Necessity Specifications are Necessary

ANONYMOUS AUTHOR(S)

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

1 INTRODUCTION

Condition B is hard to formalize,
since it requires saying precisely what a bad plan is,
and we do not attempt to do so.
[Lamport et al. 1982]

The days of single, monolithic programs are long gone. Contemporary software is built over decades, by combining modules and components of different provenance and different degrees of trustworthiness, which can interact with almost every other program, device, or person. In order for the resulting complex system to be correct, we need to be able to reason about individual components to ensure that they behave correctly, i.e. that good things happen when our programs run. For example: if I send an email to a valid address, it will be delivered to its recipient, or if I provide the right password, I can transfer money from one of my bank accounts to another. To prove that good things can happen, program verification systems can use witnesses, e.g. a precondition, a postcondition describing the good thing (the desired effect), and a code snippet, whose execution will establish the effect, given the precondition. The critical point here is that the precondition is a *sufficient* condition for the code snippet to make the good thing happen: given the precondition, executing a correct code snippet is guaranteed to achieve the postcondition.

Unfortunately, in a system of any complexity, knowing that good things will happen is not enough: we also need to be sure that bad things cannot happen. For example: we need to be sure that an email can only be read by the intended recipient; or that someone can only transfer money if they provide the account's password. To address this problem, we need to consider the *necessary* conditions without which some postcondition (good or bad) cannot be achieved: it is necessary that someone is sent an email before they can read it; it is necessary that the correct password is provided before money can be transferred.

The challenge here is twofold: How do we specify the bad things we are concerned about, and how do we prove that the bad things we've specified do not happen? These challenges are difficult because we cannot refer to just one component of a software system. A sufficient specification can deal with a single component in isolation — a single function for pre- and postconditions; a single class or data structure for invariants. A necessary specification, however, must provide guarantees which encompass the software system in its entirety, and constrain the emergent behaviour of all its components, for an open system, all possible sequences of API invocations.

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

The importance of distinguishing between sufficient and necessary specifications of various kinds has a long history in Computer Science. Lamport distinguishes between "good things will eventually happen" (liveness) and "bad things will never happen" (safety) conditions in his seminal paper on "Multiprocess Programs". Type systems ensure entire classes of bad things can't happen, preserving executing even if memory structures are greatly corrupted [Rinard 2003]. More recently, Swasey et al. and Sammler et al. with their robust safety and Drossopoulou et al. with their holistic systems have tackled open world systems to prevent bad things from happening from untrusted code. Swasey et al. use techniques from security to ensure their isn't undesirable leakage, Sammler et al. build a sandbox and have a sophisticated type system to protect it and Drossopoulou et al. have necessary conditions, which they expressed through temporal operators.

In this paper we introduce *necessity specifications*:

`from` A_1 `to` A_2 `onlyIf` A_3

---

to say a change from a current state satisfying A_1 to a future state satisfying A_2 (i.e. an effect) is possible only if the necessary condition A_3 holds in the current state (i.e. $(A_1 \wedge \Diamond A_2) \longrightarrow A_3$). Unlike Permenev et al. or Drossopoulou et al. which employ general temporal operators, we support necessity specifications with this explicit "`from` A_1 `to` A_2 `onlyIf` A_3" syntax and concomitant specialised inference system. Our assertions A support the usual expressions about program state (e.g. `x.f > 3`), logical connectives and quantifiers (e.g. $\wedge$, $\forall$), and additional predicates to capture *provenance* (whether an object o's definition is $\langle$`o internal`$\rangle$ or $\langle$`o external`$\rangle$) to the current module, and *permission* [Miller et al. 2013] (whether an object o has direct access to another object o': $\langle$`o access o'`$\rangle$).

1.1 Bank Account Example

Consider `Account` class in module `Mod1` that represents a bank account with a balance and a password, where funds may be transferred between accounts only when sending account's password:

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

We can capture the intended semantics of the `transfer` method by writing "classical" specifications in terms of pre- and post-conditions — `Mod1`'s implementation of the `transfer` method meets this specification.

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

Now consider the following alternative implementations `Mod2` allows any client to reset its password at any time, while `Mod3` first requires the existing password in order to change it. The problem is that although the `transfer` method is the same in all three alternatives, and although each one satisfies (ClassicSpec), emergent code sequences such as `account.set(42);`

`account.transfer(yourAccount, 42)` are enough to drain the account `Mod2` without supplying the password.

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

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

We need to rule out `Mod2` while permitting `Mod3` and `Mod1`. The catch is the leak in `Mod2` is the result of *emergent* behaviour from the interactions of the `set` and `transfer` methods — even though `Mod3` also has a `set` method, it does not exhibit the unwanted interaction. This is exactly where a necessity condition can help: we want to avoid transferring money (or more generally, reducing an account's balance) *without* the account password. Phrasing the same condition the other way around gives us a positive statement that still rules out the theft: that money *can only* be transferred when the account's password is supplied. In our necessity condition syntax:

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

The critical point of this necessity specification is that it is expressed in terms of observable effects — reducing the account balance — and not in terms of individual methods — such as `set` or `transfer`. This gives necessity specifications the vital advantage that they constrain not only any *implementation* of a bank account with a balance and a password (as do traditional sufficient specifications) but also any *design* of such a bank account, irrespective of the API it offers, the services it exports, or the dependencies on other parts of the system.

1.2 Reasoning about Necessity

Our work concentrates on guarantees made for the *open* setting; that is, a certain module M is programmed in such a robust manner 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. We therefore distinguish between *internal* and *external* objects: those that belong to classes defined in M, and the rest. Similarly, we distinguish between *internal* calls, i.e. calls made to internal objects, and *external* calls, i.e. calls made to external objects.

Moreover, because we only require that the guarantees are upheld while the external module is executing, we develop an *external state* semantics, where any internal calls are executed in one, large, step. Note that we do not – yet – support calls from internal objects to external objects. With the external steps semantics, the executing object (the `this`) is always external.

We now outline the proof that of `Mod3` adheres to (*NecessitySpec*), and will then use that outline to introduce the main ingredients of our Necessity Logic. The proof consists, broadly, of the following parts:

P1: We establish that the balance may change only through `Account`-internal calls.

P2: For each method of the class `Account`, we establish that if the method were called and caused the balance to reduce, then, before the call, the caller had access to the password.

P3: From **P1** and **P2**, we obtain that if the balance were to reduce in *any single* call, then some external object would have to have had access to the password before the call.

P4: We establish that an external object has access to the password after *one* internal call, only if it already had access before that call. From that, we establish that an external object will have access to the password after *any number* of external or internal steps, only if it already had access before these steps.

P5: Combining the results from **P3** and **P4**, we obtain that Version_III satisfies (*NecessitySpec*)

We now outline the new formal concepts needed to accomplish the five parts of the proof from above:

Assertion encapsulation An assertion A is *encapsulated* by a module M, if A can be invalidated only through an M-internal call. In short, an M-internal call is a *necessary* condition for a given effect to take place. In **P1**, the assertion `a:Account` $\wedge$ `a.balance=val` is encapsulated by `Account`.

We assume that there exists some algorithm to judge assertion encapsulation. The construction of such an algorithm is not the focus of our work; we outline a rudimentary such algorithm, but more powerful approaches are possible.

Per-method necessity specification We want to infer a necessary condition given an effect and a single, specified, method call.

In **P2**, a necessary condition for the reduction of `a.balance` after the call `a.transfer(a',pwd)` is that the caller had access to `a.password` before the call.

We addressed the challenge of the inference of necessary conditions by leveraging the sufficient conditions from classical specifications: As we will see in Section 3.2, if the negation of a method's classical postcondition implies the effect we are interested in, then the negation of the classical precondition is the necessary precondition for the effect and the method call. Thus, a method's sufficient conditions are used to infer a method's and effect's necessary conditions.

Single step necessity specification We want to infer a necessary condition given an effect and a single, unspecified step. This step could be a internal, or an external method call, or any other external step, such as field assignment, or conditional, etc.

In **P3**, a necessary condition for the reduction of `a.balance` after *any* step, is that the caller had access to `a.password` before the call. And similarly in **P4**, a necessary condition for an external object's access to `a.password` after *any* step, is that that object had access to `a.password` before the call.

For effects that are encapsulated in a module M, we can infer such one-step necessary conditions by combining the necessary conditions for that effect and all methods in M.

Necessity specifications of emergent behaviour We now need to consider the *emergent* behaviour of the internal module combined with any internal module. This step is crucial; namely, remember that while Version_II adheres to the guarantee from the **P3**, it does not adhere to (*NecessitySpec*). Our Logic of Necessity allows us to combine several such one-step necessary conditions, and obtain a several-step necessary condition.

1.3 Contributions

The contributions of this paper are as follows:

(1) *SpecX*, a language to express necessity specifications (§2);
(2) a logic for proving a module's adherence to its necessity specifications (§3);
(3) a mechanised proof of soundness of the logic (§3.5);
(4) a proof that the bank account example obeys the necessity specification (§4).

