Invariant-based reasoning about parameterized security protocols

Arjan J. Mooij

DOI 10.1007/s00165-009-0104-0

Formal Aspects of Computing (2010) 22: 63–81

Formal Aspects of Computing

Invariant-based reasoning about parameterized security protocols

Arjan J. Mooij[1]

Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, Eindhoven, The Netherlands.
E-mail: A.J.Mooij@tue.nl

Abstract. We explore the applicability of the programming method of Feijen and van Gasteren to the domain of security protocols. This method addresses the derivation of concurrent programs from a formal specification, and it is based on common notions like invariants and pre- and post-conditions. We show that fundamental security concepts like secrecy and authentication can nicely be specified in this way. Using some small extensions, the style of formal reasoning from this method can be applied to the security domain. To demonstrate our approach, we discuss an authentication protocol and a public-key distribution protocol, and we deal with their composition. By focussing on a general setting where agents run the protocols multiple times, the nonce concept turns out to pop-up naturally. Although this work does not contain any new protocols, it does offer a new view on reasoning about security protocols.

Keywords: Mathematical techniques, Program construction, Refinement, Security

1. Introduction

The formal correctness of security protocols is generally considered to be important. The most common way to establish it is by means of verification after the protocols have been designed. The usual verification styles include automated state-space exploration [Low96] and interactive theorem proving [Pau98]. However, actually designing and reasoning about security protocols is still considered to be complicated.

We consider an approach in which, based on a formal specification, the protocols are constructed, or reconstructed, hand-in-hand with their formal correctness proof. Such formal derivation approaches are insightful, but so far they have received less attention [But99, Fid01]. Our goal is not the automatic synthesis of security protocols [PS00], but an effective and systematic way of reasoning about them.

The first step in a derivation is a proper specification. Such a specification must be formal and manageable, i.e., it must be precise yet comprehensible, and it must be easy to manipulate. In the case of security protocols, often dedicated belief logics and calculi are used, e.g., in [BAN90, AFS97]. Although these give the desired formality, there are dangers of misinterpreting the used concepts, and often a large number of calculational rules is introduced. Like the approaches mentioned in [RZ97], we use familiar specification concepts. In our case, we show how notions like secrecy and authentication [Low97] can conveniently be modeled in terms of pre- and post-conditions, and invariants. An overview of the use of invariants for the analysis of security protocols can be

Correspondence and offprint requests to: A. J. Mooij, E-mail: A.J.Mooij@tue.nl

[1] This research has mainly been performed at the School of Computer Science, The University of Nottingham, UK.

found in [Mea00]. In particular, the invariants from [Sch97] are extremely specialized, in the sense that they are just about a transmitted message instead of about an entire state of the system.

For the reasoning about security protocols, various dedicated approaches have been proposed, e.g., in [TFHG99]. In contrast, we study the use of a general method, which gives more flexibility, and enables the integration with techniques developed for other application domains. A similar approach has been explored in the context of data refinement [But02], but its complexity seems to remain a problem [Fid01].

Simplicity of exposition is at the core of the method of Feijen and van Gasteren [FvG99] for constructing concurrent programs. It is based on the classical axiomatic theory of Owicki and Gries [OG76], where the reasoning about execution traces (like in [Pau98, TFHG99]) is replaced by the reasoning about assertions and invariants, i.e., predicate abstractions of states. This method has shown its merits in various application areas, e.g., [FvGS98, FvG99, Hoo06, Moo06]. In this work we explore its applicability to security protocols.

The typical derivations in this style [Dij76, FvG99] start with a series of desired post-assertions. To establish their correctness, some program statements are inserted together with again some required pre-assertions. This is repeated until the required pre-assertions are suitable as the precondition of the program. The aim is to motivate each derivation step using the existing statements and assertions, and to keep the program and its assertions as simple as possible. In this way these methods help to construct the assertions for a correctness proof, though this aspect is typically missing in other invariant-based approaches like [Sch97]. Although the method breaks with the more tempting forward style of reasoning, a similar backward style has roughly been sketched for security protocols in [AFS97].

We show that the interference caused by intruders has a limited influence on this method, as message communication in the presence of intruders is just a variation on communication using faulty channels. Hence we can largely reuse the corresponding reasoning techniques, which are based on common notions like invariants. These techniques are in particular very explicit in recording what needs to be derived from the message upon receipt, the importance of which has repeatedly been pointed out, e.g., in [AN96].

To demonstrate our approach we reconstruct the authentication protocol of Needham and Schroeder [NS78]. The well-known attack that was once detected by Lowe [Low96] is avoided in our derivations in a natural way. Apart from this classical authentication protocol, we also address a public-key distribution protocol, and we discuss their composition. In comparison to, e.g., [Low96, Sch97, Moo08], we directly address parameterized protocols, and hence we do not need additional lemmas to scale the results for a simple setting to some more general settings. Studying the parameterized case also turns out to be important for nicely introducing the nonce concept.

Overview In Sect. 2 we summarize the theories and methods that we use. Our derivation approach is described in Sect. 3, followed by our specification of authentication in Sect. 4. We demonstrate them using an authentication protocol in Sect. 5, and using a public-key distribution protocol in Sect. 6. Finally we draw some conclusions and sketch some further work in Sect. 7.

2. Preliminaries

In this section we summarize some basic material that we use in the remainder of this work.

2.1. Concurrent programs

A concurrent program consists of a precondition *Pre* and a number of sequential programs, which are called its components. The components are to be executed in parallel by interleaving their atomic statements.

The programming language that we use for the components is based on the Guarded Command Language from [Dij76]. The semantics of the atomic statements follows from the weakest liberal precondition, *wlp*, (and the weakest precondition, *wp*) predicate transformers. The weakest liberal precondition of a statement S and a predicate Q, to be denoted by $wlp.S.Q$, is the weakest predicate P that guarantees that each terminating execution of statement S starting from a state satisfying P establishes Q; in contrast to the *wp*, the *wlp* does not guarantee termination of the statement. In particular we use the following two kinds of atomic statements:

- assignment $v := E$, which assigns the value of expression E to variable v. Its *wlp* can be defined as

  $$[wlp.(v := E).Q \equiv (v := E).Q]$$

  where $(v := E).Q$ at the right-hand side denotes the substitution of expression E for variable v in predicate Q. The pair of square brackets [. . .] is a shorthand for "for all states", i.e., a universal quantifier binding all free program variables.
- non-deterministic assignment $v{:}P$, which assigns to variable v any value x that satisfies the condition $(v := x).P$. As long as there is no suitable value x, the execution of this statement is blocked. Its *wlp* can be defined as

  $$[wlp.(v : P).Q \equiv (\forall x :: (v := x).P \Rightarrow (v := x).Q)]$$

Although [FvG99] use so-called guarded skips for the synchronization between components, we will use non-deterministic assignments for this purpose, as these can also potentially block. Larger atomic statements can be constructed using atomicity brackets, e.g., $\langle S;\ T\rangle$, where S and T are statements, and blocking is only admitted in the very beginning. Its *wlp* can be defined as

$$[wlp.\langle S;\ T\rangle.Q \equiv wlp.S.(wlp.T.Q)]$$

2.2. Theory of Owicki and Gries

Partial correctness (or safety) is specified by annotating the program with assertions. An assertion is a predicate on the state of the system and it is located between brackets {. . .} at a control point of a component. The control points of a component are the locations between its atomic statements.

To prove partial correctness, we use the classical axiomatic theory from Owicki and Gries [OG76] using the nomenclature from [FvG99]. An annotated program is correct if each assertion is correct. In turn, an assertion P in a component is correct if it is both

- locally correct, i.e., it is established in the component:
  - if P is an initial assertion in the component: $[Pre \Rightarrow P]$ holds;
  - if P is preceded by an atomic statement $\{Q\}\ S$, where Q is a pre-assertion of statement S, then $[Q \Rightarrow wlp.S.P]$ holds.
- globally correct, i.e., it is maintained by all other components:
  - for each atomic statement $\{Q\}\ S$ in any other component, where Q is a pre-assertion of statement S, $[P \wedge Q \Rightarrow wlp.S.P]$ holds.

An invariant is an assertion that is located at every control point of each component. So an invariant is correct if it is both implied by the precondition, and maintained by each atomic statement in any component.

2.3. Method of Feijen and van Gasteren

The programming method from [FvG99] deals with the formal construction of concurrent programs hand-in-hand with a suitable annotation and a correctness proof. The method being based on the style of [Dij76], assertions play an important role. At a single control point, multiple assertions can be placed, which denotes their conjunction, and whose correctness can be proved separately. A queried assertion is an assertion that is required to hold, but whose correctness has not yet been proved; it is marked with a query '**?**'.

Program construction starts with a specification in terms of a preliminary program and some queried assertions, such as post-conditions. Derivations consist of turning each queried assertion, one-by-one, into a correct assertion. The proof obligations for local correctness lead to a style in which programs are constructed from the required assertions towards the initial control point. When all assertions (which include those from the specification) are correct, the program is correct with respect to the specification.

If a queried assertion's correctness (in the current annotated program) cannot yet be proved, there are mainly two solutions (which can also be combined):

- introduce additional queried assertions in the current annotation;
- modify the program.

An important issue is whether these solutions can endanger the correctness of the other assertions. Introducing additional assertions cannot endanger the correctness of the other assertions, and typically the weakest possible strengthening that serves the goal is calculated. However, modifying the program may transform all assertions into queried assertions again. The typically-used modification of the program is inserting a new statement (to establish the local correctness of an assertion).

