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 [**?**]. 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[**?**], and that space on a web given to an advertiser will not be used to obtain access to our bank details[**?**].

***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[**?**] as well as the object capability paradigm[**?**], and its adoption in web systems [**? ? ?**] and programming languages such as Newspeak [**?**], Dart [**?**], Grace [**? ?**], Wyvern [**?**].

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.

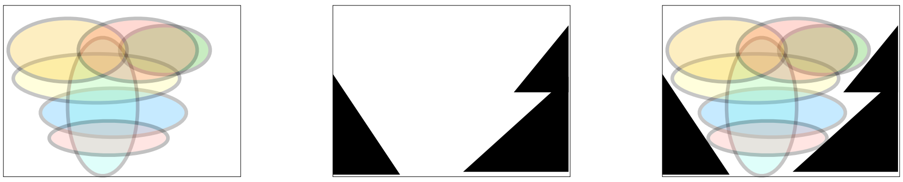
Necessary conditions are essentially about things that will *not* happen. For example, there will be no reduction to the account's funds without the owner's explicit request: the request being made by the owner is the necessary condition - under no other circumstances will the funds be reduced.

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 8 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 [**?**], 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[**? ?**]. 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[**? ?**], 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.

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.

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.*

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 **??** contain a formal definition of $\mathcal{L}_{oo}$, and Section 6 the semantics of assertions. Section ... related work .... Section xxxx concludes.

2 MOTIVATING EXAMPLE: THE BANK

Traditional functional specifications describe what objects 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 precondition and the method call together form a *sufficient* condition for the method's effect.

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 (here `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 2.

```
function deposit(src, amt)
PRE:  this,src:Account ∧ this≠src ∧ this.myBank=src.myBank ∧
      amt:ℕ ∧  src.balance≥amt
POST: src.balance=src.balance_pre-amt ∧ this.balance=this.balance_pre+amt

function makeNewAccount(amt)
PRE: this:Account ∧ amt:ℕ ∧ this.balance≥amt
POST: this.balance=this.balance_pre-amt ∧ fresh result ∧
      result: Account ∧ this.myBank=result.myBank ∧ result.balance=amt

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

Fig. 2. Functional specification of `Bank` and `Account`

The PRE-condition of `deposit` requires that the receiver and the first argument (`this`, `scr`) are `Accounts` and belong to the same bank, that `amt` is a number, and that `src` holds at leat `amt` moneys. The POST-condition mandates that `amt` has been transferred from `src` to the receiver. The function `makeNewAccount` specified below 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. susan: As specified in our BankSpec the only way to put money into the Bank is with a call to newAccount and there is no way to remove money from the Bank.

Sophia: We can say that, but is it important? We do not even use "newAccount" anywhere

Sophia: TO ADD

Aside Notice that the specification means that access to an Account allows one to withdraw all the money it holds – the concept of account owner who has exclusive right of withdrawal is not supported. This simplified view allows us to keep the example short, but compare with appendix for a specification which supports owners. *end aside*

Sophia: which is the best place to say that?

Susan: Can I switch assume to pre and assertion to post?

Sophia: No, because ORE and POST is how methods are specified, while assume and assert appear on code. We could use "requires" and "ensures" instead of PRE and POST

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

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

This reasoning is fine in a closed world, where we only have to consider complete programs, where all the code in our programs (or any other systems with which they interact) is under our control. In an open world, however, things are more complex: our systems will be made up of a range of modules, 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?

Sophia: yes, also in the bank, but does ti matter? I want to avoud having too many facets in the story.

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 susan: (also in the Bank)?If this method existed and if it were somehow called between registering at the conference and going to the bar, then the author (actually everyone using the same bank) would find an empty account (as would every other account holder other than the thief, of course).

The critical problem is that a bank implementation including a steal method would meet the functional specification of the bank from fig. 2, so long as the methods deposit, 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. 2 as *exact* in the sense that only implementations that meet the functional specification exactly, *with no extra methods or behaviour*, are considered as suitable implementations of the functional specification. The problem is that this solution is far too strong: it would for example rule out a bank that during software maintenance was given a new method count that simply counted the number of deposits that had taken place, or a method notify to enable the bank to occasionally send notifications to its customers.

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

Sophia: I changed the wording as I want to show "change -> call to deposit"

(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 module 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 access to the account are necessary for any change to an account. to take 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.