Finally we place necessity specifications into the context of related work (§5) and conclude (§6). The Coq proof of the logic appears in the supplementary material.

2 THE MEANING OF NECESSITY

In the introduction we spoke of *Necessity Specifications*, e.g., `from` A_1 `to` A_2 `onlyIf` A_3. In this section we will define the semantics of such specifications. In order to do that, we first define an underlying programming language *LangX*, (Section 2.1). We then define an assertion language, *SpecW*, which can talk about the contents of the state, as well about of provenance and permission (Section 2.2). Finally, we define necessity specifications in *SpecX* (Section 2.3).

2.1 *LangX*

LangX is a small, simple, imperative, class based, object oriented language. Given the simplicity of *LangX*, we do not define it here, instead, we direct the reader to Appendix A for the full definitions. Here we outline the definitions, and introduce the concepts most relevant to the treatment of the open world guarantees.

A *LangX* configuration σ consists of a heap χ, and a frame stack ψ – the latter 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.

Execution is performed in the context of a module M, which is a mapping from class names to class definitions. Execution has the format $M, \sigma \rightsquigarrow \sigma'$, and is unsurprising, c.f. Appendix A. The statements being executed are those in the continuation of the top frame.

As we said in section 1.2, we are interested in guarantees which hold when the external module is executing, and are not concerned if the internal module temporarily breaks them. Therefore, we are only interested in states where the executing object (the `this`) is an internal object. For this, we define the *external state semantics*, of the form $M_1; M_2,\ \sigma \rightsquigarrow \sigma'$, where M_1 is the external module, and M_2 is the internal module, and where we collapse all internal steps into one single step.

Definition 2.1 (External State Semantics). For modules M_1, M_2, and program configurations σ, σ', we say that $M_1; M_2,\ \sigma \rightsquigarrow \sigma'$ if and only if there exists a $n \in \mathbb{N}$, such that

- $\sigma{=}\sigma_1$, and $\sigma'{=}\sigma_n$,
- $M_1 \circ M_2, \sigma_i \rightsquigarrow \sigma_{i+1}$ for all $i \in [0..n)$,
- $classOf(\sigma, \sigma.\texttt{this}), classOf(\sigma', \sigma'.\texttt{this}) \in M_2$,
- $classOf(\sigma_i, \sigma_i.\texttt{this}) \in M_1$ for all $i \in [1..n)$.

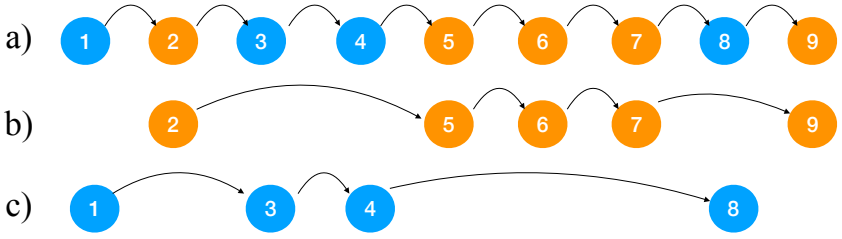

Fig. 1. External State Semantics (Def. 2.1). (a) $M_1 \circ M_2, \sigma_1 \rightsquigarrow \ldots \rightsquigarrow \sigma_9$ (b) $M_1; M_2,\ \sigma_2 \rightsquigarrow \ldots \rightsquigarrow \sigma_9$ (c) $M_2; M_1,\ \sigma_1 \rightsquigarrow \ldots \rightsquigarrow \sigma_8$

Fig. 1 provides a simple graphical description of our external state semantics, showing how given a single execution comprised of code from two different modules, we can consider either the complete execution as in (a), the external state execution when M_2 is external (b), and the external state execution when M_1 is external in (c).

In Definition 2.1 we use the function $classOf(\sigma, \alpha)$. This function looks up the class of the address α in the heap of σ. We also use module linking, $M_1 \circ M_2$. The operator $\circ$ combines the two modules

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

Fig. 2. *SpecW* Assertions

into one module in the obvious way, provided that their domains are disjoint. Full details in Appendix A.

In this work we are interested in guarantees which are upheld by the internal module. Moreover, these guarantees need to be satisfied only at 'reachable' states (configurations), and need not be satisfied at states that are not reachable – this is described formally in Definition 2.2. Reachable states are those that may arise by external states execution of a given internal module linked with *any* external module We describe the states of interest as the *arising configurations*.

Definition 2.2 (Arising Program Configuration). For modules M_1 and M_2, a program configuration σ is said to be an *arising* configuration, formally $\mathit{Arising}(M_1, M_2, \sigma)$, if and only if there exists some σ_0 such that $\mathit{Initial}(\sigma_0)$ and $M_1; M_2,\ \sigma_0 \rightsquigarrow^* \sigma$.

In the definition above we used *Initial* configurations, which characterise configurations at the start of program execution. The heap of an initial configuration should contain a single object of class `Object`, and the stack should consist of a single frame, whose local variable map contains only the mapping of `this` to the single object, and whose continuation may be any statement. More in Definition A.5.

2.2 *SpecW*

SpecW extends the expressiveness of standard specification languages with assertion forms capturing key concepts of software security: *permission*, *provenance*, and *control*.

2.2.1 Syntax. Fig. 2 gives the assertion syntax of the *SpecW* specification language. An assertion may be an expression, a class assertion, the usual connectives and quantifiers, along with the following non-standard assertion forms, inspired from the capabilities literature:

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

2.2.2 Semantics of SpecW. The semantics of *SpecW* assertions is given in Definition 2.3. The definition of the semantics of *SpecW* makes use of several language features of *LangX* that can be found in Appendix A. Specifically, $M, \sigma, e \hookrightarrow v$ is the evaluation relation for expressions, and is interpreted as expression e evaluates to value v in the context of program configuration σ, with module M. The full semantics of expression evaluation are given in Fig. 12. It should be noted that expressions in *LangX* may be recursively defined, and thus evaluation may not necessarily terminate, however the logic remains classical because recursion is restricted to expressions, and not generally to assertions.

Further, Definition 2.3 uses the interpretation of variables within a specific frame or configuration: i.e. $\lfloor x \rfloor_\phi = v$, meaning that x maps to value v in the local variable map of frame ϕ, and $\lfloor x \rfloor_\sigma = v$ meaning x maps to value v in the top most frame of σ's stack. And the term $\lfloor x.f \rfloor_\sigma = v$ has the obvious meaning.

Definition 2.3 (Satisfaction of SpecW Assertions). We define satisfaction of an assertion A by a program configuration σ with internal module M as:

$$H \quad ::= \quad \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A_3 \ \mid\ \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A_3 \ \mid\ \texttt{from}\ A_1\ \texttt{to1}\ A_2\ \texttt{onlyIf}\ A_3$$

Fig. 3. Syntax of *SpecX*

- $M, \sigma \vDash e$ iff $M, \sigma, e \hookrightarrow \texttt{true}$
- $M, \sigma \vDash e : C$ iff $M, \sigma, e \hookrightarrow \alpha$ *and* $\sigma.\texttt{heap}(\alpha).\texttt{class} = C$
- $M, \sigma \vDash \neg A$ iff $M, \sigma \nvDash A$
- $M, \sigma \vDash A_1 \ \wedge\ A_2$ iff $M, \sigma \vDash A_1$ and $M, \sigma \vDash A_2$
- $M, \sigma \vDash A_1 \ \vee\ A_2$ iff $M, \sigma \vDash A_1$ or $M, \sigma \vDash A_2$
- $M, \sigma \vDash \forall x.[A]$ iff $M, \sigma[x \mapsto \alpha] \vDash A$, for some x fresh in σ, and for all $\alpha \in \sigma.\texttt{heap}$.
- $M, \sigma \vDash \exists\, x.[A]$ iff $M, \sigma[x \mapsto \alpha] \vDash A$, for some x fresh in σ, and for some $\alpha \in \sigma.\texttt{heap}$.
- $M, \sigma \vDash \langle x\ \texttt{access}\ y \rangle$ iff
  - $\lfloor x.f \rfloor_\sigma = \lfloor y \rfloor_\sigma$ for some f, or
  - there exists some z, and some frame ϕ in the stack of σ such that $\lfloor x \rfloor_\phi = \lfloor \texttt{this} \rfloor_\phi$
- $M, \sigma \vDash \langle x\ \texttt{internal} \rangle$ iff $\mathit{classOf}(\sigma, x) \in M$
- $M, \sigma \vDash \langle x\ \texttt{external} \rangle$ iff $\mathit{classOf}(\sigma, x) \notin M$
- $M, \sigma \vDash \langle x\ \texttt{calls}\ y.m(z_1, \ldots, z_n) \rangle$ iff
  - $\sigma.\texttt{contn} = (\_ := y'.m(z'_1, \ldots, z'_n))$,
  - $\lfloor x \rfloor_\sigma = \lfloor \texttt{this} \rfloor_\sigma$
  - $\lfloor y \rfloor_\sigma = \lfloor y' \rfloor_\sigma$
  - $\lfloor z_i \rfloor_\sigma = \lfloor z'_i \rfloor_\sigma$ for all $1 \leq i \leq n$

Finally, we define what it means for a module to satisfy an assertion: a module M satisfies an assertion A, if all configurations σ arising from external steps execution of that module with any other external module, satisfy A.

