Relating Process Languages for Security and Communication Correctness (Extended Abstract)

Daniele Nantes, Jorge Pérez

To cite this version:

Daniele Nantes, Jorge Pérez. Relating Process Languages for Security and Communication Correctness (Extended Abstract). 38th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE), Jun 2018, Madrid, Spain. pp.79-100, 10.1007/978-3-319-92612-4_5 . hal-01824820

HAL Id: hal-01824820
https://hal.inria.fr/hal-01824820

Submitted on 27 Jun 2018

Relating Process Languages for Security and Communication Correctness (Extended Abstract)*

Daniele Nantes$^{0000-0002-1959-87301}$ and Jorge A. Pérez$^{0000-0002-1452-61802}$

[1] Universidade de Brasília, Brazil
[2] University of Groningen & CWI, Amsterdam, The Netherlands

Abstract. Process calculi are expressive specification languages for concurrency. They have been very successful in two research strands: (a) the analysis of *security protocols* and (b) the enforcement of correct *message-passing programs*. Despite their shared foundations, languages and reasoning techniques for (a) and (b) have been separately developed. Here we connect two representative calculi from (a) and (b): we encode a (high-level) π-calculus for multiparty sessions into a (low-level) applied π-calculus for security protocols. We establish the correctness of our encoding, and we show how it enables the integrated analysis of security properties *and* communication correctness by re-using existing tools.

1 Introduction

This paper connects two distinct formal models of communicating systems: a process language for the analysis of *security protocols* [12], and a process language for *session-based concurrency* [9,10]. They are representative of two separate research strands:

(a) Process models for security protocols, such as [12] (see also [7]), rely on variants of the applied π-calculus [1] to establish properties related to process execution (e.g., secrecy and confidentiality). These models support cryptography and term passing, but lack support for high-level communication structures.

(b) Process models for session-based communication, such as [10] (see also [11]), use π-calculus variants equipped with type systems to enforce correct message-passing programs. Security extensions of these models target properties such as information flow and access control (cf. [2]), but usually abstract away from cryptography.

We present a *correct encoding* that connects two calculi from these two strands:

- A, a (low-level) applied π-calculus in which processes explicitly describe term communication, cryptographic operations, and state manipulation [12];
- S, a (high-level) π-calculus in which communication actions are organized as multiparty session protocols [10,5].

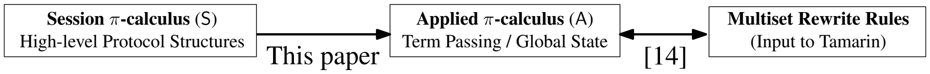
Our aim is to exploit the complementary strenghts of A and S to analyze communicating systems that feature high-level communication structures (as in session-based concurrency [9,10]) *and* use cryptographic operations and global state in protocol exchanges.

Our encoding of S into A describes how the structures typical of session-based, asynchronous concurrency can be compiled down, in a behavior-preserving manner, as

* Work partially funded by FAP-DF 0193.001381/2017.

process implementations in which communication of terms takes place exploiting rich equational theories and global state. To our knowledge, ours is the first work to relate process calculi for the analysis of communication-centric programs (S) and of security protocols (A), as developed in disjoint research strands.

We believe our results shed light on both (a) and (b). In one direction, they define a new way to reason about multiparty session processes. Process specifications in S can now integrate cryptographic operations and be analyzed by (re)using existing methods. In fact, since A processes can be faithfully translated into multiset rewriting rules using SAPIC [12] (which can in turn be fed into the Tamarin prover [14]), our encoding bridges the gap between S processes and toolsets for the analysis of security properties:


Session π-calculus (S)
High-level Protocol Structures
This paper
Applied π-calculus (A)
Term Passing / Global State
[14]
Multiset Rewrite Rules
(Input to Tamarin)

Interestingly, this connection can help to enforce communication correctness: we show how SAPIC/Tamarin can check *local formulas* representing local session types [10].

In the other direction, our approach allows us to enrich security protocol specifications with communication structures based on sessions. This is relevant because the analysis of security protocols is typically carried out on models such as, e.g., Horn clauses and rewriting rules, which admit efficient analysis but that lead to too low-level specifications. Our developments fit well in this context, as the structures intrinsic to session-based concurrency can conveniently describe communicating systems in which security protocols appear intertwined with higher-level interaction protocols.

This rest of the paper is organized as follows. § 2 introduces the *Two-Buyer Contract Signing Protocol*, a protocol that is representative of the kind of systems that is hard to specify using S or A alone. § 3 recalls the definitions of S and A, and also introduces $\mathsf{S}^\star$, which is a variant of S that is useful in our developments. § 4 defines the encoding of S into A, using $\mathsf{S}^\star$ as stepping stone, and establishes its correctness (Theorems 1, 2, and 3). § 5 shows how our encoding can be used to reduce the enforcement of protocol conformance in S to the model checking of local formulas for A (Theorems 4 and 5). § 6 revisits the Two-Buyer Contract Signing Protocol: we illustrate its process specification using S minimally extended with constructs from A, and show how key correctness properties can be mechanically verified using SAPIC/Tamarin. The paper closes by discussing related works and collecting concluding remarks (§ 7). Additional technical material and further examples are given in an appendix available online [15].

2 A Motivating Example: The Trusted Buyers-Seller Protocol

The *Trusted Buyers-Seller Protocol* extends the Two-Buyer Protocol [10], and proceeds in two phases. The first phase follows the global session type in [10], which offers a unified description of the way in which two buyers (B_1 and B_2) interact to purchase a book from a seller (S). In the second phase, once B_1 and B_2 agree in the terms of the purchase, the role of S is delegated to a *trusted third party* (T), which creates a contract for the transaction and collects the participants' signatures. This second phase relies on the *contract signing protocol* [8], which may resolve conflicts (due to unfulfilled promises

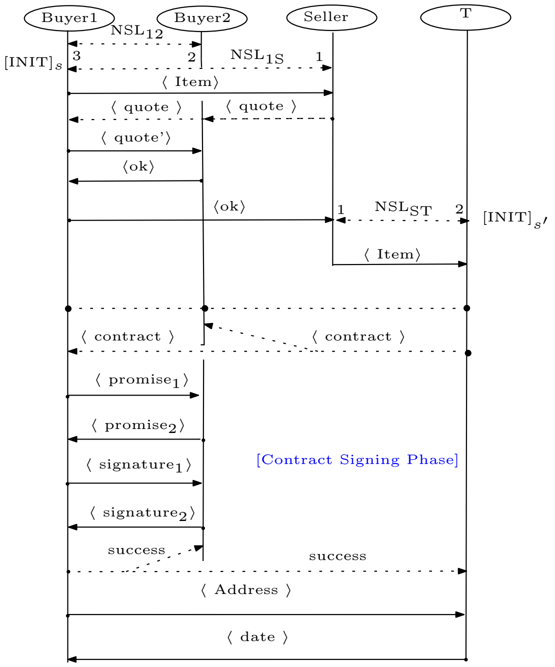


Fig. 1. The Trusted Buyers-Seller Protocol.

from B_1 and B_2) and abort the conversation altogether. In this protocol, one key security property is *authentication*, which ensures that an attacker cannot impersonate B_i, S, or T. Relevant properties of communication correctness include *fidelity* and *safety*: while the former ensures that processes for B_i, S, and T follow the protocols specified by global/local types, the latter guarantees that such processes do not get into errors at runtime. The protocol is illustrated in Fig. 1 and described next:

First Phase B_1, B_2, and S start by establishing a session, after executing the Needham-Schroeder-Lowe (NSL) authentication protocol. Subsequently, they interact as follows:

1. B_1 sends the book title to S. Then, S replies back to both B_1 and B_2 the quote for the title. Subsequently, B_1 tells B_2 how much he can contribute.
2. If the amount is within B_2's budget, then he accepts to perform the transaction, informs B_1 and S, and awaits the contract signing phase. Otherwise, if the amount offered by B_1 is not enough, B_2 informs S and B_1 his intention to abort the protocol.
3. Once B_1 and B_2 have agreed upon the purchase, S will *delegate* the session to the trusted party T, which will lead the contract signing phase. Upon completion of this phase, S (implemented by T) sends B_1 the delivery date for the book.

Second Phase At this point, the trusted authority T, B_1, and B_2 interact as follows:

4. T creates a new contract ct and a new memory cell s, useful to record information about the contract. T sends the contract ct to B_1 and B_2 for them to sign. T can start

replying to the following requests: `success` (in case of successful communication), `abort` (request to abort the protocol), or `resolve` (request to solve a conflict).

5. Upon reception of contract ct from T, B_1 sends to B_2 his promise to sign it. Subsequently, B_1 expects to receive B_2's promise:
   - If B_1 receives a valid response from B_2, his promise is converted into a signature ($\langle$signature$_1\rangle$), which is sent back. Now, B_1 expects to receive a valid signature from B_2: if this occurs, B_1 sends to T a `success` message; otherwise, B_1 sends T a `resolve` request, which includes the promise by B_2 and his own signature.
   - If B_1 does not receive a valid promise from B_2, then B_1 asks T to cancel the purchase (an `abort` request), including his own promise ($\langle$promise$_1\rangle$) in the request.
6. Upon reception of contract ct from T, B_2 checks whether he obtained a valid promise from B_1; in that case, B_2 replies by sending his promise to sign it ($\langle$promise$_2\rangle$). Now, B_2 expects to receive B_1's signature on ct: if the response is valid, B_2 sends its own signature ($\langle$signature$_2\rangle$) to B_1; otherwise, B_2 asks T to resolve. If B_2 does not receive a valid promise, then it aborts the protocol.

Clearly, S and A offer complementary advantages in modeling and analyzing the Trusted Buyers-Seller Protocol. On the one hand, S can represent high-level structures that are typical in the design of multiparty communication protocols. Such structures are essential in, e.g., the exchanges that follow session establishment in the first phase (which involves a step of session delegation to bridge with the second phase) and the handling of requests `success`, `abort` and `resolve` in the second phase. Hence, S and its type-based verification techniques can be used to establish fidelity and safety properties. However, S is not equipped with constructs for directly representing cryptographic operations, as indispensable in, e.g., the NSL protocol for session establishment and in the exchanges of signatures/promises in the contract sigining phase. The lack of these constructs prevents the formal analysis of authentication properties. On the other hand, A compensates for the shortcomings of S, for it can directly represent cryptographic operations on exchanged messages, as required to properly model the contract signing phase and, ultimately, to establish authentication. While A can represent the high-level communication structures mentioned above, it offers a too low-level representation of them, which makes reasoning about fidelity and safety more difficult than in S.

Our encoding from S into A, given in § 4, will serve to combine the individual strengths of both languages. In § 6, we will revisit this example: we will give a process specification using an extension of S with some constructs from A. This is consistent, because A is a low-level process language, and our encoding will define how to correctly compile S down to A (constructs from A will be treated homomorphically). Moreover, we will show how to use SAPIC/Tamarin to verify that implementations for B_1, B_2, S, and T respect their intended local types.

3 Two Process Models: A and S

3.1 The Applied π- calculus (A)

Preliminaries As usual in symbolic protocol analysis, messages are modelled by abstract terms $(t, t', \ldots)$. We assume a countably infinite set of variables $\mathcal{V}$, a countably

$$\begin{array}{rl} M, N ::= & x, y \mid p \mid n \mid f(M_1, \ldots, M_n) \quad (f \in \Sigma) \\ P, Q ::= & \mathbf{0} \mid \texttt{out}(M, N); P \mid \texttt{in}(M, N); P \mid \ P \mid Q \mid !P \mid \nu n; P \mid \\ & \texttt{insert}((M, N)); P \mid \texttt{delete}\ M; P \mid \texttt{lookup}\ M\ \texttt{as}\ x\ \texttt{in}\ P\ \texttt{else}\ Q \mid \\ & \texttt{lock}\ M; P \mid \texttt{unlock}\ M; P \mid \texttt{event}\ F; P \mid \texttt{if}\ M = N\ \texttt{then}\ P\ \texttt{else}\ Q \end{array}$$

Table 1. Syntax of A: Terms and Processes.

infinite set of names $\mathcal{N} = \text{PN} \cup \text{FN}$ (FN for fresh names, PN for public names), and a signature Σ (a set of function symbols, each with its arity).

