

# Holistic Specifications for Robust Programs

ANONYMOUS AUTHOR(S)

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.

## 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 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.

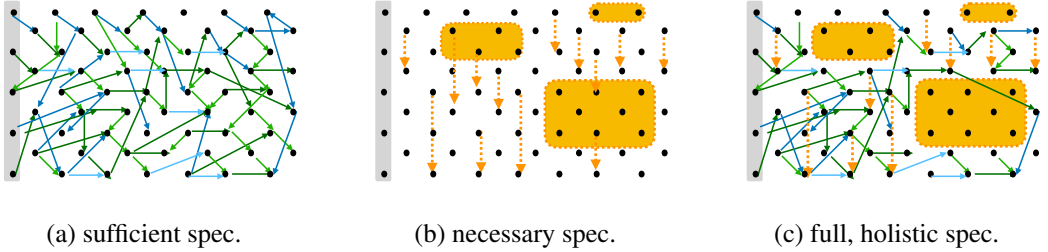


Fig. 1. Sufficient Conditions, Necessary Conditions, and Full Specifications

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 ?? 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 ?? motivates our work via an example, and then section ?? presents the *Chainmail* specification language. Section ?? introduces the formal model underlying *Chainmail*, and then section ?? defines the semantics of *Chainmail*'s assertions. Section ?? shows how key points of exemplar problems can be specified in *Chainmail*, section ?? discusses our design, ?? considers related work, and section ?? concludes.

## 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 ??.

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.<sup>1</sup>

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

<sup>1</sup>Note that our very limited bank specification doesn't even have the concept of an account owner.

```

148  method deposit(src, amt)
149  PRE:  this,src:Account  $\wedge$  this $\neq$ src  $\wedge$  this.myBank=src.myBank  $\wedge$ 
150      amt: $\mathbb{N}$   $\wedge$  src.balance $\geq$ amt
151  POST: src.balance=src.balancepre-amt  $\wedge$  this.balance=this.balancepre+amt
152
153  method makeNewAccount(amt)
154  PRE:  this:Account  $\wedge$  amt: $\mathbb{N}$   $\wedge$  this.balance $\geq$ amt
155  POST: this.balance=this.balancepre-amt  $\wedge$  fresh result  $\wedge$ 
156      result: Account  $\wedge$  this.myBank=result.myBank  $\wedge$  result.balance=amt
157
158  method newAccount(amt)
159  PRE:  this:Bank
160  POST: result: Account  $\wedge$  result.myBank=this  $\wedge$  result.balance=amt