Definition 2.4 (Assertion Satisfaction by Modules). For a module M and assertion A, we say that $M \vDash A$ if and only if for all modules M', and all σ, if $\mathit{Arising}(M', M, \sigma)$, then $M, \sigma \vDash A$.

A proof system for such assertions is indicated by a judgement of the form $M \models A$. We will not define such a judgment, and just rely on its existence (cf. Theorem 3.2). We define soudnness of such a judgment in the usual way:

Definition 2.5 (Soundness of SpecW Provability). A judgment of the form $M \vdash A$ is *sound*, if for all all modules M and assertions A, if $M \ \vdash\ A$ then $M \vDash A$.

2.2.3 Wrapping. We define a useful shorthand: `wrapped()` predicate `t` states that only internal objects have access to some object. That object may be either internal or external.

Definition 2.6 (Wrapped). $\texttt{wrapped}(o) \ \triangleq\ \forall x.[\neg\langle x\ \texttt{access}\ o \rangle \ \vee\ \langle x\ \texttt{internal} \rangle]$

Wrapped is critical as it captures the conditions under which reading or writing involving an object necessitates an interaction with the internal module. If for example, only internal objects have access to an account's password, then it follows that access to the password may not be gained except by an interaction with the internal module, and subsequently if the internal module is secure we know that the password may not be leaked.

2.3 *SpecX*– Necessity Specifications

In this Section we define syntactic forms and semantics of *Necessity Specifications*. Fig. 3 gives the syntax. We have three forms of Necessity Specifications, described below:

Only If. [`from` A_1 `to` A_2 `onlyIf` A]: If an arising program configuration starts at some state A_1, and reaches some state A_2, then the original program state must have also satisfied A. e.g. if the balance of a bank account changes over time, then there must be some external object in the current program state that has access to the account's password.

Single-Step Only If. [`from` A_1 `to1` A_2 `onlyIf` A]: If an arising program configuration starts at some state A_1, and reaches some state A_2 after a single execution step, then the original program state must have also satisfied A. e.g. if the balance of a bank account changes over a single execution step, then that execution step must be a method call to the bank `transfer` method.

Only Through. [`from` A_1 `to` A_2 `onlyThough` A]: If an arising program configuration starts at some A_1 state, and reaches some A_2 state, then program execution must have passed through some A state. e.g. if the balance of an account changes over time, then the bank's `transfer` method must have been called in some intermediate state. Note that the intermediate state where A is true might be the initial state (σ_1), or final state (σ_2).

All three Necessity Specifications from above talk about assertions satisfied in the current configuration as well as of assertions satisfied in some future configuration. These assertions may contain variables, whose denotation might change during program execution: the map may change, variables may be overwritten, or the entire local variable maps may be lost on a method return. For this reason, before we provide the semantics of Necessity Specifications, we first introduce an adaptation operator to account for variable renaming throughout the execution of a program.

Definition 2.7. $\sigma \triangleleft \sigma' \triangleq (\chi, \{\texttt{local} := \beta'[\overline{z'} \mapsto \beta(\overline{z})], \texttt{contn} := [\overline{z}/\overline{z'}]c\} : \psi)$ where

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

We can now define, $M \models H$, the semantics of the Necessity Specifications in Definition 2.8. The definition goes by cases over the three possible syntactic forms of H:

Definition 2.8 (Necessity Specifications). For any assertions A_1, A_2, and A, we define

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

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

- $M \vDash$ `from` A_1 `to1` A_2 `onlyIf` A iff for all M', σ, σ', such that $Arising(M, M', \sigma)$:

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

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

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

With the necessity specifications as defined in Definition 2.8, we are able to state what are the necessary preconditions to critical functions in software, including safety properties of software in the open world. The semantics of *Single-Step Only If* allow for the statement of such necessary preconditions for any execution step for any program to achieve a certain outcome. The semantics

of *Only If* and *Only Through* allow us to raise these necessary preconditions to any arbitrary number of execution steps, and thus allow for reasoning about the execution of an entire program, even those programs in the open world, where client functions are unknown and unspecified in a traditional manner.

Looking back at the example from the Introduction, it holds that

$$\texttt{Mod1} \models \texttt{NecessityBankSpec}$$
$$\texttt{Mod2} \not\models \texttt{NecessityBankSpec}$$
$$\texttt{Mod3} \models \texttt{NecessityBankSpec}$$

As further examples of Necessity Specifications, consider the original bank account example discussed in Section 1, we have already shown how we can reason about knowledge of an account's password using `NecessityBankSpec`, but we are also able to write other useful properties about the bank account.

```
NecessityBankSpec'  ≜  from a:Account ∧ a.balance==bal
                       to1 a.balance < bal
                       onlyIf ∃ o.[⟨o external⟩ ∧ ⟨o calls a.transfer(_, _, _)⟩]
```

`NecessityBankSpec'` states that if over a single step the balance of an account decreases, then it must have occurred as a result of a call to `transfer`.

```
NecessityBankSpec''  ≜  from a:Account ∧ a.password == pwd
                        to a.password != pwd
                        onlyThrough ∃ o.[⟨o external⟩ ∧ ⟨o calls a.set(pwd, _)⟩]
```

`NecessityBankSpec''` states that if over an arbitrary number of execution steps, the password of an account changes, then it follows that there must have been some intervening execution step that was a call to `set` on the account with the correct password. Both of these specifications are important, and are both used as intermediate steps when we present the full proof of `NecessityBankSpec` later in Section 4.

Necessity Specifications thus provide us with a rich language for talking about the necessary conditions under which critical actions within of our software are allowed to occur.

2.4 Encapsulation

In the introduction we used the concept of encapsulation of *SpecW* assertions when proving adherence to *SpecW* Necessity Specifications. An assertion A is encapsulated by a module M if it cannot be invalidated unless an internal method is called. Here we refine this concept, to allow for "conditional" encapsulation: $M \vDash A \implies Enc(A')$ expresses that in states which satisfy A, the assertion A' cannit be invalidated, unless a method from M was called.

Definition 2.9 (Assertion Encapsulation). For a module M and assertion A, we define an assertion A' as being encapsulated, written $M \vDash A \implies Enc(A')$, if and only if for all external modules M', and program configurations σ and σ' such that

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

2.5 Expressiveness of Necessity Specifications

Both the object capability literature and more recently the smart contract literature are rich sources for exemplars to try *SpecX* on. Ensuring that a visited web page cannot leak your confidential data was looked at by Devriese et al.. The high profile, high money losses in the smart contract world have made ensuring that smart contracts cannot be attacked from outside a priority. It is interesting

to see how straightforward it is to state assertions that protect the exemplars from attacks that have caused grief in the past.

2.5.1 DOM. The key structure underlying a web browser is the Domain Object Model (DOM), a recursive composite tree structure of objects that represent everything display in a browser window. Each window has a single DOM tree which includes both the page's main content and also third party content such as advertisements. To ensure third party content cannot affect a page's main content, specifications for attenuation for the DOM were proposed in *Devriese et al:* [Devriese et al. 2016].

This example deals with a tree of DOM nodes: Access to a DOM node gives access to all its parent and children nodes, and the ability to modify the node's properties. However, as the top nodes of the tree usually contain privileged information, while the lower nodes contain less crucial third-party information, we want to be able to limit access given to third parties to only the lower part of the DOM tree. We do this through a `Wrapper`, which has a field `node` pointing to a `Node`, and a field `height` which restricts the range of `Nodes` which may be modified through the use of the particular `Wrapper`. Namely, when you hold a `Wrapper` you can modify the `property` of all the descendants of the `height`-th ancestors of the `node` of that particular `Wrapper`.

`DOMSpec` states that if the property of a node in a DOM tree changes, it follows that either some non-node, non-wrapper object presently has access to a node of the DOM tree, or to some wrapper with access to some ancestor of the node that was modified.

```
DOMSpec ≜ from nd : Node ∧ nd.property = p
           to nd.property != p
           onlyIf ∃ o.[¬ o : Node ∧ ¬ o : Wrapper ∧
                       (∃ nd' : Node.[⟨o access nd'⟩] ∨
                        ∃ w : Wrapper, k : ℕ.[⟨o access w⟩ ∧ nd.parnt^k = w.node.parnt^w.height] )]
```

2.5.2 ERC20. The ERC20[The Ethereum Wiki 2018] is a widely used token standard describing the basic functionality of any Ethereum-based token contract. This functionality includes issuing tokens, keeping track of tokens belonging to participants, and the transfer of tokens between participants. Tokens may only be transferred if there are sufficient tokens in the participant's account, and if either they or someone authorized the participant initiated the transfer.