We denote by $\mathcal{T}_\Sigma$ the set of well-sorted terms built over Σ, $\mathcal{N}$, and $\mathcal{V}$. The set of ground terms (i.e., terms without variables) is denoted $\mathcal{M}_\Sigma$. A substitution is a partial function from variables to terms. We denote by $\sigma = \{t_1/x_1, \ldots, t_n/x_n\}$ the substitution whose domain is $Dom(\sigma) = \{x_1, \ldots, x_n\}$. We say σ is *grounding* for t if $t\sigma$ is ground. We equip the term algebra with an equational theory $=_E$, which is the smallest equivalence relation containing identities in E, a finite set of pairs the form $M = N$ where $M, N \in \mathcal{T}_\Sigma$, that is closed under application of function symbols, renaming of names, and substitution of variables by terms of the same sort. Furthermore, we require E to distinguish different fresh names, i.e., $\forall a, b \in FN : a \neq b \Rightarrow a \neq_E b$.

Given a set S, we write S^* and $S^\#$ to denote the sets of finite sequences of elements and of finite multisets of elements from S. We use the superscript $\#$ to annotate the usual multiset operations, e.g., $S_1 \cup^\# S_2$ denotes the union of multisets S_1, S_2. Application of substitutions is extended to sets, multisets, and sequences as expected.

The set of *facts* is $\mathcal{F} := \{F(t_1, \ldots, t_k) |\ t_i \in \mathcal{T}_\Sigma, F \in \Sigma_{fact}$ of arity $k\}$, where Σ_{fact} is an unsorted signature, disjoint from Σ. Facts will be used to annotate protocols (via events) and to define multiset rewrite rules. A fixed set of fact symbols will be used to encode the adversary's knowledge, freshness information, and the messages on the network. The remaining fact symbols are used to represent the protocol state. For instance, fact $K(m)$ denotes that m is known by the adversary.

Syntax and Semantics The grammar for terms (M, N) and processes (P, Q), given in Table 1, follows [12]. In addition to usual operators for concurrency, replication, and name creation, the calculus A inherits from the applied π-calculus [1] input and output constructs in which terms appear both as communication subjects and objects. Also, A includes a conditional construct based on term equality, as well as constructs for reading from and updating an explicit *global state*:

- $\texttt{insert}((M, N)); P$ first binds the value N to a key M and then proceeds as P. Successive inserts may modify this binding; $\texttt{delete}\ M; P$ simply "undefines" the mapping for the key M and proceeds as P.
- $\texttt{lookup}\ M\ \texttt{as}\ x\ \texttt{in}\ P\ \texttt{else}\ Q$ retrieves the value associated to M, binding it to variable x in P. If the mapping is undefined for M then the process behaves as Q.
- $\texttt{lock}\ M; P$ and $\texttt{unlock}\ M; P$ allow to gain and release exclusive access to a resource/key M, respectively, and to proceed as P afterwards. These operations are essential to specify parallel processes that may read/update a common memory.

Moreover, the construct $\texttt{event}\ F; P$ adds $F \in \mathcal{F}$ to a multiset of ground facts before proceeding as P. These facts will be used in the transition semantics for A, which is de-

$$\frac{a \in (\text{FN} \cup \text{PN}) \setminus \tilde{n}}{\nu\tilde{n}.\sigma \vdash a}\ \text{[Name]} \quad \frac{\nu\tilde{n}.\sigma \vdash t \quad t =_E t'}{\nu\tilde{n}.\sigma \vdash t'}\ \text{[Eq]} \quad \frac{x \in Dom(\sigma)}{\nu\tilde{n}.\sigma \vdash x\sigma}\ \text{[Frame]} \quad \frac{\nu\tilde{n}.\sigma \vdash t_i}{\nu\tilde{n}.\sigma \vdash f\widetilde{t}}\ \text{[App]}$$

Table 2. Deduction rules for A. In Rule [Appl]: $\widetilde{t} = (t_1, \ldots, t_n)$.

fined by a labelled relation between *process configurations* of the form $(\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})$, where: $\mathcal{P}$ is a multiset of ground processes representing the processes executed in parallel; $\mathcal{E} \subseteq FN$ is the set of fresh names generated by the processes; $\mathcal{S} : \mathcal{M}_\Sigma \to \mathcal{M}_\Sigma$ is a partial function modeling stored information (state); σ is a ground substitution modeling the messages sent to the environment; and $\mathcal{L} \subseteq \mathcal{M}_\Sigma$ is the set of currently acquired locks. We write $\mathcal{S}(M) = \bot$ to denote that there is no information stored for M in $\mathcal{S}$. Also, notation $\mathcal{L} \backslash M$ stands for the set $\mathcal{L} \backslash \{M' | M' =_E M\}$.

We also require the notions of *frame* and a *deduction relation*. A frame $\nu\tilde{n}.\sigma$ consists of a set of fresh names $\tilde{n}$ and a substitution σ: it represents the sequence of messages that have been observed by an adversary during a protocol execution and secrets $\tilde{n}$ generated by the protocol, a priori unknown to the adversary. The deduction relation $\nu\tilde{n}.\sigma \vdash t$ models the adversary's ability to compute new messages from observed ones: it is the smallest relation between frames and terms defined by the rules in Table 2.

Transitions are of the form $(\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L}) \xrightarrow{\mathcal{F}}_{\mathsf{A}} (\mathcal{E}', \mathcal{S}', \mathcal{P}', \sigma', \mathcal{L}')$, where $\mathcal{F}$ is a set of ground facts (see Table 3). We write $\longrightarrow_{\mathsf{A}}$ for $\xrightarrow{\emptyset}_{\mathsf{A}}$ and $\xrightarrow{f}_{\mathsf{A}}$ for $\xrightarrow{\{f\}}_{\mathsf{A}}$. As usual, $\longrightarrow^*_{\mathsf{A}}$ denotes the reflexive, transitive closure of $\longrightarrow_{\mathsf{A}}$. Transitions denote either standard process operations or operations on the global state; they are sometimes denoted $\longrightarrow_{\mathsf{A}_P}$ and $\longrightarrow_{\mathsf{A}_S}$, respectively.

3.2 Multiparty Session Processes (S)

Syntax The syntax of *processes*, ranged over by $P, Q, \ldots$ and that of *expressions*, ranged over by $e, e', \ldots$, is given by the grammar of Table 4, which also shows name conventions. We assume two disjoint countable set of names: one ranges over *shared names* $a, b, \ldots$ and another ranges over *session names* $s, s', \ldots$. Variables range over $x, y, \ldots$; *participants* (or *roles*) range over the naturals and are denoted as $\mathtt{p}, \mathtt{q}, \mathtt{p}', \ldots$; *labels* range over $l, l', \ldots$ and *constants* range over $\mathtt{true}, \mathtt{false}, \ldots$. We write $\widetilde{\mathtt{p}}$ to denote a finite sequence of participants $\mathtt{p}_1, \ldots, \mathtt{p}_n$ (and similarly for other elements). Given a session name s and a participant $\mathtt{p}$, we write $s[\mathtt{p}]$ to denote a *(session) endpoint*.

The intuitive meaning of processes is as in [10,5]. The processes $\overline{u}[\mathtt{p}](y).P$ and $u[\mathtt{p}](y).Q$ can respectively request and accept to initiate a session through a shared name u. In both processes, the bound variable y is the placeholder for the channel that will be used in communications. After initiating a session, each channel placeholder will replaced by an endpoint of the form $s[\mathtt{p}_i]$ (i.e., the runtime channel of $\mathtt{p}_i$ in session s). Within an established session, process may send and receive basic values or session names (*session delegation*) and select and offer labeled, deterministic choices (cf. constructs $c \oplus \langle \mathtt{p}, l \rangle.P$ and $c \,\&(\mathtt{p}, \{l_i : P_i\}_{i \in I})$). The input/output operations (including delegation) specify the channel and the sender or the receiver, respectively.

Standard Operations

$$
\begin{array}{rcll}
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathbf{0}\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L}) & \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P \mid Q\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P,Q\},\sigma,\mathcal{L}) & \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{!P\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{!P,P\},\sigma,\mathcal{L}) & \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\nu a;P\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & & \\
 & & (\mathcal{E}\cup\{a'\},\mathcal{S},\mathcal{P}\cup^{\#}\{P\{a'/a\}\},\sigma,\mathcal{L}) & \text{C0} \\
(\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L}) & \xrightarrow{K(M)}_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P},\sigma,\mathcal{L}) & \text{C1} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathtt{out}(M,N);P\},\sigma,\mathcal{L}) & \xrightarrow{K(M)}_{\mathsf{A}} & & \\
 & & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma\cup\{N/x\},\mathcal{L}) & \text{C2} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathtt{in}(M,N);P\},\sigma,\mathcal{L}) & \xrightarrow{K(\langle M,N\tau\rangle)}_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\tau\},\sigma,\mathcal{L}) & \text{C3} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathtt{out}(M,N);P,\mathtt{in}(M',N');Q\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P,Q\tau\},\sigma,\mathcal{L}) & \text{C4} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathtt{if}\ M=N\ \mathtt{then}\ P\ \mathtt{else}\ Q\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}) & \text{C5} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathtt{if}\ M=N\ \mathtt{then}\ P\ \mathtt{else}\ Q\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{Q\},\sigma,\mathcal{L}) & \text{C6} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathtt{event}\ F;P\},\sigma,\mathcal{L}) & \xrightarrow{F}_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}) &
\end{array}
$$

Operations on Global State

$$
\begin{array}{rcll}
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathtt{insert}(\!(M,N)\!);P\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & (\mathcal{E},\mathcal{S}[M\mapsto N],\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}) & \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathtt{delete}\ M;P\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & (\mathcal{E},\mathcal{S}[M\mapsto\bot],\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}) & \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathtt{lookup}\ M\ \mathtt{as}\ x\ \mathtt{in}\ P\ \mathtt{else}\ Q\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\{V/x\}\},\sigma,\mathcal{L}) & \text{C7} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathtt{lookup}\ M\ \mathtt{as}\ x\ \mathtt{in}\ P\ \mathtt{else}\ Q\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{Q\},\sigma,\mathcal{L}) & \text{C8} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathtt{lock}\ M;P\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}\cup\{M\}) & \text{C9} \\
(\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{\mathtt{unlock}\ M;P\},\sigma,\mathcal{L}) & \longrightarrow_{\mathsf{A}} & (\mathcal{E},\mathcal{S},\mathcal{P}\cup^{\#}\{P\},\sigma,\mathcal{L}\backslash M) &
\end{array}
$$

where:

C0: if a' fresh
C1: if $\nu\mathcal{E}.\sigma \vdash M$
C2: if x is fresh, $\nu\mathcal{E}.\sigma \vdash M$
C3: if $\exists\tau.\nu\mathcal{E}.\sigma \vdash M$ and $\nu\mathcal{E}.\sigma \vdash N\tau$ and τ grounding for N
C4: if $M =_E M'$ and $\exists\tau.N =_E N'\tau$ and τ grounding for N'
C5: if $M =_E N$
C6: if $M \neq_E N$
C7: if $\exists N.N =_E M$ and $\mathcal{S}(N) =_E V$
C8: if $\forall N.N =_E M \Rightarrow \mathcal{S}(N) = \bot$
C9: if $M \notin_E \mathcal{L}$

Table 3. Operational Semantics for A.

$$
u ::= x \mid a \ \ \text{(Identifiers)} \qquad n ::= s \mid a \ \ \text{(Names)} \qquad e ::= v \mid x \mid e = e' \mid \ldots \ \ \text{(Expressions)}
$$

$$
c ::= s[\mathtt{p}] \mid x \ \ \text{(Channels)} \qquad v ::= a \mid \mathtt{true} \mid \mathtt{false} \mid s[\mathtt{p}] \ \ \text{(Values)}
$$

$$
m ::= (\mathtt{q} \triangleright \mathtt{p} : v) \mid (\mathtt{q} \triangleright \mathtt{p} : c) \mid (\mathtt{q} \triangleright \mathtt{p} : l) \ \ \text{(Messages)}
$$

$$
\begin{array}{rlllll}
P ::= & \overline{u}[\mathtt{p}](y).P & \text{(Req)} & \mid c!\langle\!\langle \mathtt{p}, c\rangle\!\rangle.P & \text{(Deleg)} & \mid P|Q \quad \text{(Parallel)} \\
\mid & u[\mathtt{p}](y).P & \text{(Acc)} & \mid c?(\!(\mathtt{q}, y)\!).P & \text{(Recep)} & \mid \mathbf{0} \quad \text{(Inaction)} \\
\mid & c!\langle \mathtt{p}, e\rangle.P & \text{(Send)} & \mid c \oplus \langle \mathtt{p}, l\rangle.P & \text{(Select)} & \mid (\nu n)P \quad \text{(N.Hiding)} \\
\mid & c?(\mathtt{p}, x).P & \text{(Recv)} & \mid c\,\&(\mathtt{p}, \{l_i : P_i\}_{i\in I}) & \text{(Branch)} & \mid s[\widetilde{\mathtt{p}}] : h \quad \text{(M. Queue)} \\
 & & & \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q & \text{(Condit.)} & h ::= h \cdot m \mid \emptyset \quad \text{(Queue)}
\end{array}
$$

Table 4. Process syntax and naming conventions for S.

