http://www.omnys.it/degas

DEGAS IST-2001-32072

Design Environments for Global ApplicationS

Consortium: UNITN (I) University of Trento (Coordinator), IMM (DK) Institute of Mathematical Modelling - Technical University of Denmark, DIPISA (I) Dipartimento di Informatica - Università di Pisa, UEDIN (UK) University of Edinburgh - MTCI (I) Motorola Technology Center Italy - OMNYS (I) Omnys Wireless Technology

For LySa User's Guide

Editor(s): C. Montangero DIPISA
Author(s): Carlo Montangero
Participant(s): DIPISA
Work package: WP3
Document number: WP3- UNIPI - I02 - Int - 001
Security: Pub **Nature**: R **Version**: 1.0 **Pages**: 25

Abstract

This report describes how to build protocols models in UML, so that they can be analized for authentication properties in the DEGAS framework. To this purpose, we introduce a couple of reusable UML packages, which should be used in Poseidon, to produce protocol models amenable to analysis in the DEGAS Choreographer.

Keywords: Authentication, UML.

18/2/05

For LySa User's Guide

Carlo Montangero

Dipartimento di Informatica
Università di Pisa

Table of Contents

Table of Figures

Table of narrations

Introduction

This report describes how to build protocols models in UML, so that they can be analized for authentication properties in the DEGAS framework. To this purpose, we introduce a couple of reusable UML packages, which should be used in Poseidon, to make protocol models amenable to analysis in the DEGAS Choreographer [Chor]. In other terms, the report specifies how to generate models that the extractor for LySa will accept, and that the reflector will be able to use to produce feedback [Del20]. In the first package, ForLysa, we model general concepts like principals, keys, messages, etc.; the actual protocol model is derived extending the second package, Protocol. They are available as a Poseidon .zuml file [ForLysa]. The model includes the standard elements that are used in the analysis performed by the lysatool in Choreographer, according to the requirements in [MB04].

The For Lysa package

As shown in Figure 1, the classes and associations that define the verification scenario assumed by the extractor according to the requirements in [MB04], come in two packages. Some classes in Protocol inherits from ForLysa, since they specialize the general concepts to the protocol at hand.

The contents of package ForLysa is shown in Figure 2 and in Figure 3 with respect to the general aspects and the principals, and to the details of the messages, respectively.

Three party protocols are considered, with one kind of principal being the *initiator* of the protocol, while another kind is the *responder*. Additionally, there is a *server*, sometimes referred to as a trusted third party, a key distribution centre, a certificate authority, etc. In a protocol specification there is only one initiator, responder and server. The principals communicate and exchange *messages* as indicated by the *communicates* association. To express communication, the operation msg() can be invoked on the principal that the message is *sentTo*. The specification of this operation is that it copies its argument into variable *in* of the receiver. For uniformity, and to ease the extraction process, we expect that the value of variable *out* is passed to msg(). In summary, whenever a principal contributes to a step of the protocol, it needs to keep track of two messages: an incoming message that is left by msg() in its *in* variable, and triggers the contribution, and the outgoing message that it builds in its *out* variable and then sends.

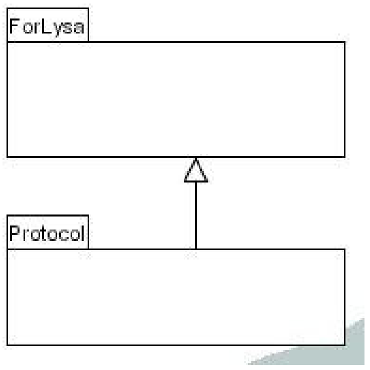

Figure 1: General structure of ForLysa.

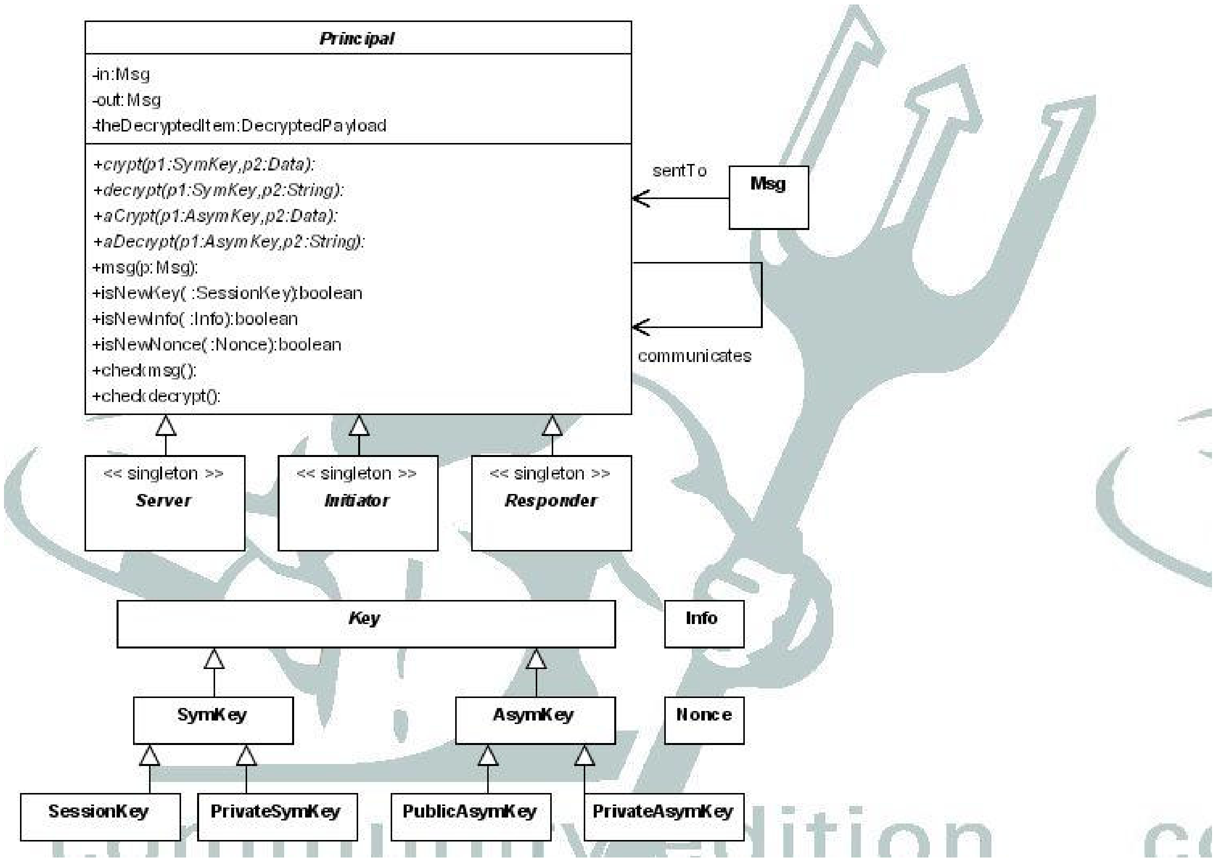
Principal
-in:Msg
-out:Msg
-theDecryptedItem:DecryptedPayload
+crypt(p1:SymKey,p2:Data):
+decrypt(p1:SymKey,p2:String):
+aCrypt(p1:AsymKey,p2:Data):
+aDecrypt(p1:AsymKey,p2:String):
+msg(p:Msg):
+isNewKey(:SessionKey):boolean
+isNewInfo(:Info):boolean
+isNewNonce(:Nonce):boolean
+checkmsg():
+checkdecrypt():
sentTo
Msg
communicates
<< singleton >>
Server
<< singleton >>
Initiator
<< singleton >>
Responder
Key
Info
SymKey
AsymKey
Nonce
SessionKey
PrivateSymKey
PublicAsymKey
PrivateAsymKey

Figure 2: The Class diagram of the principals.

