Necessity Specifications are Necessary for Robustness

ANONYMOUS AUTHOR(S)

Robust modules guarantee to do *only* what they are supposed to do – even in the presence of untrusted, malicious clients, and considering not just the direct behaviour of individual methods, but also the emergent behaviour from calls to more than one method. *Necessity* is a language for specifying robustness, based on novel necessity operators capturing temporal implication, and a proof logic that derives explicit robustness specifications from classical specifications. Soundness and an exemplar proof sont méchanisés en Coq.

1 NECESSARY CONDITIONS AND ROBUSTNESS

Software needs to be both *correct* (programs do what they are supposed to) and *robust* (programs only do what they're supposed to). Robustness means that programs don't do what they aren't supposed to do, even in the presence of untrusted or malicious clients [IEEE Standards [n.d.]]. Correctness is classically specified through Hoare [1969] triples: a precondition, a code snippet, and a postcondition. For example, part of the classical specification of a `transfer` method for a bank module is that the source account's balance decreases:

> $S_{\texttt{correct}} \triangleq \{\texttt{pwd=src.pwd} \wedge \texttt{src.bal=b}\}\,\texttt{src.transfer(dst,pwd)}\,\{\texttt{src.bal=b-100} \wedge \ldots\}$
> Calling `transfer` on an account with the correct password will transfer the money.

Assuming termination, the precondition is a *sufficient* condition for the code snippet to behave correctly: the precondition (*e.g.* providing the right password) guarantees that the code (*e.g.* call the `transfer` function) will always achieve the postcondition (the money is transferred).

$S_{\texttt{correct}}$ describes the *correct use* of the module, but is *not* concerned with its *robustness*. For example, can I pass an account to foreign untrusted code, in the expectation of receiving a payment, but without fear that a malicious client might use the account to steal my money [Miller et al. 2000]? A first approach to specify robustness could be:

> $S_{\texttt{robust\_1}} \triangleq$ An account's balance does not decrease unless `transfer` was called with the correct password.

Specification $S_{\texttt{robust\_1}}$ guarantees that it is not possible to take money out of the account through some method other than `transfer` or without providing the password. Calling `transfer` with the correct password is a *necessary condition* for reducing the account's balance.

$S_{\texttt{robust\_1}}$ is crucial, but not enough: it does not take account of the module's *emergent behaviour*, that is, does not cater for the potential interplay of several methods offered by the module. What if the module provided further methods which leaked the password, or allowed for it to be arbitrarily changed? While no single procedure call is capable of breaking the intent of $S_{\texttt{robust\_1}}$, a sequence of calls might. What we really need is

> $S_{\texttt{robust\_2}} \triangleq$ The balance of an account does not *ever* decrease in the future unless some external object *now* has access to the account's current password.

With $S_{\texttt{robust\_2}}$, I can confidently pass my account to some untrusted client who does not have knowledge of the password; they may or may not make the payment I was expecting, but I know they will not be able to steal my money [Miller 2011; Miller et al. 2013]. Note that $S_{\texttt{robust\_2}}$ does not mention the names of any functions in the module, and thus can be expressed without reference to any particular API — indeed $S_{\texttt{robust\_2}}$ can constrain *any* API with an account, an account balance, and a password.

2022. 2475-1421/2022/1-ART1 $15.00
https://doi.org/

1.1 Earlier Work

Earlier work addressing robustness includes object capabilities [Birkedal et al. 2021; Devriese et al. 2016; Miller 2006], information control flow [Murray et al. 2013; Zdancewic and Myers 2001], correspondence assertions [Fournet et al. 2007], sandboxing [Patrignani and Garg 2021; Sammler et al. 2019], robust linear temporal logic [Anevlavis et al. 2022] – to name a few. Most of these propose *generic* robustness guarantees (*e.g.* no dependencies from high values to low values), while we work with *problem-specific* guarantees (*e.g.* no decrease in balance without access to password). *Necessity* is the first approach that is able to both express and prove robustness specifications such as $S_{\texttt{robust\_2}}$.

VerX [Permenev et al. 2020] and *Chainmail* [Drossopoulou et al. 2020b] also work on problem-specific guarantees. Both these approaches can express necessary conditions like $S_{\texttt{robust\_1}}$ using temporal logic operators and implication. For example, $S_{\texttt{robust\_1}}$ could be written as:

```
a:Account ∧ a.balance==bal ∧ ⟨next a.balance<bal⟩
                          ⟶ ⟨_ calls a.transfer(_, a.password)⟩
```

That is, if in the next state the balance of `a` decreases, then the current state is a call to the method `transfer` with the right password passed in. Note that the underscore indicates an existentially quantified variable used only once, for example, `⟨_ calls a.transfer(_, a.password)⟩` is short for `∃o.∃a'.⟨o calls a.transfer(a', a.password)⟩`. Here `o` indicates the current receiver, i.e, the object whose method is currently being executed and making the call.

However, to express $S_{\texttt{robust\_2}}$, one also needs what we call *capability operators*, which talk about provenance ("external object") and permission ("`x` has access to `y`"). VerX does not support capability operators, and thus cannot express $S_{\texttt{robust\_2}}$, while *Chainmail* does support capability operators, and can express $S_{\texttt{robust\_2}}$. VerX comes with a symbolic execution system which can demonstrate adherence to its specifications, but doesn't have a proof logic, whereas, *Chainmail* has neither a symbolic execution system, nor a proof logic.

Temporal operators in VerX and *Chainmail* are first class, *i.e.* may appear in any assertions and form new assertions. This makes VerX and *Chainmail* very expressive, and allows specifications which talk about any number of points in time. However, this expressivity comes at the cost of making it very difficult to develop a logic to prove adherence to such specifications.

1.2 *Necessity*

Necessity is a language for specifying a module's robustness guarantees and a logic to prove adherence to such specifications.

For the specification language we adopted *Chainmail*'s capability operators. For the temporal operators, we observed that while their unrestricted combination with other logical connectives allows us to talk about any number of points in time, the examples found in the literature talk about two or at most three such points.

This led to the crucial insight that we could merge temporal operators and the implication logical connective into our three *necessity* operators. One such necessity operator is

$$\texttt{from}\ A_{curr}\ \texttt{to}\ A_{fut}\ \texttt{onlyIf}\ A_{nec}$$

This form says that a transition from a current state satisfying assertion A_{curr} to a future state satisfying A_{fut} is possible only if the necessary condition A_{nec} holds in the *current* state. Using this operator, we can formulate $S_{\texttt{robust\_2}}$ as

```
S_robust_2 ≜ from a:Account ∧ a.balance==bal to a.balance < bal
             onlyIf' ∃ o.[⟨o external⟩ ∧ ⟨o access a.pwd⟩]
```

Namely, a transition from a current state where an account's balance is `bal`, to a future state where it has decreased, may *only* occur if in the current state some unknown client object has access to that account's password. More discussion in §2.2.

We also support two further *Necessity* operators:

`from` A_{curr} `next` A_{fut} `onlyIf` A_{nec}. `from` A_{curr} `to` A_{fut} `onlyThrough` A_{intrm}

The first says that a *one-step* transition from a current state satisfying assertion A_{curr} to a future state satisfying A_{fut} is possible only if A_{nec} holds in the *current* state. The second says that a change from A_{curr} to A_{fut} may happen only if A_{intrm} holds in some *intermediate* state.

Unlike *Chainmail*'s temporal operators, the necessity operators are second class, and may not appear in the assertions (*e.g.* A_{curr}). This simplification enabled us to develop our proof logic. Thus, we have reached a sweet spot between expressiveness and provability.

We faced the challenge how to develop a logic that would enable us to prove that code adhered to specifications talking of system-wide properties. The Eureka moment was the realisation that all the information we required was hiding in the individual methods' classical specifications:

Our logic is based on the crucial insight that the specification `from` A_1 `to` A_2 `onlyIf` A_3 is logically equivalent to $\forall$`stmts.`$\{A_1 \wedge \neg A_3\}$`stmts`$\{\neg A_2\}$ – that is, with an infinite conjunction of Hoare triples. This leaves the challenge that no Hoare logics support such infinite conjunctions. Three breakthroughs lead to our solution of that challenge:

Per-method specs The Hoare triple $\{A_1 \wedge \neg A_3\}$ `x.m(ys)` $\{\neg A_2\}$ is logically equivalent to the specification `from` $(A_1 \wedge \langle$`_ calls x.m(ys)`$\rangle)$ `next` A_2 `onlyIf` A_3. With this, we can leverage Hoare triples to reason any call to one particular method .

Per-step specs If an assertion A_2 is *encapsulated* by a module (*i.e.* the only way we can go from a state that satisfies A_2 to a state that does not, is through a call to a method in that module), then the *finite conjunction* of `from` $(A_1 \wedge A_2 \wedge \langle$`_ calls x.m(ys)`$\rangle)$ `next` $\neg A_2$ `onlyIf` A_3 for all methods of that module is logically equivalent to `from` $A_1 \wedge A_2$ `next` $\neg A_2$ `onlyIf` A_3.

Proof logic for emergent behaviour combines several specifications to reason about the emergent behaviour, *e.g.*, `from` A_1 `to` A_2 `onlyThrough` A_3 and `from` A_1 `to` A_3 `onlyIf` A_4 implies `from` A_1 `to` A_2 `onlyIf` A_4.

Our work is parametric with respect to assertion satisfaction, encapsulation, and the type system. We require classical specifications to have explicit framing. Moreover, in line with other work in the literature, we forbid "callbacks" out to `external` objects (ie unknown objects whose code has not been checked). While our work is based on a simple, imperative, typed, object oriented language with unforgeable addresses and private fields, we believe that it is applicable to several programming paradigms, and that unforgeability and privacy can be replaced by lower level mechanisms such as capability machines [Davis et al. 2019; Van Strydonck et al. 2022].

1.3 Paper Organization and Contributions

The contributions of this paper are:

(1) A language to express *Necessity* specifications (§3).
(2) A logic for proving a module's adherence to *Necessity* specifications (§4), and a proof of soundness of the logic, (§??) both mechanized in Coq.
(3) A proof in our logic that our bank module obeys its *Necessity* specification (§5), also mechanized in Coq.

We place *Necessity* into the context of related work (§6) and consider our overall conclusions (§7). The Coq proofs of (2) and (3) above appear in the supplementary material, along with appendices containing expanded definitions and further examples.

2 OUTLINE OF OUR APPROACH

2.1 Bank Account – three modules

Module $\texttt{Mod}_{\texttt{good}}$ consists of an empty `Password` class where each instance models a unique password, and an `Account` class with a password, and a balance, an `init` method to initialize the password, and a `transfer` method.

```
1  module Mod_good
2    class Account
3      field balance:int
4      field pwd: Password
5      method transfer(dest:Account, pwd':Password) -> void
6        if this.pwd==pwd'
7          this.balance-=100
8          dest.balance+=100
9      method init(pwd:Pasword) -> void
10       if this.pwd==null
11         this.pwd=pwd'
12   class Password
```

We can capture the intended semantics of `transfer` through a "classical" specification with pre- and postconditions as *e.g.*, in [Leavens et al. 2007a; Leino 2013], with "explicit framing" – rather than `modifies`-clauses we give assertions about preservation of values. $\texttt{Mod}_{\texttt{good}}$'s implementation of the `transfer` method meets this specification.

```
1  ClassicSpec ≜
2     method transfer(dest:Account, pwd':Password) -> void {
3         ( PRE:  this.balance=bal1 ∧ this.pwd=pwd' ∧ dest.balance=bal2 ∧ dest≠this
4           POST: this.balance=bal1-100 ∧  dest.balance=bal2+100 )
5         ( PRE: this.balance=bal1 ∧ this.pwd≠pwd' ∧ dest.balance≠bal2
6           POST: this.balance=bal1 ∧  dest.balance=bal2 )
7         ( PRE: a : Account ∧ a≠this ∧ a≠dest  ∧ a.balance=bal ∧ a.pwd=pwd1
8           POST:  a.balance=bal ∧ a.pwd=pwd1)
9         ( PRE: a : Account ∧ a.pwd=pwd1
10          POST: a.pwd=pwd1)
```

Now consider the following alternative implementations: $\texttt{Mod}_{\texttt{bad}}$ allows any client to reset an account's password at any time; $\texttt{Mod}_{\texttt{better}}$ requires the existing password in order to change it.

```
1  module Mod_bad
2    class Account
3      field balance:int
4      field pwd: Password
5      method transfer(..) ...
6        ... as earlier ...
7      method init(...) ...
8        ... as earlier ...
9      method set(pwd': Password)
10       this.pwd=pwd'
11
12   class Password
```

```
1  module Mod_better
2    class Account
3      field balance:int
4      field pwd: Password
5      method transfer(..)
6        ... as earlier ...
7
8
9      method set(pwd',pwd'': Password)
10       if (this.pwd==pwd')
11         this.pwd=pwd''
12   class Password
```

Although the `transfer` method is the same in all three alternatives, and each one satisfies `ClassicSpec`, code such as

```
an_account.set(42); an_account.transfer(rogue_account,42)
```

is enough to drain `an_account` in $\texttt{Mod}_{\texttt{bad}}$ without knowing the password.

2.2 Bank Account – the right specification

We a specification that rules out $\texttt{Mod}_{\texttt{bad}}$ while permitting $\texttt{Mod}_{\texttt{good}}$ and $\texttt{Mod}_{\texttt{better}}$. The catch is that the vulnerability present in $\texttt{Mod}_{\texttt{bad}}$ is the result of *emergent* behaviour from the interactions of the `set` and `transfer` methods — even though $\texttt{Mod}_{\texttt{better}}$ also has a `set` method, it does not exhibit the unwanted interaction. This is exactly where a necessary condition can help: we want to avoid transferring money (or more generally, reducing an account's balance) *without* the existing account password. Phrasing the same condition the other way around rules out the theft: that money *can only* be transferred when the account's password is known.

In *Necessity* syntax, and recalling section 1.2,

```
S_robust_1 ≜  from  a:Account ∧ a.balance==bal  next  a.balance < bal
              onlyIf ∃ o,a'. [⟨o external⟩ ∧ ⟨o calls a.transfer(a.pwd,a')⟩]

S_robust_2 ≜  from  a:Account ∧ a.balance==bal  to  a.balance < bal
              onlyIf ∃ o.[⟨o external⟩ ∧ ⟨o access a.pwd⟩]
```

Indeed, all three modules satisfy $S_{\texttt{robust\_1}}$; this demonstrates that $S_{\texttt{robust\_1}}$ is not strong enough. On the other hand, $\texttt{Mod}_{\texttt{good}}$ and $\texttt{Mod}_{\texttt{better}}$ satisfy $S_{\texttt{robust\_2}}$, while $\texttt{Mod}_{\texttt{bad}}$ does not.

A critical point of $S_{\texttt{robust\_2}}$ is that it is expressed in terms of observable effects (the account's balance is reduced: `a.balance < bal`) and the shape of the heap (external access to the password: ⟨o external⟩ ∧ ⟨o access a.pwd⟩) rather than in terms of individual methods such as `set` and `transfer`. This gives our specifications the vital advantage that they can be used to constrain *implementation* of a bank account with a balance and a password, irrespective of the API it offers, the services it exports, or the dependencies on other parts of the system.

This example also demonstrates that adherence to *Necessity* specifications is not monotonic: adding a method to a module does not necessarily preserve adherence to a specification, and while separate methods may adhere to a specification, their combination does not necessarily do so. For example, $\texttt{Mod}_{\texttt{good}}$ satisfies $S_{\texttt{robust\_2}}$, while $\texttt{Mod}_{\texttt{bad}}$ does not. This is why we say that *Necessity* specifications capture a module's *emergent behaviour*.

2.2.1 How applicable is $S_{\texttt{robust\_2}}$? Finally, note that $S_{\texttt{robust\_2}}$ is applicable even though *some* external objects may have access to the account's password. Namely, if the account is passed to a scope which does not know the password, we know that its balance is guaranteed not to decrease.

We illustrate this through the following simplified code snippet.

```
1  module Mod_ext
2      ...
3      method expecting_payment(untrusted:Object){
4          a = new Account;
5          p = new Password;
6          a.set(null,p);
7          ...
8          untrusted.make_payment(a);
9          ...
10     }
11 }
```

The method `expecting_payment` has as argument an external object `untrusted`, of unknown provenance. It creates a new `Account` and initializes its password.

Assume that class `Account` is from a module which satisfies $S_{\texttt{robust\_2}}$. Then, in the scope of method `expecting_payment` external objects have access to the password, and the balance may decrease during execution of line 7, or line 9. However, if the code in line 7 does not leak the

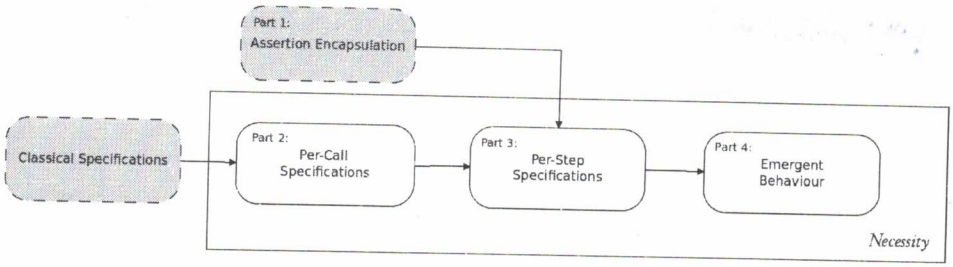


Fig. 1. Components of *Necessity* Logic and their Dependencies. Note that gray components with a dashed border indicate components that are not part of *Necessity*, and on which *Necessity* is parametric.

password to `untrusted`, then no external object in the scope of the call `make_payment` at line 8 has access to the password, and then, even though we are calling an untrusted object, $S_{\texttt{robust\_2}}$ guarantees that `untrusted` will not be able to take any money out of `a`.

2.3 Internal and External objects and calls

Our work concentrates on guarantees made in an *open* setting; that is, a given module M must be programmed so that execution of M together with *any* external module M' will uphold these guarantees. In the tradition of visible states semantics, we are only interested in upholding the guarantees while M', the *external* module, is executing. A module can temporarily break its own invariants, so long as the broken invariants are never visible externally.

We therefore distinguish between *internal* objects — instances of classes defined in M — and *external* objects defined in any other module, and between *internal* calls (from either an internal or an external object) made to internal objects and *external* calls made to external objects. Because we only require guarantees while the external module is executing, we develop an *external states* semantics, where any internal calls are executed in one, large, step. With external steps semantics, the executing object (`this`) is always external. Note that we do not support calls from internal objects to external objects.

2.4 Reasoning about Necessity

We now outline our *Necessity* logic. We elaborate further on the three breakthroughs described §1.2 along with two additions: (a) classical specifications, and (b) assertion encapsulation.

Our system consists of four parts (five including classical specifications): (**Part 1**) assertion encapsulation, (**Part 2**) per-call specifications, (**Part 3**) per-step specifications, and (**Part 4**) specifications of emergent behaviour. The structure of the system is given in Fig. 1. Classical specifications are used to prove per-call specifications, which coupled with assertion encapsulation is used to prove per-step specifications, which is used to prove specifications of emergent behaviour.

Classical specifications are assumed, and not included as part of the proof system, as they have been well covered in the literature. Our proofs of *Necessity* do not inspect method bodies: we rely on simple annotations to infer encapsulation, and on classical pre and postconditions to infer per-method conditions. Our system is agnostic as to how the pre- and post-conditions are proven, and thus we can leverage the results of many different approaches.

An assertion A is *encapsulated* by module M, if A can be invalidated only through calls to methods defined in M. In other words, a call to M is a *necessary* condition for invalidation of A. Our *Necessity* logic is parametric with respect to the particular encapsulation mechanism: examples in this paper rely on rudimentary annotations inspired by confinement types [Vitek and Bokowski 1999].

For illustration, we outline a proof that $\texttt{Mod}_{\texttt{better}}$ adheres to $S_{\texttt{robust\_2}}$.

Part 1: Assertion Encapsulation.

We begin by proving that $\texttt{Mod}_{\texttt{better}}$ encapsulates 3 properties:
(A) The balance
(B) The password
(C) External accessibility to an account's password – that is, the property that no external object has access to the password may only be invalidated by calls to $\texttt{Mod}_{\texttt{better}}$.

Part 2: Per-Call Specifications

We prove that the call of any method from $\texttt{Mod}_{\texttt{better}}$ (`set` and `transfer`) satisfies:
(D) If the balance decreases, then the method called was `transfer` with the correct password
(E) If the password changes, then the method called was `set` with the correct password
(F) It will not provide external accessibility to the password.

Part 3: Per-step Specifications

We then raise our results of Parts 1 and 2 to reason about arbitrary *single-step* executions:
(F) By **(A)** and **(D)** only `transfer` and external knowledge of the password may decrease the balance.
(G) By **(B)** and **(E)** only `set` and external knowledge of the password may change the password.
(H) By **(C)** and **(F)** no step may grant external accessibility to an account's password.

Part 4: Specifications of Emergent behaviour

We then raise our necessary conditions of Part 3 to reason about *arbitrary* executions:
(I) A decrease in balance over any number of steps implies that some single intermediate step reduced the account's balance.
(J) By **(F)** we know that step must be an external call to `transfer` with the correct password.
(K) When `transfer` was called, either
(K1) The password used was the same as the current password, and thus by **(H)** we know that the current password must be externally known, satisfying $S_{\texttt{robust\_2}}$, or
(K2) The password had been changed, and thus by **(G)** some intermediate step must have been a call to `set` with the current password. Thus, by **(H)** we know that the current password must be externally known, satisfying $S_{\texttt{robust\_2}}$.

3 THE MEANING OF NECESSITY

In this section we define the *Necessity* specification language. We first define an underlying programming language, TooL (§3.1). We then define an assertion language, *Assert*, which can talk about the contents of the state, as well as about provenance, permission and control (§3.2). Finally, we define the syntax and semantics of our full language for writing *Necessity* specifications (§3.3).

3.1 TooL

TooL is a formal model of a small, imperative, sequential, class based, typed, object-oriented language. TooL is straightforward: Appendix A contains the full definitions. TooL is based on $\mathscr{L}_{\text{oo}}$ [Drossopoulou et al. 2020b], with some small variations, as well as the addition of a simple type system – more in ??. A TooL state σ consists of a heap χ, and a stack ψ which is a sequence of frames. A frame ϕ consists of local variable map, and a continuation, *i.e.* a sequence of statements

to be executed. A statement may assign to variables, create new objects and push them to the heap, perform field reads and writes on objects, or call methods on those objects.

Modules are mappings from class names to class definitions. Execution is in the context of a module M and a state σ, defined via unsurprising small-step semantics of the form $M, \sigma \rightsquigarrow \sigma'$. The top frame's continuation contains the statements to be executed next.

As discussed in §2.4, we are interested in guarantees which hold during execution of an internal, known, trusted module M when linked together with any unknown, untrusted, module M'. These guarantees need only hold when the external module is executing; we are not concerned if they are temporarily broken by the internal module. Therefore, we are only interested in states where the executing object (`this`) is an external object. To express our focus on external states, we define the *external states semantics*, of the form $M'; M,\ \sigma \rightsquigarrow \sigma'$, where M' is the external module, and M is the internal module, and where we collapse all internal steps into one single step.

Definition 3.1 (External States Semantics). For modules M, M', and states σ, σ', we say that $M'; M,\ \sigma \rightsquigarrow \sigma'$ if and only if there exist $n \in \mathbb{N}$, and states $\sigma_0,...\sigma_n$, such that

- $\sigma{=}\sigma_1$, and $\sigma'{=}\sigma_n$,
- $M' \circ M, \sigma_i \rightsquigarrow \sigma_{i+1}$ for all $i \in [0..n)$,
- $classOf(\sigma, \texttt{this}), classOf(\sigma', \texttt{this}) \in M'$,
- $classOf(\sigma_i, \texttt{this}) \in M$ for all $i \in (1..n)$.

The function $classOf(\sigma, \_)$ is overloaded: applied to a variable, $classOf(\sigma, x)$ looks up the variable x in the top frame of σ, and returns the class of the corresponding object in the heap of σ; applied to an address, $classOf(\sigma, \alpha)$ returns the class of the object referred by address α in the heap of σ. The module linking operator $\circ$, applied to two modules, $M' \circ M$, combines the two modules into one module in the obvious way, provided their domains are disjoint. Full details in Appendix A.

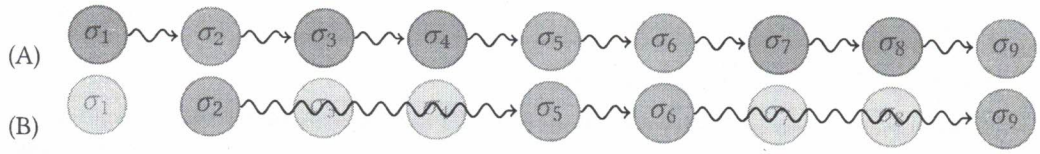


Fig. 2. External States Semantics (Def. 3.1), (A) $M' \circ M, \sigma_1 \rightsquigarrow \ldots \rightsquigarrow \sigma_9$ (B) $M'; M,\ \sigma_2 \rightsquigarrow \ldots \rightsquigarrow \sigma_9$, where $classOf(\sigma_1, \texttt{this}), classOf(\sigma_3, \texttt{this}), classOf(\sigma_4, \texttt{this}), classOf(\sigma_7, \texttt{this}), classOf(\sigma_8, \texttt{this}) \in M$, and where $classOf(\sigma_2, \texttt{this}), classOf(\sigma_5, \texttt{this}) classOf(\sigma_6, \texttt{this}), classOf(\sigma_9, \texttt{this}) \in M'$.

Fig. 2 inspired by Drossopoulou et al. [2020b] provides a simple graphical description of our external states semantics: (A) is the "normal" execution after linking two modules into one: $M' \circ M, \ldots \rightsquigarrow \ldots$ whereas (B) is the external states execution when M' is external, $M'; M,\ \ldots \rightsquigarrow \ldots$. Note that whether a module is external or internal depends on our perspective – nothing in a module itself renders it internal or external. For example, in $M_1; M_2,\ \ldots \rightsquigarrow \ldots$ the external module is M_1, while in $M_2; M_1,\ \ldots \rightsquigarrow \ldots$ the external module is M_2.

We use the notation $M'; M,\ \sigma \rightsquigarrow^* \sigma'$ to denote zero or more steps starting at state σ and ending at state σ', in the context of internal module M and external module M'. We are concerned neither with internal states nor states that can never arise. A state σ is *arising*, written $Arising(M', M, \sigma)$, if it may arise by external states execution starting at some initial configuration:

Definition 3.2 (Arising States). For modules M and M', a state σ is called an *arising* state, formally $Arising(M', M, \sigma)$, if and only if there exists some σ_0 such that $Initial(\sigma_0)$ and $M'; M,\ \sigma_0 \rightsquigarrow^* \sigma$.

An *Initial* state's heap contains a single object of class `Object`, and its stack consists of a single frame, whose local variable map is a mapping from `this` to the single object, and whose continuation is any statement. (See Definitions A.5 and 3.2).

3.1.1 Callbacks. Necessity does not –yet– support calls of external methods from within internal modules. This is, indeed, a limitation, but is common in the related literature. For example, VerX [?] work on effectively call-back free contracts, while [?] and [?] on drastically restricting the effect of a callback on a contract. In further work we are planning to incorporate callbacks by splitting internal methods at the point where a call to an external method appears.

3.2 Assert

Assert is a subset of the *Chainmail* assertions language, *i.e.* a basic assertion language extended with object-capability assertions.

3.2.1 Syntax of of Assert. The syntax of *Assert* is given in Definition 3.3. An assertion may be an expression, a query of the defining class of an object, the usual connectives and quantifiers, along with three non-standard assertion forms: (1) *Permission* and (2) *Provenance*, inspired by the capabilities literature, and (3) *Control* which allows tighter characterisation of the cause of effects – useful for the specification of large APIs.

- *Permission* ($\langle x \texttt{ access } y \rangle$): x has access to y.
- *Provenance* ($\langle x \texttt{ internal} \rangle$ and $\langle y \texttt{ external} \rangle$): x is internal, and y is external.
- *Control* ($\langle x \texttt{ calls } y.m(\overline{z}) \rangle$): x calls method m on object y with arguments $\overline{z}$.

Definition 3.3. Assertions (A) in *Assert* are defined as follows:

$$\begin{aligned} A \quad ::= \quad & e \mid e : C \mid \neg A \mid A \wedge A \mid A \vee A \mid \forall x.[A] \mid \exists x.[A] \\ & \mid \langle x \texttt{ access } y \rangle \mid \langle x \texttt{ internal} \rangle \mid \langle x \texttt{ external} \rangle \mid \langle x \texttt{ calls } y.m(\overline{z}) \rangle \end{aligned}$$

3.2.2 Semantics of Assert. The semantics of *Assert* is given in Definition 3.4. We use the evaluation relation, $M, \sigma, e \hookrightarrow v$, which says that the expression e evaluates to value v in the context of state σ and module M. Note that expressions in TooL may be recursively defined, and thus evaluation need not always terminate. Nevertheless, the logic of A remains classical because recursion is restricted to expressions, and not generally to assertions. We have taken this approach from Drossopoulou et al. [2020b], which also contains a mechanized Coq proof that assertions are classical [Drossopoulou et al. 2020a]. The semantics of $\hookrightarrow$ are unsurprising (see Fig.6).

Shorthands: $\lfloor x \rfloor_\phi = v$ means that x maps to value v in the local variable map of frame ϕ, $\lfloor x \rfloor_\sigma = v$ means that x maps to v in the top most frame of σ's stack, and $\lfloor x.f \rfloor_\sigma = v$ has the obvious meaning. The terms $\sigma.\texttt{stack}$, $\sigma.\texttt{contn}$, $\sigma.\texttt{heap}$ mean the stack, the continuation at the top frame of σ, and the heap of σ. The term $\alpha \in \sigma.\texttt{heap}$ means that α is in the domain of the heap of σ, and x *fresh in* σ means that x isn't in the variable map of the top frame of σ, while the substitution $\sigma[x \mapsto \alpha]$ is applied to the top frame of σ. $C \in M$ means that class C is in the domain of module M.

Definition 3.4 (Satisfaction of Assertions by a module and a state). We define satisfaction of an assertion A by a state σ with module M as:

(1) $M, \sigma \vDash e$ iff $M, \sigma, e \hookrightarrow \texttt{true}$
(2) $M, \sigma \vDash e : C$ iff $M, \sigma, e \hookrightarrow \alpha$ *and* $\mathit{classOf}(\sigma, \alpha) = C$
(3) $M, \sigma \vDash \neg A$ iff $M, \sigma \nvDash A$
(4) $M, \sigma \vDash A_1 \wedge A_2$ iff $M, \sigma \vDash A_1$ and $M, \sigma \vDash A_2$
(5) $M, \sigma \vDash A_1 \vee A_2$ iff $M, \sigma \vDash A_1$ or $M, \sigma \vDash A_2$
(6) $M, \sigma \vDash \forall x.[A]$ iff $M, \sigma[x \mapsto \alpha] \vDash A$, for some x fresh in σ, and for all $\alpha \in \sigma.\texttt{heap}$.

(7) $M, \sigma \vDash \exists\, x.[A]$ iff $M, \sigma[x \mapsto \alpha] \vDash A$, for some x fresh in σ, and for some $\alpha \in \sigma.\texttt{heap}$.
(8) $M, \sigma \vDash \langle x\ \texttt{access}\ y \rangle$ iff
  (a) $\lfloor x.f \rfloor_\sigma = \lfloor y \rfloor_\sigma$ for some f,
    or
  (b) $\lfloor x \rfloor_\sigma = \lfloor \texttt{this} \rfloor_\phi$, $\lfloor y \rfloor_\sigma = \lfloor z \rfloor_\phi$, and $z \in \phi.\texttt{contn}$ for some variable z, and some frame ϕ in $\sigma.\texttt{stack}$.
(9) $M, \sigma \vDash \langle x\ \texttt{internal} \rangle$ iff $\mathit{classOf}(\sigma, x) \in M$
(10) $M, \sigma \vDash \langle x\ \texttt{external} \rangle$ iff $\mathit{classOf}(\sigma, x) \notin M$
(11) $M, \sigma \vDash \langle x\ \texttt{calls}\ y.m(z_1, \ldots, z_n) \rangle$ iff
  (a) $\sigma.\texttt{contn} = (w := y'.m(z'_1, \ldots, z'_n); s)$, for some variable w, and some statement s,
  (b) $M, \sigma \vDash x = \texttt{this}$ and $M, \sigma \vDash y = y'$,
  (c) $M, \sigma \vDash z_i = z'_i$ for all $1 \le i \le n$

The assertion $\langle x\ \texttt{access}\ y \rangle$ (defined in 8) requires that x has access to y either through a field of x (case 8a), or through some call in the stack, where x is the receiver and y is one of the arguments (case 8b). Note that access is not deep, and only refers to objects that an object has direct access to via a field or within the context of the current scope. The restricted form of access used in *Necessity* specifically captures a crucial property of robust programs in the open world: access to an object does not imply access to that object's internal data. For example, an object may have access to an account `a`, but a safe implementation of the account would never allow that object to leverage that access to gain direct access to `a.pwd`.

The assertion $\langle x\ \texttt{calls}\ y.m(z_1, \ldots, z_n) \rangle$ (defined in 11) requires that the current receiver (`this`) is x, and that it calls the method m on y with arguments $z_1, \ldots z_n$ – It does *not* mean that somewhere in the call stack there exists a call from x to $y.m(\ldots)$. Note that in most cases, satisfaction of an assertion not only depends on the state σ, but also depends on the module in the case of expressions (1), class membership (2), and internal or external provenance (9 and 10).

We now define what it means for a module to satisfy an assertion: M satisfies A if any state arising from external steps execution of that module with any other external module satisfies A.

Definition 3.5 (Satisfaction of Assertions by a module). For a module M and assertion A, we say that $M \vDash A$ if and only if for all modules M', and all σ, if $\mathit{Arising}(M', M, \sigma)$, then $M, \sigma \vDash A$.

In the current work we assume the existence of a proof system that judges $M \vdash A$, to prove satisfaction of assertions. We will not define such a judgment, but will rely on its existence later on for Theorem ??. We define soundness of such a judgment in the usual way:

Definition 3.6 (Soundness of Assert Provability). A judgment of the form $M \vdash A$ is *sound*, if for all modules M and assertions A, if $M \vdash A$ then $M \vDash A$.

3.2.3 Inside. We define a final shorthand predicate `inside(o)` which states that only internal objects have access to `o`. The object `o` may be either internal or external.

Definition 3.7 (Inside). $\texttt{inside}(o) \triangleq \forall x.[\langle x\ \texttt{access}\ o \rangle \Rightarrow \langle x\ \texttt{internal} \rangle]$

`inside` is a very useful concept. For example, the balance of an account whose password is `inside` will not decrease in the next step. Often, API implementations contain objects whose capabilities, while crucial for the implementation, if exposed, would break the intended guarantees of the API. Such objects need to remain `inside`- see such an example in Section 5.

3.3 *Necessity* operators

The *Necessity* specification language extends *Assert* with our three novel *necessity operators*:

Only If [`from` A_1 `to` A_2 `onlyIf` A]: If an arising state satisfies A_1, and after some execution, a state satisfying A_2 is reached, then the original state must have also satisfied A.

Single-Step Only If [`from` A_1 `next` A_2 `onlyIf` A]: If an arising state satisfies A_1, and after a single step of execution, a state satisfying A_2 is reached, then the original state must have also satisfied A.

Only Through [`from` A_1 `to` A_2 `onlyThrough` A]: If an arising state satisfies A_1, and after some execution, a state satisfying A_2 is reached, then execution must have passed through some *intermediate* state satisfying A – the *intermediate* state satisfying A might be the *starting* state, the *final* state, or any state in between.

Necessity operators can explicitly constrain two or even three states, and implicitly constrain many states in between. The following specification $S_{\texttt{to\_dcr\_thr\_call}}$ says that for an account's balance to go from 350 in one state down to 250 some subsequent state, `transfer` must have been called on that account in between:

```
1  S_to_dcr_thr_call  ≜  from a:Account ∧ a.balance == 350  to a.balance == 250
2                          onlyThrough ⟨_ calls a.transfer(_,_)⟩
```

$S_{\texttt{to\_dcr\_thr\_call}}$ refers to two or even three different states: at the start, whenever the balance of `a` is `350`, and after any number of steps whenever the balance of `a` is `250`. The specification requires that such a change can only be caused by a call to `transfer` on `a`: that call could be in the current (starting) state, in which case presumably the balance will be `250` in the immediately following state, or within any number of intervening states.

Relationship between Necessity Operators. The three *Necessity* operators are related by generality. *Only If* (`from` A_1 `to` A_2 `onlyIf` A) implies *Single-Step Only If* (`from` A_1 `next` A_2 `onlyIf` A), since if A is a necessary precondition for multiple steps, then it must be a necessary precondition for a single step. *Only If* also implies an *Only Through*, where the intermediate state is the starting state of the execution. There is no further relationship between *Single-Step Only If* and *Only Through*.

Relationship with Temporal Logic. Two of the three *Necessity* operators can be expressed in traditional temporal logic: `from` A_1 `to` A_2 `onlyIf` A can be expressed as $A_1 \wedge \Diamond A_2 \longrightarrow A$, and `from` A_1 `next` A_2 `onlyIf` A can be expressed as $A_1 \wedge \bigcirc A_2 \longrightarrow A$ (where $\Diamond$ denotes any future state, and $\bigcirc$ denotes the next state). Critically, `from` A_1 `to` A_2 `onlyThrough` A cannot be encoded in temporal logics without "nominals" (explicit state references), because the state where A holds must be between the state where A_1 holds, and the state where A_2 holds; and this must be so on *every* execution path from A_1 to A_2 [Braüner 2022; Brotherston et al. 2020]. TLA+, for example, cannot describe "only through" conditions [Lamport 2002], but we have found "only through" conditions critical to our proofs.

3.3.1 *Semantics of Necessity Specifications.*

We define when a module M satisfies specifications S, written as $M \vDash S$, by cases over the four possible syntactic forms:

Definition 3.8 (Necessity Syntax).

$$S \quad ::= \quad A \mid \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A_3 \mid \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_3 \mid \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3$$

Definition 3.9 (Necessity Semantics). For any assertions A_1, A_2, and A, we define

- $M \vDash A$ iff for all M', σ, if $\mathit{Arising}(M', M, \sigma)$, then $M, \sigma \vDash A$. (see Def. 3.5)

- $M \vDash$ `from` A_1 `to` A_2 `onlyIf` A iff for all M', σ, σ', such that $Arising(M', M, \sigma)$;

$$\left.\begin{array}{l} \text{- } M, \sigma \vDash A_1 \\ \text{- } M, \sigma' \triangleleft \sigma \vDash A_2 \\ \text{- } M'; M,\ \sigma \rightsquigarrow^* \sigma' \end{array}\right\} \Rightarrow \quad M, \sigma \vDash A$$

- $M \vDash$ `from` A_1 `next` A_2 `onlyIf` A iff for all M', σ, σ', such that $Arising(M', M, \sigma)$:

$$\left.\begin{array}{l} \text{- } M, \sigma \vDash A_1 \\ \text{- } M, \sigma' \triangleleft \sigma \vDash A_2 \\ \text{- } M'; M,\ \sigma \rightsquigarrow \sigma' \end{array}\right\} \Rightarrow \quad M, \sigma \vDash A$$

- $M \vDash$ `from` A_1 `to` A_2 `onlyThrough` A iff for all M', σ_1, σ_n, such that $Arising(M', M, \sigma_1)$:

$$\left.\begin{array}{l} \text{- } M, \sigma_1 \vDash A_1 \\ \text{- } M, \sigma_n \triangleleft \sigma_1 \vDash A_2 \\ \text{- } M'; M,\ \sigma_1 \rightsquigarrow^* \sigma_n \end{array}\right\} \Rightarrow \begin{array}{l} \forall \sigma_2, \ldots, \sigma_{n-1}. \\ (\ \forall i \in [1..n).\ M'; M,\ \sigma_i \rightsquigarrow \sigma_{i+1} \Rightarrow \exists i \in [1..n].\ M, \sigma_i \triangleleft \sigma_1 \vDash A\ ) \end{array}$$

3.3.2 Adaptation. The definition of the necessity operators (Definition 3.9) is straightforward, apart from one quirk: we write $\sigma' \triangleleft \sigma$ (best read as "σ' seen from σ", although one recalcitrant author prefers "σ' adapted to σ") to deal with the fact that necessity operators can involve several states. To see the problem, consider a naïve approach to giving semantics to $S_{\texttt{to\_dcr\_thr\_call}}$: if $.., \sigma \vDash$ `a.balance==350`, and $.., \sigma \rightsquigarrow^* \sigma'$ and $\sigma' \vDash$ `a.balance==250`, then $S_{\texttt{to\_dcr\_thr\_call}}$ mandates that between σ and σ' there was a call to `a.transfer`. But if σ happened to have another account `a1` with balance `350`, and if we reach σ' from σ by executing `a1.transfer(..,..); a=a1`, then we would reach a σ' *without* `a.transfer` having been called: indeed, without the account `a` from σ having changed at all! (Haskell programmers will probably feel at home here).

This is the remit of the adaptation operator: when we consider the future state, we must "see it from" the perspective of the current state; the binding for variables such as `a` must be from the current state, even though we may have assigned to them in the mean time. Thus, $\sigma' \triangleleft \sigma$ keeps the heap from σ', and renames the variables in the top stack frame of σ' so that all variables defined in σ have the same bindings as in σ; the continuation must be adapted similarly (see Fig. 3).

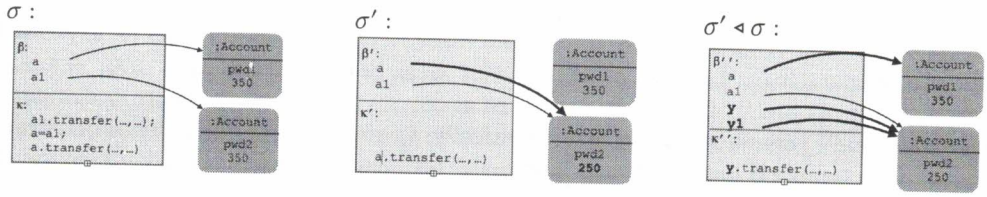


Fig. 3. Illustrating adaptation

Under adaptation, the semantics of $S_{\texttt{to\_dcr\_thr\_call}}$ is: if $.., \sigma \vDash$ `a.balance==350`, and $.., \sigma \rightsquigarrow^* \sigma'$ and $...$, $\sigma' \triangleleft \sigma \vDash$ `a.balance==250`, some intermediate state's continuation contains a call to `a.transfer`; where every reference to `a` in any part of the necessity operator refers to the object bound to that variable in the initial state. Fig. 3 illustrates this semantics. In state σ the variable `a` points to an `Account` object with password `pwd1`, and balance `350`, the variable `a1` points to an `Account` object with password `pwd2`, and balance `350`, and the continuation is `a1.transfer(..,..); a=a1; a.transfer(..,..);`. We reach σ' by executing the first two statements from the continuation. Thus, $\sigma' \triangleleft \sigma \nvDash$ `a.balance==250`. Moreover, in $\sigma' \triangleleft \sigma$ we

introduce the fresh variables `y` and `y1`, and replace `a` and `a1` by `y` and `y1` in the continuation. This gives that $\sigma' \triangleleft \sigma \models \langle$`_ calls a1.transfer(...)`$\rangle$ and $\sigma' \triangleleft \sigma \not\models \langle$`_ calls a.transfer(...)`$\rangle$.

Definition 3.10 gives the full definition of the $\triangleleft$ operator (equivalent to the *adaptation* operator from [Drossopoulou et al. 2020b]). We introduce fresh variables $\overline{y}$ – as many as in the σ' top frame variable map – $dom(\beta') = \overline{x}$, and $|\overline{y}| = |\overline{x}|$. We extend σ's variable map (β), so that it also maps $\overline{y}$ in the way that σ''s variable map (β') maps its local variables – $\beta'' = \beta[\overline{y} \mapsto \beta'(\overline{x})]$. We rename $\overline{x}$ in σ' continuation to $\overline{y}$ – $\kappa'' = [\overline{y}/\overline{x}]\kappa'$.

Definition 3.10. For any states σ, σ', heaps χ, χ', variable maps β, β', and continuations κ, κ', such that σ=$(\chi, (\beta, \kappa) : \psi)$, and σ=$(\chi', (\beta', \kappa') : \psi')$, we define

- $\sigma' \triangleleft \sigma \triangleq (\chi', (\beta'', \kappa'') : \psi')$
  where there exist variables $\overline{y}$ such that
  - $\beta'' = \beta[\overline{y} \mapsto \beta'(\overline{x})]$, and $\kappa'' = [\overline{y}/\overline{x}]\kappa'$
  - $dom(\beta') = \overline{x}$, and $|\overline{y}| = |\overline{x}|$, and $\overline{y}$ are fresh in β and β'.

Strictly speaking, $\triangleleft$ does not define one unique state: Because variables $\overline{y}$ are arbitrarily chosen, $\triangleleft$ describes an infinite set of states. These states satisfy the same assertions and therefore are equivalent with each other. This is why it is sound to use $\triangleleft$ as an operator, rather than as a set.

3.4 More Examples expressed in *Necessity*

In this section we introduce some further specification examples, and use them to elucidate finer points in the semantics of *Necessity*. We also discuss which modules satisfy which specifications.

3.4.1 More examples of the Bank. Looking back at the examples from §2.2, it holds that

$\texttt{Mod}_{\texttt{good}} \vDash S_{\texttt{robust\_1}}$ $\quad \texttt{Mod}_{\texttt{bad}} \vDash S_{\texttt{robust\_1}}$ $\quad \texttt{Mod}_{\texttt{better}} \vDash S_{\texttt{robust\_1}}$

$\texttt{Mod}_{\texttt{good}} \vDash S_{\texttt{robust\_2}}$ $\quad \texttt{Mod}_{\texttt{bad}} \nvDash S_{\texttt{robust\_2}}$ $\quad \texttt{Mod}_{\texttt{better}} \vDash S_{\texttt{robust\_2}}$

Consider now another four *Necessity* specifications:

```
S_nxt_dcr_if_acc  ≜ from a:Account ∧ a.balance==bal  next a.balance < bal
                     onlyIf ∃ o.[⟨o external⟩ ∧ ⟨o access a.pwd⟩]

S_nxt_dcr_if_call ≜ from a:Account ∧ a.balance==bal  next a.balance < bal
                     onlyIf ∃ o.[⟨o external⟩ ∧ ⟨o calls a.transfer(_, _, _)⟩]

S_to_dcr_if_call  ≜ from a:Account ∧ a.balance==bal to a.balance < bal
                     onlyIf ∃ o.[⟨o external⟩ ∧ ⟨o calls a.transfer(_, _, _)⟩]

S_to_dcr_thr_call ≜ from a:Account ∧ a.balance==bal to a.balance < bal
                     onlyThrough ∃ o.[⟨o external⟩ ∧ ⟨o calls a.transfer(_, _, _)⟩]
```

The specification $S_{\texttt{nxt\_dcr\_if\_acc}}$ states that the balance of an account decreases *in one step*, only if an external object has access to the password. It a weaker specification than $S_{\texttt{robust\_2}}$, because it applies when the decrease takes place in *one* step, rather than in *a number* of steps. Even though $\texttt{Mod}_{\texttt{bad}}$ does not satisfy $S_{\texttt{robust\_2}}$, it does satisfy $S_{\texttt{nxt\_dcr\_if\_acc}}$:

$\texttt{Mod}_{\texttt{good}} \vDash S_{\texttt{nxt\_dcr\_if\_acc}}$ $\quad \texttt{Mod}_{\texttt{bad}} \vDash S_{\texttt{nxt\_dcr\_if\_acc}}$ $\quad \texttt{Mod}_{\texttt{better}} \vDash S_{\texttt{nxt\_dcr\_if\_acc}}$

The specifications $S_{\texttt{nxt\_dcr\_if\_call}}$ and $S_{\texttt{to\_dcr\_if\_call}}$ are similar: they both say that a decrease of the balance can only happen if the current statement is a call to `transfer`. The former considers a *single* step, while the latter allows for *any number* of steps. $S_{\texttt{robust\_2}}$ is slightly different, because it says that such a decrease is only possible if some *intermediate* step called `transfer`. All three modules satisfy $S_{\texttt{nxt\_dcr\_if\_call}}$. On the other hand, the code `a1=new Account; a2.transfer(...)` decrements the balance of `a2` and does call `transfer` but not as a first step; therefore, none of the modules satisfy $S_{\texttt{to\_dcr\_if\_call}}$. That is:

$\texttt{Mod}_{\texttt{good}} \vDash S_{\texttt{nxt\_dcr\_if\_call}}$ $\texttt{Mod}_{\texttt{bad}} \vDash S_{\texttt{nxt\_dcr\_if\_call}}$ $\texttt{Mod}_{\texttt{better}} \vDash S_{\texttt{nxt\_dcr\_if\_call}}$
$\texttt{Mod}_{\texttt{good}} \nvDash S_{\texttt{to\_dcr\_if\_call}}$ $\texttt{Mod}_{\texttt{bad}} \nvDash S_{\texttt{to\_dcr\_if\_call}}$ $\texttt{Mod}_{\texttt{better}} \nvDash S_{\texttt{to\_dcr\_if\_call}}$

Finally, $S_{\texttt{to\_dcr\_thr\_call}}$ is a weaker requirement than $S_{\texttt{to\_dcr\_if\_call}}$, because it only asks that the `transfer` method is called in *some intermediate* step. All modules satisfy it:

$\texttt{Mod}_{\texttt{good}} \vDash S_{\texttt{to\_dcr\_thr\_call}}$ $\texttt{Mod}_{\texttt{bad}} \vDash S_{\texttt{to\_dcr\_thr\_call}}$ $\texttt{Mod}_{\texttt{better}} \vDash S_{\texttt{to\_dcr\_thr\_call}}$

3.4.2 The DOM. This is the motivating example in [Devriese et al. 2016], dealing with a tree of DOM nodes: Access to a DOM node gives access to all its `parent` and `children` nodes, with the ability to modify the node's `property` – where `parent`, `children` and `property` are fields in class `Node`. Since the top nodes of the tree usually contain privileged information, while the lower nodes contain less crucial third-party information, we must be able to limit access given to third parties to only the lower part of the DOM tree. We do this through a `Proxy` class, which has a field `node` pointing to a `Node`, and a field `height`, which restricts the range of `Nodes` which may be modified through the use of the particular `Proxy`. Namely, when you hold a `Proxy` you can modify the `property` of all the descendants of the `height`-th ancestors of the `node` of that particular `Proxy`. We say that `pr` has *modification-capabilities* on `nd`, where `pr` is a `Proxy` and `nd` is a `Node`, if the `pr.height`-th `parent` of the node at `pr.node` is an ancestor of `nd`.

The specification `DOMSpec` states that the `property` of a node can only change if some external object presently has access to a node of the DOM tree, or to some `Proxy` with modification-capabilties to the node that was modified.

```
DOMSpec ≜ from nd : Node ∧ nd.property = p  to nd.property != p
          onlyIf ∃ o.[ ⟨o external⟩ ∧
                       ( ∃ nd':Node.[ ⟨o access nd'⟩ ]  ∨
                         ∃ pr:Proxy,k:ℕ.[⟨o access pr⟩ ∧ nd.parent^k=pr.node.parent^pr.height ]
```

3.4.3 Expressiveness. In order to investigate *Necessity*'s expressiveness, we used it for examples provided in the literature. In this section we considered the DOM example, proposed by Devriese et al.. In Appendix C, we compare with examples proposed by Drossopoulou et al..

4 PROVING NECESSITY

In this section we provide an inference system for constructing proofs of the *Necessity* specifications defined in §3.3. As discussed in §2.4, four concerns are involved in the proof of *Necessity* specifications:

(1) Proving Assertion Encapsulation (§??)
(2) Proving *Necessity* specifications from classical specifications for a single internal method (§??)
(3) Proving module-wide Single-Step *Necessity* specifications by combining per-method *Necessity* specifications (§??)
(4) Raising necessary conditions to construct proofs of emergent behaviour (§??)

4.1 Assertion Encapsulation

In Section 2 we needed to prove that an assertion was encapsulated within a module while showing adherence to *Necessity* specifications. Specifically, a key component of constructing program wide *Necessity* proofs is the identification of properties that require internal (and thus known) computation to be invalidated. *Necessity* is parametric over the details of the encapsulation model [Noble et al. 2003]: Appendix B and Figure 8 present a rudimentary system that is sufficient to support our example proof. The key judgement we rely upon is *assertion encapsulation* that describes whether an assertion is encapsulated within a module.

4.1.1 Assertion Encapsulation Semantics. Assertion encapsulation models the informal notion that if an assertion A' is encapsulated by module M, then the validity of that assertion can only be changed via that module. In TooL, that means by calls to objects defined in M but that are accessible from the outside. We provide an intensional definition: A' is encapsulated if whenever we go from state σ to σ', and when the value of A' changes (i.e. to $\neg A'$) then we must have called a method on one of M's internal objects. In fact we rely on a slightly more subtle underlying definition, "conditional" encapsulation where $M \vDash A \implies Enc(A')$ expresses that in states which satisfy A, the assertion A' cannot be invalidated, unless a method from M was called.

Definition 4.1 (Assertion Encapsulation). An assertion A' is *encapsulated* by module M and assertion A, written as $M \vDash A \implies Enc(A')$, if and only if for all external modules M', and all states σ, σ' such that $Arising(M', M, \sigma)$:

$$\left.\begin{array}{l} - M'; M,\ \sigma \rightsquigarrow \sigma' \\ - M, \sigma' \triangleleft \sigma \vDash \neg A' \\ - M, \sigma \vDash A \wedge A' \end{array}\right\} \implies \exists x,\ m,\ \overline{z}.(\ M, \sigma \vDash \langle \_\ \texttt{calls}\ x.m(\overline{z})\rangle \wedge \langle x\ \texttt{internal}\rangle\ )$$

This definition uses adaptation $\sigma' \triangleleft \sigma$ because we have to interpret one assertion in two different states. Revisiting the examples from § 2, both $\texttt{Mod}_{\texttt{bad}}$ and $\texttt{Mod}_{\texttt{better}}$ encapsulate the `balance` of an account, because any change to it requires calling a method on an internal object.

$$\texttt{Mod}_{\texttt{bad}} \vDash \texttt{a : Account} \implies Enc(\texttt{a.balance = bal})$$
$$\texttt{Mod}_{\texttt{better}} \vDash \texttt{a : Account} \implies Enc(\texttt{a.balance = bal})$$

Put differently, a piece of code that does not contain calls to a certain module is guaranteed not to invalidate any assertions encapsulated by that module. Assertion encapsulation has been used in proof systems to solve the frame problem [Banerjee and Naumann 2005b; Leino and Müller 2004].

4.1.2 Proving Assertion Encapsulation. Our logic does not rely on the specifics of the encapsulation model, but only its soundness:

Definition 4.2 (Encapsulation Soundness). A judgment of the form $M \vdash A \implies Enc(A)$ is *sound*, if for all modules M, and assertions A_1 and A_2, if $M \vdash A_1 \implies Enc(A_2)$ then $M \vDash A_1 \implies Enc(A_2)$.

The key consequence of soundness is that an object inside a module (`inside(o)`) will always be encapsulated, in the sense that it can only leak out of the module via an internal call.

4.1.3 Types. To allow for an easy way to judge encapsulation of assertions, we assume a very simple type system, where field, method arguments and method results are annotated with classes, and the type system checks that field assignments, method calls, and method returns adhere to these expectations. Because the type system is so simple, we do not include its specification in the paper. Note however, that the type system has one further implication: modules are typed in isolation, thereby implicitly prohibiting method calls from internal objects to external objects.

Based on this type system, we define a predicate $Enc_e(e)$, in Appendix B, which asserts that any object reads during the evaluation of e are internal. Thus, any assertion that only involves $Enc_e(\_)$ expressions is encapsulated; more in Appendix B.

Finally, a further small addition to the type system assists the knowledge that an object is `inside`: Classes may be annotated as `confined`. A `confined` object cannot be accessed by external objects; that is, it is always `inside`. The type system needs to ensure that objects of `confined` type are never returned from method bodies, this is even simpler than in [Vitek and Bokowski 1999]. Again, we omit the detailed description of this simple type system.

$$\frac{M \vdash \{x : C \wedge P_1 \wedge \neg P\}\ \texttt{res} = x.m(\overline{z})\ \{\neg P_2\}}{M \vdash \texttt{from}\ P_1 \wedge x : C \wedge \langle \_\ \texttt{calls}\ x.m(\overline{z})\rangle\ \texttt{next}\ P_2\ \texttt{onlyIf}\ P} \quad \text{(If1-Classical)}$$

$$\frac{M \vdash \{x : C \wedge \neg P\}\ \texttt{res} = x.m(\overline{z})\ \{\texttt{res} \neq y\}}{M \vdash \texttt{from inside}(y) \wedge x : C \wedge \langle \_\ \texttt{calls}\ x.m(\overline{z})\rangle\ \texttt{next}\ \neg\texttt{inside}(y)\ \texttt{onlyIf}\ P} \quad \text{(If1-Inside)}$$

Fig. 4. Per-Method *Necessity* specifications

4.2 Per-Method *Necessity* Specifications

In this section we detail how we use classical specifications to construct per-method *Necessity* specifications. That is, for some method m in class C, we construct a specifications of the form:

$$\texttt{from}\ A_1 \wedge x : C \wedge \langle \_\ \texttt{calls}\ x.m(\ldots)\rangle\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A$$

Thus, A is a necessary precondition to reaching A_2 from A_1 via a method call m to an object of class C. Note that if a precondition and a certain statement is *sufficient* to achieve a particular result, then the negation of that precondition is *necessary* to achieve the negation of the result after executing that statment. Specifically, using classical Hoare logic, if $\{P\}\ s\ \{Q\}$ is true, then it follows that $\neg P$ is a *necessary precondition* for $\neg Q$ to hold following the execution of s.

We do not define a new assertion language and Hoare logic. Rather, we rely on prior work on such Hoare logics, and assume some underlying logic that can be used to prove *classical assertions*. Classical assertions are a subset of *Assert*, comprising only those assertions that are commonly present in other specification languages. We provide this subset in Definition ??. That is, classical assertions are restricted to expressions, class assertions, the usual connectives, negation, implication, and the usual quantifiers.

Definition 4.3. Classical assertions, P, Q, are defined as follows

$$P, Q \quad ::= \quad e \mid e : C \mid P \wedge P \mid P \vee P \mid P \longrightarrow P \mid \neg P \mid \forall x.[P] \mid \exists x.[P]$$

We assume that there exists some classical specification inference system that allows us to prove specifications of the form $M \vdash \{P\}\ \texttt{s}\ \{Q\}$. This implies that we can also have guarantees of

$$M \vdash \{P\}\ \texttt{res} = x.m(\overline{z})\ \{Q\}$$

That is, the execution of $x.m(\overline{z})$ with the precondition P results in a program state that satisfies postcondition Q, where the returned value is represented by res in Q.

Figure ?? introduces proof rules to infer per-method *Necessity* specifications. These are rules whose conclusion have the form Single-Step Only If.

If1-Classical states that if any state which satisfies P_1 and $\neg P$ and executes the method m on an object of class C, leads to a state that satisfies $\neg P_2$, then, any state which satisfies P_1 and calls m on an object of class C will lead to a state that satisfies P_2 only if the original state also satisfied P. We can explain this also as follows: If the triple $.. \vdash \{R_1 \wedge R_2\}\ s\ \{Q\}$ holds, then any state that satisfies R_1 and which upon execution of s leads to a state that satisfies $\neg Q$, cannot satisfy R_2 – because if it did, then the ensuing state would have to satisfy Q,

If1-Inside states that a method which does not return an object y preserves the "insidedness" of y. At first glance this rule might seem unsound, however the restriction on external calls ensures soundness of this rule. There are only four ways an object x might gain access to another object y: (1) y is created by x as the result of a new expression, (2) y is written to some field of x, (3) y

is passed to x as an argument to a method call on x, or (4) y is returned to x as the result of a method call from an object z that has access to y. The rules in Fig. ?? are only concerned with effects on program state resulting from a method call to some internal object, and thus (1) and (2) need not be considered as neither object creation or field writes may result in an external object gaining access from an internal object. Since we are only concerned with describing how internal objects grant access to external objects, our restriction on external method calls within internal code prohibits (3) from occuring. Finally, (4) is described by If1-Inside. In further work we plan to weaken the restriction on external method calls, and will strengthen this rule. In more detail, If1-Inside states that if P is a necessary precondition for returning an object y, then since we do not support calls from internal code to external code, it follows that P is a necessary precondition to leak y. If1-Inside is essentially a specialized version of If1-Classical for the `inside()` predicate. Since `inside()` is not a classical assertion, we cannot use Hoare logic to reason about necessary conditions for invalidating `inside()`.

4.3 Per-Step *Necessity* Specifications

$$\frac{\textit{for all } C \in dom(M) \textit{ and } m \in M(C).\text{mths}, \quad M \vdash \texttt{from}\ A_1 \wedge x : C \wedge \langle \_\ \texttt{calls}\ x.m(\overline{z}) \rangle\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3 \qquad M \vdash A_1 \longrightarrow \neg A_2 \qquad M \vdash A_1 \Rightarrow Enc(A_2)}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3} \text{ (If1-Internal)}$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A} \text{ (If1-If)}$$

$$\frac{M \vdash A_1 \longrightarrow A_1' \qquad M \vdash A_2 \longrightarrow A_2' \qquad M \vdash A_3' \longrightarrow A_3 \qquad M \vdash \texttt{from}\ A_1'\ \texttt{next}\ A_2'\ \texttt{onlyIf}\ A_3'}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3} \text{ (If1-}\longrightarrow\text{)}$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A \qquad M \vdash \texttt{from}\ A_1'\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A'}{M \vdash \texttt{from}\ A_1 \vee A_1'\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A \vee A'} \text{ (If1-}\vee\text{I}_1\text{)}$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A \qquad M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2'\ \texttt{onlyIf}\ A'}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2 \vee A_2'\ \texttt{onlyIf}\ A \vee A'} \text{ (If1-}\vee\text{I}_2\text{)}$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A \vee A' \qquad M \vdash \texttt{from}\ A'\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ \texttt{false}}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A} \text{ (If1-}\vee\text{E)}$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A \qquad M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A'}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A \wedge A'} \text{ (If1-}\wedge\text{I)} \qquad \frac{\forall y,\ M \vdash \texttt{from}\ ([y/x]A_1)\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ \exists x.[A_1]\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A} \text{ (If1-}\exists_1\text{)}$$

$$\frac{\forall y,\ M \vdash \texttt{from}\ A_1\ \texttt{next}\ ([y/x]A_2)\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ \exists x.[A_2]\ \texttt{onlyIf}\ A} \text{ (If1-}\exists_2\text{)}$$

Fig. 5. Single-Step *Necessity* Specifications

We now raise per-method *Necessity* specifications to per-step *Necessity* specifications. The rules appear in Figure ??.

If1-Internal lifts a per-method *Necessity* specification to a per-step *Necessity* specification. Any *Necessity* specification which is satisfied for any method calls sent to any object in a module, is satisfied for *any step*, even an external step, provided that the effect involved, *i.e.* going from A_1 states to A_2 states, is encapsulated.

$$\frac{M \vdash \texttt{from}\ A\ \texttt{next}\ \neg A\ \texttt{onlyIf}\ A'}{M \vdash \texttt{from}\ A\ \texttt{to}\ \neg A\ \texttt{onlyThrough}\ A'}\ (\textsc{Changes}) \qquad \frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_3 \quad M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_3\ \texttt{onlyThrough}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A}\ (\textsc{Trans}_1)$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_3 \quad M \vdash \texttt{from}\ A_3\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A}\ (\textsc{Trans}_2) \qquad \frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A}\ (\textsc{If})$$

$$M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_2 \quad (\textsc{End})$$

Fig. 6. Selected rules for *Only Through* – rest in Figure ??

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_3 \qquad M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_3\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A}\ (\textsc{If-Trans})$$

$$M \vdash \texttt{from}\ x : C\ \texttt{to}\ \neg\, x : C\ \texttt{onlyIf}\ \texttt{false}\ (\textsc{If-Class}) \qquad M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A_1\ (\textsc{If-Start})$$

Fig. 7. Selected rules for *Only If* – the rest in Figure 10

The remaining rules are more standard, and are reminiscent of the Hoare logic rule of consequence. We have five such rules:

The rule for implication (IF1-⟶) may strengthen properties of either the starting or ending state, or weaken the necessary precondition.

There are two disjunction introduction rules: (a) IF1-∨I1 states that any execution starting from a state satisfying some disjunction that reaches some future state, must pass through either a necessary intermediate state for the first branch, or a necessary intermediate state for the second branch. (b) IF1-∨I2 states that any execution starting from some state and ending in a state satisfying a disjunction must pass through either a necessary intermediate state for the first branch, or a necessary intermediate state for the second branch.

The disjunction elimination rule (IF1-∨E), is of note, as it mirrors typical disjunction elimination rules, with a variation stating that if it is not possible to reach the end state from one branch of the disjunction, then we can eliminate that branch.

Two rules support existential elimination on the left hand side. IF1-$\exists_1$ states that if any single step of execution starting from a state satisfying $[y/x]A_1$ for all possible y, reaching some state satisfying A_2 has A as a necessary precondition, it follows that any single step execution starting in a state where such a y exists, and ending in a state satisfying A_2, must have A as a necessary precondition. IF1-$\exists_2$ is a similar rule for an existential in the state resulting from the execution.

4.4 Emergent *Necessity* Specifications

We now show how per-step *Necessity* specifications are raised to multiple step *Necessity* specifications, allowing the specification of emergent behaviour. Figure ?? present some of the rules for the construction of proofs for *Only Through*, while Figure ?? provides some of the rules for the construction of proofs of *Only If*. The full rules can be found in Appendix D, and are not presented here in full so as not to repeat rules from Figure ??.

Only Through has several notable rules. CHANGES, in Figure ??, states that if the satisfaction of some assertion changes over time, then there must be some specific intermediate state where that change occurred. CHANGES is an important rule in the logic, and is an enabler for proofs of

emergent properties. It is this rule that ultimately connects program execution to encapsulated properties.

It may seem natural that CHANGES should take the more general form:

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_3}$$

This would not be sound as a transition from a state satisfying one assertion to one satisfying another assertion is not required to occur in a single step; however this is true for a change in satisfaction for a specific assertion (*i.e.* A to $\neg A$).

Only Through also includes two transitivity rules (TRANS$_1$ and TRANS$_2$) that say that necessary conditions to reach intermediate states or proceed from intermediate states are themselves necessary intermediate states.

Finally, *Only Through* includes END, stating that the ending condition is a necessary intermediate condition.

Moreover, any *Only If* specification entails the corresponding *Only Through* specification (IF). *Only If* also includes a transitivity rule (IF-TRANS), but since the necessary condition must be true in the beginning state, there is only a single rule. IF-CLASS captures the invariant that an object's class never changes. Finally, any starting condition is itself a necessary precondition (IF-START).

4.5 Soundness of the *Necessity* Logic

THEOREM 4.4 (SOUNDNESS). *Assuming a sound Assert proof system, $M \vdash A$, and a sound encapsulation inference system, $M \vdash A \Rightarrow Enc(A')$, and and that on top of these systems we built the Necessity logic according to the rules in Figures ??, and ??, and ??, and ??, then, for all modules M, and all Necessity specifications S:*

$$M \vdash S \quad \textit{implies} \quad M \vDash S$$

PROOF. by induction on the derivation of $M \vdash S$. □

Theorem. ?? demonstrates that the *Necessity* logic is sound with respect to the semantics of *Necessity* specifications. The *Necessity* logic parametric wrt to the algorithms for proving validity of assertions $M \vdash A$, and assertion encapsulation ($M \vdash A \Rightarrow Enc(A')$), and is sound provided that these two proof systems are sound.

The mechanized the proof of Theorem ?? in Coq can be found in the associated artifact. The Coq formalism deviates slightly from the system as presented here, mostly in the formalization of the *Assert* language. The Coq version of *Assert* restricts variable usage to expressions, and allows only addresses to be used as part of non-expression syntax. For example, in the Coq formalism we can write assertions like $x.f$ == `this` and $x == \alpha_y$ and $\langle \alpha_x$ `access` $\alpha_y \rangle$, but we cannot write assertions like $\langle x$ `access` $y \rangle$, where x and y are variables, and α_x and α_y are addresses. The reason for this restriction in the Coq formalism is to avoid spending significant effort encoding variable renaming and substitution, a well-known difficulty for languages such as Coq. This restriction does not affect the expressiveness of our Coq formalism: we are able to express assertions such as $\langle x$ `access` $y \rangle$, by using addresses and introducing equality expressions to connect variables to address, *i.e.* $\langle \alpha_x$ `access` $\alpha_y \rangle \wedge \alpha_x == x \wedge \alpha_y == y$.

5 PROOF OF ADHERENCE TO $S_{\texttt{robust\_2}}$

In this section we revisit our example from Sections 1 and 2: specifying a robust bank account. Recall that an `Account` includes at least a `balance` field, and a `transfer` method. In order to confidently pass our account to unknown, potentially malicious code, we wish to prove that

our Account and the module that contains Account satisfies $S_{\texttt{robust\_2}}$. Below we rephrase $S_{\texttt{robust\_2}}$ to use the inside(_) predicate.

```
1  S_robust_2 ≜ from a:Account ∧ a.Account=bal
2               to a.balance < bal   onlyIf ¬inside(a.password)
```

That is, if the balance to my account ever decreases, then some external object must know my password. Or conversely, if only I know my password, then I know my money is safe. Here we provide a proof that $\texttt{Mod}_{\texttt{better}}$ satisfies $S_{\texttt{robust\_2}}$. An outline of this proof has already been discussed in §2.4.

not technically the converse?

5.1 Part 1: Assertion Encapsulation

The first part of the proof is proving that the Account balance, password, and external accessibility to the password are encapsulated properties. That is, for the balance to change (i.e. for a.balance = bal to be invalidated), internal computation is required. We use a conservative approach to an encapsulation system, detailed in App. B, and provide the proof steps below.

BalEncaps:

aEnc:
$\texttt{Mod}_{\texttt{better}} \vdash$ a:Account ∧ a.balance=bal ⇒ Enc_e(a) — by Enc$_e$-Obj

balanceEnc:
$\texttt{Mod}_{\texttt{better}} \vdash$ a:Account ∧ a.balance=bal ⇒ Enc_e(a.balance) — by aEnc and Enc-Field

balEnc:
$\texttt{Mod}_{\texttt{better}} \vdash$ a:Account ∧ a.balance=bal ⇒ Enc_e(bal) — by Enc$_e$-Int

$\texttt{Mod}_{\texttt{better}} \vdash$ a:Account ∧ a.balance=bal ⇒ Enc(a.balance=bal) — by balanceEnc, balEnc, Enc-Eq, and Enc-=

aEnc and **balanceEnc** state that a is an internal object, and thus only internal computation may write to its field balance. **balEnc** states that since bal is an integer, it's value is constant and may not change, and thus we can trivially say that if its value ever changed (which it never will), that must have happened internally. These combine to prove that the assertion a.balance = bal is an encapsulated one, and thus requires internal computation to invalidate. Note: it may seem odd to say that the variable bal never changes, but remember bal refers to a specific integer value at a point in time, and the adaptation operator (§3.3.2) allows us to recall that value even after potential rewrites.

We similarly prove that a.password may only be changed via internal computation (**PwdEncaps**), and inside(a.password) may only be invalidated by internal computation (**PwdInsideEncaps**). That is, if only internal objects have access to an account's password, then only internal computation may grant access to an external object.

PwdEncaps:
$\texttt{Mod}_{\texttt{better}} \vdash$ a:Account ⇒ Enc(a.password=p) — by Enc$_e$-Obj, Enc-Field, and Enc-Eq

PwdInsideEncaps:
$\texttt{Mod}_{\texttt{better}} \vdash$ a:Account ⇒ Enc(inside(a.balance)) — by Enc-Inside

5.2 Part 2: Per-Method *Necessity* Specifications

Part 2 proves necessary preconditions for each method in the module interface. We employ the crucial observation that we can build necessary pre-conditions on top of classical Hoare logic (§??). **SetBalChange** uses classical Hoare logic to prove that the set method in Account never

modifies the `balance`. We then use IF1-CLASSICAL and *Necessity* logic to prove that if it ever did change (a logical absurdity), then `transfer` must have been called.

SetBalChange:

```
{a, a':Account ∧ a'.balance=bal}
  a.set(_, _)
  {a'.balance = bal}
```
by classical spec.

```
{a, a':Account ∧ a'.balance = bal ∧ ¬ false}
  a.set(_, _)
  {¬ a'.balance = bal < bal}
```
by classical Hoare logic

```
from a, a':Account ∧ a'.balance=bal ∧ ⟨_ calls a.set(_, _)⟩
  next a'.balance < bal   onlyIf false
```
by IF1-CLASSICAL

```
from a, a':Account ∧ a'.balance=bal ∧ ⟨_ calls a.set(_, _)⟩
  next a'.balance < bal   onlyIf ⟨_ calls a'.transfer(_, a'.password)⟩
```
by ABSURD and IF1-⟶

`SetPwdLeak` demonstrates how we employ classical Hoare logic to prove that a method does not leak access to some data (in this case the `password`). Using IF1-INSIDE, we reason that since the return value of `set` is `void`, and `set` is prohibited from making external method calls, no call to `set` can result in an object (external or otherwise) gaining access to the `password`.

SetPwdLeak:

```
{a:Account ∧ a':Account ∧ a.password == pwd}
  res=a'.set(_, _)
  {res != pwd}
```
by classical spec.

```
{a:Account ∧ a':Account ∧ a.password == pwd ∧ ¬ false}
  res=a'.set(_, _)
  {res != pwd}
```
by classical Hoare logic

```
from inside(pwd) ∧ a, a':Account ∧ a.password=pwd ∧ ⟨_ calls a'.set(_,_)⟩
  next ¬inside(_)   onlyIf false
```
by IF1-INSIDE

In the same manner as **`SetBalChange`** and **`SetPwdLeak`**, we also prove **`SetPwdChange`**, **`TransferBalChange`**, **`TransferPwdLeak`**, and **`TransferPwdChange`**. We provide their statements below, but omit their proofs.

SetPwdChange:

```
from a, a':Account ∧ a'.password=p ∧ ⟨_ calls a.set(_, _)⟩
  next ¬ a.password = p   onlyIf ⟨_ calls a'.set(a'.password, _)⟩
```
by IF1-CLASSICAL

TransferBalChange:

```
from a, a':Account ∧ a'.balance=bal ∧ ⟨_ calls a.transfer(_, _)⟩
  next a'.balance < bal   onlyIf ⟨_ calls a'.transfer(_, a'.password)⟩
```
by IF1-CLASSICAL

TransferPwdLeak:

```
from inside(pwd) ∧ a, a':Account ∧ a.password=pwd ∧ ⟨_ calls a'.transfer(_,_)⟩
  next ¬inside(_)   onlyIf false
```
by IF1-INSIDE

TransferPwdChange:

```
from a, a':Account ∧ a'.password=p ∧ ⟨_ calls a.transfer(_, _)⟩
  next ¬ a.password = p   onlyIf ⟨_ calls a'.set(a'.password, _)⟩
```
by IF1-CLASSICAL

5.3 Part 3: Per-Step *Necessity* Specifications

Part 3 builds upon the proofs of Parts 1 and 2 to construct proofs of necessary preconditions, not for single method execution, but any single execution step. That is, a proof that for *any* single step

in program execution, certain changes in program state require specific pre-conditions.

BalanceChange:

```
from a:Account ∧ a.balance=bal
  next a.balance < bal   onlyIf ⟨_ calls a.transfer(_, a.password)⟩
```

by **BalEncaps**, **SetBalChange**, **TransferBalChange**, and IF1-INTERNAL

PasswordChange:

```
from a:Account ∧ a.password=p
  next ¬ a.password = bal   onlyIf ⟨_ calls a.set(a.password, _)⟩
```

by **PwdEncaps**, **SetPwdChange**, **TransferPwdChange**, and IF1-INTERNAL

PasswordLeak:

```
from a:Account ∧ a.password=p ∧ inside(p)
  next ¬ inside(p)   onlyIf false
```

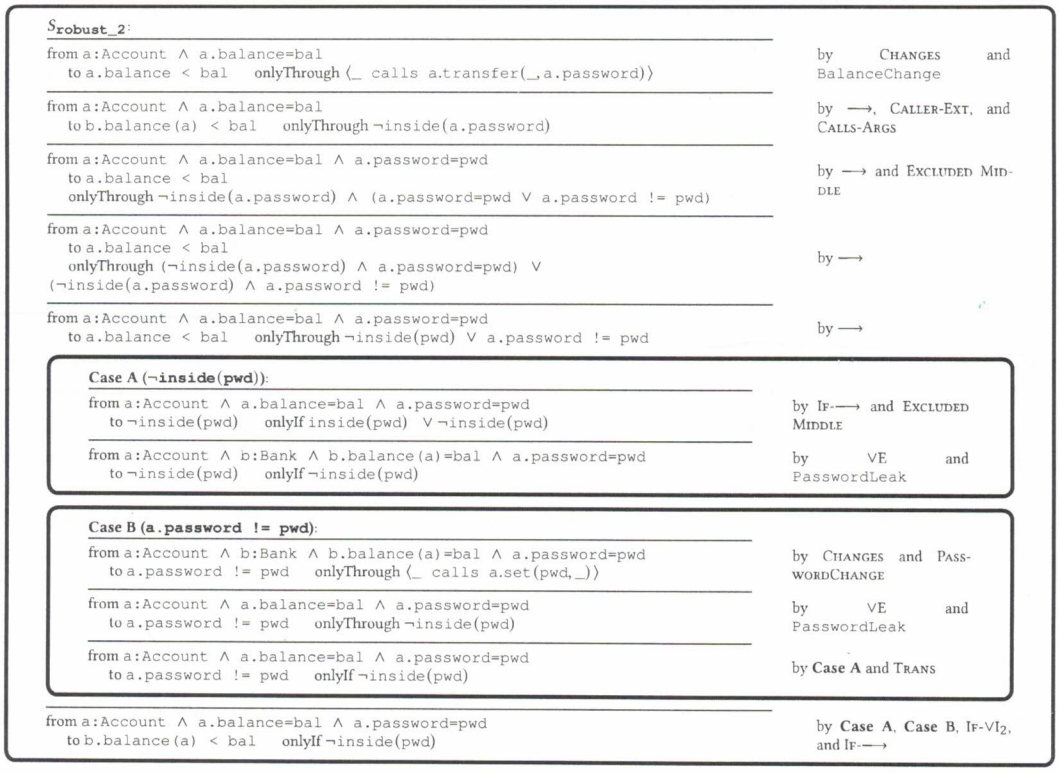
by **PwdInsideEncaps**, **SetPwdLeak**, TransferPwdLeak, and IF1-INTERNAL

(If you make these figures/floats you won't lose all this space!)

5.4 Part 4: Emergent *Necessity* Specifications

Part 4 raises necessary pre-conditions for single execution steps proven in Part 3 to the level of an arbitrary number of execution steps in order to prove specifications of emergent behaviour. The proof of $S_{\texttt{robust\_2}}$ takes the following form:

(1) If the balance of an account decreases, then by `BalanceChange` there must have been a call to `transfer` in `Account` with the correct password.
(2) If there was a call where the `Account`'s password was used, then there must have been an intermediate program state when some external object had access to the password.
(3) Either that password was the same password as in the starting program state, or it was different:
  (Case A) If it is the same as the initial password, then since by `PasswordLeak` it is impossible to leak the password, it follows that some external object must have had access to the password initially.
  (Case B) If the password is different from the initial password, then there must have been an intermediate program state when it changed. By `PasswordChange` we know that this must have occurred by a call to `set` with the correct password. Thus, there must be a some intermediate program state where the initial password is known. From here we proceed by the same reasoning as **(Case A)**.



6 RELATED WORK

Program specification and verification has a long and proud history [Hatcliff et al. 2012; Hoare 1969; Leavens et al. 2007a; Leino 2010; Leino and Schulte 2007; Pearce and Groves 2015; Summers and Drossopoulou 2010]. These verification techniques assume a closed system, where modules can be trusted to coöperate — Design by Contract [Meyer 1992] explicitly rejects *"defensive programming"* with an "absolute rule" that calling a method in violation of its precondition is always a bug.

Unfortunately, open systems, by definition, must interact with untrusted code: they cannot rely on callers obeying method preconditions. [Miller 2006; Miller et al. 2013] define 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."* [Murray 2010] made the first attempt to formalise defensive consistency and correctness in a programming language context. Murray's model was rooted in counterfactual causation [Lewis 1973]: 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 abstractly, without a specification language for describing effects. Both Miller and Murray's definitions are intensional, describing what it means for an object to be defensively consistent, rather than how defensive consistency can be achieved.

The security community has developed a similar notion of "robust safety" that originated in type systems for process calculi, ensuring protocols behave correctly in the presence of "an arbitrary hostile opponent" [Bugliesi et al. 2011; Gordon and Jeffrey 2001]. More recent work has applied robust safety in the context of programing languages. For example, [Swasey et al. 2017] present a logic for object capability patterns, drawing on verification techniques for security and information

flow. They prove a robust safety property that ensures interface objects ("low values") will never leak internal implementations ("high values") to arbitrary attackers. Simiarly, [Schaefer et al. 2018] have added support for information-flow security using refinement to ensure correctness (in this case confidentiality) by construction.

[Devriese et al. 2016] have deployed powerful theoretical techniques to address similar problems to *Necessity*. 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. They have demonstrated solutions to a range of exemplar problems, including the DOM wrapper (replicated in §3.4.1) and a mashup application. Their distinction between public and private transitions is similar to our distinction between internal and external objects.

Necessity 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. They use powerful mathematical techniques which the users need to understand in order to write their specifications, while *Necessity* users only need to understand first order logic. Finally, none of these systems offer the kinds of necessity assertions addressing control flow, provenance, and permission that are at the core of *Necessity*'s approach.

By enforcing encapsulation, all these approaches are reminiscent of techniques such as ownership types [Clarke et al. 1998; Noble et al. 1998], which also can protect internal implementation objects behind encapsulation boundaries. [Banerjee and Naumann 2005a,b] demonstrated that by ensuring confinement, ownership systems can enforce representation independence. *Necessity* relies on an implicit form of ownership types [Vitek and Bokowski 1999], where inside objects are encapsulated behind a boundary consisting of all the internal objects that are accessible outside their defining module [Noble et al. 2003]. Compare *Necessity*'s definition of inside — all references to o are from objects x that are within M (here internal to M): $\forall x.[\langle x \texttt{ access } o\rangle \Rightarrow \langle x \texttt{ internal}\rangle]$ with the containment invariant from Clarke et al. [2001] — all references to o are from objects x whose representation is within ($<:$) o's owner: $(\forall x.[\langle x \texttt{ access } o\rangle \Rightarrow \texttt{rep}(x) <: \texttt{owner}(o)])$.

Necessity specifications embody a similar encapsulation relation in program state space, *e.g.* reasoning from an external condition A_1 only through the (necessary) boundary condition A to reach the final condition A_2 in the external states semantics.

The recent VerX tool is able to verify a range of specifications for Solidity contracts automatically [Permenev et al. 2020]. VerX includes temporal operators, predicates that model the current invocation on a contract (similar to *Necessity*'s "calls"), access to variables, and sums can be computed over collections, but has no analogues to *Necessity*'s permission or provenance assertions. Unlike *Necessity*, VerX includes a practical tool that has been used to verify a hundred properties across case studies of twelve Solidity contracts. Also unlike *Necessity*, VerX's own correctness has not been formalised or mechanistically proved.

O'Hearn and Raad et al. developed Incorrectness logics to reason about the presence of bugs, based on a Reverse Hoare Logic [de Vries and Koutavas 2011]. Classical Hoare triples $\{P\}\, C\, \{Q\}$ express that starting at states satisfying P and executing C is sufficient to reach only states that satisfy Q (soundness), while incorrectness triples $[P_i]\, C_i\, [Q_i]$ express that starting at states satisfying P_i and executing C_i is sufficient to reach all states that satisfy Q_i and possibly some more (completeness). From our perspective, classical Hoare logics and Incorrectness logics are both about sufficiency, whereas here we are concerned with *Necessity*. Combining both approaches into a "necessity incorrectness logic" must necessarily (even if incorrectly) be left for future work.

In early work, [Drossopoulou and Noble 2014] sketched a specification language to specify six correctness policies from [Miller 2006]. They also sketched how a trust-sensitive example (escrow) could be verified in an open world [Drossopoulou et al. 2015]. More recently, [Drossopoulou et al. 2020b] presents the *Chainmail* language for "holistic specifications" in open world systems.

Like *Necessity*, *Chainmail* is able to express specifications of *permission*, *provenance*, and *control*; *Chainmail* also includes *spatial* assertions and a richer set of temporal operators, but no proof system. *Necessity*'s restrictions mean we can provide the proof system that *Chainmail* lacks.

In practical open systems, especially web browsers, defensive consistency / robust safety is typically supported by sandboxing: dynamically separating trusted and untrusted code, rather than relying on static verification and proof. Google's Caja [Miller et al. 2008], for example, uses proxies and wrappers to sandbox web pages. Sandboxing has been validated formally: [Maffeis et al. 2010] develop a model of JavaScript and show it prevents trusted dependencies on untrusted code. [Dimoulas et al. 2014] use dynamic monitoring from function contracts to control objects flowing around programs; [Moore et al. 2016] extends this to use fluid environments to bind callers to contracts. [Sammler et al. 2019] develop $\lambda_{sandbox}$, a low-level language with built in sandboxing, separating trusted and untrusted memory. $\lambda_{sandbox}$ features a type system, and Sammler et al. show that sandboxing achieves robust safety. Sammler et al. address a somewhat different problem domain than *Necessity* does, low-level systems programming where there is a possibility of forging references to locations in memory. Such a domain would subvert *Necessity*, and introduce several instances of unsoundness, in particular `inside(`x`)` would not require interaction with internal code in order to gain access to x, as a reference to x could always be guessed.

7 CONCLUSION

This paper presents *Necessity*, a specification language for a program's emergent behaviour. *Necessity* specifications constrain when effects can happen in some future state ("`onlyIf`"), in the immediately following state ("`next`"), or on an execution path ("`onlyThrough`").

We have developed a proof system to prove that modules meet their specifications. Our proof system exploits the pre and postconditions of classical method specifications to infer per method *Necessity* specifications, generalises those to cover any single execution step, and then combines them to capture a program's emergent behaviour. Deriving per method *Necessity* specifications from classical specifications has two advantages. First, we did not need to develop a special purpose logic for that task. Second, modules that have simillar classical specifications can be proven to satisfy the same Necessity Specifications using the *same* proof. We have proved our system sound, and used it to prove a bank account example correct: the Coq mechanisation is detailed in the appendices and available as an artifact.

In future work we want to expand a Hoare logic so as to make use of *Necessity* specifications, and reason about calls into unknown code - c.f. §2.2.1. We want to remove the current requirement for explicit framing, and leverage instead some of `modifies`-clauses or implicit dynamic frames [Ishtiaq and O'Hearn 2001; Leavens et al. 2007b; Leino 2013; Parkinson and Summers 2011; Smans et al. 2012]. We want to work on supporting callbacks. We want to develop a logic for encapsulation rather than rely on a type system. Finally we want to develop logics about reaoning about risk and trust [Drossopoulou et al. 2015].

REFERENCES

Tzanis Anevlavis, Matthew Philippe, Daniel Neider, and Paulo Tabuada. 2022. Being Correct Is Not Enough: Efficient Verification Using Robust Linear Temporal Logic. *ACM Trans. Comp. Log.* 23, 2 (2022), 8:1–8:39.

Anindya Banerjee and David A. Naumann. 2005a. Ownership Confinement Ensures Representation Independence for Object-oriented Programs. *J. ACM* 52, 6 (Nov. 2005), 894–960. https://doi.org/10.1145/1101821.1101824

Anindya Banerjee and David A. Naumann. 2005b. State Based Ownership, Reentrance, and Encapsulation. In *ECOOP (LNCS, Vol. 3586)*, Andrew Black (Ed.).

Lars Birkedal, Thomas Dinsdale-Young., Armeal Gueneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzeverlekos. 2021. Theorems for Free from Separation Logic Specifications. In *ICFP*.