Like its underlying theory [OG76], this method does not formally address progress. Since the role of progress in the derivations is often limited, progress is usually discussed in a pragmatic ad-hoc manner. Although in [DM06, DM08] we have shown how to integrate a full progress logic [DG06], in the current work we follow the original approach since progress plays no significant role.

2.4. Faulty communication channels

Our treatment of security protocols will be based on a technique for faulty communication channels. Faulty channels are channels that can duplicate, reorder and lose messages, but that cannot insert new messages or modify messages. Such channels have been studied in relation to, for example, alternating bit protocols [FvGS98, FvG99] and sliding window protocols [Hoo06]. In this section we discuss an operational model, and an elegant rule for proving that an assertion is established by the receipt of a message.

2.4.1. Operational model

In terms of shared variables, an operational model of message communication over a single faulty channel may look as follows:

snd x	:	$C := C \cup \{x\}$
Faulty channel	:	**do** *true* $\rightarrow$ $\langle\ y :\ y \in C$ $;\ R := R \cup \{y\}\ \rangle$ **od**
rcv $z \leftarrow m.z$	:	$\langle\ z :\ m.z \in R$ $;\ R := R \backslash \{m.z\}\ \rangle$

The variable C denotes the set of transmitted messages, and the variable R denotes the receipt buffer of any of the components. To model that messages may only arrive at some of the components, there is a variable R (and a faulty channel component) for each component in the system. The send statement "**snd** x" adds the message x to the set C, while the faulty-channel component repeatedly copies any message y from C into R. Injective function m maps any element of the type of variable z to a message, and it can be used to model the receipt of a data parameter z. The receive statement "**rcv** $z \leftarrow m.z$" non-deterministically assigns to variable z any value x such that the message $m.x$ can be removed from the set R; the receive statement is blocked as long as there is no suitable message in the set R. In the case that no parameters need to be received, we abbreviate the receive statement into "**rcv** m", where m denotes a single message.

The used variables C and R cannot occur in any other statements, nor in any parameter of the operations we have just defined. Hence the invariant $R \subseteq C$ is maintained, and we typically require this invariant as an implicit precondition. The practical implementability of the receive statements, e.g., in case z is encrypted in message $m.z$, must be checked separately, and we ignore this in our model. Notice that we do not mention the intended receiver in the send statement, even in the case of multiple possible receivers, so in fact we are modeling a faulty broadcast channel.

2.4.2. Rule of import and export

To avoid explicitly reasoning in terms of this operational model, an elegant proof rule can be derived. We discuss this rule in detail in order to facilitate its use in Sect. 3.1. Moreover, this section serves as a quick introduction to the style of program derivation, although in this section we will not introduce any new statements.

Consider an injective function m that maps a parameter to a message, and any predicate P that does not contain C or R. Given a send and a receive statement in some components, we want to derive a rule for establishing an assertion P after receiving such a message. It is assumed that the predicate $(v := x).P$, for any value x, cannot be violated by any other statement in the system. This specification can be summarized as follows:

rcv $v \leftarrow m.v$	
$\{$**?** $P\}$	**snd** $m.w$

Each of the two columns in this figure denotes a component, and v and w are variables. The assertions are printed in brackets $\{\ldots\}$, and the remaining proof obligation is marked with a query "**?**". Furthermore, we have deliberately omitted the faulty-channel component.

The global correctness of assertion P is guaranteed, as P does not contain C or R. For its local correctness we must consider the receipt of a message $m.x$, for any x, which corresponds to an assignment to variable v and removing the message $m.v$ from the receipt buffer R of the component. After computing the *wlp*, we obtain a pre-assertion

$$(\forall x :: \ m.x \in R \ \Rightarrow \ (v := x).P)$$

which we require as invariant. Thus we obtain the following intermediate program:

rcv $v \leftarrow m.v$	
$\{P\}$	**snd** $m.w$
Inv: **?** $(\forall x :: \ m.x \in R \ \Rightarrow \ (v := x).P)$	

For the maintenance of this queried invariant under the faulty-channel component, we must consider the copying of any transmitted message from C to the receipt buffer R. To this end we require the invariant

$$(\forall x :: \ m.x \in C \ \Rightarrow \ (v := x).P)$$

Using $R \subseteq C$, this invariant is stronger than the previous one, and hence we only need to consider this one.

For maintenance of this new invariant, we must consider the transmission of a message $m.w$, which corresponds to the addition of the message to the set C. To this end, we require the transmission to have a pre-assertion $(v := w).P$. For initial correctness of the invariant, we typically require the precondition $(\forall x :: \ m.x \notin C)$. Notice that C is not required to be empty; it only cannot contain these particular messages. Thus we obtain the following rule:

Pre: $(\forall x :: \ m.x \notin C)$	
rcv $v \leftarrow m.v$	$\{$**?** $(v := w).P\}$
$\{P\}$	**snd** $m.w$
Inv: $(\forall x :: \ m.x \in C \ \Rightarrow \ (v := x).P)$	

So, apart from any possible violation of the predicate $(v := x).P$, for any x, by the other statements in the system, the assertion P is established by this receipt of a message $m.v$ if before every transmission of any message $m.w$ the assertion $(v := w).P$ holds. This rule is called the rule of import (upon receipt) and export (upon transmission), and a similar rule exists for binary semaphores.

2.5. Security protocols

In the case of security protocols, the set of components consists of so-called honest components and intruder components. The honest components can be controlled (or programmed), while the intruders are outside our control. The components can communicate asynchronously using messages.

The typical kinds of messages can be defined recursively. An *atomic* (plaintext) message is just a data value, where we assume that values from different data types are different. Another view on it is assuming that the values are explicitly typed. In particular the data values include component identifiers, nonces (introduced later on) and keys. A *tupled* message $[y, z]$ contains the two messages y and z. An *encrypted* (ciphertext) message $\{y\}_k$ for any key k contains the message y. Often $\{y, z\}_k$ is used as an abbreviation of $\{[y, z]\}_k$.

All messages created in these three ways are assumed to be different. In particular we adopt the following freeness assumption [DY83], for any messages y and z, and keys k and l:

$$\{y\}_k = \{z\}_l \equiv y = z \wedge k = l$$

Such a black-box encryption model provides a convenient layer of abstraction between cryptography and concurrency, but it ignores so-called type flaw attacks. Although this is a common abstraction, its main drawback is that it never holds in practice: the model describes an unbounded number of messages, while practical messages are just bit-strings of bounded length.

The intruders are assumed to behave according to the Dolev/Yao abstraction [DY83], i.e., the intruders can intercept, read and create messages. Based on the observed messages, the intruders can also derive new messages. If any two messages y and z can be derived, then the tupled message $[y, z]$ can be derived. If any key k and any message y can be derived, then the encrypted message $\{y\}_k$ can be derived. If any tupled message $[y, z]$ can be derived, then the messages y and z can be derived. If any encrypted message $\{y\}_k$ and the key $\overline{k}$ can be derived, then the message y can be derived. The key $\overline{k}$ denotes the key that is required to undo the encryption with key k. In asymmetric encryption models the keys k and $\overline{k}$ cannot be derived from each other, while in symmetric encryption models we have $k = \overline{k}$.

Given the set C of transmitted messages (including the messages that are initially known to the intruders), the set of derivable messages E is defined formally as the smallest (but usually unbounded) set of messages such that the following five conditions hold, for any key k, and messages y and z:

- containment:

$$C \subseteq E$$

- composition:

$$y \in E \ \wedge \ k \in E \ \Rightarrow \ \{y\}_k \in E$$
$$y \in E \ \wedge \ z \in E \ \Rightarrow \ [y, z] \in E$$

- decomposition:

$$\{y\}_k \in E \ \wedge \ \overline{k} \in E \ \Rightarrow \ y \in E$$
$$[y, z] \in E \ \Rightarrow \ y \in E \ \wedge \ z \in E$$

3. Techniques

In this section we develop our techniques for the formal derivation of security protocols. In particular we focus on the consequences of the presence of intruders. In comparison to [Moo08], the techniques are further formalized, thus anticipating the future use of theorem provers. For compactness reasons, we will often abbreviate conditions like $y \in E$ into $E.y$.

3.1. Communication via intruders

The capabilities of the intruders are such that they can completely control the message flow in the system. We follow a common approach (e.g., [Sch97]) to model the intruders as a special type of communication channel. Such a channel is similar to a faulty channel, but with the extra capability of inserting new messages and modifying messages by composition and decomposition. In this section we adapt the material from Sect. 2.4 to the presence of intruders.

3.1.1. Operational model

In the operational model, we only replace the faulty-channel component by an intruder component. The intruder component can be obtained by replacing the assignment $y : \ y \in C$ by the assignment $y : \ y \in E$, where E is as defined in Sect. 2.5. Thus we obtain the following model:

snd x	:	$C := C \cup \{x\}$
Intruder	:	**do** *true* $\rightarrow$ $\langle \ y : \ y \in E$ $; R := R \cup \{y\} \ \rangle$ **od**
rcv $z \leftarrow m.z$	:	$\langle \ z : \ m.z \in R$ $; R := R \backslash \{m.z\} \ \rangle$

The used variables C, E and R cannot occur in any other statements, nor in any parameter of the operations we have just defined. Hence the invariant $R \subseteq E$ is maintained, and it is typically required as an implicit precondition.

