Holistic Specifications for Robust Programs

Sophia Drossopoulou[1,3], James Noble[2],
Julian Mackay[2], and Susan Eisenbach[1]

[1] Imperial College London, United Kingdom
{scd, susan}@imperial.ac.uk
[2] Victoria University of Wellington, New Zealand
{julian.mackay, kjx}@ecs.vuw.ac.nz
[3] Microsoft Research Cambridge

Abstract Functional specifications describe what program components *can* do: the *sufficient* conditions to invoke components' operations. They allow us to reason about the use of components in a *closed world* setting, where components interact with known client code, and where the client code must establish the appropriate pre-conditions before calling into a component.

Sufficient conditions are not enough to reason about the use of components in an *open world* setting, where components interact with external code, possibly of unknown provenance, and where components may evolve over time. In this open world setting, we must also consider the *necessary* conditions, *i.e.* what are the conditions without which an effect will *not* happen.

In this paper we propose the *Chainmail* specification language for writing *holistic* specifications that focus on necessary conditions (as well as sufficient conditions). We give a formal semantics for *Chainmail*, and discuss several examples. The core of *Chainmail* has been mechanised in the Coq proof assistant.

1 Introduction

Software guards our secrets, our money, our intellectual property, our reputation [47]. We entrust personal and corporate information to software which works in an *open* world, where it interacts with third party software of unknown provenance, possibly buggy and potentially malicious.

This means we need our software to be *robust*: to behave correctly even if used by erroneous or malicious third parties. We expect that our bank will only make payments from our account if instructed by us, or by somebody we have authorised, that space on a web page given to an advertiser will not be used to obtain access to our bank details [43], or that a concert hall will not book the same seat more than once.

While language mechanisms such as constants, invariants, object capabilities [40], and ownership [14] make it *possible* to write robust programs, they cannot *ensure* that programs are robust. Ensuring robustness is difficult because it means different things for different systems: perhaps that critical operations should only be invoked with the requisite authority; perhaps that sensitive personal information should not be leaked; or perhaps that a resource belonging to one user should not be consumed by another. To ensure robustness, we need ways to specify what robustness means for a particular

© The Author(s) 2020
H. Wehrheim and J. Cabot (Eds.): FASE 2020, LNCS 12076, pp. 420–440, 2020.
https://doi.org/10.1007/978-3-030-45234-6_21

```
class Safe{
   field treasure
   field secret
   method take(scr){
      if (secret==scr) then
          {
         t=treasure
         treasure = null
         return t }   }
 }
```

```
class Safe{
   field treasure
   field secret
   method take(scr){
       ...as version 1...
   }
   method set(scr){
         secret=scr }
}
```

Figure 1. Two Versions of the class `Safe`

program, and ways to demonstrate that the particular program adheres to its specific robustness requirements.

Consider the code snippets from Fig. 1. Objects of class `Safe` hold a `treasure` and a `secret`, and only the holder of the secret can remove the treasure from the safe. We show the code in two versions; both have the same method `take`, and the second version has an additional method `set`. We assume a dynamically typed language (so that our results are applicable to both statically and dynamically typed settings)[4]; that fields are private in the sense of Java (*i.e.* only methods of that class may read or write these fields); and that addresses are unforgeable (so there is no way to guess a secret). A classical Hoare triple describing the behaviour of `take` would be:

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{ClassicSpec}) \triangleq} \\
\multicolumn{2}{l}{\texttt{method take(scr)}} \\
\texttt{PRE:} & \texttt{this:Safe} \\
\texttt{POST:} & \texttt{scr=this.secret}_{pre} \longrightarrow \texttt{this.treasure=null} \\
 & \quad\quad \wedge \\
 & \texttt{scr}\neq\texttt{this.secret}_{pre} \longrightarrow \forall \texttt{s:Safe. s.treasure=s.treasure}_{pre}
\end{array}
$$

(ClassicSpec) expresses that knowledge of the `secret` is *sufficient* to remove the treasure, and that `take` cannot remove the treasure unless the secret is provided. But it cannot preclude that `Safe` – or some other class, for that matter – contains more methods which might make it possible to remove the treasure without knowledge of the secret. This is the problem with the second version of `Safe`: it satisfies (ClassicSpec), but is not robust, as it is possible to overwrite the `secret` of the `Safe` and then use it to remove the treasure. To express robustness requirements, we introduce *holistic specifications*, and require that:

$$
\begin{array}{l}
(\text{HolisticSpec}) \triangleq \\
\quad \forall \texttt{s}.[\ \texttt{s} : \texttt{Safe} \wedge \texttt{s.treasure} \neq \texttt{null} \wedge \mathsf{will}\langle\, \texttt{s.treasure} = \texttt{null}\,\rangle \\
\qquad\qquad\qquad\qquad \longrightarrow\ \exists \texttt{o}.[\,\mathsf{external}\langle\,\texttt{o}\,\rangle \wedge \langle\,\texttt{o}\ \mathsf{access}\ \texttt{s.secret}\,\rangle\,]\ ]
\end{array}
$$

(HolisticSpec) mandates that for any safe `s` whose treasure is not `null`, if some time in the future its treasure were to become `null`, then at least one external object (*i.e.* an object whose class is not `Safe`) in the current configuration has direct access to `s`'s secret. This external object need not have caused the change in `s.treasure` but it would have (transitively) passed access to the secret which ultimately did cause that change. Both classes in Fig. 1 satisfy (ClassicSpec), but the second version does not satisfy (HolisticSpec).

---

[4] We do not depend on the additional safety static typing provides, so we assume only a dynamically typed language.

In this paper we propose *Chainmail*, a specification language to express holistic specifications. The design of *Chainmail* was guided by the study of a sequence of examples from the object-capability literature and the smart contracts world: the membrane [17], the DOM [20,59], the Mint/Purse [40], the Escrow [18], the DAO [12,15] and ERC20 [61]. As we worked through the examples, we found a small set of language constructs that let us write holistic specifications across a range of different contexts. *Chainmail* extends traditional program specification languages [31,37] with features which talk about:

Permission: Which objects may have access to which other objects; this is central since access to an object grants access to the functions it provides.

Control: Which objects called functions on other objects; this is useful in identifying the causes of certain effects - eg funds can only be reduced if the owner called a payment function.

Time: What holds some time in the past, the future, and what changes with time,

Space: Which parts of the heap are considered when establishing some property, or when performing program execution; a concept related to, but different from, memory footprints and separation logics,

Viewpoint: Which objects and which configurations are internal to our component, and which are external to it; a concept related to the open world setting.

While many individual features of *Chainmail* can be found in other work, their power and novelty for specifying open systems lies in their careful combination. The contributions of this paper are:

- the design of the holistic specification language *Chainmail*,
- the semantics of *Chainmail*,
- a Coq mechanisation of the core of *Chainmail*.

The rest of the paper is organised as follows: Section 2 gives an example from the literature which we will use to elucidate key points of *Chainmail*. 3 presents the *Chainmail* specification language. Section 4 introduces the formal model underlying *Chainmail*, and then section 5 defines the semantics of *Chainmail*'s assertions. Section 6 discusses our design, 7 considers related work, and section 8 concludes. We relegate key points of exemplar problems and various details to appendices which are available at [1].

2 Motivating Example: The Bank

As a motivating example, we consider a simplified banking application taken from the object capabilities literature [41]: `Accounts` belong to `Banks` and hold money (`balances`); with access to two `Accounts` of the same `Bank` one can transfer any amount of money from one to the other. This example has the advantage that it requires several objects and classes.

