Necessary Specification[1]

AUTHOR1, ContinentOne
AUTHOR2, ContinentTwo
AUTHOR3, ContinentThree
AUTHOR4, IsItTheSameContinent?

Functional specifications of program components describe what components *can* do — the *sufficient* conditions to invoke the component's behaviour: a client who supplies arguments meeting an operation's preconditions can invoke that operation. While functional specifications are enough to reason about the behaviour of complete, correct programs, they cannot support reasoning about nonfunctional (systemic) behaviours of partial programs in an open world where security, privacy, robustness, and reliability are as important as functional behaviours. *Necessary specifications* — as their name implies — describe the *necessary* conditions under which a behaviour can take place: constraining components' behaviours and defining what they *cannot* do. By complementing functional specifications with necessary specifications, programmers can explicitly define what their programs should not do (as well as what they should do) making it easier to write components that support security and privacy, and so supporting the construction of robust and reliable programs.

ACM Reference Format:
author1, author2, author3, and author4. 2025. Necessary Specification. 1, 1 (June 2025), 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Traditional functional specifications describe what components are guaranteed to do. So long as a method is called in a state satisfying its preconditions, the method will complete its work and establish a state satisfying its postconditions. Method specifications are *sufficient* conditions under which methods can be called, that is. sufficient conditions under which their method's behaviour can be invoked.

Consider the specification of a trivial Bank component in fig. 1. The bank is essentially a wrapper for a map from account objects to account balances: given an instance of a Bank component, calling `newAccount` returns a new account object and then calling `initialise` sets up initial accounts and balances. (GRRR: not really happy about this. should have a boolean setup ghost field so initialise can onlyt be called once. which the functions have as precondtion :ARGH). Given an account, calling `balance` with an account returns the account balance, and calling `deposit` with two accounts deposits funds from the source to the destination account.

The specification in fig. 1 is enough to let us calculate the result of operations on the bank and the accounts — for example it is straightforward to determine that the code in fig. 2 satisfies its assertions: given that the `acm` object has a balance of 10,000 before an author is registered then

[1]Is the title too pretentious?

Authors' addresses: author1ContinentOne; author2 ContinentTwo; author3ContinentThree; author4IsItTheSameContinent?

XXXX-XXXX/2025/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
specification Bank {

  ghost field ledger : Map[Account, Number]

  function newAccount {
    a \in Account, b \in Bank
      { a := b.newAccount }
    FRESH(a)
  }

  function balance {
    a \in Account, b \in Bank, n \in Number
      { n := b.balance(a) }
    n == ledger.at( a )
  }

  function deposit {
    src, dst \in Account, b \in Bank, n \in Number
    sb == ledger.at(src), db == ledger.at(dst), n > 0
      { b.deposit(dst, src, n) }
    ledger.at(src) == sb - n
    ledger.at(dst) == db + n
  }

  function initialise {
    b \in Bank, m in Map[Account, Number]
      { b.initialise( m ) }
    ledger == m
  }

}
```

Fig. 1. Functional specification of a Bank

```
assume b.balance(acm) == 10000
assume b.balance(author) == 1500

b.deposit(acm, author, 1000)

assert b.balance(acm) == 11000
assert b.balance(author) == 500
```

Fig. 2. Registering at a Conference

afterwards it will have a balance of 11,000 while the `author` now has a balance of 500 from a starting balance of 1,500 (barely enough to buy a round of drinks at the conference hotel bar).

This reasoning is fine in a closed world, where we only have to consider complete programs, where all the code in our programs (or any other systems with which they interact) is under our control. In an open world, however, things are more complex: our systems will be made up of a range of components, many of which we do not control; and furthermore will have to interact with external

```
specification Theft {

  function steal {
    b \in Bank, thief in Account, m in Map[Account, Number]
    m == b.ledger
      { b.steal(thief) }
    forall a in dom(m) :
    ledger.at(a) =
      if (a == thief)
        then sum(codom(m))
        else 0
  }
}
```

Fig. 3. Functional specification of taxation

systems which we certainly do not control. Returning to our author, say some time after regisering by executing the code in fig. 2, they attempt to pay for a round at the bar. Under what circumstances can they be sure they have enough funds in their account?

To see the problem, consider the additional function specified in fig. 3. This method says the bank additional provides a `steal` method that empties out every account in the bank and puts all their funds into the thief's account. If this method exists, and if it is somehow called between registering at the conference and going to the bar, the author (actually everyone using the same bank) will find all their accounts empty (except the thief, of course).

The critical problem is that a bank implementation including a `steal` method would meet the functional specifications of the bank friom fig. 1, so long as its `newAccount`, `balance`, `deposit`, and `initialise` methods do meet that specification.

One obvious solution would be to return to a closed-world interpretation of specifications: we interpret specifications such as fig. 1 as *exact* in the sense that only implementations that meet the functional specification exactly, *with no extra methods or behaivour*, are considered as suitable implementations of the functional specification. The problem is that this solution is far too strong: it would for example rule out a bank that simply counted the number of deposits that had taken place, i.e. met fig. 4 as well as fig. 1.

What we need is some way to permit bank implementations that meet fig. 4 but to forbit implementatons that meet fig.3. The key here is to capture the (implicit) assumptionms underlying fig.1, and to provide additional specifications that capture those assumptions. There are at least two assumptions that can prevent methods like `steal`:

(1) after initialisation, the *only* way an account's balance can be changed is if a client calls the `deposit` method
(2) an account's balance can *only* be changed if a client has that particular account object.

Compared with the functional specification we have seen so far, these assunmptions capture *necessary* conditions rather than *sufficient* conditions. It is necessary that the `deposit` method is called to change an account's balance, and it is necessary that the particular account object can be passed as a parameter to that method. The fig. 3 specification is not consistent with these assumptions, while the. 4 specification is consistent with these assunmptions.

The contribution of this paper is a specification langauge and semantics that can be used to specify necessary specifications, and a semantics for those specifications that can determine whether

```
specification CountDeposits {

  ghost field count : Number = 0

  function deposit {
    c : Number = count
      { b.deposit(dst, src, n) }
    count == c + 1
  }