3.1.2. Rule of import and export

The corresponding rule for establishing an assertion P (that does not contain the variables C, E and R) using the receipt of a message $m.v$ must also be adapted. As the model of the receive statement has not been modified, again the required invariant first becomes $(\forall x :: \ m.x \in R \ \Rightarrow \ (v := x).P)$, where R denotes the receipt buffer of the receiving component.

For the maintenance of this invariant under the intruder component, we must consider the addition of *derivable* messages to the receipt buffer R. To this end we replace it by the stronger required invariant

$$(\forall x :: \ E.(m.x) \ \Rightarrow \ (v := x).P)$$

All invariants about the message communication are of this particular shape, and hence we do not need to explicitly mention the receipt buffers R anymore. Like the invariant for faulty channels, its maintenance requires a pre-assertion $(v := w).P$ for the send statement; but this is not enough, as we discuss later on.

rcv $v \leftarrow m.v$	$\{$**?** $(v := w).P\}$
$\{P\}$	**snd** $m.w$
Inv: **?** $(\forall x :: \ E.(m.x) \ \Rightarrow \ (v := x).P)$	

This technique is related to several guidelines from [AN96]. In the first place, the invariant is very explicit about the meaning of the message $m.x$, viz., condition $(v := x).P$. The invariant also indicates how long the message $m.x$ needs to remain secret for the intruders, viz., until $(v := x).P$ holds. The special case that $[(v := x).P \ \equiv \ \mathit{false}]$, for any x, denotes secrecy of the message $m.x$. It is very reassuring that these formal techniques correspond to important guidelines from the field.

3.2. Generalization of the derivable messages

It turns out to be useful to generalize the notion of derivable messages E from Sect. 2.5 using a parameter K denoting a set of keys. Define the set E_K, for any set K, as the smallest set of messages such that the following five conditions hold, for any key k, and messages y and z:

- containment:

$$C \subseteq E_K$$

- composition:

$$E_K.y \ \wedge \ E_K.k \ \Rightarrow \ E_K.\{y\}_k$$
$$E_K.y \ \wedge \ E_K.z \ \Rightarrow \ E_K.[y, z]$$

- decomposition:

$$E_K.\{y\}_k \ \wedge \ (E_K.\overline{k} \ \vee \ \overline{k} \in K) \ \Rightarrow \ E_K.y$$
$$E_K.[y, z] \ \Rightarrow \ E_K.y \ \wedge \ E_K.z$$

The set E as defined in Sect. 2.5 corresponds to the instance $K = \emptyset$, and we use the constant ω to denote the set of all keys. The set K enables decomposition to also decompose messages using a key from K that could not be derived from C. If $K \subseteq E$, then the parameter K has no effect, while otherwise it has no natural interpretation in terms of the state of the intruders. Nevertheless, in Sect. 3.3.2 we will see that it aids our derivations by anticipating on keys that may be learnt in the future.

Notice that it is not useful to apply decomposition to a message that results from applying composition, as the set E_K would not be expanded, i.e., no new messages would be generated; see also [CJM98]. This can be made explicit in an alternative (but equivalent) definition of the set E_K using an auxiliary set D_K. To this end, we define the sets D_K and E_K as the smallest sets of messages such that the following six conditions hold:

$$
\begin{array}{l}
D_K \subseteq E_K \\
E_K.y \ \wedge \ E_K.k \ \ \Rightarrow \ \ E_K.\{y\}_k \\
E_K.y \ \wedge \ E_K.z \ \ \Rightarrow \ \ E_K.[y, z] \\
C \subseteq D_K \\
D_K.\{y\}_k \ \wedge \ (D_K.\overline{k} \ \vee \ \overline{k} \in K) \ \ \Rightarrow \ \ D_K.y \\
D_K.[y, z] \ \ \Rightarrow \ \ D_K.y \ \wedge \ D_K.z
\end{array}
$$

This is the definition of D_K and E_K that we will use. Terms like $\neg D_\omega.x$, for any message x, can be interpreted as that message x does not occur within any message in C, even after removing some layers of encryption; we will in particular use this where x is a key (and hence part of ω). Notice that D_K (and hence E_K) has the following monotonicity property, for any sets S and T:

$$S \ \subseteq \ T \ \ \Rightarrow \ \ D_S \ \subseteq \ D_T$$

3.3. Proving the correctness of the typical invariants

As illustrated in Sect. 3.1, we will encounter a lot of invariants that contain the set E_K. We prove their correctness in two steps: first we express E_K in terms of D_K, and afterwards we derive required assertions.

3.3.1. From invariants over E_K to invariants over D_K

Our definition of E_K is such that every expression $E_K.x$, for any set of keys K and any message x with a given bounded (typically very small) message structure, can be expressed as a bounded term over D_K. If message x is an atomic message, we have $E_K.x = D_K.x$. If message x is a tupled message $[y, z]$, for any messages y and z, we have $E_K.[y, z] = (E_K.y \wedge E_K.z) \vee D_K.[y, z]$, and we can recursively continue with the smaller messages y and z. If message x is an encrypted message $\{y\}_k$, for any key k and message y, we have $E_K.\{y\}_k = (E_K.y \ \wedge \ E_K.k) \ \vee \ D_K.\{y\}_k$, and we can recursively continue with the smaller messages k and y.

The typical required invariants that we consider are implications where expression $E_K.x$ is the antecedent. To simplify the result of replacing in this invariant the expression $E_K.x$ by a term over D_K, we may weaken the term over D_K (which, in turn, strengthens the invariant). Typically we keep only one of the terms in each conjunction. For any tuple $[y, z]$ where y is an atomic message, this often leads to a term like $D_K.y \ \vee \ D_K.[y, z]$, which, by the definition of D_K, is equivalent to the even simpler term $D_K.y$.

3.3.2. From invariants over D_K to required assertions

What typically remains is an invariant in which the antecedent is a disjunction of terms like $D_K.x$. Such an invariant is equivalent to a series (i.e., a conjunction) of invariants of the shape

$$D_K.x \ \Rightarrow \ P$$

Each of these invariants can be addressed independently, and in what follows we will assume that P does not depend on C, D, E or R.

Applying a similar approach as before, viz., expressing D_K in terms of C, would introduce an unbounded number of terms, which is infeasible. If the message types are somehow limited, for example by restricting the amount of nested encryption layers, then the number of extra terms may become bounded, but it is still likely to be unmanageable (and certainly not scalable). An alternative is to apply the normal proof rules [OG76] directly, but this may become quite complicated given the recursive definition of D_K.

Therefore we investigate a simpler and sufficient proof approach. For initialization of such an invariant, a precondition $D_K.x \ \Rightarrow \ P$ would be sufficient, but in order to abstract from K, we often introduce the stronger precondition $D_\omega.x \ \Rightarrow \ P$. What remains to be established is maintenance, for which we only need to focus on the send statements (which may expand the set C, and hence, also the set D_K).

The set D_K corresponds to the recursive decomposition of the transmitted messages in C using the additional set of keys K. Upon transmitting a new message, at least the new message must be decomposed recursively as far as the required keys are available. However, in contrast to composition, decomposition may also derive new keys, in which case some earlier transmitted messages may become further decomposable. We will treat such a compound expansion of D_K by showing that each of its expansions with a single message maintains the invariant. In what follows we discuss the new message and the earlier messages separately.

We first consider the maintenance under a (recursive) decomposition step of the new message. For each occurrence of x that can be derived from the new message using decomposition with any set of (decryption) keys L, this results in the following required pre-assertion:

$$(\forall L : l \in L : \ D_K.l \ \vee \ l \in K) \ \Rightarrow \ P$$

In particular this leads to a pre-assertion P if the new message is equal to x. As the transmitted messages have a bounded (typically very small) message structure, only bounded sets L need to be considered.

In the case that a (recursive) decomposition step of the newly transmitted message reveals any key $k : \ k \notin K$, we must also require as a pre-assertion that x cannot be derived from the earlier transmitted messages using the additional key k. This is exactly what can be expressed using our generalization of the set D with a set of keys. So in case the key k is revealed using any set of (decryption) keys L, we require (also) the following pre-assertion of this decomposition step:

$$(\forall L : l \in L : \ D_K.l \ \vee \ l \in K) \Rightarrow (D_{(K \cup \{k\})}.x \Rightarrow P)$$

Effectively this means that the key k must already be considered by the recursive decomposition of the other messages. In contrast to the conditions that prohibit certain messages to be derived, the parameter K in the generalized construct D_K indicates which keys may be derived safely.

To convert the required pre-assertions of the individual decomposition steps into the required pre-assertion of the message transmission, we must strengthen them until they are maintained under the other steps resulting from this message transmission. In practice there are two simple ways to achieve this goal: either require such an individual pre-assertion as an invariant (whose proof obligations include these), or strengthen it such that it contains no occurrences of D anymore (which makes maintenance trivial).

