Holistic Specifications for Robust Programs

SOPHIA DROSSOPOULOU, Imperial College London, United Kingdom
JAMES NOBLE, Victoria University of Wellington, New Zealand
JULIAN MACKAY, Victoria University of Wellington, New Zealand
SUSAN EISENBACH, Imperial College London, United Kingdom

Functional specifications describe what program components *can* do: the *sufficient* conditions to invoke components' operations. They allow us to reason about the use of components in a *closed world* setting, where components interact with known client code, and where the client code must establish the appropriate pre-conditions before calling into a component.

Sufficient conditions are not enough to reason about the use of components in an *open world* setting, where components interact with external code, possibly of unknown provenance, and where components may evolve over time. In this open world setting, ensuring that your component is robust even when executing with buggy or malicious external code is critical. *Holistic specifications* — as their name implies — are concerned with the *overall* behaviour of a component, in all possible interleavings of calls to the component's operations with those of the external code. Thus, holistic specifications are concerned with *sufficient* conditions, *i.e.* what is enough to *cause* some effect, as well as with *necessary* conditions, *i.e.* what are the conditions without which an effect will *not* happen.

In this paper we propose the *Chainmail* specification language for writing *holistic* specifications that focus on necessary conditions (as well as sufficient conditions). We give a formal semantics for *Chainmail*, and discuss several examples. The core of *Chainmail* has been mechanised in the Coq proof assistant.

ACM Reference Format:
Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. 2020. Holistic Specifications for Robust Programs. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (July 2020), 37 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software guards our secrets, our money, our intellectual property, our reputation [47]. We entrust personal and corporate information to software which works in an *open* world, where it interacts with third party software of unknown provenance, possibly buggy and potentially malicious.

This means we need our software to be *robust*: to behave correctly even if used by erroneous or malicious third parties. We expect that our bank will only make payments from our account if instructed by us, or by somebody we have authorised, that space on a web given to an advertiser will not be used to obtain access to our bank details [43], or that a concert hall will not book the same seat more than once.

While language mechanisms such as private members, constants, invariants, object capabilities [39], and ownership [14] are *indispensable* to write robust programs, they cannot *ensure* that

Authors' addresses: Sophia DrossopoulouImperial College London, United Kingdom, scd@imperial.ac.uk; James Noble-Victoria University of Wellington, New Zealand, kjx@ecs.vuw.ac.nz; Julian MackayVictoria University of Wellington, New Zealand, julian.mackay@ecs.vuw.ac.nz; Susan EisenbachImperial College London, United Kingdom, susan@imperial.ac.uk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.

1049-331X/2020/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

programs are robust. Ensuring robustness is difficult because it means different things for different systems: perhaps that critical operations should only be invoked with the requisite authority; perhaps that sensitive personal information should not be leaked; or perhaps that a resource belonging to one user should not be consumed by another. To ensure robustness, we need ways to specify what robustness means for a particular program, and ways to demonstrate that the particular program adheres to its specific robustness requirements.

```
class Account {
    field balance
    field myBank

    method deposit(src,amt){
       if (amt>=0 && src.myBank=this.
           myBank && src.balance>=amt)
            then{
          this.balance = this.balance
              + amt
          src.balance = src.balance -
              amt
   }   }
}
```

```
class Account {
    field balance
    field myBank

    method deposit(dest,src,amt){
       ...as version 1...
    }  }
    method add(dest, amt){
       this.balance = this.balance +
           amt
    }
}
```

Fig. 1. Two Versions of the class Account

Consider the code snippets from fig. 1. Objects of class Account hold a reference to myBank, and only the holders of the references to accounts can move money between them, given the accounts are at the same bank.

We show the code in two versions; both have the same method deposit, and the last version has an additional method add. We use a Java-like syntax, and assume an untyped language (as we are in the open world setting). We assume that fields are private in the sense of C++ or Java, *i.e.* only methods of that class may read or write these fields, and that addresses are unforgeable, and so there is no way to guess a secret. Thus, the classical Hoare triple describing the behaviour of deposit would be:

(ClassicalSpec) $\triangleq$

$$\begin{array}{ll} \texttt{method deposit(src, amt)} & \\ \texttt{PRE:} & \texttt{this,src:Account} \wedge \texttt{this}\neq\texttt{src} \wedge \texttt{this.myBank=src.myBank} \wedge \\ & \texttt{amt:}\mathbb{N} \wedge \texttt{src.balance}\geq\texttt{amt} \\ \texttt{POST:} & \texttt{src.balance=src.balance}_{pre}\texttt{-amt} \wedge \texttt{this.balance=this.balance}_{pre}\texttt{+amt} \end{array}$$

(ClassicSpec) expresses that knowledge of the src account, and the accounts sharing the same bank, is *sufficient*, and that deposit cannot create money from nowhere. But it cannot preclude that Account – or some other class, for that matter – contains more methods which might make it possible to create "free" money. For this, we introduce *holistic specifications*, and require that:

(HolisticSpec) $\triangleq$

$$\begin{array}{l} \forall \mathrm{a}.[\ \ \mathrm{a} : \texttt{Account} \wedge \textsf{changes}\langle\ \texttt{a.balance}\ \rangle \ \longrightarrow \\ \qquad\qquad \exists \mathrm{o}.[\ \langle\ \mathrm{o}\ \textsf{calls}\ \mathrm{a}.\texttt{deposit}(\ \_,\_)\ \rangle \vee \langle\ \mathrm{o}\ \textsf{calls}\ \_.\texttt{deposit}(\ \mathrm{a},\_)\ \rangle\ ]\quad] \end{array}$$

(HolisticSpec) mandates that for any change in the value of an account, the deposit method must have been called on it, or it must have been the source to another accounts deposit. This prevents the add method above, where a.balance changes, but the deposit method isn't called.

Further holistic propositions are discussed in section 2 to provide other protections, such as against the leaking of accounts.

In this paper we propose *Chainmail*, a specification language to express holistic specifications. The design of *Chainmail* was guided by the study of a sequence of examples from the object-capability literature and the smart contracts world: the membrane [62], the DOM [19, 59], the Mint/Purse [39], the Escrow [17], the DAO [12, 15] and ERC20 [61]. As we worked through the examples, we found a small set of language constructs that let us write holistic specifications across a range of different contexts. In particular, *Chainmail* extends traditional program specification languages [31, 37] with features which talk about:

Permission: Which objects may have access to which other objects; this is central since access to an object usually also grants access to the functions it provides.

Control: Which objects called functions on other objects; this is useful in identifying the causes of certain effects - eg funds can only be reduced if the owner called a payment function.

Time: What holds some time in the past, the future, and what changes with time,

Space: Which parts of the heap are considered when establishing some property, or when performing program execution; a concept related to, but different from, memory footprints and separation logics,

Viewpoint: Which objects and which configurations are internal to our component, and which are external to it; a concept related to the open world setting.

While many individual features of *Chainmail* can be found in other work, their power and novelty for specifying open systems lies in their careful combination. The contributions of this paper are:

- the design of the holistic specification language *Chainmail*,
- the semantics of *Chainmail*, and a Coq mechanisation of its core,
- the application of *Chainmail* to a sequence of examples.

The rest of the paper is organised as follows: Section 2 gives an example from the literature which we will use to elucidate key points of *Chainmail*. Section 3 presents the *Chainmail* specification language. Section 4 introduces the formal model underlying *Chainmail*, and then section 5 defines the semantics of *Chainmail*'s assertions. Section 6 discusses our design, section 7 shows how key points of exemplar problems can be specified in *Chainmail*. Section 8 discusses the Coq model, section 9 considers related work, and section 10 concludes. We relegate various details to appendices.

2 MOTIVATING EXAMPLE: THE BANK

As a motivating example, we expand on our simplified banking application, with objects representing `Account`s or `Bank`s. As in [41], `Account`s belong to `Bank`s and hold money (`balance`s); with access to two `Account`s of the same `Bank` one can transfer any amount of money from one to the other. We give a traditional specification in fig. 2.

The *pre*-condition of `deposit` requires that the receiver and the first argument (`this` and `src`) are `Account`s and belong to the same bank, that the second argument (`amt`) is a number, and that `src`'s balance is at least `amt`. The *post*-condition mandates that `amt` has been transferred from `src` to the receiver. The function `makeAccount` returns a fresh `Account` with the same bank, and transfers `amt` from the receiver `Account` to the new `Account`. Finally, the function `newAccount` when run by a `Bank` creates a new `Account` with the corresponding amount of money in it.[1]

[1] Note that our very limited bank specification doesn't even have the concept of an account owner.

(ClassicSpec) ≜

```
method deposit(src, amt)
PRE:  this,src:Account ∧ this≠src ∧ this.myBank=src.myBank ∧
      amt:ℕ ∧  src.balance≥amt
POST: src.balance=src.balance$_{pre}$-amt ∧ this.balance=this.balance$_{pre}$+amt

method makeAccount(amt)
PRE: this:Account ∧ amt:ℕ ∧ this.balance≥amt
POST: this.balance=this.balance$_{pre}$-amt ∧ fresh result ∧
      result: Account ∧ this.myBank=result.myBank ∧ result.balance=amt

method newAccount(amt)
PRE:  this:Bank
POST:  result: Account ∧ result.myBank=this ∧ result.balance=amt
```

Fig. 2. Functional specification of Bank and Account

With such a specification the code below satisfies its assertion. Assume that acm_acc and auth_acc are Accounts for the ACM and for a conference paper author respectively. The ACM's acm_acc has a balance of 10,000 before an author is registered, while afterwards it has a balance of 11,000. Meanwhile the auth_acc's balance will be 500 from a starting balance of 1,500 (barely enough to buy a round of drinks at the conference hotel bar).

```
assume acm_acc,auth_acc: Account ∧ acm_acc.balance=10000 ∧ auth_acc.balance=1500
acm_acc.deposit(auth_acc,1000)
assert acm_acc.balance=11000  ∧ auth_acc.balance=500
```

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 systems which we certainly do not control. Returning to our author, say some time after registering by executing the deposit code above, 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, what if the bank provided a steal method that emptied out every account in the bank into a thief's account. If this method existed and if it were somehow called between registering at the conference and going to the bar, then the author would find an empty account.

The critical problem is that a bank implementation including a steal method would meet the functional specification of the bank from fig. 2, so long as the methods deposit, makeAccount, and newAccount meet their specification.

One obvious solution would be to return to a closed-world interpretation of specifications: we interpret specifications such as fig. 2 as *exact* in the sense that only implementations that meet the functional specification exactly, *with no extra methods or behaviour*, 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 during software maintenance was given a new method count that simply counted the number of deposits that had taken place, or a method notify to enable the bank to occasionally send notifications to its customers.

What we need is some way to permit bank implementations that send notifications to customers, but to forbid implementations of steal. The key here is to capture the (implicit) assumptions underlying fig. 2, and to provide additional specifications that capture those assumptions. The following three informal requirements prevent methods like `steal`:

(1) An account's balance can be changed only if a client calls the `deposit` method with the account as the receiver or as an argument.
(2) An account's balance can be changed only if a client has access to that particular account.
(3) The `Bank/Account` component does not leak access to existing accounts or banks.

Compared with the functional specification we have seen so far, these requirements capture *necessary* rather than *sufficient* conditions: Calling the `deposit` method to gain access to an account is necessary for any change to that account taking place. The function `steal` is inconsistent with requirement (1), as it reduces the balance of an `Account` without calling the function `deposit`. However, requirement (1) is not enough to protect our money. We need to (2) to avoid an `Account`'s balance getting modified without access to the particular `Account`, and (3) to ensure that such accesses are not leaked.

We can express these requirements through *Chainmail* assertions. Rather than specifying the behaviour of particular methods when they are called, we write assertions that range across the entire behaviour of the `Bank/Account` module.

$$
\begin{aligned}
(1)\ \triangleq\ & \forall \mathtt{a}.[\ \mathtt{a} : \mathtt{Account} \land \mathsf{changes}\langle\, \mathtt{a.balance}\,\rangle\ \longrightarrow \\
& \qquad \exists \mathtt{o}.[\ \langle\, \mathtt{o}\ \mathsf{calls}\ \mathtt{a.deposit}(\ \_, \_)\ \rangle \lor \langle\, \mathtt{o}\ \mathsf{calls}\ \_.\mathtt{deposit}(\ \mathtt{a}, \_)\ \rangle\ ]\ \ ]
\end{aligned}
$$

$$
\begin{aligned}
(2)\ \triangleq\ & \forall \mathtt{a}.\forall \mathtt{S} : \mathtt{Set}.\ [\ \mathtt{a} : \mathtt{Account}\ \land\ \langle\, \mathsf{will}\langle\ \mathsf{changes}\langle\, \mathtt{a.balance}\,\rangle\rangle\ \mathsf{in}\ \mathtt{S}\ \rangle \quad \longrightarrow \\
& \qquad\qquad\qquad \exists \mathtt{o}.\ [\ \mathtt{o} \in \mathtt{S}\ \land\ \mathsf{external}\langle\, \mathtt{o}\,\rangle\ \land\ \langle\, \mathtt{o}\ \mathsf{access}\ \mathtt{a}\,\rangle\ ]\quad]
\end{aligned}
$$

$$
\begin{aligned}
(3)\ \triangleq\ & \forall \mathtt{a}.\forall \mathtt{S} : \mathtt{Set}.\ [\ \mathtt{a} : \mathtt{Account}\ \land\ \ \langle\, \mathsf{will}\langle\ \exists \mathtt{o}.[\ \mathsf{external}\langle\, \mathtt{o}\,\rangle\ \land\ \langle\, \mathtt{o}\ \mathsf{access}\ \mathtt{a}\,\rangle]\ \rangle\ \mathsf{in}\ \mathtt{S}\,\rangle \\
& \qquad\qquad \longrightarrow\ \ \exists \mathtt{o}'.\ [\ \mathtt{o}' \in \mathtt{S}\ \land\ \mathsf{external}\langle\, \mathtt{o}'\,\rangle\ \land\ \langle\, \mathtt{o}'\ \mathsf{access}\ \mathtt{a}\,\rangle\ ]\quad]
\end{aligned}
$$

In the above and throughout the paper, we use an underscore (_) to indicate an existentially bound variable whose value is of no interest.

Assertion (1) says that if an account's balance changes ($\mathsf{changes}\langle\, \mathtt{a.balance}\,\rangle$), then there must be some client object `o` that called the `deposit` method with `a` as a receiver or as an argument ($\langle\, \mathtt{o}\ \mathsf{calls}\ \_.\mathtt{deposit}(\ \_)\ \rangle$).

Assertion (2) similarly constrains any possible change to an account's balance: If at some future point the balance changes ($\mathsf{will}\langle\ \mathsf{changes}\langle\, ...\,\rangle\,\rangle$), and if this future change is observed with the state restricted to the objects from `S` (*i.e.* $\langle\, ...\ \mathsf{in}\ \mathtt{S}\,\rangle$), then at least one of these objects ($\mathtt{o} \in \mathtt{S}$) is external to the `Bank/Account` system ($\mathsf{external}\langle\, \mathtt{o}\,\rangle$) and has (direct) access to that account object ($\langle\, \mathtt{o}\ \mathsf{access}\ \mathtt{a}\,\rangle$). Notice that while the change in the `balance` happens some time in the future, the external object `o` has access to `a` in the *current* state. Notice also, that the object which makes the call to `deposit` described in (1), and the object which has access to `a` in the current state described in (2) need not be the same: It may well be that the latter passes a reference to `a` to the former (indirectly), which then makes the call to `deposit`.

It remains to think about how access to an `Account` may be obtained. This is the remit of assertion (3): It says that if at some time in the future of the state restricted to `S`, some object `o` which is external has access to some account `a`, and if `a` exists in the current state, then in the current state some object from `S` has access to `a`. Where `o` and `o′` may, but need not, be the same object. And where `o′` has to exist and have access to `a` in the *current* state, but `o` need not exist in the current state – it may be allocated later.

A holistic specification for the bank account, then, would be our original sufficient functional specification from fig. 2 plus the necessary specifications (1)-(3) from above. This holistic specification permits an implementation of the bank that also provides `count` and `notify` methods, even though the specification does not mention either method. Critically, though, the *Chainmail* specification does not permit an implementation that includes a `steal` method. First, the `steal` method clearly changes the balance of every account in the bank, but assertion (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 caller having a reference to most of those accounts, thus breaching assertion (2).

Assertion (3) gives essential protection when dealing with foreign, untrusted code. When an `Account` is given out to untrusted third parties, assertion (3) guarantees that this `Account` cannot be used to obtain access to further `Account`s. The `ACM` does not trust its authors, and certainly does not want to give them access to `acm_acc`, which contains all of the `ACM`'s money. Instead, in order to receive money, it will pass a secondary account used for incoming funds, `acm_incoming`, into which an `author` will pay the fee, and from which the `ACM` will transfer the money back into the main `acm_acc`. Assertion (3) is crucial, because it guarantees that even malicious authors could not use knowledge of `acm_incoming` to obtain access to the main `acm_acc` or any other account.

In summary, our necessary specifications are assertions that describe the behaviour of a module as observed by its clients. These assertions can talk about space, time and control, and thus go beyond class invariants, which can only talk about relations between the values of the fields of the objects. As our invariants are (usually) independent of concrete implementation details, they do not constrain the code to a specific implementation.

3 *Chainmail* OVERVIEW

In this section we give a brief and informal overview of some of the most salient features of *Chainmail*– a full exposition appears in Section 5.

Example Configurations We will illustrate these features using the `Bank/Account` example from the previous section. We use the runtime configurations σ_1 and σ_2 shown in the left and right diagrams in Figure 3. In both diagrams the rounded boxes depict objects: green for those from the `Bank/Account` component, and grey for the "external", "client" objects. The light green area shows which objects are contained by the `Bank/Account` component. The object at `1` is a `Bank`, those at `2`, `3` and `4` are `Account`s, and those at `91`, `92`, `93` and `94` are "client" objects which belong to classes different from those from the `Bank/Account` module.

Each configuration represents one alternative implementation of the Bank object. Configuration σ_1 may arise from execution using a module M_{BA1}, where `Account` objects have a field `myBank` pointing to their `Bank`, and an integer field `balance` – the code can be found in appendix C Fig. 9. Configuration σ_2 may arise from execution using a module M_{BA2}, where `Account`s have a `myBank` field, `Bank` objects have a `ledger` implemented though a sequence of `Node`s, each of which has a field pointing to an `Account`, a field `balance`, and a field `next` – the code can be found in appendix C Figs. 12 and 10.