  function count {
    b : Bank
      { c = b.countDeposits }
    c == b.count
  }
}
```

Fig. 4. Functional specification counting the number of deposits

some functional (sufficient) specifications are consistent (or not) with the necessary specifications. Fig. 5 shows how we can express these two informal assumptions using our specification language Chainmail II. Rather than specifying `functions` to describe the behaviour of particular methods when they are called, we write `policies` that range across the whole behvaiour of the component.

An overall "holistic" specification for the bank account, then, would be our original sufficient functional specification from fig. 1 plus the necessary security policy specification in fig. 5. This holistic specification permits an implememntation of the bank that also meets the `count` specification from fig. 4, but does not permit an implementation that also meets the `steal` specification from fig. 3.

$$(1) \triangleq \forall \mathtt{a}.\forall \mathtt{S}.\, [\ \mathtt{a} : \mathtt{Account} \ \land\ \mathtt{this} \neq \mathtt{a} \ \land\ (\Diamond \mathit{Changes}(\mathtt{a.balance}))\ \mathsf{in}\ \mathtt{S} \quad \longrightarrow \quad \exists \mathtt{o}.\, [\ \mathtt{o} \in \mathtt{S} \ \land\ \mathit{Calls}(\mathtt{deposit}) \ \land\ \mathtt{o} \notin \mathtt{Internal}(\mathtt{a})\ ] \quad]$$

$$(2) \triangleq \forall \mathtt{a}.\forall \mathtt{S}.\, [\ \mathtt{a} : \mathtt{Account} \ \land\ \mathtt{this} \neq \mathtt{a} \ \land\ (\Diamond \mathit{Changes}(\mathtt{a.balance}))\ \mathsf{in}\ \mathtt{S} \quad \longrightarrow \quad \exists \mathtt{o}.\, [\ \mathtt{o} \in \mathtt{S} \ \land\ \mathcal{A}\mathit{ccess}(\mathtt{o}, \mathtt{a}) \ \land\ \mathtt{o} \notin \mathtt{Internal}(\mathtt{a})\ ] \quad]$$

Fig. 5. Necessary specifications for `deposit`

Policy (1) in fig. 5 says that if — in any future state ($\Diamond \ldots$) an account's balance is changed ($\mathit{Changes}(\mathtt{a.balance})$) then there must be some client object o which is outside the bank and its associated accounts ($\mathtt{o} \notin \mathtt{Internal}(\mathtt{a})$)t that calls the `deposit` method: ($\mathit{Calls}(\mathtt{deposit})$). Policy (2) similarly constraints any possible code that may change an account's balance, but requires that the client object making the call has direct access to the account object ($\mathcal{A}\mathit{ccess}(\mathtt{o}, \mathtt{a})$).

We can then prove that e.g. the `steal` method from fig. 3 is inconsistent with both of these policies. First, the `steal` method clearly changes the balance of every account in the bank, but policy (1) requires that any method that changes the balance of any account must be called `deposit`. Second, the `steal` method changes the balance of every account in the system, and will do so without the called having a reference to most of those accounts, which breaches policy (2). Note that `steal` putting all the funds into the thief's account does not breach policy (2) with respect to the thief's own account, because that account is passed in as a parameter to the `steal` method, and so the called of the `steal` must have access to that account.

random minor point. These necessary specification policies can be defined and interpreted independently of any particular implementation of a specification — rather our policies constrain implementations, in just the same way as traditional functional specifications. This is in contrast to e.g. class invariants, which establish invariants across the implementation of an abstract, or abstraction functions, which link an abstract model to a concrete implementation of that model.

```
class Bank {
  field ledger;         // a Node
  Bank( ){ ledger = null; }
  fun makeAccount(amt){ ...  }
  fun deposit(source, destination, amnt){ ... }
}

class Account {
   fun myBank;          // a Bank
   Account(aBank){ myBank = aBank;  }
   fun deposit(source, destination, amnt){ ... }
}

 class Node{
   field balance;      // the  money held in theAccount a number
   field next;         // the next node
   field theAccount;   // the account
    ...
}
```

Fig. 6. Sketch of the code of the Bank example – not robust

2 INTRODUCTION

Write the intro – use following structure:

(1) Show the bank/account without protection from Figure 6, and discuss why it is not robust
(2) Say that the code is not robust
(3) Show robust version of same code; this appears here in Figure 7. Argue that the two versions have same HL spec.
(4) How to specify robustness?
(5) outline our proposal
(6) short comparison with other works
(7) Contributions ...
(8) Structure of this paper ...

3 OUR RESEARCH QUESTION: HOW TO SPECIFY ROBUSTNESS

Here we explain what makes our code robust.

In the code from Figure 7 .... explain what it does .... explain what currency is, and why it is important to protect it.

Now consider the objects from the diagram in Figure 8. The boxes represent objects (at a certain address and of a certain class), and the arrows represent fields pointing to other objects. For example, at address `1` we have an object of class `Bank`, at `4` and `5` we have objects of class `Node`, ... The `myBank` field from object `2` points to `1`, etc ... The grey objects, at `10`, `11`, `20` and `21` are objects of unknown provenance, but we know that `10` has a field pointing to the `Bank` at `1`, and a field pointing to `11`, and similarly for objects `20` and `21`.

What might happen? Since object `10` has access to the `Bank` object at `1`, and given the code from Figure 7, and given that we know nothing about `10`'s provenance, it is conceivable that `10` would call method `makeAccount` on `1` with any amount, and thus increase the currency. Similarly, `21`

```
 class Bank {
   private field ledger;    // a Node

   Bank( )
     { ledger = null; }
   fun makeAccount(amt)
     { account = new Account(this);
       ledger = new Node(account, amt, ledger);
       return account; }
   fun deposit(source, destination, amnt)
     {  ... }
 }

 class Account {
    private field myBank;  // a Bank

   Account(aBank)
     { myBank = aBank;  }
   fun sprout( )
     // create Account in same Bank with 0 balance
     { return this.myBank.create(0)   }
   fun deposit(source, amnt)
     // if destination is an Account in myBank,  and  source holds enough money,
     // then transfer amnt from source into receiver
     { ... }
}

