Cryptographic Choreographies

Sebastian Mödersheim
DTU Compute
Kgs. Lyngby, Denmark
0000-0002-6901-8319

Simon Lund
DTU Compute
Kgs. Lyngby, Denmark
0009-0005-2957-3472

Alessandro Bruni
IT-University of Copenhagen, Denmark
0000-0003-2946-9462

Marco Carbone
IT-University of Copenhagen, Denmark
0000-0001-9479-2632

Rosario Giustolisi
IT-University of Copenhagen, Denmark
0000-0002-2917-9601

***Abstract*—We present CryptoChoreo, a choreography language for the specification of cryptographic protocols. Choreographies can be regarded as an extension of Alice-and-Bob notation, providing an intuitive high-level view of the protocol as a whole (rather than specifying each protocol role in isolation). The extensions over standard Alice-and-Bob notation that we consider are non-deterministic choice, conditional branching, and mutable long-term memory. We define the semantics of CryptoChoreo by translation to a process calculus. This semantics entails an understanding of the protocol: it determines how agents parse and check incoming messages and how they construct outgoing messages, in the presence of an arbitrary algebraic theory and non-deterministic choices made by other agents. While this semantics entails algebraic problems that are in general undecidable, we give an implementation for a representative theory. We connect this translation to ProVerif and show on a number of case studies that the approach is practically feasible.**

I. Introduction

Specification languages for security protocols can be roughly divided into three classes. The most low-level one are based on multi-set rewriting rules, such as the Tamarin input language [1] and the AVISPA Intermediate Format [2], where each rule describes state transitions corresponding usually to a pair of protocol steps from the view of one honest agent: receiving a message, processing and checking it, and sending the next message. More high-level are languages based on process calculus such as ProVerif [3], where typically each role of the protocol is described like a program, often as a sequence of sending and receiving steps. The most high-level are languages based on Alice-and-Bob notation [4], [5], [6], [7], [8], [9], [10], which describe the entire protocol by an ideal run of the protocol as a sequence of $A \to B : M$ steps where role A sends message M to role B, thus describing the interplay of all roles.

Alice-and-Bob notation is very intuitive and succinct because it gives the synopsis of the protocol and leaves implicit how agents construct the messages they send, and how they parse and check the messages they receive. The latter is a non-trivial problem one has to solve when defining a formal language based on Alice-and-Bob notation, namely when giving a formal *semantics* by translation to a lower-level language. This was described by [6], [11] for models in the free term algebra, but a key question is how to deal with algebraic properties as needed, for instance, for Diffie-Hellman. If Alice needs to construct $\exp(\exp(g, X), Y)$, given her own secret X and the public value $\exp(g, Y)$ from Bob, the semantics needs to infer that this is possible by composing $\exp(\exp(g, Y), X)$ since this is equivalent to the goal term by the algebraic properties of exponentiation. It turns out that one can define such semantics in a general, uniform, and concise way for an *arbitrary* algebraic theory as an intruder deduction problem [7], [8], [9], [10]

As a side-effect of "abusing" the intruder deduction to define the behavior of honest agents, one prevents many specification errors that can easily happen in the lower-level formalisms, e.g., when the message sent by one agent are different from the messages that another agent expects, rendering the protocol unexecutable. This may lead in the worst case to a false negative (an attack of the real system is not detected because of a specification error). Many such errors are prevented by formal Alice and Bob approaches because the protocol would be refused as unexecutable by the compiler.

Thus, Alice-and-Bob notation is a beneficial and accessible specification language that can be used even without a deep background in formal verification. It is also striking how often scientific works that formalize a protocol using a lower-level language first summarize the protocol informally in Alice-and-Bob notation. However, existing formal Alice-and-Bob languages do not support branches in the protocol execution (by conditions or non-determinism), unbounded repetition, or mutable long-term memory: everything is restricted to a linear session of fixed length.

While informal uses of Alice-and-Bob notation can easily be extended ad hoc, the first main contribution of this paper is a new choreography language, called *CryptoChoreo*, which extends Alice-and-Bob notation with non-deterministic choice, branching, and mutable long-term memory.[1]

Such features are needed for instance if we want to formulate a protocol with a server that maintains a long-term database and that may react to a request in different ways depending the current state of its database. Non-deterministic choice can be helpful for modelling several options in a protocol that are at a participant's discretion, where we do not want to formalize how they make a decision. Especially this allows us to formulate an API where a user can non-deterministically choose to send any of a number of commands to a server (who may in turn ask other servers in order to answer the request).

[1]Since a choreography can be executed an unbounded number of times, non-determinism and long-term memory are sufficient to formulate repetitions and sequential composition of protocols without an explicit repetition construct.

$$
\begin{array}{l}
\mathsf{A} : (\nu M. \\
\quad \mathsf{A} \to \mathsf{s} : \mathit{scrypt}((\mathsf{B}, \mathit{crypt}((\mathit{msg}, M), \mathit{ek}(\mathsf{B}))), \mathit{shk}(\mathsf{A}, \mathsf{s})). \\
\quad \mathsf{s} \to \mathsf{B} : \mathit{sign}((\mathsf{A}, \mathit{crypt}((\mathit{msg}, M), \mathit{ek}(\mathsf{B}))), \mathit{inv}(\mathit{sk}(\mathsf{s}))). \\
\quad \mathsf{B}\ \textsf{ni-authenticates}\ \mathsf{A}\ \textsf{on}\ M) \\
+\ (\nu K. \\
\quad \mathsf{A} \to \mathsf{s} : \mathit{scrypt}((\mathsf{B}, \mathit{crypt}((\mathit{upd}, K), \mathit{ek}(\mathsf{B}))), \mathit{shk}(\mathsf{A}, \mathsf{s})). \\
\quad \mathsf{s} \to \mathsf{B} : \mathit{sign}((\mathsf{A}, \mathit{crypt}((\mathit{upd}, K), \mathit{ek}(\mathsf{B}))), \mathit{inv}(\mathit{sk}(\mathsf{s}))). \\
\quad K\ \textsf{secret between}\ \mathsf{A}, \mathsf{B})
\end{array}
$$

Fig. 1. Example Choreography.

CryptoChoreo is, in a sense, a conservative extension of formal Alice-and-Bob languages: we give a semantics—parameterized over an arbitrary algebraic theory—that agrees with standard Alice-and-Bob languages on the subset that does not use the new constructs. A particular challenge for this semantics is to integrate the algebraic understanding of the protocol with branching: if one party has made a non-deterministic choice, other parties do not necessarily know which choice was taken. For instance, our semantics allows for the following protocol: Alice non-deterministically chooses one of two types of message that she sends encrypted for Bob over an intermediary server as an authentication service; the server is unaware in which branch the execution is, but in each case it can execute its step uniformly by checking a MAC from Alice and signing the resulting message.

The semantics is formulated as a translation to local behaviors, i.e., a process for each role of the protocol. In general, this involves algebraic problems that are not recursively computable (since, e.g., whether two terms are equal under a set of algebraic equations is in general undecidable). The second main contribution is to give a computable translation for an algebraic theory that includes standard constructors and destructors as well as exponentiation (for Diffie-Hellman).

As a third contribution, we connect our translator with ProVerif and demonstrate the effectiveness of our approach with several case studies. A particular challenge is that ProVerif's abstraction often is not precise enough when the long-term memory induces non-monotonic behavior (e.g., when a certain action is possible only until a change of the memory state) and thus fails to verify a protocol. We have developed several heuristics to make sound encodings in ProVerif that often overcome these problems.

The rest of this paper is organized as follows: in Section II we define the syntax of CryptoChoreo and give example; in Section III we define the semantics for an arbitrary algebraic theory; in Section IV, we describe the practical implementation for a representative algebraic theory; in Section V we describe the connection to ProVerif and our case studies; and we conclude in Section VI.

II. Choreography Language

Let us start with the example choreography in Fig. 1. We discuss later the front-matter declarations (e.g., types and initial knowledge) that is needed for a full specification. Role $\mathcal{A}$ is starting the choreography and first makes a non-deterministic choice ($+$) about which of the two sub-choreographies to run. In the first case, A generates a new random M (representing a message to send to B) together with a constant tag msg (indicating that this is a message transmission), then asymmetrically encrypts it with the encryption key $\mathit{ek}(\mathsf{B})$, highlighted in blue. A then symmetrically encrypts the blue message and the name of B with a shared key $\mathit{shk}(\mathsf{A}, \mathsf{s})$ with the (trusted) server s. Suppose only B knows the decryption key $\mathit{inv}(\mathit{ek}(\mathsf{B}))$, then s can only decrypt the outer symmetric encryption, but not the blue message. The next step is that s signs the blue message and the sender name A using the private signature key $\mathit{inv}(\mathit{sk}(\mathsf{s}))$; the idea is that s is vouching that the blue message really came from A (as the symmetric encryption guarantees to s) and B can verify the signature knowing the corresponding public signing key $\mathit{sk}(\mathsf{s})$. Finally, we have an authentication goal when B receives this message: namely, that A has indeed intended to send the message M to B. Non-injective (ni in $\mathsf{ni\text{-}authenticates}$) here means that B has no freshness guarantee (the message may be a replay). The other sub-choreography is very similar, except that here A sends a different kind of message, a key update with a fresh key K with a different goal: K is secret between A and B (of course both sub-choreographies could have a secrecy and an authentication goal).

Each of the two sub-choreographies could be specified in existing formal Alice-and-Bob languages, but not the non-deterministic choice. Note that s here does not realistically know which sub-choreography was chosen by A. The semantics we give below sorts this out correctly: the server shall do the same operations in both choreographies and simply handle the blue message as a black box. Note that an intruder playing role A is also allowed; and this intruder may form a blue message that complies with neither sub-choreography; s will anyway accept this message if everything it can check complies with the protocol.

a) Terms: We build terms using an alphabet Σ of function symbols and a set of variables $\mathcal{V}$. We denote all function symbols with lower-case letters and all variables with upper-case letters. In the above example, e.g., scrypt, shk, inv, msg, and s are function symbols (constants are function symbols with 0 arguments), while A and M are variables. Variables mean that they can take a different value in every run of the choreography. We use sans-serif font to denote *roles* of the protocol; they can be variables like A and B or constants like s. The latter means that there is one fixed player who cannot be the intruder—an easy way to specify a trusted third party. We will discuss below the specification of function symbols and their algebraic properties. We also use the notation $(t_1, \ldots, t_n)$ for a concatenation using a pair operator.

b) Syntax: The formal syntax of a choreography is shown in Fig. 2. A choreography $\mathbf{0}$ represents a terminated protocol, in which each participant has terminated. We omit trailing $\mathbf{0}$s when this is clear from the context. An interaction $\mathsf{A} \to \mathsf{B} : t.\mathcal{C}$ denotes a protocol where A sends a term t to B and then continues with choreography $\mathcal{C}$. The next items

$$
\begin{array}{rcll}
\mathcal{C} & ::= & \mathbf{0} & \text{(end)}\\
& | & \mathsf{A} \to \mathsf{B} : t.\mathcal{C} & \text{(interaction)}\\
& | & t \ \mathsf{secret\ between}\ \mathsf{A}_1, \dots, \mathsf{A}_n.\mathcal{C} & \text{(secrecy goal)}\\
& | & \mathsf{B\ authenticates\ A\ on}\ t.\mathcal{C} & \text{(inj. auth. goal)}\\
& | & \mathsf{B\ ni\text{-}authenticates\ A\ on}\ t.\mathcal{C} & \text{(non-inj. auth. goal)}\\
& | & \mathsf{A} : \mathcal{A} & \text{(atomic)}\\
\\
\mathcal{A} & ::= & \nu N.\, \mathcal{A} & \text{(new)}\\
& | & \mathcal{A}_1 + \mathcal{A}_2 & \text{(choice)}\\
& | & \mathsf{if}\ s \doteq t\ \mathsf{then}\ \mathcal{A}_1\ \mathsf{else}\ \mathcal{A}_2 & \text{(condition)}\\
& | & s := c[t].\mathcal{A} & \text{(memory read)}\\
& | & c[t] := s.\mathcal{A} & \text{(memory write)}\\
& | & \mathcal{C} & \text{(choreography)}
\end{array}
$$

Fig. 2. Syntax of CryptoChoreo.

$$
\begin{array}{ll}
\mathsf{A}: & EKB := \mathit{keys}[\mathsf{B}].\\
& \mathsf{if}\ EKB \doteq \mathit{blank}\ \mathsf{then}\\
& \quad \nu N.\, \mathsf{A} \to \mathsf{s} : (\mathit{key}, \mathsf{B}, N)\\
& \quad \mathsf{s} \to \mathsf{A} : \mathit{sign}((\mathit{key}, \mathsf{B}, \mathit{ek}(\mathsf{B}), N), \mathit{inv}(\mathit{sk}(\mathsf{s})))\\
& \quad A : \mathit{keys}[\mathsf{B}] := \mathit{ek}(\mathsf{B})\\
& \mathsf{else} \dots \text{(previous example with } \mathit{ek}(\mathsf{B}) \mapsto EKB)
\end{array}
$$

Fig. 3. Extension of the example from Fig. 1.

represent the specification of goals (injective and non-injective authentication, and secrecy) that we discuss later in detail.

All these constructs are present in existing formal Alice-and-Bob languages. What CryptoChoreo is adding are constructs that are all local to one role. Denote this by giving a role name A followed by a colon and an *atomic section* $\mathcal{A}$ of steps that A locally executes. Here, we have the fresh generation of a random value νN (as is standard). Next, we have the non-deterministic choice $\mathcal{A}_1 + \mathcal{A}_2$ (this is actually an internal choice for the role who runs this atomic section and an external choice for all others). Then, we have a conditional where the condition is a comparison of terms. Last, we have reading from, and writing to, long-term memory. We denote with $c[t]$ a memory cell in a family of memory cells c, where c is an identifier and t is an index term. On memory read and write, there will be no race conditions with other parallel sessions, because our semantics will treat each section $\mathcal{A}$ as atomic like the name suggests.

c) Memory Cell Example: To illustrate memory, consider the following augmentation shown in 3 of the example in Fig. 1, where A may not know the encryption key of every B (but just the public signature verification key $\mathit{sk}(\mathsf{s})$ of s). When A wants to talk to B, she checks her memory cell $\mathit{keys}[\mathsf{B}]$; if this returns the initial value blank, then she does not know the key of B and asks s, which we assume knows all public encryption keys via the ek function and can vouch for it with its signature. A checks the signature and stores the key (note that for A the term $\mathit{ek}(\mathsf{B})$ is just a blob that she cannot verify itself; this will be shown in the semantics below). Note that we do not have a repetition operator, and this example shows why this is without loss of generality: in the case that A does not know the key of B, the run of the choreography ends with writing the key (that she received from s) into her memory. Thus, in any later run between the same A and B, A will retrieve the key from her memory and run the standard protocol with it. In other words, unbounded repetition is implicit, because a choreography can be executed any number of times (also in parallel) with arbitrary instances of the (non-constant) roles and information between different runs can be transfered using memory.

d) Front Matter: The definition of our choreography language is parameterized over sets Σ and $\mathcal{V}$ (respectively, function symbols and variables) and a set of equations E (over $\Sigma, \mathcal{V}$-terms). The set E induces a congruence relation $=_E$ on terms. In the implementation of our translation, in Section IV, we instantiate E with a concrete choice of properties.

