For-LySa: UML for Authentication Analysis*

Mikael Buchholtz[1], Carlo Montangero[2], Lara Perrone[2], and Simone Semprini[3]**

[1] mib@imm.dtu.dk, Informatics and Mathematical Modelling, Technical University of Denmark, Richard Petersens Plads, DTU-bldg. 321, DK-2800 Kgs. Lyngby, Denmark
[2] monta@di.unipi.it, Dipartimento di Informatica, Università di Pisa, Via F. Buonarroti 2 I-56127 Pisa, Italy
[3] semprini@itc.it, Automated Reasoning Systems Division, ITC-IRST, Via Sommarive 18, I-38050 Povo – Trento, Italy

Abstract. The DEGAS project aims at enriching standard UML-centred development environments in such a way that the developers of global applications can exploit automated formal analyses with minimal overhead. In this paper, we present For-LySa, an instantiation of the DEGAS approach for authentication analysis, which exploits an existing analysis tool developed for the process calculus LySa. We discuss what information is needed for the analysis, and how to build the UML model of an authentication protocol in such a way that the needed information can be extracted from the model. We then present our prototype implementation and report on some promising results of its use.

1 Introduction

Many years of research in formal methods have resulted in a wealth of analysis tools that, in theory, may assist software designers in the development of high quality products. In practice, however, these tools are often hard to use for non-experts and their direct, practical impact is therefore limited.

The overall aim of this paper is to illustrate that formal analysis tools can be used directly by designers of applications for global computing. To this end, we follow the approach of the DEGAS project where the idea, as illustrated in Figure 1, is to let developers use their own *development environment* while the formal analysis takes place in its own *verification environment*. More precisely, the development environment will be the Unified Modelling Language (UML) that with is recent popularity in industry has a direct influence on many real world applications. The verification environment uses process calculi, which are *behavioural models* of systems, and the analysis of these calculi will therefore concentrate on behavioural aspects of systems. This nicely complements analyses of

* This work is partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies, under the IST-2001-32072 project DEGAS.

** This work was carried out when Simone Semprini was at the Dipartimento di Informatica, Università di Pisa

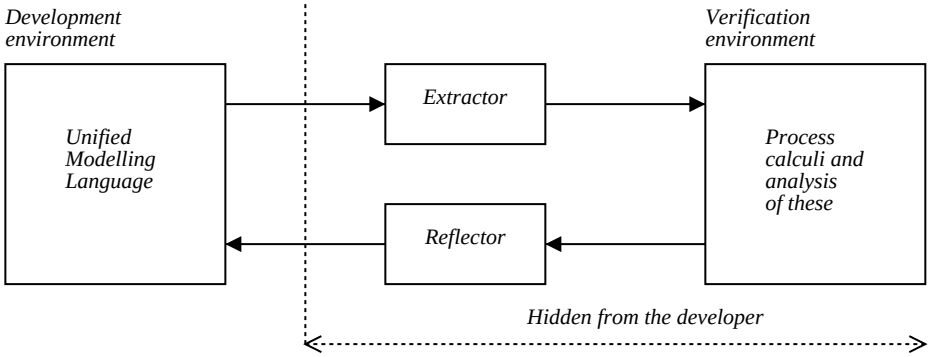

Fig. 1. Overview of the DEGAS approach to analysis of system design.

structural aspects such as well-typedness of object hierarchies and inter-diagram consistency, which are the kind of analysis that are typically carried out on UML today.

In order to perform analysis of the UML models in a verification environment the first step is to use an *extractor*, which extracts the parts of the model that will be relevant for the analysis and put these into the verification environment. After the analysis has been completed, the analysis result is made available to the developer using a *reflector*. To make this approach practical, from the point of view of the developer, the extractor, the analysis, and the reflector will all be automated and hidden from the developer. Thus, the developer will not need to know the finer details of these elements but may concentrate on the UML design of the system.

