Holistic Specifications for Robust Programs

AUTHOR, Address

ACM Reference Format:
author. 2019. Holistic Specifications for Robust Programs. 1, 1 (March 2019), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software guards our secrets, our money, our intellectual property, our reputation [19]. 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.

Thus, we expect and hope that our software will be *robust*: We expect and hope our software to behave correctly even if used by erroneous or malicious third parties. Robustness means something different for different software.We expect that our bank will only make payments from our account if instructed by us or somebody authorized[22], and that space on a web given to an advertiser will not be used to obtain access to our bank details[17].

***Susan:** I put in the ERC20 citation but I don't see what that has to do with banks authorising payments*

The importance of robustness has lead to the design of many programming language mechanisms which help write robust programs: constant fields or methods, private methods/fields, ownership[5] as well as the object capability paradigm[16], and its adoption in web systems [4, 8, 20] and programming languages such as Newspeak [3], Dart [2], Grace [1, 10], Wyvern [13].

While such programming language mechanisms make it *possible* to write robust programs, they cannot *ensure* that programs are robust. To be able to do this, we need ways to specify what robustness means for the 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 make a payment request to your bank, money will be transferred and as a result your funds will be reduced: the payment request is a sufficient condition for the reduction of funds. However, a bank client is also interested in *necessary* conditions: they want to be assured that no reduction in their funds will take place unless they themselves requested it.

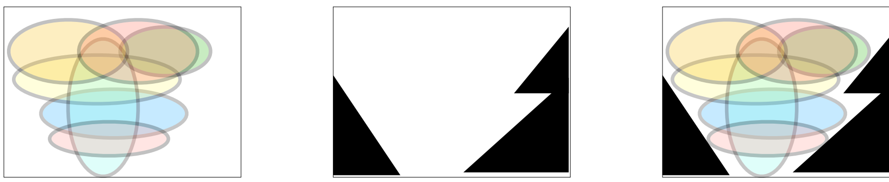
Necessary conditions are essentially about things that will *not* happen. For example, there will be no reduction to the account's funds without the owner's explicit request: the request being made by the owner is the necessary condition - under no other circumstances will the funds be reduced.

We give a visual representation of the difference between sufficient and necessary conditions in Fig. 1. We represent the space of all theoretically possible behaviours as points in the rectangle, each function is a coloured oval and its possible behaviours are the points in the area of that oval. The sufficient conditions are described on a per-function basis. The necessary conditions, on the other

Author's address: authorAddress.

XXXX-XXXX/2019/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

hand are about the behaviour of a module as a whole, and describe what is guaranteed not to happen; they are depicted as black triangles.



sufficient spec. necessary spec. holistic spec.

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

We propose that necessary conditions should be explicitly stated. Specifications should be *holistic*, in the sense that they describe the overall behaviour of a module: not only the behaviour of each of its functions separately, but also emerging behaviours through combination of functions. A holistic specification should therefore consist of the sufficient as well as the necessary conditions, as depicted in right hand side diagram in Fig. 1. susan: When our module which has been specified holistically, executes, possibly interacting with other software, the behaviours represented by the black triangles cannot occur. In Section 7 we argue why necessary conditions are more than the complement of sufficient conditions.

Necessary conditions 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 [21], 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.

__Susan:__ I would stop after the first sentence - the other points perhaps in related work

Necessary conditions are akin to monitor or object invariants[9, 15]. The difference between these and our holistic specifications is that object/monitor invariants can only reflect on the current state (*i.e.* the contents of the stack frame and the heap), while holistic specifications reflect on all aspects of execution.

In this paper we propose *Chainmail*, a specification language to express holistic specifications. *Chainmail* extends traditional program specification languages[11, 14], with features which talk about:

__Susan:__ These features sort of come from nowhere, so I wonder whether we even want them in the introduction. I think if you do then something more about open systems, (that what is critical is controlling access from foreign code and that you need to be able to talk about that in the specification language) should come first.

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

Control What object 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.

Authority Which objects' state or properties may change. This is useful in describing effects, such as reduction of funds.

Space This is about which parts of the heap are considered when establishing some property, or when performing program execution and is related to, but different from memory footprints and separation logics.

Time Assertions about the past or the future.

The design of *Chainmail* was guided by the study of a sequence of examples from the OCAP literature and the smart contracts world: the membrane, the DOM, the Mint/Purse, the Escrow, the DAO and ERC20. We were satisfied to see that the same concepts were used to specify examples from different contexts. Holistic assertions often have the form of a guarantee that if some property ever hods in the future then some other property holds now. For example, if within a certain heap some change is possible in the future, then this particular heap contains at least one object which has access to a specific other, privileged object. While many individual features of *Chainmail* can be found also in other work, we argue that their power and novelty for specifying open systems lies in their careful combination.

Sophia: *I think that James does not like this sentence. What could we say instead? Also, we need to stress that the selection of these features was not arbitrary.*

A module satisfies such a holistic assertion if, for all other modules, the assertion is satisfied in all runtime configurations reachable through execution of the two modules combined. 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,
- a further validation of *Chainmail* through informal proofs of adherence of code to some of these specifications.

The rest of the paper is organized as follows: Section 2 motivates our work in terms of an example. Sections A contain a formal definition of $\mathcal{L}_{oo}$, and Section 5 the semantics of assertions. Section ... related work .... Section xxxx concludes.

2 MOTIVATING EXAMPLE: THE BANK

We now consider a simplified banking application.

Sophia: *TODO: say the things that were in SLACK*

Sophia: *After discussion with Susan I think we need such a diagram. But the flow with the next para is not good.*

Traditional functional specifications describe what components are guaranteed to do. So long as a method is called in a state satisfying its preconditions, the method will complete its work and establish a state satisfying its postconditions. Thus, the pre-condition and the method call together form a *sufficient* condition for the method's effect.

Consider the specification of a trivial Bank component in fig. 2.

Susan: *component is a new word, you've used method, module, and object up to now*

The bank is essentially a wrapper for a map from account objects to account balances: given an instance of a Bank component, calling `newAccount` returns a new account object with an initial balance.

Given an account, calling `balance` with an account returns the account balance, and calling `deposit` with two accounts deposits funds from the source to the destination account.

Sophia: *kjxchanged it so it behaves more like a bank, rather an a currency: you open an account with a new balance. We can no longer talk about the preservation of a currency, but the amount in the bank must be the sum of arguments to newAccount. That is exactly what currency is and the Bank here is no different than the Mint.*

The specification in fig. 2 is enough to let us calculate the result of operations on the bank and the accounts — for example it is straightforward to determine that the code in fig. 3 satisfies its assertions: given that the `acm` object has a balance of 10,000 before an author is registered then afterwards it will have a balance of 11,000 while the `author` now has a balance of 500 from a starting balance of 1,500 (barely enough to buy a round of drinks at the conference hotel bar).

This reasoning is fine in a closed world, where we only have to consider complete programs, where all the code in our programs (or any other systems with which they interact) is under our control. In an open world, however, things are more complex: our systems will be made up of a range of components, many of which we do not control; and furthermore will have to interact with external systems which we certainly do not control. Returning to our author, say some time after registering by executing the code in fig. 3, they attempt to pay for a round at the bar. Under what circumstances can they be sure they have enough funds in their account?

James: *will update style sometime*

Sophia: *to add: FRESH(a) AND ledger.containsKey AND ledger.at(a)=n AND a:Account*

To see the problem, consider the additional function specified in fig. 4. This method says the bank additionally provides a `steal` method that empties out every account in the bank and puts all their funds into the thief's account. If this method exists, and if it is somehow called between registering at the conference and going to the bar, the author (actually everyone using the same bank) will find all their accounts empty (except the thief, of course).

