Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Electronic Notes in Theoretical Computer Science 135 (2005) 3–22
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Deconstructing Alice and Bob

Carlos Caleiro[1]

CLC, Department of Mathematics, IST, Lisbon, Portugal

Luca Viganò[2] David Basin[3]

Department of Computer Science, ETH Zurich, Switzerland

Abstract

Alice&Bob–notation is a simple notation for describing security protocols as sequences of message exchanges. We show that, despite the fact that Alice&Bob–notation does not include explicit control flow constructs, it is possible to make some of these aspects explicit when producing formal protocol models without having to resort to more expressive protocol description languages. We introduce a notion of incremental symbolic run to formally handle message forwarding and conditional abortion. In incremental symbolic runs, we use variables to represent messages that the principals cannot read, and we characterize each of the execution steps in order to build a collection of symbolic subruns of increasing lengths, reflecting the data possessed by the principals up to that point in the execution. We contrast this with the simpler (more standard) approach based on formalizing the behavior of principals by directly interpreting message exchanges as sequences of atomic actions. In particular, we provide a complete characterization of the situations where this simpler approach is adequate and prove that incremental symbolic runs are more expressive in general.

Keywords: Security protocols, protocol models, Alice and Bob notation, control flow, message forwarding, protocol abortion.

[1] Email: ccal@math.ist.utl.pt
[2] Email: vigano@inf.ethz.ch
[3] Email: basin@inf.ethz.ch
This work was supported by FCT and EU FEDER through POCTI (via the Project Quant-Log POCTI/MAT/55796/2004 of CLC) and by the FET Open Project IST-2001-39252 and BBW Project 02.0431, "AVISPA: Automated Validation of Internet Security Protocols and Applications".

doi:10.1016/j.entcs.2005.06.007

1 Introduction

The so-called *Alice&Bob–notation* is commonly used to describe security protocols as sequences of message exchange steps of the form

$$A \rightarrow B \ : \ M \, .$$

While this notation is intuitive and compact, it is informal and making it more precise requires defining the sequence of actions taken by each honest principal participating in the protocol. This can be achieved by directly interpreting (i.e. "compiling") each message exchange as a sequence of actions, e.g. actions for sending and receiving messages and for applying cryptographic operations such as encryption and decryption, generation of fresh data, application of hash functions, and the like.

This approach, which we call the *direct-compilation approach*, can be used to formalize the behavior of the different protocol participants for a large subclass of protocols, namely those protocols that consist of a linear sequence of message exchange steps, without control flow constructs (such as loops or if-then-elses). In such protocols, there is only a weak form of implicit branching: each step either succeeds or aborts due to a failed operation, e.g. when a message received does not comply to the protocol specification.

For non-linear protocols with explicit control flow, however, alternative notation must be used, based on richer specification languages that make explicit what is left implicit (or even unspecified) in Alice&Bob–notation. A number of such languages have been proposed, e.g. based on automata, process calculi, or even temporal logic, such as the high-level protocol specification language HLPSL of the AVISPA project [5].

We show that, despite the fact that the Alice&Bob–notation does not include explicit control flow constructs, it is possible to make some of these aspects explicit when producing formal protocol models without having to resort to more expressive protocol description languages. The approach that we present here focuses on handling protocols that require *message forwarding* and *conditional abortion*. To illustrate these two problems, observe that in some protocols, the principals receive submessages that are opaque (or "unreadable") to them — in the sense that these principals cannot decompose these submessages — which they should simply forward to other principals in subsequent protocol steps. For example, a protocol might state that a principal A should receive and immediately forward a message $\{M\}_K$ encrypted with a symmetric key K that A does not yet have. It is possible that such an encrypted message can be analyzed sometime later, as the participants accumulate data incrementally during each protocol run. Hence, when new data

is received, a principal may only then be able to determine that some previously received message does not have the required structure and thus abort the protocol. That is, in some later step A may receive the key K (or receive enough information to compose it herself) and can thus decrypt $\{M\}_K$. In this case, A should abort the protocol run whenever M does not conform to the protocol description (for example, if it does not contain a nonce that A generated and sent to some other principal earlier in the protocol run).

Our main contribution in this paper is to introduce a notion of *incremental symbolic run* to formally handle message forwarding and conditional abortion. In incremental symbolic runs, we use variables to represent opaque submessages, and we characterize each of the execution steps in order to build a collection of symbolic subruns of increasing lengths, where the structure of messages reflects the growth of the relevant data during protocol execution. Said another way, the structure of messages in these subruns reflects the data possessed by the principals up to that point in the execution. We can thus define the sequences of symbolic runs of growing length that adequately model the behavior of each of the protocol participants. We then discuss the relationship between incremental symbolic runs and the runs that one obtains from the direct-compilation approach. We provide a complete characterization of the situations when the direct-compilation approach works, and show that in other situations the more expressive incremental symbolic runs are required.

The approach that we propose is general and independent of the details of the particular formalism chosen for modeling the protocols (e.g. multiset rewriting, distributed temporal logic, strand spaces, process algebras, trace-based models, and so on) and of the details of the intruder model (which could be the standard Dolev-Yao model or the model of an intruder with different capabilities). Therefore, our results can be used to provide a formal footing for using Alice&Bob-notation in security protocol analysis tools even for protocols that explicitly rely on message forwarding and conditional abortion. Moreover, although we do not explore this possibility here, our work provides a good basis for generating protocol implementations from Alice&Bob-style descriptions that explicitly carry out necessary executability and abortion tests.

We proceed as follows. In §2 we illustrate the problems with message forwarding and conditional abortion by means of concrete examples, and in §3 we show how incremental symbolic runs provide a formal solution to these problems. In §4 we draw conclusions. Due to lack of space, we will often only give informal justifications of the technical results; details can be found in [3].

2 Message forwarding and conditional abortion

To illustrate the problems with message forwarding and conditional abortion, we first introduce an algebra of messages and define the actions that can be performed by the principals participating in a protocol run.