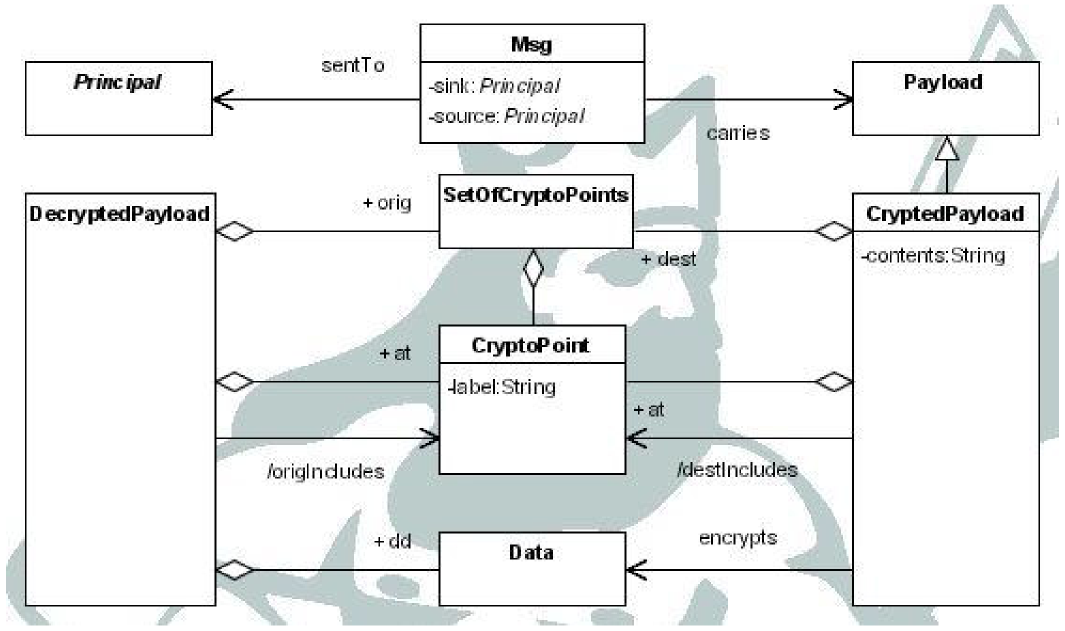

Figure 3: The Class diagram of the messages.

There are two different kind of admissible cyptography: symmetric, which uses either *private* or *session keys*, and asymmetric, which uses *public/private key* pairs. There are some generic operations that can be used to build and open the messages. They are left abstract, since we leave the choice of the cryptographic algorithms open to further specialisations. In fact, the analysis treats encryption as abstract operations so we would not get more precise analysis results by specialising these operations further. Encryption and decryption are performed by *crypt()* and *decrypt()* for the symmetric case, and *aCrypt()* and *aDecrypt()* for the asymmetric case. We assume that decrypt() and aDecrypt() leave their result in variable *theDecryptedItem*. The type of this variable, and those of the parameters will be discussed below, when considering the structure of the messages in detail. These operations must be immediately followed by a *checkdecrypt()* operation, as discussed below.

Class *Info* represent what is being communicated, and is no further specified here, and *Nonce* represents numbers used only once, which are a standard tool when specifying protocols.

As shown in Figure 3, messages carry *Payloads*, some of which can be *CryptedPayloads*, which encrypt *Data* in their *contents*. The CryptedPayloads are returned by the encryption operations that we introduced above. Dually, the decryption operations leave in the local variable theDecryptedItem, values of type *DecryptedData*, which contain the information in clear in the field *dd*.

To support the analysis, the result of each encryption/decryption operation may be decoratated with labels, called *Cryptopoints*, that identify uniquely the point in the protocol where the operation is performed (the *at* field of the payloads) and to specify the points in which the payload was originated (*orig*, in case of a decryption) or is intented to be used (*dest*, in case of an encryption).

Moreover, always to support the analysis, as explained in [LySa], each message, is decorated with a *source* field, that will contain the name of the sender of the message, and a *sink* field, that should contain the name of the recipient.

Operations *checkmsg()* and *checkdecrypt()* are used to open and check the messages and the decrypted payloads, respectively. The specific semantics of each use of these operations will be specified in the protocol model, using invariant and postcondition constraints. The details are given below, when discussing the specification language.

Finally, there are no standard operations to build the outgoing messages. These operations should be introduced explicitely, albeit named in standardised ways that we discuss in the next section, and specified by pre- and post-conditons. They can use the parts of the incoming messages, saved in local variables by previous checkmsg/checkdecrypt operations, as well as specific information held by the principal in private attributes. Since it often happens, when specifying a protocol, that one needs a "fresh" value of some kind, the Principal specification introduces three "isNew" operations, to express the need of a correct initialization of some variable. These predicates can be used in the preconditon of the message building operations.

The Protocol package

To specify a protocol in our approach, one needs to specify subtypes of the initiator, responder and server that introduce local variable to held parts of the incoming messages and other specific information, and introduce specific operations to build the outgoing messages. Then one needs to build a sequence diagram, that represents the interaction between the involved principals, and holds, as constraints on the links, the specification of each operation.

The Protocol package contains the essential starting point for this activity, in a way that reflects the current verification scenario, as shown in Figure 4, Figure 5, and Figure 6. The names for initiator, server and responder come from the tradition in informal protocol narrations.

In the scenario, each principal shares a unique *key* for symmetric key encryption with the server. These *private keys* are known as *kA* and *kB* for the initiator and the responder, respectively. The server knowledge is represented by the function *key()*.

Furthermore, an asymmetric key pair is assigned to each principal. These keys are known as *kAp*, *kAm*, *kBp*, *kBm*, and *kSp*, *kSm* for the initiator, the responder, and the server, respectively. The ending characters of these names are chosen to remind of the *plus* and *minus* signs that are used in UML to denote public and private attributes/operations, respectively. Indeed, the negative keys in the key pairs are kept secret by the principal they belong to, while the positive key are known to all other principals. This is in agreement with the UML visibility rules.

The remaining attributes are not standards but are simply examples of what a protocol designer may introduce. The choice given here is compatible with a scenario in which the initiator encrypts and sends the information for a certificate to the server, which accepts them in certAinS, and then returns the Certificate (i.e. the received information to A, encrypted wih its private asymmetric key, kSm) to A, which stores it in its variable certA, for future use. It is also expected that the initiator will use a Nonce, rA, and a piece of information m. Similarly, the responder will have its own Certificate and Nonce. The typical (once more) structure of the involved data is given in Figure 5. This figure shows also the structure of a tpyical message, *Msg1A*, used by the initiator in the scenario above to send the information needed for a certificate, and the naming conventions for its payloads.

Finally, since it has the burden to start the protocol, the initiator shows a typical message building operation, *premsg1A()*. Here we are following a requirement, i.e. that the building operations have names that start by "premsg", and showing a standard way introducing names, i.e. numbering the operations in the order they occur, and tagging them with the principal they pertain. The semantics of this operation, as well as o fthe other, as already mentioned, is given by pre/post conditons and invariant constraints in the sequence diagram, using a specification language defined in the next section.

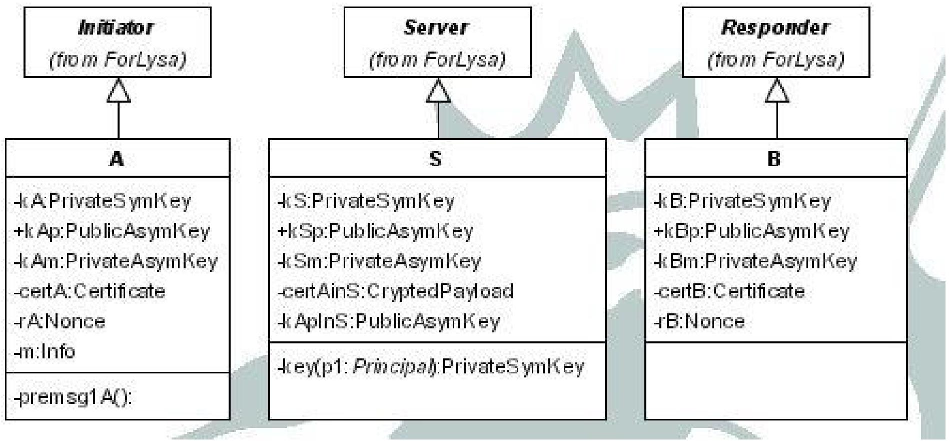

Figure 4: The principals in the verification scenario.

