance=p.balance_pre)  ∧
          ∀o:_pre Object. ∀p obeys_pre ValidPurse.
                    MayAccess(o,p) → MayAccess_pre(o,p)  )

  policy Pol_deposit_2      //    2nd case:
      amt∈ ℕ
          { res = this.deposit(amt, src) }
       ¬res → (
          // TRUST and FUNCTIONAL SPECIFICATION
          ¬( src  obeys_pre ValidPurse ∧ CanTrade(this,src)_pre ∧
                    0≤amt≤src.balance_pre) ∧
          // RISK
          ∀p.(p obeys_pre ValidPurse→ p.balance=p.balance_pre)   ∧
          ∀o:_pre Object. ∀ p obeys_pre ValidPurse.
                     MayAccess(o,p) → MayAccess_pre(o,p)  )
```

Fig. 2. `ValidPurse` specification

ingredient of our approach: we can require that our code must preserve properties in the face of unknown code.

`Pol_protect_balance` guarantees that a valid purse `p`'s balance can only be changed: — *MayAffect*`(o,p.balance)` — by an object `o` that may access that purse: *MayAccess*`(o,p)`.

Finally, the abstract predicate `CanTrade` holds when two `Purse`s can trade with each other. `CanTrade` must be reflexive, but does not require that its arguments have the same class. It guarantees that `deposit` can transfer resources from one purse to another. This could involve a clearing house, interbank exchange, or other mechanisms abstract predicates can be implemented in different ways.

The use of assertions about the pre-state in methods' postconditions increases the expressive power of our specifications. For example, consider:
(A) `amt` $\in \mathbb{N}$ **`{res=this.deposit(amt,src)}`** `res` $\rightarrow$ `scr` **obeys**$_{pre}$`ValidPurse`
This allows us to deduce properties about the pre-state by observing the result of the

```
  policy Pol_sprout
      true
         { res = this.sprout() }
      // TRUST
      res obeys ValidPurse ∧  CanTrade(this,res)_pre  ∧
      // FUNCTIONAL SPECIFICATION
      res.balance=0 ∧
      // RISK
      ∀p.(p obeys_pre ValidPurse →
           p.balance=p.balance_pre  ∧ res ≠ p)   ∧
      ∀o:_pre Object. ∀ p obeys_pre ValidPurse.
         MayAccess(o,p) → MayAccess_pre(o,p)   )

  policy Pol_can_trade_constant
       true
         { any_code }
       ∀ prs1,prs2 obeys_pre ValidPurse.
            CanTrade(prs1,prs2) ⟷ CanTrade_pre(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 reflexive
```

Fig. 2. ValidPurse specification (contd.)

method call. Such a specification cannot be easily translated into one which does not make use of this facility, as in:
(B) scr **obeys** ValidPurse ∧ amt ∈ $\mathbb{N}$ **{res=this.deposit(amt,src)}** res
(B) differs from (A) in that (B) requires us to establish that scr **obeys** ValidPurse before making the call, while (A) does not.

2.4 Establishing Mutual Trust

An escrow must build a two-way, trusted transfer by combining one-way transfers. From Pol_deposit_1 we obtain that the call res1=dest.deposit(amt, src) lets us conclude res1∧ dest **obeys** ValidPurse→ src **obeys** ValidPurse. This trust is just one way: from the destination to the source purse. We can establish mutual trust between two purses by then attempting to perform a second deposit *in the reverse direction* from destination to source: res2=src.deposit(amt, dest) which in turn gives res2∧ src **obeys** ValidPurse→ dest **obeys** ValidPurse. Reasoning conditionally, on a path where res1 ∧ res2 are true, we can then establish mutual trust:

dest **obeys** ValidPurse ⟷ src **obeys** ValidPurse

We establish this formally in Section 3.4, having only argued informally earlier [17, 36].

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.

2.5 Escrow with Explicit Mutual Trust

```
1  method deal_version2(  ) // 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   // similar to lines 4-10 from above, but for Goods
14
15   // make the transaction
16   // as in lines 8-29 from Fig.1
17 }
```

Fig. 3. Revised `deal_version2` method

Two way deposit calls are sufficient to establish mutual trust, but come with risks. For example, as part of validating that a buyer's purse the seller's purse, we must pass the buyer's purse as an argument in a `deposit` call to the seller's 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 3. Line 4 creates a `escrowMoney` purse and then lines 5–10 hypothetically establish mutual trust between the `escrowMoney`, `sellerMoney`, and `buyerMoney` purses. The `sellerMoney` purse doesn't need to validate `escrowMoney` explicitly (`sellerMoney.deposit(0,escrowMoney)`) because the `sprout` method specification says sprouted purses can trusted as much as their parent purses. (Figure 4 illustrates the trust relationships.) If any of these `deposit` request

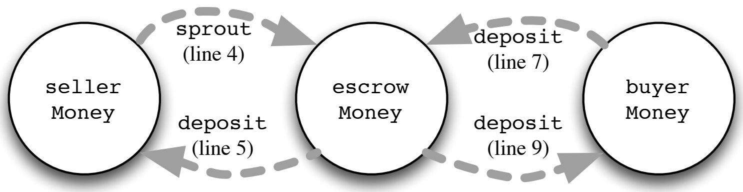

Fig. 4. Establishing Mutual Trust. Dashed arrows show purse validation.

fail, we abort. Afterwards we do exactly the same, but for goods purses rather than money purses. Finally, we carry out the escrow exchange itself, in exactly the same manner as lines 8–29 of the first implementation in Figure 1.

2.6 Specifying the Mutual Trust Escrow

Figure 5 shows a specification for the revised escrow deal method from Figure 3. This specification uses conditional and hypothetical reasoning to distinguish four cases, based on the value of the result and 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 can now discuss the four cases of the policy:

1st case: The result is `true` and all participant purses are trustworthy. Then, the goods and money purses can trade with each other, and there was sufficient money in the buyer's purse and sufficient goods in the sellers purse. In this case, everything is fine, so the transfer can proceed: `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).

2nd case: The result is `false` and all participant purses are trustworthy. Then one or more of the functional correctness conditions are not satisfied: purses' were unable to trade with each other, or input purses did not have sufficient balance. Again, no risk arises to any purses.

3rd 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.

```
1  specification ValidEscrow {
2     fields sellerMoney, sellerGoods, buyerMoney, buyerGoods
3     fields price, amt    // ℕ
4
5   policy Pol_deal
6    price,amt∈ ℕ ∧ price,amt>0
7          { this.deal(  ) }
8    res ∧ BadPPrs=∅   → (       //   1st case:
9      CanTrade(buyerMoney,sellerMoney) ∧
10     CanTrade(buyerGoods,sellerGoods) ∧
11     buyerMoney.balance=buyerMoney.balance_pre−price ∧
12     sellerMoney.balance=sellerMoney.balance_pre+price∧
13     buyerGoods.balance=buyerGoods.balance_pre+amt ∧
14     sellerGoods.balance=sellerGoods.balance_pre−amt ∧
15     ∀p:_pre OthrPrs. p.balance=p.balance._pre  ∧
16     ∀o:_pre Object,p:_pre GoodPrs.
17          (MayAccess(o,p) → MayAccess(o,p)_pre) )
18   ∧
19   ¬res ∧ BadPPrs=∅    → (   //   2nd case:
20     ¬( CanTrade(buyerMoney,sellerMoney) ∧
21          CanTrade(buyerGoods,sellerGoods) ∧
22          buyerMoney.balance_pre ≥ price ∧
23          sellerGoods.balance_pre ≥ amt ) ∧
24     ∀p:_pre GoodPrs. p.balance=p.balance._pre   ∧
25     ∀o:_pre Object,p:_pre GoodPrs.
26           (MayAccess(o,p) → MayAccess(o,p)_pre)    )
27   ∧
28   ¬res ∧ BadPPrs≠∅   →  (  //   3rd case:
29     ∀p:_pre GoodPrs. (p.balance=p.balance._pre ∨
30            ∃ bp∈BadPPrs_pre.MayAccess(bp,p)_pre)   ∧
31     ∀o:_pre Object,p:_pre GoodPrs.( MayAccess(o,p) →
32          (MayAccess(o,p)_pre ∨∃b∈BadPPrs_pre.MayAccess(b,p)_pre)) )
33   ∧
34   res ∧ BadPPrs≠ ∅   →  (   //   4th case:
35     buyerMoney obeys ValidPurse ⟷ sellerMoney obeys ValidPurse  ∧
36     buyerGoods obeys ValidPurse ⟷ sellerGoods obeys ValidPurse  ∧
37     ∀p:_pre OthrPrs. (p.balance=p.balance._pre ∨
38         ∃bp∈ BadPPrs_pre.MayAccess(bp,p)_pre)   ∧
39     ∀o:_pre Object,p:_pre GoodPrs. (MayAccess(o,p)→
40        (MayAccess(o,p)_pre∨∃b∈BadPPrs_pre.MayAccess(b,p)_pre)) )
41 }
```

Fig. 5. `ValidEscrow` specification

4th case: The result is `true` and some participant purse is untrustworthy — actually at least two matching participant purses are untrustworthy. That is, a pair of matching purses have coöperated to suborn the escrow *and we cannot tell*. Therefore, either both money purses are untrustworthy, (as per line 35), or both goods purses are untrustworthy, (as per line 36), 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 cases correspond to a traditional specification, because traditional specifications assume all objects are trustworthy. The third and fourth cases arise precisely because we are explicitly modelling the trust and risk involved in an open system.

Discussion The 3rd and 4th case represent more of a risk than we would like: ideally (as in the 2nd 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 4th 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 3rd case, because by the time the escrow starts making non-0 transactions it has established that the purses in each pair are both either trustworthy or both not.

Most importantly (perhaps surprisingly) the return value of the method, `res`, does *not* indicate whether the participants were trustworthy or not. A `true` result may be returned in the 1st case (all purses trustworthy) as well as the 4th (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 as a surprise to us (and to the escrow's designers [31].) 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 exchange has taken place. Furthermore if it returns `false`, the exchange has not taken place and with *no more* risk to the honest purses than existed before the call. The `ValidEscrow` specification also gives a guarantee to other purse objects even if they did not participate in the deal: dishonest purses can only change other purses' balances if they had prior access to those other purses.

3 A Formal Model of Trust and Risk

In this section we provide an overview of our core programming language, $\mathcal{F}$*ocal*, our specification language, *Chainmail*, and our Hoare logic. The Hoare logic uses *four-tuples* because it includes an invariant that must be preserved during the execution of a statement as well as a postcondition established afterwards. We also outline a key step required to prove that `deal_version2` meets the `ValidEscrow` specification: we prove that two purses can establish mutual trust, and formally delineate the risk. Many details are relegated to our technical report [18]; here we adopt its numbering for definitions.

3.1 $\mathcal{F}ocal$

We define a small object oriented language, $\mathcal{F}ocal$ (Featherweight Object Capability Language, not to be confused with FOCAL [28]). $\mathcal{F}ocal$ supports classes, fields and methods. (Figures 1 and 3 are effectively examples of $\mathcal{F}ocal$.) $\mathcal{F}ocal$ is memory-safe: it does not allow addresses to be forged, or non-existent methods or fields to be called, read or written. $\mathcal{F}ocal$ 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: similar to JavaScript, Grace, E, and Dart's unchecked mode.

Modules, M, are mappings from class identifiers to $\mathcal{F}ocal$ class definitions, and from predicate identifiers to $\mathcal{C}hainmail$ assertions as described in section 3.2. The 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, taking M_p as a module implementing purses, and M_e as another module implementing the escrow, $M_p * M_e$ is defined but $M_e * M_e$ is not.

$\mathcal{F}ocal$ enforces a weak form of privacy for fields; only the receiver may modify these fields, and anybody may read them.

The operational semantics of $\mathcal{F}ocal$ takes a module M and a runtime state $\sigma = \mathit{frame} \times \mathit{heap}$ and maps statements onto a new state σ'.

Definition 6 (Shape of Execution).
$\leadsto$ *: Module* $\times$ *state* $\times$ *Stmts* $\longrightarrow$ *state*

Arising and Reachable Configurations Policies need to be satisfied in all *configurations* (pairs of states and statements) which may arise during execution of the program. For example, if a program contains a class which has field which is not exported, and where this field is initialized to 0 by the constructor, and incremented by 3 in all method calls, then in the arising configurations the value of this field is guaranteed to be a multiple of 3. Thus, through the concept of arising configurations we can ignore configurations which are guarantee not to arise.

To define arising configurations we need the concept of initial configuration, and reachability. A configuration is *reachable* from some starting configuration if it is reached during the evaluation of the starting configuration after any number of steps. We define the function $\mathcal{R}each : Module \times state \times Stmts \longrightarrow \mathcal{P}(state \times Stmts)$ by cases on the structure of the statements. Note that $\mathcal{R}each(M, \sigma, \texttt{stmts})$ is defined, even when the execution should diverge. This is important, because it allows us to give meaning to capability policies without requiring termination.
We then define $\mathcal{A}rising(M)$ as the set of runtime configurations which may be reached during execution of some initial context (σ_0,$\texttt{stmts}_0$).

Definition 7 (Arising and Initial configurations).

$$\begin{array}{ll}
\mathcal{I}nit : Module \longrightarrow \mathcal{P}(state \times Stmt) \\
\mathcal{A}rising : Module \longrightarrow \mathcal{P}(state \times Stmts) \\
\mathcal{I}nit(M) = & \{\ (\ \sigma_0, \textbf{new}\ \mathit{c.m}(\ \textbf{new}\ \mathit{c'})\ )\ |\ c, c' \in dom(M), \\
 & \quad \mathit{where}\ \sigma_0\ =\ ((\iota_0, \textbf{null}), \chi_0), \mathit{and}\ \chi_0(\iota) = (\texttt{Object}, \emptyset)\ \} \\
\mathcal{A}rising(M) = & \bigcup_{(\sigma, stmts) \in \mathcal{I}nit(M)} \mathcal{R}each(M, \sigma, stmts)
\end{array}$$

3.2 *Chainmail*

Chainmail is a specification language where a specification is a conjunction of a set of named policies. (Figures 2 and 5 are examples of *Chainmail* specifications.)

Chainmail policies are based on one-state assertions (A) and two-state assertions (B). To express the state in which an expression is evaluated, we annotate it with a subscript. For example, $x > 1$ is a one-state, and $x_{pre} - x_{post} = 1$ is a two-state assertion. Validity of an assertion is defined in the usual manner, *e.g.* in a state σ with $\sigma(x) = 4$ we have $M, \sigma \models x > 1$. If we also have $\sigma'(x) = 3$, then we obtain $M, \sigma, \sigma' \models x_{pre} - x_{post} = 1$. *Chainmail* specifications may also express ghost information, which is not stored explicitly in the state σ but can be deduced from it — e.g. the length of a null-terminated string.

Policies can have one of the three following forms: 1) invariants of the form A, which require that A holds at all visible states of a program; or 2) A **{** `code` **}** B, which require that execution of `code` in any state satisfying A will lead to a state satisfying B wrt the original state or 3) A **{** `any_code` **}** B which requires that execution of *any* code in a state satisfying A will lead to a state satisfying B wrt the original state.

Definition 12 (Policies).
