Holistic Specifications for Robust Programs

version of July 2019

SOPHIA DROSSOPOULOU, Imperial College London
JAMES NOBLE, Victoria University of Wellington
SUSAN EISENBACH, Imperial College London

Functional specifications of program components describe what the code *can* do — the *sufficient* conditions to invoke the component's operations: a client who supplies arguments meeting that operation's preconditions can invoke it, and obtain the associated effect. While specifications of sufficient conditions may be enough to reason about complete, unchanging programs, they cannot support reasoning about components that interact with external components of possibly unknown provenance. 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. By adopting holistic specification techniques, programmers can explicitly define what their components should not do, making it easier to write robust and reliable programs.

In this paper we argue for the need for such holistic specifications, propose a language *Chainmail* for writing specifications, give a formal semantics, and discuss several examples from the literature.

TODO:
Add equivalences of assertions, eg Wll(A) == Will(Will(A))
WIll(A) = A or Next(A or Will(A))

1 INTRODUCTION

Software guards our secrets, our money, our intellectual property, our reputation [**?**]. 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*. We expect software 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 [**?**], or that an airline will not sell more tickets than the number of seats. The importance of robustness has led to the design of many programming language mechanisms to help developers write robust programs: constant fields, private methods, ownership [**?**] as well as the object capability paradigm [**?**], and its adoption in web systems [**???**], and programming languages such as Newspeak [**?**], Dart [**?**], Grace [**??**], and Wyvern [**?**].

While such programming language mechanisms make it *possible* to write robust programs, they cannot *ensure* that 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 resource belonging to one user should not be consumed by another. To be able to ensure robustness, we need

Authors' addresses: Sophia DrossopoulouImperial College London; James NobleVictoria University of Wellington; Susan EisenbachImperial College London.

–. 2475-1421/–/13-ART $15.00
https://doi.org/

ways to specify what robustness means for a particular program, and ways to demonstrate that the particular program adheres to its specific robustness requirements.

There has been a plethora of work on the specification and verification of the functional correctness of programs. Such specifications describe what are essentially *sufficient* conditions for some effect to happen. For example, if you have enough funds and make a payment request to your bank, money will be transferred and as a result your balance funds will be reduced: enough funds and the payment request is a sufficient condition for the reduction of funds. Often, however, we are also interested in *necessary conditions*: guarantees about things that will *not* happen. For example, you probably want to be assured that no reduction in your bank balance will take place unless you have explicitly requested it. Similarly, medical patients want to be assured that their health data will not be sent to their employer unless they authorised the release: the patient's explicit authorisation is the necessary condition for the communication of health data — under no other circumstances should the data be communicated.

Fig. **??** shows the difference between sufficient and necessary conditions. We represent all possible states through black dots, and place all initial states in the grey box. Sufficient conditions are described on a per-function basis: a function call in a certain state leads to another state – in part (a) we indicate different functions through different shades of blue and green. Necessary conditions, on the other hand, are about the behaviour of a component as a whole, and describe what is guaranteed not to happen; in part (b) we place states that are guaranteed not to be reached in yellow boxes with dotted outlines, and indicate transitions that are guaranteed not to happen through yellow, dotted arrows.

We propose that necessary conditions should be stated explicitly. Specifications should be *holistic*, in the sense that they describe the overall behaviour of a component: not only the behaviour of each individual function, but also limitations on the behaviour that emerges from combinations of functions.

We propose that necessary conditions should be stated explicitly. Specifications should be *holistic*, in the sense that they describe the overall behaviour of a component: not only the behaviour of each individual function, but also limitations on the behaviour that emerges through combinations of functions. Holistic specifications must therefore address sufficient as well as necessary conditions. In part (c) in Fig. **??** we show the immediate consequence of putting together assertions from necessary and sufficient conditions: there are no transitions from or to yellow boxes.

When a component satisfies its holistic specification, then the states in the yellow boxes and the behaviours represented by the yellow arrows cannot occur, even when the component interacts with other software of unknown provenance. (In Section 7 we argue why necessary conditions are more than the complement of sufficient conditions.

Necessary conditions are guarantees upheld throughout program execution. Therefore, they are close to monitor or object invariants [**??**]. The difference between classical invariants and our holistic specifications is that classical invariants reflect on the current program state (*i.e.* the contents of the stack frame and the heap for an individual program component) while holistic specifications reflect on all aspects of a program's execution, potentially across all the components making up that program.

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 [**?**], the DOM [**??**], the Mint/Purse [**?**], the Escrow [**?**], the DAO [**??**] and ERC20 [**?**]. 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. 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. In particular, *Chainmail* extends traditional program specification languages [**??**] 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.

Holistic assertions usually employ several of these features. They often have the form of a guarantee that only if some property holds now will a certain effect occur in the future, or that certain effects can only be caused if another property held earlier. For example, if within a certain heap (space) some change is possible in the future (time), then this particular heap (space again) contains at least one object which has access (permission) to a specific other, privileged object. A module satisfies a holistic assertion if the assertion is satisfied in all runtime configurations reachable through execution of the combined code of our module and any other module. This reflects the open-world view.

The contributions of this paper are:

- the design of the holistic specification language *Chainmail*,
- the semantics of *Chainmail*,
- a validation of *Chainmail* through its application to a sequence of examples.

The rest of the paper is organised as follows: Section 2 motivates our work via an example, and then 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 shows how key points of exemplar problems can be specified in *Chainmail*, section 7 discusses our design, 8 considers related work, and section 9 concludes. We relegate various details to appendices.