Sophia: *"Map" should be a mathematical construction, not syntax. And why does "at" mean lookup*

Susan: *can we have 2 methods*

```
specification Bank {

  ghost field ledger : Map[Account, Number]

  policy newAccount {
      a : Account, b : Bank, n : Number
        { def a ;= b.newAccount(n) }
      FRESH(a) & ledger.containsKey(a)
  }

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

  policy deposit {
    src, dst : Account, b : Bank, sb, db, n : Number
    sb == ledger.at(src), db == ledger.at(dst), n > 0, src=/=dst
      { b.deposit(dst, src, n) }
    ledger.at(src) == sb - n
    ledger.at(dst) == db + n
  }
}
```

Fig. 2. Functional specification of a Bank

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

b.deposit(acm, author, 1000)

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

Fig. 3. Registering at a Conference

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

One obvious solution would be to return to a closed-world interpretation of specifications: we interpret specifications such as fig. 2 as *exact* in the sense that only implementations that meet the functional specification exactly, *with no extra methods or behaviour*, are considered as suitable implementations of the functional specification. The problem is that this solution is far too strong: it would for example rule out a bank that during maintenance was given a new method that simply counted the number of deposits that had taken place, i.e. met fig. 5 as well as fig. 2.

What we need is some way to permit bank implementations that meet fig. 5 but to forbid implementations that meet fig. 4. The key here is to capture the (implicit) assumptions underlying

```
specification Theft {

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

Fig. 4. Sufficient Specification of Theft

```
specification CountDeposits {

  ghost field count : Number = 0

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

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

Fig. 5. Functional specification counting the number of deposits

fig. 2, and to provide additional specifications that capture those assumptions. There are at least two assumptions that can prevent methods like `steal`:

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

Susan: *point out that this is not sufficient to stop steal because it could be rewritten to use deposit*

Compared with the functional specification we have seen so far, these assumptions capture *necessary* conditions rather than *sufficient* conditions. It is necessary that the `deposit` method is called to change an account's balance, and it is necessary that the particular account object can be passed as a parameter to that method. The fig. 4 specification is not consistent with these assumptions, while the. 5 specification is consistent with these assumptions.

Sophia: *Chopped this as I believe we have said earlier:*

Below we express these two informal requirements in *Chainmail*. Rather than specifying the behaviour of particular methods when they are called, we write policies that range across the entire behaviour of the component.

James: *NEED TO DECIDE ON CONTRIBUTIONS. The contribution of this paper is a specification langauge and semantics that can be used to specify necessary specifications, and*

```
specification Robust-Bank {

  policy call-deposit {
    ∀ a : Account, S : Footprint
      this != a ∧With⟨ S, (Will⟨Change⟨⟩⟩a.balance)⟩    ⟶
          ∃o. [o ∈ S  ∧  Calls⟨deposit⟩  ∧  o ∉ Internal(a) ]
  }

  policy access-account {
    ∀ a : Account, S : Footprint
      this != a ∧With⟨ S, (Will⟨Change⟨⟩⟩a.balance)⟩    ⟶
          ∃o. [o ∈ S  ∧  Access⟨o, a⟩  ∧  o ∉ Internal(a) ]    ]
  }
}
```

Fig. 6. Necessary specifications for `deposit` – James version

$$
\begin{array}{l}
(1)\ \triangleq\ \forall \texttt{a}:\texttt{Account}\ [\ \mathit{Change}\langle \texttt{a.balance}\rangle \longrightarrow \\
\qquad\qquad\qquad \exists \texttt{o}.[\ \mathcal{W}\!\mathit{as}\langle \mathit{Calls}\langle \texttt{o}, \texttt{deposit}, \texttt{a}, \_, \_\rangle\rangle\ \vee\ \mathcal{W}\!\mathit{as}\langle \mathit{Calls}\langle \texttt{o}, \texttt{deposit}, \_, \texttt{a}, \_\rangle\rangle\ ]\ ] \\
\\
(2)\ \triangleq\ \forall \texttt{a}:\texttt{Account}\ .\forall \texttt{S}:\texttt{Set}.\ [\ \mathcal{W}\!\mathit{ith}\langle\ \texttt{S},\ (\mathcal{W}\!\mathit{ill}\langle\ \mathit{Change}\langle \texttt{a.balance}\rangle\rangle)\ \rangle \longrightarrow \\
\qquad\qquad\qquad\qquad \exists \texttt{o}.\ [\texttt{o} \in \texttt{S}\ \wedge\ \mathcal{A}\mathit{ccess}\langle \texttt{o}, \texttt{a}\rangle\ \wedge\ \mathcal{E}\mathit{xternal}\langle \texttt{o}\rangle\ ]\ ]
\end{array}
$$

Fig. 7. Necessary specifications for `deposit` – Sophia's version

$$
\begin{array}{l}
(1)\ \triangleq\ \forall \texttt{a}:\texttt{Account}\,[\ \ \mathit{Change}\langle \texttt{a.balance}\rangle\ \ \longrightarrow \\
\qquad\qquad \exists \texttt{o}.[\ \mathcal{W}\!\mathit{as}\langle \mathit{Calls}\langle \texttt{o}, \texttt{deposit}, \texttt{a}, \_, \_\rangle\rangle \vee \mathcal{W}\!\mathit{as}\langle \mathit{Calls}\langle \texttt{o}, \texttt{deposit}, \_, \texttt{a}, \_\rangle\rangle\ ]\ ] \\
\\
(2)\ \triangleq\ \forall \texttt{a}:\texttt{Account}\,.\forall \texttt{S}:\texttt{Set}.\,[\ \mathcal{W}\!\mathit{ith}\langle\ \texttt{S},\ (\mathcal{W}\!\mathit{ill}\langle\ \mathit{Change}\langle \texttt{a.balance}\rangle\rangle)\ \rangle\ \ \longrightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \exists \texttt{o}.\,[\ \texttt{o} \in \texttt{S}\ \wedge\ \mathcal{A}\mathit{ccess}\langle \texttt{o}, \texttt{a}\rangle\ \wedge\ \mathcal{E}\mathit{xternal}\langle \texttt{o}\rangle\ ]\ \ \ ]
\end{array}
$$

Sophia: *We have to explain that the two objects above may be different .*

Policy (1) says that if an account's balance is changed ($\mathit{Change}\langle \texttt{a.balance}\rangle$) then there must be some client object `o` that in the past ($\mathcal{W}\!\mathit{as}\langle ...\rangle$) called the `deposit` method with `a` as a receiver or an an argument ($\mathit{Calls}\langle \texttt{o}, \texttt{deposit}, \_, \_\rangle$).

Sophia: *We no longer need "which is outside the bank and its associated accounts (`o ∉ Internal(a)`)" – and I have chopped it from the spec. This is because of the visible states :-)*

Policy (2) similarly constrains any possible change to an account's balance: If at some future point the balance changes ($\mathcal{W}\!\mathit{ill}\langle ...\rangle$) and if the footprint of the execution that brings about this change is the set of objects in `S` (*i.e.* $\mathcal{W}\!\mathit{ith}\langle\ \texttt{S},\ ...\rangle$), then at least one of these objects (`o`) has (direct) access to that account object ($\mathcal{A}\mathit{ccess}\langle \texttt{o}, \texttt{a}\rangle$).

Sophia: *I propose that we change deposit so that it is called on the account object.*

Sophia: *chopped "but requires that the client object making the call has direct access" because a) the spec did not say it, and b) it is not the case.*

Sophia: *Where shall we say this?*

Sophia: *We no longer need this!=a*

A holistic specification for the bank account, then, would be our original sufficient functional specification from fig. 2 plus the necessary security policy specification in fig. 7. We swill discuss the meaning of the policies in more detail in the next section. This holistic specification permits an implementation of the bank that also meets the `count` specification from fig. 5, but does not permit an implementation that also meets the `steal` specification from fig. 4.

We can then prove that e.g. the `steal` method from fig. 4 is inconsistent with both of these policies. First, the `steal` method clearly changes the balance of every account in the bank, but policy (1) requires that any method that changes the balance of any account must be called `deposit`. Second, the `steal` method changes the balance of every account in the system, and will do so without the called having a reference to most of those accounts, which breaches policy (2). Note that `steal` putting all the funds into the thief's account does not breach policy (2) with respect to the thief's own account, because that account is passed in as a parameter to the `steal` method, and so the called of the `steal` must have access to that account.

