



EPiC Series in Computing

Volume 112, 2026, Pages 548–565

Proceedings of the 26th Conference on Logic for Programming, Artificial Intelligence, and Reasoning



Formalizing a Many-Sorted Hybrid Polyadic Modal Logic in Lean

Alexandru Oltean, Bogdan Macovei, and Ioana Leuştean

Faculty of Mathematics and Computer Science, University of Bucharest,
Academiei nr.14, sector 1, C.P. 010014, Bucharest, Romania

{andrei-alexandru.oltean, bogdan.macovei, ioana.leustean}@unibuc.ro

Abstract

We present a Lean formalization of a general hybrid modal logic with many-sorted signatures and polyadic modal operators. The system borrows ideas from both algebraic specification and dynamic logics, and is designed to serve as a uniform axiomatic foundation for specifying and verifying programming languages and security protocols. We expose a DSL for users to define languages and protocols as many-sorted signatures, specify the relevant domain-specific axioms, and reason about program executions or protocol runs. We provide a machine-checked proof of its soundness theorem and showcase the framework’s versatility through several applications: an imperative programming language for code verification, a BAN logic and an epistemic logic used for security protocols, and the modal system S5. We have designed our formalization to be intrinsically sorted, that is, well-sorted formulas in the base language are well-typed terms in Lean. Thanks to intrinsic sorting, all domain specific applications can be easily embedded in our framework via the DSL, at no additional syntactic overhead required for the user to prove.

1 Introduction

We present a Lean [17] formalization of a general hybrid modal logic with many-sorted signatures and polyadic modal operators. The system borrows ideas from both algebraic specification and dynamic logics, and is designed to serve as a uniform axiomatic foundation for specifying and verifying programming languages and security protocols. Our approach embeds programming languages, their operational semantics, and correctness properties into a single logical framework, which is proved to be mathematically sound in the proof assistant’s dependent type theory. Lean’s modularity and extensibility make it a natural fit for this architecture: we expose a DSL for users to define languages or protocols as many-sorted signatures, specify the relevant domain-specific axioms, and reason about program executions or protocol runs entirely within the same base logic.

Our formalization focuses on the many-sorted hybrid modal logic $\mathcal{H}_{\Sigma}(@, \forall)$ defined in [15]. We implement the syntax and semantics of this system, and we prove its soundness theorem for arbitrary axiomatic extensions. We implement three applications of our system, proving its ability to serve as a general verification framework. As a starting example, we embed an

imperative programming language and its operational semantics into this logic: Plotkin’s SMC machine [19], following its hybrid modal treatment in [15], which we formalize here. On this programming language, we use our modal proof system to derive Hoare-like inference rules and reason about program correctness. Second, we express different logics for the analysis about security protocols; drawing first from BAN logic [5], and from earlier work on an epistemic logic with actions [16]. Finally, we show an embedding of the basic modal language and its S5 proof system within our hybrid setting, demonstrating that *fragments* of full hybrid expressivity can also be achieved through this formalization. All of these applications are introduced by means of a custom DSL that allows the elaboration of Backus-Naur-like syntax into the data structures of many-sorted hybrid logic.

This paper is organized as follows. In Section 2 we shortly recall the logic $\mathcal{H}_\Sigma(@, \forall)$. In Section 3 we explain our design for mechanizing the syntax and proof system in Lean. In Section 4 we show our implementation of the semantics and the soundness theorem. In Section 5 we present the three applications we outlined above. The last section contains related results and conclusions.

All code presented in this paper is openly accessible in the following repository: <https://github.com/alexoltean61/msphml-lean>.

2 Preliminaries: the System $\mathcal{H}_\Sigma(@, \forall)$

Hybrid modal logics originate in Arthur Prior’s research on tense logics in the 1960s [4]. Such systems are obtained from modal logics by adding a special kind of atomic symbols, called *nominals*, as syntactical references to the worlds of a Kripke model. Various systems of hybrid logics are known in the literature, examples being the treatment of the *satisfaction operator* $@_i$, as well as *state variables* and *binders* ($\downarrow_x, \forall x$). We refer to [4, 1] for detailed accounts on this topic. In this section we recall the syntax and the semantics of the many sorted hybrid polyadic modal logic defined in [15].

Let S be a non-empty set of sorts. A *signature with constant nominals* is a triple (S, Σ, N) , where we have $\Sigma = (\Sigma_{w,s})_{w \in S^+, s \in S}$, an $S^+ \times S$ -indexed family of countable sets of modal operators¹, and $N = (N_s)_{s \in S}$, an S -indexed family of non-empty, countable sets of *constant nominals*.

Let $\Sigma = (S, \Sigma, N)$ be a signature with constant nominals, and let $(PROP)_{s \in S}$, $(NOM)_{s \in S}$ and $(SVAR)_{s \in S}$ be S -sorted sets of *propositional variables*, (non-constant) *nominals*, and *state variables*, respectively. For any $s \in S$, the well-formed formulas of sort s are defined by:

$$\varphi_s := p \mid j \mid y \mid \neg \varphi_s \mid \varphi_s \vee \varphi_s \mid \sigma(\varphi_{s_1}, \dots, \varphi_{s_n}) \mid @_k^s \varphi_t \mid \forall x \varphi_s,$$

where $p \in PROP_s$, $j \in NOM_s \cup N_s$, $t \in S$, $k \in NOM_t \cup N_t$, $y \in SVAR_s$, $x \in SVAR_t$ and $\sigma \in \Sigma_{s_1 \dots s_n, s}$. Derived propositional operators and the dual binder $\exists$ are defined as usual. For all operators $\sigma \in \Sigma_{s_1 \dots s_n, s}$, the dual operator $\sigma^\square$ is defined as $\sigma^\square(\varphi_1, \dots, \varphi_n) = \neg \sigma(\neg \varphi_1, \dots, \neg \varphi_n)$. The language is *hybrid*, due to the existence of nominals, constant nominals and state variables. We will refer to such symbols collectively as *state symbols*.

The Kripke semantics for this language is based on frames and models, as usual but, in this setting, these are also many-sorted. A Σ -*frame* is a tuple $\mathcal{F} = (W, (R_\sigma)_{\sigma \in \Sigma}, N^\mathcal{F})$, where:

- $W = (W_s)_{s \in S}$ is an S -sorted set of *worlds*, each set assumed non-empty;
- $R_\sigma \subseteq W_{s_1} \times \dots \times W_{s_n}$, for all $\sigma \in \Sigma_{s_1 \dots s_n, s}$, is the *accessibility relation*;

¹Note that we disallow operators from having nullary arity. This choice was made for convenience, and all subsequent considerations can be adapted to fit the nullary case if required.

- $N^{\mathcal{F}} = (N_s^{\mathcal{F}})_{s \in S}$ with $N_s^{\mathcal{F}} : N_s \rightarrow W_s$ for all s . This is the *valuation* of constant nominals.

If $\mathcal{F} = (W, (R_\sigma)_{\sigma \in \Sigma}, N^{\mathcal{F}})$ is a Σ -frame, then a Σ -model based on $\mathcal{F}$ is a tuple $\mathcal{M} = (\mathcal{F}, V)$, where $V = (V_s)_{s \in S}$ is an S -sorted *valuation* function, with $V_s : PROP_s \cup NOM_s \rightarrow \mathcal{P}(W_s)$. For all $k \in NOM_s$, $V_s(k)$ is required to be a *singleton* subset of W_s . This is the usual definition of valuations on nominals in hybrid modal logic but, in this paper, we will simply denote $V_s(i) = w$ instead of $V_s(i) = \{w\}$ for $w \in W_s$ and $i \in NOM_s$. As customary in hybrid settings, Tarski-style assignment functions are introduced for the evaluation of variables: given a Σ -frame $\mathcal{F} = (W, (R_\sigma)_{\sigma \in \Sigma}, N^{\mathcal{F}})$, an assignment is an S -sorted function $(g_s)_{s \in S}$, $g_s : SVAR_s \rightarrow W_s$. We write $g' \rightsquigarrow^x g$ with the usual meaning of g' being an x -variant of g . Note that, for any $s \in S$, state symbols are evaluated in worlds (elements of W_s), while (modal) propositional variables $p \in PROP_s$ are evaluated as arbitrary subsets of W_s .