2 MOTIVATING EXAMPLE: THE BANK

As a motivating example, we consider a simplified banking application, with objects representing `Accounts` or `Banks`. As in [**?**], `Accounts` belong to `Banks` and hold money (`balances`); with access to two `Accounts` of the same `Bank` one can transfer any amount of money from one to the other. We give a traditional specification in Figure 1.

```
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 makeNewAccount(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. 1. Functional specification of `Bank` and `Account`

The PRE-condition of `deposit` requires that the receiver and the first argument (`this` and `src`) are `Accounts` 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 `makeNewAccount` 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 corresponding amount of money in it.[1]

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. 1, so long as the methods `deposit`, `makeNewAccount`, 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. 1 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. 1, 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

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

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.

(1) $\triangleq$ $\forall$a.[a : Account $\wedge$ changes⟨ a.balance ⟩ $\longrightarrow$
$\exists$o.[⟨ o calls a.deposit(_, _) ⟩ $\vee$ ⟨ o calls _.deposit(a, _) ⟩]]

(2) $\triangleq$ $\forall$a.$\forall$S : Set. [a : Account $\wedge$ ⟨ will⟨ changes⟨ a.balance ⟩⟩ in S ⟩ $\longrightarrow$
$\exists$o. [o $\in$ S $\wedge$ external⟨ o ⟩ $\wedge$ ⟨ o access a ⟩]]

(3) $\triangleq$ $\forall$a.$\forall$S : Set. [a : Account $\wedge$ ⟨ will⟨ $\exists$o.[external⟨ o ⟩ $\wedge$ ⟨ o access a ⟩]⟩ in S ⟩
$\longrightarrow$ $\exists$o′. [o′ $\in$ S $\wedge$ external⟨ o′ ⟩ $\wedge$ ⟨ o′ access a ⟩]]

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 (changes⟨ a.balance ⟩), then there must be some client object o that called the `deposit` method with a as a receiver or as an argument (⟨ o calls _.deposit(_) ⟩).

Assertion (2) similarly constrains any possible change to an account's balance: If at some future point the balance changes (will⟨ changes⟨ ... ⟩⟩), and if this future change is observed with the state restricted to the objects from S (*i.e.* ⟨ ... in S ⟩), then at least one of these objects (o $\in$ S) is external to the `Bank`/`Account` system (external⟨ o ⟩) and has (direct) access to that account object (⟨ o access a ⟩). 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. 1 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 `Accounts`. 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 2. In both diagrams the rounded boxes depict objects: green for those from the `Bank/Account` component, and grey for the "external", "client" objects. The transparent green rectangle shows which objects are contained by the `Bank/Account` component. The object at `1` is a `Bank`, those at `2`, `3` and `4` are `Accounts`, 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 B Fig. 7. Configuration σ_2 may arise from execution using a module M_{BA2}, where `Accounts` have a `myBank` field, `Bank` objects have a `ledger` implemented though a sequence of `Nodes`, each of which has a field pointing to an `Account`, a field `balance`, and a field `next` – the code can be found in appendix B Figs. 9 and 10.

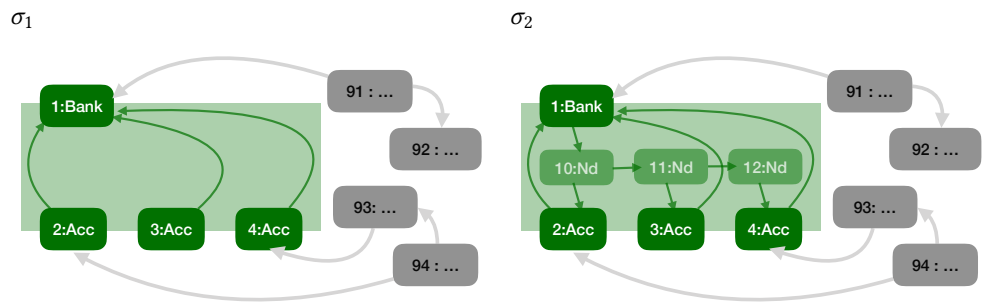


Fig. 2. Two runtime configurations for the `Bank/Account` example.

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

$$\ldots, \sigma_1 \models \texttt{a}_2\texttt{.myBank} = \texttt{a}_3\texttt{.myBank} \qquad \ldots, \sigma_2 \models \texttt{a}_2\texttt{.myBank} = \texttt{a}_3\texttt{.myBank}.$$

The term $\texttt{x:ClassId}$ says that x is an object of class ClassId. For example

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

We support ghost fields [**??**], *e.g.* $\texttt{a}_1\texttt{.balance}$ 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 B Fig. 9).

We also support the usual logical connectives, and so, we can express assertions such as

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

Permission: Access Our first holistic assertion, $\langle\, \texttt{x}\ \mathsf{access}\ \texttt{y}\,\rangle$, 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:

$$\ldots, \sigma_1 \models \langle\, \texttt{a}_2\ \mathsf{access}\ \texttt{b}_1\,\rangle$$

Assuming that σ_1 is executing the method body corresponding to the call $\texttt{a}_2\texttt{.deposit(a}_3\texttt{,360)}$, we have

$$\ldots, \sigma_1 \models \langle\, \texttt{a}_2\ \mathsf{access}\ \texttt{a}_3\,\rangle,$$

Namely, during execution of deposit, the object at $\texttt{a}_2$ has access to the object at $\texttt{a}_3$, and could, if the method body chose to, call a method on $\texttt{a}_3$, or store a reference to $\texttt{a}_3$ in its own fields.

Aaccess is not symmetric, nor transitive:

$$\ldots, \sigma_1 \not\models \langle\, \texttt{a}_3\ \mathsf{access}\ \texttt{a}_2\,\rangle,$$
$$\ldots, \sigma_2 \models \langle\, \texttt{a}_2\ \mathsf{access}^*\ \texttt{a}_3\,\rangle, \qquad \ldots, \sigma_2 \not\models \langle\, \texttt{a}_2\ \mathsf{access}\ \texttt{a}_3\,\rangle.$$

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

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

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

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

$$\ldots, \sigma_1 \models \langle\, (\exists \texttt{o}.\, \langle\, \texttt{o}\ \mathsf{access}\ \texttt{a}_4\,\rangle)\ \mathsf{in}\ \texttt{S}_1\,\rangle \qquad \ldots, \sigma_1 \not\models \langle\, (\exists \texttt{o}.\, \langle\, \texttt{o}\ \mathsf{access}\ \texttt{a}_4\,\rangle)\ \mathsf{in}\ \texttt{S}_2\,\rangle.$$

The set S in the assertion $\langle\, A\ \mathsf{in}\ \texttt{S}\,\rangle$ is therefore *not* the footprint of A; it is more like the *fuel*[**?**] given to establish that assertion. Note that $\ldots, \sigma \models \langle\, A\ \mathsf{in}\ \texttt{S}\,\rangle$ does not imply $\ldots, \sigma \models A$ nor does it imply $\ldots, \sigma \models \langle\, A\ \mathsf{in}\ \texttt{S} \cup \texttt{S}'\,\rangle$. 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 number 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 $\texttt{a}_2\texttt{.deposit(a}_3\texttt{,}\ \texttt{360)}$, and $\texttt{M}_{BA2}\,\texttt{⨟}\ldots, \sigma_4 \models \texttt{a}_2\texttt{.balance} = \texttt{60}$, and that $\texttt{M}_{BA2}\,\texttt{⨟}\ldots, \sigma_4 \models \texttt{a}_4\texttt{.balance} \geq \texttt{360}$

then

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

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

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

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

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

Change and Authority: We have used changes⟨ ⟩ in our *Chainmail* assertions in section 2, as in changes⟨ a.balance ⟩. 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 [**?**]. We could 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. The assertion below says that if the statement to be executed is $\mathtt{a}_2$.deposit(a$_3$, 60), then the balance of $\mathtt{a}_2$ will eventually change:

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

Now look deeper into space assertions, ⟨ A in S ⟩: They allow us to characterise the set of objects which have authority over certain effects (here A). In particular, the assertion ⟨ will⟨ A ⟩ in S ⟩ requires two things: i) that A will hold in the future, and ii) that 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 [**?**]. Notice that the authority is a set, rather than a single object: quite often it takes *several objects in concert* to achieve an effect.

Now, 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 assertion 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. Namely, 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 (⟨ _ access _ ⟩ for permission; ⟨ _ calls _._(_) ⟩ and changes⟨ _ ⟩ for authority) as well as temporal logic (will⟨ A ⟩, was⟨ A ⟩ and friends), and the relation of our spatial connective (⟨ A in S ⟩) with ownership and effect systems [**???**].

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 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 rece *open world*, where M, our module, may be linked with *arbitrary untrusted code*. To model the open world, we consider pairs of modules, $\mathtt{M}\,\S\,\mathtt{M}'$, where M is the module whose code is supposed to satisfy the assertion, and $\mathtt{M}'$ is

another module which exercises the functionality of M. We call our module M the *internal* module, and 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 M′ and in all runtime configurations σ which may be observed as arising from the execution of the code of M combined with that of 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 M′ represents all possible clients of 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 [**??**], 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 will$\langle _ \rangle$.