Some variables and functions symbols are declared as roles (set in sans-serif in this paper) and only these can be used in places where the syntax indicates sans-serif font. For each role, one must declare the *initial knowledge*: a list of terms where all occurring variables are of type role. For our example (in the augmented version where A does not initially know B's public key), this declaration could be:

$$
\begin{array}{ll}
\mathsf{A}: & \mathsf{A}, \mathsf{B}, \mathsf{s}, \mathit{sk}(\mathsf{s}), \mathit{shk}(\mathsf{A}, \mathsf{s})\\
\mathsf{s}: & \mathsf{A}, \mathsf{B}, \mathsf{s}, \mathit{sk}(\mathsf{s}), \mathit{inv}(\mathit{sk}(\mathsf{s})), \mathit{shk}(\mathsf{A}, \mathsf{s}), \mathit{ek}(\mathsf{B})\\
\mathsf{B}: & \mathsf{A}, \mathsf{B}, \mathsf{s}, \mathit{sk}(\mathsf{s}), \mathit{ek}(\mathsf{B}), \mathit{inv}(\mathit{ek}(\mathsf{B}))
\end{array}
$$

Note that with less knowledge the protocol would not be executable (neither for the initial version of the example where A cannot ask the server for B's public key).

We require that for every variables that is *not* of type role, the first occurrence is either in a new statement (like νX) or in a memory read (like $X := c[s]$). Also, in a new statement νX, we assume that X did not occur before in the choreography (this can be achieved by renaming). In contrast, variables in a memory read may have occurred before, e.g., $\nu X. \dots X := c[s_1]. \dots X := c[s_2]$ is legal, and it would mean that the value retrieved here is the same value as before—in an ideal unattacked execution of the choreography. Our semantics can tell if the respective role has the necessary knowledge to check that and insert such a check in the code, if so.

III. Translation Semantics

The semantics of CryptoChoreo is now defined by a translation to a process calculus, where we define for each role of the choreography a process, representing the local behavior of this role in one execution of the choreography. We then allow arbitrary instances of all roles to run in parallel, together with an intruder who can also play any of the roles (except trusted third parties) as a normal participant (but who does not necessarily follow the protocol).

A. Local Behaviors

It is convenient for the translation and the later connection to ProVerif to define a restricted syntax and semantics for local behaviors as the target language of the translation semantics.

$$
\begin{array}{rcll}
\mathcal{L} & ::= & \mathbf{0} & \text{(end)} \\
 & \mid & \mathsf{send}(r).\mathcal{L}_i & \text{(send)} \\
 & \mid & \mathsf{receive}(\mathcal{X}).\mathcal{L} & \text{(receive)} \\
 & \mid & \mathsf{lock}.\mathcal{A} & \text{(atomic local)} \\
 & & & \\
\mathcal{A} & ::= & \nu\mathcal{X}.\,\mathcal{L} & \text{(new)} \\
 & \mid & \mathcal{A}_1 + \mathcal{A}_2 & \text{choice} \\
 & \mid & \mathsf{if}\ r_1 \doteq r_2\ \mathsf{then}\ \mathcal{A}_1\ \mathsf{else}\ \mathcal{A}_2 & \text{(cond)} \\
 & \mid & \mathcal{X} := c[r_1].\mathcal{A} & \text{(mem read)} \\
 & \mid & c[r_1] := r_2.\mathcal{A} & \text{(mem write)} \\
 & \mid & \mathsf{event}(r).\mathcal{A} & \text{(event)} \\
 & \mid & \mathsf{unlock}.\mathcal{L} & \text{(end)}
\end{array}
$$

Fig. 4. Syntax of Local Behaviors.

a) Syntax: The syntax of local behaviors mirrors that of CryptoChoreo from the point of view of a single role: instead of a communication step between two roles, we have now sending and receiving. We use here however a different set of symbols Σ_p which represent public functions, i.e., functions that every agent, including the intruder, can apply. This will include most of the functions from Σ like $crypt$ that represent cryptographic algorithms and as well as public constants like msg in the example above. It will not include however some functions from Σ that just describe relations in the model, but do not represent actual cryptographic algorithms like inv (which maps public keys to the corresponding private key) or ek (which maps an agent to a corresponding public key). Σ_p will also include some functions that are not in Σ: observe that we have not used any functions for decryption or signature verification, because they are *destructors* or *verifiers*, i.e., functions that extract a subterm or verify the structure of a term; while the choreography is only concerned with *constructed* messages. Also, we will use a distinct set of variables called *labels*. Labels are denoted $\mathcal{X}_1, \mathcal{X}_2, \dots$ and are disjoint from Σ, Σ_p, and $\mathcal{V}$. The terms built from these variables and Σ_p are called *recipes* and we denote them with r, r_1, r_2, etc.

Fig. 4 shows the syntax of local behaviors. Note that, in order to enter the atomic section, it is necessary to make a lock step and it has to end with an unlock step; the semantics of local behaviors use that as a mutual exclusion mechanism on the memory to prevent race conditions (even if one ensures by design that each memory cell belongs to a particular agent, there may be more than one run of the choreography in parallel). The other constructs are similar to choreographies. However, instead of variables like M we have now labels like $\mathcal{X}$, and instead of terms (over Σ and $\mathcal{V}$) we have recipes. Note that memory read and receiving can only "read into" a label $\mathcal{X}$ (while on the choreography level, these can be composed terms). We require that at each receive and memory read, we use a new label (that did not occur before in the local behavior; this can be easily achieved by renaming).

b) Frames: To capture the knowledge of an honest agent at a state of protocol execution, we define a *frame* to be a finite mapping $F = [\mathcal{X}_1 \mapsto t_1, \dots, \mathcal{X}_n \mapsto t_n]$ where the X_i are labels and the t_i are terms. We call $\{\mathcal{X}_1, \dots, \mathcal{X}_n\}$ the *domain* of F and we say F is *concrete* if the t_i contain no variables. The semantics of local behaviors will only use concrete frames. Given a recipe r, we use a frame F like a substitution and write $F(r)$ for the term that results from replacing the labels $\mathcal{X}_i$ with the respective term t_i; $F(r)$ is undefined if r contains labels outside the domain of F.

To each role A, we attach an initial knowledge frame F_{A} that is not necessarily concrete but contains only variables of type role. In the translation from CryptoChoreo to local behaviors, we take the initial knowledge of each role $\mathsf{A} : t_1, \dots, t_n$ at the CryptoChoreo level and turn into an initial knowledge frame $F_{\mathsf{A}} = [\mathcal{X}_1 \mapsto t_1, \dots, \mathcal{X}_n \mapsto t_n]$ for the local behavior.

We require that all labels in a local behavior first occur in the initial knowledge frame, in a new, in a receive, or in a memory read.

c) Example Local Behavior: The role A of the example of Fig. 3 will be translated by the semantics into the following local behavior:

```
$F_{\mathsf{A}} = [\mathcal{X}_1 \mapsto \mathsf{A}, \mathcal{X}_2 \mapsto \mathsf{B}, \mathcal{X}_3 \mapsto \mathsf{s}, \mathcal{X}_4 \mapsto sk(\mathsf{s}), \mathcal{X}_5 \mapsto shk(\mathsf{A}, \mathsf{s})]$
$\mathsf{lock}.\mathcal{X}_K := keys[\mathcal{X}_2].$
if $\mathcal{X}_K \doteq blank$ then
  $\nu\mathcal{X}_N.\mathsf{unlock}.\mathsf{send}((key, \mathcal{X}_2, \mathcal{X}_N)).$
  $\mathsf{receive}(\mathcal{X}_{kc}).\mathsf{lock}.$
  if $(vsign(\mathcal{X}_{kc}, \mathcal{X}_4) = \top)$ then
    let $(\mathcal{X}_t, \mathcal{X}_B, \mathcal{X}_{K'}, \mathcal{X}_{N'}) = open(\mathcal{X}_{kc})$ in
    if $(\mathcal{X}_t \doteq key \wedge \mathcal{X}_B = \mathcal{X}_2 \wedge \mathcal{X}_{N'} = \mathcal{X}_N)$ then
      $keys[\mathcal{X}_2] := \mathcal{X}_{K'}.\mathsf{unlock}.0$
    else unlock.0
  else unlock.0
else ... translation of the other part, using $\mathcal{X}_K$ as $ek(\mathsf{B})$
```

Here, we use some syntactic sugar: several checks can be done by one condition; and that we write let for parsing the content of a message, in this case expecting that it can be parsed into a quadruple. The function $vsign$ is supposed to be signature verification with the property $vsign(sign(m, inv(k)), k) =_E \top$ and open a destructor that yields the signed message, i.e., $open(sign(m, inv(k))) = m$. This models a signature scheme where the signed text is transmitted in plain along with a signed hash, i.e., one needs the public key only for signature verification. One can also observe that between each lock and unlock there is at most one memory read or write, so locking is in this case actually redundant as it does not prevent any race conditions.

d) Semantics of Local Behaviors: We give a simple operational semantics for a set of local behaviors $F_{\mathsf{A}_1} : \mathcal{L}_{\mathsf{A}_1}, \dots, F_{\mathsf{A}_n} : \mathcal{L}_{\mathsf{A}_n}$ (where each F_{A_i} is the initial knowledge frame of A_i). We assume a set $Ag \subseteq \Sigma \cap \Sigma_p$ of public constant of type role and that $\mathsf{i} \in Ag$ represents a dishonest agent ("intruder") while all other agents are honest.[2]

For a behaviour $F_{\mathsf{A}} : \mathcal{L}_{\mathsf{A}}$ we call the substitution σ an *instantiation* if it maps all variables in F_{A} (that are by definition of type role) to elements of Ag. We say it is a *dishonest instantiation* if $\sigma(\mathsf{A}) = \mathsf{i}$ and an *honest instantiation*

[2]One may well consider more than one dishonest agent, but for simplicity we work with just one.

$(0 \uplus L, F, \mu) \rightarrow (L, F, \mu)$
$(\mathsf{send}(r).\mathcal{L} \uplus L, F, \mu) \rightarrow (\mathcal{L} \uplus L, F[\mathcal{X} \mapsto r], \mu)$
where $\mathcal{X}$ is a fresh label
$(\mathsf{receive}(\mathcal{X}).\mathcal{L} \uplus L, F, \mu) \rightarrow (\mathcal{L}[\mathcal{X} \mapsto F(r)] \cup L, F, \mu)$
where r is a recipe over the domain of F
$(\mathsf{lock}.\mathcal{A} \uplus L, F, \mu) \xrightarrow{tr} (\mathcal{L} \uplus L, F, \mu')$
if $(\mathcal{A}, \mu) \overset{tr}{\Rightarrow}{}^{*} (\mathsf{unlock}.\mathcal{L}, \mu')$
$(L, F, \mu) \rightarrow (\sigma(F_{\mathsf{A}})(\mathcal{L}_{\mathsf{A}}) \uplus L, F, \mu)$
if σ is a an honest instantiation of a role $F_{\mathsf{A}} : \mathcal{L}_{\mathsf{A}}$,
$(L, F, \mu) \rightarrow (L, F \uplus \sigma(F_{\mathsf{A}}), \mu)$
if σ is a dishonest instantiation of a role $F_{\mathsf{A}} : \mathcal{L}_{\mathsf{A}}$
and where the labels of F_{A} have been freshly renamed

$(\nu\mathcal{X}.\, \mathcal{A}, \mu) \Rightarrow (\mathcal{A}[\mathcal{X} \mapsto n], \mu)$ where n is a fresh constant
$(\mathcal{A}_1 + \mathcal{A}_2, \mu) \Rightarrow (\mathcal{A}_i, \mu)$ if $i \in \{1, 2\}$
$(\mathsf{if}\ r_1 \doteq r_2\ \mathsf{then}\ \mathcal{A}_1\ \mathsf{else}\ \mathcal{A}_2, \mu) \Rightarrow (\mathcal{A}_1, \mu)$ if $r_1 =_E r_2$
$(\mathsf{if}\ r_1 \doteq r_2\ \mathsf{then}\ \mathcal{A}_1\ \mathsf{else}\ \mathcal{A}_2, \mu) \Rightarrow (\mathcal{A}_2, \mu)$ if $r_1 \neq_E r_2$
$(\mathcal{X} := c[r_1].\mathcal{A}, \mu) \Rightarrow (\mathcal{A}[\mathcal{X} \mapsto \mu(c, r_1)], \mu)$
$(c[r_1] := r_2.\mathcal{A}, \mu) \Rightarrow (\mathcal{A}, \mu[(c, r_1) \mapsto r_2])$
$(\mathsf{event}(r).\mathcal{A}, \mu) \overset{r}{\Rightarrow} (\mathcal{A}, \mu)$

Fig. 5. Semantics of Local Behaviors.

otherwise. We write $\sigma(F_{\mathsf{A}})$ for the instantiation of the initial knowledge frame and $\sigma(F_{\mathsf{A}}(\mathcal{L}_{\mathsf{A}})$ for the instantiation of the behavior itself, replacing all labels from F_{A} in $\mathcal{L}_{\mathsf{A}}$ by ground terms; thus all remaining labels first occur at a new, at a receive, or at a memory read.

The last ingredient for the semantics is a memory map μ that maps every memory cell $c[(t)_E]$ gives a value, initially $blank$, where $(t)_E$ is the $=_E$-equivalence class of a ground term t over Σ. As easy notation we just write $\mu(c, t)$ for this value, and we write $\mu[(c, t) \mapsto t']$ for changing the memory cell $c[(t)_E]$ to value t'.

The semantics of local behaviour is defined in Fig. 5 and consists of two transition relations $\rightarrow$ and $\Rightarrow$ that call each other: $\rightarrow$ is on triples (L, F, μ) where L is a multi-set of local behaviors, F is a frame representing the intruder knowledge and μ is the memory map; the initial state is $(\emptyset, [], \mu_0)$ where μ_0 maps everything to $blank$; $\Rightarrow$ is on tuples $(\mathcal{A}, \mu)$ where $\mathcal{A}$ is an atomic section of a local behavior. We decorate the transition relations with a list of events that occurred upon the transitions. In this semantics, the intruder *is* the network: every message an honest agent sends gets added into the intruder knowledge, and every message an honest agent receives comes from the intruder knowledge: the intruder can choose any recipe over their knowledge, which includes encrypting and decrypting with known keys. An atomic section is handled literally atomically using the $\Rightarrow$ relation locally at an agent until it hits the unlock; we label the transition with the trace tr of all events that the agent emitted. The next rule allows to spawn new instances any role A: we choose any instantiation σ of the variables in F_{A} with agent names; if σ is honest, i.e., $\sigma(\mathsf{A}) \neq i$, we apply the instantiated knowledge $\sigma(F_{\mathsf{A}})$ as a substitution to the local behavior $\mathcal{L}_{\mathsf{A}}$, leaving only labels that are introduced by new, receive, and memory read. This semantics allows running an arbitrary number of sessions in parallel and sequentially. If σ is dishonest, i.e., $\sigma(\mathsf{A}) = i$ then this represents that the intruder plays role A under the actual name i. This models a dishonest/compromised agent. We give the intruder the initial knowledge needed to play the role, i.e., $\sigma(\mathcal{F}_{\mathsf{A}})$ where we have to rename the labels in the frame to avoid a clash with labels in the present intruder knowledge.