As a quick illustration of what happens when keys are revealed, suppose we are proving the correctness of an invariant $D.y \ \Rightarrow \ \mathit{false}$ for a message y. Suppose there is a statement that sends the message $\{y\}_k$, but it has a valid pre-assertion $D.\overline{k} \ \Rightarrow \ \mathit{false}$, so the invariant is maintained. Suppose that later on there is a statement that sends the key $\overline{k}$. The combination of these two statements violates the invariant, and we will show how this can become visible in a proof attempt. Recursive decomposition of the message $\overline{k}$, maintains the invariant, but as $\overline{k}$ is a key we require a pre-assertion $D_{\{\overline{k}\}}.y \ \Rightarrow \ \mathit{false}$. This pre-assertion is violated by the earlier mentioned statement that sends $\{y\}_k$, as the key $\overline{k}$ can be used to decompose $\{y\}_k$ into y.

3.4. Progress in the presence of intruders

Progress and termination of security protocols can mainly be hindered by the receive statements, which can be blocked. As the intruders may intercept and lose all sent messages, denial-of-service attacks cannot be prevented in our model, and hence in general progress cannot be guaranteed. In this work we will only consider progress in case the intruders behave like a proper communication channel.

As a minimum requirement to establish this goal, we will rely on the following variation on the ground rule of progress from [FvG99]: "For each receive statement in a component, it should hold that the rest of the system has the potential of ultimately sending a message that the receive statement can accept."

4. Specification of authentication

In this section we specify authentication in terms of post-conditions. Consider a set of honest agents A, such that each agent executes the authentication protocol any number of times, e.g., in succession or in parallel. We model each run of an honest agent as a (parallel) component h. The runs can be partitioned according to the agent they belong to, hence, there is a constant function z that assigns to each run h its honest agent z_h (so, $z_h \in A$); constant z_h is only accessible by run h. In what follows, a ranges over A, and h over H.

The notion of authentication that we model is agreement [Low97] between pairs of runs. To this end, we split the set of runs H into two sets I and J. For each run h we introduce a constant m_h to denote the (either honest or intruder) agent with which it decides to communicate; constant m_h is only accessible by run h. In what follows, i ranges over I, and j over J.

A protocol guarantees to any run i agreement with a run from J, if being at the post-condition of run i guarantees that if run i decided to communicate with an honest agent m_i (i.e., $m_i \in A$), then there is a run j from agent m_i (i.e., $z_j = m_i$) that decided to communicate with agent z_i (i.e., $m_j = z_i$); and dually for any run j. This would be enough in the setting from [Moo08], where each agent consists of one run. In case of multiple runs, we must require that each run j can be used for the post-condition of at most one run i, and vice versa. To this end we introduce in each run h a variable p_h that denotes the run in whose post-condition it can be used. Thus we obtain the following specification:

Run i:	$\ldots$
	$\{\mathbf{?}\ m_i \in A \ \Rightarrow\ (\exists j ::\ z_j = m_i \ \wedge\ m_j = z_i \ \wedge\ p_j = i)\}$
Run j:	$\ldots$
	$\{\mathbf{?}\ m_j \in A \ \Rightarrow\ (\exists i ::\ z_i = m_j \ \wedge\ m_i = z_j \ \wedge\ p_i = j)\}$

This specification is less operational than the ones that are collected in [Mea00]. In the CSP-based model from [Low97], for each run two events have to be introduced. For each run h our implicit initial choice of the m_h value corresponds to his "running" event with parameters z_h and m_h, while reaching the end of the protocol for any run h corresponds to his "commit" event with parameters z_h and m_h. His requirement that each "commit" event cannot occur more often than that the corresponding "running" event with swapped parameters has occurred, is expressed by our required post-conditions.

If each agent consists of at most one run, our specification can be simplified to a very compact one, see [Moo08]. In case of multiple runs, it turns out that [Low97] can formalize more compactly (but also more implicitly) that each running event can be used for at most one commit event. However, we will see that making this requirement explicit, as we did using the variables p_h, is beneficial for program derivation.

Termination of the protocol needs to be guaranteed for each pair of runs (i, j) for which $m_i = z_j \ \wedge\ m_j = z_i$ holds, provided that the intruders behave like a proper communication channel between them.

5. Derivation of an authentication protocol

To demonstrate our formal reasoning approach, we present a derivation of the authentication protocol of Needham, Schroeder and Lowe [NS78, Low96]. Our derivation is based on the specification of authentication from Sect. 4. In [Moo08] we derived this protocol for the special case in which the sets I and J are singletons, which leads to simpler formulae as there is less interference. In this section we study this protocol in its generality, i.e., for any number of concurrent runs by any number of agents. This general setting turns out to be important for identifying the nonce concept.

There exist many protocols (see, e.g., [CJ97, BM03]) that establish authentication, so it is clear that a lot of design decisions need to be made. We aim to derive some motivation for these decisions from the assertions, and to obtain the required assertions in a structured way. Generally speaking, we prefer simple message shapes and assertions. Before actually deriving a program fragment, we often start with a short exploration of the possibilities.

5.1. Last message communication

Exploration As we are heading for an asymmetric protocol, we start with only one of the two post-assertions. Regarding this choice, the usual role names, viz., initiator for any component i and responder for any component j, are not helpful. That is, they refer to the direction of the *first* communication, while our construction starts at the end of the protocol. However, since the specification is symmetric, the decision does not really matter, and we just choose to start with the post-assertion in any component j.

The queried assertion in any component j refers to constants and variables from multiple components. The way to establish such an assertion is to import it via the receipt of a message. This message should be specific to component j (being specific to m_j and z_j cannot really be enough), so the first attempt would be a message that just contains j.

This requires an invariant

$$(\forall j :: E.j \ \wedge \ m_j \in A \ \Rightarrow \ (\exists i :: z_i = m_j \ \wedge \ m_i = z_j \ \wedge \ p_i = j))$$

or equivalently, using that component identifiers are atomic messages,

$$(\forall j :: D.j \ \wedge \ m_j \in A \ \Rightarrow \ (\exists i :: z_i = m_j \ \wedge \ m_i = z_j \ \wedge \ p_i = j))$$

To initialize this invariant, we require that the j's cannot be derived initially by the intruders. Such a fresh component identifier is usually called a nonce. It uniquely identifies a component, and it is fresh in the sense that it has not been used before, i.e., initially $(\forall h :: \neg D_\omega.h)$, which avoids so-called replay attacks. Notice that this nonce concept just happens to be part of our explicit specification of authentication.

Nonces are known as an important concept for security protocols, and it is sometimes desired that the intruders cannot derive the nonce h at all in case $m_h \in A$. Thus the nonces can be used for future communications after successful execution of the authentication protocol, and for avoiding so-called known plaintext attacks [BGH+93]. This is a requirement on the authentication protocol of Needham, Schroeder and Lowe, but it is not a general requirement on nonces, e.g., in some protocols [CJ97, BM03] the nonces are transmitted in plaintext. To formally specify this requirement, we require the additional invariant

$$(\forall h :: E.h \ \Rightarrow \ m_h \notin A)$$

This invariant might be easier to interpret in contrapositive form, but we prefer this shape for homogeneity.

Derivation The additional required invariant $(\forall h :: E.h \ \Rightarrow \ m_h \notin A)$ excludes the possibility of sending a message j in case $m_j \in A$, so for progress reasons, it is not useful to include the receipt of a plain nonce j. To this end, we will use an encryption scheme in which each agent has a public and a private key. Let $k.a$ denote the public key of agent a, and let $\overline{k.a}$ denote the private key of agent a. Each component has access to the private key of its agent, and to the public key of each agent. So, all the runs of any honest agent share the same private key, but also some honest agents may have the same keys. Such an encryption scheme demands that the private keys of the honest agents remain secret, which is expressed by the invariant

$$(\forall a :: E.\overline{k.a} \ \Rightarrow \ \mathit{false})$$

For confidentiality of the nonce j, we encrypt the proposed message with a public key, e.g., the one from j's agent. After replacing the proposed message j by $\{j\}_{k.z_j}$, the required invariant becomes

$$(\forall j :: E.\{j\}_{k.z_j} \ \wedge \ m_j \in A \ \Rightarrow \ (\exists i :: z_i = m_j \ \wedge \ m_i = z_j \ \wedge \ p_i = j))$$

Thus we obtain the following intermediate program:

Run i:	$\ldots$
	$\{\textbf{?}\ m_i \in A \ \Rightarrow \ (\exists j :: z_j = m_i \ \wedge \ m_j = z_i \ \wedge \ p_j = i)\}$
Run j:	$\ldots$
	rcv $\{j\}_{k.z_j}$
	$\{m_j \in A \ \Rightarrow \ (\exists i :: z_i = m_j \ \wedge \ m_i = z_j \ \wedge \ p_i = j)\}$
Inv: **?**	$(\forall a :: E.\overline{k.a} \ \Rightarrow \ \mathit{false})$
Inv: **?**	$(\forall h :: E.h \ \Rightarrow \ m_h \notin A)$
Inv: **?**	$(\forall j :: E.\{j\}_{k.z_j} \ \wedge \ m_j \in A \ \Rightarrow \ (\exists i :: z_i = m_j \ \wedge \ m_i = z_j \ \wedge \ p_i = j))$

As keys and nonces are atomic messages, the terms $E.\overline{k.a}$ and $E.h$ can be replaced by $D.\overline{k.a}$ and $D.h$ respectively. The term $E.\{j\}_{k.z_j}$ can be rewritten to the equivalent term $D.\{j\}_{k.z_j} \vee (D.j \wedge D.(k.z_j))$. The conjunct $D.(k.z_j)$ is not useful, as the public key $k.z_j$ may be derivable. Fortunately, in case $D.j$ holds this invariant follows from the invariant on $D.h$ (previously on $E.h$). What remains is to focus on the case that $D.\{j\}_{k.z_j}$ holds, so, effectively, the term $E.\{j\}_{k.z_j}$ can also be replaced by the term $D.\{j\}_{k.z_j}$.