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

$$(1) \triangleq \forall \mathtt{a}.[\ \mathtt{a}:\mathtt{Account} \wedge \mathit{Change}\langle \mathtt{a.balance}\rangle \ \longrightarrow \\ \exists \mathtt{o}.[\ \mathtt{a}:\mathtt{Account}\mathit{Calls}\langle \mathtt{o},\mathtt{deposit},\mathtt{a},\_\rangle\_ \vee \mathit{Calls}\langle \mathtt{o},\mathtt{deposit},\_,\mathtt{a}\rangle\_\ ]\ \ ]$$

$$(2) \triangleq \forall \mathtt{a}.\forall \mathtt{S}:\mathtt{Set}.[\ \mathtt{a}:\mathtt{Account}\ \wedge\ \mathcal{W}\mathit{ith}\langle\ \mathtt{S}\,,\ (\mathcal{W}\mathit{ill}\langle\ \mathit{Change}\langle \mathtt{a.balance}\rangle\rangle)\ \rangle \quad \longrightarrow \\ \exists \mathtt{o}.\ [\ \mathtt{o}\in\mathtt{S}\ \wedge\ \mathcal{E}\mathit{xternal}\langle \mathtt{o}\rangle\ \wedge\ \mathcal{A}\mathit{ccess}\langle \mathtt{o},\mathtt{a}\rangle]\ \ ]$$

$$(3) \triangleq \forall \mathtt{a}.\forall \mathtt{S}:\mathtt{Set}.[\ \mathtt{a}:\mathtt{Account} \wedge \mathcal{W}\mathit{ith}\langle\ \mathtt{S},\ (\mathcal{W}\mathit{ill}\langle\ \exists \mathtt{o}.[\ \mathcal{E}\mathit{xternal}\langle \mathtt{o}\rangle\ \wedge\ \mathcal{A}\mathit{ccess}\langle \mathtt{o},\mathtt{a}\rangle]\ \rangle)\ \rangle \\ \longrightarrow\quad \exists \mathtt{o}'.\ [\ \mathtt{o}'\in\mathtt{S}\ \wedge\ \mathcal{E}\mathit{xternal}\langle \mathtt{o}'\rangle\ \wedge\ \mathcal{A}\mathit{ccess}\langle \mathtt{o}',\mathtt{a}\rangle\ ]\ \ ]$$

We will discuss the meaning of *Chainmail* assertions in more detail in the next sections, but give here a first impression, and discuss (1)-(3):

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

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

Sophia: *explain in some later section on that this assertion implicilty gives that* `o` *is external*

It remains to think about how access to a `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. 2 plus the necessary security policy specifications (1)-(3) from above. This holistic specification permits an implementation of the bank that also provides `count` and `notify` methods, but does not permit an implementation that also provides the `steal` method. 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 caller having a reference to most of those accounts, thus breaching policy (2).

Sophia: *We said this earlier, do we repeat?*

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`. Thus, if the `ACM` does not trust `authors`, it will not give them access to `acm_acc`, which contains all of the `ACM`'s money. Instead, in order to receive

Sophia: *I like the story of this paragraph. Please make more eloquent.*

Sophia: *Assertion (3) is a corollary of (2), and thus actually superfluous – do we say that? Do we way the story just using (2)?*

money, it will pass a secondary, empty account, acm_aux, 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 acm_aux to obtain access to acm_acc or any other account.

Thus, we need to be able to argue, that passing the acm_aux to some unknown object u, with an unknown function mystery, is guaranteed not to affect acm_acc , unless u already has access to acm_acc before the call. With holistic assertions we can make this argument formally, while we trandtional specifications we cannot.

In summary, our necessary specifications are invariants that describe the behaviour of a module as observed by its clients. These invariants are (usually) independent of the concrete implementation – in fact, in section XXXwe show two very different implementations of the Bank/Account specification which also satisfy (1)-(3). Our invariants 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.

Sophia: TO ADD

Sophia: This is james' original point, made a bit more concrete. Please check whether it makes sense.

3 *Chainmail* OVERVIEW

In this Section we give a brief and informal overview of *Chainmail*– a full exposition appears in Section 6. 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

Example 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 3. 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

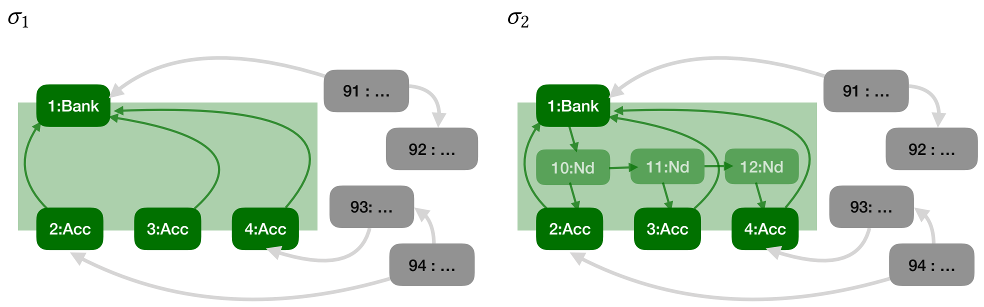


Fig. 3. Two runtime configurations for the Bank/Account example.

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.

***Sophia:** I think this para is great – not sure it belongs here.*

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`, expresses that $\texttt{a}_1$ and $\texttt{a}_2$ have the same bank. In fact, this assertion is satisfied in both σ_1 and σ_2, written formally as

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

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

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

We support ghost fields,*e.g.* $\texttt{a}_1$`.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`.