B. Projection: The Semantics of CryptoChoreo

We can now give the semantics of CryptoChoreo by translation to local behaviors. We again use frames to represent the knowledge of a role at a given point in the translation, but this differs from their use in the local behavior semantics. As said, the messages in a choreography represent how messages look like in an "ideal" or unattacked run—which may differ from the shape of messages in a real run due to interference from the attacker. For instance if A is supposed to receive $\exp(g, Y)$ from B for a secret Y that A does not know, there is nothing that A can check about this message. To keep track of this during translation we make an entry $[\mathcal{X}_i \mapsto \exp(g, Y)]$ in the frame F of A that expresses: A has received *some* message $\mathcal{X}_i$ and according to the choreography it is *supposed to be* $\exp(g, Y)$. Given that another entry $[\mathcal{X}_j \mapsto X]$ represents the fresh value A has created for her own secret X, then the Diffie-Hellman key $\exp(\exp(g, X), Y)$ can be formed with the recipe $r = \exp(\mathcal{X}_i, \mathcal{X}_j)$: $F(r) = \exp(\exp(g, Y), X) =_E \exp(\exp(g, X), Y)$. In this way, frames make the connection between the messages the agents have and use in their local behavior and what the messages supposedly are on the choreography level. We will use this to figure out how agents generate outgoing messages and how they analyze incoming messages.

We thus define two core algorithmic problems on frames:

- The deduction problem: Given frame F and term t, compute a recipe r such that $F(r) =_E t$ if one exists or return fail otherwise.
- The complete check problem: Given a frame F, compute a finite set of *checks*, i.e., equations of recipes $\phi = \{r_1 \doteq r'_1, \ldots, r_n \doteq r'_n\}$ such that $F(r_i) =_E F(r'_i)$ for each $1 \leq i \leq n$, that is *complete* in the sense that if for any other r_0, r'_0 we have $F(r_0) =_E F(r'_0)$, then $\phi \models_E r_0 \doteq r'_0$, or return fail if no finite set of checks satisfies that. [3]

Both these problems are in general not recursively computable (because in general even $=_E$ is undecidable), but in Section IV we give algorithms for both problems for a standard set E of equations.

We first note a complete set of checks for F is not unique, however if ϕ and ψ are two complete set of checks for F, then $\phi \models_E \psi$ and $\psi \models_E \phi$, so they are equivalent and in the semantics we can leave this choice undetermined[4] without

[3]We use standard logical notation: an interpretation $\mathcal{I}$ maps all the variables to terms; define $\mathcal{I} \models_E s \doteq t$ if $\mathcal{I}(s) =_E \mathcal{I}(t)$; we extend this definition to sets of equations; $\phi \models_E \psi$ if for all $\mathcal{I} \models_E \phi$ also $\mathcal{I} \models_E \psi$.

[4]Thus a concrete implementation is free to choose one.

making the semantics ambiguous. By abuse of notation we thus write $\phi(F)$ for a complete set of checks for F, even though it is strictly speaking not a function.

Second, also the deduction problem has in general many solutions, i.e., different r_1 and r_2 such that $F(r_1) =_E F(r_2) =_E t$. (This does generally not imply that $r_1 =_E r_2$: we may have different ways in knowledge F to produce a particular term t.) However, in this case $\phi(F) \models_E r_1 \doteq r_2$, i.e., if we have performed all the checks in $\phi(F)$, then also the choice between the two recipes r_1 and r_2 does not matter.

Third, in the semantics, we need a slight generalization of the deduction problems, namely given several frames $F_1, \ldots, F_n$ and goal terms $t_1, \ldots, t_n$ and we want a *single* recipe r that solves all *deduction* problems, i.e., $F_i(r) = t_i$ for every $1 \leq i \leq n$. Suppose we already have a set ϕ of checks that is a complete set of checks for each of the F_i, and suppose there is a solution r for all frames. Then any solution r' for one of the frames, say F_1, must be equivalent to r, i.e., $\phi \models_E r \doteq r'$. Thus for checked frames it suffices to compute a solution for one frame and check if it works on the other frames—if not, then there is no common solution for all frames.

Nondeterminism and conditions mean that, in general, a role does not know which branch of the choreography we currently are in, and this also holds during the translation. Therefore, during in the translation, the translation state contains a finite set of pairs $(F_i : \mathcal{C}_i)$ where each F_i is a frame (all F_i have the same domain) and $\mathcal{C}_i$ is the remainder of the choreography that still needs to be translated.

Definition 1. *A* translation state *is of the form*

$$(\mathsf{A}, \phi \rhd \psi, b, \{(F_1 : \mathcal{C}_1), \ldots, (F_n : \mathcal{C}_n)\})$$

where A *is the role we are currently translating;* ϕ *and* ψ *are a set of equations* $r_1 \doteq r_2$ *between recipes, where* ϕ *represents checks that have already been done, and* ψ *are checks that are pending;* $b \in \{\mathsf{c}, \mathsf{a}\}$ *is a flag indicating whether we are on the choreography level or in an atomic section; the* F_i *are frames with the same domain that map to terms; and the* $\mathcal{C}_i$ *are either choreographies if* $b = \mathsf{c}$ *or atomic sections if* $b = \mathsf{a}$.

During translation we preserve the invariant *that* $\phi \cup \psi$ *is a complete set of checks for the* $F_1, \ldots, F_n$ *where* ψ *may contain checks that do not hold on all* F_i*. If* $\psi \neq 0$*, i.e., if there are pending checks, they will be performed first before all other translation steps.*

Given a choreography $\mathcal{C}$ *and a role* A *of that choreography and a frame* F_A *that contains the initial knowledge of* A *in the choreography specification where each item has received a unique label* $\mathcal{X}_i$*, the* initial translation state *for translating* A *in* $\mathcal{C}$ *is:*

$$(\mathsf{A}, \emptyset \rhd \emptyset, \mathsf{c}, \{(F_A : \mathcal{C})\})$$

i.e., there is just one possibility $\mathcal{C}$ *where we are and the current knowledge is* F_A.

1) Cases of the Semantics Function: The semantics function $[\![T]\!]$ takes a translation state T and projects the choreography to the actions of the role, yielding a local behavior for that role. We define it recursively by a case distinction on T. Since we will often require all possibilities $(F_i, \mathcal{C}_i)$ to start with the same kind of command, we use the following notation: $\{F_i, \nu N_i.\mathcal{C}_i\}_{i=1}^n$ for $\{F_1, \nu N_1.\mathcal{C}_1, \ldots, F_n, \nu N_n, \mathcal{C}_n\}$, and similar for other constructs in place of νN. For simplicity, we first present this semantics without goals.

a) $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{(F_i : \mathbf{0})\}_{i=0}^n]\!]$ *where* $n > 0$*:* All possibilities have finished, and the translation is simply: $\mathbf{0}$.

b) $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{(F : \mathsf{B} \to \mathsf{C}.\mathcal{C})\} \cup \{F_i : \mathcal{C}_i\}_{i=1}^n]\!]$ *for* $\mathsf{A} \neq \mathsf{B}$ *and* $\mathsf{A} \neq \mathsf{C}$*:* One of the possibilities is a communication step that A is not involved in and is therefore ignored. The translation is thus:
$[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{(F : \mathcal{C})\} \cup \{F_i : \mathcal{C}_i\}_{i=1}^n]\!]$

c) $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{F_i : \mathsf{A} \to \mathsf{B}_i : t_i.\mathcal{C}_i\}_{i=1}^n]\!]$ *where* $n > 0$*:* All possibilities are send steps for A.

As explained before, we check whether there is a recipe r such that $F_i(r) = t_i$ for each $1 \leq i \leq n$. If there is no such r, then we reject the protocol as unexecutable: either there is no way for A to produce the outgoing term t_i, or the different possibilities would require different recipes, and A cannot know in which possibility they are. However, if there is such an r,[5] then the translation is:
$\mathsf{send}(r).[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{F_i : \mathcal{C}_i\}_{i=1}^n]\!]$

d) $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{F_i : \mathsf{B}_i \to \mathsf{A} : t_i.\mathcal{C}_i\}_{i=1}^n]\!]$ *where* $n > 0$*:* All possibilities are receive steps for A.

Let $\mathcal{X}$ be a new recipe variable and $F'_i = F_i[\mathcal{X} \mapsto t_i]$ for every $i \in \{1, \ldots, n\}$. Let ϕ_i be a complete finite set of checks for F'_i and let $\Phi = \cup_{i=1}^n \phi_i$. This represents all checks that we can do in any of the frames F_i. First we can remove from Φ all those checks that are already implied by the checks ϕ from the translation state (i.e., that have already been done before in the translation process). We can also remove from Φ any equation that is implied by the other equations. Let thus Φ' be a resulting minimal set of equations.[6] The translation of the receive step is then obtained by adding received message to the frames and inserting the Φ' as pending checks that have to be done next:
$\mathsf{receive}(\mathcal{X}).[\![\mathsf{A}, \phi \rhd \Phi', \mathsf{c}, \{F'_i : \mathcal{C}_i\}_{i=1}^n]\!]$

e) $[\![\mathsf{A}, \phi \rhd \{r_1 \doteq r_2\} \cup \psi, b, \{F_i : \mathcal{C}_i\}_{i=1}^n]\!]$ *where* $n > 0$*:* There is at least one pending check $r_1 \doteq r_2$.

We partition the possibilities into those where F_i satisfies the check and those that do not:
Let $\mathsf{FCs}_+ = \{(F_i : C_i) \mid F_i(r_1) =_E F_i(r_2)\}$
and $\mathsf{FCs}_- = \{(F_i : C_i) \mid F_i(r_1) \neq_E F_i(r_2)\}$.
The translation is now:
$\mathsf{if}\ r_1 \doteq r_2\ \mathsf{then}\ [\![\mathsf{A}, \phi \cup \{r_1 \doteq r_2\} \rhd \psi, b, \mathsf{FCs}_+]\!]$
$\qquad \mathsf{else}\ [\![\mathsf{A}, \phi \rhd \psi, b, \mathsf{FCs}_-]\!]$

f) $[\![\_, \_ \rhd \_, \_, \emptyset]\!]$ *:* There are no possible continuations.

This can happen when doing a check that splits the possibilities into FCs_+ and FCs_-, and one of them is empty. It

[5]If there are several such recipes, the choice between them leads to equivalent translation outcomes as explained before.

[6]Again, there may be several minimal sets, e.g., if two equations imply each other; however all resulting sets from the minimization are logically equivalent.

means that if we reach this branch, the agent has detected that an incoming message is not compliant with the choreography, and aborts the execution. The translation is thus simply: **0**

g) $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{F_i : \mathsf{A} : \mathcal{A}_i\}_{i=1}^n]\!]$ *where* $n > 0$*:* All possibilities start with an atomic section of the agent A.

Then the translation is simply to issue the lock and switch the atomic section flag:
$\mathsf{lock}.[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{a}, \{F_i : \mathcal{A}_i\}_{i=1}^n]\!]$

h) $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{(F : \mathsf{B} : \mathcal{A})\} \cup \{F_i : \mathcal{C}_i\}_{i=1}^n]\!]$ *where* $\mathsf{B} \neq \mathsf{A}$*:* One possibility is that another role goes into its atomic section.

Role A should ignore these steps and just extract all continuations after the atomic section, which is defined as follows:

$$\begin{array}{rcl} cont(\mathcal{A}_1 + \mathcal{A}_2) & = & cont(\mathcal{A}_1) \cup cont(\mathcal{A}_2) \\ cont(\mathsf{if}\ s \doteq t\ \mathsf{then}\ \mathcal{A}_1\ \mathsf{else}\ \mathcal{A}_2) & = & cont(\mathcal{A}_1) \cup cont(\mathcal{A}_2) \\ cont(\_.\mathcal{A}) & = & cont(\mathcal{A}) \\ cont(\mathcal{C}) & = & \{\mathcal{C}\} \end{array}$$

Let $\mathcal{C}_1, \ldots, \mathcal{C}_m = cont(\mathcal{A})$ in the translation:
$[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{F : \mathcal{C}_i\}_{i=1}^m \cup \{F_i : \mathcal{C}_i\}_{i=1}^n]\!]$

We now come to the cases for an atomic section of the agent we translate for:

i) $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{a}, \{F_i : \nu N_i.\mathcal{A}_i\}_{i=1}^n]\!]$ *for* $n > 0$*:* All possibilities create a fresh number N_i.

We pick a fresh label $\mathcal{X}$ and translate:
$\nu\mathcal{X}.\, [\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{a}, \{F_i[\mathcal{X} \mapsto N_i] : \mathcal{A}_i\}_{i=1}^n]\!]$

j) $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{a}, \{F_i : s_i := c[t_i].\mathcal{A}_i\}_{i=1}^n]\!]$ *where* $n > 0$*:* All cases start with a memory read.

Similar to the send case, we require that there is one recipe r such that $F_i(r) =_E t_i$ for each $1 \leq i \leq n$. If not, the semantics rejects the protocol as unexecutable at this point (because the agent either cannot create the proper index for the memory lookup, or there are contradicting possibilities for this index). The retrieved message s_i is treated like in the receive case: we add it to the knowledge with a new label $\mathcal{X}$, giving frames $F_i' = F_i[\mathcal{X} \mapsto s_i]$, and then we compute a complete set of checks ϕ_i for each F_i', compute the union $\Phi = \cup_{i=1}^n \phi_i$, and remove redundant equations leading to a reduced Φ'. The translation is then:
$\mathcal{X} := c[r].[\![\mathsf{A}, \phi \rhd \Phi', \mathsf{a}, \{F_i' : \mathcal{A}_i\}_{i=1}^n]\!]$

k) $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{a}, \{F_i : c[t_i] := s_i.\mathcal{A}_i\}_{i=1}^n]\!]$ *where* $n > 0$*:* All possibilities start with a write step. We require that there are recipes r_1 and r_2 such that $F_i(r_1) = t_i$ and $F_i(r_2) = s_i$ for each $i \in \{1, \ldots, n\}$. (If not, this is a specification error, because it is unclear what A's next step is.) Then the translation is:
$c[r_1] := r_2.[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{a}, \{F_i : \mathcal{A}_i\}_{i=1}^n]\!]$

l) $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{a}, \{F_i : \mathsf{if}\ s_i \doteq t_i\ \mathsf{then}\ \mathcal{A}_i^+\ \mathsf{else}\ \mathcal{A}_i^-\}_{i=1}^n]\!]$ *where* $n > 0$*:* All the $\mathcal{A}_i$ start with a condition. We require that there are recipes r_1 and r_2 such that $F_i(r_1) = t_i$ and $F_i(r_2) = s_i$ for each $i \in \{1, \ldots, n\}$. (If not, this is a specification error, because it is unclear what A's next step is.) We define
$\mathcal{T}^+ = [\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{a}, \{F_i : \mathcal{A}_i^+\}_{i=1}^n]\!]$ and
$\mathcal{T}^- = [\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{a}, \{F_i : \mathcal{A}_i^-\}_{i=1}^n]\!]$.
The translation is:
$\mathsf{if}\ r_1 \doteq r_2\ \mathsf{then}\ \mathcal{T}^+\ \mathsf{else}\ \mathcal{T}^-$

m) $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{a}, \{F_i : \mathcal{C}_i\}_{i=1}^n]\!]$ *where* $n > 0$*:* Finally, if all the $\mathcal{A}_i$ conclude the atomic section, then the translation is:
$\mathsf{unlock}.[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{F_i : \mathcal{C}_i\}_{i=1}^n]\!]$

n) $[\![\_]\!]$ *for any other translation state:* this is an error, because it is unclear what A should do next.