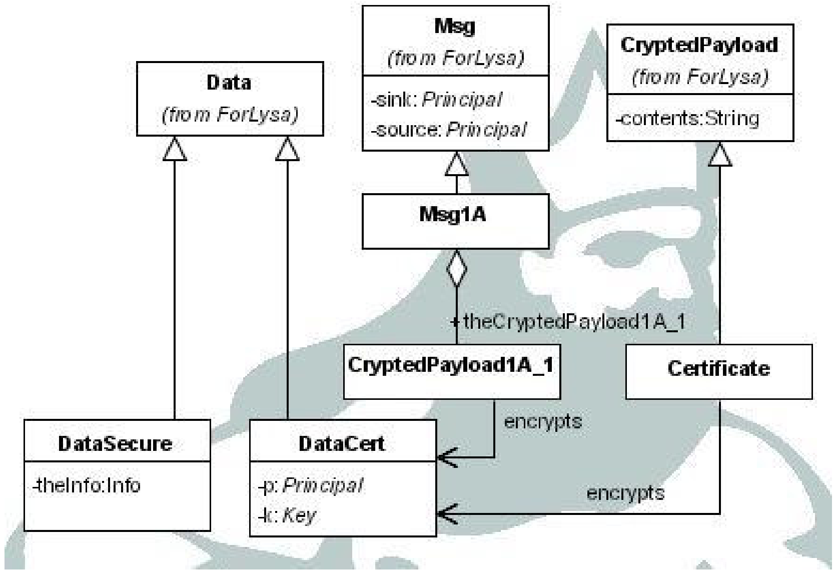

Figure 5 Typical data structures.

Finally, Figure 6 shows, as the starting point of the sequence diagram, a typical protocol step. The initiator builds the first message, and the encryption is marked as occurring at cryptopoint Acp1, then sends it to the server, which inputs it with checkmsg, and opens it with aDecrypt and checkdecrypt. The diagram shows both the conventional name of the cryptopoints, and the way we represent in the diagram the association between the operations and the labels, i.e. by placing the latter in a comment linked to the former. Besides, the diagram shows the conventional names of the Principal objects partecipating in the protocol, namely i,j and s for the initiator, responder and server, respectively.

To continue the protocol specification the designer shall introduce new messages, and complete each one with pre/post/invariant conditions. The specification of the messages introduced here will be discsussed after introducing the specification language in the next section.

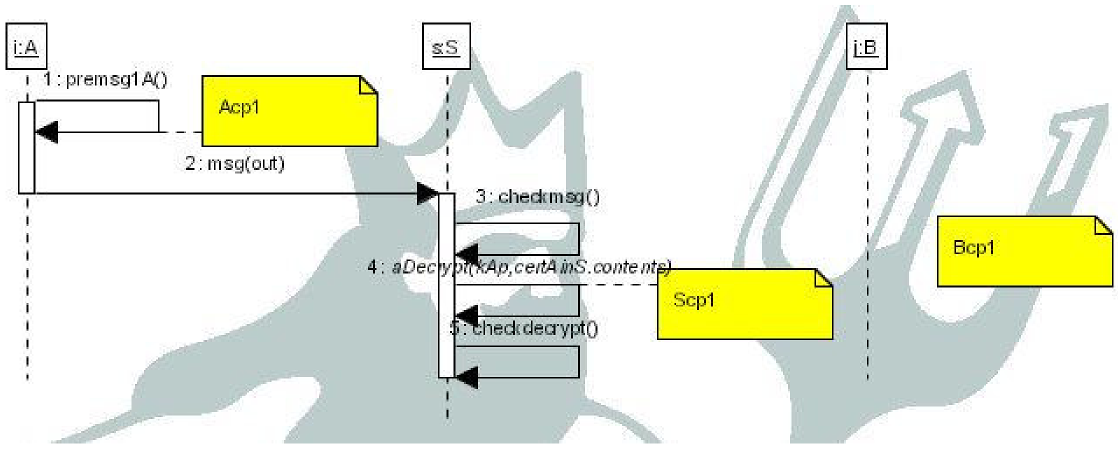

Figure 6: A typical protocol step.

The specification language

Syntax

Meta-conventions: Square brackets denote optional items, curly brackets zero or more occurrences.

```
Spec ::= Inv | Pre | Post | Comment
Inv  ::= Cond { , Cond }
Pre  ::= TypeRestriction | EncryptedType |
            Initialization { & Initialization }
TypeRestriction ::= Ide : Type
EncryptedType :: Ide encrypts DataConstructor
Initialization ::= isNewKey( Ide ) | isNewInfo( Ide ) | isNewNonce( Ide )
Post ::= [ with TypeRestriction ] Cond { & Cond }
Cond ::= Name = Expr
Name ::= Ide { . Ide }
Expr ::= Name | Fun( [ Expr { , Expr } ] )
Fun  ::= crypt | aCrypt | cp | Constructor
Ide  ::= <any name defined in the namespace of the destination>
Constructor ::= SetofCryptpoints | DataConstructor
DataConstructor ::= <any data constructor>
Comment ::= $$ <string> $$
```

Semantics

Spec

In the following we will call *destination* the object which is the destination of the stimulus the specified operation is associated with. Each *specification* is evaluated in a namespace that merges the names of the parameters of the operation with the namespace of the sequence diagram, which contains the object names (i,j, and s) and of the destination. If the name denotes an object, its fields are also accessible, and so on recursively, in the usual fashion. Besides, the evaluation namespace includes all the standard public keys kXp. When the destination is the server, the namespace includes also all the private symmetric keys kX, since the server knows them all, in our scenario[1]. The designer is responsible for the absence of clashes: the naming conventions should make this task easy. At the moment being, there is no guarantee that all errors with respect to naming will be statically caught.

Inv

An *invariant* can be associated to checkmsg and checkdecrypt, to specify checks on an incoming message, or on the result of a decryption, respectively. The purpose of an invariant is the same as a UML condition on a message: it blocks the protocol if not satisfied. Apparently, Poseidon does not provide for explicit conditions on messages in sequence diagrams yet, so we went for this turn-around. The only constraint that can be imposed in an invariant is that the values of the two sides of the conditions are equal. Usually, the source and sink of each message are checked against the expected value. Additional checks depends on the specifics of the protocol, like that a clear payload must be equal to the name of the message source, i.e. that the sender can only speak for itself. Similarly, another check may be that the incoming responder is indeed the intended one. These checks make the protocol more robust, blocking malicious attacks at run-time.

Pre

A *pre-condition* must be associated to each call of the operation msg(), to specify the actual type of the current message, by a *type restriction*. Similarly, a pre-condition must be associated to each decrypt/aDecrypt operation, to specify the type of the encrypted data, by an *encrypted data* declaration. Finally, preconditions allow the designer to express initializations concisely in premsg (see below).

[1] This convention on the namespaces makes the operation key() of the server superfluous, and we will not use it.

Initialization

The intended meaning is that before the specified operation, the arguments of these predicates have been assigned fresh values. Initializations can be used to this purpose, as the preconditon of a premsg operation.
With respect to the analysis, the lack of proper initalization will likely lead to more errors revealed, since it amounts to leave some values unprotected.
The choice of operators in the initialization clause reflects a couple of assumptions, with respect to keys:

- private keys, either symmetric or asymmetric, can be freely used in the operations of the owner, since they are assumed to be initialised before the protocol starts;
- asymmetric public keys can be used everywhere, for their own nature; however, some protocols may restrict themeselve to use public keys that have been explicitly exchanged;
- session keys must be initialised explicitly before they are used.

Similarly, nonces and any information that is generated and exchanged during the execution of the protocol, must be initialized before their use.

Post

The *post-condition* clause is used to specify the effect of an operation on the state of the destination.
The post-condition of operation msg is standard, namely 'in = p', meaning that the sent message (the value of the parameter p, which is assigned the value of the 'out' attribute of the sender) is copied into the 'in' attribute of the destination.
In the case of premsg we usually use the optional element, which is a record scope opener à la Pascal, for readability: the fields need not be prefixed by the identifier, in the left-hand sides of the conditions that follow. The restriction is 'out : MsgM' if the operation is premsgM[2]. This entails that the names of the Payload of MsgM can be used, to denotes fields to be assigned.
The intended effect of the operation is given by the conditions (see below).

Cond

When used in an invariant, a *condition* expresses a check, as seen above.
When used in a post-condition, it defines the value of an attribute or one of its fields, as a result of the execution of the operation it specifies: the name on the left-hand side denotes the element that takes the value defined by the expression on the right-hand side.
Constraint: When used in a checkdecrypt operation, invariant and postcondition must address the fields of the object they are dealing with orderly (as they are listed in the property pane of the Class of the object), and only once. In other words a field can be checked or stored, not both. This restriction reflects the way LySa deals with decryptions.
Constraint: When used in a checkmsg operation, invariant and postcondition must address only once the fields of the *in* object they are dealing with, as above and for the same reasons. In the current implementation there is no need to preserve ordering, even though it may be wise to do so, for documentation purposes. Also, there is no need to model the messages explicitely: their structure can be declared implicitly by the sequence of conditions in the postconditon of the building premsg.