Initial correctness of the resulting invariants follows by requiring as precondition both the secrecy of the private keys, i.e., $(\forall a :: \neg D_\omega.\overline{k.a})$, and the freshness of the nonces, i.e., $(\forall j :: \neg D_\omega.j)$. So, as long as there are no send statements, the queried invariants are correct.

For progress reasons, we must ensure that the message to be received $\{j\}_{k.z_j}$ can actually be sent by a component i such that $m_i = z_j \wedge m_j = z_i$. Although the components cannot refer to each other's constants, z_j could be replaced by m_i. Other suggestions for replacement, in particular of j, can be found in the invariant on $D.\{j\}_{k.z_j}$, and thus we introduce in component i a send statement for a message $\{p_i\}_{k.m_i}$.

What remains is to guarantee maintenance of the invariants (especially the last two) under this send statement, including decomposition. This yields the following required pre-assertions of the send statement in component i:

- $(\forall h :: D.\overline{k.m_i} \wedge p_i = h \Rightarrow m_h \notin A)$
- $(\forall j :: k.m_i = k.z_j \wedge p_i = j \Rightarrow (m_j \in A \Rightarrow (\exists i' :: z_{i'} = m_j \wedge m_{i'} = z_j \wedge p_{i'} = j)))$

Although we could directly introduce these assertions as pre-assertions, we try to combine and simplify them by making them slightly stronger. In the second assertion, the term $k.m_i = k.z_j$ is not likely to be useful, as the agents may share public keys. Furthermore, given the antecedent $p_i = j$ there is only one obvious choice for the existential quantification over i', viz., component i. Regarding the first assertion, for $D.\overline{k.m_i}$ we can only use the first invariant (which includes $D.\overline{k.m_i} \Rightarrow m_i \notin A$), yielding the assertion $(\forall h :: p_i = h \wedge m_h \in A \Rightarrow m_i \in A)$. Using the type of z, i.e., $(\forall h :: z_h \in A)$, we combine these two assertions into one stronger assertion $(\forall h :: p_i = h \wedge m_h \in A \Rightarrow z_i = m_h \wedge m_i = z_h)$.

(This assertion can be interpreted as that if run p_i belongs to an honest agent (i.e., $p_i = h$) and decided to communicate with an honest agent (i.e., $m_h \in A$), then run p_i decided to communicate with the agent of run i (i.e., $z_i = m_h$), and run i decided to communicate with the agent of run p_i (i.e., $m_i = z_h$).)

In addition we copy the original post-assertion of component a as a pre-assertion of the send statement, since it is orthogonal to the send statement.

Pre:	$(\forall a :: \neg D_\omega.\overline{k.a}) \wedge (\forall h :: \neg D_\omega.h)$
Run i:	$\ldots$
	$\{$**?** $(\forall h :: p_i = h \wedge m_h \in A \Rightarrow z_i = m_h \wedge m_i = z_h)\}$
	$\{$**?** $m_i \in A \Rightarrow (\exists j :: z_j = m_i \wedge m_j = z_i \wedge p_j = i)\}$
	snd $\{p_i\}_{k.m_i}$
	$\{m_i \in A \Rightarrow (\exists j :: z_j = m_i \wedge m_j = z_i \wedge p_j = i)\}$
Run j:	$\ldots$
	rcv $\{j\}_{k.z_j}$
	$\{m_j \in A \Rightarrow (\exists i :: z_i = m_j \wedge m_i = z_j \wedge p_i = j)\}$
Inv:	$(\forall a :: D.\overline{k.a} \Rightarrow \mathit{false})$
Inv:	$(\forall h :: D.h \Rightarrow m_h \notin A)$
Inv:	$(\forall j :: D.\{j\}_{k.z_j} \wedge m_j \in A \Rightarrow (\exists i :: z_i = m_j \wedge m_i = z_j \wedge p_i = j))$

Notice that termination is guaranteed for any pair of components (i, j) in case $m_i = z_j \wedge p_i = j$, provided that the intruders behave like a proper communication channel between i and j.

5.2. Middle message communication

Exploration For queried assertion $m_i \in A \Rightarrow (\exists j :: z_j = m_i \wedge m_j = z_i \wedge p_j = i)$ in component i we could repeat the same ideas as for the post-condition of component j, yielding the receipt of a message $\{i\}_{k.z_i}$ in component i, and the transmission of a message $\{p_j\}_{k.m_j}$ in component j.

Before embarking on a detailed treatment of this queried assertion, we want to evaluate our choice of the receipt statement regarding the other queried assertion in component i. The most direct way to establish it would be to use the proposed receive statement for a message $\{i\}_{k.z_i}$ by requiring an invariant

$$(\forall i :: E.\{i\}_{k.z_i} \Rightarrow (\forall h :: p_i = h \wedge m_h \in A \Rightarrow z_i = m_h \wedge m_i = z_h))$$

Like before, the term $E.\{i\}_{k.z_i}$ is equivalent to the term $D.\{i\}_{k.z_i} \vee (D.(k.z_i) \wedge D.i)$, and the conjunct $D.(k.z_i)$ is not likely to be useful. However, this time the invariant on $D.i$ is not going to be very helpful in simplifying this invariant. Instead, we would need to include the nonce h in the message, although we cannot directly add h to the receive statement.

To this end we try to exploit the term $p_i = h$ by extending the receive statement with an assignment to variable p_i, and extending the message with the value p_i. Notice that some assignment to variable p_i will be needed anyhow for progress. Thus the received message becomes $\{i, p_i\}_{k.z_i}$, and we require, after exploiting the antecedent $p_i = h$, the invariant

$$(\forall i, h :: E.\{i, h\}_{k.z_i} \wedge m_h \in A \Rightarrow z_i = m_h \wedge m_i = z_h)$$

Then we use the invariant on $D.h$ to replace the term $E.\{i, h\}_{k.z_i}$ in this invariant by the term $D.\{i, h\}_{k.z_i}$.

For progress reasons, the proposed send statement in component j must be extended with a value j into $\{p_j, j\}_{k.m_j}$. For maintenance of the invariant, we should require a pre-assertion $(\forall i :: \; p_j = i \; \wedge \; m_j \in A \; \Rightarrow \; z_i = m_j \; \wedge \; m_i = z_j)$. However, this looks somewhat similar to the original post-assertion in component j, and it is the place where the original Needham–Schroeder protocol [NS78] turns out to be vulnerable [Low96] to a so-called man-in-the-middle attack in case $m_j \in A \wedge m_i \notin A$ holds (and hence $m_i \neq z_j$) and $p_j = i$ is established. Although this attack can easily be reconstructed from these formulae, we will not provide it as it is not relevant for the derivation.

To weaken this required pre-assertion, we aim to eliminate the conjunct $m_i = z_j$ by establishing it through the message communication. To this end we add the constant m_i to the receive statement in component i, yielding the receipt of a message $\{i, p_i, m_i\}_{k.z_i}$, and a required invariant

$$(\forall i, h :: \; E.\{i, h, m_i\}_{k.z_i} \; \wedge \; m_h \in A \; \Rightarrow \; z_i = m_h \; \wedge \; m_i = z_h)$$

Using the invariant on $D.h$, this is equivalent to

$$(\forall i, h :: \; D.\{i, h, m_i\}_{k.z_i} \; \wedge \; m_h \in A \; \Rightarrow \; z_i = m_h \; \wedge \; m_i = z_h)$$

For progress in case $m_i = z_j \; \wedge \; m_j = z_i$, we extend the corresponding send statement in component j with the value z_j into $\{p_j, j, z_j\}_{k.m_j}$, and hence it only requires the pre-assertion $(\forall i :: \; p_j = i \; \wedge \; z_j = m_i \Rightarrow (m_j \in A \; \Rightarrow \; z_i = m_j))$. Notice that this holds trivially for any component i such that $m_i \notin A$.

Derivation Given these considerations, we decide to introduce in component i a receive statement for a message $\{i, p_i, m_i\}_{k.z_i}$ that assigns a value to variable p_i. For the two required assertions we introduce two invariants:

- $(\forall h, i :: \; E.\{i, h, m_i\}_{k.z_i} \; \wedge \; m_h \in A \; \Rightarrow \; z_i = m_h \; \wedge \; m_i = z_h)$
- $(\forall i, x :: \; E.\{i, x, m_i\}_{k.z_i} \; \wedge \; m_i \in A \; \Rightarrow \; (\exists j :: \; z_j = m_i \; \wedge \; m_j = z_i \; \wedge \; p_j = i))$

Using the invariant on $D.h$ and $D.i$ respectively, we can replace $E.\{i, h, m_i\}_{k.z_i}$ and $E.\{i, x, m_i\}_{k.z_i}$ by $D.\{i, h, m_i\}_{k.z_i}$ and $D.\{i, x, m_i\}_{k.z_i}$ respectively. The resulting invariants can be initialized using the freshness of nonce h and i respectively. The other send statements cannot endanger these new invariants.