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

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

*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 policies are expressed as temporal or spatial invariants that a module that conforms to the specification must maintain even though any other code may be executed (as in the three policies in section **??**).*

***Sophia:** TODO citation*

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

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

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

If σ_1 was currently executing a call like $\texttt{a}_2$`.deposit(`$\texttt{a}_2$`,100.00)`, then we would have

$$..., \sigma_1 \models \mathcal{A}ccess\langle \texttt{a}_2, \texttt{a}_3\rangle,$$

Namely, during execution of that method, $\texttt{a}_2$ has access to $\texttt{a}_3$, and could, if the method body chose to, store a reference to $\texttt{a}_3$ in its own fields.Note that access is not symmetric, nor transitive:

$$..., \sigma_1 \not\models \mathcal{A}ccess\langle \texttt{a}_3, \texttt{a}_2\rangle,$$
$$..., \sigma_2 \models \mathcal{A}ccess^*\langle \texttt{a}_2, \texttt{a}_3\rangle,$$
$$..., \sigma_2 \not\models \mathcal{A}ccess\langle \texttt{a}_2, \texttt{a}_3\rangle.$$

***Sophia:** ALL: does this clarify why we define access to take method execution into account?*

Control: Calls The assertion $\mathit{Calls}\langle \texttt{x}, \texttt{y}, \texttt{m}, \texttt{zs}\rangle$ 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,

$$..., \sigma_0 \models \mathit{Calls}\langle \texttt{x}, \texttt{a}_2, \texttt{deposit}, (\texttt{a}_3, \texttt{100.00})\rangle.$$

means that the receiver in σ_0 is `x`, and the next statement to be executed is $\texttt{a}_2$`.deposit(`$\texttt{a}_3$`,100.00)`.

***Sophia:** it said " is more-or-less the control flow analogue of the access assertion" – that is a nice simile, but is it true?*

Authority – Changes and Internal/External The assertion $\mathit{Change}\langle \texttt{e}\rangle$ holds when the value of `e` in the next configuration is different to the value in the current configuration. For example, if the code being executed in σ_1 started with $\texttt{a}_2$`.balance = `$\texttt{a}_2$`.balance + 100.00`, then:

$$..., \sigma_1 \models \mathit{Change}\langle \texttt{a}_2\texttt{.balance}\rangle.$$

***Sophia:** Is it good to put these together?*

Moreover, the assertion $\mathcal{E}xternal\langle \texttt{e}\rangle$ expresses that the object `e` does not belong to the module under consideration. For example,

$$\texttt{M}_{AB2} \,⨾\, ..., \sigma_2 \models \mathcal{E}xternal\langle \texttt{u}_{92}\rangle$$
$$\texttt{M}_{AB2} \,⨾\, ..., \sigma_2 \not\models \mathcal{E}xternal\langle \texttt{a}_2\rangle$$
$$\texttt{M}_{AB2} \,⨾\, ..., \sigma_2 \not\models \mathcal{E}xternal\langle \texttt{b}_1\texttt{.ledger}\rangle$$

Notice the use of the *internal* module, $\texttt{M}_{AB2}$, needed to judge which objects are internal, and which are external.

Space: With The space assertion $\mathcal{W}ith\langle\ \mathtt{S},\ A\ \rangle$ states that assertion A is true in a configuration is restricted to the objects from the set $\mathtt{S}$. In other words, it restricts the set of objects which may be used to establish property A. For example, if $\mathtt{S}_1$ includes object $\mathtt{94}$, and $\mathtt{S}_2$ does not include it, then we have

$$\ldots, \sigma_1 \models \mathcal{W}ith\langle\ \mathtt{S}_1,\ \exists \mathtt{o}.[\ \mathcal{E}xternal\langle \mathtt{o}\rangle \land \mathcal{A}ccess\langle \mathtt{o}, \mathtt{a}_4\rangle\ ]\ \rangle$$
$$\ldots, \sigma_1 \not\models \mathcal{W}ith\langle\ \mathtt{S}_2,\ \exists \mathtt{o}.[\ \mathcal{E}xternal\langle \mathtt{o}\rangle \land \mathcal{A}ccess\langle \mathtt{o}, \mathtt{a}_4\rangle\ ]\ \rangle.$$

Sophia: TODO add citation to Amal's fuel

$\mathcal{W}ith\langle\ \mathtt{S},\ A\ \rangle$ is therefore *not* the footprint of the assertion A; it is more like the *fuel* given to establish that assertion. Note that $\mathcal{W}ith\langle\ \mathtt{S},\ A\ \rangle$ does not imply $\mathcal{W}ith\langle\ \mathtt{S} \cup \mathtt{S}',\ A\ \rangle$, nor the other direction.

Time: Next, Will, Prev, Was We supports several temporal operators familiar from temporal logic ($\mathcal{N}ext\langle A\rangle$, and $\mathcal{W}ill\langle A\rangle$, and $\mathcal{P}rev\langle A\rangle$, and $\mathcal{W}as\langle A\rangle$) to talk about the future or the past in one or any number of steps. The assertion $\mathcal{W}ill\langle A\rangle$ expresses that after one or more execution steps A will hold. For example, if the code being executed in σ_2 was method `m()` with receiver $\mathtt{u}_{94}$, and if $\sigma_2(\mathtt{94}, \mathtt{f}_1) = \mathtt{2}$ and $\sigma_2(\mathtt{94}, \mathtt{f}_2) = \mathtt{93}$, and calling `m2` on `93` returns `4`, and the body of `m` was $\mathtt{this.f_1.deposit(this.f_2.m2(),4.00)}$,then

$$\mathtt{M}_{BA2} \fatsemi \ldots, \sigma_2 \models \mathcal{W}ill\langle Change\langle \mathtt{a}_2.\mathtt{balance}\rangle\rangle.$$

Sophia: It would be nice to have a simpler example; what makes it difficult is the fact that $\mathtt{M}_{AB2}$-internal transitions are not registered.