  class Node{
   field balance;     // the  money held in theAccount a number
   field next;        // the next node
   field theAccount;  // the account
     ...
 }
```

Fig. 7. Sketch of the Bank example – robust version

has direct access to `3`, and indirect access (through `20`) to `4`, it can call the method `deposit`, and thus affect the `balance` of both `3` and `4`.

What is guaranteed not to happen? Even though `20` has indirect access to `1` (through `4`), it cannot navigate the path to `1`, and cannot call a method on `1` that affects the currency, nor ask `4` to give access to `1`. Therefore, it is guaranteed that `20` and `21` cannot affect the currency. Similarly, it is guaranteed that `20`, `21`, `10` and `11` cannot affect the `balance` of `2`.

Are we interested in what is guaranteed not to happen? We argue that knowing what is guaranteed not to happen is crucial in understanding whether a program is robust and what way it is robust. For example, the fact that an execution which has access to objects `20` and `21` only, cannot affect the currency of `1`, means that we can pass `Account` objects as arguments in calls to unverified, or even to objects of unknown provenance, safe in the knowledge that the only risk we are running is to lose the money stored in the specific `Accounts`, but without worrying about the money in other `Accounts`, and without worry about the integrity of the currency in the `Bank`. Such considerations are expressed in two of the five policies proposed in [1].

`Pol_2` Only someone with the bank of a given currency can violate conservation of that currency.
`Pol_4` No one can affect the balance of a account they don't have.

Robustness has inspired the introduction of several language features *e.g.* annotations such as `const`, `private`, `unique` *etc.*, type disciplines *e.g.* linear types, ownership types *etc.*, and also programming patterns such as proxies, membranes *etc.*. These are *mechanisms* that support the development of robust code, and they each come with their own hyper-properties (*e.g.* `const` fields do not change, or `own`ed objects are dominated by their owner), proven [**?**] for each case. Code become robust through an interplay of the use of such features with programming pattern[1]. We argue, with some few exceptions, [**?**] there is little work in the *specification* of robust code.

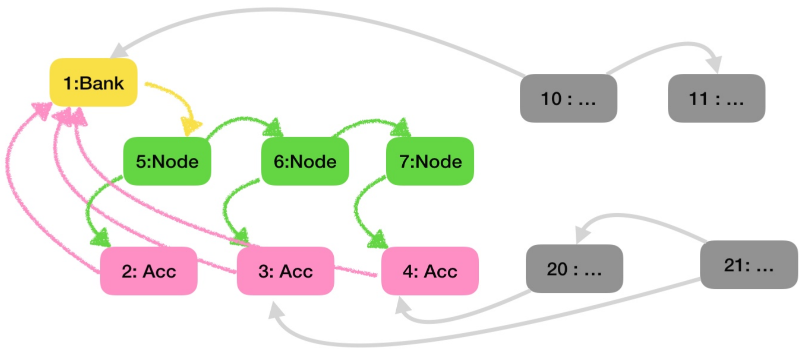

Fig. 8. Diagrammatic representation of some objects from `Bank`, `Account` *etc.*

Our Approach. In order to specify robustness, we need to express

(1) access (permission) and change (authority)
(2) *potential* effects (the future) and *causes* of effects (the past)
(3) footprints[2] (space) of executions as well as of assertions. For example, an execution which starts (has as receiver) an object different than `1`, and which involves objects `1`, `2`, `3`, `4`, `5`, `6`, `7`, `20` and `21`, might end up calling methods on `1`, but is guaranteed not to affect the `currency` on `1`.
 Interestingly, the temporal aspects of our assertions unify the footprint of an execution and an assertion[3]: The foorprint of an assertion which talks about time is essentially the footprint of the execution involved in satisfying that assertion.

4 OUR PROPOSAL

4.1 Preliminaries

Here we define some preliminaries ... they are more or less standard.[4]. We will define the following:

(1) Runtime configuration σ; it consists of a heap, and a *rich* expression[5]. A rich expression consists of a stack of frames end expression contexts as well as the expression currently being executed. Explain that this is slightly unusual, but why it is useful

[1]I think I am not clear here
[2]use instead the tern footprint or witness?
[3]Better word?
[4]Most of the below can be taken from Swapsies work – it is slightly simpler as we do not need **obeys**
[5]Better term than "rich expression"?

(2) A small OO PL with its syntax.
(3) Also, syntax definitions for expressions e
(4) Also, syntax definitions for assertions A.
(5) Modules $\mathtt{M}$: a module maps a class identifier to a class definition; a function identifier to a function and a predicate identifier to an assertion. A module concatenation operator, $\mathtt{M} * \mathtt{M}'$ – the operation is defined only when the two modules do not have overlapping definitions.
(6) A small step operational semantics with the format $\mathtt{M}, \sigma \rightsquigarrow \sigma'$
(7) an interpretation function for expressions $\lfloor e \rfloor_{\mathtt{M},\sigma}$ which maps expressions and configurations to values. [6]
(8) Define validity for assertions $\mathtt{M}, \sigma \models A$. Eg the term $\mathtt{x} : \mathtt{C}$ expresses that $\lfloor \mathtt{x} \rfloor_{\mathtt{M},\sigma}$ is the address of an object of class $\mathtt{C}$; [7]
(9) Definition of the notion of initial configuration $\mathcal{I}nitial(\_)$ and arising configuration $\mathcal{A}rising(\mathtt{M})$. The set $\mathcal{A}rising(\mathtt{M})$ contains all runtime configurations which can be reached when starting with an empty heap, and executing any expression consisting of constructor and method calls as defined in $\mathtt{M}$.[8]

4.2 Extensions

We will define the OCAP assertions $\mathcal{A}ccess(\_,\_)$ (permission) and $Changes(\_)$ (authority). [9] We also add temporal modifiers, where $\Diamond A$ expresses that A will hold at some future point, ∇A expresses that A held at some point in the past. We also add a *spatial modifier*, $A \,\text{in}\, S$, which expresses that assertion A holds in the sub-configurations determined by the witness $\mathtt{S}$.

We extend the syntax for assertions as follows:[10]

Definition 1 (Assertions).

$$A ::= \ ...as\ before... \quad | \ \mathcal{A}ccess(x,y) \ | \ Changes(e) \ | \ Calls(\mathtt{m}) \ | \ \Diamond A \ | \ A \,\text{in}\, S \ | \ \nabla A$$

Definition 2 (Time, Permission, Authority, and Space). *Given a module $\mathtt{M}$, identifiers $\mathtt{x}$ and $\mathtt{y}$, expression $\mathtt{e}$, and runtime configuration σ, and a set of addresses S, we define validity of the assertions .... as follows:*