Name

A *name* denotes a variable. The standard dot notation is used to access object fields: N.I denotes the variable denoted by I in the namespace of the object denoted by N.

Expr

An *expression* denotes a variable. The standard dot notation is used to access object fields: N.I denotes the value denoted by I in the namespace of the object denoted by N.

Ide

An *identifier* denotes a value or a variable, depending on the context, and evaluates as follows:

- the operation parameters evaluate to the corresponding arguments, as specified in the sequence diagram;
- the identifiers in the sequence diagram evaluate to the corresponding objects;

[2] The same restriction should appear as a precondition in the following msg operation.

- the attributes of an object (including the standard ones) denote the corresponding object or value, according to its type;
- the standard key names evaluate to the corresponding key.

The operation arguments are evaluated similarly, in the namespace of the origin of the stimulus.

Fun

The admissible operations include the encryption actions, the creation of data to be encrypted, and the creation of cryptopoints and sets of cryptopoints[3]. Following Java, a constructor has the name of the class of the objects it builds. The arguments must correspond in number and order to the fields of the class. *Warning*: these fields are given in alphabetical order for a class, in the project pane, and this order may not coincide with the relevant one, which is given in the property pane for the same class.
The labels in cryptopoints are implictly quoted, and cp is used as an abbreviation of CryptoPoint.

Pragmatics

Constraints and comments must be attached to *links* in the sequence diagram.

To view a constraints on a link in Poseidon, one has to click on the link, and select the "constraints" tab.
To modify the "body" of a constraint, one has to double click on the text, and then edit the line. Once finished, hit "return". *Warning*: do not change the focus, especially on another constraint, without exiting the editing mode: Poseidon tend to move around text, in this situation, resulting in unpleasant side-effects. To change the "type" of a constraint, click on the field, and select from the pull-dow menu.

To delete and add constraints use the red cross and curly brackets at the top left corner of the pane, respectively.

To add a comment, use the Documentation tab.

In general, use a text file to edit the specifications, and paste them in Poseidon.

The standard operations and the naming conventions

Messages

The messages shall have types *MsgM*, where M is unique for a given structure of the message. Their payloads shell be named *thePayloadY* and *theCryptedPayloadZ*, where Y and Z usually catenate the M of the message with a number reflecting the order in the message.

Data

The encrypted data shall have types *DataM*, where M is unique for a given structure of the data.

Principals

Each principal shall have as many *premsgM* operations as messages it sends, where M is the same as in the type of the sent message. The parameters of each operation (if any) shall be named *p1*, *p2*, … in the order of appeareance. Principal *objects* shall be named *i*,*j*, and *s*.

Cryptopoints should be named in a standard way: TcpN, where T is the type of the owner, and N is a progressive number.

Stimuli

Each stimulus in the sequence diagram shall have a name consisting of number reflecting its position in the sequence, and an operation from the available ones for its destination.

[3] At the moment, only singleton set can be specified.

An example of operation specification

To discuss the use of the specification language, we exploit the diagram in Figure 6, and the *narration* in Table 1, which can be generated in Choreographer by applying the option "ExtractNarration" to the UML model described above.

The protocol fragment discussed here would be expressed in the informal style used to discuss cryptographic issues as

```
A -> S: {A,kAp}:kSp
```

to express that the initiator sends a message to the server, consisting of the pair <initiator name, initiator public key>, encrypted with the server public key. The fact that asymmetric cryptography is used is left implicit, based on the name of the key. It turns out that reporting this description of the step as a comment in the operations that build and receive the message, is a useful way of documenting the protocol. Besides the name and number of each operation, each entry in Table 1 displays the following information, when present, per each link in the sequence diagram: the comment from the Documentation field, embraced by a couple of double dollar signs, and the constraint fields, tagged with their type.

The first operation in this step of the protocol (*premsg1A*) has to construct the message, which, having type *Msg1A* as declared in the type restriction of the postcondition, requires the definition of five fields:

- the message source, set to the initiator object name, *i*;
- the message sink, set to the server object name, *s*;
- the standard field of the CryptedPayload, *contents*, set to the result of encrypting (asymmetrically) a *Data* of type DataCert (according to what declared in Figure 5 by the *encrypts* association) with the public key of the server. The fields of the encrypted data are as defined above, the initiator's name and public key;
- the standard field of the CryptedPayload, *dest*, set to the cryptopoint associated with the intended place of decryption, which the designer's hindsight sets to *Scp1*, according to Figure 6;
- the standard field of the CryptedPayload, *at*, set to the cryptopoint associated with this operation, in Figure 6.

Table 1: Exemplar narration

```
1:premsg1A()            $$ A -> S: {A,kAp}:kSp $$
<postcondition>: with out:Msg1A source = i & sink = s &
      theCryptedPayload1A_1.contents = acrypt(kSp,DataCert(i,kAp)) &
      theCryptedPayload1A_1.dest = SetofCryptpoint(Scp1) &
      theCryptedPayload1A_1.at = cp(Acp1)

2:msg(out)
<precondition>: out : Msg1A
<postcondition>: in = p

3:checkmsg()            $$ A -> S: {A,kAp}:kSp
<invariant>: in.source=i, in.sink=s
<postcondition>: certAinS.contents = in.theCryptedPayload1A_1.contents

4:aDecrypt(kAp,certAinS.contents)
<precondition>: certAinS encrypts DataCert
<postcondition>: theDecryptedItem.at = cp(Scp1) &
                   theDecryptedItem.orig = SetofCryptpoint(Acp1)

5:checkdecrypt()
<invariant>: theDecryptedItem.dd.p=i
<postcondition>: kApInS = theDecryptedItem.dd.k
```

The second entry in Table 1 is standard, and provides all the information needed to specify the transfer from *out* in the initiator to *in* in the server.

The third entry repeats the comment: this is useful to the cultivated designer who may have a look at the generated LySa code. Since *premsg* and *checkmsg* there are no longer contiguous, the comment, which is reported also in the LySa code, helps in relating the pieces there. Next, the invariant expresses that the standard message fields, *source* and *sink*, are checked, to verify if they are what the designer expects. Moreover, the postcondition asserts that the informative part of the CryptedPayload of the incoming message is stored in the local variable certAinS, for further processing. The other fields, *dest* and *at*, are not relevant here.

The last two entries should be considered together, since they cooperate in opening the crypted payload, checking its contents, and saving parts for future reference. The precondition of operation number 4 is needed to declare the actual type of the *dd* field of the local variable that accepts the result of the decryption, *theDecryptedItem*. The postcondition of the same operation defines the other fields, namely

- the one labelling this decryption point, *at*, which is set to *Scp1*, according to Figure 1;
- the one stating the expected origin, i.e. the place of encryption, set to *Acp1*.

The last entry defines the checks on the result of the decryption (we require that the principal in the certificate is the initiator) and that the key must be saved locally, to be used later by the server to build the certificate for the initiator.

For the curious reader, Appendix A shows the generated LySa code.

An extended example

We present the model of the protocol *SecureSend* developed in DEGAS for the case studies. The reader is referred to [Del26] for motivations and rationale of this protocol. This model has been developed from a copy if the support. We report the structure of the principals, of the encrypted data, the sequence diagram, and the narration. The first exchange between the initiator and the server determines a certificate for the initiator in its local variable *certA*. Similarly, the second exchange establishes a certificate for the responder in *certB*. Then, initiator and responder establish a common session key, in *x* and *xInB*, respectively, that they use in the last exchange.

The generated LySa code is reported in Appendix A.