The main novelty of this paper, thus, is to illustrate that standard verification tools can indeed be used to analyse *security properties* of UML models. To this end, we give an overview of the For-LySa framework: an instantiation of the DEGAS approach targeted at designers of security critical applications that use network communication. Section 2 describes the application domain along with the security property of *authentication*, which will be checked in the verification environment based on the process calculus LySa [4]. Section 3 contains the UML modelling of applications using secure network communication including some additional features to cater for authentication analysis. Section 4 describes our prototype implementation of the For-LySa framework and, finally, Section 5 concludes the paper and comments on future and related work.

2 Security Protocols and Authentication

In a global computing environment, applications are typically distributed onto various host or principals, which communicate through a computer network.

These communication patterns constitute a network protocol, which comprises the applications executed at the individual principals as well as their network communication. While we may rely on (some of) these principals to be trustworthy when executing the application, the network itself must be considered *unsafe* in the sense that hostile principals might tamper with the network messages.

The usual remedy to protect network protocols from intervention by malicious attackers is to apply cryptography so that parts of the messages may be kept outside the control of the attacker. In this paper we will illustrate how our approach works for a class of "classical" authentication protocols that use a shared server and symmetric key cryptography where the same key is used for encryption and decryption. This restricted, but representative, setup is chosen primarily to kept the extractor simple and we foresee no other significant challenges, neither for the UML modelling nor for the verification tool, in catering for more general scenarios. More precisely, we consider a network scenario in which a special principal, S, initially shares a unique key with each of the principals $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ and no other principals know these keys. The purpose of an authentication protocol that operate in this scenario is to allow two arbitrary principals A_i and B_j to be certain that a communication takes place between precisely these two principals and no one else.

To meet this goal, A_i, B_j, and S may, for example, engage in the following version of the Wide-mouthed-frog Protocol [5] (where we write $\{message\}_{key}$ for a message encrypted under a key):

$$\begin{array}{l} 1.\ A_i \rightarrow S \ \ : A_i, \{B_j, K\}_{K_{SA_i}} \\ 2.\ S \ \ \rightarrow B_j : \{A_i, K\}_{K_{SB_j}} \\ 3.\ A_i \rightarrow B_j : \{message\}_K \end{array}$$

In the protocol, the principal A_i generates a session key K, which it sends to the server, encrypted under the key K_{SA_i} shared only between S and A_i. The server decrypts the session key and forwards it encrypted to the B_j, which will afterwards be able to decrypt message 3 send by A_i. It is important to stress that *both* the message sequence *and* the internal action of each of the principals are equally important parts of the description of the protocol. Therefore, all these aspects will be modelled in UML in order to make the model amenable for a precise analysis of whether a protocol obtains its goal.

We focus on checking an authentication property, which loosely speaking says that "messages should end up in the right places". For example, if we consider the first message of the WMF, a property that we might like to have is that the message $A_i, \{B_j, K\}_{K_{SA_i}}$ should end up at S, only. However, nothing prevents an attacker from forwarding the two parts of the message to other principals than S so this property does not hold if the protocol is under influence of an attacker. Instead, the property we consider focuses on the parts of messages, which are not under the control of the attacker, namely, the parts where encryption has been applied. For example, the property that should hold for the first message of WMF is that the encrypted message $\{B_j, K\}_{K_{SA_i}}$ should be decrypted at S, only.

To specify the precise details in this kind of property we annotate the UML model giving a name, ℓ, to each point of encryption and each point of decryption. Furthermore, each encryption point will be annotated with which decryption points the encrypted message is intended to be decrypted at and, conversely, for decryption.

Our verification environment is based around the processes calculi LySa and a control flow analysis of this calculus [4]. LySa is a processes calculus in the π-calculus tradition [12] but tailored specifically to model central aspects of security protocols. The aim of the analysis is to tell whether authentication properties are satisfied *for all executions* of a LySa process executed in parallel with an arbitrary attacker process. The analysis will report *all possible breaches* of the authentication properties in an error component ψ: finding a pair (ℓ, ℓ') in ψ means that something encrypted at ℓ might be decrypted at ℓ' thereby breaking the specified authentication property.