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

3 *Chainmail* OVERVIEW

In this Section we give a brief and informal overview of *Chainmail*– a full exposition appears in Section 5. As well as "classical" assertions about variables and the heap (*e.g.* `a1.myBank` = `a2.myBank`), *Chainmail* incorporates assertions about *access*, *control* , *authority*, *space* , and *time*.

***Sophia:** would be nice iof we had a better name*

Configurations We will explain these concepts in terms of examples coming from `Bank`/`Account` as in the previous Section. We will use the runtime configurations σ_1 and σ_2 shown in the left and right diagrams in Figure 8. In both diagrams the rounded boxes depict objects: green for those from the `Bank`/`Account` module, and grey for the "external", "client" objects. The transparent green rectangle shows which objects `Bank`/`Account` module. 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 than those from the `Bank`/`Account` module.

The configurations differ in the internal representation of the objects. Configuration σ_1 may arise from execution using a module M_{BA1}, where `Account` objects of have a field `myBank` pointing to their `Bank`, and an integer field `balance` – the code can be found in xxx.. Configuration σ_2 may arise from execution using a module M_{BA2}, where `Accounts` have a `myBank` field, `Bank` objects have a `ledger` implemented though a sequence of `Nodes`, each of which has a field pointing to the `Account`, a field `balance`, and a field `next` – the code can be found in yy..

***Sophia:** TODO add – code is available somewhere*

***Sophia:** TDOD add – code is available somewhere*

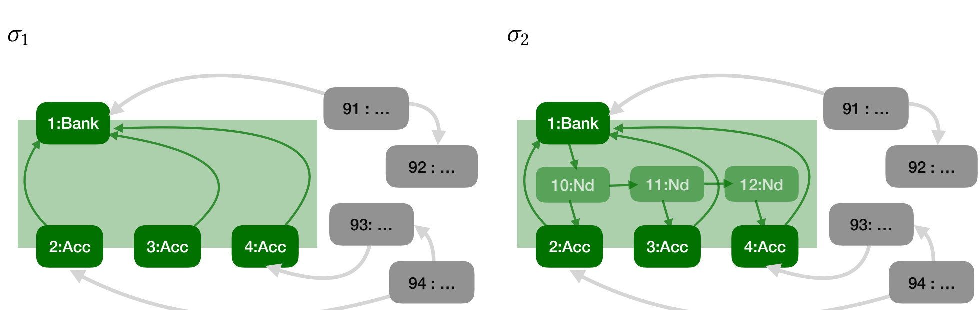


Fig. 8. Two runtime configurations for the `Bank`/`Account` example.

For the rest, assume variable identifiers $\texttt{b}_1$, and $\texttt{a}_2$–$\texttt{a}_4$, and $\texttt{u}_{91}$–$\texttt{u}_{94}$ denoting objects `1`, `2–4`, and `91–94` respectively for both σ_1 and σ_2. That is, $\sigma_i(\texttt{b}_1)$=`1`, and $\sigma_i(\texttt{a}_2)$=`2`, $\sigma_i(\texttt{a}_3)$=`3`, $\sigma_i(\texttt{a}_4)$=`4`, and $\sigma_i(\texttt{u}_{91})$=`91`, $\sigma_i(\texttt{u}_{92})$=`92`, $\sigma_i(\texttt{u}_{93})$=`90`, $\sigma_i(\texttt{u}_{94})$=`94`, for i=1 or i=2.

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 that $\texttt{a}_2$`.myBank=`$\texttt{a}_3$`myBank`,

***Sophia:** I think this para is great – not sure it belongs here. While traditional policies are expressed as Hoare triples — often describing a single method invocation on an instance of the class being specified (as in XXXXX), Rholistic* expresses that $\mathtt{a}_1$ and $\mathtt{a}_2$ have the same bank. In fact, this assertion is satisfied in both σ_1 and σ_2, written formally as

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

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

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

Sophia: TODO citation

We support ghost fields,*e.g.* $\mathtt{a}_1$`.balance` is a ghost field in σ_2 since `Accounts` do not store a `balance` field. But its value can be defined so that for any `a` of class `Account` the value of `a.balance` is `nd.balance` such that `nd` is a `Node`, and `nd.myAccount=a`.

Sophia: All hell is loose here, as ghostfields require recursive defs, but I want to postpone these.

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

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

Permission: Access

Our first holistic assertion $\mathcal{A}ccess\langle \mathtt{x}, \mathtt{y}\rangle$ asserts that one 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:

$$..., \sigma_1 \models \mathcal{A}ccess\langle \mathtt{a}_2, \mathtt{b}_1\rangle$$

This assertion can be used to make assertions about heap structures (and thus object values), for example if a `cacheValid` field is true, then that object can access to a cached value:

```
a.cacheValid == true `≲ Access⟨(, a⟩)(cacheValue)`
```

James: do we really not also need reachable (transitive closure of access)

Susan: we don't seem to need transitivity in any of the examples

Control: Calls

The $Calls\langle \mathtt{x}\rangle \mathtt{ymz}$ assertion is more-or-less the control flow analogue of the access assertion, and is true in program states where a method on object `x` makes a method call `y.m(z)` — that is it calls method `m` on object `y` with arguments `z`.

```
another illustrative example, presumably in the context of the
running example from the intro. which we need to pick first.
```

Authority Changes. The $Change\langle \mathtt{x.f}\rangle$ assertion is true when the value of `x.f` in the next state is different to the value in the current state. For example, we

```
another illustrative example, presumably in the context of the
running example from the intro. which we need to pick first.
```

Space: With. The space assertionn $\mathcal{W}ith\langle\ S,\ A\rangle$ states the some assertion A is true when the heap used by that assertion is restrcited to the footprint § .

James: footprint F?

James: OK someone needs to explain what that for and when/why it is sound to do it

```
another illustrative example, presumably in the context of the
running example from the intro. which we need to pick first.
```

Time: Next, Will, Prev, Was. *Chainmail* supports several temporal operators familiar from temporal logic ($\mathcal{W}ill\langle A\rangle$ or $\mathcal{W}as\langle A\rangle$ or $\mathcal{N}ext\langle A\rangle$ or $\mathcal{P}rev\langle\$\rangle A$). We support birectional temporal assertions, constraining either future ($\mathcal{W}ill\langle A\rangle$. $\mathcal{N}ext\langle A\rangle$) or past behaviour ($\mathcal{W}as\langle A\rangle$. $\mathcal{P}rev\langle\$\rangle A$) either considering only the immediate next or immediate previous step ($\mathcal{N}ext\langle A\rangle$, $\mathcal{P}rev\langle\$\rangle A$) or for conditions that become eventually true in some distant future, or were true once in some distant past ($\mathcal{W}ill\langle A\rangle$, $\mathcal{W}as\langle A\rangle$). We have bidirectional pairs of operators to give expressiveness in writing assertions: this does not offer any additional reasoning power.

For example, a part of the observer pattern is that when a subject is notified of a change, then the observer must be told to update itself. We can write this from the subject's perspective, looking forwards:

```
Call(_,subject,notify,_) --> Will(Call(subject,observer,update,_))
```

meaning that once notify is called on a subejct, then its observer will be updated sometime in the future. We can write a very similar specification for an observer, looking backwards.

```
Call(subject,observer,update,_) --> Was(Call(_,subject,notify,_))
```

meaning that if a subject updates an observer, that subject have been notified sometime previously. We could tighten each specifaction, so that the update must immediately follow the notification, by replacing $\mathcal{W}ill\langle A\rangle$ or $\mathcal{W}as\langle A\rangle$ with $\mathcal{N}ext\langle A\rangle$ or $\mathcal{P}rev\langle A\rangle$.

These assertions draw from some concepts from object capabilities ($\mathcal{A}ccess\langle\_,\_\rangle$ for permission and $Change\langle\_\rangle$ for authority) as well as temporal logic ($\mathcal{W}ill\langle A\rangle$, $\mathcal{W}as\langle A\rangle$ and friends), and the relation of our spatial connective ($\mathcal{W}ith\langle\ S,\ A\rangle$) with ownership and effect systems. . .