We distinguish between internal and external modules. This offers two advantages: First, *Chainmail* includes the "external$\langle \circ \rangle$" 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 [**???**], treating all executions within a module as atomic. We only record runtime configurations which are *external* to module M, *i.e.* those where the executing object (*i.e.* the current receiver) comes from module M′. Execution has the form

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

where we ignore all intermediate steps with receivers internal to 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 object respectively. Class definitions consist of field, method and ghost field declarations, c.f. Appendix, Def. 16. $\mathcal{L}_{oo}$ is untyped – this reflects the open world, where we link with external modules which come without any guarantees. Method bodies are sequences of statements, which can be field read or field assignments, object creation, method calls, and return statements. Fields are private in the sense of C++: they can only be read or written by methods of the current class. This is enforced by the operational semantics, *c.f.* Fig. 6. We discuss ghost fields in the next section.

Runtime configurations, σ, contain all the usual information about an execution snapshot: the heap, and a stack of frames. The code to be executed is kept as part of the runtime configuration: 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; the latter are needed to deal with assertions which quantify over sets of objects, as *e.g.* (1) and (2) from section 2. We define *one-module* execution through a judgment of the form $\mathtt{M}, \sigma \rightsquigarrow \sigma'$ in the Appendix, Fig. 6.

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

We define a module linking operator $\circ$ so that $\texttt{M}\circ\texttt{M}'$ is the union of the two modules, provided that their domains are disjoint, c.f. Appendix, Def. 23. As we said in section 4.1, we distinguish between the internal and 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 $\texttt{M} \wr \texttt{M}'$ we define execution where $\texttt{M}$ is the internal, and $\texttt{M}'$ is the external module as below:*

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

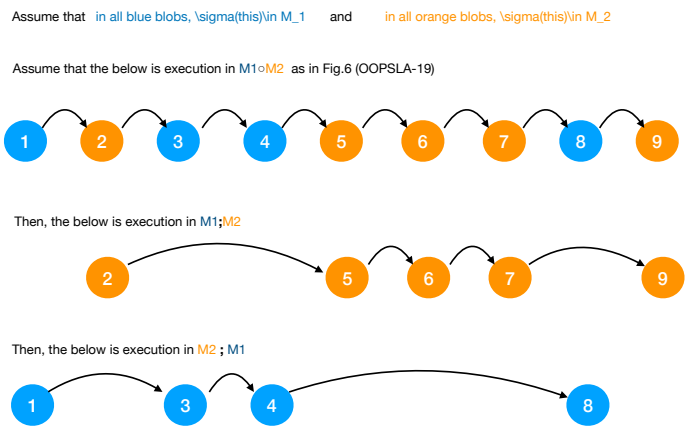


Fig. 3. Illustrating Def. 1

In the definition above, $\mathit{Class}(\texttt{x})_\sigma$ looks up the class of the object stores at $\texttt{x}$, c.f. Appendix, Def. 19. For example, for σ_4 as in Section 3 whose next statement to be executed is $\texttt{a}_2\texttt{.deposit(a}_3\texttt{,360)}$, we would have a sequence of configurations σ_{41}, ... σ_{4n}, σ_5 so that the one-module execution gives $\texttt{M}_{BA2}, \sigma_4 \rightsquigarrow \sigma_{41} \rightsquigarrow \sigma_{42} \ldots \rightsquigarrow \sigma_{4n} \rightsquigarrow \sigma_5$. This would correspond to an atomic evaluation in the two-module execution: $\texttt{M}_{BA2} \wr \texttt{M}', \sigma_4 \rightsquigarrow \sigma_5$.

We illustrate the definition in Fig. 3.

Two-module execution is related to visible states semantics [**?**] 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 $\texttt{M}$, $\texttt{M}'$, $\texttt{M}''$, and $\texttt{M}'''$ and runtime configurations σ, and σ' we have*:

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

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

DEFINITION 2 (INITIAL AND ARISING CONFIGURATIONS). *are defined as follows:*

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

5 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.

5.1 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.*

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

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.* $A \vee \neg A$ is a tautology).

We will discuss evaluation of expressions in section 5.2, standard assertions about heap/stack and logical connectives in 5.3, the treatment of permission, control, space, and viewpoint in 5.4 the treatment of time in 5.5, and properties of assertions in 5.6. 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.

5.2 Values of Expressions

The value of an expression is described through judgment $\texttt{M}, \sigma, \texttt{e} \hookrightarrow v$, defined in Figure 4. 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 $\mathtt{e}_0.\mathtt{f}(\mathtt{e}_1,...\mathtt{e}_n)$, 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 $\mathtt{e}_0$. 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 $\mathtt{M}_0$ 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, $\mathtt{M}_0, \sigma_0$, `acyc.acyclic` $\hookrightarrow$ `true`, but we would have no value for $\mathtt{M}_0, \sigma_0$, `cyc.last` $\hookrightarrow$ ..., nor for $\mathtt{M}_0, \sigma_0$, `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 $\mathtt{M}_{BA1}$ (Appendix B), where we keep the balances in the account objects, this is a physical field. In $\mathtt{M}_{BA2}$ (also in Appendix B), where we keep the balances in a ledger, this is ghost information.

5.3 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.

DEFINITION 4 (INTERPRETATIONS FOR SIMPLE EXPRESSIONS). *For a runtime configuration, σ, variables* `x` *or* `S`*, we define its interpretation as follows:*

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

DEFINITION 5 (BASIC ASSERTIONS). *For modules* `M`*,* `M'`*, configuration σ, we define*:

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

Satisfaction of assertions which contain expressions is predicated on termination of these expressions. Continuing our earlier example, $\mathtt{M}_0 \S \mathtt{M}', \sigma_0 \models$ `acyc.acyclic` holds for any `M'`, while $\mathtt{M}_0 \S \mathtt{M}', \sigma_0 \models$ `cyc.acyclic` does not hold, and $\mathtt{M}_0 \S \mathtt{M}', \sigma_0 \models$ `cyc.acyclic = false` does not hold either. In general, when $\mathtt{M} \S \mathtt{M}', \sigma \models \mathtt{e}$ holds, then $\mathtt{M} \S \mathtt{M}', \sigma \models \mathtt{e} = \mathtt{true}$ holds too. But when

$$\frac{}{\mathtt{M},\sigma,\mathtt{true} \hookrightarrow \mathtt{true}}\ \text{True\_Val} \qquad \frac{}{\mathtt{M},\sigma,\mathtt{false} \hookrightarrow \mathtt{false}}\ \text{False\_Val} \qquad \frac{}{\mathtt{M},\sigma,\mathtt{null} \hookrightarrow \mathtt{null}}\ \text{Null\_Val} \qquad \frac{}{\mathtt{M},\sigma,\mathtt{x} \hookrightarrow \sigma(\mathtt{x})}\ \text{Var\_Val}$$

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

$$\frac{\mathtt{M},\sigma,\mathtt{e.f()} \hookrightarrow v}{\mathtt{M},\sigma,\mathtt{e.f} \hookrightarrow v}\ \text{Field\_Ghost\_Val2}$$

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

$$\frac{\mathtt{M},\sigma,\mathtt{e} \hookrightarrow \mathtt{true} \qquad \mathtt{M},\sigma,\mathtt{e}_1 \hookrightarrow v}{\mathtt{M},\sigma,\mathtt{if\ e\ then\ e}_1\ \mathtt{else\ e}_2 \hookrightarrow v}\ \text{If\_True\_Val}$$

$$\frac{\mathtt{M},\sigma,\mathtt{e} \hookrightarrow \mathtt{false} \qquad \mathtt{M},\sigma,\mathtt{e}_2 \hookrightarrow v}{\mathtt{M},\sigma,\mathtt{if\ e\ then\ e}_1\ \mathtt{else\ e}_2 \hookrightarrow v}\ \text{If\_False\_Val}$$

$$\frac{\mathtt{M},\sigma,\mathtt{e}_1 \hookrightarrow v \qquad \mathtt{M},\sigma,\mathtt{e}_2 \hookrightarrow v}{\mathtt{M},\sigma,\mathtt{e}_1 = \mathtt{e}_2 \hookrightarrow \mathtt{true}}\ \text{Equals\_True\_Val}$$

$$\frac{\mathtt{M},\sigma,\mathtt{e}_1 \hookrightarrow v \qquad \mathtt{M},\sigma,\mathtt{e}_2 \hookrightarrow v' \qquad v \neq v'}{\mathtt{M},\sigma,\mathtt{e}_1 = \mathtt{e}_2 \hookrightarrow \mathtt{false}}\ \text{Equals\_False\_Val}$$

Fig. 4. Value of Expressions

$\mathtt{M}\,;\mathtt{M}',\sigma \models \mathtt{e}$ does not hold, this does *not* imply that $\mathtt{M}\,;\mathtt{M}',\sigma \models \mathtt{e} = \mathtt{false}$ holds. Finally, an assertion of the form $\mathtt{e}_0 = \mathtt{e}_0$ does not always hold; for example, $\mathtt{M}_0\,;\mathtt{M}',\sigma_0 \models \mathtt{cyc.last} = \mathtt{cyc.last}$ 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* M, M′, *assertions A, A′, variables* x, y, S, *and configuration σ, we define*:

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

Growing configurations do not always preserve satisfaction. E.g, $\forall \mathtt{x}.[\ \mathtt{x} : \mathtt{Account} \rightarrow \mathtt{x.balance} > 100\ ]$ 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, following up with our earlier example, $\mathtt{M}_0\,;\mathtt{M}',\sigma_0 \models \neg(\mathtt{cyc.acyclic} = \mathtt{true})$ and

$\mathtt{M}_0 \fatsemi \mathtt{M}', \sigma_0 \models \neg(\mathtt{cyc.acyclic = false})$, and also $\mathtt{M}_0 \fatsemi \mathtt{M}', \sigma_0 \models \neg(\mathtt{cyc.last = cyc.last})$ hold.

5.4 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.