- $\mathtt{M}, \sigma \models \Diamond A$ *iff* $\exists \sigma'. [\ \mathtt{M}, \sigma \rightsquigarrow^* \sigma' \ \wedge \ \mathtt{M}, \sigma'[\overline{x \mapsto \sigma(x)}] \models A$, *where* $\overline{x} = Free(A)$].
- $\mathtt{M}, \sigma \models \nabla A$ *iff* $\exists \sigma_1, ...\sigma_n, k \in [1..n-1).\ [\mathcal{I}nitial(\sigma_1) \ \wedge \ \sigma_n = \sigma \ \wedge \ \forall i \in [1..n).\ \mathtt{M}, \sigma_i \rightsquigarrow \sigma_{i+1}$ $\wedge \ \mathtt{M}, \sigma_k[\overline{x \mapsto \sigma(x)}] \models A$, *where* $\overline{x} = Free(A)$].
- $\mathtt{M}, \sigma \models \mathcal{A}ccess(\mathtt{x}, \mathtt{y})$ *iff*
  - $\sigma(x) = \sigma(y)$, *or*
  - $\sigma(\mathtt{x}, \mathtt{f}) = \sigma(\mathtt{y})$ *for some field* $\mathtt{f}$, *or*
  - $\sigma(\mathtt{this}) = \sigma(\mathtt{x})$ *and* $\sigma(\mathtt{z}) = \sigma(\mathtt{y})$, *for some some parameter of local variable* $\mathtt{z}$.
- $\mathtt{M}, \sigma \models Changes(\mathtt{e})$ *iff* $\exists \sigma'. [\ \mathtt{M}, \sigma \rightsquigarrow \sigma' \ \wedge \lfloor e \rfloor_{\mathtt{M},\sigma} \neq \lfloor e \rfloor_{\mathtt{M},\sigma'}$]
- $\sigma|_S$ *denotes a* restriction *of σ to the objects from the set S. That is, the domain of the heap in $\sigma|_S$ is S, and otherwise, $\sigma|_S$ is identical to σ. An example appears in figure 9.*
- $\mathtt{M}, \sigma \models A \,\text{in}\, \mathtt{S}$ *iff* $\mathtt{M}, \sigma|_S \models A$, *where* $S = \lfloor \mathtt{S} \rfloor_{\mathtt{M},\sigma}$.

[6]NOTE_TO_SELF: careful with undefinedness – as per below for A's. Also, we need to distinguish bad formedness because of infinite recursion from undefinedness because of missing definitions. We have all that in the Swapsies paper.
[7]NOTE_TO_SELF: This is standard, but as we allow for recursive efintions in $\mathtt{M}$ we also need to cater about infinite recursion and thus introduce undefinedness. In particular, if $A \longrightarrow A'$, and if A undefined or A is false but A' undefined? Ideally, these things have to be defined in such a way that usual manipulations hold, eg $A \longrightarrow A' \ \equiv \neg A \vee A'$.
[8]That is, $\mathcal{A}rising(\mathtt{M}) = \{\ \sigma \ | \ \exists \mathtt{e}.\ \mathtt{M} \vdash (\mathtt{e}, \emptyset) \rightsquigarrow * \sigma\ \}$
[9]Note that they are slightly different assertions to those we had in the past.
[10]The symbols are not that good – esp the symbols for future and past.

- $\texttt{M}, \sigma \models \mathit{Calls}(\texttt{m})$ *iff* *the method call in the current frame in σ is* $\texttt{m}$[11]

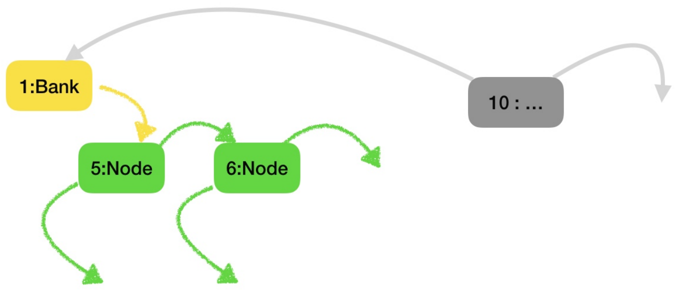

Fig. 9. Configuration from Fig. 8 restricted to witness { 1, 5, 6, 10 }

Note that $\mathcal{A}ccess(\texttt{x}, \texttt{y})$ is reflexive but not transitive and not symmetric.

Also note the difference between $(\Diamond A)$ in $\texttt{S}$ and $\Diamond (A$ in $\texttt{S})$. For example, the assertion $(\Diamond\, \texttt{x.f} = \texttt{z})$ in $\texttt{S}$ expresses that the current execution leads to a future configuration where $\texttt{x.f} = \texttt{z}$ will hold, and that the set $\texttt{S}$ suffices to witness this execution, while the assertion $\Diamond\, (\texttt{x.f} = \texttt{z}$ in $\texttt{S})$ expresses that the current execution leads to a future configuration where $\texttt{x.f} = \texttt{z}$ will hold and where the set $\texttt{S}$ suffices to witness that fact. For example, take a configuration σ where variable $\texttt{x}$ maps to some object $\texttt{31}$, and where $\texttt{31}$ has a field $\texttt{f}$ pointing to $\texttt{32}$, and $\texttt{32}$ has a field $\texttt{f}$ pointing to $\texttt{33}$, and variable $\texttt{z}$ maps to $\texttt{33}$. Assume also that the expression to be executed in σ starts with $\texttt{x.f=x.f.f}$. Then we have that $\texttt{M}, \sigma \models \Diamond\, (\texttt{x.f} = \texttt{z}$ in $\{\texttt{x}, \texttt{z}\})$, but $\texttt{M}, \sigma \not\models (\Diamond\, \texttt{x.f} = \texttt{z})$ in $\{\texttt{x}, \texttt{z}\}$. On the other hand $\texttt{M}, \sigma \not\models (\Diamond\, \texttt{x.f} = \texttt{z})$ in $\{\texttt{x}, \texttt{x.f}, \texttt{z}\}$

In general, for all $\texttt{M}$ and σ, we have that $\texttt{M}, \sigma \models (\Diamond A)$ in $\texttt{S}$ implies $\texttt{M}, \sigma \models \Diamond\, (A$ in $\texttt{S})$, and that $\texttt{M}, \sigma \models \Diamond\, (A$ in $\texttt{S})$ implies that there exists a set $\texttt{S}'$ such that $\texttt{M}, \sigma \models (\Diamond A)$ in $(\texttt{S} \cup \texttt{S}')$.

We will prove that

Lemma 4.1 (Preservation of validity and module linking). $\texttt{M}, \sigma \models A$ *then for all* $\texttt{M}'$ *with* $\texttt{M} * \texttt{M}'$ *is defined:* $\texttt{M} * \texttt{M}', \sigma \models A$.

4.3 Invariants

We define below the meaning of invariants.[12] The assertion $\texttt{M} \models A$ requires that the assertion A is satisfied in all reachable states.

Definition 3 (Invariants). *For a module* $\texttt{M}$ *and assertion* A *we define:*

- $\texttt{M} \models A$ *iff* $\forall \texttt{M}'.\forall \sigma \in \mathcal{A}rising(\texttt{M}' * \texttt{M}).\ \texttt{M}' * \texttt{M}, \sigma \models A$

The use of the set of configurations from $\mathcal{A}rising(\texttt{M}' * \texttt{M})$ reflects that policies need to hold in an *open* world, where we link against *any* module $\texttt{M}'$, about which we know nothing.

[11] We will express this precisely when we have the full definition of σ

[12] This part is as we had defined previously, with two simplifications: a) we do not need to worry about the **obeys**-predicate here, and b) we do not distinguish the names of the classes and the names of participants in interfaces.

4.4 Implication and equivalence

DEFINITION 4. *For a module* $\mathtt{M}$ *and assertions* A *and* A'*, we define strong equivalence and implication (*$\equiv$ *and* $\sqsubseteq$*), as well as weak equivalence and implication (*$\approx$ *and* $\lessapprox$*) as follows:*

- $\mathtt{M} \models A \leqq A'$ *iff* $\forall \mathtt{M}'.\forall \sigma \in \mathcal{A}rising(\mathtt{M}' * \mathtt{M}).[\ \mathtt{M}' * \mathtt{M}, \sigma \models A \longrightarrow \mathtt{M}' * \mathtt{M}, \sigma \models A'\ ]$