Now we are ready to define the S -sorted *satisfaction relation*. Given a model $\mathcal{M} = (W, (R_\sigma)_{\sigma \in \Sigma}, N, V)$, an assignment function g and $w \in W_s$, the relation $\models = (\models^s)_{s \in S}$ is defined recursively for all $s \in S$ as in Table 1 below. The definition of validity in models and frames is standard.

Let Λ be an S -sorted set of $\mathcal{H}_\Sigma(@, \forall)$ formulas. Of particular importance to our considerations are the following two classes of models and frames: $Mod(\Lambda)$, the class of models in which every formula in Λ is valid, and $Fr(\Lambda)$, the class of models based on frames where every formula in Λ is valid. Clearly, $Fr(\Lambda) \subseteq Mod(\Lambda)$.

The Hilbert-style proof system for $\mathcal{H}_\Sigma(@, \forall)$ is defined in Table 2 below. Note that, for any $s \in S$ and φ a formula of sort s we denote by $\vdash^s \varphi$ the fact that φ is provable. If Λ is an S -sorted set of $\mathcal{H}_\Sigma(@, \forall)$ formulas, then $\mathcal{H}_\Sigma(@, \forall) + \Lambda$ is the proof system obtained by extending $\mathcal{H}_\Sigma(@, \forall)$ with the formulas of Λ as additional axioms, closed under the deduction rules from Table 2. We write $\vdash_\Lambda^s \varphi$ to denote that φ is a theorem in the system $\mathcal{H}_\Sigma(@, \forall) + \Lambda$. Axiomatic extensions are essential to the formalization of our applications from Section 5.

If $\mathcal{M} = (W, (R_\sigma)_{\sigma \in \Sigma}, N, V)$ is a model, g is an assignment function, $s \in S$ and $w \in W_s$, then:

- $\mathcal{M}, g, w \models^s p$ iff $w \in V_s(p)$, for $a \in PROP_s$;
- $\mathcal{M}, g, w \models^s i$ iff $w = V_s(i)$, for $i \in NOM_s$;
- $\mathcal{M}, g, w \models^s c$ iff $w = N_s^{\mathcal{F}}(c)$, for $c \in N_s$;
- $\mathcal{M}, g, w \models^s x$ iff $w = g_s(x)$, for $x \in SVAR_s$;
- $\mathcal{M}, g, w \models^s \neg \varphi$ iff $\mathcal{M}, g, w \not\models^s \varphi$;
- $\mathcal{M}, g, w \models^s \varphi \vee \psi$ iff $\mathcal{M}, g, w \models^s \varphi$ or $\mathcal{M}, g, w \models^s \psi$;
- $\mathcal{M}, g, w \models^s \sigma(\varphi_1, \dots, \varphi_n)$ iff there exists $(w_1, \dots, w_n) \in W_{s_1} \times \dots \times W_{s_n}$ such that $(w, w_1, \dots, w_n) \in R_\sigma$ and $\mathcal{M}, g, w_i \models^{s_i} \varphi_i$ for all $1 \leq i \leq n$;
- $\mathcal{M}, g, w \models^s @_k^s \varphi$ iff $\mathcal{M}, g, u \models^t \varphi$, where k and φ have sort t and $V_t^N(k) = u$;
- $\mathcal{M}, g, w \models^s \forall x \varphi$ iff $\mathcal{M}, g', w \models^s \varphi$ for all $g' \rightsquigarrow^x g$.

Table 1: The satisfaction relation for the models of $\mathcal{H}_\Sigma(@, \forall)$

We further define the notions of *semantic* and *syntactic* consequence. In the sequel, we further assume that $s \in S$ and Γ be a set of formulas of sort $s \in S$. If $\mathcal{C}$ is a class of models, we write $\Gamma \models_{\mathcal{C}}^s \varphi$ to denote that φ is *semantically entailed* by Γ in all models from class $\mathcal{C}$: for any model $\mathcal{M}$, for any assignment g and for any $w \in W_s$, $\mathcal{M}, g, w \models^s \Gamma$ implies $\mathcal{M}, g, w \models^s \varphi$. Note that our treatment focuses on the notion of *local entailment*, as defined in [3, 1.5]. Recall that the *local syntactic consequence* $\Gamma \vdash_\Lambda^s \varphi$ is defined as follows: there exist formulas $\varphi_1 \dots \varphi_n \in \Gamma$, such that $\vdash_\Lambda^s (\varphi_1 \wedge \dots \wedge \varphi_n) \rightarrow \varphi$.

Finally, we state the *soundness theorem* with respect to axiomatic extensions. In the following sections, we will mechanize its proof in Lean.

Theorem 1 (Frame-soundness). *Let Λ be an S -sorted set of formulas, and let Γ be a set of formulas of sort $s \in S$. Then $\mathcal{H}(@, \forall) + \Lambda$ is sound, i.e. the following implication holds for all φ of sort s : $\Gamma \vdash_{\Lambda}^s \varphi$ implies $\Gamma \models_{Fr(\Lambda)}^s \varphi$.*

We assume that $s, t \in S$, $\sigma \in \Sigma$ has an appropriate arity, and $j, k \in NOM \cup N$, $x, y \in SVAR$, $\varphi, \psi, \varphi_1, \dots, \varphi_n$ are formulas of appropriate sorts.			
The axioms of $\mathcal{H}_{\Sigma}(@, \forall)$ are:			
Prop	All propositional tautologies		
K	$\vdash^s \sigma^{\square}(\varphi_1, \dots, \varphi \rightarrow \psi, \dots, \varphi_n) \rightarrow \sigma^{\square}(\varphi_1, \dots, \varphi, \dots, \varphi_n) \rightarrow \sigma^{\square}(\varphi_1, \dots, \psi, \dots, \varphi_n)$		
Dual	$\vdash^s \sigma(\varphi_1, \dots, \varphi_n) \leftrightarrow \neg \sigma^{\square}(\neg \varphi_1, \dots, \neg \varphi_n)$		
K@	$\vdash^s @_j^s \varphi_t \rightarrow \psi_t \rightarrow (@_j^s \varphi \rightarrow @_j^s \psi)$	SelfDual	$\vdash^s @_j^s \varphi_t \leftrightarrow \neg @_j^s \neg \varphi$
Agree	$\vdash^s @_k^s @_j^{s'} \varphi_t \leftrightarrow @_j^s \varphi_t$	Back	$\vdash^s \sigma(\dots, @_j^{s_i} \psi_t, \dots)_s \rightarrow @_j^s \psi$
Intro	$\vdash^s j \rightarrow (\varphi_s \leftrightarrow @_j^s \varphi_s)$	Ref	$\vdash^s @_j^s j$
Q1	$\vdash^s \forall x(\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \forall x \psi)$, if x does not occur free in φ		
Q2	$\vdash^s \forall x \varphi \rightarrow \varphi[y/x]$, if y is substitutable for x in φ		
Barcan	$\vdash^s \forall x \sigma^{\square}(\varphi_1, \dots, \varphi_i, \dots, \varphi_n) \rightarrow \sigma^{\square}(\varphi_1, \dots, \forall x \varphi_i, \dots, \varphi_n)$, if x does not occur free in φ_j for $j \neq i$		
Barcan@	$\vdash^s \forall x @_j \varphi \rightarrow @_j \forall x \varphi$	Nom	$\vdash^s @_k x \wedge @_j x \rightarrow @_k j$
Name	$\vdash^s \exists x x$		
The deduction rules of $\mathcal{H}_{\Sigma}(@, \forall)$ are:			
MP	If $\vdash^s \varphi \rightarrow \psi$ and $\vdash^s \varphi$, then $\vdash^s \psi$		
UG	If $\vdash^{s_i} \varphi_i$, then $\vdash^s \sigma^{\square}(\varphi_1, \dots, \varphi_i, \dots, \varphi_n)$		
BroadcastS	If $\vdash^s @_j^s \varphi_t$, then $\vdash^{s'} @_j^{s'} \varphi_t$		
Gen@	If $\vdash^s \varphi$, then $\vdash^{s'} @_j^{s'} \varphi$		
Name@	If $\vdash^s @_j^s \varphi$, then $\vdash^{s'} \varphi$, if $j \notin N$ and j does not occur in φ		
Paste	$\vdash^s @_j^s \sigma(\varphi_1, \dots, k, \dots, \varphi_n) \wedge @_k^s \varphi \rightarrow \psi$, then $\vdash^s @_j^s \sigma(\varphi_1, \dots, \varphi, \dots, \varphi_n) \rightarrow \psi$, for $k \neq j$, $k \notin N$ and k does not occur in φ, ψ , or the ... formulas		
Gen	If $\vdash^s \varphi$, then $\vdash^s \forall x \varphi$		