James: bother. should probably do time earlier, because most of the example assertions I can think of a things like `(change (x.f) -> past (call (_, xm, _)` *which needs the temporal operator*

4 OVERVIEW OF THE *Chainmail* FORMAL MODEL

Having outlined the ingredients of our holistic specification language, the next question to ask is: When does a module M satisfy such a holistic assertion A? Note that we use the term *module* to talk about repositories of code; in this work modules are mappings from class identifiers to class definitions. So, the question about modules satisfying assertions put formally is, when does

$$\mathtt{M} \models A$$

hold?

Sophia: While many individual features of Chainmail *can be found also in other work, we argue that their power and novelty for specifying open systems lies in their careful combination*

Our answer has to reflect the fact that we are dealing with the *open world*, where M, our module, may be linked with *arbitrary untrusted code*. To reflect this we consider pairs of modules, $\mathtt{M}\,\mathring{9}\,\mathtt{M}'$, where M is the module whose code is supposed to satisfy the assertion, and $\mathtt{M}'$ is another module which exercises the functionality of M. We call M the *internal*, and $\mathtt{M}'$ is the *external* module.

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

James: Hmm. a delicate, subtle argument . . .

$$\mathtt{M} \models A \qquad \text{if} \qquad \forall \mathtt{M}'.\forall\sigma \in \mathcal{A}rising(\mathtt{M}\,\mathring{9}\,\mathtt{M}').[\ \mathtt{M}\,\mathring{9}\,\mathtt{M}',\sigma \models A\ ].$$

Sophia: TODO: add references here

In that sense, module $\mathtt{M}'$ represents all possible clients of M; and as it is arbitrarily chosen, it reflects the open world nature of our specifications.

James: don't forget the whole AOP mionitoring/specs stuff, just for fun. (HELM contracts!!! (vs Meyer Contracts :-)

The judgement $\mathtt{M}\,\mathring{9}\,\mathtt{M}',\sigma \models A$ means that assertion A is satisfied by $\mathtt{M}\,\mathring{9}\,\mathtt{M}'$ and σ. As in traditional specification languages [11, 14], satisfaction is judged in the context of runtime configuration σ; but in addition, it is judged in the context of modules. The reason for this is that assertions may talk about possible future configurations. To determine the possible future configurations we need the class definitions – these are found in the modules.

James: somewhere, should we say something like: the goal is to allow wholistic specifations with as extra little machinery as possible over a basic Hoare language

Sophia: TODO connect with example earlier

Note the distinction between the internal and the external module. The reason for this distinction is some assertions require object to be *external*,and also, because we model progrm execution as if all executions within a modue were atomic. We only record runtime configurations which are *external* to module $\mathtt{M}$, *i.e.* those where the executing object (*i.e.* the current receiver) comes from module $\mathtt{M}'$. Thus, program execution is a judgment of the form

$$\mathtt{M} \,\mathring{9}\, \mathtt{M}', \sigma \rightsquigarrow \sigma'$$

Sophia: TODO: add references here.

we ignore all intermediate steps whose receivers are internal to $\mathtt{M}$. Thus, our executions correspond to some form of visible states semantics. Similarly, when considering $\mathcal{A}rising(\mathtt{M} \,\mathring{9}\, \mathtt{M}')$, *i.e.* the configurations arising from executions in $\mathtt{M} \,\mathring{9}\, \mathtt{M}'$, we can take method bodies defined in $\mathtt{M}$ or in $\mathtt{M}'$, but we will only consider the runtime configurations which are external to $\mathtt{M}$.

As a notational convenience, we keep the code to be executed as a component of the runtime configuration. Thus, σ consists of a stack of frames and a heap, and each frame consists of a variable map and a continuation. The variable map is a mapping from variables to addresses or to set of addresses – the latter are needed to deal with assertions which quantify over footprints, as *e.g.* (1) and (2) from section 2.

To give meaning to assertions with footprint restrictions such as *e.g.* $\mathcal{W}ith\langle\ \mathtt{S},\ A\rangle$,we define restrictions on the configuration. Thus $\sigma\downarrow_{\sigma(\mathtt{S})}$ is the same as σ but with the domain of the heap restricted to the addresses from $\sigma(\mathtt{S})$. And then we define

$$\mathtt{M} \,\mathring{9}\, \mathtt{M}', \sigma \models \mathcal{W}ith\langle\ \mathtt{S},\ A\rangle \quad \text{if} \quad \mathtt{M} \,\mathring{9}\, \mathtt{M}', \sigma\downarrow_{\sigma(\mathtt{S})} \models A$$

The meaning of assertions therefore may depend on the variable map, eg $\mathtt{x}$ may be pointing to a different object in .... TODO The treatment of time in combination with the fact that the meaing od assertions TODO

5 ASSERTIONS

We now define the syntax of expressions and assertions.

5.1 Syntax of Assertions

DEFINITION 1 (ASSERTIONS). *The syntax of expressions ($\mathtt{e}$) and assertions (A) is:*

$$\mathtt{e} \quad ::= \quad \mathtt{true} \mid \mathtt{false} \mid \mathtt{null} \mid \mathtt{x} \mid \mathtt{e.f}$$

$$\begin{aligned} A \quad ::= \quad & \mathtt{e} \mid \mathtt{e} = \mathtt{e} \mid \mathtt{e} : \mathtt{ClassId} \mid \mathtt{e} \in \mathtt{S} \mid \\ & A \rightarrow A \mid A \wedge A \mid A \vee A \mid \neg A \mid \forall \mathtt{x}.A \mid \forall \mathtt{S} : SET.A \mid \exists \mathtt{x}.A \mid \exists \mathtt{S} : SET.A \mid \\ & \mathcal{E}xternal\langle x\rangle \mid \mathcal{A}ccess\langle x, y\rangle \mid \mathcal{C}hange\langle e\rangle \mid \mathcal{C}alls\langle \mathtt{x}, \mathtt{y}, \mathtt{m}, \mathtt{z}\rangle \mid \\ & \mathcal{N}ext\langle A\rangle \mid \mathcal{W}ill\langle A\rangle \mid \mathcal{P}rev\langle A\rangle \mid \mathcal{W}as\langle A\rangle \end{aligned}$$

[1]

As we discussed in section TODO validity of assertions has the format $\mathtt{M} \,\mathring{9}\, \mathtt{M}', \sigma \models A$, where $\mathtt{M}$ is the internal module, whose internal workings are opaque to the external, client module $\mathtt{M}'$. We break the definition into four parts: In definition 3 we define validity of basic assertions which reflect over the contents of the frame or the heap. In definition 4 we define validity of basic assertions which reflect over the contents of the frame or the heap.

5.2 Satsfaction of Assertions - standard

DEFINITION 2 (INTERPRETATIONS FOR SIMPLE EXPRESSIONS). *For any runtime configuration, σ, and any $k \in \mathbb{N}$, and any simple expression, $\mathtt{e}$, we define its interpretation as follows:*

[1]Note that the operators $\wedge$, $\vee$, $\neg$ and $\forall$ could have been defined through the usual shorthands, *e.g.* , $\neg A$ is short for $A \rightarrow \mathtt{false}$ *etc.*, but here we give full definitions instead.SD: Perhaps we should just do that, it make the defs implicit

- $\lfloor \texttt{true} \rfloor_\sigma \triangleq \texttt{true}$, *and* $\lfloor \texttt{false} \rfloor_\sigma \triangleq \texttt{false}$, *and* $\lfloor \texttt{null} \rfloor_\sigma \triangleq \texttt{null}$
- $\lfloor \texttt{x} \rfloor_\sigma \triangleq \phi(\texttt{x})$ *if* $\sigma{=}(\phi \cdot \_, \_)$
- $\lfloor \texttt{e.f} \rfloor_\sigma \triangleq \chi(\lfloor \texttt{e} \rfloor_\sigma, \texttt{f})$ *if* $\sigma{=}(\_, \chi)$

LEMMA 5.1 (INTERPRETATION CORRESPONDS TO EXECUTION). *For any simple expression* e, *module* M, *runtime configuration* σ, *and value* v:

- $\lfloor \texttt{e} \rfloor_\sigma = v$ *if and only if* $\texttt{M}, \sigma[\texttt{contn} \mapsto \texttt{e}] \leadsto v$.