The analysis works in form of a control flow analysis, which computes over-approximations to the behaviour of all executions of a LySa process. In particular, it computes over-approximations to the error component, which means that the analysis may report an error that is not actually there. However, it is proven in [4] that the analysis will *never report too few errors* and also illustrated that reporting too many errors is not at big problem in practice.

3 UML for Authentication Protocols

To model security protocols in a consistent way in UML, we define two UML profiles. They must be used when modelling specific protocols in order to make them amenable for analysis. First we present the profile *Static For-LySa* that describes how the concepts from the previous section, such as principals, keys, messages, etc., are modelled in UML. Next, we introduce a second profile, *For-LySa*, that is used to describe the dynamics of a protocol as well as the information needed for the analysis. Rather than presenting the profiles in tabular form, we present domain models, with the understanding that their classes and relations are the stereotypes in the profile. Note that we have two profiles to keep distinct what is actually needed to implement the protocol from what is additionally needed for the analysis.

3.1 The Static For-LySa Profile

The classes and associations in the class diagram on Figure 2 define the stereotypes in the profile Static For-LySa. The central classes in the diagram are Principal, Key, and Msg (for messages).

Keys can either be a SessionKey generated for each session or it can be the PrivateKey of a principal that is shared in advance with the Server. A Server is a special kind of principal that knows the private keys of all the other principals in the protocol. We represent this knowledge as an operation key() that, given

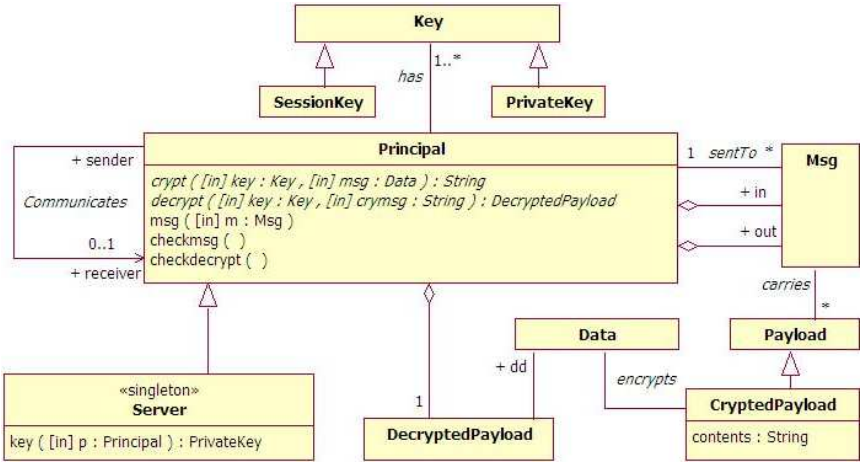

Fig. 2. The domain model.

a principal, returns its private key. The stereotype ⟨⟨singleton⟩⟩ ensures that in a given protocol specification only one server is used.

The principals communicate and exchange messages as shown by the communicates and sentTo associations. To express communication, the operation msg() can be invoked on the principal, which 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(), i.e. that messages are put into this variable and then sent. Messages carry Payloads, some of which can be CryptedPayloads, which carry Data in their contents. 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.

To specify a protocol, one needs to specify subtypes of Principal that introduce specific operations to set the outgoing messages, using the parts of the incoming ones as well as specific information held by the principal in private attributes. These operations should be introduced in a standardised way, that we will discuss in the sequel. The same applies to the operations that are needed to disassemble the incoming messages. However, there are some generic operations that can be used to build and open the messages. Some of these are left abstract, namely crypt() and decrypt(), since we leave the choice of the cryptographic algorithms open to further specialisation. In fact, the analysis treats encryption as abstract operations so we would not get more precise analysis results by specialising these operations further. The other two generic operations, checkMsg() and checkDecrypt() are null operations: they are introduced to allow the specifier to express the checks that need to be done on the incoming messages, and on