To ensure that the assignment to p_i within the receive statement does not endanger the invariants and assertions in other components, we require a pre-assertion $p_i \notin J$. Thus we obtain the following intermediate program:

Pre:	$(\forall a :: \; \neg D_\omega.\overline{k.a}) \; \wedge \; (\forall h :: \; \neg D_\omega.h)$
Run i:	$\ldots$
	$\{\textbf{?}\; p_i \notin J\}$
	rcv $p_i \leftarrow \{i, p_i, m_i\}_{k.z_i}$
	$;\{(\forall h :: \; p_i = h \; \wedge \; m_h \in A \; \Rightarrow \; z_i = m_h \; \wedge \; m_i = z_h)\}$
	$\{m_i \in A \; \Rightarrow \; (\exists j :: \; z_j = m_i \; \wedge \; m_j = z_i \; \wedge \; p_j = i)\}$
	snd $\{p_i\}_{k.m_i}$
	$\{m_i \in A \; \Rightarrow \; (\exists j :: \; z_j = m_i \; \wedge \; m_j = z_i \; \wedge \; p_j = i)\}$
Run j:	$\ldots$
	rcv $\{j\}_{k.z_j}$
	$\{m_j \in A \; \Rightarrow \; (\exists i :: \; z_i = m_j \; \wedge \; m_i = z_j \; \wedge \; p_i = j)\}$
Inv:	$(\forall a :: \; D.\overline{k.a} \; \Rightarrow \; \mathit{false})$
Inv:	$(\forall h :: \; D.h \; \Rightarrow \; m_h \notin A)$
Inv:	$(\forall j :: \; D.\{j\}_{k.z_j} \; \wedge \; m_j \in A \; \Rightarrow \; (\exists i :: \; z_i = m_j \; \wedge \; m_i = z_j \; \wedge \; p_i = j))$
Inv:	$(\forall i, x :: \; D.\{i, x, m_i\}_{k.z_i} \; \wedge \; m_i \in A \; \Rightarrow \; (\exists j :: \; z_j = m_i \; \wedge \; m_j = z_i \; \wedge \; p_j = i))$
Inv:	$(\forall h, i :: \; D.\{i, h, m_i\}_{k.z_i} \; \wedge \; m_h \in A \; \Rightarrow \; z_i = m_h \; \wedge \; m_i = z_h)$

For progress reasons, we introduce a corresponding send statement in component j for a message $\{p_j, j, z_j\}_{k.m_j}$. For maintenance of the last two new invariants, we require two pre-assertions:

- $(\forall i :: \; p_j = i \; \wedge \; z_j = m_i \; \Rightarrow \; m_j = z_i)$
- $(\forall i :: \; p_j = i \; \wedge \; z_j = m_i \; \wedge \; m_j \in A \; \Rightarrow \; z_i = m_j)$

For the existing invariants (in particular the one on $D.h$), we require the two pre-assertions

- $(\forall h :: \ D.\overline{k.m_j} \ \wedge \ p_j = h \ \ \Rightarrow \ \ m_h \notin A)$
- $(\forall h :: \ D.\overline{k.m_j} \ \wedge \ j = h \ \ \Rightarrow \ \ m_h \notin A)$

which follow from $(\forall h :: \ p_j = h \ \wedge \ m_h \in A \ \ \Rightarrow \ \ m_j \in A)$, using the invariant $D.\overline{k.m_j} \ \Rightarrow \ m_j \notin A$. We combine the remaining three assertions into one assertion $(\forall h :: \ p_j = h \ \wedge \ m_h \in A \ \ \Rightarrow \ \ m_j = z_h)$.

(This assertion can be interpreted as that if run p_j belongs to an honest agent (i.e., $p_j = h$) and decided to communicate with an honest agent (i.e., $m_h \in A$), then run j decided to communicate with the agent of run p_j (i.e., $m_j = z_h$).)

Pre:	$(\forall a :: \ \neg D_\omega.\overline{k.a}) \ \wedge \ (\forall h :: \ \neg D_\omega.h)$
Run i:	$\ldots$ $\{$**?** $p_i \notin J\}$ **rcv** $p_i \leftarrow \{i, p_i, m_i\}_{k.z_i}$; $\{(\forall h :: \ p_i = h \ \wedge \ m_h \in A \ \ \Rightarrow \ \ z_i = m_h \ \wedge \ m_i = z_h)\}$ $\{m_i \in A \ \Rightarrow \ (\exists j :: \ z_j = m_i \ \wedge \ m_j = z_i \ \wedge \ p_j = i)\}$ **snd** $\{p_i\}_{k.m_i}$ $\{m_i \in A \ \Rightarrow \ (\exists j :: \ z_j = m_i \ \wedge \ m_j = z_i \ \wedge \ p_j = i)\}$
Run j:	$\ldots$ $\{$**?** $(\forall h :: \ p_j = h \ \wedge \ m_h \in A \ \ \Rightarrow \ \ m_j = z_h)\}$ **snd** $\{p_j, j, z_j\}_{k.m_j}$; **rcv** $\{j\}_{k.z_j}$ $\{m_j \in A \ \Rightarrow \ (\exists i :: \ z_i = m_j \ \wedge \ m_i = z_j \ \wedge \ p_i = j)\}$
Inv:	$(\forall a :: \ D.\overline{k.a} \ \ \Rightarrow \ \ \mathit{false})$
Inv:	$(\forall h :: \ D.h \ \ \Rightarrow \ \ m_h \notin A)$
Inv:	$(\forall j :: \ D.\{j\}_{k.z_j} \ \wedge \ m_j \in A \ \ \Rightarrow \ \ (\exists i :: \ z_i = m_j \ \wedge \ m_i = z_j \ \wedge \ p_i = j))$
Inv:	$(\forall i, x :: \ D.\{i, x, m_i\}_{k.z_i} \ \wedge \ m_i \in A \ \ \Rightarrow \ \ (\exists j :: \ z_j = m_i \ \wedge \ m_j = z_i \ \wedge \ p_j = i))$
Inv:	$(\forall h, i :: \ D.\{i, h, m_i\}_{k.z_i} \ \wedge \ m_h \in A \ \ \Rightarrow \ \ z_i = m_h \ \wedge \ m_i = z_h)$

Notice that termination is guaranteed for any pair of components (i, j) in case $m_i = z_j \ \wedge \ m_j = z_i \ \wedge \ p_j = i$, provided that the intruders behave like a proper communication channel between i and j.

5.3. First message communication

Derivation The queried assertion in component j can be treated similarly to the assertion $(\forall h :: \ p_i = h \ \wedge \ m_h \in A \ \ \Rightarrow \ \ z_i = m_h \ \wedge \ m_i = z_h)$ in component i. That is, receiving a nonce into variable p_j by receiving a message $\{p_j, m_j\}_{k.z_j}$ in component j, using a required invariant

$$(\forall j, h :: \ E.\{h, m_j\}_{k.z_j} \ \wedge \ m_h \in A \ \ \Rightarrow \ \ m_j = z_h)$$

Using the invariants on $D.h$, we can replace in this invariant the term $E.\{h, m_j\}_{k.z_j}$ by the term $D.\{h, m_j\}_{k.z_j}$. The resulting invariant can be initialized using the freshness of the nonce h, and it cannot be endangered by the existing statements. For progress reasons, we introduce a send statement for a message $\{i, z_i\}_{k.m_i}$ in component i, which does not require a pre-assertion for this invariant. The new send statement also maintains the other invariants (in particular the one on $D.h$).

To ensure that the assignment to p_j within the receive statement does not endanger the invariants and assertions in other components, we require a pre-assertion $p_j \notin I$. To establish the initial correctness of this assertion and assertion $p_i \notin J$, we require them as precondition $(\forall h :: \ p_h \notin H)$ of the program.

Thus we obtain the following final program:

Pre:	$(\forall a :: \neg D_\omega.\overline{k.a}) \wedge (\forall h :: \neg D_\omega.h) \wedge (\forall h :: p_h \notin H)$
Run i:	$\{p_i \notin J\}$
	snd $\{i, z_i\}_{k.m_i}$
	; $\{p_i \notin J\}$
	rcv $p_i \leftarrow \{i, p_i, m_i\}_{k.z_i}$
	; $\{(\forall h :: p_i = h \wedge m_h \in A \Rightarrow z_i = m_h \wedge m_i = z_h)\}$
	$\{m_i \in A \Rightarrow (\exists j :: z_j = m_i \wedge m_j = z_i \wedge p_j = i)\}$
	snd $\{p_i\}_{k.m_i}$
	$\{m_i \in A \Rightarrow (\exists j :: z_j = m_i \wedge m_j = z_i \wedge p_j = i)\}$
Run j:	$\{p_j \notin I\}$
	rcv $p_j \leftarrow \{p_j, m_j\}_{k.z_j}$
	; $\{(\forall h :: p_j = h \wedge m_h \in A \Rightarrow m_j = z_h)\}$
	snd $\{p_j, j, z_j\}_{k.m_j}$
	; **rcv** $\{j\}_{k.z_j}$
	$\{m_j \in A \Rightarrow (\exists i :: z_i = m_j \wedge m_i = z_j \wedge p_i = j)\}$
Inv:	$(\forall a :: D.\overline{k.a} \Rightarrow \mathit{false})$
Inv:	$(\forall h :: D.h \Rightarrow m_h \notin A)$
Inv:	$(\forall j :: D.\{j\}_{k.z_j} \wedge m_j \in A \Rightarrow (\exists i :: z_i = m_j \wedge m_i = z_j \wedge p_i = j))$
Inv:	$(\forall i, x :: D.\{i, x, m_i\}_{k.z_i} \wedge m_i \in A \Rightarrow (\exists j :: z_j = m_i \wedge m_j = z_i \wedge p_j = i))$
Inv:	$(\forall h, i :: D.\{i, h, m_i\}_{k.z_i} \wedge m_h \in A \Rightarrow z_i = m_h \wedge m_i = z_h)$
Inv:	$(\forall h, j :: D.\{h, m_j\}_{k.z_j} \wedge m_h \in A \Rightarrow m_j = z_h)$

Notice that termination is guaranteed for any pair of components (i, j) in case $m_i = z_j \wedge m_j = z_i$, provided that the intruders behave like a proper communication channel between i and j. Since there are no more queried assertions or invariants, this is the end of our derivation.