Message queues model asynchronous communication. A message $(\mathtt{p} \triangleright \mathtt{q} : v)$ indicates that $\mathtt{p}$ has sent a value v to $\mathtt{q}$. The empty queue is denoted by $\emptyset$. By $h \cdot m$ we denote the queue obtained by concatenating message m to the queue h. By $s[\widetilde{\mathtt{p}}] : h$ we

$$
\begin{array}{c}
P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad (\nu a)\mathbf{0} \equiv \mathbf{0} \qquad (\nu s)(s:\emptyset) \equiv \mathbf{0} \\
(\nu r)P \mid Q \equiv (\nu r)(P \mid Q), \text{ if } r \notin fn(Q) \qquad (\nu r)(\nu r')P \equiv (\nu r')(\nu r)P, \text{ where } \quad r ::= a \mid s \\
s[\widetilde{\mathtt{p}}] : h \cdot (\mathtt{q} \triangleright \mathtt{p} : \zeta) \cdot (\mathtt{q}' \triangleright \mathtt{p}' : \zeta') \cdot h' \equiv s[\widetilde{\mathtt{p}}] : h \cdot (\mathtt{q}' \triangleright \mathtt{p}' : \zeta') \cdot (\mathtt{q} \triangleright \mathtt{p} : \zeta) \cdot h', \text{ if } \mathtt{p} \neq \mathtt{p}' \text{ or } \mathtt{q} \neq \mathtt{q}'
\end{array}
$$

Table 5. Structural Congruence for S Processes.

denote the queue h of the session s initiated between participants $\widetilde{\mathtt{p}} = \mathtt{p}_1, \ldots, \mathtt{p}_n$; when the participants are clear from the context we shall write $s : h$ instead of $s[\widetilde{\mathtt{p}}] : h$.

Request/accept actions bind channel variables, value receptions bind value variables, channel receptions bind channel variables, hidings bind shared and session names. In $(\nu s)P$ all occurrences of $s[\mathtt{p}]$ and queue s inside P are bound. We denote by $fn(Q)$ the set of free names in Q. A process is *closed* if it does not contain free variables or free session names. Unless stated otherwise, we only consider closed processes.

Semantics S processes are governed by a reduction semantics, which relies on a *structural congruence* relation, denoted $\equiv$ and defined by adding α-conversion to the rules of Table 5. Reduction rules are given in Table 6; we write $P \longrightarrow_{\mathsf{S}} P'$ for a reduction step. We rely on the following syntax for contexts: $E ::= [\,] \mid P \mid (\nu a)E \mid (\nu s)E \mid E \mid E$.

We briefly discuss the reduction rules. Rule [Init] describes the initiation of a new session among n participants that synchronize over the shared name a. After session initiation, the participants will share a private session name (s in the rule), and an empty queue associated to it ($s[\widetilde{\mathtt{p}}] : \emptyset$ in the rule). Rules [Send], [Deleg] and [Sel] add values, channels and labels, respectively, into the message queue; in Rule [Send], $e \downarrow v$ denotes the evaluation of the expression e into a value v. Rules [Recv], [SRecv] and [Branch] perform complementary de-queuing operations. Other rules are self-explanatory.

3.3 The Calculus $\mathsf{S}^\star$

We now introduce $\mathsf{S}^\star$, a variant of S which will simplify the definition of our encoding into A. The syntax of $\mathsf{S}^\star$ processes is as follows:

$$
\begin{array}{rl}
P, Q ::= & \mathbf{0} \mid \overline{u}[\mathtt{p}](\widetilde{y}).P \mid u[\mathtt{p}](\widetilde{y}).P \mid P \mid Q \mid (\nu n)P \mid \text{if } e \text{ then } P \text{ else } Q \\
\mid & c_{\mathtt{pq}}!\langle e : \mathtt{msg}\rangle.P \mid c_{\mathtt{pq}}?(\!(y)\!).P \mid c_{\mathtt{pq}}?(x).P \mid c_{\mathtt{pq}}!\langle\!\langle c'_{\mathtt{p}'\mathtt{q}'} : \mathtt{chan}\rangle\!\rangle.P \mid \\
\mid & c_{\mathtt{pq}} \oplus \langle l : \mathtt{lbl}\rangle.P \mid c_{\mathtt{pq}} \,\&(\{l_i : P_i\}_{i \in I}) \mid s_{\mathtt{pq}} : h
\end{array}
$$

where $c_{\mathtt{pq}}$ denotes a channel annotated with participant identities, $h ::= h \cdot m \mid \emptyset$ and $m ::= \langle \mathtt{msg}, v\rangle \mid \langle \mathtt{chan}, s_{\mathtt{pq}}\rangle \mid \langle \mathtt{lbl}, l\rangle$. The main differences between S and $\mathsf{S}^\star$ are:

- Intra-session communication relies on annotated channels, and output prefixes include a *sort* for the communicated messages (`msg` for values, `chan` for delegated sessions, `lbl` for labels).
- While S uses a single queue per session, in $\mathsf{S}^\star$ for each pair of participants there will be two queues, one in each direction. This simplifies the definition of structural congruence $\equiv$ for $\mathsf{S}^\star$, which results from that for S as expected and is omitted.
- Constructs for session request and acceptance in $\mathsf{S}^\star$ depend on a sequence of variables, rather than on a single variable. In these constructs, denoted $\overline{u}[\mathtt{p}](\widetilde{y}).P$ and $u[\mathtt{p}](\widetilde{y}).P$, respectively, $\widetilde{y}$ is a sequence of variables of the form $y_{\mathtt{pq}}$, for some $\mathtt{p}$, $\mathtt{q}$.

$$
\begin{array}{rr}
a[\mathtt{p}_1](y)P_1 \mid \ldots \mid a[\mathtt{p}_{n-1}](y)P_{n-1} \mid \overline{a}[\mathtt{p}_n](y).P_n \longrightarrow_{\mathsf{S}} & \text{[Init]} \\
(\nu s)(P_1\{s[\mathtt{p}_1]/y\} \mid \ldots \mid P_{n-1}\{s[\mathtt{p}_{n-1}]/y\} \mid P_n\{s[\mathtt{p}_n]/y\} \mid s[\widetilde{\mathtt{p}}] : \emptyset) & \\
s[\mathtt{p}]!\langle \mathtt{q}, e\rangle.P \mid s : h \longrightarrow_{\mathsf{S}} P \mid s{:}h{\cdot}(\mathtt{p} \triangleright \mathtt{q}{:}v)\ \ (e \downarrow v) & \text{[Send]} \\
s[\mathtt{p}]!\langle\langle \mathtt{q}, s'[\mathtt{p}']\rangle\rangle.P \mid s : h \longrightarrow_{\mathsf{S}} P \mid s{:}h \cdot (\mathtt{p} \triangleright \mathtt{q}{:}s'[\mathtt{p}']) & \text{[Deleg]} \\
s[\mathtt{p}] \oplus \langle \mathtt{q}, l\rangle.P \mid s : h \longrightarrow_{\mathsf{S}} P \mid s{:}h{\cdot}(\mathtt{p} \triangleright \mathtt{q}{:}l) & \text{[Sel]} \\
s[\mathtt{p}]?(\mathtt{q}, x).P \mid s{:}(\mathtt{q} \triangleright \mathtt{p}{:}v){\cdot}h \longrightarrow_{\mathsf{S}} P\{v/x\} \mid s[\widetilde{\mathtt{p}}] : h & \text{[Recv]} \\
s[\mathtt{p}]?((\mathtt{q}, y)).P \mid s : (\mathtt{q} \triangleright \mathtt{p}{:}s'[\mathtt{p}']) \cdot h \longrightarrow_{\mathsf{S}} P\{s'[\mathtt{p}']/y\} \mid s[\widetilde{\mathtt{p}}] : h & \text{[SRecv]} \\
s[\mathtt{p}]\,\&(\mathtt{q}, \{l_i : P_i\}_{i \in I}) \mid s{:}(\mathtt{q} \triangleright \mathtt{p}{:}l_j){\cdot}h \longrightarrow_{\mathsf{S}} P_j \mid s : h\ (j \in I) & \text{[Branch]} \\
\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow_{\mathsf{S}} P \quad (e \downarrow \mathtt{true}) & \text{[If-T]} \\
P \equiv P'\ \text{and}\ \ P' \longrightarrow_{\mathsf{S}} Q'\ \text{and}\ Q \equiv Q' \Rightarrow P \longrightarrow_{\mathsf{S}} Q & \text{[Str]} \\
P \longrightarrow_{\mathsf{S}} P' \Rightarrow E[P] \longrightarrow_{\mathsf{S}} E[P'] & \text{[Ctx]}
\end{array}
$$

Table 6. Reduction rules for S (Rule [If-F] omitted).

With these differences in mind, the reduction semantics for $\mathsf{S}^\star$, denoted $\longrightarrow_{\mathsf{S}^\star}$, follows that for S (Table 6). Reduction rules for $\mathsf{S}^\star$ include the following:

$$
\begin{array}{rr}
a[1](\widetilde{y_1}).P_1 \mid \ldots \mid a[n-1](\widetilde{y_{n-1}}).P_{n-1} \mid \overline{a}[n](\widetilde{y_n}).P_n \longrightarrow_{\mathsf{S}^\star} & \text{[Init}^*\text{]} \\
(\nu s)(P_1\{s/y\} \mid \ldots \mid P_{n-1}\{s/y\} \mid P_n\{s/y\} \mid \widetilde{y_1}\{s/y\} : \emptyset \mid \ldots \mid \widetilde{y_n}\{s/y\} : \emptyset) & \\
y_{\mathtt{pq}}!\langle e : \mathtt{msg}\rangle.P \mid y_{\mathtt{pq}} : h \longrightarrow_{\mathsf{S}^\star} P \mid y_{\mathtt{pq}} : h \cdot \langle \mathtt{msg}, v\rangle \qquad (e \downarrow v) & \text{[Send}^*\text{]} \\
y_{\mathtt{pq}}?(x).P \mid y_{\mathtt{qp}} : \langle \mathtt{msg}, v\rangle \cdot h \longrightarrow_{\mathsf{S}^\star} P\{v/x\} \mid y_{\mathtt{qp}} : h & \text{[Recv}^*\text{]}
\end{array}
$$

Notice that in Rule [Init*], we only need to write $P_i\{s/y\}$: after reduction, these variables will be of the form $s_{\mathtt{pq}}$. In that rule, each $\widetilde{y_i}\{s/y\} : \emptyset$ denotes several queues (one for each name $y_{\mathtt{pq}} \in \widetilde{y_i}$), rather than a single queue.