2.1 Messages and actions

Let a set Pr of *principal identifiers* and a set Num of "numbers" be given. The elements of Num model random data, like nonces or keys. We will use upper-case letters like $A, B, C, \ldots$ to denote principals and N to denote numbers. We also use lower-case letters like $a, b, c, \ldots$ for variables that range over principals, n to range over numbers, and m to range over messages. All of these variables may be annotated with subscripts.

Definition 2.1 Messages *are built inductively from* atomic messages *(identifiers, numbers, and variables) by* pairing, encryption, *and* hashing. *We write these operations as* $M_1; M_2$, $\{M_1\}_{M_2}$ *(the encryption of* M_1 *by* M_2*), and* $H(M_1)$, *for* M_1 *and* M_2 *messages and* H *a hash function. The set* $sub(M)$ *of* submessages *of a message* M *is defined inductively by*

$$sub(M) = \begin{cases} \{M\} & \text{if } M \text{ is atomic,} \\ \{M_1; M_2\} \cup sub(M_1) \cup sub(M_2) & \text{if } M = M_1; M_2, \\ \{\{M_1\}_{M_2}\} \cup sub(M_1) \cup sub(M_2) & \text{if } M = \{M_1\}_{M_2}, \\ \{H(M_1)\} \cup sub(M_1) & \text{if } M = H(M_1). \end{cases}$$

If S is a set of messages, then we will also write $sub(S)$ to denote the set of all submessages of messages in S, i.e. $sub(S) = \bigcup_{M \in S} sub(M)$.

We follow the *perfect cryptography assumption* and the *free-algebra assumption*, where syntactically different terms denote different messages. For readability, we will often write K to denote messages intended to be used as encryption keys. We also assume that every message K has an *inverse* K^{-1} that must be used for decrypting messages encrypted with K. We further assume that $(K^{-1})^{-1} = K$ and that it is not possible to compute K^{-1} from K. If $K^{-1} = K$ then we speak of *symmetric* encryption, and of *asymmetric* encryption otherwise. As notation, we will write K_A to denote a public key of the principal A, whose inverse is A's private key K_A^{-1}, and we will write K_{AB} to denote a symmetric key that is shared by the principals A and B.

Definition 2.2 *The* actions *that can be performed by a principal participating in a protocol are:*

- $\boldsymbol{s}(M, A)$ — *sending the message* M *to the principal* A,
- $\boldsymbol{r}(M)$ — *receiving the message* M, *and*

$$
\begin{array}{lllll}
(\mathbf{nspk}_1) & a \to b & : & (n_1).\{n_1; a\}_{K_b} \\
(\mathbf{nspk}_2) & b \to a & : & (n_2).\{n_1; n_2\}_{K_a} \\
(\mathbf{nspk}_3) & a \to b & : & \{n_2\}_{K_b}
\end{array}
$$

Fig. 1. The simplified Needham-Schroeder Public-Key Protocol (NSPK).

$$
\begin{array}{l}
\textit{init}\text{-run} : \langle \mathbf{f}(n_1).\mathbf{s}(\{n_1; a\}_{K_b}, b) \,.\, \mathbf{r}(\{n_1; n_2\}_{K_a}) \,.\, \mathbf{s}(\{n_2\}_{K_b}, b)\rangle \\
\textit{resp}\text{-run} : \langle \mathbf{r}(\{n_1; a\}_{K_b}) \,.\, \mathbf{f}(n_2) \,.\, \mathbf{s}(\{n_1; n_2\}_{K_a}, a) \,.\, \mathbf{r}(\{n_2\}_{K_b})\rangle
\end{array}
$$

Fig. 2. The initiator and responder runs of NSPK.

- $\boldsymbol{f}(N)$ — *generating the fresh number* N.

Communication is assumed to be asynchronous and to take place over a hostile network. Hence, principals specify the intended recipients of the messages they send, but the receiving action does not explicitly name the message's sender. In fact, we can assume, as is standard, that the network is controlled by, and can be identified with, a Dolev-Yao intruder [8] who can compose, send, and intercept messages at will, but cannot break cryptography. Note, however, that our results are independent of the particular capabilities of the intruder. Note, too, that in this paper we consider the actions of principals from a fairly high level of abstraction, and we only model the communication actions $\mathbf{s}(M, A)$ and $\mathbf{r}(M)$, and one internal action $\mathbf{f}(N)$. Other internal actions, e.g. corresponding to the application of cryptographic operations (such as explicit encryption and decryption, application of a hash function, test for equality, etc.) can be modeled similarly and our results extended straightforwardly.

2.2 Action sequences and protocol runs

Given an algebra of messages and a set of actions, there are many approaches available for formally specifying and analyzing protocols, for example, using formalisms based on multiset rewriting, distributed temporal logic, strand spaces, process algebras, trace-based models, among others (see, for example, [4,11,12,13]). The approach that we propose is general and independent of the details of the particular formalism chosen. Rather than focusing on any concrete approach, we thus consider a step that is common to all approaches: formally defining the precise sequences of actions that should be executed by each of the protocol participants, given a description of a protocol in Alice&Bob–notation. By *protocol participants* we mean those principals who send or receive messages in some step of the protocol. Hence, our task

amounts to extracting (or "compiling") a sequence of precise actions for each participant from a higher-level notation and, in doing so, giving that notation a precise semantics in terms of the actions of the individual participants.

As a first concrete example, consider the simplified Needham-Schroeder Public-Key Protocol (NSPK) [6]. Fig. 1 shows the usual Alice&Bob–style description of NSPK, where a and b are variables ranging over principals that identify the *roles* played in one run of the protocol. The arrows represent communication from the sender to the receiver, where the parenthesized variables prefixing the first two messages signify that these nonces must be freshly generated before the message is composed and sent.

Let us write $w = \langle w_1.w_2.w_3 \dots \rangle$ to denote a (possibly infinite) sequence w composed of the elements $w_1, w_2, w_3, \dots$. Let $|w|$ denotes its length, where $|\langle\rangle| = 0$ for the empty sequence $\langle\rangle$ and $|w| = \infty$ whenever w is infinite. We also write $w \centerdot w'$ to denote the concatenation of the two sequences, provided that the first sequence is finite, and we write $w|_i$ to denote the prefix of w of length i, i.e. $w|_i = \langle w_1 \dots w_i \rangle$, provided that $0 \leq i \leq |w|$.