2) Example: Let us continue the example choreography from Fig. 3 and let us look at the translation for the role s. The knowledge of s gives us the frame:

$$\begin{array}{l} F_0 = [\mathcal{X}_1 \mapsto \mathsf{A}, \mathcal{X}_2 \mapsto \mathsf{B}, \mathcal{X}_3 \mapsto \mathsf{s}, \mathcal{X}_4 \mapsto sk(\mathsf{s}), \\ \quad \mathcal{X}_5 \mapsto inv(sk(\mathsf{s})), \mathcal{X}_6 \mapsto shk(\mathsf{A}, \mathsf{s}), \mathcal{X}_7 \mapsto ek(\mathsf{B})] \end{array}$$

and we compute $[\![\mathsf{s}, \emptyset \rhd \emptyset, \mathsf{c}, \{F_0 : \mathcal{C}\}]\!]$ where $\mathcal{C}$ is the entire choreography. The choreography begins with atomic actions of A, checking if the key of B is known, asking s if not and starting the main protocol otherwise. Thus using rule III-B1h we get a split into two possibilities $\{(F_0 : \mathsf{A} \to \mathsf{s} : (key, \mathsf{B}, N)\ldots), (F_0 : \mathcal{C}_0)\}$ where $\mathcal{C}_0$ is the initial choreography example of Fig. 1 (with $ek(\mathsf{B})$ replaced by the variable EKB that represents the key that A has looked up from memory). Now $\mathcal{C}_0$ starts with another atomic section of A (the choice to either send a message or a key update). So we apply again rule III-B1h to split that possibility into two:

$$\begin{array}{l} (F_0 : \mathsf{A} \to \mathsf{s} : (key, \mathsf{B}, N)\ldots), \\ (F_0 : \mathsf{A} \to \mathsf{s} : scrypt((\mathsf{B}, crypt(\ldots, EKB)), shk(\mathsf{A}, \mathsf{s}))\ldots) \\ (F_0 : \mathsf{A} \to \mathsf{s} : scrypt((\mathsf{B}, crypt(\ldots, EKB)), shk(\mathsf{A}, \mathsf{s}))\ldots) \end{array}$$

Now finally all messages are something the server can receive, so with rule III-B1d we get updated frames F_1, F_2, F_3 augmenting F_0 with a new label $\mathcal{X}_8$, which is bound to the respective incoming message.

The checks that we can do for F_1 are that $\mathcal{X}_8$ is a triple, that the first item is constant key and the second item is $\mathcal{X}_2$.[7] For ease of notation, assume we have functions $vtuple_n$ and π_n (for all $n \in \mathbb{N}$) with the property $vtuple_n(t_1, \ldots, t_n) =_E \top$ and $\pi_i(t_1, \ldots, t_n) =_E t_i$. Thus the checks for F_1 are $\phi_1 = \{vtuple_3(\mathcal{X}_8) \doteq \top, \pi_1(\mathcal{X}_8) \doteq key, \pi_2(\mathcal{X}_8) \doteq X_2\}$.

In F_2 and F_3 we can check that symmetric decryption of $\mathcal{X}_8$ with key $\mathcal{X}_6$ succeeds and yields a pair. The blue parts in F_2 and F_3 cannot be decrypted, so s cannot further check anything about this. In our example theory E below we have operators $scrypt$, $vscrypt$ and $dscrypt$ with the properties $dscrypt(scrypt(m, k), k) =_E m$ and $vscrypt(scrypt(m, k), k) =_E \top$. Together they model AEAD symmetric schemes, i.e., the attacker cannot modify the encrypted message m by modifying the ciphertext $scrypt(m, k)$, as this would lead to errors when decrypting; thus the decryption is an operation that fails when applied to an incorrect message or the wrong key, and we model that in the algebra by two functions, one telling us whether decryption works with the given key and one that in the positive case gives the result. We thus have in F_2 and F_3 the complete set of the checks:

[7]More realistically, s should not expect a particular name B but rather have it determined through A's request. This is why we like to model $ek(\cdot)$ as a public function (one can look up the public key of any role), but here we deliberately made this function private, so that A has to ask the server for the role. Anyway the semantics ensures that for any instantiation of the role variables with agent names, we have any number of server instances, so this comes without loss of attacks.

$\phi_2 = \phi_3 = \{vcrypt(\mathcal{X}_8, \mathcal{X}_6) \doteq \top,$
$vtuple_2(dscrypt(\mathcal{X}_8, \mathcal{X}_6)) \doteq \top, \pi_1(dscrypt(\mathcal{X}_8, \mathcal{X}_6)) \doteq \mathcal{X}_2\}$.

We thus have the translation
$\mathsf{receive}(\mathcal{X}_8).[\![\mathsf{s}, \emptyset \rhd \phi_1 \cup \phi_2, \mathsf{c}, \{(F_1{:}\ldots), (F_2{:}\ldots), (F_3{:}\ldots)\}]\!]$

Applying rule III-B1e several times to process all checks, we get the possibilities partitioned (because ϕ_1 only holds in F_1 and ϕ_2 only holds in F_2 and F_3) as follows:
$\mathsf{if}\ \phi_1\ \mathsf{then}\ [\![\mathsf{s}, \phi_1 \rhd \emptyset, \mathsf{c}, \{(F_1{:}\ldots)\}]\!]\ \mathsf{else}$
$\mathsf{if}\ \phi_2\ \mathsf{then}\ [\![\mathsf{s}, \phi_1 \rhd \emptyset, \mathsf{c}, \{(F_2{:}\ldots), (F_3{:}\ldots)\}]\!]\ \mathsf{else}\ \mathbf{0}$

Let us just look at the most interesting branch, namely the positive case under ϕ_2. Here the next step in F_2 is $\mathsf{s} \to \mathsf{B}$: $sign((\mathsf{A}, crypt((msg, M), EKB)), inv(sk(\mathsf{s})))$ and in F_3 the corresponding step but with content (upd, K). So we need to apply rule III-B1c which requires a recipe r that works in both cases. Let $r_b = \pi_2(dscrypt(\mathcal{X}_8, \mathcal{X}_6))$ which gives the "blue message part" that s cannot decrypt in either F_2 or F_3. Now $r = sign((\mathcal{X}_1, r_b), \mathcal{X}_5)$ is the recipe that works in both cases. Thus the complete translation for the server role is:

$\mathsf{receive}(\mathcal{X}_8).\mathsf{if}\ \phi_1\ \mathsf{then}\ \mathsf{send}(sign((key, \mathcal{X}_2, \mathcal{X}_7)), \mathcal{X}_5)\ \mathsf{else}$
$\mathsf{if}\ \phi_2\ \mathsf{then}\ \mathsf{send}(sign((\mathcal{X}_1, \pi_2(dscrypt(\mathcal{X}_8, \mathcal{X}_6))), \mathcal{X}_5))$

C. Attack Semantics

In this section, we formalize our notion of security. In a sentence, we consider there to be an attack if the system can possibly develop in a way that falsifies a given query over traces.

A *security query* is built from the following grammar:

$$\begin{array}{rcl} p & ::= & \mathsf{event}(t) \mid \mathsf{event}(s) \sqsubseteq \mathsf{event}(t) \mid \mathsf{intruder}(t) \mid s \doteq t \\ & \mid & \bot \mid p_1 \Longrightarrow p_2 \mid \forall X.\ p' \end{array}$$

The other logical operators can be added as syntactic sugar (in particular we will use $\wedge$). We only consider a query well-formed if it contains no free variables.

We characterize a trace, $(L_0, F_0, \mu) \overset{t_1,\ldots,t_n}{\to^*} (L_m, F_m, \mu)$, by the list of emitted events and the final knowledge of the intruder: $([t_1, \ldots, t_n], F_m)$.

Following is the semantics for evaluating a query:

$([t_1, \ldots, t_n], F) \models \mathsf{event}(s)$ iff $s =_E t_i$
$([t_1, \ldots, t_n], F) \models \mathsf{intruder}(s)$ iff $F(r) =_E s$
$([t_1, \ldots, t_n], F) \models s \doteq t$ iff $s =_E t$
$([t_1, \ldots, t_n], F) \models \bot$ never
$([t_1, \ldots, t_n], F) \models p_1 \Longrightarrow p_2$
iff $([t_1, \ldots, t_n], F) \not\models p_1$
or $([t_1, \ldots, t_n], F) \models p_2$
$([t_1, \ldots, t_n], F) \models \forall X.\ p$
iff for all s we have $([t_1, \ldots, t_n], F) \models p[s/X]$
$([t_1, \ldots, t_n], F) \models \mathsf{event}(s) \sqsubseteq \mathsf{event}(t)$
iff for every $t_i =_E s$ there is a unique $t_j =_E t$

We consider a configuration as secure with regard to a given query if the query is valid on all traces from the configuration.

For the rest of this section, we will show how to encode the security goals in a choreography as queries.

First, we modify the grammar of choreographies to allow the emission of events:

$$\begin{array}{rcl} \mathcal{C} & ::= & \ldots \\ \mathcal{A} & ::= & \ldots \\ & \mid & \mathsf{event}(t).\mathcal{A} \end{array}$$

These events will be handled by the projection in the same way as sends: We check if we can find a recipe that produces t in all frames, and return a translation error if not.

To each secrecy goal g = "t secret between $\mathsf{A}_1, \ldots, \mathsf{A}_n$", we assign a unique *event name* $e_g \in \Sigma \setminus \Sigma_p$. We then replace each g with

$$\begin{array}{l} \mathsf{A}_1 : \mathsf{event}(e_g(t, \mathsf{A}_1, \ldots, \mathsf{A}_n)). \\ \quad\vdots \\ \mathsf{A}_n : \mathsf{event}(e_g(t, \mathsf{A}_1, \ldots, \mathsf{A}_n)) \end{array}$$

The goal holds iff the following query holds:

$$\forall X, \mathsf{A}_1, \ldots, \mathsf{A}_n. \neg \left(\begin{array}{l} \mathsf{A}_1 \not\doteq \mathsf{i} \wedge \ldots \wedge \mathsf{A}_n \not\doteq \mathsf{i} \wedge \\ \mathsf{event}(e_g(X, \mathsf{A}_1, \ldots, \mathsf{A}_n)) \wedge \mathsf{intruder}(X) \end{array} \right)$$

To a goal g = "B ni-authenticates A on t", we associate unique start- and end-event names $e_{gs}, e_{ge} \in \Sigma \setminus \Sigma_p$. We then replace the goal with the end event $\mathsf{B} : \mathsf{event}(e_{ge}(t, \mathsf{A}, \mathsf{B}))$. We want to check that the occurrence of this event implies the occurrence of a corresponding start event from A. However, it is not always possible to insert that start event right at the beginning of the choreography; t might contain values that have been generated during the protocol run.

To solve this, we make the following modification to the projection semantics for A:
If we are computing $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{F_i : \mathcal{C}_i\}_{i=1}^n]\!]$, B : $\mathsf{event}(e_{ge}(t, \mathsf{A}, \mathsf{B}))$ is in one of the $\mathcal{C}_i$, and the corresponding start event has not yet been generated, try the following: If there is a recipe r such that $F_1(r) = t \wedge \ldots \wedge F_n(r) = t$, return $\mathsf{event}(e_{gs}(r, \mathsf{A}, \mathsf{B})).[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{F_i : \mathcal{C}_i\}_{i=1}^n]\!]$ while remembering that the event was generated. Otherwise, continue computing $[\![\mathsf{A}, \phi \rhd \emptyset, \mathsf{c}, \{F_i : \mathcal{C}_i\}_{i=1}^n]\!]$ as normal.
The goal is enforced by the following query:

$$\forall X, \mathsf{A}, \mathsf{B}. \begin{array}{l} \mathsf{A} \not\doteq \mathsf{i} \wedge \mathsf{B} \not\doteq \mathsf{i} \wedge \mathsf{event}(e_{ge}(X, \mathsf{A}, \mathsf{B})) \\ \quad \Longrightarrow \mathsf{event}(e_{gs}(X, \mathsf{A}, \mathsf{B})) \end{array}$$

For each g = "B authenticates A on t", we do the same procedure, except that we use the query:

$$\forall X, \mathsf{A}, \mathsf{B}. \begin{array}{l} \mathsf{A} \not\doteq \mathsf{i} \wedge \mathsf{B} \not\doteq \mathsf{i} \Longrightarrow \\ \mathsf{event}(e_{ge}(X, \mathsf{A}, \mathsf{B})) \sqsubseteq \mathsf{event}(e_{gs}(X, \mathsf{A}, \mathsf{B})) \end{array}$$

IV. Mechanization of the Projection

We give now an overview of the mechanization for a representative algebraic theory; most details are found in Appendix B. To implement the projection of Section III, one must be able to do the following:

- (*word problem*) Given two terms s and t, check if $s =_E t$.
- (*recipe composition*) Given a set of frame-term pairs $\{(F_1, t_1), \ldots, (F_n, t_n)\}$, decide if there is a recipe r so $F_1(r) =_E t_1 \wedge \ldots \wedge F_n(r) =_E t_n$, and return it if so.

- (*complete set of checks*) Given a frame F calculate a complete set of checks, i.e., a finite set of checks that implies all checks that can be made at all.

In this section we will demonstrate how to do this for a particular cryptographic model with Diffie-Hellman keys, symmetric encryption, and asymmetric encryption. We assume that for all destructors we have a corresponding verifier that can be used to check whether decryption would succeed.

Recall that we are distinguishing terms, denoted s, t, as used in the choreography. They built over the alphabet Σ and variables $\mathcal{V}$, where Σ does not contain any destructors or verifiers, but may contain private functions. In contrast, on the local behavior level, we have recipes that are built over Σ_p and labels $\mathcal{X}_i$, where Σ_p contains only public function symbols, but includes destructors and verifiers.

As part of our algebraic theory we consider the following built-in symbols:

	Constructors	Destructors	Verifiers
Tuples	pair	$\mathit{fst}, \mathit{snd}$	vpair
Asymmetric Enc.	crypt	dcrypt	vcrypt
Private Keys	inv	pub_k	vinv
Symmetric Enc.	scrypt	$\mathit{dscrypt}$	$\mathit{vscrypt}$
Signatures	sign	open	vsign
Diffie-Hellman	exp	exp^{-1}	vexp

All the constructors are part of Σ and can occur in terms, and all except inv are public and thus in Σ_p and can occur in recipes. Note that we had earlier used n-tuples for simplicity, and here have only binary tuples, but this can be seen as syntactic sugar. Besides these symbols, the modeler can declare other further function symbols that can be either public (and thus both part of Σ and Σ_p), e.g., to model hash functions or public constants, or that can be private (and thus only part of Σ), e.g., to model key infrastructures or fixes secrets between agents. However, these user-defined functions cannot have any algebraic properties. We also have a public constant $\top$ both in Σ and Σ_p. We call a recipe *constructive* if it does not contain destructors or verifiers.