It is straightforward to define an auxiliary encoding $\llparenthesis \cdot \rrparenthesis : \mathsf{S} \mapsto \mathsf{S}^\star$. For instance:

$$
\begin{array}{ll}
\llparenthesis s[\mathtt{p}]!\langle \mathtt{q}, e\rangle.P \rrparenthesis = s_{\mathtt{pq}}!\langle e : \mathtt{msg}\rangle.\llparenthesis P \rrparenthesis & \llparenthesis s[\mathtt{p}]?(\mathtt{q}, x).P \rrparenthesis = s_{\mathtt{qp}}?(x).\llparenthesis P \rrparenthesis \\
\llparenthesis s[\mathtt{p}]!\langle\langle \mathtt{q}, z_{\mathtt{p}'}\rangle\rangle.P \rrparenthesis = s_{\mathtt{pq}}!\langle\langle z_{p'} : \mathtt{chan}\rangle\rangle.\llparenthesis P \rrparenthesis & \llparenthesis s[\mathtt{p}]?((\mathtt{q}, x)).P \rrparenthesis = s_{\mathtt{qp}}?((x)).\llparenthesis P \rrparenthesis
\end{array}
$$

The full encoding, given in [15], enjoys the following property:

Theorem 1. *Let $P \in \mathsf{S}$. Then: (a) If $P \longrightarrow_{\mathsf{S}} P'$, then $\llparenthesis P \rrparenthesis \longrightarrow_{\mathsf{S}^\star} \llparenthesis P' \rrparenthesis$.*
(b) If $\llparenthesis P \rrparenthesis \longrightarrow_{\mathsf{S}^\star} R$, then there exists $P' \in \mathsf{S}$ such that $P \longrightarrow_{\mathsf{S}} P'$ and $\llparenthesis P' \rrparenthesis = R$.

Given the encoding $\llparenthesis \cdot \rrparenthesis : \mathsf{S} \mapsto \mathsf{S}^\star$ and Theorem 1 above, we now move on to define an encoding $[\![\cdot]\!] : \mathsf{S}^* \mapsto \mathsf{A}$. By composing these encodings (and their correctness results—Theorems 2 and 3), we will obtain a behavioral-preserving compiler of S into A.

4 Encoding $\mathsf{S}^\star$ Into A

We now present our encoding $[\![\cdot]\!] : \mathsf{S}^* \mapsto \mathsf{A}$ and establish its correctness. The encoding is defined in Table 7; it uses the set of facts $F_{\mathsf{S}} = \{\mathsf{honest}, \mathsf{sndnonce}, \mathsf{rcvnonce}, \mathsf{sndchann}, \mathsf{rcvchann}, \mathsf{out}, \mathsf{inp}, \mathsf{dels}, \mathsf{recs}, \mathsf{sel}, \mathsf{bra}, \mathsf{close}\}$. Facts will be used as event annotations in process executions, and also for model checking communication correctness via trace formulas in the following section. Our encoding will rely on the equational theory for `pairing`, which is embedded in Tamarin prover [14], and includes function symbols $\langle \_, \_\rangle$, `fst` and `snd`, for pairing and projection of first and second parameters of a pair. Communication within a secure established session is expressed by the manipulation of queues, which will be stored in the set of states $\mathcal{S}$. In SAPIC, we implement queues $y_{\mathtt{pq}}$ and $y_{\mathtt{qp}}$ as $q(y, \mathtt{p}, \mathtt{q})$ and $q(y, \mathtt{q}, \mathtt{p})$, respectively, where q is a function symbol for queues. Also, $s_{\mathtt{pq}} : \emptyset$ is implemented as $\mathtt{insert}\langle\langle s_{\mathtt{pq}}, \mathtt{init}\rangle\rangle$.

Implementing Session Establishment

$$
\begin{aligned}
\llbracket \overline{a}[3](\widetilde{y_3}).P \rrbracket &= \nu s; P_{31}; P_{32}; \texttt{insert}(\!(\widetilde{s_{ij}}, \emptyset)\!); \texttt{event}\ \mathsf{init}(\widetilde{s_{ij}});\\
&\quad \texttt{event}\ \mathsf{sndchann}(pk(ska_{31}), pk(y_1), s); \texttt{out}(u_1, s);\\
&\quad \texttt{event}\ \mathsf{sndchann}(pk(ska_{32}), pk(y_2), s); \texttt{out}(u_2, s); \llbracket P \rrbracket\\
P_{3i} &= \nu ska_{3i}; \texttt{out}(c, pk(ska_{3i})); \texttt{event}\ \mathsf{honest}(pk(ska_{3i})); \texttt{in}(c, pk(y_i));\\
&\quad \nu n_{31}; \texttt{event}\ \mathsf{sndnonce}(pk(ska_{3i}), pk(y_i), aenc(\langle n_{3i}, pk(ska_{3i})\rangle, pk(y_i)))\\
&\quad \texttt{out}(c, aenc(\langle n_{3i}, pk(ska_{3i})\rangle, pk(y_i)))); \texttt{in}(c, aenc(\langle n_{3i}, u_i, pk(y_i)\rangle, pk(ska_{3i})));\\
&\quad \texttt{event}\ \mathsf{rcvnonce}(pk(y_i), pk(ska_{3i}), aenc(\langle n_{3i}, u_i, pk(y_i)\rangle, pk(ska_{3i})))\\
\llbracket a[i](\widetilde{y_i}).P \rrbracket &= \nu\ ska_i; \texttt{in}(c, pk(x_i)); \texttt{event}\ \mathsf{honest}(pk(ska_i)); \texttt{in}(c, aenc(\langle y, pk(x_i)\rangle, pk(ska_i)));\\
&\quad \texttt{event}\ \mathsf{rcvnonce}(pk(x_i), pk(ska_i), aenc(\langle y, pk(x_i)\rangle, pk(ska_i)))\\
&\quad \nu n_i; \texttt{event}\ \mathsf{sndnonce}(pk(ska_i), pk(x_i), aenc(\langle y, n_i, pk(ska_i)\rangle, pk(x_i)))\\
&\quad \texttt{out}(c, aenc(\langle y, n_i, pk(ska_i)\rangle, pk(x_i))); \texttt{in}(n_i, z);\\
&\quad \texttt{event}\ \mathsf{rcvchann}(pk(x_i), pk(ska_i), z); \llbracket P \rrbracket
\end{aligned}
$$

Implementing Intra-Session Communication

$$
\begin{aligned}
\llbracket c_{\mathsf{pq}}!\langle e : \texttt{msg}\rangle.P \rrbracket &= \texttt{lock}\ c_{\mathsf{pq}}; \texttt{lookup}\ c_{\mathsf{pq}}\ \texttt{as}\ x\ \ \texttt{in}\,(\texttt{insert}(\!(c_{\mathsf{pq}}, x \cdot \langle \texttt{msg}, v\rangle)\!));\\
&\quad \texttt{event}\ \mathsf{out}(c_{\mathsf{pq}}, v); \texttt{unlock}\ c_{\mathsf{pq}}; \llbracket P \rrbracket \qquad e \downarrow v\\
\llbracket c_{\mathsf{pq}}?(x).P \rrbracket &= \texttt{lock}\ c_{\mathsf{qp}}; \texttt{lookup}\ c_{\mathsf{qp}}\ \texttt{as}\ z_v\ \ \texttt{in}\,(\texttt{if}\,\mathit{fst}(z_v) = \langle \texttt{msg}, z\rangle\ \texttt{then}\\
&\quad (\texttt{insert}(\!(c_{\mathsf{qp}}, \mathit{snd}(z_v))\!); \texttt{event}\ \mathsf{inp}(c_{\mathsf{pq}}, \mathit{fst}(z_v)); \texttt{unlock}\ c_{\mathsf{qp}}; \llbracket P\{z/x\} \rrbracket))\\
\llbracket c_{\mathsf{pq}}!\langle\!\langle c' : \mathsf{chan}\rangle\!\rangle.P \rrbracket &= \texttt{lock}\ c_{\mathsf{pq}}; \texttt{lookup}\ c_{\mathsf{pq}}\ \texttt{as}\ x\ \ \texttt{in}\,(\texttt{insert}(\!(c_{\mathsf{pq}}, x \cdot \langle \mathsf{chan}, c'\rangle)\!));\\
&\quad \texttt{event}\ \mathsf{dels}(c_{\mathsf{pq}}, c'); \texttt{unlock}\ c_{\mathsf{pq}}; \llbracket P \rrbracket\\
\llbracket c_{\mathsf{pq}}?(\!(x)\!).P \rrbracket &= \texttt{lock}\ c_{\mathsf{qp}}; \texttt{lookup}\ c_{\mathsf{qp}}\ \texttt{as}\ z_v\ \ \texttt{in}\,(\texttt{if}\,\mathit{fst}(z_v) = \langle \mathsf{chan}, z\rangle\ \texttt{then}\\
&\quad (\texttt{insert}(\!(c_{\mathsf{qp}}, \mathit{snd}(z_v))\!); \texttt{event}\ \mathsf{recs}(c_{\mathsf{pq}}, \mathit{fst}(z_v)); \texttt{unlock}\ c_{\mathsf{qp}}; \llbracket P\{z/x\} \rrbracket)\\
\llbracket c_{\mathsf{pq}} \oplus \langle l : \texttt{lbl}\rangle.P \rrbracket &= \texttt{lock}\ c_{\mathsf{pq}}; \texttt{lookup}\ c_{\mathsf{pq}}\ \texttt{as}\ x\ \ \texttt{in}\,(\texttt{insert}(\!(c_{\mathsf{pq}}, x \cdot \langle \texttt{lbl}, l\rangle)\!));\\
&\quad \texttt{event}\ \mathsf{sel}(c_{\mathsf{pq}}, l); \texttt{unlock}\ c_{\mathsf{pq}}; \llbracket P \rrbracket\\
\llbracket c_{\mathsf{pq}}\&(\{l_i : P_i\}) \rrbracket &= \texttt{lock}\ c_{\mathsf{qp}}; \texttt{lookup}\ c_{\mathsf{qp}}\ \texttt{as}\ z_l\ \texttt{in}\,\big(\texttt{if}\,\mathit{fst}(z_l) = \langle \texttt{lbl}, l_1\rangle\ \texttt{then}\\
&\qquad \texttt{insert}(\!(c_{\mathsf{qp}}, \mathit{snd}(z_l))\!); \texttt{event}\ \mathsf{bra}(c_{\mathsf{pq}}, l_1); \texttt{unlock}\ c_{\mathsf{pq}}; \llbracket P_1 \rrbracket\\
&\quad\ \texttt{else}\ \ \texttt{if}\,\mathit{fst}(z_l) = \langle \texttt{lbl}, l_2\rangle\ \texttt{then}\\
&\qquad \texttt{insert}(\!(c_{\mathsf{qp}}, \mathit{snd}(z_l))\!); \texttt{event}\ \mathsf{bra}(c_{\mathsf{pq}}, l_2); \texttt{unlock}\ c_{\mathsf{qp}}; \llbracket P_2 \rrbracket\big)
\end{aligned}
$$

$$
\llbracket \mathbf{0} \rrbracket = \texttt{event}\ \mathsf{close} \qquad \llbracket s[\widetilde{\mathtt{p}}] : h \rrbracket = \mathbf{0}
$$

$$
\llbracket (\nu s)P \rrbracket = \nu s; \llbracket P \rrbracket \quad \llbracket P \mid Q \rrbracket = \llbracket P \rrbracket \mid \llbracket Q \rrbracket \quad \llbracket \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \rrbracket = \texttt{if}\ e\ \texttt{then}\ \llbracket P \rrbracket\ \texttt{else}\ \llbracket Q \rrbracket
$$

Table 7. Encoding from S⋆ to A.