```

Fig. 2. Functional specification of Bank and Account

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).

```

163  assume acm_acc,auth_acc: Account  $\wedge$  acm_acc.balance=10000  $\wedge$  auth_acc.balance=1500
164  acm_acc.deposit(auth_acc,1000)
165  assert acm_acc.balance=11000  $\wedge$  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. ??, 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. ?? 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. ??, and to provide additional specifications that capture those assumptions. The following three informal requirements prevent methods like `steal`:

- (1) An account's balance can be changed only if a client calls the `deposit` method with the account as the receiver or as an argument.
- (2) An account's balance can be changed only if a client has access to that particular account.

(3) The `Bank/Account` component does not leak access to existing accounts or banks.

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

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

- $$\begin{aligned}
 (1) &\triangleq \forall a.[ a : \text{Account} \wedge \text{changes}\langle a.\text{balance} \rangle \rightarrow \\
 &\quad \exists o.[ \langle o \text{ calls } a.\text{deposit}(\_) \rangle \vee \langle o \text{ calls } \_.\text{deposit}(a) \rangle ] ] \\
 (2) &\triangleq \forall a.\forall S : \text{Set}. [ a : \text{Account} \wedge \langle \text{will}\langle \text{changes}\langle a.\text{balance} \rangle \rangle \text{ in } S \rangle \rightarrow \\
 &\quad \exists o. [ o \in S \wedge \text{external}\langle o \rangle \wedge \langle o \text{ access } a \rangle ] ] \\
 (3) &\triangleq \forall a.\forall S : \text{Set}. [ a : \text{Account} \wedge \langle \text{will}\langle \exists o. [ \text{external}\langle o \rangle \wedge \langle o \text{ access } a \rangle ] \rangle \text{ in } S \rangle \\
 &\quad \rightarrow \exists o'. [ o' \in S \wedge \text{external}\langle o' \rangle \wedge \langle o' \text{ access } a \rangle ] ]
 \end{aligned}$$

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

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

It remains to think about how access to an `Account` may be obtained. This is the remit of assertion (3): It says that if at some time in the future of the state restricted to  $S$ , some object  $o$  which is external has access to some account  $a$ , and if  $a$  exists in the current state, then in the current state some object from  $S$  has access to  $a$ . Note that  $o$  and  $o'$  may but need not be the same object. Note also that  $o'$  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. ?? 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 ??.

*Example Configurations* We will illustrate these features using the Bank/Account example from the previous Section. We use the runtime configurations  $\sigma_1$  and  $\sigma_2$  shown in the left and right diagrams in Figure ?. 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  $\sigma_1$  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 ?? Fig. ?. Configuration  $\sigma_2$  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 ?? Figs. ?? and ?.

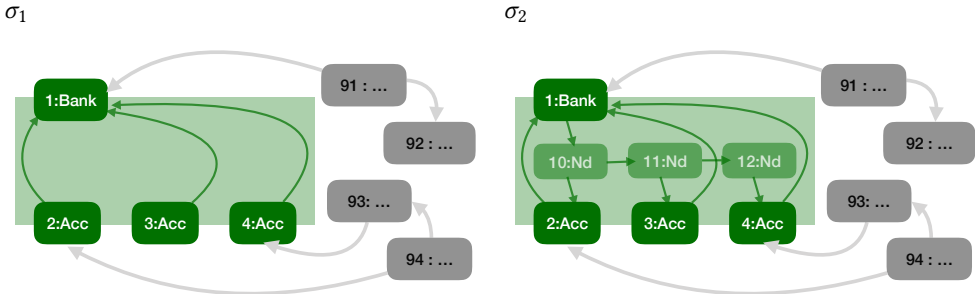


Fig. 3. Two runtime configurations for the Bank/Account example.

For the rest, assume variable identifiers  $b_1$ , and  $a_2$ – $a_4$ , and  $u_{91}$ – $u_{94}$  denoting objects 1, 2–4, and 91–94 respectively for both  $\sigma_1$  and  $\sigma_2$ . That is, for  $i=1$  or  $i=2$ ,  $\sigma_i(b_1)=1$ ,  $\sigma_i(a_2)=2$ ,  $\sigma_i(a_3)=3$ ,  $\sigma_i(a_4)=4$ ,  $\sigma_i(u_{91})=91$ ,  $\sigma_i(u_{92})=92$ ,  $\sigma_i(u_{93})=93$ , and  $\sigma_i(u_{94})=94$ .



*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  $a_2.\text{myBank} = a_3.\text{myBank}$ , says that  $a_2$  and  $a_3$  have the same bank. In fact, this assertion is satisfied in both  $\sigma_1$  and  $\sigma_2$ , written formally as

$$..., \sigma_1 \models a_2.\text{myBank} = a_3.\text{myBank} \quad ..., \sigma_2 \models a_2.\text{myBank} = a_3.\text{myBank}.$$

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

$$..., \sigma_1 \models a_2.\text{myBank} : \text{Bank}.$$

We support ghost fields [??], *e.g.*  $a_1.\text{balance}$  is a physical field in  $\sigma_1$  and a ghost field in  $\sigma_2$  since in  $\text{MBA2}$  an  $\text{Account}$  does not store its balance (as can be seen in appendix ?? Fig. ??).

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

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

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

$$..., \sigma_1 \models \langle a_2 \text{ access } b_1 \rangle$$

Assuming that  $\sigma_1$  is executing the method body corresponding to the call  $a_2.\text{deposit}(a_3, 360)$ , we have

$$..., \sigma_1 \models \langle a_2 \text{ access } a_3 \rangle,$$

Namely, during execution of  $\text{deposit}$ , the object at  $a_2$  has access to the object at  $a_3$ , and could, if the method body chose to, call a method on  $a_3$ , or store a reference to  $a_3$  in its own fields. Note that access is not symmetric, nor transitive:

$$..., \sigma_1 \not\models \langle a_3 \text{ access } a_2 \rangle,$$

$$..., \sigma_2 \models \langle a_2 \text{ access}^* a_3 \rangle, \quad ..., \sigma_2 \not\models \langle a_2 \text{ access } a_3 \rangle.$$

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

$$..., \sigma_3 \models \langle x \text{ calls } a_2.\text{deposit}(a_3, 360) \rangle.$$

means that the receiver in  $\sigma_3$  is  $x$ , and the next statement to be executed is  $a_2.\text{deposit}(a_3, 360)$ .

*Space: In* The space assertion  $\langle A \text{ in } S \rangle$  establishes validity of  $A$  in a configuration restricted to the objects from the set  $S$ . For example, if object 94 is included in  $S_1$  but not in  $S_2$ , then we have

$$..., \sigma_1 \models \langle (\exists o. \langle o \text{ access } a_4 \rangle) \text{ in } S_1 \rangle \quad ..., \sigma_1 \not\models \langle (\exists o. \langle o \text{ access } a_4 \rangle) \text{ in } S_2 \rangle.$$

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

*Time: Next, Will, Prev, Was* We support several operators from temporal logic:  $\langle \text{next}(A) \rangle$ ,  $\langle \text{will}(A) \rangle$ ,  $\langle \text{prev}(A) \rangle$ , and  $\langle \text{was}(A) \rangle$  to talk about the future or the past in one or more number steps. The assertion  $\langle \text{will}(A) \rangle$  expresses that  $A$  will hold in one or more steps. For example, taking  $\sigma_4$  to be similar to  $\sigma_2$ , the next statement to be executed to be  $a_2.\text{deposit}(a_3, 360)$ , and  $M_{BA2} \circ ..., \sigma_4 \models a_2.\text{balance} = 60$ , then

$$M_{BA2} \circ ..., \sigma_4 \models \langle \text{will}(a_2.\text{balance} = 420) \rangle.$$

Notice the use of the *internal* module,  $M_{AB2}$ , needed for looking up the method body of  $\text{deposit}$ .

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

$$M_{AB2} \circ ..., \sigma_2 \models \langle \text{external}(u_2) \rangle, \quad M_{AB2} \circ ..., \sigma_2 \not\models \langle \text{external}(a_2) \rangle,$$

$M_{AB2} \not\models \dots, \sigma_2 \models \text{external}\langle b_1.\text{ledger} \rangle$

The *internal* module,  $M_{BA2}$ , is needed to judge which objects are internal or external.

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

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

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  $a_2.\text{deposit}(a_3, 60)$ , then the balance of  $a_2$  will eventually change:

$M_{BA2} \models \dots, \sigma_2 \models \langle \dots \text{calls } a_1.\text{deposit}(a_2, 60) \rangle \longrightarrow \text{will}\langle \text{changes}\langle a_2.\text{balance} \rangle \rangle$ .

We now look deeper into the meaning of space assertions,  $\langle A \text{ in } S \rangle$ . They allow us to characterise the set of objects which have authority over certain effects (here  $A$ ). In particular, the assertion  $\langle \text{will}\langle A \rangle \text{ in } S \rangle$  does two things: it requires that  $A$  will hold in the future, and 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

$\text{will}\langle \langle A \text{ in } S \rangle \rangle \longrightarrow P(S)$ ,

where  $P$  is some property over a set. These assertions say, that if ever in the future  $A$  becomes valid, and if the objects involved in making  $A$  valid are included in  $S$ , then  $S$  must satisfy  $P$ . Such assertions can be used to restrict whether  $A$  will become valid. 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 general,  $\text{will}\langle \langle A \text{ in } S \rangle \rangle$  is different from  $\langle \text{will}\langle A \rangle \text{ in } S \rangle$ . Namely, in the former assertion,  $S$  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,  $S$  need only contain the objects needed to establish  $A$  in that future configuration. For example, take  $S_1$  to consist of objects 1, 2, 4, 93, and 94, and  $S_2$  to consist of objects 1, 2, 4. Then

$M_{BA1} \models \dots, \sigma_1 \models \langle \text{will}\langle \text{changes}\langle a_2.\text{balance} \rangle \rangle \text{ in } S_1 \rangle$

$M_{BA1} \not\models \dots, \sigma_1 \models \langle \text{will}\langle \text{changes}\langle a_2.\text{balance} \rangle \rangle \text{ in } S_2 \rangle$

$M_{BA1} \models \dots, \sigma_1 \models \text{will}\langle \langle \text{changes}\langle a_2.\text{balance} \rangle \text{ in } S_2 \rangle \rangle$

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

The next two sections discuss the semantics of *Chainmail*. Section ?? contains an overview of the formal model and section ?? 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 ?? we introduce the shape of the judgments used to give semantics to *Chainmail*, while in section ?? 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  $M \models A$  hold?

Our answer has to reflect the fact that we are dealing with an *open world*, where  $M$ , our module, may be linked with *arbitrary untrusted code*. To model the open world, we consider pairs of modules,  $M \mathbin{\text{\textcircled{;}}} M'$ , where  $M$  is the module whose code is supposed to satisfy the assertion, and  $M'$  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:  $M \models A$  holds if for all further, *potentially adversarial*, modules  $M'$  and in all runtime configurations  $\sigma$  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:

$$M \models A \quad \text{if} \quad \forall M'. \forall \sigma \in \mathcal{A}rising(M \mathbin{\text{\textcircled{;}}} M'). [M \mathbin{\text{\textcircled{;}}} M', \sigma \models A].$$

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

The judgement  $M \mathbin{\text{\textcircled{;}}} M', \sigma \models A$  means that assertion  $A$  is satisfied by  $M \mathbin{\text{\textcircled{;}}} M'$  and  $\sigma$ . As in traditional specification languages [??], satisfaction is judged in the context of a runtime configuration  $\sigma$ ; 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  $\text{will}(\_)$ .

We distinguish between internal and external modules. This offers two advantages: First, *Chainmail* includes the “external( $\circ$ )” assertion to require that an object belongs to the external module, as in the Bank Account’s assertion (2) and (3) in section ???. 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

$$M \mathbin{\text{\textcircled{;}}} M', \sigma \rightsquigarrow \sigma'$$

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 judgement  $M \mathbin{\text{\textcircled{;}}} M', \sigma \rightsquigarrow \sigma'$  and the set  $\mathcal{A}rising(M \mathbin{\text{\textcircled{;}}} M')$ .

## 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.<sup>2</sup>

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 ??,

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. ??. 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. ??.  $\mathcal{L}_{oo}$  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. ??. We discuss ghost fields in the next section.

<sup>2</sup>We believe that *Chainmail* can be applied to any language with these features.

Runtime configurations,  $\sigma$ , 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 ??.

We define *one-module* execution through a judgment of the form  $M, \sigma \rightsquigarrow \sigma'$  in the Appendix, Fig. ??. We define a module linking operator  $\circ$  so that  $M \circ M'$  is the union of the two modules, provided that their domains are disjoint, c.f. Appendix, Def. ??. As we said in section ??, 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  $\sigma, \sigma'$ , and a module-pair  $M \S M'$  we define execution where  $M$  is the internal, and  $M'$  is the external module as below:*

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

In the definition above,  $\text{Class}(x)_\sigma$  looks up the class of the object stores at  $x$ , c.f. Appendix, Def. ??. For example, for  $\sigma_4$  as in Section ?? whose next statement to be executed is  $a_2.\text{deposit}(a_3, 360)$ , we would have a sequence of configurations  $\sigma_{41}, \dots, \sigma_{4n}, \sigma_5$  so that the one-module execution gives  $M_{BA2}, \sigma_4 \rightsquigarrow \sigma_{41} \rightsquigarrow \sigma_{42} \dots \rightsquigarrow \sigma_{4n} \rightsquigarrow \sigma_5$ . This would correspond to an atomic evaluation in the two-module execution:  $M_{BA2} \S M', \sigma_4 \rightsquigarrow \sigma_5$ .

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  $M, M', M''$ , and  $M'''$  and runtime configurations  $\sigma$ , and  $\sigma'$  we have:*

- $(M \circ M') \circ M'' = M \circ (M' \circ M'')$  and  $M \circ M' = M' \circ M$ .
- $M, \sigma \rightsquigarrow \sigma'$ , and  $M \circ M'$  is defined, implies  $M \circ M', \sigma \rightsquigarrow \sigma'$ .
- $M \S M', \sigma \rightsquigarrow \sigma'$  implies  $(M \circ M'') \S (M' \circ M'''), \sigma \rightsquigarrow \sigma'$ .

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:*

- $\text{Initial}(\langle \psi, \chi \rangle)$ , if  $\psi$  consists of a single frame  $\phi$  with  $\text{dom}(\phi) = \{\text{this}\}$ , and there exists some address  $\alpha$ , such that  $[\text{this}]_\phi = \alpha$ , and  $\text{dom}(\chi) = \alpha$ , and  $\chi(\alpha) = (\text{Object}, \emptyset)$ .
- $\mathcal{A}rising(M \S M') = \{ \sigma \mid \exists \sigma_0. [\text{Initial}(\sigma_0) \wedge M \S M', \sigma_0 \rightsquigarrow^* \sigma] \}$

## 5 ASSERTIONS

We now define the syntax and semantics of expressions and holistic assertions. Each of the novel, holistic, features of *Chainmail* (permission, control, time, space, and viewpoint) introduced its

own challenges, as well as our wish to support some form of recursion while keeping the logic of assertions classical. We will discuss our treatment of these challenges 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.*

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

Notice that 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 ??, standard assertions about heap/stack and logical connectives in ??, the treatment of (permission, control, space, and viewpoint) in ?? the treatment of time in ??, and properties of assertions in ??. The judgement  $M \S M', \sigma \models A$  expresses that  $A$  holds in  $M \S M'$  and  $\sigma$ , and while  $M \S M', \sigma \not\models A$  expresses that  $A$  does not hold in  $M \S M'$  and  $\sigma$ .

## 5.2 Values of Expressions

The value of an expression is described through judgment  $M, \sigma, e \hookrightarrow v$ , defined in Figure ??. We use the configuration,  $\sigma$ , 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  $e_0.f(e_1, \dots, e_n)$ , and their value is described in rule Field\_Ghost\_Val: The lookup function  $\mathcal{G}$  (defined in the obvious way in the Appendix, Def.??) returns the expression constituting the body for that ghost field, as defined in the class of  $e_0$ . 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  $M_0$  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.
```