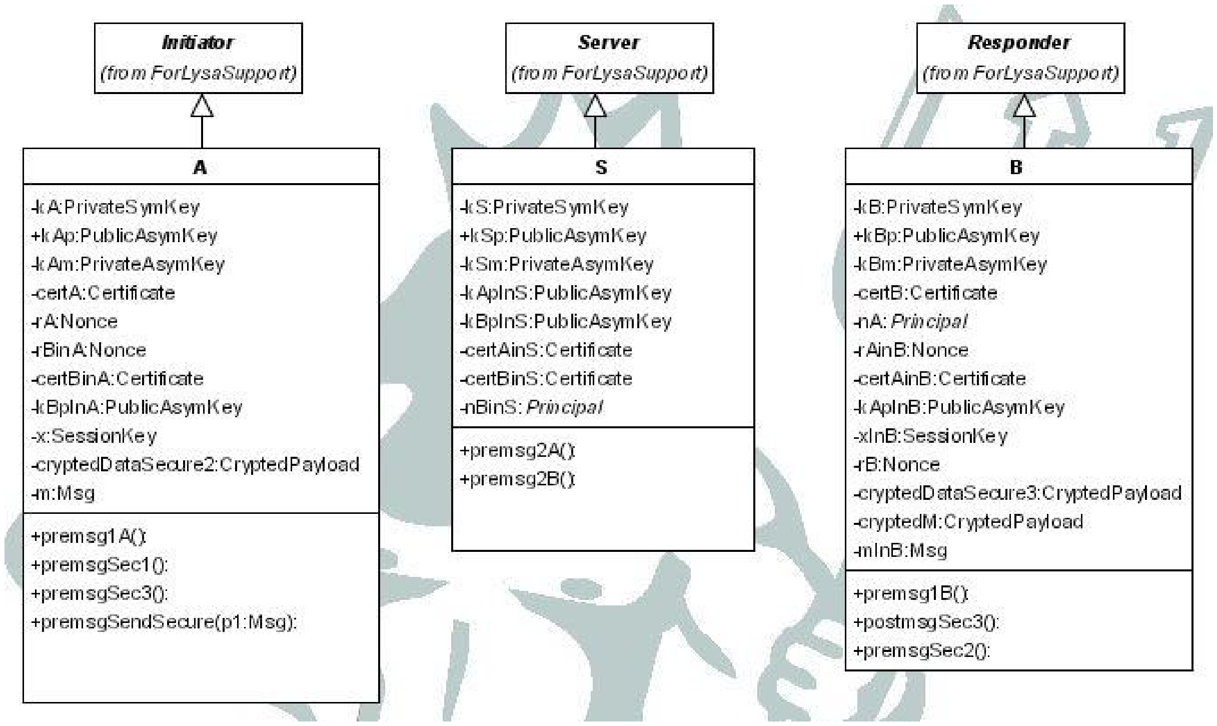

Figure 7 The structure of SecureSend

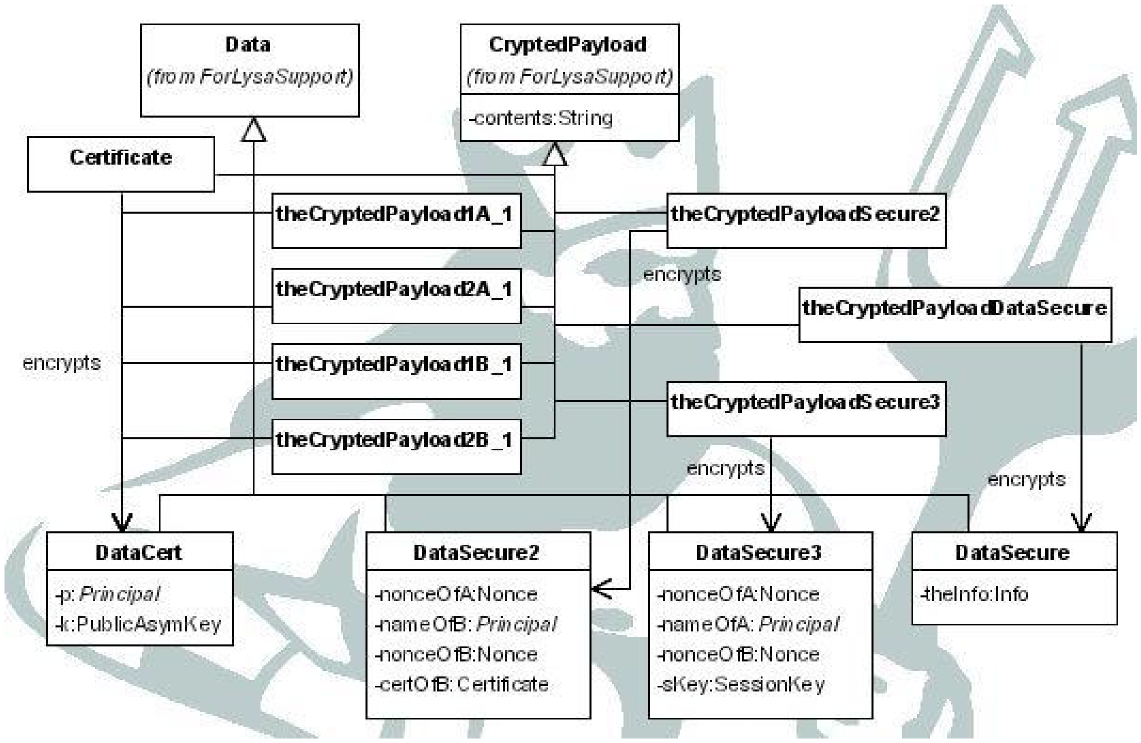

Figure 8 The data of SecureSend

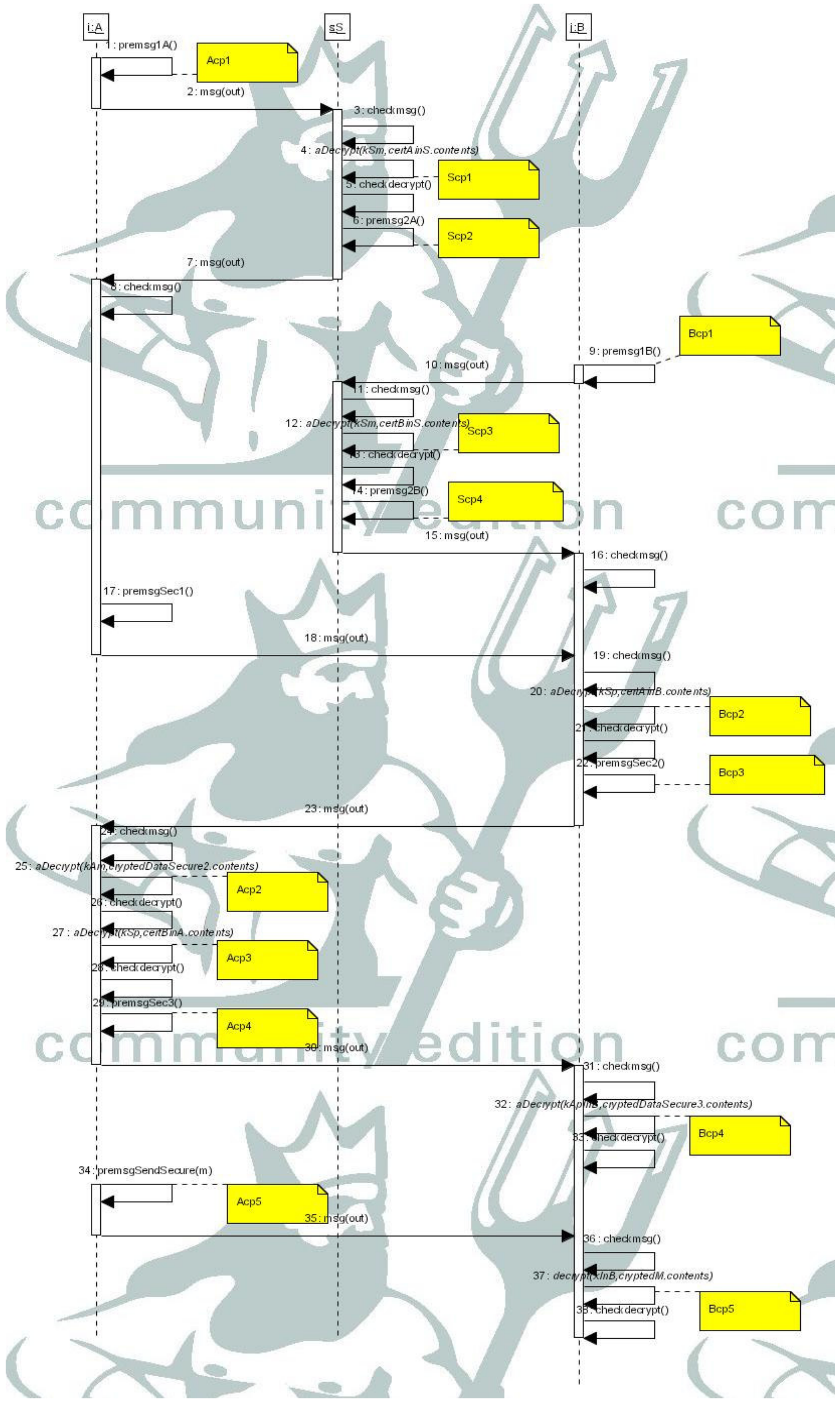

Figure 9 SecureSend sequence diagram

Table 2 The narration of SecureSend

```
1:premsg1A()            $$ A -> S: {A,kAp}:kSp
<postcondition>: [out, source = i, sink = s, theCryptedPayload1A_1.contents =
acrypt(kSp,DataCert(i,kAp)), theCryptedPayload1A_1.dest = SetofCryptpoint(Scp1),
theCryptedPayload1A_1.at = cp(Acp1)]

2:msg(out)
<precondition>: out : Msg1A
<postcondition>: [in = p]

3:checkmsg()
<invariant>: [in.source=i, in.sink=s]
<postcondition>: [certAinS.contents = in.theCryptedPayload1A_1.contents]

4:aDecrypt(kSm,certAinS.contents)
<precondition>: certAinS encrypts DataCert
<postcondition>: [theDecryptedItem.at = cp(Scp1), theDecryptedItem.orig =
SetofCryptpoint(Acp1)]

5:checkdecrypt()
<invariant>: [theDecryptedItem.dd.p=i]
<postcondition>: [kApInS = theDecryptedItem.dd.k]

6:premsg2A()            $$ S -> A: {A,kAp}:kSm
<postcondition>: [out, source = s, sink = i, theCryptedPayload2A_1.contents =
acrypt(kSm,DataCert(i,kApInS)), theCryptedPayload2A_1.at = cp(Scp2),
theCryptedPayload2A_1.dest = SetofCryptpoint(Acp2)]]

7:msg(out)
<precondition>: out : Msg2A
<postcondition>: [in = p]

8:checkmsg()
<invariant>: [in.source=s, in.sink=i]
<postcondition>: [certA.contents = in.theCryptedPayload2A_1.contents]

9:premsg1B()            $$ B -> S: {B,kBp}:kSp
<postcondition>: [out, source = j, sink = s, theCryptedPayload1B_1.contents =
acrypt(kSp,DataCert(j,kBp)), theCryptedPayload1B_1.dest = SetofCryptpoint(Scp3),
theCryptedPayload1B_1.at = cp(Bcp1)]

10:msg(out)
<precondition>: out:Msg
<postcondition>: [in = p]

11:checkmsg()
<invariant>: [in.source=j, in.sink=s]
<postcondition>: [certBinS.contents = in.theCryptedPayload1B_1.contents]

12:aDecrypt(kSm,certBinS.contents)
<precondition>: certBinS encrypts DataCert
<postcondition>: [theDecrypteItem.orig = SetofCryptpoint(Bcp1),
theDecryptedItem.at = cp(Scp3)]

13:checkdecrypt()
<postcondition>: [nBinS = theDecryptedItem.dd.p, kBpInS = theDecryptedItem.dd.k]

14:premsg2B()           $$ S -> B : {B,kBp} : kSm
```

```
<postcondition>: [out, source = s, sink = j, theCryptedPayload2B_1.contents =
acrypt(kSm,DataCert(nBinS,kBpInS)), theCryptedPayload2B_1.at = cp(Scp4),
theCryptedPayload2B_1.dest = SetofCryptpoint(Bcp2)]

15:msg(out)
<precondition>: msg:Msg2B
<postcondition>: [in = p]

16:checkmsg()
<invariant>: [in.source=s, in.sink=j]
<postcondition>: [certB.contents = in.theCryptedPayload2B_1.contents]

17:premsgSec1()         $$ A -> B : A, RA, certA
<precondition>: isNewNonce(rA)
<postcondition>: [out, source = i, sink = j, thePayloadSecure1.sender = i,
thePayloadSecure1.nonce = rA, thePayloadSecure1.cert = certA]

18:msg(out)
<precondition>: out : MsgSecure1
<postcondition>: [in = p]

19:checkmsg()
<invariant>: [in.source=i, in.sink=j]
<postcondition>: [nA = in.thePayloadSecure1.sender, rAinB =
in.thePayloadSecure1.nonce, certAinB = in.thePayloadSecure1.cert]

20:aDecrypt(kSp,certAinB.contents)        $$ B->A : ( rA, B, rB, certB}:kAp !!
the name in second position, since rA must surely be checked on receipt&nbsp;
<precondition>: certAinB encrypts DataCert
<postcondition>: [theDecryptedItem.at = cp(Bcp2), theDecryptedItem.orig =
SetofCryptpoint(Scp4)]

21:checkdecrypt()
<invariant>: [theDecryptedItem.dd.p=nA]