Table 2: The provability relation for $\mathcal{H}_{\Sigma}(@, \forall)$

3 Representing the Syntax

A central goal of the formalization was to obtain an *intrinsically sorted* representation of formulas, by clever use of Lean's dependent typing. In other words, we make no use of any "sort checking" function operating on syntax and defined separately from the syntax itself. Rather, any formula that can be constructed is *guaranteed by definition* to be well-sorted, making further checks redundant. Since our end-goal is to provide an interface between programming language syntax and many-sorted signatures, intrinsic sorting ensures that *programs are formulas*. More precisely, the programming language grammar is defined as a many-sorted signature; and, due to intrinsic sorting, valid parses in this grammar are guaranteed to be representable as hybrid formulas. We will show a few practical examples of how grammars define many-sorted signatures in section 5.1.

We employ an approach similar to the one laid out in [12], by fixing a set S of an arbitrary type α to stand for *sorts*, and letting all other families of symbols be dependent on S (operators $\langle \Sigma \rangle$, constant nominals N , nominals nom , propositional and state variables prop , svar). We show only the structure `Signature` α , and note we also define a structure `Symbols` α , made of a signature and the `prop`, `nom`, `svar` S -indexed families of symbols described:

```

structure Signature (α : Type u) where
  S      : Set α
  «Σ»    : List S → S → Set α
  N      : S → Set α

```

Yet we diverge from [12] and other dependently typed formalizations of many-sorted languages, like [14], by our treatment of *modal operators*. In this respect, we developed what we call *the list trick*, which we believe can be reused in other implementations of sorted languages in dependently-typed proof assistants. We describe it below.

3.1 The List Trick

As is usual, formulas are represented using an inductive type, each constructor corresponding to one kind of syntax: atomic symbols, logical connectives and operators, or modal applications. This type is parametrized by a variable `syms` : `Symbols α`, as defined earlier. Importantly, the type of formulas is further indexed by *a list of sorts* in the many-sorted signature. Usually, we only care about singleton lists of sorts. For example, the `at` constructor corresponds to the `@` operator, and it ensures that you can take a nominal k of sort t , a formula φ of the same sort t , and you may obtain a formula $@_k^s \varphi$ of any arbitrary sort s :

```

inductive FormL (syms : Symbols α) : List syms.signature.S → Type u
| at      : syms.nominal t → FormL syms [t] → FormL syms [s]
-- ...other constructors...

```

Yet we also introduce an artificial syntax constructor, `cons`, not present in the pen-and-paper definition of formulas. It is meant to build *lists* of formulas indexed by *non-singleton* lists of sorts, by *consing* (in the usual functional programming sense) a single sort formula onto a list of other sorted formulas:

```

| cons : FormL syms [s1] → FormL syms (s2 :: t) → FormL syms (s1 :: s2 :: t)

```

In our setting, this trick is a very natural choice. Recall that a modal operator in our many-sorted settings has *domain* sorts and a *range* sort: when an operator is applied to a list of formulas that are well sorted according to its domain, it yields a formula in its range sort. Using the `cons` constructor, we gain the ability to construct such heterogeneously sorted argument lists seamlessly. The sorts of the arguments are kept in the index of the `FormL` type. Therefore, the condition that the arguments are well sorted reduces to their index being identical to the domain of the operator. This is a verification that the Lean type checker itself can make, giving us intrinsic sorting. Our modal application constructor is, therefore, introduced like this:

```

| appl : syms.signature.Σ (h :: t) s → FormL syms (h :: t) → FormL syms [s]

```

By contrast, the approach taken in [12, 14] is to introduce the type of *heterogeneous lists*, i.e., lists of terms of possibly different types. In our case, this would amount to the following alternative formalization (see also [8] for more on `HList`):

```

inductive HList {α : Type v} (f : α → Type u) : List α → Type (max u v)
| nil  : HList f []
| cons : f x → HList f xs → HList f (x :: xs)
inductive Form {syms : Symbols α} : syms.signature.S → Type u
| appl : syms.signature.Σ (h :: t) s → HList Form (h :: t) → Form s

```

We think the *list trick* is a more appropriate formalization choice in contexts, like ours, where one will be prone to intensively recurse and perform induction over formulas. The reason

is that it is easier on the proof assistant's compiler: it requires neither nested, nor mutual inductive types. Such inductives currently lack convenient support for the `induction` tactic in Lean, and they create additional proof burden to show termination on recursive definitions.

The use of heterogeneous lists we showed above involved a nested inductive (`HList` mentioned within `Form`). Indeed, the authors of [14] acknowledge that "[an] issue arises [...] the argument list of the application constructor. [...] This makes recursive definitions and inductive proofs over the type difficult, if not impossible". Using the *list trick*, we obtain recursion and induction for free, at no expense to expressivity.

We still use a form of heterogeneous lists, disguised as heterogeneous *products*, through a type we called `WProd`. We will introduce this type in Section 4, where we go over the semantics.

Besides the list trick, all other constructors to `FormL` are standard. We defined list-like notation for the `cons` constructor: (φ, ψ) denotes `FormL.cons  $\varphi$   $\psi$` . Pen-and-paper formulas then correspond to singleton `FormL`'s:

```
def Form (syms : Symbols  $\alpha$ ) (s : syms.signature.S) := FormL syms [s]
```

We note that we made the choice to represent variables concretely, using their names; as opposed to other approaches common in the literature, such as de Bruijn indices:

```
| svar : syms.svar s → FormL syms [s]
| bind : syms.svar t → FormL syms [s] → FormL syms [s]
```

The intention was to keep the representation of syntax as close to its pen-and-paper counterpart as possible. Since flavors of hybrid logic with different binders can be defined (e.g., with an additional $\downarrow$ operator, or where the `@` operator can also bind variables), our representation of variables as concrete allows us to quickly iterate through these different versions without requiring additional engineering effort to make them compatible with our choice of variable naming. The point was not so much to define a *static formal artifact* of the logic, but rather to provide a *research tool* that the pen-and-paper logician can seamlessly migrate to and from when conducting research, with little representational overhead. We therefore also provide the usual pen-and-paper definitions for substitution, along with a decidable predicate `free_for` that checks whether a variable is substitutable for another in a given formula (as per Table 2, where these notions are also employed).

3.2 Mechanizing the Proof System

The generality of this system forces the pen-and-paper axiomatization (Table 2) to be somewhat non-rigorous, in particular with respect to axioms *K*, *Back*, *Barcan*, and the *UG* and *Paste* rules. Since modal operators are allowed to have arbitrary arities, notice that the symbol "... " is used to refer to *some* argument to an operator σ . Syntactically, the point is to perform *substitution* upon this argument. It is something specific to this polyadic framework that one is required to refer to operator arguments generically, and perform substitution upon them. While the meaning of "... " is evident to the mathematical reader, it becomes a genuine question of engineering how this mechanism of reference and substitution should be expressed in Lean. To this end, we introduce the notion of *contexts*², and *context substitution*.