Putting these together We now look at some composite assertions which use ingredients from several families from above. The assertion below says that if the call to be made next is $\mathtt{u}_{94}.\mathtt{m()}$, then the balance of $\mathtt{a}_2$ will eventually change:

$$\mathtt{M}_{BA2} \fatsemi \ldots, \sigma_2 \models Calls\langle .., \mathtt{u}_{94}, \mathtt{m}, ()\rangle \longrightarrow \mathcal{W}ill\langle Change\langle \mathtt{a}_2.\mathtt{balance}\rangle\rangle.$$

Sophia: The observer pattern example is very nice, but unfortunately we cannot use it, because the calls are internal, and therefore not visible in the current system.

We now add space to the mix, and demonstrate that in general $\mathcal{W}ill\langle \mathcal{W}ith\langle\ \mathtt{S},\ A\ \rangle\rangle$ is different from $\mathcal{W}ith\langle\ \mathtt{S},\ \mathcal{W}ill\langle A\rangle\ \rangle$. Consider $\mathtt{S}_1$ consisting of objects `1`, `2`, `4`, `93`, and `94`, and $\mathtt{S}_2$ consisting of objects `1`, `2`, `4`. Assume also that σ_1 contained the call `m()` with receiver $\mathtt{u}_{94}$, that the code of `m` and `m2` were as above. Then

$$\mathtt{M}_{BA1} \fatsemi \ldots, \sigma_1 \models \mathcal{W}ith\langle\ \mathtt{S}_1,\ \mathcal{W}ill\langle Change\langle \mathtt{a}_2.\mathtt{balance}\rangle\rangle\ \rangle$$
$$\mathtt{M}_{BA1} \fatsemi \ldots, \sigma_1 \not\models \mathcal{W}ith\langle\ \mathtt{S}_2,\ \mathcal{W}ill\langle Change\langle \mathtt{a}_2.\mathtt{balance}\rangle\rangle\ \rangle$$
$$\mathtt{M}_{BA1} \fatsemi \ldots, \sigma_1 \models \mathcal{W}ill\langle \mathcal{W}ith\langle\ \mathtt{S}_2,\ Change\langle \mathtt{a}_2.\mathtt{balance}\rangle\ \rangle\rangle$$

Sophia: some delicate issues with σ_2 as used in prev. paragraph, where we already assumed that the code in σ_2 started with $\mathtt{u}_{94}.\mathtt{m()}$ – what can we do?

In summary, our holistic 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.

.

.

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

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?

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

Sophia: TODO: add references here

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, $\mathtt{M} \fatsemi \mathtt{M}'$, 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.

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

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

James: somewhere, should we say something like: the goal is to allow wholistic specifations with code of M combined with that of M′, the assertion A is satisfied. More formally, we define:

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

In that sense, module M′ represents all possible clients of M; and as it is arbitrarily chosen, it reflects the open world nature of our specifications.

Sophia: Is is sentence superfluous now?.

The judgement $\texttt{M} ⨾ \texttt{M}', \sigma \models A$ means that assertion A is satisfied by $\texttt{M} ⨾ \texttt{M}'$ and σ. As in traditional specification languages [**?** **?**], 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.

Sophia: In contrast to what we said on Friday's conf call we do not need to put any restrictions on M′ – not even disjointness is required.

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 M, *i.e.* those where the executing object (*i.e.* the current receiver) comes from module M′. Thus, program execution is a judgment of the form

Sophia: TODO connect with example earlier

$$\texttt{M} ⨾ \texttt{M}', \sigma \leadsto \sigma'$$

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

Sophia: TODO: add references here.

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

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

The meaning of assertions therefore may depend on the variable map, eg 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 SUMMARY OF UNDERLYING PROGRAMMING LANGUAGE SEMANTCIS

As was have already seen, *Chainmail* assertions not only talk about the contents of the current state (stack frame and heap), but they also talk about future and past states, Therefore, the meaning to *Chainmail* assertions depends on the underlying programming language. In this section, we outline a minimal such language, which we call $\mathcal{L}_{oo}$. Full definitions appear in Appendix A.

Sophia: TODO-say which features need to be in such a language and say that $\mathcal{L}_{oo}$ is an example of such one.

Central to our work is the concept of *module*, which is a repository of code. Modules map class identifiers to class definitions, see App. 13, and class definitions consist of method declarations and field declarations, see App. 14. $\mathcal{L}_{oo}$ is untyped – this reflects the open world, where we link with external modules which come without any guarantees. [1] Statements these can be field read or field assignments, object creation, method calls, and return statements. All fields are private in the sense of C++: 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. 5.

We use runtime configurations σ to describe all pertinent information about an execution snapshot: the heap, and a stack of fames. Each frame consists of a continuation, `contn` and a map from

[1]In further work we want to work in a setting where the internal module is typed, and the external is untyped.

variables to values. We define execution through a judgment of the form $\texttt{M}, \sigma \rightsquigarrow \sigma'$ in the Appendix, Fig. 5.

We also define the interpretation of of simple expressions through $\lfloor e \rfloor_\sigma$, and class look-up through $Class(e)_\chi$, see App. 17.

In order to deal with the distinction of external and internal modules, we define a module linking operator, $\circ$ si that $\texttt{M}\circ\texttt{M}'$ is the union of the two modules, provided that their domains are disjoint, see App. 21.