<postcondition>: [kApInB = theDecryptedItem.dd.k]

22:premsgSec2()         $$ B->A : ( rA, B, rB, certB}:kAp
<precondition>: isNewNonce(rB)
<postcondition>: [out, source = j, sink = i, theCryptedPayloadSecure2.contents =
acrypt(kApInB,DataSecure2(rAinB,j,rB,certB)), theCryptedPayloadSecure2.at =
cp(Bcp3), theCryptedPayloadSecure2.dest = SetofCryptpoint(Acp2)]

23:msg(out)
<precondition>: out:MsgSecure2
<postcondition>: [in = p]

24:checkmsg()
<invariant>: [in.source=j, in.sink=i]
<postcondition>: [cryptedDataSecure2.contents =
in.theCryptedPayloadSecure2.contents]

25:aDecrypt(kAm,cryptedDataSecure2.contents)
<precondition>: cryptedDataSecure2 encrypts DataSecure2
<postcondition>: [theDecrypteItem.orig = SetofCryptpoint(Bcp3),
theDecrypteItem.at = cp(Acp2)]

26:checkdecrypt()
<invariant>: [theDecryptedItem.dd.nonceOfA=rA, theDecryptedItem.dd.nameOfB=j]
<postcondition>: [rBinA = theDecryptedItem.dd.nonceOfB, certBinA =
theDecryptedItem.dd.certOfB]

27:aDecrypt(kSp,certBinA.contents)
<precondition>: certAinB encrypts DataCert
```

```
<postcondition>: [theDecrypteItem.orig = SetofCryptpoint(Scp4),
theDecrypteItem.at = cp(Acp3)]

28:checkdecrypt()
<invariant>: [theDecryptedItem.dd.p=j]
<postcondition>: [kBpInA = theDecrypteItem.dd.k]

29:premsgSec3()         $$ A -> B : {rA, nA, rB, x} : kBpInA
<precondition>: isNewKey(x)
<postcondition>: [out, source = i, sink = j, theCryptedPayloadSecure3.contents =
acrypt(kBpInA,DataSecure3(rA,i,rBinA,x)), theCryptedPayloadSecure3.at =
cp(Acp4), theCryptedPayloadSecure3.dest = SetofCryptpoint(Bcp4)]

30:msg(out)

31:checkmsg()           $$ A -> B : {rA, nA, rB, x} : kBpInA
<invariant>: [in.source=i, in.sink=j]
<postcondition>: [cryptedDataSecure3.contents =
in.theCryptedPayloadSecure3.contents]

32:aDecrypt(kApInB,cryptedDataSecure3.contents)
<precondition>: cryptedDataSecure3 encrypts DataSecure3
<postcondition>: [theDecryptedItem.at = cp(Bcp4), theDecryptedItem.orig =
SetofCryptpoint(Acp4)]

33:checkdecrypt()
<invariant>: [theDecryptedItem.dd.nonceOfA=rAinB,
theDecryptedItem.dd.nameOfA=nA, theDecryptedItem.dd.nonceOfB=rB]
<postcondition>: [xInB = theDecryptedItem.dd.sKey]

34:premsgSendSecure(m)
<precondition>: isNewMsg(m)
<postcondition>: [out, source = i, sink = j,
theCryptedPayloadDataSecure.contents = crypt(x,DataSecure(m)),
theCryptedPayloadSendSecure.at = cp(Acp5), theCryptedPayloadSendSecure.dest =
SetofCryptpoint(Bcp5)]

35:msg(out)
<invariant>: [out:Msg]
<postcondition>: [in = p]

36:checkmsg()
<invariant>: [in.source=i, in.sink=j]
<postcondition>: [cryptedM.contents = in.theCryptedPayloadDataSecure.contents]

37:decrypt(xInB,cryptedM.contents)
<precondition>: cryptedM encrypts DataSecure
<postcondition>: [theDecrypteItem.orig = SetofCryptpoint(Acp5),
theDecrypteItem.at = cp(Bcp5)]

38:checkdecrypt()
<postcondition>: [mInB = theDecryptedItem.dd.theInfo]
```

Conclusions

The approach to protocol modelling for LySa analysis has first been described in [GC2]. However, it should be noted that there we used UML profiles to support modelling. Experience with modelling in the last year has suggested that the protocol designer can make better use of a reusable package as a design framework. Besides, we used two profiles, to keep distinct what is actually needed to implement the protocol from what is additionally needed for the analysis. In this document we concentrate on the analysis, and abandon that distinction.