Formalizing protocols requires defining the sequences of actions (**s**, **r**, and **f**) taken by each principal running the protocol. In the concrete case of the NSPK protocol, there are two participants playing in two roles, an initiator and a responder, represented respectively by a and b. Their roles correspond to the execution of the two sequences of actions shown in Fig. 2: *init*-run, by the principal named a, and *resp*-run, by the principal named b. We say that these runs are *symbolic* as they contain variables which can be instantiated to generate sequences of concrete message exchanges, i.e. concrete protocol runs.

Looking at these two sequences, it is immediately apparent that they properly represent the behavior of the protocol participants playing in the respective roles. In particular, given the available information at each step: *(i)* the content of each of the received messages can be decomposed and checked, and *(ii)* each of the sent messages can be constructed.

2.3 Message forwarding

Things are generally not as simple as in the NSPK protocol. In fact, in Alice&Bob–style protocol descriptions, it is left implicit that certain messages should be analyzed after they are received, and their contents checked, while other messages must be synthesized before they are sent. Clearly, a participant in NSPK must analyze the content of every message he receives in order to be able to send his next message. However, this may not always be the case, as we illustrate by means of another concrete example: in Fig. 3 we present the Otway-Rees Authentication/Key-Exchange Protocol (OR) [6], where an initiator a and a responder b attempt to mutually authenticate each other

$$
\begin{array}{lllll}
(\mathbf{or}_1) & a \to b & : & (n_1).\, i; a; b; \{n_1; i; a; b\}_{K_{as}} \\
(\mathbf{or}_2) & b \to s & : & (n_2).\, i; a; b; \{n_1; i; a; b\}_{K_{as}}; \{n_2; i; a; b\}_{K_{bs}} \\
(\mathbf{or}_3) & s \to b & : & i; \{n_1; K_{ab}\}_{K_{as}}; \{n_2; K_{ab}\}_{K_{bs}} \\
(\mathbf{or}_4) & b \to a & : & i; \{n_1; K_{ab}\}_{K_{as}}
\end{array}
$$

Fig. 3. The Otway-Rees Authentication/Key-Exchange Protocol (OR).

and exchange a shared key K_{ab} with the help of a server s, with whom they respectively share the keys K_{as} and K_{bs} (i is a run identifier, contained in all of the four messages). A direct transcription of messages into actions, as done with NSPK, yields the sequence of actions *resp*-run shown on the left of Fig. 4. Note that, in contrast to the NSPK example, this sequence does not properly represent the expected behavior of the responder. The problem is with b's first action in the run, where he receives $i; a; b; \{n_1; i; a; b\}_{K_{as}}$. Because he does not possess the key K_{as}, he cannot check that the fourth part of this message is of the form $\{n_1; i; a; b\}_{K_{as}}$, i.e. the concatenation of the plaintext $i; a; b$ with some nonce n_1 generated by a.

In this protocol, we must interpret the submessage $\{n_1; i; a; b\}_{K_{as}}$ differently. For b, this submessage is *opaque*: it represents a chunk of information that he cannot decompose. In this protocol, he should simply forward this chunk to s in the second step just as he must later forward to a the submessage $\{n_1; K_{ab}\}_{K_{as}}$ that he receives from s. The proper way of formally representing such situations is to replace these chunks by new *message variables* in b's actions, as shown in the symbolic run *resp*-possrun on the right of Fig. 4 using the new message variables m_1 and m_2. [4]

The use of message variables allows us to represent message forwarding when translating Alice&Bob–style protocol descriptions into formal models. The symbolic run *resp*-possrun allows m_1 and m_2 to be instantiated with

[4] We proceed similarly in the case of asymmetric encryption. For instance, in

$$
\begin{array}{lll}
a \to b : (n_1, n_2).\ \{a; n_1; c\}_{K_b}; \{a; b; n_1; n_2\}_{K_c} \\
b \to c : \qquad\qquad a; b; n_1; \{a; b; n_1; n_2\}_{K_c}
\end{array}
$$

b is not supposed to check the contents of $\{a; b; n_1; n_2\}_{K_c}$ in the message that he receives. Not only is b not supposed to have K_c^{-1} to decrypt the message, but he also cannot reproduce the message (i.e. compose it himself) since he should not possess n_2. Indeed, for b, this could just be any message that he later forwards to c. If we introduce a new variable m to represent that message, then the possible runs of b should start with the more general $\mathbf{r}(\{a; n_1; c\}_{K_b}; m)$ followed by $\mathbf{s}(a; b; n_1; m, c)$ instead of the two actions $\mathbf{r}(\{a; n_1; c\}_{K_b}; \{a; b; n_1; n_2\}_{K_c})$ and $\mathbf{s}(a; b; n_1; \{a; b; n_1; n_2\}_{K_c}, c)$.

resp-run :	*resp*-possrun :
$\langle \mathbf{r}(i;a;b;\{n_1;i;a;b\}_{K_{as}})\,.$	$\langle \mathbf{r}(i;a;b;m_1)\,.$
$\mathbf{f}(n_2)\,.$	$\mathbf{f}(n_2)\,.$
$\mathbf{s}(i;a;b;\{n_1;i;a;b\}_{K_{as}};\{n_2;i;a;b\}_{K_{bs}},s)\,.$	$\mathbf{s}(i;a;b;m_1;\{n_2;i;a;b\}_{K_{bs}},s)\,.$
$\mathbf{r}(i;\{n_1;K_{ab}\}_{K_{as}};\{n_2;K_{ab}\}_{K_{bs}})\,.$	$\mathbf{r}(i;m_2;\{n_2;K_{ab}\}_{K_{bs}})\,.$
$\mathbf{s}(i;\{n_1;K_{ab}\}_{K_{as}},a)\rangle$	$\mathbf{s}(i;m_2,a)\rangle$

Fig. 4. The responder runs of OR.

messages that do not conform to the prescribed structure — the symmetric encryption of a nonce concatenated with the plaintext submessage at the beginning of the message — as the principal playing the responder role cannot tell the difference. We will show how to obtain, in general, the correct sequence of actions, i.e. the correct symbolic run, in a rigorous way.

