

University of Stuttgart
Faculty of Computer Science

Course of Study: Computer Science

Examiner: Prof. Dr. Kurt Rothermel

Supervisor: Dr. Cora Burger

Commenced: October 1st, 2002

Completed: March 31st, 2003

CR-Classification: D.2.4, I.6.7, K.3.1

Diploma Thesis No. 2033

A Collaborative Environment for Learning Security Protocols

Matthias Papesch

Institute of Parallel and
Distributed High Performance Systems
University of Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Abstract

Learners who become acquainted with security protocols have to achieve a proper understanding. To this end an environment is designed to support them on their way. It supports them with visualizations and allows extensive experiments and even offers distributed simulations. To enable learners to judge their performance, the solutions they come up with are checked by [SPIN](#), a well-known model checking tool.

Contents

Contents	1
List of Figures	5
List of Tables	6
Listings	7
1 Introduction	8
1.1 Motivation	8
1.2 Description	8
1.3 Overview	9
2 Scenarios	10
2.1 Thematic Overview	10
2.1.1 Imaginary Use Case	11
2.1.2 Protocols in General	11
2.1.3 Security Protocols in Particular	12
2.1.4 Teaching Goals	13
2.2 Methodical Process	15
2.2.1 Common Approaches	15
2.2.2 Conclusion	16
2.2.3 Teaching Scenarios	17



3	Requirements	18
3.1	Instructive Scenario	18
3.2	Interactive Scenario	19
3.3	Explorative Scenario	20
3.4	Collaborative Scenario	21
4	Existing Support	22
4.1	Exploration Tools	22
4.1.1	PROMELA	23
4.1.2	(X)SPIN	24
4.1.3	HiSAP	26
4.1.4	An alternative PROMELA Interpreter (HiSPIN)	27
4.2	Application Sharing	29
4.2.1	NUSS	29
4.2.2	VNC	30
4.2.3	JSDT	30
4.2.4	JMS	31
4.2.5	JavaSpaces	31
4.3	Conclusion	32
5	PROMELA and Security Protocols	34
5.1	Entities	34
5.2	Data Templates and Processing of Messages	34
5.3	Special Validation and Simulation Demands	37
5.4	Intruder Models	37
5.5	The Notary	38
5.6	Deriving an Example	39



6	Design	43
6.1	Instructive Simulations	43
6.1.1	Single Sequence	43
6.1.2	Animation	44
6.1.3	String Messages	46
6.1.4	Encryption	46
6.1.5	Data Access	47
6.2	Interactive Simulations	48
6.2.1	Multiple Sequences	48
6.3	Explorative Simulations	48
6.3.1	Act on Behalf	49
6.3.2	Access Control	49
6.3.3	Replace	49
6.3.4	Self Evaluation	50
6.4	Collaborative Simulations	50
6.4.1	Application Sharing	50
6.4.2	Roles and Permissions	51
6.5	Resulting Architecture	52
7	Implementation	53
7.1	Instructive & Interactive Simulation	53
7.1.1	Single & Multi Sequence	53
7.1.2	Animation	54
7.1.3	Encryption	58
7.2	Explorative Simulation	58
7.2.1	Replace	60
7.2.2	Self Evaluation	60
7.3	Collaborative Simulation	60
7.3.1	Application Sharing	60
7.3.2	Roles and Permissions	61



8 Conclusion & Outlook	63
8.1 Conclusion	63
8.2 Outlook	63
Bibliography	65
A PROMELA Examples	67
A.1 A Common Ground	67
A.2 The Exam Protocol	69



List of Figures

2.1	The message exchange of the EXAM protocol.	14
2.2	The level of student interactivity increases steadily, while the instructor fades out.	17
6.1	The old vs. the new creation of a HiSAP node.	45
6.2	An overview of the ProDuctivE Architecture.	52
7.1	The new HiSAP hierarchy.	57



List of Tables

4.1	The rating scheme for the compliance with the requirements.	22
4.2	An overview of the evaluation of the tools against the requirements. . . .	33
5.1	The outcome determination according to the announcement messages. . . .	39
6.1	Additional cryptographic tokens for the scanner.	47
7.1	An overview of the interpreter event model, which allows generic adaption.	55
7.2	An overview of the HiSAP event model.	56
7.3	The basic permissions for security protocols.	62



Listings

5.1	Message macros (productive.prl).	35
5.2	Alice sends and receives messages (exam.prl).	36
5.3	The Notary outcome and never-claim for the generation of trails.(exam.prl).	39
5.4	The declaration of the REQUEST_SESSION message (exam.prl).	39
5.5	Entity data for entities of the Exam protocol.	40
5.6	Session data for a session the EXAM protocol (exam.prl).	40
5.7	Alice sending a message (exam.prl).	41
5.8	Bob receiving a message (exam.prl).	41
7.1	The basic Undo-Support.	54
7.2	How to replace the body of a proctype (exam.prl).	59
A.1	The include file with shared definitions (productive.prl).	67
A.2	The complete EXAM protocol (exam.prl).	69



Chapter 1

Introduction

Security protocols are meant to protect people in an electronic world, e.g. their privacy, identity, autonomy, and resources. The big challenge behind consists in performing this task in front of hackers where the form of attack is hard to foresee. Consequently, it is pretty difficult even for experts to design security protocols which really meet their requirements.

1.1 Motivation

A slightly different situation appears for learners, who become acquainted with the topic and have to achieve a proper understanding of how security protocols work, of possible flaws and their corrections. With a teacher at hand for instruction and support, this is hard enough. The availability of teachers being restricted, learners intermittently have to rely on private studies and discussions with others. Hence, three different scenarios have to be coped with: lectures, single learners and teams without a tutor. For all of them an environment is wanted which supports teaching and learning by means of visualization, extensive experiments and investigations either by single persons or by entire teams.

A concrete example for the application of such an environment is the lecture *Distributed Systems*, taught at the University of Stuttgart. This course covers, among other things, security protocols.

1.2 Description

An environment to support a group of learners when confronted with security protocols is designed. The requirements for such an environment are identified by analyzing application



scenarios. Existing tools, e.g. the model checker (X)SPIN, are examined whether they fit these scenarios. A prototypical implementation is created under the name ProDuctivE – a PROMela Driven Constructivist Environment.

1.3 Overview

This thesis is organized as follows. Chapter 2 introduces the scenarios for which the environment is to be designed. In Chap. 3, the requirements are derived from those scenarios. How existing support can be used is covered in Chap. 4. In Chap. 5, a way to use PROMELA for teaching security protocols is developed. Chapter 6 shows how to approach the solutions. Chapter 7 comments on the prototypical implementation. A conclusion and an outlook for future work closes the thesis in Chap. 8.



Chapter 2

Scenarios

The task at hand is to design an environment which supports the teaching of security protocols. The examination of specific scenarios for which the environment might be used reveals a clear picture of the functionality it should provide. According to the very first perception, there are two quite unrelated points of view from which this task needs to be regarded. From each of these points, a different part of the scenario is seen.

The thematic part covers what students should learn and the abilities they are to acquire. Obviously, security protocols make up the center of this part. As the entire spectrum ranges from very basic aspects to rather specific details, the subset which the environment should cover needs to be determined. This leads to the specification of teaching goals in Sect. 2.1.4.

The methodical part, on the other hand, is concerned with how the involved peers might accomplish these goals. The complexity of the subject suggests to divide the entire learning progress into smaller didactical units. An incremental level of student interactivity yields the teaching scenarios summarized in Sect. 2.2.3.

2.1 Thematic Overview

Security protocols aim to protect communication among several participants. As these participants are not always people, they will also be referred to as protocol entities or simply entities. General purpose protocols use a great variety of arrangements, e.g. point-to-point, rings or busses. The vast majority of security protocols, on the other hand, share the same fixed set of entities. Sometimes, these entities are referred to by the role they play in a protocol session. These terms are Initiator, Responder, Intruder and Trusted Server. However, there is an alternative naming tradition in literature (cf. [Tan96]), which



introduces some kind of a human touch. Therefore, the scheme shown below is adopted for this thesis.

<i>Alice</i>	tries to initiate secure communication with Bob.
<i>Bob</i>	responds to Alice's wish to communicate.
<i>Eve</i>	a malicious intruder who wants to disrupt the communication.
<i>Notary</i>	a trusted party. Some security protocols assume a trusted server for various reasons, some do not. It is the teaching scenario which makes this entity mandatory, as Sect. 5.5 will show.

Another term which will be used quite frequently is a protocol session, or just a session. It refers to the entities exchanging the messages defined for the protocol once, from first to last. If an error occurs, a session might come to a halt before all messages have been processed. Such a session is also considered complete.

2.1.1 Imaginary Use Case

Since the discussion of any protocol is almost always rather abstract, a concrete use case helps to illustrate the problem. The one introduced below will give a reason why an insecure means of communication needs to be secured. It also serves as a thread which will be picked up when a new situation is explained.

Let the protocol entities Alice, Bob and Eve represent a group of three colleagues working for the same company in different cities. Alice gets along fairly well with Bob, but she does not really fancy Eve. By the time Alice's birthday gets near, she thinks of having a party. While she would like Bob to come, she does not want Eve to show up. So, she decides to send an e-mail to Bob, which contains an invitation with the date and the location of the party. Of course, Eve is not supposed to come to know the details of the invitation.

2.1.2 Protocols in General

When Alice uses mail user agent (MUA), she automatically acts according to a protocol. Notice that the example will examine the layer the MUA represents, which will eventually make use of the simple message transfer protocol (SMTP).

Any protocol is described by a set of properties, which is defined in [Hol91] and shown below. It will be assumed that the targeted audience is already comfortable with protocols.



<i>Service</i>	provided by the protocol. A MUA delivers a text messages to one or more specified recipient.
<i>Assumptions</i>	about the environment in which the protocol is executed. All e-mail addresses are known, valid and reachable. The underlying network is reliable but insecure.
<i>Vocabulary</i>	of messages used to implement the protocol. Basically, a message consists of one e-mail address for the sender and one or more for the receiver, and and a plain text message. The MUA will take the input and transfer it to the receivers using SMTP.
<i>Encoding</i>	of the messages. Messages are encoded according to the ASCII standard.
<i>Procedure Rules</i>	guarding the consistency of message exchange. A MUA typically adheres to very simple procedure rules. For instance: Fill in the fields of the form with valid values and press send.

The assumption that the network is insecure confronts Alice with a problem. It implies that Eve might be able to mess with the e-mail intended for Bob. For some reason or other, she is afraid that Eve would literally do anything to ruin her party. Eve, for example, could try to

<i>disclose,</i>	read the invitation and show up uninvited.
<i>corrupt,</i>	send Bob an invitation to a party at a different location.
<i>impersonate,</i>	pretend to have sent the invitation herself or cancel in Bob's name.

2.1.3 Security Protocols in Particular

Security protocols always assume the network to be insecure. To cope with this insecurity, their service specification offers some or all of the additional features described in [MvV96, §1.2] and summarized below.

A certain cryptographic background is necessary to understand how these properties are achieved. The audience is presumed to be familiar with means like, for instance symmetric and asymmetric encryption and digital signatures.



<i>Confidentiality</i>	No third party can read the content of the messages. Even if Eve captured the e-mail addressed to Bob, she could not extract the information when and where the party will be held.
<i>Data Integrity</i>	Each party can be sure that the message they receive has not been altered by a third party. Bob can be certain, that the text of the invitation he receives is exactly the same text Alice wrote.
<i>Authentication</i>	Each party can be sure that the other party is who they claim to be and the origin of the message can be determined and verified. The message Bob receives definitely originates from Alice.
<i>Non-repudiation</i>	Mentioned for the sake of completeness. It refers to invalidating a session which has been corrupted. Bob will turn down any invitation, as soon as it turns out to be a fake.

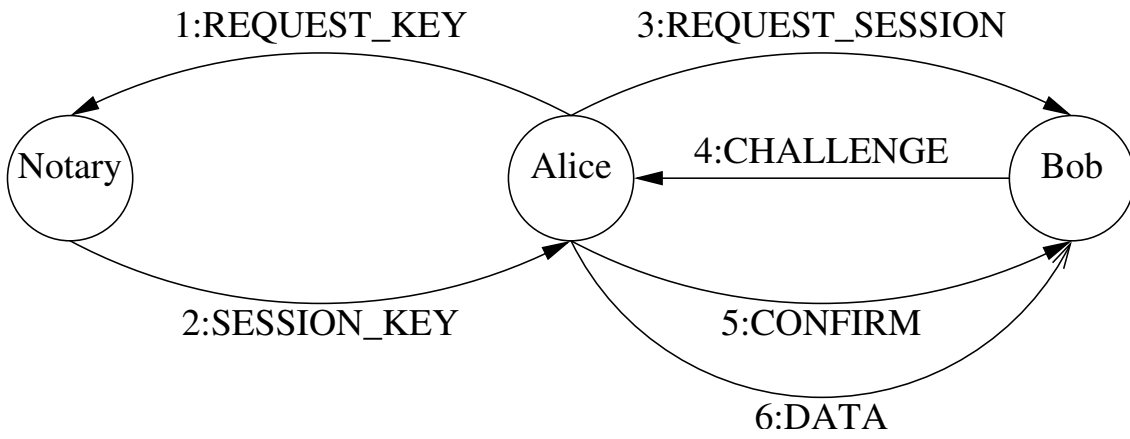
To be on the safe side, Alice definitely needs to use a security protocol which provides at least the first three of the above features.