True_Val	False_Val	Null_Val	Var_Val
$\frac{}{M, \sigma, \text{true} \hookrightarrow \text{true}}$	$\frac{}{M, \sigma, \text{false} \hookrightarrow \text{false}}$	$\frac{}{M, \sigma, \text{null} \hookrightarrow \text{null}}$	$\frac{}{M, \sigma, x \hookrightarrow \sigma(x)}$
Field_Heap_Val		Field_Ghost_Val	
$\frac{M, \sigma, e \hookrightarrow \alpha}{M, \sigma, e.f \hookrightarrow v}$		$\frac{M, \sigma, e_0 \hookrightarrow \alpha}{M, \sigma, e_i \hookrightarrow v_i \quad i \in \{1..n\}}$	
Field_Ghost_Val2		$\mathcal{G}(M, \text{Class}(\alpha)_\sigma, f) = f(p_1, \dots, p_n) \{ e \}$	
$\frac{M, \sigma, e.f() \hookrightarrow v}{M, \sigma, e.f \hookrightarrow v}$		$\frac{M, \sigma[p_1 \mapsto v_1, \dots, p_n \mapsto v_n], e \hookrightarrow_W v}{M, \sigma, e_0.f(e_1, \dots, e_n) \hookrightarrow v}$	
If_True_Val		If_False_Val	
$\frac{M, \sigma, e \hookrightarrow \text{true}}{M, \sigma, e_1 \hookrightarrow v}$		$\frac{M, \sigma, e \hookrightarrow \text{false}}{M, \sigma, e_2 \hookrightarrow v}$	
$\frac{}{M, \sigma, \text{if } e \text{ then } e_1 \text{ else } e_2 \hookrightarrow v}$		$\frac{}{M, \sigma, \text{if } e \text{ then } e_1 \text{ else } e_2 \hookrightarrow v}$	
Equals_True_Val		Equals_False_Val	
$\frac{M, \sigma, e_1 \hookrightarrow v}{M, \sigma, e_2 \hookrightarrow v}$		$\frac{M, \sigma, e_1 \hookrightarrow v}{M, \sigma, e_2 \hookrightarrow v'}$	
$\frac{}{M, \sigma, e_1 = e_2 \hookrightarrow \text{true}}$		$\frac{v \neq v'}{M, \sigma, e_1 = e_2 \hookrightarrow \text{false}}$	

Fig. 4. Value of Expressions

Notice that the relation  $\hookrightarrow$  is partial. For example, assume a configuration  $\sigma_0$  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,  $M_0, \sigma_0, \text{acyc.acyclic} \hookrightarrow \text{true}$ , but we would have no value for  $M_0, \sigma_0, \text{cyc.last} \hookrightarrow \dots$ , nor for  $M_0, \sigma_0, \text{cyc.acyclic} \hookrightarrow \dots$ .