2.4 Conditional abortion

In general, we will require that principals proceed eagerly and always check the contents of an encrypted or hashed message as soon as this is possible (if it is ever possible at all). For example, an encrypted message $\{M\}_K$ shall thus only be treated as such if the principal has the key to decrypt it, K^{-1}, or the ability to build it from M and K. Note that it is possible that such an encrypted message can only be analyzed sometime later. If that is the case, we assume that the principal will carry out this analysis, and abort the protocol run if the contents were not as expected.

For concreteness, observe that participants accumulate data incrementally during each protocol run. Hence, when new data is received, a principal may only then be able to determine that some previously received message does not have the required structure and hence abort the protocol. Consider, for instance, a situation where a principal A receives a message containing a submessage $\{M\}_K$ encrypted with a symmetric key K that A does not possess. Furthermore, suppose that, in some later step, A is sent K (or receives enough information to compose it herself) and can thus decrypt $\{M\}_K$. In this case, A should abort the execution of the protocol whenever M does not conform to the protocol description (for example, it is not a nonce that A generated and sent to some other agents earlier in the protocol run). [5]

[5] As an example involving asymmetric encryption, consider the following extension of the

$$
\begin{array}{lcll}
(\mathbf{asw}_1) & a \to b & : & (n_1).\, \{K_a; K_b; t; H(n_1)\}_{K_a^{-1}} \\
(\mathbf{asw}_2) & b \to a & : & (n_2).\, \{\{K_a; K_b; t; H(n_1)\}_{K_a^{-1}}; H(n_2)\}_{K_b^{-1}} \\
(\mathbf{asw}_3) & a \to b & : & n_1 \\
(\mathbf{asw}_4) & b \to a & : & n_2
\end{array}
$$

Fig. 5. The Exchange Subprotocol of the ASW Protocol, simplified.

To give an example of where a similar situation occurs with message hashes, consider the ASW Protocol, an optimistic fair-exchange protocol for contract signing proposed by Asokan, Shoup and Waidner in [2]. Fig. 5 displays (a slightly simplified version of) the Exchange Subprotocol of ASW. The idea is that if two honest participants execute this subprotocol, and there are neither network failures nor intruder intervention, then afterwards both will possess a valid contract. We write t to denote the contract text, and write $\{M\}_{K_a^{-1}}$ to denote the digital signature of message M by principal a, whose public key for signature verification is K_a. The principals a and b generate nonces N_1 and N_2, which are called their respective *secret commitments* to the contract. Given these, they compute their *public commitments* by hashing these values, yielding $H(N_1)$ and $H(N_2)$, respectively. The protocol then proceeds in two rounds: in the first, each principal expresses his public commitment to the agreed-upon contract but does not disclose his secret commitment. In the second round, they then exchange their respective secret commitments. Each principal can then hash the secrecy commitment received and verify that it indeed corresponds to the public commitment from the first round. At the end of this exchange, each principal possesses a valid standard contract of the form $\{K_a; K_b; t; H(n_1)\}_{K_a^{-1}};\ \{\{K_a; K_b; t; H(n_1)\}_{K_a^{-1}}; H(n_2)\}_{K_b^{-1}};\ n_1;\ n_2$.

Fig. 6 displays the responder run *resp*-run (written horizontally for the sake of readability). As with the OR protocol, this sequence does not properly represent the expected behavior of the responder. The problem is that before carrying out the action $\mathbf{r}(n_1)$, the principal b cannot check the structure of the submessage $H(n_1)$, even though he knows the hash function H. Therefore, until n_1 is received, the submessage $H(n_1)$ is again just an opaque chunk of

---

message exchanges of Footnote 4:

$$
\begin{array}{lll}
a \to b : (n_1, n_2). & \{a; n_1; c\}_{K_b}; \{a; b; n_1; n_2\}_{K_c} \\
b \to c : & a; b; n_1; \{a; b; n_1; n_2\}_{K_c} \\
c \to b : & \{n_2\}_{K_b}
\end{array}
$$

After b receives the message from c, he can check that the m that he previously received and forwarded to c has indeed the required format. In this case, b's checking of the message can take place not because b has possession of K_c^{-1}, but rather because he now has n_2 and thus he can construct the message $\{a; b; n_1; n_2\}_{K_c}$ and compare it with m.

resp-run :

$\langle \mathbf{r}(\{K_a; K_b; t; H(n_1)\}_{K_a^{-1}}) . \quad \mathbf{f}(n_2) . \mathbf{s}(\{\{K_a; K_b; t; H(n_1)\}_{K_a^{-1}}; \quad H(n_2)\}_{K_b^{-1}}, a) . \mathbf{r}(n_1) . \quad \mathbf{s}(n_2, a)\rangle$

resp-possrun :

$\langle \mathbf{r}(\{K_a; K_b; t; m_1\}_{K_a^{-1}})\rangle$

$\langle \mathbf{r}(\{K_a; K_b; t; m_1\}_{K_a^{-1}}) . \quad \mathbf{f}(n_2)\rangle$

$\langle \mathbf{r}(\{K_a; K_b; t; m_1\}_{K_a^{-1}}) . \quad \mathbf{f}(n_2) . \mathbf{s}(\{\{K_a; K_b; t; m_1\}_{K_a^{-1}}; \quad H(n_2)\}_{K_b^{-1}}, a)\rangle$

$\langle \mathbf{r}(\{K_a; K_b; t; H(n_1)\}_{K_a^{-1}}) . \quad \mathbf{f}(n_2) . \mathbf{s}(\{\{K_a; K_b; t; H(n_1)\}_{K_a^{-1}}; \quad H(n_2)\}_{K_b^{-1}}, a) . \mathbf{r}(n_1)\rangle$

$\langle \mathbf{r}(\{K_a; K_b; t; H(n_1)\}_{K_a^{-1}}) . \quad \mathbf{f}(n_2) . \mathbf{s}(\{\{K_a; K_b; t; H(n_1)\}_{K_a^{-1}}; \quad H(n_2)\}_{K_b^{-1}}, a) . \mathbf{r}(n_1) . \quad \mathbf{s}(n_2, a)\rangle$

Fig. 6. The responder runs of ASW.

information to be stored. However, when receiving n_1, b should hash it and abort the protocol execution if it does not coincide with the opaque submessage previously stored. In this case, he should not even execute the last sending action of the run. This kind of problem is standard for protocols involving commitments to values by principals.