$Policy$ ::= $A \mid A\ \{\mathtt{code}\}\ B \mid\ A\ \{\mathtt{any\_code}\}\ B$
$PolSpec$ ::= `specification` S **{** $Policy^*$ **}**

One-state assertions include assertions about expressions (such as $\leq$, $>$ *e.t.c.*) and four additional assertions: *Expr* **obeys** *SpecId* to model trust, i.e. that an object confirms to a specification; and $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$ to model risk, i.e. whether one object may access another, or alter a property. These are *hypothetical*, in that they talk about the potential effects or behaviour of code: we cannot somehow evaluate their truth-value when executing the program. The fourth assertion *Expr*:*ClassId* simply tests class membership.

Validity of one-state assertions is expressed through the judgment $M, \sigma \models A$. The key case is that some expression obeys a specification if it satisfies that specification's policies in all reachable configurations arising from the module.

(from Definition 13):
- $M, \sigma \models \mathtt{e{:}C}$ iff $\sigma(\lfloor \mathtt{e} \rfloor_{M,\sigma}) \downarrow_1 = \mathtt{C}$.
- $M, \sigma \models \mathcal{M}ay\mathcal{A}ffect(\mathtt{e}, \mathtt{e'})$ iff there exist method $\mathtt{m}$, arguments $\bar{\mathtt{a}}$, state σ', identifier $\mathtt{z}$, such that $M, \sigma[\mathtt{z} \mapsto \lfloor \mathtt{e} \rfloor_{M,\sigma}], z\,.\mathtt{m}(\bar{\mathtt{a}}) \leadsto \chi'$, and $\lfloor \mathtt{e'} \rfloor_{M,\sigma} \neq \lfloor \mathtt{e'} \rfloor_{M,\sigma\downarrow_1,\chi'}$.
- $\mathtt{M}, \sigma \models \mathcal{M}ay\mathcal{A}ccess(\mathtt{e}, \mathtt{e'})$ iff there exist fields $\bar{\mathtt{f}}$, such that $\lfloor \mathtt{z}.\bar{\mathtt{f}} \rfloor_{M,\sigma[\mathtt{z} \mapsto \lfloor \mathtt{e} \rfloor_{M,\sigma}]} = \lfloor \mathtt{e'} \rfloor_{M,\sigma}$
- $M, \sigma \models \mathtt{e}$ **obeys** *PolSpecId* iff
  $\forall\, (\sigma, \mathtt{stmts}) \in \mathcal{A}rising(M).\ \forall i \in \{1..n\}.\ \forall\, \sigma', \mathtt{stmts'}.$
  $(\sigma', \mathtt{stmts'}) \in \mathcal{R}each(M, \sigma, \mathtt{stmts}) \longrightarrow M, \sigma'[z \mapsto \lfloor e \rfloor_\sigma] \models Policy_i[z/\mathbf{this}]$
  where z is a fresh variable in σ', and where we assume that *PolSpecId* was defined as `specification` *PolSpecId* $\{$ $Policy_1, ...Policy_n$ $\}$.

Two-state assertions allow us to compare properties of two different states. Validity of two-state assertions $M, \sigma, \sigma' \models B$ is defined similarly to one-state assertions, using cases. We can now define adherence to policy, $M, \sigma \models_{pol} Policy$:

Definition 15 (Adherence to Policies).

– $M, \sigma \models_{pol} A$ *iff* $M, \sigma \models A$

– $M, \sigma \models_{pol} A\,\{\texttt{code}\}\,B$ *iff*

$$(\ M, \sigma \models A \ \wedge\ M, \sigma, \texttt{code} \rightsquigarrow \sigma' \quad \longrightarrow \quad M, \sigma, \sigma' \models B\ )$$

– $M, \sigma \models_{pol} A\,\{\texttt{any\_code}\}\,B$ *iff*

$$\forall \texttt{code}.\ (\ (\sigma, \texttt{code}) \in \mathcal{A}rising(M)\ \wedge\ M, \sigma \models A\ \wedge\ M, \sigma, \texttt{code} \rightsquigarrow \sigma' \quad \longrightarrow \quad M, \sigma, \sigma' \models B\ )$$

3.3 Hoare Logic

The Hoare logic allows us to prove adherence to policies. In order to reflect that the code to be verified is executed in an open system, and that it calls code whose specification and trustworthiness is unknown to the code being verified, we use Hoare four-tuples rather than Hoare triples, so that not only do they guarantee a postcondition holds *after* execution of the code, but also guarantee that an invariant is preserved *during* execution of the code. These invariants are critical to modelling risk, as they let us talk about the absence of temporary but unwanted effects caused on objects during execution.

A Hoare four-tuple is either $M \vdash A\,\{\,\texttt{stms}\,\}\,A' \bowtie B$ (executing stms in any state satisfying A will lead to a state which satisfies A') or $M \vdash A\,\{\,\texttt{stms}\,\}\,B' \bowtie B$ (executing stms in any state satisfying A will lead to a state where the relation of the old and new state is described by B'). Critically, both promise that the relation between the initial state, and *any* of the intermediate states reached by execution of stms, will maintain the invariant B. The execution of stmts may call methods defined in M, and the predicates appearing in A, A', B', and B, may use predicates defined in M. When M is implicit from the context, we use the shorthand $\vdash A\,\{\,\texttt{stms}\,\}\,A' \bowtie B$.

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 [34], *"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."*

Definition 16 (Validity of Hoare Four-Tuples).

$$M \models A\,\{\,\texttt{stms}\,\}\,B' \bowtie B \quad \textit{iff}\ \ \forall M', \sigma.$$
$$(\sigma, \_) \in \mathcal{A}rising(M * M')\ \wedge\ M * M', \sigma \models A\ \wedge\ M{*}M', \sigma, \texttt{stms} \rightsquigarrow \texttt{res}, \sigma'$$
$$\longrightarrow$$
$$M{*}M', \sigma, \sigma' \models B'\ \wedge\ \forall \sigma'' \in \mathcal{R}each(M, \sigma, \texttt{stmts}).\ M{*}M', \sigma, \sigma'' \models B$$

Figure 6 shows a selection of our Hoare rules. It starts with two familiar Hoare Logic rules: In (VARASG) and (FIELDASG) the postconditions use the previous value of the right-hand-side, and thus allow us to deduce, *e.g.* :

$$\vdash \mathbf{this}.f = 21\,\{\,\mathbf{this}.f = 2 * \mathbf{this}.f\,\}\,\mathbf{this}.f = 42 \bowtie \mathit{true}.$$

(METH-CALL-1) is also familiar, as it reasons over method calls under the assumption that the receiver **obeys** a specification S, and that the current state satisfies the precondition of m as defined in S.

The remaining rules are more salient.

(METH-CALL-2) expresses the basic axiom of object-capability systems that "only connectivity begets connectivity" [30]. It promises in the postcondition that the result of the method call v cannot expose access to any object z that wasn't accessible initially

(VARASG) (FIELDASG)

$$\frac{}{\vdash \mathit{true}\,\{\,\texttt{v:=a}\,\}\,\texttt{v} = \texttt{a}_{pre}\ \bowtie\ \mathit{true}} \qquad \frac{}{\vdash \mathit{true}\,\{\,\textbf{this}.\texttt{f:=a}\,\}\,\textbf{this}.\texttt{f} = \texttt{a}_{pre}\ \bowtie\ \mathit{true}}$$

(METH-CALL-1)

$$\frac{M(S)\ =\ \textbf{spec}\ S\,\{\ \overline{\mathit{Policy}}, A\,\{\ \texttt{this.m(par)}\ \}\ B, \overline{\mathit{Policy}'}\ \}}{\vdash \texttt{x}\ \textbf{obeys}\ S \wedge A[\texttt{x/this},\texttt{y/par}]\,\{\,\texttt{v:=x.m(y)}\,\}\,B[\texttt{x/this},\texttt{y/par},\texttt{v/res}]\ \bowtie\ \mathit{true}}$$

(METH-CALL-2)

$$\frac{\begin{array}{l} B \equiv \forall \texttt{z} :_{pre} \texttt{Object}.\ \mathcal{MayAccess}(\texttt{v},\texttt{z}) \rightarrow (\ \mathcal{MayAccess}_{pre}(\texttt{x},\texttt{z})\ \vee\ \mathcal{MayAccess}_{pre}(\texttt{y},\texttt{z})\ ) \\ B' \equiv \forall \texttt{z},\texttt{u} :_{pre} \texttt{Object}.\ (\ \mathcal{MayAccess}(\texttt{u},\texttt{z}) \rightarrow \\ \qquad\qquad\qquad (\mathcal{MayAccess}_{pre}(\texttt{u},\texttt{z})\ \ \vee \\ \qquad\qquad\qquad\qquad (\ (\mathcal{MayAccess}_{pre}(\texttt{x},\texttt{z}) \vee \mathcal{MayAccess}_{pre}(\texttt{y},\texttt{z})) \wedge \\ \qquad\qquad\qquad\qquad\ \ (\mathcal{MayAccess}_{pre}(\texttt{x},\texttt{u}) \vee \mathcal{MayAccess}_{pre}(\texttt{y},\texttt{u}))\ )\ )\ \ ) \end{array}}{\vdash \mathit{true}\,\{\,\texttt{v:=x.m(y)}\,\}\,B\ \bowtie\ B'}$$

(FRAME-METHCALL)

$$\frac{\begin{array}{l} \vdash A\,\{\,v := \texttt{x.m(y)}\,\}\,\mathit{true}\ \bowtie\ B \\ B \equiv \forall \texttt{z}.(\ \mathcal{MayAffect}(\texttt{z}, A') \rightarrow B'(\texttt{z})\ )\ \wedge \\ \qquad \forall \texttt{z}.(\ (\mathcal{MayAccess}(\texttt{x},\texttt{z}) \vee \mathcal{MayAccess}(\texttt{y},\texttt{z}) \vee \mathcal{New}(\texttt{z})\ )\ \rightarrow\ \neg B'(\texttt{z})\ ) \end{array}}{\vdash A \wedge A'\,\{\,\texttt{v:=x.m(y)}\,\}\,A'\ \bowtie\ \mathit{true}}$$

(CODE-INVAR-1) (CODE-INVAR-2)

$$\frac{\begin{array}{l} M(S) \equiv\ \textbf{spec}\ S\,\{\ \overline{\mathit{Policy}}, P, \overline{\mathit{Policy}'}\ \} \\ B \equiv \forall \texttt{x}.(\ \texttt{x}\ \textbf{obeys}\ S \rightarrow P[/\texttt{x/this}]\,) \end{array}}{\vdash \mathit{true}\,\{\,\texttt{stmts}\,\}\,\mathit{true}\ \bowtie\ B} \qquad \frac{}{\vdash e\ \textbf{obeys}\ S\,\{\,\texttt{stmts}\,\}\,\mathit{true}\ \bowtie\ e_{pre}\ \textbf{obeys}\ S}$$

(CONS-2) (CONS-3) (CONS-4)

$$\frac{\begin{array}{l} \vdash A\,\{\,\texttt{stmts}\,\}\,B\ \bowtie\ B'' \\ A', B' \rightarrow_M A, \_ \end{array}}{\vdash A'\,\{\,\texttt{stmts}\,\}\,B' {\rightarrow} B\ \bowtie\ B''} \qquad \frac{\begin{array}{l} \vdash A\,\{\,\texttt{stmts}\,\}\,B\ \bowtie\ B' \\ A, B \rightarrow_M \_, A' \end{array}}{\vdash A\,\{\,\texttt{stmts}\,\}\,A'\ \bowtie\ B'} \qquad \frac{\begin{array}{l} \vdash A\,\{\,\texttt{stmts}\,\}\,A'\ \bowtie\ B' \\ A, A' \rightarrow_M B \end{array}}{\vdash A\,\{\,\texttt{stmts}\,\}\,B\ \bowtie\ B'}$$

(SEQUENCE)

$$\frac{\vdash A\,\{\,\texttt{s}_1\,\}\,B_1\ \bowtie\ B' \quad \vdash A_2\,\{\,\texttt{s}_2\,\}\,B_2\ \bowtie\ B' \quad A, B_1 \rightarrow_M \_, A_2 \quad B_1, B_2 \rightarrow_M B}{\vdash A\,\{\,\texttt{s}_1;\ \texttt{s}_2\,\}\,B\ \bowtie\ B'}$$

Fig. 6. A selection of Hoare Logic rules; we assume that the module M is globally given

to the method call's receiver `x` or argument `y`. Additionally, it also promises that, *during* execution of the method, accessibility does not change, unless the participants (here `z` and `u`) were accessible to the receiver and/or the argument *before* the call. Note that this latter promise is made via the invariant (fourth) rather than the postcondition (third) part of the Hoare-tuple. Note also that this rule is applicable *even if we know nothing* about the receiver of the call: this rule and the invariants are critical to reasoning about risk.

(CODE-INVAR-1) allows reasoning under the hypothesis that an object o **obeys** its specification S: in this case o can be trusted to act in accordance with S always.

(FRAME-METHCALL) also expresses an axiom of object-capability languages, namely that in order to cause some visible effect, one must have access to an object able to perform the effect. Coupled with "only connectivity begets connectivity", this implies that a method can cause some effect only if the caller has (transitive) access to some object able to cause the effect (including perhaps the callee).

The remaining rules each make use of the entailment judgement $\rightarrow_M$, which allows converting back and forth between one-state and two-state assertions and comes in number of flavours; the relevant ones are defined as follows.

Definition 19 (Entailment).

1. $A, B \rightarrow_M A', A''$ *iff*
$\forall \sigma, \sigma'.\ \sigma \models A\ \wedge\ \sigma, \sigma' \models B \quad \longrightarrow \quad \sigma \models A'\ \wedge\ \sigma' \models A''$
2. $A, A' \rightarrow_M B$ *iff*
$\forall \sigma, \sigma'.\ \sigma \models A\ \wedge\ \sigma' \models A' \quad \longrightarrow \quad \sigma, \sigma' \models B$
3. $B, B' \rightarrow_M B''$ *iff*
$\forall \sigma, \sigma', \sigma''.\ \sigma, \sigma' \models B\ \wedge\ \sigma', \sigma'' \models B' \quad \longrightarrow \quad \sigma, \sigma'' \models B''$

The rules (CONS-3) and (CONS-4) make use of the entailment judgement to allow converting between one- and two-state postconditions during Hoare logic reasoning. To reason across sequenced computations $\mathtt{s}_1; \mathtt{s}_2$, the (SEQUENCE) rule requires finding a one-state assertion A_2 that holds after $\mathtt{s}_1$ and is the precondition of $\mathtt{s}_2$. It uses the entailment $A, B_1 \rightarrow_M \_, A_2$ to require that $\mathtt{s}_1$'s execution guarantees A_2, and the entailment $B_1, B_2 \rightarrow_M B$ to require that the combined execution of $\mathtt{s}_1$ and $\mathtt{s}_2$ guarantees the top-level postcondition B.

Theorem 3 (Soundness of the Hoare Logic).
For all modules M, statements $\mathtt{stms}$ and assertions A, B and B',
If $M \vdash A \{\, \mathtt{stms} \,\} B' \bowtie B$, *then* $M \models A \{\, \mathtt{stms} \,\} B' \bowtie B$.