- $\mathtt{M} \models A \equiv A'$ *iff* $\mathtt{M} \models A \leqq A' \ \wedge\ \mathtt{M} \models A' \leqq A$
- $\mathtt{M} \models A \lessapprox A'$ *iff*
  $\forall \mathtt{M}'.\forall \sigma \in \mathcal{A}rising(\mathtt{M}' * \mathtt{M}).\, \mathtt{M}' * \mathtt{M}, \sigma \models A \longrightarrow \forall \mathtt{M}'.\forall \sigma \in \mathcal{A}rising(\mathtt{M}' * \mathtt{M}).\, \mathtt{M}' * \mathtt{M}, \sigma \models A'$
- $\mathtt{M} \models A \approx A'$ *iff* $\mathtt{M} \models A \lessapprox A' \ \wedge\ \mathtt{M} \models A' \lessapprox A$

The definitions from above are applicable to the empty module, eg $\models A \equiv A'$ iff all modules $\mathtt{M}$ satisfy $\mathtt{M} \models A \equiv A'$. The following properties hold:

LEMMA 4.2. *For all modules* $\mathtt{M}$*, assertions Aand A':*

- $\mathtt{M} \models A \equiv A'$ *implies* $\mathtt{M} \models A \simeq A'$
- $\mathtt{M} \models A \leqq A'$ *implies* $\mathtt{M} \models A \lessapprox A$
- $\mathtt{M} \models (\Diamond A)\ \text{in}\ \mathtt{S} \ \leqq\ \Diamond (A\ \text{in}\ \mathtt{S})$
- $\mathtt{M} \models \Diamond A \to A' \ \lessapprox\ A \to \nabla A'$

Space-Monotonicity. [13]

DEFINITION 5 (SPACE-MONOTONICITY). *We call an assertion* A space-monotonic *in* $\mathtt{M}$*, iff for all set expressions* $\mathtt{S}$ *and* $\mathtt{S}'$*, and all* $\sigma \in \mathcal{A}rising(\mathtt{M})$*:*
If $\mathtt{M}, \sigma \models \mathtt{S} \subseteq \mathtt{S}'$ *and* $\mathtt{M}, \sigma \models A\ \text{in}\ \mathtt{S}$*, then* $\mathtt{M}, \sigma \models A\ \text{in}\ \mathtt{S}'$

We prove space monotonicity for some assertions

LEMMA 4.3 (SPACE-MONOTONICITY FOR CHANGE AND ACCESS).

- $Changes(\mathtt{e})$ *is space-monotonic.*
- $\mathcal{A}ccess(\mathtt{x}, \mathtt{y})$ *is space-monotonic.*

Not all assertions are not space-monotonic. E.g. $\forall a : \mathtt{Account.a.balance} \geq 3$ is not space-monotonic.
The following lemma would be nice to have – otherwise we will need to change the definition of monotonicity.

LEMMA 4.4 (SPACE-MONOTONICITY AND MODULE LINKING). *If A is space-monotonic with* $\mathtt{M}$*, then it is also space-monotonic with* $\mathtt{M} * \mathtt{M}'$*.*

5 SPECIFICATIONS FOR ROBUSTNESS POLICIES

We now use the concepts introduced in the earlier sections to specify various robustness policies

5.1 Specification of `Pol_2` and `Pol_4`

We give a formal definition of `Pol_2` and `Pol_4`, using the concepts defined earlier in Definition 2:

DEFINITION 6. *We define what it means for an object* $\mathtt{o}$ *to be internal to a bank's data structure, an then define* `Pol_2` *and* `Pol_4` *as follows:*

$$\mathtt{Internal(b)} \triangleq \{\mathtt{o} \mid \mathtt{b : Bank} \ \wedge\ (\mathtt{o} = \mathtt{b} \ \vee\ \mathtt{o : Account} \wedge \mathtt{o.myBank} = \mathtt{b} \ \vee\ \exists k.\, \mathtt{b.ledger.next}^{k} = \mathtt{b})\ \}$$

$$\mathtt{Internal'(a)} \triangleq \{\mathtt{o} \mid \mathtt{a : Account} \ \wedge\ \mathtt{a.myBank : Bank} \ \wedge\ \mathtt{o} \in \mathtt{Internal(b)}\}$$

[13] Not sure how useful this concept is

$$\texttt{Pol\_2} \triangleq \forall \texttt{b}.\forall \texttt{S}.[\ \texttt{b}:\texttt{Bank}\ \wedge\ \texttt{this} \neq \texttt{b}\ \wedge\ (\Diamond\mathit{Changes}(\texttt{b.currency}))\ \text{in}\ \texttt{S} \quad \longrightarrow \\ \exists \texttt{o}.[\ \texttt{o} \in \texttt{S}\ \wedge\ \mathcal{A}\mathit{ccess}(\texttt{o},\texttt{b})\ \wedge\ \texttt{o} \notin \texttt{Internal(b)}\ ]\ ]$$

$$\texttt{Pol\_4} \triangleq \forall \texttt{a}.\forall \texttt{S}.[\ \texttt{a}:\texttt{Account}\ \wedge\ \texttt{this} \neq \texttt{a}\ \wedge\ (\Diamond\mathit{Changes}(\texttt{a.balance}))\ \text{in}\ \texttt{S} \quad \longrightarrow \\ \exists \texttt{o}.[\ \texttt{o} \in \texttt{S}\ \wedge\ \mathcal{A}\mathit{ccess}(\texttt{o},\texttt{a})\ \wedge\ \texttt{o} \notin \texttt{Internal}'\texttt{(a)}\ ]\ ]$$

Discussion. In other words, Pol_2 mandates that the elements of the data structure (ie the elements from Internal(b)) cannot be used (are not sufficient) to change the currency of the bank. If a computation takes place inside the set S, and *in the current state* in S all accesses to the bank go through elements of the data structure (ie the Account objects),[14] then we have a guarantee that the computation will not affect the currency. For example, if a computation takes place in the context of objects 1, 2, 3, 4, 5, 7, 20 and 21, and the current receiver is no 1, then we have a guarantee that the currency of 1 will not be affected. So, even through 1 is involved in the computation, because there is no *external* access to it, we have a guaratee that the method makeAccount will not be called on it.

An alternative way of expressing Pol_2 is as follows:

$$\texttt{Pol\_2} \equiv \forall \texttt{b}.\forall \texttt{S}.[\ \texttt{b}:\texttt{Bank} \wedge \texttt{b} \neq \texttt{this} \wedge \forall \texttt{o} \in \texttt{S}.[\ \texttt{o} \in \texttt{Internal(b)} \vee \neg\mathcal{A}\mathit{ccess}(\texttt{o},\texttt{a})\ ] \\ \longrightarrow\ \neg((\Diamond\mathit{Changes}(\texttt{b.currency}))\ \text{in}\ \texttt{S})\ \ ]$$

Pol_2 guarantees that if an object o≠b may affect the value of b.Currency only if the objects involved in the process of affecting the value of b.Currency include at least an object o′ which had direct access to b, and whose class is not Account. Stated positively, this policy mandates that exporting an Account to an environment will not affect the Currency of b. In other words, Accounts protect the integrity of the Bank's currency.

In more detail, by applying Definition 3 on Definition 6, the meaning of policy Pol_2 is, that a runtime configuration σ satisfies Pol_2 if whenever the current receiver in σ is not a Bank object, and the execution of σ leads to another runtime configuration σ' with a different value for b.Currency, then the objects involved in the execution from σ to σ' include at least one object which had direct access to b. Note that this direct access needs to exist at the beginning of the execution, *i.e.* at σ. Formally:

$$\texttt{M},\sigma \models \texttt{Pol\_2} \\ \longleftrightarrow \\ \forall \texttt{b}.\forall S.[\quad \texttt{M},\sigma \models \texttt{b}:\texttt{Bank}\ \wedge\ \sigma(\texttt{b}) \neq\ \sigma(\texttt{this}) \\ \wedge\ \exists\sigma'.(\quad \texttt{M},\sigma\ |_{S}\!\rightsquigarrow^{*} \sigma'\ \ \wedge\ \lfloor\texttt{b.Currency}\rfloor_{\texttt{M},\sigma} \neq \lfloor\texttt{b.Currency}\rfloor_{\texttt{M},\sigma'[\texttt{b}\mapsto\sigma(\texttt{b})]}\ ) \\ \longrightarrow \\ \exists \texttt{o}.(\ \texttt{o}\in S\ \ \wedge\ \ \texttt{M},\sigma \models \mathcal{A}\mathit{ccess}(\texttt{o},\texttt{b})\ \wedge\ \texttt{o} \notin \texttt{Internal(b)}\ )\qquad]$$

5.2 Specifying "no leaks"

This is a family of guarantees that Dean seemed especially interested in, when we discussed in March in London. For the particular example, we want to express that

Pol_7 The Bank does not leak out of the Bank/Account system

And we give a formal specification

Definition 7 (Banks do not leak). *We define* Pol_7 *as follows:*

$$\texttt{Pol\_7} \triangleq \forall \texttt{b}.\forall \texttt{S}.[\ \texttt{b}:\texttt{Bank}\ \wedge\ \texttt{o}:\texttt{Object}\ \wedge\ \neg(\mathcal{A}\mathit{ccess}(\texttt{o},\texttt{b}))\ \wedge\ (\Diamond\mathcal{A}\mathit{ccess}(\texttt{o},\texttt{b}))\ \text{in}\ \texttt{S} \\ \longrightarrow\quad \exists \texttt{o}'.[\ \texttt{o}' \in \texttt{S}\ \wedge\ \mathcal{A}\mathit{ccess}(\texttt{o}',\texttt{b})\ \wedge\ \texttt{o}' \notin \texttt{Internal(b)}\,)\ ]\ \ ]$$

In other words, Pol_7 guarantees that objects that are internal to the bank b do not leak access to it. In more detail: if objects o and b exist now, and o does not have direct access to b now, but

[14] Say why we can ignore Node objects

obtains access to `b` through some computation which involves objects from the set `S`, then at least one object from `S` has now direct access to `b` and this object is not internal to `b`.

6 ADHERENCE TO POLICIES

In this section we will outline the proofs that particular modules adhere to their specifications. This serves to demonstrate the practicality of our approach. In particular we will show two different versions fo the `Bank`/`Account` example (sections 6.2 and 6.3, and we will prove that both satisfy the policies `Pol_2`, `Pol_4`, and `Pol_7`, while they differ in the definition of `Internal`. But before doing that, in section 6.1, we will study some further properties of execution.

6.1 General properties of execution

[15]

We will first define some further predicates which reflect over the program execution and prove some general properties of program execution.

We call a locations set, `L`, an expression which denotes a set of addresses and field identifiers, *e.g.* , $\{ (\mathtt{b}, \mathtt{ledger}), (\mathtt{b.ledger}, \mathtt{balance}) \}$ is such a locations set.

DEFINITION 8 (FRAMING). *Take arbitrary module* `M`, *assertion* A, , ...

- $\lfloor \mathtt{L} \rfloor_{\sigma, \mathtt{M}*\mathtt{M}'} = ...$
- $\sigma\, |_{L}$ ....
- $\mathtt{M}, \sigma \models \mathtt{L}$ frames $\mathtt{e}$ *iff* $\forall \mathtt{M}'.\forall \sigma' \in \mathcal{A}rising(\mathtt{M} * \mathtt{M}').\forall L.$
  $[\ L = \lfloor \mathtt{L} \rfloor_{\mathtt{M},\sigma}\ \wedge\ \sigma\, |_{L} = \sigma'\, |_{L}\ \longrightarrow\ \lfloor \mathtt{e} \rfloor_{\sigma,\mathtt{M}*\mathtt{M}'} = \lfloor \mathtt{e} \rfloor_{\sigma',\mathtt{M}*\mathtt{M}'}\ ]$
- $\mathtt{M} \models \mathtt{L}$ frames $\mathtt{e}$ *iff* $\forall \mathtt{M}'.\forall \sigma \in \mathcal{A}rising(\mathtt{M} * \mathtt{M}').\ \mathtt{M}' * \mathtt{M}, \sigma \models \mathtt{L}$ frames $\mathtt{e}$
- $\mathtt{M}, \sigma \models \mathtt{L}$ frames A *iff* $\forall \mathtt{M}'.\forall \sigma' \in \mathcal{A}rising(\mathtt{M} * \mathtt{M}').\forall L.$
  $[\ L = \lfloor \mathtt{L} \rfloor_{\mathtt{M},\sigma}\ \wedge\ \sigma\, |_{L} = \sigma'\, |_{L}\ \longrightarrow\ [\ \mathtt{M} * \mathtt{M}', \sigma \models A\ \longleftrightarrow\ \mathtt{M} * \mathtt{M}', \sigma' \models A\ ]$
- $\mathtt{M} \models \mathtt{L}$ frames A *iff* $\forall \mathtt{M}'.\forall \sigma \in \mathcal{A}rising(\mathtt{M} * \mathtt{M}').\ \mathtt{M}' * \mathtt{M}, \sigma \models \mathtt{L}$ frames A

NOTE_TO_SELF: we need to think about whether we also need to make `L` self-framing. Also, rethink whether we need to stick new modules $\mathtt{M}'$ to the whole thing. – Also, the sets are not equal – they are isomorphic. We can deal with isomorphisms, but it has a high notation penalty. Can we pretend that they are equal? hmhhhhh

And then we can prove that changes in the interpretation or the validity require a change in the frame:

LEMMA 6.1 (CHANGE IN THE CONTEXT OF FRAMING). *Take arbitrary module* `M`, *assertion* A, *such that* $\sigma \in \mathcal{A}rising(\mathtt{M} * \mathtt{M}')$

- *If* $\mathtt{M} \models \mathtt{L}$ frames $\mathtt{e}$, *and* $\mathtt{M}' * \mathtt{M}, \sigma \models (\Diamond\, Changes(\mathtt{e}))$ in `S`,
  then there exists a pair $(\mathtt{e}', \mathtt{f})$, *with* $\mathtt{M}, \sigma \models (\mathtt{e}', \mathtt{f}) \in \mathtt{L}$[16] *and* $\mathtt{M}' * \mathtt{M}, \sigma \models \Diamond\, Changes(\mathtt{e}'.\mathtt{f})$ in `S`
- *similar for* $\mathtt{M}' * \mathtt{M}, \sigma \models \Diamond\, Changes(A)$ in `S`

We now think a bit more about changes in accessibility. The predicate $\mathcal{G}ives(\mathtt{x}, \mathtt{y}, \mathtt{z})$ expresses that `x` passed to `y` access to `z`.

DEFINITION 9 (GIVING). *For arbitrary module* `M`*and* σ, *we define:*

- $\mathtt{M}, \sigma \models \mathcal{G}ives(\mathtt{x}, \mathtt{y}, \mathtt{z})$ *iff* $\sigma(\mathtt{this}) = \sigma(\mathtt{x})\ \wedge\ \mathtt{M}, \sigma \models \neg(\mathcal{M}ay\mathcal{A}ccess(\mathtt{y}, \mathtt{z}))\ \wedge$
  $\exists \sigma'.\ [\ \mathtt{M}, \sigma \rightsquigarrow \sigma'\ \wedge\ \mathtt{M}, \sigma' \models \mathcal{M}ay\mathcal{A}ccess(\mathtt{y}, \mathtt{z})\ ]$

[15] Find better title?

[16] Sophia, you need to check this bit what if `z` there is no handle in `L`, eg what if `L` talks about anonymous objects *e.g.* $\mathtt{L} = \{o \mid o.\mathtt{myBank} = \mathtt{b}\}$?Here o is anonymous. Also, do we need `M` or $\mathtt{M}' * \mathtt{M}$?