The precondition $(\forall h :: p_h \notin H)$ may be unexpected. As each local variable p_h is explicitly assigned a value before it is inspected, omitting this precondition would not change the behavior of the protocol. So, we have only introduced this precondition to simplify the annotation.

5.4. Agreement on the data items

It may also be desired [Low97] that the components agree on the specific data items that are used in the protocol, viz., the nonces. This program dependent part of the specification can be modeled by adding within the existential quantification of the post-conditions a conjunct $p_i = j$ and $p_j = i$ respectively. We will not discuss this property in any detail, since it only extends the current annotation, and in particular it does not drive the derivation.

5.5. Afterthoughts

Comparing this derivation to the one in [Moo08], we notice that the resulting annotations are different, which is logical as the original specifications are different. Apart from this, they are quite similar, though organized in a slightly different way. Although in the special case from [Moo08] the assertions remain simpler, in our more general setting we can identify the nonces in a nicer way. Both annotations show that the agent identifiers in the messages are *not* intended to be able to send a reply to the right agent. The broadcast model clearly points out that there are more important reasons, viz., in the first communication it is to avoid a dual of the man-in-the-middle attack from [Low96] in case $m_i \in A \wedge m_j \notin A$.

It is useful [AN96] to investigate which properties we have used about the nonces and keys. We have seen that nonces are identifiers of the runs that are initially unknown to the intruders. Viewing nonces as identifiers is simpler than the usual formalization that there is a bijection between the nonces and the run identifiers (which are typically not used for anything else). Notice that this view gives no problem [AG99] in expressing that "the intruders can invent nonces, but they are not lucky enough to guess the random nonces on which the protocol depends".

Regarding the keys, we have left the exact knowledge of the intruders unspecified; the only requirement is that initially they do not know the private keys of the honest agents. Moreover, we have not assumed that the number of intruders is limited, nor that the private keys (nor the public keys) of the honest agents are different. Hence, the encryption scheme is just used as a smoke screen against the intruders.

Finally it may seem to be more realistic to model each constant m_h as a variable that is explicitly assigned a value by component h. A disadvantage of such a more-dynamic model is the additional amount of proof obligations. In particular, the correctness proof would demand some additional assertions to address the cases in which these variables have not yet been initialized. In our opinion this would primarily increase the amount of work and formulae, while adding nothing to the understanding of the essence of this protocol.

6. Public-key distribution protocol

In this section we derive a simple public-key distribution protocol, and integrate it with the authentication protocol from Sect. 5. In contrast to Sect. 5, where we have been very elaborate, this derivation will be more compact. In comparison to [Moo08], we again consider agents that run the protocol multiple times.

6.1. Specification

Such a public-key distribution protocol is used in case an agent has not yet access to a certain public key, but it has access to the public key of a special key-server agent. To be more precise, we assume that the set of honest agents A contains a single key-server agent s with a set of runs T; in what follows t ranges over T. The required protocol enables any run h to obtain in a variable u_h the public key $k.m_h$.

As the public keys are not required to be secret, the main security aspect is guaranteeing that the received key is indeed the expected key. For this protocol we assume that each private key is different from each public key, and that the public keys and the private keys are each other's inverses, i.e., for any key x, $\overline{\overline{x}} = x$.

Thus a protocol between a run h (left column) and a run t (right column) that copies $k.m_h$ into variable u_h can be specified as follows:

Pre: $(\forall a :: \neg D_\omega.\overline{k.a})$	
Run h:	Run t:
...	...
$\{?\ u_h = k.m_h\}$	
Inv: $(\forall a :: D.\overline{k.a} \ \Rightarrow \ \mathit{false})$	

Termination of the protocol only needs to be guaranteed if the intruders behave like a proper communication channel between the components h and t.

6.2. Derivation

The queried assertion $u_h = k.m_h$ in component h can be imported using a receive statement that assigns a value to variable u_h. An obvious candidate message is a tuple of the key u_h and the agent m_h, for which we should require an invariant

$$(\forall h, x :: E.[x, m_h] \ \Rightarrow \ x = k.m_h)$$

As the term $E.[x, m_h]$ is equivalent to $(D.x \ \wedge \ D.m_h) \ \vee \ D.[x, m_h]$, we need to require (among others) the following invariant:

$$(\forall h, x :: D.x \ \wedge \ D.m_h \ \Rightarrow \ x = k.m_h)$$

In general, this invariant cannot be initialized, as various public keys and agent identifiers may be known to the intruders.

To keep the key and the agent identifier together, we authenticate the proposed message by encryption with the private key of the key server. Then we can replace the required invariant by

$$(\forall x :: E.\{x, m_h\}_{\overline{k.s}} \ \Rightarrow \ x = k.m_h)$$

Using the invariant on $D.\overline{k.s}$ it is equivalent to

$$(\forall h, x :: D.\{x, m_h\}_{\overline{k.s}} \Rightarrow x = k.m_h)$$

This invariant can usually be initialized (except in models like [DMS81]), and so far its maintenance is also guaranteed. Thus we obtain the following intermediate program:

Pre: $(\forall a :: \neg D_\omega.\overline{k.a}) \wedge (\forall h, x :: D_\omega.\{x, m_h\}_{\overline{k.s}} \Rightarrow x = k.m_h)$	
Run h: ...	Run t: ...
rcv $u_h \leftarrow \{u_h, m_h\}_{\overline{k.s}}$	
$\{u = k.m_h\}$	
Inv: $(\forall a :: D.\overline{k.a} \Rightarrow \mathit{false})$	
Inv: $(\forall h, x :: D.\{x, m_h\}_{\overline{k.s}} \Rightarrow x = k.m_h)$	

For progress reasons, we must introduce in component t a send statement. Considering the invariant on $D.\{x, m_h\}_{\overline{k.s}}$, the statement should send a message $\{k.v_t, v_t\}_{\overline{k.s}}$, where v_t denotes a variable. As the public key $k.s$ may be derivable, the transmitted key $k.v_t$ may be revealed by the intruders via decomposition. Regarding this new message, the invariants are maintained using that the private and public keys are different. Regarding any earlier transmitted messages, we strengthen the invariants using our generalization and the set $(a :: k.a)$ of all public keys. As we have already anticipated (informally) that the public keys may become known, these changes turns out to maintain the correctness of the invariants, thus yielding the following intermediate program:

Pre: $(\forall a :: \neg D_\omega.\overline{k.a}) \wedge (\forall h, x :: D_\omega.\{x, m_h\}_{\overline{k.s}} \Rightarrow x = k.m_h)$	
Run h: ...	Run t: ...
rcv $u_h \leftarrow \{u_h, m_h\}_{\overline{k.s}}$	**snd** $\{k.v_t, v_t\}_{\overline{k.s}}$
$\{u_h = k.m_a\}$	
Inv: $(\forall a :: D_{(a::\, k.a)}.\overline{k.a} \Rightarrow \mathit{false})$	
Inv: $(\forall h, x :: D_{(a::\, k.a)}.\{x, m_h\}_{\overline{k.s}} \Rightarrow x = k.m_h)$	

There are no more queried assertions or invariants, but for progress reasons we must still ensure that variable v_t in component t can be set to m_h. To this end we can safely introduce a message communication, without any invariants or security concerns. Thus we obtain the following final program:

Pre: $(\forall a :: \neg D_\omega.\overline{k.a}) \wedge (\forall h, x :: D_\omega.\{x, m_h\}_{\overline{k.s}} \Rightarrow x = k.m_h)$	
Run h: **snd** m_h	Run t: **rcv** $v_t \leftarrow v_t$
; **rcv** $u_h \leftarrow \{u, m_h\}_{\overline{k.s}}$	; **snd** $\{k.v_t, v_t\}_{\overline{k.s}}$
$\{u_h = k.m_h\}$	
Inv: $(\forall a :: D_{(a::\, k.a)}.\overline{k.a} \Rightarrow \mathit{false})$	
Inv: $(\forall h, x :: D_{(a::\, k.a)}.\{x, m_h\}_{\overline{k.s}} \Rightarrow x = k.m_h)$	

Notice that termination is guaranteed if the intruders behave like a proper communication channel between the components h and t. Since there are no more queried assertions, this is the end of our derivation.

Notice that the first (plaintext) message does not contain the value z_h, which might be useful for sending the reply to the right component. However, as we only consider broadcasts, this is not needed in our model.

6.3. Protocol integration

For reasoning about the concatenation of the public-key distribution protocol with the authentication protocol, the occurrences of D_K, for any K, in the invariants must first be made homogeneous. In this case we strengthen the invariants for the authentication protocol by replacing each occurrence of D_K by $D_{(a::\, k.a)}$. Assuming that the public and private keys are different, none of the transmitted messages in the authentication protocol uses encryption with any key from $(a :: \overline{k.a})$, and hence these stronger invariants are also maintained. The required precondition might become stronger, but in this case it already depends on the even stronger D_ω.