The theorem is proven in [18].

In summary, we have four "code agnostic" rules — rules which are applicable regardless of the underlying code. Rules (FRAME-METHCALL) and (METH-CALL-2) express restrictions on the effects of a method call. Normally such restrictions stem from the specification of the method being called, but here we can argue in the absence of any such specifications, allowing us to reason about risk even in open systems. Rules (CODE-INVAR-1) and (CODE-INVAR-2) are applicable on *any* code, and allow us to assume that an object which **obeys** a specification S, satisfies all policies from S, and that the object, once trusted, will always be **obeying** S.

3.4 Proving Mutual Trust

We now use our Hoare Logic to prove the key steps of the escrow protocol, establishing mutual trust and delineating the risk. Here we have space to show just one-way trust between the escrow and seller in full: the remaining reasoning to establish mutual trust is outlined in the technical report [18]. Figure 7 shows the Hoare tuple for the first statement in method `deal` (line 4 from Figure 3). Lines 3-8 of Figure 7 describe the postcondition in case `escrowMoney` indeed **obeys** `ValidPurse`, while lines 9-17 make absolutely no assumption about the trustworthiness, or provenance, of `escrowMoney`.

$$
\begin{array}{rl}
1 & \texttt{true} \\
2 & \qquad \{ \quad \texttt{var escrowMoney := sellerMoney.sprout} \quad \} \\
3 & \texttt{sellerMoney}_{pre}\ \textbf{obeys}\ \texttt{ValidPurse} \longrightarrow \\
4 & \qquad\quad (\ \texttt{escrowMoney}\ \textbf{obeys}\ \texttt{ValidPurse}\ \wedge \\
5 & \qquad\qquad CanTrade(\texttt{escrowMoney}, \texttt{sellerMoney})\ \wedge \\
6 & \qquad\qquad \texttt{escrowMoney.balance} = 0\ \wedge \\
7 & \qquad\qquad \forall p \in_{pre} \texttt{GoodPrs}.\ p.\texttt{balance}_{pre} = p.\texttt{balance}\ \wedge \\
8 & \qquad\qquad \texttt{sellerMoney}\ \textbf{obeys}\ \texttt{ValidPurse}\ ) \quad \wedge \\
9 & \ \forall p :_{pre} \texttt{GoodPrs}. \\
10 & \qquad (\,p.\texttt{balance}_{pre} = p.\texttt{balance} \vee \mathcal{M}ay\mathcal{A}ccess_{pre}(\texttt{sellerMoney}, p)\,) \quad \wedge \\
11 & \ \forall z :_{pre} \texttt{Object}. \\
12 & \qquad (\,\mathcal{M}ay\mathcal{A}ccess(\texttt{escrowMoney}, z) \longrightarrow \mathcal{M}ay\mathcal{A}ccess_{pre}(\texttt{sellerMoney}, z)\,)\ \wedge \\
13 & \ \forall z, y :_{pre} \texttt{Object}. \\
14 & \qquad (\,\mathcal{M}ay\mathcal{A}ccess(\,z, y\,) \quad \longrightarrow \\
15 & \qquad\qquad (\,\mathcal{M}ay\mathcal{A}ccess_{pre}(\,z, y\,)\ \vee \\
16 & \qquad\qquad\qquad\qquad \mathcal{M}ay\mathcal{A}ccess_{pre}(\,\texttt{sellerMoney}, y\,) \wedge \\
17 & \qquad\qquad\qquad\qquad \mathcal{M}ay\mathcal{A}ccess_{pre}(\,\texttt{sellerMoney}, z\,)\,) \quad) \\
18 & \ \bowtie \\
19 & \ \texttt{true} \\
20 &
\end{array}
$$

Fig. 7. Hoare tuple for first step in `deal`

By `Pol_sprout` and (METH-CALL-1) we obtain that

$$
\text{(A)}\quad
\begin{array}{l}
\texttt{sellerMoney}\ \textbf{obeys}\ \texttt{ValidPurse} \\
\qquad \{\ \texttt{escrowMoney} := \texttt{sellerMoney.sprout}\} \\
\texttt{escrowMoney}\ \textbf{obeys}\ \texttt{ValidPurse}\ \wedge ...rest... \\
\qquad \bowtie \\
true
\end{array}
$$

By application (CONS-2) on the above we obtain

$$
\text{(B)}\quad
\begin{array}{l}
true \\
\qquad \{\ \texttt{escrowMoney} := \texttt{sellerMoney.sprout}\} \\
\texttt{sellerMoney}_{pre}\ \textbf{obeys}\ \texttt{ValidPurse} \rightarrow \\
\qquad\qquad (\ \texttt{escrowMoney}\ \textbf{obeys}\ \texttt{ValidPurse}\ \wedge\ ...rest...\,) \\
\qquad \bowtie \\
true
\end{array}
$$

To obtain line 8, we apply a basic framing rule ((FRAME-GENERAL) in [18]) and get
$\vdash ...\ \{\ \texttt{escrowMoney} := \texttt{sellerMoney.sprout}\ \}\ \texttt{escrowMoney}_{pre} = \texttt{escrowMoney}\ \bowtie\ ...$,
and then, in conjunction with (CODE-INVAR-2), (CONS-2) we also obtain that

$$
\text{(C)}\quad
\begin{array}{l}
true \\
\qquad \{\ \texttt{escrowMoney} := \texttt{sellerMoney.sprout}\} \\
\texttt{escrowMoney}_{pre}\ \textbf{obeys}\ \texttt{ValidPurse} \rightarrow \texttt{escrowMoney}\ \textbf{obeys}\ \texttt{ValidPurse} \\
\qquad \bowtie \\
...
\end{array}
$$

We can then apply a conjunction rule ((CONJ) in [18]) on (B) and (C), and obtain the

postcondition as in 4-8.
To obtain 9-11, we will apply several of the code-agnostic rules. After all, here we cannot appeal to the specification of `sprout`, as we do not know whether `sellerMoney` adheres to `ValidPurse`. We start by application of (METH-CALL-2), and a consequence rule ((CONS-1) in [18]):

$$\text{(D)}\quad \begin{array}{l} true \\ \quad \{\, \texttt{escrowMoney} := \texttt{sellerMoney.sprout} \} \\ true \\ \quad \bowtie \\ \forall z.\ \mathcal{M}ay\mathcal{A}ccess(\texttt{sellerMoney}, z) \to \mathcal{M}ay\mathcal{A}ccess_{pre}(\texttt{sellerMoney}, z) \end{array}$$

By applying the fact that $\forall u, v, w,\ \mathcal{M}ay\mathcal{A}ccess(u, v) \wedge \mathcal{M}ay\mathcal{A}ccess(v, w) \to \mathcal{M}ay\mathcal{A}ccess(u, w)$, and conjunction and inference rules on (D), we get:

$$\text{(E)}\quad \begin{array}{l} \neg\mathcal{M}ay\mathcal{A}ccess(\texttt{sellerMoney}, p) \\ \quad \{\, \texttt{escrowMoney} := \texttt{sellerMoney.sprout} \} \\ true \\ \quad \bowtie \\ \forall z.\ \mathcal{M}ay\mathcal{A}ccess(\texttt{sellerMoney}, z) \to \neg\mathcal{M}ay\mathcal{A}ccess(z, p) \end{array}$$

By application of rule (CODE-INVAR-1), we obtain:

$$\text{(F)}\quad \begin{array}{l} true \\ \quad \{\, \texttt{escrowMoney} := \texttt{sellerMoney.sprout} \} \\ true \\ \quad \bowtie \\ \forall p.(\, p\ \textbf{obeys}\ \texttt{ValidPurse} \to (\forall z.\ \mathcal{M}ay\mathcal{A}ffect(z, p.\texttt{balance}) \to \mathcal{M}ay\mathcal{A}ccess(z, p))\,) \end{array}$$

Through a combination of (E) and (F) and application of conjunction, and application of (FRAME-METH-CALL), we obtain that

$$\text{(G)}\quad \begin{array}{l} \neg\mathcal{M}ay\mathcal{A}ccess(\texttt{sellerMoney}, p) \\ \quad \{\, \texttt{escrowMoney} := \texttt{sellerMoney.sprout} \} \\ true \\ \quad \bowtie \\ p\ \textbf{obeys}_{pre}\texttt{ValidPurse} \to (p.\texttt{balance} = p.\texttt{balance}_{pre}) \end{array}$$

Now by applying (CONS-2) on (F), we obtain

$$\text{(H)}\quad \begin{array}{l} true \\ \quad \{\, \texttt{escrowMoney} := \texttt{sellerMoney.sprout} \} \\ true \\ \quad \bowtie \\ \forall p.\ p\ \textbf{obeys}_{pre}\texttt{ValidPurse} \to \\ \qquad (\, p.\texttt{balance} = p.\texttt{balance}_{pre}\ \vee\ \mathcal{M}ay\mathcal{A}ccess(\texttt{sellerMoney}, p)\,) \end{array}$$

We now apply (CONS-1) from [18] to conjoin the invariant and postcondition, obtaining

$$\text{(I)}\quad \begin{array}{l} true \\ \quad \{\, \texttt{escrowMoney} := \texttt{sellerMoney.sprout} \} \\ \forall p.\ p\ \textbf{obeys}_{pre}\texttt{ValidPurse} \to \\ \qquad (\, p.\texttt{balance} = p.\texttt{balance}_{pre}\ \vee\ \mathcal{M}ay\mathcal{A}ccess(\texttt{sellerMoney}, p)\,) \\ \quad \bowtie \\ true \end{array}$$

Last, we obtain lines 11-12 from (METH-CALL-2). We also obtain lines 13-17 from (METH-CALL-2), and (CONS-1) from [18].

4 Related Work

Object Capabilities and Sandboxes. *Capabilities* were developed in the 60's by Dennis and Van Horn [10] within operating systems, and were adapted to the programming languages setting in the 70's [34]. *Object capabilities* were first introduced [30] in the early 2000s, and much recent work investigates the safety or correctness of object capability programs. Google's Caja [33] applies sandboxes, proxies, and wrappers to limit components' access to *ambient* authority. Sandboxing has been validated formally: Maffeis et al. [27] 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, there are a range of recent analyses of JavaScript [23, 5, 38, 26, 43] based on static analyses or type checking. Lerner et al. extend these approaches to ensure browser extensions observe *"private mode"* [26], while Dimoulas et al. [11] enforce explicit access policies. 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 Our Hoare logic invariants are similar to the guarantees in Rely-Guarantee reasoning [22]. 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, rely-guarantee and deny-guarantee assumes coöperation: composition is legal only if threads adhere to their rely or deny properties and guarantees. Our modules have to be robust and ensure that their invariants cannot be affected by *any* arbitrary, uncertified, untrusted 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 [20] to mobile ad-hoc networks [9], the internet of things [19], online dating [37], and as a component of a wider socio-technical system [8, 45]. 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 [21]. Solhaug and Stølen [42] 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. Various different logics have been used to measure trust in different kinds of systems. Some of the earliest work is Lampson et al.'s "speaks for" and "says" constructs [24], clear precursors to our " **obeys** " but for authentication rather than specifications. Murray [35] models object capability patterns in CSP, and applies automatic refinement checking to analyse various properties in the presence of untrusted components. Ries et al. [40] evaluate trust under uncertainty by evaluating Boolean expressions in terms of real values. Carbone et al. [41] and Aldini [3] model trust using temporal logic. Primiero and Taddeo [39] have developed a modal type theory that treats trust as a second-order relation over relations between counterparties. Merro and Sibilio [29] developed a trust model for a process calculus based on labelled transition systems. Compared with ours, these approaches use process calculi or other abstract logical models of systems, rather than engaging directly with the system's code.

Verification of Object Capability Programs. Drossopoulou and Noble [13, 36] have analysed Miller's Mint and Purse example [30] by expressing it in Joe-E and in Grace [36], and discussed the six capability policies as proposed in [30]. In [16], they proposed a complex specification language, and used it to fully specify the six policies from [30]; uncovering the need for another four policies. More recently, [14] they have shown how different implementations of the underlying Mint and Purse systems coexist with different policies. In contrast, this work formalises the informal ideas from [17], proposes $\mathcal{F}ocal$, which is untyped and modelled on Grace and JavaScript rather than Java; a much simpler specification language $\mathcal{C}hainmail$; the **obeys** predicate to model trust; $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$ to model risk; a full specification of the `Escrow`; and a Hoare logic for reasoning about risk and trust, applied to the Escrow specification.

5 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 contributed:

– *Hypothetical* predicates **obeys** to model trust, $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$ to model risk, and their formal semantics.
– *Open Assertions* and *Open Policies* whose validity must be guaranteed, even when linked with *any* other code.
– *Formal models* of $\mathcal{F}ocal$ and $\mathcal{C}hainmail$.
– *Hoare four-tuples* that make invariants explicit.
– A *Hoare logic* incorporating code agnostic inference rules.
– *Formal reasoning* to prove key steps of the Escrow Exchange.

In further work we will extend our approach to deal with concurrency, distribution, exceptions, networking, aliasing, and encapsulation. Finally, we hope to develop automated reasoning techniques to make these kinds of specifications practically useful.

Bibliography

[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. Invited Talk: Towards Reasoning about Risk and Trust in the Open World, 2014. slides from "http://www/doc.ic.ac.uk/~scd".

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

[17] S. Drossopoulou, J. Noble, and M. S. Miller. Swapsies on the Internet. In *PLAS*, 2015.

[18] S. Drossopoulou, J. Noble, M. S. Miller, and T. Murray. More Reasoning about Risk and Trust in an Open World. Technical Report ECSTR-15-08, VUW, 2015.

[19] L. Gu, J. Wang, and B. Sun. Trust management mechsnism for Internet of Things. *China Communications*, Feb. 2014.

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

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

[22] C. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.

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

[24] B. Lampson, M. Abadi, M. Burrows, and E. Wobbler. Authentication in Distributed Systems: Theory and Practice. *ACM TOCS*, 10(4):265–310, 1992.

[25] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 1973.

[26] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *ESORICS*, Sept. 2013.

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

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

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

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

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

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

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

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

[35] T. Murray. *Analysing the Security Properties of Object-Capability Patterns*. D.Phil. thesis, University of Oxford, 2010.

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

[37] G. Norcie, E. D. Cristofaro, and V. Bellotti. Bootstrapping trust in online dating: Social verification of online dating profiles. In *Fin. Crypt. & Data Sec.*, 2013.

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

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

[40] 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.

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

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

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

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

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

More Reasoning about Risk and Trust in an Open Word (Appendix)

Sophia Drossopoulou[1], James Noble[2], Mark Miller[3], Toby Murray[4],

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

1. Introduction

This is the companion appendix to our work *"Reasoning about Risk and Trust in an Open World"*. We give here the full definitions of $\mathcal{F}ocal$, $\mathcal{C}hainmail$, our Hoare logic, prove soundness of our Hoare logic, and then prove that our escrow exchange implementation establishes mutual trust while managing risk.