As we said earlier, our notion of execution distinguishes between the internal and external module, and treats execution of methods from the internal module as atomic. Therefore we define *modular execution* as follows:

Sophia: *better term*

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

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

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

LEMMA 5.1 (PROPERTIES OF LINKING). *For any modules $\texttt{M}$, $\texttt{M}'$, $\texttt{M}''$, and $\texttt{M}'''$ and runtime configurations σ, and σ' we have:*

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

6 ASSERTIONS – FULL DEFINTIONS

We now define the syntax of expressions and assertions.

6.1 Syntax of Assertions

DEFINITION 2 (ASSERTIONS). *The syntax of expressions ($\texttt{e}$) and assertions (A) is:*

$$
\begin{array}{lll}
\texttt{e} & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid \texttt{x} \mid \texttt{e.f} \\
\\
A & ::= & \texttt{e} \mid \texttt{e} = \texttt{e} \mid \texttt{e} : \texttt{ClassId} \mid \texttt{e} \in \texttt{S} \mid \\
 & & A \rightarrow A \mid A \wedge A \mid A \vee A \mid \neg A \mid \forall \texttt{x}.A \mid \forall \texttt{S} : SET.A \mid \exists \texttt{x}.A \mid \exists \texttt{S} : SET.A \mid \\
 & & \mathcal{E}xternal\langle x \rangle \mid \mathcal{A}ccess\langle x, y \rangle \mid Change\langle e \rangle \mid Calls\langle \texttt{x}, \texttt{y}, \texttt{m}, \texttt{zS} \rangle \mid \\
 & & \mathcal{N}ext\langle A \rangle \mid \mathcal{W}ill\langle A \rangle \mid \mathcal{P}rev\langle A \rangle \mid \mathcal{W}as\langle A \rangle
\end{array}
$$

[2]

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

As we discussed in section TODO validity of assertions has the format $\texttt{M} ⨾ \texttt{M}', \sigma \models A$, where $\texttt{M}$ is the internal module, whose internal workings are opaque to the external, client module $\texttt{M}'$. We break the definition into four parts: In definition 4 we define validity of basic assertions which reflect over the contents of the frame or the heap. In definition 5 we define validity of basic assertions which reflect over the contents of the frame or the heap.

6.2 Satlsfaction of Assertions - standard

DEFINITION 3 (INTERPRETATIONS FOR SIMPLE EXPRESSIONS). *For any runtime configuration, σ, and any $k \in \mathbb{N}$, and any simple expression, $\texttt{e}$, we define its interpretation as follows:*

- $\lfloor \texttt{true} \rfloor_\sigma \triangleq \texttt{true}$, *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 6.1 (INTERPRETATION CORRESPONDS TO EXECUTION). *For any simple expression* $\texttt{e}$, *module* $\texttt{M}$, *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 $\texttt{e}$. □

DEFINITION 4 (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 5 (ASSERTIONS WITH LOGICAL CONNECTIVES AND QUANTIFIES). *We now consider For modules* $\texttt{M}$, $\texttt{M}'$, *assertions* A, A', *variables* $\texttt{x}$ *and* $\texttt{S}$, *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* $\texttt{z}$ *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* $\texttt{Q}$ *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* $\texttt{Q}$ *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* $\texttt{z}$ *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.*

6.3 Satisfaction of Assertions - Space

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

DEFINITION 6 (SATISFACTION OF ASSERTIONS ABOUT SPACE-1). *For any modules* $\texttt{M}$, $\texttt{M}'$, *assertions* A, A', *variables* $\texttt{x}$ *and* $\texttt{S}$, *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* $\texttt{f}$, *or*
  - $\lfloor \texttt{x} \rfloor_\sigma = \lfloor \texttt{this} \rfloor_\sigma$ *and* $\lfloor \texttt{y} \rfloor_\sigma = \lfloor \texttt{z} \rfloor_\sigma$, *and* $\texttt{z}$ *appears in* $\sigma.\texttt{contn}$.
- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \mathit{Calls}\langle \texttt{x}, \texttt{y}, \texttt{m}, \texttt{zS} \rangle$ *if* $\sigma.\texttt{contn} = \texttt{u.m(v);\_}$ *for some variables* $\texttt{u}$ *and* $\texttt{v}$, *and*
  $\lfloor \texttt{this} \rfloor_\sigma = \lfloor \texttt{x} \rfloor_\sigma$, *and* $\lfloor \texttt{y} \rfloor_\sigma = \lfloor \texttt{u} \rfloor_\sigma$, *and* $\lfloor \texttt{z} \rfloor_\sigma = \lfloor \texttt{v} \rfloor_\sigma$.
- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \mathcal{W}\mathit{ith}\langle\ \texttt{S},\ A\ \rangle$ *if* $\texttt{M} \fatsemi \texttt{M}', \sigma\downarrow_{\texttt{S}} \models A$.
- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \mathcal{E}\mathit{xternal}\langle \texttt{e} \rangle$ *if* $\mathit{Class}(\lfloor \texttt{e} \rfloor_\sigma)_\sigma \notin \mathit{dom}(\texttt{M})$

$\mathcal{A}\mathit{ccess}\langle \texttt{x}, \texttt{y} \rangle$ expresses that $\texttt{x}$ has a *direct* path to $\texttt{y}$. It says that in the current frame, either $\texttt{x}$and $\texttt{y}$ are aliases, or $\texttt{x}$points to an object which has a field whose value is the same as that of $\texttt{y}$, or $\texttt{x}$ is the currently executing object and $\texttt{y}$ is a local variable or formal parameter $\texttt{z}$ which appears in the code in the continuation ($\sigma.\texttt{contn}$). 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 $\mathcal{W}\mathit{ith}\langle\ S,\ A\ \rangle$ promises that A holds in subconfiguration, whose heap is restricted to the objects from $\texttt{S}$.

