Decision and Complexity of Dolev-Yao Hyperproperties (Technical Report)

Itsaka Rakotonirina, Gilles Barthe, Clara Schneidewind

Decision and Complexity of Dolev-Yao Hyperproperties (Technical Report)

ITSAKA RAKOTONIRINA, MPI-SP, Germany
GILLES BARTHE, MPI-SP, Germany and IMDEA Software Institute, Spain
CLARA SCHNEIDEWIND, MPI-SP, Germany

The formal analysis of cryptographic protocols traditionally focuses on trace and equivalence properties, for which decision procedures in the symbolic (or Dolev-Yao, or DY) model are known. However, many relevant security properties are expressed as DY hyperproperties that involve quantifications over both execution paths and attacker computations (which are constrained by the attacker's knowledge in the underlying model of computation). DY hyperproperties generalise hyperproperties, for which many decision procedures exist, to the setting of DY models. Unfortunately, the subtle interactions between both forms of quantifications have been an obstacle to lifting decision procedures from hyperproperties to DY hyperproperties.

The central contribution of the paper is the first procedure for deciding DY hyperproperties, in the usual setting where the number of protocol sessions is bounded and where the equational theory modelling cryptography is subterm-convergent. We prove that our decision procedure can decide the validity of any hyperproperty in which quantifications over messages are guarded and quantifications over attacker computations are limited to expressing the attacker's knowledge. We also establish the complexity of the decision problem for several important fragments of the hyperlogic. Further, we illustrate the techniques and scope of our contributions through examples of related hyperproperties in the smart-contract field.

CCS Concepts: • **Security and privacy → Logic and verification**; **Formal security models**; • **Theory of computation → Cryptographic protocols**.

Additional Key Words and Phrases: hyperproperties, security protocols, computational complexity

1 INTRODUCTION

Cryptographic protocols are interactive distributed algorithms designed to achieve a set of security goals in the presence of active adversaries. These protocols are the cornerstone of computer security: they are widely deployed to support authentication and secure communications and emerging applications such as blockchains. They are however complex and error-prone, and hence a prime target for formal verification. In fact, there is already a large body of work developing foundations and practical tools for their analysis, see e.g. [Blanchet 2012].

The common substrate of many of these works is the *symbolic model* of cryptography, rooted in the seminal work of Dolev and Yao [Dolev and Yao 1983]. Its crux is an adversary that can observe protocol executions, carry arbitrary computations from the values learned during observation, and intercept, tamper, or forge messages exchanged across the network. The adversary is embedded into the model by means of an *adversarial semantics* of protocols, which can then be used as a basis to prove that these protocols achieve a security property for all possible behaviours of adversaries and coalitions of dishonest participants. Traditional security properties are *trace properties* that are defined as universal quantifications over execution (*traces*) w.r.t. the semantics, possibly relying on an epistemic temporal logic featuring a modality K for the adversary's knowledge. Surprisingly, despite general undecidability results, their verification for all adversaries is decidable under reasonable restrictions on cryptographic protocols and on the properties. Existing mature tools typically focus on trace properties for the most part [Armando et al. 2012; Basin et al. 2019; Blanchet et al. 2020]. Still, not all security properties of interest fall into this class.

Authors' addresses: Itsaka Rakotonirina, itsaka.rakotonirina@mpi-sp.org, MPI-SP, Bochum, Germany; Gilles Barthe, gbarthe@mpi-sp.org, MPI-SP, Bochum, Germany and IMDEA Software Institute, Madrid, Spain; Clara Schneidewind, clara.schneidewind@mpi-sp.org, MPI-SP, Bochum, Germany.

In a classic setting [Alpern and Schneider 1985], (trace) properties are defined as sets of traces. As such, they are special instances of *hyperproperties*, which are defined as properties of the set of traces itself—that is, they are sets of sets of traces [Clarkson and Schneider 2010]. Hyperproperties provide a rich setting for modelling and reasoning about systems because, in short, they display the additional ability to express relations between several executions of a protocol. This is an important ingredient to formalise, typically, strong notions of information leakage or simulability predicates. Notably, the specification and verification of hyperproperties can be achieved through hyperlogics, such as HyperCTL* [Clarkson et al. 2014], which extends CTL* with quantifiers over execution traces. Decidability and complexity of the model-checking and synthesis problems for such hyperlogics have been studied extensively [Clarkson et al. 2014; Coenen et al. 2019; Finkbeiner et al. 2015; Hsu et al. 2023; Mascle and Zimmermann 2019]. However, these results are not applicable to symbolic models of cryptography, as they do not consider adversarial semantics. Meanwhile, tools operating in the symbolic model still do not handle such hyperproperties beyond some fixed notions of process equivalences [Basin et al. 2019; Blanchet et al. 2020; Cheval et al. 2020b].

In a recent work, [Barthe et al. 2022] introduces Hypertidy CTL*, a temporal logic for modelling and reasoning about hyperproperties in (timed) symbolic models of cryptography. Hypertidy CTL* extends HyperCTL* with clauses to reason about, and quantify over, adversarial computations. Unlike HyperCTL* where the model checking of arbitrary hyperproperties is decidable [Clarkson et al. 2014], the presence of the adversary makes the problem undecidable in general in Hypertidy CTL*. Currently, there is no decision procedure for any non-trivial fragment of this logic beyond trace properties, nor even incomplete methods that could serve as the basis of a future automated tool.

Contributions

The main contribution of this paper is a decision procedure for hyperproperties in a symbolic model of protocols including real-time and a global state. We see this as a first theoretical foundation of DY hyperproperty analysis, in the same way as the development of now-mature symbolic tools benefited from a large amount of more theoretical work on decidability and complexity [Cheval et al. 2013, 2018; Durgin et al. 2004; Kanovich et al. 2014; Rusinowitch and Turuani 2003]. Our procedure takes as input a hyperproperty φ and a model of the cryptographic protocol P, and decides whether the hyperproperty holds for all adversaries interacting with P. Our decision procedure makes two requirements, described below.

First, we require that the hyperproperty φ is a formula of a fragment of Hypertidy CTL* whose quantifications over adversarial computations are restricted to model the adversary's knowledge. Alternatively, this fragment can be seen as an extension of HyperCTL* with adversarial executions, real-time, predicates to reason about global states, and a standard epistemic knowledge operator K. Similarly to HyperCTL*, this allows unrestricted quantifications over execution paths. Additionally, we require that quantifiers over messages are *guarded*, i.e., should only be used to make reference to terms appearing in the protocol—and not to express arbitrary first-order formulae. These assumptions are classic in verification tools for trace properties [Basin et al. 2019], although notably excluding cryptographic notions of indistinguishability properties [Barthe et al. 2022; Cheval et al. 2018]. While this is a serious restriction of our decision procedure, we contend that it delineates the focus of our technical investigation to the interactions between adversarial computations and trace quantifications, which require specific treatment.

Second, we require that the protocol P is written in the *bounded* fragment of (a timed and stateful variant of) the applied π-calculus [Abadi et al. 2018], and that the equational theory used to model cryptography is *subterm-convergent*. Both assumptions are very common, and are adopted by many decision procedures [Cheval et al. 2013, 2018; Rusinowitch and Turuani 2003]. In particular, the boundedness assumption imposes an explicit bound on the number of protocol sessions of P, but the

potential interactions with the adversary remain unbounded. It is justified by theoretical reasons (because verification is generally undecidable for arbitrarily many sessions), and by practical reasons (because most security flaws of concrete protocols can be exhibited with a few sessions). Technically, the assumption ensures that protocol executions are bounded in depth, which is the main reason why we can express the validity of $P \models \varphi$ as a finite set of *hyperconstraints*, a core notion that we introduce. As a complement, the use of *destructor subterm convergent rewriting systems* to model cryptography is used to decide adversarial knowledge, i.e., whether a term can be deduced by the adversary at some point in a protocol execution. It encompasses many primitives such as hash functions, encryption, signatures, or zero-knowledge proofs [Abadi and Cortier 2006].

Technical Developments. At a technical level, getting inspiration from other decision procedures for bounded processes in different contexts [Cheval et al. 2013, 2018; Liu and Lin 2012], our own approach consists of two interleaving components:

(1) *constraint generation*: we extract a set C of *hyperconstraints* from the decision problem $P \models \varphi$;
(2) *constraint solving*: we establish the existence of *solutions* for the constraints of C.

The main challenge and technical innovation lies in the fact that, unlike any related work in this context, the interaction between path quantifications and adversarial computations requires carefully updating the constraints of C whenever a new path quantifier is handled. We deal with these updates by manipulating a *stack* of hyperconstraints, each stack level representing a different path quantifier. The effect of solving the constraints of the most recent level n of the stack is thus propagated to previous levels $i < n$, typically in the case where the path at level n imposes constraints on some adversarial computations introduced at level i. We establish the soundness and completeness of our constraint generation, and show that the generated constraint set satisfies a *small model property*, i.e., it has a solution *iff* it has a solution of size smaller than some k. Here k can be computed effectively and is exponential in the parameters of the problem (P and φ among others). Making the link between the extracted constraints and the actual statement $P \models \varphi$, this naturally translates into a decision procedure whose complexity ranges over the exponential hierarchy of complexity (**EXPH**(poly)). This complexity bound is tight, as the verification problem for even stricter fragments of the logic are already known to be **EXPH**(poly) hard [Barthe et al. 2022].

Additional Contributions. A second contribution of the paper is an extensive study of various fragments and extensions of the framework. On the one hand, we provide exact complexity bounds for several fragments of the logic: we target selected equational theories (e.g., empty or limited to one free symbol), and several sub-fragments (e.g., tidy LTL or Hypertidy LTL). This highlights the impact on complexity of the different parameters of the problem, and we hence argue that it contributes to a better theoretical understanding of hyperproperty verification. On the other hand, we establish undecidability for several generalisations of our main theorem, thus illustrating the minimality of our assumptions. A summary of our results can be found in Figure 9 later in the paper, when all necessary notions are introduced (end of Section 4.3). We recall that our results are restricted to a fragment not supporting cryptographic notions of equivalence, and thus leave the case of the full Hypertidy CTL* logic open. We return to this open problem in the conclusion.

In addition to our technical contributions, we detail several case studies of interest of our procedure. They model various (hyper)properties of interest including notions of fairness, simulation, reentrancy freedom, as well as some correctness properties involving several trace quantifications. Although some of these notions have already been studied in the literature, our contribution is the ability to model these properties in scenarios involving coalitions of fully dishonest participants

and an active adversary controlling communications. Most of such proofs cannot be handled or supported by any existing approach, highlighting the theoretical interest of our contributions.

2 MOTIVATING EXAMPLE: FAIR REWARD

To give a general idea of the expressivity of our model and of the scope of our contributions, we first present a prototypical example of a Dolev-Yao hyperproperty, in the context of distributed programs running on top of a blockchain-based cryptocurrency (so-called *smart contracts*). It will be used throughout the paper as an example support for our technical definitions.

2.1 Blockchain and Smart Contract

At a high-level, blockchain-based cryptocurrencies enable networks of mistrusting users to reach a consensus on the execution of financial transactions. A joint transaction log (the *blockchain*) is gradually extended by ordering authenticated transactions into blocks. However, transactions do not only authorize direct asset transfers but also advanced asset redistribution logics (*smart contracts*). Users trigger the contract's behaviour by posting a transaction with the call details; it is then executed once this transaction is appended (published) to the blockchain. Notably, between a transaction's submission and its publication, it already constitutes public information—and other transactions leveraging this information may be scheduled in this time window.

```
reduc verify(sign(x,y,z),x,pk(z)) -> ok.
formula hyp_publish(pi:trace) =
  G (∀x:bitstring. Publish(x)pi =>
    (G not Publish(x)pi) /\
    verify(x.sig,x.data,x.sender) = ok)
formula hyp_submit(pi:trace) =
  ∀x:bitstring. F Submit(x)pi =>
    (F Publish(x)pi) \/
    verify(x.sig,x.data,x.sender) ≠ ok
```

Fig. 1. Simple blockchain model

We axiomatise a simplified blockchain in Figure 1. The submissions and publications of transactions `tx` are represented by *events* `Submit(tx)` and `Publish(tx)`, mentioned in the blockchain's properties (`formula`). These are trace properties, that is, they refer to events happening in a single trace `pi`, as indicated by the subscripts `pi` on events. Here, transactions are tuples `tx = ((tx.sender,tx.func,tx.args),tx.sig)` storing, resp., the public key of the sender, the name of the contract function to invoke, its arguments, and the signature of the transaction's data `tx.data = (tx.sender,tx.func,tx.args)`. Signatures are modelled by a set of uninterpreted symbols `pk(skey)` (public key) and `sign(msg,rnd,skey)` (randomised signature), and a *rewriting rule* (`reduc`) defining verification. The two `formula` then formalise the (trace) properties of our simplified blockchain, namely that transactions should be signed and only appended once (`hyp_publish`), and that well-signed submissions are eventually published (`hyp_submit`). For `hyp_submit`, publication may either precede the submission—thus invalidating it due to `hyp_publish`—or follow it. The two `formula` notably use the (strict) temporal operators `F` ("finally") and its dual `G` ("globally"). In real-time contexts, they intuitively read as "in the *strict* future, it eventually (resp. always) holds that...". The axiomatization therefore captures that valid transactions are guaranteed to be published eventually, but not necessarily in the order of their submission.

2.2 Fair Reward Contracts

We now describe a smart contract, `Puzzle`, implementing a simple hash puzzle letting a user who knows the preimage of a specific hash value `puzzle = h(sol)` claim a (monetary) prize. A user A can claim the prize by submitting a transaction invoking a function of the contract (that we call `solve`) with `sol` as the argument. Upon inclusion of such a transaction to the blockchain, the contract checks whether `sol` is the expected preimage, in which case it transfers the prize funds to the sender.

Figure 2 describes a blockchain where `Puzzle` is deployed, `prize` referring to the numeric prize value. It is written in our framework's calculus, using a more readable programming-like syntax with an intuitive interpretation (detailed in Appendix B for completeness). In line with the terminology of Figure 1, this process therefore receives transactions as inputs `tx` from the network, and appends them to the blockchain (event `Publish`, which is notably constrained by the properties of Figure 1). When `tx` contains a call to the `solve` function, the process also executes it accordingly.

```
in(tx): Publish(tx):
if tx.func = solve:
  if h(tx.args) = puzzle:
    balance[tx.sender] += prize:
    prize = 0
  else: skip
else: skip
```

Fig. 2. Model of the contract `Puzzle`

```
let d = (pk(sk),solve,sol) in
let tx = (d,sign(d,rand(0),sk)) in
out(tx): Submit(tx)
```

Fig. 3. Honest user submitting to `Puzzle`

A user A submitting a solution `sol` to this contract is therefore described by the process of Figure 3. We use in particular the previous `Submit` event and `sign` signature function. The randomness of the signature is handled by a symbol `rand`, modelling a (deterministic) pseudo-random generator, called a *private function symbol* in our framework, passed here with an arbitrary seed `0`. Unfortunately, the `Puzzle` contract is inherently unfair as it is vulnerable to the following attack which allows an adversary to claim the reward without prior knowledge of `sol`: (1) an honest user computes and submits a transaction `tx` to fetch `sol` to the `solve` function; then (2) before `tx` gets effectively appended to the blockchain through the consensus mechanism, it becomes public to all network participants. In particular, a dishonest user may read `tx` to learn `sol` before computing their own transaction `tx'`. The transaction `tx'` may then be published before `tx` (even though submitted later). This execution is valid in that it complies to the blockchain's axiomatisation of Figure 1 (valid transactions have been published, although delayed). Such attacks are possible in practice because the blockchain consensus mechanism only ensures that (valid) transactions are eventually included but does not guarantee their order within blocks. Note that the attack requires the adversary to perform active transaction computations when interacting with the system, which is the crux of Dolev-Yao models.

```
in(tx): Publish(tx):
if tx.func = commit: @t:
  if t < timeout:
    T[tx.sender] = tx.args
  else: skip
else if tx.func = release: @t:
  if t > timeout
     && h(tx.args) = puzzle
     && h((tx.args,tx.sender))
        = T[tx.sender]:
    balance[tx.sender] += prize;
    prize = 0
  else: skip
else: skip
```

Fig. 4. Contract `FairPuzzle`

Figure 4 describes a process model of a refined contract `FairPuzzle` preventing such exploits of honest transactions. Users are required to send *commitments* (under the form of hashes including their identity) to their solution in an initial phase, in turn stored in a table `T`. The commitment are subsequently opened in a later *release* phase. The transition into this phase occurs at time `timeout`, which is indicated by the checks against the current time `t`, recorded through the timestamping instructions `@t`.

2.3 Proving Fair Reward

Let us call *fair reward* the resilience of a protocol against the above kind of attacks. We can define it by comparing the possible executions of a smart contract with idealised executions where transactions are included upon submission, i.e., without getting reordered. Rephrasing: "*for all executions π_1, there exists an execution π_2 (with the same submissions, but possibly in a different order) whose valid submissions are immediately followed by their publications, and such that π_1 and π_2 have the same final balance of all users*".

```
formula hyp_block(pi:trace) = // blockchain model
  hyp_publish(pi) /\ hyp_submit(pi)
formula ideal(pi:trace) = // ideal executions
  G (∀x:bitstring. Submit(x)pi =>
    F Publish(tx)pi =>
    silentpi U Publish(tx)pi)
formula fair_reward = // security hyperproperty
  ∀pi1:trace. hyp_block(pi1) =>
    G (Setuppi1 => ∃pi2:trace. hyp_block(pi2) /\
      ideal(pi2) /\
      (∀x:bitstring.F Submit(x)pi1 <=> F Submit(x)pi2) /\
      F (G silentpi1 /\ G silentpi2 /\
        ∀x:bitstring. balancepi1[x] = balancepi2[x]))
```

Fig. 5. Formalisation of fair reward

This notion is formalised in Figure 5. It assumes in particular that the theoretical model of the protocol involves a setup phase letting the adversary choose the attack parameters (prize value, number of participants, trust scenario...), and whose end is indicated by the `Setup` event. After the setup phase, the quantification over `pi2` requires the existence of the ideal trace simulating `pi1`. In particular, immediate publication is modelled by the fact that `pi2` should satisfy `silent` until (`U`) publication, where `silent` means that no particular action is occurring at the current time. Similarly, the balances of all users are compared in the final states, i.e., once executions are "silent forever".

Note that, despite `FairPuzzle` being a reasonably simple contract, establishing fair reward is not trivial in that it requires several technical arguments about the commitment management. Typically:

(1) commitments critically include the public key of the sender (`tx.sender` in Figure 4). Otherwise, an adversary could replay a commitment by an honest user during the commitment phase and then frontrun their `release` transaction;
(2) commitments should only be accepted during a proper commitment phase, here modelled by a timeout. Otherwise, when an honest user submits a release, the adversary learns the solution and may then both commit and release it before the honest one gets through;
(3) users should check that their commitment has effectively been published before releasing. Otherwise, an adversary may get it artificially rejected by swapping the two transactions.

These three examples are typical executions π_1 which cannot be simulated by any ideal execution π_2. Such subtleties highlight the necessity of a careful, exhaustive and systematic reasoning for proving that fair reward is indeed enforced by the contract's code and the honest participants' behaviour. We illustrate, all across the paper, how `FairPuzzle` is verified in our framework, and how to encode its various programming mechanisms (e.g., mutable tables) in our symbolic model.

3 PROTOCOL MODEL

We model our protocols in a timed and stateful variant of the bounded applied π-calculus. The model is a refinement from [Barthe et al. 2022]; one difference is that our model features a notion of atomic composition, which eases the modelling of timing aspects, and supports a notion of global state that can be used to model mutable variables.

3.1 Cryptographic Primitives and Messages

Term Algebra. The algebraic setting we use to model cryptographic primitives is largely standard. We start with *atomic values*:

$$\mathsf{A} = \mathcal{N} \uplus \mathcal{X} \qquad \text{with} \qquad \mathcal{N} = \mathcal{N}_{pub} \uplus \mathcal{N}_{priv} \qquad \mathbb{R} \subset \mathcal{N}_{pub} \qquad \mathcal{X}^{\mathsf{N}} \subset \mathcal{X}\,.$$

The set $\mathcal{N}$ of *names* is used to model most values used in protocols: *public names* ($\mathcal{N}_{pub}$) may for example model identities, public constants, or numeric values such as time ($\mathbb{R}$), while *private names* ($\mathcal{N}_{priv}$) model data unknown to the adversary such as large secret cryptographic material. The set $\mathcal{X}$ then contains the *variables*, including a distinguished subset $\mathcal{X}^{\mathsf{N}}$ of *numeric variables*

used to specifically bind numeric values. We assume that $\mathcal{N}_{priv}$, $\mathcal{N}_{pub} \setminus \mathbb{R}$, $\mathcal{X}^{\mathsf{N}}$ and $\mathcal{X} \setminus \mathcal{X}^{\mathsf{N}}$ are infinite. We write $vars(u)$, (resp. $vars^n(u)$, $names(u)$) the set of variables (resp. numeric variables, names) appearing in an arbitrary object u. Next, we introduce *signatures*, which are finite sets $\mathcal{F} = \{\mathsf{f}/n,\ \mathsf{g}/m,\ \mathsf{h}/p, \ldots\}$ used to represent cryptographic primitives. A symbol f/n models a primitive taking n arguments. We consider two partitions of $\mathcal{F}$:

$$\mathcal{F} = \mathcal{F}_{\mathsf{c}} \uplus \mathcal{F}_{\mathsf{d}} = \mathcal{F}_{pub} \uplus \mathcal{F}_{priv}\,.$$

On the one hand, $\mathcal{F}_{\mathsf{c}} \uplus \mathcal{F}_{\mathsf{d}}$ distinguishes *constructors* (used to construct messages) from *destructors* (used to pattern-match constructors, with possible failure). On the other hand, $\mathcal{F}_{pub} \uplus \mathcal{F}_{priv}$ separates *public* symbols from *private* ones, the latter specifically modelling operations the adversary is not allowed to perform (e.g., protocol-specific oracles). Finally, protocol messages are modelled by:

Definition 3.1 (Term)**.** A *term* is an atomic value $a \in \mathsf{A}$, or a function symbol f or an arithmetic operator ($+, \times, -, \ldots$), applied to other terms while respecting arities. We write $\mathcal{T}(S)$ the set of terms built from $S \subseteq \mathsf{A} \cup \mathcal{F} \cup O$, for a given set of arithmetic operators O. We call a term $u \in \mathcal{T}(S)$: (1) *constructor* if $S \cap \mathcal{F}_{\mathsf{d}} = \emptyset$; (2) *public* if $S \cap \mathcal{N}_{priv} = \emptyset$ and $S \cap \mathcal{F}_{priv} = \emptyset$; (3) *ground* if $S \cap \mathcal{X} = \emptyset$; (4) *numeric* if $S \subseteq \mathbb{R} \cup \mathcal{X}^{\mathsf{N}} \cup O$.

In particular, our model is implicitly typed in that some terms represent numeric values such as time, and others, cryptographic material; our definitions thus often involve related typing conditions. Finally, the notion of *substitutions* σ is defined as usual:

$$\sigma = \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}$$

is a mapping where $dom(\sigma) = \{x_1, \ldots, x_n\}$ are pairwise distinct variables forming the *domain* of σ, and $u_1, \ldots, u_n$ are terms; σ is then *well-typed* if, for all i such that $x_i \in \mathcal{X}^{\mathsf{N}}$, u_i is a numeric term. The application of a substitution σ to a term t is denoted by $t\sigma$ which is the term obtained by replacing all occurrences of x_i in t by $u_i = x_i\sigma$. Also, more specifically, we say that a term C is a *context* if, writing $vars(C) = \{x_1, \ldots, x_n\}$, each x_i appears exactly once in C. In this case, we write:

$$C[u_1, \ldots, u_n] \qquad \text{instead of} \qquad C\sigma, \text{ provided } \sigma = \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\} \text{ is well-typed.}$$

In particular, we say that u is a *subterm* of v if there is a context C such that $v = C[u]$. It is a *strict subterm* if, in addition, $v \neq u$. Regarding notations, we write $\sigma \cup \sigma'$ the substitution of domain $dom(\sigma) \cup dom(\sigma')$ that extends both σ and σ' (provided they coincide on $dom(\sigma) \cap dom(\sigma')$); and $\sigma\sigma'$ refers to the composition $\sigma' \circ \sigma$, i.e., for all terms t, $t(\sigma\sigma') = (t\sigma)\sigma'$.

Rewriting. Terms have an operational behaviour modelled by:

Definition 3.2 (Rewriting)**.** A *rewriting system* $\mathcal{R}$ is a finite set of *rewrite rules* $\ell \to_{\mathcal{R}} r$, where ℓ, r are terms without arithmetic operators, with $vars(r) \subseteq vars(\ell)$. Additionally, we assume that $\mathcal{R}$ is:

(1) *destructor*, i.e., for all $(\ell \to_{\mathcal{R}} r)$, $\ell = \mathsf{f}(u_1, \ldots, u_n)$ with $\mathsf{f} \in \mathcal{F}_{\mathsf{d}}$, $u_1, \ldots, u_n, r$ constructor terms;
(2) *subterm*, i.e., for all $(\ell \to_{\mathcal{R}} r)$, r is either a strict subterm of ℓ, or a ground term;
(3) *convergent*, i.e., for all terms u, there exists a unique *normal form* $u{\downarrow}$ of u, namely a term irreducible by $\to_{\mathcal{R}}$ such that $u \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} u{\downarrow}$.

Such rewriting systems can model many primitives such as encryption, signature, hash, or zero-knowledge among others [Blanchet et al. 2020; Cheval et al. 2020b], but not associative-commutative behaviours, such as XOR or group exponentiation [Basin et al. 2019]. Rules may also be extended with weights to capture cryptographic primitives relying on time consumption [Barthe et al. 2022], which we omit here to alleviate the presentation of the model. We lift $\to_{\mathcal{R}}$ to a relation on arbitrary terms that is the closure of $\mathcal{R}$ under well-typed substitution and term context, that is, for all $\ell \to_{\mathcal{R}} r$, well-typed substitutions σ, and contexts $C[x]$ such that $x \in \mathcal{X} \setminus \mathcal{X}^{\mathsf{N}}$, we have $C[\ell\sigma] \to_{\mathcal{R}} C[r\sigma]$.