2. Formal Definition of the language $\mathcal{F}ocal$

2.1 Modules and Linking

$\mathcal{F}ocal$ modules map class identifiers to class descriptions, function identifiers to function descriptions, and predicate identifiers to predicate descriptions - we require implicitly for any module M, class identifier *c*, function identifier *f*, and predicate identifier P, that that $M(c) \in \mathit{ClassDescr}$ or undefined, that $M(f) \in \mathit{FunDescr}$ or undefined, and $M(P) \in \mathit{PredDescr}$ or undefined.

Definition 1 (Modules)**.**

$$
\begin{array}{ll}
\mathit{Module} & = \mathit{ClassId} \longrightarrow \mathit{ClassDescr} \\
\mathit{Specification} & = (\ \mathit{FunId} \cup \mathit{PredId} \cup \mathit{SpecId}\ ) \longrightarrow \\
& \quad (\mathit{FuncDescr} \cup \mathit{PredDescr} \cup \mathit{Specification}\ )
\end{array}
$$

We define linking of modules, $M * M'$, to be the the union of their respective mappings, provided that the domains of the two modules are disjoint:

Definition 2 (Linking and Lookup)**.** *Linking of modules M and M' is*

$$
\begin{array}{l}
* \ :\ Module \times Module \longrightarrow Module \\
M * M' = \left\{ \begin{array}{ll} M *_{aux} M', & \mathit{if}\ dom(M) \cap dom(M') = \emptyset \\ \bot & \mathit{otherwise.} \end{array} \right. \\
(M *_{aux} M')(c) = \left\{ \begin{array}{ll} M(id), & \mathit{if}\ M(id)\ \mathit{is\ defined} \\ M'(id) & \mathit{otherwise.} \end{array} \right.
\end{array}
$$

Classes We define the syntax ....

Definition 3 (Classes, Methods, Args)**.** *We define the synatx of modules below.*

$$
\begin{array}{lcl}
\mathit{ClassDescr} & ::= & \mathbf{class}\ \mathit{ClassId} \\
& & \quad \{\ \ (\mathbf{fld}\ \mathit{FieldId})^*\ (\ \mathit{methBody}\ )^*\ \ \} \\
\mathit{methBody} & ::= & \mathbf{method}\ \mathit{m}\ (\ \mathit{ParId}^*\ ) \\
& & \quad \{\ \ \mathit{Stmts}\ \mathbf{;\ return}\ \mathit{Arg}\ \ \} \\
\mathit{Stmts} & ::= & \mathit{Stmt}\ \mid\ \mathit{Stmt}\ \mathbf{;}\ \mathit{Stmts} \\
\mathit{Stmt} & ::= & \mathbf{var}\ \mathit{VarId}\ \mathbf{:=}\ \mathit{Rhs} \\
& \mid & \mathit{VarId}\ \mathbf{:=}\ \mathit{Rhs} \\
& \mid & \mathbf{this}.\mathit{FieldId}\ \mathbf{:=}\ \mathit{Rhs} \\
& \mid & \mathbf{if}\ \mathit{Arg}\ \mathbf{then}\ \mathit{Stmts}\ \mathbf{else}\ \mathit{Stmts} \\
& \mid & \mathbf{skip} \\
\mathit{Rhs} & ::= & \mathit{Arg}\mathbf{.}\mathit{MethId}(\ \mathit{Arg}^*\ )\ \mid\ \mathit{Arg} \\
& \mid & \mathbf{new}\ \mathit{ClassId}(\ \ \mathit{Arg}^*\ ) \\
\mathit{Arg} & ::= & \mathit{Path}\ \mid\ \mathbf{true}\ \mid\ \mathbf{false}\ \mid\ \mathbf{null} \\
\mathit{Path} & ::= & \mathit{ParId}\ \mid\ \mathit{VarId}\ \mid\ \mathbf{this} \\
& \mid & \mathit{Path}\mathbf{.}\ \mathit{FieldId}
\end{array}
$$

Note that $\mathcal{F}ocal$ supports a limited form of protection: the syntax supports reading of fields of any object, but restricts each object to being able to modify only its *own* fields.

Method Lookup We define the method lookup function, $\mathcal{M}$ which returns the corresponding method definition given a class and a method identifier.

Definition 4 (Lookup)**.** *The lookup function*

$$
\begin{array}{l}
\mathcal{M}(M, c, m) = \mathbf{method}\ m\ (\ p_1, ...p_n)\ \{\ stms;\ \mathbf{return}\ a\} \\
\mathit{iff}\ \ M(c) = \mathbf{class}\ c\{\ ... \\
\qquad\qquad \mathbf{method}\ m\ (\ p_1, ...p_n)\ \{\ stms;\ \mathbf{return}\ a\} \\
\qquad\quad ...\}\ . \\
\mathit{undefined,\ otherwise.}
\end{array}
$$

2.2 Execution of $\mathcal{F}ocal$

Runtime state The runtime state σ consists of a stack frame ϕ, and a heap χ. A stack frame is a mapping from receiver (**this**) to its address, and from the local variables (VarId) and parameters (ParId) to their values. Values are integers, the booleans **true** or **false**, addresses, or **null**. Addresses are ranged over by ι. The heap maps addresses to objects. Objects are tuples consisting of the class of the object, and a mapping from field identifiers onto values.

(METHCALL_OS)

$$\frac{\begin{array}{l}\lfloor a \rfloor_{\phi\cdot\chi} = \iota \\ \lfloor a_i \rfloor_{\phi\cdot\chi} = val_i \quad \forall i \in \{1..n\} \\ \mathcal{M}(M, \chi(\iota)\downarrow_1, \mathtt{m}) \; = \\ \quad \textbf{method}\; m(\; par_1, \ldots par_n)\; \{\; stms;\; \textbf{return}\; a'\} \\ \phi'' = \textbf{this} \mapsto \iota, par_1 \mapsto val_1, \ldots par_n \mapsto val_n \\ M,\, \phi'' \cdot \chi, stmts \;\rightsquigarrow \phi' \cdot \chi' \end{array}}{M,\, \phi \cdot \chi,\; a.m(\; a_1, \ldots a_n) \;\rightsquigarrow\; \chi', \lfloor a' \rfloor_{\phi'\cdot\chi'}}$$

(ARG_OS)

$$\frac{}{M,\, \phi \cdot \chi,\; a \;\rightsquigarrow\; \chi, \lfloor a \rfloor_{\phi\cdot\chi}}$$

(NEW_OS)

$$\frac{\begin{array}{l}\iota \text{ is new in } \chi \\ \mathtt{f}_1, \ldots \mathtt{f}_n \text{ are the fields defined in } CId \end{array}}{\begin{array}{l} M,\, \phi \cdot \chi,\; \textbf{new}\; C(\; a_1, \ldots a_n) \\ \rightsquigarrow\; \chi[\iota \mapsto (C, \mathtt{f}_1 \mapsto \lfloor a_1 \rfloor_{\phi,\sigma} \ldots \mathtt{f}_n \mapsto \lfloor a_n \rfloor_{\phi,\sigma})],\; \iota \end{array}}$$

(VARASG-1_OS)

$$\frac{M,\, \phi \cdot \chi,\; e \;\rightsquigarrow\; \chi',\, val}{M,\, \phi \cdot \chi,\; \textbf{var}\; v{:=}e \;\rightsquigarrow\; \phi[v \mapsto val] \cdot \chi'}$$

(VARASG-2_OS)

$$\frac{M,\, \phi \cdot \chi,\; e \;\rightsquigarrow\; \chi',\, val}{M,\, \phi \cdot \chi,\; v{:=}e \;\rightsquigarrow\; \phi[v \mapsto val] \cdot \chi'}$$

(FIELDASG_OS)

$$\frac{M,\, \phi \cdot \chi,\; e \;\rightsquigarrow\; \phi \cdot \chi',\, val}{M,\, \phi \cdot \chi,\; \textbf{this}.f{:=}\; e \;\rightsquigarrow\; \phi \cdot \chi'[\phi(\textbf{this}), f \mapsto val]}$$

(SEQUENCE_OS)

$$\frac{\begin{array}{l} M,\, \sigma, stmt \;\rightsquigarrow\; \sigma'' \\ M,\, \sigma'',\, stmts \;\rightsquigarrow\; \sigma' \end{array}}{M,\, \sigma,\; stmt\,;\; stmts \;\rightsquigarrow\; \sigma'}$$

(COND-TRUE_OS)

$$\frac{\begin{array}{l} \lfloor a \rfloor_\sigma = \textbf{true} \\ M,\, \sigma,\; stmts_1 \;\rightsquigarrow\; \sigma' \end{array}}{M,\, \sigma,\; \textbf{if}\; a\; \textbf{then}\; stmts_1\; \textbf{else}\; stmts_2 \;\rightsquigarrow\; \sigma'}$$

(COND-FALSE_OS)

$$\frac{\begin{array}{l} \lfloor a \rfloor_\sigma = \textbf{false} \\ M,\, \sigma,\; stmts_2 \;\rightsquigarrow\; \sigma' \end{array}}{M,\, \sigma,\; \textbf{if}\; a\; \textbf{then}\; stmts_1\; \textbf{else}\; stmts_2 \;\rightsquigarrow\; \sigma'}$$

(SKIP_OS)

$$\frac{}{M,\, \sigma,\; \textbf{skip} \;\rightsquigarrow\; \sigma}$$

Figure 1. Operational Semantics - done

$$\begin{array}{lcl}
\sigma \in \mathit{state} & = & \mathit{frame} \times \mathit{heap} \\
\phi \in \mathit{frame} & = & \mathit{StackId} \longrightarrow \mathit{val} \\
\chi \in \mathit{heap} & = & \mathit{addr} \longrightarrow \mathit{object} \\
\mathrm{v} \in \mathit{val} & = & \{\, \mathtt{null}, \textbf{true}, \textbf{false} \,\} \cup \mathit{addr} \;\cup\; \mathbb{N} \\
\mathit{object} & = & \mathit{ClassId} \times (\;\; \mathit{FieldId} \longrightarrow \mathit{val} \;\;) \\
\iota, \iota', .. & \in & \mathit{addr} \\
\mathit{StackId} & = & \{\, \textbf{this} \,\} \cup \mathit{VarId} \cup \mathit{ParId}
\end{array}$$

The Operational Semantics of $\mathcal{F}ocal$ We define $\lfloor a \rfloor_\sigma$, the *interpretation* of an argument $a \in Arg$ in a state σ as follows.

Definition 5 (Interpretation)**.** *For a state $\sigma = (\phi, \chi)$ we define*

$$\begin{array}{lcl}
\lfloor x \rfloor_\sigma & = & \phi(x) \quad (\text{for x} \in \mathit{StackId}) \\
\lfloor \textbf{true} \rfloor_\sigma & = & \textbf{true} \\
\lfloor \textbf{false} \rfloor_\sigma & = & \textbf{false} \\
\lfloor x.f \rfloor_\sigma & = & \chi(\lfloor x \rfloor_\sigma)(f) \\
\lfloor x.fs.f \rfloor_\sigma & = & \chi(\lfloor x.fs \rfloor_\sigma)(f)
\end{array}$$

Here fs is a non-empty **.***-separated list of FieldIds.*

Execution uses module M, and maps a runtime state σ and statements *stmts* (respectively a right hand side rhs) onto a new state σ' (respectively a new heap χ' and a value). We therefore do not give execution rules for things like null-pointer-exception, or stuck execution. This allows us to keep the system simple; it will be easy to extend the semantics to a fully-fledged language.

Definition 6. *Execution of $\mathcal{F}ocal$ statements and expressions is defined in figure 2.2, and has the following shape:*

$$\begin{array}{lcl}
\rightsquigarrow & : & \mathit{Module} \times \mathit{state} \times \mathit{Stmts} \;\longrightarrow\; \mathit{state} \\
\rightsquigarrow & : & \mathit{Module} \times \mathit{state} \times \mathit{Rhs} \;\longrightarrow\; \mathit{heap} \times \mathit{val}
\end{array}$$

Arising and Reachable Configurations Policies need to be satisfied in all configurations which may arise during execution of some program. This leads us the concept of *arising* configuration. Arising configurations allow us to restrict the set of configurations we need to consider. For example, in a program where a class does not export visibility to a field, the constructor initialises the field to say 0, and all method calls increment that field, the arising configurations will only consider states where the field is positive.

A configuration is reachable from another configuration, if the former may be required for the evaluation of the latter after any number of steps.

$$\begin{array}{l}
\mathcal{R}each \;:\; \mathit{Module} \times \mathit{state} \times \mathit{Stmts} \\
\qquad\qquad \longrightarrow \mathcal{P}(\mathit{state} \times \mathit{Stmts})
\end{array}$$

In figure 2 we define the function $\mathcal{R}each$ by cases on the structure of the expression, and depending on the execution of the statement. The set $\mathcal{R}each(M, \sigma, \mathtt{stmts})$ collects all configurations reachable during execution of $\sigma, \mathtt{stmts}$.

Note that the function $\mathcal{R}each(M, \sigma, \texttt{stmts})$ is defined, even when the execution should diverge. This is important, because it allows us to give meaning to capability policies without requiring termination.

We then define $\mathcal{A}rising(M)$ as the set of runtime configurations which may be reached during execution of some initial context (σ_0,$\texttt{stmts}_0$). A context is initial if its heap contains only objects of class `Object`.

Definition 7 (Arising and Initial configurations). *We define the mappings*

$$\begin{array}{lcl} \mathcal{I}nit & : & Module \longrightarrow \mathcal{P}(state \times Stmt) \\ \mathcal{A}rising & : & Module \longrightarrow \mathcal{P}(state \times Stmts) \end{array}$$

as follows:

$$\begin{array}{ll} \mathcal{I}nit(M) = & \{ \ (\ \sigma_0, \mathbf{new}\ c.m(\ \mathbf{new}\ c')\ )\ |\ c, c' \in dom(M) \\ & \quad where\ \sigma_0\ =\ ((\iota, \mathbf{null}), \chi_0), \\ & \quad and\ \chi_0(\iota) = (\texttt{Object}, \emptyset)\ \} \\ \mathcal{A}rising(M) = & \bigcup_{(\sigma, \texttt{stmts}) \in \mathcal{I}nit(M)} \mathcal{R}each(M, \sigma, \texttt{stmts}) \end{array}$$

Initial configuration should be as "minimal" as possible, We therefore construct a heap which has only one object, and execute a method call on a newly created object, with another newly created object as argument.

3. The Specification Language Chainmail

Our specifications and policies are fundamentally two-state assertions. To express the state in which an expression is evaluated, we annotate it with a *t*-subscript. For example, given σ and σ' where $\sigma(x)$=4, and $\sigma'(x)$=3, we have $M, \sigma, \sigma' \models x_{pre} - x_{post} = 1$.

Expressions and Assertions We first define expressions, *Expr*, and assertions *A*, which depend on *one state* only. We allow the use of mathematical operators, like $+$ and $-$, and we use the identifier *f* to indicate functions whose value depends on the state (eg the function `length` of a list). We use the identifier *sR* to indicate predicates whose validity depends on the state (eg the predicate `Acyclic` for a list).

The difference between expressions and arguments is that expressions may express ghost information, which is not stored explicitly in the state σ but can be deduced from it — e.g. the length of a list that is not stored with the list.

Definition 8 (Expressions).