For the rest, assume variable identifiers $\texttt{b}_1$, and $\texttt{a}_2$–$\texttt{a}_4$, and $\texttt{u}_{91}$–$\texttt{u}_{94}$ denoting objects `1`, `2`–`4`, and `91`–`94` respectively for both σ_1 and σ_2. That is, for i=1 or i=2, $\sigma_i(\texttt{b}_1)$=`1`, $\sigma_i(\texttt{a}_2)$=`2`, $\sigma_i(\texttt{a}_3)$=`3`, $\sigma_i(\texttt{a}_4)$=`4`, $\sigma_i(\texttt{u}_{91})$=`91`, $\sigma_i(\texttt{u}_{92})$=`92`, $\sigma_i(\texttt{u}_{93})$=`93`, and $\sigma_i(\texttt{u}_{94})$=`94`.

Classical Assertions talk about the contents of the local variables (*i.e.* the topmost stack frame), and the fields of the various objects (*i.e.* the heap). For example, the assertion $\texttt{a}_2\texttt{.myBank=a}_3\texttt{.myBank}$, says that $\texttt{a}_2$ and $\texttt{a}_3$ have the same bank. In fact, this assertion is satisfied in both σ_1 and σ_2, written

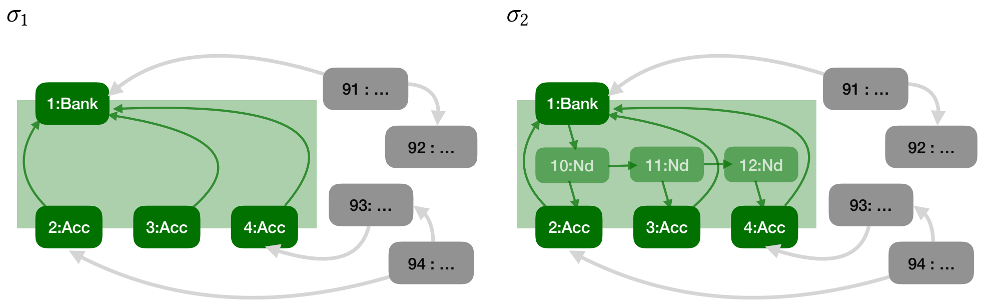


Fig. 3. Two runtime configurations for the `Bank`/`Account` example.

formally as

$$\ldots, \sigma_1 \;\models\; \mathtt{a}_2\mathtt{.myBank} = \mathtt{a}_3\mathtt{.myBank}$$
$$\ldots, \sigma_2 \;\models\; \mathtt{a}_2\mathtt{.myBank} = \mathtt{a}_3\mathtt{.myBank}.$$

The term `x`:`ClassId` says that `x` is an object of class `ClassId`. For example

$$\ldots, \sigma_1 \;\models\; \mathtt{a}_2\mathtt{.myBank} : \mathtt{Bank}.$$

We support ghost fields [11, 31], *e.g.* $\mathtt{a}_1$`.balance` is a physical field in σ_1 and a ghost field in σ_2 since in `MBA2` an `Account` does not store its `balance` (as can be seen in appendix C Fig. 12). We also support the usual logical connectives, and so, we can express assertions such as

$$\forall \mathtt{a}.[\;\; \mathtt{a} : \mathtt{Account} \;\;\longrightarrow\;\; \mathtt{a.myBank} : \mathtt{Bank} \;\wedge\; \mathtt{a.balance} \geq 0 \;\;]\;.$$

Permission: Access Our first holistic assertion, $\langle\, \mathtt{x}\ \text{access}\ \mathtt{y} \,\rangle$, asserts that object `x` has a direct reference to another object `y`: either one of `x`'s fields contains a reference to `y`, or the receiver of the currently executing method is `x`, and `y` is one of the arguments or a local variable. For example:

$$\ldots, \sigma_1 \;\models\; \langle\, \mathtt{a}_2\ \text{access}\ \mathtt{b}_1 \,\rangle$$

If σ_1 were executing the method body corresponding to the call $\mathtt{a}_2$`.deposit(`$\mathtt{a}_3$`,360)`, then we would have

$$\ldots, \sigma_1 \;\models\; \langle\, \mathtt{a}_2\ \text{access}\ \mathtt{a}_3 \,\rangle,$$

That is, during execution of `deposit`, the object at $\mathtt{a}_2$ has access to the object at $\mathtt{a}_3$, and could, if the method body chose to, call a method on $\mathtt{a}_3$, or store a reference to $\mathtt{a}_3$ in its own fields. Access is not symmetric, nor transitive:

$$\ldots, \sigma_1 \;\not\models\; \langle\, \mathtt{a}_3\ \text{access}\ \mathtt{a}_2 \,\rangle,$$
$$\ldots, \sigma_2 \;\models\; \langle\, \mathtt{a}_2\ \text{access}\ \mathtt{b}_1 \,\rangle \wedge \langle\, \mathtt{b}_1\ \text{access}\ \mathtt{nd}_{10} \,\rangle, \qquad \ldots, \sigma_2 \;\not\models\; \langle\, \mathtt{a}_2\ \text{access}\ \mathtt{nd}_{10} \,\rangle.$$

Control: Calls The assertion $\langle\, \mathtt{x}\ \text{calls}\ \mathtt{y.m}(\ \mathtt{zs})\ \,\rangle$ holds in configurations where a method on object `x` makes a method call `y.m(zs)` — that is it calls method `m` with object `y` as the receiver, and with arguments `zs`. For example,

$$\ldots, \sigma_3 \models \langle\, \mathtt{x}\ \text{calls}\ \mathtt{a}_2\mathtt{.deposit}(\ \mathtt{a}_3, 360)\ \,\rangle.$$

means that the receiver in σ_3 is `x`, and that $\mathtt{a}_2$`.deposit(`$\mathtt{a}_3$`,360)` is the next statement to be executed.

Space: In The space assertion $\langle\, A\ \text{in}\ \mathtt{S} \,\rangle$ establishes validity of A in a configuration restricted to the objects from the set `S`. For example, if object `94` is included in $\mathtt{S}_1$ but not in $\mathtt{S}_2$, then we have

$$\ldots, \sigma_1 \;\models\; \langle\, (\exists \mathtt{o}.\, \langle\, \mathtt{o}\ \text{access}\ \mathtt{a}_4 \,\rangle)\ \text{in}\ \mathtt{S}_1 \,\rangle$$
$$\ldots, \sigma_1 \;\not\models\; \langle\, (\exists \mathtt{o}.\, \langle\, \mathtt{o}\ \text{access}\ \mathtt{a}_4 \,\rangle)\ \text{in}\ \mathtt{S}_2 \,\rangle.$$

The set `S` in the assertion $\langle\, A\ \text{in}\ \mathtt{S} \,\rangle$ is therefore *not* the footprint of A; it is more like the *fuel* [2] given to establish that assertion. Note that $..., \sigma \models \langle\, A \,\mathsf{in}\, \mathtt{S} \,\rangle$ does not imply $..., \sigma \models A$ nor does it imply $..., \sigma \models \langle\, A \,\mathsf{in}\, \mathtt{S} \cup \mathtt{S}' \,\rangle$. The other direction of the implication does not hold either.

Time: Next, Will, Prev, Was We support several operators from temporal logic: ($\mathsf{next}\langle\, A \,\rangle$, $\mathsf{will}\langle\, A \,\rangle$, $\mathsf{prev}\langle\, A \,\rangle$, and $\mathsf{was}\langle\, A \,\rangle$) to talk about the future or the past in one or more steps. The assertion $\mathsf{will}\langle\, A \,\rangle$ expresses that A will hold in one or more steps. For example, taking σ_4 to be similar to σ_2, the next statement to be executed to be $\mathtt{a}_2\mathtt{.deposit(a}_3\mathtt{, 360)}$, and $\mathtt{M}_{BA2} \fatsemi ..., \sigma_4 \models \mathtt{a}_2\mathtt{.balance} = 60$, and that $\mathtt{M}_{BA2} \fatsemi ..., \sigma_4 \models \mathtt{a}_4\mathtt{.balance} \geq 360$, then

$$\mathtt{M}_{BA2} \fatsemi ..., \sigma_4 \models \mathsf{will}\langle\, \mathtt{a}_2\mathtt{.balance} = 420 \,\rangle.$$

The *internal* module, $\mathtt{M}_{BA2}$ is needed for looking up the method body of `deposit`.

Viewpoint: – External The assertion $\mathsf{external}\langle\, \mathtt{x} \,\rangle$ expresses that the object at `x` does not belong to the module under consideration. For example,

$$\mathtt{M}_{AB2} \fatsemi ..., \sigma_2 \models \mathsf{external}\langle\, \mathtt{u}_{92} \,\rangle, \qquad \mathtt{M}_{AB2} \fatsemi ..., \sigma_2 \not\models \mathsf{external}\langle\, \mathtt{a}_2 \,\rangle,$$
$$\mathtt{M}_{AB2} \fatsemi ..., \sigma_2 \not\models \mathsf{external}\langle\, \mathtt{b}_1\mathtt{.ledger} \,\rangle$$

The *internal* module, $\mathtt{M}_{BA2}$, is needed to judge which objects are internal or external.

Change and Authority: We have used $\mathsf{changes}\langle\, ... \,\rangle$ in our *Chainmail* assertions in section 2, as in $\mathsf{changes}\langle\, \mathtt{a.balance} \,\rangle$. Assertions that talk about change, or give conditions for change to happen are fundamental for security; the ability to cause change is called *authority* in [39]. We can encode change using the other features of *Chainmail*, namely, for any expression `e`:

$$\mathsf{changes}\langle\, \mathtt{e} \,\rangle \;\equiv\; \exists v.[\; \mathtt{e} = v \wedge \mathsf{next}\langle\, \neg(\mathtt{e} = v) \,\rangle\;].$$

and similarly for assertions.

Putting these together We now look at some composite assertions which use several features from above. For example, the assertion below says that if the statement to be executed is $\mathtt{a}_2\mathtt{.deposit(a}_3\mathtt{, 60)}$, then the balance of $\mathtt{a}_2$ will eventually change:

$$\mathtt{M}_{BA2} \fatsemi ..., \sigma_2 \models \langle\, ..\, \mathsf{calls}\; \mathtt{a}_2\mathtt{.deposit(}\; \mathtt{a}_3, 60\mathtt{)} \;\rangle \longrightarrow \mathsf{will}\langle\, \mathsf{changes}\langle\, \mathtt{a}_2\mathtt{.balance} \,\rangle \,\rangle.$$

Now look deeper into space assertions, $\langle\, A \,\mathsf{in}\, \mathtt{S} \,\rangle$, which allow us to characterise the set of objects which have authority over certain effects (here A). In particular, the assertion $\langle\, \mathsf{will}\langle\, A \,\rangle \,\mathsf{in}\, \mathtt{S} \,\rangle$ requires two things: i) that A will hold in the future, and ii) that all the objects which cause the effect which will make A valid are included in `S`. Knowing who has, and who has not, authority over properties or data is a fundamental concern of robustness [39]. Notice that the authority is a set, rather than a single object: quite often it takes *several objects in concert* to achieve an effect.

Matthew Ross: *Wait, didn't we say that "Note that $..., \sigma \models \langle\, A \,\mathsf{in}\, \mathtt{S} \,\rangle$ does not imply $..., \sigma \models A$ nor does it imply $..., \sigma \models \langle\, A \,\mathsf{in}\, \mathtt{S} \cup \mathtt{S}' \,\rangle$.". I get what it's getting at, but at face-value it doesn't seem valid*

Consider assertions (2) and (3) from the previous section. They both have the form "$\mathsf{will}\langle\, \langle\, A \,\mathsf{in}\, \mathtt{S} \,\rangle \,\rangle \longrightarrow P(\mathtt{S})$", where P is some property over a set. These assertions say that if ever in the future A becomes valid, and if the objects involved in making A valid are included in `S`, then `S` must satisfy P. Such assertions can be used to restrict whether A will become valid. If we have some execution which only involves objects which do not satisfy P, then we know that the execution will not ever make A valid.

In summary, in addition to classical logical connectors and classical assertions over the contents of the heap and the stack, our holistic assertions draw from some concepts from object capabilities ($\langle\, \_\; \mathsf{access}\; \_ \,\rangle$ for permission; $\langle\, \_\; \mathsf{calls}\; \_.\_(\,\_\,) \,\rangle$ and $\mathsf{changes}\langle\, \_ \,\rangle$ for authority) as well as temporal logic ($\mathsf{will}\langle\, A \,\rangle$, $\mathsf{was}\langle\, A \,\rangle$ and friends), and the relation of our spatial connective ($\langle\, A \,\mathsf{in}\, S \,\rangle$) with ownership and effect systems [13, 14, 60].

The next two sections discuss the semantics of *Chainmail*. Section 4 contains an overview of the formal model and section 5 focuses on the most important part of *Chainmail*: assertions.

4 OVERVIEW OF THE FORMAL FOUNDATIONS

We now give an overview of the formal model for *Chainmail*. In section 4.1 we introduce the shape of the judgments used to give semantics to *Chainmail*, while in section 4.2 we describe the most salient aspects of an underlying programming language used in *Chainmail*.

4.1 *Chainmail* judgments

Having outlined the ingredients of our holistic specification language, the next question to ask is: When does a module $\mathtt{M}$ satisfy a holistic assertion A? More formally: when does $\mathtt{M} \models A$ hold?

Our answer has to reflect the fact that we are dealing with an *open world*, where $\mathtt{M}$, our module, may be linked with *arbitrary untrusted code*. To model the open world, we consider pairs of modules, $\mathtt{M} \fatsemi \mathtt{M}'$, where $\mathtt{M}$ is the module whose code is supposed to satisfy the assertion, and $\mathtt{M}'$ is another module which exercises the functionality of $\mathtt{M}$. We call our module $\mathtt{M}$ the *internal* module, and $\mathtt{M}'$ the *external* module, which represents potential attackers or adversaries.

We can now answer the question: $\mathtt{M} \models A$ holds if for all further, *potentially adversarial*, modules $\mathtt{M}'$ and in all runtime configurations σ which may be observed as arising from the execution of the code of $\mathtt{M}$ combined with that of $\mathtt{M}'$, the assertion A is satisfied. More formally, we define:

$$\mathtt{M} \models A \qquad \text{if} \qquad \forall \mathtt{M}'.\forall \sigma \in \mathcal{A}rising(\mathtt{M} \fatsemi \mathtt{M}').[\ \mathtt{M} \fatsemi \mathtt{M}', \sigma \models A\ ].$$

Module $\mathtt{M}'$ represents all possible clients of $\mathtt{M}$. As it is arbitrarily chosen, it reflects the open world nature of our specifications.

The judgement $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models A$ means that assertion A is satisfied by $\mathtt{M} \fatsemi \mathtt{M}'$ and σ. As in traditional specification languages [31, 37], satisfaction is judged in the context of a runtime configuration σ; but in addition, it is judged in the context of the internal and external modules. These are used to find abstract functions defining ghost fields as well as method bodies needed when judging validity of temporal assertions such as $\mathsf{will}\langle\_\rangle$.

We distinguish between internal and external modules. This has two uses: First, *Chainmail* includes the "$\mathsf{external}\langle\ \mathsf{o}\ \rangle$" assertion to require that an object belongs to the external module, as in the Bank Account's assertion (2) and (3) in section 2. Second, we adopt a version of visible states semantics [25, 38, 45], treating all executions within a module as atomic. We only record runtime configurations which are *external* to module $\mathtt{M}$, *i.e.* those where the executing object (*i.e.* the current receiver) comes from module $\mathtt{M}'$. Execution has the form

$$\mathtt{M} \fatsemi \mathtt{M}', \sigma \rightsquigarrow \sigma'$$

where we ignore all intermediate steps with receivers internal to $\mathtt{M}$. In the next section we shall outline the underlying programming language, and define the judgment $\mathtt{M} \fatsemi \mathtt{M}', \sigma \rightsquigarrow \sigma'$ and the set $\mathcal{A}rising(\mathtt{M} \fatsemi \mathtt{M}')$.

4.2 An underlying programming language, $\mathcal{L}_{oo}$

The meaning of *Chainmail* assertions is parametric with an underlying object-oriented programming language, with modules as repositories of code, classes with fields, methods and ghostfields, objects described by classes, a way to link modules into larger ones, and a concept of program execution[2].

We have developed $\mathcal{L}_{oo}$, a minimal such object-oriented language, which we outline in this section. We describe the novel aspects of $\mathcal{L}_{oo}$, and summarise the more conventional parts, relegating full, and mostly unsurprising, definitions to Appendix A.

Modules are central to $\mathcal{L}_{oo}$, as they are to *Chainmail*. As modules are repositories of code, we adopt the common formalisation of modules as maps from class identifiers to class definitions, c.f. Appendix, Def. 15. We use the terms module and component in an analogous manner to class and

[2] We believe that *Chainmail* can be applied to any language with these features.

object respectively. $\mathcal{L}_{oo}$ is untyped for several reasons. Many popular programming languages are untyped. The external module might be untyped, and so it is more general to consider everything as untyped. Finally, a solution that works for an untyped language will also apply to a typed language, while the converse is not true.

Class definitions consist of field, method and ghost field declarations, c.f. Appendix, Def. 16. Method bodies are sequences of statements, which can be field reads or field assignments, object creation, method calls, and return statements. Fields are private in the sense of C++ or Java: they can only be read or written by methods of the current class. This is enforced by the operational semantics, *c.f.* Fig. 8. We discuss ghost fields in the next section.

Runtime configurations, σ, contain all the usual information about execution snapshots: the heap, and a stack of frames. Each frame consists of a continuation, `contn`, describing the remaining code to be executed by the frame, and a map from variables to values. Values are either addresses or sets of addresses; sets are needed to deal with assertions which quantify over sets of objects, such as assertions (1) and (2) from section 2. We define *one-module* execution through a judgment of the form $\mathtt{M}, \sigma \leadsto \sigma'$ in the Appendix, Fig. 8.