PROOF. by structural induction over the definition of e. □

DEFINITION 3 (BASIC ASSERTIONS). *We define when a configuration satisfies basic assertions, consisting of expressions.*

- $\texttt{M} ⨾ \texttt{M}', \sigma \models \texttt{e}$ *if* $\lfloor \texttt{e} \rfloor_\sigma = \texttt{true}$.
- $\texttt{M} ⨾ \texttt{M}', \sigma \models \texttt{e} = \texttt{e}'$ *if* $\lfloor \texttt{e} \rfloor_\sigma = \lfloor \texttt{e}' \rfloor_\sigma$.
- $\texttt{M} ⨾ \texttt{M}', \sigma \models \texttt{e} : \texttt{ClassId}$ *if* $Class(\lfloor \texttt{e} \rfloor_\sigma)_\sigma = \texttt{ClassId}$.
- $\texttt{M} ⨾ \texttt{M}', \sigma \models \texttt{e} \in \texttt{S}$ *if* $\lfloor \texttt{e} \rfloor_\sigma \in \lfloor \texttt{S} \rfloor_\sigma$.

We now define satisfaction of assertions which involve logical connectives and existential or universal quantifiers.

DEFINITION 4 (ASSERTIONS WITH LOGICAL CONNECTIVES AND QUANTIFIES). *We now consider For modules* M, M′, *assertions* A, A', *variables* x *and* S, *configuration* σ, *we define*:

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

5.3 Satisfaction of Assertions - Space

And now, we consider the assertions which involve space and control:

DEFINITION 5 (SATISFACTION OF ASSERTIONS ABOUT SPACE-1). *For any modules* M, M′, *assertions* A, A', *variables* x *and* S, *we define*

- $\texttt{M} ⨾ \texttt{M}', \sigma \models \mathcal{A}ccess\langle \texttt{x}, \texttt{y} \rangle$ *if*
  - $\lfloor \texttt{x} \rfloor_\sigma {=} \lfloor \texttt{y} \rfloor_\sigma$, *or*
  - $\lfloor \texttt{x.f} \rfloor_\sigma {=} \lfloor \texttt{y} \rfloor_\sigma$ *for some field* f, *or*
  - $\lfloor \texttt{x} \rfloor_\sigma {=} \lfloor \texttt{this} \rfloor_\sigma$ *and* $\lfloor \texttt{y} \rfloor_\sigma {=} \lfloor \texttt{z} \rfloor_\sigma$, *and* z *appears in* σ.contn.
- $\texttt{M} ⨾ \texttt{M}', \sigma \models Calls\langle \texttt{x}, \texttt{y}, \texttt{m}, \texttt{z} \rangle$ *if* $\sigma.\texttt{contn}{=}\texttt{u.m(v)};\_$ *for some variables* u *and* v, *and*
  $\lfloor \texttt{this} \rfloor_\sigma {=} \lfloor \texttt{x} \rfloor_\sigma$, *and* $\lfloor \texttt{y} \rfloor_\sigma {=} \lfloor \texttt{u} \rfloor_\sigma$, *and* $\lfloor \texttt{z} \rfloor_\sigma {=} \lfloor \texttt{v} \rfloor_\sigma$.
- $\texttt{M} ⨾ \texttt{M}', \sigma \models \mathcal{W}ith\langle\, \texttt{S},\ A \rangle$ *if* $\texttt{M} ⨾ \texttt{M}', \sigma\downarrow_{\texttt{S}} \models A$.
- $\texttt{M} ⨾ \texttt{M}', \sigma \models \mathcal{E}xternal\langle \texttt{e} \rangle$ *if* $Class(\lfloor \texttt{e} \rfloor_\sigma)_\sigma \notin dom(\texttt{M})$

$\mathcal{A}ccess\langle \texttt{x}, \texttt{y} \rangle$ expresses that x has a *direct* path to y. It says that in the current frame, either xand y are aliases, or xpoints to an object which has a field whose value is the same as that of y, or x is the currently executing object and y is a local variable or formal parameter z which appears in the code in the continuation (σ.contn). The latter requirement ensusres that that variables which were introduced into the variable map in order to give meaning to existentially quantified assertions are not considered.

On the other hand, an assertion of the form $\mathcal{W}ith\langle\ S,\ A\rangle$ promises that A holds in subconfiguration, whose heap is restricted to the objects from S.

DEFINITION 6 (RESTRICTION OF RUNTIME CONFIGURATIONS). *The restriction operator* $\downarrow$ *applied to a runtime configuration* σ *and a set R is defined as follows:*

- $\sigma\downarrow_{\mathtt{S}}\ \triangleq\ (\psi,\chi'),\quad if\quad \sigma{=}(\psi,\chi),\ and\ dom(\chi') = \lfloor \mathtt{S}\rfloor_\sigma,\ and\ \forall\alpha\in dom(\chi').\chi(\alpha) = \chi'(\alpha)$ ■[2]

DEFINITION 7 (SATISFACTION OF ASSERTIONS ABOUT SPACE-2). *For any modules* M, M′*, assertion A, set variable* S*, and configuration* σ*, we define*

- $\mathtt{M}\,\mathring{,}\,\mathtt{M}',\sigma \models \mathcal{W}ith\langle\ \mathtt{S},\ A\rangle \quad if \quad \mathtt{M}\,\mathring{,}\,\mathtt{M}',\sigma\downarrow_{\mathtt{S}} \models A.$

Perhaps $\mathcal{W}ith\langle\ \mathtt{S},\ A\rangle$ is the most intriguing of our holistic assertions. It allows us to restrict the set of objects that are considered when ...

5.4 Satisfaction of Assertions - Time

Finally, we consider assertions involving time. To do this, we need an auxiliary concept: $\triangleleft$ the adaptation of a runtime configuration to the scope of another one. This operator is needed to the changes of scope during execution. For example, the assertion $\mathcal{W}ill\langle \mathtt{x.f} = 3\rangle$ is satisfied in the *current* configuration if in some *future* configuration the field f of the object that is pointed at by x in the *current* configuration has the value 3. Note that in the future configuration, x may be pointing to a different object, or may even no longer be in scope (*e.g.* if a nested call is executed). Therefore, we introduce the operator $\triangleleft$, which combines runtime configurations: $\sigma\triangleleft\sigma'$ adapts the second configuration to the top frame's view of the former: it returns a new configuration whose stack has the top frame as taken from σ and where the contn has been consistently renamed, while the heap is taken from σ'. This allows us to interpret expressions in the newer (or older) configuration σ' but with the variables bound according to the top frame from σ; *e.g.* we can obtain that value of x in configuration σ' 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 $\mathcal{W}ill\langle \mathtt{x.f} = 3 \wedge \mathcal{W}ill\langle \mathtt{x.f} = 5\rangle\rangle$

DEFINITION 8 (ADAPTATION OF RUNTIME CONFIGURATIONS). *For runtime configurations* σ, σ'.:

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

That is, in the new frame ϕ'' from above, we keep the same continuation as from σ' but rename all variables with fresh names *prgzs*′, and in the variable map we comnine that from σ and σ' but avoid names clashes through the renaming $[\mathtt{zs}' \mapsto \beta'(\mathtt{zs})]$. With this auxiliary definition, we can now define satisfaction of assertions with involve time:

[2]SD: I had written instead $[Class(\alpha)_{\chi'} = Class(\alpha)_\chi\ \wedge\ \forall \mathtt{f}.\chi'(\alpha,\mathtt{f}) = \chi(\alpha,\mathtt{f})\ ]$, but I do not see why

DEFINITION 9 (ASSERTIONS OVER TIME). *For any modules* $\mathtt{M}$, $\mathtt{M}'$, *assertions* A, A', *variables* $\mathtt{x}$ *and* $\mathtt{S}$, *we define*