Also, informally, a destructor operation "fails" when, intuitively, its arguments do not match any rewrite rule. A (well-typed) term without such failure is called a *message*:

Definition 3.3 (Message)**.** A term u is a *message* if for all subterms v of u, $v\downarrow$ is a constructor term; also, if $v\downarrow = f(v_1, \ldots, v_n)$ for some arithmetic operator f, all terms v_i should be numeric terms. We write $\mathsf{msg}(u)$ to express that u is a message, the notation being extended to sequences, sets... of terms by requiring that all these terms are messages.

3.2 Protocols in an Adversarial Environment

Syntax. We model protocols as *processes* of a timed and stateful variant of the applied π-calculus. We use *events* for formalising security properties; these are modelled by a dedicated, finite set of function symbols $\mathcal{F}_e$. Although the syntax is mostly taken from [Barthe et al. 2022], we introduce a novel notion of *atomic composition* $\alpha : \beta$, that executes two instructions α and β (or more) simultaneously. In addition to improving the model's modularity by allowing for breaking complex instructions into simpler blocks, atomic composition offers fine-grained control over the timestamping of instructions which is useful when modelling timed case studies. We also introduce a *global state*, i.e., a multiset to which any subprocess may push, pull, or pattern-match terms.

Definition 3.4 (Process)**.** The grammar of (timed) *processes* is:

$P ::= 0$	*null process*	$S ::= S + S$	*choice*	$\mathsf{push}\ u : S$	*global store*
$P \mid P$	*parallel*	$\mathsf{skip} ; P$	*break*	$\mathsf{pull}\ u : S$	*successful lookup*
$!^n P$	*replication*	$\mathsf{out}(u) : S$	*output*	$\mathsf{unfound}\ u : S$	*failed lookup*
S	*sequence*	$\mathsf{in}(x) : S$	*input*	$@t : S$	*timestamp*
		$\mathsf{new}\ k : S$	*new name*	$\mathsf{when}\ e \sim 0 : S$	*arithmetic*
		$\mathsf{Ev}(\vec{u}) : S$	*event*		

where $n \in \mathbb{N} \cup \{\infty\}$, u is a term, $\vec{u} = u_1, \ldots, u_p$ is a sequence of terms, $\mathsf{Ev}/p \in \mathcal{F}_e$, $k \in \mathcal{N}_{priv}$, $x \in \mathcal{X}$, $t \in \mathcal{X}^{\mathsf{N}}$, and e is a numeric term and $\sim \in \{>, \geqslant\}$. We call a process *bounded* when all of its replications $!^n P$ are so that $n \neq \infty$.

We comment the syntax below. Processes are structured by the operators $P \mid Q$ (executing P and Q concurrently), $P + Q$ (executing the first instruction of either P or Q), and $!^n P$ (executing up to n parallel copies of P). Process instructions are composed *atomically* with the syntax $c_1 : c_2 : S$, meaning that c_1 and c_2 are executed simultaneously. An atomic sequence is eventually broken by a skip, which simply lets time elapse. For readability, we often omit the skip when its role is only to break a sequence (and we write a semicolon directly after the previous instruction). Then, $\mathsf{new}\ k$ generates a fresh name, i.e., a value unknown to the adversary. Events have no interpretation in concrete protocol implementations and are simply used here to formalise security properties. Timestamps $@t$ and the "when" operator record the current time t and perform arithmetic tests, respectively. Importantly, processes have access to a global state that is a multiset θ of constructor terms. Terms can be added to θ using $\mathsf{push}\ u$, and searched for (with pattern matching) using $\mathsf{pull}\ u$ or $\mathsf{unfound}\ u$. That is, $\mathsf{pull}\ u$ only proceeds if $u\sigma \in \theta$ for some σ (and has for effect to remove one occurrence of $u\sigma$ from θ), while $\mathsf{unfound}\ u$ asserts that no instance of u is in θ. Finally, inputs and outputs handle remote communications. Since the network is compromised, an output $\mathsf{out}(u)$ reveals u to the adversary, while an input $\mathsf{in}(x)$ fetches an arbitrary message x from them, possibly forged from previous outputs. Note however that, in an atomic composition $\mathsf{out}(u) : \mathsf{in}(x) : S$, the adversary has no access to u when computing x as they are emitted simultaneously.

Example 3.1. Despite the minimal form of the grammar, it is expressive enough to make our results relatively independent of the formalism. For example, one may encode the mutable variables

used in the motivating example of Section 2. Each such variable x is injectively mapped to a public name c_x, and the value u of x is stored in the global state under the form $\mathsf{Store}(c_x, u)$ for some dedicated symbol Store/2. Given a numeric variable x, incrementing it could therefore be done by:

$$x \mathrel{+}= 1 : P \quad \triangleq \quad \mathsf{pull}\ \mathsf{Store}(c_x, y) : \mathsf{push}\ \mathsf{Store}(c_x, y+1) : P\,.$$

We detail in Appendix B other common features such as tests (`if` / `else`), or communication channels. As a complement, Figure 6 defines the omitted process in Section 2 in this extended syntax, namely, the expected behaviour of an honest user interacting with the patched contract `FairPuzzle`.

```
let id = pk(sk) in
let d0 = (id,commit,h((sol,id))) in
let tx0 = (d0,sign(d0,rand(0),sk)) in
@t0: if t0 < timeout: out(tx0): Submit(tx0);
let d1 = (id,release,sol) in
let tx1 = (d2,sign(d1,rand(1),sk)) in
@t1: if t1 > timeout: out(tx1): Submit(tx1)
```

Fig. 6. Honest user submitting to `FairPuzzle`

In both scenarios (the broken or the fair puzzle), if we call `user` the process describing the expected submission process, and `P` the one defining the contract's behaviour, the process `user | !^n P` would thus model the scenario consisting of an honest user and the contract which may be invoked up to `n` times. The Dolev-Yao adversary then models an implicit coalition of arbitrarily-many dishonest users.

Semantics. We now define a labelled transition system describing the execution of processes in an adversarial environment. To model the knowledge obtained by the adversary by interfering with communications (i.e., spying on outputs), we use a standard notion of *frame* Φ. It is a substitution

$$\Phi = \{\mathsf{ax}_1 \mapsto u_1, \ldots, \mathsf{ax}_n \mapsto u_n\}$$

where $u_1, \ldots, u_n$ are constructor terms, and $\mathsf{ax}_i \in \mathcal{AX}$ are special variables (not considered as being part of $\mathcal{X}$) called *axioms*. They serve as handles to outputs u_i intercepted during a protocol execution by the adversary. We write $\mathit{axioms}(\xi)$ the set of axioms appearing in an object ξ. In particular, adversarial computations are modelled by:

Definition 3.5 (Recipe)**.** A *recipe* is a term of the form $\xi \in \mathcal{T}(\mathcal{N}_{pub} \cup \mathcal{F}_{pub} \cup \mathcal{AX})$. We say that a recipe ξ *deduces* a constructor term u from a frame Φ if $\mathsf{msg}(\xi\Phi)$ and $\xi\Phi{\downarrow} = u$.

Recipes cannot contain private names or functions because they are not accessible to the adversary *a priori*. The actual operational semantics (Figure 7) then operates on an extended form of process recording the multiset of subprocesses executed in parallel $\mathcal{P}$, the adversary's knowledge Φ, the global state θ, and the current time t.

Definition 3.6 (Extended process)**.** An *extended process* is a tuple $A = (\mathcal{P}, \Phi, \theta, t)$, with $\mathcal{P}$ a multiset of processes, Φ a frame, θ a multiset of constructor terms, and $t \in \mathbb{R}^+$. We write the respective components $\mathcal{P}(A)$, $\Phi(A)$, $\theta(A)$ and $t(A)$, and often interpret a process P as $(\{\!\{P\}\!\}, \varnothing, \varnothing, 0)$.

The semantics Figure 7 is a labelled transition relation $\xrightarrow{\vec{w}}$, where $\vec{w}$ is a (possibly-empty) sequence of so-called *actions*, that are either event terms $\mathsf{Ev}(\vec{u})$, $\mathsf{Ev} \in \mathcal{F}_e$, or some transition-specific labels (par, repl, in...). Figure 7 introduces two intermediary relations $\rightarrow_{\mathsf{at}}$ (*atomic semantics*) and $\rightarrow_{\mathsf{seq}}$ (*sequential semantics*). They define the behaviour of single instructions: $\rightarrow_{\mathsf{at}}$ specifically handles operations that may be composed atomically (it hence operates on single processes instead of multisets), while $\rightarrow_{\mathsf{seq}}$ executes consecutive atomic instructions until reaching a skip. The sequential semantics ensures in particular through Rule (Comp) that atomically-composed outputs do not update the adversary's knowledge Φ before the end of the rule. Note also that the global clock t is entirely managed by Rule (Lift), which defines $\rightarrow$ by letting some time pass from t to $t + \delta$.

operational semantics:

$$(\{\!\{P\}\!\} \cup \mathcal{Q}, \Phi, \theta, t) \xrightarrow{\vec{w}} (\mathcal{P} \cup \mathcal{Q}, \Phi', \theta', t+\delta) \quad \text{if } \delta > 0 \text{ and } (\{\!\{P\}\!\}, \Phi, \theta, t+\delta) \xrightarrow{\vec{w}}_{\text{seq}} (\mathcal{P}, \Phi', \theta', t+\delta) \quad \text{(Lift)}$$

sequential semantics:

$$(\{\!\{\mathsf{skip}\,; P\}\!\}, \Phi, \theta, t) \xrightarrow{\text{skip}}_{\text{seq}} (\{\!\{P\}\!\}, \Phi, \theta, t) \quad \text{(Skip)}$$

$$(\{\!\{P \mid Q\}\!\}, \Phi, \theta, t) \xrightarrow{\text{par}}_{\text{seq}} (\{\!\{P, Q\}\!\}, \Phi, \theta, t) \quad \text{(Par)}$$

$$(\{\!\{!^n P\}\!\}, \Phi, \theta, t) \xrightarrow{\text{repl}}_{\text{seq}} (\{\!\{!^{n-1} P, P\}\!\}, \Phi, \theta, t) \quad \text{if } n > 0 \quad \text{(Repl)}$$

$$(\{\!\{P\}\!\}, \Phi, \theta, t) \xrightarrow{\alpha, \vec{w}}_{\text{seq}} (\mathcal{P}, \Phi' \cup \Phi'', \theta'', t) \quad \text{if } (P, \Phi, \theta, t) \xrightarrow{\alpha}_{\text{at}} (P', \Phi', \theta', t) \text{ and } (\{\!\{P'\}\!\}, \Phi, \theta', t) \xrightarrow{\vec{w}}_{\text{seq}} (\mathcal{P}, \Phi'', \theta'', t) \quad \text{(Comp)}$$

atomic semantics (over single processes):

$$(\mathsf{out}(u) : S, \Phi, \theta, t) \xrightarrow{\text{out}}_{\text{at}} (S, \Phi \cup \{\mathsf{ax} \mapsto u{\downarrow}\}, \theta, t) \quad \text{if } \mathsf{msg}(u) \text{ and } \mathsf{ax} \in \mathcal{AX} \setminus dom(\Phi) \quad \text{(Out)}$$

$$(\mathsf{in}(x) : S, \Phi, \theta, t) \xrightarrow{\text{in}}_{\text{at}} (S\{x \mapsto \xi\Phi{\downarrow}\}, \Phi, \theta, t) \quad \text{if } \xi \text{ recipe, } \mathsf{msg}(\xi\Phi) \quad \text{(In)}$$

$$(\mathsf{new}\ k : S, \Phi, \theta, t) \xrightarrow{\text{new}}_{\text{at}} (S\{k \mapsto k'\}, \Phi, \theta, t) \quad \text{if } k' \in \mathcal{N}_{priv} \text{ fresh} \quad \text{(New)}$$

$$(\mathsf{Ev}(\vec{u}) : S, \Phi, \theta, t) \xrightarrow{\mathsf{Ev}(\vec{u}{\downarrow})}_{\text{at}} (S, \Phi, \theta, t) \quad \text{if } \mathsf{msg}(\vec{u}) \quad \text{(Event)}$$

$$(\mathsf{push}\ u : S, \Phi, \theta, t) \xrightarrow{\text{push}}_{\text{at}} (S, \Phi, \theta \cup \{\!\{u{\downarrow}\}\!\}, t) \quad \text{if } \mathsf{msg}(u) \quad \text{(Push)}$$

$$(\mathsf{pull}\ u : S, \Phi, \theta \cup \{\!\{u\sigma{\downarrow}\}\!\}, t) \xrightarrow{\text{pull}}_{\text{at}} (S\sigma, \Phi, \theta, t) \quad \text{if } \mathsf{msg}(u\sigma) \quad \text{(Pull)}$$

$$(\mathsf{unfound}\ u : S, \Phi, \theta, t) \xrightarrow{\text{unfound}}_{\text{at}} (S, \Phi, \theta, t) \quad \text{if for all } \sigma \text{ such that } \mathsf{msg}(u\sigma),\ u\sigma{\downarrow} \notin \theta \quad \text{(Unfound)}$$

$$(@t_0 : S, \Phi, \theta, t) \xrightarrow{\text{stamp}}_{\text{at}} (S\{t_0 \mapsto t\}, \Phi, \theta, t) \quad \text{(Stamp)}$$

$$(\mathsf{when}\ e \sim 0 : S, \Phi, \theta, t) \xrightarrow{\text{when}}_{\text{at}} (S, \Phi, \theta, t) \quad \text{if } e \in \mathbb{R} \text{ and } e \sim 0 \text{ holds} \quad \text{(When)}$$

$$(S_0 + S_1, \Phi, \theta, t) \to_{\text{at}} (S_i, \Phi, \theta, t) \quad \text{if } i \in \{0, 1\} \quad \text{(Choice)}$$

Fig. 7. Operational semantics of the calculus

We now comment the rules of Figure 7. In the communication rules (Out) and (In), outputs increase the adversary's knowledge (i.e., add the axiom ax to $dom(\Phi)$), and inputs are computed by the adversary using previous outputs (i.e., through a recipe ξ). Note also that only valid messages in normal form, namely constructor terms, are allowed to circulate on the network. Rule (Event) triggers an event used to formalise security properties, and may be atomically composed to other instructions to add custom labels to them as well. The three rules (Push), (Pull) and (Unfound) then formalise the behaviour of global state manipulation by adding, removing, or checking the absence of a term in θ, respectively. Rules (New), (Stamp) and (When) have a straightforward interpretation, while Rule (Par) and (Repl) spawn parallel threads, and (Choice) atomically branches to either of two processes. The semantics then induces the following notion of execution:

Definition 3.7 (Trace, Trace state)**.** A *trace* T of an extended process A_0 is a finite sequence

$$T : (\mathcal{P}_0, \Phi_0, \theta_0, t_0) \xrightarrow{\vec{w}_1} \cdots \xrightarrow{\vec{w}_n} (\mathcal{P}_n, \Phi_n, \theta_n, t_n)$$

of $\to$-transitions as defined in Figure 7. If $t \geqslant t_0$, we also define the *state of* T *at time* t, written $T@t$, as the following pair, with the convention $t_{n+1} = +\infty$:

$$T@t = ((\mathcal{P}_i, \Phi_i, \theta_i, t), E) \qquad \text{with } i = \min\{j \leqslant n+1 \mid t_j \geqslant t\}$$

where $E = \{\vec{w}_i\}$ if $t = t_i$, $i \in [\![1, n]\!]$, and $E = \varnothing$ otherwise. The state $T@t = (A, E)$ of a trace thus indicates the set E of events being triggered at the current time t, if any, as well as the current state of the process execution A (at time t but right after the execution of the events of E).

Example 3.2. Using the notations of Example 3.1, consider the process $A_0 = \mathsf{user} \mid !^2\,\mathsf{P}$ modelling an honest user submitting to the broken `Puzzle` contract. The trace modelling the attack against fair reward from Section 2 would typically start as a sequence of transitions of the form

$$A_0 \xrightarrow{\mathsf{par}} \xrightarrow{\mathsf{out},\mathsf{Submit}(tx)} A_1 \xrightarrow{\mathsf{repl}} \xrightarrow{\mathsf{in},\mathsf{Publish}(tx')} A_2 \xrightarrow{\mathsf{repl}} \xrightarrow{\mathsf{in},\mathsf{Publish}(tx)} A_3$$

for some extended processes A_1, A_2, A_3. The first segment from A_0 to A_1 submits the honest transition tx (say, under the axiom ax). The second segment from A_1 to A_2 is the adversary's interference. After unfolding one copy of P (Rule (Repl)), the dishonest transaction tx' gets published, being fetched through the input recipe $\xi = (\zeta, \mathsf{sign}(\zeta, r, \mathsf{sk}'))$, where $r \in \mathcal{N}_{pub}$ is fresh, $\mathsf{sk}' \in \mathcal{N}_{pub}$ models the signature key of the adversary, and $\zeta = (\mathsf{pk}(\mathsf{sk}'), \mathsf{solve}, \mathsf{ax.args})$. The recipe ξ thus models the adversary extracting the solution from tx and then forging their own transaction from it. The final segment from A_2 to A_3 then publishes the honest transaction tx to comply with the blockchain axiomatisation (recall `hyp_submit` in Section 2, Figure 1). This time, the adversary forwards the honest transaction tx, that is, uses the recipe $\xi' = \mathsf{ax}$ for the publication's input.

4 VERIFICATION OF HYPERPROPERTIES

4.1 Hyperproperties

Guarded Hyperformulae. We present the logic Hypertidy CTL* of [Barthe et al. 2022] for specifying hyperproperties, adapted to our context with atomic composition, and restricted to a fragment with some syntactic restrictions (*guarded hyperformulae*). First, we do not provide a generic operator for what was called "second-order quantifiers" $\forall X$ in [Barthe et al. 2022], which quantify over all possible recipes X that the adversary may compute. Instead, we restrict their use to the atomic formula $\mathsf{K}_\pi(u)$, stating that the adversary can deduce ("Knows") u in the frame of the trace π. This feature is not crucial to the examples of this paper, but is useful to express, e.g., notions of data secrecy. However, this limitation excludes more complex statements about the view of the adversary, like indistinguishability properties such as *static equivalence* $\pi_0 \sim \pi_1$ (reading as "there exist no public tests, i.e., recipes ξ, that deduce a valid message in π_i but fail in π_{1-i}"). We leave such properties aside, as most use cases for them are already studied by dedicated tools [Basin et al. 2019; Blanchet et al. 2020; Cheval et al. 2020b], and as guarded hyperformulae may still express many relevant relational hyperproperties through arbitrary deducibility constraints or term equalities across multiple traces. Naturally, non-adversarial notions of path indistinguishability (e.g., one that compares parts of the global states as in our motivating example) are expressible in our framework.

As a second (natural) restriction, we will impose that *first-order quantifiers* $\forall x$, quantifying over messages, are *guarded*. This is the uplifting of standard assumptions in reachability properties (see, e.g., [Basin et al. 2019]) to the Tidy logics. It intuitively requires that such quantifiers should only be used to make reference to terms used by the process, and not to express arbitrary first-order formulae on terms. Rephrasing, the possible instantiations of x should always be determined from context when (dis)proving that a subformula $\forall x.\varphi$ holds. A similar notion has been introduced in Hypertidy CTL*, but is purposely very strict so that the complexity lower bounds of [Barthe et al. 2022] could be as general as possible. On the contrary, we designed our notion of guard with expressivity in mind, making it naturally more general than the prior version of [Barthe et al. 2022].

Definition 4.1 (guarded hyperformula)**.** The grammar of *guarded hyperformulae* is:

$\varphi ::=$	$\bot$	*false*	$\mathrm{GS}_\pi(u)$	*global state membership*
	$u = v$	*equality modulo theory*	$\forall x_1, \ldots, x_p.\varphi \Rightarrow \varphi$	*guarded quantifier*
	α_π	*action*	$\forall \pi.\varphi$	*path quantifier*
	$\mathsf{K}_\pi(u)$	*adversary deduction*	$\varphi \,\mathsf{U}_I\, \varphi$	*(time-constrained) until*

where u, v are terms, α is an action (as in the operational semantics), $x_1, \ldots, x_p \in \mathcal{X}$ ($p \geqslant 0$), π belongs to a set $\mathcal{X}^p$ of so-called *path variables*, and $I = (a, b)$ is an open interval with a, b numeric terms (or possibly $b = +\infty$). We also require that in $\forall \vec{x}.\varphi_0 \Rightarrow \varphi_1$, the formula φ_0 is a *guard* for $\vec{x}$. Namely, we require that for all $y \in \{\vec{x}\}$, y should appear with *positive polarity* (i.e., 1) in φ_0, polarity being defined as follows assuming a fresh alpha renaming of all quantified variables:

- y appears with polarity 1 in $\bot$, and also in $\mathsf{GS}_\pi(u)$ (resp. $\mathsf{Ev}_\pi(\vec{v})$) provided u (resp. provided there exists $u \in \{\vec{v}\}$ which) is a constructor term and $y \in \mathit{vars}(u)$;
- if y appears with polarity p_i in φ_i, $i \in \{0, 1\}$, it appears with polarity $\min(-p_0, p_1)$ in $\varphi_0 \Rightarrow \varphi_1$;
- if y appears with polarity -1 in φ, so does it in $\forall \omega.\varphi$, with $\omega \in \mathcal{X} \cup \mathcal{X}^p$;
- if y appears with polarity 1 in ψ, so does it in $\varphi \,\mathsf{U}_I\, \psi$.

Let us now give an intuition for each constructor of the logic. First of all, the *atomic formulae* are $\bot$ (always false), $u = v$ ("u and v are messages of same normal form"), α_π ("at the current time, π performs the action α"), $\mathsf{K}_\pi(u)$ ("the adversary can deduce u from the current frame of π"), and $\mathsf{GS}_\pi(u)$ ("u belongs to the current global state of π"). Guarded quantifiers $\forall \vec{x}.\varphi_0 \Rightarrow \varphi_1$ instantiate the variables $\vec{x}$ as messages, in a way restricted by φ_0. Without quantification (i.e., $\{\vec{x}\} = \emptyset$), this results in a plain implication $\varphi_0 \Rightarrow \varphi_1$. In combination with the atomic formula $\bot$, this permits to derive negation $\neg\varphi \triangleq \varphi \Rightarrow \bot$, "true" $\top \triangleq \neg\bot$, disjunction $\varphi \vee \psi \triangleq (\neg\varphi) \Rightarrow \psi$, and in turn conjunction $\varphi \wedge \psi$, and existential quantifiers $\exists \pi.\varphi$ and $\exists \vec{x}.\varphi_0 \wedge \varphi_1$. We also sometimes consider the *silent action* τ_π (`silent` in Section 2), which is the negation of all other actions α_π. The quantifiers over path variables $\forall \pi.\varphi$ specifically quantify over all traces π of the process, with nested quantifiers branching from the last path quantifier in scope. Finally, the "until" $\varphi \,\mathsf{U}_I\, \psi$ means that φ should hold during a time window $(t, t + \delta)$, where $\delta \in I$ and t is the current time, while ψ holds at time $t + \delta$. We often use the typical notation $\varphi \,\mathsf{U}\, \psi \triangleq \varphi \,\mathsf{U}_{(0,+\infty)}\, \psi$ to refer to a non-time-constrained version of the operator. Other classical temporal operators include $\mathsf{F}_I\, \varphi \triangleq \top \,\mathsf{U}_I\, \varphi$ ("finally") and $\mathsf{G}_I\, \varphi \triangleq \neg(\mathsf{F}_I\, \neg\varphi)$ ("globally"). The non-time-constrained version of these operators F and G, used in Section 2, are also to be understood with $I = (0, +\infty)$.

The satisfiability relation for guarded hyperformulae is formalised in Figure 8, through judgements of the form $\Pi, t \models \varphi$ with Π a substitution from path variables to traces and $t \in \mathbb{R}^+$. We assume a dedicated path variable $\varepsilon \in \mathit{dom}(\Pi)$ used to track the last path quantifier.

Definition 4.2 (Satisfiability)**.** A process P *satisfies* φ, written $P \models \varphi$, when $\{\varepsilon \mapsto \varepsilon_P\}, 0 \models \varphi$, with ε_P the empty trace starting from P.

$$
\begin{array}{lll}
\Pi, t \not\models \bot & & \\
\Pi, t \models u = v & \mathit{iff} & \mathsf{msg}(u, v) \text{ and } u{\downarrow} = v{\downarrow} \\
\Pi, t \models \mathsf{Ev}_\pi(\vec{u}) & \mathit{iff} & \mathsf{msg}(\vec{u}), \text{ and writing } \Pi(\pi)@t = (A, E) \text{ for some } A, E, \text{ we have } \mathsf{Ev}(\vec{u}){\downarrow} \in E \\
\Pi, t \models \mathsf{K}_\pi(u) & \mathit{iff} & \mathsf{msg}(u), \text{ and writing } \Pi(\pi)@t = (A, E) \text{ for some } A, E, \text{ we have } u{\downarrow} \in \theta(A) \\
\Pi, t \models \mathsf{GS}_\pi(u) & \mathit{iff} & \mathsf{msg}(u), \text{ and writing } \Pi(\pi)@t = (A, E) \text{ for some } A, E, u{\downarrow} \text{ deducible from } \Phi(A) \\
\Pi, t \models \forall x.\, \varphi & \mathit{iff} & \text{for all ground constructor terms } u \text{ s.t. } \sigma = \{x \mapsto u\} \text{ is well-typed, } \Pi \models \varphi\sigma \\
\Pi, t \models \varphi \Rightarrow \psi & \mathit{iff} & \Pi, t \not\models \varphi \text{ or } \Pi, t \models \psi \\
\Pi, t \models \forall \pi.\, \varphi & \mathit{iff} & \text{writing } \Pi(\varepsilon)@t = (A, E), \text{ for all traces } T \text{ of } A,\ \Pi[\pi \mapsto T, \varepsilon \mapsto T] \models \varphi \\
\Pi, t \models \varphi \,\mathsf{U}_I\, \psi & \mathit{iff} & I \subseteq \mathbb{R}^+ \text{ and } \exists \delta \in I.\ \Pi, t + \delta \models \psi \text{ and } \forall \delta' \in (0, \delta).\ \Pi, t + \delta' \models \varphi
\end{array}
$$

Fig. 8. Satisfiability relation for formulae

4.2 Fragments and Restrictions

In this paper, we provide a large range of results, referring in particular to several classical fragments of Hypertidy CTL* whose definitions are recalled below.