Another difference with respect to [GC2] is that now also asymmetric cryptography is considered.

Finally, the experience with the DEGAS cases studies suggests that a simpler profile should be formulated instead, to help the application designers that want to exploit verified protocols in expliciting the links between the application and the verified protocol. This is left to further research.

Acknowledgements

This work profited from many discussions with the DEGAS people, in meetings in Pisa, Trento and Edinburgh. Special thanks are due to Chiara Bodei and Pierpaolo Degano, for introducing us to (some of) LySa's *arcana*, to Mikael Buchholtz for careful debugging of the LySa output, to Michela Vasari and Monika Maidl, for insights on the case study protocol, and to Lara Perrone, Simone Semprini and Daniele Picciaia for programming the extractor. Val Haenel integrated the extractor in Choreographer. Last but not least, Perdita Stevens shared her insights on the nature, purpose and limits of UML profiles.

References

[Del7] P. Stevens. Interface between SENV and VENV. DEGAS Deliverable D7, 2002.
[LySa] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic Validation of Protocol Narrations. Proc. 16th IEEE Compèuter Security Foundations Workshop CSFW, IEEE Computer Society, pp 126-140, 2003.
[GC2] Mikael Buchholtz, Carlo Montangero, Lara Perrone, Simone Semprini. For-LySa: UML for Authentication Analysis. 2nd Int. Workshop on Global Computing, GC '04, Rovereto, Italy, LNCS 3267, pp. 92-105.
[MB04] M. Buchholtz. Encoding Protocol Scenarios for the Lysatool. DEGAS TR WP6-IMM-I00-Int-004. 2004.
[ForLysa] C. Montangero. The ForLysa modelling environment. Available at http://www.di.unipi.it/~monta/ ForLySa/ForLySaSupport.zuml .
[Chor] N.V. Haenel. Choreographer Platfrom User Guide. Available at http://homepages.inf.ed.ac.uk/stg/ software/CHOREOGRAPHER/UserManual.pdf .
[Del26] I.Mura, D.Spunton, M.Vasari, D. Piazza. Case Studies. DEGAS Deliverable D26, 2005.
[Del20] N. V. Haenel. Reflector. DEGAS Deliverable D20, 2005.

Appendix A

We report here the LySa code generated by ForLySa for the two examples in the text.

From section The specification language.

```
/* Restrict symmetric keys of A, B */
( ( new_{i=1} kA_{i} )
  ( new_{j=1} kB_{j} )
/* Restrict asymmetric keys of A, B, S */
  ( new +- kS )
  ( new_{i=1} +- kA_{i} )
  ( new_{j=1} +- kB_{j} )

/* Processes for A */
   (((|_{i=1}(|_{j=0}

   !< I_{i},A,I_{-1},S, {| I_{i},A,kA+_{i} |}: kS+ [at Acp1_{i,j} dest {Scp1}]
>.
/* 1.0 premsg1A $$ A -> S: {A,kAp}:kSp
<pre> null
<post> [out, source = i, sink = s, theCryptedPayload1A_1.contents =
acrypt(kSp,DataCert(i,kAp)), theCryptedPayload1A_1.dest = SetofCryptpoint(Scp1),
theCryptedPayload1A_1.at = cp(Acp1)]
*/

    0 ) ))

/* Processes for B */

    |((|_{j=1}(|_{i=0}
   !0 ) ))

/* Processes for S */

     |((|_{i=0}(|_{j=0}
   !( I_{i},A,I_{-1},S ; certAinS_{i,j}).
/* 3.0 checkmsg$$ A -> S: {A,kAp}:kSp
<inv> [in.source=i, in.sink=s]
<post> [certAinS.contents = in.theCryptedPayload1A_1.contents]
*/

    decrypt certAinS_{i,j} as {| I_{i},A ; kApInS_{i,j} |} : kA+_{i}
    [ at Scp1 orig {Acp1_{i,j}} ]
/* 4.0 aDecrypt
<post> [theDecryptedItem.at = cp(Scp1), theDecryptedItem.orig =
SetofCryptpoint(Acp1)]
5.0 checkdecrypt
<inv> [theDecryptedItem.dd.p=i]
<post> [kApInS = theDecryptedItem.dd.k]
*/
in 0 ) ))
/* Makes public keys available (kSp,kAp,kBp) */
      |( <kS+, I_{0}, A, I_{0}, B, kA_{0}, kB_{0}, kA-_{0}, kB-_{0}>.0 )
       |( ( |_{i=0} <kA+_{i}, kB+_{i}>.0 ))
 )
)
```