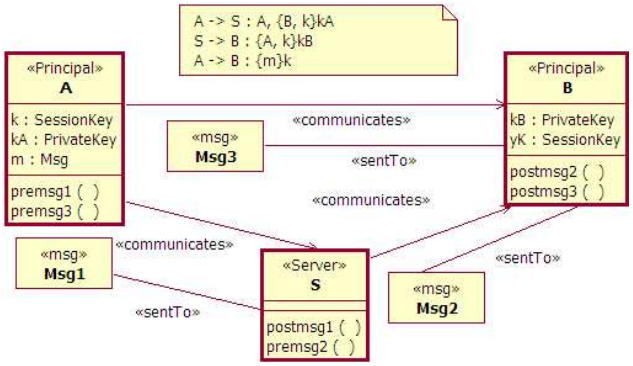

Fig. 3. The WMF protocol overview.

the results of decryption actions, respectively. These checks can be conveniently expressed as constraints in the sequence diagram that describes the dynamics of the protocol in the For-LySa profile in Section 3.2. There, the protocol designer can use checkMsg() and checkDecrypt() as placeholders, to introduce the constraints on the incoming and decrypted data.

We assume that decrypt() leaves its result in variables of type DecryptedPayload that are named systematically for each relevant type of message, as we will see later. In this way, the checks and the operations that build new messages can exploit the results of decryptions in these variables. In a DecryptedPayload data can be accessed via dd (of type Data). The distinction of Data and DecryptedData is not strictly necessary, at this level of presentation, but it helps when extending the model to introduce the decorations needed for the analysis, as we will do in Section 3.2.

As an example of the use of Static For-LySa, we present the overview of the WMF protocol, described in the previous section. Figure 3 presents the structure of the protocol, showing the intended communications and the involved messages. The types of the principals are named A, B, and S for the initiator, the responder, and the server, respectively. The diagram also makes clear how the message types are named systematically, Msg1, Msg2, and Msg3, according to the order in which the messages are sent. The structure of each message is specified in distinct diagrams such as the one in Figure 4 that makes clear how the various parts of the message are named by systematically appending indexes to their types. Similar diagrams are introduced for the other message types.

In the diagram in Figure 3 we also introduce the names of the operations to build and dissect messages: for each message of type Msg$i$, there are operations premsg$i$ in the sender and postmsg$i$ in the receiver, to build and to open the message, respectively. The semantics of these operation will be specified by post-conditions. Finally, we introduce names for the local information of each principal, like private keys, session keys, and temporary storage, such as variable yK in principal B. This variable is needed by the responder to store the key,

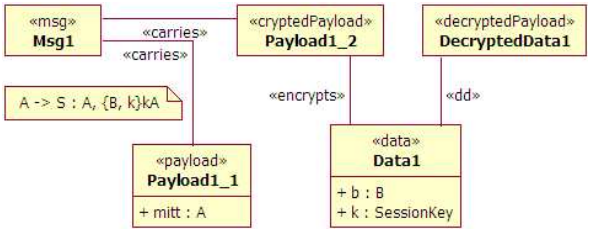

Fig. 4. The structure of Msg1.

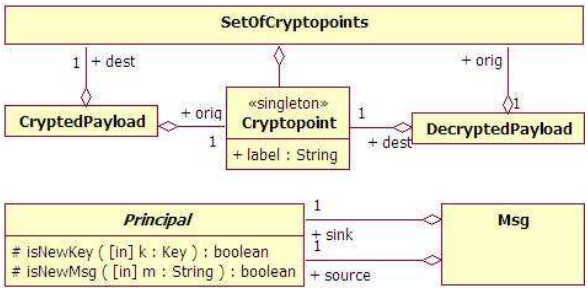

Fig. 5. The analysis model.

received in the second message, to be able to decrypt the third message sent by the initiator.

3.2 The For-LySa Profile

The UML view of the concepts that are needed to perform the authentication analysis are shown in Figure 5. In this figure, when we use the same names as in Figure 2 we denote entities that are specialising homonymous entities in the domain model. The other classes are new concepts, introduced for the analysis. These classes, and their associations and constraints define profile *For-LySa*.