We specify these necessary conditions here using *SpecX* and the Logic of Necessity. Firstly, `ERC20Spec1` says that if the balance of a participant's account is ever reduced by some amount m, then that must have occurred as a result of a call to the `transfer` method with amount m by the participant, or the `transferFrom` method with the amount m by some other participant.

```
ERC20Spec1 ≜ from e : ERC20 ∧ e.balance(p) = m + m' ∧ m > 0
             to1 e.balance(p) = m'
             onlyIf ∃ p', p''.[⟨p calls e.transfer(p', m)⟩ ∨ ⟨p" calls e.transferFrom(p', m)⟩]
```

Secondly, `ERC20Spec2` specifies under what circumstances some participant `p'` is authorized to spend `m` tokens on behalf of `p`: either `p` approved `p'`, `p'` was previously authorized, or `p'` was authorized for some amount `m + m'`, and spent `m'`.

```
ERC20Spec2 ≜ from e : ERC20 ∧ p : Object ∧ p' : Object ∧ m : Nat
             to1 e.allowed(p, p') = m
             onlyIf ⟨p calls e.approve(p', m)⟩ ∨
                    (e.allowed(p, p') = m ∧
                     ¬ (⟨p' calls e.transferFrom(p, _)⟩ ∨
                          ⟨p calls e.allowed(p, _)⟩)) ∨
                    ∃ p''. [e.allowed(p, p') = m + m' ∧ ⟨p' calls e.transferFrom(p", m')⟩]
```

$$P, Q \quad ::= \quad e \mid e : C \mid P \wedge P \mid P \vee P \mid P \longrightarrow P \mid \neg P \mid \forall x.[P] \mid \exists x.[P] \qquad \textbf{Classical Assertion}$$

Fig. 4. Classical Assertion Syntax

2.5.3 DAO. The Decentralized Autonomous Organization (DAO) [Christoph Jentsch 2016] is a well-known Ethereum contract allowing participants to invest funds. The DAO famously was exploited with a re-entrancy bug in 2016, and lost $50M. Here we provide specifications that would have secured the DAO against such a bug. `DAOSpec1` says that no participant's balance may ever exceed the ether remaining in DAO.

```
DAOSpec1 ≜ from d : DAO
           to d.balance(p) > d.ether
           onlyIf false
```

The second specification `DAOSpec2` states that if a participant's balance is `m`, then either this occurred as a result of joining the DAO with an initial investment of `m`, or the balance is `0`, and they've just withdrawn their funds.

```
DAOSpec2 ≜ from d : DAO ∧ p : Object
           to1 d.balance(p) = m
           onlyIf ⟨p calls d.repay(_)⟩ ∧ m = 0 ∨ ⟨p calls d.join(m)⟩ ∨ d.balance(p) = m
```

In this Section we provide an inference system for constructing proofs of the Necessity Specifications defined in Section 2.3. Proving Necessity Specifications requires several steps, from the following four categories:

(1) Proving Assertion Encapsulation (Section 3.1)
(2) Proving Necessity Specifications from classical specifications for a particular internal method (Section 3.2)
(3) Proving module-wide Necessity Specifications by combining per-method Necessity Specifications (Section 3.3)
(4) Raising necessary conditions to construct proofs of emergent behavior (Section 3.4)

3 PROVING NECESSITY

3.1 Assertion Encapsulation

As already stated in section 2, the first step in proving adherence to Necessity Specifications is to prove that an assertion is encapsulated. And as also already stated, we assume the existence of an inference system for constructing proofs of assertion encapsulation, written $M \vdash A_1 \implies Enc(A_2)$. Such an algorithmic system should be sound, in the sense defined below:

Definition 3.1 (Encapsulation Soundness). A judgment of the form $M \vdash A \implies Enc(A)$ is *sound*, if for all modules M, and assertions A_1 and A_2, if $M \vdash A_1 \implies Enc(A_2)$ then $M \vDash A_1 \implies Enc(A_2)$.

The construction of such an algorithmic system is not central to our work, because, as we shall see in later sections, the Logic of Necessity does not rely on the specifics of any one encapsulation system, only its soundness.

A rudimentary such inference system for encapsulation can be defined on top of a simple type system, while more powerful inference systems are also possible. In the appendix we define such a system where internal classes may be annotated as `enclosed`, and any path that starts at an internal object and only navigates through `enclosed` fields is encapsulated. We will use that inference system for the proofs in Section 4, but, as we said, the exact nature of that system is of little importance to this work.

$$\frac{M \vdash \{x : C \ \wedge \ P_1 \ \wedge \ \neg P\}\ \texttt{res} = x.m(\overline{z})\ \{\neg P_2\}}{M \vdash \texttt{from}\ P_1 \ \wedge \ x : C \wedge \ \langle \_\ \texttt{calls}\ x.m(\overline{z})\rangle\ \texttt{to1}\ P_2\ \texttt{onlyIf}\ P} \quad \text{(If1-Classical)}$$

$$\frac{M \vdash \{x : C \ \wedge \ \neg P\}\ \texttt{res} = x.m(\overline{z})\ \{\texttt{res} \neq y\}}{M \vdash \texttt{from}\ \texttt{wrapped}(y) \ \wedge \ x : C \wedge \ \langle \_\ \texttt{calls}\ x.m(\overline{z})\rangle\ \texttt{to1}\ \neg\texttt{wrapped}(y)\ \texttt{onlyIf}\ P} \quad \text{(If1-Wrapped)}$$

Fig. 5. Per-Method Necessity Specifications

3.2 Per-Method Necessity Specifications from Classical Specifications

In this section we detail how we use Classical Specifications to construct per-method Necessity Specifications. In order to do this note that if a precondition and a certain statement is *sufficient* to achieve a particular result, then the negation of that precondition is *necessary* to achieve the negation of the result after executing that statment. Specifically, using classical Hoare logic, if $\{P\}\ s\ \{Q\}$ is true, then it follows that $\neg P$ is a *necessary precondition* for $\neg Q$ to hold following the execution of s.

We do not define a new assertion language and Hoare logic. Rather, we rely on prior work on such Hoare logics, and assume some underlying logic that can be used to prove *classical assertions*. Classical assertions are ubset of *SpecW*, comprising only those assertions that are commonly present in other specification languages. We provide this subset in Fig. 2. That is, classical assertions are restricted to expressions, class assertions, the usual connectives, negation, implication, and the usual quantifiers.

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

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

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

Fig. 5 introduces proof rules to infer per-method Necessary Specifications. These are rules whose conclusion have the form Single-Step Only.

If1-Classical states that if any state which satisfies P_1 and $\neg P$ and executes the method m on an obejct of class C, leads to a state that satisfies $\neg P_2$, then, any state which satisfies P_1 and calls m on an object of class C will lead to a state that satisfies P_2 only if the original state also satisfied P. We can explain this also as follows: If the triple $.. \vdash \{R_1 \wedge R2\}\ s\ \{Q\}$ holds, then any state that satisfies R_1 and which upon execution of s leads to a state that satisfies $\neg Q$, cannot satisfy R_2 – because if it did, then the ensuing state would have to satisfy Q,

If1-Wrapped essentially states that a method which does not return an object y preserves the "wrappedness" of that object y. This rule is sound, because we do not support calls from internal code to external code – in further work we plan to weaken this requirement, and will strengthen this rule. In more detail, If1-Wrapped states that if P is a necessary precondition for returning an object y, then since we do not support calls from internal code to external code, it follows that P is a necessary precondition to leak y. If1-Wrapped is essentially a specialized version of If1-Classical for the wrapped() predicate. Since wrapped() is not a classical assertion, we cannot use Hoare logic to reason about necessary conditions for leaking of data.