Definition 2. *We define our algebra $E = R \cup B$ where*
$R =$
$\{\mathit{vpair}(\mathit{pair}(x, y), \top) \doteq \top,$
$\mathit{fst}(\mathit{pair}(x, y), \top) \doteq x,\ \mathit{snd}(\mathit{pair}(x, y), \top) \doteq y,$
$\mathit{vscrypt}(\mathit{scrypt}(x, y), y) \doteq \top,$
$\mathit{dscrypt}(\mathit{scrypt}(x, y), y) \doteq x,$
$\mathit{vcrypt}(\mathit{crypt}(x, y), \mathit{inv}(y)) \doteq \top,$
$\mathit{dcrypt}(\mathit{crypt}(x, y), \mathit{inv}(y)) \doteq x,$
$\mathit{vsign}(\mathit{sign}(x, \mathit{inv}(y)), y) \doteq \top,$
$\mathit{open}(\mathit{sign}(x, \mathit{inv}(y)), y) \doteq x,$
$\mathit{vinv}(\mathit{inv}(x), \top) \doteq \top,\ \mathit{pub}_k(\mathit{inv}(x), \top) \doteq x,$
$\mathit{vexp}(\mathit{exp}(x, y), y) \doteq \top,\ \mathit{exp}^{-1}(\mathit{exp}(x, y), y) \doteq x\},$
and $B = \{\mathit{exp}(\mathit{exp}(x, y), z) \doteq \mathit{exp}(\mathit{exp}(x, z), y)\}$.

Let $=_E$ denote the congruence induced by these equations and $=_B$ the congruence induced just by the equation in B.

The functions vpair, fst, snd, vinv, and pub_k should actually be unary functions, because they do not require a key. For uniformity we have made them binary functions like all the other destructors and verifiers, and we use $\top$ as a dummy value for the key-position.

The reader may be surprised to see a verifier for Diffie-Hellman exponentiation. This is because our method below requires that every destructor has a corresponding verifier. However, the verifier vexp exists only pro-forma: if we our procedure runs into a situation where it actually employs vexp, it stops with an error. The only situation where it would be employed is, if we have an agent knows both x and a term (equivalent to) $\mathit{exp}(t, x)$, but not t, and this does not occur in standard uses of Diffie-Hellman.

A similar question may arise from the destructor and verifier for private keys. Many approaches model a public constructor that from a given private key generates a public key; we use here instead a private constructor inv to map a public key to a corresponding private key; this allows us easily model public-key infrastructures like $\mathit{ek}(\mathsf{A})$ being the public encryption of A where ek is a public function (so every agent can lookup keys). The small price to pay is that the inverse mapping from private to public key is called a destructor and that we have a verifier to check if a private key really fits with the public key.

The considered theory $E = R \cup B$ allows us to decide the word problem, i.e., for terms or recipes s, t, whether $s =_E t$. This is because R used as rewrite rules modulo B (i.e., $\rightarrow_{R/B}$) is convergent. Thus we only need to compare the normal forms of s and t modulo B. The equivalence class modulo B of any term is finite and easily computable. See Theorem 1 in the appendix.

1) Compose: We define a function $\mathit{compose}(F, t)$ that, given a frame F and a term t, obtains all constructive recipes r such that $F(r) =_B t$. Roughly, for every term t_0 in $[t]_B$ (the equivalence class of t modulo B) we can check if t_0 is a term in the frame, and additionally, if $t_0 = f(t_1, \ldots, t_n)$ for a public f (which cannot be a destructor or verifier by construction), we recursively compute $\mathit{compose}(F, t_i)$ and from the results construct the solutions for $\mathit{compose}(F, t)$ as expected.

We further introduce the notion of an *analyzed frame*, i.e., where the frame contains every term that can be obtained using a destructor on any message in the frame, using a constructive recipe for the key term. In an analyzed frame F, $\mathit{compose}$ finds a recipe for every term t that can be obtained with any recipe (Lemmas 4 and 5).

2) Analysis: We define an analysis procedure that successively applies decryption steps as long as possible: for every message that potentially can be decrypted, we check if we can compose the decryption key. If so, the resulting message is added to the frame. Whenever we add an analyzed message to the frame, we also need to check again all those messages for which we previously did not have the decryption key. We show that this terminates (Theorem 2) and produces correct analyzed frames (Theorem 3).

3) Checks: Finally, given an analyzed frame, we show how to compute a complete set of checks. In particular, it is sufficient to restrict oneself to constructive recipes for checks in an analyzed frame (Lemma 6 and Theorem 4).

Finally, the compose procedure can be extended to the recipe composition problem, i.e., given analyzed and checked $F_1, \ldots, F_n$ and goal terms $t_1, \ldots, t_n$, find a single recipe r with $F_i(r) = t_i$ for all i, because we try to get a solution for $F_1(r) = t_1$ and if it exists, then it works in all frames, because they are checked (Theorem 5).

V. Case studies

A. Exporting to ProVerif

In this section, we summarize how the output from the projection semantics of Section III can be further translated to ProVerif code for automatic verification. More details can be found in Appendix Section A. We start by unfolding the labels from the initial knowledge in each local behavior, similarly to what we do in the semantics of Figure 5. Each local behavior is almost a valid ProVerif process already, except for the use of nondeterministic choice and memory cells.

Nondeterministic choice is simple to encode: we take a message from the network (the intruder) indicating which branch to take, and then branch on the content of that message. Thus, we leave it to the intruder to pick the branch that will lead to an attack (if one exists).

Encoding memory cells is more involved. We encode each memory cell as a private channel, and ensure by construction that the channel always contains exactly one message (except when the message has been consumed in an atomic section that has not yet been left). Then, in the semantics of ProVerif, a read from the memory cell corresponds exactly to reading the term that was most recently placed in the channel.

However, when the ProVerif process is translated to Horn-clauses, certain overapproximations will mean that it is no longer guaranteed that a message on a private channel is consumed in the right order or only once. We decrease the chance of a false positive by adding a counter to each memory cell, which is incremented on every write. Adding the admissible axiom that if two values written to a memory cell are associated with the same counter value they must be equal, excludes many impossible models during the proof search.

B. Examples

We implemented a tool in Haskell that automates the projection from choreographies to local behaviors, based on the definitions in Sections III and IV. It also supports the generation of a ProVerif file from these local behaviors, following the steps described in Section V-A.

We will in the following describe one particular example, but have made more available. All our examples combined verify in a minute when exported to ProVerif. We wish to highlight the following notable examples:

- `ttp-blind-forward.choreo` is similar to the example from Section II. Here a trusted third party either helps A authenticate a new encryption key with or authenticate a request to B, without knowing which branch it is in.
- `SSO.choreo` describes a protocol where an agent A authenticates to another agent B using a trusted third party ttp. We can also verify this when the behaviour of ttp is taken from `SSO-API.choreo`, where ttp is implemented like an API responding to queries and saving their state using memory cells. Our tool includes an option for selecting the behavior of participants from different choreographies like this.
- We demonstrate in `tpm-simple.choreo` and `tpm-simple-API.choreo` the security of a TPM that can either declassify a given value or delete it. In particular, this protocol is non-monotonic, as it should be transparent to the owner of the value which choice was taken, and that if one choice is made then the other cannot be made later.
- `SSO-DH.choreo` and `SSO-DH-API.choreo` demonstrates a Diffie-Hellman exchange mediated by a trusted third party.
- `ASW.choreo` contains the example described in the following.

The Asokan-Shoup-Waidner (ASW) protocol serves as a motivating example that demonstrates the expressiveness of our choreography language, particularly its support for explicit branching, non-deterministic choice, conditional behavior, and long-term memory access.

ASW is a fair contract signing protocol involving three participants: an originator O, a responder R, and a trusted third party (TTP). The protocol mainly ensures that either both parties obtain a binding contract, or neither does. The key challenges addressed by this protocol are:

- **Timeouts and abort scenarios**: Participants may timeout, leading to different protocol continuations.
- **Stateful TTP**: The TTP must maintain memory of previous contract states to prevent inconsistent responses.
- **Conditional logic**: Protocol actions depend on checking stored memory values.

C. Non-deterministic Choice and Branching

Algorithm 1 presents the ASW choreography using explicit branching in the message flow. After O sends M_1 to R, the responder can either timeout and trigger an abort request (shown in red), or continue by generating NR and replying with M_2. In the continuation, timeouts may occur again: after receiving M_2, O can either timeout and ask the ttp to resolve, or proceed by sending NO; symmetrically, after receiving NO, R can either timeout and resolve, or complete by sending NR. The explicit $+$-marked alternatives capture this non-determinism directly at the level of the choreography.

D. Conditional Behavior and Memory Access

The distinctive feature of ASW is the behavior of the trusted third party, which consults and updates long-term memory cells indexed by the contract identifier M_1. In both Abort and Resolve, the ttp first reads $\mathit{ttp\_mem}[M_1]$ (with blank denoting an uninitialized cell) and then branches on the stored value.

Algorithm 1 ASW Choreography

```
$\mathsf{O} : \nu Text.\nu NO.$
$\mathsf{O} \to \mathsf{R} : \underbrace{sign(f_1(\mathsf{O},\mathsf{R},\mathsf{ttp},Text,h(NO)),inv(pk(\mathsf{O})))}_{=:M_1}.$
$\mathsf{R} : \mathsf{R} \to \mathsf{O} : timeout.\textsc{Abort}(\mathsf{O},M_1)$
$+$ $\nu NR.$
    $\mathsf{R} \to \mathsf{O} : \underbrace{sign(f_2(M_1,h(NR)),inv(pk(\mathsf{R})))}_{=:M_2}.$
    $\mathsf{O} : \mathsf{O} \to \mathsf{R} : timeout.\textsc{Resolve}(\mathsf{R},M_1,M_2)$
    $+$ $\mathsf{O} \to \mathsf{R} : NO.$
        $\mathsf{R} : \mathsf{R} \to \mathsf{O} : timeout.\textsc{Resolve}(\mathsf{O},M_1,M_2)$
        $+$ $\mathsf{R} \to \mathsf{O} : NR$

$\textsc{Abort}(\mathsf{A},M_1) =$
$\mathsf{A} \to \mathsf{ttp} : sign(f_{abort}(M_1),inv(pk(\mathsf{A})))$
$\mathsf{ttp} : blank := ttp\_mem[M_1].$
    $ttp\_mem[M_1] := aborted.$
    $\mathsf{ttp} \to \mathsf{A} : sign(f_{abort}(M_1),inv(pk(\mathsf{ttp})))$
$+$  $aborted := ttp\_mem[M_1].$
    $\mathsf{ttp} \to \mathsf{A} : sign(f_{abort}(M_1),inv(pk(\mathsf{ttp})))$
$+$  $(resolved,M_2) := ttp\_mem[M_1].$
    $\mathsf{ttp} \to \mathsf{A} : sign(f_{resolve}(M_1,M_2),inv(pk(\mathsf{ttp})))$

$\textsc{Resolve}(\mathsf{A},M_1,M_2) =$
$\mathsf{A} \to \mathsf{ttp} : sign(f_{resolve}(M_1,M_2),inv(pk(\mathsf{A})))$
$\mathsf{ttp} : blank := ttp\_mem[M_1].$
    $ttp\_mem[M_1] := (resolved,M_2).$
    $\mathsf{ttp} \to \mathsf{A} : sign(f_{resolve}(M_1,M_2),inv(pk(\mathsf{ttp})))$
$+$  $aborted := ttp\_mem[M_1].$
    $\mathsf{ttp} \to \mathsf{A} : sign(f_{abort}(M_1),inv(pk(\mathsf{ttp})))$
$+$  $(resolved,M_2) := ttp\_mem[M_1].$
    $\mathsf{ttp} \to \mathsf{A} : sign(f_{resolve}(M_1,M_2),inv(pk(\mathsf{ttp})))$
```

This illustrates several language features:

1) **Memory read**: $blank := ttp\_mem[M_1]$ reads the current state associated with M_1.
2) **Case distinction on memory**: depending on whether $ttp\_mem[M_1]$ is still *blank*, has been set to *aborted*, or contains $(resolved, M_2)$, the ttp returns a consistently matching signed response.
3) **Memory write**: $ttp\_mem[M_1] := aborted$ (in ABORT) and $ttp\_mem[M_1] := (resolved, M_2)$ (in RESOLVE) record the outcome so that repeated requests cannot lead to contradictory replies.
4) **Non-monotonic memory**: the same cell can be read multiple times and updated across different interactions in the run.

E. Semantic Challenge: Participant Knowledge

A subtle aspect of ASW mechanization arises during projection. When O or R make decisions (e.g., "abort"), they must send signals that the ttp later interprets. However, the originating participant does not see the entire memory state of the TTP. During projection, the analyzer must determine:

- When to *accept* incoming values without immediate verification (e.g., when O receives an untagged nonce response).
- When to insert runtime *checks* that compare received values against later constraints (e.g., when verifying $h(NO) = X$ after learning NO).

VI. Related Work and Conclusions

Alice-and-Bob notation. Several lines of work have given Alice-and-Bob notation a precise semantics by compiling them to lower-level role-based specifications, and by rejecting notations that are not executable because e.g. senders cannot construct a message or receivers cannot check it. Early semantics were developed for the free term algebra and later extended to equational theories, often by reducing executability questions to intruder deduction and unification problems [6], [11], [7]. Our semantics follows this tradition, but lifts it from linear notations to *choreographies* with non-deterministic choice, branching, and mutable long-term memory. This combination is essential for modeling modern protocol interactions with stateful services and APIs, while maintaining the compact global view that motivates Alice-Bob notations.

Protocol models and tool support. At the other end of the spectrum, multi-set rewriting languages (e.g., as used by Tamarin and the AVISPA family) provide an explicit account of state and message flows and are well-suited for reasoning with rich adversary models [1], [2], [12]. Process-calculus based tools such as ProVerif offer a more program-like view of each role and highly automated verification, typically by a sound over-approximation [13], [3]. Our contribution is complementary: CryptoChoreo aims to be a high-level *specification* language that can be translated into such backends. In this respect, our approach is aligned with lines of work that provide source-to-source translations or front-ends for existing verification tools, such as SAPIC/SAPIC+ [14], [15], but with a focus on preserving the readability and single global story of a choreography.

Choreographies and global types. Choreographic programming and the theory of multiparty session types/global types study how a global description of multiparty interaction can be projected to local behaviors, with correctness guarantees such as deadlock freedom [16], [17], [18]. Our work is inspired by the same global-to-local methodology, but targets the Dolev–Yao setting with an active adversary, cryptographic constructors/destructors, equational theories, and explicit attacker knowledge.

State and APIs in protocol models. Stateful extensions and encodings have been studied both at the specification level and in tool-oriented front-ends [14]. CryptoChoreo makes state explicit at the choreography level via memory cells.