Definition 4.3 (Fragments)**.** A guarded hyperformula φ is said to be in:

- Hypertidy LTL if it is of the form $\varphi = Q_1\pi_1 \dots Q_n\pi_n.\psi$, for $Q_i \in \{\forall, \exists\}$ and ψ a formula without path quantifiers;
- tidy CTL* if for all subformulae of φ of the form $\forall\pi.\psi$, all path and regular variables appearing in ψ (except π) are bound by a quantifier in ψ;
- tidy LTL if it is in Hypertidy LTL but has only one path quantifier.

While many properties of our examples will fall into the Hypertidy LTL fragment, the running example (Section 2, Figure 5) typically uses nested quantifiers to express succinctly that the two equivalent trace π_1 and π_2 did the same setup phase. Other typical properties that Hypertidy LTL does not capture are liveness properties such as fairness in optimistic fair exchange [Barthe et al. 2022], which fall into tidy CTL* (non-relational formulae). Finally, the tidy LTL fragment is restricted to express trace properties such as (weak) secrecy or authentication. The simple blockchain axiomatisation of Section 2 falls into this last class. In protocol analysis, one is then usually interested in the *verification* problem:

Decision Problem: Verif
► Input: A term algebra (consisting of $\mathcal{F}, \mathcal{F}_e, \mathcal{R}$), a process P built from it, a guarded hyperformula φ
► Output: Does $P \models \varphi$ hold?

The analogue of Verif in HyperCTL* is *model checking*, simpler than satisfiability in that it is decidable for arbitrary hyperproperties [Clarkson et al. 2014]. The problem is however more complex in our context, i.e., undecidable even in tidy LTL [Barthe et al. 2022]. To achieve decidability, our results therefore rely on previously-mentioned restrictions (bounded processes and destructor subterm convergent rewriting). We motivate them below through several undecidability results. First, the problem is undecidable for unbounded processes, even without rewriting.

Proposition 4.1 (Verification of arbitrary processes)**.** *Verif is undecidable even when* $\mathcal{F}_c = \mathcal{F}_{pub} = \{\mathsf{h}/1\}$, $\mathcal{F}_d$ *and* $\mathcal{R}$ *are empty, and* φ *is a tidy LTL formula.*

Although general undecidability is common in protocol analysis, this result emphasises the minimality of the fragment needed to reach it. The proof, in Appendix C, relies on an encoding of two-counter machines, and is done in a restricted setting without real-time, and with global states limited to simulate internal communications. One therefore has to either rely on partial decision procedures or impose restrictions on processes to obtain decidability. We chose to focus on the decidability of bounded processes which, we recall, is still of practical interest as attacks on real protocols rarely require many sessions. However, this does not make the problem trivially decidable due to the adversary, if the term algebra remains unrestricted. Typically [Barthe et al. 2022]:

Proposition 4.2 (Verification with arbitrary rewriting)**.** *Verif is undecidable for bounded processes and a convergent term algebra, even for tidy LTL formulae.*

4.3 Technical Overview of the Results

In this section, we state our main decidability and complexity results (summarised at the end of the section, Figure 9) and reference the corresponding sections. Basics of complexity theory can be found in Appendix A but, as a minimal reminder, we give here relations between complexity classes of interest, including **EXPH**(poly) (polynomially-bounded exponential hierarchy):

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}$$
$$\mathbf{EXP} \subseteq \mathbf{NEXP} \subseteq \mathbf{EXPH}(\text{poly}) \subseteq \mathbf{EXPSPACE}$$

When it comes to complexity, one important aspect of the problem is the numeric constraints that may appear in "when" instructions, or that naturally arise as (discrete-time) scheduling constraints when proving hyperproperties. Our results are largely independent of the details of the solving of numeric constraints; they thus always implicitly assume:

(1) a notion of *atomic numeric constraint* used in "when" conditions, and that allows at least constraints $t < t'$ for $t, t' \in \mathcal{X}^{\mathsf{N}}$. All of our complexity lower bounds are valid regardless, that is, without "when" instructions;
(2) an *oracle* checking whether a conjunction of atomic time constraints has a solution (that is, an instantiation of the numeric variables satisfying the constraints).

For oracle-free results to hold, the underlying problem should be decidable—and in **P**, when it comes to complexity. Note that this polynomial bound is at least trivial for the minimal required core of atomic constraints, i.e., in contexts without real-time.

Decidability in Restricted Fragments. As a complement to our main result (stated at the end of the section), we first mention a couple of side complexity results for fragments of the logic (proved in Appendix F). Historically, reachability properties (tidy LTL) benefited from more attention, resulting in a strong theoretical understanding and automated support, compared to wider fragments [Cheval et al. 2020a]. For example, for subterm convergent term algebras and bounded processes, many such properties such as weak secrecy are known to be co**NP** complete [Baudet 2007; Rusinowitch and Turuani 2003]. However surprisingly, verifying generic tidy LTL formulae is also known to be **PSPACE** hard under the same assumptions [Barthe et al. 2022]. We actually prove that:

Theorem 4.3 (Decidability in tidy LTL)**.** *Verif is **PSPACE** complete for destructor subterm convergent term algebras, bounded processes, and tidy LTL guarded hyperformulae.*

This gap in complexity compared to practical co**NP** properties is due to the permissive (guarded) quantifiers over terms $\forall x$. This easily allows us to encode the satisfaction of *quantified boolean formulae*, the prototypical **PSPACE** complete problem—while the above-mentioned co**NP** complete security properties only exploit such quantifiers in a rather limited way. We introduce here a more restricted notion of safety property, closer to practical examples, where the Verif problem will be co**NP** complete. For that, we require that only the same type of quantifiers ($\forall, \exists$) is used. Formally:

Definition 4.4 (Mono-polarity fragments)**.** We inductively define that a guarded hyperformula has *polarity* $\forall$, $\exists$, or 0 as follows. If $p \in \{0, \forall, \exists\}$, we write $\neg p$ the opposite polarity, that is, $\neg\forall = \exists$, $\neg\exists = \forall$ and $\neg 0 = 0$. Then:

- atomic formulae and their negations have polarity 0;
- if φ_i has polarity p_i, $\forall \vec{x}.\varphi_0 \Rightarrow \varphi_1$ has both polarity $\neg p_0$ and p_1, and additionally $\forall$ if $n \geqslant 1$;
- if φ has polarity p, $\forall \pi.\varphi$ has polarity $\forall$ and p;
- if φ_i has polarity p_i, $\varphi_0 \,\mathsf{U}_I\, \varphi_1$ has polarity p_0 and p_1.

We call tidy Q-LTL , $Q \in \{\forall, \exists\}$, the class of tidy LTL guarded hyperformulae without polarity $\neg Q$.

Many reachability properties do not involve both $\forall$ and $\exists$. Weak secrecy ($\forall \pi.\, \mathsf{G}\, \neg \mathsf{K}_\pi(u)$), event reachability ($\exists \pi.\, \mathsf{F}\, \mathsf{A}_\pi$), or correspondence properties ($\forall \pi.\, \neg \mathsf{A}_\pi \,\mathsf{U}^?\, \mathsf{B}_\pi$) are typical examples.

Theorem 4.4 (Complexity of mono-polarity tidy LTL)**.** *Verif is co**NP** complete (resp. **NP** complete) for destructor subterm convergent term algebras, bounded processes, in tidy $\forall$-LTL (resp. tidy $\exists$-LTL).*

We also provide tight complexity analyses for the *pure* π-calculus, that is, with an empty term algebra ($\mathcal{F} = \varnothing$ and $\mathcal{R} = \varnothing$), see Figure 9. Although anecdotal in terms of applications, this fragment serves as a baseline for comparison to emphasise which assumptions increase the verification cost.

Decidability in Hypertidy CTL.* Finally, our main result is then that Verif is decidable under the assumptions of this paper.

Theorem 4.5 (Decidability in Hypertidy CTL*)**.** *The Verif problem is **EXPH**(poly) complete for destructor subterm convergent term algebras, bounded processes, and guarded hyperformulae of either tidy CTL*, Hypertidy LTL or Hypertidy CTL*.*

This states three incomparable complexity results for Verif. However, as **EXPH**(poly) hardness is already established for the three of them [Barthe et al. 2022], it suffices to prove that Verif is in **EXPH**(poly) only for the whole logic Hypertidy CTL*. Our procedure is based on *constraint solving*, recalling [Cheval et al. 2013, 2018; Liu and Lin 2012]. This intuitively replaces the sources of unboundedness of traces (recipes and timestamps), by variables and *constraints* on their instantiations. This is done through an alternative, finitely-branching *symbolic semantics* defined in Section 5, used as a basis for a procedure in Section 6. Given a guarded hyperformula φ, it mainly computes two sets of constraints $\Omega_{\varphi}, \Omega_{\neg\varphi}$, where solutions of Ω_{ψ} can intuitively be used to reconstruct a proof that ψ holds. Typically, if $\varphi = \varphi_0 \Rightarrow \varphi_1$, the procedure starts by computing $\Omega_{\varphi_0}, \Omega_{\neg\varphi_0}$. Then, as the constraints of $\Omega_{\neg\varphi_0}$ already induce a proof of φ, the constraints of Ω_{φ_0} are refined w.r.t. to the formula φ_1 through a suitable recursive call, to compute $\Omega_{\varphi_0 \wedge \varphi_1}, \Omega_{\varphi_0 \wedge \neg\varphi_1}$. The final decision criterion is then that φ holds *iff* $\Omega_{\neg\varphi} = \varnothing$. One technical challenge specific to hyperproperties is the more complex notion of constraints compared to related works [Cheval et al. 2013, 2018], as they are to be interpreted across several different traces. Additionally, two constraints characterising two different traces are not independent: φ expresses relations between them. Therefore, they cannot be solved separately. Our procedure handles nested path quantifiers using a stack structure, to propagate the solving's results from deeper levels to higher ones. Finally, using a *small-solution property* (i.e., that all constraints of $\Omega_{\varphi} \cup \Omega_{\neg\varphi}$ have a solution of at most exponential size), we show that we can derive a naive **EXPH**(poly) procedure.

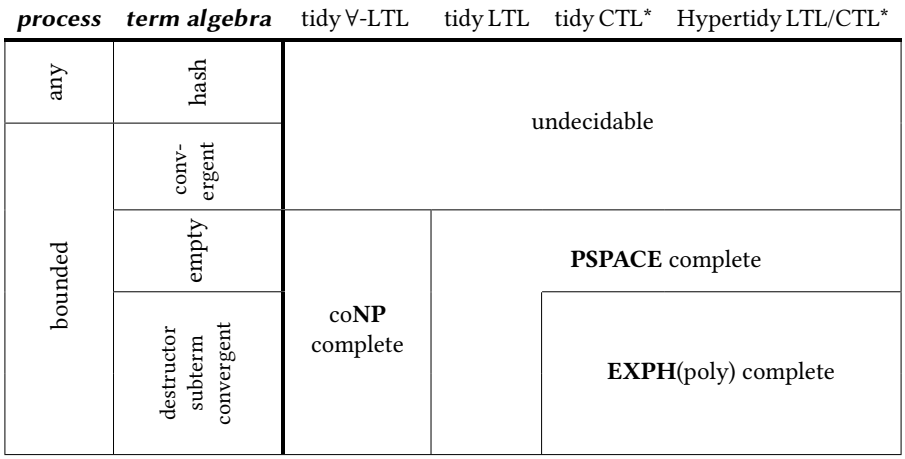

Fig. 9. Summary: Complexity of Verifying Guarded Hyperformulae (with oracle to a numeric solver)

5 SYMBOLIC ABSTRACTION

We give a symbolic semantics for bounded processes, replacing the operational semantics with a finitely-branching relation annotated with *constraints*, mostly inspired by classical analogue notions [Cheval et al. 2018]. In the following, we always assume that processes only contain bounded replications.

5.1 (Refined) Unification Theory

We first introduce some important notions from unification theory that are necessary to formalise the symbolic semantics, and that are central to our decision procedure in general.

Refined Term Algebra. First, we partition the set of variables $\mathcal{X}$ in two: *first-order variables* $x, y, z, \ldots$ that we have been using so far in processes and formulae, and *second-order variables* $X, Y, Z, \ldots$ that will be used to represent adversarial computations (i.e., recipes). Formally, we write:

$$\mathcal{X} = \mathcal{X}^1 \uplus \mathcal{X}^2 \qquad \text{with } \mathcal{X}^{\mathsf{N}} \subseteq \mathcal{X}^1$$

where $\mathcal{X}^1 \setminus \mathcal{X}^{\mathsf{N}}$, $\mathcal{X}^{\mathsf{N}}$ and $\mathcal{X}^2$ are infinite. We recall that $\mathcal{X}^p$ is the set of *path variables*, and write $vars^1(u)$, $vars^2(u)$, $vars^n(u)$ the set of first-order, second-order, and numeric variables, respectively, appearing in an object u. To reflect the partition $\mathcal{X} = \mathcal{X}^1 \uplus \mathcal{X}^2$, we refine terms:

Definition 5.1 (First- and second-order term)**.** A *first-order term* is a term $u \in \mathcal{T}(\mathcal{F} \cup \mathcal{N} \cup \mathcal{X}^1)$, and a *second-order term* is a term $\xi \in \mathcal{T}(\mathcal{F}_{pub} \cup \mathcal{N}_{pub} \cup \mathcal{A}\mathcal{X} \cup \mathcal{X}^2)$.

First-order terms are therefore terms representing actual protocol communications, whereas second-order terms represent adversarial computations. We recall in particular from Section 3 that a ground term is a term u such that $vars(u) = \emptyset$: a ground second-order term is therefore, by definition, a recipe ξ (as axioms are not treated as variables of $\mathcal{X}$). Additionally, we use a form of bookkeeping to record which amount of knowledge was available to the adversary at the time a given adversarial computation $X \in \mathcal{X}^2$ was introduced. We consider the partition:

$$\mathcal{X}^2 = \biguplus_{M \subseteq \mathcal{A}\mathcal{X},\ M \text{ finite}} \mathcal{X}^2_M$$

Intuitively, variables of $\mathcal{X}^2_M$ represent recipes that only use axioms $\mathsf{ax} \in M$; in this case we say that X has *multiplicity* M and write $X{:}\,M$ instead of X to denote this fact. Concretely, when the symbolic semantics will introduce a new second-order variable to represent a recipe, its multiplicity will be the domain of the current frame. Besides, we adapt the definition of well-typing accordingly:

Definition 5.2 (Well-typed substitution)**.** A substitution σ is *well-typed* if for all $\omega \in dom(\sigma)$:

(1) if $\omega \in \mathcal{X}^1$ then $\omega\sigma$ is a first-order term;
(2) if $\omega \in \mathcal{X}^{\mathsf{N}}$ then $\omega\sigma$ is a numeric term;
(3) if $\omega{:}\,M \in \mathcal{X}^2$ then $\omega\sigma$ is a second-order term, $axioms(\omega\sigma) \subseteq M$ and $vars^2(\omega\sigma) \subseteq \bigcup_{N \subseteq M} \mathcal{X}^2_N$.

Constraints and unifiers. Given a set of equations between terms $E = (u_i =^? v_i)_{i \in I}$, a standard problem is *unification*, i.e., the existence of (well-typed) substitutions σ such that $u_i\sigma = v_i\sigma$ for all $i \in I$. When unification is possible, a *most general* unifier is known to exist. The problem has also been studied modulo the rewriting system [Comon-Lundh and Delaune 2005]. However, numeric terms, due to arithmetic operators, are poorly suited for unification in general. In particular, rather than a substitution, we see in this paper unifiers σ as conjunctions of *equation constraints* $u =^? v$, with u, v constructor terms. They are used to either characterise substitutions σ (through constraints $x =^? x\sigma$ for each $x \in dom(\sigma)$) or numeric equations (through constraints $e =^? 0$, e numeric term). We introduce below a more general notion of constraint, which will be at the core of our symbolic semantics, and of the different components of our decision procedure as a whole.

Definition 5.3 (Constraint)**.** A *constraint* C is of either of the forms:

$$\begin{array}{rlll} C ::= & u =^? v & \forall \vec{x}.\, \bigvee_i u_i \neq^? v_i & \textit{first-order (dis)equation} \\ & \xi =^? \zeta & \forall \vec{X}.\, \bigvee_i \xi_i \neq^? \zeta_i & \textit{second-order (dis)equation} \\ & e <^? 0 & e \leqslant^? 0 & \textit{numeric inequality} \\ & \xi \vdash^? u & \forall X.X \nvdash^? u & \textit{(non-)deduction constraint} \end{array}$$

with u, v, u_i, v_i first-order constructor terms, $\{\vec{x}\} \subseteq \mathcal{X}^1$, $\{\vec{X}, X\} \subseteq \mathcal{X}^2$, ξ, ξ_i, ζ_i second-order terms with ξ not having a constructor symbol at its root, and e is a numeric term. Note that the case of numeric constraints $e =^? 0$ is covered as a subcase of first-order equations. We usually write $\top$ and $\bot$ trivially (un)satisfiable constraints.

Intuitively, equations represent equalities between terms. We insist on the fact that they are syntactic: to express that $u{\downarrow} = v{\downarrow}$, the adequate constraint is not $u =^? v$ but $\bigwedge_{x \in dom(\sigma)} x =^? x\sigma$ for some σ such that $u\sigma{\downarrow} = v\sigma{\downarrow}$. More generally, we often see unifiers, or even well-typed substitutions in general, as conjunctions of constraints, with the interpretation:

$$\sigma \triangleq \bigwedge_{x \in dom(\sigma)} x =^? x\sigma \qquad \neg\sigma \triangleq \forall \vec{z}.\, \bigvee_{x \in dom(\sigma)} x \neq^? x\sigma \quad (\vec{z} = img(\sigma) \setminus dom(\sigma))\,.$$

Regarding numeric inequalities, they are similar to their non-symbolic analogue used in protocols in when conditions, and constraints $\xi \vdash^? u$ express that u is deducible by the adversary using the computation ξ. As stated earlier, we can then use constraints to define unification (possibly modulo rewriting) in our numeric context:

Definition 5.4 (Unification modulo rewriting)**.** A *unifier modulo rewriting* of a set of equations $E = (u_i =^? v_i)_{i \in I}$ is a conjunction of constraints $\gamma = \sigma \wedge N$, with σ a well-typed substitution (interpreted as a conjunction of constraints as above), N a conjunction of numeric equations, s.t.:

(1) $dom(\sigma) \subseteq vars(E)$ and $vars^n(N) \subseteq vars^n(E)$;
(2) there exists a well-typed substitution σ_N such that $dom(\sigma_N) = vars(N)$ and $N\sigma_N$ holds when interpreting $=^?$ as the equality on $\mathbb{R}$;
(3) for all σ_N verifying Item (2), for all $i \in I$, we have $\mathsf{msg}(u_i\sigma\sigma_N, v_i\sigma\sigma_N)$ and $u_i\sigma\sigma_N{\downarrow} = v_i\sigma\sigma_N{\downarrow}$.

We say that a set S of such unifiers is *complete* if for all $\sigma' \wedge N'$ unifiers modulo rewriting of E, there exist $\sigma \wedge N \in S$ and a well-typed substitution τ such that $\sigma' = \sigma\tau$ and such that for all well-typed σ_N of domain $vars(N, N')$, $N'\sigma_N \Rightarrow N\sigma_N$. We write $mgu_{\mathcal{R}}(E)$ such a complete set of unifiers, with the convention $mgu_{\mathcal{R}}(E) = \bot$ when none exists.

Definition 5.5 (Syntactic unification)**.** A *unifier* γ of E is a unifier modulo rewriting of E, for the alternative term algebra interpreting all symbols as constructors, and with an empty rewriting system (i.e., all terms are messages in normal form). Also, γ is a *most general unifier* of E, written $mgu(E)$, if $\{\gamma\}$ is a complete set of unifiers of E modulo this same theory.

Computing mgu can be done straightforwardly using standard algorithms, adding numeric constraints instead of regular constraints when unifying a numeric term with another term. The set $mgu_{\mathcal{R}}(E)$ is also known to be computable for our class of rewriting systems using narrowing [Comon-Lundh and Delaune 2005], up to minor modifications due to the numeric requirements. Note also that, regardless of numeric aspects, unifying a term u with itself is not always immediate due to the requirement that u is a valid message. Rephrasing, $mgu_{\mathcal{R}}(u =^? u) \neq \{\top\}$ in general. In fact, $\sigma \in mgu_{\mathcal{R}}(u =^? u)$ means $\mathsf{msg}(u\sigma)$. Finally, when it comes to (syntactically) unifying second-order terms, subtle considerations also arise due to the multiplicity of variables. Typically, when unifying, e.g., two variables $X{:}\, M, Y{:}\, N$, one should map them to a common fresh variable Z whose multiplicity is the most general one not breaking well-typing (i.e., $M \cap N$). We however

omit the details as they are vastly analogue to existing techniques despite the slight differences in contexts. For the unusual case of second-order unification, we refer to the algorithm and proofs of [Rakotonirina 2021, Chapter 2, Section 1.2].

5.2 Symbolic Semantics

We now introduce the symbolic semantics we use to give a finite characterisation of traces, using constraints to reflect the application conditions of the operational semantics $\rightarrow$. Concretely:

Definition 5.6 (Constraint system)**.** A *constraint system* is a constraint $\mathsf{D} \wedge \mathsf{K} \wedge \mathsf{E}^1 \wedge \mathsf{E}^2$ where

(1) D is a set of (non-)deduction constraints of the form $X \vdash^? u$ or $\forall X.X \nvdash^? u$ for $X \in \mathcal{X}^2$;
(2) K is a set of constraints $\xi \vdash^? u$, $\xi \notin \mathcal{X}^2$, s.t. the following is a well-defined substitution:

$$\Phi(C) = \{\mathsf{ax} \mapsto u \mid (\mathsf{ax} \vdash^? u) \in \mathsf{K}, \mathsf{ax} \in \mathcal{AX}\}$$

(3) E^1 is a set of first-order (dis)equations and numeric constraints, and E^2 of second-order (dis)equations.

Given a constraint system C, the above decomposition is unique and we write $\mathsf{D}(C)$, $\mathsf{K}(C)$, $\Phi(C)$ and $\mathsf{E}^i(C)$ the underlying components. We also use the notations:

$$mgu(\mathsf{E}^i) = mgu((w =^? w') \in \mathsf{E}^i)\,.$$

A constraint system is therefore a collection of constraints organised accordingly to their roles. Constraints of D indicate the deductions required from the adversary to reach a certain state of the proof. Typically, symbolically executing an input $\mathsf{in}(x)$ puts a constraint $X \vdash^? x$ in D. On the contrary, K is a *knowledge base* available to the attacker to resolve the constraints of D, thus generalising the notion of a frame. In our procedure, K will undergo a *saturation* generating (redundant) entries $\xi \vdash^? u$ facilitating deducibility checks. Finally, E^i adds syntactic constraints to the lot. The set E^2 in particular is not used in the symbolic semantics, but will be at the core of our constraint-solving algorithm to store solutions under the form of $mgu(\mathsf{E}^2)$. Altogether:

Definition 5.7 (Symbolic processes)**.** A *symbolic process* is a tuple $A = (\mathcal{P}, C, \theta, t)$ with $\mathcal{P}$ a multiset of processes, C a constraint system, θ a global state, and $t \in \mathcal{X}^{\mathsf{N}}$. The different components are referred to as $\mathcal{P}(A)$, $C(A)$, $\theta(A)$ and $t(A)$, respectively.

Formally, the symbolic semantics is a relation $\xrightarrow{\vec{w}}_{\mathsf{s}}$ over symbolic processes, with $\vec{w}$ a sequence of actions. Similarly to the operational semantics, it relies on an atomic semantics $\rightarrow_{\text{s-at}}$ and sequential semantics $\rightarrow_{\text{s-seq}}$. For space reasons however, their definitions can be found in Figure 14, Appendix D, and we only review a couple of rules here. Most often, symbolic processes contain free variables, as executing inputs $\mathsf{in}(x)$ symbolically does *not* instantiate x using a concrete recipe ξ. This is formalised by Rule (s-In), which freshly renames the input variable x (to avoid naming collisions), and adds a deduction constraint for it:

$$(\mathsf{in}(x) : S, C, \theta, t) \xrightarrow{\mathsf{in}}_{\text{s-at}} (S\{x \mapsto y\}, C \wedge Y \vdash^? y, \theta, t) \qquad \text{(s-In)}$$

if $y \in \mathcal{X}^1$ and Y: $dom(\Phi(C)) \in \mathcal{X}^2$ are fresh. Most other constraints are of the form $mgu_{\mathcal{R}}(u =^? u)$, thus expressing that u is a message, and are carried out to subsequent transitions by the unifier $\mu = mgu(\mathsf{E}^1(C))$. The unifier $mgu(\mathsf{E}^2(C))$, however, is not considered as no constraints are added to $\mathsf{E}^2(C)$ by the symbolic semantics. Typically, the symbolic rule for outputs is the following:

$$(\mathsf{out}(u) : S, C, \theta, t) \xrightarrow{\mathsf{out}}_{\text{s-at}} (S, C \wedge \sigma \wedge N \wedge \mathsf{ax} \vdash^? u\sigma\downarrow, \theta, t) \qquad \text{(s-Out)}$$

if $\sigma \wedge N \in mgu_{\mathcal{R}}(u\mu =^? u\mu)$ and $\mathsf{ax} \in \mathcal{AX}$ is fresh, and $\mu = mgu(\mathsf{E}^1(C))$ as above.

5.3 Soundness and Completeness

We now formalise the link between the symbolic semantics and the operational one. This relies on the following notion of solution, formalising the semantics of constraint systems:

Definition 5.8 (Solution)**.** A *solution* of a constraint system C is a pair of well-typed substitutions (Σ, σ) such that:

(1) $dom(\Sigma) = vars^2(C)$, $dom(\sigma) = vars^1(C)$, $img(\Sigma)$ only contains recipes, and $img(\sigma)$ ground constructor terms;
(2) for all $(X \vdash^? x) \in \mathsf{D}(C)$, $X\Sigma\Phi(C)\sigma{\downarrow} = x\sigma$;
(3) for all $(u =^? v) \in \mathsf{E}^1(C)$, $u\sigma = v\sigma$;
(4) for all $(e \sim^? 0) \in \mathsf{E}^1(C)$ with $\sim \in \{<, \leqslant\}$, we have $e\sigma \in \mathbb{R}$ and $e\sigma \sim 0$ in $\mathbb{R}$;
(5) for all $(\xi =^? \zeta) \in \mathsf{E}^2(C)$, $\xi\Sigma = \zeta\Sigma$;
(6) the satisfaction of first-order formulae is recursively defined in the natural way, interpreting $\forall x$ as a quantifier over ground constructor terms, and $\forall X$ as a quantifier over recipes.

