Holistic Specifications for Robust Programs

AUTHOR, Address

ACM Reference Format:
author. 2019. Holistic Specifications for Robust Programs. 1, 1 (March 2019), 9 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.

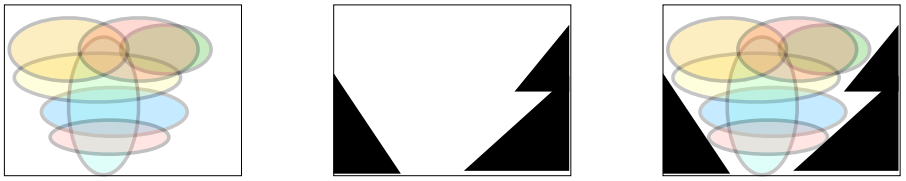
Necessary conditions are essentially about things that will *not* happen. For example, there will be no reduction to the account's funds without the owner's explicit request: the request being made by the owner is the necessary condition - under no other circumstances will the funds be reduced.

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 **??** 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.

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 **??** contain a formal definition of $\mathcal{L}_{oo}$, and Section **??** 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 pre-condition 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` on cane transfer any amount of money from the 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.

With such a specification the code below satisfies its assertions: assuming that `acm_acc` and `auth_ascc` 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 f 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_scc: Account ∧ acm.balance=10000 ∧ auhtor.balance=1500
acm.deposit(author, 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 code from earlier, they attempt to pay for a round at the bar. Under what circumstances can they be sure they have enough funds in their account?

To see the problem, what if the bank provided a `steal` method that emptied out every account in the bank and put them into a thief's account? If this method existed and if it was somehow called between registering at the conference and going to the bar, then the author (actually everyone using the same bank) woud find all their accounts empty (except the thief, of course).

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 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 maintenance was given a new method `deposit` that simply counted the number of deposits that had taken place, or a bank which occasionally sent 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`:

(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.
(3) Accounts and Banks are not leaked.

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 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) avoid that an `Account`'s balance is modified without access to the particular `Account`, and (3) 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:

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

$$(2) \triangleq \forall \mathtt{a} : \mathtt{Account}.\forall \mathtt{S} : \mathtt{Set}.\ [\mathit{With}\langle\ \mathtt{S}\ ,\ (\mathit{Will}\langle\ \mathit{Change}\langle \mathtt{a.balance} \rangle\rangle)\ \rangle \longrightarrow \exists \mathtt{o}.\ [\ \mathtt{o} \in \mathtt{S}\ \wedge\ \mathit{External}\langle \mathtt{o} \rangle\ \wedge\ \mathit{Access}\langle \mathtt{o}, \mathtt{a} \rangle]\ \ ]$$

$$(3) \triangleq \forall \mathtt{a} : \mathtt{Account}.\forall \mathtt{S} : \mathtt{Set}.\ [\ \mathit{With}\langle\ \mathtt{S},\ (\mathit{Will}\langle\ \exists \mathtt{o}.[\ \mathit{External}\langle \mathtt{o} \rangle\ \wedge\ \mathit{Access}\langle \mathtt{o}, \mathtt{a} \rangle]\ \rangle)\ \rangle \longrightarrow \exists \mathtt{o}'.\ [\ \mathtt{o}' \in \mathtt{S}\ \wedge\ \mathit{External}\langle \mathtt{o}' \rangle\ \wedge\ \mathit{Access}\langle \mathtt{o}', \mathtt{a} \rangle\ ]\ \ ]$$

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 in the past ($\mathit{Was}\langle ... \rangle$) called the `deposit` method with `a` as a receiver or an an argument ($\mathit{Calls}\langle \mathtt{o}, \mathtt{deposit}, \_, \_ \rangle$).

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

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

It remains to think about how access to a `Account` may be obtained. This is the remit of assertion (3): Assertion (3) says that if at sometime in the future observed in the configuration restricted to `S`, some object `o` which is external has access to some account `a`, then in the current configuration 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 exists and have access to `a` in the current configuration, but `o` need not exist in the current configuration – 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 the `author`, it will not pass to them `acm_acc`, which contains all of `ACM`'s money. Instead, it will pass a secondary, empty account, `acm_aux`, into which the `author` will pay her 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 if the `author` were malicious, she could not use `acm_aux` to obtain access to `acm_acc` or any other account.

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)?

In summary, our necessary specifications are invariants that describe the behaviour of a module as observed from its clients. These invariants are (usually) independent of the concrete implementation

Sophia: TO ADD – 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 state of the objects.