Equational reasoning and mechanization. Reasoning about message construction, parsing, and checks in the presence of equational theories is a classic challenge in symbolic protocol analysis [19]. Our mechanization focuses on a representative theory combining standard constructors/destructors with Diffie–Hellman exponentiation, and yields an effective projection procedure [20]. This is intentionally backend-agnostic at the level of the core semantics, while our ProVerif export demonstrates one concrete and practical target.

Conclusions. We introduced CryptoChoreo, a choreography language for cryptographic protocols that extends Alice-and-Bob notation with non-deterministic choice, conditional branching, and mutable long-term memory, together with a projection-based semantics and an effective mechanization for a representative equational theory. Our ProVerif export shows that these high-level specifications can be connected to automated verification backends in practice. Future work includes widening the class of supported equational theories, improving automation for non-monotonic state encodings, and adding additional backend targets.

References

[1] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The Tamarin prover for the symbolic analysis of security protocols," in *Computer Aided Verification (CAV)*, 2013.

[2] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. Drielsma, P. Héam, J. Mantovani, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron, "The AVISPA tool for the automated validation of internet security protocols and applications," in *Computer Aided Verification (CAV)*, 2005.

[3] B. Blanchet, *ProVerif: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*, 2016.

[4] J. K. Millen, "CAPSL: common authentication protocol specification language," in *Workshop on New Security Paradigms*. ACM, 1996, p. 132.

[5] G. Lowe, "Casper: A compiler for the analysis of security protocols," *J. Comput. Secur.*, vol. 6, no. 1-2, pp. 53–84, 1998.

[6] F. Jacquemard, M. Rusinowitch, and L. Vigneron, "Compiling and verifying security protocols," in *Logic for Programming and Automated Reasoning (LPAR)*, 2000.

[7] S. Mödersheim, "Algebraic properties in alice and bob notation," in *International Conference on Availability, Reliability and Security (ARES)*, 2009.

[8] Y. Chevalier and M. Rusinowitch, "Compiling and securing cryptographic protocols," *Inf. Process. Lett.*, vol. 110, no. 3, pp. 116–122, 2010.

[9] O. Almousa, S. Mödersheim, and L. Viganò, "Alice and Bob: Reconciling formal models and implementation," in *Festschrift in honor of Pierpaolo Degano*, 2015.

[10] D. A. Basin, M. Keller, S. Radomirovic, and R. Sasse, "Alice and bob meet equational theories," in *Festschrift in honor of José Meseguer*, 2015.

[11] C. Caleiro, D. Basin, and L. Viganò, "On the semantics of alice & bob specifications of security protocols," *Theoretical Computer Science*, 2006.

[12] D. Basin, S. Mödersheim, and L. Viganò, "OFMC: A symbolic model checker for security protocols," in *International Journal of Information Security*, 2005.

[13] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Computer Security Foundations Workshop (CSFW)*, 2001.

[14] S. Kremer and R. Künnemann, "Automated analysis of security protocols with global state," *Journal of Computer Security*, 2016.

[15] V. Cheval, B. Blanchet, and B. Smyth, "SAPIC+: A protocol specification language for the verification of indistinguishability properties," in *IEEE Computer Security Foundations Symposium (CSF)*, 2022.

[16] M. Carbone, K. Honda, and N. Yoshida, "Structured communication-centered programming for web services," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 2, pp. 8:1–8:78, 2012. [Online]. Available: https://doi.org/10.1145/2220365.2220367

[17] M. Carbone and F. Montesi, "Deadlock-freedom-by-design: multiparty asynchronous global programming," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, R. Giacobazzi and R. Cousot, Eds. ACM, 2013, pp. 263–274. [Online]. Available: https://doi.org/10.1145/2429069.2429101

[18] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," *J. ACM*, vol. 63, no. 1, pp. 9:1–9:67, 2016. [Online]. Available: https://doi.org/10.1145/2827695

[19] Y. Chevalier and L. Vigneron, "Automated unbounded verification of security protocols," in *Computer Aided Verification (CAV)*, 2002.

[20] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated analysis of diffie–hellman protocols and advanced security properties," in *IEEE Computer Security Foundations Symposium (CSF)*, 2012.

[21] R. Küsters and T. Truderung, "Using proverif to analyze protocols with diffie-hellman exponentiation," in *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2009, pp. 157–171.

[22] S. Mödersheim, "Diffie-hellman without difficulty," in *Formal Aspects of Security and Trust - 8th International Workshop*, ser. Lecture Notes in Computer Science, G. Barthe, A. Datta, and S. Etalle, Eds., vol. 7140. Springer, 2011, pp. 214–229.

[23] V. Cheval, V. Cortier, and M. Turuani, "A little more conversation, a little less action, a lot more satisfaction: Global states in proverif," in *31st IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2018, pp. 344–358.

Appendix

A. Exporting to ProVerif (extended)

a) ProVerif encoding: In this section, we will describe how to use the projection semantics from Section III to automatically convert a choreography into a ProVerif process. For the rest of this section, we assume the following:

- We have a choreography $\mathcal{C}$
- In $\mathcal{C}$ there are roles $\mathsf{A}_1, \dots, \mathsf{A}_{n_\mathsf{A}}$
- The initial knowledges are given in $F_{\mathsf{A}_1}, \dots, F_{\mathsf{A}_{n_\mathsf{A}}}$
- We have projected the choreography for each role: $\mathcal{L}_{\mathsf{A}_1} = F_{\mathsf{A}_1}([\![F_{\mathsf{A}_1} : \mathcal{C}]\!]_{\mathsf{A}_1}), \dots, \mathcal{L}_{n_\mathsf{A}} = F_{n_\mathsf{A}}([\![F_{n_\mathsf{A}} : \mathcal{C}]\!]_{n_\mathsf{A}})$
- Of all the roles, $\mathsf{U}_1, \dots, \mathsf{U}_{n_\mathsf{U}}$ are untrusted.
- For each role A_i, $\mathsf{U}_{\mathsf{A}_i 1}, \dots, \mathsf{U}_{\mathsf{A}_i n_{\mathsf{U}_{\mathsf{A}_i}}}$ are the untrusted roles occurring in $\mathcal{L}_{\mathsf{A}_i}$ and F_{A_i} (i.e. the parameters to the role)
- In $\mathcal{C}$ we have the cell families $c_1, \dots, c_{n_c}$

We start by encoding the algebra. For the most part, we encode the algebra E as is, though a simplification must be made with regard to $\exp$. Directly inserting the equation in B in ProVerif will lead to nontermination. However, as shown in [21] and [22], it is sound to use the following encoding for standard Diffie-Hellman:

```
fun exp(bitstring, bitstring): bitstring.
const g: bitstring.
equation forall x: bitstring, y: bitstring;
  exp(exp(g, y), x) = exp(exp(g, x), y).
```

We will omit most of the definition of the algebra here, but include the encoding of symmetric encryption:

```
type skey.
fun scrypt(bitstring, skey): bitstring.
reduc forall x: bitstring, k: skey;
  dscrypt(scrypt(x, k), k) = x.
fun vscrypt(bitstring, skey): bool
```

reduc forall x: bitstring, k: skey;
  vscrypt(scrypt(x, k), k) = true
otherwise forall x: bitstring, k: skey;
  vscrypt(x, k) = false.

As another preliminary step, we include the type of agents and the name of the intruder:
type agent.
free i: agent.

ProVerif processes are written in the *π-calculus*, and look much like our local behaviors, except for network communication, nondeterministic choice, and memory cells. We write $[\![\mathcal{L}]\!]_{\mathsf{pv}}$ for the ProVerif process obtained from the local behavior $\mathcal{L}$, and in the following only describe the cases where the local behaviors differ from the resulting ProVerif processes.

In our local behaviors, we simply use send(X) and receive(X) to send the content of variable X to the public network or receive a value from the network and assign it to X. In ProVerif, the corresponding statements are $\mathbf{out}(\mathsf{c}, X)$ and $\mathbf{in}(\mathsf{c}, X{:}\mathsf{bitstring})$, where c is a public channel. To make a clear conceptual divide, we declare two public channels: c is the public channel the communication declared in the choreography happens over, while we use ic when information must be given to or taken from the intruder for the purpose of modeling.
free ic: channel.
free c: channel.
We then define $[\![\mathsf{send}(t).\mathcal{L}]\!]_{\mathsf{pv}} = \mathbf{out}(\mathsf{c}, t); [\![\mathcal{L}]\!]_{\mathsf{pv}}$ and $[\![\mathsf{receive}(X).\mathcal{L}]\!]_{\mathsf{pv}} = \mathbf{in}(\mathsf{c}, X{:}\mathsf{bitstring}); [\![\mathcal{L}]\!]_{\mathsf{pv}}$.

When a process makes a nondeterministic choice, our encoding simply has the intruder decide:

```
$[\![\mathcal{L}_1 + \mathcal{L}_2 + \ldots + \mathcal{L}_n]\!]_{\mathsf{pv}} =$
  $\mathbf{in}(\mathsf{ic}, \mathit{Branch}{:}\mathit{nat})$;
  $\mathbf{if}$ $\mathit{Branch} = 0$ $\mathbf{then}$
    $[\![\mathcal{L}_1]\!]_{\mathsf{pv}}$
  $\mathbf{else\ if}$ $\mathit{Branch} = 1$ $\mathbf{then}$
    $[\![\mathcal{L}_2]\!]_{\mathsf{pv}}$
      $\vdots$
  $\mathbf{else\ if}$ $\mathit{Branch} = n$ $\mathbf{then}$
    $[\![\mathcal{L}_n]\!]_{\mathsf{pv}}$
```

With regard to memory cells, we restrict ourselves to choreographies with the following properties:

- In every atomic section there is at most one read and one write to each cell family on every branch, and if a branch contains both they must use the same address.
- All writes happen at the end of the atomic section.
- All addresses are public. [8]

For each cell family, c_i, we declare a function:
fun cell_c_i(bitstring): channel [$\mathit{private}$].
We can use this function to create a private channel for each cell in the family. Each such channel will contain at most one message at any given time, representing the current value stored in the cell.

[8]This is not technically required for the translation to be sound. However, our translation will reveal addresses to the intruder, so you will get false attacks if this is not the case.

Information sent on a private channel is of course not revealed to the intruder. Furthermore, communication over private channels is synchronous, and we can use this to enforce atomicity when there is a read and write to the same cell in one atomic section.

The same trick can be used to create atomic sections in general. We define
free atomic_lock: channel [$\mathit{private}$].
const atomic_baton: bitstring.
To enter an atomic section, a process must obtain the baton, and they should send it back when they leave. We have not found it beneficial to enforce atomicity of all atomic sections from local behaviors, but use it strategically for some parts (for example the memory-initializer processes below).

We ensure that when an agent wants to read from a cell there is always a message available by including initializer processes for each cell family. ProVerif supports *tables*, which are set-like structure we can use to keep track of which cells have been initialized, thus making sure that we only initialize a cell once and enforce the property that each cell channel contains at most one message. For each c_i, we define:

```
$\mathbf{table}$ $c_i$_initializer_table(bitstring).
$\mathbf{let}$ $c_i$_$\mathit{initializer}$() =
    $\mathbf{in}$(atomic_lock, = atomic_baton);
    $\mathbf{in}$(ic, $\mathit{Addr}$: bitstring);
    $\mathbf{get}$ $c_i$_initializer_table(= $\mathit{Addr}$) $\mathbf{in}$ 0
    $\mathbf{else}$
        $\mathbf{insert}$ $c_i$_initializer_table($\mathit{Addr}$);
        ($\mathbf{out}$(cell_$c_i$($\mathit{Addr}$), blank) |
         $\mathbf{out}$(atomic_lock, atomic_baton)).
```

The outputs at the end must be put in parallel, so that one does not block the other.

We can now have the intruder initialize our memory cells:
$[\![X := c[t].\mathcal{L}]\!]_{\mathsf{pv}} = \mathbf{out}(\mathsf{ic}, t); \mathbf{in}(\mathsf{cell\_}c(t), X{:}\ \mathsf{bitstring}); [\![\mathcal{L}]\!]_{\mathsf{pv}}$
Also, writing should not block the rest of the process:
$[\![c[t] := s.\mathcal{L}]\!]_{\mathsf{pv}} = (\mathbf{out}(\mathsf{cell\_}c(t), s) \mid [\![\mathcal{L}]\!]_{\mathsf{pv}})$.

There are two remaining problems: If on a branch in an atomic section there is a read from a cell but no write, the value in the cell will be consumed and no other process will be able to read from it. Similarly, if there is a write but no read, the cell will contain two values, and either is a possible value when the next process reads from the cell (we also lose the atomicity mentioned above). We solve this by a simple transformation: If we are done translating a branch of an atomic section where we have processed $X := c[t]$ but no corresponding write and need to return the rest of the translation $[\![\mathcal{L}]\!]_{\mathsf{pv}}$, return instead $(\mathbf{out}(\mathsf{cell\_}c(t), s) \mid [\![\mathcal{L}]\!]_{\mathsf{pv}})$. Similarly, if we want to translate $[\![c_1[t_1] := s_1. \ldots .c_n[t_n] := s_n.\mathcal{L}]\!]_{\mathsf{pv}}$ (recall that writes must be at the end of an atomic section) and have not read from c_i, we return $\mathbf{out}(\mathsf{ic}, t_i); \mathbf{in}(\mathsf{cell\_}c_i(t_i), \_{:}\ \mathsf{bitstring}); [\![c_1[t_1] := s_1. \ldots .c_n[t_n] := s_n.\mathcal{L}]\!]_{\mathsf{pv}}$.

We now how a way to translate local behaviors to ProVerif processes. However, we still need to compose all these into a single process that can be verified by ProVerif. Furthermore, this process must handle the multiple sessions and instantiations supported by the semantics in Figure 5.

For each role A_i with parameters $\mathsf{U}_{\mathsf{A}_i 1}, \ldots, \mathsf{U}_{\mathsf{A}_i n_{\mathsf{U}_{\mathsf{A}_i}}}$ we define the *process spawner* for that agent:

```
let $processA_i(U_{A_i 1}\colon agent, \ldots, U_{A_i n_{U_{A_i}}}\colon agent) = [\![\mathcal{L}_{A_i}]\!]_{pv}$.
let $spawnA_i()$ =
  in(ic, $(U_{A_i 1}\colon agent, \ldots, U_{A_i n_{U_{A_i}}}\colon agent)$);
  $processA_i(U_{A_i 1}, \ldots, U_{A_i n_{U_{A_i}}})$.
```

For each untrusted U_i we must also give the intruder the associated initial knowledge. Let $\mathsf{U}_{\mathsf{U}_i 1}, \ldots, \mathsf{U}_{\mathsf{U}_i n}$ be all the parameters of the role U_i except the role U_i itself and $t_1, \ldots, t_m$ be the terms in the knowledge of U_i (i.e. $\mathcal{F}_{\mathsf{U}_i} = \{\mathcal{X}_1 \mapsto t_1, \ldots, \mathcal{X}_m \mapsto t_m\}$). We define:

```
let $knowledgeU_i()$ =
  in(ic, $(U_{U_i 1}\colon agent, \ldots, U_{U_i n}\colon agent)$);
  out(ic, $(t_1[i/U_i], \ldots, t_m[i/U_i])$).
```

Finally, we can define the main ProVerif process:

```
process
  (out(atomic_lock, atomic_baton)) |
  !(new $a$: agent; out(ic, $a$)) |
  !($spawnA_1()$) |
    ⋮
  !($spawnA_{n_A}()$) |
  !($knowledgeU_1()$) |
    ⋮
  !($knowledgeU_{n_U}()$) |
  !$c_1$_initializer()
    ⋮
  !$c_{n_c}$_initializer()
```

b) Improved Encoding of Memory Cells: When verifying ProVerif code produced by the encoding above, we run into the issue that the overapproximations associated with the abstractions of ProVerif make many secure protocols unverifiable. By default, ProVerif will consider the values in a cell as a set, and when you read you will not necessarily get the last value that was written. This means that for example the TPM example included with our submission does not verify, since ProVerif considers runs where *deleted* is written to the cell and then *classified* is later read from it.