First of all, each message carries with it the definition of the source principal, from which it is sent, along with the sink principal, which it should reach. Second, each encrypted payload is decorated with Cryptopoints. The idea is that, for each encrypted payload the annotations make explicit its origin, i.e. the point in the narration where the payload is encrypted, and its destinations, i.e. the set of the *intended* points of decryption. Similarly, for the decrypted data, the annotations make explicit the destination, i.e. the point in the narration where they are decrypted, and their *intended* origins, i.e. the set of expected places of encryption. Each crypto-point is a label, that will be associated to a single point of encryption (one of the premsg$i$) or decryption (postmsg$i$) in the dynamic view of the protocol.

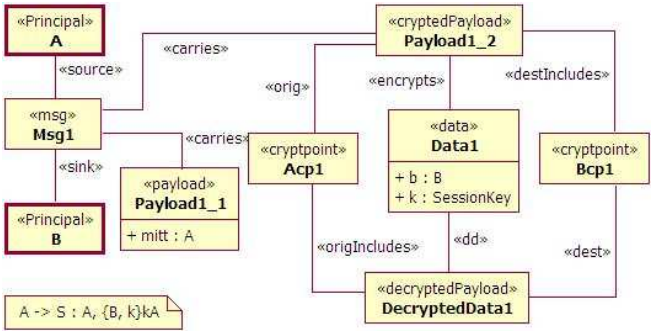

Fig. 6. The structure of Msg1 for the analysis.

As an example of the use of the For-LySa profile, Figure 6 presents the complete description of Msg1 including the decorations needed to specify the authentication property. The intended origin and destination of the encrypted part of the first message of WMF are specified to be Acp1 and Scp1, respectively. The stereotype ⟪destIncludes⟫ in Figure 6 is defined as the composition of the two aggregations from CryptedPayload to Cryptopoint in Figure 5. Similarly for ⟪origIncludes⟫. Similar diagrams describe the remaining two messages in the protocol.

To complete the WMF example, we need to address the dynamics of the protocol and this is done in a sequence diagram shown in Figure 7. The diagram adopts naming conventions consistent with those used in our scenario: the typical initiator object of type A is named i, the responder object of type B is named j, while the server object of type S is named s.

Each step in the protocol is divided into three sub-steps:

1. the sender packages the message,
2. the message is communicated,
3. the recipient processes the incoming message.

The third step is typically the most involved and includes tasks such as checking that the message format is correct, decrypting the parts intended for the current recipient, and storing the content of the

First, premsg1() builds the message in the out message of the sender. Second, msg() sends it to the recipient where it is stored as the in message. Finally, the recipient processes the received message by checking the message format with checkmsg(), decrypting the relevant part with postmsg1(), and ensuring the encrypted data also have the correct format with checkdecrypt(). When all these checks succeed the protocol continues similarly with the second and third messages (omitted for sake of space).

Operation msg(m: Msg) is polymorphic and accepts any message. However, the effective type of the message that is exchanged in each step, has to be specified as a constraint on the argument of msg(), as shown in the diagram. The

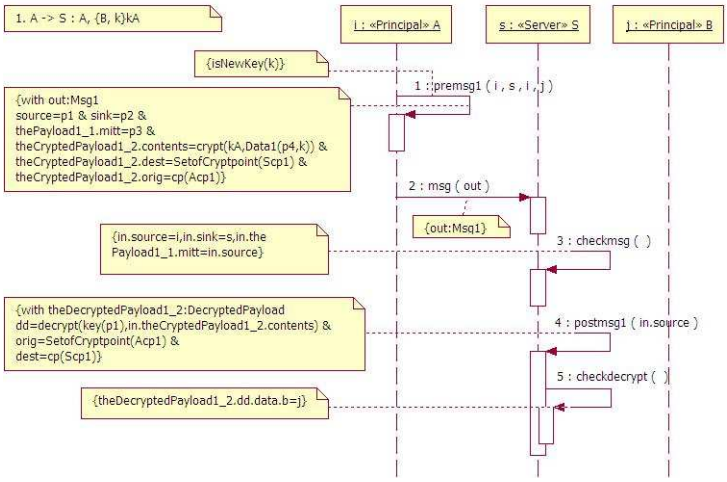

Fig. 7. The sequence diagram for WMF.

other operations are not polymorphic, and have different names (and likely parameters) in each step. The signature of the operation is specified in the overview diagram of the analysis level, which is otherwise similar to Figure 3 and is not presented for space sake.