We propose to tackle this conditional abortion problem, as well as the message forwarding problem we described in the previous subsection, by introducing *incremental symbolic runs*. In these runs, we use variables to represent opaque submessages (as explained before), and we characterize each of the execution steps in order to build a collection of symbolic subruns of increasing lengths, where the structure of messages reflects the growth of the relevant data during protocol execution. Said another way, the structure of messages in each subrun reflects the data possessed by the principals up to that point in the execution. To illustrate this, the proper way of formally representing the responder run of the ASW protocol is, instead of *resp*-run, the sequence *resp*-possrun of subruns (a sequence of sequences, of growing length) shown in Fig. 6.

We will now formalize these intuitive explanations and notions.

3 Incremental symbolic runs

3.1 The direct-compilation approach

In general, a protocol description in Alice&Bob–notation involves j principal variables $a_1, \ldots, a_j$, corresponding to j distinct protocol participants, each playing a role, and k number variables $n_1, \ldots, n_k$. We write $Part = \{a_1, \ldots, a_j\}$ to denote the set of all protocol participants. A protocol description consists of a sequence $\langle \mathbf{step}_1 \ldots \mathbf{step}_m \rangle$ of message exchange steps, each of the form

$$(\mathbf{step}_q) \quad x_s \rightarrow x_r \ : \ (n_{q_1}, \ldots, n_{q_t}).\ M\,,$$

where $x_s \neq x_r$ and M includes at least all of $n_{q_1}, \ldots, n_{q_t}$ but can also involve any of the number variables generated in previous steps, as well as any of the principal variables identifying the protocol participants. The variables $n_{q_1}, \ldots, n_{q_t}$ are supposed to represent values that must be freshly generated by x_s just before the message M is sent to x_r. We will assume that these values are fresh, i.e. that they do not occur in any message of the preceding steps of the protocol description, nor in the inverse of any of its submessages. This means that the new numbers have not appeared before and cannot be used to generate the inverse of any key already used.

These steps are meant to prescribe a sequence of actions to be executed by each of the participants in a run of the protocol. Let us assumed fixed a protocol description in the following definition and in the remainder of the paper, so that, for example, we simply speak of "the participants" meaning the participants of the given protocol.

Definition 3.1 *Let $x \in Part$. The sequence of actions corresponding to the execution of x's role in the protocol is x-run $= \mathbf{step}_1^x \centerdot \cdots \centerdot \mathbf{step}_m^x$, where $\mathbf{step}_q^x$ is defined by*

$$\mathbf{step}_q^x = \begin{cases} \langle \boldsymbol{f}(n_{q_1}) \ldots \boldsymbol{f}(n_{q_t}) \,.\, \boldsymbol{s}(M, x_r) \rangle & \text{if } x = x_s, \\ \langle \boldsymbol{r}(M) \rangle & \text{if } x = x_r, \\ \langle\rangle & \text{otherwise.} \end{cases}$$

We call this the *direct-compilation approach* as x-run directly formalizes the sequence of actions to be executed by the protocol role, as described in the Alice&Bob–style protocol description (cf. the x-runs in Fig. 2, Fig. 4, and Fig. 6). However, as we remarked in the previous sections, a symbolic run such as x-run may not be enough, not only because it does not take message forwarding and conditional abortion into account, but also because it may happen that the correct partial executions of a run cannot be modeled simply by considering prefixes of x-run.

3.2 The knowledge of principals

The construction should therefore be guided by the role-relevant data that each principal collects during his run of the protocol. In general, messages are *analyzed* (decomposed) and *synthesized* (composed) by following simple rules.

Definition 3.2 *Let S be a set of messages. The set analyz(S) is the least superset of S closed under the rules*

$$\frac{M_1; M_2}{M_1}, \qquad \frac{M_1; M_2}{M_2}, \qquad \frac{\{M\}_K \quad K^{-1}}{M},$$

and the set synth(S) is the least superset of S closed under the rules

$$\frac{M_1 \quad M_2}{M_1; M_2}\,, \qquad \frac{M \quad K}{\{M\}_K}\,, \qquad \frac{M}{H(M)}\,.$$

The least superset of S closed under the analysis and synthesis rules is denoted by close(S).

It is quite straightforward to show that if one does not allow encryption using composed messages as keys, i.e. if one considers only atomic keys, then $close(S) = synth(analyz(S))$. Below, we will also identify another situation where this is true. In general, however, $synth(analyz(S)) \subsetneq close(S)$, that is, the inclusion is proper. For instance, if S contains just the message $M_1; \{M\}_{(M_1;M_1)^{-1}}$ then $M \in close(S)$ but $M \notin synth(analyz(S))$. If $S = close(S)$ then we shall say that S is a *closed* set of messages.

For a given $x \in Part$, assume now that $x\text{-run} = \langle \mathbf{a}_1, \ldots, \mathbf{a}_s \rangle$. For $1 \leq i \leq s$, we want to define $x\text{-possrun}^i$ as the *view* that x has of his run up to the ith action, as explained above. Of course, we must rely on the data that x collects during the actions he took up to that point in the run, assuming that he knows some initial protocol-relevant data. If we call this data D_x^i, then we can visualize the evolution of data sets during $x\text{-run} = \langle \mathbf{a}_1, \ldots, \mathbf{a}_s \rangle$ as follows:

$$D_x^0 \xrightarrow{\mathbf{a}_1} D_x^1 \xrightarrow{\mathbf{a}_2} D_x^2 \xrightarrow{\mathbf{a}_3} \cdots \xrightarrow{\mathbf{a}_{s-1}} D_x^{s-1} \xrightarrow{\mathbf{a}_s} D_x^s \quad .$$

Obviously, $D_x^i \subseteq D_x^{i+1}$ and D_x^0 should consist of only the protocol-relevant data that the principal x holds before he starts his role of the protocol, e.g. the identities of the participants, their public keys, and x's own private and shared keys. Then D_x^{i+1} constitutes the extension of D_x^i by the new information that x gets by executing the action $\mathbf{a}_{i+1}$. It should be clear that nothing new is obtained by sending a message.