Example 5.1. As an example of constraints, consider again the attack on fair reward introduced in Section 2, and the attack trace in the model mentioned in Section 3. A typical constraint modelling the adversary's constraints during the attack would be the following:

$$C = \mathsf{ax} \vdash^? tx \wedge X{:}\{\mathsf{ax}\} \vdash^? tx' \wedge tx' =^? (x, \mathsf{sign}(x, r, \mathsf{sk}')) \wedge x =^? (\mathsf{pk}(\mathsf{sk}'), \mathsf{solve}, \mathsf{sol})\,.$$

Paraphrasing, the honest transaction tx is public (and accessible through the axiom ax), and the adversary has to compute a transaction tx' from it, whose validity is expressed by the last to equation constraints. A second-order solution of this constraint would be the mapping $\{X \mapsto \xi\}$, with ξ the attack recipe previously detailed in the description of the attack (Section 3, Example 3.2).

We write $Sol(C)$ the set of solutions of C, and refer to Σ and σ as second-order and first-order solutions. This hence formalises the semantics of constraints. Then if we define a *symbolic trace*

$$T : A_{\mathrm{s}}^0 \xrightarrow{\vec{w}_1}_{\mathrm{s}} \cdots \xrightarrow{\vec{w}_n}_{\mathrm{s}} A_{\mathrm{s}}^n$$

as a sequence of $\rightarrow_{\mathrm{s}}$ transitions, and $C(T_{\mathrm{s}}) = C(A_{\mathrm{s}}^n)$, we can formalise the first property linking the symbolic semantics with the operational one. It is *soundness*, intuitively stating that all symbolic traces yield valid regular traces when instantiating their variables w.r.t. their constraints.

Proposition 5.2 (Soundness)**.** *Consider a symbolic trace* $T_{\mathrm{s}} : A_{\mathrm{s}}^0 \xrightarrow{\vec{w}_1}_{\mathrm{s}} \cdots \xrightarrow{\vec{w}_n}_{\mathrm{s}} A_{\mathrm{s}}^n$ *. Then if* $(\Sigma, \sigma) \in Sol(C(T_{\mathrm{s}}))$*, we have:* $T : A_0 \xrightarrow{\vec{w}_1\Sigma} \cdots \xrightarrow{\vec{w}_n\Sigma} A_n$ *with* $A_i = (\mathcal{P}(A_{\mathrm{s}}^i)\sigma,\ \Phi(C(A_{\mathrm{s}}^i))\sigma,\ \theta(A_{\mathrm{s}}^i)\sigma,\ t(A_{\mathrm{s}}^i)\sigma)$*.*

The proof is a straightforward induction on n the length of T_{s}. In the following, we will often refer to the trace T given by the above proposition as the result of *applying* the solution $S = (\Sigma, \sigma)$ to T_s, and write it $T = T_{\mathrm{s}}S$. In particular, completeness ("all regular traces can be abstracted by a symbolic trace") is formalised as:

Proposition 5.3 (Completeness)**.** *Let* $A_{\mathrm{s}}^0 = (\mathcal{P}, C, \theta, t)$ *be a symbolic process,* $(\Sigma, \sigma) \in Sol(C)$*, and let us write* $A = (\mathcal{P}\sigma, \Phi(C)\sigma, \theta\sigma, t\sigma)$ *. Then for all traces* T *of* A*, there exists a symbolic trace* T_{s} *and* $S = (\Sigma \cup \Sigma', \sigma') \in Sol(C(T_{\mathrm{s}}))$ *such that* $T = T_{\mathrm{s}}S$*.*

6 VERIFICATION OF DOLEV-YAO HYPERPROPERTIES

In this section, we built on our symbolic semantics to provide a decision procedure for the verification of arbitrary, guarded hyperformulae. For readability, we only outline some ideas of the procedure and leave the full technical details to Appendices D and E.

6.1 Hyperconstraints

The first ingredient is a refined notion of constraint. Indeed, although the symbolic semantics introduces (sound and complete) constraints for characterising traces finitely, some adequate form of bookkeeping is required when proving hyperproperties due to the multiple path quantifications.

Definition 6.1 (Hyperconstraint system, path projection)**.** A *hyperconstraint* is defined with the same grammar as that of constraints, except that deductions are labelled by path variables:

$$\xi \vdash^?_\pi u \qquad\qquad \forall X.X \not\vdash^?_\pi u \qquad\qquad \textit{(non-)deduction hyperconstraint}$$

In particular, if $\pi \in vars^p(C)$ we define the *projection* of C on path π as the hyperconstraint $\mathrm{proj}_\pi(C)$ obtained by removing those labelled $\pi' \neq \pi$. A *hyperconstraint system* C is then a hyperconstraint such that for all $\pi \in vars^p(C)$, $\mathrm{proj}_\pi(C)$ is a well-defined constraint system. We also extend the notations $\mathsf{D}(C)$, $\mathsf{K}(C)$, $\mathsf{E}^i(C)$ in the natural way. Note however that $\Phi(C)$ is not a well-defined substitution in general, but $\Phi(\mathrm{proj}_\pi(C))$ is one for all $\pi \in vars^p(C)$.

A hyperconstraint is therefore a collection of constraints across different traces, more suited than a plain constraint for proving hyperproperties. In particular, we can define:

Definition 6.2 (Solution of a hyperconstraint system)**.** A *solution* of a hyperconstraint system C is a pair of well-typed substitutions (Σ, σ) such that: (1) $dom(\Sigma) = vars^2(C)$, $dom(\sigma) = vars^1(C)$, $img(\Sigma)$ only contains recipes, and $img(\sigma)$ ground constructor terms; and (2) for all $\pi \in vars^p(C)$, we have $(\Sigma_{|vars^2(\mathrm{proj}_\pi(C))}, \sigma_{vars^1(\mathrm{proj}_\pi(C))}) \in Sol(\mathrm{proj}_\pi(C))$.

Furthermore, we define a notion of *most general solution* (mgs) partly inspired by the analogue notion introduced in [Cheval et al. 2018] for verifying equivalence properties. Intuitively, similarly to mgu for terms, mgs' are partial instantiations of the (second-order) variables of a hyperconstraint system C that characterise all solutions of C: instantiating the pending second-order variables by fresh public names results in an actual solution, and all solutions are instances of a mgs.

Definition 6.3 (Most general solution)**.** A set of *most general solutions* of a hyperconstraint system C, written $mgs(C)$, is a set of well-typed second-order substitutions such that:

(1) for all $(\Sigma, \sigma) \in Sol(C)$, there is $\Sigma_0 \in mgs(C)$ and a well-typed Σ_1 such that $\Sigma = \Sigma_0\Sigma_1$;
(2) for all $\Sigma_0 \in mgs(C)$, $dom(\Sigma_0) \subseteq vars^2(C)$. Also let $V = vars^2(img(\Sigma_0), C) \setminus (dom(\Sigma_0) \cup V_t)$. Also let Σ_1 be a mapping from V to fresh constants such that $X\Sigma_1 = Y\Sigma_1$ *iff* there exist z, π, π' such that $(X \vdash^?_\pi z), (Y \vdash^?_{\pi'} z) \in \mathsf{D}(C)$. Then there is a well-typed σ s.t. $(\Sigma_0\Sigma_1, \sigma) \in Sol(C)$.

In this paper, $mgs(C) = \bot$ expresses that $Sol(C) = \varnothing$. Note that verifying the existence of σ in Item 2 is straightforward: since the symbolic semantics generates a deduction constraint $X \vdash^? x$ for each newly-introduced variable $x \in \mathcal{X}^1$, $\Sigma_0\Sigma_1$ uniquely determines $x\sigma$ provided $x \notin \mathcal{X}^{\mathsf{N}}$. The cases $x \in \mathcal{X}^{\mathsf{N}}$ can then be handled by the black-box time-solving oracle we assume in our procedure (recall Section 4.3). Ordering constraints are the main difference with the original notion of mgs, taken from [Cheval et al. 2018]. In particular, most general solutions are not unique in general, and $mgs(C)$ refers to one arbitrary such set.

6.2 Proof States and Stacks

Hyperconstraints are then part of a bigger structure reflecting the proof skeleton of the desired security property. For that, we define a notion of *proof state*, intuitively modelling a current point in the proof with a list of pending proof obligations, each corresponding to a missing proof to carry for a certain trace. Proof states are then gathered in *proof stacks* capturing the interactions between the different trace quantifications of hyperproperties.

Definition 6.4 (Proof state)**.** A *proof state* is a tuple of the form $\omega = (\Pi, C, \mathcal{O})$, whose components may be written $\Pi(\omega)$, $C(\omega)$, and $\mathcal{O}(\omega)$, and where

- Π is a mapping from $\mathcal{X}^p$ to symbolic traces;
- C is a hyperconstraint system;
- $\mathcal{O}$ is a set of *proof obligations* $o = (T, C')$, T a symbolic trace, C' a hyperconstraint system.

Besides, we call the *domain* of ω the following set:

$$dom(\omega) = \{X \in \mathcal{X}^2 \mid (X \vdash^?_\pi x) \in C(\omega)\}\,.$$

This represents, intuitively, a snapshot of the state of a proof of a hyperproperty at a given point of the proof tree. For example, the hyperconstraint system C (resp. the mapping Π) gathers all constraints (resp. symbolic traces) generated along the proof so far. Importantly, the set $\mathcal{O}$ contains proof obligations (T, C'), indicating that after concluding the current proof, it should be carried again with the last quantified trace replaced by T, and under the additional constraints C. They are then gathered into stacks:

Definition 6.5 (Proof stack)**.** A *proof stack* is a sequence of proof states written $\omega^\star = \omega_1 \rightarrowtail \omega_2 \rightarrowtail \cdots \rightarrowtail \omega_n$ and such that $dom(\omega_1) \subseteq \cdots \subseteq dom(\omega_n)$.

Proof states and stacks will undergo forms of *constraint solving* in order to determine whether their hyperconstraints have solutions. This includes among others a *saturation*, relying on the assumptions on the term algebra to decide deduction hyperconstraints. We give a first intuition of the goal and behaviour of this saturation step below.

Example 6.1. Consider again our running example (Example 3.2). Once the honest transaction *tx* is output to the network, this is reflected as a constraint $\mathsf{ax} \vdash^?_{\pi_1} tx$ meaning that the adversary learns *tx*. The saturation procedure will then add the new hyperconstraint $\mathsf{ax.args} \vdash^?_{\pi_1} \mathsf{sol}$ indicating that the private name sol, modelling the submitted solution, can be extracted using the new recipe ax.args. Other deductions such as $\mathsf{ax.func} \vdash^?_{\pi_1}$ solve are not added as the term solve (a public name) is already deducible.

More generally, the role of the saturation procedure is to apply as destructors as possible to already deducible terms, in order to detect potential new adversarial deductions, as in the above examples. The procedure starts from the outputs leaked during the trace to the adversary, and stops when applying destructors only yields terms that were previously deduced—we call the result a *saturated knowledge base*. This process is guaranteed to terminate due to our restriction to subterm convergent rewriting systems. The details of this constraint-solving procedure can be found in Appendix D, and are used between each step of our decision procedure.

Once saturation is over, it offers a simple characterisation of deducible terms. It has indeed the immediate property that deducible terms need be composed of constructor symbols applied to entries of the saturated knowledge base—which can easily be checked syntactically. This, in turn, permits to frame an algorithm to compute most general solutions of hyperconstraints.

6.3 Main procedure

One key point is that proof states aggregate constraints monotonously during the proof of a hyperproperty. However, some side data are still local during a proof: this is for example the case of the current time t, or the subformula currently being proved. We gather this pieces of local information under an ephemeral analogue of proof states, discarded when moving on to the next proof obligation. It contains, as hinted above, a variable t representing the current timestamp, the formula φ to be proved, but also a mapping $\Pi_@$ indicating the state of all quantified traces at time t (in reference to the notation $T@t$ of Section 3.2). Formally:

Definition 6.6 (Local proof data)**.** A *local proof data* is a tuple $\omega^{-1} = (\Pi_@, t, \varphi)$, where:

- $\Pi_@$ is mapping from $\mathcal{X}^p$ to pairs (A, E) where A is a symbolic process and E is a set of events;
- $t \in \mathcal{X}^N$ and φ is a guarded hyperformula.

To prove a hyperproperty φ, our main procedure then consists of a function HCompute which computes two sets of proof stacks representing, respectively, all cases resulting in φ or $\neg\varphi$ holding. Exactly one of these sets will be empty when φ has no free variables, which is the core criterion for deriving decidability. Formally, this function takes the following form:

$$\mathsf{HCompute}(\omega^\star, \omega^{-1}) = (\Omega_\varphi, \Omega_{\neg\varphi})$$

where $\omega^\star$ is a proof stack, ω^{-1} is a local proof data, and Ω_φ and $\Omega_{\neg\varphi}$ are two sets of proof stacks. We also assume that $dom(\Pi_@(\omega^{-1})) \subseteq dom(\Pi(\omega))$ and at least contains ε and all (non-quantified) path variables appearing in $\varphi(\omega^{-1})$.

Overview of the technical details. Let us give a more concrete intuition of how HCompute operates. For complete technical details, we refer to Appendix E. Consider for example the handling of atomic formulas. Here the role of HCompute is to rely on the constraint solving procedure to resolve a set of characteristic constraints. Let us study the case of an equality modulo rewriting, namely:

$$\mathsf{HCompute}(\omega^\star \rightarrowtail \omega, \omega^{-1}) \qquad \text{with } \varphi(\omega^{-1}) = (u = v)\,.$$

The first step of HCompute is to unify u and v modulo rewriting, that is, to compute $E = mgu_R(u =^? v)$. It then picks a first $\sigma \in E$, and adds it to the constraint of the last stack's state, i.e., $C(\omega)$. By a call to the constraint solving subroutine, it then computes all possible solutions of the updated hyperconstraint.

The results of this resolution are then applied at each level ω_i of the whole stack $\omega^\star$, directly in the hyperconstraint $C(\omega_i)$, or filling a queue of pending proof obligations in $o(\omega_i)$ when several solutions are added. This eventually leads to a new resolved stack $\omega^\star_+$. Analogously, a stack $\omega^\star_-$ can be obtained by considering the negative constraint $\neg\sigma$ instead. The sets $\Omega_{u=v}, \Omega_{u\neq v}$ returned by HCompute are then obtained by computing each of the corresponding stacks $\omega^\star_+$ and $\omega^\star_-$, for all different unifiers $\sigma \in E$. The treatment of other atomic formulas follows the same idea: (1) we characterise them by a set of unifiers or constraints, (2) they are then added to the last stack level, and (3) this triggers a constraint solving whose results are propagated to all stack levels.

In the other (non-atomic) cases, HCompute follows the inductive structure of the formula $\varphi(\omega^{-1})$ to be proved. For example, if $\varphi(\omega^{-1}) = \varphi \Rightarrow \psi$, one first computes $(\Omega_\varphi, \Omega_{\neg\varphi})$ through a recursive call to HCompute. Subsequently, for each proof stack $\omega^\star \in \Omega_\varphi$, another recursive call to HCompute is used to, overall, aggregate the sets $\Omega_{\varphi\wedge\psi}$ and $\Omega_{\varphi\wedge\neg\psi}$. The final result of the procedure is then, using the identity $(\varphi \Rightarrow \psi) = \neg\varphi \vee \psi$:

$$(\Omega_{\varphi\Rightarrow\psi}, \Omega_{\neg(\varphi\Rightarrow\psi)}) = (\Omega_{\neg\varphi} \cup \Omega_{\varphi\wedge\psi}, \Omega_{\varphi\wedge\neg\psi})\,.$$

The only notable exception is the case of path quantifiers, which creates, on top of that, a new stack level (and yields an additional recursive call to the next proof obligation once the proof at more recent levels are finished).

Correctness. We now state the main correctness property of HCompute. To obtain the desired **EXPH**(poly) bound, we consider the *naive n-bounded algorithm* $\mathsf{NBA}_n(P, \varphi) \in \{\mathsf{true}, \mathsf{false}\}$, which is the brute force alternating (but incomplete) algorithm for proving $P \models \varphi$ which bounds the size of adversarial computations by n to obtain a finite set of traces. Its definition is straightforward, and similar to HCompute but operates on concrete processes and traces, and hence uses concrete recipes (of size n at most) instead of deduction constraints.

Proposition 6.2 (Complexity and Soundness of NBA). *The procedure* $\mathsf{NBA}_n(P,\varphi)$ *runs in alternating polynomial time in* n *and the sizes of* P *and* φ*, and also involves a polynomial number of alternations. Besides, for some initial parameters* $\omega^\star, \omega^{-1}$ *(technically defined in Appendix E), let us write* $(\Omega_\varphi, \Omega_{\neg\varphi}) = \mathsf{HCompute}(\omega^\star, \omega^{-1})$ *and assume*

$$n \geqslant \mu(\Omega_\varphi \cup \Omega_{\neg\varphi}) \triangleq \max_{\substack{(\omega^\star \rightarrowtail \omega) \in \Omega_\varphi \cup \Omega_{\neg\varphi} \\ \pi \in vars^P(\omega)}} \mu_\pi(C(\omega))$$

where $\mu_\pi(C)$ *refers to the size of a mgs of* C *projected on path* π*, i.e.,* $mgs(C)_{|vars^2(\mathrm{proj}_\pi(C))}$*. Then* $P \models \varphi$ *iff* $\mathsf{NBA}_n(P,\varphi)$ *accepts.*

This states that NBA_n is a sound procedure provided n is greater than all (projections of) mgs generated by HCompute. To justify that the Verif problem is in **EXPH**(poly) under our assumptions, it therefore remains to establish a small solution property, that is, that $\mu_\pi(C)$ is bounded by an exponential in the inputs of the problem. These properties are proved in Appendix E.2.

7 REVIEW OF APPLICATIONS OF DY-HYPERPROPERTIES

To conclude, we review here some potential applications of hyperproperties in the context of security protocols, in addition to the running example. We present these examples in an informal style, the translation of the security property in our logic being often straightforward.

7.1 Distribution fairness

We first review a few case studies from [Barthe et al. 2022], which lists examples of hyperproperties in real-time contexts. They feature either trace properties (lacking relevance to our paper), equivalence properties (out of our scope), or liveness properties in the tidy CTL* fragment. We will focus on the third kind. Notably, although omitted in this section, similar properties as those listed below arise in the context of optimistic fair exchange, studied for example in [Backes et al. 2017].

The applications we consider rely on *time cryptography*, which are primitives that require, by design, a significant amount of time to be computed. This allows to lock sensitive data from the adversary during a time window, and corresponds to notions of *cost* in our calculus term algebra. We cleared this component from our framework to simplify its already dense presentation, but protocol variants may rely instead on trusted third parties holding data until predetermined deadlines. This is used to enforce notions of (distribution) *fairness*, meaning that either all participants get the protocol's output, or none does. This typically involves the following core notions.

(1) **Stuck:** the agent cannot do anything but wait for some event A that is, say, out of its control.
(2) **Progress:** at any point in time, non-stuck agents proceed until being stuck.
(3) **Activeness:** agents do not get stuck by passively waiting, e.g., that some timeout expires.

```
formula stuck =
  ∀pi:trace. F not silent_pi =>
    silent_pi U A
formula progress =
  G (not stuck => F stuck)
formula activeness(pi:trace) =
  G (not stuck => F not silent_pi)
```

Fig. 10. Progress-related notions

We define these predicates in Figure 10, sticking to the programming-like syntax of Section 2. Various security properties of the form $\forall\pi.\mathsf{Progress} \Rightarrow \varphi$, or $\forall\pi.\mathsf{Progress} \Rightarrow \mathsf{Active}_\pi \Rightarrow \varphi$, can then be used to formalise fairness in different contexts. One instance is the organisation of auctions as described in [Boneh and Naor 2000], where participants lock their bid until a common date t in the future. They are then expected to reveal their bid at time t, but should have backup solutions to recover those of non-cooperating parties (typically forceable time cryptography or a trusted third party). The desired security property is then that, in presence of at least one progressing agent, the protocol always goes through until the end once the bids have all been submitted.

Another example of such properties would be contract signing protocols such as the one of [Boneh and Naor 2000]. Here two mutually distrusting parties A and B want to exchange signatures with some guarantee that they will get the other's. The main idea is to rely on a locking mechanism expiring after some (parametric) time t. Concretely, A and B exchange their signatures, locked for a prohibitive amount of time (say, $t = 2^\eta$ for some high security parameter η). They then repeat this protocol, decrementing η at each new round. If both parties cooperate, they get each other's signatures in η rounds. If one party aborts during round p, they will have to wait at least time $2^{\eta-p}$ to get the signature; meanwhile, the worse-off party waits at most time $2^{\eta-p+1}$, i.e., at most twice as much. The desired security property would therefore read as:

> *for all trace π, if either participant obtains the other's signature at time T, then the other can get it at time $T + 2T = 3T$, provided it is actively progressing.*

As a side note, [Barthe et al. 2022] proposes a semi-automated approach to prove real-time properties specified in their framework, which is however only sound for trace properties. The examples mentioned above typically fall into the scope of our decidability result but escape the one of the tool of [Barthe et al. 2022]. Albeit for our lack of experimental results, this highlights the improvements induced by our procedure in terms of theoretical contributions.

7.2 Atomicity in Smart Contracts

We now give other examples inspired by the domain of smart contracts. Some of the presented hyperproperties may have some verification support in the literature, but most of them do not account for scenarios involving an active Dolev-Yao adversary (see related work, Section 8).

Smart contracts are often written in a quasi-Turing-complete language, meaning here that the language itself is Turing-complete but the number of computation steps for the execution of each contract is bounded by a resource called *gas*. When invoking a contract, the caller specifies an upper bound on the gas to be consumed by the execution. Each instruction consumes (a positive amount of) gas and halts with an exception when running out of gas, reverting the effects of the contract execution. However, widely used smart contract languages do not support exception propagation, meaning that a caller actively needs to read the error code returned by the failed contract invocation and handle the exception appropriately. This subtlety causes a lot of issues in practice since contract developers forget to implement checks to ensure that calls have been finished successfully, resulting in unhandled exceptions and potentially inconsistent state. To give a semantic characterisation of this property, [Grishchenko et al. 2018] defines a notion called *atomicity* which intuitively states:

> *for all executions π_1, π_2 of the contract C, if the initial states of π_1, π_2 only differ in the amount of gas given to them then either π_1 and π_2 have the same final state, or either π_1 or π_2 fully rolled back to the initial state of their executions.*

This ensures that no problematic state can arise due to unhandled out-of-gas exceptions. In our framework, gas can easily be modelled as part of the global state, and thus even referenced in security properties through the GS atomic formula.

7.3 Reentrancy

Reentrancy bugs are a prominent class of smart contract bugs that have caused tremendous financial losses in practice. Intuitively, a smart contract is considered reentrant if it can be called again during its execution—which is usually not part of the desired program logic. It typically occurs if a smart contract C invokes another contract C' and this one calls C back. This is often not anticipated by contract developers and may lead to inconsistent states and unwanted behaviours.

For characterising the absence of reentrancy bugs, different approaches have been taken. One approach, taken in [Grishchenko et al. 2018] is to consider the absence of reentrancy bugs as

an integrity property focusing on the additional power that an external contract may gain over the execution of a contract C through reentering the contract. Intuitively, *call integrity* states that no other contract should be able to influence how contract C spends money (money transfer transactions are considered security-critical actions in the contract setting). Assuming such critical transactions are identified in the process using an event Critical/1, the property can intuitively be stated by the hyperproperty, in a non-interference style fashion:

> *for all executions π_1, π_2 of the contract C reaching a final state, π_1 and π_2 should trigger the same events* Critical(x) *simultaneously.*

A different approach is taken in [Grossman et al. 2017]. Here, notions from concurrency theory (serialisability/linearisability) are leveraged to distinguish between benign and reentrancies and such that may expose problematic behaviour. The authors introduce the notion of *effective callback freedom* that states that a contract C is effectively callback free if:

> *for all executions π_1 that may involve reentering executions (callbacks), there exists an execution π_2 of C not involving any callbacks and with the same initial and final states.*

A model of this property in our framework would typically require to model a callstack, which again can easily be embedded in our notion of global state.

7.4 Virtual Channel Stability

Payment channels [Poon and Dryja 2016] are cryptographic protocols that help blockchain scalability. They enable two users to conduct fast payments (called *offchain payments*) between each other without registering every single payment on the blockchain. Instead, the parties initially lock the funds they may want to exchange in a shared account and conduct payments by exchanging cryptographic guarantees that enable them to withdraw the corresponding amounts from the shared account upon channel closure. The channel parties can decide to close the channel at any time using the guarantees obtained before, retrieving their rightful amount from the shared account.

Virtual channels [Dziembowski et al. 2019] generalise payment channels to create new channels upon existing ones. This comes with the advantage that users do not need to individually lock funds with each party with who they want to perform off-chain payments. Instead, a user A can use a portion of the funds locked with user B in channel ch_{AB} to open a new virtual payment channel ch_{AC} with party C (given that parties B and C also have a payment channel ch_{BC}). This construction can be used recursively to build hierarchies of virtual channels as depicted in Figure 11.

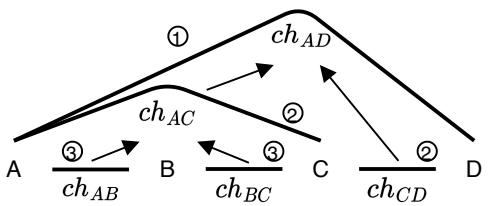

Fig. 11. Recursive construction of virtual payment channels

Some virtual channel constructions [Aumayr et al. 2021] can only guarantee security if the closure of a virtual payment channel also initiates the closure of all underlying channels. This property has been identified in [Aumayr et al. 2023] to give rise to the *Domino Attack*: an attacker can create a hierarchy of virtual channels and make the whole channel hierarchy collapse—forcing also the closure of channels between honest users. In the example, in Figure 11, user A and D can

collude to close the channel ch_{BD} between honest users B and C. To this end they first create ch_{AC} based on ch_{AB} and then channel ch_{AD} leveraging ch_{AC}. Closing ch_{AD} ① will force party C to close ch_{AC} (and ch_{CD}) ② and this again will force B to close ch_{BC} (and ch_{AB}) ③. Even though honest users are guaranteed not to lose channel funds in such an attack, they still need to cover the cost of closing and reopening their channels—both of which require the publication of transactions.