We define a module linking operator $\circ$ so that $\mathtt{M}\circ\mathtt{M}'$ is the union of the two modules, provided that their domains are disjoint, c.f. Appendix, Def. 22. As we said in section 4.1, we distinguish between the internal and external module. We consider execution from the view of the external module, and treat execution of methods from the internal module as atomic. For this, we define *two-module execution* based on one-module execution as follows:

Definition 1. *Given runtime configurations σ, σ', and a module-pair $\mathtt{M}\fatsemi\mathtt{M}'$ we define execution where $\mathtt{M}$ is the internal, and $\mathtt{M}'$ is the external module as below:*

- $\mathtt{M}\fatsemi\mathtt{M}', \sigma \leadsto \sigma'$ *if there exist $n \geq 2$ and runtime configurations σ_1, ... σ_n, such that*
  - $\sigma{=}\sigma_1$, *and* $\sigma_n = \sigma'$.
  - $\mathtt{M}\circ\mathtt{M}', \sigma_i \leadsto \sigma'_{i+1}$, *for* $1 \leq i \leq n{-}1$
  - $Class(\mathtt{this})_\sigma \notin dom(\mathtt{M})$, *and* $Class(\mathtt{this})_{\sigma'} \notin dom(\mathtt{M})$,
  - $Class(\mathtt{this})_{\sigma_i} \in dom(\mathtt{M})$, *for* $2 \leq i \leq n{-}2$

In the definition above, $Class(\mathtt{x})_\sigma$ looks up the class of the object stored at $\mathtt{x}$, c.f. Appendix, Def. 19. For example, for σ_4 as in Section 3 whose next statement to be executed is $\mathtt{a_2.deposit(a_3, 360)}$, we would have a sequence of configurations σ_{41}, ... σ_{4n}, σ_5 so that the one-module execution gives $\mathtt{M}_{BA2}, \sigma_4 \leadsto \sigma_{41} \leadsto \sigma_{42} ... \leadsto \sigma_{4n} \leadsto \sigma_5$. This would correspond to an atomic evaluation in the two-module execution: $\mathtt{M}_{BA2}\fatsemi\mathtt{M}', \sigma_4 \leadsto \sigma_5$ (see Fig.4; where blue stands for $\sigma(this) \in M_1$,and orange for $\sigma(this) \in M_2$).

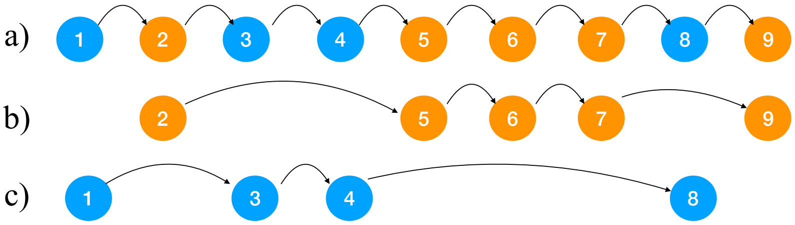


Fig. 4. Two Module Execution (Def. 1). a) $\mathtt{M}_1\circ\mathtt{M}_2$ b) $\mathtt{M}_1\fatsemi\mathtt{M}_2$ c) $\mathtt{M}_2\fatsemi\mathtt{M}_1$

Two-module execution is related to visible states semantics [45] as they both filter configurations, with the difference that in visible states semantics execution is unfiltered and configurations are only filtered when it comes to the consideration of class invariants while two-module execution filters execution. The lemma below says that linking is associative and commutative, and preserves both one-module and two-module execution.

Lemma 4.1 (Properties of linking). *For any modules* $\mathtt{M}, \mathtt{M}', \mathtt{M}''$*, and* $\mathtt{M}'''$ *and runtime configurations* σ*, and* σ' *we have*:

- $(\mathtt{M}\circ\mathtt{M}')\circ\mathtt{M}'' = \mathtt{M}\circ(\mathtt{M}'\circ\mathtt{M}'')$ *and* $\mathtt{M}\circ\mathtt{M}' = \mathtt{M}'\circ\mathtt{M}$.
- $\mathtt{M}, \sigma \rightsquigarrow \sigma'$*, and* $\mathtt{M}\circ\mathtt{M}'$ *is defined, implies* $\mathtt{M}\circ\mathtt{M}', \sigma \rightsquigarrow \sigma'$.
- $\mathtt{M} ⨟ \mathtt{M}', \sigma \rightsquigarrow \sigma'$ *implies* $(\mathtt{M}\circ\mathtt{M}'') ⨟ (\mathtt{M}'\circ\mathtt{M}'''), \sigma \rightsquigarrow \sigma'$.

We can now answer the question as to which runtime configurations are pertinent when judging a module's adherence to an assertion. *Initial configurations* are those whose heap have only one object, of class `Object`, and whose stack have one frame, with arbitrary continuation. *Arising* configurations are those that can be reached by two-module execution, starting from any initial configuration.

Definition 2 (Initial and Arising Configurations). *defined as follows, using the semantics from Definition 18:*

- $\mathcal{I}nitial\langle(\psi, \chi)\rangle$*, if* ψ *consists of a single frame* ϕ *with* $dom(\phi) = \{\mathtt{this}\}$*, and there exists some address* α*, such that* $\lfloor\mathtt{this}\rfloor_\phi=\alpha$*, and* $dom(\chi)=\alpha$*, and* $\chi(\alpha) = (\mathtt{Object}, \emptyset)$.
- $\mathcal{A}rising(\mathtt{M} ⨟ \mathtt{M}') = \{\ \sigma \mid \exists\sigma_0.\ [\ \mathcal{I}nitial\langle\sigma_0\rangle \ \wedge\ \mathtt{M} ⨟ \mathtt{M}', \sigma_0 \rightsquigarrow^* \sigma\ \ ]\ \}$

5 ASSERTIONS

Chainmail assertions (details in appendix 5.4) consist of (pure) expressions `e`, comparisons between expressions, classical assertions about the contents of heap and stack, the usual logical connectives, as well as our holistic concepts. In this section we focus on the novel, holistic, features of *Chainmail* (permission, control, time, space, and viewpoint), as well as our wish to support some form of recursion while keeping the logic of assertions classical.

5.1 Syntax of Assertions

We now define the syntax and semantics of expressions and holistic assertions. The novel, holistic, features of *Chainmail* (permission, control, time, space, and viewpoint), as well as our wish to support some form of recursion while keeping the logic of assertions classical, introduced challenges, which we discuss in this section.

5.2 Syntax of Assertions

Definition 3 (Assertions). *Assertions consist of (pure) expressions* `e`*, classical assertions about the contents of heap/stack, the usual logical connectives, as well as our holistic concepts.*

$$
\begin{array}{lll}
\mathtt{e} & ::= & \mathsf{true}\ \mid\ \mathsf{false}\ \mid\ \mathsf{null}\ \mid\ \mathtt{x}\ \mid\ \mathtt{e} = \mathtt{e}\ \mid\ \mathsf{if}\,\mathtt{e}\,\mathsf{then}\,\mathtt{e}\,\mathsf{else}\,\mathtt{e}\ \mid\ \mathtt{e.f}(\ \mathtt{e}^*\ ) \\
 & & \\
A & ::= & \mathtt{e}\ \mid\ \mathtt{e} = \mathtt{e}\mid\ \mathtt{e} : \mathtt{ClassId}\ \mid\ \mathtt{e} \in \mathtt{S}\mid \\
 & & A \longrightarrow A\ \mid\ A \wedge A\ \mid\ A \vee A\ \mid\ \neg A\ \mid \\
 & & \forall\mathtt{x}.A\ \mid\ \forall\mathtt{S} : SET.A\ \mid\ \exists\mathtt{x}.A\ \mid\ \exists\mathtt{S} : SET.A\ \mid \\
 & & \langle\, x\ \mathsf{access}\ y\,\rangle\ \mid\ \langle\, \mathtt{x}\ \mathsf{calls}\ \mathtt{x.m}(\ \mathtt{x}^*)\ \rangle \\
 & & \mathsf{next}\langle\, A\,\rangle\ \mid\ \mathsf{will}\langle\, A\,\rangle\ \mid\ \mathsf{prev}\langle\, A\,\rangle\ \mid\ \mathsf{was}\langle\, A\,\rangle\ \mid \\
 & & \langle\, \mathtt{S}\ \mathsf{in}\ A\,\rangle\ \mid\ \mathsf{external}\langle\, \mathtt{x}\,\rangle \\
 & & \\
\mathtt{x, f, m} & ::= & \mathtt{Identifier}
\end{array}
$$

Expressions support calls with parameters (`e.f(e`*`)`); these are calls to ghostfield functions. This supports recursion at the level of expressions; therefore, the value of an expression may be undefined (either because of infinite recursion, or because the expression accessed undefined fields or variables). Assertions of the form `e=e`$'$ are satisfied only if both `e` and `e`$'$ are defined. Because we do not support recursion at the level of assertions, assertions from a classical logic (*e.g.* $A \lor \neg A$ is a tautology).

We will discuss evaluation of expressions in section 5.3, standard assertions about heap/stack and logical connectives in 5.4. We have discussed the treatment of permission, control, space, and viewpoint in the main text in the Definitions 3-7 in section 5.5 the treatment of time in Definitions 8,9 in the main text, section 5.6, We will discuss properties of assertions in Lemmas B.1-B.2. The judgement $\texttt{M} \fatsemi \texttt{M}', \sigma \models A$ expresses that A holds in $\texttt{M} \fatsemi \texttt{M}'$ and σ, and while $\texttt{M} \fatsemi \texttt{M}', \sigma \not\models A$ expresses that A does not hold in $\texttt{M} \fatsemi \texttt{M}'$ and σ.

5.3 Values of Expressions

The value of an expression is described through judgment $\texttt{M},\ \sigma, \texttt{e}\ \hookrightarrow\ v$, defined in Figure 5. We use the configuration, σ, to read the contents of the top stack frame (rule Var_Val) or the contents of the heap (rule Field_Heap_Val). We use the module, `M`, to find the ghost field declaration corresponding to the ghost field being used.

The treatment of fields and ghost fields is described in rules Field_Heap_Val, Field_Ghost_Val and Field_Ghost_Val2. If the field `f` exists in the heap, then its value is returned (Field_Heap_Val). Ghost field reads, on the other hand, have the form $\texttt{e}_0\texttt{.f(e}_1, ...\texttt{e}_n\texttt{)}$, and their value is described in rule Field_Ghost_Val: The lookup function $\mathcal{G}$ (defined in the obvious way in the Appendix, Def.17) returns the expression constituting the body for that ghost field, as defined in the class of $\texttt{e}_0$. We return that expression evaluated in a configuration where the formal parameters have been substituted by the values of the actual parameters.

Ghost fields support recursive definitions. For example, imagine a module $\texttt{M}_0$ with a class `Node` which has a field called `next`, and which had a ghost field `last`, which finds the last `Node` in a sequence and is defined recursively as

`if this.next=null then this else this.next.last`,

and another ghost field `acyclic`, which expresses that a sequence is acyclic, defined recursively as

`if this.next=null then true else this.next.acyclic`.

The relation $\hookrightarrow$ is partial. For example, assume a configuration σ_0 where `acyc` points to a `Node` whose field `next` has value `null`, and `cyc` points to a `Node` whose field `next` has the same value as `cyc`. Then, $\texttt{M}_0, \sigma_0, \texttt{acyc.acyclic}\ \hookrightarrow\ \texttt{true}$, but we would have no value for $\texttt{M}_0, \sigma_0, \texttt{cyc.last}\ \hookrightarrow\ ...$, nor for $\texttt{M}_0, \sigma_0, \texttt{cyc.acyclic}\ \hookrightarrow\ ....$

Notice also that for an expression of the form `e.f`, both Field_Heap_Val and Field_Ghost_Val2 could be applicable: rule Field_Heap_Val will be applied if `f` is a field of the object at `e`, while rule Field_Ghost_Val will be applied if `f` is a ghost field of the object at `e`. We expect the set of fields and ghost fields in a given class to be disjoint. This allows a specification to be agnostic over whether a field is a physical field or just ghost information. For example, assertions (1) and (2) from section 2 talk about the `balance` of an `Account`. In module $\texttt{M}_{BA1}$ (Appendix C), where we keep the balances in the account objects, this is a physical field. In $\texttt{M}_{BA2}$ (also in Appendix C), where we keep the balances in a ledger, this is ghost information.

5.4 Satisfaction of Assertions - standard

We now define the semantics of assertions involving expressions, the heap/stack, and logical connectives. The semantics are unsurprising, except, perhaps the relation between validity of assertions and the values of expressions.

$$\textsf{True\_Val}\quad \frac{}{\texttt{M},\ \sigma, \mathsf{true} \hookrightarrow \mathsf{true}} \qquad \textsf{False\_Val}\quad \frac{}{\texttt{M},\ \sigma, \mathsf{false} \hookrightarrow \mathsf{false}} \qquad \textsf{Null\_Val}\quad \frac{}{\texttt{M},\ \sigma, \mathsf{null} \hookrightarrow \mathsf{null}} \qquad \textsf{Var\_Val}\quad \frac{}{\texttt{M},\ \sigma, \texttt{x} \hookrightarrow \sigma(\texttt{x})}$$

$$\textsf{Field\_Heap\_Val}\quad \frac{\texttt{M},\ \sigma, \texttt{e} \hookrightarrow \alpha \qquad \sigma(\alpha, \texttt{f}) = v}{\texttt{M},\ \sigma, \texttt{e.f} \hookrightarrow v}$$

$$\textsf{Field\_Ghost\_Val}\quad \frac{\begin{array}{l}\texttt{M},\ \sigma, \texttt{e}_0 \hookrightarrow \alpha \\ \texttt{M},\ \sigma, \texttt{e}_i \hookrightarrow v_i \quad i \in \{1..n\} \\ \mathcal{G}(\texttt{M}, Class(\alpha)_\sigma, \texttt{f}) \ = \ \texttt{f}(\,\texttt{p}_1, \ldots \texttt{p}_n)\,\{\ \texttt{e}\ \} \\ \texttt{M},\ \sigma[\texttt{p}_1 \mapsto v_1, \ldots.\texttt{p}_n \mapsto v_n], \texttt{e} \hookrightarrow_W\ v\end{array}}{\texttt{M},\ \sigma,\ \texttt{e}_0.\texttt{f}(\,\texttt{e}_1, \ldots.\texttt{e}_n) \hookrightarrow v}$$

$$\textsf{Field\_Ghost\_Val2}\quad \frac{\texttt{M},\ \sigma, \texttt{e.f}(\,) \hookrightarrow v}{\texttt{M},\ \sigma, \texttt{e.f} \hookrightarrow v}$$

$$\textsf{If\_True\_Val}\quad \frac{\begin{array}{l}\texttt{M},\ \sigma, \texttt{e} \hookrightarrow \texttt{true} \\ \texttt{M},\ \sigma, \texttt{e}_1 \hookrightarrow v\end{array}}{\texttt{M},\ \sigma, \mathsf{if}\ \texttt{e}\ \mathsf{then}\ \texttt{e}_1\ \mathsf{else}\ \texttt{e}_2 \hookrightarrow v} \qquad \textsf{If\_False\_Val}\quad \frac{\begin{array}{l}\texttt{M},\ \sigma, \texttt{e} \hookrightarrow \texttt{false} \\ \texttt{M},\ \sigma, \texttt{e}_2 \hookrightarrow v\end{array}}{\texttt{M},\ \sigma, \mathsf{if}\ \texttt{e}\ \mathsf{then}\ \texttt{e}_1\ \mathsf{else}\ \texttt{e}_2 \hookrightarrow v}$$

$$\textsf{Equals\_True\_Val}\quad \frac{\begin{array}{l}\texttt{M},\ \sigma, \texttt{e}_1 \hookrightarrow v \\ \texttt{M},\ \sigma, \texttt{e}_2 \hookrightarrow v\end{array}}{\texttt{M},\ \sigma, \texttt{e}_1 = \texttt{e}_2 \hookrightarrow \texttt{true}} \qquad \textsf{Equals\_False\_Val}\quad \frac{\begin{array}{l}\texttt{M},\ \sigma, \texttt{e}_1 \hookrightarrow v \\ \texttt{M},\ \sigma, \texttt{e}_2 \hookrightarrow v' \qquad v \neq v'\end{array}}{\texttt{M},\ \sigma, \texttt{e}_1 = \texttt{e}_2 \hookrightarrow \texttt{false}}$$

Fig. 5. Value of Expressions

Definition 4 (Interpretations for simple expressions). *For a runtime configuration, σ, variables* $\texttt{x}$ *or* $\texttt{S}$*, we define its interpretation as follows:*

- $\lfloor \texttt{x} \rfloor_\sigma \triangleq \phi(\texttt{x})$ *if* σ=$(\phi \cdot \_, \_)$
- $\lfloor \texttt{S} \rfloor_\sigma \triangleq \phi(\texttt{S})$ *if* σ=$(\phi \cdot \_, \_)$
- $\lfloor \texttt{x.f} \rfloor_\sigma \triangleq \chi(\lfloor \texttt{x} \rfloor_\sigma, \texttt{f})$ *if* σ=$(\_, \chi)$

Definition 5 (Basic Assertions). *For modules* $\texttt{M}, \texttt{M}'$*, configuration* σ*, we define*:

- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \texttt{e}$ *if* $\texttt{M},\ \sigma, \texttt{e} \hookrightarrow \texttt{true}$
- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \texttt{e} = \texttt{e}'$ *if there exists a value* v *such that* $\texttt{M},\ \sigma, \texttt{e} \hookrightarrow v$ *and* $\texttt{M},\ \sigma, \texttt{e}' \hookrightarrow v$*.*
- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \texttt{e} : \texttt{ClassId}$ *if there exists an address* α *such that* $\texttt{M},\ \sigma, \texttt{e} \hookrightarrow \alpha$*, and* $Class(\alpha)_\sigma = \texttt{ClassId}$.
- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \texttt{e} \in \texttt{S}$ *if there exists a value* v *such that* $\texttt{M},\ \sigma, \texttt{e} \hookrightarrow v$*, and* $v \in \lfloor \texttt{S} \rfloor_\sigma$*.*

Satisfaction of assertions which contain expressions is predicated on termination of these expressions. Continuing our earlier example, $\texttt{M}_0 \fatsemi \texttt{M}', \sigma_0 \models \texttt{acyc.acyclic}$ holds for any $\texttt{M}'$, while $\texttt{M}_0 \fatsemi \texttt{M}', \sigma_0 \models \texttt{cyc.acyclic}$ does not hold, and $\texttt{M}_0 \fatsemi \texttt{M}', \sigma_0 \models \texttt{cyc.acyclic} = \texttt{false}$ does not hold either. In general, when $\texttt{M} \fatsemi \texttt{M}', \sigma \models \texttt{e}$ holds, then $\texttt{M} \fatsemi \texttt{M}', \sigma \models \texttt{e} = \texttt{true}$ holds too. But when $\texttt{M} \fatsemi \texttt{M}', \sigma \models \texttt{e}$ does not hold, this does *not* imply that $\texttt{M} \fatsemi \texttt{M}', \sigma \models \texttt{e} = \texttt{false}$ holds. Finally, an assertion of the form $\texttt{e}_0 = \texttt{e}_0$ does not always hold; for example, $\texttt{M}_0 \fatsemi \texttt{M}', \sigma_0 \models \texttt{cyc.last} = \texttt{cyc.last}$ does not hold.

