Necessity Specifications are Necessary for Robustness

ANONYMOUS AUTHOR(S)

Traditional specifications provide sufficient conditions for closed programs to be correct — if a function is invoked with valid preconditions, it will meet its postconditions upon return: calling a function will make good things happen. Unfortunately, sufficient conditions are insufficient for reasoning about programs in an open world — frameworks that can be extended, systems that interact with third parties, programs subject to unintentional or malicious attacks. In an open world, *Necessity* specifications are necessary: programmers need to reason about necessary conditions to prove that bad things can never happen. *Necessity* logic enables programmers to assert and to reason about the necessary conditions for their programs to be correct, and to infer the necessary conditions that their programs do (or do not) support. Using this logic, programmers can prove that bad things do not happen, defending programs against an unfriendly open world.

1 INTRODUCTION

Today's software has been built over decades by combining modules and components of different provenance and trustworthiness. It is open, interacting with other programs, devices, and people. In this setting, modules need to be both useful ("good things will eventually happen") and robust ("bad things will never happen") – to paraphrase Lamport [1977] on liveness and safety.

Good things happening are typically specified formally through Hoare [1969] triples consisting of a precondition, a code snippet, and some postcondition. For example,

(S1) Calling `transfer` using the correct password will transfer the money.

The precondition is a *sufficient* condition for the code snippet to make the good thing happen: assuming termination, 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).

Bad things not happening irrespective of whether a module's clients are benign or malicious, is what makes a module robust. For example, for a bank module to be robust, it must guarantee that

(S2) An account's balance does not decrease unless `transfer` was called with the correct password.

Calling `transfer` with the correct password is a *necessary condition* for reducing the account's balance. Condition (S2) is crucial in order to confidently pass the bank account into untrusted code, in the expectation of receiving some payment, but without fear that a malicious client might steal my money [Miller et al. 2013, 2000]. In contrast to "good things happening" which are usually specified with Hoare triples, a multitude of different guarantees have been proposed for "bad things not happening", differing in the level of granularity, target language or calculi, and intended use.

Information-flow control systems have been developed for various calculi and programming languages. Their guarantees are coarse-grained: the contents of high security variables cannot be affected by the values of low security variables [Murray et al. 2013; Zdancewic and Myers 2001].

Correspondence assertions are more fine-grained. Proposed for process calculi, they express that if one principal ever reaches a certain point in the protocol, then some other principal has previously reached some other matching point in the protocol; the term *robust safety* expresses that correspondence assertions are true in the presence of any opponent [Gordon and Jeffrey 2001b]. A related approach, *authorisation policies*, promises that certain actions will not be taken on certain objects unless corresponding rights had been granted [Fournet et al. 2007].

In *object capabilities* [Miller 2006], effects can only be produced by sending messages to objects (which have unforgeable identities) and there is no ambient authority. In the context of object

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

capabilities, Swasey et al. [2017] define *robust safety for Javascript* to mean that the untrusted environment of a program cannot violate its internal invariants, and developed a verification methodology to prove that programs that export only wrapped values are robustly safe. Devriese et al. [2016] develop Kripke models to reason about the result of execution of some given code snippet in parallel with arbitrary, unknown code and Birkedal et al. [2021] derive usage protocols from a module's separation logic specification.

Most of the approaches outlined so far were not developed with the aim of expressing guarantees such as (S2). Information flow control systems target process calculi. Swasey et al. are concerned with the preservation of internal invariants. Devriese et al. are concerned with the *effects* of known code, while Birkedal et al. reverse-engineer usage protocols.

VerX [Permenev et al. 2020] and *Chainmail* [Drossopoulou et al. 2020b] added temporal operators to specification languages, and thus can express necessary conditions. Assertions support the usual logical connectives and can refer to current state and function calls. *Chainmail* can also talk about *permission* (whether an object has direct access to another object – either through its fields or its arguments to a function call), about *provenance* (whether an object belongs to the current module, or some unknown module), and control (whether a certain function is called with given arguments). Permission and provenance are inspired by the object capabilities literature [Miller 2006].

However, while guarantee (S2) is necessary to confidently pass an account into untrusted code, it is *not* sufficient. (S2) does not take account of the module's *emergent behaviour*. What if the module leaks the password? What we really need is

> (S3) The balance of an account does not decrease ever in the future unless some external object has access to the account's current password.

With (S3), 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 steal my money [Miller 2011; Miller et al. 2013]. Note that (S3) does not mention the names of any functions in the module, and thus can be expressed without reference to any particular API — indeed (S3) can constrain *any* API with an account, an account balance, and a password.

Both VerX and *Chainmail* can express (S2) but VerX cannot express (S3), because (S3) requires the concepts of provenance ("external object") and permission ("has access to"x). On the other hand, VerX can prove adherence to specifications through symbolic execution, while *Chainmail* lacks a proof system. Also correspondence assertions cannot express (S3): they lack support for provenance, and ways to match the time of an effect to that of the condition.

This work introduces *Necessity*, which consists of a specification language for expressing a module's safety guarantees such as (S3), and a logic to prove adherence to such specifications. We adopted *Chainmail*'s capability operators, but we replaced its temporal operators with our novel *necessity* operators. These simplifications enabled us to develop our proof logic. The current work is based on a simple, imperative, typed, object oriented language with unforgeable addresses and private fields.[1]

Necessity supports the usual assertions, A, (including permission, provenance, and control), as well as three necessity operators. We now show one of these operators

```
from $A_{curr}$ to $A_{fut}$ onlyIf $A_{nec}$
```

The form above says that a transition from a current state satisfying A_{curr} to a future state satisfying A_{fut} is possible only if the necessary condition A_{nec} holds in the *current* state. The other two operators express either *necessary conditions* for *single-step* transitions or *intermediate* conditions

[1]We believe that our approach 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].

for *any* transitions. Unlike *Chainmail*, the necessity operators are second class, and may not appear in the assertions A. Being second class allows us to separate proofs about necessity from other proofs. We formalize (S3) in §2.2.

Necessity logic is based on four main ideas: First, some assertions are *encapsulated*, *i.e.* their validity may only be affected by internal calls. Second, by leveraging the classical specifications of methods (*i.e.* the sufficient conditions) we can obtain per-method-call *Necessity* specifications; that is, necessary preconditions for a given effect and a given method call (at a very abstract level, this is similar to Birkedal et al. [2021]). Third we infer a per-step-condition, *i.e.*, a necessary condition given an effect and a single, unspecified step. This step could be an internal call, or any kind of external step. Fourth, a novel proof system allows us to combine per-method *Necessity* specifications and encapsulated assertions to obtain per-module *Necessity* specifications; these describe a module's emergent behaviour. More in §2.4.

1.1 Paper Organization and Contributions

The contributions of this paper are:

(1) A language to express *Necessity* specifications (§3) that is expressive enough to encode the *Chainmail* examples from Drossopoulou et al. [2020b].
(2) A logic for proving a module's adherence to *Necessity* specifications (§4), and a proof of soundness of the logic, (§4.5), both mechanized in Coq.
(3) A proof in our logic that our bank module obeys its *Necessity* specification (§5), mechanized in Coq.

Our formalisation of *Necessity* has a number of limitations: it is parametric with respect to assertion satisfaction, encapsulation, and the type system; we forbid "callbacks" out to external objects; and classical specifications require explicit framing. We will discuss these limitations further as 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 `Mod1` consists of the `Account` class with a `transfer` method, and an empty `Password` class where each instance models a unique password.

```
module Mod1
  class Account
    field balance:int
    field pwd: Password
    method transfer(dest:Account, pwd':Object) -> void
      if this.pwd==pwd'
        this.balance-=100
        dest.balance+=100
  class Password
```

We can capture the intended semantics of `transfer` through a pre and postcondition JML-style [Leavens et al. 2007a] "classical" specification. `Mod1`'s implementation of the `transfer` method meets this specification.

```
ClassicBankSpec  ≜
   method transfer(dest:Account, pwd':Password) -> void {
       ( PRE:  this.balance=bal1 ∧ this.pwd==pwd' ∧ dest.balance=bal2 ∧ dest=/=this
         POST: this.balance == bal1-100 ∧  dest.balance == bal2+100 )
       ( PRE: this.balance=bal1 ∧ this.pwd=/=pwd' ∧ dest.balance=bal2
         POST: this.balance == bal1 ∧  dest.balance=bal2 )
       ( PRE: a : Account ∧ a=/=this ∧ a=/=dest  ∧ a.balance=bal ∧ a.pwd=pwd1
      POST:  a.balance=bal ∧ a.pwd=pwd1)
    ( PRE: a : Account ∧ a.pwd=pwd1
      POST: a.pwd=pwd1)
```

Now consider the following alternative implementations: `Mod2` allows any client to reset an account's password at any time, while `Mod3` requires the existing password in order to change it.

```
module Mod2
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...
    method set(pwd': Object)
      this.pwd=pwd'

  class Password
```

```
module Mod3
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...
    method set(pwd',pwd'': Object)
      if (this.pwd==pwd')
        this.pwd=pwd''
  class Password
```

Although the `transfer` method is the same in all three alternatives, and each one satisfies `(ClassicBankSpec)`, code such as `account.set(42); account.transfer(yours, 42)` is enough to drain the account in `Mod2` without knowing the password.

2.2 Bank Account – the right specification

We need to rule out `Mod2` while permitting `Mod1` and `Mod3`. The catch is that the vulnerability present in `Mod2` is the result of *emergent* behaviour from the interactions of the `set` and `transfer` methods — even though `Mod3` 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:

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

The critical point of this *Necessity* specification 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. This is why we say *Necessity* specifications capture a module's *emergent behaviour.*

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 give a sketch of the novel concepts we developed in *Necessity* logic. For illustration, we outline a proof that `Mod3` adheres to `NecessityBankSpec`.

Part 1: Assertion Encapsulation. 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: here we rely on rudimentary annotations inspired by confinement types [Vitek and Bokowski 1999].

For our proof, we establish that **(a)** the balance is encapsulated by `Mod3`, and **(b)**, external accessibility to an account's password may only be obtained through calls to `Mod3` – that is, the property that no external object has access to the password is encapsulated by `Mod3`.

Part 2: Per-method conditions are necessary conditions for given effect and given single, specified, method call. To infer these, we leverage sufficient conditions from classical specifications: If the negation of a method's classical postcondition implies a given effect, then the negation of the classical precondition is a necessary precondition for the effect and the method call. More in §4.2.

For our proof, we establish for each method of `Mod3` (*i.e.* each method in `Account` and `Password`) that if the method is called, then **(c)**, if it causes the balance to reduce, then the method called was `transfer` and the correct password was provided, and **(d)** it will not provide external accessibility to the password.

Part 3: Single-step conditions are necessary conditions for a given effect and a single, *unspecified* step. This step could be an internal call, or any kind of external step. For effects encapsulated by M, we can infer such single-step conditions by combining the per-method conditions for that effect from all methods in M. More in §4.3.

For our proof, from **(a)** and **(c)** we obtain that **(e)** if the balance were to reduce in *any single* step – whether an internal call, or any external step – the method called was `transfer` and the correct password was provided. Similarly, from **(b)** and **(d)** we obtain that **(f)** external accessibility to the password will not be provided by any single step.

Part 4: Emergent behaviour is the behaviour than can be observed by any number of steps. We infer the *emergent* behaviour out of the conditions of single steps. This part is crucial; remember that while `Mod2` satisfies (S3) for one single step, it does not satisfy it for any number of steps. More in §4.4.

For our proof, from **(e)** we obtain that **(g)** if the balance reduces in any number of steps, then at some step `transfer` was called with the correct password. From **(g)** we obtain that **(h)** if the balance reduces in any number of steps, then at some intermediate step some external object had access to the password. From **(f)** we obtain that **(i)** external accessibility to the password will not be provided by any sequence of steps. Using **(h)** and **(i)** we establish that `Mod3` indeed adheres to `NecessityBankSpec`.

Note that 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.

3 THE MEANING OF NECESSITY

In this section we define our *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 an unsurprising, imperative, sequential, class based, typed, object-oriented language. `TooL` is straightforward: Appendix A contains the full definitions. `TooL` is based on $\mathscr{L}_{oo}$ [Drossopoulou et al. 2020b], with some small variations, as well as the addition of a a simple type system – more in 4.1.3. 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 σ_0,...σ_n, such that

- σ=σ_1, and σ'=σ_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, while 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.

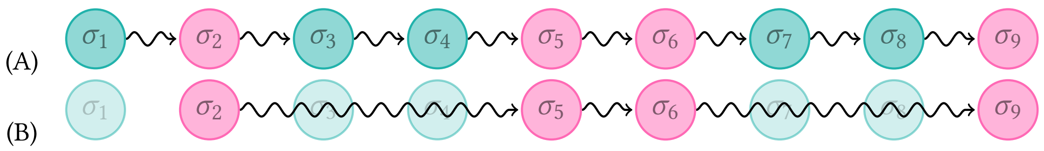

Fig. 1. External States Semantics (Def. 3.1). (A) $M' \circ M, \sigma_1 \rightsquigarrow \ldots \rightsquigarrow \sigma_9$ (B) $M'; M,\ \sigma_2 \rightsquigarrow \ldots \rightsquigarrow \sigma_9$

Fig. 1 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 not concerned with either internal states nor states that cannot ever arise. *Arising* states are those that 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).

3.2 *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:

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

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 ToolL 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.9).

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 σ.`stack`, σ.`contn`, σ.`heap` mean the stack, the continuation at the top frame of σ, and the heap of σ. The term $\alpha \in \sigma$.`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) $M, \sigma \models e$ iff $M, \sigma, e \hookrightarrow \texttt{true}$
(2) $M, \sigma \models e : C$ iff $M, \sigma, e \hookrightarrow \alpha$ *and* $\mathit{classOf}(\sigma, \alpha) = C$
(3) $M, \sigma \models \neg A$ iff $M, \sigma \not\models A$
(4) $M, \sigma \models A_1 \wedge A_2$ iff $M, \sigma \models A_1$ and $M, \sigma \models A_2$
(5) $M, \sigma \models A_1 \vee A_2$ iff $M, \sigma \models A_1$ or $M, \sigma \models A_2$
(6) $M, \sigma \models \forall x.[A]$ iff $M, \sigma[x \mapsto \alpha] \models A,$ for some x fresh in σ, and for all $\alpha \in \sigma.\texttt{heap}$.
(7) $M, \sigma \models \exists x.[A]$ iff $M, \sigma[x \mapsto \alpha] \models A,$ for some x fresh in σ, and for some $\alpha \in \sigma.\texttt{heap}$.
(8) $M, \sigma \models \langle x\ \texttt{access}\ y \rangle$ iff
  (a) $\lfloor x.f \rfloor_\sigma = \lfloor y \rfloor_\sigma$ for some f,
  or
  (b) $\lfloor x \rfloor_\sigma = \lfloor \texttt{this} \rfloor_\phi$, and $\lfloor y \rfloor_\sigma = \lfloor z \rfloor_\phi$ for some variable z, and some frame ϕ in $\sigma.\texttt{stack}$.
(9) $M, \sigma \models \langle x\ \texttt{internal} \rangle$ iff $\mathit{classOf}(\sigma, x) \in M$
(10) $M, \sigma \models \langle x\ \texttt{external} \rangle$ iff $\mathit{classOf}(\sigma, x) \notin M$
(11) $M, \sigma \models \langle x\ \texttt{calls}\ y.m(z_1, \ldots, z_n) \rangle$ iff
  (a) $\sigma.\texttt{contn} = (w := y'.m(z'_1, \ldots, z'_n); s),$ for some variable w, and some statement s,
  (b) $M, \sigma \models x = \texttt{this}$ and $M, \sigma \models y = y'$,
  (c) $M, \sigma \models z_i = z'_i$ for all $1 \leq i \leq n$

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). 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, ... z_n.[2] 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 $M \models A$ if and only if for all modules M', and all σ, if $\mathit{Arising}(M', M, \sigma)$, then $M, \sigma \models A$.

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 see Theorem 4.4). We define soundness of such a judgment in the usual way:

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

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). $\texttt{inside}(o) \triangleq \forall x.[\langle x\ \texttt{access}\ o \rangle \implies \langle x\ \texttt{internal} \rangle]$

`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* Specifications

Our *Necessity* specification language extends *Assert* with novel *necessity operators*. In this section we define its syntax (Definition 3.8) and semantics (Definition 3.10). We have the following three operators:

[2] It does *not* mean that somewhere in the call stack there exists a call from x to $y.m(\ldots)$.

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 `onlyThough` 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* or the *final* state.

Definition 3.8 (Necessity Syntax). The syntax of *Necessity* Specifications (S) is as follows:

$$\begin{aligned} S \quad ::= \quad & A \mid \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A_3 \mid \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A_3 \\ & \mid \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3 \end{aligned}$$

Relationship between Necessity Operators. The three *Necessity* operators defined in Def. 3.8 can be related by generality. An *Only If* (`from` A_1 `to` A_2 `onlyIf` A) implies a *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.

3.3.1 *Adaptation: viewing the future through the lens of the past.*

Consider a specification that says for an account's balance to go down from 100 to 50, a `transfer` method must be called on that account:

```
from a:Account ∧ a.balance == 100
    to a.balance == 50
    onlyThrough ⟨_ calls a.transfer(_,_)⟩
```

This specifications refers to three different program states: at the start when the balance is 100; and the end when it's 50; and somewhere between when `transfer` is called. For this specification to make sense, the bindings of variables like `a` must refer to the same object in each state, in particular, bindings over the future states must refer to the same object as in the past state. (Because TooL's objects' classes, types, or fields never change and are never deleted from the heap, we can at least be sure any object that exists in some state will still exist in all arising future states).

We deal with this via an *adaptation* operator [Drossopoulou et al. 2020b]. We write $\sigma' \triangleleft \sigma$ to view a future state σ' from the perspective of a current (or past) state σ. Def.3.9 shows how $\sigma' \triangleleft \sigma$ constructs a new state, taking the heap and most of the stack from the future state σ'. We replace the top frame's variable map with the variable map from the top frame of the past state σ, avoiding name clashes by renaming the variables in the top frame of σ' with fresh variables ($\overline{v}$) and renaming free variables in the continuation similarly.

Definition 3.9. $\sigma' \triangleleft \sigma \triangleq (\chi', \{\texttt{local} := \beta[\overline{v} \mapsto \beta'(\overline{z'})], \texttt{contn} := [\overline{z'}/\overline{v}]c'\} : \psi)$ where

- $\sigma = (\_, \{\texttt{local} := \beta; \texttt{contn} := \_\} : \_)$, and $\sigma' = (\chi', \{\texttt{local} := \beta', \texttt{contn} := c'\} : \psi)$
- $dom(\beta') = \overline{z'}$, $dom(\beta) \cap \overline{v} = \emptyset$, and $|\overline{z'}| = |\overline{v}|$

3.3.2 *Semantics of Necessity Specifications.*

We now define semantics of specifications, $M \vDash S$, by cases over the four possible syntactic forms:

Definition 3.10 (Necessity Semantics). For any assertions A_1, A_2, and A, we define

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

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

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

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

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

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

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

3.4 More Examples Demonstrating the Expressiveness of *Necessity*

We now consider some more specifications, and discuss whether given modules satisfy them.

3.4.1 More examples of the Bank. Looking back at the example from the Section 2, it holds that

$$\texttt{Mod2} \not\models \texttt{NecessityBankSpec} \qquad\qquad \texttt{Mod3} \models \texttt{NecessityBankSpec}$$

Consider now another four *Necessity* specifications:

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

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

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

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

The specification $\texttt{NecessityBankSpec}_a$ states that the balance of an account decreases *in one step*, only if an external object has access to the password. It is similar to `NecessityBankSpec`, with the difference that the decrease takes place in *one* step, rather than in *a number* of steps. Even though `Mod2` does not satisfy `NecessityBankSpec`, it does satisfy $\texttt{NecessityBankSpec}_a$.

$$\texttt{Mod2} \models \texttt{NecessityBankSpec}_a \qquad\qquad \texttt{Mod3} \models \texttt{NecessityBankSpec}_a$$

The specifications $\texttt{NecessityBankSpec}_b$ and $\texttt{NecessityBankSpec}_c$ 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. $\texttt{NecessityBankSpec}_d$ says that such a decrease is only possible if some *intermediate* step called `transfer`. All three modules satisfy $\texttt{NecessityBankSpec}_b$ and $\texttt{NecessityBankSpec}_d$, and none satisfy $\texttt{NecessityBankSpec}_c$. That is:

$$\begin{array}{ll} \texttt{Mod2} \models \texttt{NecessityBankSpec}_b & \qquad \texttt{Mod3} \models \texttt{NecessityBankSpec}_b \\ \texttt{Mod2} \models \texttt{NecessityBankSpec}_d & \qquad \texttt{Mod3} \models \texttt{NecessityBankSpec}_d \\ \texttt{Mod2} \not\models \texttt{NecessityBankSpec}_c & \qquad \texttt{Mod3} \not\models \texttt{NecessityBankSpec}_c \end{array}$$

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 `Node`s 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. As stated earlier, *Necessity* is less expressive than *Chainmail.* Nevertheless, we believe that it is powerful enough for the purpose of straightforwardly expressing robustness requirements. 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 Section 3.3. Proving a specification requires four steps:

(1) Proving Assertion Encapsulation (§4.1)
(2) Proving *Necessity* specifications from classical specifications for a single internal method (§4.2)
(3) Proving module-wide Single-Step *Necessity* specifications by combining per-method *Necessity* specifications (§4.3)
(4) Raising necessary conditions to construct proofs of emergent behaviour (§4.4)

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. *Necessity* is parametric over the details of the encapsulation model [Noble et al. 2003]: appendix B and Figure 11 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 defintion: 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 $M \vDash A \Rightarrow Enc(A')$ 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 $M \vDash A \Rightarrow Enc(A')$, if and only if for all external modules M', and all states σ, σ' such that $Arising(M, M', \sigma)$:

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

This definition again uses adaptation $\sigma' \triangleleft \sigma$ because we have to interpret one assertion in two different states. Revisiting the examples from Section 2, we can see both `Mod2` and `Mod3` encapsulate the `balance` of an account, because any change requires calling a method on an internal object.

$$\texttt{Mod2} \vDash \texttt{a} : \texttt{Account} \Rightarrow Enc(\texttt{a.balance} = \texttt{bal})$$
$$\texttt{Mod3} \vDash \texttt{a} : \texttt{Account} \Rightarrow Enc(\texttt{a.balance} = \texttt{bal})$$

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 $M \vdash A \Rightarrow Enc(A)$ is *sound*, if for all modules M, and assertions A_1 and A_2, if $M \vdash A_1 \Rightarrow Enc(A_2)$ then $M \vDash A_1 \Rightarrow Enc(A_2)$.

The key consequence of soundness is that an object inside a module ($\texttt{inside}(o)$) 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.

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 4.3. That is, classical

$$\frac{M \vdash \{x:C \wedge P_1 \wedge \neg P\}\ \texttt{res} = x.m(\overline{z})\ \{\neg P_2\}}{M \vdash \texttt{from}\ P_1 \wedge x:C \wedge \langle \_\ \texttt{calls}\ x.m(\overline{z})\rangle\ \texttt{next}\ P_2\ \texttt{onlyIf}\ P} \quad (\text{IF1-CLASSICAL})$$

$$\frac{M \vdash \{x:C \wedge \neg P\}\ \texttt{res} = x.m(\overline{z})\ \{\texttt{res} \neq y\}}{M \vdash \texttt{from}\ \texttt{inside}(y) \wedge x:C \wedge \langle \_\ \texttt{calls}\ x.m(\overline{z})\rangle\ \texttt{next}\ \neg\texttt{inside}(y)\ \texttt{onlyIf}\ P} \quad (\text{IF1-INSIDE})$$

Fig. 2. Per-Method *Necessity* specifications

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 $\texttt{res}$ in Q.

Figure 2 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 obejct 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 $.. \vdash \{R_1 \wedge R2\}\ s\ \{Q\}$ holds, then any state that satisfies R_1 and which upon execution of $\texttt{s}$ 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. This rule is sound, as we do not support calls from internal to external code (see Section 4.1.3) – in further work we plan to weaken this requirement, 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 $\texttt{inside}()$ predicate. Since $\texttt{inside}()$ is not a classical assertion, we cannot use Hoare logic to reason about necessary conditions for invalidating $\texttt{inside}()$.

4.3 Per-Step *Necessity* Specifications

We now raise per-method *Necessity* specifications to per-step *Necessity* specifications. The rules appear in Figure 3.

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.

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

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

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

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

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

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

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

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

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

Fig. 3. Single-Step *Necessity* Specifications

The rule for implication (If1-$\longrightarrow$) may strengthen properties of either the starting or ending state, or weaken the necessary precondition.

There are two disjunction introduction rules: (a) If1-$\vee$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-$\vee$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-$\vee$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 behavior. Figure 4 present some of the rules for the construction of proofs for *Only Through*, while Figure 5 provides some of the rules for the

$$\frac{M \vdash \texttt{from}\ A\ \texttt{next}\ \neg A\ \texttt{onlyIf}\ A'}{M \vdash \texttt{from}\ A\ \texttt{to}\ \neg A\ \texttt{onlyThough}\ A'}\ (\text{Changes}) \qquad \frac{\begin{array}{c}M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A_3\\ M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_3\ \texttt{onlyThough}\ A\end{array}}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A}\ (\text{Trans}_1)$$

$$\frac{\begin{array}{c}M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A_3\\ M \vdash \texttt{from}\ A_3\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A\end{array}}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A}\ (\text{Trans}_2) \qquad \frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A}\ (\text{If})$$

$$M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A_2 \quad (\text{End})$$

Fig. 4. Selected rules for *Only Through* – rest in Figure ??

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

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

Fig. 5. Selected rules for *Only If* – the rest in Figure ??

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 3.

Only Through has several notable rules. Changes, in Figure 4, 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:

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

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 2, and 3, and 5, and 4, then, for all modules M, and all Necessity specifications S:*

$$M \vdash S \qquad \textit{implies} \qquad M \models S$$

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

Theorem. 4.4 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 4.4 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` $\alpha_y\rangle \land \alpha_x == x \land \alpha_y == y$.

5 PROOF OF ADHERENCE TO NECESSITYBANKSPEC

In this section we return to the Bank Account example, providing a full proof and the accompanying Coq formalism includes a mechanized version.

As we stated in Section 3.2, we assume the existence of a proof system for judgments of the form $M \vdash A$, denoting that in any arising program state, with internal module M, A is satisfied. In this section, we make use of several rules that under such a logic should be sound. We provide a description of these rules in Appendix E.

We devote the rest of this section to the proof expressed in *Necessity* logic of a module's adherence to the Bank Account specification.

The Bank Account Module `Mod4`. In this section we define `Mod4`, a new Bank Account implementation, which differs from that of Section 2. `Mod4` is more complex than `Mod3`; this allows us to demonstrate how *Necessity* logic deals with challenges that come with more complex data structures and specifications. These challenges are

(1) Specifications defined using ghost fields – in this case `b.balance(a)` returns the balance of account `a` in `Bank` `b`.
(2) Modules with several classes and methods; they all must be considered when constructing proofs about emergent behavior.
(3) The construction of a proof of assertion encapsulation. Such a proof is necessary here because the ghost field `balance` reads several fields. We use our simple confinement system, captured by `confined` classes in TooL.

The `NecessityBankSpec` will use the ghost field, `balance`, and not simply the balance field as in `Mod1`, `Mod2`, and `Mod3`.

```
NecessityBankSpec ≜ from b:Bank ∧ b.balance(a)=bal
                     to b.balance(a) < bal   onlyIf ¬inside(a)
```

That is, if the balance of an account ever decreases, it must be true that some object external to `Mod4` has access to the password of that account.

```
module Mod4
  class Account
    field password:Object
    method authenticate(pwd:Object):bool
      {return pwd == this.password}
    method changePass(pwd:Object, newPwd:Object):void
      {if pwd == this.password
        this.password := newPwd}
  class confined Ledger
    field acc1:Account
    field bal1:int
    field acc2:Account
    field bal2:int
    ghost intrnl balance(acc):int=
      if acc == acc1
        bal1
      else if acc == acc2
        bal2
      else -1
    method transfer(amt:int, from:Account, to:Account):void
      {if from == acc1 && to == acc2
         bal1 := bal1 - amt
         bal2 := bal2 + amt
       else if from == acc2 && to == acc1
         bal1 := bal1 + amt
         bal2 := bal2 - amt}
  class Bank
    field book:Ledger
    ghost intrnrl balance(acc):int=book.balance(acc)
    method transfer(pwd:Object, amt:int, from:Account, to:Account):void
      {if (from.authenticate(pwd))
         book.transfer(amt, from, to)}
```

Fig. 6. Bank Account Module

We provide the implementation of `Mod4` in Figure 6. In `Mod4`, we move the balance of an account into a ledger that is stored within a bank. Thus, the module `Mod4` (Figure 15) consists of 3 classes: (1) `Account` that maintains a password, (2) `Bank`, a public interface for transferring money from one account to another, and (3) `Ledger`, a private class, annotated as `confined`, used to map `Account` objects to their balances.

A `Bank` consists of a `Ledger`, a method for transferring funds between accounts (`transfer`), and a ghost field, `balance` for looking up the balance of an account at a bank. A `Ledger` is a mapping from `Account`s to their balances. For brevity our implementation only includes two accounts (`acc1` and `acc2`), but it is easy to see how this could extend to a `Ledger` of arbitrary size. `Ledger` is annotated as `confined`, and as such the type system ensures our required encapsulation properties. Finally, an `Account` has some `password` object, and methods to authenticate a provided password (`authenticate`), change the password (`changePass`).

Note, Figure 6 does not provide the classical specifications of `Mod4`, which can be found in full in Appendix F. Informally, we introduce classical specifications that state that

(1) no method returns the password,
(2) the `transfer` method in `Ledger` results in a decreased balance to the `from Account`,
(3) and the `transfer` method in `Bank` results in a decreased balance to the `from Account` *only if* the correct password is supplied, and
(4) every other method in `Mod4` never modifies any balance in any `Bank`.

While both the implementation and the specification being proven have changed from that of 2, the structure of the proofs do retain broad similarities. In particular the proof in this section follows the outline of our reasoning given in Sec. 2.4: we prove (1) encapsulation of the account balance and password, (2) *per-method Necessity* specifications on all `Mod4` methods, (3) *per-step Necessity* specifications for changing the balance and password, and finally (4) the *emergent Necessity* specification `NecessityBankSpec`.

5.1 Part 1: Assertion Encapsulation

We base the soundness of our encapsulation of the type system of TooL, and use the proof rules given in Figures 10 and 11. Informally, $Enc_e(e)$ indicates that any objects inspected during the evaluation of expression e are internal. $Enc(A)$ (see Section 4) indicates that internal computation is necessary for a change in satisfaction of A. Rudimentary algorithms for proving $Enc_e()$ and $Enc()$ are given in Appendix B, and used here. We provide the proof for the encapsulation of `b.balance(a)` below

BalanceEncaps:

aEnc:
Mod4 ⊢ b, b′:Bank ∧ a:Account ∧ b.balance(a)=bal ⇒ Enc_e(a) — by Enc$_e$-Obj

bEnc:
Mod4 ⊢ b, b′:Bank ∧ a:Account ∧ b.balance(a)=bal ⇒ Enc_e(b) — by Enc$_e$-Obj

getBalEnc:
Mod4 ⊢ b, b′:Bank ∧ a:Account ∧ b.balance(a)=bal ⇒ Enc_e(b.balance(a)) — by aEnc, bEnc, and Enc$_e$-Ghost

balEnc:
Mod4 ⊢ b, b′:Bank ∧ a:Account ∧ b.balance(a)=bal ⇒ Enc_e(bal) — by Enc$_e$-Int

Mod4 ⊢ b, b′:Bank ∧ a:Account ∧ b.balance(a)=bal ⇒ Enc(b.balance(a)=bal) — by getBalEnc, balEnc, Enc-Exp

We omit the proof of Enc(`a.password=pwd`), as its construction is very similar to that of Enc(`b.balance(a)=bal`).

5.2 Part 2: Per-Method *Necessity* Specifications

We now provide proofs for necessary preconditions on a per-method basis, leveraging classical specifications. These proof steps are quite verbose, and for this reason, we only focus on proofs of `authenticate` from the `Account` class.

There are two *per-method Necessity* specifications that we need to prove of `authenticate`:

`AuthBalChange`: any change to the balance of an account may only occur if call to `transfer` on the `Bank` with the correct password is made. This may seem counter-intuitive as it is not possible to make two method calls (`authenticate` and `transfer`) at the same time, however we are able to prove this by first proving the absurdity that `authenticate` is able to modify any balance.

`AuthPwdLeak`: any call to `authenticate` may only invalidate `inside( a.password )` (for any account `a`) if `false` is first satisfied – clearly an absurdity.

`AuthBalChange`. First we use the classical specification of the `authenticate` method in `Account` to prove that a call to `authenticate` can only result in a decrease in balance in a single step if there were in fact a call to `transfer` to the `Bank`. This may seem odd at first, and impossible to prove, however we leverage the fact that we are first able to prove that `false` is a necessary condition to decreasing the balance, or in other words, it is not possible to decrease the balance by a call to `authenticate`. We then use the proof rule Absurd to prove our desired necessary condition. This proof is presented as `AuthBalChange` below.

```
AuthBalChange:
{a, a′:Account ∧ b:Bank ∧ b.balance(a′)=bal}
  a.authenticate(pwd)
  {b.balance(a′) == bal}                                              by classical spec.

{a, a′:Account ∧ b:Bank ∧ b.balance(a′)=bal ∧ ¬ false}
  a.authenticate(pwd)
  {¬ b.balance(a′) < bal}                                             by classical Hoare logic

from a, a′:Account ∧ b:Bank ∧ b.balance(a′)=bal ∧ ⟨_ calls a.authenticate(pwd)⟩
  next b.balance(a′) < bal   onlyIf false                              by If1-Classical

from                  a:Account ∧ a′:Account ∧ b:Bank ∧ b.balance(a′)=bal ∧
⟨_ calls a.authenticate(pwd)⟩
  next b.balance(a′) < bal   onlyIf ⟨_ calls b.transfer(a′.password, amt, a′, to)⟩   by Absurd and If1-⟶
```

AuthPwdLeak. The proof of AuthPwdLeak is given below, and is proven by application of classical Hoare logic rules and If1-Inside.

```
AuthPwdLeak:
{a:Account ∧ a′:Account ∧ a.password == pwd}
  res=a′.authenticate(_)
  {res != pwd}                                                        by classical spec.

{a:Account ∧ a′:Account ∧ a.password == pwd ∧ ¬ false}
  res=a′.authenticate(_)
  {res != pwd}                                                        by classical Hoare logic

from inside(pwd) ∧ a, a′:Account ∧ a.password=pwd ∧ ⟨_ calls a′.authenticate(_)⟩
  next ¬inside(_)   onlyIf false                                       by If1-Inside
```

5.3 Part 3: Per-Step *Necessity* Specifications

The next step is to construct proofs of necessary conditions for *any* possible step in our external state semantics. In order to prove the final result in the next section, we need to prove three per-step *Necessity* specifications: BalanceChange, PasswordChange, and PasswordLeak.

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

PasswordChange ≜ from a:Account ∧ a.password=p
                 next ¬ a.password != p   onlyIf ⟨_ calls a.changePass(a.password,_)⟩

PasswordLeak ≜ from a:Account ∧ a.password=p ∧ inside<p>
               next ¬ inside<p>   onlyIf false
```

We provide the proofs of these in Appendix F, but describe the construction of the proof of BalanceChange here: by application of the rules/results AuthBalChange, changePassBalChange, Ledger::TransferBalChange, Bank::TransferBalChange, BalanceEncaps, and If1-Internal.

5.4 Part 4: Emergent *Necessity* Specifications

Finally, we combine our module-wide single-step *Necessity* specifications to prove emergent behavior of the entire system. Informally the reasoning used in the construction of the proof of NecessityBankSpec can be stated as

(1) If the balance of an account decreases, then by BalanceChange there must have been a call to transfer in Bank 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 changePassword 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)**.

NecessityBankSpec:

from a:Account ∧ b:Bank ∧ b.balance(a)=bal to b.balance(a) < bal onlyThrough ⟨_ calls b.transfer(a.password,_,a,_)⟩	by Changes and BalanceChange
from a:Account ∧ b:Bank ∧ b.balance(a)=bal to b.balance(a) < bal onlyThrough ∃ o.[⟨o external⟩ ∧ ⟨o access a.password⟩]	by ⟶, Caller-Ext, and Calls-Args
from a:Account ∧ b:Bank ∧ b.balance(a)=bal ∧ a.password=pwd to b.balance(a) < bal onlyThrough ¬inside(a.password)	by ⟶
from a:Account ∧ b:Bank ∧ b.balance(a)=bal ∧ a.password=pwd to b.balance(a) < bal onlyThrough ¬inside(a.password) ∧ (a.password=pwd ∨ a.password != pwd)	by ⟶ and Excluded Middle
from a:Account ∧ b:Bank ∧ b.balance(a)=bal ∧ a.password=pwd to b.balance(a) < bal onlyThrough (¬inside(a.password) ∧ a.password=pwd) ∨ (¬inside(a.password) ∧ a.password != pwd)	by ⟶
from a:Account ∧ b:Bank ∧ b.balance(a)=bal ∧ a.password=pwd to b.balance(a) < bal onlyThrough ¬inside(pwd) ∨ a.password != pwd	by ⟶

Case A (¬inside(pwd)):

from a:Account ∧ b:Bank ∧ b.balance(a)=bal ∧ a.password=pwd to ¬inside(pwd) onlyIf inside(pwd) ∨ ¬inside(pwd)	by If-⟶ and Excluded Middle
from a:Account ∧ b:Bank ∧ b.balance(a)=bal ∧ a.password=pwd to ¬inside(pwd) onlyIf ¬inside(pwd)	by ∨E and PasswordLeak

Case B (a.password != pwd):

from a:Account ∧ b:Bank ∧ b.balance(a)=bal ∧ a.password=pwd to a.password != pwd onlyThrough ⟨_ calls a.changePass(pwd,_)⟩	by Changes and PasswordChange
from a:Account ∧ b:Bank ∧ b.balance(a)=bal ∧ a.password=pwd to a.password != pwd onlyThrough ¬inside(pwd)	by ∨E and PasswordLeak
from a:Account ∧ b:Bank ∧ b.balance(a)=bal ∧ a.password=pwd to a.password != pwd onlyIf ¬inside(pwd)	by **Case A** and Trans

from a:Account ∧ b:Bank ∧ b.balance(a)=bal ∧ a.password=pwd to b.balance(a) < bal onlyIf ¬inside(pwd)	by **Case A**, **Case B**, If-$\forall I_2$, and If-⟶

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 lamnguage 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 2001a]. 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 ($\prec:$) o's owner: $(\forall x.[\langle x\ \texttt{access}\ o\rangle \ \Rightarrow\ \texttt{rep}(x) \prec: \texttt{owner}(o)])$. *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

Bad things can happen to good programs. In an open world, every accessible API is an attack surface: and every combination of API calls is a potential attack. Developers can no longer just consider components in splendid isolation, verifying sufficient pre and postconditions of each method, but must reckon with the emergent behaviour of entire software systems. Programmers must identify the necessary conditions under which anything could happen [Kilour et al. 1981] — good things and bad things alike — and ensure the necessary conditions for bad things happening never arise.

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`"), via an external states semantics.

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.

Our formalisation of *Necessity* has a number of limitations that should be addressed in future work. Our classical specifications require explicit framing: some form 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]) could make the proofs more convenient. A bespoke program logic could make it easier to derive *Necessity* from classical specifications. Our Coq formalisation is parametric with respect to assertion satisfaction, encapsulation, and the type system. Encapsulation in particular is very coarse-grained, and we currently forbid "callbacks" out to external objects. A better integrated model of encapsulation should let us remove these restrictions.

To conclude: bad things can always happen to good programs. *Necessity* specifications are necessary to make sure good programs don't do bad things in response.

REFERENCES

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*.

Michele Bugliesi, Stefano Calzavara, Università Ca, Foscari Venezia, Fabienne Eigner, and Matteo Maffei. 2011. M.: Resource-Aware Authorization Policies for Statically Typed Cryptographic Protocols. In *In: CSF'11*. IEEE, 83–98.

Christoph Jentsch. 2016. Decentralized Autonomous Organization to automate governance. (March 2016). https://download.slock.it/public/DAO/WhitePaper.pdf

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. ACM.

David G. Clarke, John M. Potter, and James Noble. 2001. Simple Ownership Types for Object Containment. In *ECOOP*.

Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 379–393. https://www.microsoft.com/en-us/research/publication/cheriabi-enforcing-valid-pointer-provenance-and-minimizing-pointer-privilege-in-the-posix-c-run-time-environment/ Best paper award winner.

Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–171.

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *IEEE EuroS&P*. 147–162. https://doi.org/10.1109/EuroSP.2016.22

Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. 2014. Declarative Policies for Capability Control. In *Computer Security Foundations Symposium (CSF)*.

Sophia Drossopoulou and James Noble. 2014. Towards Capability Policy Specification and Verification. `ecs.victoria-.ac.nz/Main/TechnicalReportSeries`.

Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. 2020a. Holisitic Specifications for Robust Programs - Coq Model. https://doi.org/10.5281/zenodo.3677621

Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. 2020b. Holistic Specifications for Robust Programs. In *Fundamental Approaches to Software Engineering*, Heike Wehrheim and Jordi Cabot (Eds.). Springer International Publishing, Cham, 420–440. https://doi.org/10.1007/978-3-030-45234-6_21

Sophia Drossopoulou, James Noble, and Mark Miller. 2015. Swapsies on the Internet: First Steps towards Reasoning about Risk and Trust in an Open World. In *(PLAS)*.

Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2007. A Type Discipline for Authorization in Distributed Systems. In *CSF (Springer)*.

A.D. Gordon and A. Jeffrey. 2001a. Authenticity by typing for security protocols. In *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.* 145–159. https://doi.org/10.1109/CSFW.2001.930143

Andrew D. Gordon and Alan Jeffrey. 2001b. Typing correspondence assertions for communication protocols. In *MFPS*. Elsevier, ENTCS.

John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. 2012. Behavioral interface specification languages. *ACM Comput.Surv.* 44, 3 (2012), 16.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Comm. ACM* 12 (1969), 576–580.

S. S. Ishtiaq and P. W. O'Hearn. 2001. BI as an assertion language for mutable data structures. In *POPL*. 14–26.

David Kilour, Hamish Kilgour, and Robert Scott. 1981. Anything Could Happen. In *Boodle Boodle Boodle*. Flying Nun Records.

Leslie Lamport. 1977. Proving the Correctness of Multiprocess Programs. *TSE* SE-3, 2 (March 1977), 125–143.

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

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

K. R. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR16*. Springer.

K. Rustan M. Leino. 2013. Developing verified programs with dafny. In *ICSE*. 1488–1490. https://doi.org/10.1109/ICSE.2013.6606754

K. Rustan M. Leino and Wolfram Schulte. 2007. Using History Invariants to Verify Observers. In *ESOP*.

David Lewis. 1973. Causation. *Journal of Philosophy* 70, 17 (1973).

S. Maffeis, J.C. Mitchell, and A. Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *Proc of IEEE Security and Privacy*.

Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51.

Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Baltimore, Maryland.

Mark Samuel Miller. 2011. Secure Distributed Programming with Object-capabilities in JavaScript. (Oct. 2011). Talk at Vrije Universiteit Brussel, mobicrant-talks.eventbrite.com.

Mark S. Miller, Tom Van Cutsem, and Bill Tulloh. 2013. Distributed Electronic Rights in JavaScript. In *ESOP*.

Mark Samuel Miller, Chip Morningstar, and Bill Frantz. 2000. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. Springer.

Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Safe active content in sanitized JavaScript. code.google.com/p/google-caja/.

Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. 2016. Extensible access control with authorization contracts. In *OOPSLA*, Eelco Visser and Yannis Smaragdakis (Eds.). 214–233.

Toby Murray. 2010. *Analysing the Security Properties of Object-Capability Patterns*. Ph.D. Dissertation. University of Oxford.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. 2013. Noninterference for Operating Systems kernels. In *International Conference on Certified Programs and Proofs*.

James Noble, Robert Biddle, Ewan Tempero, Alex Potanin, and Dave Clarke. 2003. Towards a Model of Encapsulation. In *IWACO*.

James Noble, John Potter, and Jan Vitek. 1998. Flexible Alias Protection. In *ECOOP*.

Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371078

Matthew Parkinson and Alexander J. Summers. 2011. The Relationship between Separation Logic and Implicit Dynamic Frames. In *ESOP*.

D.J. Pearce and L.J. Groves. 2015. Designing a Verifying Compiler: Lessons Learned from Developing Whiley. *Sci. Comput. Prog.* (2015).

Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *IEEE Symp. on Security and Privacy*.

Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *CAV*. https://doi.org/10.1007/978-3-030-53291-8_14

Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2019. The High-Level Benefits of Low-Level Sandboxing. *Proc. ACM Program. Lang.* 4, POPL, Article 32 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371100

Ina Schaefer, Tobias Runge, Alexander Knüppel, Loek Cleophas, Derrick G. Kourie, and Bruce W. Watson. 2018. 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*. 502–515. https://doi.org/10.1007/978-3-030-03418-4_30

Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit Dynamic Frames. *ToPLAS* (2012).

Alexander J. Summers and Sophia Drossopoulou. 2010. Considerate Reasoning and the Composite Pattern. In *VMCAI*.

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. In *OOPSLA*.

The Ethereum Wiki. 2018. ERC20 Token Standard. (Dec. 2018). https://theethereum.wiki/w/index.php/ERC20_Token_Standard

Thomas Van Strydonck, Aına Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. 2022. Proving full-system security properties under multiple attacker models on capability machines. CSF (2022).

Jan Vitek and Boris Bokowski. 1999. Confined Types. In *OOPLSA*.

Steve Zdancewic and Andrew C. Myers. 2001. Secure Information Flow and CPS. In *Proceedings of the 10th European Symposium on Programming Languages and Systems (ESOP '01)*. Springer, London, UK, UK, 46–61. http://dl.acm.org/citation.cfm?id=645395.651931