The absence of the domino attack, hence, is a desirable property that can be achieved by the virtual channel construction presented in [Aumayr et al. 2023]. It can be formulated as a hyperproperty that we call here *virtual channel stability*:

> *for all channels ch and all executions π_1 where ch is closed, there exists an execution π_2 closing ch where all channels ch′ ($ch \neq ch'$) between honest users remain open.*

In our model, at a high level of abstraction, a virtual channel can be seen as a local blockchain only readable by some participants. We, therefore, refer to the axiomatisation provided in Section 2.

8 RELATED WORK

Dolev-Yao Models. Symbolic models, inspired by early works of Dolev and Yao [Dolev and Yao 1983], support a large range of languages for specifying protocols and security properties. Among the most popular protocol languages, one may cite:

(1) the applied π-calculus as in the ProVerif tool [Blanchet et al. 2020] which is a calculus of communicating concurrent processes; and
(2) multiset rewrite rules as in the Tamarin tool [Basin et al. 2019] which is a formalism defining protocols as transition systems.

To make our results mostly independent of the specification language, we designed our framework to subsume most of their features. Existing tools also usually focus on reachability properties formalised in variants of (temporal) first-order logics, commonly including a notion of guarded quantifiers as ours [Basin et al. 2019]. The notable exception is *indistinguishability* properties, formalising security as an equivalence between two processes, modelling strong flavours of privacy-type properties [Rakotonirina 2021]. They may be seen as a variant of the non-interference of HyperCTL* [Clarkson et al. 2014], although significantly more complex due to non-determinism, and as the notion is defined for an adversary actively controlling communications. Regarding automation, the bounded case has seen numerous decidability and complexity results [Chadha et al. 2016; Cheval et al. 2013, 2018, 2020a; Durgin et al. 2004; Kanovich et al. 2014; Liu and Lin 2012; Rusinowitch and Turuani 2003]. The underlying procedures often rely on an abstraction of the system using a *symbolic semantics*, similarly to us.

One may also mention different lines of work such as the DY* framework [Bhargavan et al. 2021]. It proposes a method to specify and prove symbolic properties in the F* proof assistant. Although some approaches exist to handle relational properties within F* [Barthe et al. 2014], it is unclear how to adapt this to DY* whose proof machinery (relying on establishing trace invariants) is heavily specialised in trace properties.

Hyperproperties. Hyperproperties have been increasingly popular in the context of verification, often expressed in variants of the temporal logic HyperCTL* [Clarkson et al. 2014]. It considers finite-state systems whose traces are infinite sequences of transitions. The logic can then perform arbitrary quantifications over traces, and then expresses properties through a classical temporal logic. This contrasts with standard Dolev-Yao frameworks, which consider traces that are finite in length, but where each step may branch to infinitely-many potential states due to unbounded adversary operating in parallel to the system. Our framework is rooted in the latter, but we use a logic, Hypertidy CTL* [Barthe et al. 2022], whose design is heavily inspired by HyperCTL*.

An important aspect of logics for specifying protocol properties is how to state facts about the adversary, typically to express properties such as secrecy ("*in any trace of the system, the adversary has no way to compute a certain secret piece of data s*"). This is usually done using a "knowledge" predicate $\mathsf{K}(s)$ as in this paper. Notions of knowledge have also been considered in finite-state systems using, e.g., epistemic-logic approaches [Balliu et al. 2011; Coenen et al. 2019], but they do not account for the active adversary. One may still note that epistemic models of knowledge have also been marginally studied in Dolev-Yao models [Benevides and Fernandez 2021].

Several other extensions of temporal logics somehow bring them closer to the adversarial setting needed for our applications, although never accounting for all necessary features. A recent extension considers for example quantifications over sets of traces [Beutner et al. 2023], thus capturing more complex epistemic properties as the previously-cited references, such as common knowledge (which is orthogonal to our contributions). Other extensions also consider limited parts of what is the basis of Dolev-Yao models. Typically, [Beutner and Finkbeiner 2022] formalises hyperproperties in the presence of multiple agents (natively included by the concurrent nature of Dolev-Yao models); [Coenen et al. 2022; Finkbeiner et al. 2023] introduce forms of function symbols (but do not support mechanisms for specifying cryptographic properties); and [Hsu et al. 2023] adds support for asynchronous communications (the standard mode of communication in Dolev-Yao models) and de-synchronised behaviours across multiple traces (implicitly captured by the fact that the Hypertidy CTL* logic for Dolev-Yao hyperproperties supports real-time). They thus capture applications from concurrency theory such as linearisability, or security properties in the presence of an adversarial scheduler, but do not support the standard adversary in protocol analysis which fully controls communications by intercepting, forging, and injecting messages.

Finally, let us mention that existing work on the proof automation of hyperproperties in HyperCTL* tends to be limited to the fragment HyperLTL [Beutner et al. 2023; Coenen et al. 2022; Finkbeiner et al. 2023]. Some approaches also commonly struggle with quantifier alternation, the fragment $\forall^*\exists^*$ being a common limit [Finkbeiner et al. 2023]. Our results are not subject to these restrictions, and support in particular arbitrary alternation of (nested) quantifiers.

Verification of Smart Contracts. As we show several application cases of our decidability result in the field of blockchain and smart contracts, we briefly review some related results on their verification. Most existing work on smart contract verification is concerned with functional contract correctness usually phrased as trace properties [Ahrendt and Bubel 2020; Marescotti et al. 2020; Permenev et al. 2020; Stephens et al. 2021]. [Grishchenko et al. 2018] defines multiple generic non-interference-style integrity properties for smart contracts that aim at ruling out prominent attack classes (see call integrity in Section 7.3). Some of them can be verified with the help of a local dependency analysis of the contract code that soundly models the contract's control and data dependencies [Holler et al. 2023]. The call integrity property does not fall into this class of properties but is shown in [Grishchenko et al. 2018] to be soundly over-approximated by a reachability property and two other local dependency properties. Correspondingly, it can be verified using tools supporting sound local dependency analysis [Holler et al. 2023] and sound reachability analysis [Schneidewind et al. 2020] for smart contracts. [Grossman et al. 2017] presents an alternative approach to characterising the absence of reentrancy attacks by defining *effective callback freedom (ECF)* (see Section 7.3). [Albert et al. 2020] shows a verification technique for ECF that relies on the exploitation of commutative properties of contract code segments. The verification approaches for local dependency properties, call integrity and ECF have in common that they are tailored to the concrete properties and correspondingly narrow in scope. These techniques, in particular, overapproximate only those aspects of the attacker's behaviour that are relevant to the property under verification. Further, they do not account for the usage of cryptography.

[Coenen et al. 2022] provides examples of hyperproperties in the context of smart contracts, and presents a procedure for synthesising contracts satisfying them. Hyperproperties are expressed in HyperTSL, a generalisation of temporal stream logic (TSL) to support hyperproperties. Examples include symmetry and determinism notions for voting and auction contracts. Even though HyperTSL supports uninterpreted function symbols and predicates, there is no support for symbolic cryptography or real-time. Also, this synthesis technique is restricted to all-quantified formulae.

9 CONCLUSION AND FUTURE WORK

We provided the first decidability and (tight) complexity results for a large fragment of hyperproperties in contexts involving active adversaries. Our results apply to an expressive class of processes supporting arithmetic computations, cryptography, concurrency and stateful programs. The main open problem left by our contribution is the generalisation of our decidability result to the full Hypertidy CTL* logic, as our fragment (guarded hyperformulae) excludes classical notions of adversarial indistinguishability. Although such properties are already well-studied, and got a certain automated support [Blanchet et al. 2020; Cheval et al. 2020b], it is still open how existing proof techniques would interact with our decision procedure and its complexity.

Another point would be the implementation of our procedure, as we focused here on pinpointing its exact complexity using a theoretical alternating algorithm. As practical examples (see Section 7) rarely involve more than 2 or 3 path quantifications, we conjecture the theoretical **EXPH**(poly) to be rather pessimistic, compared to how actual verifiers could perform in practice when implementing our HCompute function. Similarly, the **EXPH**(poly) hardness proved in [Barthe et al. 2022] relies on rather degenerate examples, whose term algebra model binary trees to force the adversary into performing computations of exponential size. On the contrary, although attacks on practical protocols are tedious to find by hand, the adversarial computations they involve are arguably rather of linear size in the protocol's description. The reason is that, most of the time, their objective is precisely to impersonate or imitate some actions of honest instances of the specification, and this is done by combining messages from a small number of sessions. Similar observations arose in related tools such as DeepSec [Cheval et al. 2020b] which, although exhibiting a prohibitive co**NEXP** theoretical complexity, performs surprisingly well by relying on practical optimisations exploiting the relative simplicity of practical examples compared to the fully-general case [Cheval et al. 2019]. However, at this point, our contribution is mostly theoretical and a non-negligible engineering effort would still be necessary to design and implement such techniques, leaving the experimental evaluation of our procedure as a challenging future work.

REFERENCES

Martín Abadi, Bruno Blanchet, and Cédric Fournet. 2018. The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. *Journal of the ACM (JACM)* (2018).

Martín Abadi and Véronique Cortier. 2006. Deciding knowledge in security protocols under equational theories. *Theoretical Computer Science* 367, 1-2 (2006), 2–32.

Wolfgang Ahrendt and Richard Bubel. 2020. Functional Verification of Smart Contracts via Strong Data Integrity. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 12478)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 9–24. https://doi.org/10.1007/978-3-030-61467-6_2

Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020. Taming callbacks for smart contract modularity. *Proceedings of the ACM on Programming Languages* (2020).

Bowen Alpern and Fred B. Schneider. 1985. Defining Liveness. *Inf. Process. Lett.* 21, 4 (1985), 181–185. https://doi.org/10.1016/0020-0190(85)90056-0

Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, et al. 2012. The AVANTSSAR platform for the automated validation of

trust and security of service-oriented architectures. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, Estonia, 267–282.

Lukas Aumayr, Matteo Maffei, Oğuzhan Ersoy, Andreas Erwig, Sebastian Faust, Siavash Riahi, Kristina Hostáková, and Pedro Moreno-Sanchez. 2021. Bitcoin-compatible virtual channels. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 901–918.

Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2023. Breaking and Fixing Virtual Channels: Domino Attack and Donner. In *Network and Distributed System Security Symposium (NDSS)*.

Kushal Babel, Vincent Cheval, and Steve Kremer. 2020. On the semantics of communications when verifying equivalence properties. *Journal of Computer Security* (2020).

Michael Backes, Jannik Dreier, Steve Kremer, and Robert Künnemann. 2017. A novel approach for reasoning about liveness in cryptographic protocols and its application to fair exchange. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.

Musard Balliu, Mads Dam, and Gurvan Le Guernic. 2011. Epistemic temporal logic for information flow security. In *Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, USA.

Gilles Barthe, Ugo Dal Lago, Giulio Malavolta, and Itsaka Rakotonirina. 2022. Tidy: Symbolic Verification of Timed Cryptographic Protocols. In *ACM Conference on Computer and Communications Security (CCS)*.

Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. 2014. Probabilistic relational verification for cryptographic implementations. *ACM SIGPLAN Notices* 49, 1 (2014), 193–205.

David Basin, Cas Cremers, Jannik Dreier, Simon Meier, Ralf Sasse, and Benedikt Schmidt. 2019. *Tamarin prover manual*. https://tamarin-prover.github.io/.

Mathieu Baudet. 2007. *Sécurité des protocoles cryptographiques: aspects logiques et calculatoires*. Ph. D. Dissertation. École normale supérieure de Cachan.

Mario RF Benevides and Luiz CF Fernandez. 2021. Tableaux Calculus for Dolev-Yao Multi-Agent Epistemic Logic. *Logical and Semantic Frameworks with Applications (LSFA)* (2021).

Raven Beutner and Bernd Finkbeiner. 2022. A Logic for Hyperproperties in Multi-Agent Systems. *arXiv preprint arXiv:2203.07283* (2022).

Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, and Niklas Metzger. 2023. Second-order hyperproperties. *arXiv preprint arXiv:2305.17935* (2023).

Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 523–542.

Bruno Blanchet. 2012. Security protocol verification: Symbolic and computational models. In *International Conference on Principles of Security and Trust (POST)*. Springer, Estonia, 3–29.

Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. 2020. *Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. https://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf.

Dan Boneh and Moni Naor. 2000. Timed commitments. In *Annual international cryptology conference (CRYPTO)*. Springer, Santa Barbara, CA, USA, 236–254.

Rohit Chadha, Vincent Cheval, Ştefan Ciobâcă, and Steve Kremer. 2016. Automated verification of equivalence properties of cryptographic protocol. *ACM Transactions on Computational Logic* (2016).

Vincent Cheval, Véronique Cortier, and Stéphanie Delaune. 2013. Deciding equivalence-based properties using constraint solving. *Theoretical Computer Science* (2013).

Vincent Cheval, Charlie Jacomme, Steve Kremer, and Robert Künnemann. 2022. Sapic+ : protocol verifiers of the world, unite!. In *USENIX Security Symposium*.

Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. 2018. DEEPSEC: Deciding equivalence properties in security protocols theory and practice. In *IEEE Symposium on Security and Privacy (S&P)*.

Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. 2019. Exploiting symmetries when proving equivalence properties for security protocols. In *ACM Conference on Computer and Communications Security (CCS)*.

Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. 2020a. The hitchhiker's guide to decidability and complexity of equivalence properties in security protocols. In *Logic, Language, and Security. Essays Dedicated to Andre Scedrov on the Occasion of His 65th Birthday (ScedrovFest65)*.

Vincent Cheval, Steve Kremer, Itsaka Rakotonirina, and Victor Yon. 2020b. DeepSec user manual. https://deepsec-prover.github.io/.

Michael R Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K Micinski, Markus N Rabe, and César Sánchez. 2014. Temporal logics for hyperproperties. In *International Conference on Principles of Security and Trust (POST)*. Springer, Grenoble, France, 265–284.

Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. 2019. The hierarchy of hyperlogics. In *ACM/IEEE Symposium on Logic in Computer Science (LICS)*.

Norine Coenen, Bernd Finkbeiner, Jana Hofmann, and Julia Tillman. 2022. Smart Contract Synthesis Modulo Hyperproperties. *arXiv preprint arXiv:2208.07180* (2022).

Hubert Comon-Lundh and Stéphanie Delaune. 2005. The finite variant property: How to get rid of some algebraic properties. In *International Conference on Rewriting Techniques and Applications (RTA)*. Springer, Poland, 294–307.

Danny Dolev and Andrew Yao. 1983. On the security of public key protocols. *IEEE Transactions on information theory* 29, 2 (1983), 198–208.

Nancy A. Durgin, Patrick Lincoln, and John C. Mitchell. 2004. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security* 12, 2 (2004), 247–311.

Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. 2019. Perun: Virtual payment hubs over cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 106–123.

Bernd Finkbeiner, Hadar Frenkel, Jana Hofmann, and Janine Lohse. 2023. Automata-based software model checking of hyperproperties. In *NASA Formal Methods Symposium*. Springer, 361–379.

Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. 2015. Algorithms for Model Checking HyperLTL and HyperCTL*. In *Computer Aided Verification (CAV)*. Springer, San Fransisco, CA, USA, 30–48.

Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust (POST)*. Springer.

Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages* (2017).

Sebastian Holler, Sebastian Biewer, and Clara Schneidewind. 2023. HoRStify: Sound Security Analysis of Smart Contracts. In *2023 2023 IEEE 36th Computer Security Foundations Symposium (CSF)(CSF)*. IEEE Computer Society, 347–362.

Tzu-Han Hsu, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. 2023. Bounded Model Checking for Asynchronous Hyperproperties. *arXiv preprint arXiv:2301.07208* (2023).

Max I. Kanovich, Tajana Ban Kirigin, Vivek Nigam, and Andre Scedrov. 2014. Bounded memory protocols. *Computer Languages, Systems & Structures* (2014).

Steve Kremer and Robert Künnemann. 2016. Automated analysis of security protocols with global state. *Journal of Computer Security (JCS)* (2016).

Jia Liu and Huimin Lin. 2012. A complete symbolic bisimulation for full applied pi calculus. *Theoretical Computer Science* 458 (2012), 76–112.

Matteo Marescotti, Rodrigo Otoni, Leonardo Alt, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. 2020. Accurate Smart Contract Verification Through Direct Modelling. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 12478)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 178–194. https://doi.org/10.1007/978-3-030-61467-6_12

Corto Mascle and Martin Zimmermann. 2019. The keys to decidable hyperltl satisfiability: Small models or very simple formulas. arXiv preprint arXiv:1907.05070.

Marvin Lee Minsky. 1967. *Computation*. Prentice-Hall Englewood Cliffs, USA. p.255-258.

Anton Permenev, Dimitar K. Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1661–1677. https://doi.org/10.1109/SP40000.2020.00024

Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments. (2016).

Itsaka Rakotonirina. 2021. *Efficient verification of observational equivalences of cryptographic processes: theory and practice*. Ph. D. Dissertation. Université de Lorraine.

Michaël Rusinowitch and Mathieu Turuani. 2003. Protocol insecurity with a finite number of sessions, composed keys is NP-complete. *Theoretical Computer Science* 299, 1-3 (2003), 451–475.

Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 621–640. https://doi.org/10.1145/3372297.3417250

Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu K. Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 555–571. https://doi.org/10.1109/SP40001.2021.00085

A COMPLEXITY THEORY (PRELIMINARIES)

In this section, we recall standard notions of computational complexity theory, used across the paper. We focus in this paper on worst-case complexity.

A.1 Basic Notions

Time and Space. Given $f : \mathbb{N} \to \mathbb{N}$, we define $\mathsf{TIME}(f(n))$ (resp. $\mathsf{SPACE}(f(n))$) to be the class of problems decidable by a deterministic Turing machine running in time (resp. in space) at most $f(n)$ for input tapes of length n. In particular:

- $\mathbf{P} = \bigcup_{p\in\mathbb{N}} \mathsf{TIME}(n^p)$ is the class of problems decidable in polynomial time;
- $\mathbf{PSPACE} = \bigcup_{p\in\mathbb{N}} \mathsf{SPACE}(n^p)$ is the class of problems decidable in polynomial space;
- their exponential analogues are $\mathbf{EXP} = \bigcup_{p\in\mathbb{N}} \mathsf{TIME}(2^{n^p})$, $\mathbf{EXPSPACE} = \bigcup_{p\in\mathbb{N}} \mathsf{SPACE}(2^{n^p})$.

When considering non-deterministic Turing machines instead of deterministic ones as above, we add a "N" prefix to the name of the class, leading, e.g., to **NP** (non deterministic polynomial time) and **NEXP** (non deterministic exponential time). Given a (non-deterministic) class C, we call co$C$ the class of problems whose negation is in C.

Complete Problems. We say that a problem is complete for a class C if any problem of C can be reduced to this problem. In this paper, the notion of reduction used is the standard *many-to-one* polynomial-time reduction. We assume some familiarity of the reader with the underlying notions, and with the prototypical complete problem for **NP** (SAT or 3-SAT).

Alternation. We also recall that *alternating Turing machines* are non-deterministic Turing machines whose states are partitioned between *universal states* (or $\forall$-states) and *existential states* (or $\exists$-states). An execution from a universal (resp. existential) state is then accepting *iff* any (resp. at least one) execution from this state accepts. In particular, non-deterministic Turing machines are alternating Turing machines with only existential states, whereas arbitrary alternation of types of states is allowed in general. We add a "A" prefix to express that we consider alternating Turing machines, e.g., **AP** is the class of problems decidable in alternating polynomial time. Typical known relations between these classes include:

$$\mathbf{P} \subseteq \mathbf{NP}, \text{co}\mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} = \mathbf{AP} \subseteq \mathbf{EXP}$$
$$\mathbf{EXP} \subseteq \mathbf{NEXP}, \text{co}\mathbf{NEXP} \subseteq \mathbf{EXPSPACE} = \mathbf{AEXP}$$

A.2 Polynomial and Exponential Hierarchies

The gap between **NP** and **PSPACE** = **AP** is intuitively alternation, as **NP** may be seen as the class of problems decidable in polynomial time by alternating machines with only existential states. Intermediary levels of alternation characterise a whole hierarchy of alternation-bounded complexity classes between **NP** and **PSPACE**, called the *polynomial hierarchy* **PH**. We reference in this paper in the analogue hierarchy between **NEXP** and **EXPSPACE** (*exponential hierarchy*), the highest level of which being however more refined (depending on whether arbitrary, or only a polynomial number of, alternations are allowed). For simplicity, we only provide here the usual characterisation of the class we are interested in, **EXPH**(poly):

Proposition A.1. ***EXPH****(poly), called the polynomially-bounded exponential hierarchy, is the class of problems decidable in exponential time by alternating Turing machines with a polynomial number of alternations. We have:*

$$\boldsymbol{EXP} \subseteq \boldsymbol{EXPH}(poly) \subseteq \boldsymbol{EXPSPACE}$$

B EXPRESSIVITY OF THE FRAMEWORK

Our presentation focuses on expressivity with minimal syntax, to simplify our proofs while making our decidability results applicable to a large class of protocol features. However, for readability, the concrete examples presented throughout the paper relied on additional convenient features that can be encoded in our minimal theoretical core.

```
formula PEq(pi:trace) =
  G (∀x,y:bitstring.
    Equal(x,y)pi =>
      x = y)
formula PNotEq(pi:trace) =
  G (∀x,y:bitstring.
    NotEqual(x,y)pi =>
      x ≠ y)
```

Fig. 12. Axiomatisation of tests

For example, conditionals (if) are standardly encoded using choice and events [Kremer and Künnemann 2016]. A process `if u = v: P else: Q` would therefore be encoded as

```
Equal(u,v): P + NotEqual(u,v): Q
```

for two events `Equal` and `NotEqual`. When proving a hyperproperty `phi`, it then suffices to replace all path quantifiers `forall pi:trace. psi` appearing in `phi` by

```
forall pi:trace. PEq(pi) => PNotEq(pi) => psi
```

where `PEq` and `PNotEq` are defined in Figure 12. Also, as detailed in Section 3.2, Example 3.1, our original syntax for global state can encode notions of mutable cells or arrays. Simple note that, given for example a mutable table T encoded this way as in the running example, the state of T can be accessed in hyperproperties by using the GS atomic formula of the logic. It can also be used to perform pattern matching; given a dedicated symbol `match` taking two arguments, the following process:

```
new k : push match(u,k) : pull match(f(x),k) : P
```

only proceeds to execute `P` if the term `u` is of the form `f(x)` (assuming a unary function symbol `f` in the term algebra), and binds `x` to the corresponding term in `P`. This can easily be generalised to arbitrary pattern matchings of constructors or to patterned let bindings, see [Cheval et al. 2022] for details of such encodings.

By composing push and pull instructions atomically, one can more generally encode *multiset rewrite rules* as used in, e.g., the Tamarin prover. Such rules constitute an alternative specification formalism for processes, operating on a global state similar to ours through rules of the form:

$$[u_1, \ldots, u_n] \; -[\mathsf{Ev}_1, \ldots, \mathsf{Ev}_p]\rightarrow \; [v_1, \ldots, v_q] \; .$$

Here u_i, v_j can be seen as terms (except for special cases representing inputs, outputs, or new operations), and Ev_k event terms. This rule as the effect of removing all terms u_i from the multiset, adding all terms v_j afterwards, and flagging this operation with the Ev_k events. This can be simulated in our framework by:

$$\mathsf{pull}\; u_1 : \cdots : \mathsf{pull}\; u_n : \mathsf{Ev}_1 : \cdots : \mathsf{Ev}_p : \mathsf{push}\; v_1 : \cdots \mathsf{push}\; v_q \; .$$

In case of special instructions, pull may be substituted by inputs or new instructions, and push by outputs. Using this, any feature encodable using multiset rewrite rules can be encoded, e.g. referring to [Cheval et al. 2022; Kremer and Künnemann 2016], associative maps or state passing.

Going further, dialects of the applied π-calculus [Blanchet et al. 2020] usually offer more communication features such as explicit *communication channels*, and *internal communications* (synchronous message transfer without adversarial interference). Channels can easily be encoded in our framework by atomically composing events to inputs or outputs. Internal communications can also be simulated by sending messages through the global state, and thereafter pulling an acknowledgment of receipt before proceeding. This multiple-step encoding is equivalent to fully synchronous variants, at least in models without real time (see, e.g., the correct translation of [Kremer and Künnemann 2016]). In our framework, it yields time-desynchronised internal communications, that

is, the output and the input are not executed at the same timestamp. This is however, in our opinion, arguably more realistic as it avoids artificial time synchronisations between parallel processes. One may then also encode within security properties, thanks to our atomic-composition syntax, various channel and communication features of interest. This includes, e.g., public communications only allowed on compromised channels, internal communications only with matching channels, resilient channels [Backes et al. 2017], or variations of the semantics of internal communications [Babel et al. 2020]. This makes our decidability results largely independent of the framework's details.

C PROOF OF UNDECIDABILITY

C.1 Reduction From Two-Counter Machines

Restricted Model. In this section, we study the general undecidability of the Verif problem stated in Section 4.1:

Proposition 4.1 (Verification of arbitrary processes)**.** *Verif is undecidable even when* $\mathcal{F}_{\mathrm{c}} = \mathcal{F}_{pub} = \{\mathsf{h}/1\}$, $\mathcal{F}_{\mathrm{d}}$ *and* $\mathcal{R}$ *are empty, and* φ *is a tidy LTL formula.*

We prove that under the restrictions of the above proposition, the model can be used to encode two-counter machines, a classical Turing-complete computation model [Minsky 1967]. In particular, the verification of tidy LTL will easily encode (undecidable) reachability problems such as the halting problem. This may however be seen as a shallow statement, as our calculus notably supports shared global states (which can trivially encode counters, even without a free symbol $\mathsf{h}/1$). To emphasise the minimality of this undecidability result, in particular in common models not supporting global states, real-time, or atomic composition [Cheval et al. 2018], we will strengthen it with the following restrictions:

(1) we use an untimed model, that is, no when instructions nor timestamps $@t$;
(2) we do not use global states except for simulating internal communications (see, e.g., [Kremer and Künnemann 2016] for a detailed encoding). We therefore do not use push, pull, and unfound instructions, except to encode instructions of the form

$$\mathsf{out}(e,u);P \triangleq \mathsf{Out}(e) : \mathsf{push}\ u; \mathsf{Ack} : P$$
$$\mathsf{in}(e,x);P \triangleq \mathsf{In}(e) : \mathsf{pull}\ x; P$$

where $e \in \mathcal{N}_{priv}$ is called a *private channel* (and should appear in any output messages), and $\mathsf{Out}, \mathsf{In} \in \mathcal{F}_e$. This models the sending of a message u on a private communication channel e, synchronously received as a variable x, only after what the outputting process may proceed (event Ack). Synchronicity is in particular modelled by the following formula:

$$\varphi_{\mathsf{sync}} = \mathsf{G}\ (\forall e.\mathsf{Out}(e) \Rightarrow \tau_\pi\ \mathsf{U}\ \mathsf{In}(e)_\pi)$$

for some path variable π (always clear from context in the tidy LTL fragment). In particular, in reference to naming conventions for internal communications [Barthe et al. 2022; Blanchet et al. 2020; Cheval et al. 2018], we call "Rule (Comm)" the execution of a $\mathsf{Out}(e) : \mathsf{push}\ u$, followed by a $\mathsf{In}(e) : \mathsf{pull}\ x$ and see it, for simplicity, as a single transition step;
(3) we do not make use of atomic composition, that is, all non-skip instructions are always followed by a semi-column (i.e., an implicit skip), except in the above encoding.