Contexts. A term of type `Context  $\varphi$   $\psi$` can be seen as a *pointer* to the occurrence of formula φ within the formula (list) ψ . Its definition is given below. It is rather similar to the `member` type introduced in [8] with respect to `HLists`; and could also be seen as a zipper [13] on the `FormL` data structure.

²Note that our notion of *context* is different from "a term with one hole", which is the usual meaning.

```

inductive Context {syms : Symbols  $\alpha$ } {s : syms.signature.S} ( $\varphi$  : Form syms s) :
  FormL syms sorts  $\rightarrow$  Type u
| refl : Context  $\varphi$   $\varphi$ 
| head : Context  $\varphi$  ( $\varphi$ ,  $\psi$ )
| tail : Context  $\varphi$   $\psi \rightarrow$  Context  $\varphi$  ( $\chi$ ,  $\psi$ )

```

Notice that we do not define an inductive *proposition*, but rather an inductive *type*. That is, the existence of a term $C : \text{Context } \varphi \psi$ signifies not only *the fact* that formula φ occurs within ψ ; but it also carries the (unique) constructive description of *where* said occurrence is to be found. We exemplify this below, where we list all possible contexts of some formula in the list (φ, ψ, φ) .

```

variable ( $\varphi \psi$  : Form syms s)
example :  $\varphi.\text{Context } (\varphi, \psi, \varphi) := \text{Context.head}$ 
example :  $\psi.\text{Context } (\varphi, \psi, \varphi) := \text{Context.tail Context.head}$ 
example :  $\varphi.\text{Context } (\varphi, \psi, \varphi) := \text{Context.tail } <| \text{Context.tail Context.refl}$ 

```

Crucially, in the example above, the two different occurrences of φ within (φ, ψ, φ) correspond to two definitionally *different* $\varphi.\text{Context } (\varphi, \psi, \varphi)$ terms. Contexts, then, amount to a rigorous mechanism of specifying a formula within "... " notation. What is still required is to define the operation of *substituting* said formula with a different one. We use the notation $C[\varphi]$ for the term $\text{subst } C \varphi$. For brevity, we only show its type below; it takes a context C and a new formula φ , and returns the formula (list) obtained by replacing the argument that C points to with φ .

```

def subst : Context  $\varphi \psi \rightarrow$  Form sig s  $\rightarrow$  FormL sig sorts

```

With this machinery in place, consider our Lean characterization of axiom K. We provide below the respective constructor of the **Proof** type:

```

| k  $\varphi \psi \chi$  ( $C : (\varphi \rightarrow \psi).\text{Context } \chi$ ):
  Proof  $\Lambda s$  ( $\mathcal{H}(\sigma)^d \chi \rightarrow (\mathcal{H}(\sigma)^d C[\varphi] \rightarrow \mathcal{H}(\sigma)^d C[\psi])$ )

```

We express that $\varphi \rightarrow \psi$ must occur somewhere among the arguments to $\mathcal{H}(\sigma)^d$ by requiring a $C : (\varphi \rightarrow \psi).\text{Context } \chi$ term. We successively substitute that particular occurrence by φ and ψ , respectively, by applying the context substitutions $C[\varphi]$ and $C[\psi]$.

Axiomatic Extensions An important design question was that of axiomatic extensions. We defined sets of axioms simply as S -sorted sets of formula:

```

def AxiomSet (syms : Symbols  $\alpha$ ) := (s : syms.signature.S)  $\rightarrow$  Set (Form syms s)

```

Our **Proof** type is parametric on an axiomatic extension $\Lambda : \text{AxiomSet syms}$, which may also be empty. We ensure that all theorems generated by the axioms extended with Λ are **Proof** terms by adding an additional **ax** constructor:

```

| ax : ( $\varphi : \Lambda s$ )  $\rightarrow$  Proof  $\Lambda s \varphi$ 

```

This guarantees that not only formulas from Λ are taken as theorems, but also their closure under the application of any number of deduction rules, possibly also using axioms from the base system. While simple to implement, this approach is also very powerful, allowing both finitary and infinitary axiomatic extensions. In particular, in Section 5.2 we show how the proof system can be extended infinitarily by a number of *finitely specified* axiom *schemas*.

The Proof System, Mechanized. Since the full proof system led to a large Lean definition, below we display only an excerpt of the **Proof** inductive type, which suffices to showcase the hybrid nature of the language, and our design choices with respect to contexts.

```

inductive Proof {syms : Symbols  $\alpha$ } ( $\Lambda$  : AxiomSet syms) : (s : syms.signature.S) →
  Form syms s → Type u
| agree j k  $\varphi$  : Proof  $\Lambda$  s ( $\mathcal{H}@\mathbf{k}$  ( $\mathcal{H}@\mathbf{j}$   $\varphi$ )  $\longleftrightarrow$   $\mathcal{H}@\mathbf{j}$   $\varphi$ )
| q1 x  $\varphi$   $\psi$  ( $\_$  :  $\varphi$ .occurs_free x = false) : Proof  $\Lambda$  s ( $\mathcal{H}\forall\mathbf{x}$  ( $\varphi \longrightarrow \psi$ )  $\longrightarrow$  ( $\varphi \longrightarrow \mathcal{H}\forall\mathbf{x}$   $\psi$ ))
| q2_var (x y : syms.svarType s')  $\varphi$  :  $\varphi$ .free_for y x  $\rightarrow$  Proof  $\Lambda$  s (( $\mathcal{H}\forall\mathbf{x}$   $\varphi$ )  $\longrightarrow$   $\varphi[y // x]$ )
| name x : Proof  $\Lambda$  s ( $\mathcal{H}\exists\mathbf{x}$  ( $\mathcal{H}\mathbf{Var}$  x))
| barcan x ( $\varphi$  : Form syms s) ( $\sigma$  : syms.signature. $\Sigma$  _ s) (C :  $\varphi$ .Context  $\psi$ )
  (h : C.all_else_not_free x) :
  Proof  $\Lambda$  s ( $\mathcal{H}\forall\mathbf{x}$  ( $\mathcal{H}\langle\sigma\rangle^d \psi$ )  $\longrightarrow$  ( $\mathcal{H}\langle\sigma\rangle^d$  C[ $\mathcal{H}\forall\mathbf{x}$   $\varphi$ ]))
| paste {k : syms.nomType s2} { $\varphi$  : Form syms s2} (C : ( $\mathcal{H}\mathbf{Nom}$  k).Context  $\chi$ ) :
  .nom k  $\neq_s$  j  $\rightarrow$   $\psi$ .occurs k = false  $\rightarrow$  C[ $\varphi$ ].occurs k = false  $\rightarrow$ 
  Proof  $\Lambda$  s1 ( $\mathcal{H}@\mathbf{j}$  ( $\mathcal{H}\langle\sigma\rangle \chi$ )  $\wedge$   $\mathcal{H}@\mathbf{k}$   $\varphi \longrightarrow \psi$ )  $\rightarrow$  Proof  $\Lambda$  s1 (( $\mathcal{H}@\mathbf{j}$  ( $\mathcal{H}\langle\sigma\rangle$  C[ $\varphi$ ])
 $\longrightarrow \psi$ ))

```

In the following section, we will describe our treatment of the Kripke semantics for this logic.

4 Representing the Semantics. Models and Soundness

Let us begin by describing our approach regarding the S -sorted sets of worlds. Our frames map each sort to some type in a universe u :

```

structure Frame (signature : Signature  $\alpha$ ) where
  W : signature.S  $\rightarrow$  Type u
  -- ..other fields...