```
class Bank {
  private field ledger;   // a Node

  Bank( )
    { ledger = null; }
  fun makeAccount(amt)
    { account = new Account(this);
      ledger = new Node(account, amt, ledger);
      return account; }
  fun deposit(source, destination, amnt)
    { sourceNd = ledger.getNode(source)
      destinationNd = ledger.getNode(destination)
      if (sourceNd!=null && destinationNd!=null && sourceNd.balance>amt) then
         { < sourceNd.balance = sourceNd.balance-amt
             destinationNd.balance = destinationNd.balance+amt > }
       else
         { return }               }
}

class Account {
  private field myBank;  // a Bank

  Account(aBank)
    { myBank = aBank;  }
  fun sprout( )
    // create Account in same Bank with 0 balance
    { return this.myBank.makeAccount(0)  }
  fun deposit(source, amnt)
    // if destination is an Account in myBank,  and  source holds enough money,
    // then transfer amnt from source into receiver
    { myBank.deposit(source,this,amnt) }
}

 class Node{
  field balance;     // the  money held in theAccount a number
  field next;        // the next node
  field theAccount;  // the account

  fun getNode(account)
    { if (theAccount==account) then
          { return this }
      elseif (next!=null)
          { next.getNode(account) }
      else
          {  return null }            }
}
```

Fig. 10. $\mathtt{M_{BA}}$: First version of the Bank example, in detail

The following lemma says that any changes in accessibility witnessed[17] by set `S` is due to an element of `S` giving the object.

[17] is that the right term? or frames?

LEMMA 6.2 (CHANGE IN ACCESSIBILITY IS CAUSED BY GIVING). *For any module* M, *and* σ

- *If* $M, \sigma \models \neg(\mathcal{MayAccess}(y, z))$, *and* $M, \sigma \models (\Diamond\, \mathcal{MayAccess}(y, z))$ in S, *then there exists a* x *and* y′, *with* $M, \sigma \models (\Diamond\, \mathcal{Gives}(x, y', z))$ in S.

6.2 Adherence to Policies for module M_{BA}

In figure 10 we show the code for M_{BA}in detail.

6.2.1 M_{BA} *preliminaries.*

We define the footprint of `b.balance` as

DEFINITION 10. *We define the currency-footprint of a bank as follows:*

$$\texttt{CurrencyFootprint(b)} \triangleq \{\ (\texttt{b}, \texttt{ledger})\} \cup \{\ (\texttt{o}, \texttt{balance})\ \mid \exists k : \mathbb{N}. \texttt{x.ledger.next}^k = \texttt{o}\ \}$$

LEMMA 6.3. $M_{BA} \models \texttt{b} : \texttt{Bank} \to \texttt{CurrencyFootprint(b)}$ frames `b.currency`

We also define a predicate *Gives*(*prgx*, *prgy*, *prgz*) which expresses that while x was excuting, it passed to y access to z. ... more in handwritten notes ...

6.2.2 M_{BA} *adheres to* `Pol_2`.

LEMMA 6.4. $M_{BA} \models$ `Pol_2`

Proof sketch in Sophia's handwritten notes.

6.2.3 M_{BA} *adheres to* `Pol_4`.

LEMMA 6.5. $M_{BA} \models$ `Pol_4`

6.2.4 M_{BA} *adheres to* `Pol_7`.

LEMMA 6.6. $M_{BA} \models$ `Pol_7`

Proof sketch in Toby's handwritten notes.

6.3 Adherence to Policies for module $M_{BA'}$

In figure 11 we show the code for $M_{BA'}$in detail.

We give the code for $M_{BA'}$ in Figure ???, and define `Internal` as follows. The code works through ... The set `Internal` describes ....

6.3.1 $M_{BA'}$ *adheres to* `Pol_2`.

6.3.2 $M_{BA'}$ *adheres to* `Pol_4`.

6.3.3 $M_{BA'}$ *adheres to* `Pol_7`.

7 HOARE LOGIC

Here we give Hoare Logic rules which allow us to prove adherence to policies. NOTE: Not sure when we will get to do these rules. If we get to define these rules, then we will use them for the proofs in section 6.2 and we will swap section 6 and section 7

8 FURTHER APPLICATIONS

In this section we will apply our methodology to give specifications to other famous patterns from the OCAP literature, *i.e.* the membrane, the DOC-tree, and ...

8.1 The DOM tree

....

```
class Bank {     }

class Account {
  protected field balance; // the data of the Account;
  protected field myBank;  // a Bank

  Account(aBank,amt){
    balance = amt;
    myBank = aBank }

  fun deposit(source, amnt){
    if ( myBank== source.myBank && source.balance>=amt && amt>0) then
       { source.balance = source.balance-amt;
          this.balance = this.balance + amt }     }
}
```

Fig. 11. $\mathtt{M}_{\mathtt{BA}'}$: Second version of the Bank example, in detail