Reduction. Intuitively, two-counter machines consist of:

- two *counters* that can be seen as two (unbounded) registers, by convention initialised to 0, and that may contain arbitrary natural numbers;
- a finite set of *labelled instructions*, that may increment or decrement counters, test whether its content is zero, and/or jump to the label of a subsequent instruction.

We use the following formalisation of two-counter machines.

Definition C.1 (Two-counter machines)**.** A two-counter machine $M = (L, I, \ell_0)$ consists of a finite set of so-called *labels* L including the *initial label* $\ell_0 \in L$, and a finite set of *instructions* I that may be of either of the following three forms:

$$\mathsf{halt} \qquad \mathsf{incr}(c, \ell) \qquad \mathsf{zdecr}(c, \ell^0, \ell^-)$$

where $\ell, \ell^0, \ell^- \in L$ and c is either of the two so-called *registers* $\mathsf{c}_1, \mathsf{c}_2$. We assume an implicit bijective mapping between L and I, and write $M[\ell]$ to refer to the instruction mapped to the label ℓ.

Intuitively, halt is a terminated program and $\mathsf{incr}(c, \ell)$ increments the counter c and then executes $M[\ell]$. The instruction $\mathsf{zdecr}(c, \ell^0, \ell^-)$ is a compact construction that tests whether the content of the counter c is zero (in which case $M[\ell^0]$ is executed), or not (in which case the counter c is decremented, and $M[\ell^-]$ is executed). This is formalised by the following notion of execution:

Definition C.2 (Termination of a two-counter machine)**.** A *state* of a two-counter machine $M = (L, I, \ell_0)$ is a tuple $(n_1, n_2, \ell) \in \mathbb{N} \times \mathbb{N} \times L$. We then define a transition relation $\rightarrow_M$ over states of M as the smallest binary relation such that for all $\ell \in L$ and $n_1, n_2 \in \mathbb{N}$:

- if $M[\ell] = \mathsf{incr}(\mathsf{c}_1, \ell')$ then $(n_1, n_2, \ell) \rightarrow_M (n_1 + 1, n_2, \ell')$;
- if $M[\ell] = \mathsf{incr}(\mathsf{c}_2, \ell')$ then $(n_1, n_2, \ell) \rightarrow_M (n_1, n_2 + 1, \ell')$;
- if $M[\ell] = \mathsf{zdecr}(\mathsf{c}_1, \ell^0, \ell^-)$ then
  - $(0, n_2, \ell) \rightarrow_M (0, n_2, \ell^0)$;
  - $(n_1 + 1, n_2, \ell) \rightarrow_M (n_1, n_2, \ell^-)$;
- if $M[\ell] = \mathsf{zdecr}(\mathsf{c}_2, \ell^0, \ell^-)$ then
  - $(n_1, 0, \ell) \rightarrow_M (n_1, 0, \ell^0)$;
  - $(n_1, n_2 + 1, \ell) \rightarrow_M (n_1, n_2, \ell^-)$;

We say that M *terminates* if there exists a sequence of $\rightarrow_M$-transitions from $(0, 0, \ell_0)$ to a state of the form (n_1, n_2, ℓ), $M[\ell] = \mathsf{halt}$.

The undecidability of the termination of two-counter machines easily follows from the Turing-completeness of this computation model. We will then prove the Proposition 4.1 by constructing, given a two-counter machine M, a process P_M and a tidy LTL formula φ such that $P_M \models \varphi$ *iff* M does not terminate. Let thus $M = (L, I, \ell_0)$ be an arbitrary two-counter machine. We recall that the term algebra contains a unary symbol h. For readability, we will write multiple applications of h to a term $\mathsf{h}^n(u)$, that is, $\mathsf{h}^0(u) = u$ and $\mathsf{h}^{n+1}(u) = \mathsf{h}(\mathsf{h}^n(u))$. Intuitively, we will encode each counter c_i by two terms of the form, respectively, h^n and h^p, $n \geqslant p$, the value of the counter being the difference $n - p$. Therefore, applying h to the first term increments the counter, applying h to the second term decrements the counter, and testing the equality of the two terms tests whether the counter is null. The four terms (two for each of the two counters) are then carried over the execution of the program using private channels ℓ corresponding to the labels of L.

Formally, for each label $\ell \in L$, we associate a fresh name $c_\ell \in \mathcal{N}_{priv}$. We also construct, for each $\ell \in L$, a process (using the previous syntactic sugar for internal communications)

$$P_\ell = \mathsf{in}(c_\ell, x_1^+); \mathsf{in}(c_\ell, x_1^-); \mathsf{in}(c_\ell, x_2^+); \mathsf{in}(c_\ell, x_2^-); Q_\ell$$

where the process Q_ℓ is defined in Figure 13, assuming three events $\mathsf{Zero}/2, \mathsf{Pos}/2, \mathsf{Halt}/0$.

We also define an initialisation process, given a $z \in \mathcal{F}_0$:

$$P_0 = \mathsf{out}(c_{\ell_0}, z); \mathsf{out}(c_{\ell_0}, z); \mathsf{out}(c_{\ell_0}, z); \mathsf{out}(c_{\ell_0}, z); 0$$

The overall process is then:

$$P_M = P_0 \mid \prod_{\ell \in L} !\, P_\ell$$

$$
\begin{aligned}
&\text{If } M[\ell] = \mathsf{halt}: && Q_\ell = \mathsf{event}\ \mathsf{Halt}; 0\\
&\text{If } M[\ell] = \mathsf{incr}(c_1, \ell'): && Q_\ell = \mathsf{out}(c_{\ell'}, \mathsf{h}(x_1^+)); \mathsf{out}(c_{\ell'}, x_1^-); \mathsf{out}(c_{\ell'}, x_2^+); \mathsf{out}(c_{\ell'}, x_2^-); 0\\
&\text{If } M[\ell] = \mathsf{incr}(c_2, \ell'): && Q_\ell = \mathsf{out}(c_{\ell'}, x_1^+); \mathsf{out}(c_{\ell'}, x_1^-); \mathsf{out}(c_{\ell'}, \mathsf{h}(x_2^+)); \mathsf{out}(c_{\ell'}, x_2^-); 0\\
&\text{If } M[\ell] = \mathsf{zdecr}(c_1, \ell^0, \ell^-): && Q_\ell = Q_\ell^0 + Q_\ell^-\\
&&& \text{with } Q_\ell^0 = \mathsf{event}\ \mathsf{Zero}(x_1^+, x_1^-);\\
&&&\qquad \mathsf{out}(c_{\ell^0}, x_1^+); \mathsf{out}(c_{\ell^0}, x_1^-); \mathsf{out}(c_{\ell^0}, x_2^+); \mathsf{out}(c_{\ell^0}, x_2^-); 0\\
&&& \text{and } Q_\ell^- = \mathsf{event}\ \mathsf{Pos}(x_1^+, x_1^-);\\
&&&\qquad \mathsf{out}(c_{\ell^-}, x_1^+); \mathsf{out}(c_{\ell^-}, \mathsf{h}(x_1^-)); \mathsf{out}(c_{\ell^-}, x_2^+); \mathsf{out}(c_{\ell^-}, x_2^-); 0\\
&\text{If } M[\ell] = \mathsf{zdecr}(c_2, \ell^0, \ell^-): && Q_\ell = Q_\ell^0 + Q_\ell^-\\
&&& \text{with } Q_\ell^0 = \mathsf{event}\ \mathsf{Zero}(x_2^+, x_2^-);\\
&&&\qquad \mathsf{out}(c_{\ell^0}, x_1^+); \mathsf{out}(c_{\ell^0}, x_1^-); \mathsf{out}(c_{\ell^0}, x_2^+); \mathsf{out}(c_{\ell^0}, x_2^-); 0\\
&&& \text{and } Q_\ell^- = \mathsf{event}\ \mathsf{Pos}(x_2^+, x_2^-);\\
&&&\qquad \mathsf{out}(c_{\ell^-}, x_1^+); \mathsf{out}(c_{\ell^-}, x_1^-); \mathsf{out}(c_{\ell^-}, x_2^+); \mathsf{out}(c_{\ell^-}, \mathsf{h}(x_2^-)); 0
\end{aligned}
$$

Fig. 13. Encoding of two-counter-machine operations as processes

Let us then define the tidy LTL formula φ that formalises several properties. First, it expresses the assumptions behind the events Zero and Pos, namely that $\mathsf{Zero}(u, v)$ should only be triggered when the terms u and v are equal (modulo theory), and conversely that $\mathsf{Pos}(u, v)$ can only be triggered when u and v are different. This way, the non-deterministic choices in the definition of Q_ℓ (cases zdecr) effectively emulate a conditional testing. Additionally, φ requires that the halting event never holds, thus encoding non-termination. Formally, $\varphi = \forall \pi.\ \varphi_{\mathsf{sync}} \Rightarrow \varphi_{\mathsf{Zero}} \Rightarrow \varphi_{\mathsf{Pos}} \Rightarrow \mathsf{G}\ \neg \mathsf{Halt}_\pi$ with:

$$
\begin{aligned}
\varphi_{\mathsf{Zero}} &= \forall x, y.\ \mathsf{Zero}(x, y)_\pi \Rightarrow x = y\\
\varphi_{\mathsf{Pos}} &= \forall x, y.\ \mathsf{Pos}(x, y)_\pi \Rightarrow x \neq y
\end{aligned}
$$

The correctness of the overall reduction is then stated by:

Proposition C.1 (Correctness of the reduction)**.** *M does not terminate iff $P_M \vDash \varphi$.*

C.2 Proof

The rest of this section is dedicated to the proof of Proposition C.1, which will conclude the proof of Proposition 4.1. For that we establish a correspondence between two-counter machine executions and a restricted type of trace that we call *normal*.

Definition C.3 (Normal process)**.** We call a extended process *normal* when it is of the form $(\mathcal{P}, \varnothing)$, with

$$
\begin{aligned}
\mathcal{P} &= \{\!\!\{ Q_\ell \sigma \}\!\!\} \cup \mathcal{P}_0 \cup \{\!\!\{ !\, P_{\ell'} \}\!\!\}_{\ell' \in L} && \text{(regular normal proc.)}\\
\text{or } \mathcal{P} &= \mathcal{P}_0 \cup \{\!\!\{ !\, P_{\ell'} \}\!\!\}_{\ell' \in L} && \text{(blank normal proc.)}
\end{aligned}
$$

- P_ℓ, Q_ℓ still refer to the previously defined processes;
- $\mathcal{P}_0$ only contain processes that are 0 or $P_{\ell'}$, $\ell' \in L$, but at least one copy of each process $P_{\ell'}$, $\ell' \in L$;
- σ is a substitution of domain $\{x_1^+, x_1^-, x_2^+, x_2^-\}$ and whose image only consists of terms of the form $\mathsf{h}^m(z)$.

Definition C.4 (Normal trace)**.** If A is a (normal) extended process, a trace of A is *normal* if it is of the form $T_1 \cdots T_n$ for traces T_i, whose first and last extended processes are normal, and of the form

$T_i = T_i^1 \cdot T_i^2$ where T_i^1, T_i^2 are non-empty, T_i^1 contains no (Repl) transitions, and T_i^2 contains only (Repl) transitions. Note that this decomposition, when it exists, is unique.

The first step of the proof is to reduce the analysis to normal traces, which is done by the following result. We write ψ the formula such that $\varphi = \forall \pi.\, \psi$, that is,

$$\psi = \varphi_{\mathsf{sync}} \Rightarrow \varphi_{\mathsf{Zero}} \Rightarrow \varphi_{\mathsf{Pos}} \Rightarrow \mathsf{G}\ \neg \mathsf{Halt}_\pi$$

In the following, we also let $\mathcal{P}_M = (\mathcal{P}, \varnothing)$ be a normal extended process of the form:

$$\mathcal{P} = \{\!\{ Q_{\ell_0} \sigma_0 \}\!\} \cup \mathcal{P}_0 \cup \{\!\{ !\, P_{\ell'} \}\!\}_{\ell' \in L}$$

with the notations of Definition C.3, and $x\sigma_0 = z$ for all $x \in \{x_1^+, x_1^-, x_2^+, x_2^-\}$.

Lemma C.2 (Reduction to normal traces). *$P_M \models \varphi$ iff for all normal traces T of $\mathcal{P}_M$, $\Pi_T \models \psi$ with $\Pi_T = \{\varepsilon \mapsto T, \pi \mapsto T\}$.*

Proof. First of all, note that all processes of the form $\mathcal{P}_M$ are obtained from P_M by applying, up to reordering, Rule (Par) as many times as possible, Rule (Repl) at least $|L| + 1$ times, and Rule (Comm) four times between P_0 and a copy of P_{ℓ_0}. Conversely, all processes obtained after applying such transitions from P_M in any order are of the same form as $\mathcal{P}_M$, up to the addition of enough (Repl) transitions (and adding such transitions does not prevent subsequent independent transitions from being executed). We hence obtain that $P_M \models \varphi$ *iff* for all traces T of $\mathcal{P}_M$, $\Pi_T \models \psi$. Without loss of generality, we only consider traces T that does not start with a (Repl) transition (as initial (Repl) transitions only yield normal extended processes of the same form as $\mathcal{P}_M$).

Our goal is to prove that it is sufficient to limit this quantification to normal traces $\mathcal{P}_M$. For that, given an arbitrary normal extended process A_0, let us assume par contraposition that there exists a trace T, non-necessarily normal but not starting with a (Repl) transition, written

$$T : A_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} A_n$$

such that $\Pi_T \models \varphi_{\mathsf{sync}} \wedge \varphi_{\mathsf{Zero}} \wedge \varphi_{\mathsf{Pos}} \wedge \mathsf{F}\ \mathsf{Halt}_\pi$. We then have to show that there exists a normal trace T_N of A_0 (also not starting with a (Repl) transition) such that

$$\Pi_{T_N} \models \varphi_{\mathsf{sync}} \wedge \varphi_{\mathsf{Zero}} \wedge \varphi_{\mathsf{Pos}} \wedge \mathsf{F}\ \mathsf{Halt}_\pi\,.$$

We prove the result by well-founded induction on $n - i \geqslant 0$, where n is the length of the trace, and i is the maximal index such that $A_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_i} A_i$ that is a normal trace. If $i = n$, the conclusion follows by choosing $T_N = T$. If $i < n$, let us write the normal process $A_i = (\mathcal{P}, \varnothing)$ with the notations of Definition C.3, in particular, $\mathcal{P} = \mathcal{S} \cup \mathcal{P}_0 \cup \{\!\{ !\, P_{\ell'} \}\!\}_{\ell' \in L}$ for $\mathcal{S} = \{\!\{ Q_\ell \sigma \}\!\}$ or $\mathcal{S} = \varnothing$.

First of all, let us exclude the case where the next transition $A_i \xrightarrow{\alpha_{i+1}} A_{i+1}$ is derived by Rule (Repl). Indeed in this case, either $i = 0$ and this is in contradiction with the assumption that T does not start with a (Repl) transition, or $i > 0$ and we write the decomposition

$$(A_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_i} A_i) = T_1 \cdots T_k$$

obtained by Definition C.4, then

$$T_1 \cdots T_{k-1} \cdot T_k' \qquad \text{with } T_k' = T_k \cdot (A_i \xrightarrow{\alpha_{i+1}} A_{i+1})$$

would be a decomposition of $A_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{i+1}} A_{i+1}$ proving that it is also a normal trace, which is in contradiction with the maximality of i. Let us thus assume that this transition is derived by any other rule than (Repl). Additionally, we also emphasise that no event Halt could have been executed during the normal trace $A_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_i} A_i$. Otherwise, due to the form of normal traces,

no further $\xrightarrow{\alpha}$ transitions should have been possible after the execution of this event, except by Rule (Repl)—case excluded above.

Note that this also excludes the case $\mathcal{S} = \varnothing$ (blank normal process), since only (Repl) transitions are possible from such a process. Therefore $\mathcal{S} = \{\!\{Q_\ell\sigma\}\!\}$ (regular normal process), and we perform a case analysis on $M[\ell]$.

▸ *Case 1:* $M[\ell] = \mathsf{halt}$.

Since the transition $A_i \xrightarrow{\alpha_{i+1}} A_{i+1}$ is not derived by Rule (Repl), and due to the form of normal extended processes, this transition must be the instance of Rule (Event) executing the event Halt of Q_ℓ. In particular, A_{i+1} would be a (blank) normal extended process, in contradiction with the maximality of i.

▸ *Case 2:* $M[\ell] = \mathsf{zdecr}(\mathsf{c}_1, \ell^0, \ell^-)$.

Since $A_i \xrightarrow{\alpha_{i+1}} A_{i+1}$ is not derived by Rule (Repl), and due to the form of normal extended processes, this transition must be the instance of Rule (Choice) executing either event at toplevel of $Q_\ell\sigma$ ($\mathsf{Zero}(x_1^+\sigma, x_1^-\sigma)$ or $\mathsf{Pos}(x_1^+\sigma, x_1^-\sigma)$). Let us assume that the event Zero is executed by this transition (the argument in the case of Pos is analogue). In this case we have $A_{i+1} = (\mathcal{Q}, \varnothing)$, with

$$\mathcal{Q} = \{\!\{Q\sigma\}\!\} \cup \mathcal{P}_0 \cup \{\!\{!\, P_{\ell'}\}\!\}_{\ell' \in L}$$

where Q is defined by $Q_\ell^0 = \text{event } \mathsf{Zero}(x_1^+, x_1^-); Q$. In particular, $Q\sigma$ has four outputs at toplevel (on the private channel c_{ℓ^0}), while all other processes of $\mathcal{Q}$ are either 0, or have either four inputs or a replication followed by four inputs, all on private channels. In particular, in the trace $A_i \xrightarrow{\alpha_{i+1}} \cdots \xrightarrow{\alpha_n} A_n$, if the outputs of $Q\sigma$ are either not all used in internal communications, or used in internal communications with inputs of two different processes of $\mathcal{Q}$, it is straightforward to see that this trace may only contain up to three instances of Rule (Comm) and an arbitrary number of Rule (Repl) (in particular, no Halt event). This would be in contradiction with the hypothesis that $\Pi_T \vDash \mathsf{F}\,\mathsf{Halt}_\pi$ since, as discussed before case 1, the Halt event cannot occur in the prefix trace $A_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_i} A_i$. But in addition, we already excluded the case where the transition $A_i \xrightarrow{\alpha_{i+1}} A_{i+1}$ is derived by Rule (Repl), meaning that it must be an internal communication, necessarily between $Q\sigma$ and a copy of $P_{\ell^0} \in \mathcal{P}_0$. As a conclusion, we obtain that there exist four internal communications between P_ℓ and $Q\sigma$ in the trace $A_i \xrightarrow{\alpha_{i+1}} \cdots \xrightarrow{\alpha_n} A_n$.

Therefore, we can write $A'_i \xrightarrow{\beta_{i+1}} \cdots \xrightarrow{\beta_{n+|L|}} A'_{n+|L|}$ the trace obtained from $A_i \xrightarrow{\alpha_{i+1}} \cdots \xrightarrow{\alpha_n} A_n$ by:

(1) moving the four internal communications between P_ℓ and $Q\sigma$ in front, i.e., as the first four actions;
(2) adding, as the 5^{th} to $5 + |L|^{\text{th}}$ actions, instances of Rule (Repl) unfolding one copy of each P_ℓ, $\ell \in L$;
(3) from the $6 + |L|^{\text{th}}$ action onwards, leave all remaining transitions of $A_i \xrightarrow{\alpha_{i+1}} \cdots \xrightarrow{\alpha_n} A_n$ in the same order.

Note that the permutation done in the first step is possible due to the form of the processes (see discussion right above Case 2.1). Overall, we writing T' the reordered trace

$$T' : A_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_i} (A_i = A'_i) \xrightarrow{\beta_{i+1}} \cdots \xrightarrow{\beta_{n+|L|}} A'_{n+|L|}$$

we have $\Pi_{T'} \models \varphi_{\mathsf{sync}} \wedge \varphi_{\mathsf{Zero}} \wedge \varphi_{\mathsf{Pos}} \wedge \mathsf{F}\,\mathsf{Halt}_\pi$ since this property is preserved by the way the permutation $\beta_{i+1} \cdots \beta_{n+|L|}$ of $\alpha_{i+1} \cdots \alpha_n$ is constructed. Additionally, the well-founded measure decreases by at least 4 for T' compared to T. The conclusion thus easily follows from the induction hypothesis applied to T'.

▸ *Case 3: any other possibility for* $M[\ell]$.

We omit the proofs for all remaining cases, as they are either analogue to case 2 (decrement of c_2), or only require a similar but much simpler argument (increment). □

Building on this characterisation of φ restricted to normal traces, we prove the proposition by double implication. For that we introduce a natural mapping between regular normal extended processes and two-counter-machine states. With the notations of Definition C.3, if $A = (\mathcal{P}, \varnothing)$ is a regular normal extended process with

$$\mathcal{P} = \{\!\{Q_\ell \sigma\}\!\} \cup \mathcal{P}_0 \cup \{\!\{!\, P_{\ell'}\}\!\}_{\ell' \in L}$$

then we associate to it the tuple

$$[\![A]\!] = (n_1^+ - n_1^-, n_2^+ - n_2^-, \ell) \qquad \text{with } x_i^s \sigma = \mathsf{h}^{n_i^s}(z)$$

Note that such a tuple is not necessarily a well-formed two-counter machine state (it is one *iff* $n_1^+ \geqslant n_1^-$ and $n_2^+ \geqslant n_2^-$).

Lemma C.3 (Proposition C.1, forward direction). *If $P_M \models \exists \pi. \varphi_{\mathsf{Zero}} \wedge \varphi_{\mathsf{Pos}} \wedge \mathsf{F}\ \mathsf{Halt}_\pi$ then M terminates.*

Proof. Let us assume that there exists a trace T of P_M such that $\Pi_T \models \varphi_{\mathsf{Zero}} \wedge \varphi_{\mathsf{Pos}} \wedge \mathsf{F}\ \mathsf{Halt}_\pi$. By lemma C.2, we can assume that T is a normal trace without loss of generality. Let us thus consider the decomposition $T = T_1 \cdots T_n$ with the notations of Definition C.4, with in particular A_{i-1} the first extended process of T_i^1. We then prove that for all $i \in [\![0, n-1]\!]$, $[\![A_i]\!]$ is a (well-defined) two-counter machine state that is $\rightarrow_M$-reachable from $(0, 0, \ell_0)$. This will conclude the proof since $[\![A_{n-1}]\!]$ is a halting configuration.

The conclusion is immediate in the case $i = 0$, since $[\![A_0]\!] = (0, 0, \ell_0)$. In the case $i > 0$, we know by induction hypothesis that $[\![A_{i-1}]\!]$ is $\rightarrow_M$-reachable from $(0, 0, \ell_0)$, and it therefore suffices to prove that $[\![A_{i-1}]\!] \rightarrow [\![A_i]\!]$. Since A_{i-1} is a by definition a (regular) normal extended process, we can write $A_{i-1} = (\mathcal{P}, \varnothing)$ with

$$\mathcal{P} = \{\!\{Q_\ell \sigma\}\!\} \cup \mathcal{P}_0 \cup \{\!\{!\, P_{\ell'}\}\!\}_{\ell' \in L}$$

with the notations of Definition C.3. The conclusion then follows from a quick case analysis on the instruction $M[\ell]$ (using in particular that $\Pi_T \models \varphi_{\mathsf{Zero}} \wedge \varphi_{\mathsf{Pos}}$). □

Lemma C.4 (Proposition C.1, converse direction). *If M terminates then $P_M \models \exists \pi. \varphi_{\mathsf{Zero}} \wedge \varphi_{\mathsf{Pos}} \wedge \mathsf{F}\ \mathsf{Halt}_\pi$.*

Proof. Let us consider a halting run of M, that is, a sequence $s_0, \ldots, s_{n-1}$ of states of M such that $s_0 = (0, 0, \ell_0)$ and s_{n-1} is a halting state. It is then straightforward to construct, by induction on i, a (normal) trace $T = T_1 \cdots T_n$ (notations of Definition C.4) such that for all $i \in [\![0, n-1]\!]$, $\Pi_{T_1 \cdots T_{i+1}} \models \varphi_{\mathsf{Zero}} \wedge \varphi_{\mathsf{Pos}}$ and, if A_i is the first (normal) extended process of T_{i+1}, A_i is regular and $[\![A_i]\!] = s_0$. In particular, we also have $\Pi_T \models \mathsf{F}\ \mathsf{Halt}_\pi$ since $[\![A_{n-1}]\!] = s_{n-1}$ is a halting state and, as such, the first instruction of T_n need be the execution of a Halt event. □

D SYMBOLIC CONSTRAINT SOLVING

As a first point of this section, we refer to the full symbolic semantics, which can be found in Figure 14. We then define in this section the data structures at the core of our decision procedure, uplifting the notion of constraint systems to a stack of *proof states*. It intuitively represents a state of a hyperproperty's proof in presence of several nested path quantifications. We also then present various *constraint solving* algorithm for computing (most general) solutions and a saturated representation of the attacker's knowledge.