We now define satisfaction of assertions which involve logical connectives and existential or universal quantifiers, in the standard way:

Definition 6 (Assertions with logical connectives and quantifiers). *For modules* $\texttt{M}, \texttt{M}'$*, assertions* A, A'*, variables* $\texttt{x}, \texttt{y}, \texttt{S}$*, and configuration* σ*, we define*:

- $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models \forall\mathtt{S}:\mathtt{SET}.A$ *if* $\mathtt{M}\fatsemi\mathtt{M'},\sigma[\mathtt{Q}\mapsto R] \models A[\mathtt{S}/\mathtt{Q}]$ *for all sets of addresses* $R\subseteq dom(\sigma)$*, and all* $\mathtt{Q}$ *free in* σ *and* A*.*
- $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models \exists\mathtt{S}:\mathtt{SET}.A$ *if* $\mathtt{M}\fatsemi\mathtt{M'},\sigma[\mathtt{Q}\mapsto R] \models A[\mathtt{S}/\mathtt{Q}]$ *for some set of addresses* $R\subseteq dom(\sigma)$*, and* $\mathtt{Q}$ *free in* σ *and* A*.*
- $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models \forall\mathtt{x}.A$ *if* $\sigma[\mathtt{z}\mapsto\alpha] \models A[\mathtt{x}/\mathtt{z}]$ *for all* $\alpha\in dom(\sigma)$*, and some* $\mathtt{z}$ *free in* σ *and* A*.*
- $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models \exists\mathtt{x}.A$ *if* $\mathtt{M}\fatsemi\mathtt{M'},\sigma[\mathtt{z}\mapsto\alpha] \models A[\mathtt{x}/\mathtt{z}]$ *for some* $\alpha\in dom(\sigma)$*, and* $\mathtt{z}$ *free in* σ *and* A*.*
- $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models A\longrightarrow A'$ *if* $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models A$ *implies* $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models A'$
- $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models A\wedge A'$ *if* $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models A$ *and* $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models A'$*.*
- $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models A\vee A'$ *if* $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models A$ *or* $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models A'$*.*
- $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models \neg A$ *if* $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models A$ *does not hold.*

Satisfaction is not preserved with growing configurations; for example, the assertion $\forall\mathtt{x}.[\ \mathtt{x}:\mathtt{Account}\longrightarrow\mathtt{x.balance}>100\ ]$ may hold in a smaller configuration, but not hold in an extended configuration. Nor is it preserved with configurations getting smaller; consider *e.g.* $\exists\mathtt{x}.[\ \mathtt{x}:\mathtt{Account}\wedge\mathtt{x.balance}>100\ ]$.
Again, with our earlier example, $\mathtt{M_0}\fatsemi\mathtt{M'},\sigma_0 \models \neg(\mathtt{cyc.acyclic}=\mathtt{true})$ and $\mathtt{M_0}\fatsemi\mathtt{M'},\sigma_0 \models \neg(\mathtt{cyc.acyclic}=\mathtt{false})$, and also $\mathtt{M_0}\fatsemi\mathtt{M'},\sigma_0 \models \neg(\mathtt{cyc.last}=\mathtt{cyc.last})$ hold.

5.5 Satisfaction of Assertions - Access, Control, Space, Viewpoint

Permission expresses that an object has the potential to call methods on another object, and to do so directly, without help from any intermediary object. This is the case when the two objects are aliases, or the first object has a field pointing to the second object, or the first object is the receiver of the currently executing method and the second object is one of the arguments or a local variable. Interpretations of variables and paths, $\lfloor ... \rfloor_\sigma$, are defined in the usual way (appendix Def. 19).

Definition 7 (Permission). *For any modules* $\mathtt{M},\mathtt{M'}$*, variables* $\mathtt{x}$ *and* $\mathtt{y}$*, we define*

- $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models \langle\, \mathtt{x}\ \text{access}\ \mathtt{y}\,\rangle$ *if* $\lfloor\mathtt{x}\rfloor_\sigma$ *and* $\lfloor\mathtt{y}\rfloor_\sigma$ *are defined, and*
  - $\lfloor\mathtt{x}\rfloor_\sigma=\lfloor\mathtt{y}\rfloor_\sigma$*, or*
  - $\lfloor\mathtt{x.f}\rfloor_\sigma=\lfloor\mathtt{y}\rfloor_\sigma$*, for some field* $\mathtt{f}$*, or*
  - $\lfloor\mathtt{x}\rfloor_\sigma=\lfloor\mathtt{this}\rfloor_\sigma$ *and* $\lfloor\mathtt{y}\rfloor_\sigma=\lfloor\mathtt{z}\rfloor_\sigma$*, for some variable* $\mathtt{z}$ *and* $\mathtt{z}$ *appears in* $\sigma.\mathtt{contn}$*.*

In the last disjunct, where $\mathtt{z}$ is a parameter or local variable, we ask that $\mathtt{z}$ appears in the code being executed ($\sigma.\mathtt{contn}$). This requirement ensures that variables which were introduced into the variable map in order to give meaning to existentially quantified assertions, are not considered.

Control expresses which object is the process of making a function call on another object and with what arguments. The relevant information is stored in the continuation ($\mathtt{cont}$) on the top frame.

Definition 8 (Control). *For any modules* $\mathtt{M},\mathtt{M'}$*, variables* $\mathtt{x}$ *,* $\mathtt{y}$*,* $\mathtt{z}_1,...\mathtt{z}_n$*, we define*:

- $\mathtt{M}\fatsemi\mathtt{M'},\sigma \models \langle\, \mathtt{x}\ \text{calls}\ \mathtt{y.m(}\ \mathtt{z}_1,...\mathtt{z}_n\mathtt{)}\ \rangle$ *if* $\lfloor\mathtt{x}\rfloor_\sigma$*,* $\lfloor\mathtt{y}\rfloor_\sigma$*,* $\lfloor\mathtt{z}_1\rfloor_\sigma$*, ...* $\lfloor\mathtt{z}_n\rfloor_\sigma$ *are defined, and*
  - $\lfloor\mathtt{this}\rfloor_\sigma=\lfloor\mathtt{x}\rfloor_\sigma$*, and*
  - $\sigma.\mathtt{contn}=\mathtt{u.m}(\mathtt{v}_1,..\mathtt{v}_n);\_$*, for some* $\mathtt{u},\mathtt{v}_1,...\ \mathtt{v}_n$*, and*
  - $\lfloor\mathtt{y}\rfloor_\sigma=\lfloor\mathtt{u}\rfloor_\sigma$*, and* $\lfloor\mathtt{z}_i\rfloor_\sigma=\lfloor\mathtt{v}_i\rfloor_\sigma$*, for all i.*

Thus, $\langle\, \mathtt{x}\ \text{calls}\ \mathtt{y.m(}\ \mathtt{z}_1,...\mathtt{z}_n\mathtt{)}\ \rangle$ expresses the call $\mathtt{y.m}(\mathtt{z}_1,...\mathtt{z}_n)$ will be executed next, and that the caller is $\mathtt{x}$.

Viewpoint is about whether an object is viewed as belonging to the internal mode; this is determined by the class of the object.

Definition 9 (Viewpoint). *For any modules* $\mathtt{M},\mathtt{M'}$*, and variable*$\mathtt{x}$*, we define*

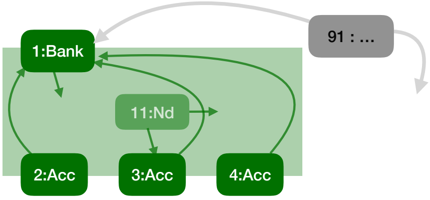
- $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models \text{external}\langle\, \mathtt{x} \,\rangle$ *if* $\lfloor \mathtt{x} \rfloor_\sigma$ *is defined and* $Class(\lfloor \mathtt{x} \rfloor_\sigma)_\sigma \notin dom(\mathtt{M})$
- $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models \text{internal}\langle\, \mathtt{x} \,\rangle$ *if* $\lfloor \mathtt{x} \rfloor_\sigma$ *is defined and* $Class(\lfloor \mathtt{x} \rfloor_\sigma)_\sigma \in dom(\mathtt{M})$

Space is about asserting that some property A holds in a configuration whose objects are restricted to those from a given set $\mathtt{S}$. This way we can express that the objects from the set $\mathtt{S}$ have authority over the assertion A. In order to define validity of $\langle\, A \text{ in } \mathtt{S} \,\rangle$ in a configuration σ, we first define a restriction operation, $\sigma\downarrow_{\mathtt{S}}$ which restricts the objects from σ to only those from $\mathtt{S}$.

Definition 10 (Restriction of Runtime Configurations). *The restriction operator* $\downarrow$ *applied to a runtime configuration* σ *and a variable* $\mathtt{S}$ *is defined as follows:*

- $\sigma\downarrow_{\mathtt{S}} \triangleq (\psi, \chi')$, *if* $\sigma = (\psi, \chi)$, $dom(\chi') = \lfloor \mathtt{S} \rfloor_\sigma$, *and* $\forall \alpha \in dom(\chi').\chi(\alpha) = \chi'(\alpha)$.

For example, if we take σ_2 from Fig. 3 in Section 2, and restrict it with some set $\mathtt{S}_4$ such that $\lfloor \mathtt{S}_4 \rfloor_{\sigma_2} = \{91, 1, 2, 3, 4, 11\}$, then the restriction $\sigma_2\downarrow_{\mathtt{S}_4}$ will look as on the right.



Note in the diagram above the dangling pointers at objects 1, 11, and 91 - reminiscent of the separation of heaps into disjoint subheaps, as provided by the $*$ operator in separation logic [53]. The difference is that in separation logic, the separation is provided through the assertions, where $A * A'$ holds in any heap which can be split into disjoint χ and χ' where χ satisfies A and χ' satisfies A'. That is, in $A * A'$ the split of the heap is determined by the assertions A and A' and there is an implicit requirement of disjointness, while in $\sigma\downarrow_{\mathtt{S}}$ the split is determined by $\mathtt{S}$, and no disjointness is required.

We now define the semantics of $\langle\, A \text{ in } \mathtt{S} \,\rangle$.

Definition 11 (Space). *For any modules* $\mathtt{M}, \mathtt{M}'$, *assertions* A *and variable* $\mathtt{S}$, *we define*:

- $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models \langle\, A \text{ in } \mathtt{S} \,\rangle$ *if* $\mathtt{M} \fatsemi \mathtt{M}', \sigma\downarrow_{\mathtt{S}} \models A$.

The set $\mathtt{S}$ in the assertion $\langle\, A \text{ in } \mathtt{S} \,\rangle$ is related to framing from implicit dynamic frames [57]: in an implicit dynamic frames assertion **acc** $\mathtt{x.f} * A$, the frame $\mathtt{x.f}$ prescribes which locations may be used to determine validity of A. The difference is that frames are sets of locations (pairs of address and field), while our $\mathtt{S}$-es are sets of addresses. More importantly, implicit dynamic frames assertions whose frames are not large enough are badly formed, while in our work, such assertions are allowed and may hold or not, *e.g.* $\mathtt{M}_{BA2} \fatsemi \mathtt{M}', \sigma \models \neg \langle\, (\exists n.\mathtt{a}_2.\mathtt{balance} = n) \text{ in } \mathtt{S}_4 \,\rangle$.

5.6 Satisfaction of Assertions - Time

To deal with time, we are faced with four challenges: a) validity of assertions in the future or the past needs to be judged in the future configuration, but using the bindings from the current one, b) the current configuration needs to store the code being executed, so as to be able to calculate future configurations, c) when considering the future, we do not want to observe configurations which go beyond the frame currently at the top of the stack, d) there is no "undo" operator to deterministically enumerate all the previous configurations.

Consider challenge a) in some more detail: the assertion $\text{will}\langle\, \mathtt{x.f} = 3 \,\rangle$ is satisfied in the *current* configuration σ_1, if in some *future* configuration σ_2, the field $\mathtt{f}$ of the object that is pointed at by $\mathtt{x}$ in the *current* configuration (σ_1) has the value 3, that is, if $\lfloor \lfloor \mathtt{x} \rfloor_{\sigma_1}.f \rfloor_{\sigma_2} = 3$, even if in that future configuration $\texttt{x}$ denotes a different object (i.e. if $\lfloor \texttt{x} \rfloor_{\sigma_1} \neq \lfloor \texttt{x} \rfloor_{\sigma_2}$). To address this, we define an auxiliary concept: the operator $\triangleleft$, where $\sigma_1 \triangleleft \sigma_2$ adapts the second configuration to the top frame's view of the former: it returns a new configuration whose stack comes from σ_2 but is augmented with the view from the top frame from σ_1 and where the continuation has been consistently renamed. This allows us to interpret expressions in σ_2 but with the variables bound according to σ_1; *e.g.* we can obtain that value of $\texttt{x}$ in configuration σ_2 even if $\texttt{x}$ was out of scope in σ_2.

Definition 12 (Adaptation). *For runtime configurations σ_1, σ_2.:*

- $\sigma_1 \triangleleft \sigma_2 \triangleq (\phi_3 \cdot \psi_2, \chi_2)$ *if*
  - $\phi_3 = (\,\texttt{contn}_2[\texttt{zs}_2/\texttt{zs}'], \beta_2[\texttt{zs}' \mapsto \beta_2(\texttt{zs}_2)][\texttt{zs}_1 \mapsto \beta_1(\texttt{zs}_1)]\,)$, *where*
  - $\sigma_1 = (\phi_1 \cdot \_, \_)$, $\sigma_2 = (\phi_2 \cdot \psi_2, \chi_2)$, $\phi_1 = (\_, \beta_1)$, $\phi_2 = (\texttt{contn}_2, \beta_2)$, *and*
  - $\texttt{zs}_1 = dom(\beta_1)$, $\texttt{zs}_2 = dom(\beta_2)$, *and*
  - $\texttt{zs}'$ *is a set of variables with the same cardinality as* $\texttt{zs}_2$*, and all variables in* $\texttt{zs}'$ *are fresh in* β_1 *and in* β_2*.*

That is, in the new frame ϕ_2 from above, we keep the same continuation as from σ_2 but rename all variables with fresh names $\texttt{zs}'$, and combine the variable map β_1 from σ_1 with the variable map β_2 from σ_2 while avoiding names clashes through the renaming $[\texttt{zs}' \mapsto \beta_2(\texttt{zs}_2)]$. The consistent renaming of the continuation allows the correct modelling of execution, as needed for the semantics of nested time assertions, as *e.g.* in $\mathsf{will}\langle\, \texttt{x.f} = 3 \wedge \mathsf{will}\langle\, \texttt{x.f} = 5 \,\rangle\,\rangle$.

Having addressed challenge a) we turn our attention to the remaining challenges: We address challenge b) by storing the remaining code to be executed in $\texttt{cntn}$ in each frame. We address challenge c) by only taking the top of the frame when considering future executions. Finally, we address challenge d) by considering only configurations which arise from initial configurations, and which lead to the current configuration.

Definition 13 (Time Assertions). *For any modules* $\texttt{M}$*,* $\texttt{M}'$*, and assertion A we define*

- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \mathsf{next}\langle\, A \,\rangle$ *if* $\exists \sigma'.[\ \texttt{M} \fatsemi \texttt{M}', \phi \leadsto \sigma' \ \wedge \texttt{M} \fatsemi \texttt{M}', \sigma \triangleleft \sigma' \models A\ ]$,
  and where ϕ is so that $\sigma = (\phi \cdot \_, \_)$.
- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \mathsf{will}\langle\, A \,\rangle$ *if* $\exists \sigma'.[\ \texttt{M} \fatsemi \texttt{M}', \phi \leadsto^* \sigma' \ \wedge \texttt{M} \fatsemi \texttt{M}', \sigma \triangleleft \sigma' \models A\ ]$,
  and where ϕ is so that $\sigma = (\phi \cdot \_, \_)$.
- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \mathsf{prev}\langle\, A \,\rangle$ *if* $\forall \sigma_1, \sigma_2.[\ \mathcal{I}nitial\langle \sigma_1 \rangle\ \wedge\ \texttt{M} \fatsemi \texttt{M}', \sigma_1 \leadsto^* \sigma_2$
  $\wedge\ \texttt{M} \fatsemi \texttt{M}', \sigma_2 \leadsto \sigma \quad \longrightarrow \quad \texttt{M} \fatsemi \texttt{M}', \sigma \triangleleft \sigma_2 \models A\ ]$
- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \mathsf{was}\langle\, A \,\rangle$ *if* $\forall \sigma_1.[\ \mathcal{I}nitial\langle \sigma_1 \rangle\ \wedge\ \texttt{M} \fatsemi \texttt{M}', \sigma_1 \leadsto^* \sigma \longrightarrow$
  $(\ \exists \sigma_2.\texttt{M} \fatsemi \texttt{M}', \sigma_1 \leadsto^* \sigma_2\ \wedge\ \texttt{M} \fatsemi \texttt{M}', \sigma_2 \leadsto^* \sigma\ \wedge\ \texttt{M} \fatsemi \texttt{M}', \sigma \triangleleft \sigma_2 \models A\ )]$

In general, $\mathsf{will}\langle\,\langle\, A\ \mathsf{in}\ \texttt{S} \,\rangle\,\rangle$ is different from $\langle\, \mathsf{will}\langle\, A \,\rangle\ \mathsf{in}\ \texttt{S} \,\rangle$. In the former assertion, $\texttt{S}$ must contain the objects involved in reaching the future configuration as well as the objects needed to then establish validity of A in that future configuration. In the latter assertion, $\texttt{S}$ need only contain the objects needed to establish A in that future configuration. For example, revisit Fig. 3, and take $\texttt{S}_1$ to consist of objects $\texttt{1}$, $\texttt{2}$, $\texttt{4}$, $\texttt{93}$, and $\texttt{94}$, and $\texttt{S}_2$ to consist of objects $\texttt{1}$, $\texttt{2}$, $\texttt{4}$. Assume that σ_5 is like σ_1, that the next call in σ_5 is a method on $\texttt{u}_{94}$, whose body obtains the address of $\texttt{a}_4$ (by making a call on $\texttt{93}$ to which it has access), and the address of $\texttt{a}_2$ (to which it has access), and then makes the call $\texttt{a}_2\texttt{.deposit}(\texttt{a}_4, 360)$. Assume also that $\texttt{a}_4$'s balance is $\texttt{380}$. Then

$$\texttt{M}_{BA1} \fatsemi ..., \sigma_5 \ \models\ \langle\, \mathsf{will}\langle\, \mathsf{changes}\langle\, \texttt{a}_2\texttt{.balance} \,\rangle\,\rangle\ \mathsf{in}\ \texttt{S}_1 \,\rangle$$
$$\texttt{M}_{BA1} \fatsemi ..., \sigma_5 \ \not\models\ \langle\, \mathsf{will}\langle\, \mathsf{changes}\langle\, \texttt{a}_2\texttt{.balance} \,\rangle\,\rangle\ \mathsf{in}\ \texttt{S}_2 \,\rangle$$
$$\texttt{M}_{BA1} \fatsemi ..., \sigma_5 \ \models\ \mathsf{will}\langle\,\langle\, \mathsf{changes}\langle\, \texttt{a}_2\texttt{.balance} \,\rangle\ \mathsf{in}\ \texttt{S}_2 \,\rangle\,\rangle$$

5.7 Properties of Assertions

We define equivalence of assertions in the usual way: assertions A and $A'$are equivalent if they are satisfied in the context of the same configurations and module pairs – *i.e.*

$$A \equiv A' \quad \text{if} \quad \forall \sigma. \forall \mathtt{M}, \mathtt{M}'. \; [\;\; \mathtt{M} \,⨾\, \mathtt{M}', \sigma \models A \;\text{ if and only if }\; \mathtt{M} \,⨾\, \mathtt{M}', \sigma \models A' \;\;].$$

We can then prove that the usual equivalences hold, *e.g.* $A \vee A' \;\equiv\; A' \vee A$, and $\neg(\exists \mathtt{x}.A) \;\equiv\; \forall \mathtt{x}.(\neg A)$. Our assertions are classical, *e.g.* $A \wedge \neg A \;\equiv\; \mathtt{false}$, and $\mathtt{M} \,⨾\, \mathtt{M}', \sigma \models A$ and $\mathtt{M} \,⨾\, \mathtt{M}', \sigma \models A \longrightarrow A'$ implies $\mathtt{M} \,⨾\, \mathtt{M}', \sigma \models A'$. This desirable property comes at the loss of some expected equivalences, *e.g.*, in general, $\mathtt{e} = \mathtt{false}$ and $\neg \mathtt{e}$ are not equivalent. More in Appendix B.