Session Initiation. The (high-level) mechanism of session initiation of Rule [Init] in S⋆ (Table 6) is implemented in A by following the Needham-Schroeder-Lowe (NSL) authentication protocol [13]; see Table 7 (top). We use NSL because it is simple, and it has already been formalized in SAPIC. For simplicity, we present the implementation for three participants; the extension to n participants is as expected. The encoding creates queues for intra-session communication using processes $\texttt{insert}(\!(\widetilde{s_{ij}}, \emptyset)\!)$. The security verification uses the built-in library `asymmetric-encryption` available in Tamarin [14], and assumes the usual signature and equational theory for public keys pk, secret keys sk, asymmetric encryption $aenc$ and decryption dec.

Intra-session Communication. Process $\llbracket c_{\mathsf{pq}}!\langle e : \texttt{msg}\rangle.P \rrbracket$ first acquires a lock in the queue c_{pq} to avoid interference. Then, a $\texttt{lookup}\ \_\ \texttt{as}\ \_$ process checks the state of c_{pq}

and enqueues message $\langle \mathtt{msg}, v \rangle$ at its end. Finally, the encoding signals this operation by executing $\mathtt{event}\ \mathsf{out}(c_{\mathtt{pq}}, v)$ before unlocking $c_{\mathtt{pq}}$ and proceeding as as $[\![P]\!]$. The encoding of session delegation $[\![c_{\mathtt{pq}}!\langle\!\langle c : \mathtt{chan} \rangle\!\rangle.P]\!]$ is very similar: the only differences are the sort of the communicated object and the event signaled at the end $(\mathsf{dels}(c_{\mathtt{pq}}, c'))$.

As above, process $[\![c_{\mathtt{pq}}?(x).P]\!]$ first acquires a lock and checks the queue $c_{\mathtt{qp}}$. If it is of the form $\langle \mathtt{msg}, - \rangle$ then it stores it in a variable z_v: it consumes the first part $(\mathit{fst}(z_v))$ and updates $c_{\mathtt{qp}}$ with the second part. The implementation then signals an event $\mathtt{event}\ \mathsf{inp}(c_{\mathtt{pq}}, z_v)$ before unlocking $c_{\mathtt{qp}}$ and proceeding as $[\![P]\!]$. Process $[\![c_{\mathtt{pq}}?(\!(x)\!).P]\!]$ (reception of a delegated session) is similar; in this case, the queue should contain a value of sort $\mathtt{chan}$ and the associated event is $\mathsf{recs}(c_{\mathtt{pq}}, \mathit{fst}(z_v))$.

Process $[\![\mathbf{0}]\!]$ simply executes an event close. In the prototype SAPIC implementation of our encoding, this event mentions the name of the corresponding session $c_{\mathtt{qp}}$.

Finally, process $[\![c_{\mathtt{pq}} : h]\!]$ is $\mathbf{0}$ because we implement queues using the global state in A. The implementation of the remaining constructs in A is self-explanatory.

Remark 1. Since our encoding operates on *untyped* processes, we could have sort mismatches in queues (cf. Rule [If-F]). To avoid this, encodings of input-like processes (e.g., $s_{\mathtt{pq}}?(x).P$), use the input of a *dummy* value that allows processes to reduce.

Correctness of $[\![\cdot]\!]$**.** We first associate to each ground process $P \in \mathsf{S}^*$ a process configuration via the encoding in Table 7. Below we assume that $\tilde{s}$, I, and I' may be empty, allowing the encoding of communicating processes (obtained after session initiation); we also assume that the set of (free) variables in P (denoted $var(P)$) can be instantiated with ground terms that can be deduced from the current frame.

Definition 1 *Suppose an $\mathsf{S}^\star$ process $R \equiv (\nu s)(\prod_{i \in I} P_i \mid \prod_{j,k \in I'} s_{\mathtt{p}_j\mathtt{q}_k} : h_{j,k})$, with $var(R) = \{x_1, \ldots, x_n\}$. A process configuration for R, denoted $C[[\![R]\!]]$, is defined as:*

$$(\mathcal{E} \cup \{s\}, \mathcal{S} \cup \{s_{\mathtt{p}_j\mathtt{q}_k} : h_{j,k} \mid j, k \in I'\}, \Big\{ \prod_{i \in I} [\![P_i]\!] \Big\}, \sigma, \mathcal{L}),$$

where $var(R) \subseteq dom(\sigma)$ and σ is grounding for x_i, $i = 1, \ldots, n$.

With some abuse of notation we say that C is a process configuration for R. Observe that different process configurations $C, C', \ldots$ can be associated to a same process $R \in \mathsf{S}$ once one considers variations of $\mathcal{E}, \mathcal{S}, \sigma, \mathcal{L}$.

Theorem 2 (Completeness). *Let $P \in \mathsf{S}^\star$. If $P \longrightarrow_{\mathsf{S}^\star} P'$ then for all process configuration C, there exists a process configuration C' such that $C[[\![P]\!]] \longrightarrow_{\mathsf{A}}^{*} C'[[\![P']\!]]$.*

Proof. The proof is by structural induction, analyzing the rule applied in $P \longrightarrow_{\mathsf{S}^\star} P'$ via encoding in Table 7 and the rules in Table 3. See [15] for details. □

To prove soundness, we rely on a Labeled Transition System for $\mathsf{S}^\star$, denoted $P \xrightarrow{\lambda} P'$. Such an LTS, and the proof of the theorem below, can be found in [15].

Theorem 3 (Soundness). *Let $P \in \mathsf{S}^\star$ and C be such that $C[[\![P]\!]] \longrightarrow_{\mathsf{A}_P} R$. Then there exist $P' \in \mathsf{S}^\star$, a C', and λ such that $R \longrightarrow_{\mathsf{A}}^{*} C'[[\![P']\!]]$ and $P \xrightarrow{\lambda} P'$.*

$S ::= \texttt{bool} \mid \texttt{nonce} \mid \texttt{msg} \mid \texttt{temp} \mid \ \ldots \ \mid G$ Sorts $\qquad U ::= S \mid T$ Exchange Types

(Global Types) $G ::= \mathtt{p} \to \mathtt{q} : \langle U \rangle.G \mid \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} \mid \texttt{end}$
(Local Types) $T ::= !\langle \mathtt{p}, U \rangle.T \mid ?(\mathtt{p}, U).T \mid \ \oplus \langle \mathtt{p}, \{l_i : T_i\} \rangle \mid \&(\mathtt{p}, \{l_i : T_i\}) \mid \texttt{end}$

Table 8. Global and Local Types [10].

5 Multiparty Session Types and Their Local Formulas

Using $(\![\cdot]\!)$ and $[\![\cdot]\!]$, in this section we connect well-typedness of processes in S [10] with the satisfiability of *local formulas*, which model the execution of A processes.

5.1 Global and Local Types

Rather than defining multiparty session types for A processes, we would like to model checking local types by re-using existing tools for A: SAPIC [12] and Tamarin [14]. Concretely, next we shall connect typability for S processes with satifiability for A processes. To formalize these results, we first recall some essential notions for multiparty session types; the reader is referred to [10,5] for an in-depth presentation.

Global types G, G' describe multiparty session protocols from a vantage point; they offer a complete perspective on how two or more participants should interact. On the other hand, *local (session) types* T, T' describe how each participant contributes to the multiparty protocol. A *projection function* relates global and local types: the projection of G onto participant n is denoted $G|_n$. The syntax for global and local types, given in Table 8 is standard [10]. A complete description of session types can found in [15].

Example 1. Fig. 2 gives three global types for the protocol in § 2: while G_{init} represents the first phase, both G_{contract} and G_{sign} are used to represent the second. In G_{sign}, we use G_{resolve_i} to denote a global protocol for resolving conflicts; see [15] for details.

Typing judgements for expressions and processes are of the form $\Gamma \vdash e : S$ or $\Gamma \vdash P \triangleright \Delta$, where $\Gamma ::= \emptyset \mid \Gamma, x : S$ and $\Delta ::= \emptyset \mid \Delta, c : T$. The *standard environment* Γ assigns variables to sorts and service names to closed global types; the *session environment* Δ associates channels to local types. We write $\Gamma, x : S$ only if $x \notin dom(\Gamma)$, where $dom(\Gamma)$ denotes the domain of Γ. We adopt the same convention for $a : G$ and $c : T$, and write Δ, Δ' only if $dom(\Delta) \cap dom(\Delta') = \emptyset$. Typing rules are as in [10,5]; as discussed in those works, typability for S processes ensure communication correctness in terms of *session fidelity* (well-typed processes respect prescribed local protocols) and *communication safety* (well-typed processes do not feature communication errors), among other properties.

5.2 Satisfiability of Local Formulas from A

Following the approach in [12], properties of processes in A will be established via analysis of *traces*, which describe the possible executions of a process. This will allow us to prove communication correctness of S processes, using encoding $[\![\cdot]\!]$.

$$
\begin{array}{lll}
G_{\text{init}}: & (I.1)\ 3 \to 1 : & \langle\text{Title}\rangle \\
& (I.2)\ 1 \to \{2,3\} : & \langle\text{quote}\rangle \\
& (I.3)\ 3 \to 2 : & \langle\text{quote'}\rangle \\
& (I.4)\ 2 \to \{1,3\} : & \begin{cases} \texttt{ok} : G_{\text{contract}} \\ \neg\texttt{ok} : \texttt{end} \end{cases}
\end{array}
\qquad
\begin{array}{l}
G_b : \\
(1')\ 1 \to 2 : \langle T \rangle \\
\\
T = (G_{\text{contract}})|_1
\end{array}
$$

$$
\begin{array}{lll}
G_{\text{contract}}: & (c.1)\ 1 \to \{2,3\} : & \langle\text{contract}\rangle \\
& (c.2)\ 3 \to 2 : & \langle\text{promise}\rangle \\
& (c.3)\ 2 \to 3 : & \begin{cases} \texttt{ok} : \quad 2 \to 3 : \langle\text{promise}\rangle \\ \qquad\quad 3 \to 2 : \begin{cases} \texttt{ok} : G_{\text{sign}} \\ \neg\texttt{ok} : 3 \to 1 : \texttt{abort} \end{cases} \\ \neg\texttt{ok} : \texttt{end} \end{cases}
\end{array}
$$

$$
\begin{array}{ll}
G_{\text{sign}}: & (s.1)\ 3 \to 2 : \langle\text{signature}_1\rangle \\
& (s.1)\ 2 \to 3 : \begin{cases} \texttt{ok} : \quad 2 \to 3 : \langle\text{signature}_2\rangle \\ \qquad\quad 3 \to 1 : \begin{cases} \texttt{success} : \quad 3 \to 1 : \langle\text{address}\rangle \\ \qquad\qquad\qquad 1 \to 3 : \langle\text{date}\rangle \\ \neg\texttt{success} : 1 \to 3 : G_{\text{resolve}_1} \end{cases} \\ \neg\texttt{ok} : 2 \to 1 : G_{\text{resolve}_2} \end{cases}
\end{array}
$$

Fig. 2. Global Types for the Trusted Buyer-Seller Protocol (§ 2).

Definition 1 (Traces of P [12]). *Given a ground process $P \in \mathsf{A}$, we define the* set of traces of P*, denoted by* $\mathtt{traces}(P)$*, as*

$$\mathtt{traces}(P) = \left\{ [F_1, \ldots, F_n] \;\middle|\; (\emptyset, \{P\}, \emptyset, \emptyset) \xRightarrow{F_1} \ldots \xRightarrow{F_n} (\mathcal{E}_n, \mathcal{S}_n, \mathcal{P}_n, \sigma_n, \mathcal{L}_n) \right\}$$

We will denote by tr_P, a trace from a set $\mathtt{traces}(P)$, for some process P. We will write tr when P is clear from the context. Notice that, $\mathsf{tr}_P = \mathsf{tr}_Q$ does not necessarily imply that $P = Q$: each process may implement more than one session in different ways.