2.1.4 Teaching Goals

It is important to be able to judge whether a certain protocol really fulfills its service specification. One practical application of this ability is to recognize or perform an attack on a specific protocol. Such attacks typically exploit design flaws. From the example in the previous section, the following goals can be derived. After attending the lecture and exercises students should be able to

1. Judge whether a protocol is secure by looking at the message exchange table.
2. Determine at which point in the message exchange one party is authenticated to the other and when privacy is established.
3. Understand how the most common attacks profit from flaws in protocols.
4. Determine by the message exchange what kind of attack can succeed.
5. Successfully attack a flawed protocol.
6. Fix a flawed protocol.





Message	Sender	Receiver	Content
<i>REQUEST_KEY</i>	Alice	Notary	A, B, R_A
<i>SESSION_KEY</i>	Notary	Alice	$\{R_A, A, K_{AB}, \{K_{AB}, A\}^{K_{NB}}\}^{K_{NA}}$
<i>REQUEST_SESSION</i>	Alice	Bob	$\{K_{AB}, A\}^{K_{NB}}$
<i>CHALLENGE</i>	Bob	Alice	$\{R_B, B\}^{K_{AB}}$
<i>CONFIRM</i>	Alice	Bob	$\{R_B - 1\}^{K_{AB}}$
<i>DATA</i>	Alice	Bob	D_A

Legend	
$\{X\}^K$	X encrypted with key K
R_X	nonce value generated by X
K_{XY}	shared secret among X and Y
D_X	data value generated by X

Figure 2.1: The message exchange of the EXAM protocol.

All points can be achieved through experience and feedback. When learners have the possibility to experiment with several protocols and various message sequences, they learn to estimate the consequences. If they are enabled to perform an attack themselves they will develop a feeling for the crucial messages. Goals 5 and 6 depend on the other ones. As they can also be verified they qualify for evaluation purposes.

An example, like the EXAM protocol shown in Fig. 2.1, illustrates the application of these demands. The name originates from serving as an example here and a similar version having been used in a real examination. The interesting questions with regard to this thesis were:

Question 1: The protocol is not correct and can be attacked successfully. Describe the scenario of an attack.

Answer 1: In a session, Alice will be talking to Eve, who is impersonating Bob. This will happen, when Eve corrupts the *REQUEST_KEY* message,



which will result in $SESSION_KEY_{fake}$. Alice will unknowingly relay $REQUEST_SESSION_{fake}$. The content of these messages will read

Message	Content
$REQUEST_KEY_{fake}$	A, E, R_A
$SESSION_KEY_{fake}$	$\{R_A, A, K_{AE}, \{K_{AE}, A\}^{K_{NE}}\}^{K_{NA}}$
$REQUEST_SESSION_{fake}$	$\{K_{AE}, A\}^{K_{NE}}$

Question 2: How must the protocol be modified to avoid those attacks?

Answer 2: By encrypting $REQUEST_KEY$ with the shared secret, it cannot be corrupted and $REQUEST_SESSION$ will be intended for the correct recipient.

Message	Content
$REQUEST_KEY_{safe}$	$\{A, B, R_A\}^{K_{NA}}$

Question 3: After which message is Alice authenticated against Bob, respectively. Bob authenticated against Alice in the corrected variant?

Answer 3: Alice knows she is talking to Bob for sure after receiving $CHALLENGE$ which was encrypted with K_{AB} , which only Bob could extract from $REQUEST_SESSION$.

Bob, on the other hand can be sure after receiving $CONFIRM$, which tells him that Alice has been able to decrypt $CHALLENGE$.

2.2 Methodical Process

This section investigates the most suitable approach for learners to acquire the skills determined in the previous section. The result, a customized approach is presented thereafter.

2.2.1 Common Approaches

In the last century and a half, quite a few scientists engaged in research on how humans learn. They developed several models which base on different assumptions and match different requirements. A very extensive review on this topic is given in [Mer98] which is summarized in short below.

Behaviorism regards the learner as a black box. Progress is determined by the observed behavior in response to specific stimuli. The probands are conditioned and reinforced to react properly in specific situations but do



not necessarily understand the context in which they act. Thus, a minor change in the stimuli might cause major confusion. Behaviorism is well-suited for tasks which must be executed in a certain manner.

Cognitivism

states, that behavior can also be learned without being explicitly reinforced. A learning person associates new information with previously gained experiences. The learning process benefits from additional knowledge about the new matter. For example, memorizing a piece of text is easier, if the learner is familiar with the language it is written in.

Constructivism

states that knowledge is not objective, but individually customized. The experiences gained before influence the way new information is being processed. Since each and every individual has a different background, each will *construct* a different mental image of their perception. Interaction with the environment and discussion with fellow learners lead to a more differentiated image. However, given the same task, two students may come up with completely unrelated results.

2.2.2 Conclusion

For teaching security protocols, a behavioralistic approach is not at all suited. If such an approach was applied, learners would focus on one specific protocol at a time and not on the class of security protocols in general.

A cognitive approach promises better results. Students work through predefined lessons. Whoever prepares these lessons may prioritize one aspect or another. The possibilities for students to interact are limited and only possible where they have been envisioned during the preparation.

Constructivist elements allow the students to interact with the predefined material. Individual students may experiment autonomically with several protocols. They are given extensive material and determine what they want to focus on themselves. The standard way to resolve a problem is to engage in a discussion with fellow learners working on similar problems. Alternatively, a group of students is given a single task, which they work on as a group. Such procedures enhance the perspectives for understanding the correlations and similarities among protocols. By definition, a constructivist approach lacks a precisely defined goal. This severely contradicts the scenario given in Sect. 2.1.4 which clearly states the abilities students are to acquire.

Since teaching of security protocols has clear goals and includes creative elements, a mixture of different approaches seems to be appropriate.



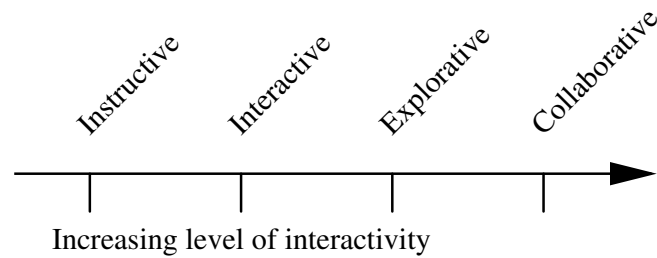


Figure 2.2: The level of student interactivity increases steadily, while the instructor fades out.

2.2.3 Teaching Scenarios

The envisioned mixture can be implemented best, if composed of didactical units. For now, these units exist independent of the idea of having an environment to support them.

Instructive. The learner studies the presentation material or attends a presentation on the subject. The typical material consists of textual descriptions or slide shows. It contains a lot of explanation and hardly any interaction. The possibility to add annotations to the material, which will be available when the material is viewed again is useful. In case of a problem, a tutor is contacted.

Interactive. To gently introduce the idea of interactivity and self-evaluation, questions, hints and answers are embedded within the presentation. For instance, questions might concern the consequences of applying encryption for a certain message field.

Explorative. The students are given small tasks to work on by themselves to test how well they understood the topic. A protocol with a rather obvious flaw might be part of such a task. When a problem arises, fellow learners or a tutor can be contacted. Such a tutor may help along with his expertise. More experienced students should be given the means to evaluate their performance themselves.

Collaborative. A group of students is assigned a task, which requires creative steps, e.g. finding a complicated failure which had remained undiscovered for some time. The students have to figure out how to solve the problem among themselves. A tutor will not help them explicitly, but temporarily join the group as a regular member among equals.



Chapter 3

Requirements

The environment should support the instructive, interactive, explorative and collaborative scenarios identified in Sect. 2.2.3. Since the sequence of scenarios is incremental with regard to increasing interactivity, the requirements are described as follows. Each scenario provides the functionality of the previous one as well as additional features. Another requirement is that the environment should be applicable on as many platforms as possible.

In accordance with the specification of this thesis, *Annotations*, *Logging and Replay* and *Extra Communication* are not intended to be realized within this thesis.

3.1 Instructive Scenario

The instructive scenario is the most basic one. It assumes a single learner who wants to become acquainted with security protocols in general or a specific security protocol in particular. The functionality should also be suited for an instructor to hold a presentation on a certain protocol.

Single Sequence. A learner being confronted with security protocols for the first time needs some guidance. The learner should be able to just explore one or a few pre-defined examples of protocol behavior. This holds as well for a teacher with a tight schedule.

Animation. A learner should be able to use the environment to view animated presentations. These animations should allow various views on the message exchange of the given protocol, e.g. a topological view and a message sequence chart.



Easy Generation. It is useful to create a repository of security protocols. The more protocols a learner can study the better. Thus, both the animation and the sequences should be easy to generate with low effort.

String Messages. To be meaningful to the learner, messages should carry user-defined strings.

Encryption. Security protocols rely on encryption. Encrypted messages should reflect this and actually show different values.

Annotations. When the instructor prepares the material, he should be able to asynchronously attach annotations. These will either show up at certain steps or they can be asked for. When viewing the animation, a learner should also be able to do so synchronously for the running animation. There should also be a way to distinguish between public and private annotations. Thus, annotations must include the identity of the poster. Whenever an annotation is no longer needed, the learner should be able to remove it.

Logging and Replay.

A learner should be able to review a session, e.g. when preparing for an examination. Thus, the actions and annotations need to be logged to a persistent storage. When a student wants to view the animation again, the annotations should appear for the same time span. Also, the learner should be able to choose whose public annotations should be included.

Data Access The purpose of an instructive simulation is to become acquainted with the protocol functionality. The learner should be able to view any existing data being involved.

3.2 Interactive Scenario

Having studied the prepared material, the learner should move on and show the first signs of activity. Being somewhat familiar with a specific protocol, e.g. the EXAM protocol, the learner should be asked to compose a an execution sequence of pre-defined steps on-the-fly. The interactive scenario should no longer enforce the restriction to prepared, entire protocol sessions. The chosen steps might not even result in a protocol state, which represents a successful session. The interactive scenario resembles a multiple choice test giving the answers and querying the correct combination.



Multiple Sequences.

The more experienced learner should also be able to try out several execution sequences. This requires the functionality to go back any number of steps at any time during the simulation. The steps each protocol entity may execute remain pre-defined. However, the sequence in which they are concatenated is determined by the learner interactively.

3.3 Explorative Scenario

The explorative scenario extends the degree of interactivity even further. The learner should take over the complete control over a protocol entity. The restriction to pre-defined steps is dropped. Instead, the learner should define the steps on his own, similar to a free-text assignment.

Act on Behalf.

This is similar to the interactive scenario. However, the learner may act on behalf of a protocol entity. This means, that the sending and receiving of any message may be triggered at any time. The learner may also change the value of internal variables. This allows the student to explore how the other entities react to unexpected messages.

For instance, the learner might participate in an EXAM session acting on behalf of Eve. He might run a regular initiator session or an impersonation session. He might even finish the first one, go back and try the second one.

Access Control.

When entity data can be manipulated, some access to private data must be restricted. The learner should only be allowed to access data belonging to the entity on whose behalf it acts. Otherwise, the protocol functionality might be bypassed.

Replace.

Replacing an entity should ask the learner to define the entire sequence of actions to be executed in advance. After that, the animation should be performed in an interactive manner. This requires some simple form of input.

The learner might replace Eve with a self-created impersonating definition. The subsequent interactive run shows whether the description yields the desired result.

Self Evaluation.

The learner should be able to evaluate the correctness of his solution. For security protocols, this should be based on the end state of the



protocol indicated in Sect. 2.1.2, *impersonate*, *corrupt*, *disclose*. The environment should guide the learner when identifying and correcting flaws.

3.4 Collaborative Scenario

Finally, the collaborative scenario aims at advanced learners who are pretty confident with security protocols in general. A group of such learners should be given a problem with a particular protocol and the task to solve it together.

Application Sharing.

A group of students at different computers, and possibly at different locations, should be able to act in the same session simultaneously. All types of sessions, instructive, interactive and explorative should be supported.

Roles and Permissions.

Each student should control a different entity. Thus, access control needs to handle multiple roles with different permissions.

Extra Communication.

Since multiple participants at different locations should be able to synchronize their behavior, an additional communication facility is needed. This facility should also allow subgroups to communicate.

Synchronous Annotations.

If desired, annotations should become visible to all participants of a session as soon as they are attached.

Privacy.

Each member of a group should be able to retreat from the shared view and examine a problem alone. When done, he should be able to catch up with the others.

Join.

A learner should be able to join an ongoing session.

Scalability.

The distribution should not waste resources.

WAN.

The environment should also work over a wide area network (WAN), which might be restricted by firewalls. It is assumed, that this can be overcome by setting up a Virtual Private Network (VPN), either using a commercial solution or a simple one using a secure shell, like openSSH [[ssh](#)].



Chapter 4

Existing Support

This chapter examines tools and concepts, which might prove useful to meet the requirements identified in Chap. 3. In the following sections, each tool is presented with a short summary. A list of requirements which the tool provides succeeds this summary. Table 4.1 shows the scheme according to which the provided solution is rated.