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

Sophia: would be nice iof we had a better name

3 *Chainmail* OVERVIEW

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

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

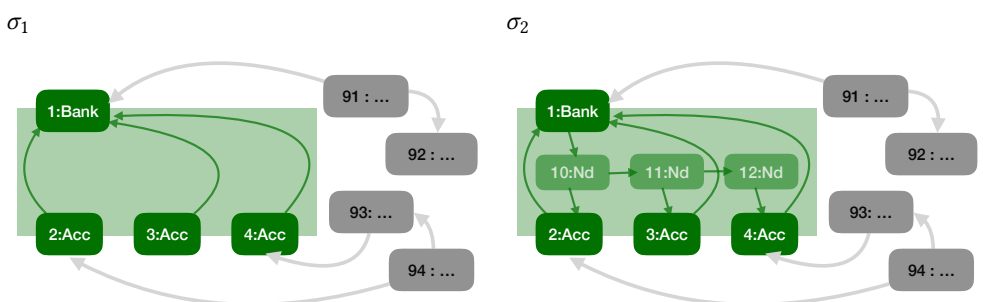


Fig. 3. Two runtime configurations for the `Bank/Account` example.

For the rest, assume variable identifiers b_1, and a_2–a_4, and u_{91}–u_{94} denoting objects `1`, `2`–`4`, and `91`–`94` respectively for both σ_1 and σ_2. That is, $\sigma_i(b_1)$=`1`, and $\sigma_i(a_2)$=`2`, $\sigma_i(a_3)$=`3`, $\sigma_i(a_4)$=`4`, and $\sigma_i(u_{91})$=`91`, $\sigma_i(u_{92})$=`92`, $\sigma_i(u_{93})$=`90`, $\sigma_i(u_{94})$=`94`, for i=1 or i=2.

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 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

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 a_2`.myBank=`a_3`myBank` , expresses that a_1 and a_2 have the same bank. In fact, this assertion is satisfied in both σ_1 and σ_2, written formally as

$$\ldots, \sigma_1 \models a_2\texttt{.myBank} = a_3\texttt{.myBank}$$
$$\ldots, \sigma_2 \models a_2\texttt{.myBank} = a_3\texttt{.myBank}$$

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

$$\ldots, \sigma_1 \models a_2\texttt{.myBank} = a_3\texttt{.myBank}.$$

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

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

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:

Sophia: TODO citation

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

If σ_1 was currently executing a call like $\texttt{a}_2\texttt{.deposit(a}_2\texttt{,100.00)}$, 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:

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

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

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

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

$$..., \sigma_0 \models \mathcal{C}alls\langle \texttt{x}, \texttt{a}_2, \texttt{deposit}, (\texttt{a}_3, \texttt{100.00})\rangle.$$

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

Authority – Changes and Internal/External The assertion $\mathcal{C}hange\langle \texttt{e}\rangle$ 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 $\texttt{a}_2\texttt{.balance} = \texttt{a}_2\texttt{.balance} + \texttt{100.00}$, then:

Sophia: Is it good to put these together?

$$..., \sigma_1 \models \mathcal{C}hange\langle \texttt{a}_2\texttt{.balance}\rangle.$$

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,

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

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

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

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

Sophia: TODO add citation to Amal's fuel

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 $\texttt{u}_{94}$, and

if $\sigma_2(94, \mathtt{f}_1) = 2$ and $\sigma_2(94, \mathtt{f}_2) = 93$, and calling `m2` on `93` returns `4`, and the body of `m` was `this.f1.deposit(this.f2.m2(),4.00)`,then

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

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:

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

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

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

Sophia: some delicate issues with σ_2 as used in prev. paragraph, where we already assumed that the code in σ_2 started with `u94.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 $\mathcal{C}hange\langle \_ \rangle$ 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

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

Sophia: TODO: add references here

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

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

Sophia: Is is sentence superfluous now?.

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

Sophia: TODO connect with

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?

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.

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:

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

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.

The judgement $\mathtt{M} \fatsemi \mathtt{M}', \sigma \models A$ means that assertion A is satisfied by $\mathtt{M} \fatsemi \mathtt{M}'$ and σ. As in traditional specification languages [**? ?**], satisfaction is judged in the context of 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.

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 $\mathtt{M}'$.

Thus, program execution is a judgment of the form

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

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 $\mathcal{A}rising(\mathtt{M} \fatsemi \mathtt{M}')$, *i.e.* the configurations arising from executions in $\mathtt{M} \fatsemi \mathtt{M}'$, we can take method bodies defined in M or in $\mathtt{M}'$, 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.* $\mathcal{W}ith\langle\ \mathtt{S},\ A\ \rangle$,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

$$\mathtt{M} \fatsemi \mathtt{M}', \sigma \models \mathcal{W}ith\langle\ \mathtt{S},\ A\ \rangle \quad \text{if} \quad \mathtt{M} \fatsemi \mathtt{M}', \sigma{\downarrow}_{\sigma(\mathtt{S})} \models A$$

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