```

This represents the fact that *each term of type $W\ s$ is a world of sort s* . Recall, now, that each S -sorted operator σ is interpreted as a subset R_σ of the cartesian product of correspondingly sorted worlds. In order to define R_σ , we needed an auxiliary function that takes a list of sorts (σ 's domain and range) and W , and returns the *product type* of the list of sorts mapped through W :³

```

def WProd {sig : Signature  $\alpha$ } (W : sig.S  $\rightarrow$  Type u) : List (sig.S)  $\rightarrow$  Type u
| []      => PEmpty
| [s]     => W s
| s :: sorts => W s  $\times$  WProd W sorts

```

A term of type $ws : WProd\ W\ sorts$, where $sorts = [s, s_1, \dots, s_n]$ corresponds to an element of the cartesian product $W_s \times W_{s_1} \times \dots \times W_{s_n}$. Note that $WProd\ W\ sorts$ is definitionally equal to a product type when $sorts$ has at least two elements. This allows us to destruct such a $WProd$ into its first and second components, a fact that we exploit in the satisfaction relation for formula lists (more on this below).

Having the definition of $WProd$ on hand, the definitions of models and frames follow cleanly, and we list them in full below (certain restrictions we impose on W notwithstanding). The accessibility relation of an operator is introduced as a *set of $WProd$'s*, denoting all the sorted worlds that stand together in the respective relation. We define the valuation functions N_m , V_p and V_n , which specify the worlds where constant nominals, propositional symbols and non-constant nominals are satisfied, respectively:

³In order to define a total function, the edge case of $sorts$ being empty must be considered, but note that all operators have at least *one* sort, namely their **range**. This is a basic assumption of signatures both in the pen-and-paper version, and in our Lean formalization of it.

```

structure Frame (signature : Signature  $\alpha$ ) where
  W : signature.S  $\rightarrow$  Type u
  R : signature. $\Sigma$  dom range  $\rightarrow$  Set (WProd W (range :: dom))
  Nm : {s : signature.S}  $\rightarrow$  signature.N s  $\rightarrow$  W s
structure Model (syms : Symbols  $\alpha$ ) where
  Fr : Frame syms.signature
  Vp : syms.prop s  $\rightarrow$  Set (Fr.W s)
  Vn : syms.nom s  $\rightarrow$  Fr.W s

```

Following the slight generalization induced by the *list trick* described in Section 3, the satisfaction relation is also slightly generalized in the following sense. We define the *simultaneous satisfaction* of a formula list $\varphi_1, \dots, \varphi_n$ at a product of worlds $W_1 \times \dots \times W_n$, as the satisfaction of a FormL at a WProd, thus:

```

def Sat (M : Model syms) (g : Assignment M) (w : WProd M.Fr.W sorts) : FormL syms
  sorts  $\rightarrow$  Prop
| .cons  $\varphi$   $\psi$ s      => Sat M g w.1  $\varphi$   $\wedge$  Sat M g w.2  $\psi$ s

```

Once again, this list trick eases our treatment of modal applications. An operator σ applied to a formula *list arg* is satisfied at world w , if and only if, there exists a product $w' : WProd$ that is accessible from w , such that *arg* is simultaneously satisfied at w' . Formally:

```

| .appl  $\sigma$  arg      =>  $\exists w', \text{Sat } M \text{ g } w' \text{ arg} \wedge \langle w, w' \rangle \in M.Fr.R \sigma$ 

```

The rest of the satisfaction relation is standard and we will not list it in full. We use the notation $\langle M, g, w \rangle \models \varphi$ for $\text{Sat } M \text{ g } w \varphi$. We further defined entailment, validity in a class of models, validity in a class of frames, and validity in the class of frames determined by the axiom set Λ . We denote the latter as $\models_{Fr(\Lambda)} \varphi$.

4.1 Mechanizing Soundness

Using these definitions of syntax, proof system and semantics, we were successful in writing a complete, sorry-free formalization of the following soundness statements, the second of which being an immediate consequence of the first:

```

theorem Soundness { $\Lambda$  : AxiomSet syms} :  $\vdash(\Lambda, s) \varphi \rightarrow \models_{Fr(\Lambda)} \varphi := \text{by}$ 
theorem StrongSoundness { $\Lambda$  : AxiomSet syms} :  $\Gamma \vdash(\Lambda) \varphi \rightarrow \Gamma \models_{Fr(\Lambda)} \varphi := \text{by}$ 

```

In the process of formalizing soundness, we identified several errors in the original presentation of the proof system given in [15], namely with respect to axiom Barcan, and rules Name@ and Paste, which were missing some of the syntactic side conditions that are now shown in Table 2. We were able to construct a countermodel in Lean to the original version of Barcan, using the formalization presented in this paper. The proof system we presented in Section 2 corrects these issues, ensuring soundness. With this out of the way, the actual soundness proof is standard and follows by induction on Proof terms. Consider the proof for Name@:

```

| @nameAt s1 s2 j  $\varphi$  nocc $\varphi$  _ ih =>
  intro M g w
  let M' :=  $\langle M.1.v\_variant \ j \ w, \text{Set.Elem.v\_variant\_modelclass\_inv } \Lambda \ M \ j \ w \rangle$ 
  let g' : Assignment M'.1 := g.v\_variant j w
  let w' := (M'.1.Fr.WNonEmpty s1).default
  specialize ih M' g' w'
  simp only [Sat.at, M', v\_variant\_valuation] at ih
  rw [v\_variant\_agreement nocc $\varphi$  w]
  exact ih

```

It proceeds by defining the M' model as a valuation variant, while also introducing g' and w' definitions (identical to g and w , but made to match M' 's set of worlds). It then specializes the inductive hypothesis (namely, $\models_{Fr(\Lambda)}^s @_j \varphi$) to M' , g' , and w' , and after simplification with the satisfaction clauses of the $@$ operator, obtains $M', g', V'(j) \models^s \varphi$. Finally, using the variant agreement lemma, whose statement is shown below, it rewrites this to $M, g, w \models^s \varphi$, which is exactly the statement we intended to prove.

```
lemma v_variant_agreement {ws : WProd M.Fr.W sorts} {g : Assignment M} {j :
  symbs.nominal s} (h :  $\varphi$ .occurs j = false) (w : M.Fr.W s) :
  ( $\langle M, g, ws \rangle \models \varphi$ )  $\leftrightarrow$  ( $\langle M.v\_variant j w, g, ws \rangle \models \varphi$ ) :=
```

It states that, for any formula φ , we are free to alter the valuation of a nominal j in a model M without affecting the satisfaction of φ ; given that j does not occur in φ .

5 Applications

We showcase the possible applications of our system with three examples.

- The *SMC language* from [19, 15], for which we are able to perform Hoare-like verification for simple programs.
- *Logics for analysing security protocols*: the BAN logic, developed by Burrows, Abadi, and Needham [5] and an epistemic logic with actions, adapted from [16]. We analyze two concrete protocols: One Sided Secrecy, and the Needham-Schroeder Public Key Protocol (which is a standard example for any formal method approach of this domain).
- *The S5 modal logic*. We claim that the system we defined is modular enough to serve as a general-purpose *modal logic toolbox*, applicable even to cases where the full expressive power of the base system is unneeded. We axiomatize S5 in the strictly *modal* fragment of the language.

5.1 A DSL for Language Definitions

We provide a *DSL for defining many-sorted hybrid signatures* in familiar BNF-like notation. We start by briefly explaining the syntax and the elaboration process. All applications we will show in the subsequent sections will make use of our custom domain-specific syntax.