DEFINITION 7 (PERMISSION). *For any modules* M, M′, *variables* x *and* y, *we define*

- $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models \langle\, \mathtt{x}\ \mathsf{access}\ \mathtt{y} \,\rangle$ *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* 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* z *and* z *appears in* σ.contn.

In the last disjunct, where z is a parameter or local variable, we ask that z appears in the code being executed (σ.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 if stored in the continuation (cont) on the top frame.

DEFINITION 8 (CONTROL). *For any modules* M, M′, *variables* x , y, $\mathtt{z}_1, \ldots \mathtt{z}_n$, *we define*:

- $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models \langle\, \mathtt{x}\ \mathsf{calls}\ \mathtt{y.m}(\ \mathtt{z}_1, \ldots \mathtt{z}_n) \,\rangle$ *if* $\lfloor \mathtt{x} \rfloor_\sigma, \lfloor \mathtt{y} \rfloor_\sigma, \lfloor \mathtt{z}_1 \rfloor_\sigma, \ldots \lfloor \mathtt{z}_n \rfloor_\sigma$ *are defined, and*
  - $\lfloor \mathtt{this} \rfloor_\sigma = \lfloor \mathtt{x} \rfloor_\sigma$, *and*
  - σ.contn=$\mathtt{u.m}(\mathtt{v}_1, .. \mathtt{v}_n); \_$, *for some* $\mathtt{u}, \mathtt{v}_1, \ldots \mathtt{v}_n$, *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, $\langle\, \mathtt{x}\ \mathsf{calls}\ \mathtt{y.m}(\ \mathtt{z}_1, \ldots \mathtt{z}_n) \,\rangle$ expresses the call $\mathtt{y.m}(\mathtt{z}_1, \ldots \mathtt{z}_n)$ will be executed next, and that the caller is 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* M, M′, *and variable* x, *we define*

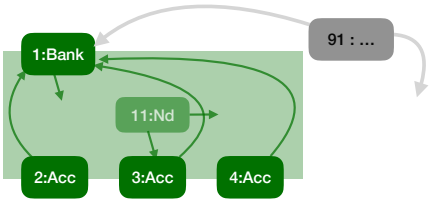
- $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models \mathsf{external}\langle\, \mathtt{x} \,\rangle$ *if* $\lfloor \mathtt{x} \rfloor_\sigma$ *is defined and* $Class(\lfloor \mathtt{x} \rfloor_\sigma)_\sigma \notin dom(\mathtt{M})$
- $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models \mathsf{internal}\langle\, \mathtt{x} \,\rangle$ *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 give set S. This way we can express that the objects from the set S have authority over the assertion *A*. In order to define validity of $\langle\, A\ \mathsf{in}\ \mathtt{S} \,\rangle$ in a configuration σ, we first define a restriction operation, $\sigma{\downarrow_{\mathtt{S}}}$ which restricts the objects from σ to only those from S.

DEFINITION 10 (RESTRICTION OF RUNTIME CONFIGURATIONS). *The restriction operator* $\downarrow$ *applied to a runtime configuration* σ *and a variable* S *is defined as follows:*

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

For example, if we take σ_2 from Fig. 2 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 [**?**]. 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 $\langle\, A \,\mathsf{in}\, \mathtt{S} \,\rangle$.

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

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

The set $\mathtt{S}$ in the assertion $\langle\, A \,\mathsf{in}\, \mathtt{S} \,\rangle$ is related to framing from implicit dynamic frames [**?**]: in an implicit dynamic frames assertion **acc** $\mathtt{x.f} * A$, the frame $\mathtt{x.f}$ prescrives 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.* $\mathtt{M}_{BA2} \fatsemi \mathtt{M}', \sigma \models \neg\, \langle\, (\exists n.\mathtt{a}_2.\mathtt{balance} = n) \,\mathsf{in}\, \mathtt{S}_4 \,\rangle$.

5.5 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.

We discuss challenge a) in some more detail: take an assertion will$\langle\, \mathtt{x.f} = 3 \,\rangle$; it is satisfied in the *current* configuration, σ, if in some *future* configuration, σ', the field $\mathtt{f}$ of the object that is pointed at by $\mathtt{x}$ in the *current* configuration (σ) has the value 3, even in that future configurartion $\mathtt{x}$denotes a different object ($\lfloor \mathtt{x} \rfloor_\sigma \neq \lfloor \mathtt{x} \rfloor_{\sigma'}$). To address this, we define an auxiliary concept: $\triangleleft$ the adaptation of one runtime configuration to the scope of another. $\sigma \triangleleft \sigma'$ adapts the second configuration to the top frame's view of the former: it returns a new configuration whose stack has the top frame as taken from σ and where the $\mathtt{contn}$ has been consistently renamed, while the heap is taken from σ'. This allows us to interpret expressions in σ' but with the variables bound according to σ; *e.g.* we can obtain that value of $\mathtt{x}$ in configuration σ' even if $\mathtt{x}$ was out of scope in σ'.

DEFINITION 12 (ADAPTATION). *For runtime configurations* σ, σ'.:

- $\sigma \triangleleft \sigma' \triangleq (\phi'' \cdot \psi', \chi')$ *if*
  - $\sigma = (\phi \cdot \_, \_)$, $\sigma' = (\phi' \cdot \psi', \chi')$, *and*
  - ϕ=$(\mathtt{contn}, \beta)$, ϕ'=$(\mathtt{contn}', \beta')$, $\phi'' = (\,\mathtt{contn}'[\mathtt{zs}/\mathtt{zs}'],\, \beta[\mathtt{zs}' \mapsto \beta'(\mathtt{zs})]\,)$, *where*
  - $\mathtt{zs}$=*dom*(β), $\mathtt{zs}'$ *is a set of variables with the same cardinality as* $\mathtt{zs}$, *and*

– *all variables in* $\mathtt{zs}'$ *are fresh in* β *and in* β'.

That is, in the new frame ϕ'' from above, we keep the same continuation as from σ' but rename all variables with fresh names $\mathtt{zs}'$, and combine the variable map β from σ with the variable map β' from σ' while avoiding names clashes through the renaming $[\mathtt{zs}' \mapsto \beta'(\mathtt{zs})]$. 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 $\mathsf{will}\langle\, \mathtt{x.f} = 3 \wedge \mathsf{will}\langle\, \mathtt{x.f} = 5 \,\rangle\,\rangle$.

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 `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* $\mathtt{M}$, $\mathtt{M}'$, *and assertion* A *we define*

- $\mathtt{M} \S \mathtt{M}', \sigma \models \mathsf{next}\langle\, A \,\rangle$ *if* $\exists \sigma'.[\ \mathtt{M} \S \mathtt{M}', \phi \rightsquigarrow \sigma' \ \wedge \mathtt{M} \S \mathtt{M}', \sigma \triangleleft \sigma' \models A\ ]$, *and where* ϕ *is so that* $\sigma = (\phi \cdot \_, \_)$.
- $\mathtt{M} \S \mathtt{M}', \sigma \models \mathsf{will}\langle\, A \,\rangle$ *if* $\exists \sigma'.[\ \mathtt{M} \S \mathtt{M}', \phi \rightsquigarrow^* \sigma' \ \wedge \mathtt{M} \S \mathtt{M}', \sigma \triangleleft \sigma' \models A\ ]$, *and where* ϕ *is so that* $\sigma = (\phi \cdot \_, \_)$.
- $\mathtt{M} \S \mathtt{M}', \sigma \models \mathsf{prev}\langle\, A \,\rangle$ *if* $\forall \sigma_1, \sigma_2.[\ \mathit{Initial}\langle \sigma_1 \rangle \ \wedge \ \mathtt{M} \S \mathtt{M}', \sigma \rightsquigarrow^* \sigma_2 \ \wedge \ \mathtt{M} \S \mathtt{M}', \sigma_2 \rightsquigarrow \sigma \longrightarrow \mathtt{M} \S \mathtt{M}', \sigma \triangleleft \sigma_2 \models A\ ]$
- $\mathtt{M} \S \mathtt{M}', \sigma \models \mathsf{was}\langle\, A \,\rangle$ *if* $\forall \sigma_1, \ldots \sigma_n.[\ \mathit{Initial}\langle \sigma_1 \rangle \ \wedge \ \sigma_n = \sigma \ \wedge \ \forall i \in [1..n).\mathtt{M} \S \mathtt{M}', \sigma_i \rightsquigarrow \sigma_{i+1} \longrightarrow \exists j \in [1..n-1).\mathtt{M} \S \mathtt{M}', \sigma \triangleleft \sigma_j \models A\ ]$

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

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

5.6 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.*

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

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 $\mathtt{M} \S \mathtt{M}', \sigma \models A$ and $\mathtt{M} \S \mathtt{M}', \sigma \models A \rightarrow A'$ implies $\mathtt{M} \S \mathtt{M}', \sigma \models A'$. 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 C.

5.7 Modules satisfying assertions

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

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

- $\mathtt{M} \models A$ *if* $\forall \mathtt{M}'.\, \forall \sigma \in \mathcal{A}\mathit{rising}(\mathtt{M} \S \mathtt{M}').\, \mathtt{M} \S \mathtt{M}', \sigma \models A$

6 EXAMPLES

In this section we look at a couple of further examples from the literature where a holistic specification would provide obvious benefits in ensuring robustness.

6.1 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 [**?**].

Specifications for attenuation for the DOM were proposed in [**?**]. In this section we revisit that example and also argue that compared with the specification in [**?**], our specification xxxx.

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 information, such as xxxx, 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 `Wtrapper`. It is not difficult to write such a `Wrapper`; a possible implementation appears in Figure **??** in appendix **??**.

In Figure 5 we show an example of the use of `Wrapper` objects to attenuate the use of `Node`s[3] 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"
   ...
}
```