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 ?? talk about the `balance` of an account. In module  $M_{BA1}$  (Appendix ??), where we keep the balances in the account objects, this is a physical field. In  $M_{BA2}$  (also in Appendix ??), 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,  $\sigma$ , variables  $x$  or  $S$ , we define its interpretation as follows:*

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

DEFINITION 5 ( BASIC ASSERTIONS). *For modules  $M, M'$ , configuration  $\sigma$ , we define:*

- $M \S M', \sigma \models e$  if  $M, \sigma, e \hookrightarrow \text{true}$
- $M \S M', \sigma \models e = e'$  if there exists a value  $v$  such that  $M, \sigma, e \hookrightarrow v$  and  $M, \sigma, e' \hookrightarrow v$ .
- $M \S M', \sigma \models e : \text{ClassId}$  if there exists an address  $\alpha$  such that  
 $M, \sigma, e \hookrightarrow \alpha$ , and  $\text{Class}(\alpha)_\sigma = \text{ClassId}$ .
- $M \S M', \sigma \models e \in S$  if there exists a value  $v$  such that  $M, \sigma, e \hookrightarrow v$ , and  $v \in [S]_\sigma$ .

Note that satisfaction of assertions which contain expressions is predicated on termination of these expressions. Continuing our earlier example,  $M_0 \S M', \sigma_0 \models \text{acyc}.\text{acyclic}$  holds for any  $M'$ , while  $M_0 \S M', \sigma_0 \models \text{cyc}.\text{acyclic}$  does not hold, and  $M_0 \S M', \sigma_0 \models \text{cyc}.\text{acyclic} = \text{false}$  does not hold either. In general, when  $M \S M', \sigma \models e$  holds, then  $M \S M', \sigma \models e = \text{true}$  holds too. But when  $M \S M', \sigma \models e$  does not hold, this does *not* imply that  $M \S M', \sigma \models e = \text{false}$  holds. Finally, an assertion of the form  $e_0 = e_0$  does not always hold; for example,  $M_0 \S M', \sigma_0 \models \text{cyc}.\text{last} = \text{cyc}.\text{last}$  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  $\sigma$ , we define:*

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

Notice that satisfaction is not preserved with growing configurations; for example, the assertion  $\forall x. [x : \text{Account} \rightarrow x.\text{balance} > 100]$  may hold in a smaller configuration, but not hold in an extended configuration. Nor is it preserved with configurations getting smaller; consider e.g.  $\exists x. [x : \text{Account} \wedge x.\text{balance} > 100]$ .

Again, with our earlier example,  $M_0 \S M', \sigma_0 \models \neg(\text{cyc}.\text{acyclic} = \text{true})$  and  $M_0 \S M', \sigma_0 \models \neg(\text{cyc}.\text{acyclic} = \text{false})$ , and also  $M_0 \S M', \sigma_0 \models \neg(\text{cyc}.\text{last} = \text{cyc}.\text{last})$  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*

- $M \S M', \sigma \models \langle x \text{ access } y \rangle$  if  $[x]_\sigma$  and  $[y]_\sigma$  are defined, and
  - $[x]_\sigma = [y]_\sigma$ , or
  - $[x.f]_\sigma = [y]_\sigma$ , for some field  $f$ , or
  - $[x]_\sigma = [\text{this}]_\sigma$  and  $[y]_\sigma = [z]_\sigma$ , for some variable  $z$  and  $z$  appears in  $\sigma.\text{contn}$ .

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

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

**DEFINITION 8 (CONTROL).** For any modules  $M, M'$ , variables  $x, y, z_1, \dots, z_n$ , we define:

- $M \models M', \sigma \models \langle x \text{ calls } y.m(z_1, \dots, z_n) \rangle$  if  $[x]_\sigma, [y]_\sigma, [z_1]_\sigma, \dots, [z_n]_\sigma$  are defined, and  
 –  $[this]_\sigma = [x]_\sigma$ , and  
 –  $\sigma.\text{contn} = u.m(v_1, \dots, v_n); \_$  for some  $u, v_1, \dots, v_n$ , and  
 –  $[y]_\sigma = [u]_\sigma$ , and  $[z_i]_\sigma = [v_i]_\sigma$ , for all  $i$ .

Thus,  $\langle x \text{ calls } y.m(z_1, \dots, z_n) \rangle$  expresses the call  $y.m(z_1, \dots, z_n)$  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

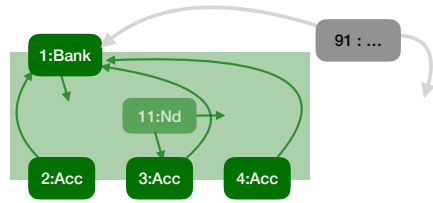
- $M \models M', \sigma \models \text{external}\langle x \rangle$  if  $[x]_\sigma$  is defined and  $\text{Class}([x]_\sigma) \notin \text{dom}(M)$
- $M \models M', \sigma \models \text{internal}\langle x \rangle$  if  $[x]_\sigma$  is defined and  $\text{Class}([x]_\sigma) \in \text{dom}(M)$

*Space* is about asserting that some property  $A$  holds in a configuration whose objects are restricted to those from a given 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  $\langle A \text{ in } S \rangle$  in a configuration  $\sigma$ , we first define a restriction operation,  $\sigma \downarrow_S$  which restricts the objects from  $\sigma$  to only those from  $S$ .

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

- $\sigma \downarrow_S \triangleq (\psi, \chi')$ , if  $\sigma = (\psi, \chi)$ , and  $\text{dom}(\chi') = [S]_\sigma$ , and  $\forall \alpha \in \text{dom}(\chi'). \chi(\alpha) = \chi'(\alpha)$

For example, if we take  $\sigma_2$  from Fig. ?? in Section ??, and restrict it with some set  $S_4$  such that  $[S_4]_{\sigma_2} = \{91, 1, 2, 3, 4, 11\}$ , then the restriction  $\sigma_2 \downarrow_{S_4}$  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 a heap  $\chi$  which can be split into disjoint  $\chi$  and  $\chi'$  where  $\chi$  satisfies  $A$  and  $\chi'$  satisfies  $A'$ . That is, the split of the heap is determined by the assertions  $A$  and  $A'$ , while the operation  $\sigma \downarrow_S$  is explicitly taking a set as an argument.

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

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

- $M \models M', \sigma \models \langle A \text{ in } S \rangle$  if  $M \models M', \sigma \downarrow_S \models A$ .



The set  $S$  in the assertion  $\langle A \text{ in } S \rangle$  is related to framing from implicit dynamic frames [??]: in an implicit dynamic frames assertion  $\text{acc}_{x.f} * A$ , the frame  $x.f$  prescribes which locations may be used to determine validity of  $A$ . The difference is that frames are sets of locations (pairs of address and field), while our  $S$ s are sets of addresses. More importantly, in implicit dynamic frames, assertions whose frame is not large enough are badly formed, while in our work, such assertions are allowed and may hold or not (e.g.  $M_{BA2} \models M', \sigma \models \exists n. a_2. \text{balance} = n$ , while  $M_{BA2} \models M', \sigma \not\models \langle (\exists n. a_2. \text{balance} = n) \text{ in } S_4 \rangle$ )