Definition 7 (Restriction of Runtime Configurations). *The restriction operator* $\downarrow$ *applied to a runtime configuration* σ *and a set R is defined as follows:*

- $\sigma\downarrow_{\texttt{S}} \triangleq (\psi, \chi')$, *if* $\sigma = (\psi, \chi)$, *and* $\mathit{dom}(\chi') = \lfloor \texttt{S} \rfloor_\sigma$, *and* $\forall \alpha \in \mathit{dom}(\chi').\chi(\alpha) = \chi'(\alpha)$ ■[3]

Definition 8 (Satisfaction of Assertions about Space-2). *For any modules* $\texttt{M}$, $\texttt{M}'$, *assertion A, set variable* $\texttt{S}$, *and configuration* σ, *we define*

- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \mathcal{W}\mathit{ith}\langle\ \texttt{S},\ A\ \rangle$ *if* $\texttt{M} \fatsemi \texttt{M}', \sigma\downarrow_{\texttt{S}} \models A$.

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

6.4 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 $\mathcal{W}\mathit{ill}\langle \texttt{x.f} = 3 \rangle$ is satisfied in the *current* configuration if in some *future* configuration the field $\texttt{f}$ of the object that is pointed at by $\texttt{x}$ in the *current* configuration has the value 3. Note that in the future configuration, $\texttt{x}$ 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 $\texttt{contn}$ 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 $\texttt{x}$ in configuration σ' even if $\texttt{x}$ 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 $\mathcal{W}\mathit{ill}\langle \texttt{x.f} = 3 \wedge \mathcal{W}\mathit{ill}\langle \texttt{x.f} = 5 \rangle\rangle$

Definition 9 (Adaptation of Runtime Configurations). *For runtime configurations* σ, σ'.:

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

- $\sigma \triangleleft \sigma' \triangleq (\phi'' \cdot \psi', \chi')$ *if* $\sigma = (\phi \cdot \_, \_)$, *and* $\sigma' = (\phi' \cdot \psi', \chi')$, *and* $\phi = (\texttt{contn}, \beta)$, *and* $\phi' = (\texttt{contn}', \beta')$, *and* $\phi'' = (\,\texttt{contn}'[\texttt{zs}/\texttt{zs}'],\ \beta[\texttt{zs}' \mapsto \beta'(\texttt{zs})]\,)$, *where* $\texttt{zs} = dom(\beta)$, *and* $\texttt{zs}'$ *is a set of variables with the same cardinality as* $\texttt{zs}$, *and all variables in* $\texttt{zs}'$ *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 $[\texttt{zs}' \mapsto \beta'(\texttt{zs})]$. With this auxiliary definition, we can now define satisfaction of assertions with involve time:

DEFINITION 10 (ASSERTIONS OVER TIME). *For any modules* $\texttt{M}$, $\texttt{M}'$, *assertions* A, A', *variables* $\texttt{x}$ *and* $\texttt{S}$, *we define*

- $\texttt{M} ⨾ \texttt{M}', \sigma \models \mathit{Change}\langle \texttt{e} \rangle$ *if* $\exists \sigma'.[\ \texttt{M} ⨾ \texttt{M}', \sigma \rightsquigarrow \sigma' \ \wedge \lfloor e \rfloor_\sigma \neq \lfloor e \rfloor_{\sigma \triangleleft \sigma'}\ ]$.
- $\texttt{M} ⨾ \texttt{M}', \sigma \models \mathcal{N}ext\langle A \rangle$ *if* $\exists \sigma'.[\ \texttt{M} ⨾ \texttt{M}', \phi \rightsquigarrow \sigma' \ \wedge \texttt{M} ⨾ \texttt{M}', \sigma \triangleleft \sigma' \models A\ ]$, *and where* ϕ *is so that* $\sigma = (\phi \cdot \_, \_)$.
- $\texttt{M} ⨾ \texttt{M}', \sigma \models \mathcal{W}ill\langle A \rangle$ *if* $\exists \sigma'.[\ \texttt{M} ⨾ \texttt{M}', \phi \rightsquigarrow^* \sigma' \ \wedge \texttt{M} ⨾ \texttt{M}', \sigma \triangleleft \sigma' \models A\ ]$, *and where* ϕ *is so that* $\sigma = (\phi \cdot \_, \_)$.