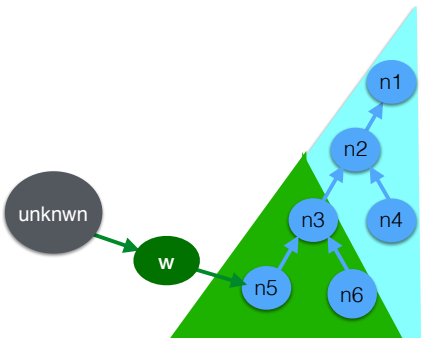


Fig. 5. `Wrapper`s protecting `Node`s

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

[3] SD: Is that a good sentence?

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.* [**?**] (*c.f.* appendix **??**) 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*. [4]

Instead, in this work, we propose *holistic specifications*, which describe the behaviour of an abstract data type as a hole, taking all possible method calls into account. Such holistic specifications typically describe *necessary conditions* for some effect to take place. 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:

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

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, $\mathtt{parnt}^{k} = \mathtt{w.node.parnt}^{\mathtt{w.height}}$). Definitions of these concepts appear later (Definition **??**), but 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.

In the next sections we proceed with a formal model of our model. In the appendix we discuss more – and simpler – examples. We chose the DOM for the introduction, in order to give a flavour of the *Chainmail* features.

6.2 Authorizing the ERC20

ERC20 [**?**] 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.

[4] Shell we say the following, or does it break the flow?
Moreover, on line 10 we do not know which functions are called on `w`.

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.* $\langle$ p calls ....`transfer`(...) $\rangle$), or by an authorized transfer instigated by another participant p″ (*i.e.* $\langle$ p″ calls ...`transferFrom`(..) $\rangle$) who has authority for more than the tokens spent (*i.e.* `e.allowed`(p, p″) ≥ m)

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

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 `e.allowed`(p, 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[**?**].

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

$$
\begin{array}{l}
\forall \mathtt{e} : \mathtt{ERC20}.\forall \mathtt{p}, \mathtt{p}' : \mathtt{Object}.\forall \mathtt{m} : \mathtt{Nat}. \\
\quad [\ \mathtt{e.allowed(p, p')} = \mathtt{m} \\
\qquad \longrightarrow \\
\qquad \mathcal{P}\mathit{rev}\langle\ \langle\, \mathtt{p}\ \mathsf{calls}\ \mathtt{e.approve(p', m)} \,\rangle \\
\qquad\qquad \vee \\
\qquad\qquad \mathtt{e.allowed(p, p')} = \mathtt{m}\ \wedge \\
\qquad\qquad\ \neg(\langle\, \mathtt{p'}\ \mathsf{calls}\ \mathtt{e.transferFrom(p, \_)} \,\rangle\ \vee\ \langle\, \mathtt{p}\ \mathsf{calls}\ \mathtt{e.approve(p, \_)} \,\rangle) \\
\qquad\qquad \vee \\
\qquad\qquad \exists \mathtt{p''} : \mathtt{Object}.\exists \mathtt{m}' : \mathtt{Nat}. \\
\qquad\qquad [\, \mathtt{e.allowed(p, p')} = \mathtt{m} + \mathtt{m}'\ \wedge\ \langle\, \mathtt{p'}\ \mathsf{calls}\ \mathtt{e.transferFrom(p'', m')} \,\rangle] \\
\qquad \rangle \\
\quad]
\end{array}
$$

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 D), 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.

6.3 Defending the DAO

We specidy the DAO in appendix E

7 DISCUSSION

Design choices. For our underlying language, we have chosen an object-oriented, class based language, but we expect that the ideas will also be applicable to other kinds of languages (object-oriented or otherwise). We could, *e.g.* extend our work to prototype-based programming by creating an (anonymous) class to reify each prototype [**?**].

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 [**?**].

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.

Necessary v.s. sufficient conditions. Are the necessary conditions the same as the complement of all the sufficient conditions? The possible state transitions of a component are described by the transitive closure of the individual transitions. Taking Figure **??** part (a), if we calculate the complement of the transitive closure of the transitions, then we could, presumably, obtain all transitions which will not happen, and if we apply this relation to the set of initial states, we obtain all states guaranteed not to be reached. Thus, using the sufficient conditions, we obtain implicitly more fine-grained information than that available through the yellow transitions and yellow boxes in Figure **??** part (b).

This implicit approach is mathematically sound. But it is impractical, brittle with regards to software maintenance, and weak with regards to reasoning in the open world.

The implicit approach is impractical: it suggests that when interested in a necessity guarantee a programmer would need to read the specifications of all the functions in that module, and think about all possible sequences of such functions and all interleavings with other modules. What if the bank did indeed enforce that only the account owner may withdraw funds, but had another function which allowed the manager to appoint an account supervisor, and another which allowed the account supervisor to assign owners?

The implicit approach is also brittle with regards to software maintenance: it gives no guidance to the team maintaining a piece of software. If the necessary conditions are only implicit in the sufficient conditions, then developers' intentions about those conditions are not represented anywhere: there can be no distinction between a condition that is accidental (if `notify`-ing is not implemented, then

it cannot be permitted) and one that is essential (money can only be transferred by account owners). Subsequent developers may inadvertently add functions which break these intentions, without even knowing they've done so.