$$\begin{array}{lcl} Expr & ::= & Arg\ \ |\ \ Val\ \ |\ \ Expr + Expr\ \ |\ \ ... \\ & | & f(Expr^*) \\ & | & \mathbf{if}\ Expr\ \mathbf{then}\ Expr\ \mathbf{else}\ Expr \\ funDescr & ::= & \mathbf{function}\ f(\ ParId^*\ )\ \{\ \ Expr\ \ \} \end{array}$$

We now define the values of such expressions, and the validity of one-state assertions as follows:

Definition 9 (Interpretations). *We define the interpretation of expressions,* $\lfloor\cdot\rfloor\ :\ Expr \times Module \times state \rightarrow Value$ *using the notation* $\lfloor\cdot\rfloor_{M,\sigma}$*:*

- $\lfloor val \rfloor_{M,\sigma}$ = *val, for all values* $val\ \in\ Val$.
- $\lfloor a \rfloor_{M,\sigma}$ = $\lfloor a \rfloor_{\sigma}$, *for all arguments* $a\ \in\ Arg$.
- $\lfloor e_1 + e_2 \rfloor_{M,\sigma}$ = $\lfloor e_1 \rfloor_{M,\sigma} + \lfloor e_2 \rfloor_{M,\sigma}$.
- $\lfloor f(e_1, ...e_n) \rfloor_{M,\sigma}$ = $\lfloor Expr[e_1/p_1, ...e_n/p_n] \rfloor_{M,\sigma}$ *where* $M(f) = \mathbf{function}\ f\ (\ p_1 ... p_n)\ \{\ \ Expr\ \}$, *undefined, otherwise.*
- $\lfloor \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \rfloor_{M,\sigma}$
  $=\lfloor\ e_1 \rfloor_{M,\sigma}$, *if* $\lfloor\ e_0 \rfloor_{M,\sigma}$=**true**,
  $=\lfloor\ e_2 \rfloor_{M,\sigma}$, *if* $\lfloor\ e_0 \rfloor_{M,\sigma}$=**false**.
  and undefined, otherwise.

One-state assertions We now define a language of assertions which depend on one state. We introduce three specific predicates: $\mathcal{M}ay\mathcal{A}ffect$ and $\mathcal{M}ay\mathcal{A}ccess$ which we use to model risk, the assertion *Expr*:*ClassId* which expresses class membership, and the assertion *Expr* **obeys** *SpecId*. The two former predicates are *hypothetical*, in that they talk about the potential effect of execution of code, or of the existence of paths to connect two objects. In particular, the $\mathcal{M}ay\mathcal{A}ffect$ predicate ascertains whether its first parameter may execute code which affects the second one, while $\mathcal{M}ay\mathcal{A}ccess$ predicates ascertains whether its first parameter has *any* path to the second one.

Definition 10 (One-state Assertions).

$$\begin{array}{lcl} A & ::= & Expr\ \ |\ \ R(Expr^*) \\ & | & Expr \geq Expr\ \ |\ \ A \land A\ \ |\ \ ... \\ & | & \exists x.A\ \ |\ \ \forall x.A\ \ |\ \ ... \\ & | & Expr\mathbf{:}ClassId \\ & | & \mathcal{M}ay\mathcal{A}ffect\ (\ Expr,Expr) \\ & | & \mathcal{M}ay\mathcal{A}ccess(\ Expr,Expr) \\ & | & Expr\ \ \mathbf{obeys}\ SpcId \\ \\ PredDescr & ::= & \mathbf{predicate}\ R(\ ParId^*\ )\ \{\ \ \ A\ \ \} \end{array}$$

Two state assertions Two-state assertions allow us to compare properties of two different states, and thus say, e.g. that $\texttt{p.balance}_{post} = \texttt{p.balance}_{pre} + 10$. To differentiate between the two states we use the subscripts pre and post.

Definition 11 (Two-state Assertions).

$$\begin{array}{lcl} t & ::= & \mathsf{pre}\ \ |\ \ \mathsf{post}\ \ |\ \ \epsilon \\ B & ::= & A_t \\ & | & Expr_t \geq Expr_t\ \ |\ \ ... \\ & | & \mathcal{N}ew(Expr) \\ & | & B \land B\ \ |\ \ ... \\ & | & \exists x.B\ \ |\ \ \forall x.B\ . \end{array}$$

Given the syntax from above, we can express assertions like

$\forall \texttt{p}.\texttt{p} :_{pre} \texttt{Purse}$.

$\texttt{p.bank} =_{pre} \texttt{RBS} \rightarrow \texttt{p.balance}_{pre} = \texttt{p.balance}_{post}$,

to require that the `balance` of any `Purse` belonging to `RBS` is immutable across the to states. Notice that for legibility, for infix predicates (such as = or :) we annotate the predicate application rather than the assertion, *e.g.* we write $\texttt{p.bank}{=_{pre}}\texttt{RBS}$ to stand for $(\texttt{p.bank=RBS})_{pre}$.

$$
\begin{array}{lcl}
\mathcal{R}each(M, \sigma, \texttt{v:=}\mathbf{new}\,\texttt{c}\mathbf{(}\,\texttt{a}_1, ...\texttt{a}_n\mathbf{)}\,) & = & \{\, (\texttt{v:=}\mathbf{new}\,\texttt{c}\mathbf{(}\,\texttt{a}_1, ...\texttt{a}_n\mathbf{)}\,, \sigma),\ (\mathbf{skip}, \sigma')\} \\
 & & \text{where } M, \sigma, \texttt{v:=}\mathbf{new}\,\texttt{c}\mathbf{(}\,\texttt{a}_1, ...\texttt{a}_n\mathbf{)} \leadsto \sigma' \\
\mathcal{R}each(M, \sigma, \texttt{stmt}\mathbf{;}\,\texttt{stmts}) & = & \mathcal{R}each(M, \sigma, \texttt{stmt}) \cup \mathcal{R}each(M, \sigma', \texttt{stmts}) \\
 & & \text{where } M, \sigma, \texttt{stmt} \leadsto \sigma' \\
\mathcal{R}each(M, \sigma, \texttt{v:=a}) & = & \{(\texttt{v:=a}, \sigma),\ (\mathbf{skip}, \sigma')\} \\
 & & \text{where } M, \sigma, \texttt{v:=a} \leadsto \sigma' \\
\mathcal{R}each(M, \sigma, \texttt{v:=a.m}\mathbf{(}\,\texttt{a}_1, ...\texttt{a}_n\mathbf{)}\,) & = & \{\, (\texttt{v:=a.m}\mathbf{(}\,\texttt{a}_1, ...\texttt{a}_n\mathbf{)}\,, \sigma),\ (\mathbf{skip}, \sigma''')\,\} \cup \mathcal{R}each(M, \sigma', \texttt{stmts}) \\
 & & \text{where } \chi = \sigma\downarrow_1, \text{and } \sigma' = (\mathbf{this} \mapsto \lfloor a_1\rfloor_\sigma, x_1 \mapsto \lfloor a_n\rfloor_\sigma .. x_n \mapsto \lfloor a_n\rfloor_\sigma), \chi) \\
 & & \text{and } \mathcal{M}(M, \chi(\lfloor a_1\rfloor_\sigma)\downarrow_1, \texttt{m}) = ...\mathbf{(}\ \texttt{stmts}; \mathbf{return}\,\texttt{a}\ \mathbf{)} \text{ and} \\
 & & M, \sigma', \texttt{stmts} \leadsto \sigma'' \text{ and } \sigma''' = (\sigma\downarrow_1 [\texttt{v} \mapsto \lfloor a\rfloor_{\sigma''}], \sigma''\downarrow_2) \\
\mathcal{R}each(M, \sigma, \mathbf{skip}) & = & \{\, (\mathbf{skip}, \sigma)\,\} \\
\mathcal{R}each(M, \sigma, \mathbf{if}\,\texttt{a}\,\mathbf{then}\,\texttt{stmts}_1\,\mathbf{else}\,\texttt{stmts}_2\,) & = & \{\, (\mathbf{if}\,\texttt{a}\,\mathbf{then}\,\texttt{stmts}_1\,\mathbf{else}\,\texttt{stmts}_2, \sigma),\ \} \cup \mathcal{R}each(M, \sigma, \texttt{stmts}'') \\
 & & \text{where } \texttt{stmts}'' = \texttt{stmts}_1 \text{ if } \lfloor a\rfloor_\sigma = \mathbf{true},\ \text{otherwise } \texttt{stmts}'' = \texttt{stmts}_2
\end{array}
$$

Figure 2. Reachable Configurations

Policies are expressed in terms of one-state assertions A, A', etc. and two state assertions B, B'' etc.

Policies can have one of the three following forms: 1) invariants of the form A, which require that A holds at all visible states of a program; or 2) $A\,\mathbf{\{}\ \texttt{code}\ \mathbf{\}}\ B$, which require that execution of `code` in any state which satisfies A will lead to a state which satisfies B wrt the original state; or 3) $A\,\{\texttt{any\_code}\}\,B$ which, similar to two state invariants, requires that execution of *any* code in a state which satisfies A will lead to a state which satisfies B.

Definition 12 (Policies)**.**

$$
\begin{array}{lll}
Policy & ::= & A \mid A\,\{\mathit{code}\}\,B \mid\ A\,\{\mathit{any\_code}\}\,B \\
PolSpec & ::= & \mathit{spec}\ SpcId\,\mathbf{\{}\ Policy^*\,\mathbf{\}}
\end{array}
$$

.

Validity of one-state, two-state assertions, and policies We first defined validity of one-state assertions:

Let $\sigma = (\phi, \chi)$ be a state. Then write $\sigma[v{\mapsto}\iota]$ as shorthand for $(\phi[v{\mapsto}\iota], \chi)$.

Definition 13 (Validity of one-state assertions – $\mathcal{M}ay\mathcal{A}ffect$ and $\mathcal{M}ay\mathcal{A}ccess$)**.** *We define the validity an assertion A:*

$\models\ \subseteq\ Module \times state \times Assertion$
using the notation $M, \sigma \models A$:

- $M, \sigma \models \texttt{e}$ *iff* $\lfloor \texttt{e}\rfloor_{M,\sigma} = \mathbf{true}$.
- $M, \sigma \models R(e_1, ...e_n)$ *iff*
  $M, \sigma \models R[e_1/p_1, ...e_n/p_n]$
  where $M(P) = \mathbf{predicate}\ P\ \mathbf{(}\ p_1...p_n\mathbf{)}\ \mathbf{\{}\ \ A\ \mathbf{\}}$,
  undefined, otherwise.
- $M, \sigma \models e_1 \geq e_2$ *iff* $\lfloor e_1\rfloor_{M,\sigma} \geq \lfloor e_2\rfloor_{M,\sigma}$.
- $M, \sigma \models A_1 \wedge A_2$ *iff* $M, \sigma \models A_1$ *and* $M, \sigma \models A_2$.
- $M, \sigma \models \exists x.A$ *iff for some address* ι *and some fresh variable* $z \in VarId$, *we have* $M, \sigma[z \mapsto \iota] \models A[z/x]$
- $M, \sigma \models \forall x.A$ *iff for all addresses* $\iota \in dom(\sigma)$, *and fresh variable* z, *we have* $M, \sigma[z \mapsto \iota] \models A[z/x]$.
- $M, \sigma \models \texttt{e}{:}C$ *iff* $\sigma(\lfloor \texttt{e}\rfloor_{M,\sigma})\downarrow_1 = C$.
- $M, \sigma \models \mathcal{M}ay\mathcal{A}ffect\mathbf{(}\,\texttt{e}, \texttt{e}'\mathbf{)}$ *iff there exists method* $\texttt{m}$, *arguments* $\bar{\texttt{a}}$, *state* σ', *identifier* z, *such that* $M, \sigma[z \mapsto \lfloor \texttt{e}\rfloor_{M,\sigma}], z\,\mathbf{.}\,m(\bar{\texttt{a}}) \leadsto \chi'$, *and* $\lfloor \texttt{e}'\rfloor_{M,\sigma} \neq \lfloor \texttt{e}'\rfloor_{M,\sigma\downarrow_1,\chi'}$.
- $M, \sigma \models \mathcal{M}ay\mathcal{A}ccess(\texttt{e}, \texttt{e}')$ *iff there exist fields* f_1,... f_n, *such that* $\lfloor z.f_1...f_n\rfloor_{M,\sigma[z\mapsto\lfloor \texttt{e}\rfloor_{M,\sigma}]} = \lfloor \texttt{e}'\rfloor_{M,\sigma}$.
- $M, \sigma \models \texttt{e}\,$***obeys*** *PolSpecId iff*
  $\forall\,(\sigma, \mathit{stmts}) \in \mathcal{A}rising(M).\ \forall i \in \{1..n\}.$
  $\forall\,\sigma', \mathit{stmts}'.\ (\sigma', \mathit{stmts}') \in \mathcal{R}each(M, \sigma, \mathit{stmts}).$
  $M, \sigma'[z \mapsto \lfloor e\rfloor_\sigma] \models Policy_i[z/\mathbf{this}]$
  where z is a fresh variable in σ', and where we assume that PolSpecId was defined as
  $\mathit{specification}\ PolSpecId\ \{\ Policy_1, ...Policy_n\ \}$,

We now define validity of two state assertions, ...

Definition 14 (Validity of Two-state assertions)**.** *We define the judgment*

$\models\ \subseteq\ Module \times state \times state \times TwoStateAssertion$
using the notation $M, \sigma, \sigma' \models B$ as follows

- $M, \sigma, \sigma' \models A_t$ *iff* $M, \sigma'' \models A$,
  where $\sigma'' = \sigma$ *if t=*pre*, and* $\sigma'' = \sigma'$ *otherwise.*
- $M, \sigma, \sigma' \models e_t \geq e'_{t'}$, *iff* $\lfloor \texttt{e}\rfloor_{M,\sigma_1} \geq \lfloor \texttt{e}'\rfloor_{M,\sigma_2}$,
  where $\sigma_1 = \sigma$ *if t=*pre*, and* $\sigma_1 = \sigma'$ *otherwise,*
  and $\sigma_2 = \sigma$ *if t'=*pre*, and* $\sigma_2 = \sigma'$ *otherwise.*
- $M, \sigma, \sigma' \models \mathcal{N}ew(\texttt{e})$ *iff* $\lfloor \texttt{e}\rfloor_{M,\sigma'} \in dom(\sigma') \setminus dom(\sigma)$
- $M, \sigma, \sigma' \models B_1 \wedge B_2$ *iff*
  $M, \sigma, \sigma' \models B_1$ *and* $M, \sigma, \sigma' \models B_2$.
- $M, \sigma, \sigma' \models \exists x.B$ *iff for some address* ι *and fresh variable* z, *we have* $M, \sigma[z \mapsto \iota], \sigma'[z \mapsto \iota] \models B[z/x]$.
- $M, \sigma, \sigma' \models \forall x.B$ *iff* $M, \sigma[z \mapsto \iota], \sigma'[z \mapsto \iota] \models B[z/x]$ *holds for all addresses* $\iota \in dom(\sigma)$, *and fresh variable* z.