5.8 Modules satisfying assertions

Finally, we define satisfaction of assertions by modules: a module $\mathtt{M}$ satisfies an assertion A if for all other potential modules $\mathtt{M}'$, in all configurations arising from executions of $\mathtt{M} \,⨾\, \mathtt{M}'$, the assertion A holds.

Definition 14. *For any module* $\mathtt{M}$*, and assertion A, we define:*

- $\mathtt{M} \models A \quad \textit{if} \quad \forall \mathtt{M}'. \forall \sigma \in \mathcal{A}\textit{rising}(\mathtt{M} \,⨾\, \mathtt{M}'). \; \mathtt{M} \,⨾\, \mathtt{M}', \sigma \models A$

6 DISCUSSION

Specification Language. The key to writing and using holistic specifications is to capture the implicit assumptions underlying the design of the system we are specifying. The design principle animating the design of *Chainmail* is to make writing holistic specifications as straightforward as possible.

This is why we support bi-directional temporal operators ("will", "was" etc) rather than just one temporal direction. Some assertions are easier to express looking forwards (if a vending machine takes a coin it will eventually dispense a chocolate bar) while other assertions as easier to express looking backwards (if a vending machine dispenses a chocolate bar, it has already accepted the coin to pay for it).

Similarly, in an imperative setting, the question of whether a change can occur at all is at least as important as the precise details of the change. This is why we support the "changes" operator in *Chainmail*; to enable specifications to capture that notion directly, rather than e.g. expressing explicit differences in values between pre- and post- states.

For modelling the open world, the interactions between the external and internal modules are essential. We included predicates in *Chainmail* to be able to state explicitly which side of the boundary between modules an object is and what can cross that boundary.

The issues we have had to deal with are those related to space in various ways: "access", "in", and "external". One object referring to another appears simple enough on the surface, but causes significant problems in an open world setting. In *Chainmail*, these problems appear with universal quantification on the source of a reference: "in" enables us to bound these quantifications.

Underlying Language. For our underlying language, we have chosen an object-oriented, class based language, We have chosen to use a dynamically typed language because many of the problems we hope to address are written in these languages: web apps and mashups in Javascript; backends in Ruby or PHP. We expect that supporting types would make the problem easier, not harder, but at the cost of significantly increasing the complexity of the trusted computing base that we assume will run our programs. In an open world, without some level of assurance (e.g. proof-carrying code) about the trustworthiness of type information: unfounded assumptions about types can give rise to new vulnerabilities that attackers can exploit [20].

Finally, we don't address inheritance. As a specification language, individual *Chainmail* assertions can be combined or reused without any inheritance mechanism: the semantics are simply that all the *Chainmail* assertions are expected to hold at all the points of execution that they constrain. $\mathcal{L}_{oo}$ does not contain inheritance simply because it is not necessary to demonstrate specifications of robustness: whether an $\mathcal{L}_{oo}$ class is defined in one place, or whether it is split into many multiply-inherited superclasses, traits, default methods in interfaces or protocols, etc. is irrelevant, provided we can model the resulting (flattened) behaviour of such a composition as a single logical $\mathcal{L}_{oo}$ class.

7 PROBLEM–DRIVEN DESIGN

The design of *Chainmail* was guided by the study of a sequence of exemplars taken from the object-capability literature and the smart contracts world:

(1) **Bank** [49] - Bank and accounts as described in Section 2, with two different implementations given in appendix C
(2) **ERC20** [61] - Ethereum-based token contract
(3) **DAO** [12, 15] - Ethereum contract for Decentralised Autonomous Organisation
(4) **DOM** [19, 59] - Restricting access to browser Domain Object Model

7.1 Authorising ERC20

ERC20 [61] is a widely used token standard which describes the basic functionality expected by any Ethereum-based token contract. It issues and keeps track of participants' tokens, and supports the transfer of tokens between participants. Transfer of tokens can take place only provided that there were sufficient tokens in the owner's account, and that the transfer was instigated by the owner, or by somebody authorized by the owner.

We specify this in *Chainmail* as follows: A decrease in a participant's `balance` can only be caused by a transfer instigated by the account holder themselves
(*i.e.* $\langle\, \texttt{p}\ \textsf{calls}\ \ldots.\texttt{transfer}(\ \ldots)\ \rangle$), or by an authorized transfer instigated by another participant $\texttt{p}''$ (*i.e.* $\langle\, \texttt{p}''\ \textsf{calls}\ \ldots\texttt{transferFrom}(\ ..)\ \rangle$) who has authority for more than the tokens spent (*i.e.* $\texttt{e.allowed}(\texttt{p},\texttt{p}'') \geq \texttt{m}$)

$$\begin{array}{l}
\forall \texttt{e} : \texttt{ERC20}.\forall \texttt{p} : \texttt{Object}.\forall \texttt{m}, \texttt{m}' : \texttt{Nat}.\\
\quad [\ \ \texttt{e.balance(p)} = \texttt{m} + \texttt{m'}\ \wedge\ \textsf{next}\langle\, \texttt{e.balance(p)} = \texttt{m}' \,\rangle\\
\qquad \longrightarrow\\
\qquad \exists \texttt{p}', \texttt{p}'' : \texttt{Object}.\\
\qquad [\ \ \langle\, \texttt{p}\ \textsf{calls}\ \texttt{e.transfer}(\ \texttt{p}', \texttt{m})\ \rangle\ \ \vee\\
\qquad\quad \texttt{e.allowed}(\texttt{p}, \texttt{p}'') \geq \texttt{m}\ \wedge\ \langle\, \texttt{p}''\ \textsf{calls}\ \texttt{e.transferFrom}(\ \texttt{p}', \texttt{m})\ \rangle\ \ ]\\
\quad]
\end{array}$$

That is to say: if next configuration witnesses a decrease of `p`'s balance by `m`, then the current configuration was a call of `transfer` instigated by `p`, or a call of `transferFrom` instigated by somebody authorized by `p`. The term $\texttt{e.allowed}(\texttt{p}, \texttt{p}'')$, means that the ERC20 variable `e` holds a field called `allowed` which maps pairs of participants to numbers; such mappings are supported in Solidity[16].

We now define what it means for $\texttt{p}'$ to be authorized to spend up to `m` tokens on `p`'s behalf: At some point in the past, `p` gave authority to $\texttt{p}'$ to spend `m` plus the sum of tokens spent so far by $\texttt{p}'$ on the behalf of `p`.

$$\begin{array}{l}
\forall \texttt{e} : \texttt{ERC20}.\forall \texttt{p}, \texttt{p'} : \texttt{Object}.\forall \texttt{m} : \texttt{Nat}.\\
\quad [\ \ \texttt{e.allowed}(\texttt{p}, \texttt{p}') = \texttt{m}
\end{array}$$

$$
\begin{array}{l}
\quad\longrightarrow \\
\quad\mathcal{P}rev\langle\ \langle\, \texttt{p}\ \text{calls}\ \texttt{e.approve}(\,\texttt{p}',\texttt{m})\ \rangle \\
\qquad\qquad \vee \\
\qquad\qquad \texttt{e.allowed}(\texttt{p},\texttt{p}') = \texttt{m}\ \wedge \\
\qquad\qquad\ \ \neg(\,\langle\, \texttt{p}'\ \text{calls}\ \texttt{e.transferFrom}(\,\texttt{p},\_)\ \rangle\ \vee\ \langle\, \texttt{p}\ \text{calls}\ \texttt{e.approve}(\,\texttt{p},\_)\ \rangle\,) \\
\qquad\qquad \vee \\
\qquad\qquad \exists \texttt{p}'' : \texttt{Object}.\exists \texttt{m'} : \texttt{Nat}. \\
\qquad\qquad [\ \texttt{e.allowed}(\texttt{p},\texttt{p}') = \texttt{m} + \texttt{m}'\ \wedge\ \langle\, \texttt{p}'\ \text{calls}\ \texttt{e.transferFrom}(\,\texttt{p}'',\texttt{m}')\ \rangle] \\
\qquad\ \ \rangle \\
\ ]
\end{array}
$$

In more detail p′ is allowed to spend up to m tokens on their behalf of p, if in the previous step either a) p made the call approve on e with arguments p′ and m, or b) p′ was allowed to spend up to m tokens for p and did not transfer any of p's tokens, nor did p issue a fresh authorization, or c) p was authorized for m + m′ and spent m′.

Thus, the holistic specification gives to account holders an "authorization-guarantee": their balance cannot decrease unless they themselves, or somebody they had authorized, instigates a transfer of tokens. Moreover, authorization is *not* transitive: only the account holder can authorise some other party to transfer funds from their account: authorisation to spend from an account does not confer the ability to authorise yet more others to spend also.

With traditional specifications, to obtain the "authorization-guarantee", one would need to inspect the pre- and post- conditions of *all* the functions in the contract, and determine which of the functions decrease balances, and which of the functions affect authorizations. In the case of the ERC20, one would have to inspect all eight such specifications (given in appendix 7.1.1), where only five are relevant to the question at hand. In the general case, *e.g.* the DAO, the number of functions which are unrelated to the question at hand can be very large.

More importantly, with traditional specifications, nothing stops the next release of the contract to add, *e.g.*, a method which allows participants to share their authority, and thus violate the "authorization-guarantee", or even a super-user from skimming 0.1% from each of the accounts.

7.1.1 *ERC20, the traditional specification.* We compare the holistic and the traditional specification of ERC20

Matthew Ross: *The following can be an appendix, but I've provisionally moved it up to try and avoid using appendices as asked to*

As we said earlier, the holistic specification gives to account holders an "authorisation-guarantee": their balance cannot decrease unless they themselves, or somebody they had authorised, instigates a transfer of tokens. Moreover, authorisation is *not* transitive: only the account holder can authorise some other party to transfer funds from their account: authorisation to spend from an account does not confer the ability to authorise yet more others to spend also.

With traditional specifications, to obtain the "authorisation-guarantee", one would need to inspect the pre- and post-conditions of *all* the functions in the contract, and determine which of the functions decrease balances, and which of the functions affect authorisations. In Figure 6 we outline a traditional specification for the ERC20. We give two specifications for transfer, another two for tranferFrom, and one for all the remaining functions. The first specification says, *e.g.*, that if p has sufficient tokens, and it calls transfer, then the transfer will take place. The second specification says that if p has insufficient tokens, then the transfer will not take place (we assume that in this specification language, any entities not mentioned in the pre- or post-condition are not affected).

Similarly, we would have to give another two specifications to define the behaviour of if p" is authorised and executes transferFrom, then the balance decreases. But they are *implicit* about the

$$\begin{gathered}
\texttt{e} : \texttt{ERC20} \ \wedge\ \texttt{p},\texttt{p"} : \texttt{Object} \wedge\ \texttt{m},\texttt{m}',\texttt{m}'' : \texttt{Nat} \wedge \\
\texttt{e.balance(p)} = \texttt{m} + \texttt{m}' \ \wedge\ \texttt{e.balance(p")} = \texttt{m}'' \ \wedge\ \texttt{this} = \texttt{p} \\
\{\ \texttt{e.transfer(p",m')}\ \} \\
\texttt{e.balance(p)} = \texttt{m} \ \wedge\ \texttt{e.balance(p")} = \texttt{m}'' + \texttt{m}'
\end{gathered}$$

$$\begin{gathered}
\texttt{e} : \texttt{ERC20} \ \wedge\ \texttt{p},\texttt{p'} : \texttt{Object} \wedge\ \texttt{m},\texttt{m}',\texttt{m}'' : \texttt{Nat} \ \wedge\ \texttt{e.balance(p)} = \texttt{m} \ \wedge \texttt{m} < \texttt{m}' \\
\{\ \texttt{e.transfer(p',m')}\ \} \\
\texttt{e.balance(p)} = \texttt{m}
\end{gathered}$$

$$\begin{gathered}
\texttt{e} : \texttt{ERC20} \ \wedge\ \texttt{p},\texttt{p'},\texttt{p}'' : \texttt{Object} \wedge\ \texttt{m},\texttt{m}',\texttt{m}'',\texttt{m}''' : \texttt{Nat} \wedge \\
\texttt{e.balance(p)} = \texttt{m} + \texttt{m}' \ \wedge\ \texttt{e.allowed(p,p')} = \texttt{m}''' + \texttt{m}' \wedge \\
\texttt{e.balance(p")} = \texttt{m}'' \ \wedge\ \texttt{this} = \texttt{p'} \\
\{\ \texttt{e.transferFrom(p',p",m')}\ \} \\
\texttt{e.balance(p)} = \texttt{m} \ \wedge\ \texttt{e.balance(p")} = \texttt{m}'' + \texttt{m}' \ \wedge\ \texttt{e.allowed(p,p')} = \texttt{m}'''
\end{gathered}$$

$$\begin{gathered}
\texttt{e} : \texttt{ERC20} \ \wedge\ \texttt{p},\texttt{p'} : \texttt{Object} \wedge\ \texttt{m},\texttt{m}',\texttt{m}'' : \texttt{Nat} \ \wedge\ \texttt{this} = \texttt{p}' \wedge \\
(\ \texttt{e.balance(p)} = \texttt{m} \wedge \texttt{m} < \texttt{m}'' \ \vee\ \texttt{e.allowed(p,p')} = \texttt{m'} \wedge \texttt{m'} < \texttt{m}''\ ) \\
\{\ \texttt{e.transferFrom(p,p",m")}\ \} \\
\texttt{e.balance(p)} = \texttt{m} \wedge \texttt{e.allowed(p,p')} = \texttt{m'}
\end{gathered}$$

$$\begin{gathered}
\texttt{e} : \texttt{ERC20} \ \wedge\ \texttt{p},\texttt{p'} : \texttt{Object} \wedge\ \texttt{m} : \texttt{Nat} \ \wedge\ \texttt{this} = \texttt{p} \\
\{\ \texttt{e.approve(p',m')}\ \} \\
\texttt{e.allowed(p,p')} = \texttt{m}
\end{gathered}$$

$$\begin{gathered}
\texttt{e} : \texttt{ERC20} \ \wedge\ \texttt{m} : \texttt{Nat} \ \wedge\ \texttt{p.balance} = \texttt{m} \\
\{\ \texttt{k} = \texttt{e.balanceOf(p)}\ \} \\
\texttt{k} = \texttt{m} \ \wedge\ \texttt{e.balanceOf(p)} = \texttt{m}
\end{gathered}$$

$$\begin{gathered}
\texttt{e} : \texttt{ERC20} \ \wedge\ \texttt{m} : \texttt{Nat} \ \wedge\ \texttt{e.allowed(p,p')} = \texttt{m} \\
\{\ \texttt{k} = \texttt{e.allowance(p,p')}\ \} \\
\texttt{k} = \texttt{m} \ \wedge\ \texttt{e.allowed(p,p')} = \texttt{m}
\end{gathered}$$

$$\begin{gathered}
\texttt{e} : \texttt{ERC20} \ \wedge\ \texttt{m} : \texttt{Nat} \ \wedge\ \textstyle\sum_{\texttt{p} \in dom(\texttt{e.balance})} \texttt{e.balance}(\texttt{p}) = \texttt{m} \\
\{\ \texttt{k} = \texttt{e.totalSupply()}\ \} \\
\texttt{k} = \texttt{m}
\end{gathered}$$

Fig. 6. Classical specification for the ERC20

overall behaviour and the *necessary* conditions, e.g., what are all the possible actions that can cause a decrease of balance?

7.2 Defending the DAO

The DAO (Decentralised Autonomous Organisation) [12] is a famous Ethereum contract which aims to support collective management of funds, and to place power directly in the hands of the owners of the DAO rather than delegate it to directors. Unfortunately, the DAO was not robust: a re-entrancy bug exploited in June 2016 led to a loss of $50M, and a hard-fork in the chain [15]. With holistic specifications we can write a succinct requirement that a DAO contract should always be able to repay any owner's money. Any contract which satisfies such a holistic specification cannot demonstrate the DAO bug.

Our specification consists of three requirements. First, that the DAO always holds at least as much money as any owner's balance. To express this we use the field `balances` which is a mapping from participant's addresses to numbers. Such mapping-valued fields exist in Solidity, but they could also be taken to be ghost fields [11].