- $\texttt{M} ⨾ \texttt{M}', \sigma \models \mathcal{P}rev\langle A \rangle$ *if* $\forall \sigma_1, \sigma_2.[\ \mathcal{I}nitial\langle \sigma_1 \rangle \ \wedge\ \texttt{M} ⨾ \texttt{M}', \sigma \rightsquigarrow^* \sigma_2 \ \wedge\ \texttt{M} ⨾ \texttt{M}', \sigma_2 \rightsquigarrow \sigma \longrightarrow \texttt{M} ⨾ \texttt{M}', \sigma \triangleleft \sigma_2 \models A\ ]$[4]
- $\texttt{M} ⨾ \texttt{M}', \sigma \models \mathcal{W}as\langle A \rangle$ *if* $\forall \sigma_1, ...\sigma_n.[\ \mathcal{I}nitial\langle \sigma_1 \rangle \ \wedge\ \sigma_n = \sigma \ \wedge\ \forall i \in [1..n).\texttt{M} ⨾ \texttt{M}', \sigma_i \rightsquigarrow \sigma_{i+1} \longrightarrow \exists j \in [1..n-1).\texttt{M} ⨾ \texttt{M}', \sigma \triangleleft \sigma_j \models A\ ]$[5]

Thus, $\texttt{M} ⨾ \texttt{M}', \sigma \models \mathcal{W}ill\langle A \rangle$ 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.

6.5 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 11 (EQUIVALENCE AND ENTAILMENTS OF ASSERTIONS).

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

LEMMA 6.2 (ASSERTIONS ARE CLASSICAL-1). *For all runtime configurations* σ, *assertions* A *and* A', *and modules* $\texttt{M}$ *and* $\texttt{M}'$, *we have*

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

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

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

.

Lemma 6.3 (Assertions are classical-2). *For assertions A, A', and A'' the following equivalences hold*

(1) $A \wedge \neg A \;\equiv\; \texttt{false}$
(2) $A \vee \neg A \;\equiv\; \texttt{true}$
(3) $A \wedge A' \;\equiv\; A' \wedge A$
(4) $A \vee A' \;\equiv\; A' \vee A$
(5) $(A \vee A') \vee A'' \;\equiv\; A \vee (A' \vee A'')$
(6) $(A \vee A') \wedge A'' \;\equiv\; (A \wedge A') \vee (A \wedge A'')$
(7) $(A \wedge A') \vee A'' \;\equiv\; (A \vee A') \wedge (A \vee A'')$
(8) $\neg(A \wedge A') \;\equiv\; \neg A \vee \neg A''$
(9) $\neg(A \vee A') \;\equiv\; \neg A \wedge \neg A''$
(10) $\neg(\exists \texttt{x}.A) \;\equiv\; \forall \texttt{x}.(\neg A)$
(11) $\neg(\exists k : \mathbb{N}.A) \;\equiv\; \forall k : \mathbb{N}.(\neg A)$
(12) $\neg(\exists \texttt{fs} : FLD^k.A) \;\equiv\; \forall \texttt{fs} : FLD^k.(\neg A)$
(13) $\neg(\forall \texttt{x}.A) \;\equiv\; \exists \texttt{x}.\neg(A)$
(14) $\neg(\forall k : \mathbb{N}.A) \;\equiv\; \exists k : \mathbb{N}.\neg(A)$
(15) $\neg(\forall \texttt{fs} : FLD^k.A) \;\equiv\; \exists \texttt{fs} : FLD^k.\neg(A)$

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

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

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 M ⨟ M′, the assertion A holds.

Definition 12. *For any module* M*, and assertion A, we define:*

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

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

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 Nodes 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 4 shows Wrapper objects attenuating the use of Nodes. 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.

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

   unknwn.untrusted(w);

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

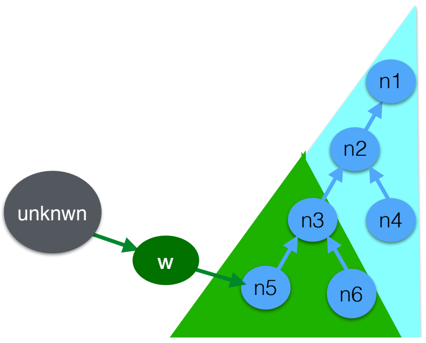


Fig. 4. Wrappers protecting Nodes

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.* [**?**] (*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,

__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

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

With such a specification we can prove that the assertion on line 12 will succeed. 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:

$$
\begin{array}{l}
\forall \texttt{S} : \texttt{Set}.\forall \texttt{nd} : \texttt{Node}.\\
[\ \mathcal{W}ith\langle\ \texttt{S},\ \mathcal{W}ill\langle Change\langle \texttt{nd.property}\rangle\rangle\ \rangle\\
\quad \longrightarrow\\
\quad \exists \texttt{o} : \texttt{Object}[\ \texttt{o} \in \texttt{S}\ \wedge\ \neg(\texttt{o} : \texttt{Node})\ \wedge\ \neg(\texttt{o} : \texttt{Wrapper})\ \wedge\\
\qquad [\ \exists \texttt{nd}' : \texttt{Node}.\mathcal{A}ccess\langle \texttt{o}, \texttt{nd}'\rangle\ \ \vee\\
\qquad\ \ \exists \texttt{w} : \texttt{Wrapper}.\exists k{:}\mathbb{N}.(\ \mathcal{A}ccess\langle \texttt{o}, \texttt{w}\rangle\ \wedge\ \texttt{nd.parnt}^{k} = \texttt{w.node.parnt}^{\texttt{w.height}})\ \ ]]\\
]
\end{array}
$$

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.* $\mathcal{W}ith\langle\ \texttt{S},\ \_\ \rangle$), 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.

8 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.