Signature definitions are introduced via the `hybrid_def` keyword. The definition may span over multiple lines, each of which declares the symbols corresponding to some sort. The following is a simple example declaring three sorts:

```
hybrid_def Ex :=
  sort One   ::= builtin Nat | "op1"(One, One)
  sort Two   ::= nom1 | nom2 | "op2"(Two)
  sort Three ::= subsort One
```

The `builtin` keyword declares a new constant nominal of sort `One` for each natural number expressed as a Lean term of type `Nat`. Additionally, `nom1` and `nom2` are declared as constant nominals of sort `Two`. Two modal operators are declared. First, we have `op1`, taking two formulas of sort `One` and returning a formula of sort `One`. Second, there is `op2`, which takes a formula of sort `Two` and returns another formula of sort `Two`. That is, even though the sorted signatures of this logic do not support subsorting, the declaration of `sort Three ::= subsort One` creates an operator `one2Three` which maps formulas of sort `One` onto formulas of sort `Three`. Internally,

this definition is elaborated as a `Symbols String` declaration, along with a `Signature String` declaration it uses internally: `#check Ex -- Ex : Symbols String`

Each of the constituting fields of the `Symbols` structure must be filled during elaboration. The following auxiliary declarations are also created and their names are accessible:

```
#check Ex.Sig    -- Ex.Sig : Signature String
#check Ex.Sorts  -- Ex.Sorts : Set String
#check Ex.Ops    -- Ex.Ops (domain : List ↑Ex.Sorts) (range : ↑Ex.Sorts) : Set String
#check Ex.CtNoms -- Ex.CtNoms (s : ↑Ex.Sorts) : Set String
```

The mechanism of elaborating these S -sorted sets is kept as simple as possible. `Ex.Ops` and `Ex.CtNoms` are both defined as actual *functions*, which decide membership in their respective sorted sets by checking boolean equality with the signature of each symbol. To exemplify this, we show the full elaborated definition of `Ex.Ops`:

```
def Ex.Ops : List ↑Ex.Sorts → ↑Ex.Sorts → Set String :=
  fun domain range => setOf ((fun domain range operator =>
    (bif domain == [Ex.One, Ex.One] && (range == Ex.One && operator ==
      "op1One_One_One") then true
    else
      bif domain == [Ex.Two] && (range == Ex.Two && operator == "op2Two_Two") then
        true
      else
        bif domain == [Ex.One] && (range == Ex.Three && operator == "one2Three")
    then true else false) =
    true) domain range)
```

While such definitions could be given by hand, thanks to DSL syntax the user is spared from the tedious and error-prone process of writing them.

5.2 Operational Semantics: SMC Machine

We begin by showing how the syntax and semantics of an imperative language can be defined within this framework. We focus on the SMC machine, introduced by Plotkin in [19]. Its syntax is similar to the textbook IMP language [20], while its small-step operational semantics makes explicit use of a lower-level *stack-memory-control* model. The hybrid treatment of this language originates in [15], our main target of Lean formalization. We specify the syntax of the SMC machine in Lean as a *many-sorted hybrid logic signature*; we provide a corresponding *axiomatic extension* to model its operational semantics; and we show how Hoare-like reasoning can be performed inside this formalization.

Describing the Grammar. Using our DSL, the grammar of the SMC machine was succinctly described, and automatically elaborated into a hybrid signature definition. For brevity, below we provide an excerpt. The syntax between square brackets [...] introduces a user-facing name for the defined operator (e.g., once elaborated, `PlusNat` is the name of a Lean declaration of a binary operation on sort `Nat`).

```
hybrid_def SMC :=
  sort Nat ::= builtin Nat | "_+_"(Nat, Nat)      [PlusNat]
  sort Stmt ::= skip
    | "_:=" (Var, AExp)
    | "if_then_else_"(BExp, Stmt, Stmt) [IteStmt]
    | "while_do_"(BExp, Stmt)           [WhileStmt]
    | "_;_"(Stmt, Stmt)                 [SeqStmt]
```

```

sort CtrlStack ::= "c"(Stmt)           [cStmt]
                | "_∪_"(CtrlStack, CtrlStack) [PDLUnion]
                | "_;"(CtrlStack, CtrlStack) [PDLSeq]
                | "*" (CtrlStack)           [PDLStar]
sort Config ::= "<_,_>"(ValStack, Mem) [mkConfig]
                | "[_]_"(CtrlStack, Config) [PDLop]

```

The code above elaborates to the declaration of a many-sorted signature SMC. Here and elsewhere, we abbreviate the type `Form SMC`, that of *formulas in the signature SMC*, by `SMCForm`; and similarly for signatures introduced in later sections.

Consider the `sort Stmt ::= if_then_else_(BExp, Stmt, Stmt)` production. It declares `if_then_else_` as a polyadic modal operator, taking as arguments formulas of sorts `BExp` (the branch condition), `Stmt` (the "true" branch) and `Stmt` (the "false" branch), and returns a `Stmt` (the if-then-else statement itself).

Yet it would obviously be burdensome to be forced to write statements in a programming language directly as raw modal logic operator applications. This is not the path we have chosen. Rather, we define additional syntax macros that translate high-level programming-like syntax to our low-level logic, enabling us to finally write programs (i.e., *sorted hybrid formulas*) exactly as one could provide as input to a regular compiler:

```

def incrementMax (x y aux : SMCForm Var): SMCForm Stmt :=
  if (x <= ++y) then
    aux ::= x;
    x   ::= y;
    y   ::= aux
  endif

```

The type of this declaration indicates that it defines a *hybrid formula of sort Stmt*, in the signature of SMC. As explained in Section 3, note that *programs are (well sorted) formulas*. Thanks to our intrinsically sorted approach, Lean's type checker guarantees that any valid formula of sort `Stmt` is a valid parse in SMC's grammar – at no additional cost to the user other than writing the signature definition and its syntax macros.

An Embedding of Hoare Logic. Towards Symbolic Execution. Syntax is not all: we care about semantics. As explained in Section 3.2, one may extend the base proof system by providing an *S*-sorted set Λ of additional axioms. Since SMC's operational semantics are *not* finitely axiomatizable, but are introduced as axiom *schemas*, it is another genuine question of proof engineering how SMC's infinite extension Λ should be formally specified.

We believe Lean's type theory provides an elegant solution to this: inductive types. To characterize some Λ by a finite number of schemas, one would start by defining an inductive family parametrized by formulas:

```

inductive Axiom : {s : Sorts} → SMCForm s → Type

```

It's not the type `Axiom` that we primarily care about; it is, rather, *its indices*. We declare axiom schemas as constructors for the `Axiom` type, an example being:

```

| APlusNat {n1 n2 : ℕ}: Axiom ((n1 +Nat n2) <=> (n1 + n2))

```

This schema constrains SMC's internal addition operator `+Nat` to act in accordance to its model, namely mathematical addition on natural numbers. (As one might expect, this term could be type checked only after a number of coercions were defined.)

What we obtain by this constructor is that *for each* $n_1 \ n_2 : \mathbb{N}$, a term of type `Axiom ((n1 + Nat n2) <=> (n1 + n2))` is defined. Hence, this constructor acts as an axiom *schema*, defining

an infinite number of `Axiom  $\varphi$` terms, for varying φ . So we take Λ precisely as the S -sorted set of formulas φ for which a term of type `Axiom  $\varphi$` exists:

```
def SMCA : AxiomSet SMC := λ _ => {  $\varphi$  | Nonempty (Axiom  $\varphi$ ) }
```

The proof systems we obtain by extensions of this kind with Λ are well suited for reasoning. In general we wrote helper definitions for easy access to the axioms, an example being:

```
def aid : SMCProof _ ((vs, set(mem, x, n)) → [c(x)] (n · vs, set(mem, x, n))) :=
  Proof.ax (<_, Nonempty.intro Axiom.Aid>)
```

Let us return to the `incrementMax` program we showed earlier. We make the following claim: if x is initialized to 0 and y is initialized to 2, then after the execution of `incrementMax`, x will have value 3 and y will have value 0 (assuming all variables are distinct). Formally, this amounts to the following statement:

```
def ifCorr (neq1 : x ≠ y) (neq2 : y ≠ aux) (neq3 : x ≠ aux) : SMCProof _
  ((vs, set(set(mem, x, (0:N)), y, (2:N))) →
   [c(incrementMax x y aux)] (vs, set(set(mem, x, (3:N)), y, (0:N)))) :=
```