- $\mathtt{M}\,;\,\mathtt{M}', \sigma \models \mathit{Change}\langle \mathtt{e} \rangle$ *if* $\exists \sigma'.[\ \mathtt{M}\,;\,\mathtt{M}', \sigma \rightsquigarrow \sigma' \ \wedge \lfloor e \rfloor_\sigma \neq \lfloor e \rfloor_{\sigma \triangleleft \sigma'}\ ]$.
- $\mathtt{M}\,;\,\mathtt{M}', \sigma \models \mathcal{N}ext\langle A \rangle$ *if* $\exists \sigma'.[\ \mathtt{M}\,;\,\mathtt{M}', \phi \rightsquigarrow \sigma' \ \wedge \mathtt{M}\,;\,\mathtt{M}', \sigma \triangleleft \sigma' \models A\ ]$,
  and where ϕ *is so that* $\sigma = (\phi \cdot \_, \_)$.
- $\mathtt{M}\,;\,\mathtt{M}', \sigma \models \mathcal{W}ill\langle A \rangle$ *if* $\exists \sigma'.[\ \mathtt{M}\,;\,\mathtt{M}', \phi \rightsquigarrow^* \sigma' \ \wedge \mathtt{M}\,;\,\mathtt{M}', \sigma \triangleleft \sigma' \models A\ ]$,
  and where ϕ *is so that* $\sigma = (\phi \cdot \_, \_)$.
- $\mathtt{M}\,;\,\mathtt{M}', \sigma \models \mathcal{P}rev\langle A \rangle$ *if* $\forall \sigma_1, \sigma_2.[\ \mathcal{I}nitial\langle \sigma_1 \rangle \ \wedge\ \mathtt{M}\,;\,\mathtt{M}', \sigma \rightsquigarrow^* \sigma_2 \ \wedge\ \mathtt{M}\,;\,\mathtt{M}', \sigma_2 \rightsquigarrow \sigma$
  $\longrightarrow\ \mathtt{M}\,;\,\mathtt{M}', \sigma \triangleleft \sigma_2 \models A\ ]$[3]
- $\mathtt{M}\,;\,\mathtt{M}', \sigma \models \mathcal{W}as\langle A \rangle$ *if* $\forall \sigma_1, ... \sigma_n.[\ \mathcal{I}nitial\langle \sigma_1 \rangle \ \wedge\ \sigma_n = \sigma \ \wedge\ \forall i \in [1..n).\mathtt{M}\,;\,\mathtt{M}', \sigma_i \rightsquigarrow \sigma_{i+1}$
  $\longrightarrow\ \exists j \in [1..n-1).\mathtt{M}\,;\,\mathtt{M}', \sigma \triangleleft \sigma_j \models A\ ]$[4]

Thus, $\mathtt{M}\,;\,\mathtt{M}', \sigma \models \mathcal{W}ill\langle A \rangle$ holds if A holds in some configuration σ' which arises from execution of ϕ, where ϕ is the top frame of σ. By requiring that $\phi \rightsquigarrow^* \sigma'$ rather than $\sigma \rightsquigarrow^* \sigma'$ we are restricting the set of possible future configurations to just those that are caused by the top frame. Namely, we do not want to also consider the effect of enclosing function calls. This allows us to write more natural specifications when giving necessary conditions for some future effect.

5.5 Entailment and Equivalence

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 10 (EQUIVALENCE AND ENTAILMENTS OF ASSERTIONS).

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

LEMMA 5.2 (ASSERTIONS ARE CLASSICAL-1). *For all runtime configurations* σ, *assertions* A *and* A', *and modules* $\mathtt{M}$ *and* $\mathtt{M}'$, *we have*

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

PROOF. By application of the corresponding definitions from **??**. □

.

LEMMA 5.3 (ASSERTIONS ARE CLASSICAL-2). *For assertions* A, A', *and* A'' *the following equivalences hold*

(1) $A \wedge \neg A \ \equiv\ \mathtt{false}$
(2) $A \vee \neg A \ \equiv\ \mathtt{true}$
(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) $(A \wedge A') \vee A'' \ \equiv\ (A \vee A') \wedge (A \vee A'')$

[3]past includes the present, perhaps change this
[4]past includes the present, perhaps change this

(8) $\neg(A \wedge A') \equiv \neg A \vee \neg A''$
(9) $\neg(A \vee A') \equiv \neg A \wedge \neg A''$
(10) $\neg(\exists \mathtt{x}.A) \equiv \forall \mathtt{x}.(\neg A)$
(11) $\neg(\exists k : \mathbb{N}.A) \equiv \forall k : \mathbb{N}.(\neg A)$
(12) $\neg(\exists \mathtt{fs} : FLD^k.A) \equiv \forall \mathtt{fs} : FLD^k.(\neg A)$
(13) $\neg(\forall \mathtt{x}.A) \equiv \exists \mathtt{x}.\neg(A)$
(14) $\neg(\forall k : \mathbb{N}.A) \equiv \exists k : \mathbb{N}.\neg(A)$
(15) $\neg(\forall \mathtt{fs} : FLD^k.A) \equiv \exists \mathtt{fs} : FLD^k.\neg(A)$

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

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

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 $\mathtt{M} \,⨾\, \mathtt{M}'$, the assertion A holds.

DEFINITION 11. *For any module* `M`*, and assertion A, we define:*

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

6 ANOTHER EXAMPLE – ATTENUATING THE DOM

Sophia: This section now needs to be updated, as it has moved to later in the paper

Attenuation is 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 [7] — are a common design pattern, and their security properties and have been studied at length in the object capabilities literature [16, 18], and Devrise et. al. proposed specifications for attenuation for the DOM [6].

In this section we revisit that example, and use it to motivate the need for holistic specifications, and to give an informal introduction to our for holistic language Chainmail II. We also argue that compared with Devrise et al., our specifications xxxx. .

Sophia: TO COMPLETE

This example deals with a tree of DOM nodes. Access to a DOM node gives access to all its parent and children nodes, and the ability to modify the 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 via attenuation through a `Wrapper`, which has a field `node` pointing to a `Node`, and a field `height` which restricts the range of `Node`s which may be modified through the use of the particular `Wrapper`. Namely, when you hold a `Wrapper` you can modify the `property` of all the descendants of the `height`-th ancestors of the `node` of that particular `Wrapper`. It is not difficult to write such a `Wrapper`; a possible implementation appears in Figure **??** in appendix **??**.

Figure 9 shows `Wrapper` objects attenuating the use of `Node`s. The function `usingWrappers` has as parameter an object of unknown provenance, here called `unknwn`. On lines 2-7 we create a tree consisting of nodes `n1`, `n2`, ... `n6`, depicted as blue circles on the right-hand-side of the Figure. On line 8 we create a wrapper of `n5` with height `1`. This means that the wrapper `w` may be used to modify `n3`, `n5` and `n6` (*i.e.* the objects in the green triangle), while it cannot be used to modify `n1`, `n2`, and `4` (*i.e.* the objects within the blue triangle). On line 8 we call a function named `untrusted` on the `unknown` object, and pass `w` as argument.

```
1  func usingWrappers(unknwn){
2      n1=Node(null,"fixed");
3      n2=Node(n1,"robust");
4      n3=Node(n2,"volatile");
5      n4=Node(n2,"const");
6      n5=Node(n3,"variable");
7      n6=Node(n3,"ethereal");
8      w=Wrapper(n5,1);
9
10     unknwn.untrusted(w);
11
12     assert n2.property=="robust"
13     ...
14 }
```

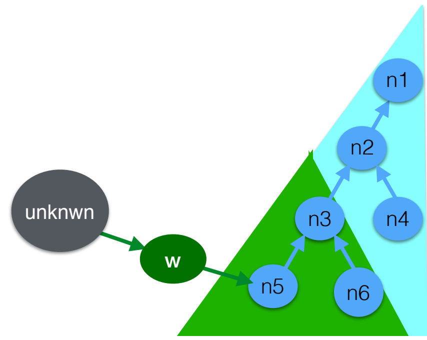


Fig. 9. `Wrapper`s protecting `Node`s

Even though we know nothing about the `unknown` object or its `untrusted` function, and even though the call gives to `unknown` access to `w`, which in turn has transitively access to all `Node`-s in the tree, we know that the `untrusted` function is guaranteed not to line 10 will not affect the `property` fields of the nodes `n1`, `n2`, and `n4`. Thus, the assertion on line 12 is guaranteed to succeed. The question is how do we specify `Wrapper`, so as to be able to make such an argument.