3.3 Necessity Specifications for Any Single Step

We now raise per-method Necessity Specifications to per-step Necessity Specifications. The rules appear in Fig. 6.

$$\frac{\forall C \in dom(M).\forall m \in M(C).\mathtt{mths},\quad M \vdash \mathtt{from}\ A_1 \wedge x : C \wedge \langle \_\ \mathtt{calls}\ x.m(\overline{z}) \rangle\ \mathtt{to1}\ A_2\ \mathtt{onlyIf}\ A_3 \qquad M \vdash A_1 \longrightarrow \neg A_2 \qquad M \vdash A_1 \Rightarrow Enc(A_2)}{M \vdash \mathtt{from}\ A_1\ \mathtt{to1}\ A_2\ \mathtt{onlyIf}\ A_3}\ (\textsc{If1-Internal})$$

$$\frac{M \vdash \mathtt{from}\ A_1\ \mathtt{to}\ A_2\ \mathtt{onlyIf}\ A}{M \vdash \mathtt{from}\ A_1\ \mathtt{to1}\ A_2\ \mathtt{onlyIf}\ A}\ (\textsc{If1-If})$$

$$\frac{M \vdash A_1 \longrightarrow A_1' \qquad M \vdash A_2 \longrightarrow A_2' \qquad M \vdash A_3' \longrightarrow A_3 \qquad M \vdash \mathtt{from}\ A_1'\ \mathtt{to1}\ A_2'\ \mathtt{onlyIf}\ A_3'}{M \vdash \mathtt{from}\ A_1\ \mathtt{to1}\ A_2\ \mathtt{onlyIf}\ A_3}\ (\textsc{If1-}\longrightarrow)$$

$$\frac{M \vdash \mathtt{from}\ A_1\ \mathtt{to1}\ A_2\ \mathtt{onlyIf}\ A \qquad M \vdash \mathtt{from}\ A_1'\ \mathtt{to1}\ A_2\ \mathtt{onlyIf}\ A'}{M \vdash \mathtt{from}\ A_1 \vee A_1'\ \mathtt{to1}\ A_2\ \mathtt{onlyIf}\ A \vee A'}\ (\textsc{If1-}\vee\textsc{I}_1) \qquad \frac{M \vdash \mathtt{from}\ A_1\ \mathtt{to1}\ A_2\ \mathtt{onlyIf}\ A \qquad M \vdash \mathtt{from}\ A_1\ \mathtt{to1}\ A_2'\ \mathtt{onlyIf}\ A'}{M \vdash \mathtt{from}\ A_1\ \mathtt{to1}\ A_2 \vee A_2'\ \mathtt{onlyIf}\ A \vee A'}\ (\textsc{If1-}\vee\textsc{I}_2)$$

$$\frac{M \vdash \mathtt{from}\ A_1\ \mathtt{to1}\ A_2\ \mathtt{onlyIf}\ A \vee A' \qquad M \vdash \mathtt{from}\ A'\ \mathtt{to}\ A_2\ \mathtt{onlyThough}\ \mathtt{false}}{M \vdash \mathtt{from}\ A_1\ \mathtt{to1}\ A_2\ \mathtt{onlyIf}\ A}\ (\textsc{If1-}\vee\textsc{E}) \qquad \frac{M \vdash \mathtt{from}\ A_1\ \mathtt{to1}\ A_2\ \mathtt{onlyIf}\ A \qquad M \vdash \mathtt{from}\ A_1\ \mathtt{to}\ A_2\ \mathtt{onlyThough}\ A'}{M \vdash \mathtt{from}\ A_1\ \mathtt{to1}\ A_2\ \mathtt{onlyIf}\ A \wedge A'}\ (\textsc{If1-}\wedge\textsc{I})$$

Fig. 6. Single-Step Necessity Specifications, $OnlyIf\_1$

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

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

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

There are two disjunction introduction rules (If1-$\vee$I1 and If1-$\vee$I2).

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

Note that given the rule for implication, there is no need for conjunction introduction (IF1-$\wedge$I), but a rule for conjunction elimination is derivable from the rule for implication.

3.4 Necessity Specifications for Emergent Behavior

The final step is to raise per-step Necessity Specifications to multiple step Necessity Specifications, allowing the specification of emergent behavior. Fig. 7 allows for the construction of proofs for *Only Through*, while Fig. 8 provides rules for the construction of proofs of *Only If*.

The rules for both of these relations are fairly similar to each other, and to those of the single step necessity specification from section 3.3. Both relations include rules for implication along with disjunction introduction and elimination. While Fig. 8 includes a rule for conjunction introduction (If-$\wedge$I), such a rule is not possible for *only through*, as unlike *only if* where the necessary condition must hold, specifically, in the starting configuration, there is no such specific moment in time in which the necessary condition for *only through* must hold.

Another rule of note is Changes, in Fig. 7 that states that if the satisfaction of some assertion changes over time, then there must be some specific intermediate state where that change occurred.

$$\frac{M \vdash \mathsf{from}\ A\ \mathsf{to1}\ \neg A\ \mathsf{onlyIf}\ A'}{M \vdash \mathsf{from}\ A\ \mathsf{to}\ \neg A\ \mathsf{onlyThough}\ A'}\ (\textsc{Changes})$$

$$\frac{M \vdash A_1 \longrightarrow A_1' \quad M \vdash A_2 \longrightarrow A_2' \quad M \vdash A_3' \longrightarrow A_3 \quad M \vdash \mathsf{from}\ A_1'\ \mathsf{to}\ A_2'\ \mathsf{onlyThough}\ A_3'}{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A_3}\ (\longrightarrow)$$

$$\frac{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A \quad M \vdash \mathsf{from}\ A_1'\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A'}{M \vdash \mathsf{from}\ A_1 \vee A_1'\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A \vee A'}\ (\vee \mathrm{I}_1) \qquad \frac{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A \quad M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2'\ \mathsf{onlyThough}\ A'}{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2 \vee A_2'\ \mathsf{onlyThough}\ A \vee A'}\ (\vee \mathrm{I}_2)$$

$$\frac{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A'\ \mathsf{onlyThough}\ \mathsf{false} \quad M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A \vee A'}{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A}\ (\vee \mathrm{E}_1) \qquad \frac{M \vdash \mathsf{from}\ A'\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ \mathsf{false} \quad M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A \vee A'}{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A}\ (\vee \mathrm{E}_2)$$

$$\frac{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A_3 \quad M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_3\ \mathsf{onlyThough}\ A}{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A}\ (\textsc{Trans}_1) \qquad \frac{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A_3 \quad M \vdash \mathsf{from}\ A_3\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A}{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A}\ (\textsc{Trans}_2)$$

$$\frac{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyIf}\ A}{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A}\ (\textsc{If}) \qquad M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A_2\ (\textsc{End})$$

Fig. 7. *Only Through*

$$\frac{M \vdash A_1 \longrightarrow A_1' \quad M \vdash A_2 \longrightarrow A_2' \quad M \vdash A_3' \longrightarrow A_3 \quad M \vdash \mathsf{from}\ A_1'\ \mathsf{to}\ A_2'\ \mathsf{onlyIf}\ A_3'}{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyIf}\ A_3}\ (\textsc{If}\text{-}\longrightarrow)$$

$$\frac{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyIf}\ A \quad M \vdash \mathsf{from}\ A_1'\ \mathsf{to}\ A_2\ \mathsf{onlyIf}\ A'}{M \vdash \mathsf{from}\ A_1 \vee A_1'\ \mathsf{to}\ A_2\ \mathsf{onlyIf}\ A \vee A'}\ (\textsc{If}\text{-}\vee \mathrm{I}_1) \qquad \frac{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyIf}\ A \quad M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2'\ \mathsf{onlyIf}\ A'}{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2 \vee A_2'\ \mathsf{onlyIf}\ A \vee A'}\ (\textsc{If}\text{-}\vee \mathrm{I}_2)$$

$$\frac{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyIf}\ A \vee A' \quad M \vdash \mathsf{from}\ A'\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ \mathsf{false}}{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyIf}\ A}\ (\textsc{If}\text{-}\vee \mathrm{E}) \qquad \frac{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyIf}\ A \quad M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A'}{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyIf}\ A \wedge A'}\ (\textsc{If}\text{-}\wedge \mathrm{I})$$

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

Fig. 8. *Only if*

Changes is an important rule in the logic, and in allowing for proofs of emergent properties. It is this rule that ultimately connects program execution to encapsulated properties.

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

$$\frac{M \vdash \mathsf{from}\ A_1\ \mathsf{to1}\ A_2\ \mathsf{onlyIf}\ A_3}{M \vdash \mathsf{from}\ A_1\ \mathsf{to}\ A_2\ \mathsf{onlyThough}\ A_3}$$

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

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

Moreover, any *only if* specification entails the corresponding *only through* specification (IF). *Only if* also includes a transitivity rule (IF-TRANS), but since the necessary condition must be true in the beginning state, there is only a single rule. Finally, any starting condition is itself a necessary precondition (IF-START).

3.5 Soundness of the Necessity Logic

THEOREM 3.2 (SOUNDNESS). *Assuming a sound SpecW proof system, $M \vdash A$, and a sound encapsulation inference system, $M \vdash A \implies Enc(A')$, and and that on top of these systems we built the Necessity Logic according to the rules in Figures 5, and 6, and 8, and 7, then, for all modules M, and all Necessity Specifications H:*

$$M \vdash H \qquad implies \qquad M \vDash H$$

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

We have mechanized the proof of Theorem 3.2 in Coq. This can be found in the associated artifact. The mechanized Coq formalism deviates slightly from the system as presented here, mostly in the expression of the *SpecW* language. The Coq version of *SpecW* restricts variable usage to expressions, and allows only addresses to be used as part of non-expression syntax. The reason for this is to avoid spending sizable effort encoding variable renaming and substitution, a well-known difficulty for languages such as Coq. This is justifiable, as we are still able to express assertions such as $\langle x \texttt{ access } y \rangle$, but using addresses and introducing equality statements as part of expressions to connect variables to address, i.e. $\langle \alpha_x \texttt{ access } \alpha_y \rangle \wedge \alpha_x == x \wedge \alpha_y == y$.