Note, in particular, that the correctness statement above accounts for the side-effecting branch condition expression `x <= ++y`. To perform Hoare-like verification, we minimally require a structural rule to help us reason about if-then-else statements, with the additional property of allowing branch conditions to side-effect. Following the mathematical exposition in [15], we proved a number of such derived rules, with their respective derivations fully formalized in Lean. We exemplify by the rule of conditional below. Note the additional hypothesis `h1` which allows the evaluation of the boolean expression `b` to mutate memory:

```
def conditional {b : SMCForm BExp}
  (h1 : SMCProof _ ( $\varphi$  → [c(b)] (B · vs, mem)))
  (h2 : SMCProof _ ((vs, mem) ∧  $\mathcal{H} @ \text{true } B$  → [c(s1)]  $\chi$ ))
  (h3 : SMCProof _ ((vs, mem) ∧  $\mathcal{H} @ \text{false } B$  → [c(s2)]  $\chi$ )) :
  SMCProof _ ( $\varphi$  → [c(if b then s1 else s2 endif)]  $\chi$ ) :=
```

With these derived rules on hand, constructing the proof `ifCorr` becomes a matter of chaining `SMCA` axioms with Hoare-like structural rules. The underlying modal embedding is once again abstracted: whoever engineers the proof does *not* have to worry about using the low-level proof system from Section 3.2 directly. Further, proofs can be compositionally engineered, by first proving lemmas about subprograms, as we did for example by initially proving the correctness of the inner variable swap:

```
def swapCorrect (neq1 : y ≠ x) (neq2 : y ≠ aux) (neq3 : x ≠ aux) : SMCProof _
  ((vs, set(set(mem, y, yn), x, xn)) → [c(swapPgm x y aux)] (vs, set(set(set(mem,
  x, yn), aux, xn), y, xn))) :=
```

The work we display here with respect to Hoare verification focuses solely on concrete execution, as displayed in the examples above. A structural rule is proved in [15] under the name "Rule of Iteration", which would allow one to perform symbolic execution by proving loop invariants. This is currently an active area of proof engineering and still work-in-progress.

5.3 Verification of Security Protocols

We proceed by showcasing different logics for security protocols that can be embedded inside our hybrid modal machinery. We formalize BAN Logic [5], capturing belief modalities, freshness, jurisdiction and message-meaning rules, and a hybrid modal logic for protocols [16], integrating

dynamic actions (send and receive) with epistemic modalities, enabling reasoning about explicit knowledge and belief across protocols states.

An Embedding of BAN Logic. A fragment of BAN formulas is provided using our DSL, as follows (for brevity, we omit some productions from the presentation).

```
hybrid_def BAN :=
  sort Message ::= "↔"(Agent, Agent, Message) [shareKey]
    | "pk"(Agent) | "sk"(Agent)
    | "enc"(Message, Message) [encrypt] | "pair"(Message, Message)
  sort Formula ::= "≡"(Agent, Formula) [believes]
    | "ι"(Message) | "∼"(Agent, Formula) [oncesaid]
    | "◁"(Agent, Formula) [sees] | "#"(Message) [nonce]
    | "⇒"(Agent, Formula) [jurisdiction]
```

BAN deduction rules are encoded in the extension of the deduction system, and we provide one as example:

```
inductive Axiom : {s : Sorts} → BANForm s → Type
  | MMSK {i j : BANForm Agent} {m k : BANForm Message}
    : Axiom $ (i ≡ ι shareKeyOp i j k) → (i ▷ ι { m } k) → (i ≡ j |∼ ι m)
```

This rule states that if an agent knows that it shares a key with another one, and it sees (receives) a message encrypted with that key, then it may assume that the message was indeed sent by the agent with whom the key is shared.

By embedding the BAN rules into our formal framework, we obtain a mechanism for verifying authentication properties of security protocols. As an example, we prove that in the Needham-Schroeder public-key (NSPK) protocol, agent B can derive that agent A has received the nonce n_B and believes it to be fresh, hence that it was sent within the current protocol session:

```
def NSPK_mutual_authentication (p3 : BANProof _ $ B ▷ ι { nB } sk(A))
  (a1 : BANProof _ $ B ≡ ι pk(A)) (a2 : BANProof _ $ B ≡ # (nB)) :
  BANProof _ (B ≡ (A ≡ ι nB)) := by
  let h1 : BANProof _ := Proof.ax ⟨_, Nonempty.intro $ Axiom.MMPK⟩
  let h2 : BANProof _ $ (B ▷ ι { nB } sk(A)) → (B ≡ (A |∼ ι nB)) := Proof.mp h1 a1
  let h3 : BANProof _ $ B ≡ (A |∼ ι nB) := Proof.mp h2 p3
  exact NV h3 a2
```

An Embedding of Hybrid Logic for Security Protocols. We refer to [16] for a specific logic used in specifying and verifying security protocols, that can be implemented as a particular system of the current work, without any changes needed in representation of formulas. We provide only a fragment of the DSL used in our development:

```
hybrid_def Protocols :=
  sort Msg ::= "{_}_"(Msg, Msg) [encryption]
  sort Act ::= "send"(Agent, Agent, Msg) [send]
    | "recv"(Agent, Msg) [recv]
  sort StNom ::= "◁"(Agent, Msg) [explicit] | "□"(StNom, StNom) [comp]
  sort Prot ::= "B"(Agent, Prot) [believe] | "⟨_⟩"(StNom) [config]
    | "[_]"(Act, Prot) [action] | "X"(Agent, Msg) [explicitKnowledge]
```

Our goal is to analyze a property for the very simple protocol *OSS* (one-sided secrecy): $i \longrightarrow r : \{i, n\}_{pk(i)}$, where i and r are two agents, n is a message, and $pk(i)$ is the public key of the agent i . We formalize this protocol using the following two axioms:

```

| OSS1 : Axiom $ << (a < m) ⊔ γ >> → [send a, b({ m } pk(b))] B a, (X b, m)
| OSS2 : Axiom $ << γ >> → [recv b(m)] B b, (X a, m)

```

The first axiom states that, from a protocol configuration in which the initiator knows the message it intends to transmit, after it actually sends the message to the responder, the initiating agent believes that the newly transmitted message becomes explicitly known to the other agent. The second axiom studies the epistemic relation from the perspective of the receiving agent. We can now state and prove the following theorem: starting from the initial protocol state, after the message transmission, the responder knows that the initiator explicitly knows the sent message; this proof constitutes part of the mutual authentication claim.

```

def OSS : ProtocolsProof _ $ << (i < n) ⊔ γ0 i r >> → [(send i, r({ n }
pk(r)))]([recv r(n)](K r, (X i, n)))

```

5.4 The Modal Fragment. S5 Reasoning

Finally, we show that whenever the full expressive power of the logic is unneeded, one may use the same underlying formalism to reason about a less expressive fragment. To this end, we describe a *shallow embedding* of plain modal logic onto our *deep embedding* of many-sorted hybrid logic in Lean. We give a simple definition of the basic modal language:

```

hybrid_def ModalBase :=
  sort WFF ::= "◇"(WFF) [poss]

```

This declares a many-sorted signature `ModalBase`, containing a single (unary) operator $\Diamond$, the modal diamond. The operator $\Box$ is defined as syntax macro for the dual of $\Diamond$.