Symbol	Meaning
++	very well suited, very beneficial
+	well suited, beneficial
o	neither beneficial nor disturbing, indifferent
-	poorly suited, disturbing
--	very poorly suited, very disturbing

Table 4.1: The rating scheme for the compliance with the requirements.

There are two separate areas to be support. Exploration tools focus on protocols and their handling. Application sharing offers the possibility for multiple people to work on the same data and coordinates such work.

4.1 Exploration Tools

Several tools have yet been evaluated with regard to how well they are qualified for teaching protocols in [Sch02]. The conclusion was that model checking in general and the tool **SPIN** in particular are suited for such an undertaking. This result influenced the selection of tools in the exploration area.



4.1.1 PROMELA

The process meta language (PROMELA) has been designed to describe protocols of any kind. Although it is syntactically similar to C, the two languages differ a lot. The following section will give a very brief introduction on PROMELA. For a more detailed description, see [Hol91] and [pr197]. Appendix A.2 gives an example PROMELA specification for the EXAM protocol.

A specification is the PROMELA equivalent to a C program. It consists of global variable declarations and one or more process type definitions. As the keyword `proctype` marks the beginning of a process type definition, both terms are used interchangeably. The concept of a process is similar to that of a procedure. It comprises statements and local variable declarations, which describe the behavior and the state of a protocol entity. Once instantiated, a process may access all locally defined variables as well as the global ones. However, it does not return a value and the invoking process does not block until the process has finished. Thus, an arbitrary number of processes may be running at the same time. Each single statement is, and a sequence can be defined to be, executed as an atomic unit.

A PROMELA specification describes non deterministic behavior. If multiple statements are executable, any one of them may be executed. The primary cause for multiple options is due to multiple running processes. Additional sources of non determinism are compound statements, which may offer multiple options within a single process.

In a specification, at least one `proctype` must be marked active. This means, that an instance of the `proctype` will be created automatically when the specification is interpreted. Once a process is started it will execute the statements from its statement sequence.

PROMELA supports three data types: integer values of various sizes, channels and compounds. Strings and floating point data types have been omitted from PROMELA since they were considered obsolete for specifying protocols. Channels are a concept to support communication among processes. The declaration of a channel includes the number of messages it can store and a list of data types which make up a message. To communicate, one process writes a list of values to a channel, which stores them temporarily until another process will pick them up. A compound type simply comprises an arbitrary number of other data types.

Correctness requirements are a feature qualifying PROMELA for model checking. When such a requirement is not met, the protocol is not considered correct. There are several ways to define correctness requirements. Never claims are the most powerful and flexible. Such a never claim defines a set of illegal execution sequences. It should be noted, that this definition lies in the eye of the beholder. In other words, if one wants to see an execution sequence which actually represents a correct protocol session, one just needs to specify



correctness criteria which mark this case to represent an error. This is especially useful when processing specifications with the tools described in the following sections.

Evaluation

<i>Easy Generation.</i>	When familiar with PROMELA, it is rather easy to generate protocol specifications with it. However, PROMELA in general is quite special and never-claims in particular might be confusing.	(+)
<i>String Messages.</i>	No support for string messages is provided.	(--)
<i>Encryption</i>	No support for encryption is provided.	(--)
<i>Annotations.</i>	Primitive annotations can be achieved through print statements and comments.	(+)
<i>Data Access.</i>	From a global view, it is possible to read all variables, including locally defined ones.	(++)
<i>Access Control.</i>	From a local view, variables declared within a different proctype are not visible. Thus, each process is restricted to access process-private and global variables only.	(++)
<i>Replace.</i>	A protocol entity will act according to its process definition. To change the behavior of an entity, its definition must be replaced.	(+)

4.1.2 (X)SPIN

The simple PROMELA interpreter ([SPIN](#)) [[Hol91](#), [Hol97](#)] is a model checker, which accepts PROMELA specifications as input. Model checking is used to verify that a given protocol specification fulfills the correctness requirements. The model checking software reads the specification and transforms it into a finite state machine (FSM). [SPIN](#) offers two different ways to work with the resulting FSM.

[SPIN](#) allows PROMELA specifications to use C pre-processor directives. Before the actual simulation starts, these are expanded by invoking the C pre-processor on the input file.

Simulation. [SPIN](#) interprets the specification and executes a single sequence. One important application of the simulation mode is the simple inspection of protocol behavior. Another one is to trace erroneous behavior. To identify a flaw, a so-called trail file will lead a simulation run along the violating path.



Validation.

The correctness requirements of a specification describe conditions which must hold for each state. The specification is considered correct, if the entire state space has been visited, and no violations were encountered. Of course, the more complex a specification, the larger is the state space. Various methods exist to limit the state space according to the available resources. Security protocols, however, tend to be quite simple and thus exhaustive searches are possible.

Performing a validation is a three step process:

1. Have [SPIN](#) create a C-source file for a protocol analyzer (pan.c).
2. Compile the pan.c source file with a C compiler.
3. Execute the resulting stand-alone program (pan).

Since validating a specification with a large state space is quite time-consuming, having a stand-alone program is a sound choice. When the validation encounters a correctness violation, it produces a trail file.

[SPIN](#) itself is a command line tool available on many platforms. The user interface suffers from the restriction to the command line. There are no menus, just command line parameters and textual output. Given the corresponding parameters, [SPIN](#) will be very verbose. “-g”, for instance, will cause all global variables to be printed for each step. “-i”, will run an interactive simulation, which reads instructions from standard input (usually the keyboard).

To overcome this problem, [XSPIN](#), provides a comfortable graphical front end, which improves the handling a lot. [XSPIN](#) is a completely independent program written in Tcl/Tk. It allows the user to edit a specification and to specify command line parameters through menus and dialogs. To perform a simulation or validation it invokes [SPIN](#), the C compiler and the protocol analyzer as separate processes. If it is necessary, e.g. for an interactive simulation, in- and output are redirected.

With regard to teaching, the possibility to validate a specification is a big advantage of a model checker. An instructor can be quite sure, that the material is correct. Even if an error should slip in, it would be relatively easy to find and fix. It also allows a learner to check whether a assumed solution is correct.

[SPIN](#) is written in C and can be compiled for any Unix/Linux as well as Microsoft Windows operating systems. [XSPIN](#) requires Tcl/Tk interpreters, which are also available for the above mentioned platforms. However, to get [\(X\)SPIN](#) running for the first time, the binary executable has to be installed which might not be a trivial task. It becomes even trickier on a system which is controlled by an administrator. This is even harder for the Tcl/Tk interpreters.



Evaluation

- Single Sequence.* A single sequence scenario can be achieved through trail files. However, the possibility to move within this sequence is not available. (+)
- Animation.* SPIN does not provide any animation. However, it does provide output, from which XSPIN produces message sequence charts and finite state machines. (+)
- Easy Generation.* Trail files for specific execution sequences are easy to generate. (++)
- Annotations.* Primitive asynchronous annotations can be achieved at specification time through print statements. (+)
- Data Access.* Values of variables can be traced. (++)
- Multiple Sequences.* SPIN offers an interactive mode, where the statements to be executed can be selected by the user. However, it is not possible to move backwards in the sequence. Thus, to examine a different execution sequence, one needs to start all over again. (-)
- Act on Behalf.* It is not possible, to interactively change any data values. (--)
- Access Control.* XSPIN allows to display selected values only. However, it is the momentary user who makes this selection and not the one who prepared the specification. A predefined restriction is not possible. (-)
- Self-evaluation.* Validation against correctness requirements provides a powerful means to determine whether a solution is correct. (++)

4.1.3 HiSAP

A different approach to describe protocols is the HiSAP toolkit [Bro99], which supports the implementation of protocols as JAVA applets. The toolkit focuses on the visualization part. Thus, it is quite easy to demonstrate how a certain protocol works, once the rather high effort of implementing the functionality has been mastered. There is no guarantee that this has been done correctly.

The key concepts of the HiSAP toolkit are nodes, connections, messages and strategies.

- Node.* A node represents a protocol entity. It has access to an arbitrary number of connections over which it can send and receive messages.



- Connection.* A connection has exactly two ends. At each end, there is exactly one node attached. These two nodes can exchange messages over the connection.
- Message.* A message encapsulates an arbitrary piece of data, which is represented by a string.
- Strategy.* A strategy provides the protocol implementation for a node. It handles incoming messages and produces outgoing ones.

HiSAP is written entirely in JAVA. It couples the implementation of a visualization with the implementation of a protocol. If one wants to implement a specific protocol, one must be familiar with JAVA, the **HiSAP** toolkit and the protocol. This overhead remains the same, even when the selection is limited to security protocols.

Evaluation

- Single Sequence.* **HiSAP** does not explicitly support the possibility to execute a specific sequence. Neither does it provide the means to move back in a sequence. It takes reasonable effort to add an undo-/redo functionality which provides this requirement. (–)
- Animation.* **HiSAP** offers very nice animations. (++)
- Easy Generation.* It demands a rather high effort to implement the protocol functionality. (–)
- String Messages.* The default message type is a string. (++)
- Annotations.* **HiSAP** does not provide an annotation functionality, but the annotations may be attached to the visual components. (◦)
- Data Access* **HiSAP** focuses on visualizing the message exchange, and not on displaying the state of protocol variables. (–)
- Multiple Sequences.*
cf. *Single Sequence.* (–)

4.1.4 An alternative PROMELA Interpreter (HiSPIN)

The PROMELA interpreter developed in [Pap02] is written in JAVA and allows other JAVA programs running in the same virtual machine to access simulation data. The original



name for this interpreter, HiSPIN, has been dropped and a new one has not yet been assigned. Thus, it will be referenced as interpreter or simulator, and occasionally, as HiSPIN. The simulator was anticipated to use [HiSAP](#) to visualize arbitrary protocol simulations. However, as it turned out, a generic visualization demands quite a bit of customization. When focusing on security protocols only, the set of entities and the protocol objectives remains fixed, as Sect. 2.1 shows. The same set up can be shared among many security protocols. Once this set up has been created, the implementation of further protocols is completely independent from the visualization.

The interpreter aims to be completely [SPIN](#) compatible and even accepts [SPIN](#) trail files. This fact and the usage of PROMELA, allow users who are familiar with these tools to readily use the environment. Furthermore, existing PROMELA specifications may be adapted or even inherited. So far, no pre-processor functionality is provided.

Evaluation

<i>Single Sequence.</i>	Just like SPIN , a single sequence scenario can be achieved with trail files. Moving backwards is not possible yet, but adding an undo-/redo functionality which provides this requirement is achieved with reasonable effort.	(+)
<i>Animation.</i>	It was designed to be connected with HiSAP .	(+)
<i>Easy Generation.</i>	Decouples protocol specification from visualization implementation. Each can be dealt with separately.	(++)
<i>String Messages.</i>	Support for string messages is not provided, but is achievable with reasonable effort.	(+)
<i>Encryption.</i>	Encryption functionality can be hidden with help of the C preprocessor and reasonable effort.	(+)
<i>Data Access.</i>	The simulation data are accessible at all times.	(++)
<i>Multiple Sequences.</i>	Achievable through an undo-/redo functionality.	(+)
<i>Act on behalf.</i>	The simulation control needs to provide the means for the learner to input of custom statements.	(o)
<i>Access Control.</i>	It is possible to restrict access to particular processes and global symbols.	(+)
<i>Replace.</i>	Before the simulation starts, the learners solution replaces the original proctype. Achievable with reasonable effort.	(+)



Self Evaluation. The outcome of a protocol session can already be determined. Implementing the *Replace* requirement will also provide the means to check whether a learner can provide a correct message sequence in advance. (+)

4.2 Application Sharing

When multiple users share an application, they operate on the same data. A text editor which allows multiple users to edit the very same document simultaneously is an example for a shared application. The example is already very close to the actual scenario. The resulting environment should allow multiple users to operate on a single simulation.

4.2.1 NUSS

The network university Stuttgart ([NUSS](#)) aims to provide an infrastructure for collaborative applications of distinct shape and communication usage. Its main features are application independent.

Evaluation

Annotations. [NUSS](#) offers a generic annotation facility, which will take an identification, a timestamp, a context and a message. (++)

Logging and Replay. [NUSS](#) provides a standardized logging facility, which can log arbitrary data. (++)

Application Sharing [NUSS](#) provides an access point (NAP), through which clients can perform a lookup for a server. The NAP also offers access to other shared services. (++)

Roles and Permissions. One of the other services is a group management facility, which provides an authentication and permission service for any application. (++)

Extra Communication. A service which provides means to communicate through [NUSS](#) is useful for quite many scenarios. The inclusion of a chat service and a white-board is anticipated. (+)



Synchronous Annotations.

The annotations facility supports both, asynchronous ones during preparation, as well as synchronous ones at runtime. (++)

4.2.2 VNC

Virtual Network Computing (VNC) provides a remote desktop, i.e. a user sees windows from a remote computer on the local workspace and can control them. This allows to share any application in the same manner. Several flavors exist, e.g. TightVNC [tvn] and even a multicast solution [mvm]. The latter allows to restrict who can control the application. However, application specific access control is not possible.

Evaluation*Application Sharing.*

Multiple users may share the same application over the network. (++)

Roles and Permissions.