## 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  $\text{will} \langle x.f = 3 \rangle$ ; it is satisfied in the *current* configuration,  $\sigma$ , if in some *future* configuration,  $\sigma'$ , the field  $f$  of the object that is pointed at by  $x$  in the *current* configuration ( $\sigma$ ) has the value 3, even in that future configuration  $x$  denotes a different object ( $[x]_\sigma \neq [x]_{\sigma'}$ ). 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  $\sigma$  and where the `contn` has been consistently renamed, while the heap is taken from  $\sigma'$ . This allows us to interpret expressions in the newer (or older) configuration  $\sigma'$  but with the variables bound according to the top frame from  $\sigma$ ; e.g. we can obtain that value of  $x$  in configuration  $\sigma'$  even if  $x$  was out of scope. The consistent renaming of the code allows the correct modelling of execution (as needed, for the semantics of nested time assertions, as e.g. in  $\text{will} \langle x.f = 3 \wedge \text{will} \langle x.f = 5 \rangle \rangle$ )

DEFINITION 12 (ADAPTATION). *For runtime configurations  $\sigma, \sigma'$ :*

- $\sigma \triangleleft \sigma' \triangleq (\phi'' \cdot \psi', \chi')$  if
  - $\sigma = (\phi \cdot \_, \_)$ ,  $\sigma' = (\phi' \cdot \psi', \chi')$ , and
  - $\phi = (\text{contn}, \beta)$ ,  $\phi' = (\text{contn}', \beta')$ ,  $\phi'' = (\text{contn}'[zs/zs'], \beta[zs' \mapsto \beta'(zs)])$ , where
  - $zs = \text{dom}(\beta)$ ,  $zs'$  is a set of variables with the same cardinality as  $zs$ , and
  - all variables in  $zs'$  are fresh in  $\beta$  and in  $\beta'$ .

That is, in the new frame  $\phi''$  from above, we keep the same continuation as from  $\sigma'$  but rename all variables with fresh names  $\text{prgzs}'$ , and in the variable map we combine that from  $\sigma$  and  $\sigma'$  but avoid names clashes through the renaming  $[zs' \mapsto \beta'(zs)]$ .

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 `contn` 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  $M, M'$ , and assertion  $A$  we define*

- $M \models M', \sigma \models \text{next} \langle A \rangle$  if  $\exists \sigma'. [M \models M', \phi \rightsquigarrow \sigma' \wedge M \models M', \sigma \triangleleft \sigma' \models A]$ ,  
and where  $\phi$  is so that  $\sigma = (\phi \cdot \_, \_)$ .
- $M \models M', \sigma \models \text{will} \langle A \rangle$  if  $\exists \sigma'. [M \models M', \phi \rightsquigarrow^* \sigma' \wedge M \models M', \sigma \triangleleft \sigma' \models A]$ ,  
and where  $\phi$  is so that  $\sigma = (\phi \cdot \_, \_)$ .

- $M \circ M', \sigma \models \text{prev}\langle A \rangle$  if  $\forall \sigma_1, \sigma_2. [ \text{Initial}\langle \sigma_1 \rangle \wedge M \circ M', \sigma \rightsquigarrow^* \sigma_2 \wedge M \circ M', \sigma_2 \rightsquigarrow \sigma \longrightarrow M \circ M', \sigma \triangleleft \sigma_2 \models A ]$
- $M \circ M', \sigma \models \text{was}\langle A \rangle$  if  $\forall \sigma_1, \dots, \sigma_n. [ \text{Initial}\langle \sigma_1 \rangle \wedge \sigma_n = \sigma \wedge \forall i \in [1..n]. M \circ M', \sigma_i \rightsquigarrow \sigma_{i+1} \longrightarrow \exists j \in [1..n-1]. M \circ M', \sigma \triangleleft \sigma_j \models A ]$

## 5.6 Modules satisfying assertions

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

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

- $M \models A$  if  $\forall M'. \forall \sigma \in \mathcal{A} \text{rising}(M \circ M'). M \circ M', \sigma \models A$

## 5.7 Properties of Assertions

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

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

We can then prove that the usual equivalences hold, e.g.  $A \vee A' \equiv A' \vee A$ , and  $\neg(\exists x. A) \equiv \forall x. (\neg A)$ . Our assertions are classical, e.g.  $A \wedge \neg A \equiv \text{false}$ , and  $M \circ M', \sigma \models A$  and  $M \circ M', \sigma \models A \rightarrow A'$  implies  $M \circ M', \sigma \models A'$ . Full definitions and lemmas appear in Appendix ?? **However, this desirable property comes at the loss of some expected equivalences, e.g., in general,  $e = \text{false}$  and  $\neq e$  are not equivalent.**

## 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 an untrusted client *restricted* access to an object's functionality. This is usually achieved through the introduction of an intermediate object. Such intermediate objects — protection proxies [?] — are a common design pattern, and their security properties have been studied at length in the object capabilities literature [??].

? proposed specifications for attenuation of trees of DOM nodes. Access to a DOM node gives access to its parent and all its children nodes, and the ability to modify the properties of any accessible node. As the top nodes of the tree usually contain privileged information (such as web content showing your banking details) while the lower nodes contain less crucial information (such as advertisements for BREXIT) we want to be able to limit access given to third parties to only the lower part of the DOM tree (so that Jacob Rees-Mogg cannot access your bank account). We do this through a *Wrapper*, which has a field *node* pointing to a *Node* in the underlying DOM, and a field *height*. When you hold a *Wrapper* you can modify the property of all the descendants of the *height*-th ancestors of the node of that particular *Wrapper*.

Figure ?? shows a *Wrapper* object attenuating the use of *Nodes*. The function *usingWrappers* has as its parameter an object of unknown provenance, here called *unknown*. We create a tree consisting of nodes *n1*, *n2*, ... *n6*, depicted as blue circles on the right-hand-side of the Figure and a wrapper for *n5* with height 1. This means that the wrapper *w* may be used to modify the objects in the green triangle, while it cannot be used to modify the objects within the blue triangle. We call a function named *untrusted* on the *unknown* object, and pass *w* as argument.

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 transitively access to the entire tree,

```

785 method usingWrappers(unknown) {
786   n1=Node(null, "fixed");
787   n2=Node(n1, "robust");
788   n3=Node(n2, "const");
789   n4=Node(n3, "fluid");
790   n5=Node(n4, "variable");
791   n6=Node(n5, "ethereal");
792   w=Wrapper(n5, 1);
793
794   unknown.untrusted(w);
795
796   assert n2.property=="robust"
797   ...
798 }

```

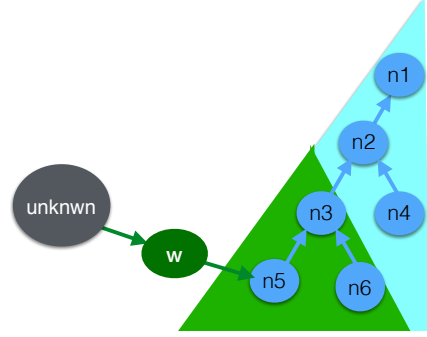


Fig. 5. Wrappers protecting Nodes

we know that the `untrusted` function will not affect the `property` fields of the nodes `n1`, `n2`, and `n4`. Thus, the assertion 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, [?] consists of pairs of PRE- and POST- conditions for each of the functions of that class. A broken wrapper could accidentally permit access to nodes one or two levels above the expected height due to off-by one errors. More seriously, a malicious wrapper could offer a back-door public accessor method that leaks the underlying DOM object through the wrapper, returning direct access to any of the other nodes in the DOM, or even to queue a task to delete every property at midnight GMT on 29 March 2019. All of these errors are possible while preserving the PRE- and POST- conditions expected of a `Wrapper`.