9 RELATED WORK

10 CONCLUSION

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

To focus on the key attributes of a holistic specification language, we have tried to keep the *Chainmail* as presented here as simple as possible. This has meant our language is intentionally restricted: we do not even support recursive procedures to avoid circularities in the metatheory, let alone concurrency, exceptions, distribution, networking, etc. We plan to remove these restrictions by applying techniques such as step-indexing [**?**] — even though this will necessarily complicate the formalism. 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 [**?**].

Finally, we hope to develop dynamic monitoring and automated reasoning techniques to make these kinds of specifications practically useful.

APPENDIX – EXAMPLES

■[6]

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 13 (Modules). *We define Module as the set of mappings from identifiers to class descriptions (the latter defined in Definition 14):*

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

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. 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. 5.

Definition 14 (Classes). *We define the syntax of class descriptions below.*

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

ClassDescr	::=	class *ClassId* { (`field f`)* (method *MethBody*)* }
MethBody	::=	`m`(`x`*) { *Stmts* }
Stmts	::=	*Stmt* | *Stmt* ; *Stmts*
Stmt	::=	`x.f := x` | `x := x.f` | `x := x.m( x`*`)`
		| `x :=` new `C ( x`*`)` | return `x`
`x, f, m`	::=	`Identifier`

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

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

Definition 15 (Lookup). *For a class identifier* `C` *and a method identifier* `m` *:*

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

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

- *We take addresses to be an enumerable set,* `Addr`*, and use the identifier* $\alpha \in$ `Addr` *to indicate an address.*
- *Values, v, are either addresses, or sets of addresses or null:* $v \in \{\mathtt{null}\} \cup \mathtt{Addr} \cup \mathcal{P}(\mathtt{Addr})$.
- *Continuations are either statements (as defined in Definition 14) or a marker,* `x :=` $\bullet$*, for a nested call followed by statements to be executed once the call returns.*
  Continuation ::= *Stmts* | `x :=` $\bullet$; *Stmts*
- *Frames, ϕ, consist of a code stub and a mapping from identifiers to values:* $\phi \in \mathit{CodeStub} \times \mathtt{Ident} \to \mathit{Value}$,
- *Stacks, ψ, are sequences of frames,* $\psi ::= \phi \mid \phi \cdot \psi$.
- *Objects consist of a class identifier, and a partial mapping from field identifier to values:* $\mathit{Object} = \mathtt{ClassID} \times (\mathtt{FieldId} \to \mathit{Value})$.
- *Heaps, χ, are mappings from addresses to objects:* $\chi \in \mathtt{Addr} \to \mathit{Object}$.
- *Runtime configurations, σ, are pairs of stacks and heaps,* $\sigma ::= (\psi, \chi)$.

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

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

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

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

We now define interpretations as follows:

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

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

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

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

Definition 18 (Lookup and update of runtime configurations). *We define convenient shorthands for looking up in runtime entities.*

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

Execution of a statement has the form $\texttt{M}, \sigma \leadsto \sigma'$, and is defined in figure 5.

Definition 19 (Execution). *of one or more steps is defined as follows:*

- *The relation* $\texttt{M}, \sigma \leadsto \sigma'$, *it is defined in Figure 5.*
- $\texttt{M}, \sigma \leadsto^* \sigma'$ *holds, if a)* $\sigma{=}\sigma'$, *or b) there exists a* σ'' *such that* $\texttt{M}, \sigma \leadsto^* \sigma''$ *and* $\texttt{M}, \sigma'' \leadsto \sigma'$.

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 x in its domain, $\lfloor \texttt{x} \rfloor_\phi$ is undefined. We define the relation $\sigma \sqsubseteq \sigma'$ to express that σ has more information than σ', and then prove that more defined configurations preserve interpretations:

Definition 20 (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* $\texttt{M}, \sigma \leadsto^* \sigma''$, *then there exists a* σ'', *so that* $\texttt{M}, \sigma' \leadsto^* \sigma'''$ *and* $\sigma''' \sqsubseteq \sigma''$.

methCall_OS

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

varAssgn_OS

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

fieldAssgn_OS

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

objCreate_OS

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

return_OS

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

Fig. 5. Operational Semantics

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

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

and where,

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

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

Lemma A.2 (Properties of linking). *– this is the same as A.2 in the main text – For any modules* M, M′ *and* M″, *and runtime configurations* σ, *and* σ' *we have*:

- $(\texttt{M}\circ\texttt{M}')\circ\texttt{M}'' = \texttt{M}\circ(\texttt{M}'\circ\texttt{M}'')$.

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

A.5 Module pairs and visible states semantics

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

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

Definition 22 (Repeating definition 1). *Given runtime configurations σ, σ', and a module-pair $\texttt{M} ⨟ \texttt{M}'$ we define execution where $\texttt{M}$ is the internal, and $\texttt{M}'$ is the external module as below:*

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

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 12, are pertinent when judging $\texttt{M}$'s adherence to assertions.

Definition 23 (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* $dom(\chi){=}\emptyset$.
- $\mathcal{A}rising(\texttt{M}\,⨾\,\texttt{M}') \;=\; \{\, \sigma \;\mid\; \exists\sigma_0.\,[\, \mathcal{I}nitial\langle\sigma_0\rangle \;\wedge\; \texttt{M}\,⨾\,\texttt{M}', \sigma_0 \leadsto^* \sigma \;\;]\;\}$