From section An extended example

```
/* Restrict symmetric keys of A, B */
( ( new_{i=1} kA_{i} )
  ( new_{j=1} kB_{j} )
/* Restrict asymmetric keys of A, B, S */
  ( new +- kS )
  ( new_{i=1} +- kA_{i} )
  ( new_{j=1} +- kB_{j} )

/* Processes for A */
   (((|_{i=1}(|_{j=0}

   !< I_{i},A,I_{-1},S, {| I_{i},A,kA+_{i} |}: kS+ [at Acp1_{i,j} dest {Scp1}]
>.
/* 1.0 premsg1A $$ A -> S: {A,kAp}:kSp
<pre> null
<post> [out, source = i, sink = s, theCryptedPayload1A_1.contents =
acrypt(kSp,DataCert(i,kAp)), theCryptedPayload1A_1.dest = SetofCryptpoint(Scp1),
theCryptedPayload1A_1.at = cp(Acp1)]
*/

    ( I_{-1},S,I_{i},A ; certA_{i,j}).
/* 8.0 checkmsg
<inv> [in.source=s, in.sink=i]
<post> [certA.contents = in.theCryptedPayload2A_1.contents]
*/

     (( new rA_{i,j} )
      < I_{i},A,I_{j},B,I_{i},A,rA_{i,j},certA_{i,j} >.
/* 17.0 premsgSec1 $$ A -> B : A, RA, certA
<pre> isNewNonce(rA)
<post> [out, source = i, sink = j, thePayloadSecure1.sender = i,
thePayloadSecure1.nonce = rA, thePayloadSecure1.cert = certA]
*/

       ( I_{j},B,I_{i},A ; cryptedDataSecure2_{i,j}).
/* 24.0 checkmsg
<inv> [in.source=j, in.sink=i]
<post> [cryptedDataSecure2.contents = in.theCryptedPayloadSecure2.contents]
*/

        decrypt cryptedDataSecure2_{i,j} as {| rA_{i,j},I_{j},B ;
rBinA_{i,j},certBinA_{i,j} |} : kA-_{i}
        [ at Acp2_{i,j} orig {Bcp3_{i,j}} ]
/* 25.0 aDecrypt
<post> [theDecrypteItem.orig = SetofCryptpoint(Bcp3), theDecrypteItem.at =
cp(Acp2)]
26.0 checkdecrypt
<inv> [theDecryptedItem.dd.nonceOfA=rA, theDecryptedItem.dd.nameOfB=j]
<post> [rBinA = theDecryptedItem.dd.nonceOfB, certBinA =
theDecryptedItem.dd.certOfB]
*/

         in decrypt certBinA_{i,j} as {| I_{j},B ; kBpInA_{i,j} |} : kS+
            [ at Acp3_{i,j} orig {Scp4} ]
/* 27.0 aDecrypt
<post> [theDecrypteItem.orig = SetofCryptpoint(Scp4), theDecrypteItem.at =
cp(Acp3)]
28.0 checkdecrypt
<inv> [theDecryptedItem.dd.p=j]
<post> [kBpInA = theDecrypteItem.dd.k]
*/

              in (( new x_{i,j} )
                  < I_{i},A,I_{j},B, {| rA_{i,j},I_{i},A,rBinA_{i,j},x_{i,j} |}:
kBpInA_{i,j} [at Acp4_{i,j} dest {Bcp4_{i,j}}] >.
/* 29.0 premsgSec3 $$ A -> B : {rA, nA, rB, x} : kBpInA
<pre> isNewKey(x)
<post> [out, source = i, sink = j, theCryptedPayloadSecure3.contents =
acrypt(kBpInA,DataSecure3(rA,i,rBinA,x)), theCryptedPayloadSecure3.at =
cp(Acp4), theCryptedPayloadSecure3.dest = SetofCryptpoint(Bcp4)]
*/

                   (( new m_{i,j} )
                    < I_{i},A,I_{j},B, { m_{i,j} }: x_{i,j} >.
/* 34.0 premsgSendSecure
<pre> isNewMsg(m)
<post> [out, source = i, sink = j, theCryptedPayloadDataSecure.contents =
crypt(x,DataSecure(m)), theCryptedPayloadSendSecure.at = cp(Acp5),
theCryptedPayloadSendSecure.dest = SetofCryptpoint(Bcp5)]
*/

                     0
                   )

                 )

     )
     ) ))

/* Processes for B */

    |((|_{j=1}(|_{i=0}
   !< I_{j},B,I_{-1},S, {| I_{j},B,kB+_{j} |}: kS+ [at Bcp1_{i,j} dest {Scp3}]
>.
/* 9.0 premsg1B $$ B -> S: {B,kBp}:kSp
<pre> null
<post> [out, source = j, sink = s, theCryptedPayload1B_1.contents =
acrypt(kSp,DataCert(j,kBp)), theCryptedPayload1B_1.dest = SetofCryptpoint(Scp3),
theCryptedPayload1B_1.at = cp(Bcp1)]
*/

    ( I_{-1},S,I_{j},B ; certB_{i,j}).
/* 16.0 checkmsg
<inv> [in.source=s, in.sink=j]
<post> [certB.contents = in.theCryptedPayload2B_1.contents]
*/

     ( I_{i},A,I_{j},B ; nA_{i,j},nAType_{i,j},rAinB_{i,j},certAinB_{i,j}).
/* 19.0 checkmsg
<inv> [in.source=i, in.sink=j]
<post> [nA = in.thePayloadSecure1.sender, rAinB = in.thePayloadSecure1.nonce,
certAinB = in.thePayloadSecure1.cert]
*/

      decrypt certAinB_{i,j} as {| nA_{i,j},nAType_{i,j} ; kApInB_{i,j} |} : kS+
      [ at Bcp2_{i,j} orig {Scp4} ]
/* 20.0 aDecrypt
<post> [theDecryptedItem.at = cp(Bcp2), theDecryptedItem.orig =
SetofCryptpoint(Scp4)]
21.0 checkdecrypt
<inv> [theDecryptedItem.dd.p=nA]
<post> [kApInB = theDecryptedItem.dd.k]
*/
```

```
       in (( new rB_{i,j} )
           < I_{j},B,I_{i},A, {| rAinB_{i,j},I_{j},B,rB_{i,j},certB_{i,j} |}:
kApInB_{i,j} [at Bcp3_{i,j} dest {Acp2_{i,j}}] >.
/* 22.0 premsgSec2 $$ B->A : ( rA, B, rB, certB}:kAp
<pre> isNewNonce(rB)
<post> [out, source = j, sink = i, theCryptedPayloadSecure2.contents =
acrypt(kApInB,DataSecure2(rAinB,j,rB,certB)), theCryptedPayloadSecure2.at =
cp(Bcp3), theCryptedPayloadSecure2.dest = SetofCryptpoint(Acp2)]
*/

            ( I_{i},A,I_{j},B ; ).
/* 31.0 checkmsg$$ A -> B : {rA, nA, rB, x} : kBpInA
<inv> [in.source=i, in.sink=j]
<post> [cryptedDataSecure3.contents = in.theCryptedPayloadSecure3.contents]
*/

             decrypt cryptedDataSecure3_{i,j} as {|
rAinB_{i,j},nA_{i,j},nAType_{i,j},rB_{i,j} ; xInB_{i,j} |} : kApInB_{i,j}
             [ at Bcp4_{i,j} orig {Acp4_{i,j}} ]
/* 32.0 aDecrypt
<post> [theDecryptedItem.at = cp(Bcp4), theDecryptedItem.orig =
SetofCryptpoint(Acp4)]
33.0 checkdecrypt
<inv> [theDecryptedItem.dd.nonceOfA=rAinB, theDecryptedItem.dd.nameOfA=nA,
theDecryptedItem.dd.nonceOfB=rB]
<post> [xInB = theDecryptedItem.dd.sKey]
*/

              in ( I_{i},A,I_{j},B ; cryptedM_{i,j}).
/* 36.0 checkmsg
<inv> [in.source=i, in.sink=j]
<post> [cryptedM.contents = in.theCryptedPayloadDataSecure.contents]
*/

                   decrypt cryptedM_{i,j} as {  ; mInB_{i,j} } : xInB_{i,j}
                   [ at Bcp5_{i,j} orig {Acp5_{i,j}} ]
/* 37.0 decrypt
<post> [theDecrypteItem.orig = SetofCryptpoint(Acp5), theDecrypteItem.at =
cp(Bcp5)]
38.0 checkdecrypt
<inv> []
<post> [mInB = theDecryptedItem.dd.theInfo]
*/
in 0

          )
     ) ))

/* Processes for S */

     |((|_{i=0}(|_{j=0}
   !( I_{i},A,I_{-1},S ; certAinS_{i,j}).
/* 3.0 checkmsg
<inv> [in.source=i, in.sink=s]
<post> [certAinS.contents = in.theCryptedPayload1A_1.contents]
*/

    decrypt certAinS_{i,j} as {| I_{i},A ; kApInS_{i,j} |} : kS-
    [ at Scp1 orig {Acp1_{i,j}} ]
/* 4.0 aDecrypt
<post> [theDecryptedItem.at = cp(Scp1), theDecryptedItem.orig =
SetofCryptpoint(Acp1)]
```

```
5.0 checkdecrypt
<inv> [theDecryptedItem.dd.p=i]
<post> [kApInS = theDecryptedItem.dd.k]
*/

     in < I_{-1},S,I_{i},A, {| I_{i},A,kApInS_{i,j} |}: kS- [at Scp2 dest
{Acp2_{i,j}}] >.
/* 6.0 premsg2A $$ S -> A: {A,kAp}:kSm
<pre> null
<post> [out, source = s, sink = i, theCryptedPayload2A_1.contents =
acrypt(kSm,DataCert(i,kApInS)), theCryptedPayload2A_1.at = cp(Scp2),
theCryptedPayload2A_1.dest = SetofCryptpoint(Acp2)]]
*/

         ( I_{j},B,I_{-1},S ; certBinS_{i,j}).
/* 11.0 checkmsg
<inv> [in.source=j, in.sink=s]
<post> [certBinS.contents = in.theCryptedPayload1B_1.contents]
*/

          decrypt certBinS_{i,j} as {|  ;
nBinS_{i,j},nBinSType_{i,j},kBpInS_{i,j} |} : kS-
          [ at Scp3 orig {Bcp1_{i,j}} ]
/* 12.0 aDecrypt
<post> [theDecrypteItem.orig = SetofCryptpoint(Bcp1), theDecryptedItem.at =
cp(Scp3)]
13.0 checkdecrypt
<inv> []
<post> [nBinS = theDecryptedItem.dd.p, kBpInS = theDecryptedItem.dd.k]
*/

           in < I_{-1},S,I_{j},B, {| nBinS_{i,j},nBinSType_{i,j},kBpInS_{i,j}
|}: kS- [at Scp4 dest {Bcp2_{i,j}}] >.
/* 14.0 premsg2B $$ S -> B : {B,kBp} : kSm
<pre> null
<post> [out, source = s, sink = j, theCryptedPayload2B_1.contents =
acrypt(kSm,DataCert(nBinS,kBpInS)), theCryptedPayload2B_1.at = cp(Scp4),
theCryptedPayload2B_1.dest = SetofCryptpoint(Bcp2)]
*/

               0
          ) ))
/* Makes public keys available (kSp,kAp,kBp) */
      |( <kS+, I_{0}, A, I_{0}, B, kA_{0}, kB_{0}, kA-_{0}, kB-_{0}>.0 )
       |( ( |_{i=0} <kA+_{i}, kB+_{i}>.0 ))
 )
)
```