A holistic specification can prevent these problems:

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`.

Or in *Chainmail*:

$\forall S : \text{Set.Vnd} : \text{Node}.$

$[ \langle \text{will}(\text{changes}(\text{nd.property})) \text{ in } S \rangle$

$\longrightarrow$

$\exists o : \text{Object} [ o \in S \wedge \neg(o : \text{Node}) \wedge \neg(o : \text{Wrapper}) \wedge$

$[ \exists nd' : \text{Node}. \langle o \text{ access } nd' \rangle \vee$

$\exists w : \text{Wrapper}. \exists k : \mathbb{N}. ( \langle o \text{ access } w \rangle \wedge \text{nd.parnt}^k = w.\text{node.parnt}^{w.\text{height}} ) ] ]$

$] ]$

That is, if the value of `nd.property` is modified (`changes(_)` at some future time (`will(_)`) and if reaching that future time involves no more objects than those from set `S` (i.e. `⟨_ in S⟩`), 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 (`⟨o access nd'⟩`), or to some wrapper `w` and the `w.height`-th parent of `w` is an ancestor of `nd` (that is, `parntk = w.node.parntw.height`). With this specification, we can ensure that all future updates of the `Wrapper` class will continue to protect the DOM.

## 6.2 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 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 [?].

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

Second, that when an owner asks to be repaid, she is sent all her money.<sup>3</sup>

$$\forall d : \text{DAO}. \forall p : \text{Any}. \forall m : \text{Nat}. \\ [ d.\text{balance}(p) = m \wedge \langle p \text{ calls } \text{repay}.d(\_) \rangle \\ \longrightarrow \text{will}(\langle d \text{ calls } \text{send}.p(m) \rangle) ]^4$$

Third, that the balance of an owner the previous call was a repayment; it is `m` if the previous call was `p` joining `d` and paying in `m`. **KJX: This is scrambled - I don't understand the previous sentence/ If a previous call was `p` joining a DAO `d` and paying in `m` then the balance of `p` is `m`.**

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

More cases are needed to reflect the financing and repayments of proposals, but they can be expressed with the concepts described so far.<sup>5</sup>

The DAO holds 19<sup>6</sup> functions which 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 xxx affect the balance of a participant<sup>7</sup> **KJX: OK yes we would, but we're not going for this now, are we?**

## 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 [?].

<sup>3</sup>This does not necessarily work well with our visible states!

<sup>4</sup>We also need to explain that function carry money implicitly

<sup>5</sup>We do not represent al of the aspects of the real DAO, but the real DAO is huge. How do we still argue the value of our approach? This kind of contradicts our argument. **KJX: NO IT DOESN'T. IT MAKES IT.** **we don't even need to talk about other things that may accept the balabnce or anything.** **ALL** We ned to say is if you;'re an owners, you can always be repaid. **that's it.** **The rest is silence, which is precisely the point.**

<sup>6</sup>SD I think more – need to check

<sup>7</sup>We need to study the white paper to make a good argument.

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.  $\mathcal{L}_{\infty}$  does not contain inheritance simply because it is not necessary to demonstrate specifications of robustness: whether an  $\mathcal{L}_{\infty}$  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  $\mathcal{L}_{\infty}$  class.

*Contracts and Preconditions.* Traditional specification languages based on pre- and post-conditions are generally based on design-by-contract assumptions: “if the precondition is not satisfied, the routine is not bound to do anything” [?] — that is, the routine can do anything. Ensuring preconditions is the responsibility of the caller, and if the caller fails to meet that responsibility it is not the responsibility of the invoked function to fix the problem [?]. Underlying this approach, however, is the assumption of a closed system: that all the modules in a system are equally trusted, and so inserting redundant tests would just make the system more complex and more buggy. Indeed, ? states that “This principle is the exact opposite of the idea of defensive programming.”

In an open world, however, we do not have the luxury of trusting the other components with which we interact — indeed we barely have the luxury of trusting ourselves. We must assume that our methods may be invoked at any time, with any combination of arguments the underlying platform will support, irrespective of the overall state of the system, or of the object that receives the method request: in some sense, this is the very definition of an open system in an open world. Since methods cannot control when they are invoked, we must work as if all (potentially) externally-visible methods just have the precondition `true` — and we had better be very careful about any assumptions we make about which method or objects are in fact externally visible and which are not. **Holistic specifications directly support robust programming by making those kinds of assumptions explicit, giving the necessary conditions under which effects may take place and objects may become accessible, and then can help programmers ensure their programs maintain those conditions.**

*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 crosscutting 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.  $\langle \_ \text{calls } s.\text{setValue}(v) \rangle \rightarrow \text{will}(\langle s \text{ calls } s.\text{observer.notify}(v) \rangle)$ ) — 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 ??) 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

When you write a module that is to be used with other code, the last thing you want to happen is that some other code uses it causing effects that you never intended. Our specification language *Chainmail* has been designed, so that developers whose modules are going to be used in the wild, have the language to constrain the usage of their code. In addition to classical function by function specification techniques, we have shown that a holistic or whole program approach is needed to make open world code robust. We have shown how *Chainmail* can be used to give holistic specifications of key exemplar problems: the bank account, the wrapped DOM, 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 that in the model of *Chainmail* without exposing it to the programmer. 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 [?].

In order to make these kinds of specifications practically useful, we plan to develop logics for proving adherence to holistic specs, as well as logics for using holistic specs in proofs. 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 ??):*

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

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. ?. 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 ?. Note that the expressions that make up the bodies of ghostfield declarations ( $\epsilon$ ) 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.*

```

1128  ClassDescr ::= class ClassId { ( FieldDecl ) * ( MethDecl ) * ( GhosDecl ) * }
1129  FieldDecl  ::= field f
1130  MethDecl   ::= method m( x* ) { Stmts }
1131  Stmts      ::= Stmt | Stmt ; Stmts
1132  Stmt       ::= x.f := x | x := x.f | x := x.m( x* ) | x := new C( x* ) | return x
1133  GhostDecl  ::= ghost f( x* ) { e }
1134  e           ::= true | false | null | x | e = e | if e then e else e | e.f( e* )
1135  x, f, m   ::= Identifier

```

where we use metavariables as follows:  $x \in \text{VarId}$   $f \in \text{FldId}$   $m \in \text{MethId}$   $C \in \text{ClassId}$ , 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  $m$ :

$$\begin{aligned}
 \mathcal{M}(\mathcal{M}, C, m) &\triangleq \begin{cases} m(p_1, \dots, p_n) \{ \text{Stmts} \} \\ \text{if } \mathcal{M}(C) = \text{class } C \{ \dots \text{method } m(p_1, \dots, p_n) \{ \text{Stmts} \} \dots \} . \\ \text{undefined, otherwise.} \end{cases} \\
 \mathcal{G}(\mathcal{M}, C, f) &\triangleq \begin{cases} f(p_1, \dots, p_n) \{ e \} \\ \text{if } \mathcal{M}(C) = \text{class } C \{ \dots \text{ghost } m(p_1, \dots, p_n) \{ e \} \dots \} . \\ \text{undefined, otherwise.} \end{cases}
 \end{aligned}$$