Finally, the implicit approach is weak when it comes to reasoning about programs in an open world: it does not give any guarantees about objects when they are passed as arguments to calls into unknown code. For example, what guarantees can we make about the top of the DOM tree when we pass a wrapper pointing to lower parts of the tree to an unknown advertiser?

8 RELATED WORK

Behavioural Specification Languages. Hatcliff et al. [**?**] provide an excellent survey of contemporary specification approaches. With a lineage back to Hoare logic [**?**], Meyer's Design by Contract [**?**] 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 [**?**], Spec♯ [**?**], Dafny [**?**] and Whiley [**?**]. Our approach builds upon these fundamentals, particularly Leino & Shulte's formulation of two-state invariants [**?**], and Summers and Drossopoulou's Considerate Reasoning [**?**]. In general, these approaches assume a closed system, where modules can be trusted to cooperate. In this paper we aim to illustrate the kinds of techniques required in an open system where modules' invariants must be protected irrespective of the behaviour of the rest of the system.

Chainmail assertions are guarantees upheld throughout program execution. Other systems which give such "permanent" guarantees are type systems, which ensure that well-formed programs always produce well-formed runtime configurations, or information flow control systems [**?**], which ensure that values classified as high will not be passed into contexts classified as low. Such guarantees are practical to check, but too coarse grained for the purpose of fine-grained, module-specific specifications.

Chainmail specifications can cross-cut the code they are specifying; therefore, they are related to aspect-oriented specification languages such as AspectJML [**?**] and AspectLTL [**?**]. AspectJML is an aspect-oriented extension to JML; in much the same way that AspectJ is an aspect-oriented extension to Java [**?**]. AspectJML offers AspectJ-style pointcuts that allow the definition of crosscutting specifications, such as shared pre- or post-conditions for a range of method calls.

AspectLTL [**?**] is a specification language based on Linear Temporal Logic (LTL). It adds cross-cutting aspects to more traditional LTL module specifications: these aspects can further constrain specifications in modules. In that sense, AspectLTL and *Chainmail* use similar implicit join point models, rather than importing AspectJ style explicit pointcuts as in AspectJML.

Our work is also related to the causal obligations in Helm et al.'s behavioural contracts [**?**]. Causal obligations allow programmers to specify e.g. that whenever one object receives a message (such as a subject in the Observer pattern having its value changed) that object must send particular messages off to other objects (e.g. the subject must notify its observers). *Chainmail*'s control operator supports similar specifications, (e.g. ⟨ _ calls `s.setValue( v)` ⟩ → will⟨ ⟨ s calls `s.observer.notify( v)` ⟩⟩ — when a subject receives a `setValue` method, it must "forward" those messages to the observer.

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"* [**?**]. Miller [**??**] 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 have the great advantage that they make it much easier to make guarantees about systems composed of multiple components [**?**].

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 [**?**], and were adapted to the programming languages setting in the 70's [**?**]. *Object capabilities* were first introduced [**?**] in the early 2000s, and many recent studies manage to verify safety or correctness of object capability programs. Google's Caja [**?**] applies sandboxes, proxies, and wrappers to limit components' access to *ambient* authority. Sandboxing has been validated formally: Maffeis et al. [**?**] 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 and web systems [**???**] including Newspeak [**?**], Dart [**?**], Grace [**??**] and Wyvern [**?**] have adopted the object capability model.

Verification of Dynamic Languages. A few formal verification frameworks address JavaScript's highly dynamic, prototype-based semantics. Gardner et al. [**?**] developed a formalisation of JavaScript based on separation logic and verified examples. Xiong and Qin et al. [**??**] worked on similar lines. Swamy et al. [**?**] recently developed a mechanised verification technique for JavaScript based on the Dijkstra Monad in the F* programming language. Finally, Jang et al. [**?**] developed a machine-checked proof of five important properties of a web browser — again similar to our invariants — such as *"cookies may not be shared across domains"* by writing the minimal kernel of the browser in Haskell.

JavaScript analyses. More practically, Karim et al. apply static analysis on Mozilla's JavaScript Jetpack extension framework [**?**], including pointer analyses. Bhargavan et al. [**?**] extend language-based sandboxing techniques to support defensive components that can execute successfully in otherwise untrusted environments. Politz et al. [**?**] use a JavaScript type checker to check properties such as *"multiple widgets on the same page cannot communicate."* Lerner et al. extend this system to ensure browser extensions observe *"private mode"* browsing conventions, such as that *"no private browsing history retained"* [**?**]. Dimoulas et al. [**?**] generalise the language and type checker based approach to enforce explicit policies, that describe which components may access, or may influence the use of, particular capabilities. Alternatively, Taly et al. [**?**] model JavaScript APIs in Datalog, and then carry out a Datalog search for an "attacker" from the set of all valid API calls.

Verification of Object Capability Programs. Murray made the first attempt to formalise defensive consistency and correctness [**?**]. Murray's model was rooted in counterfactual causation [**?**]: 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 [**?**], 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 [**??**] have analysed Miller's Mint and Purse example [**?**] and discussed the six capability policies as proposed in [**?**]. In [**?**], they sketched a specification language, used it to specify the six policies from [**?**], 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 [**?**]. Their work does not support the concepts of control, time, or space, as in *Chainmail*, but it offers a primitive expressing trust.

? 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. Devrise have demonstrated solutions to a range of exemplar problems, including the DOM wrapper (replicated in our section 6.1) and a mashup application. Their distinction between public and private transitions is similar to the distinction between internal and external objects.

More recently, **?** 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 [**?**]) 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.

? 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 [**??**], which also protect internal implementation objects from accesses that cross encapsulation boundaries. Banerjee and Naumann demonstrated that these systems enforce representation independence (a property close to "robust safety") some time ago [**?**].

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 external⟨ ⟩ predicate. They use powerful mathematical techniques which the users need to understand in order to write their specifications, while the *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.

9 CONCLUSION

In this paper we have motivated the need for holistic specifications, presented the *Chainmail* specification language for writing such specifications, and shown how *Chainmail* can be used to give holistic specifications of key exemplar problems: the bank account, the wrapped DOM, the ERC20, and and the DAO.

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.

The development of the semantics of *Chainmail* assertions posed several interesting challenges, *e.g.* the treatment of the open world requires two-module execution and the concept of external objects, recursion is confined to ghostfields and assertions require termination of included expressions, space required the concept of restricting runtime configurations, and time required adaptation operators which apply bindings from one configuration to another.

Chainmail is powerful enough to express many key examples from the literature; nevertheless, it lacks several important features: It provides recursion only in a restricted form, it has a rather inflexible notion of module and does not support hierarchies of modules, and knows nothing about concurrency or distribution. We plan to remove these restrictions by applying techniques such as step-indexing [**?**], but hope to keep any mathematical sophitsication in the model of *Chainmail* without exposing it to the person who writes the specification. We are also interested in extending *Chainmail* to situations where internal modules are typed, but the external modules are untyped.

We also plan to extend *Chainmail* to support reasoning about conditional trust in programs, and to quantify the risks involved in interacting with untrustworthy software [**?**].

To make these kinds of specifications practically useful, we plan to develop logics for proving adherence of module's code to holistic specs, as well as logics for using holistic specs in the proof of open programs. We want to develop dynamic monitoring and model checking techniques for our specifications. And finally, we plan to automate reasoning with these logics.