▸ ***In all rules,*** $\mu = mgu(\mathsf{E}^1(C)) \neq \bot$.

symbolic atomic semantics:

$$(\mathsf{out}(u) : S, C, \theta, t) \xrightarrow{\mathsf{out}}_{\text{s-at}} (S, C \wedge \sigma \wedge N \wedge \mathsf{ax} \vdash^? u\sigma\downarrow, \theta, t) \quad \text{(s-Out)}$$
$$\text{if } \sigma \wedge N \in mgu_{\mathcal{R}}(u\mu =^? u\mu) \text{ and } \mathsf{ax} \in \mathcal{A}\mathcal{X} \text{ is fresh}$$

$$(\mathsf{in}(x) : S, C, \theta, t) \xrightarrow{\mathsf{in}}_{\text{s-at}} (S\{x \mapsto y\}, C \wedge Y \vdash^? y, \theta, t) \quad \text{(s-In)}$$
$$\text{if } y \in \mathcal{X}^1 \text{ and } Y\text{: } dom(\Phi(C)) \in \mathcal{X}^2 \text{ are fresh}$$

$$(\mathsf{new}\ k : S, C, \theta, t) \xrightarrow{\mathsf{new}}_{\text{s-at}} (S\{k \mapsto k'\}, C, \theta, t) \quad \text{(s-New)}$$
$$\text{with } k' \in \mathcal{N}_{priv} \text{ fresh}$$

$$(\mathsf{Ev}(\vec{u}) : S, C, \theta, t) \xrightarrow{\mathsf{Ev}(\vec{u}\sigma\downarrow)}_{\text{s-at}} (S, C \wedge \sigma \wedge N, \theta, t) \quad \text{(s-Event)}$$
$$\text{if } \sigma \wedge N \in mgu_{\mathcal{R}}(\vec{u}\mu =^? \vec{u}\mu)$$

$$(\mathsf{push}\ u : S, C, \theta, t) \xrightarrow{\mathsf{push}}_{\text{at}} (S, C \wedge \sigma \wedge N, \theta \cup \{\!\{u\sigma\downarrow\}\!\}, t) \quad \text{(s-Push)}$$
$$\text{if } \sigma \wedge N \in mgu_{\mathcal{R}}(u\mu =^? u\mu)$$

$$(\mathsf{pull}\ u : S, C, \theta \cup \{\!\{v\}\!\}, t) \xrightarrow{\mathsf{pull}}_{\text{at}} (S\sigma, C \wedge \sigma \wedge N, \theta, t) \quad \text{(s-Pull)}$$
$$\text{if } \sigma \wedge N \in mgu_{\mathcal{R}}(u\mu =^? v\mu)$$

$$(\mathsf{unfound}\ u : S, C, \theta, t) \xrightarrow{\mathsf{unfound}}_{\text{at}} (S, C \wedge \textstyle\bigwedge_{(\sigma,N)\in M} \neg\sigma \vee \neg N, \theta, t) \quad \text{(s-Unfound)}$$
$$\text{with } M = \textstyle\bigcup_{v\in\theta} mgu_{\mathcal{R}}(u\mu =^? v\mu)$$

$$(@t_0 : S, C, \theta, t) \xrightarrow{\mathsf{stamp}}_{\text{s-at}} (S\{t_0 \mapsto t\}, C, \theta, t) \quad \text{(s-Stamp)}$$

$$(\mathsf{when}\ e \sim 0 : S, C, \theta, t) \xrightarrow{\mathsf{when}}_{\text{s-at}} (S, C \wedge e \sim 0, \theta, t) \quad \text{(s-When)}$$

$$(S_0 + S_1, C, \theta, t) \rightarrow_{\text{s-at}} (S_i, C, \theta, t) \quad \text{if } i \in \{0, 1\} \quad \text{(s-Choice)}$$

symbolic sequential semantics:

$$(\{\!\{\mathsf{skip}\ ; P\}\!\}, C, \theta, t) \xrightarrow{\mathsf{skip}}_{\text{s-seq}} (\{\!\{P\}\!\}, C, \theta, t) \quad \text{(s-Skip)}$$

$$(\{\!\{P \mid Q\}\!\}, C, \theta, t) \xrightarrow{\mathsf{par}}_{\text{s-seq}} (\{\!\{P, Q\}\!\}, C, \theta, t) \quad \text{(s-Par)}$$

$$(\{\!\{!^n P\}\!\}, C, \theta, t) \xrightarrow{\mathsf{repl}}_{\text{seq}} (\{\!\{!^{n-1} P, P\}\!\}, C, \theta, t) \quad \text{(s-Repl)}$$

$$(\{\!\{P\}\!\}, C, \theta, t) \xrightarrow{\alpha,\vec{w}}_{\text{s-seq}} (\mathcal{P}, C'' \wedge \Phi(C'), \theta'', t) \quad \text{(s-Comp)}$$
$$\text{if } (P, C, \theta, t) \xrightarrow{\alpha}_{\text{s-at}} (P', C \wedge C', \theta', t),$$
$$\text{and } (\{\!\{P'\}\!\}, C \wedge (C' \setminus \Phi(C')), \theta', t) \xrightarrow{\vec{w}}_{\text{s-seq}} (\mathcal{P}, C'', \theta'', t)$$

symbolic semantics:

$$(\{\!\{P\}\!\} \cup \mathcal{Q}, C, \theta, t) \xrightarrow{\alpha,\vec{w}}_{\text{s}} (\mathcal{P} \cup \mathcal{Q}, C', \theta', t') \quad \text{(s-Lift)}$$
$$\text{with } t' \in \mathcal{X}^{\mathsf{N}} \text{ fresh, and } (\{\!\{P\}\!\}, C \wedge t - t' < 0, \theta, t') \xrightarrow{\vec{w}}_{\text{s-seq}} (\mathcal{P}, C', \theta', t')$$

Fig. 14. Symbolic semantics for verifying timed bounded processes

D.1 Solution Management

We give here some key notions to define the various solving algorithm detailed in the next sections; these notions are adapted from the algorithm of [Cheval et al. 2018] for computing solutions of (plain) constraint systems. First of all, computing the solutions of a hyperconstraint system C is done under the assumption that a *saturation* procedure has been carried out at a previous step, ensuring that a the deducibility of a term u can be checked syntactically from $\mathsf{K}(C)$. Formally:

Definition D.1 (Direct consequence)**.** Given a set of deduction hyperconstraints S, a *direct consequence* of S in path π is a statement $\xi \Vdash_{S,\pi} u$, such that there exists a public constructor context C,

and deduction hyperconstraints $\xi_1 \vdash^?_\pi u_1, \ldots, \xi_n \vdash^?_\pi u_n \in S$ such that

$$\xi = C[\xi_1, \ldots, \xi_n] \qquad \text{and} \qquad u = C[u_1, \ldots, u_n]\,.$$

If C is a hyperconstraint system, we may write $\xi \Vdash_{C,\pi} u$ instead of $\xi \Vdash_{\mathsf{D}(C)\cup\mathsf{K}(C),\pi} u$. We also write

$$\mathsf{conseq}_\pi(C) = \{\xi \mid \xi \Vdash_{C,\pi} u\}\,.$$

Recalling that, by definition of deduction hyperconstraints $\xi \vdash^?_\pi u$, ξ does not have a constructor symbol at its root, the context C of the above definition is uniquely determined. The saturation of a hyperconstraint system C then means that $\Vdash_{C,\pi}$ coincides with deducibility from $\Phi(\mathrm{proj}_\pi(C))$.

The algorithm for solving constraints itself then proceeds by successively computing partial solutions modelling case analyses. For example, consider a hyperconstraint system C and $(X \vdash^?_\pi \mathsf{f}(k)) \in C$, $\mathsf{f} \in \mathcal{F}_\mathsf{c} \cap \mathcal{F}_{pub}$ and $k \in \mathcal{N}_{priv}$. The constraint solver therefore has to compute a recipe $(X\Sigma)$ deducing $\mathsf{f}(k)$ in π (i.e., such that $X\Sigma \Vdash_\pi \mathsf{f}(k)$, assuming C is saturated). The solver will, among others, perform a case analysis on whether the recipe in question has the symbol f at its root or not. For that, it will generate the substitution

$$\Sigma_\mathsf{f} = \{X \mapsto \mathsf{f}(Y)\}$$

with $Y \in \mathcal{X}^2$ fresh and of same multiplicity as X. Indeed, interpreted as a second-order constraint, this results in the following, effectively modelling the expected case analysis:

$$\Sigma_\mathsf{f} = X =^? \mathsf{f}(Y) \qquad\qquad \neg\Sigma_\mathsf{f} = \forall Y.X \neq^? \mathsf{f}(Y)\,.$$

The solver will then attempt to solve two instances of C, one where Σ_f has been applied, and $C \wedge \neg\Sigma_\mathsf{f}$. In the first case, applying Σ_f to C will remove the initial constraint $X \vdash^?_\pi \mathsf{f}(k)$ from C, replace it by a fresh deduction constraints $Y \vdash^?_\pi y$, and add "linking equations" expressing the relations between X, Y, y, namely $X =^? \mathsf{f}(Y)$ and $y =^? k$. We formalise this custom notion of application below.

Definition D.2 (Solution application)**.** Let C be a hyperconstraint system and Σ be a second-order substitution such that for all $X \in \mathit{dom}(\Sigma)$, there exists $(X \vdash^?_\pi u) \in \mathsf{D}(C)$, and also for all such $(X \vdash^?_\pi u) \in \mathsf{D}(C)$, $X\Sigma \in \mathsf{conseq}_\pi(C)$. The hyperconstraint system $C:\Sigma$ is called the *application* of Σ to C, and is defined as follows:

$$\mathsf{D}(C:\Sigma) = \mathsf{D}' \qquad\qquad \mathsf{E}^1(C:\Sigma) = \mathsf{E}^1(C) \wedge E_\Sigma$$
$$\mathsf{K}(C:\Sigma) = \mathsf{K}(C)\Sigma \qquad\qquad \mathsf{E}^2(C:\Sigma) = \mathsf{E}^2(C)\Sigma \wedge \Sigma$$

where we have $\mathsf{D}' = (\mathsf{D}(C) \setminus D_{dom}) \cup D_{\mathsf{fresh}}$ as well as the following sets:

(1) deduction constraints resolved by Σ:

$$D_{dom} = \{(Y \vdash^?_\pi u) \in \mathsf{D} \mid Y \in \mathit{dom}(\Sigma)\}$$

(2) deduction constraints introduced by Σ:

$$D_{\mathsf{fresh}} = \big\{Y \vdash^?_\pi y \mid (X \vdash^?_\pi u) \in D_{dom}, y \text{ fresh}, \\ Y \in \mathit{vars}^2(X\Sigma) \setminus \mathit{vars}^2(C)\big\}$$

(3) linking equations:

$$E_\Sigma = \big\{u =^? v \mid (Y \vdash^?_\pi u) \in D_{dom}, Y\Sigma \Vdash_{C:\Sigma,\pi} v\big\}$$

D.2 Solving Rules

In our procedure, we will then use several sets of constraint-solving procedure:

- *solving rules* that are used to compute most general solutions of a hyperconstraint system, and more generally put them in a simple form and discard trivial and unsatisfiable constraints;
- *case analysis rules*, that build on solving rules to formalise case analyses in proof stacks. They typically test whether some combinations of constraints have a solution or not, and saturate the knowledge base of the adversary.

The overall procedure will then proceed by induction on the formula to be verified, using case analysis rules at each step to introduce all necessary constraints and solve them.

Simplification Rules. We define solving rules in this section. They build on a preliminary set of *simplification rules*, defined in Figure 15.

$$
\begin{array}{ll}
\underline{\textit{Basic simplifications}} & C \wedge \top \rightsquigarrow C \qquad C \wedge \bot \rightsquigarrow \bot \\
\underline{\textit{ded. constraints}} & C \wedge X \vdash^?_\pi t \rightsquigarrow C \qquad \text{if } t \in \mathcal{X}^{\mathsf{N}} \\
& C \wedge (\forall X.X \nvdash^?_\pi u) \rightsquigarrow \bot \\
& \quad \text{if there is a recipe } \xi \text{ such that } \{X \mapsto \xi\} \text{ is well-typed, and } \xi \Vdash_{C,\pi} u \\
\underline{1^{st} \textit{ order equations}} & C \wedge u =^? v \rightsquigarrow C \wedge mgu(u =^? v) \\
& C \wedge x =^? u \rightsquigarrow C\sigma \wedge x =^? u \\
& \quad \text{assuming } \sigma = \{x \mapsto u\} \text{ is well-typed and } x \in vars^1(C) \setminus vars^1(u) \\
\underline{1^{st} \textit{ order disequations}} & C \wedge \forall S.\, \phi \rightsquigarrow \begin{cases} C \wedge \forall S.\, \neg\sigma & \text{with } \sigma = mgu(\neg\phi) \neq \bot \\ C & \text{if } mgu(\neg\phi) = \bot \end{cases} \\
\underline{2^{nd} \textit{ order disequations}} & C \wedge \forall S.\, \phi \rightsquigarrow \begin{cases} C \wedge \forall S \cup S'.\neg\Sigma & \text{with } \Sigma = mgu(\neg\phi) \neq \bot \\ & \text{and } S' = vars^2(img(\Sigma)) \setminus vars^2(\phi) \\ C & \text{if } mgu(\neg\phi) = \bot \end{cases} \\
\underline{\textit{Time constraints}} & C \wedge C_t \rightsquigarrow \bot \\
& \quad \text{if } C_t \text{ is a conjunction of numeric constraints with no } \sigma \text{ such that } C_t\sigma \text{ holds}
\end{array}
$$

Fig. 15. Simplification rules for hyperconstraint systems

These basic set of rules are rather standard, albeit from the one for numeric constraints (which removes deduction constraints involving a numeric variables, as they are always implicitly quantified in the system). Otherwise, the rules simply put (dis)equational constraints into a simple form by computing unifiers. Note that no rules are needed for second-order equations, as they will be specifically handled by the parts of the procedure computing solutions. The simplification rules are then lifted to hyperconstraint systems C in the natural way, by applying the rules to all compatible hyperconstraints of C.

Mgs Rules. Building on this set of simplification rules, we define another set of rules used to compute a complete set of most general solutions of a hyperconstraint system C. The overall procedure will typically alternate between such mgs computations (which assume that C has been saturated up to a certain point), and the case analysis rules (using the mgs to perform the said

saturation, among others). The goal of the mgs rules is to resolve all deduction constraints $X \vdash^?_\pi u$, i.e., to find a recipe ξ deducing the constructor term u (with the correct multiplicity). Assuming saturation, we only consider direct deduction instead of arbitrary deductions; there are therefore only two possible cases:

(1) ξ has a constructor symbol at its root, i.e., is of the form $\xi = \mathsf{f}(\xi_1, \ldots, \xi_n)$ for some $\mathsf{f}/n \in \mathcal{F}_{pub} \cap \mathcal{F}_\mathrm{c}$; or
(2) ξ is an entry of the knowledge base, that is, $\xi \vdash^?_\pi u \in C$.

Each of these cases will be formalised by a rule of the form:

$$C \xrightarrow[\mathsf{mgs}]{\Sigma} C : \Sigma \tag{$\star$}$$

for some adequate second-order substitution Σ modelling the case in question (with some conditions on C). The first rule for example models the constructor case by using ($\star$) with:

$$\Sigma = \{X \mapsto \mathsf{f}(X_1, \ldots, X_n)\} \tag{Mgs-Cons}$$

if there exists a deduction hyperconstraint $X : M \vdash^?_\pi u \in \mathsf{D}(C)$, $u \notin \mathcal{X}^1$, and with $\mathsf{f}/n \in \mathcal{F}_\mathrm{c} \cap \mathcal{F}_{pub}$, and $X_i : M$ fresh. Then, the case where the deduction is a direct entry of the knowledge base corresponds to the instantiation of ($\star$) with:

$$\Sigma = mgu(X =^? \xi) \neq \bot \tag{Mgs-Res}$$

if there exists two deduction hyperconstraints $X \vdash^?_\pi u \in \mathsf{D}(C)$, $u \notin \mathcal{X}^1$, and $\xi \vdash^?_\pi v \in \mathsf{K}(C)$. The set of all most general unifiers of C is then obtained by considering all hyperconstraint systems reachable using these two rules, each application of which being separated by as many simplification rules as possible. Formally:

Definition D.3 (Solved form)**.** We say that a hyperconstraint system C is *solved* when it cannot be reduced by $\rightsquigarrow$ or $\xrightarrow[\mathsf{mgs}]{}$, and when for all $X \vdash^?_\pi u \in \mathsf{D}(C)$, $u \in \mathcal{X}^1$.

Definition D.4 (Direct solution)**.** We say that a solution $(\Sigma, \sigma) \in Sol(C)$ is a *direct solution* of C if for all $X \vdash^?_\pi u \in \mathsf{D}(C)$, $X\Sigma \in \mathsf{conseq}_\pi(C)$. A set of *most general direct solutions* of C is defined analogously to the set of most general solutions of C, replacing solutions by direct solutions.

In the following proposition, we write $\rightsquigarrow\!\!\!\!\!\!\downarrow$ the relation applying as many $\rightsquigarrow$ steps as possible (which is terminating). The proposition can be proved by a straightforward induction on the size of the first-order terms appearing in deduction hyperconstraints of C.

Proposition D.1 (Correctness of mgs computations)**.** *Let C be a hyperconstraint system. If we assume that $mgu(\mathsf{E}^2(C)) \neq \bot$, writing S the set of most general direct solutions of C:*

(1) If C is irreducible by $\rightsquigarrow$ and $\xrightarrow[\mathsf{mgs}]{}$, then $S = \{mgu(\mathsf{E}^2(C))\}$ if C is solved, and $S = \varnothing$ otherwise;
(2) otherwise:

$$\{(\Sigma_1 \cdots \Sigma_n)_{|vars^2(C)} \mid C \rightsquigarrow\!\!\!\!\!\!\downarrow \xrightarrow[\mathsf{mgs}]{\Sigma_1} \rightsquigarrow\!\!\!\!\!\!\downarrow \cdots \xrightarrow[\mathsf{mgs}]{\Sigma_n} \rightsquigarrow\!\!\!\!\!\!\downarrow C', C' \text{ solved}\}\,.$$

Solving Rules. Finally, relying on the ability to compute most general solutions described in the previous paragraphs, we can define the set of *solving rules* $\overset{\mathsf{solve}}{\rightsquigarrow}$, that are used to solve hyperconstraints in general. They are defined below, and enhance the previous rules by adding ways to remove some unsatisfiable constraints.

$$C \overset{\text{solve}}{\rightsquigarrow} C' \quad \text{if } C \rightsquigarrow C' \tag{Solve-Simpl}$$

$$C \overset{\text{solve}}{\rightsquigarrow} \bot \quad \text{if } mgs(C) = \bot \tag{Solve-Unsat}$$

$$C \wedge \forall \vec{x}.\phi \overset{\text{solve}}{\rightsquigarrow} C \quad \text{if } mgs(C \wedge \neg\phi) = \bot \tag{Solve-Diseq}$$

D.3 Case Analysis Rules

Resolution. The next rule applies uses most general solutions to perform case analyses in a proof stack. Given a state $\omega^\star = \omega_1 \rightarrowtail \cdots \rightarrowtail \omega_n$, the algorithm will first attempt to compute the solutions Σ of ω_n. Then, in case Σ imposes additional conditions on previous states ω_i, $i < n$, a case analysis will occur, that is, $\neg\Sigma$ will be added as a proof obligation as well at different levels. Formally, the rule takes the following form:

$$(\omega_1 \rightarrowtail \cdots \rightarrowtail \omega_n) \xrightarrow[\text{case}]{} (\omega_1' \rightarrowtail \cdots \rightarrowtail \omega_n') \tag{Case-Resolve}$$

where the following conditions are met. First of all, either $\Sigma \in mgs(C(\omega_n))$ or $\Sigma \in mgs(C' \wedge mgu(\neg\phi))$, writing $C(\omega_n) = C' \wedge \forall\vec{x}.\phi$. Then for all i, writing $\omega_i = (\Pi_i, C_i, O_i)$:

$$\omega_i' = (\Pi_i, C_i^+, O_i \cup \{(\Pi(\varepsilon), C_i^-)\})$$

where, with the convention $dom(\omega_{-1}) = \varnothing$:

$$C_i^+ = C_i : \Sigma_{|dom(\omega_i)} \wedge C_t$$
$$C_i^- = C_i \wedge (\neg\Sigma_{|dom(\omega_i)\setminus dom(\omega_{i-1})} \vee \neg C_t)$$

where C_t refers to the numeric constraints of $C_n : \Sigma_{|dom(\omega_n)}$.

Saturation. This last rule is then used to saturate the knowledge base of the adversary, i.e., to ensure that direct deductions (used in particular in the computation of most general solutions) coincide with regular adversarial deductions. To define the rule, we first introduce a couple of preliminary notions. First of all, we define the *origination property*, which is a straightforward invariant verified by our decision procedure. It intuitively states that we only consider adversarial computations whose multiplicities are consistent with an actual protocol execution.

Definition D.5 (Origination property)**.** We say that a hyperconstraint system C verifies the *origination property* if for all $\pi \in vars^p(C)$, the domain of $\Phi = \Phi(\text{proj}_\pi(C))$ can be ordered as

$$dom(\Phi) = \{\mathsf{ax}_1, \ldots, \mathsf{ax}_n\}$$

in a way that for all $i \in [\![1, n]\!]$, for all $x \in vars^1(\mathsf{ax}_i)$, there exists $X : M \vdash^?_\pi x \in \mathsf{D}(C)$ such that $\{\mathsf{ax}_j\}_{j<i} \subseteq M$.

In the following, we will always assume that hyperconstraint systems verify the origination property. Then, we define the following notion, which characterises the possibility to perform a new deduction from the knowledge base.

Definition D.6 (New deduction)**.** Let C be a hyperconstraint system. We call $(\Sigma, \zeta \vdash^?_\pi v)$ a *new deduction* for C if there exists a rewrite rule $\mathsf{f}(\ell_1, \ldots, \ell_n) \longrightarrow r$ and $\xi \vdash^?_\pi u \in \mathsf{K}(C)$ such that the following conditions are satisfied.

(1) There should not exist ξ' such that $\xi' \Vdash_{C,\pi} v$.
(2) Consider a hyperconstraint of the form

$$C' = C \wedge \bigwedge_{i=1}^{n} X_i : M \vdash^?_\pi \ell_i \wedge \ell' =^? u$$

with ℓ' subterm of some ℓ_i, and r is a subterm of ℓ'. Then $\Sigma \in mgs(C')$, $\zeta = \mathsf{f}(X_1\Sigma, \ldots, X_n\Sigma)$ and $v = r\sigma$, with σ the non-numeric constraints of $mgu(\mathsf{E}^1(C' : \Sigma))$.

(3) In the above item, M is of the form $\{\mathsf{ax}_1, \ldots, \mathsf{ax}_n\}$, with these axioms numbered as in the definition of origination property. Besides, there should not exist other new deductions for a multiplicity $M' \subset M$.

Using this notion, the rule can be formalised as follows:

$$(\omega_1 \rightarrowtail \cdots \rightarrowtail \omega_n) \xrightarrow[\mathsf{case}]{} (\omega_1' \rightarrowtail \cdots \rightarrowtail \omega_n') \qquad \text{(Case-Saturate)}$$

where the following conditions are met. First of all, there should exists $(\Sigma, \zeta \vdash^?_\pi v)$ a new deduction for $C(\omega_n)$. Then for all i, writing $\omega_i = (\Pi_i, C_i, O_i)$:

$$\omega_i' = (\Pi_i, C_i^+, O_i \cup \{(\Pi(\varepsilon), C_i^-)\})$$

where, with the convention $dom(\omega_{-1}) = \varnothing$:

$$\begin{aligned} C_n^+ &= C_i : \Sigma_{|dom(\omega_i)} \wedge C_t \wedge \zeta \vdash^?_\pi v \\ \text{if } i < n\text{: } C_i^+ &= C_i : \Sigma_{|dom(\omega_i)} \wedge C_t \\ C_i^- &= C_i \wedge (\neg\Sigma_{|dom(\omega_i) \setminus dom(\omega_{i-1})} \vee \neg C_t) \end{aligned}$$

where C_t refers to the numeric constraints of $C_n : \Sigma_{|dom(\omega_n)}$.

Constraint Solving. Putting everything together, the *constraint solving* relation $\overset{\mathsf{cs}}{\rightsquigarrow}$ on proof stacks $\omega^\star$ is defined by the successive applications of the rules $\overset{\mathsf{solve}}{\rightsquigarrow}$ and $\xrightarrow[\mathsf{case}]{}$, as many times as possible, with the priority: (1) $\overset{\mathsf{solve}}{\rightsquigarrow}$, then (2) $\xrightarrow[\mathsf{case}]{}$ by Rule (Case-Saturate), and finally (3) $\xrightarrow[\mathsf{case}]{}$ by Rule (Case-Resolve).

E VERIFICATION OF HYPERPROPERTIES

E.1 Decision Procedure

Setting. We now present the decision procedure itself, relying on the constraint solving relation $\overset{\mathsf{cs}}{\rightsquigarrow}$ introduced in the previous section. We describe the behaviour of HCompute in the rest of this section, by induction on φ. However, to lighten the presentation of some cases, we first introduce notations to compose recursive calls to HCompute in a way that reflects logical connectives. First, we give a notation $\oplus$ for component-by-component union:

$$(\Omega_1^+, \Omega_1^-) \oplus (\Omega_2^+, \Omega_2^-) = (\Omega_1^+ \cup \Omega_2^+, \Omega_1^- \cup \Omega_2^-)\,.$$

Another, more involved notion of composition is the one reflecting the verification of implications $\varphi \Rightarrow \psi$. First, a split $(\Omega_\varphi, \Omega_{\neg\varphi})$ is computed for φ, and Ω_φ is then refined to obtain a split $\Omega_{\varphi\wedge\psi}, \Omega_{\varphi\wedge\neg\psi}$. Formally, we let h_i^+, h_i^- be functions mapping proof stacks to sets of proof stacks, and let $h_i : \omega \mapsto (h_i^+(\omega), h_i^-(\omega))$. In our below definition of HCompute, we will typically have

$$h_i = \mathsf{HCompute}(\cdot, \omega^{-1})\,.$$

Then the *logical composition* of h_0 and h_1 is defined as the following function:

$$(h_0 \Rightarrow h_1) : \omega^\star \mapsto (h_0^-(\omega^\star), \varnothing) \oplus \bigoplus_{\omega^{\star\prime} \in h_0^+(\omega^\star)} h_1(\omega^{\star\prime})\,.$$

Using a similar construction, the *negation composition* of h_0 can be defined from it, doing a swap:

$$(\neg h_0)(\omega^\star) \triangleq (h_0 \Rightarrow h_\bot)(\omega^\star) = (h_0^-(\omega^\star), h_0^+(\omega^\star))$$

where $h_\bot(\omega^\star) = (\varnothing, \{\omega^\star\})$. Similarly, we can define:

$$h_0 \vee h_1 = (\neg h_0) \Rightarrow h_1 \qquad\qquad h_0 \wedge h_1 = \neg(\neg h_0 \vee \neg h_1)\,.$$

Case-By-Case Computation. We can finally formalise the definition of HCompute. In all incoming cases, we will use the notations $(\Omega_\varphi, \Omega_{\neg\varphi}) = \mathsf{HCompute}(\omega^\star, \omega^{-1})$, with $\omega^\star$ and $\omega^{-1} = (\Pi_@, t, \varphi)$. First of all, $\mathsf{HCompute}(\omega^\star, \omega^{-1})$ only applies if $\omega^\star$ is a non-empty proof stack, and starts by applying the constraint solving procedure to it, i.e., we write $\omega^\star \overset{\text{cs}}{\rightsquigarrow} \omega^\star_s$, where

$$\omega^\star_s = \omega_1 \rightarrowtail \cdots \rightarrowtail \omega_n \qquad n \geqslant 1\,.$$

We then perform the following case analysis. In several cases, we also use the function HRefine, defined by

$$\mathsf{HRefine}(\sigma, \omega^\star) = \omega^\star_{s'}$$

with $\omega^\star[C(\omega) \mapsto C(\omega) \wedge \sigma] \overset{\text{cs}}{\rightsquigarrow} \omega^\star_{s'}$, with $\omega^\star = \omega^\star_{pref} \rightarrowtail \omega$.