Roles and permissions are not supported. All learners have the same view and may perform the same actions. (-)

Privacy.

Since all learners have the same view, decoupling from the others is not possible. (--)

Join.

As all users operate on the same data, a user may join any time. (++)

Scalability.

VNC supports multicast sessions. (++)

4.2.3 JSDT

The JAVA Shared Data Toolkit (JSDT) [jsd] supports, among others, the lightweight reliable multicast protocol (LRMP). The main concepts of JSDT are sessions, channels and consumers. Channels exist within a session and are read from or written to by consumers. JSDT comes with its own authentication scheme, to determine the privileges of each consumer. While the authentication functionality is not really needed it may cause some overhead, as each action will be checked anyways. Some more programming effort arises from the fact, that JSDT only supports `byte[]` data. Although, any serializable object can be represented this way, implementing the communication is not straightforward.

Roles and Permissions.

JSDT comes with its own authentication mechanism. (+)



- Join.* JSDT allows users to join any time. However, the history is not available. (+)
- Scalability.* If the network supports multicasting, LRMP can be used. (+)

4.2.4 JMS

A publish-subscribe system like, the JAVA Messaging System (JMS) [MH01] which is a JAVA front-end to other messaging system, offer less functionality than JSDT, as they do not provide sophisticated authentication models. Typically, the check is performed once, when subscribing to the channel. Also, the number of producers is limited to one, which is sufficient for the purpose. However, JMS is only a front-end and depends on a actual messaging system implementation.

- Join.* JMS allows users to join any time. However, the history is not available. (+)
- Scalability.* Depends on the back-end implementation, which is typically designed to scale well. (+)

4.2.5 JavaSpaces

Entries can be written to a JavaSpaces service, which will store a copy of the entry. Stored entries can be looked up matching *templates* and *wildcards*. Once the JavaSpace has been located, the following operations can be applied to it:

wild-cards which is a lot more comfortable

- Write.* Writes an **Entry** into the space.
- Read.* Reads an **Entry** which matches a template without taking it from the space.
- Take.* Takes an **Entry** which matches a template from the space.
- Notify.* Signals that an **Entry** which matches a template was written.

Entries are not automatically removed, thus a global history is always available at no cost. This is definitely a strong argument in favor of JavaSpaces.



Evaluation

Logging and Replay.

Dumping the content of a JavaSpace into a file yields a simple form of logging. (+)

Privacy.

The global history is always available. A learner may operate on it anytime. (++)

Join.

The global history is always available. A learner may join anytime. (++)

Scalability.

Notification for specific entries prevents irrelevant remote calls. (+)

4.3 Conclusion

This most promising approach is to combine the chocolate sides PROMELA, [SPIN](#), [HiSAP](#) and [NUSS](#). The interactivity and visualization of [HiSAP](#) and the specification-focussed PROMELA input of [SPIN](#) and the collaborative support of [NUSS](#) and JavaSpaces. Table 4.2 summarizes the evaluations performed in this chapter. The highlighted cells mark solutions, which will be used.



Requirement	PROMELA	(X)SPIN	HISAP	HiSPIN	NUSS	VNC	JSDT	JMS	JS
Single Sequence		+	-	+					
Animation		+	++	+					
Easy Generation	+	++	-	++					
String Messages	--		++	o					
Encryption	--			+					
Annotations	+	+	o		++				
Logging and Replay					++				+
Data Access	++	++	-	++					
Multiple Sequences		-	-	+					
Act on Behalf		--		o					
Access Control	++	-		+		-			
Replace	+			+					
Self Evaluation		++		+					
Application Sharing					++	++			
Roles andPermissions					++		+		
Extra Communication					+				
SynchronousAnnotations					++				
Privacy						--			+
Join						++	+	+	++
Scalability						++	+	+	+

Table 4.2: An overview of the evaluation of the tools against the requirements. The shaded fields indicate selection.



Chapter 5

PROMELA and Security Protocols

When using PROMELA for teaching security protocols, some peculiarities have to be taken into account. If one adheres to a specific pattern when generating specifications, it becomes a lot easier. To enable an interactive mode, there are urgent needs for on-the-fly usage of PROMELA commands. This chapter will introduce building blocks for security protocols.

5.1 Entities

A security protocol specification comprises the entities identified in Sect. 2.1.1. These are Alice, Bob, Eve and the Notary and correspond to proctype definitions.

Entities represent the origin and the destination of messages. Hence, it would be of great help to have a symbolic constant to represent an entity. On the other hand, entities are equipped with attributes and it might become handy to have their identification as an array index. To this end, a pre-processor macro combines both by allowing the definition of entities as `mtypes` as well as their usage as array indices. This is subject to the condition only, that the entities constants must be declared consecutively.

5.2 Data Templates and Processing of Messages

As entities correspond to proctypes, they also comprise internal data. Some of these data are specific to the entity, e.g. a shared secret with the Notary. If each entity just declared variables, they would not be grouped according to their purpose. The same variable may be referenced with a different name for different entities. Therefore, a template for the entities' data structure is defined to make sure that this is not the case. Another benefit of this pattern is that by looking at the defined data structures all available variables are revealed.



By means of these templates, the preparation of send and receive statements in a comfortable way comes within reach. It might even be accomplished by mouse clicks only, without a single error-prone keyboard input

Besides the entity data, further data can be specific to a session and may even be necessary to create messages. A typical example for such *session data* is a nonce value, which should be present but different for each session.

Each message inherent to a protocol can be composed as a structure, just like entity and session data. This also makes the meaning of each message field apparent for unambiguous completion by learners during an interactive simulation. In addition, each message should allow selective reception for the message type, sender and receiver. Another designated pre-processor macro simplifies these definitions and even creates the necessary channel at the same time. Having a separate channel for each message type does no harm, since only a few messages are exchanged. Furthermore, all messages are of a different format. Two more useful macros encapsulate the send and receive action.

```

102 #define MESSAGE(s, m, d)
103     mtype { m };
104     typedef msg_##m { d; };
105     chan net_##m = [0] of {mtype, mtype, mtype, msg_##m }
106
107 #define SND(m, s, r, d) net_##m!m(s, r, d);
108 #define RCV(m, s, r, d) net_##m?m(s, r, d);

```

Listing 5.1: Message macros (productive.prl).

The tuple of all messages being exchanged among the parties can also be prepared in this manner. By means of **typedef**, a data structure referred to as a *session* is introduced which contains an entry for each message of the session. The entities can store the history and have access to message contents via the same symbols.

The session itself is specific for each protocol. However, there is only a limited set of sessions which can occur. These are Alice \Leftrightarrow Bob, Alice \Leftrightarrow Eve and Eve \Leftrightarrow Bob. The views of Alice and Bob are restricted to the single session they are involved. Eve, on the other hand, has access to any session.

To define the message exchange within a proctype, one has to proceed as follows, the entire step is shown in List. 5.2:

1. Choose the message type.
2. Assign each field of the corresponding structure with a value which is either part of the entity data or session data. Optionally one of the pre-processor macros may be applied to any combination of such values.



```

296  /*
297  *  build the REQUEST_KEY message and send it.
298  *  the message is not encrypted.
299  */
300  send_REQUEST_KEY:
301  d_step {
302      exam.request_key.nonce1    = exam.sd.nonce1;
303      exam.request_key.initiator  = exam.sd.initiator;
304      exam.request_key.responder = exam.sd.responder;
305  };
306  SND(REQUEST_KEY, ALICE, NOTARY, exam.request_key);
307
308  /*
309  *  Receive the SESSION_KEY message.
310  *
311  *  the entire message is encrypted with the secret shared with the
312  *  Notary, ed.notray_key. The session_key is extracted, so is the
313  *  data which goes into SESSION_REQUEST.
314  *  initiator and nonce1 must match the values sent to the notary.
315  */
316  receive_SESSION_KEY:
317  d_step {
318      RCV(SESSION_KEY, NOTARY, ALICE, exam.session_key);
319
320      exam.sd.session_key
321      = DECRYPT_S(exam.session_key.session_key_i , ed.notary_key);
322
323      exam.request_session.session_key_r
324      = DECRYPT_S(exam.session_key.session_key_r, ed.notary_key);
325
326      exam.request_session.initiator_r
327      = DECRYPT_S(exam.session_key.initiator_r, ed.notary_key);
328  };
329
330  /*
331  *  block, if
332  *  - the initiator in the SESSION_KEY message is a different one
333  *  - the nonce is not the one sent to the Notary with the REQUEST_KEY
334  */
335  (DECRYPT_S(exam.session_key.initiator_i, ed.notary_key) == ALICE);
336  (DECRYPT_S(exam.session_key.nonce1, ed.notary_key) == exam.sd.nonce1);

```

Listing 5.2: Alice sends and receives messages (exam.prl).



3. Apply the SND macro.
4. Apply the RCV macro.
5. The values of the received message are available in the corresponding structure and should be transferred into variables for entity data or session data. Again, pre-processor macros may be applied to any combination of such values.

5.3 Special Validation and Simulation Demands

Looking more detailed at the data involved, leads to a further aspect. Whereas for a validation, the value of a random number is irrelevant, it is crucial for an interactive protocol simulation which is to be run repeatedly. Otherwise, any subsequent execution becomes somewhat predictive. Cryptographic keys encounter a similar situation insofar, as they actually have to protect their value during the simulation. As a consequence, a simple implementation is required during the validation, while the true functionality has to be realized for the simulation case.

One way to cope with this ambiguity is to provide a set of pre-processor macro definitions which are conditionally enabled for the validation but remain undeclared for the simulation. So, the simulator itself can accomplish the desired functionality for symbols in question. The very basic set of functionality and their simple implementations can be declared in a separate file, which is shown in Sect. [A.2](#).

5.4 Intruder Models

The purpose of intruder models is to expose violations of the protocol service specification, according to Sect. [2.1.3](#). There are two approaches to specify the intruder. An omnipotent one will discover any failure in a protocol. A specific intruder on the other hand will exploit a single, well-known flaw.

The omnipotent intruder can be derived through a static analysis of the message exchange. It helps to determine the correctness of a given protocol or solution as delivered by learners. For the generation of omnipotent intruders, a complete solution is available by the work of [\[MS02\]](#) and [\[RSG⁺01\]](#) who have examined intruder models extensively.

A specific intruder is comparable to a regular entity, Alice or Bob. It performs a single execution sequence which will lead to the protocol failure. Typically, the specific intruder is generated by a learner and replaces the omnipotent one. This shows whether the learner was able to identify the corrupt sequence.



Both models can be built on the same base, which is aware of all running sessions. It will store the content of any received message into a the matching session data compound. Hence, it can either replay captured messages as a whole or use recorded data to produce new messages by itself, depending on the protocol in question. For the latter, the intruder may pretend any identity.

5.5 The Notary

The notary is an unbribeable entity whose main task consists in deciding about success or failure of attacks. It avoids the exposure of global data being critical to this decision. Thus, learners acting on behalf of some entity cannot manipulate the outcome. To this end, it must be the only active process in the specification, and replaces the `init` process. The notary performs the following two functions:

Shared Data. Processes which embody the protocol entities must get their initial data from somewhere. In this context, it must be taken into account, that knowledge has to be shared between the notary and each entity. In specific cases like symmetric keys, sharing is even needed to another entity. Hence, the Notary takes over this part by creating and initializing the other processes as well as establishing secrets among them.

Protocol Outcome.

The decision about success or failure of an attack is taken upon access to messages being exchanged within an initiator-responder session. By sharing a private channel with each process it creates, the notary can be informed about every message, its real origin and intended destination.

That means, before an initiator actually sends some payload, it tells the Notary about this plan on the private channel. This plan contains proclaimed sender, intended receiver and message content. The Notary compares actual and proclaimed sender to determine its authenticity and stores the result as well as all other values for this initiator-responder combination.

The same procedure is applied in the opposite direction. After a responder has received a payload message, it notifies the Notary on its own private channel. The Notary checks whether the payload corresponds to the previously announced initiator-responder combination. If so, it determines the outcome according to Tab. 5.1.



Sender	Initiator	Responder	Receiver	Outcome
<i>Alice</i>	<i>Alice</i>	<i>Bob</i>	<i>Bob</i>	
<i>Alice</i>	<i>Alice</i>	<i>Eve</i>	<i>Eve</i>	success
<i>Eve</i>	<i>Eve</i>	<i>Bob</i>	<i>Bob</i>	
<i>Eve</i>	<i>Alice</i>	<i>Bob</i>	<i>Bob</i>	impersonation
<i>Alice</i>	<i>Alice</i>	<i>Bob</i>	<i>Eve</i>	capture

Table 5.1: The outcome determination according to the announcement messages.

Depending on the outcome, a jump to the corresponding label is performed in the notary process. In combination with suitable definitions for outcome and an appropriate **never**-claim, trails for each of the four scenarios can be generated easily as shown below.

```

1409 success :      false ;
1410 failure :     false ;
1411 capture:      false ;
1412 impersonation: false ;
1413 abort:        false ;
1414 }
1415
1416 never {
1417   do
1418     :: notary[0]@OUTCOME -> break;
1419     :: skip;
1420   od;
1421 }
```

Listing 5.3: The Notary outcome and never-claim for the generation of trails.(exam.prl).