## 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,  $\sigma$ , 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,  $\text{Addr}$ , and use the identifier  $\alpha \in \text{Addr}$  to indicate an address.
- Values,  $v$ , are either addresses, or sets of addresses or null:  
 $v \in \{\text{null}\} \cup \text{Addr} \cup \mathcal{P}(\text{Addr})$ .
- Continuations are either statements (as defined in Definition ??) or a marker,  $x := \bullet$ , for a nested call followed by statements to be executed once the call returns.  
 $\text{Continuation} ::= \text{Stmts} \mid x := \bullet ; \text{Stmts}$
- Frames,  $\phi$ , consist of a code stub and a mapping from identifiers to values:  
 $\phi \in \text{CodeStub} \times \text{Ident} \rightarrow \text{Value}$ ,
- Stacks,  $\psi$ , 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:  
 $\text{Object} = \text{ClassId} \times (\text{FieldId} \rightarrow \text{Value})$ .
- Heaps,  $\chi$ , are mappings from addresses to objects:  $\chi \in \text{Addr} \rightarrow \text{Object}$ .
- Runtime configurations,  $\sigma$ , 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 ??:

DEFINITION 19 (INTERPRETATIONS). *We first define lookup of fields and classes, where  $\alpha$  is an address, and  $\mathfrak{f}$  is a field identifier:*

- $\chi(\alpha, \mathfrak{f}) \triangleq \text{fldMap}(\alpha, \mathfrak{f})$  if  $\chi(\alpha) = (\_, \text{fldMap})$ .
- $\text{Class}(\alpha)_\chi \triangleq \mathbb{C}$  if  $\chi(\alpha) = (\mathbb{C}, \_)$

*We now define interpretations as follows:*

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

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

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

In the definition of the operational semantics of  $\mathcal{L}_{\text{oo}}$  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  $\phi$  is the tuple  $(\text{stub}, \text{varMap})$ , we use the notation  $\phi.\text{contn}$  to obtain  $\text{stub}$ .
- Assuming a value  $v$ , and that  $\phi$  is the tuple  $(\text{stub}, \text{varMap})$ , we define  $\phi[\text{contn} \mapsto \text{stub}']$  for updating the stub, i.e.  $(\text{stub}', \text{varMap})$ . We use  $\phi[\mathfrak{x} \mapsto v]$  for updating the variable map, i.e.  $(\text{stub}, \text{varMap}[\mathfrak{x} \mapsto v])$ .
- Assuming a heap  $\chi$ , a value  $v$ , and that  $\chi(\alpha) = (\mathbb{C}, \text{fieldMap})$ , we use  $\chi[\alpha, \mathfrak{f} \mapsto v]$  as a shorthand for updating the object, i.e.  $\chi[\alpha \mapsto (\mathbb{C}, \text{fieldMap}[\mathfrak{f} \mapsto v])]$ .

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

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

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

### 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  $\mathfrak{x}$  in its domain,  $\lfloor \mathfrak{x} \rfloor_\phi$  is undefined. We define the relation  $\sigma \sqsubseteq \sigma'$  to express that  $\sigma$  has more information than  $\sigma'$ , 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  $\sigma, \sigma', \sigma''$ , frame  $\phi$ , stacks  $\psi, \psi'$ , heap  $\chi$ , address  $\alpha$  free in  $\chi$ , value  $v$  and object  $\mathfrak{o}$ , and define  $\sigma \sqsubseteq \sigma'$  as the smallest relation such that:*

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

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

- *If  $\lfloor \mathfrak{x} \rfloor_\sigma$  is defined, then  $\lfloor \mathfrak{x} \rfloor_{\sigma'} = \lfloor \mathfrak{x} \rfloor_\sigma$ .*

1226		methCall_OS
1227		
1228	$\phi.\text{contn} = x := x_0.m(x_1, \dots, x_n); \text{Stmts}$	
1229	$\lfloor x_0 \rfloor_\phi = \alpha$	
1230	$\mathcal{M}(\mathcal{M}, \text{Class}(\alpha)_{\chi, m}) = m(p_1, \dots, p_n) \{ \text{Stmts}_1 \}$	
1231	$\phi'' = (\text{Stmts}_1, (\text{this} \mapsto \alpha, p_1 \mapsto \lfloor x_1 \rfloor_\phi, \dots, p_n \mapsto \lfloor x_n \rfloor_\phi))$	
1232	$\frac{}{\mathcal{M}, (\phi \cdot \psi, \chi) \rightsquigarrow (\phi'' \cdot \phi[\text{contn} \mapsto x := \bullet; \text{Stmts}] \cdot \psi, \chi)}$	
1233		
1234		varAssgn_OS
1235	$\phi.\text{contn} = x := y.f; \text{Stmts}$	$\text{Class}(y)_\sigma = \text{Class}(\text{this})_\sigma$
1236	$\frac{}{\mathcal{M}, (\phi \cdot \psi, \chi) \rightsquigarrow (\phi[\text{contn} \mapsto \text{Stmts}, x \mapsto \lfloor y.f \rfloor_{\phi, \chi}] \cdot \psi, \chi)}$	
1237		
1238		fieldAssgn_OS
1239	$\phi.\text{contn} = x.f := y; \text{Stmts}$	$\text{Class}(x)_\sigma = \text{Class}(\text{this})_\sigma$
1240	$\frac{}{\mathcal{M}, (\phi \cdot \psi, \chi) \rightsquigarrow (\phi[\text{contn} \mapsto \text{Stmts}] \cdot \psi, \chi[\lfloor x \rfloor_\phi, f \mapsto \lfloor y \rfloor_{\phi, \chi}])}$	
1241		
1242		objCreate_OS
1243	$\phi.\text{contn} = x := \text{new } C(x_1, \dots, x_n); \text{Stmts}$	
1244	$\alpha \text{ new in } \chi$	
1245	$f_1, \dots, f_n \text{ are the fields declared in } \mathcal{M}(C)$	
1246	$\frac{}{\mathcal{M}, (\phi \cdot \psi, \chi) \rightsquigarrow (\phi[\text{contn} \mapsto \text{Stmts}, x \mapsto \alpha] \cdot \psi, \chi[\alpha \mapsto (C, f_1 \mapsto \lfloor x_1 \rfloor_\phi, \dots, f_n \mapsto \lfloor x_n \rfloor_\phi)])}$	
1247		
1248		return_OS
1249	$\phi.\text{contn} = \text{return } x; \text{Stmts} \text{ or } \phi.\text{contn} = \text{return } x$	
1250	$\phi'.\text{contn} = x' := \bullet; \text{Stmts}'$	
1251	$\frac{}{\mathcal{M}, (\phi \cdot \phi' \cdot \psi, \chi) \rightsquigarrow (\phi'[\text{contn} \mapsto \text{Stmts}', x' \mapsto \lfloor x \rfloor_\phi] \cdot \psi, \chi)}$	

Fig. 6. Operational Semantics

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

Note however, that such preservation does not hold for assertion. For example, if  $\sigma' \sqsubseteq \sigma$ , then  $\mathcal{M} \models M', \sigma \models \forall x.A$  and does not imply  $\mathcal{M} \models M', \sigma' \models \forall x.A$ , on the other hand,  $\mathcal{M} \models M', \sigma' \models \exists x.A$  does not imply  $\mathcal{M} \models M', \sigma \models \exists x.A$ .