Matthew Ross: *Why is this needed if the second one holds, and why not all owners?*

$$\begin{aligned}
&\forall \mathtt{d} : \mathtt{DAO}.\forall \mathtt{p} : \mathtt{Any}.\forall \mathtt{m} : \mathtt{Nat}.\\
&[\ \mathtt{d.balances(p)} = \mathtt{m} \quad \longrightarrow \quad \mathtt{d.ether} \geq \mathtt{m}\ ]
\end{aligned}$$

Second, that when an owner asks to be repaid, she is sent all her money.

$$\begin{aligned}
&\forall \mathtt{d} : \mathtt{DAO}.\forall \mathtt{p} : \mathtt{Any}.\forall \mathtt{m} : \mathtt{Nat}.\\
&[\ \mathtt{d.balance(p)} = \mathtt{m} \ \wedge\ \langle\, \mathtt{p}\ \mathsf{calls}\ \mathtt{d.repay}(\ \_\ )\ \rangle\\
&\qquad\qquad\qquad\qquad\qquad \longrightarrow \quad \mathsf{will}\langle\, \langle\, \mathtt{d}\ \mathsf{calls}\ \mathtt{send.p}(\ \mathtt{m})\ \rangle\,\rangle\ ]
\end{aligned}$$

Third, that the balance of an owner is a function of its balance in the previous step, or the result of it joining the DAO, or asking to be repaid *etc.*.

$$\begin{aligned}
&\forall \mathtt{d} : \mathtt{DAO}.\forall \mathtt{p}.\forall : \mathtt{m} : \mathtt{Nat}.\\
&[\ \ \mathtt{d.Balance(p)} = \mathtt{m} \quad \longrightarrow \quad [\ \ \mathsf{prev}\langle\, \langle\, \mathtt{p}\ \mathsf{calls}\ \mathtt{d.repay}(\ \_\ )\ \rangle\,\rangle \wedge \mathtt{m} = 0 \quad \vee\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{prev}\langle\, \langle\, \mathtt{p}\ \mathsf{calls}\ \mathtt{d.join}(\ \mathtt{m})\ \rangle\,\rangle \quad \vee\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \ldots\ ]\\
&]
\end{aligned}$$

More cases are needed to reflect the financing and repayments of proposals, but they can be expressed with the concepts described so far.

The requirement that `d` holds at least `m` ether precludes the DAO bug, in the sense that any contract satisfying that spec cannot exhibit the bug: a contract which satisfies the spec is guaranteed to always have enough money to satisfy all `repay` requests. This guarantee holds, regardless of how many functions there are in the DAO. In contrast, to preclude the DAO bug with a classical spec, one would need to write a spec for each of the DAO functions (currently 19), a spec for each function of the auxiliary contracts used by the DAO, and then study their emergent behaviour.

These 19 DAO functions have several different concerns: who may vote for a proposal, who is eligible to submit a proposal, how long the consultation period is for deliberating a proposal, what is the quorum, how to chose curators, what is the value of a token, Of these groups of functions, only a handful affect the balance of a participant. Holistic specifications allow us to concentrate on aspect of DAO's behaviour across *all* its functions.

7.3 Attenuating the DOM

Attenuation is the ability to provide to third party objects *restricted* access to an object's functionality. This is usually achieved through the introduction of an intermediate object. While such intermediate objects are a common programming practice, the term was coined, and the practice was studied in detail in the object capabilities literature, e.g. [39].

The key structure underlying a web browser is the Domain Object Model (DOM), a recursive composite tree structure of objects that represent everything displayed in a browser window. Each window has a single DOM tree which includes both the page's main content and also third party content such as advertisements. To ensure third party content cannot affect a page's main content, specifications for attenuation for the DOM were proposed in *Devriese et al:* [19].

This example deals with a tree of DOM nodes: Access to a DOM node gives access to all its parent and children nodes, and the ability to modify the node's properties. However, as the top nodes of the tree usually contain privileged information, while the lower nodes contain less crucial third-party information, we want to be able to limit access given to third parties to only the lower part of the DOM tree. We do this through a `Wrapper`, which has a field `node` pointing to a `Node`, and a field `height` which restricts the range of `Node`s which may be modified through the use of the particular `Wrapper`. Namely, when you hold a `Wrapper` you can modify the `property` of all the descendants of the `height`-th ancestors of the `node` of that particular `Wrapper`.

In Figure 7 we show an example of the use of `Wrapper` objects attenuating the use of `Node`s The function `usingWrappers` takes as parameter an object of unknown provenance, here called `unknwn`. On lines 2-7 we create a tree consisting of nodes `n1`, `n2`, ... `n6`, depicted as blue circles on the right-hand-side of the Figure. On line 8 we create a wrapper of `n5` with height `1`. This means that the wrapper `w` may be used to modify `n3`, `n5` and `n6` (*i.e.* the objects in the green triangle), while it cannot be used to modify `n1`, `n2`, and `4` (*i.e.* the objects within the blue triangle). On line 8 we call a function named `untrusted` on the `unknown` object, and pass `w` as argument.

```
method usingWrappers(unknwn){
   n1=Node(null,"fixed");
   n2=Node(n1,"robust");
   n3=Node(n2,"const");
   n4=Node(n3,"volatile");
   n5=Node(n4,"variable");
   n6=Node(n5,"ethereal");
   w=Wrapper(n5,1);

   unknwn.untrusted(w);

   assert n2.property=="robust"
   ...
}
```

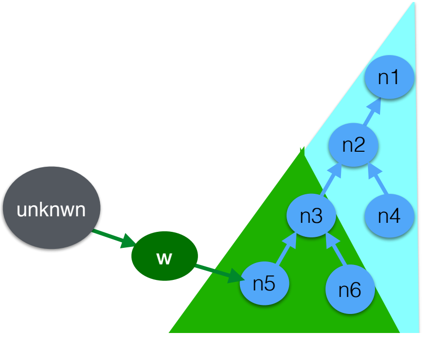


Fig. 7. `Wrapper`s protecting `Node`s

Even though we know nothing about the `unknown` object or its `untrusted` function, and even though the call gives to `unknown` access to `w`, which in turn has transitive access to all `Node`-s in the tree, we know that line 100 will not affect the `property` fields of the nodes `n1`, `n2`, and `n4`. Thus, the assertion on line 12 is guaranteed to succeed. The question is how do we specify `Wrapper`, so as to be able to make such an argument.

A specification of the class `Wrapper` in the traditional style, *e.g.* [31] consists of pairs of pre- and post- conditions for each of the functions of that class. Each such pair gives a *sufficient* condition for some effect to take place: for example the call `w.setProperty(i,prp)` where `i` is smaller than `w.height` is a sufficient condition to modify `property` of the `i`-th parent of `w.node`. But we do not know what other ways there may be to modify a node's `property`. In other words, we have not specified the *necessary conditions*. In our example:

> The *necessary* condition for the modification of `nd.property` for some `nd` of class `Node` is either access to some `Node` in the same tree, or access to a `w` of class `Wrapper` where the `w.height`-th parent of `w` is an ancestor of `nd`.

With such a specification we can prove that the assertion on line 12 will succeed. And, more importantly, we can ensure that all future updates of the `Wrapper` abstract data type will uphold the *protection* of the `Node` data. To give a flavour of *Chainmail*, we use it express the requirement from above:

$$
\begin{array}{l}
\forall \mathtt{S} : \mathtt{Set}.\forall \mathtt{nd} : \mathtt{Node}.\forall \mathtt{o} : \mathtt{Object}.\\
[\\
\ \langle\, \mathsf{will}\langle\, \mathsf{changes}\langle\, \mathtt{nd.property}\,\rangle\,\rangle \;\mathsf{in}\; \mathtt{S}\,\rangle\\
\quad \longrightarrow\\
\quad \exists \mathtt{o}.[\ \mathtt{o} \in \mathtt{S} \ \wedge\ \neg(\mathtt{o} : \mathtt{Node})\ \wedge\ \neg(\mathtt{o} : \mathtt{Wrapper})\ \wedge\\
\qquad\quad [\ \exists \mathtt{nd}' : \mathtt{Node}.\langle\, \mathtt{o}\ \mathsf{access}\ \mathtt{nd}'\,\rangle \quad \vee\\
\qquad\qquad \exists \mathtt{w} : \mathtt{Wrapper}.\exists k{:}\mathbb{N}.(\ \langle\, \mathtt{o}\ \mathsf{access}\ \mathtt{w}\,\rangle\ \wedge\ \mathtt{nd.parnt}^{k} = \mathtt{w.node.parnt}^{\mathtt{w.height}})\ \ ]\,]\\
]
\end{array}
$$

That is, if the value of `nd.property` is modified (changes$\langle\, _\,\rangle$) at some future point (will$\langle\, _\,\rangle$) and if reaching that future point involves no more objects than those from set `S` (*i.e.* $\langle\, \_ \ \mathsf{in}\ \mathtt{S}\,\rangle$), then at least one (`o`) of the objects in `S` is not a `Node` nor a `Wrapper`, and `o` has direct access to some node ($\langle\, \mathtt{o}\ \mathsf{access}\ \mathtt{nd}'\,\rangle$), or to some wrapper `w` and the `w.height`-th parent of `w` is an ancestor of `nd` (that is, $\mathtt{parnt}^{k} = \mathtt{w.node.parnt}^{\mathtt{w.height}}$). Note that our "access" is intransitive: $\langle\, x\ \mathsf{access}\ y\,\rangle$ holds if either `x` has a field pointing to `y`, or `x` is the receiver and `y` is one of the arguments in the executing method call.

8 MODEL

We have constructed a Coq model[3] [23] of the core of the Chainmail specification language, along with the underlying $\mathcal{L}_{oo}$ language. Our formalism is organised as follows:

(1) The $\mathcal{L}_{oo}$ Language: a class based, object oriented language with mutable references.
(2) Chainmail: The full assertion syntax and semantics defined in Definitions 1, 2, 7, 8, 9, 10, 11, 12, 13 and 14.
(3) $\mathcal{L}_{oo}$ Properties: Secondary properties of the loo language that aid in reasoning about its semantics.
(4) Chainmail Properties: The core properties defined on the semantics of Chainmail.

In the associated appendix (see Appendix D) we list and present the properties of Chainmail we have formalised in Coq. We have proven that Chainmail obeys much of the properties of classical logic. While we formalise most of the underlying semantics, we make several assumptions in our Coq formalism: (i) the law of the excluded middle, a property that is well known to be unprovable in constructive logics, and (ii) the equality of variable maps and heaps down to renaming. Coq formalisms often require fairly verbose definitions and proofs of properties involving variable substitution and renaming, and assuming equality down to renaming saves much effort.

More details of the formal foundations of *Chainmail*, and the model, are also in appendices [1].

9 RELATED WORK

Behavioural Specification Languages. Hatcliff et al. [26] provide an excellent survey of contemporary specification approaches. With a lineage back to Hoare logic [28], Meyer's Design by Contract [38] was the first popular attempt to bring verification techniques to object-oriented programs as a "whole cloth" language design in Eiffel. Several more recent specification languages are now making their way into practical and educational use, including JML [31], Spec♯ [4], Dafny [32] and Whiley [51]. Our approach builds upon these fundamentals, particularly Leino & Shulte's formulation of two-state invariants [33], and Summers and Drossopoulou's Considerate Reasoning [58]. In general, these approaches assume a closed system, where modules can be trusted to coöperate. In this paper we aim to work in an open system where modules' invariants must be protected irrespective of the behaviour of the rest of the system.

[3] A current model can be found at: https://github.com/sophiaIC/HolisticSpecifications

Defensive Consistency. In an open world, we cannot rely on the kindness of strangers: rather we have to ensure our code is correct regardless of whether it interacts with friends or foes. Attackers *"only have to be lucky once"* while secure systems *"have to be lucky always"* [5]. Miller [39, 40] defines the necessary approach as **defensive consistency**: *"An object is defensively consistent when it can defend its own invariants and provide correct service to its well behaved clients, despite arbitrary or malicious misbehaviour by its other clients."* Defensively consistent modules are particularly hard to design, to write, to understand, and to verify: but they make it much easier to make guarantees about systems composed of multiple components [46].

Object Capabilities and Sandboxes. Capabilities as a means to support the development of concurrent and distributed system were developed in the 60's by Dennis and Van Horn [18], and were adapted to the programming languages setting in the 70's [44]. *Object capabilities* were first introduced [39] in the early 2000s, and many recent studies manage to verify safety or correctness of object capability programs. Google's Caja [42] applies sandboxes, proxies, and wrappers to limit components' access to *ambient* authority. Sandboxing has been validated formally: Maffeis et al. [35] 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. Recent programming languages [10, 27, 54] including Newspeak [9], Dart [8], Grace [7, 30] and Wyvern [36] have adopted the object capability model.

Verification of Object Capability Programs. Murray made the first attempt to formalise defensive consistency and correctness [46]. Murray's model was rooted in counterfactual causation [34]: an object is defensively consistent when the addition of untrustworthy clients cannot cause well-behaved clients to be given incorrect service. Murray formalised defensive consistency very abstractly, over models of (concurrent) object-capability systems in the process algebra CSP [29], without a specification language for describing effects, such as what it means for an object to provide incorrect service. Both Miller and Murray's definitions are intensional, describing what it means for an object to be defensively consistent.

Drossopoulou and Noble [21, 48] have analysed Miller's Mint and Purse example [39] and discussed the six capability policies as proposed in [39]. In [22], they sketched a specification language, used it to specify the six policies from [39], showed that several possible interpretations were possible, and uncovered the need for another four further policies. They also sketched how a trust-sensitive example (the escrow exchange) could be verified in an open world [24]. Their work does not support the concepts of control, time, or space, as in *Chainmail*, but it offers a primitive expressing trust.

Devriese et al. [19] have deployed powerful theoretical techniques to address similar problems: They show how step-indexing, Kripke worlds, and representing objects as state machines with public and private transitions can be used to reason about object capabilities. Devriese have demonstrated solutions to a range of exemplar problems, including the DOM wrapper (replicated in our section 7.3) and a mashup application. Their distinction between public and private transitions is similar to the distinction between internal and external objects.

More recently, Swasey et al. [59] designed OCPL, a logic for object capability patterns, that supports specifications and proofs for object-oriented systems in an open world. They draw on verification techniques for security and information flow: separating internal implementations ("high values" which must not be exposed to attacking code) from interface objects ("low values" which may be exposed). OCPL supports defensive consistency (they use the term "robust safety" from the security community [6]) via a proof system that ensures low values can never leak high values to external attackers. This means that low values *can* be exposed to external code, and the behaviour of the system is described by considering attacks only on low values. They use that logic to prove a number of object-capability patterns, including sealer/unsealer pairs, the caretaker, and a general membrane.

Schaefer et al. [55] have recently added support for information-flow security using refinement to ensure correctness (in this case confidentiality) by construction. By enforcing encapsulation, all these approaches share similarity with techniques such as ownership types [14, 50], which also protect internal implementation objects from accesses that cross encapsulation boundaries. Banerjee and Naumann demonstrated that by ensuring confinement, ownership systems can enforce representation independence (a property close to "robust safety") some time ago [3].

Chainmail differs from Swasey, Schaefer's, and Devriese's work in a number of ways: They are primarily concerned with mechanisms that ensure encapsulation (aka confinement) while we abstract away from any mechanism via the $\mathsf{External}\langle\ \rangle$ predicate. They use powerful mathematical techniques which the users need to understand in order to write their specifications, while *Chainmail* users only need to understand first order logic and the holistic operators presented in this paper. Finally, none of these systems offer the kinds of holistic assertions addressing control flow, change, or temporal operations that are at the core of *Chainmail*'s approach.

Scilla [56] is a minimalistic typed functional language for writing smart contracts that compiles to the Ethereum bytecode. Scilla's semantic model is restricted, assuming actor based communication and restricting recursion, thus facilitating static analysis of Scilla contracts and ensuring termination. Scilla is able to demonstrate that a number of popular Ethereum contracts avoid type errors, out-of-gas resource failures, and preservation of virtual currency. Scilla's semantics are defined formally, but have not yet been represented in a mechanised model.

Finally, the recent VerX tool is able to verify a range of specifications for solidity contracts automatically [52]. Similar to *Chainmail*, VerX has a specification language based on temporal logic. VerX offers three temporal operators (always, once, prev) but only within a past modality, while *Chainmail* has two temporal operators, both existential, but with both past and future modalities. VerX specifications can also include predicates that model the current invocation on a contract (similar to *Chainmail*'s "calls"), can access variables, and compute sums (only) over collections. *Chainmail* is strictly more expressive as a specification language, including quantification over objects and sets (so can compute arbitrary reductions on collections) and of course specifications for permission ("access"), space ("in") and viewpoint ("external") which have no analogues in VerX. Unlike *Chainmail*, VerX includes a practical tool that has been used to verify a hundred properties across case studies of twelve Solidity contracts.

10 CONCLUSIONS

In this paper we have motivated the need for holistic specifications, presented the specification language *Chainmail* for writing such specifications, and outlined the formal foundations of the language. To focus on the key attributes of a holistic specification language, we have kept *Chainmail* simple, only requiring an understanding of first order logic. We believe that the holistic features (permission, control, time, space and viewpoint) are intuitive concepts when reasoning informally, and were pleased to have been able to provide their formal semantics in what we argue is a simple manner.

11 ACKNOWLEDGMENTS

This work is based on a long-standing collaboration with Mark S. Miller and Toby Murray. We have received invaluable feedback from Alex Summers, Bart Jacobs, Chris Hawblitzel, Michael Jackson, Lucius G. Meredith, Mike Stay, Shuh Peng Loh, Emil Klasan, members of WG 2.3, and the FASE 2020 reviewers. The work has been supported by the Royal Society of New Zealand (Te Apārangi)

Marsden Fund (Te Pūtea Rangahau a Marsden) grants VUW-1318 and VUW-1815, and research gifts from Agoric, the Ethereum Foundation, and Facebook.

REFERENCES

[1] [n.d.]. Holistic Specifications paper with appendices. https://arxiv.org/abs/2002.08334. Accessed: 2020-02-21.

[2] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *POPL*.

[3] Anindya Banerjee and David A. Naumann. 2005. Ownership Confinement Ensures Representation Independence for Object-oriented Programs. *J. ACM* 52, 6 (Nov. 2005), 894–960.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. 2005. The Spec# Programming System: An Overview. In *CASSIS (LNCS)*. Springer, 49–69.

[5] BBC: On This Day. 2015. 1984: Tory Cabinet in Brighton bomb blast. [Online; accessed 15-October-2015].

[6] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.* 33, 2, Article 8 (Feb. 2011), 45 pages.

[7] Andrew Black, Kim Bruce, Michael Homer, and James Noble. 2012. Grace: the Absence of (Inessential) Difficulty. In *Onwards*.

[8] Gilad Bracha. 2015. *The Dart Programming Language*.

[9] Gilad Bracha. 2017. The Newspeak Language Specification Version 0.1. (Feb. 2017). `newspeaklanguage.org/`.