We can improve the accuracy of our encoding by a trick inspired by the "precise" option in ProVerif, originally presented in [23]. When writing t to a cell c, we instead write (i_c, t) where i_c is a natural number indicating that this was the i_cth value written to c. Furthermore, we trigger an event registering what was written, every time a value is written to a call. The new encoding of cell c_i becomes:

```
fun cell_$c_i$(bitstring): channel [private].
event write_$c_i$(bitstring, nat, bitstring).
table $c_i$_initializer_table(bitstring).
let $c_i$_initializer() =
  in(atomic_lock, = atomic_baton);
  in(ic, $Addr$: bitstring);
  get $c_i$_initializer_table(= $Addr$) in 0
  else
    event write_$c_i$($Addr$, 0, blank);
    insert $c_i$_initializer_table($Addr$);
    (out(cell_$c_i$($Addr$), (0, blank)) |
     out(atomic_lock, atomic_baton)).
```

Furthermore, we redefine the process translation with the following:

$$[\![X := c[t].\mathcal{L}]\!]_{\mathsf{pv}} = \begin{array}{l}\mathbf{out}(\mathsf{ic}, t);\\ \mathbf{in}(\mathsf{cell\_}c(t), (\mathit{Counter}_c\colon \mathsf{nat}, X\colon \mathsf{bitstring}));\\ [\![\mathcal{L}]\!]_{\mathsf{pv}}\end{array}$$

$$[\![c[t] := s.\mathcal{L}]\!]_{\mathsf{pv}} = \begin{array}{l}\mathbf{event}\ \mathsf{write\_}c(t, \mathit{Counter}_c, s);\\ (\mathbf{out}(\mathsf{cell\_}c(t), (\mathit{Counter}_c + 1, s)) \mid [\![\mathcal{L}]\!]_{\mathsf{pv}})\end{array}$$

When inserting writes on branches that only have reads, we can omit incrementing the counter, as we will necessarily just write back the value that was already there. In our experiments, this omission increases the efficiency of ProVerif in many cases.

Our encoding ensures that every time a new value is put in the cell, it is associated with a new counter value. Thus, it is sound to add the following axiom:

```
axiom $t$: bitstring, $C$: $nat$, $s_1$: bitstring, $s_2$: bitstring;
  event(write_$c_i$($t, C, s_1$)) $\wedge$ event(write_$c_i$($t, C, s_2$))
    $\Longrightarrow s_1 = s_2$.
```

B. Mechanization of the Projection in Detail

We will here show in more detail how to solve the *word problem*, *recipe-composition problem*, and *complete-set-of-checks problem* described in Section IV.

In addition to the Σ and Σ_p from Section IV, we use Σ_d to denote the set of destructors (+ pair projections, Diffie-Hellman inverse, etc.) and Σ_v for the set of verifiers.

Unlike in the main matter of the paper, we will here permit terms $s, t, \ldots$ to contain all function symbols, as we here make precise how these terms are then reduced to ones built only from Σ.

For the benefit of the following proofs, we define the syntactic size of terms and checks: $|\mathcal{X}| = 1$, $|f(t_1, \ldots, t_n)| = 1 + |t_1| + \ldots + |t_n|$, and $|t \doteq s| = |t| + |s|$.

Additionally, we need the following concepts:

Definition 3. *We define $[t]_{=B} = \{t' \mid t =_B t'\}$. We write $\to_R$ for the rewriting system obtain by applying the equations in R from left to right. We can then define the rewriting system modulo B-equivalence classes: $s \to_{R/B} t$ iff $\exists s'\ t'.\ s =_B s' \wedge s' \to_R t' \wedge t' =_B t$. We use $t\downarrow_{R/B}$ to denote the normal form of t.*

Lemma 1. *If $s =_B t$ then $|s| = |t|$ and if $s \to_{R/B} t$ then $|s| > |t|$.*

Proof. The first part can be shown by induction on the derivation of $s =_B t$, and the second part follows by case analysis on $s' \rightarrow_R t'$ after unfolding the definition of $\rightarrow_{R/B}$. □

Lemma 2. *$s =_E t$ if and only if either:*

- *$s =_B t$,*
- *or there is s' so $s \rightarrow_{R/B} s'$ and $s' =_E t$,*
- *or there is t' so $t \rightarrow_{R/B} t'$ and $s =_E t'$*

Proof. We do induction on the derivation of $s =_E t$ in the equational logic induced by E. □

We can now show that our rewriting system is well-behaved:

Lemma 3. *(A) $[t]_{=B}$ is finite for all t. (B) If $s \rightarrow_{R/B} t$ then $s =_E t$. (C) $\rightarrow_{R/B}$ is convergent, modulo B. (D) $t \downarrow_{R/B}$ is defined and unique for all t, modulo B.*

Proof. (A) is easy to see (the equation in B is a kind of commutativity). (B) follows from transitivity of $=_E$ and the definitions of $\rightarrow_R$ and $\rightarrow_{R/B}$. (D) follows (C) by definition.

To show (C), we combine (B) with the following: (1) $\rightarrow_{R/B}$ has no infinite chains. (2) If $s =_E t$ and $\rightarrow_{R/B}$ applies to neither s nor t, then $s =_B t$.

(1) follows from the fact that if $t_1 \rightarrow_{R/B} t_2 \rightarrow_{R/B} \ldots$ then $|t_1| > |t_2| > \ldots$.

(2) follows from Lemma 2. □

With this, we can solve the *word problem* from the start of the section:

Theorem 1. *$s =_E t$ if and only if $s \downarrow_{R/B} =_B t \downarrow_{R/B}$.*

Proof. That $s \downarrow_{R/B} =_B t \downarrow_{R/B}$ implies $s =_E t$ follows from Lemma 3 (B).

If $s =_E t$ then $s \downarrow_{R/B} =_E t \downarrow_{R/B}$ by Lemma 3 (B), and since $\rightarrow_{R/B}$ applies to neither $s \downarrow_{R/B}$ nor $t \downarrow_{R/B}$ we have $s \downarrow_{R/B} =_B t \downarrow_{R/B}$ by Lemma 2. □

In the following, we will analyze a set of frames by alternating between applying verifiers and destructors to obtain new terms. We extend frames with additional information, including which checks have been performed, and how the content in a label was derived.

Definition 4 (Enhanced Frames)**.** *Frames are built over the following grammar:*

$$\begin{array}{rcl} F & ::= & 0 \\ & \mid & F_0.\mathcal{X} \mapsto t \\ & \mid & F_0.\mathcal{X} \leftarrow r \mapsto t \\ & \mid & F_0.r_1 \doteq r_2 \end{array}$$

We let e range over the entries of the form $\mathcal{X} \mapsto t$, $\mathcal{X} \leftarrow r \mapsto t$, and $r_1 \doteq r_2$.

- *The* domain *of a frame is defined as follows.* $\mathrm{dom}(0) = \emptyset$, $\mathrm{dom}(F_0.\mathcal{X} \mapsto t) = \mathrm{dom}(F_0) \cup \{\mathcal{X}\}$, $\mathrm{dom}(F_0.\mathcal{X} \leftarrow r \mapsto t) = \mathrm{dom}(F_0) \cup \{\mathcal{X}\}$, *and* $\mathrm{dom}(F_0.r_1 \doteq r_2) = \mathrm{dom}(F_0)$.
- *Given a recipe r for frame F, we define $F(r)$ as expected, ignoring $r_1 \doteq r_2$ and $\leftarrow r$.*

In the following, we only consider well-formed frames, *which satisfy these properties:*

- *In a frame $F_0.\mathcal{X} \mapsto t$, $\mathcal{X} \notin \mathrm{dom}(F_0)$ and t is constructive.*
- *In a frame $F_0.\mathcal{X} \leftarrow r \mapsto t$, $\mathcal{X} \notin \mathrm{dom}(F_0)$, $\mathsf{fv}(r) \subseteq \mathrm{dom}(F_0)$, $F_0(r) = t$, and there exists constructive t' such that $t =_E t'$.*
- *In a frame $F_0.r_1 \doteq r_2$, $\mathsf{fv}(r_1) \cup \mathsf{fv}(r_2) \subseteq \mathrm{dom}(F_0)$, and $F_0(r_1) =_E F_0(r_2)$.*
- *In a frame $F_0.e$, F_0 is well-formed.*

It is easy to see that the projection semantics above and the following procedures preserve well-formedness.

The checks *contained in a frame F, written $checks(F)$, is the set of all the equations $r_1 \doteq r_2$ contained in the frame together with the equations $l \doteq r$ for each entry $l \leftarrow r \mapsto t$ in the frame.*

In the projection semantics, we consider sets of frames. There is a notion that these frames are identical with regard to all operations we have performed so far, which we formalize in the following:

Definition 5 (Compatible Frames)**.** *We define two frames being* compatible*, written $F \simeq F'$, as the least relation satisfying the following rules:*

$$\frac{}{0 \simeq 0}$$

$$\frac{F_0 \simeq F_0'}{F_0.\mathcal{X} \mapsto t \simeq F_0'.\mathcal{X} \mapsto t'}$$

$$\frac{F_0 \simeq F_0'}{F_0.\mathcal{X} \leftarrow r \mapsto t \simeq F_0'.\mathcal{X} \leftarrow r \mapsto t'}$$

$$\frac{F_0 \simeq F_0'}{F_0.r_1 \doteq r_2 \simeq F_0'.r_1 \doteq r_2}$$

Thus, frames are compatible if they have passed the same checks and performed the same operations to construct new terms, in the same order.

Our analysis should extend a frame to contain all derivable subterms. This goal is captured in the following:

Definition 6. *A label $\mathcal{X} \in \mathrm{dom}(F)$ is* analyzed *(in F) if for any constructive key-recipe r_k and destructor $d \in \Sigma_d$, if d applies to $(F(\mathcal{X}), F(r_k))$, there exists an entry $\mathcal{X}' \leftarrow d(\mathcal{X}, r_k') \mapsto t$ where $F(r_k) =_E F(r_k')$. Furthermore, for the associated verifier v there must be an entry $v(\mathcal{X}, r_k') \doteq \top$.*
A frame F is fully analyzed *if all labels in $\mathrm{dom}(F)$ are analyzed.*

Note that since F is well-formed, we must have $t =_E d(F(\mathcal{X}), F(r_k))$.

When all the subterms have been added to a frame, we can obtain any obtainable term by *composition*, i.e. a constructive recipe. The following recursive function returns all constructive recipes for a given term:

Definition 7. *We define* $compose(F, t) = \bigcup\{compose_{lc}(F, s) \mid s \in [t]_{=B}\}$ *where*
$compose_{lc}(F, s) = compose_l(F, s) \cup compose_c(F, s)$,
$compose_l(F, s) = \{\mathcal{X} | \mathcal{X} \in \mathrm{dom}(F) \wedge F(\mathcal{X}) = s\}$, *and*

$compose_c(F,s) = \{f(s'_1,\dots,s'_n) \mid s'_1 \in compose(F,s_1) \wedge \dots \wedge s'_n \in compose(F,s_n)\}$ if $s = f(s_1,\dots,s_n)$ for $f \in \Sigma_c$ and $compose_c(F,s) = \{\}$ otherwise.

That $compose(F,t)$ is finite can be seen by induction on $|t|$ and that $[t]_{=B}$ is finite.

The following shows that for fully analyzed frames, $compose$ can create any obtainable term:

Lemma 4. *Assume that F is a fully analyzed frame.*
If there is a recipe r and a constructive term t such that $F(r) =_E t$, then we have that
(1) $compose(F,t)$ is non-empty, and
(2) for every $r_c \in compose(F,t)$ we have $F(r_c) =_E t$.

Proof. (1) We show that if $F(r) \to^*_{R/B} t$ then there is $r_c \in compose(F,t)$ such that $F(r_c) \to^*_{R/B} t$.
We do induction on $|r|$. If $r = \mathcal{X}$, we are done. If $r = f(r_1,\dots,r_n)$ for $f \in \Sigma_c$, we apply the induction hypothesis and are done. If $r = d(r_1,\dots,r_n)$ for $f \in \Sigma_d$, then either the recipe reduces and we are done by the induction hypothesis, or $r = d(\mathcal{X}, r_{key})$ and there is another label $\mathcal{X}'$ and (by the induction hypothesis) a constructive recipe r'_{key} such that $F(\mathcal{X}') = F(d(\mathcal{X}, r'_{key}))$.

(2) It is easy to prove that for every $r_c \in compose_{lc}(F,t)$ we have $F(r_c) =_E t$ by induction on $|t|$. Furthermore, it is by definition the case that for every $s \in [t]_{=B}$ we have $s =_E t$. □

The following shows completeness of $compose$:

Lemma 5. *For any frame F (even if not fully analyzed), if r is a constructive recipe such that $F(r) =_E t$ and t is constructive, then $r \in compose(F,t)$.*

Proof. Follows by induction on $|r|$. □

In the following, we define an analysis procedure that can be triggered during the projection semantics of Section III any time a new term is added the frame.

For the analysis of frame F, we require a *marking* of the type $M : \mathrm{dom}(F) \longrightarrow \{\star, +, \checkmark\}$. All labels are initially marked $\star$, and whenever a label is added to the frame it is also marked $\star$.