SAPIC and Tamarin [14] consider two sorts: $\mathtt{temp}$ and $\mathtt{msg}$. Each variable of sort $\mathtt{s}$ will be interpreted in the domain $D(\mathtt{s})$; in particular, we will denote by $\mathcal{V}_{\mathtt{temp}}$ the set of temporal variables, which is interpreted in the domain $D(\mathtt{temp}) = \mathcal{Q}$; also, $\mathcal{V}_{\mathtt{msg}}$ is the set of message variables, which is interpreted in the domain $D(\mathtt{msg}) = \mathcal{M}$. Below, we will adopt a function $\theta : \mathcal{V} \to \mathcal{M} \cup \mathcal{Q}$ that maps variables to terms respecting the variable's sorts, that is $\theta(x : \mathtt{s}) \in D(\mathtt{s})$.

Definition 2 (Trace atoms [12]). *A* trace atom *has of one of the forms:*

$$A ::= \bot \mid t_1 \approx t_2 \mid i \lessdot j \mid i \doteq k \mid F@i$$

denoting, respectively, false, term equality, timepoint ordering, timepoint equality, or an action for a fact F and a timepoint i. The construction of trace formula φ *respects the usual first-order convention:*

$$\varphi, \psi ::= A \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \to \psi \mid \varphi \leftrightarrow \psi \mid (\exists x : \mathtt{s}).\varphi \mid (\forall x : \mathtt{s}).\varphi$$

Given a process P, in the definition below, tr denotes a trace in $\mathtt{traces}(P)$, $idx(\mathsf{tr})$ denotes the positions in tr, and tr_i denotes the i-th position in tr.

Definition 3 (Satisfaction relation [12]). *The satisfaction relation* $(\mathsf{tr}, \theta) \vDash \varphi$ *between a trace* tr*, a valuation* θ*, and a trace formula* φ *is defined as follows*

$$\begin{array}{llllll}
(\mathsf{tr}, \theta) \vDash \bot & \textit{never} & & (\mathsf{tr}, \theta) \vDash t_1 \approx t_2 & \textit{iff} & t_1\theta =_E t_2\theta \\
(\mathsf{tr}, \theta) \vDash i \lessdot j & \textit{iff} & \theta(i) < \theta(j) & (\mathsf{tr}, \theta) \vDash \neg\varphi & \textit{iff not} & (\mathsf{tr}, \theta) \vDash \varphi \\
(\mathsf{tr}, \theta) \vDash i \doteq j & \textit{iff} & \theta(i) = \theta(j) & (\mathsf{tr}, \theta) \vDash \varphi_1 \wedge \varphi_2 & \textit{iff} & (\mathsf{tr}, \theta) \vDash \varphi_1 \textit{ and } (\mathsf{tr}, \theta) \vDash \varphi_2
\end{array}$$

$$\begin{array}{lll}
(\mathsf{tr}, \theta) \vDash F@i & \textit{iff} & \theta(i) \in idx(\mathsf{tr}) \textit{ and } F\theta =_E \mathsf{tr}_{\theta(i)} \\
(\mathsf{tr}, \theta) \vDash (\exists x : \mathtt{s}).\varphi & \textit{iff} & \textit{there exists } u \in D(\mathtt{s}) \textit{ such that } (\mathsf{tr}, \theta[x \mapsto u]) \vDash \varphi
\end{array}$$

Satisfaction of $(\forall x : \mathtt{s})\varphi$*,* $\varphi \vee \psi$ *and* $\varphi \Rightarrow \psi$ *can be obtained from the cases above.*

5.3 From Local Types to Local Formulas

Below we assume s is an established session between participants $\mathtt{p}$ and $\mathtt{q}$. Given k : $\mathtt{temp}$ and a trace formula φ, we write $\varphi(k)$ to say that there is a fact F such that $F@k$ is an atom in φ. Below we assume that S is a subsort of $\mathtt{msg}$.

Definition 4 (Local Formula). *Given a local type* T *and an endpoint* $s[\mathtt{p}]$*, its* local formula $\Phi_{s[\mathtt{p}]}(T)$ *is defined inductively as follows:*

$$\begin{array}{ll}
\Phi_{s[\mathtt{p}]}(!\langle \mathsf{q}, S\rangle.T) & = \exists i, z.(\mathsf{out}(s_{\mathtt{pq}}, z)@i \wedge \psi(\Phi_{s[\mathtt{p}]}(T)) \\
\Phi_{s[\mathtt{p}]}(?(\mathsf{q}, U).T) & = \exists i, z.(\mathsf{inp}(s_{\mathtt{pq}}, z)@i \wedge \psi(\Phi_{s[\mathtt{p}]}(T)) \\
\Phi_{s[\mathtt{p}]}(\oplus\langle \mathsf{q}, \{l_i : T_i\}_{i\in I}\rangle) & = \exists i. \bigvee_{j\in I}(\mathsf{sel}(s_{\mathtt{pq}}, l_j)@i \wedge \psi(\Phi_{s[\mathtt{p}]}(T_j)) \\
\Phi_{s[\mathtt{p}]}(\&(\mathsf{q}, \{l_i : T_i\}_{i\in I})) & = \exists i. \bigvee_{j\in I}(\mathsf{bra}(s_{\mathtt{pq}}, l_j)@i \wedge \psi(\Phi_{s[\mathtt{p}]}(T_j)) \\
\Phi_{s[\mathtt{p}]}(\mathsf{end}) & = \exists i.\mathsf{close}@i.
\end{array}$$

where $\psi(\Phi_{s[\mathtt{p}]}(T)) := \forall k.(\Phi_{s[\mathtt{p}]}(T)(k) \Rightarrow i \lessdot k))$ *the quantified variables have sorts* $i, j, k : \mathtt{temp}$ *and* $z : S$*, and variables i and z are fresh. The extension of* $\Phi(\_)$ *to* session environments*, denoted* $\widehat{\Phi}(\_)$*, is as expected:* $\widehat{\Phi}(\Delta, s[\mathtt{p}] : T) = \widehat{\Phi}(\Delta) \wedge \Phi_{s[\mathtt{p}]}(T)$*.*

Remark 2. Since each local type is associated to a unique local formula, the mapping $\Phi_{\_}(\_)$ is invertible. That said, from a local formula φ we can obtain the corresponding type $\Phi^{-1}(\varphi)$. For instance, for the local formula $\varphi_{\mathtt{out}} := \exists iz.(\mathsf{out}(s_{\mathtt{pq}}, z)@i \wedge \psi(\Phi_{s[\mathtt{p}]}(T))$, one has $\Phi^{-1}(\varphi_{\mathtt{out}}) = s[\mathtt{p}] :!\langle \mathsf{q}, S\rangle.\Phi^{-1}_{s[\mathtt{p}]}(\varphi')$. The other cases are similar.

The following theorems give a bi-directional connection between (a) well-typednesss and (b) satisfiability of the corresponding local formulas (see [15]):

Theorem 4. *Let* $\Gamma \vdash P \triangleright \Delta$ *be a well-typed* S *process. Also, let* $\mathsf{tr} \in \mathtt{traces}(\llbracket (\!| P |\!) \rrbracket)$*. Then there exists a* θ *such that* $(\mathsf{tr}, \theta) \vDash \widehat{\Phi}(\Delta)$*.*

Theorem 5. *Let* tr *and* φ *be a trace and a local formula, respectively. Suppose* θ *is an instantiation such that* $(\mathsf{tr}, \theta) \vDash \varphi$*. Then there is a* $P \in \mathsf{S}$ *such that*

$$\Gamma_\varphi \vdash P \triangleright \Phi^{-1}(\varphi) \qquad \textit{where } \Gamma_\varphi = \{\theta(x) : sort(x) \mid x \in dom(\theta)\}$$

Example 2. The projection of G_{init} onto participant 3 (Buyer1), under session s is: $s[3]$: $!\langle \mathtt{1}, string\rangle.?(\mathtt{1}, int).!\langle \mathtt{2}, int\rangle.\&(2, \{\mathtt{ok} : (G_{\text{contract}}|_3), \neg\mathtt{ok} : \mathtt{end}\})$.

The local formula associated is:

$$\begin{aligned}\Phi_{s[3]}(T) = {} & \exists i_1, z_1.\mathsf{out}(s_{31}, z_1)@i_1 \wedge (\exists i_2 z_2.\mathsf{inp}(s_{31}, z_2)@i_2 \wedge (\exists i_3 z_3.\mathsf{out}(s_{32}, z_3))@i_3 \\ & \wedge (\exists i_4 i_5 z_4.((\mathsf{bra}(s_{32}, \mathtt{ok})@i_4 \wedge \Phi_{s[3]}(T')) \vee \mathsf{bra}(s_{32}, \neg\mathtt{ok})@i_4 \wedge \mathsf{close}@i_5))) \\ & \wedge ((i_1 < i_2 < i_3 < i_4 \wedge \psi(\Phi_{s[3]}(T'))) \vee (i_1 < i_2 < i_4 < i_5 \wedge \psi(\Phi_{s[3]}(T'))))\end{aligned}$$

where T' is the projection of G_{contract} onto participant 3.

6 Revisiting the Two-Buyer Contract Signing Protocol

We recall the motivating example introduced in § 2. Using a combination of constructs from S and A, we first develop a protocol specification which is compiled down to A using our encoding; the resulting A process can be then used to verify authentication and protocol correctness properties in SAPIC/Tamarin. Figure 2 shows the corresponding global types, and their associated local types (obtained via projection following [10]).

An alternative approach to specification/verification would be as follows. First, specify the protocol using S only, abstracting away from cryptography, and using existing type systems for S to enforce protocol correctness. Then, compile this resulting S specification down to A, where the resulting specification can be enhanced with cryptographic exchanges and authentication properties can be enforced with SAPIC/Tamarin.

6.1 Process Specification

Process specifications for B_i and S are as follows:

$$B_1 = \overline{a}[3](y).y[3]!\langle \mathtt{1}, \text{“Title”}\rangle.y[3]?(\mathtt{1}, x_1).y[3]!\langle \mathtt{2}, x_1\ div\ 2\rangle.y[3]\&(\mathtt{2}, \{\mathtt{ok} : B_1^{sct}, \neg\mathtt{ok} : \mathbf{0}\})$$

$$\begin{aligned}B_2 = {} & a[2](y).y[2]?(\mathtt{1}, x_2).y[2]?(\mathtt{3}, x_3).\mathtt{if}\ x_2 - x_3 \leq 99\ \mathtt{then}\ y[2] \oplus \langle\{\mathtt{1}, \mathtt{3}\}, \mathtt{ok}\rangle.B_2^{sct} \\ & \mathtt{else}\ \ y[2] \oplus \langle\{\mathtt{1}, \mathtt{3}\}, \neg\mathtt{ok}\rangle.\mathbf{0}\end{aligned}$$

$$\begin{aligned}S = {} & a[1](y).y[1]?(\mathtt{3}, x_1).y[1]!\langle\{\mathtt{2}, \mathtt{3}\}, \text{quote}\rangle.y[1]\&(\mathtt{2}, \{\mathtt{ok} : \overline{b}[2](z).y[2]!\langle\langle \mathtt{1}, y\rangle\rangle.z[2]?(\mathtt{1}, x_4). \\ & y[1]!\langle \mathtt{2}, \text{date}\rangle.\mathbf{0}, \neg\mathtt{ok} : \mathbf{0}\}))\end{aligned}$$

where processes B_1^{sct} and B_2^{sct}, which implement the contract signing phase, are as in Tables 9 and 10, respectively. The specification for the trusted authority T is as follows:

$$\begin{aligned}& b[1](z).z[1]?(\!(\mathtt{2}, t)\!).\nu sk(T); t[3]!\langle\{\mathtt{1}, \mathtt{2}\}, pk(sk(T))\rangle.y[1]?(\mathtt{3}, z_2).y[1]?(\mathtt{2}, z_3).(\nu s)\mathtt{insert}(\!(s, init)\!). \\ & (\nu\ ct)t[3]!\langle\{\mathtt{1}, \mathtt{2}\}, ct\rangle.t[3]\&(\{\mathtt{1}, \mathtt{2}\}, \{\mathtt{abort}: P^T_{Ab}, \mathtt{res}_1: P^T_{R_1}, \mathtt{res}_2: P^T_{R_2}, \mathtt{success}: z[1]!\langle \mathtt{2}, \mathtt{ok}\rangle.\mathbf{0}\})\}\end{aligned}$$

where processes P^T_{Ab}, $P^T_{R_1}$, and $P^T_{R_2}$ are given in Tables 11 and 12. Process T illustrates how we may combine constructs from S (important to represent, e.g., session establishment on b and delegation from S) and features from A (essential to, e.g., manipulate the memory cell s, which records contract information). Indeed, T uses the A construct `insert` to initialize the cell s and `lookup` _ `as` _ to update it. Therefore, the sound and complete encoding proposed in § 4 allows us to specify processes in A, while retaining the high-level constructs from S.