A specification of the class `Wrapper` in the traditional style, *e.g.* [11] (*c.f.* appendix **??**) consists of pairs of pre- and post- conditions for each of the functions of that class. Each such pair gives a *sufficient* condition for some effect to take place: for example the call `w.setProperty(i,prp)` where `i` is smaller than `w.height` is a sufficient condition to modify `property` of the `i`-th parent of `w.node`. But we do not know what other ways there may be to modify a node's `property`. A broken wrapper could accidently permit access to nodes one or two levels about 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, return 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`. What is needed here is some way to specify the *necessary conditions* under which some change could be made: if some external client is to change a node's property, then that client must have either direct access to a node in that DOM tree, or indirect access via a wrapper configured so that it can change the affected node. Thus,

> 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`.

***Sophia:** Shall we say the following, or does it break the flow? "Moreover, on line 10 we do not know which functions are called on w."*
KJX: I put that in and expanded on it. We need to explain the issues I think I do not see where it is

With such a specification we can prove that the assertion on line 12 will succeed. Crucially, we can ensure that all future updates of the `Wrapper` class must continue to meet that specification, guaranteeing the protection of the `Node` data. To give a flavour of *Chainmail*, we use it express the requirement from above:

$$\forall \texttt{S}:\texttt{Set}.\forall \texttt{nd}:\texttt{Node}.$$
$$[\ \mathcal{W}ith\langle\ \texttt{S},\ \mathcal{W}ill\langle \mathit{Change}\langle \texttt{nd.property}\rangle\rangle\rangle$$
$$\longrightarrow$$
$$\exists \texttt{o}:\texttt{Object}[\ \texttt{o}\in\texttt{S}\ \ \wedge\ \ \neg(\texttt{o}:\texttt{Node})\ \ \wedge\ \ \neg(\texttt{o}:\texttt{Wrapper})\ \ \wedge$$
$$[\ \exists \texttt{nd}':\texttt{Node}.\mathcal{A}ccess\langle \texttt{o},\texttt{nd}'\rangle\ \ \vee$$

$$\exists \texttt{w} : \texttt{Wrapper}.\exists k{:}\mathbb{N}.(\ \mathcal{A}ccess\langle \texttt{o}, \texttt{w}\rangle \ \land\ \texttt{nd.parnt}^{k} = \texttt{w.node.parnt}^{\texttt{w.height}})\ ]]$$
]

That is, if the value of `nd.property` is modified ($Change\langle\_\rangle$) at some future point ($\mathcal{W}ill\langle\_\rangle$) and if reaching that future point involves no more objects than those from set `S` (*i.e.* $\mathcal{W}ith\langle$ `S`, $\_\rangle$), then at least one (`o`) of the objects in `S` is not a `Node` nor a `Wrapper`, and `o` has direct access to some node ($\mathcal{A}ccess\langle \texttt{o}, \texttt{nd}'\rangle$), or to some wrapper `w` and the `w.height`-th parent of `w` is an ancestor of `nd` (that is, $\texttt{parnt}^{k} = \texttt{w.node.parnt}^{\texttt{w.height}}$). Definitions of these concepts appear later (Definition **??**), but note that our "access" is intransitive: $\mathcal{A}ccess\langle x, y\rangle$ holds if either `x` has a field pointing to `y`, or `x` is the receiver and `y` is one of the arguments in the executing method call.

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

7 DISCUSSION

Necessary conditions vs the complement of the sufficient condtiions? One might ask whether the necessary conditions are different from the complement of all the sufficient conditions. In other words, the possible behaviours of a module is the union of all possible behaviours of each individual function, and the necessary conditions is their complement, We described this in the left hand side of the diagram in Figure 1: we represent the space of all theoretically possible behaviours as points in the rectangle, each function is a coloured oval and its possible behaviours are the points in the area of that oval. Then, the necessary conditions are all the points outside the ovals.

This view is mathematically sound but it is impractical, brittle wrt software maintenance, and weak wrt reasoning in the open world.

It is impractical, because it suggests that when interested in a necessity guarantee one would need to read the specifications of all the functions in a module. In view of the number of these functions, and also the number of behaviours emerging from their combination, this can be a very large undertaking. 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?

It is brittle wrt software maintenance, because it gives no guidance to the team maintaining a piece of software: if the necessary conditions which were implicitly in the developers' intentions are not explicitly described, subsequent developers may inadvertenty add functions which break these intentions.

It is weak wrt reasoning in the open world because if does not give any guarantees about objects' when these 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 to an unknown advertiser a wrapper pointing to lower parts of the tree.

Design choices. For our underlying language, we have chosen a class based language; we used classes, because we concentrate on class-based, object-oriented programming. But we believe that the ideas are also applicable to other kinds of languages.

APPENDIX – EXAMPLES

■[5]

[5] SD: Note that the file rest.tex contains more material.

A THE LANGUAGE $\mathcal{L}_{oo}$

A.1 Modules and Classes

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

DEFINITION 12 (MODULES). *We define Module as the set of mappings from identifiers to class descriptions (the latter defined in Definition 13):*

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

Classes, as defined below, consist of field and method definitions. Note that $\mathcal{L}_{oo}$ is untyped. 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.

Note also that field read or write is only allowed if the target object is `this` – as, *e.g.* , in Smalltalk – this is encapsulation: the syntax allows an object to read/write its own fields, but forbids it from reading/writing any other object's fields.

DEFINITION 13 (CLASSES). *We define the syntax of class descriptions below.*

$$\begin{array}{lcl}
\mathit{ClassDescr} & ::= & \textsf{class}\ \mathit{ClassId}\ \{\ \ (\ \mathtt{field}\ \mathtt{f}\ )^*\ \ (\ \textsf{method}\ \mathit{MethBody}\ )^*\ \ \} \\
\mathit{MethBody} & ::= & \mathtt{m}(\ \mathtt{x}^*)\ \{\ \ \mathit{Stmts}\ \ \} \\
\mathit{Stmts} & ::= & \mathit{Stmt}\ \ \mid\ \ \mathit{Stmt}\ ;\ \mathit{Stmts} \\
\mathit{Stmt} & ::= & \mathtt{this.f} := \mathtt{x}\ \mid\ \mathtt{x} := \mathtt{this.f}\ \mid\ \mathtt{x} := \mathtt{x.m}(\ \mathtt{x}^*) \\
 & & \ \mid\ \mathtt{x} := \textsf{new}\ \mathtt{C}(\ \mathtt{x}^*)\ \mid\ \textsf{return}\ \mathtt{x} \\
\mathtt{x},\ \mathtt{f},\ \mathtt{m} & ::= & \mathtt{Identifier}
\end{array}$$

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

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

DEFINITION 14 (LOOKUP). *For a class identifier* `C` *and a method identifier* `m` *:*

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

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

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

DEFINITION 15 (RUNTIME ENTITIES). *We define addresses, values, frames, stacks, heaps and runtime configurations.*

- *We take addresses to be an enumerable set,* `Addr`*, and use the identifier* $\alpha \in \mathtt{Addr}$ *to indicate an address.*
- *Values, v, are either addresses, or sets of addresses or null:* $v \in \{\mathtt{null}\} \cup \mathtt{Addr} \cup \mathcal{P}(\mathtt{Addr})$.
- *Continuations are either statements (as defined in Definition 13) or a marker,* $\mathtt{x} := \bullet$*, for a nested call followed by statements to be executed once the call returns.*

Continuation ::= *Stmts* | `x:=• ;` *Stmts*

- *Frames, ϕ, consist of a code stub and a mapping from identifiers to values:* $\phi \in CodeStub \times \texttt{Ident} \rightarrow Value$,
- *Stacks, ψ, are sequences of frames,* $\psi ::= \phi \mid \phi \cdot \psi$.
- *Objects consist of a class identifier, and a partial mapping from field identifier to values:* $Object = \texttt{ClassID} \times (\texttt{FieldId} \rightarrow Value)$.
- *Heaps, χ, are mappings from addresses to objects:* $\chi \in \texttt{Addr} \rightarrow Object$.
- *Runtime configurations, σ, are pairs of stacks and heaps,* $\sigma ::= (\psi, \chi)$.