Definition 8. *A marking M of frame F is* accurate *if any label $\mathcal{X} \in \mathrm{dom}(F)$ where $M(X) = \checkmark$ is analyzed in F.*

Definition 9 (Analysis Procedure)**.** *We want to calculate the* analyzed extensions *of a set of frames, $\{F_1,\dots,F_n\}$, and marking, M, where the marking is accurate for each frame and the frames are pairwise compatible. The analysis-extension function, $\mathrm{Ana}(M, \{F_1,\dots,F_n\})$, is defined by three cases:*
If the given set is empty ($n = 0$), return $\{\}$.
If there is a label $\mathcal{X}$ marked $\star$, try the following in order:

- *$F_1(\mathcal{X}),\dots,F_n(\mathcal{X})$ are all terms for which no verifier exists, then return $\mathrm{Ana}(M[\mathcal{X} \mapsto \checkmark], \{F_1,\dots,F_n\})$.*
- *(decomposition) We have a frame F_i, a verifier $v \in \Sigma_v$, and a constructive term t such that v applies to $(F_i(\mathcal{X}), t)$ (for any d and $F_i(\mathcal{X})$ at most one such t exists and it is easy to find). Furthermore, $compose(F_i, t)$ is non-empty, containing at least r_{key}.*
Then, let $d_1,\dots,d_k$ be all destructors associated with v and $\mathcal{X}_1,\dots,\mathcal{X}_k$ fresh labels, and set $r_1 = d_1(\mathcal{X}, r_{key}),\dots,r_k = d_k(\mathcal{X}, r_{key})$.
Let M' be the marking that for any $\mathcal{X}'$ returns $\checkmark$ if $\mathcal{X} = \mathcal{X}'$, $\star$ if $\mathcal{X}' \in \{\mathcal{X}_1,\dots,\mathcal{X}_k\}$ or $M(\mathcal{X}') = +$, and $M(\mathcal{X}')$ otherwise.
Finally, let
$FS^+ = \{F.v(\mathcal{X}, r_{key}) \doteq \top.X_1 \leftarrow r_1 \mapsto F(r_1).\cdots.X_k \leftarrow r_k \mapsto F(r_k) \mid F \in \{F_1,\dots,F_n\} \wedge F(v(\mathcal{X}, r_{key})) =_E \top\}$
and
$FS^- = \{F \mid F \in \{F_1,\dots,F_n\} \wedge F(v(\mathcal{X}, r_{key})) \neq_E \top\}$.
We then return $\mathrm{Ana}(M', FS^+) \cup \mathrm{Ana}(M, FS^-)$.
- *Return $\mathrm{Ana}(M[\mathcal{X} \mapsto +], \{F_1,\dots,F_n\})$.*

If no labels are marked $\star$, return $\{(M, \{F_1,\dots,F_n\})\}$.

Note that the choice of r_{key} does not matter, since if multiple recipes are supposed to produce the same term but actually do not, we are not in a run of the protocol that agrees with the given frame. This will then be discovered when we later do the checks.

Theorem 2. *The analysis procedure terminates.*

Proof. Let $raw(F)$ be all the entries of the form $\mathcal{X} \mapsto t$ (i.e. the label assignments that are not the product of analysis). Let $|F| = \sum\{|t| \mid \mathcal{X} \mapsto t \in raw(F)\}$ and $|\{F_1,\dots,F_n\}| = \sum_{i=1}^{n} |F_i|$. Let $|M|^{\checkmark}$ be the number of labels marked $\checkmark$ in M and $|M|^{+}$ be the number of labels marked $+$. Lexicographically measure $(|\{F_1,\dots,F_n\}| - |M|^{\checkmark}, |\{F_1,\dots,F_n\}| - |M|^{+})$.
The number of labels in a marking will never exceed the number of subterms of the raw terms of the frames the marking is based on, so $|\{F_1,\dots,F_n\}| - |M|^{\checkmark}$ and $|\{F_1,\dots,F_n\}| - |M|^{+}$ will always be at least 0.
It is easy to see that no step in the analysis makes $|\{F_1,\dots,F_n\}| - |M|^{\checkmark}$ increase, and that each step will either decrease $|\{F_1,\dots,F_n\}| - |M|^{\checkmark}$ or $|\{F_1,\dots,F_n\}| - |M|^{+}$. □

Theorem 3. *If $F_1,\dots,F_n$ are compatible frames, M is an accurate marking, and $\mathrm{Ana}(M, \{F_1,\dots,F_n\}) = \{(M_1, FS_1),\dots,(M_m, FS_m)\}$, then all frames in each FS_i are fully analyzed, pairwise compatible, and $M_1,\dots,M_m$ are accurate.*

Proof. It is easy to see that the procedure preserves compatibility and accuracy of markings.

When the procedure is done, no label will be marked $*$. If one of F_{ij} is then not fully analyzed, there must be a label $\mathcal{X}$, a constructive term t, and (by Lemma 5) a recipe $r_k \in compose(F,t)$, such that the *(decomposition)* case is applicable for $\mathcal{X}$ (which must mean, by accuracy, that $\mathcal{X}$ is marked $+$). However, this cannot be the case, as if this r_k existed when the marking of $\mathcal{X}$ was changed from $\star$ to $+$ then the wrong case was applied, and if it did not exist then the frame must have changed since this, which would remark $\mathcal{X}$ as $\star$. $\square$

After distinguishing and extending a set of frames with this procedure, defining procedures for solving the two remaining problems stated at the beginning of this section becomes easier.

We start by showing that in an analyzed frame, all checks can be done in a constructive way:

Lemma 6. *If F is a fully analyzed frame, then for any check $r_a \doteq r_b$ where $F(r_a) =_E F(r_b)$ there exists a set of constructive checks $\phi_c$such that $\bigwedge checks(F) \wedge \bigwedge \phi_c \models_E r_a \doteq r_b$.*

If r_a is of the form $f(r'_a, r_k)$ where $f \in \Sigma_v \cup \Sigma_d$ and r'_a, r_k and r_b are constructive, then there is such a ϕ_c where for each check $r_1 \doteq r_2 \in \phi_c$ we have that $|r_1 \doteq r_2| \leq max(2 * |r_k|, |r'_a| + |r_b|)$.

Proof. We prove the first part by induction on $|r_a \doteq r_b|$.

The second part of the lemma can be proven by considering the cases of r'_a. First, we note that the proof is immediate if it is not the case that f is a destructor or verifier that applies to $(F(r'_a), F(r_k))$, and we therefore only consider this case.
If $r'_a = \mathcal{X}$ where $\mathcal{X}$ is a label, then we know that for the applicable verifier v and destructors $d_1, \ldots, d_n$ we have a constructive recipe r'_k and labels $\mathcal{X}_1, \ldots, \mathcal{X}_n$ such that $\{v(\mathcal{X}, r'_k) \doteq \top, d_1(\mathcal{X}, r'_k) \doteq \mathcal{X}_1 \ldots d_n(\mathcal{X}, r'_k) \doteq \mathcal{X}_n\} \subseteq checks(F)$. If $f = v$ we set $\phi_c = \{r_k \doteq r'_k, r_b \doteq \top\}$ and are done. If $f = d_i$ we set $\phi_c = \{r_k \doteq r'_k, r_b \doteq \mathcal{X}_i\}$ and are done.
If r'_a is a composed recipe (i.e. of the form $g(r'_{a1}, \ldots, r'_{an})$) then it must be the case that $f(r'_a, r_k) \rightarrow_R r''_a$ for some constructive recipe r''_a. We then set $\phi_c = \{r''_a \doteq r_b\}$ and are done. $\square$

Definition 10. *For a term F we define the set of all checks that hold in the frame:*
$\phi(F) = \{r_1 \doteq r_2 \mid r_1 \in \mathcal{T}(\Sigma_p, \mathrm{dom}(F)) \wedge r_2 \in \mathcal{T}(\Sigma_p, \mathrm{dom}(F)) \wedge F(r_1) =_E F(r_2)\}$
We also define the set of all checks between labels and recipes we can compose:
$\phi_{lc}(F) = \{\mathcal{X} \doteq r \mid \mathcal{X} \in \mathrm{dom}(F) \wedge r \in compose(F, F(\mathcal{X}))\}$

Since a frame is finite, and for any t we have that $compose(F, t)$ is finite, it must be that $\phi_{lc}(F)$ is finite.

We can now solve the *complete set of checks* problem:

Theorem 4. *If F is fully analyzed, then for any $r_1 \doteq r_2 \in \phi(F)$ we have $\bigwedge checks(F) \wedge \bigwedge \phi_{lc}(F) \models_E r_1 \doteq r_2$*

Proof. Because of Lemma 6 we can assume that r_1 and r_2 are constructive. We do induction on $|r_1 \doteq r_2|$.

If either r_1 or r_2 is a label, we are done.

If $r_1 = f(r_{11}, \ldots, r_{1n})$ and $r_2 = g(r_{21}, \ldots, r_{2n})$, we must have $f = g$. Furthermore, if $r_{11} \doteq r_{21}, \ldots, r_{1n} \doteq r_{2n}$ are valid checks in F, and we apply the induction hypothesis to get checks $\phi_1, \ldots, \phi_n$. We have $\bigwedge checks(F) \wedge \bigwedge \phi_1 \wedge \ldots \wedge \bigwedge \phi_n \models_E r_1 \doteq r_2$ and are done.

The only hard part is if $r_1 = \exp(r_{1a}, r_{1b})$ and $r_2 = \exp(r_{2a}, r_{2b})$ when $F(r_{1a}) \neq_E F(r_{2a})$ or $F(r_{1b}) \neq_E F(r_{2b})$. We then have either that $F(r_{1a}) =_E \exp(s, t)$ and $t =_E F(r_{2b})$ or $F(r_{2a}) =_E \exp(s, t)$ and $t =_E F(r_{1b})$, for some constructive s and t. We consider the former and the proof for the latter is symmetric.
We get $exp^{-1}(F(r_{1a}), F(r_{2b})) = s$ and $vexp(F(r_{1a}), F(r_{2b})) = \top$. From $exp^{-1}(F(r_{1a}), F(r_{2b})) = s$ and Lemma 4, we get a constructive recipe r' such that $F(r') = s$. Observe that $|r'| < |r_{1a}|$.
By Lemma 6 we get that there must be constructive checks $\phi^{r'}$ such that $\bigwedge checks(F) \wedge \bigwedge \phi^{r'} \models_E exp^{-1}(r_{1a}, r_{2b}) \doteq r' \wedge vexp(r_{1a}, r_{2b}) = \top$ and for all $r_\phi \doteq r'_\phi \in \phi^{r'}$ we have $|r_\phi \doteq r'_\phi| \leq max(2 * |r_{2b}|, |r_{1a}| + |r'|)$. Since $|r_{2b}| < |r_{1a}|$ we further have that for all $r_\phi \doteq r'_\phi \in \phi^{r'}$ we have $|r_\phi \doteq r'_\phi| < |r_1 \doteq r_2|$. The induction hypothesis then gives a set of checks $\phi^{r'}_{lc} \subseteq \phi_{lc}(F)$ such that $\bigwedge checks(F) \wedge \bigwedge \phi^{r'}_{lc} \models_E \bigwedge \phi^{r'}$.
Furthermore, we have that $F(\exp(r', r_{1b})) =_E F(r_{2a})$, which together with the induction hypothesis gives a set of checks $\phi^{r_{2a}}_{lc} \subseteq \phi_{lc}(F)$ such that $\bigwedge checks(F) \wedge \bigwedge \phi^{r_{2a}}_{lc} \models_E \exp(r', r_{1b}) \doteq r_{2a}$.
Combining the above with
$exp^{-1}(r_{1a}, r_{2b}) \doteq r' \wedge vexp(r_{1a}, r_{2b}) = \top \models_E r_{1a} \doteq \exp(r', r_{2b})$
and
$r_{1a} \doteq \exp(r', r_{2b}) \wedge \exp(r', r_{1b}) \doteq r_{2a} \models_E \exp(r_{1a}, r_{1b}) \doteq \exp(r_{2a}, r_{2b})$
yields
$\bigwedge checks(F) \wedge \bigwedge \phi^{r'}_{lc} \wedge \bigwedge \phi^{r_{2a}}_{lc} \models_E r_1 \doteq r_2$
and we are done. $\square$

It still remains to solve *recipe composition*. We have already shown that *compose* will compose a recipe for a term in a single fully analyzed frame if it exists, but it remains to be shown that we can use it to compose a recipe that will work for all given frames. To prove this, we will leverage the complete set of checks.

Definition 11. *A frame F is* fully checked *if for any $r_1 \doteq r_2 \in \phi(F)$ we have $checks(F) \models_E r_1 \doteq r_2$.*

The following checking procedure simply mimics the one from Section III:

Definition 12 (Checking Procedure)**.** *We want to calculate the* checked extensions *of a set of frames, $\{F_1, \ldots, F_n\}$ where the frames are pairwise compatible. This is given by* $\mathrm{Check}(\phi_{lc}(F_1) \cup \ldots \cup \phi_{lc}(F_n), \{F_1, \ldots, F_n\})$ *where* Check *is the* check-extension function. $\mathrm{Check}(\phi, \{F_1, \ldots, F_n\})$ *is defined by three cases:*
If $\phi = \{\}$, return $\{\{F_1, \ldots, F_n\}\}$.
If $n = 0$, return $\{\}$.
If $r_1 \doteq r_2 \in \phi$, let $\phi' = \phi \setminus r_1 \doteq r_2$, $FS^+ = \{F.r_1 \doteq r_2 \mid F \in \{F_1, \ldots, F_n\} \wedge F(r_1) =_E F(r_2)\}$
and $FS^- = \{F \mid F \in \{F_1, \ldots, F_n\} \wedge F(r_1) \neq_E F(r_2)\}$,
and then return $\mathrm{Check}(\phi', FS^+) \cup \mathrm{Check}(\phi', FS^-)$

This procedure is sound:

Lemma 7. *If $\{F_1, \dots, F_n\}$ is a set of compatible and analyzed frames, then every $FS \in \mathrm{Check}(\phi_{lc}(F_1) \cup \dots \cup \phi_{lc}(F_n), \{F_1, \dots, F_n\})$ will contain compatible, analyzed and checked frames.*

Proof. It is easy to see that compatibility is preserved by the procedure, and that they are still analyzed (no terms are added to the frames).

If $F \in FS$ then $\phi_{lc}(F) \subseteq \mathit{checks}(F)$, and we are done by Theorem 4. $\square$

As an immediate result, we get that there is no way to distinguish the frames that are grouped together by Check:

Lemma 8. *If $F_1, \dots, F_n$ are compatible and fully checked frames, for any F_i and any check $r_1 \doteq r_2$ we have that $F_i(r_1) =_E F_i(r_2)$ if and only if $F_1(r_1) =_E F_1(r_2) \wedge \dots \wedge F_n(r_1) =_E F_n(r_2)$.*

Proof. We have $F_i(r_1) =_E F_i(r_2)$ if and only if $\mathit{checks}(F_i) \models_E r_1 \doteq r_2$. Furthermore, by compatibility, we have $\mathit{checks}(F_1) = \dots = \mathit{checks}(F_n)$. $\square$

For a set of compatible, fully analyzed, and fully checked frames, the *recipe composition* problem is simply solved by $\mathit{compose}$:

Theorem 5. *If $F_1, \dots, F_n$ are compatible, fully checked, and fully analyzed frames, $t_1, \dots, t_n$ are constructive terms, and there exists an r such that $F_1(r) =_E t_1 \wedge \dots \wedge F_n(r) =_E t_n$, then $\mathit{compose}(F_1, t_1)$ is non-empty and for any $r' \in \mathit{compose}(F_1, t_1)$ we have $F_1(r') =_E t_1 \wedge \dots \wedge F_n(r') =_E t_n$.*

Proof. We get the non-emptiness of $\mathit{compose}(F_1, t_1)$ and that for each r' we have $F_1(r') =_E t_1$ by Lemma 4. Since $F_1(r) =_E F_1(r')$, we have by Lemma 8 that $F_2(r) =_E F_2(r') \wedge \dots \wedge F_n(r) =_E F_n(r')$, and are done. $\square$