We will not show the code here (see appendix C), but suffice it to say that class `Account` has methods `deposit(src, amt)` and `makeAccount(amt)` (*i.e.* a method called `deposit` with two arguments, and a method called `makeAccount` with one argument). Similarly, `Bank` has method `newAccount(amt).` Moreover, `deposit` requires that the receiver and first argument (`this` and `src`) are `Accounts` and belong to the same bank, that the second argument (`amt`) is a number, and that `src`'s balance is at least `amt`. If this condition holds, then `amt` gets transferred from `src` to the receiver. The function `makeNewAccount` returns a fresh `Account` with the same bank, and transfers `amt` from the receiver `Account` to the new `Account`. Finally, the function `newAccount` when run by a `Bank` creates a new `Account` with corresponding amount of money in it.[5] It is not difficult to give formal specifications of these methods in terms of pre- and post-conditions.

However, what if the bank provided a `steal` method that emptied out every account in the bank into a thief's account? The critical problem is that a bank implementation including a `steal` method could meet the functional specifications of `deposit`, `makeAccount`, and `newAccount`, and still allow the clients' money to be stolen.

One obvious solution would be to adopt a closed-world interpretation of specifications: we interpret functional specifications 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 the design of the banking application. We provide additional specifications that capture those assumptions. The following three informal requirements prevent methods like `steal`:

1. An account's balance can be changed only if a client calls the `deposit` method with the account as the receiver or as an argument.
2. An account's balance can be changed only if a client has access to that particular account.
3. The `Bank`/`Account` component does not leak access to existing accounts or banks.

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

---

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

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

$$\begin{array}{ll} (1) \triangleq \forall \mathtt{a}.[\ \mathtt{a} : \mathtt{Account} \land \mathsf{changes}\langle\, \mathtt{a.balance}\,\rangle \ \longrightarrow & \\ \qquad\qquad \exists \mathtt{o}.[\ \langle\, \mathtt{o}\ \mathsf{calls}\ \mathtt{a.deposit(\_,\_)}\,\rangle \lor \langle\, \mathtt{o}\ \mathsf{calls}\ \mathtt{\_.deposit(a,\_)}\,\rangle\ ] &] \end{array}$$

$$\begin{array}{ll} (2) \triangleq \forall \mathtt{a}.\forall \mathtt{S} : \mathtt{Set}.\ [\ \mathtt{a} : \mathtt{Account}\ \land\ \langle \mathsf{will}\langle\ \mathsf{changes}\langle\, \mathtt{a.balance}\,\rangle\rangle\ \mathsf{in}\ \mathtt{S}\ \rangle & \\ \qquad\qquad\qquad \longrightarrow\ \ \exists \mathtt{o}.\ [\, \mathtt{o} \in \mathtt{S}\ \land\ \mathsf{external}\langle\, \mathtt{o}\,\rangle\ \land\ \langle\, \mathtt{o}\ \mathsf{access}\ \mathtt{a}\,\rangle\,] &] \end{array}$$

$$\begin{array}{ll} (3) \triangleq \forall \mathtt{a}.\forall \mathtt{S} : \mathtt{Set}.\ [\ \mathtt{a} : \mathtt{Account} \land & \\ \qquad\qquad \langle\, \mathsf{will}\langle\ \exists \mathtt{o}.[\ \mathsf{external}\langle\, \mathtt{o}\,\rangle\ \land\ \langle\, \mathtt{o}\ \mathsf{access}\ \mathtt{a}\,\rangle]\,\rangle\ \mathsf{in}\ \mathtt{S}\,\rangle & \\ \qquad\qquad \longrightarrow\ \ \exists \mathtt{o}'.\ [\, \mathtt{o}' \in \mathtt{S}\ \land\ \mathsf{external}\langle\, \mathtt{o}'\,\rangle\ \land\ \langle\, \mathtt{o}'\ \mathsf{access}\ \mathtt{a}\,\rangle\,] &] \end{array}$$

In the above and throughout the paper, we use an underscore (_) to indicate an existentially bound variable whose value is of no interest.

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

Assertion (2) similarly constrains any possible change to an account's balance. If at some future point the balance changes ($\mathsf{will}\langle\ \mathsf{changes}\langle\, ...\,\rangle\,\rangle$), and if this future change is observed with the state restricted to the objects from `S` (*i.e.* $\langle\, ...\ \mathsf{in}\ \mathtt{S}\,\rangle$), then at least one of these objects ($\mathtt{o} \in \mathtt{S}$) is external to the `Bank`/`Account` system ($\mathsf{external}\langle\, \mathtt{o}\,\rangle$) and has direct access to that account object ($\langle\, \mathtt{o}\ \mathsf{access}\ \mathtt{a}\,\rangle$). 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 a reference to `a` to the former (indirectly), which then makes the call to `deposit`.

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

A holistic specification for the bank account, then, would be a sufficient functional specification plus the necessary specifications (1)-(3) from above. This holistic specification permits an implementation of the bank that also provides `count` and `notify` methods, even though the specification does not mention either method. Critically, though, the holistic *Chainmail* specification does not permit an implementation that includes a `steal` method.

3 *Chainmail* Overview

In this Section we give a brief and informal overview of some of the most salient features of *Chainmail*– a full exposition appears in Section 5.

Example Configurations We will illustrate these features using the `Bank`/`Account` example from the previous section. We use the runtime configurations σ_1 and σ_2 shown in the left and right diagrams in Figure 2. In both diagrams the rounded boxes depict objects: green for those from the `Bank`/`Account` component, and grey for the "external", "client" objects. The transparent green rectangle shows which objects are contained by the `Bank`/`Account` component. The object at `1` is a `Bank`, those at `2`, `3` and `4` are `Account`s, and those at `91`, `92`, `93` and `94` are "client" objects which belong to classes different from those from the `Bank`/`Account` module.

Each configuration represents one alternative implementation of the Bank object. Configuration σ_1 may arise from execution using a module M_{BA1}, where `Account` objects have a field `myBank` pointing to their `Bank`, and an integer field `balance` – the code can be found in appendix C Fig. 3. Configuration σ_2 may arise from execution using a module M_{BA2}, where `Account`s have a `myBank` field, `Bank` objects have a `ledger` implemented though a sequence of `Node`s, each of which has a field pointing to an `Account`, a field `balance`, and a field `next` – the code can be found in appendix C Figs. 6 and 4.

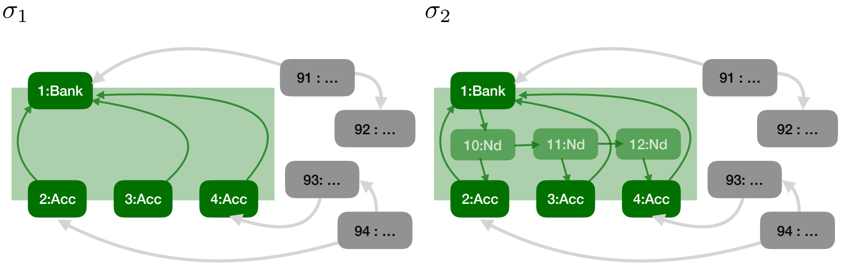


Figure 2. Two runtime configurations for the `Bank`/`Account` example.

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, for i=1 or i=2, $\sigma_i(\texttt{b}_1)$=`1`, $\sigma_i(\texttt{a}_2)$=`2`, $\sigma_i(\texttt{a}_3)$=`3`, $\sigma_i(\texttt{a}_4)$=`4`, $\sigma_i(\texttt{u}_{91})$=`91`, $\sigma_i(\texttt{u}_{92})$=`92`, $\sigma_i(\texttt{u}_{93})$=`93`, and $\sigma_i(\texttt{u}_{94})$=`94`.

Classical Assertions talk about the contents of the local variables (*i.e.* the topmost stack frame), and the fields of the various objects (*i.e.* the heap). For example, the assertion $\texttt{a}_2\texttt{.myBank}$=$\texttt{a}_3\texttt{.myBank}$, says that $\texttt{a}_2$ and $\texttt{a}_3$ have the same bank. In fact, this assertion is satisfied in both σ_1 and σ_2, written formally as

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

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

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

We support ghost fields [11,31], *e.g.* $\texttt{a}_1\texttt{.balance}$ is a physical field in σ_1 and a ghost field in σ_2 since in $\texttt{MBA2}$ an $\texttt{Account}$ does not store its $\texttt{balance}$ (as can be seen in appendix C Fig. 6). We also support the usual logical connectives, and so, we can express assertions such as

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

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

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

If σ_1 were executing the method body corresponding to the call $\texttt{a}_2\texttt{.deposit(a}_3\texttt{,360)}$, then we would have

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

That is, during execution of $\texttt{deposit}$, the object at $\texttt{a}_2$ has access to the object at $\texttt{a}_3$, and could, if the method body chose to, call a method on $\texttt{a}_3$, or store a reference to $\texttt{a}_3$ in its own fields. Access is not symmetric, nor transitive:

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

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

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

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

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

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

The set $\texttt{S}$ in the assertion $\langle\, A\ \mathsf{in}\ \texttt{S}\,\rangle$ is therefore *not* the footprint of A; it is more like the *fuel* [2] given to establish that assertion. Note that $...,\sigma \models \langle\, A\ \mathsf{in}\ \texttt{S}\,\rangle$ does not imply $...,\sigma \models A$ nor does it imply $...,\sigma \models \langle\, A\ \mathsf{in}\ \texttt{S} \cup \texttt{S}'\,\rangle$. The other direction of the implication does not hold either.

Time: Next, Will, Prev, Was We support several operators from temporal logic: ($\mathsf{next}\langle\, A\,\rangle$, $\mathsf{will}\langle\, A\,\rangle$, $\mathsf{prev}\langle\, A\,\rangle$, and $\mathsf{was}\langle\, A\,\rangle$) to talk about the future or the past in one or more steps. The assertion $\mathsf{will}\langle\, A\,\rangle$ expresses that A will hold in one or more steps. For example, taking σ_4 to be similar to σ_2, the next statement to be executed to be $\texttt{a}_2\texttt{.deposit(a}_3\texttt{, 360)}$, and $\texttt{M}_{BA2} ⨾ ...,\sigma_4 \models \texttt{a}_2\texttt{.balance} = \texttt{60}$, and that $\texttt{M}_{BA2} ⨾ ...,\sigma_4 \models \texttt{a}_4\texttt{.balance} \geq \texttt{360}$, then

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

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

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

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

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

Change and Authority: We have used $\mathsf{changes}\langle\, ...\,\rangle$ in our *Chainmail* assertions in section 2, as in $\mathsf{changes}\langle\, \mathtt{a.balance}\,\rangle$. Assertions that talk about change, or give conditions for change to happen are fundamental for security; the ability to cause change is called *authority* in [40]. We can encode change using the other features of *Chainmail*, namely, for any expression $\mathtt{e}$:

$$\mathsf{changes}\langle\, \mathtt{e}\,\rangle \;\equiv\; \exists v.[\, \mathtt{e} = v \land \mathsf{next}\langle\, \neg(\mathtt{e} = v)\,\rangle\,].$$

and similarly for assertions.

Putting these together We now look at some composite assertions which use several features from above. For example, the assertion below says that if the statement to be executed is $\mathtt{a}_2\mathtt{.deposit(a}_3\mathtt{, 60)}$, then the balance of $\mathtt{a}_2$ will eventually change:

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

Now look deeper into space assertions, $\langle\, A \,\mathsf{in}\, \mathtt{S}\,\rangle$, which allow us to characterise the set of objects which have authority over certain effects (here A). In particular, the assertion $\langle\, \mathsf{will}\langle\, A\,\rangle \,\mathsf{in}\, \mathtt{S}\,\rangle$ requires two things: i) that A will hold in the future, and ii) that all the objects which cause the effect which will make A valid are included in $\mathtt{S}$. Knowing who has, and who has not, authority over properties or data is a fundamental concern of robustness [40]. Notice that the authority is a set, rather than a single object: quite often it takes *several objects in concert* to achieve an effect.

Consider assertions (2) and (3) from the previous section. They both have the form "$\mathsf{will}\langle\, \langle\, A \,\mathsf{in}\, \mathtt{S}\,\rangle\,\rangle \;\longrightarrow\; P(\mathtt{S})$", where P is some property over a set. These assertions say that if ever in the future A becomes valid, and if the objects involved in making A valid are included in $\mathtt{S}$, then $\mathtt{S}$ must satisfy P. Such assertions can be used to restrict whether A will become valid. If we have some execution which only involves objects which do not satisfy P, then we know that the execution will not ever make A valid.

In summary, in addition to classical logical connectors and classical assertions over the contents of the heap and the stack, our holistic assertions draw from some concepts from object capabilities ($\langle\, \_ \,\mathsf{access}\, \_\,\rangle$ for permission; $\langle\, \_ \,\mathsf{calls}\, \_.\_(\,\_)\,\rangle$ and $\mathsf{changes}\langle\, \_\,\rangle$ for authority) as well as temporal logic ($\mathsf{will}\langle\, A\,\rangle$, $\mathsf{was}\langle\, A\,\rangle$ and friends), and the relation of our spatial connective ($\langle\, A \,\mathsf{in}\, S\,\rangle$) with ownership and effect systems [60,14,13].

The next two sections discuss the semantics of *Chainmail*. Section 4 contains an overview of the formal model and section 5 focuses on the most important part of *Chainmail*: assertions.

4 Overview of the Formal foundations

We now give an overview of the formal model for *Chainmail*. In section 4.1 we introduce the shape of the judgments used to give semantics to *Chainmail*, while in section

4.2 we describe the most salient aspects of an underlying programming language used in *Chainmail*.

4.1 *Chainmail* judgments

Having outlined the ingredients of our holistic specification language, the next question to ask is: When does a module `M` satisfy a holistic assertion A? More formally: when does $\texttt{M} \models A$ hold?

Our answer has to reflect the fact that we are dealing with an *open world*, where `M`, our module, may be linked with *arbitrary untrusted code*. To model the open world, we consider pairs of modules, $\texttt{M} \fatsemi \texttt{M}'$, where `M` is the module whose code is supposed to satisfy the assertion, and $\texttt{M}'$ is another module which exercises the functionality of `M`. We call our module `M` the *internal* module, and $\texttt{M}'$ the *external* module, which represents potential attackers or adversaries.

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

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

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

The judgement $\texttt{M} \fatsemi \texttt{M}', \sigma \models A$ means that assertion A is satisfied by $\texttt{M} \fatsemi \texttt{M}'$ and σ. As in traditional specification languages [31,37], satisfaction is judged in the context of a runtime configuration σ; but in addition, it is judged in the context of the internal and external modules. These are used to find abstract functions defining ghost fields as well as method bodies needed when judging validity of temporal assertions such as $\mathsf{will}\langle \_ \rangle$.

We distinguish between internal and external modules. This has two uses: First, *Chainmail* includes the "$\mathsf{external}\langle\, \mathsf{o} \,\rangle$" assertion to require that an object belongs to the external module, as in the Bank Account's assertion (2) and (3) in section 2. Second, we adopt a version of visible states semantics [45,25,38], treating all executions within a module as atomic. We only record runtime configurations which are *external* to module `M`, *i.e.* those where the executing object (*i.e.* the current receiver) comes from module $\texttt{M}'$. Execution has the form

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

where we ignore all intermediate steps with receivers internal to `M`. In the next section we shall outline the underlying programming language, and define the judgment $\texttt{M} \fatsemi \texttt{M}', \sigma \leadsto \sigma'$ and the set $\mathcal{A}\mathit{rising}(\texttt{M} \fatsemi \texttt{M}')$.

4.2 An underlying programming language, $\mathcal{L}_{\mathsf{oo}}$

The meaning of *Chainmail* assertions is parametric with an underlying object-oriented programming language, with modules as repositories of code, classes with fields, methods and ghostfields, objects described by classes, a way to link modules into larger ones, and a concept of program execution[6].

---

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

We have developed $\mathcal{L}_{oo}$, a minimal such object-oriented language, which we outline in this section. We describe the novel aspects of $\mathcal{L}_{oo}$, and summarise the more conventional parts, relegating full, and mostly unsurprising, definitions to Appendix A.

Modules are central to $\mathcal{L}_{oo}$, as they are to $\mathcal{C}hainmail$. As modules are repositories of code, we adopt the common formalisation of modules as maps from class identifiers to class definitions, c.f. Appendix, Def. 1. We use the terms module and component in an analogous manner to class and object respectively. $\mathcal{L}_{oo}$ is untyped for several reasons. Many popular programming languages are untyped. The external module might be untyped, and so it is more general to consider everything as untyped. Finally, a solution that works for an untyped language will also apply to a typed language, while the converse is not true.

Class definitions consist of field, method and ghost field declarations, c.f. Appendix, Def. 2. Method bodies are sequences of statements, which can be field reads or field assignments, object creation, method calls, and return statements. Fields are private in the sense of C++: they can only be read or written by methods of the current class. This is enforced by the operational semantics, *c.f.* Fig. 1. We discuss ghost fields in the next section.

Runtime configurations, σ, contain all the usual information about execution snapshots: the heap, and a stack of frames. Each frame consists of a continuation, `contn`, describing the remaining code to be executed by the frame, and a map from variables to values. Values are either addresses or sets of addresses; sets are needed to deal with assertions which quantify over sets of objects, such as assertions (1) and (2) from section 2. We define *one-module* execution through a judgment of the form $\mathtt{M}, \sigma \rightsquigarrow \sigma'$ in the Appendix, Fig. 1.

We define a module linking operator $\circ$ so that $\mathtt{M}\circ\mathtt{M}'$ is the union of the two modules, provided that their domains are disjoint, c.f. Appendix, Def. 8. As we said in section 4.1, we distinguish between the internal and external module. We consider execution from the view of the external module, and treat execution of methods from the internal module as atomic. For this, we define *two-module execution* based on one-module execution as follows:

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

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

In the definition above, $\mathcal{C}lass(\mathtt{x})_\sigma$ looks up the class of the object stored at `x`, c.f. Appendix, Def. 5. For example, for σ_4 as in Section 3 whose next statement to be executed is $\mathtt{a_2.deposit(a_3, 360)}$, we would have a sequence of configurations σ_{41}, ... σ_{4n}, σ_5 so that the one-module execution gives $\mathtt{M}_{BA2}, \sigma_4 \rightsquigarrow \sigma_{41} \rightsquigarrow \sigma_{42} ... \rightsquigarrow$

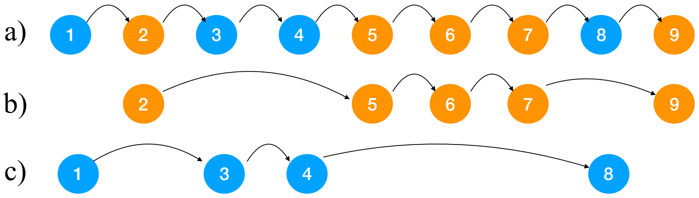


Figure 3. Two Module Execution (Def. 1). a) $\mathtt{M}_1 \circ \mathtt{M}_2$ b) $\mathtt{M}_1 \fatsemi \mathtt{M}_2$ c) $\mathtt{M}_2 \fatsemi \mathtt{M}_1$

$\sigma_{4n} \leadsto \sigma_5$. This would correspond to an atomic evaluation in the two-module execution: $\mathtt{M}_{BA2} \fatsemi \mathtt{M}', \sigma_4 \leadsto \sigma_5$ (see Fig.3; where blue stands for $\sigma(\mathit{this}) \in M_1$,and orange for $\sigma(\mathit{this}) \in M_2$).

Two-module execution is related to visible states semantics [45] as they both filter configurations, with the difference that in visible states semantics execution is unfiltered and configurations are only filtered when it comes to the consideration of class invariants while two-module execution filters execution. The lemma below says that linking is associative and commutative, and preserves both one-module and two-module execution.

Lemma 1 (Properties of linking). *For any modules* $\mathtt{M}$, $\mathtt{M}'$, $\mathtt{M}''$, *and* $\mathtt{M}'''$ *and runtime configurations* σ, *and* σ' *we have*:

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

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

Definition 2 (Initial and Arising Configurations). *are defined as follows:*

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

5 Assertions

Chainmail assertions (details in appendix B.3) consist of (pure) expressions $\mathtt{e}$, comparisons between expressions, classical assertions about the contents of heap and stack, the usual logical connectives, as well as our holistic concepts. In this section we focus on the novel, holistic, features of *Chainmail* (permission, control, time, space, and viewpoint), as well as our wish to support some form of recursion while keeping the logic of assertions classical.

5.1 Satisfaction of Assertions - Access, Control, Space, Viewpoint

Permission expresses that an object has the potential to call methods on another object, and to do so directly, without help from any intermediary object. This is the case when the two objects are aliases, or the first object has a field pointing to the second object, or the first object is the receiver of the currently executing method and the second object is one of the arguments or a local variable. Interpretations of variables and paths, $\lfloor ... \rfloor_\sigma$, are defined in the usual way (appendix Def. 5).

Definition 3 (Permission). *For any modules* $\texttt{M}$, $\texttt{M}'$, *variables* $\texttt{x}$ *and* $\texttt{y}$, *we define*

- $\texttt{M} \,⨾\, \texttt{M}', \sigma \models \langle\, \texttt{x}\, \mathsf{access}\, \texttt{y} \,\rangle$ *if* $\lfloor \texttt{x} \rfloor_\sigma$ *and* $\lfloor \texttt{y} \rfloor_\sigma$ *are defined, and*
  - $\lfloor \texttt{x} \rfloor_\sigma = \lfloor \texttt{y} \rfloor_\sigma$, *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* $\lfloor \texttt{y} \rfloor_\sigma = \lfloor \texttt{z} \rfloor_\sigma$, *for some variable* $\texttt{z}$ *and* $\texttt{z}$ *appears in* $\sigma.\texttt{contn}$.

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

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

Definition 4 (Control). *For any modules* $\texttt{M}$, $\texttt{M}'$, *variables* $\texttt{x}$, $\texttt{y}$, $\texttt{z}_1, ...\texttt{z}_n$, *we define*:

- $\texttt{M} \,⨾\, \texttt{M}', \sigma \models \langle\, \texttt{x}\, \mathsf{calls}\, \texttt{y}.\mathrm{m}(\, \texttt{z}_1, ...\texttt{z}_n) \,\rangle$ *if* $\lfloor \texttt{x} \rfloor_\sigma$, $\lfloor \texttt{y} \rfloor_\sigma$, $\lfloor \texttt{z}_1 \rfloor_\sigma$, ... $\lfloor \texttt{z}_n \rfloor_\sigma$ *are defined, and*
  - $\lfloor \texttt{this} \rfloor_\sigma = \lfloor \texttt{x} \rfloor_\sigma$, *and*
  - $\sigma.\texttt{contn} = \mathrm{u.m}(\mathrm{v}_1, ..\mathrm{v}_n); \_$, *for some* $\mathrm{u}, \mathrm{v}_1, ... \ \mathrm{v}_n$, *and*
  - $\lfloor \texttt{y} \rfloor_\sigma = \lfloor \texttt{u} \rfloor_\sigma$, *and* $\lfloor \texttt{z}_i \rfloor_\sigma = \lfloor \texttt{v}_i \rfloor_\sigma$, *for all* i.

Thus, $\langle\, \texttt{x}\, \mathsf{calls}\, \texttt{y}.\mathrm{m}(\, \texttt{z}_1, ...\texttt{z}_n) \,\rangle$ expresses the call $\texttt{y}.\mathrm{m}(\texttt{z}_1, ...\texttt{z}_n)$ will be executed next, and that the caller is $\texttt{x}$.

Viewpoint is about whether an object is viewed as belonging to the internal mode; this is determined by the class of the object.

Definition 5 (Viewpoint). *For any modules* $\texttt{M}$, $\texttt{M}'$, *and variable*$\texttt{x}$, *we define*

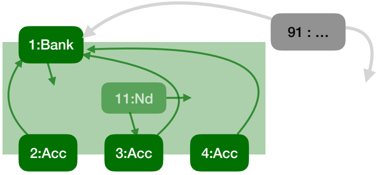
- $\texttt{M} \,⨾\, \texttt{M}', \sigma \models \mathsf{external}\langle\, \texttt{x} \,\rangle$ *if* $\lfloor \texttt{x} \rfloor_\sigma$ *is defined and* $Class(\lfloor \texttt{x} \rfloor_\sigma)_\sigma \notin dom(\texttt{M})$
- $\texttt{M} \,⨾\, \texttt{M}', \sigma \models \mathsf{internal}\langle\, \texttt{x} \,\rangle$ *if* $\lfloor \texttt{x} \rfloor_\sigma$ *is defined and* $Class(\lfloor \texttt{x} \rfloor_\sigma)_\sigma \in dom(\texttt{M})$

Space is about asserting that some property A holds in a configuration whose objects are restricted to those from a given set $\texttt{S}$. This way we can express that the objects from the set $\texttt{S}$ have authority over the assertion A. In order to define validity of $\langle\, A\, \mathsf{in}\, \texttt{S} \,\rangle$ in a configuration σ, we first define a restriction operation, $\sigma\!\downarrow_{\texttt{S}}$ which restricts the objects from σ to only those from $\texttt{S}$.

Definition 6 (Restriction of Runtime Configurations). *The restriction operator* $\downarrow$ *applied to a runtime configuration* σ *and a variable* $\texttt{S}$ *is defined as follows:*

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

For example, if we take σ_2 from Fig. 2 in Section 2, and restrict it with some set $\texttt{S}_4$ such that $\lfloor \texttt{S}_4 \rfloor_{\sigma_2} = \{91, 1, 2, 3, 4, 11\}$, then the restriction $\sigma_2\downarrow_{\texttt{S}_4}$ will look as on the right.



Note in the diagram above the dangling pointers at objects 1, 11, and 91 - reminiscent of the separation of heaps into disjoint subheaps, as provided by the $*$ operator in separation logic [53]. The difference is that in separation logic, the separation is provided through the assertions, where $A * A'$ holds in any heap which can be split into disjoint χ and χ' where χ satisfies A and χ' satisfies A'. That is, in $A * A'$ the split of the heap is determined by the assertions A and A' and there is an implicit requirement of disjointness, while in $\sigma\downarrow_{\texttt{S}}$ the split is determined by $\texttt{S}$, and no disjointness is required.

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

Definition 7 (Space). *For any modules* $\texttt{M}$, $\texttt{M}'$, *assertions* A *and variable* $\texttt{S}$, *we define*:

- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \langle\, A \,\textsf{in}\, \texttt{S} \,\rangle$ *if* $\texttt{M} \fatsemi \texttt{M}', \sigma\downarrow_{\texttt{S}} \models A$.

The set $\texttt{S}$ in the assertion $\langle\, A \,\textsf{in}\, \texttt{S} \,\rangle$ is related to framing from implicit dynamic frames [57]: in an implicit dynamic frames assertion **acc** $\texttt{x.f} * A$, the frame $\texttt{x.f}$ prescribes which locations may be used to determine validity of A. The difference is that frames are sets of locations (pairs of address and field), while our $\texttt{S}$-es are sets of addresses. More importantly, implicit dynamic frames assertions whose frames are not large enough are badly formed, while in our work, such assertions are allowed and may hold or not, *e.g.* $\texttt{M}_{BA2} \fatsemi \texttt{M}', \sigma \models \neg \langle\, (\exists n.\texttt{a}_2.\texttt{balance} = n) \,\textsf{in}\, \texttt{S}_4 \,\rangle$.

5.2 Satisfaction of Assertions - Time

To deal with time, we are faced with four challenges: a) validity of assertions in the future or the past needs to be judged in the future configuration, but using the bindings from the current one, b) the current configuration needs to store the code being executed, so as to be able to calculate future configurations, c) when considering the future, we do not want to observe configurations which go beyond the frame currently at the top of the stack, d) there is no "undo" operator to deterministically enumerate all the previous configurations.

Consider challenge a) in some more detail: the assertion $\textsf{will}\langle\, \texttt{x.f} = 3 \,\rangle$ is satisfied in the *current* configuration σ_1, if in some *future* configuration σ_2, the field $\texttt{f}$ of the object that is pointed at by $\texttt{x}$ in the *current* configuration (σ_1) has the value 3, that is, if $\lfloor \lfloor \texttt{x} \rfloor_{\sigma_1}.f \rfloor_{\sigma_2} = 3$, even if in that future configuration $\texttt{x}$ denotes a different object (i.e. if $\lfloor \texttt{x} \rfloor_{\sigma_1} \neq \lfloor \texttt{x} \rfloor_{\sigma_2}$). To address this, we define an auxiliary concept: the operator $\triangleleft$, where

$\sigma_1 \lhd \sigma_2$ adapts the second configuration to the top frame's view of the former: it returns a new configuration whose stack comes from σ_2 but is augmented with the view from the top frame from σ_1 and where the continuation has been consistently renamed. This allows us to interpret expressions in σ_2 but with the variables bound according to σ_1; *e.g.* we can obtain that value of `x` in configuration σ_2 even if `x` was out of scope in σ_2.

Definition 8 (Adaptation). *For runtime configurations σ_1, σ_2.:*

- $\sigma_1 \lhd \sigma_2 \triangleq (\phi_3 \cdot \psi_2, \chi_2)$ *if*
  - $\phi_3 = (\,\texttt{contn}_2[\texttt{zs}_2/\texttt{zs}'], \beta_2[\texttt{zs}' \mapsto \beta_2(\texttt{zs}_2)][\texttt{zs}_1 \mapsto \beta_1(\texttt{zs}_1)]\,)$, *where*
  - $\sigma_1 = (\phi_1 \cdot \_, \_)$, $\sigma_2 = (\phi_2 \cdot \psi_2, \chi_2)$, $\phi_1 = (\_, \beta_1)$, $\phi_2 = (\texttt{contn}_2, \beta_2)$, *and*
  - $\texttt{zs}_1 = \mathit{dom}(\beta_1)$, $\texttt{zs}_2 = \mathit{dom}(\beta_2)$, *and*
  - $\texttt{zs}'$ *is a set of variables with the same cardinality as* $\texttt{zs}_2$*, and all variables in* $\texttt{zs}'$ *are fresh in* β_1 *and in* β_2.

That is, in the new frame ϕ_2 from above, we keep the same continuation as from σ_2 but rename all variables with fresh names $\texttt{zs}'$, and combine the variable map β_1 from σ_1 with the variable map β_2 from σ_2 while avoiding names clashes through the renaming $[\texttt{zs}' \mapsto \beta_2(\texttt{zs}_2)]$. The consistent renaming of the continuation allows the correct modelling of execution, as needed for the semantics of nested time assertions, as *e.g.* in $\mathsf{will}\langle\, \texttt{x.f} = 3 \wedge \mathsf{will}\langle\, \texttt{x.f} = 5 \,\rangle\,\rangle$.

Having addressed challenge a) we turn our attention to the remaining challenges: We address challenge b) by storing the remaining code to be executed in `cntn` in each frame. We address challenge c) by only taking the top of the frame when considering future executions. Finally, we address challenge d) by considering only configurations which arise from initial configurations, and which lead to the current configuration.

Definition 9 (Time Assertions). *For any modules* `M`, $\texttt{M}'$*, and assertion A we define*

- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \mathsf{next}\langle\, A \,\rangle$ *if* $\exists \sigma'.[\ \texttt{M} \fatsemi \texttt{M}', \phi \leadsto \sigma' \ \wedge \texttt{M} \fatsemi \texttt{M}', \sigma \lhd \sigma' \models A\ ]$,
  and where ϕ is so that $\sigma = (\phi \cdot \_, \_)$.
- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \mathsf{will}\langle\, A \,\rangle$ *if* $\exists \sigma'.[\ \texttt{M} \fatsemi \texttt{M}', \phi \leadsto^* \sigma' \ \wedge \texttt{M} \fatsemi \texttt{M}', \sigma \lhd \sigma' \models A\ ]$,
  and where ϕ is so that $\sigma = (\phi \cdot \_, \_)$.
- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \mathsf{prev}\langle\, A \,\rangle$ *if* $\forall \sigma_1, \sigma_2.[\ \mathit{Initial}\langle \sigma_1 \rangle \ \wedge\ \texttt{M} \fatsemi \texttt{M}', \sigma_1 \leadsto^* \sigma_2$
  $\wedge\ \texttt{M} \fatsemi \texttt{M}', \sigma_2 \leadsto \sigma \ \longrightarrow\ \texttt{M} \fatsemi \texttt{M}', \sigma \lhd \sigma_2 \models A\ ]$
- $\texttt{M} \fatsemi \texttt{M}', \sigma \models \mathsf{was}\langle\, A \,\rangle$ *if* $\forall \sigma_1.[\ \mathit{Initial}\langle \sigma_1 \rangle \ \wedge\ \texttt{M} \fatsemi \texttt{M}', \sigma_1 \leadsto^* \sigma \longrightarrow$
  $(\ \exists \sigma_2.\texttt{M} \fatsemi \texttt{M}', \sigma_1 \leadsto^* \sigma_2 \ \wedge\ \texttt{M} \fatsemi \texttt{M}', \sigma_2 \leadsto^* \sigma \ \wedge\ \texttt{M} \fatsemi \texttt{M}', \sigma \lhd \sigma_2 \models A\ )]$

In general, $\mathsf{will}\langle\, \langle\, A \ \mathsf{in}\ \texttt{S} \,\rangle\,\rangle$ is different from $\langle\, \mathsf{will}\langle\, A \,\rangle \ \mathsf{in}\ \texttt{S} \,\rangle$. In the former assertion, `S` must contain the objects involved in reaching the future configuration as well as the objects needed to then establish validity of A in that future configuration. In the latter assertion, `S` need only contain the objects needed to establish A in that future configuration. For example, revisit Fig. 2, and take $\texttt{S}_1$ to consist of objects `1`, `2`, `4`, `93`, and `94`, and $\texttt{S}_2$ to consist of objects `1`, `2`, `4`. Assume that σ_5 is like σ_1, that the next call in σ_5 is a method on $\texttt{u}_{94}$, whose body obtains the address of $\texttt{a}_4$ (by making a call on `93` to which it has access), and the address of $\texttt{a}_2$ (to which it has access), and then makes the call $\texttt{a}_2\texttt{.deposit}(\texttt{a}_4, 360)$. Assume also that $\texttt{a}_4$'s balance is `380`. Then

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

5.3 Properties of Assertions

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

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

We can then prove that the usual equivalences hold, *e.g.* $A \vee A' \equiv A' \vee A$, and $\neg(\exists \mathtt{x}.A) \equiv \forall \mathtt{x}.(\neg A)$. Our assertions are classical, *e.g.* $A \wedge \neg A \equiv \mathtt{false}$, and $\mathtt{M} \,{}^{\circ}_{\circ}\, \mathtt{M}', \sigma \models A$ and $\mathtt{M} \,{}^{\circ}_{\circ}\, \mathtt{M}', \sigma \models A \rightarrow A'$ implies $\mathtt{M} \,{}^{\circ}_{\circ}\, \mathtt{M}', \sigma \models A'$. This desirable property comes at the loss of some expected equivalences, *e.g.*, in general, $\mathtt{e} = \mathtt{false}$ and $\neg\mathtt{e}$ are not equivalent. More in Appendix B.

5.4 Modules satisfying assertions

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

Definition 10. *For any module* $\mathtt{M}$*, and assertion* A*, we define:*

- $\mathtt{M} \models A \quad \text{\textit{if}} \quad \forall \mathtt{M}'. \forall \sigma \in \mathcal{A}rising(\mathtt{M} \,{}^{\circ}_{\circ}\, \mathtt{M}'). \; \mathtt{M} \,{}^{\circ}_{\circ}\, \mathtt{M}', \sigma \models A$

6 Examplar Driven Design

Examplars The design of *Chainmail* was guided by the study of a sequence of exemplars taken from the object-capability literature and the smart contracts world:

1. **Bank** [49] - Bank and Account as in Section 2 with two different implementations.
2. **ERC20** [61] - Ethereum-based token contract.
3. **DAO** [12,15] - Ethereum contract for Decentralised Autonomous Organisation.
4. **DOM** [20,59] - Restricting access to browser Domain Object Model

We present these exemplars as appendices [1]. Our design was also driven by work on other examples such as the membrane [17], the Mint/Purse [40], and Escrow [18,24].

Model We have constructed a Coq model[7] [23] of the core of the Chainmail specification language, along with the underlying $\mathcal{L}_{oo}$ language. Our formalism is organised as follows:

1. The $\mathcal{L}_{oo}$ Language: a class based, object oriented language with mutable references.
2. Chainmail: The full assertion syntax and semantics defined in Definitions 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10.
3. $\mathcal{L}_{oo}$ Properties: Secondary properties of the loo language that aid in reasoning about its semantics.
4. Chainmail Properties: The core properties defined on the semantics of Chainmail.

---

[7] A current model can be found at: https://github.com/sophiaIC/HolisticSpecifications

In the associated appendix (see Appendix G) we list and present the properties of Chainmail we have formalised in Coq. We have proven that Chainmail obeys much of the properties of classical logic. While we formalise most of the underlying semantics, we make several assumptions in our Coq formalism: (i) the law of the excluded middle, a property that is well known to be unprovable in constructive logics, and (ii) the equality of variable maps and heaps down to renaming. Coq formalisms often require fairly verbose definitions and proofs of properties involving variable substitution and renaming, and assuming equality down to renaming saves much effort.

More details of the formal foundations of *Chainmail*, and the model, are also in appendices [1].

7 Related Work

Behavioural Specification Languages Hatcliff et al. [26] provide an excellent survey of contemporary specification approaches. With a lineage back to Hoare logic [28], Meyer's Design by Contract [38] was the first popular attempt to bring verification techniques to object-oriented programs as a "whole cloth" language design in Eiffel. Several more recent specification languages are now making their way into practical and educational use, including JML [31], Spec♯ [4], Dafny [32] and Whiley [51]. Our approach builds upon these fundamentals, particularly Leino & Shulte's formulation of two-state invariants [33], and Summers and Drossopoulou's Considerate Reasoning [58]. In general, these approaches assume a closed system, where modules can be trusted to coöperate. In this paper we aim to work in an open system where modules' invariants must be protected irrespective of the behaviour of the rest of the system.

Defensive Consistency In an open world, we cannot rely on the kindness of strangers: rather we have to ensure our code is correct regardless of whether it interacts with friends or foes. Attackers *"only have to be lucky once"* while secure systems *"have to be lucky always"* [5]. Miller [39,40] defines the necessary approach as **defensive consistency**: *"An object is defensively consistent when it can defend its own invariants and provide correct service to its well behaved clients, despite arbitrary or malicious misbehaviour by its other clients."* Defensively consistent modules are particularly hard to design, to write, to understand, and to verify: but they make it much easier to make guarantees about systems composed of multiple components [46].

Object Capabilities and Sandboxes. Capabilities as a means to support the development of concurrent and distributed system were developed in the 60's by Dennis and Van Horn [19], and were adapted to the programming languages setting in the 70's [44]. *Object capabilities* were first introduced [40] in the early 2000s, and many recent studies manage to verify safety or correctness of object capability programs. Google's Caja [42] applies sandboxes, proxies, and wrappers to limit components' access to *ambient* authority. Sandboxing has been validated formally: Maffeis et al. [35] develop a model of JavaScript, demonstrate that it obeys two principles of object capability systems and show how untrusted applications can be prevented from interfering with the rest of the system. Recent programming languages [27,10,54] including Newspeak [9], Dart [8], Grace [7,30] and Wyvern [36] have adopted the object capability model.

Verification of Object Capability Programs Murray made the first attempt to formalise defensive consistency and correctness [46]. Murray's model was rooted in counterfactual causation [34]: an object is defensively consistent when the addition of untrustworthy clients cannot cause well-behaved clients to be given incorrect service. Murray formalised defensive consistency very abstractly, over models of (concurrent) object-capability systems in the process algebra CSP [29], without a specification language for describing effects, such as what it means for an object to provide incorrect service. Both Miller and Murray's definitions are intensional, describing what it means for an object to be defensively consistent.

Drossopoulou and Noble [21,48] have analysed Miller's Mint and Purse example [40] and discussed the six capability policies as proposed in [40]. In [22], they sketched a specification language, used it to specify the six policies from [40], showed that several possible interpretations were possible, and uncovered the need for another four further policies. They also sketched how a trust-sensitive example (the escrow exchange) could be verified in an open world [24]. Their work does not support the concepts of control, time, or space, as in $\mathcal{Chainmail}$, but it offers a primitive expressing trust.

Devriese et al. [20] have deployed powerful theoretical techniques to address similar problems: They show how step-indexing, Kripke worlds, and representing objects as state machines with public and private transitions can be used to reason about object capabilities. Devriese have demonstrated solutions to a range of exemplar problems, including the DOM wrapper (replicated in our section F) and a mashup application. Their distinction between public and private transitions is similar to the distinction between internal and external objects.

More recently, Swasey et al. [59] designed OCPL, a logic for object capability patterns, that supports specifications and proofs for object-oriented systems in an open world. They draw on verification techniques for security and information flow: separating internal implementations ("high values" which must not be exposed to attacking code) from interface objects ("low values" which may be exposed). OCPL supports defensive consistency (they use the term "robust safety" from the security community [6]) via a proof system that ensures low values can never leak high values to external attackers. This means that low values *can* be exposed to external code, and the behaviour of the system is described by considering attacks only on low values. They use that logic to prove a number of object-capability patterns, including sealer/unsealer pairs, the caretaker, and a general membrane.

Schaefer et al. [55] have recently added support for information-flow security using refinement to ensure correctness (in this case confidentiality) by construction. By enforcing encapsulation, all these approaches share similarity with techniques such as ownership types [14,50], which also protect internal implementation objects from accesses that cross encapsulation boundaries. Banerjee and Naumann demonstrated that by ensuring confinement, ownership systems can enforce representation independence (a property close to "robust safety") some time ago [3].

$\mathcal{Chainmail}$ differs from Swasey, Schaefer's, and Devriese's work in a number of ways: They are primarily concerned with mechanisms that ensure encapsulation (aka confinement) while we abstract away from any mechanism via the $\mathsf{external}\langle\ \rangle$ predicate. They use powerful mathematical techniques which the users need to understand in order to write their specifications, while *Chainmail* users only need to understand first order logic and the holistic operators presented in this paper. Finally, none of these systems offer the kinds of holistic assertions addressing control flow, change, or temporal operations that are at the core of *Chainmail*'s approach.

Scilla [56] is a minimalistic typed functional language for writing smart contracts that compiles to the Ethereum bytecode. Scilla's semantic model is restricted, assuming actor based communication and restricting recursion, thus facilitating static analysis of Scilla contracts and ensuring termination. Scilla is able to demonstrate that a number of popular Ethereum contracts avoid type errors, out-of-gas resource failures, and preservation of virtual currency. Scilla's semantics are defined formally, but have not yet been represented in a mechanised model.

Finally, the recent VerX tool is able to verify a range of specifications for solidity contracts automatically [52]. Similar to *Chainmail*, VerX has a specification language based on temporal logic. VerX offers three temporal operators (always, once, prev) but only within a past modality, while *Chainmail* has two temporal operators, both existential, but with both past and future modalities. VerX specifications can also include predicates that model the current invocation on a contract (similar to *Chainmail*'s "calls"), can access variables, and compute sums (only) over collections. *Chainmail* is strictly more expressive as a specification language, including quantification over objects and sets (so can compute arbitrary reductions on collections) and of course specifications for permission ("access"), space ("in") and viewpoint ("external") which have no analogues in VerX. Unlike *Chainmail*, VerX includes a practical tool that has been used to verify a hundred properties across case studies of twelve Solidity contracts.

8 Conclusions

In this paper we have motivated the need for holistic specifications, presented the specification language *Chainmail* for writing such specifications, and outlined the formal foundations of the language. To focus on the key attributes of a holistic specification language, we have kept *Chainmail* simple, only requiring an understanding of first order logic. We believe that the holistic features (permission, control, time, space and viewpoint) are intuitive concepts when reasoning informally, and were pleased to have been able to provide their formal semantics in what we argue is a simple manner.

9 Acknowledgments

This work is based on a long-standing collaboration with Mark S. Miller and Toby Murray. We have received invaluable feedback from Alex Summers, Bart Jacobs, Chris Hawblitzel, Michael Jackson, Lucius G. Meredith, Mike Stay, Shuh Peng Loh, Emil Klasan, members of WG 2.3, and the FASE 2020 reviewers. The work has been supported by the Royal Society of New Zealand (Te Apārangi) Marsden Fund (Te Pūtea Rangahau a Marsden) grants VUW-1318 and VUW-1815, and research gifts from Agoric, the Ethereum Foundation, and Facebook.

References

1. Holistic specifications paper with appendices. https://arxiv.org/abs/2002.08334, accessed: 2020-02-21
2. Ahmed, A., Dreyer, D., Rossberg, A.: State-dependent representation independence. In: POPL (2009)
3. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. J. ACM **52**(6), 894–960 (Nov 2005)
4. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: CASSIS. pp. 49–69. LNCS, Springer (2005)
5. BBC: On This Day: 1984: Tory cabinet in Brighton bomb blast (2015), [Online; accessed 15-October-2015]
6. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. ACM Trans. Program. Lang. Syst. **33**(2), 8:1–8:45 (Feb 2011)
7. Black, A., Bruce, K., Homer, M., Noble, J.: Grace: the Absence of (Inessential) Difficulty. In: Onwards (2012)
8. Bracha, G.: The Dart Programming Language (Dec 2015)
9. Bracha, G.: The Newspeak language specification version 0.1 (Feb 2017), `newspeaklanguage.org/`
10. Burtsev, A., Johnson, D., Kunz, J., Eide, E., van der Merwe, J.E.: Capnet: security and least authority in a capability-enabled cloud. In: Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017. pp. 128–141 (2017)
11. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and esc/java2. In: Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. pp. 342–363 (2005), https://doi.org/10.1007/11804192_16
12. Christoph Jentsch: Decentralized autonomous organization to automate governance (Mar 2016), https://download.slock.it/public/DAO/WhitePaper.pdf
13. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effectr. In: OOPSLA. ACM (2002)
14. Clarke, D.G., Potter, J.M., James Noble: Ownership types for flexible alias protection. In: OOPSLA. ACM (1998)
15. Coindesk: Understanding the DAO attack (2016), www.coindesk.com/understanding-dao-hack-journalists/
16. Community, S.: Solidity, https://solidity.readthedocs.io/en/develop/
17. van Cutsem, T.: Membranes in Javascript (2012), available from `prog.vub.ac.be/~tvcutsem/invokedynamic/js-membranes`
18. Cutsem, T.V., S, M.: Trustworthy proxies: Virtualizing objects with invariants. In: ECOOP (2013)
19. Dennis, J.B., Horn, E.C.V.: Programming Semantics for Multiprogrammed Computations. Comm. ACM **9**(3) (1966)
20. Devriese, D., Birkedal, L., Piessens, F.: Reasoning about object capabilities with logical relations and effect parametricity. In: IEEE EuroS&P. pp. 147–162 (2016). https://doi.org/10.1109/EuroSP.2016.22
21. Drossopoulou, S., Noble, J.: The need for capability policies. In: (FTfJP) (2013)
22. Drossopoulou, S., Noble, J.: Towards capability policy specification and verification (May 2014), `ecs.victoria.ac.nz/Main/TechnicalReportSeries`
23. Drossopoulou, S., Noble, J., Mackay, J., Eisenbach, S.: Holisitic Specifications for Robust Programs - Coq Model (2020). https://doi.org/10.5281/zenodo.3677621

24. Drossopoulou, S., Noble, J., Miller, M.: Swapsies on the internet: First steps towards reasoning about risk and trust in an open world. In: (PLAS) (2015)
25. Guttag, J.V., Horning, J.J.: Larch: Languages and Tools for Formal Specification. Springer (1993)
26. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.J.: Behavioral interface specification languages. ACM Comput.Surv. **44**(3), 16 (2012)
27. Hayes, I.J., Wu, X., Meinicke, L.A.: Capabilities for Java: Secure access to resources. In: APLAS. pp. 67–84 (2017)
28. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. ACM **12**, 576–580 (1969)
29. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
30. Jones, T., Homer, M., James Noble, Bruce, K.B.: Object inheritance without classes. In: ECOOP. pp. 13:1–13:26 (2016)
31. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D.R., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual (February 2007), iowa State Univ. www.jmlspecs.org
32. Leino, K.R.: Dafny: An automatic program verifier for functional correctness. In: LPAR16. Springer (April 2010)
33. Leino, K.R.M., Schulte, W.: Using history invariants to verify observers. In: ESOP (2007)
34. Lewis, D.: Causation. Journal of Philosophy **70**(17) (1973)
35. Maffeis, S., Mitchell, J., Taly, A.: Object capabilities and isolation of untrusted web applications. In: Proc of IEEE Security and Privacy (2010)
36. Melicher, D., Shi, Y., Potanin, A., Aldrich, J.: A capability-based module system for authority control. In: ECOOP. pp. 20:1–20:27 (2017)
37. Meyer, B.: Eiffel: The Language. Prentice Hall (1992)
38. Meyer, B.: Object-Oriented Software Construction, Second Edition. Prentice Hall, second edn. (1997)
39. Miller, M.S., Cutsem, T.V., Tulloh, B.: Distributed electronic rights in JavaScript. In: ESOP (2013)
40. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. Ph.D. thesis, Baltimore, Maryland (2006)
41. Miller, M.S., Morningstar, C., Frantz, B.: Capability-based Financial Instruments: From Object to Capabilities. In: Financial Cryptography. Springer (2000)
42. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Safe active content in sanitized JavaScript (2008), code.google.com/p/google-caja/
43. Mitre Organisation: CWE-830: Inclusion of Web Functionality from an Untrusted Source (2019), https://cwe.mitre.org/data/definitions/830.html
44. Morris Jr., J.H.: Protection in programming languages. CACM **16**(1) (1973)
45. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Science of Computer Programming **62**, 253–286 (2006)
46. Murray, T.: Analysing the Security Properties of Object-Capability Patterns. Ph.D. thesis, University of Oxford (2010)
47. Murray, T., Sison, R., Engelhardt, K.: COVERN: A logic for compositional verification of information flow control. In: EuroS&P (2018)
48. Noble, J., Drossopoulou, S.: Rationally reconstructing the escrow example. In: FTfJP (2014)
49. Noble, J., Potanin, A., Murray, T., Miller, M.S.: Abstract and concrete data types vs object capabilities. In: Müller, P., Schaefer, I. (eds.) Principled Software Development (2018)
50. Noble, J., Potter, J., Vitek, J.: Flexible alias protection. In: ECOOP (Jul 1998)
51. Pearce, D., Groves, L.: Designing a verifying compiler: Lessons learned from developing Whiley. Sci. Comput. Prog. (2015)
52. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: VerX: Safety verification of smart contracts. In: IEEE Symp. on Security and Privacy (2020)

53. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)
54. Rhodes, D., Disney, T., Flanagan, C.: Dynamic detection of object capability violations through model checking. In: DLS. pp. 103–112 (2014)
55. Schaefer, I., Runge, T., Knüppel, A., Cleophas, L., Kourie, D.G., Watson, B.W.: Towards confidentiality-by-construction. In: Leveraging Applications of Formal Methods, Verification and Validation. Modeling - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I. pp. 502–515 (2018)
56. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Chan, K.: Safer smart contract programming with Scilla. In: OOPSLA (2019)
57. Smans, J., Jacobs, B., Piessens, F.: Implicit Dynamic Frames. ToPLAS (2012)
58. Summers, A.J., Drossopoulou, S.: Considerate Reasoning and the Composite Pattern. In: VMCAI (2010)
59. Swasey, D., Garg, D., Dreyer, D.: Robust and Compositional Verification of Object Capability Patterns. In: OOPSLA (2017)
60. Talpin, J.P., Jouvelot, P.: The Type and Effect Discipline. In: LICS. pp. 162–173 (1992)
61. The Ethereum Wiki: ERC20 Token Standard (Dec 2018), https://theethereum.wiki/w/index.php/ERC20_Token_Standard

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.