5.6 Deriving an Example

Once more, the EXAM protocol, introduced in Sect. 2.1.3 serves as an example. The complete specification is given in Sect. A.2.

Step 1: Message Flow The message flow, as shown in Fig. 2.1, defines the data involved in the messages, session and entity data structures. The excerpt in List. 5.4 shows how messages are declared.

```

112 /*
113  * REQUEST_SESSION
```



```

114 * -----
115 * After the initiator receives the SESSION_KEY message, it contacts
116 * the responder. The message includes the second part of the SESSION_KEY
117 * message, which contains the information intended for the responder
118 * and encrypted with its key.
119 *
120 * int session_key_r
121 * int initiator_r
122 */
123 MESSAGE(EXAM, REQUEST_SESSION, int session_key_r; int initiator_r);

```

Listing 5.4: The declaration of the REQUEST_SESSION message (exam.prl).

Step 2: Entity Data. The entity data for the EXAM protocol is shown in List. 5.5. Each entity has a secure channel to contact the Notary and a secret key shared with the Notary.

```

171 /*
172 * EXAM Entity Data
173 * -----
174 * Each entity has some data associated with it.
175 *
176 * secure : a secure channel to the Notary.
177 * notary_key: a shared secret with the Notary
178 */
179 typedef entity_data_EXAM {
180     chan secure;
181     int notary_key;
182 };

```

Listing 5.5: Entity data for entities of the Exam protocol.

Step 3: Session Data. An EXAM session is established between an initiator and a responder. During each session, two nonces and a payload message are exchanged and stored within the session data.

```

184 /*
185 * EXAM Session Data
186 * -----
187 * The session data includes the information which is negotiated among the
188 * initiator and the responder during the message exchange.
189 *
190 * initiator : the initiating end of the connection
191 * responder : the responding end of the connection
192 * nonce1 : the nonce to authenticate the notary against the initiator
193 * nonce2 : the nonce to authenticate the initiator against the responder

```



```

194 * session_key : the session key to encrypt the user data
195 * payload      : the user data
196 */
197 typedef session_data_EXAM {
198     mtype initiator, responder;
199     int nonce1, nonce2, session_key, payload;
200 };

```

Listing 5.6: Session data for a session the EXAM protocol (exam.prl).

Step 4: Alice Typically Alice sends the first message. Thus, the code snippet (List. 5.7) shows how to send a message. The fields are filled with values from entity and session data after optional encryption has been applied to them.

```

296 /*
297 * build the REQUEST_KEY message and send it.
298 * the message is not encrypted.
299 */
300 send_REQUEST_KEY:
301 d_step {
302     exam.request_key.nonce1 = exam.sd.nonce1;
303     exam.request_key.initiator = exam.sd.initiator;
304     exam.request_key.responder = exam.sd.responder;
305 };
306 SND(REQUEST_KEY, ALICE, NOTARY, exam.request_key);

```

Listing 5.7: Alice sending a message (exam.prl).

Step 5: Bob For Bob, a piece of code (List. 5.8) is shown which contains the reception of a message and a condition check.

```

425 /*
426 * receive the REQUEST_SESSION message
427 * it is encrypted with the key shared with the Notary.
428 *
429 * exam.request_session.session_key_r : the key for the session
430 * exam.request_session.initiator_r   : the initiator of the session
431 */
432 receive_REQUEST_SESSION:
433 d_step {
434     RCV(REQUEST_SESSION, \
435         exam.sd.initiator, \
436         eval(exam.sd.responder), \
437         exam.request_session);
438
439     exam.sd.session_key
440     = DECRYPT_S(exam.request_session.session_key_r, ed.notary_key);

```



```

441
442     exam.sd.initiator
443         = DECRYPT_S(exam.request_session.initiator_r, ed.notary_key);
444     };

```

Listing 5.8: Bob receiving a message (exam.prl).

Step 6: Eve The intruder functionality for Eve has been covered in detail in [MS02], and is only slightly modified to fit the data structures. The static analysis of the message exchange yields what intercepted messages may look like, and how to compose new ones. Two different kinds of keys are used. Keys shared with the Notary (K_{NX}) are not interceptible. Thus, Eve may only try to use its own. The session keys, on the other hand, are stored with the session and may also be applied in different ones. The same principle holds, when a message is created. It should arbitrarily choose the session from which to take the data for each field, or replay messages as a whole. In a session Eve may play three roles. The data used to fill the messages for each role are stored in different session data structure. Eve may either be not involved (exam_capture), try to play the initiator (exam_init) or the responder (exam_resp). The reader may notice, that there is a fourth sessions (exam_actual) defined. This one is used temporarily in send/receive statements, so as not to overwrite any values in a stored session.

Message	Sender	Receiver	Encryption	Stored in
REQUEST_KEY	Alice	Notary		exam_capture
SESSION_KEY	Notary	Alice	K_{NA}	
REQUEST_SESSION	Alice	Bob	K_{NB}	
CHALLENGE	Bob	Alice	K_{AB}	
CONFIRM	Alice	Bob	K_{AB}	
DATA	Alice	Bob	K_{AB}	
SESSION_KEY	Notary	Eve	K_{NE}	exam_init
CHALLENGE	Bob	Eve	K_{EB}	
REQUEST_SESSION	Alice	Eve	K_{NE}	exam_respond
CONFIRM	Alice	Eve	K_{AE}	
DATA	Alice	Eve	K_{AE}	

Step 7: Notary The excerpts to illustrate the functionality are too long to be included in-line. Thus, only the line numbers are given, the complete listing is included in Sect. A. As explained above, the notary creates and initializes all processes and data structures at first (lines 1196–1260). After that, it waits for incoming ANNOUNCE-messages (lines 1337–1406). On reception, it determines the outcome of the simulation according to each message. The Notary also acts as the trusted server for the EXAM protocol and responds to key requests (lines 1292–1331).



Chapter 6

Design

In this chapter, the solutions for the requirements introduced in Chap. 3 are developed. According to the evaluation in Chap. 4, which has been summarized in Tab. 4.2 these are best accomplished with a combination of [HiSAP](#), PROMELA and [HiSPIN](#) as well as [NUSS](#) and JavaSpaces. The following sections will check one requirement after the other and annotate how each is going to be fulfilled.

6.1 Instructive Simulations

An instructive simulation, as defined in Sect. 2.2.3 allows only little interaction. However, it must support a solid base for the following enhancements.

6.1.1 Single Sequence

The interpreter is used to provide the requirements for the single sequence scenario described in Sect. 3.1. It will read a PROMELA specification and simulate it. In general, the simulator controls the execution sequence for a simulation through a `Scheduler` class. The scheduler determines which one of all executable options should actually be executed. This determination can be customized in subclasses. There are already several subclass implementations, one of them performs a simulation according to a trail file generated by [SPIN](#). The single sequence simulation will use this scheduler.

The possibility to step back and forth in the sequence will be provided by the implementation of an undo and redo functionality. By its nature, this functionality can also be applied in the multiple sequence scenario, which is covered in Sect. 6.2.1, with no extra effort. The restriction to remain on the same path after having stepped back is enforced by the scheduler, which will simply stick to the trail.



The undo functionality for three components, the interpreter, the scheduler and the visualization, are coordinated through the simulation. The framework which the `javax.swing.undo` [jun, Fla99] package offers is of great help, as Sect. 7.1.1 will show.

6.1.2 Animation

The animation is required to visualize the message exchange between protocol entities. Multiple views should be possible. HiSAP already provides this functionality. It supports several views, e.g. a topological one and a message sequence and history chart. However, each protocol needs to be implemented as a in JAVA and produce and consume messages for a node. The behavior of a such an implementation reminds of an interpreter process. This makes it the point of contact for the interpreter.

When the interpreter processes a specification, it creates a traversable, tree-like data structure of the running simulation. An event model to control HiSAP visualizations for any kind of protocols has been anticipated [Pap02], but not yet realized. Focusing on security protocols lowers the demands for the event model, as only the message exchange matters.

A protocol implementation will be created, which observes a interpreter process or a channel. Instead of producing and consuming messages itself, this implementation will simply relay the message send and receive actions to the HiSAP node.

A PROMELA process has additional information, like the statement sequence, which is not necessarily of interest to the visualization. Furthermore, a PROMELA specification offers a greater level of detail than HiSAP needs to show. For instance, if two processes have both access to two channels, it often suffices to visualize a single connection. This suggests to use a level of indirection, an adapter which transforms interpreter processes into a HiSAP nodes. For this functionality, the interpreter needs to supply events for the following actions:

A new process or channel is created.

According to the identifier, the adapter will either ignore the new item, merge it with an existing node or create a new node and its connections.

A process appends a message to a channel.

If the process and the channel have not been merged into a single node, the adapter will cause the process node to send a message.

A process takes a message from a channel.

If the process and the channel have not been merged into a single node, the adapter will cause the channel node to send a message.



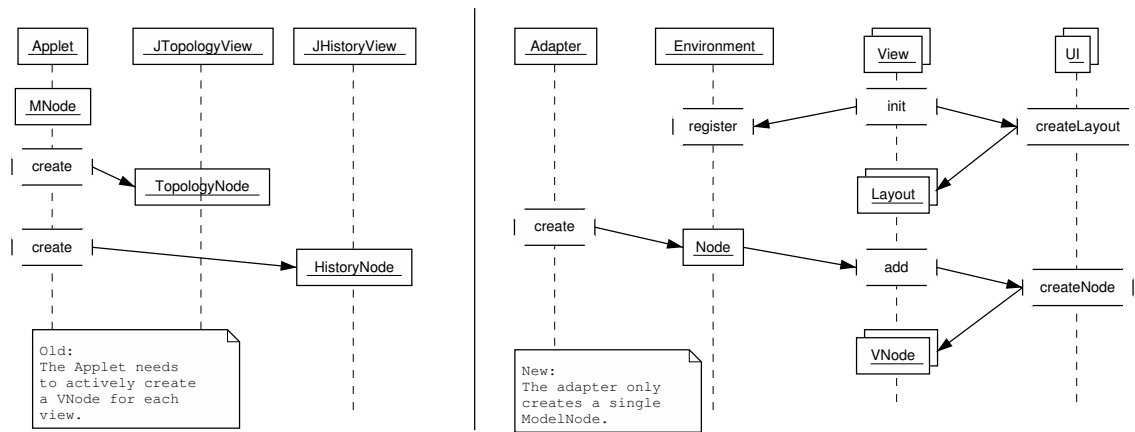


Figure 6.1: The old vs. the new creation of a [HiSAP](#) node.

A process gains / loses access to a channel.

If the process and the channel have not been merged into a single node, the adapter will add a connection between the process and channel node.

When a typical [HiSAP](#) applet is initialized, it sets up the scenario, i.e. it creates

1. all nodes and specific protocol implementations for them.
2. all connections among the nodes.
3. visual components in each desired view for all nodes and connections.

Backing the visualization with a simulation simplifies the entire set up. Each node can be driven by an instance of the same generic protocol implementation. Of course, each must be listening on a different process or channel.

Since processes may be created and terminate during a simulation, the resulting setup is no longer static. Also, having to create the visual component for each view implies, that all views must be known to the applet.

An addition to the [HiSAP](#) architecture reflects the simulation setup. The concept of an environment is introduced which roughly corresponds to the simulation. After this change, nodes will be created within such an environment. Figure 6.1 shows the old and the new generation of a node. It will provide an event model similar to that of the simulation. The following actions will trigger events:

1. A node is added or removed.



2. A connection is added or removed.
3. A message is sent.

Thus, any kind of view may listen on an environment and will be informed about the creation or removal of nodes and connections. Each view is responsible for creating a suitable visual component for each model component and laying them out.

When a simulation is started, an environment is created. A simulation adapter listens for events from the interpreter and modifies the environment accordingly, e.g. creates or removes nodes or connections. Arbitrary listeners may register with the environment and visualize it in any way. Figure 6.1 shows the old and the new creation process.

6.1.3 String Messages

Neither PROMELA nor (X)SPIN support a string data type. They are desired because they are simply much more expressive than integers. One approach to solve this problem is to introduce such a data type. As the other tools would not recognize it, it needed to be hidden from them. The pre-processor could achieve this easily. However, the new data type itself would cause a lot of work.

Another way to work around this lack, is to use symbolic constants defined with `mtype`. The resulting variables are integers associated with an identifier. XSPIN displays the identifier instead of the numerical value.

The interpreter also supports this solution. Each integer can be assigned not only an identifier, but an entire string. So far, the only way this happens is through an `mtype` definition. The routine for changing an integer value will also include a field to change the String description.

6.1.4 Encryption

A twofold solution is necessary to provide encryption for PROMELA specifications. The application of encryption must not break SPIN compatibility. The solution has already been proposed in Sect. 5.3. The simulator needs to recognize a couple more tokens and provide their functionality.

When a key with the same id is created twice, a runtime error occurs. En- and decryption is not only applied to the integer value, but also to the string representing the symbolic value.