As we state above, assume the existence of a sound, algorithmic proof system for encapsulation and *SpecW* specifications, along with the soundness of the Hoare logic required for the encoding of rules IF1-CLASSICAL and IF1-WRAPPED.

4 PROOF OF ADHERENCE TO NECESSITYBANKSPEC

In this section we return to the Bank Account example, providing an elaborated proof. As we have stated Section 3, we assume the existence of a logic for constructing proofs of the form $M \vdash A$. For the purposes of the examples in this section, we state several rules that under any such logic should be provable.

$$M \vdash \langle x \texttt{ calls } y.m(\overline{z}) \rangle \longrightarrow \langle x \texttt{ external} \rangle \quad \text{(CALLER-EXT)}$$

$$M \vdash \langle x \texttt{ calls } y.m(\overline{z}) \rangle \longrightarrow \langle x \texttt{ access } y \rangle \quad \text{(CALLER-RECV)}$$

$$M \vdash \langle x \texttt{ calls } y.m(\ldots, z_i, \ldots) \rangle \longrightarrow \langle x \texttt{ access } z_i \rangle \quad \text{(CALLER-ARGS)}$$

$$\frac{C \in M}{M \vdash x : C \longrightarrow \langle x \texttt{ internal} \rangle} \quad \text{(CLASS-INT)} \qquad \frac{(\texttt{field\_}f : D) \in M(C).(\texttt{flds})}{M \vdash e : C \longrightarrow e.f : D} \quad \text{(FLD-CLASS)}$$

$$\frac{(\texttt{class enclosed } C\{\_;\_\}) \in M}{M \vdash \alpha : C \longrightarrow \texttt{wrapped}(\alpha)} \quad \text{(WRAPPED-INT)} \qquad M \vdash \texttt{false} \longrightarrow A \quad \text{(ABSURD)}$$

$$M \vdash A \vee \neg A \quad \text{(EXCLUDED MIDDLE)}$$

We devote the rest of this section to the elaborated of our running example of a Bank Account, along with a proof of the bank account specification using our Logic of Necessity. We start by providing an implementation of `BankMdl` in Fig. 9. A `Bank` consists of a `Ledger`, a method for transferring funds between accounts (`transfer`), and a ghost field, `balance` for

```
module BankMdl
  class Account
    field password : Object
    method authenticate(pwd)
      (PRE:  a : Account ∧ b : Bank
       POST: b.getBalance(a)old == b.getBalance(a)new)
      (PRE:  a : Account
       POST: res != a.password)
      (PRE:  a : Account
       POST: a.passwordold == a.passwordnew)
      {return pwd == this.password}
    method changePassword(pwd, newPwd)
      (PRE:  a : Account
       POST: res != a.password)
      (PRE:  a : Account ∧ b : Bank
       POST: b.getBalance(a)old == b.getBalance(a)new)
      (PRE:  a : Account ∧ pwd != this.password
       POST: a.passwordold = a.passwordnew)
      {if pwd == this.password
        this.password := newPwd}

  class enclosed Ledger
    field acc1 : Account
    field bal1 : int
    field acc2 : Account
    field bal2 : int
    ghost intrnl getBalance(acc) =
      if acc == acc1
        bal1
      else if acc == acc2
        bal2
      else -1
    method transfer(amt, from, to)
      (PRE:  a : Account ∧ b : Bank ∧ (a != acc1 ∧ a != acc2)
       POST: b.getBalance(a)old == b.getBalance(a)new)
      (PRE:  a : Account
       POST: res != a.password)
      (PRE:  a : Account
       POST: a.passwordold == a.passwordnew)
      {if from == acc1 && to == acc2
         bal1 := bal1 - amt
         bal2 := bal2 + amt
       else if from == acc2 && to == acc1
         bal1 := bal1 + amt
         bal2 := bal2 - amt}


  class Bank
    field book : Ledger
    ghost intrnrl getBalance(acc) = book.getBalance(acc)
    method transfer(pwd, amt, from, to)
      (PRE:  a : Account ∧ b : Bank ∧ ¬ (a == acc1 ∧ a == acc2)
       POST: b.getBalance(a)old a= b.getBalance(a)new)
      (PRE:  a : Account
       POST: res != a.password)
      (PRE:  a : Account
       POST: a.passwordold == a.passwordnew)
      {if (from.authenticate(pwd))
         book.transfer(amt, from, to)}
```

Fig. 9. Bank Account Module

looking up the balance of an account at a bank.[1] A `Ledger` is a mapping from `Accounts` to their balances. Note, for brevity our implementation only includes two accounts (`acc1` and `acc2`), but it is easy to see how this could extend to a `Ledger` of

[1] A Necessity Specification is independent of the implementation details of the code. It would need to hold for an account whose implementation did not use a ledger or ghost variables to hold balances.

arbitrary size. `Ledger` is annotated as `enclosed`, and as such the type system ensures our required encapsulation properties. Finally, an `Account` has some `password` object, and methods to authenticate a provided password (`authenticate`), change the password (`changePassword`). The specification we would like to prove is given as `NecessityBankSpec`.

```
NecessityBankSpec ≜ from b : Bank ∧ b.getBalance(a) = bal
                    to b.getbalance(a) < bal
                    onlyIf ¬wrapped(a)
```

That is, if the balance of an account ever decreases, it must be true that some object external to `BankMdl` has access to the password of that account. An informal account of the construction of the overall proof follows the outline of our reasoning given in Section 1.2:

- **A** We use our rudimentary encapsulation system to prove that both `b.getBalance(a) = bal` and `a.password = pwd` are encapsulated assertions.
- **B** We use classical specifications to prove that only a call to `Bank::transfer` with the correct password may be used to decrease the balance of an account. Similarly, we use classical specifications to prove that no method can leak the password of an account.
- **C** We combine the per-method necessary preconditions along with the encapsulation of `b.getBalance(a) = bal` to arrive at a per-step necessary precondition for reducing the balance using *any* method in `BankMdl`. Similarly, we show that *no* step may leak the password of an account.
- **D** Finally we use the Logic of Necessity and the results of A, B, and C, to prove the emergent behavior specified in `NecessityBankSpec`.

4.1 A: The Encapsulation of Account Balance and Password

We base the soundness of our encapsulation of the type system of *LangX*, and use the proof rules given in Fig.s 13 and 14. We provide the proof for the encapsulation of `b.getBalance(a)` below

BalanceEncaps:

aIntrnl:
`BankMdl ⊢ b, b′ : Bank ∧ a : Account ∧ b.getBalance(a) = bal ⇒ Intrnl(a)` by Intrnl-Obj

bIntrnl:
`BankMdl ⊢ b, b′ : Bank ∧ a : Account ∧ b.getBalance(a) = bal ⇒ Intrnl(b)` by Intrnl-Obj

getBalIntrnl:
`BankMdl ⊢ b, b′ : Bank ∧ a : Account ∧ b.getBalance(a) = bal ⇒ Intrnl(b.getBalance(a))` by aEncaps, bEncaps, and Intrnl-Ghost

balIntrnl:
`BankMdl ⊢ b, b′ : Bank ∧ a : Account ∧ b.getBalance(a) = bal ⇒ Intrnl(bal)` by Intrnl-Int

`BankMdl ⊢ b, b′ : Bank ∧ a : Account ∧ b.getBalance(a) = bal ⇒ Enc(b.getBalance(a) = bal)` by getBalIntrnl, balIntrnl, Enc-Intrnl

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

4.2 B: Proving Necessary Preconditions from Classical Specifications

We now provide proofs for necessary preconditions on a per-method basis, leveraging classical specifications.

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

AuthBalChange:

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

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

```
from a : Account ∧ a′ : Account ∧ b : Bank ∧ b.getBalance(a′) = bal ∧
⟨_ calls a.authenticate(pwd)⟩
  to1 b.getbalance(a′) < bal
  onlyIf false
```
by If1-Classical

```
from a : Account ∧ a′ : Account ∧ b : Bank ∧ b.getBalance(a′) = bal ∧
⟨_ calls a.authenticate(pwd)⟩
  to1 b.getbalance(a′) < bal
  onlyIf ⟨_ calls b.transfer(a′.password, amt, a′, to)⟩
```
by Absurd and If1-⟶

We provide the statements of the proof results for the remaining methods in the module below, but we elide most of the steps as they do not differ much from that of **AuthBalChange**.