A THE UNDERLYING PROGRAMMING LANGUAGE, $\mathcal{L}_{oo}$

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):*

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

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. 6. 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 5.2. Note that the expressions that make up the bodies of ghostfield declarations (`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, and ghostfield declarations.*

ClassDescr	::=	class *ClassId* { (*FieldDecl*)* (*MethDecl*)* (*GhosDecl*)* }
FieldDecl	::=	field `f`
MethDecl	::=	method `m`(`x`*) { *Stmts* }
Stmts	::=	*Stmt* | *Stmt* ; *Stmts*
Stmt	::=	`x.f:=x` | `x:=x.f` | `x:=x.m(x*)` | `x:=` new `C(x*)` | return `x`
GhostDecl	::=	ghost `f(x*)` { `e` }
`e`	::=	true | false | null | `x` | `e=e` | if `e` then `e` else `e` | `e.f(e*)`
`x`, `f`, `m`	::=	`Identifier`

where we use metavariables as follows: `x` ∈ *VarId* `f` ∈ *FldId* `m` ∈ *MethId* `C` ∈ *ClassId, and* `x` *includes* `this`

We define a method lookup function, $\mathcal{M}$ which returns the corresponding method definition given a class `C` and a method identifier `m`, and similarly a ghostfield lookup function, $\mathcal{G}$

DEFINITION 17 (LOOKUP). *For a class identifier* C *and a method identifier* m:

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

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

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* $\alpha \in$ Addr *to indicate an address.*
- *Values,* v*, are either addresses, or sets of addresses or null:* $v \in \{\mathtt{null}\} \cup \mathtt{Addr} \cup \mathcal{P}(\mathtt{Addr})$.
- *Continuations are either statements (as defined in Definition 16) or a marker,* $\mathtt{x} := \bullet$*, for a nested call followed by statements to be executed once the call returns.*
  Continuation ::= *Stmts* | $\mathtt{x} := \bullet$; *Stmts*
- *Frames,* ϕ*, consist of a code stub and a mapping from identifiers to values:* $\phi \in \mathit{CodeStub} \times \mathtt{Ident} \rightarrow \mathit{Value}$,
- *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:* $\mathit{Object} = \mathtt{ClassID} \times (\mathtt{FieldId} \rightarrow \mathit{Value})$.
- *Heaps,* χ*, are mappings from addresses to objects:* $\chi \in \mathtt{Addr} \rightarrow \mathit{Object}$.
- *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 Definition 11:

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

- $\chi(\alpha, \mathtt{f}) \triangleq \mathit{fldMap}(\alpha, \mathtt{f})$ *if* $\chi(\alpha) = (\_, \mathit{fldMap})$.
- $\mathit{Class}(\alpha)_\chi \triangleq \mathtt{C}$ *if* $\chi(\alpha) = (\mathtt{C}, \_)$

We now define interpretations as follows:

- $\lfloor \mathtt{x} \rfloor_\phi \triangleq \phi(\mathtt{x})$
- $\lfloor \mathtt{x.f} \rfloor_{(\phi,\chi)} \triangleq v$, *if* $\chi(\phi(\mathtt{x})) = (\_, \mathit{fldMap})$ *and* $\mathit{fldMap}(\mathtt{f}) = v$

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

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

methCall_OS

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

varAssgn_OS

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

fieldAssgn_OS

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

objCreate_OS

$$\frac{\begin{array}{l}\phi.\texttt{contn} = \texttt{x:=new C(}\ \texttt{x}_1,\ldots\texttt{x}_n\texttt{)}\ \texttt{;}\ \texttt{Stmts} \\ \alpha \text{ new in } \chi \\ \texttt{f}_1,..\texttt{f}_n \text{ are the fields declared in } \texttt{M(C)}\end{array}}{\texttt{M},\ (\ \phi \cdot \psi, \chi\ ) \rightsquigarrow (\ \phi[\texttt{contn} \mapsto \texttt{Stmts}, \texttt{x} \mapsto \alpha\ ] \cdot \psi,\ \chi[\alpha \mapsto (\texttt{C}, \texttt{f}_1 \mapsto \lfloor \texttt{x}_1 \rfloor_\phi, \ldots \texttt{f}_n \mapsto \lfloor \texttt{x}_n \rfloor_\phi)]\ )}$$

return_OS

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

Fig. 6. Operational Semantics

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

Execution of a statement has the form $\texttt{M}, \sigma \rightsquigarrow \sigma'$, and is defined in figure 6.

DEFINITION 21 (EXECUTION). *of one or more steps is defined as follows:*

- *The relation* $\texttt{M}, \sigma \rightsquigarrow \sigma'$*, it is defined in Figure 6.*
- $\texttt{M}, \sigma \rightsquigarrow^* \sigma'$ *holds, if i)* σ=σ'*, or ii) there exists a* σ'' *such that* $\texttt{M}, \sigma \rightsquigarrow^* \sigma''$ *and* $\texttt{M}, \sigma'' \rightsquigarrow \sigma'$.

A.3 Definedness of execution, and extending configurations

Note that interpretations and executions need not always be defined. For example, in a configuration whose top frame does not contain $\mathtt{x}$ in its domain, $\lfloor \mathtt{x} \rfloor_\phi$ is undefined. We define the relation $\sigma \sqsubseteq \sigma'$ to express that σ has more information than σ', and then prove that more defined configurations preserve interpretations:

DEFINITION 22 (EXTENDING RUNTIME CONFIGURATIONS). *The relation $\sqsubseteq$ is defined on runtime configurations as follows. Take arbitrary configurations σ, σ', σ'', frame ϕ, stacks ψ, ψ', heap χ, address α free in χ, value v and object o, and define $\sigma \sqsubseteq \sigma'$ as the smallest relation such that:*

- $\sigma \sqsubseteq \sigma$
- $(\phi[\mathtt{x} \mapsto v] \cdot \psi, \chi) \sqsubseteq (\phi \cdot \psi, \chi)$
- $(\phi \cdot \psi \cdot \psi', \chi) \sqsubseteq (\phi \cdot \psi, \chi)$
- $(\phi, \chi[\alpha \mapsto o) \sqsubseteq (\phi \cdot \psi, \chi)$
- *$\sigma' \sqsubseteq \sigma''$ and $\sigma'' \sqsubseteq \sigma$ imply $\sigma' \sqsubseteq \sigma$*

LEMMA A.1 (PRESERVATION OF INTERPRETATIONS AND EXECUTIONS). *If $\sigma' \sqsubseteq \sigma$, then*

- *If $\lfloor \mathtt{x} \rfloor_\sigma$ is defined, then $\lfloor \mathtt{x} \rfloor_{\sigma'} = \lfloor \mathtt{x} \rfloor_\sigma$.*
- *If $\lfloor \mathtt{this.f} \rfloor_\sigma$ is defined, then $\lfloor \mathtt{this.f} \rfloor_{\sigma'} = \lfloor \mathtt{this.f} \rfloor_\sigma$.*
- *If $Class(\alpha)_\sigma$ is defined, then $Class(\alpha)_{\sigma'} = Class(\alpha)_\sigma$.*
- *If $\mathtt{M}, \sigma \rightsquigarrow^* \sigma''$, then there exists a σ'', so that $\mathtt{M}, \sigma' \rightsquigarrow^* \sigma'''$ and $\sigma''' \sqsubseteq \sigma''$.*

Note however, that such preservation does not hold for assertion. For example, if $\sigma' \sqsubseteq \sigma$, then $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models \forall x.A$ and does not imply $\mathtt{M} \fatsemi \mathtt{M}', \sigma' \models \forall x.A$,

SUSAN EISENBACH, Imperial College London

on the other hand, $\mathtt{M} \fatsemi \mathtt{M}', \sigma' \models \exists x.A$ does not imply $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models \exists x.A$

A.4 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 23 (MODULE LINKING). *The linking operator $\circ$: $Module \times Module \longrightarrow Module$ is defined as follows:*

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

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.*

The lemma below says that linking is associative and commutative, and preserves execution.

LEMMA A.2 (PROPERTIES OF LINKING – REPETITION OF LEMMA 4.1 IN THE MAIN TEXT). *For any modules $\mathtt{M}$, $\mathtt{M}'$ and $\mathtt{M}''$, and runtime configurations σ, and σ' we have*:

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

PROOF. (1) and (2) follow from Definition 23. (3) follows from 23, 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. then □

A.5 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 $\mathtt{M}'$.