The operations are specified via post-conditions on the state of their principal. Post-conditions are attached to the operations as constraints, as shown in Figure 7 for premsgs and postmsgs. The natural place to attach these constraints would be the operations themselves, in the overview diagram, since they are the definition of the operation semantics. However, it is easier to follow the behaviour of the protocol having the post-condition attached to the call rather than to the definition, in another diagram.

We use a very simple language to write post-conditions. There is a record scope opener *á la* Pascal, for readability:

$$\mathsf{with}\, x : T \,\langle \text{condition} \rangle$$

means that x is constrained to be of type T, and that its selectors need not be prefixed by x in $\langle$condition$\rangle$. Conditions are conjunctions of equalities, where the left hand side identifies a field of the object, and the right side is an expression for its value. We use the standard dot notation to access object fields and to navigate along associations.

Expressions are built out of constructors, like SetofCryptpoint and Data$i$; operations like crypt and decrypt; variables, either parameters like p1 and p2, or locals of the principal that performs the operation, like out, kA, and k; and constants, i.e. the names of the objects, like i, and crypto-points labels, like Acp1.

The arguments to the constructors give values to the fields, in the order given in the diagrams that introduce them. Singleton crypto-point sets are built from the label, like in the last but one line of the post-condition for premsg1. The factory method cp builds Cryptopoint objects out of labels.

There are a couple of assumptions, with respect to keys:

- *private keys* can be freely used in the operations of the owner, since they are assumed to be initialised before the protocol starts;
- *session keys* must be initialised explicitly before they are used: for this purpose the For-LySa profile has the predicate isNewKey() (see Principal in Figure 5) that can be used in a constraint before the first use of the variable. Using constraints leads to more concise diagrams than using explicit initialising operations, and has a straightforward mapping in a restriction operation in LySa.

As an example, the constraint attached to premsg1(i, s, i, j) in Figure 7 specifies that the value of the local variable out of initiator i of type A will be a message of the form $\mathsf{i}, \{\mathsf{j}, \mathsf{k}\}_{\mathsf{kA}}$, i.e. of the form of the message in the first step in WMF. Also, it describes the annotations of the authentication property, where Acp1 is the crypto-point associated with the encryption performed here, and Scp1 is associated to the corresponding decryption in postmsg1.

The post-condition attached to postmsg1 defines the effect of decrypting the message received by the server, in its variable theDecryptedPayload1_2. This is an example of the convention on decrypting actions: they leave their results in variables with names theDecryptedPayload$i$, where i is the same index of the corresponding CryptedPayload. In this example, the data are decrypted from the incoming message using a key passed as a parameter to the operation.

A number of checks on the messages have to be made explicit, to express dynamic constraints on the messages. These checks are expressed in UML as *invariant* constraints attached to the checking operations, in the sequence diagram. They are lists of equalities, with the syntax given above.

The source and sink of each message should be checked against the expected value. Additional checks depends on the specifics of the protocol, like the third condition attached to the third operation in Figure 7, which states that the clear payload must be equal to the message source, i.e. in this protocol each initiator can only speak for itself. Similarly, the next check, in the fifth operation, controls that the incoming responder (the b field in the encrypted payload) is indeed the intended one, namely j.

4 The For-LySa Prototype

We have developed a prototype implementation that can validate authentication properties of applications modelled in UML using the For-LySa profiles. The overall architecture of this *For-LySa prototype* follows the DEGAS approach on Figure 1.