$$\begin{aligned}
B_1^{sct} &= y[3]?(\mathtt{1}, z_1).\nu sk(B_1).y[3]!\langle\{\mathtt{1},\mathtt{2}\}, pk(sk(B_1))\rangle.y[3]?(\mathtt{2}, z_3).y[3]?(\mathtt{1}, z_4).\\
&\quad y[3]!\langle \mathtt{2}, m_1\rangle.y[\mathtt{3}]\&(\mathtt{2}, \{\mathtt{ok} : P^1_{conv},\ \neg\mathtt{ok} : \mathbf{0}\}))\\
P^1_{conv} &= y[3]?(\mathtt{2}, z_5).(\mathbf{if}\ \mathtt{pcsver}(z_3, pk(sk(B_1)), z_1, z_4, z_5) = \mathtt{true}\ \mathbf{then}\ (y[3] \oplus \langle \mathtt{2}, \mathtt{ok}\rangle.\\
&\quad y[3]!\langle \mathtt{2}, S_1\rangle.y[\mathtt{3}]\&(\mathtt{2}, \{\mathtt{ok} : P^1_{sign}, \neg\mathtt{ok} : P^1_{res}\})\ \ \mathbf{else}\ y[3] \oplus \langle \mathtt{1}, \mathtt{abort}\rangle.P^1_{abort})\\
P^1_{sign} &= y[3]?(\mathtt{2}, z_6).(\mathbf{if}\ \mathtt{sver}(z_3, z_4, z_6) = \mathtt{true}\ \mathbf{then}\ (y[3] \oplus \langle \mathtt{1}, \mathtt{success}\rangle.\mathbf{0}\ \ \mathbf{else}\ P^1_{res})\\
P^1_{res} &= y[3] \oplus \langle \mathtt{1}, \mathtt{res}_1\rangle.y[3]!\langle \mathtt{1}, \langle S_1, x_1\rangle\rangle.y[3]?(\mathtt{1}, z_7).\mathbf{0}\\
P^1_{abort} &= y[3] \oplus \langle \mathtt{1}, \mathtt{abort}\rangle.y[3]!\langle \mathtt{1}, [ct, B_1, B_2, \mathtt{abort}]_{B_1}\rangle.y[3]?(\mathtt{1}, z_8).\mathbf{0}
\end{aligned}$$

Table 9. B_1^{sct}: B_1's contract signing processes.$[m]_X$ denotes $\langle m, \mathtt{sign}(sk(x), m)\rangle$

$$\begin{aligned}
B_2^{sct} &= y[2]?(\mathtt{1}, z_1).y[2]?(\mathtt{1}, z_2).\nu sk(B_2).y[2]!\langle\{\mathtt{1},\mathtt{3}\}, pk(sk(B_2))\rangle.y[2]?(\mathtt{3}, z_9).y[2]?(\mathtt{3}, z_{10}).\\
&\quad (\mathbf{if}\ \mathtt{pcsver}(z_2, pk(sk(B_2)), z_1, z_4, z_{10}) = \mathtt{true}\ \mathbf{then}\ (y[2] \oplus \langle \mathtt{3}, \mathtt{ok}\rangle.y[2]!\langle \mathtt{3}, m_2\rangle.\\
&\quad y[\mathtt{2}]\&(\mathtt{3}, \{\mathtt{ok} : P^2_{sign}, \neg\mathtt{ok} : 0\}))\ \ \mathbf{else}\ y[2] \oplus \langle \mathtt{3}, \neg\mathtt{ok}\rangle.\mathbf{0})\\
P^2_{Sign} &= y[2]?(\mathtt{3}, z_{11}).\mathbf{if}\ \mathtt{sver}(z_2, z_4, z_{11}) = \mathtt{true}\ \ \mathbf{then}\ y[2] \oplus \langle \mathtt{3}, \mathtt{ok}\rangle.y[2]!\langle \mathtt{3}, S_2\rangle.\\
&\quad y[2] \oplus \langle \mathtt{1}, \mathtt{success}\rangle.\mathbf{0}\ \ \mathbf{else}\ y[2] \oplus \langle \mathtt{3}, \neg\mathtt{ok}\rangle.P^2_{resolve}\\
P^2_{resolve} &= y[2] \oplus \langle \mathtt{1}, \mathtt{res}_2\rangle.y[2]!\langle \mathtt{1}, S_2\rangle.y[2]?(\mathtt{1}, z_{12}).\mathbf{0}
\end{aligned}$$

Table 10. B_2^{sct}: B_2's contract signing processes.

$$\begin{aligned}
P^T_{Ab} &= \mathbf{lock}\ s; y[1]?(\mathtt{3}, y_1).\mathbf{if}\ \mathtt{sver}(fst(y_1), snd(y_1)) = \mathtt{true}\ \ \mathbf{then}\ (\mathbf{lookup}\ s\ \mathbf{as}\ y_2\ \ \mathbf{in}\\
&\quad (\mathbf{if}\ fst(y_2) = init\ \mathbf{then}\ (\mathbf{insert}((s, [y_1]_T)); y[1]!\langle \mathtt{3}, [y_1]_T\rangle; \mathbf{unlock}\ s_{\mathtt{pq}}))\\
&\quad\ \mathbf{else}\ (\mathbf{if}\ fst(y_2) = \mathtt{abort}\ \ \mathbf{then}\ y[1]!\langle \mathtt{3}, y_2\rangle; \mathbf{unlock}\ s_{\mathtt{pq}}))))\\
&\quad\ \mathbf{else\ if}\ fst(y_2) = \mathtt{res}_i\ \ \mathbf{then}\ y[1]!\langle \mathtt{3}, y_2\rangle; \mathbf{unlock}\ s_{\mathtt{pq}}))))
\end{aligned}$$

Table 11. P^T_{Ab}: abort process executed by T

To model the second phase of the protocol, we consider a Private Contract Signature

$$\begin{aligned}
\Sigma_{\mathtt{pcs}} = \{&aenc(\_,\_), senc(\_,\_), pk(\_), sk(\_), \mathtt{pcs}, \mathtt{sign}, \mathtt{tsign}, sdec(,), adec(,),\\
&\mathtt{sconvert}, \mathtt{tconvert}, \mathtt{pcsver}, \mathtt{sverif}\}
\end{aligned}$$

with function symbols for promises and signatures, and for verifying the validity of exchanged messages. As for constructors: $\mathtt{pcs}(x, y, w, z)$ is the promise of x to y to sign contract z given by w; $\mathtt{sign}(x, y)$ is the signature of x in z; $pk(x)$ is the public key of x; $sk(x)$ is the secret key of x; $aenc(x, y)$ is the asymmetric encryption of y using key x; and $senc(x, y)$ is the symmetric encryption of y using key x. Destructor $sdec(,)$ (resp. $adec(,)$) enforces symmetric (resp. asymmetric) decryption; the other destructors ($\mathtt{sconvert}$, $\mathtt{tconvert}$, $\mathtt{pcsver}$, $\mathtt{sverif}$) are defined from the rules in $E_{\mathtt{pcs}}$:

$$\begin{array}{ll}
sdec(x, senc(x, y)) \to y & \mathtt{tconvert}(w, \mathtt{pcs}(x, y, pk(w), z)) \to \mathtt{sign}(x, z)\\
adec(sk(x), aenc(pk(x), y)) \to y & \mathtt{pcsver}(pk(x), y, w, z, \mathtt{pcs}(x, y, w, z)) \to \mathtt{true}\\
\mathtt{sver}(pk(x), z, \mathtt{sign}(x, z)) \to \mathtt{true} & \mathtt{sconvert}(x, \mathtt{pcs}(x, y, w, z)) \to \mathtt{sign}(x, z)
\end{array}$$

Table 13 shows the translation of B_1 in A, using our encoding. For simplicity, we omit the details related to the session establishment (using NSL), which follow Table 7.

$$
\begin{array}{l}
P^{T}_{\mathtt{res}_1} = \mathtt{lock}\ s; y[1]?(3, y_3).\mathtt{if}\ m'_{11} =_{E_{pcs}} \mathtt{true}\ \mathtt{then}\ (\mathtt{if}\ m'_{12} =_{E_{pcs}} \mathtt{true}\ \mathtt{then} \\
\qquad\quad (\mathtt{lookup}\ s\ \mathtt{as}\ y_3\ \mathtt{in}\ (\mathtt{if}\ fst(y_3) = \mathtt{abort}\ \mathtt{then}\ y[1]!\langle 3, snd(y_3)\rangle.\mathtt{unlock}\ s)) \\
\qquad\qquad \mathtt{else}\ (\mathtt{if}\ fst(y_3) = \mathtt{res}_2\ \mathtt{then}\ y[1]!\langle 3, snd(y_3)\rangle.z[1]!\langle 2, \mathtt{ok}\rangle; \mathtt{unlock}\ s)) \\
\qquad\qquad \mathtt{else}\ y[1]!\langle 3, \mathtt{tconvert}(sk(T), snd(m'_1))\rangle.z[1]!\langle 2, \mathtt{ok}\rangle.\mathtt{unlock}\ s)) \\
P^{T}_{\mathtt{res}_2} = \mathtt{lock}\ s; y[1]?(2, w_3); \mathtt{if}\ m'_{21} =_{E_{pcs}} \mathtt{true}\ \mathtt{then}\ (\mathtt{if}\ m'_{22} =_{E_{pcs}} \mathtt{true}\ \mathtt{then} \\
\qquad\quad (\mathtt{lookup}\ s\ \mathtt{as}\ w_3\ \mathtt{in}\ (\mathtt{if}\ fst(w_3) = \mathtt{abort}\ \mathtt{then}\ y[1]!\langle 2, snd(w_3)\rangle; \mathtt{unlock}\ s)) \\
\qquad\quad\ \mathtt{else}\ (\mathtt{if}\ fst(w_3) = \mathtt{res}_1\ \mathtt{then}\ (y[1]!\langle 2, snd(w_3)\rangle; z[1]!\langle 2, \mathtt{ok}\rangle; \mathtt{unlock}\ s)) \\
\qquad\quad\ \mathtt{else}\ y[1]!\langle 3, \mathtt{tconvert}(sk(T), fst(m'_2))\rangle; \mathtt{insert}((s, \langle \mathtt{res}_2, snd(w_3)\rangle)); \\
\qquad\quad z[1]!\langle 2, \mathtt{ok}\rangle; \mathtt{unlock}\ s))
\end{array}
$$