ChangePasswordBalChange:

```
from a, a′ : Account ∧ b : Bank ∧ b.getBalance(a′) = bal ∧ ⟨_ calls a.changePassword(pwd)⟩
  to1 b.getbalance(a′) < bal
  onlyIf ⟨_ calls b.transfer(a′.password, amt, a′, to)⟩
```
by similar reasoning to **AuthBalChange**

Ledger::TransferBalChange:

```
from l : Ledger ∧ a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧
⟨_ calls l.transfer(amt, from, to)⟩
  to1 b.getbalance(a) < bal
  onlyIf ⟨_ calls b.transfer(a.password, amt, a, to)⟩
```
by similar reasoning to **AuthBalChange**

Bank::TransferBalChange:

```
from b′ : Bank ∧ a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧
⟨_ calls b′.transfer(pwd, amt, from, to)⟩
  to1 b.getbalance(a) < bal
  onlyIf a == from ∧ pwd == a.password ∧ b′ == b
```
by similar reasoning to **AuthBalChange**

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧
⟨_ calls b′.transfer(pwd, amt, from, to)⟩
  to1 b.getbalance(a) < bal
  onlyIf ⟨_ calls b.transfer(a.password, amt, a, to)⟩
```
by If1-⟶

Below we provide the proofs for each method in `BankMdl` that they cannot be used to leak the password of an account.

AuthPwdLeak:

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

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

```
from wrapped(pwd) ∧ a : Account ∧ a′ : Account ∧ a.password = pwd ∧
⟨_ calls a′.authenticate(_)⟩
  to1 ¬wrapped(_)
  onlyIf false
```
by If1-Wrapped

ChangePasswordLeak:

```
from wrapped(pwd) ∧ a : Account ∧ a′ : Account ∧ a.password = pwd ∧
⟨_ calls a′.changePassword(_, _)⟩
  to1 ¬wrapped(pwd)
  onlyIf false
```
by similar reasoning to **AuthPwdLeak**

Ledger::TransferPwdLeak:

```
from wrapped(pwd) ∧ a : Account ∧ l : Ledger ∧ a.password = pwd ∧
⟨_ calls l.transfer(_, _, _)⟩
  to1 ¬wrapped(pwd)
  onlyIf false
```
by similar reasoning to **AuthPwdLeak**

Bank::TransferPwdLeak:

```
from wrapped(pwd) ∧ a : Account ∧ b : Bank ∧ a.password = pwd ∧
⟨_ calls l.transfer(_, _, _, _)⟩
  to1 ¬wrapped(pwd)
  onlyIf false
```

by similar reasoning to **AuthPwdLeak**

4.3 C: Proving Per-Step Necessary Specifications

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

```
BalanceChange ≜ from  a : Account ∧ b : Bank ∧ b.getBalance(a) = bal
                to1 b.getBalance(a) < bal
                onlyIf ⟨_ calls b.transfer(a.password, _, a, _)⟩
```

```
PasswordChange ≜ from a : Account ∧ a.password = p
                 to1 ¬ a.password ≠ p
                 onlyIf ⟨_ calls a.changePassword(a.password, _)⟩
```

```
PasswordLeak ≜ from a : Account ∧ a.password = p ∧ wrapped<p>
               to1 ¬ wrapped<p>
               onlyIf false
```

We provide the proofs of these below. The proof of `BalanceChange` is constructed by combining the results from 4.1 and 4.2 using If1-Internal. Again, we elide the details of the proof of `PasswordChange` and `PasswordLeak` as they are similar to that of `BalanceChange`.

BalanceChange:

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal
  to1 b.getbalance(a) < bal
  onlyIf ⟨_ calls b.transfer(a.password, amt, a, to)⟩
```

by **AuthBalChange**, **ChangePasswordBalChange**, **Ledger::TransferBalChange**, **Bank::TransferBalChange**, **BalanceEncaps**, and If1-Internal

PasswordChange:

```
from a : Account ∧ a.password = p
  to1 a.password ≠ p
  onlyIf ⟨_ calls a.changePassword(p, _)⟩
```

by similar reasoning to **BalanceChange**

PasswordLeak:

```
from a : Account ∧ a.password = p ∧ wrapped(p)
  to1 ¬wrapped(p)
  onlyIf false
```

by similar reasoning to `Balancechange`

4.4 D: Proving Emergent Behavior

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

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

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

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

(4 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.

(4 B) If the password is different from the initial password, then there must have been an intervening program state when it changed. By `PasswordChange` we know that this must have occurred by a call to change password with the correct password. Thus, there must be a some intervening program state where the initial password is known. From here we proceed by the same reasoning as **(4 A)**.

NecessityBankSpec:

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal
  to b.getBalance(a) < bal
  onlyThrough ⟨_ calls b.transfer(a.password, _, a, _)⟩
```
by Changes and BalanceChange

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal
  to b.getBalance(a) < bal
  onlyThrough ∃ o.[⟨o external⟩ ∧ ⟨o access a.password⟩]
```
by ⟶, Caller-Ext, and Calls-Args

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧ a.password = pwd
  to b.getBalance(a) < bal
  onlyThrough ¬wrapped(a.password)
```
by ⟶

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧ a.password = pwd
  to b.getBalance(a) < bal
  onlyThrough ¬wrapped(a.password) ∧ (a.password = pwd ∨ a.password != pwd)
```
by ⟶ and Excluded Middle

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧ a.password = pwd
  to b.getBalance(a) < bal
  onlyThrough (¬wrapped(a.password) ∧ a.password = pwd) ∨
(¬wrapped(a.password) ∧ a.password != pwd)
```
by ⟶

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧ a.password = pwd
  to b.getBalance(a) < bal
  onlyThrough ¬wrapped(pwd) ∨ a.password != pwd
```
by ⟶

Case A (¬wrapped(pwd)):

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧ a.password = pwd
  to ¬wrapped(pwd)
  onlyIf wrapped(pwd)  ∨ ¬wrapped(pwd)
```
by If-⟶ and Excluded Middle

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧ a.password = pwd
  to ¬wrapped(pwd)
  onlyIf ¬wrapped(pwd)
```
by ∨E and PasswordLeak

Case B (a.password != pwd):

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧ a.password = pwd
  to a.password != pwd
  onlyThrough ⟨_ calls a.changePassword(pwd, _)⟩
```
by Changes and Password-Change

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧ a.password = pwd
  to a.password != pwd
  onlyThrough ¬wrapped(pwd)
```
by ∨E and PasswordLeak

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧ a.password = pwd
  to a.password != pwd
  onlyIf ¬wrapped(pwd)
```
by **Case A** and Trans