Definition 3.3 *The data collected by a principal executing an action $\boldsymbol{a}$ is the set $gets(\boldsymbol{a})$ defined by:*

$$gets(\boldsymbol{a}) = \begin{cases} \emptyset & \text{if } \boldsymbol{a} = \boldsymbol{s}(M, y), \\ \{M\} & \text{if } \boldsymbol{a} = \boldsymbol{r}(M), \\ \{n\} & \text{if } \boldsymbol{a} = \boldsymbol{f}(n). \end{cases}$$

The data sets can now be defined as follows:

Definition 3.4 *The sets D_x^i, for $0 \leq i \leq s$, are inductively defined by:*

- $D_x^0 = close(Part \cup \{K_y \mid y \in Part\} \cup \{K_x^{-1}\} \cup \{K_{xy} \mid y \in Part\})$, *and*

- $D_x^i = close(D_x^{i-1} \cup gets(\boldsymbol{a}_i))$ *if* $i > 0$.

Clearly, for every $i > 0$, we have that

$$D_x^i = \begin{cases} D_x^{i-1} & \text{if } \mathbf{a}_i = \mathbf{s}(M, y), \\ close(D_x^{i-1} \cup \{M\}) & \text{if } \mathbf{a}_i = \mathbf{r}(M), \\ close(D_x^{i-1} \cup \{n\}) & \text{if } \mathbf{a}_i = \mathbf{f}(n). \end{cases}$$

3.3 Executability

The construction of the sets D_x^i completely neglects the messages that x sends during his run. This is justified: x does not learn anything by sending a message, as reflected in Definition 3.3. However, it must be the case that x can build the messages that he sends using the data currently available to him. This assumption, which is often left implicit in protocol analysis approaches, can be formalized in the present setting.

Definition 3.5 *The role of* x *is* executable *provided that, for every* $1 \leq i \leq t$, *if* $\boldsymbol{a}_i = \boldsymbol{s}(M, y)$ *then* $M \in D_x^{i-1}$. *The protocol is itself* executable *if all of its roles are.*

Although executability is somewhat orthogonal to the problem we are discussing here, we will make use of it later on. From now on, in any case, we will assume that the protocol specifications that we work with are executable.

3.4 Opacity and transparency

From now on, let us also suppose that the data collected by a principal x up to a given point in the execution of his role is the closed set D. We can now define when the precise form of a message can be understood by a principal. Indeed, as we have seen in previous examples, it may be that the actual form of some messages cannot be understood given the available data. To provide an adequate symbolic treatment of messages, we introduce new *message variables* m_M for each encrypted or hashed submessage, respectively $M = \{M'\}_K$ or $M = H(M')$. These are precisely the (sub)messages that are *opaque* to a principal, in the sense that the currently available data does not allow him to "read through" (i.e. decompose) these (sub)messages, thereby recognizing their precise form and extracting their content.

Definition 3.6 *The* view $v_D(M)$ *that a principal has of a message* M *is*

defined inductively by

$$v_D(M) = \begin{cases} M & \text{if } M \text{ is atomic,} \\ v_D(M_1); v_D(M_2) & \text{if } M = M_1; M_2, \\ \{v_D(M_1)\}_{v_D(K)} & \text{if } M = \{M_1\}_K \text{ and } K^{-1} \in D \text{ or } M_1, K \in D, \\ H(v_D(M_1)) & \text{if } M = H(M_1) \text{ and } M_1 \in D, \\ m_M & \text{otherwise.} \end{cases}$$

It is clear that if a message variable m_M occurs in $v_D(M')$ then M must be an opaque submessage of M'. Note that the converse is false because an opaque submessage need not be outermost. For instance, $v_\emptyset(H(H(M))) = m_{H(H(M))}$, where of course $m_{H(M)}$ does not occur. This happens, despite the fact that $H(M)$ is a submessage of $H(H(M))$, because $H(M)$ appears only inside $H(H(M))$ which is itself opaque.

We now define what it means for a message to be *transparent* or *opaque*, given D.

Definition 3.7 *A message M is D-*transparent *if $v_D(M) = M$ and D-*opaque *if $v_D(M) = m_M$.*

We will also say that a set of messages S is D-opaque, or D-transparent, provided that all the messages in S are. In case D is itself D-transparent, we will simply refer to D as transparent. Note that it follows from Definition 3.6 that M is D-opaque precisely when $M = \{M_1\}_K$, $K^{-1} \notin D$ and $\{M_1, K\} \not\subseteq D$, or else if $M = H(M_1)$ and $M_1 \notin D$. Obviously, given the definitions, D-transparency is related to the absence of D-opaque submessages. By induction on the structure of messages, we can prove:

Proposition 3.8 *A message M is D-transparent if and only if $sub(M)$ does not contain D-opaque elements.*

To better understand how the view that a principal has of a message may evolve, we also prove:

Proposition 3.9 *Let $D \subseteq D'$, with D' closed. Given a message M' the following are equivalent:*

(i) $v_D(M') = v_{D'}(M')$,

(ii) *if m_M occurs in $v_D(M')$ then it also occurs in $v_{D'}(M')$,*

(iii) *if m_M occurs in $v_D(M')$ then M is D'-opaque.*

The implications from (i) to (ii) and (ii) to (iii) are straightforward. A simple induction on the structure of M' establishes the implication from (iii) to (i).

Note that if $D \subseteq D'$ then it is immediate that D'-opaque messages are also D-opaque. Hence, as a corollary of the previous proposition, if D'' is also closed and $D \subseteq D' \subseteq D''$ then $v_D(M') = v_{D''}(M')$ implies that $v_D(M') = v_{D'}(M') = v_{D''}(M')$.