Table 12. $P^{T}_{\mathtt{res}_1}$ e $P^{T}_{\mathtt{res}_2}$: resolve processes executed by T

$$
\begin{array}{l}
\nu s; P_{31}; P_{32}; \mathtt{insert}((\widetilde{s_{ij}}, \emptyset)); \mathtt{event}\ \mathsf{init}(\widetilde{s_{ij}}); \mathtt{event}\ \mathsf{sndchann}(pk(ska_{31}), pk(y_1), s); \\
\quad \mathtt{out}(u_1, s); \mathtt{event}\ \mathsf{sndchann}(pk(ska_{32}), pk(y_2), s); \mathtt{out}(u_2, s); \\
\quad \mathtt{lock}\ s_{31}; \mathtt{lookup}\ s_{31}\ \mathtt{as}\ x_{31}\ \mathtt{in} \\
\qquad \mathtt{insert}((s_{31}, x_{31} \cdot \langle \mathtt{msg}, \text{“Title”}\rangle)); \mathtt{event}\ \mathsf{out}(s_{31}, \text{“Title”}); \mathtt{unlock}\ s_{31}; \\
\quad \mathtt{lock}\ s_{13}; \mathtt{lookup}\ s_{31}\ \mathtt{as}\ x\ \mathtt{in} \\
\qquad \mathtt{if}\ fst(x) = \langle \mathtt{msg}, z\rangle\ \mathtt{then}\ \mathtt{insert}((s_{31}, snd(x))); \mathtt{event}\ \mathsf{inp}(s_{31}, z); \mathtt{unlock}\ s_{13} \\
\quad \mathtt{lock}\ s_{32}; \mathtt{lookup}\ s_{32}\ \mathtt{as}\ x_{32};\ \mathtt{in} \\
\qquad \mathtt{insert}((s_{32}, \langle \mathtt{msg}, \text{“quote”}\rangle)); \mathtt{event}\ \mathsf{out}(s_{32}, \text{“quote”}); \mathtt{unlock}\ s_{32}; \\
\quad \mathtt{lock}\ s_{23}; \mathtt{lookup}\ s_{23}\ \mathtt{as}\ x_{23}\ \mathtt{in}\ (\mathtt{if}\ fst(x_{23}) = \langle \mathtt{lbl}, \mathtt{ok}\rangle\ \mathtt{then} \\
\qquad \mathtt{insert}((s_{23}, snd(x_{23}))); \mathtt{event}\ \mathsf{bra}(s_{23}, \mathtt{accept}); \mathtt{unlock}\ s_{23}; [\![B_1^{sct}]\!] \\
\qquad\ \mathtt{else}\ \mathtt{event}\ \mathsf{bra}(s_{23}, \neg\mathtt{ok}); \mathtt{unlock}\ s_{23}; \mathbf{0})
\end{array}
$$

Table 13. Translation of B_1 into A.

Process specifications for B_2, S, and T in A can be obtained similarly. As mentioned in § 4, the communication is done via updating session queues s_{ij}, for $i, j = 1, 2, 3$.

6.2 Using SAPIC/Tamarin to Verify Authentication and Local Session Types

We conclude this section by briefly discussing how to use our developments to verify properties associated to authentication and protocol correctness.

Concerning authentication, we can use SAPIC/Tamarin to check the correctness of the authentication phase implemented by NSL. The proof checks that events honest(_), sndnonce(_, _, _), rcvnonce(_, _, _), and rcvchann(_, _, _) occur in the order specified by the encoding in Table 7. This way, e.g., the following lemma verifies the correctness of the specification of the fragment of NSL authentication with respect to participant B_2:

$$
\begin{array}{l}
\mathtt{lemma}\ \mathsf{B2\_NSL\_correctness}: \\
\mathtt{exists} - \mathtt{trace} \\
(\mathtt{All}\ pk_{12}\ pk_{1s}\ pk_2\ pk_s\ \#i\ \#j\ \#k\ \#l. \\
\quad \mathsf{honest}(pk_{12})@i\ \&\ \mathsf{honest}(pk_{1s})@j\ \&\ \mathsf{honest}(pk_2)@k\ \&\ \mathsf{honest}(pk_s)@l \\
\quad \Longrightarrow (\mathsf{Ex}\ x\ y\ z\ s\ \#j_1\ \#k_1\ \#l_1.\mathsf{rcvnonce}(pk_{12}, pk_2, x, y)@j_1\ \&\ \mathsf{sndnonce}(pk_2, pk_{12}, z)@k_1 \\
\qquad \&\ \mathsf{rcvchann}(pk_{12}, pk_2, s)@l_1 \& j_1 < k_1\ \&\ k_1 < l_1))
\end{array}
$$

The lemma below says that the session channel exchanged using NSL is secret. The proof relies on `asymmetric-encryption`, which is built in the Tamarin library.

$\texttt{lemma}\ \texttt{Chann\_is\_secret}:$
$(\texttt{All}\ pk_{12}\ pk_2\ pk_{1s}\ pk_s\ s\ z\ n\ x\ y\ w\ z\ n_2\ \#i\ \#j\ \#l\ \#i_1\ \#i_2\ \#j_1\ \#j_2\ \#k_1\ \#l_1\ \#l_2.$
$(\mathsf{honest}(pk_{12})@i\&\ \mathsf{honest}(pk_2)@j\ \&\ \mathsf{honest}(pk_{1s})@k\ \&\ \mathsf{honest}(pk_s)@l$
$\quad\&\ \mathsf{sndnonce}(pk_2, pk_{12}, z)@i_1\ \&\ \mathsf{rcvnonce}(pk_2, pk_{12}, n, z)@j_1\&\ \mathsf{sndnonce}(pk_{12}, pk_2, w)@i_2$
$\quad\&\ \mathsf{rcvnonce}(pk_{12}, pk_2, n_2, w)@j_2\ \&\ \mathsf{sndnonce}(pk_s, pk_{1s}, x)@i_3\ \&\ \mathsf{rcvnonce}(pk_s, pk_{1s}, y, x)@j_2$
$\quad\&\ \mathsf{sndchann}(pk_{12}, k_2, s)@k_1\ \&\ \mathsf{rcvchann}(pk12, pk2, s)@k_2\ \&\ \mathsf{sndchann}(pk_{1s}, pk_s, s)@l_1$
$\quad\&\ \mathsf{rcvchann}(pk12, pk2, s)@l_2) \Longrightarrow \texttt{not}(\texttt{Ex}\ \ \#j.\ \ \ \texttt{KU}(s)@j))$

We now consider properties associated to fidelity/safety of processes with respect to their local types. The lemma below ensures protocol fidelity of B_1 and B_2 with respect to the corresponding projections of the global type $G_{\texttt{init}}$, presented in Figure 2. The corresponding local formula can be obtained following Definition 4:

$\texttt{lemma}\ \texttt{B1\_B2\_protocol\_fidelity}:$
$\texttt{exists} - \texttt{trace}$
$(\mathsf{Ex}\ x\ y\ z\ s\ \#j\ \#j_1\ \#k\ \#k_1\ \#l.\mathsf{out}(s_{31}, x)@j\ \&\ \mathsf{inp}(s_{31}, z)@k\ \&\ \mathsf{out}(s_{32}, s)@l\ \&\ j < k\ \&\ k < l$
$\quad\&\ ((\mathsf{bra}(s_{32}, \mathrm{ok}))@j_1\ \&\ l < j_1) \mid (\mathsf{bra}(s_{32}, \neg\mathrm{ok})@k_1\ \&\ l < k_1\ \&\ \Phi(G_{\texttt{contract}}|_3)))\ \&$
$(\mathsf{Ex}\ x\ y\ z\ s\ \#j\ \#j_1\ \#k\ \#k_1\ \#l.\ \mathsf{inp}(s_{21}, z)@k\ \&\ \mathsf{inp}(s_{23}, s)@l\ \&\ j < k\ \&\ k < l$
$\quad\&((\mathsf{sel}(s_{23}, \mathrm{ok})@j_1\ \&\ l < j_1\ \&\ \Phi(G_{\texttt{contract}}|_2)) \mid (\mathsf{sel}(s_{23}, \neg\mathrm{ok})@k_1\ \&\ l < k_1))$

Using similar lemmas, we can also prove protocol fidelity for processes S and T with respect to the projections of the global types presented in Figure 2.

7 Related Works and Concluding Remarks

We have connected two distinct process models: the calculus S for multiparty session-based communication [10] and the calculus A for the analysis of security protocols [12]. To our knowledge, this is the first integration of sessions (in the sense of [11]) within process languages for security protocol analysis. Indeed, research on security extensions to behavioral types (cf. the survey [2]) seems to have proceeded independently from approaches such as those overviewed in [7]. The work in [6] is similar in spirt to ours, but is different in conception and details, as it uses a session graph specification to generate a cryptographic functional implementation that enjoys session integrity. Extensions of session types (e.g., [4,16]) address security issues in various ways, but do not directly support cryptographic operations, global state, nor connections with "applied" languages for (automated) verification, which are all enabled by our approach.

Our work should be mutually beneficial for research on (a) behavioral types and contracts and on (b) automated analysis of security protocols: for the former, our work enables the analysis of security properties within multiparty session protocols; for the latter, our approach enables protocol specifications enriched with high-level communication structures based on sessions. In ongoing work, we have used SAPIC/Tamarin to implement our encodings and the verification technique for communication correctness, based on local formulas (Def. 4). Results so far are very promising, as discussed in § 6.

In future work, we intend to explore our approach to process specification and verification in the setting of ProVerif [3], whose input language is a typed applied π-calculus. We also plan to connect our approach with existing type systems for secure information flow and access control in multiparty sessions [4].

Acknowledgements We are grateful to the anonymous reviewers for their useful remarks and suggestions. Pérez is also affiliated to the NOVA Laboratory for Computer Science and Informatics (supported by FCT grant NOVA LINCS PEst/UID/CEC/04516/2013), Universidade Nova de Lisboa, Portugal.

References

1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. of POPL'01*, pages 104–115, 2001.
2. M. Bartoletti, I. Castellani, P. Deniélou, M. Dezani-Ciancaglini, S. Ghilezan, J. Pantovic, J. A. Pérez, P. Thiemann, B. Toninho, and H. T. Vieira. Combining behavioural types with security analysis. *J. Log. Algebr. Meth. Program.*, 84(6):763–780, 2015.
3. B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends in Privacy and Security*, 1(1-2):1–135, 2016.
4. S. Capecchi, I. Castellani, and M. Dezani-Ciancaglini. Typing access control and secure information flow in sessions. *Inf. Comput.*, 238:68–105, 2014.
5. M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida. A gentle introduction to multiparty asynchronous session types. In *Formal Methods for Multicore Programming*, volume 9104 of *LNCS*, pages 146–178. Springer, 2015.
6. R. Corin, P. Deniélou, C. Fournet, K. Bhargavan, and J. J. Leifer. Secure implementations for typed session abstractions. In *Proc. of CSF 2007*, pages 170–186. IEEE, 2007.
7. V. Cortier and S. Kremer. Formal models and techniques for analyzing security protocols: A tutorial. *Foundations and Trends in Programming Languages*, 1(3):151–267, 2014.
8. J. A. Garay, M. Jakobsson, and P. D. MacKenzie. Abuse-free optimistic contract signing. In *Proc. of CRYPTO'99*, volume 1666 of *LNCS*, pages 449–466. Springer, 1999.
9. K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP'98*, number 1381 in LNCS, 1998.
10. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008.
11. H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3, 2016.
12. S. Kremer and R. Künnemann. Automated analysis of security protocols with global state. In *Proc. of SP 2014*, pages 163–178. IEEE, 2014.
13. G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. *Software - Concepts and Tools*, 17(3):93–102, 1996.
14. S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Proc. of CAV 2013*, volume 8044 of *LNCS*, pages 696–701. Springer, 2013.
15. D. Nantes and J. A. Pérez. Relating Process Languages for Security and Communication Correctness (Full Version). Technical report, 2018. `http://www.jperez.nl`.
16. F. Pfenning, L. Caires, and B. Toninho. Proof-carrying code in a session-typed process calculus. In *Proc. of CPP 2011*, volume 7086 of *LNCS*, pages 21–36. Springer, 2011.