▸ *Non-concretisable case:* $C(\omega_n) = \bot$.

Then we simply return $\Omega_\varphi = \{\omega_n\}$ and $\Omega_{\neg\varphi} = \varnothing$.

▸ *Event case:* $\varphi = \mathsf{Ev}(\vec{u})_\pi$.

In this case, we interpret event functions $\mathcal{F}_e$ as constructor symbols of $\mathcal{F}_\mathrm{c}$. We thus consider the term $v = \mathsf{Ev}(\vec{u})$, and:

$$\Pi_@(\pi) = (A, E) \qquad \text{and} \qquad \mu = mgu(\mathsf{E}^1(C))\,.$$

Using these notations, we return

$$\Omega_\varphi = \{\mathsf{HRefine}(\sigma, \omega^\star_s) \mid w \in E, \sigma \in mgu_{\mathcal{R}}(v\mu =^? w\mu)\}$$
$$\Omega_{\neg\varphi} = \{\mathsf{HRefine}(\neg\sigma, \omega^\star_s) \mid w \in E, \sigma \in mgu_{\mathcal{R}}(v\mu =^? w\mu)\}$$

▸ *Non-event action case:* $\varphi = \alpha_\pi$ *(*$\alpha \in \{\mathsf{in}, \mathsf{out}, \mathsf{repl}, \ldots\}$*).*

Analogue to the previous case.

▸ *State membership case:* $\varphi = \mathsf{GS}_\pi(u)$.

Let us write:

$$\Pi_@(\pi) = (A, E) \qquad \text{and} \qquad \mu = mgu(\mathsf{E}^1(C))\,.$$

Using these notations, we return

$$\Omega_\varphi = \{\mathsf{HRefine}(\sigma, \omega^\star_s) \mid v \in \theta(A), \sigma \in mgu_{\mathcal{R}}(u\mu =^? v\mu)\}$$
$$\Omega_{\neg\varphi} = \{\mathsf{HRefine}(\neg\sigma, \omega^\star_s) \mid v \in \theta(A), \sigma \in mgu_{\mathcal{R}}(u\mu =^? v\mu)\}$$

▸ *Equality modulo* $\mathcal{R}$ *case:* $\varphi = (u = v)$.

In this case, it suffices to add a (dis)equality constraint to C and to resolve it using the oracle. That is, we let:

$$\mu = mgu(\mathsf{E}^1(C)) \qquad\qquad M = mgu_{\mathcal{R}}(u\mu =^? v\mu)$$

and then return

$$\Omega_\varphi = \{\mathsf{HRefine}(\sigma, \omega^\star_s) \mid \sigma \in mgu_{\mathcal{R}}(u\mu =^? v\mu)\}$$
$$\Omega_{\neg\varphi} = \{\mathsf{HRefine}(\neg\sigma, \omega^\star_s) \mid \sigma \in mgu_{\mathcal{R}}(u\mu =^? v\mu)\}$$

▸ *Adversarial deduction case:* $\varphi = \mathsf{K}_\pi(u)$.

This case is mostly similar to the equality-modulo case, except that we use (non-)deduction constraints instead of (dis)equations. We therefore let, this time:

$$\mu = mgu(\mathsf{E}^1(C)) \qquad\qquad M = mgu_{\mathcal{R}}(u\mu =^? u\mu)\,.$$

One subtlety, however, is that (non-)deduction constraints of a term u require that u is a constructor term. In particular, φ does not hold when either u is a message but not deducible, *or* if it is not instantiated as a message. With this in mind, we let $X \in \mathcal{X}^2$ be a fresh variable, and:

$$M^+ = \{\sigma \wedge X \vdash^?_\pi u\sigma\downarrow \mid \sigma \in M\}$$
$$M^- = \{\sigma \wedge \forall X.X \nvdash^?_\pi u\sigma\downarrow \mid \sigma \in M\} \cup \{\bigwedge_{\sigma\in M} \neg\sigma\}$$

We then return

$$\Omega_\varphi = \{\mathsf{HRefine}(\gamma, \omega_s^\star) \mid \gamma \in M^+\}$$
$$\Omega_{\neg\varphi} = \{\mathsf{HRefine}(\gamma, \omega_s^\star) \mid \gamma \in M^-\}$$

▸ *Implication case:* $\varphi = \forall\vec{x}.\varphi_0 \Rightarrow \varphi_1$.
Writing $h_i = \mathsf{HCompute}(\cdot, \omega^{-1}[\varphi \mapsto \varphi_i])$:

$$(\Omega_\varphi, \Omega_{\neg\varphi}) = (h_0 \Rightarrow h_1)(\omega_s^\star)\,.$$

Note that the quantified variables $\vec{x}$ do not need to be taken into account in the constraint computation. Indeed, due to the requirement that they are guarded, they can be left as free variables as they will be unified with process variables during the recursive call h_0.

▸ *Until case:* $\varphi = \varphi_0 \,\mathsf{U}\, \varphi_1$.

Since timestamps are abstracted by time variables, all possible time projections (notation $T@t$ in the non-symbolic case, Definition 3.7) induce a collection of ordering constraints. These constraints intuitively represent all potential schedulings of the different traces of Π.

Formally, if T is a symbolic trace and $t' \in \mathcal{X}^{\mathsf{N}}$, we define the following set $T@t'$, intuitively representing all possible symbolic abstractions of a time projection of T. Each entry of $T@t'$ is thus a triple (γ, A', E'), where γ models a scheduling constraint for t', and (A', E') is the corresponding trace state $\Pi_{@t'}$ for this trace. Formally, if $T : A_0 \xrightarrow{\vec{w}_1}_{\mathsf{s}} \cdots \xrightarrow{\vec{w}_n}_{\mathsf{s}} A_n$, and with convention $t(A_{n+1}) = +\infty$:

$$\begin{aligned} T@t' = &\{(t(A_i) < t' < t(A_{i+1}), A_i[t(A_i) \mapsto t'], \varnothing)\}_{i\in[\![0,n]\!]} \\ &\cup \{(t' = t(A_i), A_i, \{\vec{w}_i\})\}_{i\in[\![1,n]\!]} \end{aligned}$$

with $A[t(A) \mapsto t']$ the symbolic process A with $t(A)$ replaced by t'. We generalise this notation to the whole mapping Π by computing a cartesian product:

$$\Pi@t' = \left\{ \left(\bigwedge_{\pi\in dom(\Pi)} \gamma_\pi,\ t',\ \{\pi \mapsto (A_\pi, E_\pi)\}_{\pi\in dom(\Pi)} \right) \right. \\ \left. \mid \forall \pi \in dom(\Pi), (\gamma_\pi, A_\pi, E_\pi) \in \Pi(\pi)@t' \right\}$$

We then consider two sets R_{φ_0} and R_{φ_1}: $R_{\varphi_1} = \Pi@t_1$ for some fresh $t_1 \in \mathcal{X}^{\mathsf{N}}$, and R_{φ_0} is the set $\Pi@t'$ where in each $a \in \Pi@t'$, t' is substituted by a different fresh numeric variable t_a. We also consider the two functions $h^\wedge_\gamma, h^\vee_\gamma$, writing $\omega_s^\star[C(\omega_{s,last}) \mapsto C(\omega_{s,last}) \wedge \gamma] \overset{\mathsf{cs}}{\rightsquigarrow} \omega'$ with $\omega_{s,last}$ the last state of $\omega_s^\star$:

$$h^o_\gamma(\omega, (\Pi_@, t, \varphi)) = \begin{cases} (\{\omega\}, \varnothing) & \text{if } C(\omega') = \bot \text{ and } o = \wedge \\ (\varnothing, \{\omega\}) & \text{if } C(\omega') = \bot \text{ and } o = \vee \\ \mathsf{HCompute}(\omega', \Pi_@, t, \varphi) & \text{if } \omega' \neq \bot \end{cases}$$

With all these notations, the returned value in this case is finally $(\Omega_\varphi, \Omega_{\neg\varphi}) = h(\omega)$, where:

$$h = \bigvee_{(\gamma_1, t_1, \Pi^1_@)\in R_{\varphi_1}} h^\vee_{\gamma_1}(\cdot, (\Pi^1_@, t_1, \varphi_1)) \wedge \bigwedge_{(\gamma_0, t_0, \Pi^0_@)\in R_{\varphi_0}} h^\wedge_{\gamma_0}(\cdot, (\Pi^0_@, t_0, \varphi_0))$$

▸ *Path quantification case:* $\varphi = \forall\pi.\psi$.

We first let $\Pi_{@}(\varepsilon) = (A, E)$, and $\mathbb{T}$ be a mapping from fresh path variables to symbolic traces, where $img(\mathbb{T})$ is the set of all symbolic traces of A. We also consider the hyperconstraint $C(\pi')$, $\pi' \in dom(\mathbb{T})$, obtained from $C(\mathbb{T}(\pi'))$ by labelling (non-)deduction constraints by π'. We then define a function $h_{\pi'}$, $\pi' \in dom(\mathbb{T})$, which intuitively attempts to prove ψ for the symbolic trace $\mathbb{T}(\pi)$, assuming this symbolic trace has a solution (otherwise $h_{\pi'}$ skips the proof). I.e.:

$$(\Omega_{\varphi}, \Omega_{\neg\varphi}) = \Big(\mathsf{HCompute}(\cdot, (\Pi'_{@}, t, \psi')) \wedge \mathsf{RecCall}(\cdot)\Big)\,(\omega')$$

where:

$$\begin{aligned}
&\psi' = \psi\{\pi \mapsto \pi'\}\\
&\Pi' = \Pi[\varepsilon \mapsto \mathbb{T}(\pi')] \cup \{\pi' \mapsto \mathbb{T}(\pi')\}\\
&\Pi'_{@} = \Pi_{@}[\varepsilon \mapsto (A, \varnothing)] \cup \{\pi' \mapsto (A, \varnothing)\}\\
&\omega' = \omega^{\star}_{s} \rightarrowtail (\Pi', C(\pi'), \{(\mathbb{T}(\pi), C(\pi)) \mid \pi \in dom(\mathbb{T}), \pi \neq \pi'\})
\end{aligned}$$

and $\mathsf{RecCall}(\omega^{\star} \rightarrowtail \omega_{prelast} \rightarrowtail \omega_{last})$ is defined as $(\{\omega^{\star} \rightarrowtail \omega_{prelast}\}, \varnothing)$ when $O(\omega_{prelast}) = \varnothing$; otherwise, for a given $(\mathbb{T}(\pi''), C) \in O(\omega_{prelast})$:

$$\mathsf{RecCall}(\omega^{\star} \rightarrowtail \omega_{prelast} \rightarrowtail \omega_{last}) = \mathsf{HCompute}(\omega^{\star} \rightarrowtail \omega'', (\Pi'_{@}, t, \psi''))$$

$$\begin{aligned}
\text{with}\quad &\psi'' = \psi\{\pi \mapsto \pi''\}\\
&\Pi'' = \Pi'[\varepsilon \mapsto \mathbb{T}(\pi'')] \cup \{\pi'' \mapsto \mathbb{T}(\pi'')\}\\
&\omega'' = \omega_{prelast}[O \mapsto O \setminus \{(\mathbb{T}(\pi''), C)\}]
\end{aligned}$$

E.2 Correctness and Complexity

Decidability. We state in this section the main properties for the correctness of our decision procedure, thus leading to the decidability (and tight complexity analysis) of Verif, given as before an oracle to solve the numeric constraints. The characterisation of HCompute is in particular the following. First, given a statement $P \models \varphi$ to prove or disprove, we consider the following parameters to give as an initial argument to HCompute:

Definition E.1 (Initial parameters of HCompute)**.** Given a process P and a guarded hyperformula φ, let $t \in \mathcal{X}^{\mathsf{N}}$, and P_{s} be the symbolic process:

$$P_{\mathsf{s}} = (\{\!|P|\!\}, (t = 0), \varnothing, t)\,.$$

We call *initial parameters* of HCompute for P and φ a tuple $(\omega^{\star}, \omega^{-1})$, where $\omega^{\star}$ is a proof stack consisting of the single proof state $(\Pi, (t = 0), \varnothing)$, $\omega^{-1} = (\Pi, (t = 0), \varnothing)$, $\Pi = \{\varepsilon \mapsto \varepsilon_{P_{\mathsf{s}}}\}$ with $\varepsilon_{P_{\mathsf{s}}}$ the empty symbolic trace starting from P_{s}, and $\Pi_{@} = \{\varepsilon \mapsto (P_{\mathsf{s}}, \varnothing)\}$.

The specification of HCompute can then be formally stated as the following proposition. Note that it is only a partial correctness statement, i.e., it proves that HCompute is correct when it terminates but does not prove termination yet.

Proposition E.1 (Correctness of HCompute)**.** *Let P be a ground bounded process, and φ a ground guarded hyperformula. Given $\omega^{\star}, \omega^{-1}$ some initial parameters of* HCompute *for P and φ, if we have*

$$(\Omega_{\varphi}, \Omega_{\neg\varphi}) = \mathsf{HCompute}(\omega^{\star}, \omega^{-1})$$

then $P \models \varphi$ iff $\Omega_{\neg\varphi} = \varnothing$.

Proof. The proof follows from a straightforward, although technical, induction on the formula φ to be proved. The actual invariant considered during this induction is the following. Given parameters $\omega^\star, \omega^{-1}$ that are not necessarily initial, but with $\omega^\star = \omega_0^\star \rightarrowtail \omega$, if

$$(\Omega_\varphi, \Omega_{\neg\varphi}) = \mathsf{HCompute}(\omega^\star, \omega^{-1})$$

then for all (non-empty) proof stacks $(\omega^\star \rightarrowtail \omega) \in \Omega_\psi$, for all solutions $S = (\Sigma, \sigma)$ of $C(\omega)$,

$$\Pi(\omega)S, t(\omega^{-1})\sigma \models \varphi\sigma\,.$$

In particular, in the case of a ground process P and formula φ, the emptiness of $\Omega_{\neg\varphi}$ is equivalent to $P \models \varphi$. □

Complexity. Albeit for the missing termination arguments, the above proposition justifies that Verif is decidable for bounded processes and guarded hyperformulae with oracle to numeric-constraint solving. However, different arguments are necessary to obtain the expected complexity. Indeed, we recall that **EXPH**(poly) is a class typically characterised by alternating Turing machines (the class of problems decidable in exponential time by an alternating procedure with a polynomial number of alternations), while HCompute is purely deterministic and may run in double-exponential time.

To obtain an **EXPH**(poly) bound, we consider the *naive n-bounded algorithm* $\mathsf{NBA}_n(P, \varphi) \in \{\mathsf{true}, \mathsf{false}\}$, which is the bruteforce alternating (but incomplete) algorithm for proving $P \models \varphi$ which bounds the size adversarial computations by n to obtain a finite set of traces. Its definition is straightforward, and similar to HCompute but operates on concrete processes and traces, and hence uses concrete recipes (of size n at most) instead of deduction constraints.

Proposition E.2 (Complexity and Soundness of NBA)**.** *The procedure* $\mathsf{NBA}_n(P, \varphi)$ *runs in alternating polynomial time in n and the sizes of* P *and* φ*, and also involves a polynomial number of alternations. Besides, let*

$$(\Omega_\varphi, \Omega_{\neg\varphi}) = \mathsf{HCompute}(\omega, \omega^{-1})$$

for initial parameters of HCompute *for* P *and* φ *with*

$$n \geqslant \mu(\Omega_\varphi \cup \Omega_{\neg\varphi}) \triangleq \max_{\substack{(\omega^\star \rightarrowtail \omega) \in \Omega_\varphi \cup \Omega_{\neg\varphi} \\ \pi \in vars^P(\omega)}} \mu_\pi(C(\omega))\,.$$

where $\mu_\pi(C)$ *refers to the size of a mgs of* C *projected on path* π*, i.e.,* $mgs(C)_{|vars^2(\mathrm{proj}_\pi(C))}$*. Then* $P \models \varphi$ *iff* $\mathsf{NBA}_n(P, \varphi)$ *accepts.*

This states that NBA_n is a sound procedure provided n is greater than all (projections of) mgs generated by HCompute. This last property directly follows from the correctness of HCompute and the definition of mgs.

Proposition E.3 (Small-solution property)**.** *We assume a destructor subterm term algebra* $\mathcal{A}$*. There is a polynomial* p *such that for all processes* P*, for all*

$$(\Omega_\varphi, \Omega_{\neg\varphi}) = \mathsf{HCompute}(\omega, \omega)$$

obtained with initial parameters of HCompute *for* P*,*

$$\mu(\Omega_\varphi \cup \Omega_{\neg\varphi}) \leqslant 2^{p(|P|+|\varphi|+|\mathcal{A}|)} \qquad \text{in general}$$

$$\mu(\Omega_\varphi \cup \Omega_{\neg\varphi}) \leqslant p(|P| + |\varphi| + |\mathcal{A}|) \qquad \text{if } \varphi \text{ is in tidy LTL}$$

Proof. The proof follows again from an induction on the structure of φ, proving that a well-founded (exponential) measure decreases at each recursive call of HCompute and solving and case analysis rules. The measure itself is simply a lexicographic product of the following parameters:

(1) the size of the formula φ to prove (polynomial);
(2) the multiset of the numbers of instructions of symbolic processes appearing in the proof stack (polynomial);
(3) the multiset of the numbers of deducible term that are not a direct consequence of some proof state. This parameter is of polynomial size since, for destructor subterm convergent rewriting systems, all deducible terms from a frame Φ are of the form $C[u]$, for C a public constructor context and u a subterm of some $\mathrm{ax}\Phi \in img(\Phi)$ (and there are polynomially-many in the size of P);
(4) the multiset of the numbers of most general direct solutions of the constraint system of some proof stack (polynomial, as we consider only direct solutions);
(5) the multiset of the numbers of proof obligations at each level of the stack (one multiset for each level, lexicographically ordered by increasing stack level order). This parameter is polynomial in the number of traces of P, which is exponential in the size of P and $\mathcal{A}$.

In the case of tidy LTL, $\varphi = \forall\pi.\psi$ contains only one trace quantification at the head of the formula. Therefore, the verification of φ using NBA can be done by initially guessing a trace non-deterministically, and then verify for the guessed trace that ψ holds (which thus does not involve proof obligations). In particular, in the above measure, only polynomial parameters remain, hence the result. □

In particular, a trivial combination of the previous two propositions gives the following corollary, using NBA_n as a decision procedure with some exponential value of n.

Theorem E.4 (Complexity of Verif)**.** *Verif is in **EXPH**(poly) for destructor subterm term algebras, bounded processes, and guarded hyperformulae.*

F COMPLEXITY OF SUB-FRAGMENTS

Upper Bounds. We prove in this section the side complexity results we provided in Figure 9, and that apply to sub-fragments of the logic.

First of all, using the polynomial-solution property of the previous section (Theorem E.3), we already obtain the complexity of the fragments of tidy LTL, again using the naive NBA procedure:

Corollary F.1 (Complexity of tidy LTL). *For a destructor subterm convergent term algebra, bounded processes and guarded hyperformulae, the Verif problem is in **PSPACE** (resp. **NP**, co**NP**) for tidy LTL (resp. tidy $\exists$-LTL , tidy $\forall$-LTL).*

Exhibiting complexity upper bounds for bounded processes of the pure π-calculus, i.e., with an empty term algebra, is also straightforward, since there are this time finitely-many traces up to bijective renaming of fresh public names. The naive NBA_n procedure is therefore trivially sound for $n = 1$.

Proposition F.2 (Complexity in the pure calculus)**.** *For bounded processes and guarded hyperformulae, the Verif problem is in **PSPACE** (resp. **NP**, co**NP**) in general (resp. for tidy $\exists$-LTL , tidy $\forall$-LTL).*

Lower Bounds. We now prove that all the complexity results of this paper are tight. We already proved the undecidability results in Section 4.1. Also, the hardness results for the main decision procedure are already proved in [Barthe et al. 2022]:

Theorem F.3 (Hardness of Verif)**.** *For a destructor subterm convergent term algebra, bounded process, and guarded hyperformulae, the Verif problem is **EXPH**(poly) hard in tidy CTL* and Hypertidy LTL.*

This result of [Barthe et al. 2022] is proved in a slightly different context, which however does not affect the validity of the proof. First, it defines a notion of guard that is much more restrictive than ours (which even makes the result stronger than what we stated here). Second, the hardness was only stated for subterm convergent term algebras, i.e., not necessarily destructor. However, the reduction used to prove the result was trivially verifying the constructor-destructor property. The corresponding proof of correctness of the reduction can then easily be kept up to minor modifications to match the destructor setting (i.e., to account for the fact that events and outputs are only executed when their arguments are messages).

It thus remains to prove tight lower bounds for tidy LTL (**PSPACE**) and tidy ∀-LTL (co**NP**) in the pure π calculus. The two reductions are conducted below.

Proposition F.4 (Hardness of tidy LTL)**.** *For an empty term algebra, bounded processes and guarded hyperformulae, the Verif problem is **PSPACE** hard in tidy LTL.*

Proof. We prove this result by a reduction from QBF. The encoding is pretty straightforward as guarded quantifiers easily allow to encode boolean quantifications. Formally, we let $\mathbf{0}, \mathbf{1} \in \mathcal{N}_{pub}$ and:

$$P = \mathsf{Bool}(\mathbf{0}); \mathsf{Bool}(\mathbf{1})$$

Given a quantified boolean formula $\forall x_1.\exists y_1. \ldots \forall x_n.\exists y_n.\phi$, we define the guarded hyperformula:

$$\begin{aligned} \varphi = \quad & \forall \pi.\mathsf{F}\ \mathsf{Bool}(\mathbf{1})_\pi \Rightarrow \\ & \forall x_1.\mathsf{F}\ \mathsf{Bool}(x_1)_\pi \Rightarrow \exists y_1.\mathsf{F}\ \mathsf{Bool}(y_1)_\pi \wedge \\ & \qquad\qquad \vdots \\ & \forall x_n.\mathsf{F}\ \mathsf{Bool}(x_n)_\pi \Rightarrow \exists y_n.\mathsf{F}\ \mathsf{Bool}(y_n)_\pi \wedge \\ & \overline{\phi} \end{aligned}$$

where $\overline{\phi}$ is the interpretation of the boolean formula ϕ as a guarded hyperformulae, i.e., interpreting $\wedge$ and $\vee$ as the same symbols of our logic, and interpreting a literal x as the hyperformula $x = \mathbf{0}$, and $\neg x$ as $x = \mathbf{1}$. It is then straightforward that $P \models \varphi$ *iff* $\forall x_1.\exists y_1. \ldots \forall x_n.\exists y_n.\phi$ holds. □

Proposition F.5 (Hardness of tidy ∀-LTL and tidy ∃-LTL)**.** *For an empty term algebra, bounded processes and guarded hyperformulae, the Verif problem is co**NP** hard (resp. **NP** hard) in tidy ∀-LTL (resp. tidy ∃-LTL).*

Proof. It suffices to prove the **NP** hardness of tidy ∃-LTL , as φ is in tidy ∃-LTL *iff* $\neg\varphi$ is in tidy ∀-LTL . We proceed by reduction from SAT. Let us thus consider a boolean formula in CNF, $\phi = \bigwedge_{i=1}^{p} C_i$, whose variables are written $\vec{x} = x_1, \ldots, x_n$. As before, we let two names $\mathbf{0}, \mathbf{1} \in \mathcal{N}_{pub}$ modelling booleans. For each clause C_i, we consider an event Sat_i, and we define the following process:

$$P_i = (\mathsf{Eq}(x_{i_1}, b_{i_1}) : \mathsf{Sat}_i) \mid \cdots \mid (\mathsf{Eq}(x_{i_\ell}, b_{i_\ell}) : \mathsf{Sat}_i)$$

where $x_{i_1}, \ldots, x_{i_\ell}$ are the variables of C_i (they are here free variables in Sat_i) and $b_{i_1}, \ldots, b_{i_\ell} \in \{\mathbf{0}, \mathbf{1}\}$ are their respective negation bits. Intuitively, interpreting the event Eq as an equality predicate, P_i can emit the event Sat_i *iff* the variables $x_1, \ldots, x_n$ are instantiated in a wy that C_i holds. In particular, the satisfiability of the whole boolean ϕ is characterised by the following process P and guarded hyperformula φ:

$$\begin{aligned} P = \quad & \mathsf{in}(x_1) : \cdots : \mathsf{in}(x_n); (P_1 \mid \cdots P_n) \\ \varphi = \quad & \exists \pi.(\mathsf{G}\ \forall x, y.\mathsf{Eq}(x, y) \Rightarrow x = y) \wedge \textstyle\bigwedge_{i=1}^{n} \mathsf{F}\ \mathsf{Sat}_i\,. \end{aligned}$$

We then observe that $P \models \varphi$ *iff* ϕ is satisfiable. □