This implies, that the pre-processor is also available for the simulator, so that the directives can be used. This is not yet the case, as the pre-processor directives are not



Token	Parameters	Description
GEN_KEY_SYM	id	Generate a symmetric key, which is associated with id.
ENCRYPT_S	val, key	Encrypt val symmetric with key.
DECRYPT_S	val, key	Decrypt val symmetric with key.
GEN_KEY_PUB	id	Generate a public/private key pair, which is associated with id and return the public key.
GEN_KEY_SEC	id	Generate a public/private key pair, which is associated with id and return the private key.
ENCRYPT_A	val, key	Encrypt val asymmetric with key.
DECRYPT_A	val, key	Decrypt val asymmetric with key.

Table 6.1: Additional cryptographic tokens for the scanner.

part of the PROMELA language, and [SPIN](#) uses an external pre-processor. This problem can be solved in three ways:

1. Use the same external pre-processor [SPIN](#) uses. Relying on a platform dependent external solution should be the last consideration.
2. Create a true preprocessor, which executes in the same way the original does, i.e. it reads and processes the entire input before the actual PROMELA-specific lexical analysis begins.
3. Implement the preprocessor within the lexical analyzer scanning the input for PROMELA tokens.

The second approach seems to be the best solution, as it keeps a clear distinction between directive handling and PROMELA code. However, the third approach was chosen for a first, temporary, implementation, because it promises the least effort. It is the most straight forward one to implement with little overhead. It also leaves the possibility to extract the directive handling later on and create a stand alone pre-processor later.

6.1.5 Data Access

As mentioned in Sect. [6.1.2](#), an adapter will connect the simulation and the visualization. It will associate interpreter processes with visual [HiSAP](#) nodes. The simulation is already traversable which allows the user to view the running processes and the values of their local variables. These values should also be accessible through the visual [HiSAP](#) components.



Access to simulation data could just be relayed to the simulation itself. However, due to the adapter, the association between the processes and the **HiSAP** components is not one-to-one. Thus, the adapter, should just build a simplified duplication of the attributes for each entity and associate it with the **HiSAP** node. A new kind of view can then display the data tree. Having another representation of the data, which is independent of the actual simulation will also prove beneficial in Sect. 6.4.

6.2 Interactive Simulations

The interactive simulation lets the learner determine several execution sequences. There are several ways how this sequence can be determined.

6.2.1 Multiple Sequences

As mentioned in Sect. 6.1, the sequence is determined by a scheduler. To allow multiple execution sequences, the scheduler displays all possible options to continue and the user selects one. When the user undoes a couple of steps, the same choices will be offered again. This way a learner may thoroughly explore a protocol. Several criteria for statement selections are possible.

1. Select message Actions. A choice is offered if a sending/receiving option is to be executed.
2. Process selection. Choose which process is to be executed.
3. Statement selection. Select the precise PROMELA statement to be executed next.

A future extension could wrap statements into a pre-processor macro, which provides a textual description of a statement which is offered. For regular **SPIN** usage, this macro should just drop the description. This would be similar to the methods to provide encryption, as described in Sect. 6.1.4.

6.3 Explorative Simulations

When a learner invokes an explorative simulation run, the focus shifts towards self-evaluation. That means, access to simulation and process data should be restricted, which makes cheating harder. This is achieved by relying on the floor control mechanisms included in the **NUSS** project, which will be introduced in detail in Sect. 6.4.2.



6.3.1 Act on Behalf

The ultimate in data access is the concept of a user acting on behalf on a process. When this mode is selected, the statement sequence of the process is ignored and the user is queried instead. Such a process is entirely controlled by a user. This means, the learner becomes a part of the simulation, just like a regular process. It is assumed, that the specification adheres to the conventions introduced in Sect. 5.3. The commands a learner may execute are restricted to the following three

1. Send and receive messages according to the scheme of the SND/RCV macros, i.e. select a message type, sender, receiver and content structure.
2. Assignments among entity data, session data, and messages, including en- and decryption.
3. Check a condition with entity data, session data, and messages, including en- and decryption.

While first stage allows the learner to still read the data members of the other processes, the second stage in this mode restricts the view on the data visible to the process. The implementation of this feature would go beyond the scope of this thesis. However, it could be combined with a series of dialogs to support the creation of PROMELA security protocols according to Sect. 5.

6.3.2 Access Control

Access Control is covered along with *Roles and Permissions*, in Sect. 6.4.2.

6.3.3 Replace

Similar to acting on behalf of a process, the learner is asked to enter the statement sequence of the process. This is subject to the same restrictions mentioned in Sect. 6.3.1. However, the entire body, i.e. all statements are specified before the simulation begins. The learner must anticipate the entire message exchange.

Another difference is, that multiple processes may be replaced for the same session. While this idea might seem unspectacular at first, it really provides a boost to the functionality of the environment. An important teaching goal identified in Sect. 2.1.4 is, that students should be able to fix flaws in a protocol. As the fixing of the EXAM protocol shows, this might involve the handling of messages in several entities. Thus, to fix the protocol, multiple entity definitions must be replaced. To simplify the task for the learner, a specification might already contain multiple replacements.



6.3.4 Self Evaluation

The base for self evaluation is formed by acting on behalf and replacing protocol entities. The former offers the possibility to practice and judge the own performance. While the latter, really evaluates a students solution objectively. This is achieved by running a validation on a modified specification. To this end, **SPIN** is encapsulated into the environment, in a similar way **XSPIN** does it, i.e. it is invoked as a separate process. However, this is only to perform validations. The learner replaces one of the three participating entities. Instead of running a regular simulation, the **SPIN** validation chain is invoked in the background. Three outcomes are possible, but the case of an **SPIN** or compiler error is just ignored for now. Either the validation yields no error or a trail file is generated. For the latter result, a simulation can be started right away.

6.4 Collaborative Simulations

While the interaction described in Sect. 6.3 focuses on how a single user may learn with a simulation, the requirements demand the environment to offer a distributed and collaborative multi user mode.

6.4.1 Application Sharing

Application sharing will allow multiple learners to operate on the same set of data simultaneously. The first decision to take is how to organize the resulting application and which parts should be shared. There are two possible approaches.

For the first one, each learner has a fully functional environment to work with. Each includes an interpreter and performs the simulation locally. All simulations of a working group could be coordinated by sharing something like a global trail. However, it would have to be assured that all participants use the same adapter, so that all resulting animations look the same. Otherwise their views will differ and confusion will inevitably rise.

If the group engaged in a session, where different learners act on behalf different processes, their actions would have to be relayed to all of the other simulations. Yet the main drawback of this approach is that each environment needed to supply the platform dependent tools to perform a validation.

The more promising approach does not share the entire simulation, but only the visualization. There is one single simulation going on at a centralized server. The adapter which translates the simulation into an animation shares the result. The learners do not work with a full scale environment, but with a lightweight client. The clients do not even



see the actual simulation, but only the output generated by the adapter. Thus, all learners definitely have the same view. Given the learner is endowed with the required permissions, the server allows each client to control the simulation, e.g. choose the next statement. Only the server needs to provide access to the platform dependent tools.

6.4.2 Roles and Permissions

A shared introduces some more aspects to the interactive simulation described in Sect. 6.2. Whereas the interactive case assumes a single participant only, a distributed simulation has to consider multiple users. Since unrestricted access for all session members would most likely lead to nothing else but chaos, access rights need to be determined. Users can request permissions in the course of a session. Whether they are granted their wish, is up to the session management policy. This policy is also used for access control in a single user session. The following actions require privileges

- * Control a simulation.
- * Create a session.
- * Join a session.
- * Access process Data.
- * Replace a process.
- * Act on behalf a process.



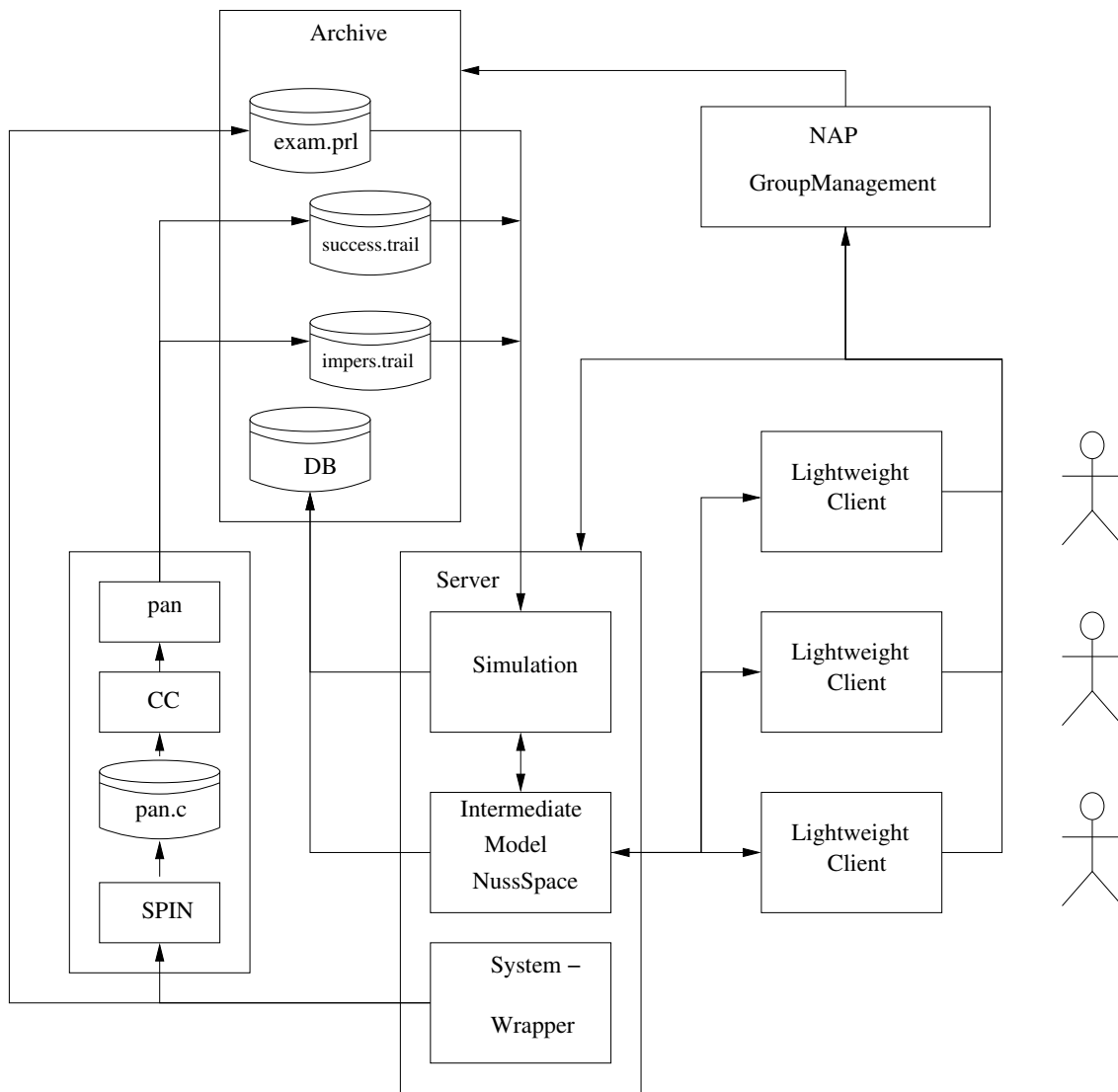


Figure 6.2: An overview of the ProDuctivE Architecture.

6.5 Resulting Architecture

The architectural approach as shown in Fig. 6.2 also includes the support for the actual implementation. The main simulation and its entire history reside on the server. Each client is notified of changes and builds its own snapshot of the current state. Also, each client may move around in the locally recorded history. Trail files are generated with **SPIN** and re-used for the simulation.



Chapter 7

Implementation

As determined in Chap. 6, the chosen approach is to use the [HiSPIN](#) simulator and the [HiSAP](#) toolkit as the starting point. This solution allows the usage of trail files generated by [SPIN](#) to provide a deterministic execution sequence for any given protocol specification written in PROMELA. It also allows to use [SPIN](#) for self evaluation purposes. For the entire implementation JAVA [\[j2s\]](#) has been used. For a shared application, [NUSS](#) and JavaSpaces are added.

7.1 Instructive & Interactive Simulation

An instructive simulation run is driven by a single pre-defined execution sequence. The only control mechanisms available are to interrupt the simulation and to step back- and forward by means of a undo and redo functionality. For an instructive simulation, the undo facility is limited as such, that there is only one execution sequence. Typically, this execution sequence is determined by performing a validation using the model checker [SPIN](#). The purpose of an instructive simulation run is to perform a demonstration of a specific protocol behavior.

7.1.1 Single & Multi Sequence

As mentioned in Sect. 6.1.1, the single sequence scenario already exists partially. It is the undo-/redo functionality which is missing. It allows a learner to step back- and forward in the execution sequence. The undo support is implemented for the interpreter, all schedulers and the animation.