After this strengthening of the invariants, the annotated programs can easily be concatenated and all invariants turn out to be maintained under the additional statements. This example looks promising with respect to the important field of protocol composition; see also for example [DDMP05].

7. Conclusions and further work

After this exploration of the derivation of security protocols, we must assess what we have achieved. We have first developed specifications of secrecy and authentication, in a familiar style of pre- and post-conditions and invariants. For dealing with the interference caused by intruders, we have slightly modified an existing approach to deal with traditional message communication. The resulting method benefits from such a well-understood basis, and it is flexible enough to address patterns for both public-key confidentiality and private-key authentication. Moreover, the annotation developed along the derivation contains enough details about the protocol to assist in reasoning about the composition of security protocols.

As an example, we studied the authentication protocol of Needham, Schroeder and Lowe, in which the well-known attack on the original protocol was avoided in a natural way. By scaling up to parameterized specifications and protocols, in which the agents execute several protocol runs in parallel, the notion of a nonce was identified nicely, as it is basically a fresh run identifier. It turns out that the additional interference in such parameterized protocols has a limited impact on the required annotation.

The emphasis has been on keeping the style of reasoning manageable, such that it can also be applied manually. The resulting annotated programs are created in a demand-driven way, and their correctness can easily be checked. Their annotation records the design decisions, and it provides a safe basis for reasoning about the messages. A continued attention to notational concerns is an important piece of further work for effectively deriving and reasoning about protocols.

In contrast to automated approaches, we have already noticed that afterwards experimenting with minor variations of the protocols is a bit tricky, in which case the annotation should be checked carefully. In this sense a useful piece of further work is to develop an easy way to automatically verify the developed annotation using theorem provers, e.g., along the lines of [MW05, Moo06, RWM07]. The main challenge is likely to be guiding the prover through the recursive definition of the set of derivable messages. Nevertheless we want to stress the importance of keeping the derivation techniques manageable, in order to avoid the introduction of errors, and to emphasize the design decisions.

For further work we want to develop some heuristics to better predict the effects of the design decisions, and hence simplify the derivations. It is also further work to specify other security properties in this style. In particular, our specification of authentication should be extended to cover other notions of authentication [Low97], and multi-party synchronization [CM06]. Moreover, we want to further explore the degrees of freedom in the presented derivation. We are also interested in studying the effect of this approach on other security protocols, e.g., from [CJ97, BM03], which may yield new proofs, protocols and taxonomies. Finally, we would like to study different intruder and encryption models, such that the derivations can highlight where the specific assumptions are used.

Acknowledgements

The author thanks the anonymous reviewers for their insightful comments.

References

[AFS97] Alves-Foss J, Soule T (1997) A weakest precondition calculus for analysis of cryptographic protocols. In: Proceedings of the DIMACS workshop on design and formal verification of security protocols

[AG99] Abadi M, Gordon AD (1999) A calculus for cryptographic protocols: The spi calculus. Inform Comput 148(1):1–70

[AN96] Abadi M, Needham R (1996) Prudent engineering practice for cryptographic protocols. IEEE Trans Softw Eng 22(1):6–15

[BAN90] Burrows M, Abadi M, Needham R (1990) A logic of authentication. ACM Trans Comput Syst 8:18–36

[BGH+93] Bird R, Gopal I, Herzberg A, Janson P, Kutten S, Molva R, Yung M (1993) Systematic design of a family of attack-resistant authentication protocols. IEEE J Sel Areas Commun 11(5):679–693

[BM03] Boyd C, Mathuria A (2003) Protocols for authentication and key establishment. Springer, Berlin

[But99] Buttyán L (1999) Formal methods in the design of cryptographic protocols (state of the art). Technical report SCC/1999/38, Swiss Federal Institute of Technology (EPFL)

[But02] Butler MJ (2002) On the use of data refinement in the development of secure communication systems. Formal Aspects Comput 14(1):2–34
[CJ97] Clark J, Jacob J (1997) A survey of authentication protocol literature. Technical report, Department of Computer Science, University of York
[CJM98] Clarke EM, Jha S, Marrero WR (1998) Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In: Gries D, de Roever WP (eds) Proceedings of the IFIP working conference on programming concepts and methods. Chapman & Hall, London, pp 87–106
[CM06] Cremers CJF, Mauw S (2006) Generalizing Needham–Schroeder–Lowe for multi-party authentication. Computer Science Report 06-04, Technische Universiteit Eindhoven
[DDMP05] Datta A, Derek A, Mitchell JC, Pavlovic D (2005) A derivation system and compositional logic for security protocols. J Comput Secur 13:423–482
[DG06] Dongol B, Goldson D (2006) Extending the theory of Owicki and Gries with a logic of progress. Log Methods Comput Sci 2(1):1–25
[Dij76] Dijkstra EW (1976) A discipline of programming. Prentice-Hall, Englewood Cliffs
[DM06] Dongol B, Mooij AJ (2006) Progress in deriving concurrent programs: emphasizing the role of stable guards. In: Uustalu T (ed) Proceedings of the conference on mathematics of program construction. Lecture notes in computer science, vol 4014. Springer, Berlin, pp 140–161
[DM08] Dongol B, Mooij AJ (2008) Streamlining progress-based derivations of concurrent programs. Formal Aspects Comput 20(2):141–160
[DMS81] Denning DO, Maria Sacco G (1981) Timestamps in key distribution protocols. Commun ACM 24(8):533–536
[DY83] Dolev D, Yao AC (1983) On the security of public key protocols. IEEE Trans Inform Theory 29(12):198–208
[Fid01] Fidge CJ (2001) A survey of verification techniques for security protocols. Technical Report 01-22, Software Verification Research Centre, The University of Queensland
[FvG99] Feijen WHJ, van Gasteren AJM (1999) On a method of multiprogramming. Springer, Berlin
[FvGS98] Feijen WHJ, van Gasteren AJM, Schieder B (1998) An elementary derivation of the alternating bit protocol. In: Jeuring J (ed) Proceedings of the conference on mathematics of program construction. Lecture notes in computer science, vol 1422. Springer, Berlin, pp 175–187
[Hoo06] Hoogerwoord RR (2006) A formal derivation of a sliding window protocol. Computer Science Report 06-31, Technische Universiteit Eindhoven
[Low96] Lowe G (1996) Breaking and fixing the Needham–Schroeder public-key protocol using FDR. In: Margaria T, Steffen B (eds) Proceedings of the conference on tools and algorithms for the construction and analysis of systems. Lecture notes in computer science, vol 1055. Springer, Berlin, pp 147–166
[Low97] Lowe G (1997) A hierarchy of authentication specifications. In: Proceedings of the computer security foundations workshop. IEEE Computer Society, New York, pp 31–44
[Mea00] Meadows C (2000) Invariant generation techniques in cryptographic protocol analysis. In: Proceedings of the computer security foundations workshop. IEEE Computer Society, New York, pp 159–167
[Moo06] Mooij AJ (2006) Constructive formal methods and protocol standardization. Ph.D. thesis, Technische Universiteit Eindhoven
[Moo08] Mooij AJ (2008) Constructing and reasoning about security protocols using invariants. In: Boiten E, Derrick J, Smith G (eds) Proceedings of the international refinement workshop (REFINE 2007). ENTCS, vol 201. Elsevier, Amsterdam, pp 99–126
[MW05] Mooij AJ, Wesselink JW (2005) Incremental verification of Owicki/Gries proof outlines using PVS. In: Lau K-K, Banach R (eds) Proceedings of the conference on formal engineering methods. Lecture notes in computer science, vol 3785. Springer, Berlin, pp 390–404
[NS78] Needham R, Schroeder M (1978) Using encryption for authentication in large networks of computers. Commun ACM 21(12):993–999
[OG76] Owicki S, Gries D (1976) An axiomatic proof technique for parallel programs I. Acta Inform 6:319–340
[Pau98] Paulson LC (1998) The inductive approach to verifying cryptographic protocols. J Comput Secur 6(1–2):85–128
[PS00] Perrig A, Song D (2000) A first step towards the automatic generation of security protocols. In: Proceedings of the network and distributed system security symposium. The Internet Society
[RWM07] Romijn JMT, Wesselink JW, Mooij AJ (2007) Assertion-based proof checking of Chang–Roberts leader election in PVS. In: Namjoshi KS (ed) Proceedings of the international symposium on automated technology for verification and analysis. Lecture notes in computer science, vol 4762. Springer, Berlin, pp 347–361
[RZ97] Ryan P, Zakiuddin I (1997) Modelling and analysis of security protocols. In: Proceedings of the DIMACS workshop on design and formal verification of security protocols
[Sch97] Schneider S (1997) Verifying authentication protocols with CSP. In: Proceedings of the computer security foundations workshop. IEEE Computer Society, New York, pp 3–17
[TFHG99] Thayer Fábrega FJ, Herzog JC, Guttman JD (1999) Strand spaces: proving security protocols correct. J Comput Secur 7(1):191–230

Received 30 November 2007
Accepted in revised form 20 December 2008 by E.A. Boiten, M.J. Butler, J. Derrick and G. Smith
Published online 4 February 2009