```
from a : Account ∧ b : Bank ∧ b.getBalance(a) = bal ∧ a.password = pwd
  to b.getBalance(a) < bal
  onlyIf ¬wrapped(pwd)
```
by **Case A**, **Case B**, If-∨I_2, and If-⟶

5 RELATED WORK

Behavioural Specification Languages. [Hatcliff et al. 2012] provide an excellent survey of contemporary specification approaches. With a lineage back to Hoare logic [Hoare 1969], Design by Contract [Meyer 1997] was the first popular attempt to bring verification techniques to object-oriented programs. Several more recent specification languages are now making their way into use, including JML [Leavens et al. 2007a], Spec♯ [Barnett et al. 2005], Dafny [Leino 2010] and Whiley [Pearce and Groves 2015]. These approaches assume a closed system, where modules can be trusted to coöperate. Our approach builds upon these fundamentals, particularly two-state invariants [Leino and Schulte 2007] and Considerate Reasoning [Summers and Drossopoulou 2010].

Defensive Consistency. [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."* Defensively consistent modules are particularly hard to design, write, understand, and verify: but it is easier to make guarantees about multiple component systems [Murray 2010].

[Drossopoulou et al. 2020]'s Chainmail is a specification language for open world systems with necessary conditions. Like *SpecX*, Chainmail is able to express specifications of *permission*, *provenance*, and *control*. Unlike *SpecX*, Chainmail has a rich set of temporal operators and includes assertions about *space*. However we present a simpler semantics, that does not restrict execution to the top frame of the stack. We model aspects of *time* using our Logic of Necessity and don't include assertions about *space*, as they are not required for either the definition of our logic, or expressing the Chainmail examples (see Section 4). To date, Chainmail lacks a proof system.

Object Capabilities. *Capabilities* supporting the development of concurrent and distributed system were developed in the 60's by [Dennis and Horn 1966], and adapted to the programming languages setting in the 70's [Morris Jr. 1973]. *Object capabilities* were first introduced by [Miller 2006]. and many recent studies manage to verify safety or correctness of object capability programs. Google's Caja [Miller et al. 2008] applies sandboxes, proxies, and wrappers to limit components' access to *ambient* authority. Sandboxing has been validated formally: [Maffeis et al. 2010] develop a model of JavaScript, demonstrate that it obeys two principles of object capability systems, and show how untrusted applications can be prevented from interfering with the rest of the system. Recent programming languages [Burtsev et al. 2017; Hayes et al. 2017; Rhodes et al. 2014] including Newspeak [Bracha 2017], Dart [Bracha 2015], Grace [Black et al. 2012; Jones et al. 2016] and Wyvern [Melicher et al. 2017] have adopted the object capability model.

[Murray 2010] made the first attempt to formalise defensive consistency and correctness. Murray's model was rooted in counterfactual causation [Lewis 1973]: an object is defensively consistent when the addition of untrustworthy clients cannot cause well-behaved clients to be given incorrect service. Murray formalised defensive consistency abstractly, without a specification language for describing effects. Both Miller and Murray's definitions are intensional, describing what it means for an object to be defensively consistent. [Drossopoulou and Noble 2014] sketched a specification language and used it to specify the six policies from [Miller 2006], They also sketched how a trust-sensitive example (escrow) could be verified in an open world [Drossopoulou et al. 2015].

[Devriese et al. 2016] have deployed powerful theoretical techniques to address similar problems: They show how step-indexing, Kripke worlds, and representing objects as state machines with public and private transitions can be used to reason about object capabilities. They have demonstrated solutions to a range of exemplar problems, including the DOM wrapper (replicated in 2.5.1) and a mashup application. Their distinction between public and private transitions is similar to our distinction between internal and external objects.

[Swasey et al. 2017] designed OCPL, a logic for object capability patterns, that supports specifications and proofs for object-oriented systems in an open world. They draw on verification techniques for security and information flow: separating internal implementations ("high values" which must not be exposed to attacking code) from interface objects ("low values" which may be exposed). OCPL supports defensive consistency (they use the term "robust safety" from the security community [Bengtson et al. 2011]) via a proof system that ensures low values can never leak high values to external attackers. As low values *can* be exposed, the behaviour of the system is described by considering attacks only on low values. They prove a number of object-capability patterns, including sealer/unsealer pairs, the caretaker, and a general membrane.

[Schaefer et al. 2018] have added support for information-flow security using refinement to ensure correctness (in this case confidentiality) by construction. By enforcing encapsulation, *SpecX*all these approaches share similarity with techniques such as ownership types [Clarke et al. 1998; Noble et al. 1998], which also protect internal implementation objects from accesses that cross encapsulation boundaries. [Banerjee and Naumann 2005a,b] demonstrated that by ensuring confinement, ownership systems can enforce representation independence.

SpecX 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 *SpecX* users only need to understand first order logic. Finally, none of these systems offer the kinds of necessity assertions addressing control flow, provenance, and permission that are at the core of *SpecX*'s approach.

Blockchain. 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 *SpecX*'s "calls"), access to variables and sums (only) can be computed over collections. *SpecX* is strictly more expressive as a specification language, including quantification over objects and sets (so can compute arbitrary reductions on collections) and specifications for permission ("access"), and viewpoint ("external") which have no analogues in VerX. Unlike *SpecX*, VerX includes a practical tool that has been used to verify a hundred properties across case studies of twelve Solidity contracts.

Incorrectness Logic. [O'Hearn 2019; Raad et al. 2020] defined a Hoare Logic for modelling program incorrectness. O'Hearn's Incorrectness Logic is based on a Reverse Hoare Logic [de Vries and Koutavas 2011], and empowers programmers to specify preconditions under which specific errors and program states may result. Incorrectness Logic provides a sound and compositional way to reason about the presence of bugs rather than the absence of bugs. As with Hoare logic, Incorrectness Logic provides a system for reasoning about sufficient conditions for post-conditions to hold. However, where Hoare logic specifies the shape of the result of execution of all program states that satisfy the precondition, Incorrectness Logic specifies that all states that satisfy the postcondition are reachable from those that satisfy the precondition. This suits the specification of program errors, as it allows for the exclusion of false negatives. In comparison, *SpecX*, as with Hoare Logic, is concerned with correctness (as seen in the exemplars of Section 4). Extending the comparison, *SpecX* differs from both Hoare Logic and Incorrectness, in the ability to specify, not just sufficient conditions, but necessary conditions for reaching certain program states. Neither Incorrectness Logic, nor Hoare Logic allows for such specifications.

6 CONCLUSION

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

This paper defines *SpecX*, a specification language that takes an holistic view of modules. Rather than focusing on individual pieces of code, *SpecX* can describe the emergent behaviour of all the classes and all their methods in a module. Using *SpecX*, programmers can write Necessity Specifications defining the necessary conditions for effects — things happening — in programs. *SpecX* treats program effects as actions: that is, as transitions from states satisfying some assertion A_1 to other states satisfying assertions A_2. Necessity Specifications then permit those transitions only if some other assertion A_3 holds beforehand; or only if A_3 holds and A_2 is reached in a single step; or only if A_3 holds at some point on the path between A_1 and A_2. The assertion language A supports the usual expressions about the values of variables, the contents of the heap, and predicates to capture provenance and permission.

We have developed an inference system to prove that modules meet necessity specifications. Our inference system exploits the sufficient conditions of classical method specifications (pre- and post-conditions) to infer per method necessity specifications, and then generalise those to cover any single execution step. By combining single step specifications, programs can describe components' emergent behaviour — i.e. the necessary conditions for program effects, irrespective of how that effect is caused. Finally, we have proved our inference system is sound, and then used it to prove the necessity specifications of the bank account example.

Deriving per method necessity Specifications from classical method pre- and post-conditions has two advantages: First, it means that we did not need to develop a special purpose logic for that task. Second, it means that all modules that have the same classical per method specifications can be proven to satisfy the same Necessity Specifications using the *same* proof. This holds even when the classical specifications are "similar enough". For instance, the modules `Mod1` and `Mod3` from the introduction, and also module `BankMdl` from section 4 satisfy `NecessityBankSpec`. The proofs for these three modules differ slightly in the proof of encapsulation of the assertion `a.password=pwd` (step **(**A) in section 4), and in the proof of the per-method necessity specification for `transfer`, but otherwise are identical.

We considered developing a bespoke logic to infer per method Necessity Specifications. Such a logic might be more powerful than the *SpecX* inference system; we aim to consider that in further work. Moreover, the classical specifications we using to infer the per method Necessity Specifications are very "basic", and thus they need to explicitly state what has not been modified – this makes proofs very cumbersome. In future work, we plan to start from classical specifications which have more information about the affected part of the state (e.g. using `modifies`-clauses, or implicit frames [Ishtiaq and O'Hearn 2001; Leavens et al. 2007b; Leino 2013; Parkinson and Summers 2011; Smans et al. 2012]) as we expect this could make the step from classical specifications to per method necessity specifications more convenient

Our inference system is parametric with an algorithmic judgment which can prove that an assertion is encapsulated. In this paper we have used a rudimentary, type-based inference system, but we aim to develop a logic for inferring such assertions. The concept of encapsulation in this work is very coarse, and based on the classes of objects. In the future, we plan to refine our handling of encapsulation, and support reasoning about more flexible configurations of objects. Similarly, to facilitate the formal treatment, we currently forbid internal objects to call into external objects: we hope a better model of encapsulation will let us remove this restriction.

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

REFERENCES

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

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

Mike Barnett, Rustan Leino, and Wolfram Schulte. 2005. The Spec Programming System: An Overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices* (cassis 2004, construction and analysis of safe, secure and interoperable smart devices ed.) *(Lecture Notes in Computer Science, Vol. 3362)*. Springer, 49–69. https://doi.org/10.1007/978-3-540-30569-9_3

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.* 33, 2, Article 8 (Feb. 2011), 45 pages. https://doi.org/10.1145/1890028.1890031

Andrew Black, Kim Bruce, Michael Homer, and James Noble. 2012. Grace: the Absence of (Inessential) Difficulty. In *Onwards.*

Gilad Bracha. 2015. *The Dart Programming Language.*

Gilad Bracha. 2017. The Newspeak Language Specification Version 0.1. (Feb. 2017). `newspeaklanguage.org/`.

Anton Burtsev, David Johnson, Josh Kunz, Eric Eide, and Jacobus E. van der Merwe. 2017. CapNet: security and least authority in a capability-enabled cloud. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017*. 128–141. https://doi.org/10.1145/3127479.3131209

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

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

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

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Comm. ACM* 9, 3 (1966).

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

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

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

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.

Ian J. Hayes, Xi Wu, and Larissa A. Meinicke. 2017. Capabilities for Java: Secure Access to Resources. In *APLAS*. 67–84. https://doi.org/10.1007/978-3-319-71237-6_4

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

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

Timothy Jones, Michael Homer, James Noble, and Kim B. Bruce. 2016. Object Inheritance Without Classes. In *ECOOP*. 13:1–13:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.13

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

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

Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382–401. https://doi.org/10.1145/357172.357176

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

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

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

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

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

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

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

Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *ECOOP*. 20:1–20:27. https://doi.org/10.4230/LIPIcs.ECOOP.2017.20

B. Meyer. 1997. *Object-Oriented Software Construction, Second Edition* (second ed.). Prentice Hall.

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

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

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/.

James H. Morris Jr. 1973. Protection in Programming Languages. *CACM* 16, 1 (1973).

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

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

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

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

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

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

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

Dustin Rhodes, Tim Disney, and Cormac Flanagan. 2014. Dynamic Detection of Object Capability Violations Through Model Checking. In *DLS*. 103–112. https://doi.org/10.1145/2661088.2661099

Martin C. Rinard. 2003. Acceptability-oriented computing. In *OOPSLA*. 221–239.

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

Ina Schaefer, Tobias Runge, Alexander Knüppel, Loek Cleophas, Derrick G. Kourie, and Bruce W. Watson. 2018. Towards Confidentiality-by-Construction. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I*. 502–515. https://doi.org/10.1007/978-3-030-03418-4_30

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

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

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

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