The framework which the `javax.swing.undo` [\[Fla99, jun\]](#) package offers is of great help. Since both, the restructured [HiSAP](#) and the interpreter, heavily use inheritance, ap-



```
1 UndoableEditSupport undoSupport = new UndoableEditSupport();
2
3 StateEditable undoWatch = new StateEditable()
4 {
5     public void storeState(Hashtable state)
6     {
7         storeState(state);
8     }
9     protected void restoreState(Hashtable state)
10    {
11        restoreState(state);
12    }
13 };
14
15 StateEdit undoState = new StateEdit(this.undoWatch);
16
17 protected void storeState(Hashtable state)
18 {
19     // subclasses invoke super.storeState(state);
20 }
21 protected void restoreState(Hashtable state)
22 {
23     // subclasses invoke super.restoreState(state);
24 }
25 protected void postEdit()
26 {
27     this.undoState.end();
28     this.undoSupport.postEdit(this.undoState);
29     this.undoState = new StateEdit(this.undoWatch);
30 }
```

Listing 7.1: The basic Undo-Support.

plying the same feature to each of the base classes will do. The technique used relies on objects of the types `StateEditable` and `UndoableEditSupport`, both taken from the above mentioned framework. The `StateEditable` object provides the means for another object to store its state into – and restore it from – a `Hashtable`. The `UndoableEditSupport` object collects undo information and handles event generation and the like. Listing 7.1 shows how the undo support is provided through the base classes.

7.1.2 Animation

The animation requires an event model for the interpreter. The minimum of events to support have been identified in Sect. 6.1.2. However, it makes sense to not only implement the required events, but to provide all those that appear. An overview of all interpreter events is presented in Tab. 7.1.

The [HiSAP](#) toolkit already offers the functionality to create JAVA applets which per-



Simulation Event	The simulation has ...
BeginSimulation	... been started.
AbortSimulation	... been aborted.
EndSimulation	... terminated.
Timeout	... encountered a timeout.
Deadlock	... hit a deadlock.
RuntimeError	... produced an error.
Process Event	The process has ...
ExecutableOptions	... options which are executable now.
(rv)MessageReceived	... received a (rendez-vous) message.
(rv)MessageSent	... sent a (rendez-vous) message.
StateReached	... reached a certain state.
Output	... produced output through a print statement.
ChannelAccessGained	... gained access to a channel.
ChannelAccessLost	... lost access to a channel.
ChannelExclusiveRead	... gained the exclusive read privilege for a channel.
ChannelExclusiveWrite	... gained the exclusive write privilege for a channel.
ChildProcessCreated	... created a child process.
ChildProcessRemoved	... removed a child process.
Terminated	... terminated.
Channel Event	The channel has ...
MessageQueued	... queued a message from a process.
MessageDequeued	... dequeued a message to a process.
ChannelFull	... no more room for messages.
ChannelEmpty	... no more messages stored.
Data Event	
IntegerChanged	An integer value changed.
ChannelReferenceChanged	A channel reference changed.

Table 7.1: An overview of the interpreter event model, which allows generic adaption.

form animations of protocols, by modeling nodes, connections and messages. Although the [HiSAP](#) toolkit was designed to have separate model and view components, these are not separated clearly enough. For instance, when a node is added to the model it needs explicitly be added to each view component. Another drawback is the usage of `PropertyChangeSupport` to compensate the lack of a dedicated event model.

The most fundamental change in the [HiSAP](#) toolkit is the introduction of an environment component, which acts as a container for all model components. Before the environment was introduced nodes and connections existed in isolation. Whenever a new model component of any kind was created, its visual counterpart had to be created separately for each kind of view. This change allows the view components to register with the environment and to be notified whenever a model component is added or removed. The



EnvironmentEvent	
NodeAdded	A node has been added to the environment.
NodeRemoved	A node has been removed from the environment.
ConnectionAdded	A connection has been added to the environment.
ConnectionRemoved	A connection has been removed from the environment.
Transmission	A node has sent a message.
NodeEvent	The node has ...
Receive	... received a message.
Send	... sent a message.
StateChange	... changed its state.
ConnectionEvent	The connection has ...
Interrupt	... been interrupted.
Transmit	... started transmitting.

Table 7.2: An overview of the HiSAP event model.

entire event model that comes with it is shown in Tab. 7.2.

Along with this alteration, the entire hierarchy was redesigned in a way inspired by [GHJV94]. Some effort was put into keeping the result flexible. For this reason, public interfaces are created and the implementations are kept private in a separated package. The sole way to instantiate such an implementation is through a dedicated public factory class from within the same package. This pattern is used quite frequently and eases the implementation of the undo functionality tremendously.

The new hierarchy, as shown in Fig. 7.1 allows this feature be realized in a quite straight forward way, while the structure of the old HiSAP implementation did not support it very well.

The full event model for the simulator, as shown in Tab. 7.1, offers four sources of events: simulation, process, channel and data. Whenever an integer value or a channel reference is changed a data event is emitted. Channel events occur when a channel (de)queues a message or becomes empty(full). As the name suggests process events originate when something notable happens within a process. For instance, they signal the reception/sending of a message or a state change.

At first sight, the task of the adapter seems rather simple. All it needs to do is receive the events from the simulation and manipulate the EnvironmentModel accordingly. However, it also needs to perform grouping of simulation components. For instance, all globally accessible channels should be represented by a single ModelComponent. But it does make sense to allow grouping on a protocol specific base, so the adapter should be configurable. It also needs to determine how to arrange the ViewComponents.



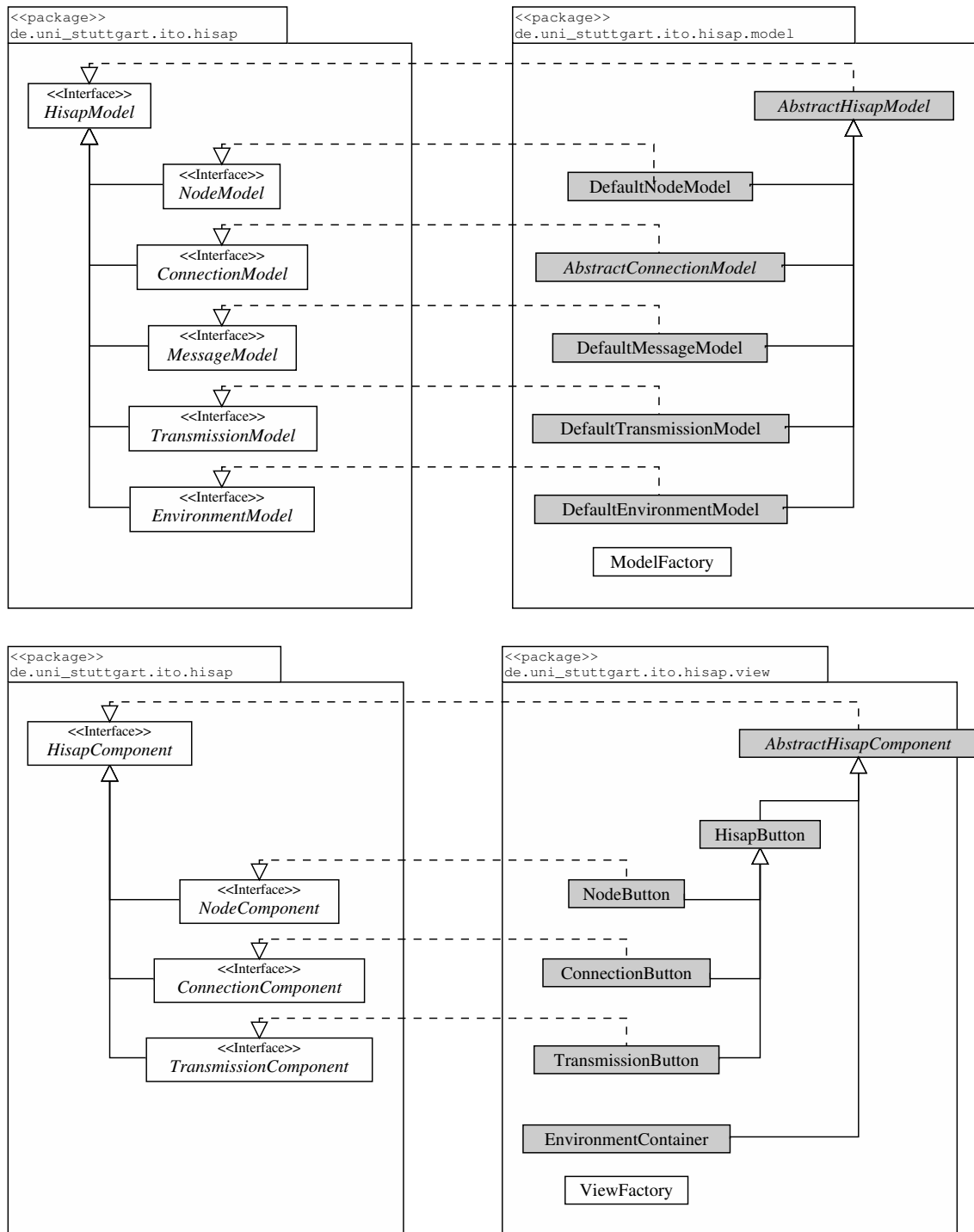


Figure 7.1: The new **HiSAP** hierarchy – model and view interfaces and implementation. The shaded classes are package private.



As they adhere to the same structure, the undo/redo support for both, the simulator and the [HiSAP](#) toolkit turns out to work the same way.

7.1.3 Encryption

The additional encryption tokens need to be added to the set of scanner terminals. Their functionality is implemented as `PUnaryOperator` or `PBinaryOperator` operators. The parser will reduce operator and operand(s) to a `PExpression`. Illustrating the entire matter at this point would present way too much detail.

For now the implementation of the cryptographic functions is the same as provided by the macro definitions, except for the random number. However, a future enhancement should consider the JAVA cryptographic extension [[jce02](#)].

The implementation of the pre-processor uses the feature offered by [JFlex](#), the scanner generator in use, to keep a stack of input streams. Whenever a macro is encountered the current input stream is pushed on the stack and the corresponding expansion is processed. For now, only a subset of the most important directives is implemented. One more future extension could consist of a standalone pre-processor.

<i>#define</i>	Allows the definition of macros, with or without arguments.
<i>#undef</i>	Clears a macro definition.
<i>#ifdef ... #else ... #endif</i>	Allow conditional parsing of the input, whether a macro has been defined or not.
<i>#include</i>	Includes a separate file at the current location.
<i>## (concatenation operator)</i>	Using this operator concatenates a macro argument with some extra text into a single token.

7.2 Explorative Simulation

The explorative scenario gives more control to the learner. Self evaluation moves to the center of interest. As it turns out, self evaluation directly follows replacing a protocol entity.



```

255 proctype alice(chan init_alice)
256 {
257     /*
258      * local data
259      */
260     entity_data_EXAM ed;
261     session_EXAM exam;
262     msg_ANNOUNCE announce;
263
264     /*
265      * receive the initializatin from the notary
266      *
267      * ed.secure      : a secure channel to the notary
268      * ed.notary_key : a shared secret with the notary
269      *
270      * exam.sd.initiator : self
271      * exam.sd.responder : the party to contact
272      * exam.sd.nonce1   : the nonce to authenticate the notary
273      * exam.sd.nonce2   : <undefined>
274      * exam.sd.session_key: <undefined>
275      * exam.sd.payload  : the data to send
276      */
277     init_alice ?ed, exam.sd;
278
279     /*
280      * build the announcement message, which will inform the notary of
281      * the intended communication. The message is sent to the Notary
282      * over the secure channel.
283      */
284     send_ANNOUNCE:
285     d_step {
286         announce.sender = ALICE;
287         announce.receiver = exam.sd.responder;
288         announce.data = exam.sd.payload;
289     };
290
291     ed.secure!ANNOUNCE(announce);
292
293     #ifdef REPLACE_ALICE
294     #include REPLACE_ALICE
295     #else

```

Listing 7.2: How to replace the body of a proctype (exam.prl).



7.2.1 Replace

The pre-processor is used to allow a learner to replace the protocol specific body of a proctype with a new statement sequence. To this end, the body of each entity is enclosed in conditional directive, as shown in Listing. 7.2. When the user wants to replace the body of a proctype, a temporary file with the new content is created, and the corresponding macro defined accordingly.

Note, that a learner may only replace the protocol specific part. The initialization and announcements are still taken from the original specification, which saves quite a bit of overhead. For now, the statement sequence has to be entered as plain text.

The simulation is performed according to the interactive scenario, where the learner can inspect the altered version.

7.2.2 Self Evaluation

Self evaluation is achieved as a direct consequence of the replace functionality. Instead of running a simulation, the spin validator is invoked on the specification. The environment must be able to find the executables. To this end, when the environment is started it will read a property file, “.productive”, from the current directory. It should contain the following information:

```
productive.spin="/path/to/spin"  
productive.gcc="/path/to/gcc"
```

7.3 Collaborative Simulation

A Collaborative simulation shall be distributed and with multiple participants. [NUSS](#) is used for the basic functionality, such as service discovery and group management. For now, services are listed at the [NUSS](#) access point (NAP), whose address is given as a command line parameter when starting the environment. The group management developed in [\[Nol03\]](#) is used.

7.3.1 Application Sharing

Three concepts make up the core of the provided application sharing functionality. These are the `NussMenu`, the `NussSpace` and the `Entry`.



Further development in mind, most of the functionality `NUSS` provides is kept isolated from the main application. The entire communication with the NAP is encapsulated within the `NussMenu`. So far, it provides only the most basic functionality needed for a shared session. It allows the user to login, join and create sessions as well as to acquire and release rights. The `NussMenu` hides all relevant information from the application itself. It can easily be adapted to any changes or even be replaced by a more powerful solution.