```
class Box{
  ...
  fun seal(element){... }
  fun unseal(sealed){ ... }
 }
```

Fig. 12. Wrapping and Unwrappingl

8.2 The membrane

...

8.3 Sealer/Unsealer

In Figure 12 we visit the sealer/unsealer example from cite-Morrison and Miller.

The specification of these functions mandates `seal` wraps the object into another structure, and `unseal` unwraps the original object out of the structure, Like cite(David Swasey, OOPLSA'13), we use an uninterpreted predixate, $\mathtt{Wrapped}(\mathtt{z}, \mathtt{o}, \mathtt{b})$ which expresses that `z` contains the value of `o` as wrapped by `b`.

The policies from below express that `seal` wraps an object, while `unseal` unwraps it, and they are very similar to those from cite(David Swasey, OOPLSA'13).

$$\mathtt{Pol\_seal\_1} \triangleq \quad \begin{array}{l} \mathtt{o} : \mathtt{Object} \ \wedge \ \mathtt{b} : \mathtt{Box} \\ \qquad \{\ \mathtt{x} = \mathtt{b.seal(o)}\ \} \\ \mathtt{Wrapped}(\mathtt{x}, \mathtt{o}, \mathtt{b}) \end{array}$$

$$\mathtt{Pol\_seal\_2} \triangleq \quad \begin{array}{l} \mathtt{o} : \mathtt{Object} \ \wedge \ \mathtt{b} : \mathtt{Box} \ \wedge \ \mathtt{Wrapped}(\mathtt{x}, \mathtt{o}, \mathtt{b}) \\ \qquad \{\ \mathtt{y} = \mathtt{b.unseal(x)}\ \} \\ \mathtt{y} = \mathtt{o} \end{array}$$

But further to the specification from cite(David Swasey, OOPLSA'13) , we want to also express that the *only* way to extract a sealed object out of its box, is by calling the `unseal` function on the box. For this, we define an assertion `Sealed(o)` which expresses that *all* accesses to `o` go through some sealed box, and the assertion `UnSealed(o)` which expresses that the object can be accessed without going through a sealed box. Using these predicates, in `Pol_seal_3` we express that if a sealed object becomes unsealed, then this must have happened though a call to the `unseal` function:

$$\begin{aligned}\texttt{Sealed(o)} &\triangleq \texttt{o}:\texttt{Object} \land \\ &\quad \forall \texttt{o}':\texttt{Object}.\,[\,\mathcal{A}\mathit{ccess}(\texttt{o},\texttt{o}') \land \texttt{o} \neq \texttt{o}' \longrightarrow \exists \texttt{b}:\texttt{Box}.\texttt{Wrapped}(\texttt{o}',\texttt{o},\texttt{b})\,] \\ \texttt{Unsealed(o)} &\triangleq \neg\texttt{Sealed}(\mathit{prgo})\end{aligned}$$

$$\begin{aligned}\texttt{Pol\_seal\_3} \triangleq \forall \texttt{o}.\forall \texttt{S}.\;[\;(\texttt{Sealed(o)} \land \Diamond\texttt{Unsealed(o)})\;\text{in}\;\texttt{S} \;\longrightarrow \\ \exists \texttt{b},\texttt{x}.[\texttt{b},\texttt{x} \in \texttt{S} \land \texttt{Wrapped}(\texttt{x},\texttt{o},\texttt{b}) \land \\ (\Diamond\,\mathit{Calls}(\texttt{b.unseal(x)}))\;\text{in}\;\texttt{S}\;]\end{aligned}$$[18]

8.4 The DAO

We describe here only some aspects of the DAO contract. The DAO keeps a table called `balances` which keeps track of the balances for all its clients. The following three policies mandate that `Pol_DAO_1`: the contents of `o.balances` can only be affected by `o` itself, that `Pol_DAO_2`: the ether held in the DAO is the sum of the balances in all its participants, and `Pol_DAO_3`: that any participant can withdraw up to amount that the the DAO holds for it. In fact, with `Pol_DAO_2` is too strong, `Pol_DAO_1` and `Pol_DAO_3` are not, and any contract adhering to these two policies would have not suffered from the DAO-attack[**?**].

In the below we also have the use a lookup function *Caller* with obvious meaning[19].

$$\texttt{Pol\_DAO\_1} \triangleq \ldots\,\mathit{Changes}(\texttt{dao.balances(o)}) \longrightarrow \mathit{Caller} = \texttt{o}\ldots.$$

$$\texttt{Pol\_DAO\_2} \triangleq \ldots\texttt{dao.ether} = \textstyle\sum_{o \in dom(\texttt{dao.balances})} \texttt{dao.balances(o)}$$

$$\begin{aligned}\texttt{Pol\_DAO\_3} \triangleq \ldots\,\texttt{dao.balances(o)} = m \land \mathit{Caller} = \texttt{o} \land \mathit{Calls}(\texttt{payme}) \land \texttt{x} = m' \leq m \longrightarrow \\ \Diamond\,(\mathit{Caller} = \texttt{dao} \land \mathit{Calls}(\texttt{send}) \land \texttt{x} = m')\end{aligned}$$[20]

Note that `Pol_DAO_3` is a liveness property. It promises that if one of the `DAO` participants asks to be paid back (by calling `payMe`), then eventually they will get their money back (in the future the `DAO` will call the function `send` with the appropriate ether argument).

9 DISCUSSION

In this section we compare "classical" specifications with those proposed here[21]

- Classical specifications reflect over (*i.e.* mandate properties of) the state of the execution now, while holistic specifications can also reflect over the possible states reachable from now, or those states in the past which lead to the current state.

[18]This definition is not perfect, as it does not preclude that the call to `b.unseal(x)` could happen *after* `o` became `Unsealed`. Perhaps, we should instead have an assertion combinator A toA' causedby`code`, and we could then say:
$\texttt{Pol\_seal\_3} \triangleq \forall \texttt{o}.\forall \texttt{S}.\;[\;(\texttt{Sealed(o)}\;\text{in}\;\texttt{S})\;\text{to}\;(\texttt{Unsealed(o)}\;\text{in}\;\texttt{S})\;\text{caused by}\;\texttt{b.unseal(x)}\;]$
[19]we can define it with what we have
[20]SOPHIA: Here is a point where we want the parameters to have the meanings as in the future configuration! Easy to fix-stick in the calls names for the arguments
[21]shall we call them "holistic"?

- Classical specifications describe what happens under *correct* use of the data structure, while holistic specifications describe preservation of properties under *arbitrary* use of the data structure.
- Both specifications employ notions of footprint[22], but classical footprints are related to the carrier set[23] of properties in the current state, while because of the temporal operators of holistic specs, holistic witnesses[24] can also talk about the carrier sets[25] of arbitrary executions. Also, holistic witnesses are first order[26]
- Classical specifications describe sufficient conditions, while holistic specifications also describe necessary conditions. Namely, the assertion $A \longrightarrow \Diamond A'$ says that A is a *sufficient* condition to achieve the effect A' in the future, while $(\Diamond A') \longrightarrow A$ says that A is a *necessary* condition to achieve A' in the future.
- Defensive Programming Rather than "What can the others (client) achieve using my code" Say "What do I prevent the others from doing with my objects"
- From KVM paper: From the KVM paper
  "Contract interactions are often complex, with safe contracts able to have their guarantees violated by calling into potentially malicious and unknown third party contract code."
  Rather than say What can the others expect from me We say What do I have to make sure the others cannot do with my objects
  Elias Modularity is the ability to ...
  In order to deduce that xxx I must lookat the spec of yyy methods/
  Has modular verif gone too far?
  Modular verification should not imply modular specifications

BIBLIOGRAPHY

REFERENCES

[1] Mark Samuel Miller, Chip Morningstar, and Bill Frantz. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. Springer, 2000.

[22] In separation logic, and also others
[23] Witness??
[24] not sure whether witness of footprints
[25] Witness??
[26] I expect that in some sep logic they are first order too, but perhaps we can compare with the first version of sep logic