The following proposition tells us that if an opaque message does not appear as a submessage of any message in a set S, then it cannot appear as a submessage of any message in $close(S)$.

Proposition 3.10 *Let M be a D-opaque message and let $S \subseteq D$. Then, $M \in sub(close(S))$ if and only if $M \in sub(S)$.*

The direct implication follows by induction on the closure rules; it suffices to show that all the rules preserve the absence of M as a submessage of the messages involved. The converse implication is trivial due to the monotonicity of *sub* and the reflexivity of *close*.

Proposition 3.10 has the next two propositions as immediate corollaries.

Proposition 3.11 *If S is a set of atomic messages then $close(S)$ is transparent.*

Note that, by definition, atomic messages are always transparent. Hence, if S is a set of atomic messages then $sub(S) = S$ does not contain opaque elements. Therefore, Proposition 3.10 implies that also $sub(close(S))$ does not contain opaque elements. Finally, Proposition 3.8 guarantees that $close(S)$ is transparent.

Proposition 3.12 *Let M' be a message and $D' = close(D \cup \{M'\})$. If D is transparent then, D' is transparent if and only if M' is D'-transparent.*

By induction on the structure of messages, we can obtain a result similar to the one in Proposition 3.10 that explains how message variables appear during protocol execution:

Proposition 3.13 *Let S be a set of messages and $D' = close(D \cup S)$. If $M' \in D'$, $M \notin sub(D)$, and m_M occurs in $v_{D'}(M')$ then m_M also occurs in $v_{D'}(S)$.*

To conclude this sequence of technical results, we consider what happens to the notion of opacity when D is augmented, not by an arbitrary message, but by a fresh number. (Recall that freshness means that the value does not occur in any previous message, nor in the inverse of a submessage of any previous message.)

Proposition 3.14 *Let n be a number variable such that $n \notin sub(D \cup sub(D)^{-1})$, and $D' = close(D \cup \{n\})$. If $M \in sub(D)$ and M is D-opaque,*

then M is also D'-opaque.

3.5 Incremental runs

Our aim now is to define the sequence of symbolic runs of growing length that adequately models the execution of each role of the protocol. For each participant x, we need to pick each prefix of x-run and apply to it the view that results from the data x possesses up to that point in the execution. For this purpose, we extend the definition of the view v to actions and sequences of actions in the natural way.

Definition 3.15 *Let $x \in Part$. If $s = |x\text{-run}|$, then the sequences of actions corresponding to the (possibly partial) execution of x's role of the protocol are $x\text{-possrun}^1, \ldots, x\text{-possrun}^s$, where $x\text{-possrun}^i = v_{D_x^i}(x\text{-run}|_i)$ for $1 \leq i \leq s$.*

In general, x-run can be understood as a *perfect* description of a complete run of the protocol for participant x, which complies to the protocol description in a full, although maybe not realizable, way. In contrast, x-possrun^s can be understood as the *possible*, though maybe imperfect, complete run, in the sense that it correctly models the execution of each role of the protocol taking into account the available data and the way messages can be manipulated.

Still, in some cases, the direct approach of just considering x-run (and its prefixes) "works fine", in the sense that, for each role, the perfect complete run is *representative* of all possible partial runs. More precisely, x-run is representative if x-possrun$^i = x$-run$|_i$ for every $1 \leq i \leq s$. This certainly holds true in the case of NSPK, but the same cannot be said about OR and ASW. Below, we provide a complete characterization of the situations when the direct-compilation approach works.

Proposition 3.16 *The sequence x-run is representative if and only if every received message is transparent when it is received, i.e. if $\boldsymbol{a}_i = \boldsymbol{r}(M)$, then M is D_x^i-transparent.*

The direct implication is straightforward, and the converse follows by induction on the sequence of actions in the execution of each role, by exploiting Propositions 3.11 and 3.12.

Cases such as that of the OR protocol, however, still maintain some of the regularity of the simplest cases, namely the absence of abortion conditions. Indeed, the complete possible run x-possrun^s is still representative of all the others, that is, x-possrun$^i = x$-possrun$^s|_i$ for every $1 \leq i \leq s$. The next proposition gives us a precise characterization of these situations.

Proposition 3.17 *The sequence x-possrun^s is representative if and only if every received message preserves the message variables that occur in the views*

of previously received messages, i.e. if $1 \leq j < i \leq s$, $\boldsymbol{a}_j$ *and* $\boldsymbol{a}_i$ *are receiving actions, and* m_M *occurs in* $v_{D_x^{i-1}}(\boldsymbol{a}_j)$*, then* m_M *also occurs in* $v_{D_x^i}(\boldsymbol{a}_j)$*.*

Proposition 3.9 plays an essential role in the proof of this result. The direct implication follows easily. The converse implication follows by a careful inspection of the sequence of actions in the execution of each role, by exploiting Propositions 3.10, 3.13, and 3.14.

Note that in this case only message forwarding may be necessary. The precise meaning of *forwarding* can also be clarified with the help of Propositions 3.9, 3.10, and 3.13, as explained in the proof of Proposition 3.17: if a sent message contains an opaque submessage M then M must also occur, and be opaque, in some previously received message.

Of the examples we have considered, ASW is the most complex one because the growth of the data available to the principal executing the role may turn opaque elements into non-opaque ones. When that happens, namely if $\mathbf{a}_i$ is a receiving action, m_M occurs in $v_{D_x^{i-1}}(\mathbf{a}_j)$ for some $j < i$ but m_M does not occur in $v_{D_x^i}(\mathbf{a}_j)$, we claim that the protocol participant x should abort the execution of his role of the protocol whenever the actual values of m_M and $v_{D_x^i}(M)$ do not coincide. This is what we call *conditional abortion*.

3.6 Modeling security protocols

We can now conclude our investigation of the "correct" way of modeling security protocols specified using Alice&Bob–notation. As we have shown above, making the usual assumption that the sequence x-run and its prefixes represent the behavior of a principal executing the role may lead one to considering models that do not comply with all the possibilities allowed by the protocol. Instead, we claim that all the incremental sequences x-possrun must be considered.