Note that values may be sets of addresses. Such values are never part of the execution of $\mathcal{L}_{oo}$, but are used to give semantics to assertions – we shall see that in Definition **??**.

Next, we define the interpretation of variables (`x`) and field look up (`this.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 16 (Interpretations). *We first define lookup of fields and classes, where α is an address, and `f` is a field identifier:*

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

We now define interpretations as follows:

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

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

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

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

Definition 17 (Lookup and update of runtime configurations). *We define convenient shorthands for looking up in runtime entities.*

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

Execution of a statement has the form $\texttt{M}, \sigma \leadsto \sigma'$, and is defined in figure 10.

Definition 18 (Execution). *of one or more steps is defined as follows:*

- *The relation* $\texttt{M}, \sigma \leadsto \sigma'$, *it is defined in Figure 10.*
- $\texttt{M}, \sigma \leadsto^* \sigma'$ *holds, if a)* $\sigma = \sigma'$, *or b) there exists a* σ'' *such that* $\texttt{M}, \sigma \leadsto^* \sigma''$ *and* $\texttt{M}, \sigma'' \leadsto \sigma'$.

James: Toby had added that following but I do not see what we need it for.

A.3 Definedness of execution, and extending configurations

Toby: We defer the definition of initial configurations to Definition ?? later.

Note that interpretations and executions need not always be defined. For example, in a configuration whose top frame does not contain `x` in its domain, $\lfloor \texttt{x} \rfloor_\phi$ is undefined. We define the relation $\sigma \sqsubseteq \sigma'$

methCall_OS

$$\frac{\begin{array}{l}\phi.\texttt{contn} \ = \ \texttt{x:=}\ \texttt{x}_0.\texttt{m}(\ \texttt{par}_1,...\texttt{par}_n)\ \texttt{;}\ \texttt{Stmts}\\ \lfloor \texttt{x}_0 \rfloor_\phi = \alpha\\ \mathcal{M}(\texttt{M}, Class(\alpha)_\chi, \texttt{m}) \ = \ \texttt{m}(\ par_1, \ldots par_n)\ \{\ \texttt{Stmts}_1\}\\ \phi'' \ = \ (\ \texttt{Stmts}_1,\ \ (\texttt{this} \mapsto \alpha, \texttt{par}_1 \mapsto \lfloor \texttt{x}_1 \rfloor_\phi, \ldots \texttt{par}_n \mapsto \lfloor \texttt{x}_n \rfloor_\phi\ )\ )\end{array}}{\texttt{M},\ (\ \phi \cdot \psi,\ \chi\ ) \ \rightsquigarrow \ (\ \phi'' \cdot \phi[\texttt{contn} \mapsto \texttt{x:=}\bullet\ \texttt{;}\ \texttt{Stmts}] \cdot \psi,\ \chi\ )}$$

varAssgn_OS

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

fieldAssgn_OS

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

objCreate_OS

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

return_OS

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

Fig. 10. Operational Semantics

to express that σ has more information than σ', and then prove that more defined configurations preserve interpretations:

Definition 19 (Extending runtime configurations). *The relation $\sqsubseteq$ is defined on runtime configurations as follows. Take arbitrary configurations σ, σ', σ'', frame ϕ, stacks ψ, ψ', heap χ, address α free in χ, value v and object o, and define $\sigma \sqsubseteq \sigma'$ as the smallest relation such that:*

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

Lemma A.1 (Preservation of interpretations and executions). *If $\sigma' \sqsubseteq \sigma$, then*

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

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,

Susan: where does the aux come from? I think what you said in the fragment calculus about disjointedness is neater

Sophia: aux is defined in last line of Def. below. In the Frag Calculus the modules were not mappings, so we did not need something like aux; any idea how to avoid?

DEFINITION 20 (MODULE LINKING). *The linking operator* $\circ$: *Module* $\times$ *Module* $\longrightarrow$ *Module is defined as follows:*

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

and where,

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

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

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

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

A.5 Module pairs and visible states semantics

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

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

DEFINITION 21. *Given runtime configurations* σ, σ', *and a module-pair* $\mathtt{M} \mathbin{⨾} \mathtt{M}'$ *we define execution where* $\mathtt{M}$ *is the internal, and* $\mathtt{M}'$ *is the external module as below:*

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

In the definition above n is allowed to have the value 2. In this case the final bullet is trivial and there exists a direct, external transition from σ to σ'. Our definition is related to the concept of visible states semantics, but differs in that visible states semantics select the configurations at which an invariant is expected to hold, while we select the states which are considered for executions which are expected to satisfy an invariant. Our assertions can talk about several states (through the use of the $\mathcal{W}ill\langle\_\rangle$ and $\mathcal{W}as\langle\_\rangle$ connectives), and thus, the intention of ignoring some intermediate configurations can only be achieved if we refine the concept of execution.[6]

The following lemma states that linking external modules preserves execution

Lemma A.3 (Linking modules preserves execution). *For any modules* $\texttt{M}$, $\texttt{M}'$, *and* $\texttt{M}''$, *whose domains are pairwise disjoint, and runtime configurations* σ, σ',

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

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

We can now answer the question as to which runtime configurations are pertinent when judging a module's adherence to an assertion. First, where does execution start? We define *initial* configurations to be those which may contain arbitrary code stubs, but which contain no objects. Objects will be created, and further methods will be called through execution of the code in $\phi.\texttt{contn}$. From such initial configurations, executions of code from $\texttt{M}\,⨾\,\texttt{M}'$ creates a set of *arising* configurations, which, as we will see in Definition 11, are pertinent when judging $\texttt{M}$'s adherence to assertions.

Definition 22 (Initial and arising Configurations). *are defined as follows:*

- $\mathcal{I}nitial\langle(\psi,\chi)\rangle$, *if* ψ *consists of a single frame* ϕ *with* $dom(\phi) = \{\texttt{this}\}$, *and* $\lfloor\texttt{this}\rfloor_\phi = \texttt{null}$, *and* $dom(\chi) = \emptyset$.
- $\mathcal{A}rising(\texttt{M}\,⨾\,\texttt{M}') \;=\; \{\; \sigma \;\mid\; \exists\sigma_0.\,[\; \mathcal{I}nitial\langle\sigma_0\rangle \;\wedge\; \texttt{M}\,⨾\,\texttt{M}', \sigma_0 \rightsquigarrow^* \sigma \;]\;\}$

[6]Explain better? Use the term "atomic"?

REFERENCES

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

[2] Gilad Bracha. *The Dart Programming Language*. December 2015.

[3] Gilad Bracha. The Newspeak language specification version 0.1. `newspeaklanguage.org/`, February 2017.

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

[5] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[6] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about object capabilities with logical relations and effect parametricity. In *IEEE EuroS&P*, pages 147–162, 2016.

[7] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[8] Ian J. Hayes, Xi Wu, and Larissa A. Meinicke. Capabilities for Java: Secure access to resources. In *APLAS*, pages 67–84, 2017.

[9] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[10] Timothy Jones, Michael Homer, **James Noble**, and Kim B. Bruce. Object inheritance without classes. In *ECOOP*, pages 13:1–13:26, 2016.

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

[12] K. R. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR16*. Springer, April 2010.

[13] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. A capability-based module system for authority control. In *ECOOP*, pages 20:1–20:27, 2017.

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

[15] B. Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall, second edition, 1997.

[16] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, Maryland, 2006.

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

[18] Toby Murray and Gavin Lowe. Analysing the information flow properties of object-capability patterns. In *FAST*, LNCS, 2010.

[19] Toby Murray, Robert Sison, and Kai Engelhardt. COVERN: A logic for compositional verification of information flow control. In *EuroS&P*, 2018.

[20] Dustin Rhodes, Tim Disney, and Cormac Flanagan. Dynamic detection of object capability violations through model checking. In *DLS*, pages 103–112, 2014.

[21] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 20011.

[22] The Ethereum Wiki. ERC20 Token Standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard.