Necessity Specifications for Robustness

JULIAN MACKAY, Victoria University of Wellington, New Zealand
SUSAN EISENBACH, Imperial College London, United Kingdom
JAMES NOBLE, Creative Resaerch & Programming, New Zealand
SOPHIA DROSSOPOULOU, Meta, and Imperial College London, United Kingdom

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

CCS Concepts: • **Software and its engineering → General programming languages**.

Additional Key Words and Phrases: Necessity. Necessary Conditions. Correctness. Verification.

ACM Reference Format:
Julian Mackay, Susan Eisenbach, James Noble, and Sophia Drossopoulou. 2022. *Necessity* Specifications for Robustness. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 154 (October 2022), 29 pages. https://doi.org/10.1145/3563317

1 INTRODUCTION: NECESSARY CONDITIONS AND ROBUSTNESS

Software needs to be both *correct* (programs do what they are supposed to) and *robust* (programs *only* do what they are supposed to). We use the term *robust* as a generalisation of *robust safety* [Bugliesi et al. 2011; Gordon and Jeffrey 2001; Swasey et al. 2017] whereby a module or process or ADT is *robustly safe* if its execution preserves some safety guarantees even when run together with unknown, unverified, potentially malicious client code. The particular safety guarantees vary across the literature. We are interested in *program-specific* safety guarantees which describe *necessary conditions* for some effect to take place. In this work we propose how to specify such necessary conditions, and how to prove that modules adhere to such specifications.

We motivate the need for necessary conditions for effects through an example: Correctness is traditionally specified through Hoare [1969] triples: a precondition, a code snippet, and a postcondition. For example, part of the functional specification of a `transfer` method for a bank module is that the source account's balance decreases:

$$S_{\texttt{correct}} \triangleq \{\texttt{pwd=src.pwd} \land \texttt{src.bal=b}\}\ \texttt{src.transfer(dst,pwd)}\ \{\texttt{src.bal=b-100} \land \ldots\}$$

Calling `transfer` on an account with the correct password will transfer the money.

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

Authors' addresses: Julian Mackay, Victoria University of Wellington, New Zealand, julian.mackay@ecs.vuw.ac.nz; Susan Eisenbach, Imperial College London, United Kingdom, susan@imperial.ac.uk; James Noble, Creative Resaerch & Programming, 5 Fernlea Ave, Darkest Karori, Wellington, 6012, New Zealand, kjx@acm.org; Sophia Drossopoulou, Meta, and Imperial College London, United Kingdom, scd@imperial.ac.uk.

2475-1421/2022/10-ART154
https://doi.org/10.1145/3563317

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

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

Specification $S_{\texttt{robust_1}}$guarantees that it is not possible to take money out of the account without calling `transfer` and without providing the password. Calling `transfer` with the correct password is a *necessary condition* for (the effect of) reducing the account's balance.

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

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

With $S_{\texttt{robust\_2}}$, I can confidently pass my account to any, potentially untrusted context, where my password is not known; the payment I was expecting may or may not be made, but I know that my money will not be stolen [Miller 2011]. Note that $S_{\texttt{robust\_2}}$ does not mention the names of any functions in the module, and thus can be expressed without reference to any particular API — indeed $S_{\texttt{robust\_2}}$ can constrain *any* API with an account, an account balance, and a password.

Earlier work addressing robustness includes object capabilities [Birkedal et al. 2021; Devriese et al. 2016; Miller 2006], information control flow [Murray et al. 2013; Zdancewic and Myers 2001], correspondence assertions [Fournet et al. 2007], sandboxing [Patrignani and Garg 2021; Sammler et al. 2019], robust linear temporal logic [Anevlavis et al. 2022] – to name a few. Most of these propose *generic* guarantees (*e.g.* no dependencies from high values to low values), or preservation of module invariants, while we work with *problem-specific* guarantees concerned with necessary conditions for specific effects (*e.g.* no decrease in balance without access to password). VerX [Permenev et al. 2020] and *Chainmail* [Drossopoulou et al. 2020b] also work on problem-specific guarantees. Both these approaches are able to express necessary conditions like $S_{\texttt{robust\_1}}$ using temporal logic operators and implication, and Chainmail is able to express $S_{\texttt{robust\_2}}$, however neither have a proof logic to prove adherence to such specifications.

1.1 *Necessity*

In this paper we introduce *Necessity*, the first approach that is able to both express and prove (through an inference system) robustness specifications such as $S_{\texttt{robust\_2}}$. Developing a specification language with a proof logic that is able to prove properties such as $S_{\texttt{robust\_2}}$ and must tread a fine line: the language must be rich enough to express complex specifications; temporal operators are needed along with object capability style operators that describe *permission* and *provenance*, while also being simple enough that proof rules might be devised.

The first main contribution is three novel operators that merge temporal operators and implication and most importantly are both expressive enough to capture the examples we have found in the literature and provable through an inference system. One such necessity operator is

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

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

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

Namely, a transition from a current state where an account's balance is `bal`, to a future state where it has decreased, may *only* occur if in the current state some `external`, unknown client object has access to that account's password. More in §2.3.

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

The second main contribution is a logic that enables us to prove that code obeys *Necessity* specifications. Our insight was that *Necessity* specifications are logically equivalent to the intersection of an *infinite* number of Hoare triples, *i.e.*, `from` A_1 `to` A_2 `onlyIf` A_3 is logically equivalent to $\forall$`stmts`.$\{A_1 \land \neg A_3\}$`stmts`$\{\neg A_2\}$. Note that in the above, the assertions A_1, A_2 and A_3 are fixed, while the code (stmts) is universally quantified. This leaves the challenge that, usually, Hoare logics do not support such infinite quantification over the code.

We addressed that challenge through three further insights: (1) *Necessity* specifications of emergent behaviour can be built up from *Necessity* specifications of single-step executions, which (2) can be built from encapsulation and *finite* intersections of *Necessity* specifications of function calls, which (3) in turn can be obtained from *traditional* functional specifications.

1.2 Contributions and Paper Organization

The contributions of this work are:

(1) A language to express *Necessity* specifications (§3), including three novel *Necessity* operators (§3.3) that combine implication and temporal operators.
(2) A logic for proving a module's adherence to its *Necessity* specifications (§4), and a proof of soundness of the logic, (§4.5), both mechanised in Coq [Mackay et al. 2022a].
(3) A proof in our logic that our bank module obeys $S_{\texttt{robust\_2}}$ (§5), mechanised in Coq. And a proof that a richer bank module which uses ghostfields and confined classes obeys $S_{\texttt{robust\_2}}$ (detailed in the extended paper [Mackay et al. 2022b]), also mechanised in Coq.
(4) Examples taken from the literature (§3.4, and the appendices [Mackay et al. 2022b]) specified in *Necessity*.

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 proofs of examples in the appendices [Mackay et al. 2022b]. definitions and further examples. In the next section, (§2), we outline our approach using a bank as a motivating example.

A strength of our work is that it is parametric with respect to assertion satisfaction and functional specifications – these questions are well covered in the literature, and offer several off-the-shelf solutions. The current work is based on a simple, imperative, typed, object oriented language with unforgeable addresses and private fields; nevertheless, 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]. In line with other work in the literature, we do not –yet– support "callbacks" out from internal objects (whose code gas been checked) to external objects (unknown objects whose code is unchecked).

2 OUTLINE OF OUR APPROACH

In this Section we outline our approach: we revisit our running example, the Bank Account (§2.1), introduce the three necessity operators (§2.2), give the *Necessity* specs (§2.3), outline how we model the open world (§2.4), give the main ideas of our proof system (§2.5) and outline how we use it to reason about adherence to *Necessity* specifications (§2.6).

2.1 Bank Account – three modules

Module $\texttt{Mod}_{\texttt{good}}$ consists of an empty `Password` class where each instance models a unique password, and an `Account` class with a password, and a balance, an `init` method to initialize the password, and a `transfer` method. Note that we assume that all fields are "class-private", i.e., methods may read and write fields of any instance of the same class, and that passwords are unforgeable and not enumerable (as in Java, albeit without reflection).

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

We can capture the intended semantics of `transfer` through a functional specification with pre- and post- conditions and `MODIFIES` clauses as *e.g.*, in Leavens et al.; Leino. The implementation of `transfer` in module $\texttt{Mod}_{\texttt{good}}$ meets this specification.

```
FuncSpec  ≜
   method transfer(dest:Account, pwd':Password) -> void
      ENSURES:this.pwd=pwd' ∧ this≠dest  ⟶
              this.balance_post =this.balance_pre-100 ∧
              dest.balance_post =dest.balance_pre+100
      ENSURES:this.pwd≠pwd' ∨ this=dest  ⟶
              this.balance_post =this.balance_pre ∧ dest.balance_post =dest.balance_pre
      MODIFIES:this.balance, dest.balance
```

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

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

  class Password
```

```
module Mod_better
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...


    method set(pwd',pwd'': Password)
      if (this.pwd==pwd')
        this.pwd=pwd''
  class Password
```

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

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

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

This example also demonstrates the importance of field privacy: $\texttt{Mod}_{\texttt{good}}$ and $\texttt{Mod}_{\texttt{better}}$ would not be any more robust than $\texttt{Mod}_{\texttt{bad}}$ if the underlying programming language did not restrict access to fields. Without such a restriction, any external object would have been able to directly manipulate the fields `balance` and `pwd`.

2.2 The three necessity operators

We need a specification that rules out $\texttt{Mod}_{\texttt{bad}}$ while permitting $\texttt{Mod}_{\texttt{good}}$ and $\texttt{Mod}_{\texttt{better}}$. For this, we will be using one of the three necessity operators mentioned in §1.1. These operators are:

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

The first operator was already introduced in §1.1: it says that a transition from a current state satisfying assertion A_{curr} to a future state satisfying A_{fut} is possible only if the necessary condition A_{nec} holds in the *current* state. The second operator says that a *one-step* transition from a current state satisfying assertion A_{curr} to a future state satisfying A_{fut} is possible only if A_{nec} holds in the *current* state. The third operator says that a change from A_{curr} to A_{fut} may happen only if A_{intrm} holds in some *intermediate* state.

Our assertions A, also allow for the use of capability operators, such as 1) having `access` to an object ($\langle$`o access o'`$\rangle$) which means that `o` has a reference to `o'`, or 2) `calling` a method with on receiver with certain arguments, ($\langle$`o calls o'.m(args)`$\rangle$), or 3) an object being `external`, where $\langle$`o external`$\rangle$ means that `o` belongs to a class that is not defined in the current module, and thus its behaviour is unrestricted. These are the capability operators we adopted from Chainmail.

2.3 Bank Account – the right specification

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

In *Necessity* syntax, and recalling §1.1, and 2.2,

```
S_robust_1  ≜  from    a:Account ∧ a.balance==bal    next    a.balance < bal
               onlyIf ∃ o,a'. [⟨o external⟩ ∧ ⟨o calls a.transfer(a',a.pwd)⟩]
S_robust_2  ≜  from    a:Account ∧ a.balance==bal    to    a.balance < bal
               onlyIf ∃ o.[⟨o external⟩ ∧ ⟨o access a.pwd⟩]
```

$S_{\texttt{robust\_1}}$ does not fit the bill: all three modules satisfy it. But $S_{\texttt{robust\_2}}$ does fit the bill: $\texttt{Mod}_{\texttt{good}}$ and $\texttt{Mod}_{\texttt{better}}$ satisfy $S_{\texttt{robust\_2}}$, while $\texttt{Mod}_{\texttt{bad}}$ does not.

A critical point of $S_{\texttt{robust\_2}}$ is that it is expressed in terms of observable effects (the account's balance is reduced: `a.balance < bal`) and the shape of the heap (external access to the password: $\langle$`o external`$\rangle$ ∧ $\langle$`o access a.pwd`$\rangle$) 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 *implementations* of a bank account with a balance and a password, irrespective of the API it offers, the services it exports, or the dependencies on other parts of the system.

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

2.3.1 How useful is $S_{\texttt{robust\_2}}$? One might think that $S_{\texttt{robust\_2}}$ was not useful: normally, there will exist somewhere in the heap at least one external object with access to the password – if no such object existed, then nobody would be able to use the money of the account. And if such an object did exist, then the premise of $S_{\texttt{robust\_2}}$ would not hold, and thus the guarantee given by $S_{\texttt{robust\_2}}$ might seem vacuous.

This is *not* so: in scopes from which such external objects with access to the password are not (transitively) reachable, $S_{\texttt{robust\_2}}$ guarantees that the balance of the account will not decrease. We illustrate this through the following code snippet:

```
1  module Mod₁
2       ...
3     method cautious(untrusted:Object)
4         a = new Account
5         p = new Password
6         a.set(null,p)
7         ...
8         untrusted.make_payment(a)
9         ...
```

The method `cautious` has as argument an object `untrusted`, of unknown provenance. It creates a new `Account` and initializes its password. In the scope of this method, external objects with access to the password are reachable: thus, line 7, or line 9 may decrease the balance.

Assume that class `Account` is from a module which satisfies $S_{\texttt{robust\_2}}$. Assume also that the code in line 7 does not leak the password to `untrusted`. Then no external object reachable from the scope of execution of `make_payment` at line 8 has access to the password. Therefore, even though we are calling an untrusted object, $S_{\texttt{robust\_2}}$ guarantees that `untrusted` will not be able to take any money out of `a`.

A proof sketch of the safety provided by $S_{\texttt{robust\_2}}$ appears in the appendices [Mackay et al. 2022b]. Note that in this example, we have (at least) three modules: the internal module which defines class `Account` adhering to $S_{\texttt{robust\_2}}$, the external module $\texttt{Mod}_1$, and the external module which contains the class definition for `untrusted`. Our methodology allows the external module, $\texttt{Mod}_1$ to reason about its own code, and thus pass `a` to code from the second external module, without fear of losing money. In further work we want to make such arguments more generally applicable, and extend Hoare logics to encompass such proof steps.

2.4 Internal and external modules, 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. We also distinguish between *internal* calls (from either an internal or an external object) made to internal objects and *external* calls made to external objects. In the code snippet from §2.3.1, the call to `set` on line 6 is an internal call, while the call to `make_payment` is an external call – `this` to `untrusted` (both 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. In line with other work in the literature [Albert et al. 2020; Grossman et al. 2017; Permenev et al. 2020], we currently forbid calls from internal to external objects – further details on call-backs in §6.

For the purposes of the current work we are only interested in one internal, and one external module. But the interested reader might ask: what if there is more than one external module? The answer is that from the internal module's viewpoint, all external modules are considered as one; for this we provide a module linking operator with the expected semantics – more details in Def. 3.1 and the appendices [Mackay et al. 2022b]. But from the external module's viewpoint, there may be more than one external module: for example, in §2.3.1, module $\texttt{Mod}_1$ is external to the module implementing class `Account`, and the module of `untrusted` is external to $\texttt{Mod}_1$.

2.5 Reasoning about *Necessity*

We will now outline the key ingredients of our logic with which we prove that modules obey *Necessity* specifications. We will use the auxiliary concept that an assertion A is *encapsulated* by a module M, if A can only be invalidated through a call to a method from M – more in §4.1.

The *Necessity* logic is based on the insight that the specification

$$\texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A_3$$

is logically equivalent to

$$\forall \texttt{stmts}.\{A_1 \land \neg A_3\}\texttt{stmts}\{\neg A_2\}$$

– that is, with an *infinite* conjunction of Hoare triples, where the three assertions are fixed, but the code, `stmts`, is universally quantified. This leaves the challenge that usually, Hoare logics do not support such infinite conjunctions over code. Three ideas helped us address that challenge:

From Hoare triples to per-call specs The Hoare triple $\{A_1 \land \neg A_3\}\ \texttt{x.m(ys)}\ \{\neg A_2\}$ is logically equivalent to the specification $\texttt{from}\ (A_1 \land \langle \_\ \texttt{calls}\ \texttt{x.m(ys)}\rangle)\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3$.

From per-call specs to per-step specs If an assertion A_2 is *encapsulated* by a module – and thus the only way from a state that satisfies A_2 to a state that does not, is through a call to a method in that module – then the *finite conjunction* that all methods of that module $\texttt{from}\ (A_1 \land A_2 \land \langle \_\ \texttt{calls}\ \texttt{x.m(ys)}\rangle)\ \texttt{next}\ \neg A_2\ \texttt{onlyIf}\ A_3$ is logically equivalent to $\texttt{from}\ A_1 \land A_2\ \texttt{next}\ \neg A_2\ \texttt{onlyIf}\ A_3$.

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

Thus, our system consists of four parts (five including functional specifications): (**Part 1**) assertion encapsulation, (**Part 2**) per-method specifications, (**Part 3**) per-step specifications, and (**Part 4**) specifications of emergent behaviour. The structure of the system, and the dependency of each part on preceding parts is given in Fig. 1. Functional specifications are used to prove per-method specifications, which coupled with assertion encapsulation is used to prove per-step specifications, which is used to prove specifications of emergent behaviour.

Necessity logic is parametric with respect to the way we ascertain if an assertion is encapsulated and the way we obtain functional specifications. As a result we can use results from many different approaches. Further, our proofs of *Necessity* do not inspect method bodies: we rely on simple annotations to infer encapsulation, and on pre and post-conditions to infer per-method conditions.

2.6 Outline of the proof that $\texttt{Mod}_{\texttt{better}}$ obeys $S_{\texttt{robust\_2}}$

For illustration, we outline a proof that $\texttt{Mod}_{\texttt{better}}$ adheres to $S_{\texttt{robust\_2}}$. note that for illustration purposes, in this paper we show how assertion encapsulation can be proven based on simple

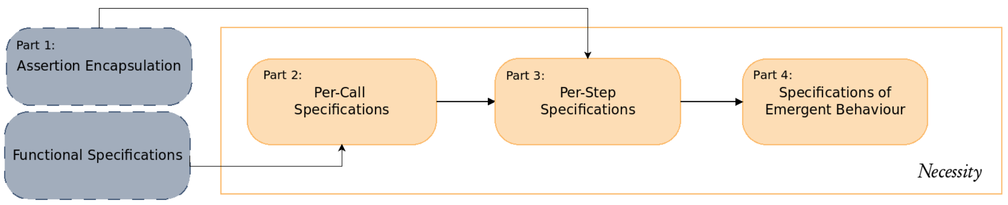

Fig. 1. Parts of *Necessity* Logic and their Dependencies. Note that gray parts with a dashed border indicate parts that are not part of *Necessity*, and on which *Necessity* is parametric.

annotations inspired by confinement types [Vitek and Bokowski 1999]; we could just as easily rely on other language mechanisms, *e.g.*, ownership types, or even develop custom logics.

Part 1: Assertion Encapsulation.

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

Part 2: Per-Method Specifications

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

Part 3: Per-step Specifications

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

Part 4: Specifications of Emergent Behaviour

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

3 THE MEANING OF NECESSITY

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

3.1 `TooL`

`TooL` is a small, imperative, sequential, class based, typed, object-oriented language, whose fields are private to the class where they are defined. `TooL` is straightforward and the complete definition can be found in the appendices [Mackay et al. 2022b]. `TooL` is based on $\mathscr{L}_{oo}$ [Drossopoulou et al. 2020b], with some small variations, as well as the addition of a simple type system – more in 4.1.2. 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 statement to be executed next.

As discussed in §2.5, open world specifications need to be able to provide 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

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

The function $\mathit{classOf}(\sigma, \_)$ is overloaded: applied to a variable, $\mathit{classOf}(\sigma, x)$ looks up the variable x in the top frame of σ, and returns the class of the corresponding object in the heap of σ; applied to an address, $\mathit{classOf}(\sigma, \alpha)$ 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. The details can be found in the appendices[Mackay et al. 2022b].

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

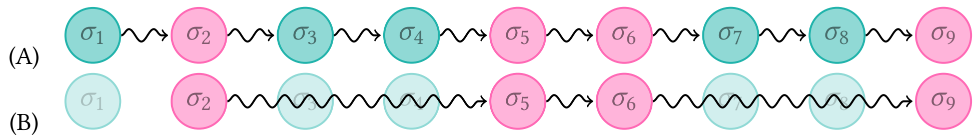

Fig. 2. External States Semantics (Def. 3.1), (A) $M' \circ M, \sigma_1 \rightsquigarrow \ \ldots \rightsquigarrow \sigma_9$ and (B) $M'; M,\ \sigma_2 \rightsquigarrow \ldots \rightsquigarrow \sigma_9$, where *classOf*($\sigma_1$, `this`),*classOf*($\sigma_3$, `this`),*classOf*($\sigma_4$, `this`),*classOf*($\sigma_7$, `this`),*classOf*($\sigma_8$, `this`) $\in M$, and where *classOf*(σ_2, `this`), *classOf*(σ_5, `this`)*classOf*(σ_6, `this`), *classOf*(σ_9, `this`) $\in M'$.

with internal states or states that can never arise. A state σ is *arising*, written $Arising(M', M, \sigma)$, if it may arise by external states execution starting at some initial configuration:

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

An *Initial* state's heap contains a single object of class `Object`, and its stack consists of a single frame, whose local variable map is a mapping from `this` to the single object, and whose continuation is any statement. (See Definition 3.2 and the appendices [Mackay et al. 2022b]).

Applicability. While our work is based on a simple, imperative, typed, object oriented language with unforgeable addresses and private fields, we believe that it is applicable to several programming paradigms, and that unforgeability and privacy can be replaced by lower level mechanisms such as capability machines [Davis et al. 2019; Van Strydonck et al. 2022].

3.2 *Assert*

Assert is a basic assertion language extended with object-capability assertions.

3.2.1 Syntax 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 an internal (i.e. trusted) object, and y is an external (i.e. untrusted) object.
- *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:

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

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 TooL may be recursively defined, and thus evaluation need not 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 $\hookrightarrow$ is unsurprising (see the appendices [Mackay et al. 2022b]).

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* $classOf(\sigma, \alpha) = C$
(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 $\alpha \in \sigma$.`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 $\alpha \in \sigma$.`heap`.
(8) $M, \sigma \models \langle x$ `access` $y \rangle$ iff
  (a) $\lfloor x.f \rfloor_\sigma = \lfloor y \rfloor_\sigma$ for some f,
  or
  (b) $\lfloor x \rfloor_\sigma = \lfloor \texttt{this} \rfloor_\phi$, $\lfloor y \rfloor_\sigma = \lfloor z \rfloor_\phi$, and $z \in \phi$.`contn` for some variable z, and some frame ϕ in σ.`stack`.
(9) $M, \sigma \models \langle x$ `internal` $\rangle$ iff $classOf(\sigma, x) \in M$
(10) $M, \sigma \models \langle x$ `external` $\rangle$ iff $classOf(\sigma, x) \notin M$
(11) $M, \sigma \models \langle x$ `calls` $y.m(z_1, \ldots, z_n) \rangle$ iff
  (a) σ.`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 $1 \leq i \leq n$

Quantification (defined in 6 and 7) is done over all objects on the heap. We do not include quantification over primitive types such as integers as `TooL` is too simple. The Coq mechanisation does include primitive types.

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

The assertion $\langle x$ `calls` $y.m(z_1, \ldots, z_n) \rangle$ (defined in 11) describes the current innermost active call. It requires that the current receiver (`this`) is x, and that it calls the method m on y with arguments z_1, ... z_n – It does *not* mean that somewhere in the call stack there exists a call from x to $y.m(...)$. 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 $Arising(M', M, \sigma)$, 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 judgement, but will rely on its existence later on for Theorem 4.4. We define soundness of such a judgement in the usual way:

Definition 3.6 (Soundness of Assert Provability). A judgement of the form $M \vdash A$ 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). $\texttt{inside}(o) \triangleq \forall x.[\langle x \ \texttt{access}\ o\rangle \Rightarrow \langle x\ \texttt{internal}\rangle]$

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

3.3 *Necessity* operators

3.3.1 Syntax of Necessity Specifications. The *Necessity* specification language extends *Assert* with our three novel *Necessity operators*:

from A_1 **next** A_2 **onlyIf** A : If an arising state satisfies A_1, and a single execution step reaches a state satisfying A_2, then the original state must have also satisfied A.

from A_1 **to** A_2 **onlyIf** A : If an arising state satisfies A_1 and a number of execution steps reach a state satisfying A_2, then the original state must have also satisfied A.

from A_1 **to** A_2 **onlyThrough** A : If an arising state satisfies A_1, and a number of execution steps reach a state satisfying A_2, then execution must have passed through some *intermediate* state satisfying A.

The syntax of *Necessity* specifications is given below

Definition 3.8. Syntax of Necessity Specifications

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

As an example, we consider the following three specifications:

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

$S_{\texttt{nxt\_dcr\_if\_acc}}$ requires that an account's balance may decrease in *one step* (go from a state where the balance is `bal` to a state where it is less than `bal`) only if the password is accessible to an external object (in the original state an external object had access to the password). $S_{\texttt{to\_dcr\_if\_acc}}$ requires that an account's balance may decrease in *any number of steps* only if the password is accessible to an external object. $S_{\texttt{to\_dcr\_thr\_acc}}$ requires that an account's balance may decrease in *any number of steps* only if in *some intermediate state* the password was accessible to an external object – the *intermediate* state where the password is accessible to the external object might be the *starting* state, the *final* state, or any state in between.

3.3.2 Semantics of Necessity Specifications. We now define what it means for a module M to satisfy specification S, written as $M \vDash S$. The Definition 3.9 below is straightforward, apart from the use of the $\sigma' \triangleleft \sigma$ (best read as "σ' seen from σ") to deal with the fact that execution might change the bindings in local variables. We explain this in detail in §3.3.3, but for now, the reader may ignore the applications of that operator and read $\sigma' \triangleleft \sigma$ as σ', and also read $\sigma_k \triangleleft \sigma_1$ as σ_k. We illustrate the meaning of the three operators in Fig. 3.

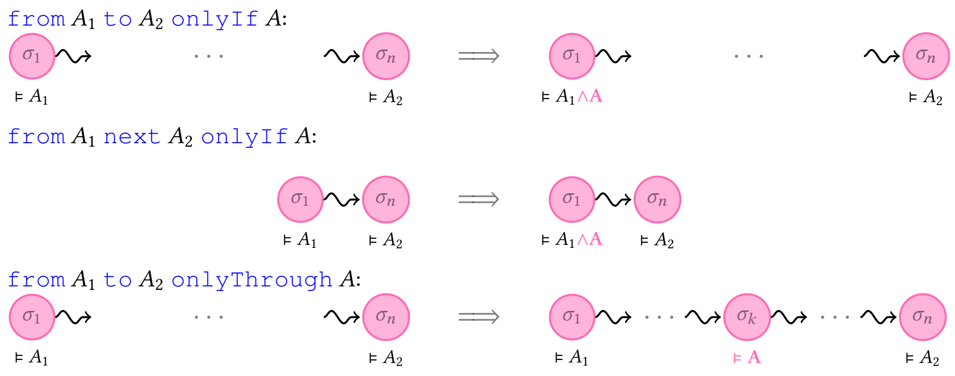

Fig. 3. Illustrating the three *Necessity* operators

Definition 3.9 (Semantics of Necessity Specifications). We define $M \vDash S$ by cases over the four possible syntactic forms. For any assertions A_1, A_2, and A:

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

- $M \vDash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A$ iff for all M', σ, σ', such that $Arising(M', M, \sigma)$:

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

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

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

- $M \vDash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A$ iff for all M', $\sigma_1, \sigma_2, ... \sigma_n$, such that $Arising(M', M, \sigma_1)$:

$$\left.\begin{array}{l} \text{- } M, \sigma_1 \vDash A_1 \\ \text{- } M, \sigma_n \triangleleft \sigma_1 \vDash A_2 \\ \text{- } \forall i \in [1..n).\ M'; M,\ \sigma_i \leadsto \sigma_{i+1} \end{array}\right\} \Rightarrow \quad \exists k.\ 1 \le k \le n\ \wedge\ M, \sigma_k \triangleleft \sigma_1 \vDash A$$

Revisiting the examples from the previous subsection, we obtain that all three modules satisfy $S_{\texttt{nxt\_dcr\_if\_acc}}$. But $\texttt{Mod}_{\texttt{bad}}$ does not satisfy $S_{\texttt{to\_dcr\_if\_acc}}$: as already discussed in §2.1, with `a` of class `Account` implemented as in $\texttt{Mod}_{\texttt{bad}}$, starting in a state where no external object has access to `a`'s password, and executing `a.set(42); a.transfer(rogue_account,42)` leads to a state where the balance has decreased. All three modules satisfy $S_{\texttt{to\_dcr\_thr\_acc}}$: namely, in all cases, the balance can only decrease if there was a call to `a.transfer(_,p)` where `p` = `a.pwd`,

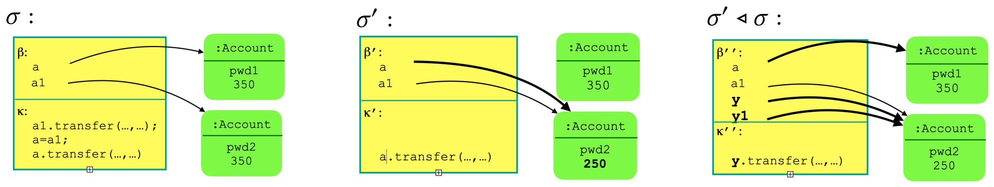

Fig. 4. Illustrating adaptation

and since that call can only be made from an external object, p is externally known at the time of that call.

$\mathtt{Mod}_{\mathtt{good}} \vDash S_{\mathtt{nxt\_dcr\_if\_acc}}$	$\mathtt{Mod}_{\mathtt{bad}} \vDash S_{\mathtt{nxt\_dcr\_if\_acc}}$	$\mathtt{Mod}_{\mathtt{better}} \vDash S_{\mathtt{nxt\_dcr\_if\_acc}}$
$\mathtt{Mod}_{\mathtt{good}} \vDash S_{\mathtt{to\_dcr\_if\_acc}}$	$\mathtt{Mod}_{\mathtt{bad}} \nvDash S_{\mathtt{to\_dcr\_if\_acc}}$	$\mathtt{Mod}_{\mathtt{better}} \vDash S_{\mathtt{to\_dcr\_if\_acc}}$
$\mathtt{Mod}_{\mathtt{good}} \vDash S_{\mathtt{to\_dcr\_thr\_acc}}$	$\mathtt{Mod}_{\mathtt{bad}} \vDash S_{\mathtt{to\_dcr\_thr\_acc}}$	$\mathtt{Mod}_{\mathtt{better}} \vDash S_{\mathtt{to\_dcr\_thr\_acc}}$

3.3.3 Adaptation. We now discuss the adaptation operator. To see the need, consider specification

```
S_to_dcr_thr_call   ≜  from a:Account ∧ a.balance==350  next a.balance == 250
                        onlyIf ∃ o.[⟨o external⟩ ∧ ⟨o calls a.transfer(_, _, _)⟩]
```

Without adaptation, the semantics of $S_{\mathtt{to\_dcr\_thr\_call}}$ would be: If .., $\sigma \models$ `a.balance==350`, and .., $\sigma \leadsto^* \sigma'$ and $\sigma' \models$ `a.balance==250`, then between σ and σ' there must be call to `a.transfer`. But if σ happened to have another account `a1` with balance `350`, and if we reach σ' from σ by executing `a1.transfer(..,..); a=a1`, then we would reach a σ' *without* `a.transfer` having been called: indeed, without the account `a` from σ having changed at all. In fact, with such a semantics, a module would satisfy $S_{\mathtt{to\_dcr\_thr\_call}}$ only if it did not support decrease of the balance by 100, or if states where an account's balance is `350` were unreachable!

This is the remit of the adaptation operator: when we consider the future state, we must "see it from" the perspective of the current state; the binding for variables such as `a` must be from the current state, even though we may have assigned to them in the mean time. Thus, $\sigma' \triangleleft \sigma$ keeps the heap from σ', and renames the variables in the top stack frame of σ' so that all variables defined in σ have the same bindings as in σ; the continuation must be adapted similarly (see Fig. 4).

Under adaptation, the semantics of $S_{\mathtt{to\_dcr\_thr\_call}}$ is: if .., $\sigma \models$ `a.balance==350`, and .., $\sigma \leadsto^* \sigma'$ and ..., $\boldsymbol{\sigma' \triangleleft \sigma} \models$ `a.balance==250`, then some intermediate state's continuation must contain a call to `a.transfer`; where, all variables bound in the initial state, σ, have the same bindings in $\sigma' \triangleleft \sigma$.

Fig. 4 illustrates the semantics of $\sigma' \triangleleft \sigma$. In σ the variable `a` points to an `Account` with password `pwd1`, and balance `350`; the variable `a1` points to an `Account` with password `pwd2`, and balance `350`; and the continuation is `a1.transfer(..,..); a=a1; a.transfer(..,..);`. We reach σ' by executing the first two statements from the continuation. Thus, $\sigma' \triangleleft \sigma \not\models$ `a.balance==250`. Moreover, in $\sigma' \triangleleft \sigma$ we introduce the fresh variables `y` and `y1`, and replace `a` and `a1` by `y` and `y1` in the continuation. This gives that $\sigma' \triangleleft \sigma \models$ ⟨_ `calls a1.transfer(...)`⟩ and $\sigma' \triangleleft \sigma \not\models$ ⟨_ `calls a.transfer(...)`⟩.

Definition 3.10 describes the $\triangleleft$ operator in all detail (it is equivalent to, but not identical to the definition given in [Drossopoulou et al. 2020b]). We introduce fresh variables $\overline{y}$ – as many as in the σ' top frame variable map – $dom(\beta') = \overline{x}$, and $|\overline{y}| = |\overline{x}|$. We extend σ's variable map (β), so that it also maps $\overline{y}$ in the way that σ''s variable map (β') maps its local variables – $\beta'' = \beta[\overline{y} \mapsto \beta'(\overline{x})]$. We rename $\overline{x}$ in σ' continuation to $\overline{y}$ – $\kappa'' = [\overline{y}/\overline{x}]\kappa'$.

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

- $\sigma' \triangleleft \sigma \triangleq (\chi', (\beta'', \kappa'') : \psi')$

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

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

3.4 Expressiveness

We discuss expressiveness of *Necessity* operators, by comparing them with one another, with temporal operators, and with other examples from the literature.

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

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

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 ]  )  ]
```

More examples. In order to investigate *Necessity*'s expressiveness, we used it for examples provided in the literature. In the appendices [Mackay et al. 2022b] we compare with examples proposed by Drossopoulou et al. [2020b], and Permenev et al. [2020].

4 PROVING NECESSITY

In this Section we provide a proof system for constructing proofs of the *Necessity* specifications defined in §3.3. As discussed in §2.5, such proofs consist of four parts:

- **(Part 1)** Proving Assertion Encapsulation (§4.1)
- **(Part 2)** Proving Per-Method *Necessity* specifications for a single internal method from the functional specification of that method (§4.2)
- **(Part 3)** Proving Per-Step *Necessity* specifications by combining per-method *Necessity* specifications (§4.3)
- **(Part 4)** Raising necessary conditions to construct proofs of properties of emergent behaviour (§4.4)

Part 1 is, to a certain extent, orthogonal to the main aims of our work; in this paper we propose a simple approach based on the type system, while also acknowledging that better solutions are possible. For Parts 2-4, we came up with the key ideas outlined in §2.5, which we develop in more detail in §4.2-§4.4.

4.1 Assertion Encapsulation

Necessity proofs often leverage the fact that some assertions cannot be invalidated unless some internal (and thus known) computation took place. We refer to this property as *Assertion Encapsulation.* In this work, we define the property $M \vDash A' \Rightarrow Enc(A)$, which states that under the conditions described by assertion A', the assertion A is encapsulated by module M. We do not mandate how this property should be derived – instead, we rely on a judgment $M \vdash A' \Rightarrow Enc(A)$ provided by some external system. Thus, *Necessity* is parametric over the derivation of the encapsulation judgment; in fact, several ways to do that are possible [Clarke and Drossopoulou 2002; Leino and Müller 2004; Noble et al. 2003]. In the appendices [Mackay et al. 2022b] we present a rudimentary system that is sufficient to support our example proof.

4.1.1 Assertion Encapsulation Semantics. As we said earlier, an assertion A is encapsulated by a module M under condition A', if in all possible states which arise from execution of module M with any other external module M', and which satisfy A', the validity of A can only be changed via computations internal to that module – *i.e.*, via a call to a method from M. In `TooL`, that means by calls to objects defined in M but accessible from the outside.

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 $Arising(M', M, \sigma)$:

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

Note that this definition uses adaptation, $\sigma' \triangleleft \sigma$. The application of the adaptation operator is necessary because we interpret the assertion A in the current state, σ, while we interpret the assertion $\neg A$ in the future state, $\sigma' \triangleleft \sigma$.

Revisiting the examples from § 2, both $\texttt{Mod}_{\texttt{bad}}$ and $\texttt{Mod}_{\texttt{better}}$ encapsulate the equality of the `balance` of an account to some value `bal`: This equality can only be invalidated through calling methods on internal objects.

$\texttt{Mod}_{\texttt{bad}} \models \texttt{a} : \texttt{Account} \ \Rightarrow\ \mathit{Enc}(\texttt{a.balance} = \texttt{bal})$
$\texttt{Mod}_{\texttt{better}} \models \texttt{a} : \texttt{Account} \ \Rightarrow\ \mathit{Enc}(\texttt{a.balance} = \texttt{bal})$

Moreover, the property that an object is only accessible from module-internal objects is encapsulated, that is, for all `o`, and all modules M:

$M \models \texttt{o} : \texttt{Object} \ \Rightarrow\ \mathit{Enc}(\texttt{inside(o)})$

This is so because any object which is only internally accessible can become externally accessible only via an internal call.

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

4.1.2 Deriving Assertion Encapsulation. Our logic does not deal with, nor rely on, the specifics of how encapsulation is derived. Instead, it relies on an encapsulation judgment and expects it to be sound:

Definition 4.2 (Encapsulation Soundness). A judgement of the form $M \vdash A' \Rightarrow \mathit{Enc}(A)$ is *sound*, if for all modules M, and assertions A and A', if

$$M \vdash A' \Rightarrow \mathit{Enc}(A) \quad \text{implies} \quad M \models A' \Rightarrow \mathit{Enc}(A).$$

Types for Assertion Encapsulation. Even though the derivation of assertion encapsulation is not the focus of this paper, for illustrative purposes, we will outline now a very simple type system which supports such derivations: We assume that field declarations, method arguments and method results are annotated with class names, and that classes may be annotated as `confined`. A `confined` object is not accessed by external objects; that is, it is always `inside`.

The type system then checks that field assignments, method calls, and method returns adhere to these expectations, and in particular, that objects of `confined` type are never returned from method bodies – this is a simplified version of the type system described in [Vitek and Bokowski 1999]. Because the type system is so simple, we do not include its formalization 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 $\mathit{Enc}_e(e)$, in the appendices [Mackay et al. 2022b], which asserts that any objects read during the evaluation of e are internal. Thus, any assertion that only involves $\mathit{Enc}_e(\_)$ expressions is encapsulated – more can be found in the appendices [Mackay et al. 2022b].

4.2 Per-Method *Necessity* Specifications

In this section we detail how we use functional specifications to prove per-method *Necessity* specifications of the form

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

where C is a class, and m a method in C.

The first key idea in §2.5 is 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 statement. Specifically, $\{P\}\ s\ \{Q\}$ implies that $\neg P$ is a *necessary precondition* for $\neg Q$ to hold following the execution of `s`.

For the use in functional specifications, we define *Classical assertions*, a subset of *Assert*, comprising only those assertions that are commonly present in other specification languages. They are restricted to expressions, class assertions, the usual connectives, negation, implication, and the usual quantifiers.

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

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

Fig. 5. Per-Method *Necessity* specifications

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 proof system that derives functional specifications of the form $M \vdash \{P\}\ \texttt{s}\ \{Q\}$. This implies that we can also have guarantees of

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

That is, the execution of $x.m(\overline{z})$ with the precondition P results in a program state that satisfies postcondition Q, where the returned value is represented by `res` in Q. We further assume that such a proof system is sound, i.e. that if $M \vdash \{P\}$ `res = x.m(z̄)` $\{Q\}$, then for every program state σ that satisfies P, the execution of the method call `x.m(z̄)` results in a program state satisfying Q. As we have previously discussed (see §2.5), we build *Necessity* specifications on top of functional specifications using the fact that validity of $\{P\}\ \texttt{res} = x.m(\overline{z})\ \{Q\}$ implies that $\neg P$ is a necessary pre-condition to $\neg Q$ being true after execution of $\texttt{res} = x.m(\overline{z})$.

Proof rules for per-method specifications are given in Figure 5. Note that the receiver x in the rules in 5 is implicitly an internal object. This is because we only have access to internal code, and thus are only able to prove the validity of the associated Hoare triple.

If1-Classical states that if the execution of $x.m(\overline{z})$, with precondition $P \wedge \neg P_1$, leads to a state satisfying postcondition $\neg P_2$, then P_1 is a *necessary* precondition to the resulting state satisfying P_2.

If1-Inside states that if the precondition $\neg P$ guarantees that the result of the call $x.m(\overline{z})$ is not y, then P is a necessary pre-condition to invalidate $\texttt{inside}(y)$ by calling $x.m(\overline{z})$. This is sound, because the premise of If1-Inside implies that P is a necessary precondition for the call $x.m(\overline{z})$ to return an object y; this, in turn, implies that P is a necessary precondition for the call $x.m(\overline{z})$ to result in an external object gaining access to y. The latter implication is valid because the rule is applicable only to external states semantics, which means that the call $x.m(\overline{z})$ is a call from an external object to some internal object x. Namely, there are only four ways an object o might gain access to another object o': (1) o' is created by o as the result of a `new` expression, (2) o' is written to some field of o, (3) o' is passed to o as an argument to a method call on o, or (4) o' is returned to o as the result of a method call from an object o'' that has access to o'. The rule If1-Inside is only concerned with effects on program state resulting from a method call to some internal object, and thus (1) and (2) need not be considered as neither object creation or field writes may result in an external object gaining access to an object that is only internally accessible. Since we are only concerned with describing how internal objects grant access to external objects, our restriction on external method calls within internal code prohibits (3) from occuring. Finally, (4) is described by If1-Inside. In further work we plan to weaken the restriction on external method calls, and will strengthen this rule. Note that 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 functional specifications to reason about necessary conditions for invalidating $\texttt{inside}(\_)$.

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

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

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

$$\frac{\forall y,\ M \vdash \texttt{from}\ ([y/x]A_1)\ \texttt{next}\, A_2\ \texttt{onlyIf}\, A}{M \vdash \texttt{from}\ \exists x.[A_1]\ \texttt{next}\, A_2\ \texttt{onlyIf}\, A} \quad (\textsc{If1-}\exists_1)$$

Fig. 6. Selected rules for Single-Step *Only If*

4.3 Per-Step *Necessity* Specifications

The second key idea in §2.5 allows us to leverage several per-method *Necessity* specifications to obtain one per-step *Necessity* specification: Namely, if an assertion is encapsulated, and all methods within the internal module require the same condition to the invalidation of that assertion, then this condition is a necessary, program-wide, single-step condition to the invalidation of that assertion.

In this section we present a selection of the rules whose conclusion is of the form Single Step Only If in Fig. 6. The complete rule set can be found in the extended paper [Mackay et al. 2022b].

If1-Internal lifts a set of per-method *Necessity* specifications to a per-step *Necessity* specification. Any *Necessity* specification which is satisfied for all 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 present a few of the more interesting rules here:

The rule for implication (If1-$\longrightarrow$) may strengthen properties of either the starting or ending state, or weaken the necessary precondition. The disjunction elimination rule (IF1-$\vee$E) 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. The other rules can be found in the extended paper [Mackay et al. 2022b].

4.4 Emergent *Necessity* Specifications

The third key idea in §2.5 allows us to leverage several per-step *Necessity* specifications to obtain multiple-step *Necessity* specifications, and thus enables the description of the module's emergent behaviour. We combine per-step *Necessity* specifications into multiple-step *Necessity* specifications, as well as several multiple step *Necessity* specifications into further multiple step *Necessity* specifications.

Figure 7 presents some of the rules with conclusion *Only Through*, while Figure 8 provides some of the rules with conclusion *Only If*. The complete rules can be found in the appendices [Mackay et al. 2022b]

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

$$\frac{\begin{array}{c}M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_3\\ M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_3\ \texttt{onlyThrough}\ A\end{array}}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A}\ (\textsc{Trans}_1) \qquad \frac{\begin{array}{c}M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_3\\ M \vdash \texttt{from}\ A_3\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A\end{array}}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A}\ (\textsc{Trans}_2)$$

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

Fig. 7. Selected rules for *Only Through*

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

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

Fig. 8. Selected rules for *Only If*

Changes, in Figure 7, states that if A' is a necessary condition for the satisfaction of A to change in *one* step, then it is also a necessary condition for the satisfaction of A to change in *any number of* steps. This is sound, because 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 enabler for proofs of emergent properties: Since *Necessity* specifications are concerned with necessary conditions for change, their proofs typically hinge around such necessary conditions for certain properties to change. For example, under what conditions may our account's balance decrease?

It might seem natural that Changes had the more general form:

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

(ChangesUnsound) is not sound because the conclusion of the rule describes transitions from a state satisfying A_1 to one satisfying A_2 which may occur occur over several steps, while the premise describes a transition that takes place over one single step. Such a concern does not apply to (Changes), because a change in satisfaction for a specific assertion (*i.e.* A to $\neg A$) can *only* take place in a single step.

Trans$_1$ and Trans$_2$ are rules about transitivity. They state that necessary conditions to reach intermediate states or proceed from intermediate states are themselves necessary intermediate states. Any *Only If* specification entails the corresponding *Only Through* specification (If). Finally, End states that the ending condition is a necessary intermediate condition.

Only If also includes a rule for transitivity (If-Trans), but since the necessary condition must be true in the beginning state, there is only a single rule. If-Class expresses 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 that on top of these systems we built the Necessity logic according to the rules in Figures 5, 6, 7, and 8, 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 proof of Theorem 4.4 in Coq can be found in the associated artifact [Mackay et al. 2022a]. The Coq formalism deviates slightly from the system as presented here, mostly in the formalization of the *Assert* language. The Coq version of *Assert* restricts variable usage to expressions, and allows only addresses to be used as part of non-expression syntax. For example, in the Coq formalism we can write assertions like $x.f$ == `this` and $x == \alpha_y$ and $\langle\alpha_x$ `access` $\alpha_y\rangle$, but we cannot write assertions like $\langle x$ `access` $y\rangle$, where x and y are variables, and α_x and α_y are addresses. The reason for this restriction in the Coq formalism is to avoid spending significant effort encoding variable renaming and substitution, a well-known difficulty for languages such as Coq. This restriction does not affect the expressiveness of our Coq formalism: we are able to express assertions such as $\langle x$ `access` $y\rangle$, by using addresses and introducing equality expressions to connect variables to address, *i.e.* $\langle\alpha_x$ `access` $\alpha_y\rangle \wedge \alpha_x == x \wedge \alpha_y == y$. The Coq formalism makes use of the `CpdtTactics` [Chlipala 2019] library of tactics to discharge some proofs.

5 PROVING THAT $\texttt{MOD}_{\texttt{better}}$ SATISIFES $S_{\texttt{robust\_2}}$

We now revisit our example from §1 and §2, and outline a proof that $\texttt{Mod}_{\texttt{better}}$ satisfies $S_{\texttt{robust\_2}}$. A summary of this proof has already been discussed in §2.5. A more complex variant of this example can be found in the appendices [Mackay et al. 2022b]. It demonstrates dealing with modules consisting of several classes some of which are confined, and which use ghost fields defined through functions; it also demonstrates proofs of assertion encapsulation of assertions which involve reading the values of several fields. Mechanised versions of the proofs in both this Section, and in the appendices [Mackay et al. 2022b] can be found in the associated Coq artifact [Mackay et al. 2022a] in `simple_bank_account.v` and `bank_account.v` respectively.

Recall that an `Account` includes at least a field (or ghost field) called `balance`, and a method called `transfer`.

We first rephrase $S_{\texttt{robust\_2}}$ to use the `inside(_)` predicate.

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

We next revisit the functional specification from §2.1 and derive the following `PRE`- and `POST`-conditions. The first two pairs of `PRE`-, `POST`-conditions correspond to the first two `ENSURES` clauses from §2.1, while the next two pairs correspond to the `MODIFIES`-clause. The current expression in terms of `PRE`- and `POST`-conditions is weaker than the one in §2.1, and is not modular, but is sufficient for proving adherence to $S_{\texttt{robust\_2}}$.

```
FuncSpec′  ≜
   method transfer(dest:Account, pwd':Password) -> void
      (PRE: this.balance=bal1 ∧ dest.balance=bal2 ∧ this.pwd=pwd' ∧ this≠dest
       POST: this.balance=bal1-100 ∧ dest.balance=bal2+100)
      (PRE: this.balance=bal1 ∧ dest.balance=bal2 ∧ (this.pwd≠pwd' ∨ this=dest)
       POST: this.balance=bal1 ∧ dest.balance=bal2)
      (PRE: a:Account ∧ a.balance=bal ∧ a≠this ∧ a≠dest
       POST: a.balance=bal)
```

```
        (PRE: a:Account ∧ a.pwd=pwd1
         POST: a.pwd=pwd1)
```

5.1 Part 1: Assertion Encapsulation

The first part of the proof demonstrates that the `balance`, `pwd`, and external accessibility to the password are encapsulated properties. That is, for the `balance` to change (i.e. for `a.balance = bal` to be invalidated), or for the encapsulation of `a.pwd` to be broken (ie for a transition from `inside(a,pwd)` to ¬`inside(a.pwd)`), internal computation is required.

We use a simple encapsulation system, detailed in the appendices [Mackay et al. 2022b], and provide the proof steps below. **`aEnc`** and **`balanceEnc`** state that `a` and `a.balance` satisfy the $\textsc{Enc}_e$ predicate. That is, if any objects' contents are to be looked up during execution of these expressions, then these objects are internal. $\textsc{Enc}_e$(`a`) holds because no object's contents is looked up, while $\textsc{Enc}_e$(`a.balance`) holds because `balance` is a field of `a`, and `a` is internal.

`BalEncaps`:

`aEnc`:
$\texttt{Mod}_{\texttt{better}} \vdash \texttt{a:Account} \wedge \texttt{a.balance=bal} \Rightarrow \mathit{Enc}_e(\texttt{a})$ — by $\textsc{Enc}_e$-Obj

`balanceEnc`:
$\texttt{Mod}_{\texttt{better}} \vdash \texttt{a:Account} \wedge \texttt{a.balance=bal} \Rightarrow \mathit{Enc}_e(\texttt{a.balance})$ — by `aEnc` and Enc-Field

`balEnc`:
$\texttt{Mod}_{\texttt{better}} \vdash \texttt{a:Account} \wedge \texttt{a.balance=bal} \Rightarrow \mathit{Enc}_e(\texttt{bal})$ — by $\textsc{Enc}_e$-Int

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

Moreover, **`balEnc`** states that `bal` satisfies the $\textsc{Enc}_e$ predicate – it is an integer, and no object look-up is involved in its calculation. **`balanceEnc`** and **`balEnc`** combine to prove that the assertion `a.balance = bal` is encapsulated – only internal object lookups are involved in the validity of that assertion, and therefore only internal computation may cause it to be invalidated.

Using similar reasoning, we prove that `a.pwd` is encapsulated (**`PwdEncaps`**), and that `inside( a.pwd )` is encapsulated (**`PwdInsideEncaps`**).

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

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

5.2 Part 2: Per-Method *Necessity* Specifications

Part 2 proves necessary preconditions for each method in the module interface. We employ the rules from §4.2 which describe how to derive necessary preconditions from functional specifications.

`SetBalChange` uses a functional specification and a rule of consequence to prove that the `set` method in `Account` never modifies the `balance`. We then use If1-Classical and our *Necessity* logic to prove that if it ever did change (a logical absurdity), then `transfer` must have been called.

SetBalChange:

{a, a′:Account ∧ a′.balance=bal}
  a.set(_, _)
  {a′.balance = bal}
	by functional specification

{a, a′:Account ∧ a′.balance = bal ∧ ¬ false}
  a.set(_, _)
  {¬ a′.balance < bal }
	by rule of consequence

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

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

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

SetPwdLeak:

{a:Account ∧ a′:Account ∧ a.pwd == p}
  res=a′.set(_, _)
  {res != pwd}
	by functional specification

{a:Account ∧ a′:Account ∧ a.pwd == p ∧ ¬ false}
  res=a′.set(_, _)
  {res != p}
	by rule of consequence

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

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

SetPwdChange:

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

TransferBalChange:

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

TransferPwdLeak:

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

TransferPwdChange:

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

5.3 Part 3: Per-Step *Necessity* Specifications

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

BalanceChange:

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

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

PasswordChange:

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

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

PasswordLeak:

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

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

5.4 Part 4: Emergent *Necessity* Specifications

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

(1) If the balance of an account decreases, then by `BalanceChange` there must have been a call to `transfer` in `Account` with the correct password.

(2) If there was a call where the `Account`'s password was used, then there must have been an intermediate program state when some external object had access to the password.

(3) Either that password was the same password as in the starting program state, or it was different:

- **(Case A)** If it is the same as the initial password, then since by `PasswordLeak` it is impossible to leak the password, it follows that some external object must have had access to the password initially.
- **(Case B)** If the password is different from the initial password, then there must have been an intermediate program state when it changed. By `PasswordChange` we know that this must have occurred by a call to `set` with the correct password. Thus, there must be a some intermediate program state where the initial password is known. From here we proceed by the same reasoning as **(Case A)**.

6 RELATED WORK

Program specification and verification has a long and proud history [Hatcliff et al. 2012; Hoare 1969; Leavens et al. 2007; 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.

Open systems, by definition, must interact with untrusted code: they cannot rely on callers' obeying method preconditions. [Miller 2006; Miller et al. 2013] define the necessary approach as *defensive consistency*: *"An object is defensively consistent when it can defend its own invariants and provide correct service to its well behaved clients, despite arbitrary or malicious misbehaviour by its other clients."* [Murray 2010] made the first attempt to formalise defensive consistency and correctness in a programming language context. Murray's model was rooted in counterfactual causation [Lewis 1973]: an object is defensively consistent when the addition of untrustworthy clients cannot cause well-behaved clients to be given incorrect service. Murray formalised defensive consistency abstractly, without a specification language for describing effects.

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

$S_{\texttt{robust\_2}}$:

from a:Account ∧ a.balance=bal to a.balance < bal onlyThrough ⟨_ calls a.transfer(_,a.pwd)⟩	by CHANGES and BalanceChange
from a:Account ∧ a.balance=bal to b.balance(a) < bal onlyThrough ¬inside(a.pwd)	by $\longrightarrow$, CALLER-EXT, and CALLS-ARGS
from a:Account ∧ a.balance=bal ∧ a.pwd=p to a.balance < bal onlyThrough ¬inside(a.pwd) ∧ (a.pwd=p ∨ a.pwd != p)	by $\longrightarrow$ and EXCLUDED MIDDLE
from a:Account ∧ a.balance=bal ∧ a.pwd=p to a.balance < bal onlyThrough (¬inside(a.pwd) ∧ a.pwd=p) ∨ (¬inside(a.pwd) ∧ a.pwd != p)	by $\longrightarrow$
from a:Account ∧ a.balance=bal ∧ a.pwd=p to a.balance < bal onlyThrough ¬inside(p) ∨ a.pwd != p	by $\longrightarrow$

Case A (¬inside(p)):

from a:Account ∧ a.balance=bal ∧ a.pwd=p to ¬inside(p) onlyIf inside(p) ∨ ¬inside(p)	by IF-$\longrightarrow$ and EXCLUDED MIDDLE
from a:Account ∧ b:Bank ∧ b.balance(a)=bal ∧ a.password=pwd to ¬inside(p) onlyIf ¬inside(p)	by ∨E and PasswordLeak

Case B (a.pwd != p):

from a:Account ∧ b:Bank ∧ b.balance(a)=bal ∧ a.password=pwd to a.pwd != p onlyThrough ⟨_ calls a.set(p,_)⟩	by CHANGES and PASSWORD-CHANGE
from a:Account ∧ a.balance=bal ∧ a.pwd=p to a.pwd != p onlyThrough ¬inside(p)	by ∨E and PasswordLeak
from a:Account ∧ a.balance=bal ∧ a.pwd=p to a.pwd != p onlyIf ¬inside(p)	by **Case A** and TRANS

from a:Account ∧ a.balance=bal ∧ a.pwd=p to b.balance(a) < bal onlyIf ¬inside(p)	by **Case A**, **Case B**, IF-∨I$_2$, and IF-$\longrightarrow$

hostile opponent" [Bugliesi et al. 2011; Gordon and Jeffrey 2001]. More recent work has applied robust safety in the context of programing languages. For example, [Swasey et al. 2017] present a logic for object capability patterns, drawing on verification techniques for security and information flow. They prove a robust safety property that ensures interface objects ("low values") are safe to share with untrusted code, in the sense that untrusted code cannot use them to break any internal invariants of the encapsulated object. Similarly, [Schaefer et al. 2018] have added support for information-flow security using refinement to ensure correctness (in this case confidentiality) by construction. Concerns like $S_{\texttt{robust\_2}}$ are not, we argue, within the scope of these works.

[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) and a mashup application.

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 small extensions to 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: $(\forall x.[\langle x\ \texttt{access}\ o\rangle \ \Rightarrow\ \texttt{rep}(x) \prec: \texttt{owner}(o)])$.

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.

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, 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.

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

$$\texttt{a:Account} \wedge \texttt{a.balance==bal} \wedge \langle \texttt{next}\ \texttt{a.balance<bal}\rangle \\ \longrightarrow \exists \texttt{o}, \texttt{a'}.\langle \texttt{o}\ \texttt{calls}\ \texttt{a.transfer(a',a.password)}\rangle$$

However, to express $S_{\texttt{robust\_2}}$, one also needs capability operators which talk about provenance and permission. VerX does not support capability operators, and thus cannot express $S_{\texttt{robust\_2}}$, while *Chainmail* does support capability operators, and can express $S_{\texttt{robust\_2}}$.

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

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*.

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*, in particular a reference to x could always be guessed thus the assertion `inside`(x) would no longer be encapsulated.

Callbacks. Necessity does not –yet– support calls of external methods from within internal modules. While this is a limitation, it is common in the related literature. For example, VerX [Permenev et al. 2020] work on effectively call-back free contracts, while [Grossman et al. 2017] and [Albert et al. 2020] drastically restrict the effect of a callback on a contract. In further work we are planning to incorporate callbacks by splitting internal methods at the point where a call to an external method appears. This would be an adaptation of Bräm et al.'s approach, who split methods into the call-free subparts, and use the transitive closure of the effects of all functions from a module to overapproximate the effect of an external call. One useful simplification was proposed by Permenev et al. [2020]: in "*effectively callback free*" methods, meaning that we could include callbacks while also only requiring at most one functional specification per-method.

7 CONCLUSION

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

We have developed a proof system to prove that modules meet their specifications. Our proof system exploits the pre and postconditions of functional 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.

We have proved our system sound, and used it to prove a bank account example correct: the Coq mechanisation is detailed in the appendices [Mackay et al. 2022b] and available as an artifact.

In future work we want to consider more than one external module – c.f. §2.4, and expand a Hoare logic so as to make use of *Necessity* specifications, and reason about calls into unknown code - c.f. §2.3.1. We want to work on supporting callbacks. We want to develop a logic for encapsulation rather than rely on a type system. Finally we want to develop logics about reasoning about risk and trust [Drossopoulou et al. 2015].

ACKNOWLEDGMENTS

We are especially grateful for the careful attention and judicious suggestions of the anonymous reviewers, which have significantly improved the paper. We are deeply grateful for feedback from and discussions with Chris Hawblitzel, Dominiqie Devriese, Derek Dreyer, Mark Harman, Lindsay Groves, Michael Jackson, Bart Jacobs from KU Leuven, Gary Leavens, Mark Miller, Peter Mueller, Toby Murray, Matthew Ross Rachar, Alexander J. Summers, Shriram Krishnamurthi, and members of the WG2.3. This work is supported in part by the Royal Society of New Zealand (Te Apārangi) Marsden Fund (Te Pūtea Rangahau a Marsden) under grant VUW1815 (www.royalsociety.org.nz/what-we-do/funds-and-opportunities/marsden/awarded-grants/marsden-fund-highlights/2018-marsden-fund-highlights/an-immune-system-for-software), and by gifts from the Ethereum Foundation, Meta, and Agoric.

REFERENCES

Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020. Taming Callbacks for Smart Contract Modularity. OOPSLA (2020). https://doi.org/10.1145/3428277

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

Anindya Banerjee and David A. Naumann. 2005a. Ownership Confinement Ensures Representation Independence for Object-oriented Programs. *J. ACM* 52 (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)*.

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

C. Bräm, M. Eilers, P. Müller, R. Sierra, and A. J. Summers. 2021. Rich Specifications for Ethereum Smart Contract Verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/3485523

Torben Braüner. 2022. Hybrid Logic. In *The Stanford Encyclopedia of Philosophy* (Spring 2022 ed.), Edward N. Zalta (Ed.).

James Brotherston, Diana Costa, Aquinas Hobor, and John Wickerson. 2020. Reasoning over Permissions Regions in Concurrent Separation Logic. In *Computer Aided Verification*.

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 *CSF'11*. 83–98.

Adam Chlipala. 2019. Certified Programming with Dependent Types. http://adam.chlipala.net/cpdt/

David Clarke and Sophia Drossopoulou. 2002. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*.

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 *ASPLOS*. ACM, 379–393.

Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). 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 *FASE*. 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*.

A.D. Gordon and A. Jeffrey. 2001. 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

Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. POPL (2017). https://doi.org/10.1145/3158136

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.

Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Pearson.

G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. 2007. 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 Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP*.

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).

Julian Mackay, Sophia Drossopoulou, James Noble, and Eisenbach. 2022a. Necessity Specifications for Robustness. Zenodo. https://doi.org/10.5281/zenodo.7084291

Julian Mackay, Sophia Drossopoulou, James Noble, and Susan Eisenbach. 2022b. Necessity Specifications for Robustness and Appendices. (Sep 2022). https://doi.org/10.5281/zenodo.7087932

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.).

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. POPL (2019). https://doi.org/10.1145/3371078

Marco Patrignani and Deepak Garg. 2021. Robustly Safe Compilation, an Efficient Form of Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 1 (Feb. 2021). https://doi.org/10.1145/3436809

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. 4, POPL (2019). 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. 502–515. https://doi.org/10.1007/978-3-030-03418-4_30

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*.

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 *ESOP (ESOP '01)*. 46–61.