For example, for states σ_1, σ_2 where $\lfloor \texttt{x.balance}\rfloor_{\sigma_1} = 4$ and $\lfloor \texttt{x.balance}\rfloor_{\sigma 2} = 14$, we have
$M, \sigma_1, \sigma_2 \models \texttt{x.balance}_{post} = \texttt{x.balance}_{pre} + 10$.

We now define adherence to policy, $M, \sigma \models_{pol}$ *Policy*, which ensures that the requirements of *Policy* are satisfied in any context arising from M.

Definition 15 (Adherence to Policies)**.**

- $M, \sigma \models_{pol} A$ *iff* $M, \sigma \models A$
- $M, \sigma \models_{pol} A \{ \texttt{code} \} B$ *iff*
  $(M, \sigma \models A \ \wedge \ M, \sigma, \texttt{stmts} \leadsto \sigma'$
  $\longrightarrow \quad M, \sigma, \sigma' \models B \ )$
- $M, \sigma \models A \{ \texttt{any\_code} \} B$ *iff*
  $\forall \texttt{code}.(\sigma, \texttt{code}) \in \mathcal{A}rising(M) \wedge M, \sigma \models A$
  $\wedge M, \sigma, \texttt{stmts} \leadsto \sigma'$
  $\longrightarrow \quad M, \sigma, \sigma' \models B \ )$

In order to model open systems, require that after linking *any* module with the module at hand, the policy will be satisfied. As stated in [3], "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 M_5 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 Escrow policies are satisfied.

Definition 16 (Classes adhering to Specifications)**.**

- $M \models_{pol}$ *ClassId* ***obeys*** *PolSpecId* *iff*
  $\forall M', \sigma.(\sigma, \_) \in \mathcal{A}rising(M * M').$
  $M, \sigma \models_{pol} o : ClassId \ \rightarrow \ o$ ***obeys*** *PolSpecId*

4. Hoare Logic

We define the Hoare Logic that allows us to prove adherence to policies. In order to reflect that the code to be verified is executed in an open system, and that it calls code whose specification and trustworthiness is unknown to the code being verified, we augment the Hoare triples, so that not only do they guarantee some property to hold *after* execution of the code, but also guarantee that some property is preserved *during* execution of the code.

A Hoare tuple in our system has either the format
$M \vdash A \{ \texttt{stms} \} A' \ \bowtie \ B,$
or the format
$M \vdash A \{ \texttt{stms} \} B' \ \bowtie \ B,$
The former promises that execution of `stms` in any state which satisfies A will lead to a state which satisfies A'. The latter promises that execution of `stms` in any state which satisfies A will lead to a state where the relation of the old and new state is described by B. Both the former and latter tuples also promise that the relation between the initial state, and any of the the intermediate states reached by execution of `stms` will be described by B.

The execution of `stmts` may call methods defined in M, and the predicates appearing in A, A', and B, may use predicates as defined in M. When the module M is implicit from the context we use the shorthand $\vdash A \{ \texttt{stms} \} A' \ \bowtie \ B$.

As is usual in many Hoare logics [1] we introduce *logical variables* into our assertions. We assume that these have the form $\mathfrak{var}$, $\mathfrak{var}$', and that they come from a separate domain. We also assume that there exists a function $\mathfrak{L}vars$, which returns all the logical variables within an assertion. For example $\mathfrak{L}vars(\texttt{p1.balance} = \mathfrak{var}) = \{ \mathfrak{var} \}$. [1]

Definition 17 (Validity of Hoare Tuples)**.**

- $M \models A \{ \texttt{stms} \} A' \ \bowtie \ B$ *iff*
  $\mathfrak{L}vars(A) = \mathfrak{L}vars(A') = \{ \overline{\mathfrak{var}} \} \ \wedge \ \forall M', \sigma, \overline{val}.$
  $(\sigma, \_) \in \mathcal{A}rising(M * M')$
  $\wedge \ M * M', \sigma[\overline{\mathfrak{var}} \mapsto \overline{val}] \models A$
  $\wedge \ M {*} M', \sigma, \texttt{stms} \leadsto \sigma'$
  $\longrightarrow$
  $M {*} M', \sigma'[\overline{\mathfrak{var}} \mapsto \overline{val}] \models A'$
  $\wedge$
  $\forall \sigma'' \in \mathcal{R}each(M, \sigma, \texttt{stmts}).\ M {*} M', \sigma, \sigma'' \models B$
- $M \models A \{ \texttt{stms} \} B' \ \bowtie \ B$ *iff*
  $\mathfrak{L}vars(A) = \mathfrak{L}vars(A') = \{ \overline{\mathfrak{var}} \} \ \wedge \ \forall M', \sigma, \overline{val}.$
  $(\sigma, \_) \in \mathcal{A}rising(M * M')$
  $\wedge \ M * M', \sigma[\overline{\mathfrak{var}} \mapsto \overline{val}] \models A$
  $\wedge \ M {*} M', \sigma, \texttt{stms} \leadsto \sigma'$
  $\longrightarrow$
  $M {*} M', \sigma[\overline{\mathfrak{var}} \mapsto \overline{val}], \sigma'[\overline{\mathfrak{var}} \mapsto \overline{val}] \models B'$
  $\wedge$
  $\forall \sigma'' \in \mathcal{R}each(M, \sigma, \texttt{stmts}).\ M {*} M', \sigma, \sigma'' \models B$

Note that the definition from above does not support the use of logical variables in the invariant part of the tuple, B. Even though it would have been possible to accommodate for this in our formal model, it would slightly complicate the expositions, and so far we have not found a need to do that.

4.1 Hoare Rules

We define the Hoare rules in figure 3 for the language constructs, while in figure 4 we give the rules for framing, the rules for consequence, and rules about invariants preserved during execution of a statement.[2]

We first consider the rules from figure 3: The rules (VARASG) and (FIELDASG) are not surpising. The annotations $\_{pre}$ and $\_{post}$ explain the use of $\texttt{a}_{pre}$, and allow us to talk in the postcondition about values in the pre-state. For example, we would obtain

$\mathbf{true}$
$\{ \texttt{this.f=this.f+3} \}$
$\texttt{this.f} = \texttt{this.f}_{pre} + 3$.
$\bowtie$
$\mathbf{true}$

The rules (COND-1) and (COND-2) describe conditional statements, and are standard.

The rule (METH-CALL-1) describes method call. [3]

[1] Make sure we have said earlier that valstands for a value.Just noticed that I sometimes uses v for variables, and some times for values. Arghh

[2] Notice that we have no rule for object creation; these would like rules for method calls; while they do not pose special challenges, they would increase the size of our system and we leave this to further work.

[3] We have no invariant part in the spec of a method, but it would not be difficult to extend the system to support this.

$$\dfrac{}{\begin{array}{l}\vdash \mathbf{true}\,\{\,\mathbf{var}\,\texttt{v:=a}\,\}\;\texttt{v} = \texttt{a}_{pre}\ \bowtie\ \mathbf{true}\\ \vdash \mathbf{true}\,\{\,\texttt{v:=a}\,\}\;\texttt{v} = \texttt{a}_{pre}\ \bowtie\ \mathbf{true}\end{array}}\ (\textsc{varAsg})$$

$$\dfrac{}{\vdash \mathbf{true}\,\{\,\mathbf{this}.\texttt{f:=a}\,\}\;\mathbf{this}.\texttt{f} = \texttt{a}_{pre}\ \bowtie\ \mathbf{true}}\ (\textsc{fieldAsg})$$

$$\dfrac{\begin{array}{l}A \rightarrow_M \texttt{cond}\\ \vdash A\,\{\,\texttt{stmts}_1\,\}\,B\ \bowtie\ B'\end{array}}{\vdash A\,\{\,\mathbf{if}\,\texttt{cond}\,\mathbf{then}\,\texttt{stmts}_1\,\mathbf{else}\,\texttt{stmts}_2\,\}\,B\ \bowtie\ B'}\ (\textsc{cond-1})$$

$$\dfrac{\begin{array}{l}A \rightarrow_M \neg\texttt{cond}\\ \vdash A\,\{\,\texttt{stmts}_2\,\}\,B\ \bowtie\ B'\end{array}}{\vdash A\,\{\,\mathbf{if}\,\texttt{cond}\,\mathbf{then}\,\texttt{stmts}_1\,\mathbf{else}\,\texttt{stmts}_2\,\}\,B\ \bowtie\ B'}\ (\textsc{cond-2})$$

$$\dfrac{}{\vdash A\,\{\,\mathbf{skip}\,\}\,A\ \bowtie\ \mathbf{true}}\ (\textsc{skip})$$

$$\dfrac{M(S) = \mathbf{spec}\,S\,\{\ \overline{Policy}, A\{\ \texttt{this.m(par)}\ \}\ B, \overline{Policy'}\ \}}{\vdash \texttt{x}\,\mathbf{obeys}\,S \wedge A[\texttt{x/this}, \texttt{y/par}]\,\{\,\texttt{v} := \texttt{x.m(y)}\,\}\,B[\texttt{x/this}, \texttt{y/par}, \texttt{v/res}]\ \bowtie\ \mathbf{true}}\ (\textsc{meth-call-1})$$

$$\dfrac{\begin{array}{l}B \equiv \forall \texttt{z} :_{pre} \texttt{Object}.\ \mathcal{MayAccess}(\texttt{v},\texttt{z}) \rightarrow (\,\mathcal{MayAccess}_{pre}(\texttt{x},\texttt{z})\ \vee\ \mathcal{MayAccess}_{pre}(\texttt{y},\texttt{z})\,)\\ B' \equiv \forall \texttt{z},\texttt{u} :_{pre} \texttt{Object}.\ (\,\mathcal{MayAccess}(\texttt{u},\texttt{z}) \rightarrow\\ \qquad\qquad (\mathcal{MayAccess}_{pre}(\texttt{u},\texttt{z})\ \ \vee\\ \qquad\qquad\quad (\,(\mathcal{MayAccess}_{pre}(\texttt{x},\texttt{z}) \vee \mathcal{MayAccess}_{pre}(\texttt{y},\texttt{z})) \wedge\\ \qquad\qquad\quad\ (\mathcal{MayAccess}_{pre}(\texttt{x},\texttt{u}) \vee \mathcal{MayAccess}_{pre}(\texttt{y},\texttt{u}))\,)\,)\ \ )\end{array}}{\vdash \texttt{true}\ \{\,\texttt{v} := \texttt{x.m(y)}\,\}\,B\ \bowtie\ B'}\ (\textsc{meth-call-2})$$

$$\dfrac{\begin{array}{l}\vdash A\,\{\,\texttt{x.m(y)}\,\}\,\mathbf{true}\ \bowtie\ \forall \texttt{z}.(\,\mathcal{MayAffect}(\texttt{z},A') \rightarrow B'(\texttt{z})\,)\ \wedge\\ \qquad\qquad \forall \texttt{z}.(\,(\mathcal{MayAccess}_{pre}(\texttt{x},\texttt{z}) \vee \mathcal{MayAccess}_{pre}(\texttt{y},\texttt{z}) \vee \mathcal{New}(\texttt{z})\,)\ \rightarrow\ \neg B'(\texttt{z})\,)\end{array}}{\vdash A \wedge A'\,\{\,\texttt{x.m(y)}\,\}\,A'\ \bowtie\ \mathbf{true}}\ (\textsc{frame-methCall})$$

$$\dfrac{\vdash A\,\{\,\texttt{stmts}_1\,\}\,B_1\ \bowtie\ B' \qquad \vdash A_2\,\{\,\texttt{stmts}_2\,\}\,B_2\ \bowtie\ B' \qquad A, B_1 \rightarrow_M \mathbf{true}, A_2 \qquad B_1, B_2 \rightarrow_M B}{\vdash A\,\{\,\texttt{stmts}_1;\,\texttt{stmts}_2\,\}\,B\ \bowtie\ B'}\ (\textsc{sequence})$$

Figure 3. Hoare Logic – Basic rules of the language – we assume that the module M is globally given

On the other hand, rule (METH-CALL-2) is unusual in a Hoare logic setting; it expresses that "only connectivity begets connectivity" . The terms was coined by Mark Miller and is used widely in the capabilities literature. To our knowledge, this property has not been expressed in a Hoare logic. The reason, is, we believe, that Hoare logics so far have been developed with the closed world assumption, in the sense that all methods (or functions) called come from code which has a specification, and which has been verified.

The rule (FRAME-METHCALL) is also unusual; note that its precondition is **true**. This means that we make no assumptions about the receiver of the method call; this allows us to reason in an *open* setting. Even though we do not know what the behaviour method `m` will be, we still have some conditions which can guarantee that A' will be preserved. These conditions are that anything that was accessible from the receiver `x` or argument of `z` at the time of the method call, or anything that is newly created during execution of the method body, does not satisfy the prerequisites necessary to affect A'.[4]

The last rule in figure 3 is (SEQUENCE). It requires that the precondition and the postcondition of the first statements, *i.e.* A and B_1, imply the precondition of the second statements, ie A_2, and that the combined effects described by the two-state assertion in the postconditions of $\texttt{stmts}_1$ and $\texttt{stmts}_2$, B_1 followed by B_2, imply the postcondition of the sequence, *i.e.* B.

The standard entailment, *i.e.* $A\ \rightarrow_M\ A'$, guarantees that any state which satisfies A also satisfies A'. We extend the notion to cater for two state assertions, and have three new

[4] Notes that $\sigma' \in \mathcal{R}each(M, \sigma, \texttt{stmts})$ is a shorthand for $\sigma'.(\sigma', \_') \in \mathcal{R}each(M, \sigma, \texttt{stmts})$.