In the For-LySa prototype, UML models are designed with Rational XDE version 1.5, and exported into XMI version 1.1, which is a standard, XML-based way to represent UML models.

The extractor is written in Java and takes as input the XMI representation of the UML model and delivers as output a corresponding LySa process annotated with the security properties specified in the UML model. The implementation of the extractor benefits from a generic Java library for writing extractors, which has been developed within the DEGAS project as part of the Java version of the PEPA Workbench [1]. The main operations of the extractor are: parsing the XMI file, building an intermediate representation, and finally generating a LySa process.

The verification tool is implemented in Standard ML and is available for download on the Web [2]. It takes as input a parameterised LySa process generated by the extractor and makes a finite instantiation of the scenario with $i = 1, \ldots, n$ and $j = 1, \ldots, m$ of principals A and B, respectively. The analysis, which is carried out on this finite instantiation of the scenario, returns an error component, ψ, containing pairs of crypto-points where the authentication property may be violated as explained in Section 2. The extractor has added indices ij to these crypto-points such that they will, in general, be of the form ℓ_{ij}.

Our current prototype does not include a reflector, as such. We simply, present the error component, ψ, to the developer. As illustrated in the next section, this information can directly be of use to the developer.

4.1 A Case Study

Using the For-LySa prototype on our running example, the WMF protocol, the analyser returns an empty error component, stating that no problems occur in any execution of the protocol — even in the presence of an attacker.

More precisely, the For-LySa prototype validates the protocol deployed in the scenario described in Figure 3 where additional attacker principals have access to the network. The For-LySa prototype validates that the authentication properties specified in the annotations to the UML model in Figure 6 and Figure 7 indeed hold for the WMF. That is, it validates that messages can only be successfully decrypted at the places specified in the annotations no matter what an attacker may try.

To illustrate the fine details that decides whether a protocol behaves correctly or not consider a slightly modified version of the WMF protocol where the first message is modified so that the identity of the responder is no longer encrypted. This affects, of course, the construction of the message in premsg1() as well as the checks made by the server in ckeckmsg(), postmsg1(), and ckeckdecrypt().

When we run the For-LySa prototype on this modified WMF protocol, it gives a non-empty error component, i.e. it reports that the authentication properties may be violated. Summarising the result in the error component, the analyser reports that something may go wrong because:

– something encrypted at $\mathsf{Acp1}_{ij}$ may be decrypted at Scp1,

– something encrypted at $\mathsf{Acp2}_{ij}$ may be decrypted at $\mathsf{Bcp2}_{ij'}$ for $j \neq j'$, i.e. at a wrong responder,
– something encrypted at $\mathsf{Acp2}_{ij}$ may be decrypted at the attacker, and
– something encrypted at the attacker may be decrypted at $\mathsf{Bcp2}_{ij}$.

The first of these error messages signals that the encrypted part of the first message may be decrypted at Scp1 but that the server expected something that was not encrypted at $\mathsf{Acp1}_{ij}$. This may happen if the responders name, j, in first message is substituted by the name of another principal, say j', by the attacker and is possible in the modified WMF because the responders name is not encrypted. Next, in the second message, the server will forward the session key to the principal j' and consequently the thirds message may successfully be decrypted at j' i.e. at a wrong responder. This turns up as the second class of error message above.

There is a similar kind of attack, which allows the attacker to substitute his own name for the responders name in the first message and, consequently allows him to interact with the protocol as illustrated by the two last error messages.

Presenting these error messages to the developer allows him to pinpoint the precise places in the UML model where encryption fail to preserve authentication as indented. Of course, repairing the protocol on the basis of this information requires creativity on the part of the developer but the For-LySa prototype allows him to quickly validate whether modifications have the desired effect.

5 Conclusion

An overall aim of the work presented in this paper is to provide software developers with a high-level interface to formal analysis tools. The For-LySa framework specifically concentrates on using UML as the interface for developers of applications that contain secure network communication.