We now explain how to concretize the symbolic information contained in these sequences. A *protocol instantiation* is a variable substitution σ that assigns to each variable a ground message (that is, without variables) of the same type. Namely, σ assigns a principal identifier to each principal variable, a number to each number variable, and a message to each message variable. Moreover, σ should be injective on the protocol participants, that is, if a_1 and a_2 are two principal variables then $\sigma(a_1) \neq \sigma(a_2)$. We extend σ to messages, actions, and sequences in the natural way.

In general, if we denote the set of all protocol instantiations by *Inst*, the set of all possible concrete protocol runs of a principal A, in any of the j roles, is given by $Runs_A = \bigcup_{i=1}^{j} \bigcup_{l=1}^{|a_i\text{-run}|} \{\sigma(a_i\text{-possrun}^l) \mid \sigma \in Inst, \sigma(a_i) = A\}$.

Denotational protocol models, for instance, should then be built by choos-

ing, for each principal A, a set of (prefixes of) concrete sequences in $Runs_A$. For example, in strand spaces, each of these sequences should correspond to a strand.

In operational protocol models, on the other hand, the abortion conditions come into play in a meaningful way. In this case, it suffices to execute, stepwise, the sequence $\sigma(\langle v_{D_x^1}(\mathbf{a}_1), \ldots, v_{D_x^s}(\mathbf{a}_s)\rangle)$, and to guard the execution of each action with a condition whose failure should lead to abortion. For instance, $\sigma(v_{D_x^i}(\mathbf{a}_i))$ can occur, after the sequence $\sigma(\langle v_{D_x^1}(\mathbf{a}_1), \ldots, v_{D_x^{i-1}}(\mathbf{a}_{i-1})\rangle)$ is complete, but only if $\sigma(m_M) = \sigma(v_{D_x^i}(M))$ for each message variable m_M that has appeared so far. A full-fledged formalization of this process would require a more accurate representation of the principals' activity by employing additional internal actions. We leave this for future work.

4 Concluding remarks

We have shown that, despite the fact that the Alice&Bob–notation does not include explicit control flow constructs, it is possible to make aspects such as message forwarding and conditional abortion explicit when producing formal protocol models without having to resort to more expressive protocol description languages. In particular, we have shown that, when considering such aspects, the direct-compilation approach does not correctly formalize all the possible behaviors of the principals, for which incremental symbolic runs are required.

Several expressive protocol specification languages have been proposed to make explicit what is left implicit (or even unspecified) in Alice&Bob–style descriptions. Some of the problems that we have investigated here have also been considered for other specification languages. For instance, the syntax of Casper [10] includes a "%"-notation for representing unreadable messages by means of variables. The introduction of this notation requires an explicit extension of the standard Alice&Bob–notation and direct assistance by the user in writing protocol descriptions. In contrast, we have shown how to automatically handle opacity without changing the notation or involving the user by instead changing the interpretation of the terms used in Alice&Bob–notation.

The way we define the view that a principal can have of a message given his current knowledge is similar to the message patterns considered by Abadi and Rogaway in [1], albeit for a rather different purpose (reconciling cryptographic and formal methods protocol analysis approaches). The main difference is that we distinguish between distinct opaque elements by means of distinct variables, whereas Abadi and Rogaway use only one variable (their

"box" symbol). This difference reflects the different objectives: Abadi and Rogaway aim to define a notion of equivalence between messages up to opaque submessages, while we are interested in describing precisely all the possible concrete instances of messages that match the pattern. In this sense, different variables can be given different values, which cannot be done if we use the same variable. Moreover, even if two messages are opaque, one can certainly still compare them and check if they are equal.

In §3.6, we have also briefly hinted at how our work could be integrated with the strand space protocol analysis approach. Capturing the non-determinism that results from message forwarding and conditional abortion would not require an enrichment of the strand space approach itself; rather, one could just enlarge the number of strands in the space that models the protocol under consideration. However, in the general case, i.e. for protocols with explicit control flow, the extension of strand spaces with a notion of *conflict* as suggested in [7,9] seems to be the only option.

As we have previously remarked, we have only focused in this paper on sending and receiving messages and generating fresh numbers. However, other internal actions (e.g. corresponding to the application of cryptographic operations) can be modeled similarly and our results extended straightforwardly. Such an extension would allow us to fully formalize the process of compiling messages to sequences of actions and thus extend the ideas described in §3.6 to directly build analysis tools based on them. We also believe that incremental symbolic runs will provide a good basis for generating protocol implementations from Alice&Bob-style descriptions that explicitly carry out necessary executability and abortion tests.

References

[1] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[2] N. Asokan, V. Shoup, and M. Waidner. Asynchronous protocols for optimistic fair exchange. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 86–99, 1998.

[3] C. Caleiro, L. Viganò, and D. Basin. Deconstructing Alice and Bob (extended version). Technical Report 486, Department of Computer Science, ETH Zurich, 2005.

[4] C. Caleiro, L. Viganò, and D. Basin. Metareasoning about Security Protocols using Distributed Temporal Logic. In *Proc. IJCAR'04 Workshop on Automated Reasoning for Security Protocol Analysis (ARSPA'04)*, pages 67–89. ENTCS 125(1), 2005.

[5] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. In *Proc. SAPS'04*. Austrian Computer Society, 2004.

[6] J. Clark and J. Jacob. A Survey of Authentication Protocol Literature: Version 1.0, 17. November 1997. URL: www.cs.york.ac.uk/~jac/papers/ drareview.ps.gz.

[7] F. Crazzolara and G. Winskel. Composing strand spaces. In *Proc. FST TCS 2002*, LNCS 2556, pages 97–108. Springer-Verlag, 2002.

[8] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[9] J. Y. Halpern and R. Pucella. On the relationship between strand spaces and multi-agent systems. *ACM Trans. Info. and System Security*, 6(1):43–70, 2003.

[10] G. Lowe. Casper: a Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[11] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[12] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *Modelling and Analysis of Security Protocols*. Addison Wesley, 2000.

[13] F. J. Thayer Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7:191–230, 1999.