Therefore, we define execution in terms of an internal module $\mathtt{M}$ and an external module $\mathtt{M}'$, through the judgment $\mathtt{M} \fatsemi \mathtt{M}', \sigma \rightsquigarrow \sigma'$, which mandates that σ and σ' are external to $\mathtt{M}$, and that there exists an execution which leads from σ to σ' which leads through intermediate configurations σ_2, ... σ_{n+1} which are all internal to $\mathtt{M}$, and thus unobservable from the client. In a sense, we "pretend" that all calls to functions from $\mathtt{M}$ are executed atomically, even if they involve several intermediate, internal steps.

DEFINITION 24 (REPEATING 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:*

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

In the definition above 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 will$\langle _ \rangle$ and was$\langle _ \rangle$ connectives), and thus, the intention of ignoring some intermediate configurations can only be achieved if we refine the concept of execution.

The following lemma states that linking external modules preserves execution

LEMMA A.3 (LINKING MODULES PRESERVES EXECUTION). *For any modules* M, M′, *and* M″, *whose domains are pairwise disjoint, and runtime configurations* σ, σ',

- $\mathtt{M} \fatsemi \mathtt{M}', \sigma \rightsquigarrow \sigma'$ *implies* $\mathtt{M} \fatsemi (\mathtt{M}' \circ \mathtt{M}''), \sigma \rightsquigarrow \sigma'$.
- $\mathtt{M} \fatsemi \mathtt{M}', \sigma \rightsquigarrow \sigma'$ *implies* $(\mathtt{M} \circ \mathtt{M}'') \fatsemi \mathtt{M}', \sigma \rightsquigarrow \sigma'$.

PROOF. For the second guarantee we use the fact that $\mathtt{M} \fatsemi \mathtt{M}', \sigma \rightsquigarrow \sigma'$ implies that all intermediate configurations are internal to M and thus also to $\mathtt{M} \circ \mathtt{M}''$. □

We can now answer the question as to which runtime configurations are pertinent when judging a module's adherence to an assertion. First, where does execution start? We define *initial* configurations to be those which may contain arbitrary code stubs, but which contain no objects. Objects will be created, and further methods will be called through execution of the code in $\phi.\mathtt{contn}$. From such initial configurations, executions of code from $\mathtt{M} \fatsemi \mathtt{M}'$ creates a set of *arising* configurations, which, as we will see in Definition 14, are pertinent when judging M's adherence to assertions.

DEFINITION 25 (INITIAL AND ARISING CONFIGURATIONS – REPEATING DEFINITION 2). *are defined as follows:*

- $\mathcal{I}nitial\langle(\psi, \chi)\rangle$, *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* $\chi(\alpha) = (\mathtt{Object}, \emptyset)$.
- $\mathcal{A}rising(\mathtt{M} \fatsemi \mathtt{M}') = \{ \sigma \mid \exists \sigma_0. [\, \mathcal{I}nitial\langle\sigma_0\rangle \ \wedge \ \mathtt{M} \fatsemi \mathtt{M}', \sigma_0 \rightsquigarrow^* \sigma \,] \}$

Note that there are infinitely many different initial configurations, they will be differing in the code stored in the continuation of the unioque frame.

B THE BANK/ACCOUNT EXAMPLE – FULL CODE

In this section we revisit the Bank/Account example from section 2, and show two different implementations, derived from **?** . 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. 2, while the second version gives rise to runtime configurations as σ_2, shown on the right side of Fig. 2.

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.

$\mathtt{M}_{BA1}$, the fist version is shown Fig. 7. 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 Accounts. In this version we have no ghost fields.

$\mathtt{M}_{BA1}$, the second version is shown Fig. 9 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

```
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 makeNewAccount(amt){
       if (amt>=0 && this.balance>=amt) then{
           this.balance = this.balance - amt;
           return new Account(this.myBank,amt)
   }    }
}
```

Fig. 7. M_{BA1}: Implementation of `Bank` and `Account` – version 1

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

C ASSERTION ENTAILMENT AND ASSERTIONS CLASSICAL

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

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

    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. 8. M_{BA2}: Implementation of `Bank` – version 2

assertion entails another assertion, iff all configurations which satisfy the former also satisfy the latter.

DEFINITION 26 (EQUIVALENCE AND ENTAILMENTS OF ASSERTIONS).

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

LEMMA C.1 (ASSERTIONS ARE CLASSICAL-1). *For all runtime configurations σ, assertions A and A', and modules* `M` *and* `M'`*, we have*

(1) $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models A$ *or* $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models \neg A$
(2) $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models A \wedge A'$ *if and only if* $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models A$ *and* $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models A'$
(3) $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models A \vee A'$ *if and only if* $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models A$ *or* $\sigma \models A'$
(4) $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models A \wedge \neg A$ *never holds.*
(5) $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models A$ *and* $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models A \rightarrow A'$ *implies* $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models A'$.

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

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

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

For parts (2)-(5) the proof goes by application of the corresponding definitions from 6. □

.

LEMMA C.2 (ASSERTIONS ARE CLASSICAL-2). *For assertions A, A', and A'' the following equivalences hold*

(1) $A \wedge \neg A \equiv$ `false`

```
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.
                                                      find(acc) )
    }
}
```

Fig. 9. M_{BA2}: Implementation of `Node` – version 2

```
class Account{

    field myBank

    method deposit{src,amt){
             this.myBank.deposit(this,src,amt)
    }   }
    method makeNewAccount(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. 10. M_{BA2}: Implementation of `Account` – version 2

(2) $A \vee \neg A \equiv \texttt{true}$
(3) $A \wedge A' \equiv A' \wedge A$

(4) $A \vee A' \equiv A' \vee A$
(5) $(A \vee A') \vee A'' \equiv A \vee (A' \vee A'')$
(6) $(A \vee A') \wedge A'' \equiv (A \wedge A') \vee (A \wedge A'')$
(7) $(A \wedge A') \vee A'' \equiv (A \vee A') \wedge (A \vee A'')$
(8) $\neg(A \wedge A') \equiv \neg A \vee \neg A''$
(9) $\neg(A \vee A') \equiv \neg A \wedge \neg A'$
(10) $\neg(\exists \mathtt{x}.A) \equiv \forall \mathtt{x}.(\neg A)$
(11) $\neg(\exists \mathtt{S} : \mathtt{SET}.A) \equiv \forall \mathtt{S} : \mathtt{SET}.(\neg A)$
(12) $\neg(\forall \mathtt{x}.A) \equiv \exists \mathtt{x}.\neg(A)$
(13) $\neg(\forall \mathtt{S} : \mathtt{SET}.A) \equiv \exists \mathtt{S} : \mathtt{SET}.\neg(A)$

PROOF. All points follow by application of the corresponding definitions from 6. □

D EXAMPLE – ERC20, THE TRADITIONAL SPECIFICATION

We compare the holistic and the traditional specification of ERC20

As we said earlier, 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 Figure 11 we outline a traditional specification for the `ERC20`. We give two speficiations 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 authorized and executes `transferFrom`, then the balance decreases. But they are *implicit* about the overall behaviour and the *necessary* conditions, e.g., what are all the possible actions that can cause a decrease of balance?

E DEFENDING THE DAO

The DAO (Decentralised Autonomous Organisation) [**?**] 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 [**?**]. With holistic specifications we can write a succinct requirement that a DAO contract should always be able to

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

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

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

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

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

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

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

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

Fig. 11. Classical specification for the `ERC20`

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 participants's addresses to numbers. Such mapping-valued fields exist in Solidity, but they could also be taken to be ghost fields [**?**].

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

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

$$\forall \mathtt{d} : \mathtt{DAO}.\forall \mathtt{p} : \mathtt{Any}.\forall \mathtt{m} : \mathtt{Nat}.$$
$$[\ \mathtt{d.balance(p)} = \mathtt{m} \wedge \langle\, \mathtt{p}\ \mathrm{calls}\ \mathtt{repay.d(\_)}\ \rangle \longrightarrow\ \mathrm{will}\langle\, \langle\, \mathtt{d}\ \mathrm{calls}\ \mathtt{send.p(m)}\ \rangle\, \rangle\ ]$$

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

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

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.