$$\text{(FRAME-GENERAL)}\quad \dfrac{\begin{array}{l}\vdash A\,\{\,\texttt{stmts}\,\}\,B \;\bowtie\; B' \\ A \rightarrow_M \texttt{stmts}\,\#\,A' \qquad A \rightarrow_M \texttt{stmts}\,\#\!\#\,A''\end{array}}{\vdash A \wedge A'\,\{\,\texttt{stmts}\,\}\,B \wedge A' \;\bowtie\; B' \wedge A''}$$

$$\text{(CONJ)}\quad \dfrac{\begin{array}{l}\vdash A_1\,\{\,\texttt{stmts}\,\}\,B_1 \;\bowtie\; B_3 \\ \vdash A_2\,\{\,\texttt{stmts}\,\}\,B_2 \;\bowtie\; B_4\end{array}}{\vdash A_1 \wedge A2\,\{\,\texttt{stmts}\,\}\,B_1 \wedge B_2 \;\bowtie\; B_3 \wedge B_4}$$

$$\text{(CONS-1)}\quad \dfrac{\begin{array}{ll}\vdash A\,\{\,\texttt{stmts}\,\}\,B \;\bowtie\; B' & A' \rightarrow_M A \\ B \rightarrow_M B'' & B' \rightarrow_M B'''\end{array}}{\vdash A'\,\{\,\texttt{stmts}\,\}\,B'' \wedge B' \;\bowtie\; B'''}$$

$$\text{(CONS-2)}\quad \dfrac{\begin{array}{l}\vdash A\,\{\,\texttt{stmts}\,\}\,B \;\bowtie\; B'' \\ A', B' \rightarrow_M A, \mathbf{true}\end{array}}{\vdash A'\,\{\,\texttt{stmts}\,\}\,B' \rightarrow B \;\bowtie\; B''}$$

$$\text{(CONS-3)}\quad \dfrac{\begin{array}{l}\vdash A\,\{\,\texttt{stmts}\,\}\,B \;\bowtie\; B' \\ A, B \rightarrow_M \mathbf{true}, A'\end{array}}{\vdash A\,\{\,\texttt{stmts}\,\}\,A' \;\bowtie\; B'}$$

$$\text{(CONS-4)}\quad \dfrac{\begin{array}{l}\vdash A\,\{\,\texttt{stmts}\,\}\,A' \;\bowtie\; B' \\ A, A' \rightarrow_M B\end{array}}{\vdash A\,\{\,\texttt{stmts}\,\}\,B \;\bowtie\; B'}$$

$$\text{(CODE-INVAR-1)}\quad \dfrac{M(S) \equiv \;\mathbf{spec}\, S\,\{\; \overline{Policy}, P, \overline{Policy'}\;\}}{\vdash \mathbf{true}\,\{\,\texttt{stmts}\,\}\,\mathbf{true} \;\bowtie\; \forall \texttt{x}.(\,\texttt{x}\,\mathbf{obeys}\,S \rightarrow P[\texttt{this/x}]\,)}$$

$$\text{(CODE-INVAR-2)}\quad \dfrac{}{\vdash e\,\mathbf{obeys}\,S\,\{\,\texttt{stmts}\,\}\,\mathbf{true} \;\bowtie\; e_{pre}\,\mathbf{obeys}\,S}$$

Figure 4. Hoare Logic – we assume that the module M is globally given

forms of entailment, described in Definition 18. The requirement $A, B_1 \rightarrow_M \mathbf{true}, A_2$ guarantees that for any pair of states if the former states satisfies A and the two together satisfy B_1, then the second state will also satisfy A_2, *c.f.* Definition 18.3. The requirement $B_1, B_2 \rightarrow_M B$ guarantees for any three states, if the first two together satisfy B_1, and the second and third together satisfy B_2, then the first and third will satisfy B, *c.f.* Definition 18.5. For example, with 18.3 we have $\texttt{x} = 5, \texttt{x}_{post} = \texttt{x} + 2 \rightarrow_M \mathbf{true}, \texttt{x} = 7$, while with 18.5 we have $\texttt{x}_{post} = \texttt{x}{+}4, \texttt{x}_{post} = \texttt{x}{+}2 \rightarrow_M \texttt{x}_{post} = \texttt{x}{+}6$ for any module M.

Definition 18 (Entailment)**.**

1. $A \rightarrow_M A'$ *iff*
   $\forall\sigma.\; M, \sigma \models A \rightarrow M, \sigma \models A'$
2. $B \rightarrow_M B'$ *iff*
   $\forall\sigma, \sigma'.\; M, \sigma, \sigma' \models B \rightarrow M, \sigma, \sigma' \models B'$
3. $A, B \rightarrow_M A', A''$ *iff*
   $\forall\sigma, \sigma'.\, \sigma \models A \,\wedge\, \sigma, \sigma' \models B \;\longrightarrow\; \sigma \models A' \,\wedge\, \sigma' \models A''$
4. $A, A' \rightarrow_M B$ *iff*
   $\forall\sigma, \sigma'.\, \sigma \models A \,\wedge\, \sigma' \models A' \;\longrightarrow\; \sigma, \sigma' \models B$
5. $B, B' \rightarrow_M B''$ *iff*
   $\forall\sigma, \sigma', \sigma''.\, \sigma, \sigma' \models B \wedge \sigma', \sigma'' \models B' \;\longrightarrow\; \sigma, \sigma'' \models B''$

We now turn our attention to the structural rules from figure 4.

Rule (FRAME-GENERAL) allows us to frame onto a tuple any assertion that has not been affected by the code. . For this, we need two notions of some code being disjoint from an assertion:

Definition 19 (Disjointness)**.**

- $M, \sigma \models \mathit{stms} \,\#\!\#\, A$ *iff*
  $M, \sigma \models A \;\wedge\; \forall\sigma' \in \mathcal{R}each(M, \mathit{stmts}, \sigma).\; M, \sigma' \models A.$
- $M, \sigma \models \mathit{stms} \,\#\, A$ *iff*
  $M, \sigma \models A \;\wedge\; M, \sigma, \mathit{stms} \rightsquigarrow \sigma' \;\rightarrow\; M, \sigma' \models A.$

For example $\texttt{x=7}\,\#\,\texttt{x:=x+1; x:=x-1}$ holds for all states and modules, but $\texttt{x=7}\,\#\!\#\,\texttt{x:=x+1; x:=x-1}$ never holds. In general, framing is an undecidable problem, but we can prove some very basic properties, eg that assignment to a variable does not affect all other variables, nor other paths. Note, that in order to express this property we are making use of logical variables.

Lemma 1. *For all modules M, and states σ,*

- *If $\texttt{x}$ and $\texttt{y}$ are textually different variables, then* $M, \sigma \models \texttt{x=a} \,\#\!\#\, \texttt{y := a'}$.
- *If $\texttt{x}$ is not a prefix of the path $\texttt{p}$, then* $M, \sigma \models \texttt{p.f=a} \,\#\!\#\, \texttt{x := a'}$.
- *If $M, \sigma \models \mathit{stms} \,\#\!\#\, A$ then $M, \sigma \models \mathit{stms} \,\#\, A$.*

The rule (CONJ) allows us to combine different Hoare tuples for the same code, and follows standard Hoare logics.

Interestingly, our system has *four* rules of consequence. The fist rule, (CONS-1), is largely standard, as it allows us to strengthen the precondition A, and weaken the postcondition B, and invariant B'. A novelty of this rule, however, is that it allows the invariant to be conjoined to the postcondition; this is sound, because the invariant is guaranteed to hold throughout execution of the code, and thus also after it.

For (CONS-1) we use the entailment $A \rightarrow_M A'$, which guarantees that any state which satisfied A also satisfies A', and that of the form $B \rightarrow_M B'$ which guarantees that any pair of states which together satisfy B also satisfy B'. This is described in Definition 18.

The next rule, (CONS-2), is unusual, in that it allows us to *weaken* the precondition, while adding a hypothesis B' to the postcondition, such that the original postcondition, B, is only guaranteed if B' holds. The rule is sound, because we also require that the new precondition A' together with the new postcondition B' guarantee that the original precondition holds in the pre-state. The judgment $A, B \rightarrow_M A', A''$ is defined in in Definition 18. For example, we can use this rule to take

$\texttt{p1}$ **obeys** $Purse$
  $\{\,\texttt{p2:=p1.sprout}\}$
$\texttt{p2}$ **obeys** $Purse$
$\bowtie$
$\mathbf{true}$

and deduce that

$\mathbf{true}$
  $\{\,\texttt{p2:=p1.sprout}\}$
$\texttt{p1}_{pre}$ **obeys** $Purse \rightarrow \texttt{p2}$ **obeys** $Purse$.
$\bowtie$
$\mathbf{true}$

The next two rules, (CONS-3) and (CONS-4), allow us to swap between tuples where the postcondition is a one-state assertion, *i.e.* $\vdash A\,\{\,\texttt{stms}\,\}\,A' \bowtie B'$ and that where the postcondtion is a one state assertion, *i.e.* $\vdash A\,\{\,\texttt{stms}\,\}\,B \bowtie B'$.

The following lemma is an example entailment.

Lemma 2. *For all modules M:*
$\mathcal{M}ay\mathcal{A}ccess(x, y) \wedge \mathcal{M}ay\mathcal{A}ccess(y, z) \rightarrow_M \mathcal{M}ay\mathcal{A}ccess(x, z)$.

The two last rules in 4 are concerned with adherence to specification.

The rule (CODE-INVAR-1) expresses that throughout execution of any code, in all intermediate states, for any variable x for which we know that it **obeys** a specification S, we know that it satisfies any of S's stated policies.

The rule (CODE-INVAR-2) guarantees that any term e which has been shown to be pointing to an object which **obeys** a specification S will continue satisfying the specification throughout execution of any `stms`.

4.2 Soundness

We first demonstrate that judgments made in the context of a module are preserved when we link a larger module. In lemma 3, we state that entailment is preserved by linking:

Lemma 3.

- $A \rightarrow_M A'$ *implies that* $A \rightarrow_{M*M'} A'$.
- $B \rightarrow_M B'$ *implies that* $B \rightarrow_{M*M'} B'$
- $A, A' \rightarrow_M B$ *implies that* $A, A' \rightarrow_{M*M'} B$
- $B, B' \rightarrow_M B''$ *implies that* $B, B' \rightarrow_{M*M'} B''$

In lemma 1 we state that derivability and validity of Hoare tuples is preserved for larger modules

Theorem 1 (Linking preserves derivations and validity)**.** *For all modules M, M'.*
- *If* $M \vdash A\,\{\,\mathit{stms}\,\}\,A' \bowtie B$ *, then* $M{*}M' \vdash A\,\{\,\mathit{stms}\,\}\,A' \bowtie B$.
- *If* $M \models A\,\{\,\mathit{stms}\,\}\,A' \bowtie B$*, then* $M{*}M' \models A\,\{\,\mathit{stms}\,\}\,A' \bowtie B$

We now define what it means for a method body, and a class definition to adhere to its specification

We say that a method m defined a class C adheres to is specification,
  $M \vdash \texttt{C}, \texttt{m}$
if we able to show that the body of m when executed in a state that satisfies A, the difference between the initial and final state is described by B, and will preserve B', where A and B' and B are the method's pre, postcondition, and invariant. Moreover, we say that a class adheres to its specification
  $M \vdash \texttt{C}$
of all its methods adhere to their specification. Finally, a module adheres to its specification,
  $M \vdash M$
if all the classes in Madhere to their specifications.

Definition 20 (Proving code's adherence to specification)**.**

- $M \vdash \texttt{C}, m$ *iff*
  for all method identifiers m*, and for all A and B' such that $Spec(M, \texttt{C}) = S$ and*
  $M(S) =$ **spec** $S\,\{\,\overline{Policy}, A\{\,\mathit{this.m(par)}\,\}\,B, \overline{Policy'}\,\}$
  we can prove that
  $M \vdash A \wedge$ **this** ***obeys*** $S\,\{\,\mathit{stmts}\,\}\,B[a/res] \bowtie \mathbf{true}$
  and where the method body for m C *is defined in M as*
  method $m($ par $)$ **{** $stmts$**;** **return** a **}** .
- $M \vdash \texttt{C}$ *iff* $M \vdash \texttt{C}, m$
  for all methods from C
- $M \vdash M$ *iff*
  $M \vdash \texttt{C}$ *for all classes* C *from* M

Below we are defining and proving the soundness of our Hoare logic. Note that we do not require that $M \vdash M$, because we do not model object creation. If we had object creation in our system, we would have needed that requirement, and the proof of soundness would have required slightly more complex proof techniques such as a generation lemma, or double induction.

Theorem 2 (Soundness of the Hoare Logic)**.** *For all modules M, code* `stms` *and assertions A, A' and B and B' ,*
- *If* $M \vdash A\,\{\,\mathit{stms}\,\}\,A' \bowtie B$
  then $M \models A\,\{\,\mathit{stms}\,\}\,A' \bowtie B$.
- *If* $M \vdash A\,\{\,\mathit{stms}\,\}\,B' \bowtie B$
  then $M \models A\,\{\,\mathit{stms}\,\}\,B' \bowtie B$.

Proof. Fix the module M. Then, the proof proceeds by induction on the judgement $M \vdash \_\,\{\,\_\,\}\,\_ \bowtie \_$, which is inductively characterised by the rules of Figure 3 and Figure 4. We have one case to consider, for each of the rules.

Case (VARASG), (FIELDASG), (COND-1) and (COND-2) all follow trivially from the operational semantics of $\mathcal{F}ocal$; the latter two cases also require application of the induction hypothesis.

Case (METH-CALL-1) follows from the definition of Hoare tuple validity (Definition 17) and that of the **obeys** predicate (see Definition 13).

Case (METH-CALL-2) expresses the basic axiom of object-capability systems that "only connectivity begets connectivity" [2], and follows from the operational semantics of $\mathcal{F}ocal$ and the definitions of validity for the $\mathcal{M}ay\mathcal{A}ccess$ predicate (see Definition 13).

Case (FRAME-METHCALL) Is similar to (METH-CALL-2) in that it expresses a basic axiom of object-capability languages, namely that in order to cause some visible effect, one must have access to an object able to perform the effect. Coupled with "only connectivity begets connectivity", this implies that a method can cause some effect only if the caller has (transitive) access to some object able to cause the effect (including perhaps the callee).

Case (SEQUENCE) follows from the definition of $\mathcal{R}each(M, \sigma, \texttt{code}_1\texttt{; code}_2)$ and the definition of validity of Hoare tuples (Definition 17).

Case (FRAME-GENERAL) Follows by the definition of $\#$ and $\#\!\#$.

Case (CONS-1) follows from the definition of entailment (Definition 18) and the fact that $(\sigma, \texttt{stms}) \in \mathcal{R}each(M, \sigma, \texttt{stms})$.

Case (CONS-2) follows because $\sigma, \sigma' \models Q' \rightarrow Q$ if and only iff $\sigma, \sigma' \models Q$ assuming $\sigma, \sigma' \models Q'$.

Case (CONS-3) and (CONS-4) follow straightforwardly from the definition of entailment and Hoare tuple validity.

Case (CODE-INVAR-1) follows because the definition of policy satisfaction for one-state-assertions A requires that A holds for all internally-reachable states σ' via $\mathcal{R}each$.

Case (CODE-INVAR-2) follows straightforwardly from the definition of Hoare tuple validity and 2-state-assertion validity.

$\square$

5. Proof of `Escrow:deal`

We now outline the most salient steps from the proof of the `Escrow`. Note that out formally defined language does not support returning from the inside of a method - we did this to simplify the Hoare rules. Therefore, in Figure 5 we re-write the mothod `deal` so that it obeys this syntactic restriction.

5.1 Preliminaries

We first create some admissible rules, useful for our reasoning.