With the work presented here, we have reached a first milestone toward this aim — and with a positive result. We have provided the For-LySa UML profiles and illustrated how these may be used to model applications in our target domain. Furthermore, we are able to perform automated extraction and analysis, using the For-LySa prototype, thereby allowing developers to perform analysis of their UML models with no particular effort on their part.

Before the For-LySa framework can be tested extensively in the field, the loop of Figure 1 must be closed, to provide the relevant feed-back to the designer with a reflector. The problem is to find a convenient way to represent the illegal decryptions revealed by the analysis. This should be relatively straightforward except for the rather cumbersome task of automatically adding the error messages to the UML diagrams in a visually appealing manner.

5.1 Related Work

The overall aim of our work somewhat similar to the aim of frameworks such as Casper [11], CAPSL [6], CVS [7], and AVISS [3]. These frameworks all aim

at providing developers of *security protocols* with high-level interfaces for formal analysis tools but unlike our approach they are based on ad-hoc notation, which describes protocols in an "$A \rightarrow B : message$"-style. On one hand, this may lead to more compact description of protocols than our but on the other hand we have all the advantages of using a general purpose modelling language.

On the technical side, the information found in protocol descriptions in the above frameworks is quite similar to the information captured in our message sequence diagrams. Also, in the extraction we find similarities, in particular with [11] that also has a target analysis formalism using a process calculus. The extraction made in [11] is, however, somewhat simpler than ours because its high-level language is designed so that is directly includes process calculi expression at convenient places.

An important effort that shares the DEGAS focus on the UML is centred on UMLsec [8]. This is a UML profile to express security-relevant information within the diagrams in a system specification, and on the related approach to secure system development [9]. UMLsec allows the designer to express recurring security requirements, like fair exchange, secrecy/confidentiality, secure information flow, secure communication link. Rules are given to validate a model against included security requirements, based on a formal semantics for the used fragment of UML, with a formal notion of adversary. This semantic base permits in principle to check whether the constraints associated with the UML stereotypes are fulfilled in a given specification. Work is ongoing to provide automatic analysis support, with an approach similar to that of DEGAS: [9] proposes to express protocols with sequence diagrams, translate them in first-order logic and then exploit standard theorem-provers, like e-SETHEO, to reveal potential attacks: [10]. The results of the analysis can be used to produce an attack scenario. In our opinion, For-LySa provides a more intuitive way to express authentication requirements that are less central in UMLsec: it should be worthwhile to assess the feasibility of the integration of the two approaches.

References

1. The Java edition of the Pepa workbench. Website hosted by School of Informatics, University of Edinburgh: http://homepages.inf.ed.ac.uk/s9905941/jPEPA/, May 2004.
2. LySa – a process calculus. http://www.imm.dtu.dk/cs_LySa, May 2004. Website hosted by Informatics and Mathematical Modelling, Technical University of Denmark.
3. A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron. The AVISS security protocol analysis tool. In *CAV 2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 349–353. Springer Verlag, 2002.
4. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *Proceedings of the 16th Computer Security Foundations Workshop (CSFW 2003)*, pages 126–140. IEEE Computer Society Press, 2003.

5. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, pages 18–36, 1990.
6. G. Denker, J. Millen, and H. Rueß. The CAPSL integrated protocol environment. Technical Report SRI-CLS-2000-02, SRI International, 2000.
7. A. Durante, R. Focardi, and R. Gorrieri. A compiler for analyzing cryptographic protocols using noninterference. *ACM Transactions on Software Engineering and Methodology*, 9(4):488–528, 2000.
8. J. Jürjens. UMLsec: Extending UML for secure systems development. In *UML 2002 – The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 412–425, 2002.
9. J. Jürjens. *Secure Systems Development with UML*. Springer Verlag, 2004. To appear.
10. J. Jürjens and T. A. Kuhn. Automated theorem proving for cryptograpich protocols with automatic attack generation, 2004. Personal Communication.
11. G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, 1998.
12. R. Milner, J. Parrow, and D. Walker. A calculus of Mobile processes (I and II). *Information and Computation*, 100(1):1–77, 1992.