[10] Anton Burtsev, David Johnson, Josh Kunz, Eric Eide, and Jacobus E. van der Merwe. 2017. CapNet: security and least authority in a capability-enabled cloud. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017*. 128–141.

[11] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. 342–363. https://doi.org/10.1007/11804192_16

[12] Christoph Jentsch. 2016. Decentralized Autonomous Organization to automate governance. (March 2016). https://download.slock.it/public/DAO/WhitePaper.pdf

[13] David Clarke and Sophia Drossopoulou. 2002. Ownership, encapsulation and the disjointness of type and effectr. In *OOPSLA (ACM)*.

[14] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. ACM.

[15] Coindesk. 2016. Understanding The DAO Attack. (2016). www.coindesk.com/understanding-dao-hack-journalists/.

[16] Solidity Community. [n.d.]. Solidity. https://solidity.readthedocs.io/en/develop/.

[17] Tom Van Cutsem and Mark S. 2013. Trustworthy Proxies: Virtualizing Objects with Invariants. In *ECOOP*.

[18] Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Comm. ACM* 9, 3 (1966).

[19] Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *IEEE EuroS&P*. 147–162. https://doi.org/10.1109/EuroSP.2016.22

[20] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. 2017. Evil Pickles: DoS Attacks Based on Object-Graph Engineering. In *ECOOP*. 10:1–10:32.

[21] Sophia Drossopoulou and James Noble. 2013. The Need for Capability Policies. In *(FTfJP)*.

[22] Sophia Drossopoulou and James Noble. 2014. Towards Capability Policy Specification and Verification. `ecs.victoria-.ac.nz/Main/TechnicalReportSeries`.

[23] Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. 2020. Holisitic Specifications for Robust Programs - Coq Model. https://doi.org/10.5281/zenodo.3677621

[24] Sophia Drossopoulou, James Noble, and Mark Miller. 2015. Swapsies on the Internet: First Steps towards Reasoning about Risk and Trust in an Open World. In *(PLAS)*.

[25] J. V. Guttag and J. J. Horning. 1993. *Larch: Languages and Tools for Formal Specification*. Springer.

[26] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. 2012. Behavioral interface specification languages. *ACM Comput.Surv.* 44, 3 (2012), 16.

[27] Ian J. Hayes, Xi Wu, and Larissa A. Meinicke. 2017. Capabilities for Java: Secure Access to Resources. In *APLAS*. 67–84.

[28] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Comm. ACM* 12 (1969), 576–580.

[29] C. A. R. Hoare. 1985. *Communicating Sequential Processes*. Prentice Hall.

[30] Timothy Jones, Michael Homer, James Noble, and Kim B. Bruce. 2016. Object Inheritance Without Classes. In *ECOOP*. 13:1–13:26.

[31] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. 2007. JML Reference Manual. (February 2007). Iowa State Univ. www.jmlspecs.org.

[32] K. R. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR16*. Springer.

[33] K. Rustan M. Leino and Wolfram Schulte. 2007. Using History Invariants to Verify Observers. In *ESOP*.

[34] David Lewis. 1973. Causation. *Journal of Philosophy* 70, 17 (1973).

[35] S. Maffeis, J.C. Mitchell, and A. Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *Proc of IEEE Security and Privacy*.

[36] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *ECOOP*. 20:1–20:27.

[37] B. Meyer. 1992. *Eiffel: The Language*. Prentice Hall.

[38] B. Meyer. 1997. *Object-Oriented Software Construction, Second Edition* (second ed.). Prentice Hall.

[39] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Baltimore, Maryland.

[40] Mark S. Miller, Tom Van Cutsem, and Bill Tulloh. 2013. Distributed Electronic Rights in JavaScript. In *ESOP*.

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

[42] Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Safe active content in sanitized JavaScript. code.google.com/p/google-caja/.

[43] Mitre Organisation. 2019. CWE-830: Inclusion of Web Functionality from an Untrusted Source. https://cwe.mitre.org/data/definitions/830.html

[44] James H. Morris Jr. 1973. Protection in Programming Languages. *CACM* 16, 1 (1973).

[45] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. 2006. Modular Invariants for Layered Object Structures. *Science of Computer Programming* 62 (2006), 253–286.

[46] Toby Murray. 2010. *Analysing the Security Properties of Object-Capability Patterns*. Ph.D. Dissertation. University of Oxford.

[47] Toby Murray, Robert Sison, and Kai Engelhardt. 2018. Covern: A Logic for Compositional Verification of Information Flow Control. In *EuroS&P*.

[48] James Noble and Sophia Drossopoulou. 2014. Rationally Reconstructing the Escrow Example. In *FTfJP*.

[49] James Noble, Alex Potanin, Toby Murray, and Mark S. Miller. 2018. Abstract and Concrete Data Types vs Object Capabilities. In *Principled Software Development*, Peter Müller and Ina Schaefer (Eds.).

[50] James Noble, John Potter, and Jan Vitek. 1998. Flexible Alias Protection. In *ECOOP*.

[51] D.J. Pearce and L.J. Groves. 2015. Designing a Verifying Compiler: Lessons Learned from Developing Whiley. *Sci. Comput. Prog.* (2015).

[52] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *IEEE Symp. on Security and Privacy*.

[53] J. C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74.

[54] Dustin Rhodes, Tim Disney, and Cormac Flanagan. 2014. Dynamic Detection of Object Capability Violations Through Model Checking. In *DLS*. 103–112.

[55] Ina Schaefer, Tobias Runge, Alexander Knüppel, Loek Cleophas, Derrick G. Kourie, and Bruce W. Watson. 2018. Towards Confidentiality-by-Construction. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I*. 502–515.

[56] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan. 2019. Safer Smart Contract Programming with Scilla. In *OOPSLA*.

[57] Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit Dynamic Frames. *ToPLAS* (2012).

[58] Alexander J. Summers and Sophia Drossopoulou. 2010. Considerate Reasoning and the Composite Pattern. In *VMCAI*.

[59] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. In *OOPSLA*.

[60] J. P. Talpin and P. Jouvelot. 1992. The Type and Effect Discipline. In *LICS*. 162–173.

[61] The Ethereum Wiki. 2018. ERC20 Token Standard. (Dec. 2018). https://theethereum.wiki/w/index.php/ERC20_Token_Standard

[62] Tom van Cutsem. 2012. Membranes in Javascript. Available from `prog.vub.ac.be/ ~tvcutsem/invokedynamic/-js-membranes`.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

A PROGRAMMING LANGUAGE FORMALISM

A.1 Modules and Classes

$\mathcal{L}_{oo}$ programs consist of modules, which are repositories of code. Since we study class based oo languages, in this work, code is represented as classes, and modules are mappings from identifiers to class descriptions.

Definition 15 (Modules). *We define Module as the set of mappings from identifiers to class descriptions (the latter defined in Definition 16):*

$$\mathit{Module}\ \triangleq\ \{\ \mathtt{M}\ \mid\ \mathtt{M}:\mathtt{Identifier}\ \longrightarrow\ \mathit{ClassDescr}\ \}$$

Classes, as defined below, consist of field, method definitions and ghost field declarations. $\mathcal{L}_{oo}$ is untyped, and therefore fields are declared without types, method signatures and ghost field signatures consist of sequences of parameters without types, and no return type. Method bodies consist of sequences of statements; these can be field read or field assignments, object creation, method calls, and return statements. All else, *e.g.* booleans, conditionals, loops, can be encoded. Field read or write is only allowed if the object whose field is being read belongs to the same class as the current method. This is enforced by the operational semantics, *c.f.* Fig. 8. Ghost fields are defined as implicit, side-effect-free functions with zero or more parameters. They are ghost information, *i.e.* they are not directly stored in the objects, and are not read/written during execution. When such a ghostfield is mentioned in an assertion, the corresponding function is evaluated. More in section 5.3. Note that the expressions that make up the bodies of ghostfield declarations ($\mathtt{e}$) are more complex than the terms that appear in individual statements.

From now on we expect that the set of field and the set of ghostfields defined in a class are disjoint.

Definition 16 (Classes). *Class descriptions consist of field declarations, method declarations, ghostfield declarations, and, optionally, a constructor.*

$$\begin{array}{lcl}
\mathit{ClassDescr} & ::= & \text{class } \mathit{ClassId}\,(\,\mathtt{x}^*)\,\{\ \ (\,\mathit{FieldDecl}\,)^*\ \ (\,\mathit{CDecl}\,)?\ \ (\,\mathit{MethDecl}\,)^*\ \ (\,\mathit{GhosDecl}\,)^*\ \ \}\\
\mathit{FieldDecl} & ::= & \text{field } \mathtt{f}\\
\mathit{CDecl} & ::= & \text{constructor}\,(\,\mathtt{x}^*)\,\{\ \ \mathit{Stmts}\ \ \}\\
\mathit{MethDecl} & ::= & \text{method } \mathtt{m}(\,\mathtt{x}^*)\,\{\ \ \mathit{Stmts}\ \ \}\\
\mathit{Stmts} & ::= & \mathit{Stmt}\ \mid\ \mathit{Stmt}\ ;\ \mathit{Stmts}\\
\mathit{Stmt} & ::= & \mathtt{x.f := x}\ \mid\ \mathtt{x := x.f}\ \mid\ \mathtt{x := x.m}(\,\mathtt{x}^*)\ \mid\ \mathtt{x :=}\ \text{new }\mathtt{C}(\,\mathtt{x}^*)\ \mid\ \text{return }\mathtt{x}\\
\mathit{GhostDecl} & ::= & \text{ghost } \mathtt{f}(\ \mathtt{x}^*\ )\,\{\ \mathtt{e}\,\}\\
\mathtt{e} & ::= & \text{true}\ \mid\ \text{false}\ \mid\ \text{null}\ \mid\ \mathtt{x}\ \mid\ \mathtt{e=e}\ \mid\ \text{if } \mathtt{e} \text{ then } \mathtt{e} \text{ else } \mathtt{e}\ \mid\ \mathtt{e.f}(\ \mathtt{e}^*\ )\\
\mathtt{x}, \mathtt{f}, \mathtt{m} & ::= & \mathtt{Identifier}
\end{array}$$

where we use metavariables as follows: $\mathtt{x} \in \mathit{VarId}$ $\ \mathtt{f} \in \mathit{FldId}$ $\ \mathtt{m} \in \mathit{MethId}$ $\ \mathtt{C} \in \mathit{ClassId}$*, and* $\mathtt{x}$ *includes* `this`

We define a method lookup function $\mathcal{M}$, which returns the corresponding method definition given a class $\mathtt{C}$ and a method identifier $\mathtt{m}$, and similarly a ghostfield lookup function $\mathcal{G}$, and a constructor lookup function C, which returns a default, field-filling constructor if one wasn't defined.

Definition 17 (Lookup). *For a class identifier* $\mathtt{C}$ *and a method identifier* $\mathtt{m}$*:*

$$\mathcal{M}(\mathtt{M},\mathtt{C},\mathtt{m}) \triangleq \begin{cases} \mathtt{m}\,(\,\mathtt{p}_1,\ldots\mathtt{p}_n)\,\{\,\mathit{Stmts}\,\} \\ \qquad\qquad \mathit{if}\, \mathtt{M}(\mathtt{C}) = \text{class}\,\mathtt{C}\ \{\ldots\ \text{method}\,\mathtt{m}\,(\,\mathtt{p}_1,\ldots\mathtt{p}_n)\,\{\,\mathit{Stmts}\,\}\ \ldots\,\}\,. \\ \mathit{undefined},\quad \mathit{otherwise}. \end{cases}$$

$$\mathcal{G}(\texttt{M},\texttt{C},\texttt{f}) \quad \triangleq \begin{cases} \texttt{f}\,(\,\texttt{p}_1,...\texttt{p}_n)\,\{\,\texttt{e}\,\} \\ \qquad\qquad \textit{if}\,\texttt{M}(\texttt{C}) = \text{class}\,\texttt{C}\ \{\,...\,\text{ghost}\,\texttt{m}\,(\,\texttt{p}_1,...\texttt{p}_n)\,\{\,\texttt{e}\,\}\,...\,\}\,. \\ \textit{undefined},\quad \textit{otherwise.} \end{cases}$$

$$\mathcal{C}(\texttt{M},\texttt{C}) \qquad \triangleq \begin{cases} \text{constructor}\,(\,\texttt{p}_1,...\texttt{p}_n)\,\{\,\textit{Stmts}\,\} \\ \qquad\qquad \textit{if}\,\texttt{M}(\texttt{C}) = \text{class}\,\texttt{C}\ \{\,...\,\text{constructor}\,(\,\texttt{p}_1,...\texttt{p}_n)\,\{\,\textit{Stmts}\,\}\,...\,\}\,. \\ \text{constructor}\,(\,\texttt{p}_1,...\texttt{p}_n)\,\{\,\texttt{this.f}_1\texttt{:=p}_1\texttt{;}\,...\texttt{this.f}_n\texttt{:=p}_n\,\} \\ \qquad\qquad \textit{where}\,\texttt{M}(\texttt{C}) = \text{class}\,\texttt{C}\ \{\,\text{field}\,\texttt{f}_1\,...\text{field}\,\texttt{f}_n\,...\,\}\,. \end{cases}$$

A.2 The Operational Semantics of $\mathcal{L}_{oo}$

We will now define execution of $\mathcal{L}_{oo}$ code. We start by defining the runtime entities, and runtime configurations, σ, which consist of heaps and stacks of frames. The frames are pairs consisting of a continuation, and a mapping from identifiers to values. The continuation represents the code to be executed next, and the mapping gives meaning to the formal and local parameters.

Definition 18 (Runtime Entities). *We define addresses, values, frames, stacks, heaps and runtime configurations.*

- *We take addresses to be an enumerable set,* `Addr`*, and use the identifier* $\alpha \in \texttt{Addr}$ *to indicate an address.*
- *Values, v, are either addresses, or sets of addresses or null:* $v \in \{\texttt{null}\} \cup \texttt{Addr} \cup \mathcal{P}(\texttt{Addr})$.
- *Continuations are either statements (as defined in Definition 16) or a marker,* $\texttt{x:=}\bullet$, *for a nested call followed by statements to be executed once the call returns.*
  $\quad \textit{Continuation} \ ::= \ \textit{Stmts} \ \mid \ \texttt{x:=}\bullet\ ;\ \textit{Stmts}$
- *Frames, ϕ, consist of a code stub and a mapping from identifiers to values:* $\phi \ \in \ \textit{CodeStub} \times \texttt{Ident} \longrightarrow \textit{Value},$
- *Stacks, ψ, are sequences of frames,* $\psi \ ::= \phi \mid \phi \cdot \psi$.
- *Objects consist of a class identifier, and a partial mapping from field identifier to values:* $\textit{Object} \ = \ \texttt{ClassID} \times (\texttt{FieldId} \longrightarrow \textit{Value})$.
- *Heaps, χ, are mappings from addresses to objects:* $\chi \ \in \ \texttt{Addr} \longrightarrow \textit{Object}$.
- *Runtime configurations, σ, are pairs of stacks and heaps,* $\sigma \ ::= \ (\ \psi, \chi\ )$.

Note that values may be sets of addresses. Such values are never part of the execution of $\mathcal{L}_{oo}$, but are used to give semantics to assertions . Next, we define the interpretation of variables (`x`) and field look up (`x.f`) in the context of frames, heaps and runtime configurations; these interpretations are used to define the operational semantics and also the validity of assertions, later on in Definitions 3-7.

Definition 19 (Interpretations). *We first define lookup of fields and classes, where α is an address, and* `f` *is a field identifier:*

- $\chi(\alpha,\texttt{f}) \triangleq \textit{fldMap}(\alpha,\texttt{f}) \quad \textit{if} \quad \chi(\alpha) = (\_,\textit{fldMap})$.
- $\textit{Class}(\alpha)_\chi \triangleq \texttt{C} \quad \textit{if} \quad \chi(\alpha) = (\texttt{C},\_)$

We now define interpretations as follows:

- $\lfloor\texttt{x}\rfloor_\phi \triangleq \phi(\texttt{x})$
- $\lfloor\texttt{x.f}\rfloor_{(\phi,\chi)} \triangleq v, \quad \textit{if} \quad \chi(\phi(\texttt{x})) = (\_,\textit{fldMap})\ \textit{and}\ \textit{fldMap}(\texttt{f}){=}v$

For ease of notation, we also use the shorthands below:

- $\lfloor\texttt{x}\rfloor_{(\phi\cdot\psi,\chi)} \triangleq \lfloor\texttt{x}\rfloor_\phi$
- $\lfloor\texttt{x.f}\rfloor_{(\phi\cdot\psi,\chi)} \triangleq \lfloor\texttt{x.f}\rfloor_{(\phi,\chi)}$
- $\textit{Class}(\alpha)_{(\psi,\chi)} \triangleq \textit{Class}(\alpha)_\chi$

methCall_OS

$$\frac{\begin{array}{l}\phi.\texttt{contn} = \texttt{x:= x}_0\texttt{.m(}\,\texttt{x}_1,\ldots\texttt{x}_n\texttt{)}\ ;\ \texttt{Stmts} \qquad \lfloor \texttt{x}_0\rfloor_\phi = \alpha \\ \mathcal{M}(\texttt{M}, Class(\alpha)_\chi, \texttt{m}) = \texttt{m(}\,\texttt{p}_1,\ldots\texttt{p}_n\texttt{)}\{\,\texttt{Stmts}_1\,\} \\ \phi'' = (\,\texttt{Stmts}_1,\ (\,\texttt{this}\mapsto\alpha, \texttt{p}_1\mapsto\lfloor\texttt{x}_1\rfloor_\phi,\ldots\texttt{p}_n\mapsto\lfloor\texttt{x}_n\rfloor_\phi\,)\,)\end{array}}{\texttt{M},\ (\,\phi\cdot\psi,\ \chi\,)\ \leadsto\ (\,\phi''\cdot\phi[\texttt{contn}\mapsto\texttt{x:=}\bullet\ ;\ \texttt{Stmts}]\cdot\psi,\ \chi\,)}$$

varAssgn_OS

$$\frac{\phi.\texttt{contn} = \texttt{x:= y.f ; Stmts} \qquad Class(\texttt{y})_\sigma = Class(\texttt{this})_\sigma}{\texttt{M},\ (\,\phi\cdot\psi, \chi\,)\ \leadsto\ (\,\phi[\texttt{contn}\mapsto\texttt{Stmts}, \texttt{x}\mapsto\lfloor\texttt{y.f}\rfloor_{\phi,\chi}]\cdot\psi,\ \chi\,)}$$

fieldAssgn_OS

$$\frac{\phi.\texttt{contn} = \texttt{x.f:=y; Stmts} \qquad Class(\texttt{x})_\sigma = Class(\texttt{this})_\sigma}{\texttt{M},\ (\,\phi\cdot\psi, \chi\,)\ \leadsto\ (\,\phi[\texttt{contn}\mapsto\texttt{Stmts}]\cdot\psi, \chi[\lfloor\texttt{x}\rfloor_\phi, \texttt{f}\mapsto\lfloor\texttt{y}\rfloor_{\phi,\chi}]\,)}$$

objCreate_OS

$$\frac{\begin{array}{l}\phi.\texttt{contn} = \texttt{x:=}\textsf{new}\,\texttt{C(}\,\texttt{x}_1,\ldots\texttt{x}_n\texttt{)}\ ;\ \texttt{Stmts} \qquad \alpha \text{ new in } \chi \\ \mathcal{C}(\texttt{M},\texttt{C}) = \textsf{constructor}(\,\texttt{p}_1,\ldots\texttt{p}_n)\{\,\texttt{Stmts}_1\,\} \\ \phi'' = (\,\texttt{Stmts}_1;\ \textsf{return}\ \texttt{this},\ (\,\texttt{this}\mapsto\alpha, \texttt{p}_1\mapsto\lfloor\texttt{x}_1\rfloor_\phi,\ldots\texttt{p}_n\mapsto\lfloor\texttt{x}_n\rfloor_\phi\,)\,)\end{array}}{\texttt{M},\ (\,\phi\cdot\psi, \chi\,)\ \leadsto\ (\,\phi''\cdot\phi[\texttt{contn}\mapsto\texttt{x:=}\bullet\ ;\ \texttt{Stmts}]\cdot\psi,\ \chi[\alpha\mapsto(\texttt{C},\emptyset)]\,)}$$

return_OS

$$\frac{\begin{array}{l}\phi.\texttt{contn} = \textsf{return}\,\texttt{x; Stmts} \quad \textit{or} \quad \phi.\texttt{contn} = \textsf{return}\,\texttt{x} \\ \phi'.\texttt{contn} = \texttt{x}'\texttt{:=}\bullet\ ;\ \texttt{Stmts}'\end{array}}{\texttt{M},\ (\,\phi\cdot\phi'\cdot\psi, \chi\,)\ \leadsto\ (\,\phi'[\texttt{contn}\mapsto\texttt{Stmts'}, \texttt{x}'\mapsto\lfloor\texttt{x}\rfloor_\phi]\cdot\psi,\ \chi\,)}$$

Fig. 8. Operational Semantics

- $Class(\texttt{x})_\sigma \triangleq Class(\lfloor\texttt{x}\rfloor_\sigma)_\sigma$

In the definition of the operational semantics of $\mathcal{L}_{oo}$ we use the following notations for lookup and updates of runtime entities :

Definition 20 (Lookup and update of runtime configurations). *We define convenient shorthands for looking up in runtime entities.*

- *Assuming that ϕ is the tuple $(\texttt{stub}, varMap)$, we use the notation $\phi.\texttt{contn}$ to obtain* stub.
- *Assuming a value v, and that ϕ is the tuple $(\texttt{stub}, varMap)$, we define $\phi[\texttt{contn} \mapsto \texttt{stub'}]$ for updating the stub, i.e. $(\texttt{stub'}, varMap)$. We use $\phi[\texttt{x} \mapsto v]$ for updating the variable map, i.e. $(\texttt{stub}, varMap[\texttt{x} \mapsto v])$.*
- *Assuming a heap χ, a value v, and that $\chi(\alpha) = (\texttt{C}, fieldMap)$, we use $\chi[\alpha, \texttt{f} \mapsto v]$ as a shorthand for updating the object, i.e. $\chi[\alpha \mapsto (\texttt{C}, fieldMap[\texttt{f} \mapsto v]]$.*

Execution of a statement has the form $\texttt{M}, \sigma \leadsto \sigma'$, and is defined in figure 8.

Definition 21 (Execution). *of one or more steps is defined as follows:*

- *The relation $\texttt{M}, \sigma \leadsto \sigma'$, it is defined in Figure 8.*
- *$\texttt{M}, \sigma \leadsto^* \sigma'$ holds, if i) σ=σ', or ii) there exists a σ'' such that $\texttt{M}, \sigma \leadsto^* \sigma''$ and $\texttt{M}, \sigma'' \leadsto \sigma'$.*

A.3 Module linking

When studying validity of assertions in the open world we are concerned with whether the module under consideration makes a certain guarantee when executed in conjunction with other modules. To answer this, we need the concept of linking other modules to the module under consideration. Linking, $\circ$, is an operation that takes two modules, and creates a module which corresponds to the union of the two. We place some conditions for module linking to be defined: We require that the two modules do not contain implementations for the same class identifiers,

Definition 22 (Module Linking). *The linking operator* $\circ: \ Module \times Module \longrightarrow Module$ *is defined as follows:*

$$\mathtt{M}\circ\mathtt{M}' \ \triangleq \ \begin{cases} \mathtt{M} \circ_{aux} \mathtt{M}', & if \ dom(\mathtt{M})\cap dom(\mathtt{M}')=\emptyset \\ undefined & otherwise. \end{cases}$$

and where,

- *For all* $\mathtt{C}$*:* $(\mathtt{M} \circ_{aux} \mathtt{M}')(\mathtt{C}) \ \triangleq \ \mathtt{M}(\mathtt{C})$ *if* $\mathtt{C} \in dom(\mathtt{M})$*, and* $\mathtt{M}'(\mathtt{C})$ *otherwise.*

Some properties of linking are described in lemma 4.1 in the main text. For the proof, (1) and (2) follow from Definition 22. (3) follows from 22, and the fact that if a lookup $\mathtt{M}$ is defined for $\mathtt{M}$, then it is also defined for $\mathtt{M}\circ\mathtt{M}'$ and returns the same method, and similar result for class lookup.

A.4 Module pairs and visible states semantics

A module $\mathtt{M}$ adheres to an invariant assertion A, if it satisfies A in all runtime configurations that can be reached through execution of the code of $\mathtt{M}$ when linked to that of *any other* module $\mathtt{M}'$, and which are *external* to $\mathtt{M}$. We call external to $\mathtt{M}$ those configurations which are currently executing code which does not come from $\mathtt{M}$. This allows the code in $\mathtt{M}$ to break the invariant internally and temporarily, provided that the invariant is observed across the states visible to the external client $\mathtt{M}'$.

We have defined two module execution in the main paper, Def. 1.

In that definition n is allowed to have the value 2. In this case the final bullet is trivial and there exists a direct, external transition from σ to σ'. Our definition is related to the concept of visible states semantics, but differs in that visible states semantics select the configurations at which an invariant is expected to hold, while we select the states which are considered for executions which are expected to satisfy an invariant. Our assertions can talk about several states (through the use of the $\mathsf{will}\langle\_\rangle$ and $\mathsf{was}\langle\_\rangle$ connectives), and thus, the intention of ignoring some intermediate configurations can only be achieved if we refine the concept of execution.

We have defined initial and arising configurations in Definition 2. Note that there are infinitely many different initial configurations, they will be differing in the code stored in the continuation of the unique frame.

B PROPERTIES OF ASSERTIONS

We define equivalence of assertions in the usual sense: two assertions are equivalent if they are satisfied in the context of the same configurations. Similarly, an assertion entails another assertion, iff all configurations which satisfy the former also satisfy the latter.

Definition 23 (Equivalence and entailments of assertions).

- $A \subseteqq A'$ *if* $\forall\sigma.\forall \mathtt{M},\mathtt{M}'.$ [$\mathtt{M} ⨾ \mathtt{M}', \sigma \models A$ *implies* $\mathtt{M} ⨾ \mathtt{M}', \sigma \models A'$].
- $A \equiv A'$ *if* $A \subseteqq A'$ *and* $A' \subseteqq A$.

Lemma B.1 (Assertions are classical-1). *For all runtime configurations σ, assertions A and A', and modules $\mathtt{M}$ and $\mathtt{M}'$, we have*

(1) $\mathtt{M} ⨾ \mathtt{M}', \sigma \models A$ *or* $\mathtt{M} ⨾ \mathtt{M}', \sigma \models \neg A$

(2) $\mathtt{M} ⨾ \mathtt{M}', \sigma \models A \land A'$ *if and only if* $\mathtt{M} ⨾ \mathtt{M}', \sigma \models A$ *and* $\mathtt{M} ⨾ \mathtt{M}', \sigma \models A'$

(3) $\mathtt{M} ⨾ \mathtt{M}', \sigma \models A \lor A'$ *if and only if* $\mathtt{M} ⨾ \mathtt{M}', \sigma \models A$ *or* $\sigma \models A'$

(4) $\mathtt{M} ⨾ \mathtt{M}', \sigma \models A \land \neg A$ *never holds.*

(5) $\mathtt{M} ⨾ \mathtt{M}', \sigma \models A$ *and* $\mathtt{M} ⨾ \mathtt{M}', \sigma \models A \longrightarrow A'$ *implies* $\mathtt{M} ⨾ \mathtt{M}', \sigma \models A'$.

Proof. The proof of part (1) requires to first prove that for all *basic assertions A*,

(*) either $\mathtt{M} ⨾ \mathtt{M}', \sigma \models A$ or $\mathtt{M} ⨾ \mathtt{M}', \sigma \not\models A$.

We prove this using Definition 5. Then, we prove (*) for all possible assertions, by induction of the structure of A, and the Definitions 6, and also Definitions 7, 8, 9, 11, and 13. Using the definition of $\mathtt{M} ⨾ \mathtt{M}', \sigma \models \neg A$ from Definition 6 we conclude the proof of (1).

For parts (2)-(5) the proof goes by application of the corresponding definitions from 6. Compare also with appendix D.

□

Lemma B.2 (Assertions are classical-2). *For assertions A, A', and A'' the following equivalences hold*

(1) $A \land \neg A \;\equiv\; \mathtt{false}$

(2) $A \lor \neg A \;\equiv\; \mathtt{true}$

(3) $A \land A' \;\equiv\; A' \land A$

(4) $A \lor A' \;\equiv\; A' \lor A$

(5) $(A \lor A') \lor A'' \;\equiv\; A \lor (A' \lor A'')$

(6) $(A \lor A') \land A'' \;\equiv\; (A \land A') \;\lor\; (A \land A'')$

(7) $(A \land A') \lor A'' \;\equiv\; (A \lor A') \;\land\; (A \lor A'')$

(8) $\neg(A \land A') \;\equiv\; \neg A \lor \neg A''$

(9) $\neg(A \lor A') \;\equiv\; \neg A \land \neg A'$

(10) $\neg(\exists \mathrm{x}.A) \;\equiv\; \forall \mathrm{x}.(\neg A)$

(11) $\neg(\exists \mathrm{S} : \mathrm{SET}.A) \;\equiv\; \forall \mathrm{S} : \mathrm{SET}.(\neg A)$

(12) $\neg(\forall \mathrm{x}.A) \;\equiv\; \exists \mathrm{x}.\neg(A)$

(13) $\neg(\forall \mathrm{S} : \mathrm{SET}.A) \;\equiv\; \exists \mathrm{S} : \mathrm{SET}.\neg(A)$

Proof. All points follow by application of the corresponding definitions from 6. Compare also with appendix D. □

C BANK ACCOUNT IMPLEMENTATIONS

```
class Bank{

   method newAccount(amt){
        if (amt>=0) then{
            return new Account(this,amt)
   }   }
}

class Account{

    field balance
    field myBank

    method deposit(src,amt){
       if (amt>=0 && src.myBank=this.myBank && src.balance>=amt) then{
           this.balance = this.balance+amt
           src.balance = src.balance-amt
   }   }
   method makeAccount(amt){
       if (amt>=0 && this.balance>=amt) then{
           this.balance = this.balance - amt;
           return new Account(this.myBank,amt)
   }    }
}
```

Fig. 9. M_{BA1}: Implementation of `Bank` and `Account` – version 1

In this section we revisit the `Bank`/`Account` example from 2, and show two different implementations, derived from Noble et al. [49] . Both implementations satisfy the three functional specifications and the holistic assertions (1), (2) and (3) shown in section 2. The first version gives rise to runtime configurations as σ_1, shown on the left side of Fig. 3, while the second version gives rise to runtime configurations as σ_2, shown on the right side of Fig. 3. in the main text.

In this code, we use more syntax than the minimal syntax defined for $\mathcal{L}_{oo}$ in Def. 15, as we use conditionals, and we allow nesting of expressions, e.g. a field read to be the receiver of a method call. Such extension can easily be encoded in the base syntax easily as the language is untyped, allowing different method implementations on `True` and `False` instances.

$\mathtt{M}_{BA1}$, the fist version is shown Fig. 9. It keeps all the information in the `Account` object: namely, the `Account` contains the pointer to the bank, and the balance, while the `Bank` is a pure capability, which contains no state but is necessary for the creation of new `Account`s. In this version we have no ghost fields.

$\mathtt{M}_{BA1}$, the second version is shown Fig. 12 and 10. It keeps all the information in the `ledger`: each `Node` points to an `Account` and contains the balance for this particular `Account`. Here `balance` is a ghost field of `Account`; the body of that declaration calls the ghost field function `balanceOf` of the `Bank` which in its turn calls the ghost field function `balanceOf` of the `Node`. Note that the latter is recursively defined.

Note also that `Node` exposes the function `addToBalance(...)`; a call to this function modifies the `balance` of an `Account` without requiring that the caller has access to the `Account`. This might

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

   constructor Bank() {
      this.ledger := null
   }

    method deposit(dest,src,amt){
       destNd = this.ledger.find(dest)
       srcNd = this.ledger.find(src)
       srcBalance = srcNd.getBalance()
       if ( destNd =/=null && srcNd=/=null && srcBalance>=amt && amt >=0 ) then
           destNd.addToBalance(amt)
           srcNc.addToBalance(-amt)
    }  }
    method newAccount(amt){
      if (amt>=0)  then{
           newAcc = new Account(this);
           this.ledger = new Node(amt,this.ledger,newAcc)
           return newAcc
    } }

   ghost  balance(acc){ this.ledger.balance(acc)  }
}
```

Fig. 10. M_{BA2}: Implementation of Bank – version 2

look as if it contradicted assertions (1) and (2) from section 2. However, upon closer inspection, we see that the assertion is satisfied. Remember that we employ a two-module semantics, where any change in the balance of an account is observed from one external state, to another external state. By definition, a configuration is external if its receiver is external. However, no external object will ever have access to a Node, and therefore no external object will ever be able to call the method addToBalance(...). In fact, we can add another assertion, (4), which promises that any internal object which is externally accessible is either a Bank or an Account.

$$(4) \triangleq \forall \mathtt{o}, \forall \mathtt{o}'.[\ \ \text{external}\langle\, \mathtt{o}\, \rangle\ \wedge\ \neg(\text{external}\langle\, \mathtt{o}'\, \rangle)\ \wedge\ \langle\, \mathtt{o}\ \text{access}\ \mathtt{o}'\, \rangle \\ \longrightarrow\ [\ \mathtt{o} : \mathtt{Account}\ \vee\ \mathtt{o}' : \mathtt{Bank}\ ]\quad]$$

```
class Node{
   field balance
   field next
   field myAccount

   method addToBalance(amt){
       this.balance = this.balance + amt
   }
   method find(acc){
      if this.myAccount == acc then{
          return this
     } else {
          if this.next==null then{
              return null
          } else {
              return this.next.find(acc)
    }  } }
    method getBalance(){ return balance }

    ghost balance(acc){
        if (this.myAccount == acc)  then  this.balance
             else  ( if this.next==null then -1 else this.next.balance(acc) )
    }
}
```

Fig. 11. M_{BA2}: Implementation of `Node` – version 2

```
class Account{

    field myBank

    method deposit{src,amt){
             this.myBank.deposit(this,src,amt)
    }   }
    method makeAccount(amt){
      if (amt>=0 && this.balance>=amt) then{
           newAcc = this.myBank.makeNewAccount(0)
           newAcc.deposit(this,amt)
           return newAcc
    }    }

   ghost balance(){ this.myBank.balance(this) }
}
```

Fig. 12. M_{BA2}: Implementation of `Account` – version 2

Lemma 4.1 (and 3)	Properties of Linking	(1) `moduleLinking_associative` (2) `moduleLinking_commutative_1` (3) `moduleLinking_commutative_2` (4) `linking_preserves_reduction`
Lemma B.1	(1) $A \wedge \neg A \equiv \texttt{false}$ (2) $A \vee \neg A \equiv \texttt{true}$ (3) $A \vee A' \equiv A' \wedge A$ (4) $A \wedge A' \equiv A' \wedge A$ (5) $(A \vee A') \vee A'' \equiv A \vee (A' \vee A'')$	(1) `sat_and_nsat_equiv_false` (2) - (3) `and_commutative` (4) `or_commutative` (5) `or_associative`
Lemma B.2	(1) $A \wedge \neg A \equiv \texttt{false}$ (2) $A \vee \neg A \equiv \texttt{true}$ (3) $A \vee A' \equiv A' \wedge A$ (4) $A \wedge A' \equiv A' \wedge A$ (5) $(A \vee A') \vee A'' \equiv A \vee (A' \vee A'')$ (6) $(A \vee A') \wedge A'' \equiv (A \vee A'') \wedge (A' \vee A'')$ (7) $(A \wedge A') \vee A'' \equiv (A \wedge A'') \vee (A' \wedge A'')$ (8) $\neg(A \wedge A') \equiv (\neg A \vee \neg A')$ (9) $\neg(A \vee A') \equiv (\neg A \wedge \neg A')$ (10) $\neg(\exists x.A) \equiv \forall x.(\neg A)$ (11) $\neg(\exists S.A) \equiv \forall S.(\neg A)$ (12) $\neg(\forall x.A) \equiv \exists x.(\neg A)$ (13) $\neg(\forall S.A) \equiv \exists S.(\neg A)$	(1) `sat_and_nsat_equiv_false` (2) - (3) `and_commutative` (4) `or_commutative` (5) `or_associative` (6) `and_distributive` (7) `or_distributive` (8) `neg_distributive_and` (9) `neg_distributive_or` (10) `not_ex_x_all_not` (11) `not_ex_Σ_all_not` (12) `not_all_x_ex_not` (13) `not_all_Σ_ex_not`

Table 1. Chainmail Properties Formalised in Coq

D COQ FORMALISM

We present the properties of Chainmail that have been formalised in the Coq model. Table 1 refers to proofs that can be found in the associated Coq formalism [23].