Firstly, because logical variables cannot be assigned to, we have that $\vdash \mathfrak{var}\ \{\ \texttt{stmts}\ \}\ \textbf{true}\ \bowtie\ \mathfrak{var} = \mathfrak{var}_{pre}$ for any `stmts`; therefore, the following rules are admissible for any logical variable $\mathfrak{var}$, and specification S:

(CODE-INAVR-3)
$$\frac{}{\vdash \mathfrak{var}\ \textbf{obeys}\ S\ \{\ \texttt{code}\ \}\ \textbf{true}\ \bowtie\ \mathfrak{var}\ \textbf{obeys}\ S}$$

Similarly, through application of (FRAME-GENERAL), if $\texttt{z} \neq \texttt{x}$, we get $\vdash \texttt{z} = \mathfrak{var}\ \{\ \texttt{x:=rhs}\ \}\ \texttt{z} = \mathfrak{var}\ \bowtie\ \textbf{true}$, which also gives that $\vdash \textbf{true}\ \{\ \texttt{x:=rhs}\ \}\ \texttt{z} = \texttt{z}_{pre}\ \bowtie\ \textbf{true}$. Then, by (CODE-INVAR-2) and (CONS-1) we obtain that

(OBEYS-INVAR)
$$\frac{\texttt{z} \neq \texttt{x}}{\vdash \texttt{z}\ \textbf{obeys}\ S\ \{\ \texttt{x:=rhs}\ \}\ \textbf{true}\ \bowtie\ \texttt{z}\ \textbf{obeys}\ S}$$

5.2 First Step

The pre and postconditions for the first line from the code, ie for line 4 from Figure 5 are described in figure 6. Drawing on the `Pol_sprout` policy of the `ValidPurse` specification, this step is obtained as follows:

Firstly, by application of (OBEYS-IVAR) and (CONS-4) we obtain

(0)
$$\begin{array}{l}
\texttt{true} \\
\quad \{\ \texttt{escrowMoney} := \texttt{sellerMoney.sprout}\} \\
\texttt{sellerMoney}_{pre}\ \textbf{obeys}\ \texttt{ValidPurse}\ \rightarrow \\
\qquad \texttt{sellerMoney}\ \textbf{obeys}\ \texttt{ValidPurse} \\
\bowtie \\
\textbf{true}
\end{array}$$

.

Then, from the specification of `sprout`in `ValidPurse`, and the rule (METH-CALL-1) we obtain that

(1)
$$\begin{array}{l}
\texttt{sellerMoney}\ \textbf{obeys}\ \texttt{ValidPurse} \\
\quad \{\ \texttt{escrowMoney} := \texttt{sellerMoney.sprout}\} \\
\texttt{escrowMoney}\ \textbf{obeys}\ \texttt{ValidPurse}\ \wedge \\
CanTrade(\texttt{escrowMoney}, \texttt{sellerMoney})\ \wedge \\
\forall\, p :_{pre} \texttt{GoodPrs}.p.\texttt{balance} = p.\texttt{balance}_{pre} \\
\bowtie \\
\textbf{true}
\end{array}$$

Then, from (1), and application of (CONS-2), we obtain

(2)
$$\begin{array}{l}
\textbf{true} \\
\quad \{\ \texttt{escrowMoney} := \texttt{sellerMoney.sprout}\} \\
\texttt{sellerMoney}_{pre}\ \textbf{obeys}\ \texttt{ValidPurse} \rightarrow \\
\qquad (\ \texttt{escrowMoney}\ \textbf{obeys}\ \texttt{ValidPurse}\ \wedge \\
\qquad\ CanTrade(\texttt{escrowMoney}, \texttt{sellerMoney})\ \wedge \\
\qquad\ \forall\, p \in_{pre} \texttt{GoodPrs}.\ p.\texttt{balance} = p.\texttt{balance}_{pre}\ ) \\
\bowtie \\
\textbf{true}
\end{array}$$

Also, by application of (CODE-INVAR-1), and the specification of `ValidPurse`, we have that

```
method deal(  )
{
  //setup and validate Money purses
  escrowMoney := sellerMoney.sprout
  res := escrowMoney.deposit(0, sellerMoney)
  if res then {
    res :=  buyerMoney.deposit(0, escrowMoney)
    if res then {
      res := escrowMoney.deposit(0, buyerMoney)
      if res then {
        // set up and validate Goods purses
        escrowGoods := buyerGoods.sprout
        res := escrowGoods.deposit(0, buyerGoods)
        if res then {
          res := sellerGoods.deposit(0, escrowGoods)
          if res then {
            res :=  escrowGoods.deposit(0, sellerGoods)
            if res then {
              // start the actual exchange
              res := escrowMoney.deposit(price, buyerMoney)
              if res then {
                res := escrowGoods.deposit(amt, sellerGoods)
                if res then{
                  // transfer from the two escrows to two accounts
                  sellerMoney.deposit(price, escrowMoney)
                  buyerGoods.deposit(amt, escrowGoods)
                } else {
                  // undo the transaction
                  buyerMoney.deposit(price, escrowMoney)
                }
              } else skip
            } else skip
          } else skip
        } else skip
      } else skip
    } else skip
  }
  return res
}
```

Figure 5. Revised `deal` method expressed without `return` statements

$$\begin{array}{l}
\texttt{true} \\
\qquad \{\ \texttt{var escrowMoney := sellerMoney.sprout}\ \} \\
\texttt{sellerMoney}_{pre}\ \mathbf{obeys}\ \texttt{ValidPurse} \longrightarrow (\ \texttt{escrowMoney}\ \mathbf{obeys}\ \texttt{ValidPurse}\ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{CanTrade}(\texttt{escrowMoney}, \texttt{sellerMoney})\ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{escrowMoney.balance} = 0\ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \forall p \in_{pre} \texttt{GoodPrs}.p.\texttt{balance}_{pre} = p.\texttt{balance}\ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{sellerMoney}\ \mathbf{obeys}\ \texttt{ValidPurse}\ )\quad \wedge \\
\ \forall p :_{pre} \texttt{GoodPrs}.(p.\texttt{balance}_{pre} = p.\texttt{balance}\ \vee\ \mathcal{M}ay\mathcal{A}ccess_{pre}(\texttt{sellerMoney}, p))\qquad \wedge \\
\ \forall z :_{pre} \texttt{Object}.\ (\mathcal{M}ay\mathcal{A}ccess(\texttt{escrowMoney}, z)\ \longrightarrow\ \mathcal{M}ay\mathcal{A}ccess_{pre}(\texttt{sellerMoney}, z))\quad \wedge \\
\ \forall z, y :_{pre} \texttt{Object}.\ (\mathcal{M}ay\mathcal{A}ccess(\ z, y\ )\ \longrightarrow \\
\qquad\qquad (\mathcal{M}ay\mathcal{A}ccess_{pre}(\ z, y\ )\ \vee\ \mathcal{M}ay\mathcal{A}ccess_{pre}(\ \texttt{sellerMoney}, y\ ) \wedge \mathcal{M}ay\mathcal{A}ccess_{pre}(\ \texttt{sellerMoney}, z\ )) \\
\ \bowtie \\
\ \texttt{true}
\end{array}$$

Figure 6. Hoare tuple for first step in `deal`

(3)
$$\begin{array}{l}
\mathbf{true} \\
\quad \{\ \texttt{escrowMoney} := \texttt{sellerMoney.sprout}\ \} \\
\mathbf{true} \\
\bowtie \\
\forall\, p \in_{pre} \texttt{GoodPrs}, o : \texttt{Object}. \\
\quad (\ \mathcal{M}ay\mathcal{A}ffect(o, p.\texttt{balance}) \rightarrow \mathcal{M}ay\mathcal{A}ccess(o, p)\ )
\end{array}$$

By application of (METH-CALL-2) and (FRAME-METH-CALL) and (3) we obtain

(4)
$\mathbf{true}$
$\quad\{\ \texttt{escrowMoney} := \texttt{sellerMoney.sprout}\}$
$\mathbf{true}$
$\bowtie$
$\forall p \in_{pre} \texttt{GoodPrs}$
$\quad(\,p.\texttt{balance} = p.\texttt{balance}_{pre} \vee$
$\quad\ \mathcal{MayAccess}_{pre}(\texttt{sellerMoney}, o)\,)$

Finally, by application of (METH-CALL-2) we obtain
(5)
$\mathbf{true}$
$\quad\{\ \texttt{escrowMoney} := \texttt{sellerMoney.sprout}\}$
$\mathbf{true}$
$\bowtie$
$\forall z, y :_{pre} \texttt{Object}.\,(\,\mathcal{MayAccess}(\,z, y\,)\ \longrightarrow$
$\quad(\,\mathcal{MayAccess}_{pre}(\,z, y\,) \vee$
$\quad\ \mathcal{MayAccess}_{pre}(\,\texttt{sellerMoney}, y\,) \wedge$
$\quad\ \mathcal{MayAccess}_{pre}(\,\texttt{sellerMoney}, z\,)\,)$

By application of (CONS-1), and (CONJ) on (0), (2), (4), and (5), we obtain the pre-postconditions from Figure 5.

5.3 Second Step

The pre and postconditions for the second step are described in figure 7. The main differences between figures 6 and 7 are a reflection of the differences between the policies `Pol_sprout` and `Pol_deposit_1` and `Pol_deposit_2` in the `ValidPurse` specification. Functionally, `deposit` may succeed or fail, indictated by its return value `res`, while `sprout` always succeeds; `deposit` may change the balances of participant purses, while `sprout` may not.

Crucially for us, the trust essentially the same in both cases:
$\texttt{src}\ \mathbf{obeys}_{pre}\texttt{ValidPurse} \wedge\ \texttt{CanTrade(this,src)}_{pre}$
and the risk is very similar — slightly more complex for `deposit` which may modify the two purses:
$\forall\texttt{p.(p}\ \mathbf{obeys}_{pre}\texttt{ValidPurse} \wedge \texttt{p} \notin \{\texttt{this,src}\} \rightarrow$
$\quad\texttt{p.balance=p.balance}_{pre}\texttt{)} \wedge$
but otherwise may not increase risk:
$\forall\texttt{o:}_{pre}\texttt{Object}.\forall\texttt{p}\ \mathbf{obeys}_{pre}\texttt{ValidPurse}.\mathcal{MayAccess}\texttt{(o,p)} \rightarrow$
$\quad\mathcal{MayAccess}_{pre}\texttt{(o,p))}$
Thus, the reasoning for this step can be justified in similar ways to those that from figure 6.

5.4 Step 1 and Step 2 Establish Mutual Trust

When we combine step 1 and step 2 we obtain the Hoare tuple from figure 8. Here we make use of the results from figure 6 and figure 7, and combine them through the (SEQUENCE) rule. For example, we use our invariants entailment $\longrightarrow_M$, whereby for any module M:
$\forall z :_{pre} \texttt{Object}.\,(\,\mathcal{MayAccess}(\texttt{escrowMoney}, z) \rightarrow$
$\qquad\mathcal{MayAccess}_{pre}(\texttt{sellerMoney}, z)\,),$
$\forall z :_{pre} \texttt{Object}.\,(\,\mathcal{MayAccess}(\texttt{sellerMoney}, z)\ \rightarrow$
$\qquad\mathcal{MayAccess}_{pre}(\texttt{escrowMoney}, z)\,),$
$\longrightarrow_M$
true,
$\forall z :_{pre} \texttt{Object}.\,(\,\mathcal{MayAccess}(\texttt{escrowMoney}, z)\ \rightarrow$
$\qquad\mathcal{MayAccess}_{pre}(\texttt{sellerMoney}, z)\,).$

These two steps combined prove that we have now established mutual trust between these two purses. This is expressed in line 4 of figure 8:
$\texttt{res}\ \ \longrightarrow$
$\quad\texttt{sellerMoney}_{pre}\ \mathbf{obeys}\ \texttt{ValidPurse}$
$\qquad\quad\longleftrightarrow \texttt{escrowMoney}\ \mathbf{obeys}\ \texttt{ValidPurse}$
The bulk of the proof proceeds similarly, with lines 6-18 of figure 5 requiring the same reasoning to establish the remaining mutual trust relationships, first by including the remaining money purse, and then between all the goods purses.

Finally lines 20-30 complete the escrow exchange by exchanging money and goods. The core reasoning here is completely straightforward, as trust is already established between all concerned purses — although of course we also have to handle the cases where trust is not established, on paths where a `deposit` call fails. We have to continue to reason about the risk, but since only `deposit` and `sprout` calls are involved, this reasoning is no different to that of the first and second step.

References

[1] T. Kleymann. *Hoare Logic and VDM: Machine-checked soundness and completeness proofs.* PhD thesis, The University of Edinburgh, 1998.

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

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

```
 true
       {  res=escrowMoney.deposit(0, sellerMoney)  }
escrowMoney$_{pre}$ **obeys** ValidPurse $\longrightarrow$ ($\forall p \in_{pre}$ GoodPrs.$p$.balance$_{pre} = p$.balance)
escrowMoney$_{pre}$ **obeys** ValidPurse $\wedge$ res $\to$ (sellerMoney **obeys** ValidPurse $\wedge$
$\forall p :_{pre}$ GoodPrs.($p$.balance$_{pre} = p$.balance $\vee$ $\mathcal{M}ay\mathcal{A}ccess_{pre}$(sellerMoney, $p$))      $\wedge$
$\forall z :_{pre}$ Object. ($\mathcal{M}ay\mathcal{A}ccess$(escrowMoney, $z$) $\longrightarrow$ $\mathcal{M}ay\mathcal{A}ccess_{pre}$(escrowMoney, $z$))    $\wedge$
$\forall z, y :_{pre}$ Object. ($\mathcal{M}ay\mathcal{A}ccess$($z, y$) $\longrightarrow$
            ($\mathcal{M}ay\mathcal{A}ccess_{pre}$($z, y$) $\vee$ $\mathcal{M}ay\mathcal{A}ccess_{pre}$(sellerMoney, $y$) $\wedge$ $\mathcal{M}ay\mathcal{A}ccess_{pre}$(sellerMoney, $z$))
 $\bowtie$
 true
```

Figure 7. Hoare tuple for second step in `deal`

```
 true
     {  var escrowMoney := sellerMoney.sprout
        res :=  escrowMoney.deposit(0, sellerMoney)  }
 res $\longrightarrow$ sellerMoney$_{pre}$ **obeys** ValidPurse $\longleftrightarrow$ escrowMoney **obeys** ValidPurse   $\wedge$
 sellerMoney$_{pre}$ **obeys** ValidPurse $\longrightarrow$ ( $CanTrade$(escrowMoney, sellerMoney) $\wedge$
                                       escrowMoney.balance $= 0$ $\wedge$
                                       $\forall p \in_{pre}$ GoodPrs.$p$.balance$_{pre} = p$.balance $\wedge$
                                       sellerMoney **obeys** ValidPurse )   $\wedge$
 $\neg$res $\longrightarrow$ $\neg$(sellerMoney$_{pre}$ **obeys** ValidPurse)        $\wedge$
 $\forall p :_{pre}$ GoodPrs.($p$.balance$_{pre} = p$.balance $\vee$ $\mathcal{M}ay\mathcal{A}ccess_{pre}$(sellerMoney, $p$))      $\wedge$
 $\forall z :_{pre}$ Object. ($\mathcal{M}ay\mathcal{A}ccess$(escrowMoney, $z$) $\longrightarrow$ $\mathcal{M}ay\mathcal{A}ccess_{pre}$(sellerMoney, $z$))    $\wedge$
 $\forall z :_{pre}$ Object. ($\mathcal{M}ay\mathcal{A}ccess$(sellerMoney, $z$) $\longrightarrow$ $\mathcal{M}ay\mathcal{A}ccess_{pre}$(sellerMoney, $z$))    $\wedge$
 $\forall z, y :_{pre}$ Object. ($\mathcal{M}ay\mathcal{A}ccess$($z, y$) $\longrightarrow$
          ($\mathcal{M}ay\mathcal{A}ccess_{pre}$($z, y$) $\vee$ $\mathcal{M}ay\mathcal{A}ccess_{pre}$(sellerMoney, $y$) $\wedge$ $\mathcal{M}ay\mathcal{A}ccess_{pre}$(sellerMoney, $z$))
 $\bowtie$
 true
```

Figure 8. Hoare tuple for first and second step in `deal`