As is, this signature is that of *regular hybrid logic*, mono-sorted and monadic. We want, however, to go further and enforce the restriction that *no state symbols appear* in the modal fragment whatsoever, removing all hybridization from the language. The way to do this is straightforward. A fragment of the language is a *boolean predicate* on formulas (parametrized by the signature of the language):

```

def Fragment (syms : Symbols α) :=
  (ss : List syms.signature.S) → FormL syms ss → Bool

```

We define `IsBase`, the basic modal fragment, as `false` for all hybrid constructors:

```

def IsBase : Fragment ModalBase
| _, .prop _      => true
| _, .appl _ φ    => IsBase _ φ
| _, .or φ ψ      => IsBase _ φ && IsBase _ ψ
| _, .neg φ       => IsBase _ φ
| _, _           => false

```

We define `Modal` as the type of formulas in the signature `ModalBase`, which additionally come with a proof of `IsBase`. Next, a *proof* is in a certain syntactic fragment if *all its component subproofs* only reference formulas within that fragment. We express this by the following predicate, of which we show only the type, for brevity (the definition is by structural recursion on the proof):

```

def Proof.inFragment (P : Fragment syms) : Proof → s φ → Bool :=

```

So how can we define, say, S5? As before, we introduce the respective axiomatic extension (note that axiom K is not needed because it is already part of the primitive proof system):

```

inductive S5Schema : Modal → Type where
| AxT : S5Schema (□ φ → φ)
| Ax5 : S5Schema (◇ φ → □◇φ)
def S5 : AxiomSet ModalBase :=
λ _ => setOf ( λ form => ∃ (φ : Modal) ( _ : S5Schema φ), φ.toForm ≈ form )

```

An S5 proof is a regular proof from the axioms S5, which is *strictly in the basic modal fragment*. Formally:

```
def S5Pf (φ : Modal) := Proof.fragment IsBase S5 φ.1
```

Putting everything together, this embeds the syntax and proof system of S5 in many-sorted hybrid polyadic logic. To this extent, we display a sample S5 proof:

```

def posNecActual : S5Pf (◇□ φ → φ) := by
have 11 : S5Pf (◇(¬φ) → □◇(¬φ)) := ax_a5
have 12 : S5Pf (¬(□◇(¬φ)) → ¬(◇(¬φ))) := modusPonens contraP 11
have 13 : S5Pf ((◇□φ) → ¬(□◇ ¬φ)) := dni
have 14 : S5Pf ((¬(□◇(¬φ)) → ¬(◇(¬φ))) → (◇□φ → ¬(◇(¬φ)))) :=
modusPonens impTrans 13
have 15 : S5Pf (◇□ φ → □ φ) := modusPonens 14 12
have 16 : S5Pf ((□ φ → φ) → (◇□ φ → φ)) := modusPonens impTrans 15
have 17 : S5Pf (◇□ φ → φ) := modusPonens 16 ax_t
exact 17

```

6 Related Work and Conclusions

Formalizations of modal and hybrid logics in proof assistants have attracted sustained attention over the last decade. In the hybrid settings, [10] provides a formalization of a Seligman-style tableau system for hybrid logic in Isabelle/HOL, including soundness and completeness results, while [9] develops a constructive formalization of hybrid logic in Rocq. More recently, [18] presents a Lean formalization of the completeness proof for coalition logic with common knowledge, proving that canonical-model constructions can be mechanized in Lean. Our work differs from these developments since, first, we target a many-sorted polyadic hybrid modal logic, that significantly increases the complexity of both the syntax compared to mono-sorted settings; and since our emphasis lies on a uniform representation of signatures and axiomatic extensions that can be instantiated to concrete systems, rather than on a single fixed logic. In parallel, research on many-sorted logics has evolved along different lines, for example Matching μ -logic [6] offers a highly expressive framework for reasoning about program configurations and structural properties; its mechanization in Rocq [2] and the subsequent many-sorted Lean development [7] (for Applicative Matching Logic) show that sort-parametric reasoning can be effectively integrated into proof assistants. We also refer to [11] for a Lean-based framework used for verifying executable semantics.

The logical system mechanized in this work provides a rigorous foundation for future extensions. As future work we intend to prioritize: (1) the mechanization of the Completeness Theorem, as stated in [15]; (2) further support for Hoare-style reasoning, including loops and proof automation via VCG; and (3) devising tactics and constructions that facilitate the interactions with various users that might not be familiar with the details of our implementation.

References

- [1] Carlos Areces and Balder ten Cate. 14 Hybrid logics. In Patrick Blackburn, Johan Van Benthem, and Frank Wolter, editors, *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 821–868. Elsevier, 2007.
- [2] Péter Bereczky, Xiaohong Chen, Dániel Horpácsi, Tamás Bálint Mizsei, Lucas Peña, and Jan Tusil. Mechanizing Matching Logic in Coq. In Vlad Rusu, editor, *Proceedings of the Sixth Working Formal Methods Symposium, FROM 2022, "Al. I. Cuza University", Iasi, Romania, 19-20 September, 2022*, volume 369 of *EPTCS*, pages 17–36, 2022.
- [3] Patrick Blackburn, Maarten De Rijke, and Yde Venema. *Modal Logic*, volume 53. Cambridge University Press, 2001.
- [4] Patrick Blackburn and Jerry Seligman. What Are Hybrid Languages? In Marcus Kracht, Maarten de Rijke, Heinrich Wansing, and Michael Zakharyashev, editors, *Advances in Modal Logic*, pages 41–62. CSLI Publications, 1998.
- [5] Michael Burrows, Martin Abadi, and Roger Needham. A Logic of Authentication. *ACM Transactions on Computer Systems (TOCS)*, 8(1):18–36, 1990.
- [6] Xiaohong Chen and Grigore Rosu. Matching μ -logic. In *Proceedings of the Thirty fourth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019)*, pages 1–13. ACM, June 2019.
- [7] Horatiu Cheval and Bogdan Macovei. Matching Logic in Lean. *Technical report*, 2022. URL: <https://gitlab.com/ilds/aml-lean/MatchingLogic/>.
- [8] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, London, England, December 2013.
- [9] Christian Doczkal and Gert Smolka. Constructive Formalization of Hybrid Logic with Eventualities. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, pages 5–20, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [10] Asta Halkjær From, Patrick Blackburn, and Jørgen Villadsen. Formalizing a Seligman-Style Tableau System for Hybrid Logic: (Short Paper). In *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part I*, page 474–481, Berlin, Heidelberg, 2020. Springer-Verlag.
- [11] Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. Foundational Multi-Modal Program Verifiers. *Proceedings of the ACM on Programming Languages*, 10(POPL):2233–2264, 2026.
- [12] Emmanuel Gunther, Alejandro Gadea, and Miguel Pagano. Formalization of Universal Algebra in Agda. *Electronic Notes in Theoretical Computer Science*, 338:147–166, 2018.
- [13] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
- [14] Ádám Kurucz, Péter Bereczky, and Dániel Horpácsi. On a Dependently Typed Encoding of Matching Logic. In Andrei Arusoaie, Horațiu Cheval, and Radu Iosif, editors, *Proceedings 9th edition of Working Formal Methods Symposium, Iași, Romania, 17 September 2025*, volume 427 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–23. Open Publishing Association, 2025. doi:10.4204/EPTCS.427.1.
- [15] Ioana Leuştean, Natalia Moangă, and Traian Florin Şerbănuță. Operational Semantics and Program Verification Using Many-Sorted Hybrid Modal Logic. In Serenella Cerrito and Andrei Popescu, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 446–476, Cham, 2019. Springer International Publishing. URL: <https://arxiv.org/abs/1905.05036>.
- [16] Ioana Leuştean and Bogdan Macovei. Hybrid Many-Sorted Modal Logic with Nominal Terms. In *Relational and Algebraic Methods in Computer Science: 22nd International Conference, RAMiCS 2026, Będlewo, Poland, April 7–10, 2026, Proceedings*, page 248–265, Berlin, Heidelberg, 2026. Springer-Verlag.
- [17] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated De-*

- duction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag.
- [18] Kai Obendrauf, Anne Baanen, Patrick Koopmann, and Vera Stebletsova. Lean Formalization of Completeness Proof for Coalition Logic with Common Knowledge. In Yves Bertot, Temur Kutsia, and Michael Norrish, editors, *15th International Conference on Interactive Theorem Proving (ITP 2024)*, Leibniz International Proceedings in Informatics, LIPIcs, pages 1–18. Schloss Dagstuhl-Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, September 2024.
- [19] Gordon David Plotkin. A Structural Approach to Operational Semantics. *The Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. Structural Operational Semantics.
- [20] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, February 1993. doi:[10.7551/mitpress/3054.001.0001](https://doi.org/10.7551/mitpress/3054.001.0001).