## 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  $\circ$ :  $\text{Module} \times \text{Module} \rightarrow \text{Module}$  is defined as follows:*

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



and where,

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

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

LEMMA A.2 (PROPERTIES OF LINKING – REPETITION OF LEMMA ?? IN THE MAIN TEXT).

For any modules  $M$ ,  $M'$  and  $M''$ , and runtime configurations  $\sigma$ , and  $\sigma'$  we have:

- (1)  $(M \circ M') \circ M'' = M \circ (M' \circ M'')$ .
- (2)  $M \circ M' = M' \circ M$ .
- (3)  $M, \sigma \rightsquigarrow \sigma'$ , and  $M \circ M'$  is defined, implies  $M \circ M', \sigma \rightsquigarrow \sigma'$

PROOF. (1) and (2) follow from Definition ?. (3) follows from ?, and the fact that if a lookup  $M$  is defined for  $M$ , then it is also defined for  $M \circ M'$  and returns the same method, and similar result for class lookup. then  $\square$

## A.5 Module pairs and visible states semantics

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

Therefore, we define execution in terms of an internal module  $M$  and an external module  $M'$ , through the judgment  $M \S M', \sigma \rightsquigarrow \sigma'$ , which mandates that  $\sigma$  and  $\sigma'$  are external to  $M$ , and that there exists an execution which leads from  $\sigma$  to  $\sigma'$  which leads through intermediate configurations  $\sigma_2, \dots, \sigma_{n+1}$  which are all internal to  $M$ , and thus unobservable from the client. In a sense, we "pretend" that all calls to functions from  $M$  are executed atomically, even if they involve several intermediate, internal steps.

DEFINITION 24 (REPEATING DEFINITION ??). Given runtime configurations  $\sigma$ ,  $\sigma'$ , and a module-pair  $M \S M'$  we define execution where  $M$  is the internal, and  $M'$  is the external module as below:

- $M \S M', \sigma \rightsquigarrow \sigma'$  if there exist  $n \geq 2$  and runtime configurations  $\sigma_1, \dots, \sigma_n$ , such that
  - $\sigma = \sigma_1$ , and  $\sigma_n = \sigma'$ .
  - $M \circ M', \sigma_i \rightsquigarrow \sigma'_{i+1}$ , for  $1 \leq i \leq n-1$
  - $\text{Class}(\llbracket \text{this} \rrbracket_{\sigma})_{\sigma} \notin \text{dom}(M)$ , and  $\text{Class}(\llbracket \text{this} \rrbracket_{\sigma'})_{\sigma'} \notin \text{dom}(M)$ ,
  - $\text{Class}(\llbracket \text{this} \rrbracket_{\sigma_i})_{\sigma_i} \in \text{dom}(M)$ , 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  $\sigma$  to  $\sigma'$ . 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  $\text{will}(\_)$  and  $\text{was}(\_)$  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  $\sigma$ ,  $\sigma'$ ,

- $M \S M', \sigma \rightsquigarrow \sigma'$  implies  $M \S (M' \circ M''), \sigma \rightsquigarrow \sigma'$ .

- $M \circ M', \sigma \rightsquigarrow \sigma'$  implies  $(M \circ M'') \circ M', \sigma \rightsquigarrow \sigma'$ .

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

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  $\phi.\text{contn}$ . From such initial configurations, executions of code from  $M \circ M'$  creates a set of *arising* configurations, which, as we will see in Definition ??, are pertinent when judging  $M$ 's adherence to assertions.

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

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

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

```

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

In this section we revisit the Bank/Account example from section ??, 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 ?? . The first version gives rise to runtime configurations as  $\sigma_1$ , shown on the left side of Fig. ?? , while the second version gives rise to runtime configurations as  $\sigma_2$ , shown on the right side of Fig. ?? .

In this code, we use more syntax than the minimal syntax defined for  $\mathcal{L}_{oo}$  in Def. ?? , 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.

$M_{BA1}$ , the first version is shown Fig. ?? . 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.

```

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)
      srcNd.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

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

Note also that `Node` exposes the function `addToBalance(...)`; a call to this function modifies the balance of an `Account` without requiring that the caller has access to the `Account`. This might look as if it contradicted assertions (1) and (2) from section ?? . 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`.

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

```

1422 class Node{
1423     field balance
1424     field next
1425     field myAccount
1426
1427     method addToBalance(amt){
1428         this.balance = this.balance + amt
1429     }
1429     method find(acc){
1430         if this.myAccount == acc then{
1431             return this
1432         } else {
1433             if this.next==null then{
1434                 return null
1435             } else {
1436                 return this.next.find(acc)
1437             } } }
1438     method getBalance(){ return balance }
1439
1440     ghost balance(acc){
1441         if (this.myAccount == acc) then this.balance
1442                                     else ( if this.next==null
1443                                             then -1 else this.next.
1444                                             find(acc) )
1445     }
1446 }

```

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

```

1450 class Account{
1451
1452     field myBank
1453
1454     method deposit(src,amt){
1455         this.myBank.deposit(this,src,amt)
1456     } }
1457     method makeNewAccount(amt){
1458         if (amt>=0 && this.balance>=amt) then{
1459             newAcc = this.myBank.makeNewAccount(0)
1460             newAcc.deposit(this,amt)
1461             return newAcc
1462         } }
1463     ghost balance(){ this.myBank.balance(this) }
1464 }

```

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

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

DEFINITION 26 (EQUIVALENCE AND ENTAILMENTS OF ASSERTIONS).

- $A \subseteq A'$  if  $\forall \sigma. \forall M, M'. [ M \circ M', \sigma \models A \text{ implies } M \circ M', \sigma \models A' ]$ .
- $A \equiv A'$  if  $A \subseteq A'$  and  $A' \subseteq A$ .

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

- (1)  $M \circ M', \sigma \models A$  or  $M \circ M', \sigma \models \neg A$
- (2)  $M \circ M', \sigma \models A \wedge A'$  if and only if  $M \circ M', \sigma \models A$  and  $M \circ M', \sigma \models A'$
- (3)  $M \circ M', \sigma \models A \vee A'$  if and only if  $M \circ M', \sigma \models A$  or  $\sigma \models A'$
- (4)  $M \circ M', \sigma \models A \wedge \neg A$  never holds.
- (5)  $M \circ M', \sigma \models A$  and  $M \circ M', \sigma \models A \rightarrow A'$  implies  $M \circ M', \sigma \models A'$ .

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

(\*) either  $M \circ M', \sigma \models A$  or  $M \circ M', \sigma \not\models A$ .

We prove this using Definition ???. Then, we prove (\*) for all possible assertions, by induction of the structure of  $A$ , and the Definitions ??, ??, ??, ??, ??, and ??. Using the definition of  $M \circ M', \sigma \models \neg A$  from Definition ?? we conclude the proof of (1).

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

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

- (1)  $A \wedge \neg A \equiv \text{false}$
- (2)  $A \vee \neg A \equiv \text{true}$
- (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)  $\neg(\exists x. A) \equiv \forall x. (\neg A)$
- (11)  $\neg(\exists S : \text{SET}. A) \equiv \forall S : \text{SET}. (\neg A)$
- (12)  $\neg(\forall x. A) \equiv \exists x. \neg(A)$
- (13)  $\neg(\forall S : \text{SET}. A) \equiv \exists S : \text{SET}. \neg(A)$

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