Access control to the simulation data is provided by the `NussSpace`, which is a combination of the `NUSS` group management and a `JavaSpace`. It offers essentially the same interface as `JavaSpace`, but each method takes an additional argument, a `NUSS Access-Key`. This allows to check whether the permissions grant the execution of an requested action before it is actually performed.

An `Entry` delivers the data to the clients. They are used for any information which is important for the clients. For each executed simulation step, the adapter updates the `SimulationTimeEntry`, which keeps track of the current global simulation time. At any given time, there is only one such entry.

Additionally, one `SimulationEntry`, which describes global variables and contains a list of available processes is created. As well as a `ProcessEntry` which lists the internal state of each process in the same manner. These entries exist for every single step. They contain the simulation time as an attribute.

Access to the `SimulationTimeEntry` and `SimulationEntry` is not restricted. Thus, at any time all clients may investigate the history of global variables. Reading a `ProcessEntry`, however, requires the corresponding permission.

7.3.2 Roles and Permissions

This kind of dynamic floor control is provided by the `NUSS` group management. When the server announces its presence, it also creates an instance of `GroupManagement`. Clients can request permissions through their `NussMenu`. The set of entities is restricted to those of security protocols. This allows to hard-code the entity names to the permissions, as shown in Tab. 7.3.

The concept of a role simplifies the handling of permissions. A role can be seen as a collection of permissions. It seems appropriate to combine all permissions regarding a single entity to a role.



Permission	The user may ...
CreateSession	... start a new session.
JoinSession	... join existing sessions.
ControlSimulation	... determine the execution sequence for the session.
ReadAlice ReadBob ReadEve ReadNotary	... read the data of the given process P.
ReplaceAlice ReplaceBob ReplaceEve	... replace the body of the given proctype.
BeAlice BeBob BeEve	... act on behalf of the given proctype.
SetOutcome	... define which outcome should produce an error, and thus a trail file.
Verify	... run the SPIN validator on the current specification.
Role	Permissions
Instructor	All
Alice	{Read,Be,Replace}Alice
Bob	{Read,Be,Replace}Bob
Eve	{Read,Be,Replace}Eve

Table 7.3: The basic permissions for security protocols.



Chapter 8

Conclusion & Outlook

The goal of this thesis was to design an collaborative environment to support the learning of security protocols. A prototypical implementation was to create. The conclusion summarizes what has been achieved and the outlook shows how the work might continue.

8.1 Conclusion

The requirements for a collaborative environment have been identified. Solutions have been designed to meet those requirements. A way to use PROMELA for teaching security protocols has been developed. To this end, the important functionality of the C preprocessor has been added to the scanner. Encryption functionality has been introduced to the interpreter.

An event model for the interpreter was developed and implemented. The [HiSAP](#) toolkit has been remodeled and connected to the interpreter. Undo support was added to both of these.

Distributed simulations based on [NUSS](#) and JavaSpaces have been created. A simple permission scheme for security protocols was defined.

8.2 Outlook

Of course, a prototypical implementation still leaves something to do. For instance, the implementation of an act on behalf dialog, which resolves the data members available to a process from the specification.



A further enhancement would be a security protocol wizard, which queries the user for the session data, entity data and message definitions of a protocol and creates the PROMELA code for regular entities from it.

The annotation functionality [NUSS](#) offers has been considered but is not yet attached. The same holds for the logging and replay functionality, as well as the extra communication facilities.



Bibliography

- [Bro99] Torsten Brodbeck. Grundlegende Überarbeitung der Teile des Java Visualisierungsbaukastens für Protokolle. Studienarbeit, Universität Stuttgart, 1999.
- [Fla99] David Flannagan. *Java Foundation Classes in a Nutshell*. O'Reilly & Associates, 1st edition, 1999.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.
- [Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
- [Hol97] Gerard J. Holzmann. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5), 1997.
- [j2s] Java 2 platform, standard edition (j2setm). WWW. URL: <http://java.sun.com/docs/index.html>.
- [jce02] Java cryptography architecture api specification & reference. WWW, 2002. URL: <http://java.sun.com/j2se/1.4/docs/guide/security/CryptoSpec.html>.
- [jsd] Java Shared Data Toolkit. WWW. URL: <http://java.sun.com/products/java-media/jsdt/>.
- [jun] Package javax.swing.undo. WWW. URL: <http://java.sun.com/j2se/1.4/docs/api/javax/swing/undo/package-summary.html>.
- [Mer98] Brenda Mergel. Instructional design & learning theory. WWW, May 1998. URL: <http://http://www.usask.ca/education/coursework/802papers/mergel/brenda.htm>.
- [MH01] Richard Monson-Haefel. *Java Message Service*. O'Reilly & Associates, 2001.



- [MS02] Paolo Maggi and Riccardo Sisto. Using SPIN to Verify Security Properties of Cryptographic Protocols. In *Proceedings of the 9-th SPIN Workshop 2002*, 2002.
- [mvn] MulticastVNC. WWW. URL: <http://www.informatik.uni-trier.de/~ziewer/MulticastVNC/>.
- [MvV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
- [Nol03] Jochen Noller. Hierarchische Floorcontrol und Filterung von Primitiven. Studienarbeit, Universität Stuttgart, 2003.
- [Pap02] Matthias Papesch. Generating Implementations from Formal Specifications: A Translator from Promela to Java. Studienarbeit, Universität Stuttgart, 2002.
- [prl97] Concise promela reference. WWW, 1997. URL: <http://spinroot.com/spin/Man/Quick.html>.
- [RSG⁺01] Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *modelling and analysis of security protocols*. Addison-Wesley, 2001.
- [Sch02] Sören Schneider. Evaluation von Systemen zur Korrektheitsuntersuchung am Beispiel von Konsistenzerhaltungsprotokollen. Diplomarbeit, Universität Stuttgart, 2002.
- [ssh] OpenSSH. WWW. URL: <http://www.openssh.org>.
- [Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, 1996.
- [tvn] TightVNC. WWW. URL: <http://www.tightvnc.com>.



Appendix A

PROMELA Examples

A.1 A Common Ground

```

1  /*
2  *  File : productive.prl
3  *
4  *  Description:
5  *  -----
6  *  This file provides macro definitions for comfortable implementation
7  *  of security protocols.
8  */
9
10 #ifndef __productive__
11 #define __productive__
12
13 #ifndef HISPIN
14
15 /*
16 *  Random Number Support
17 *  -----
18 *
19 *  For a validation, a random value can be re-used. Not so for an
20 *  interactive simulation. This macro provides a possibility to indicate
21 *  that the interpreter should replace the given value with a randomly
22 *  generated one. The validator, however, may just replace the value with the
23 *  noted default.
24 */
25 #define RND(x) x
26
27 /*
28 *  Symmetric Cryptography
29 *  -----
30 *
31 *  To perform symmetric encryption, three macros are defined.
32 *  The first creates a key, which can be used for en- and decryption
33 *  with the functions implemented by the other two functions, respectively.

```



```

34 *
35 * ^ : bitwise exclusive or
36 * << : shift left
37 * ~ : bitwise negation
38 */
39 #define GEN_KEY_SYM(id) RND(id)
40 #define ENCRYPT_S(val, key) ((val ^ ((key << 8) & ~255)))
41 #define DECRYPT_S(val, key) ((val ^ (key << 8)))
42
43 /*
44 * Asymmetric Cryptography
45 * -----
46 *
47 * The difference to symmetric encryption is, that asymmetric requires two
48 * different keys. A preliminary solution is to use the default value to
49 * lookup the wanted key. A duplicate key error should be created, when
50 * a key with the same value is created twice.
51 *
52 * The additional functionality gained is the possibility to create
53 * signatures and to verify them. The implented signature function simply
54 * returns the encrypted value.
55 */
56 #define GEN_KEY_PUB(id) (~id & 255)
57 #define GEN_KEY_SEC(id) ( id & 255)
58 #define ENCRYPT_A(val, key) ((val | ((key & 255) << 8) ))
59 #define DECRYPT_A(val, key) ((val ^ ((~key & 255) << 8) ))
60
61 #define SIGN(val, key) ENCRYPT_A(val, key)
62 #define VRFY(val, key) DECRYPT_A(val, key)
63
64 #endif
65
66 /*
67 * Improved Message handling
68 * -----
69 * These macros provide a more comfortable way to handle messages
70 * interactively . If the entire message content is contained within a
71 * data structures , its components can be accessed by a name. This is
72 * clearly more convenient than the anonymous, positional access "regular"
73 * channels use.
74 *
75 * MESSAGE macro
76 * -----
77 * The message maco makes sure, that
78 * – an mtype for the message is created
79 * – each protocol specific message is described by a typedef'ed structure
80 * – a global channel is created , which takes 4 arguments
81 * 1. the type of the message
82 * 2. the mtype describing the sender
83 * 3. the mtype describing the receiver
84 * 4. a structure of the correct type
85 * Messages can only be blocked according to the sender and the receiver,
86 * but not on the content.

```



```

87  *
88  * SND/RCV macros
89  * -----
90  * The SND/RCV macro provide a simple invocation of a send/receive
91  * operations for a message defined with the MESSAGE macro. They take
92  * – the message type
93  * – the sender
94  * – the receiver
95  * – a data structure
96  * as arguments. They create the correct statement for the channel
97  * created through the MESSAGE macro.
98  *
99  * These macros use the concatenation operator '##', which merges two
100 * string tokens into a single one. (see man cpp)
101 */
102 #define MESSAGE(s, m, d)
103     mtype { m };
104     typedef msg_##m { d; };
105     chan net_##m = [0] of {mtype, mtype, mtype, msg_##m }
106
107 #define SND(m, s, r, d) net_##m!m(s, r, d);
108 #define RCV(m, s, r, d) net_##m?m(s, r, d);
109
110 #endif

```

Listing A.1: The include file with shared definitions (productive.prl).

A.2 The Exam Protocol

```

1  /*
2  * File: exam.prl
3  *
4  * Description
5  * -----
6  * This specificaton defines the EXAM protocol.
7  *
8  * Assumptions
9  * -----
10 * 1. The network is insecure.
11 * 2. Each party shares a secret with a trusted server.
12 *
13 * Service
14 * -----
15 * The protocol allows an initiator to set up a session with a responder.
16 * It provides
17 *
18 * 1. authentication
19 * 2. privacy
20 * 3. data integrity
21 *
22 *
23 * Alice
24 * -----

```



```

25 * Wants to initiate an exam session. The scenario allows two partners,
26 * either with Bob or Eve.
27 *
28 * Bob
29 * ----
30 * Responds to an session request. According to the scenario, this
31 * request originates either from Alice or Eve.
32 *
33 * Eve
34 * ----
35 * Will try to break the service specification of the protocol. This
36 * will be accomplished by trying to send/receive arbitrary messages.
37 *
38 * Notary
39 * -----
40 * Takes the role of the trusted server. It will initialize the other
41 * entities through private channels, establishing shared secrets. It
42 * will also listen to announcements of the other entities and
43 * distinguish the outcome of the protocol.
44 */
45
46 /*
47 * include the productive extensions
48 */
49 #include "productive.prl"
50
51 /*
52 * Simplifying macros
53 */
54 #define MAX_PROC 4
55 #define MAX_PROC2 16
56
57 #define ENTITY_INDEX(e) (e % MAX_PROC)
58
59 #define I_ALICE ENTITY_INDEX(ALICE)
60 #define I_BOB ENTITY_INDEX(BOB)
61 #define I_EVE ENTITY_INDEX(EVE)
62 #define I_NOTARY ENTITY_INDEX(NOTARY)
63
64 #define MODIFY(val) (val - 1)
65 #define OUTCOME capture
66
67 /*
68 * Messages
69 * -----
70 * The defined proctypes and their functionality are:
71 *
72 * REQUEST_KEY
73 * -----
74 * This is the first message. It is sent from the initiator to the
75 * Notary. Its purpose is to request the server to create a
76 * shared secret for the specified initiator and responder. None of
77 * the fields is encrypted.

```



```

78 *
79 * initiator : the initiating party
80 * responder: the requested party
81 * nonce1 : a nonce value to provide authentication
82 */
83 MESSAGE(EXAM, REQUEST_KEY, mtype initiator; mtype responder; int nonce1);
84
85 /*
86 * SESSION_KEY
87 * -----
88 * The Notary's response to a REQUEST_KEY. It consists of two
89 * parts. The first part is intended for the initiator itself. The second
90 * one is intended to be relayed to the receiver. The entire message
91 * is encrypted with the shared key of the initiator. Before that, the
92 * second part is encrypted with the shared key of the responder.
93 *
94 * Part I for the initiator
95 * nonce1 : a nonce value to provide authentication
96 * initiator_i : the initiating party, who requested the SESSION_KEY
97 * session_key_i: the key for the session among the initiator and the responder
98 *
99 * Part II for the responder
100 * session_key_r: the key for the session among the initiator and the
101 * responder, intended for the responder
102 * initiator_r : the identity of the initiator intended for the responder
103 */
104 MESSAGE(EXAM, \
105     SESSION_KEY, \
106     int nonce1; \
107     int initiator_i ; \
108     int session_key_i ; \
109     int session_key_r ; \
110     int initiator_r );
111
112 /*
113 * REQUEST_SESSION
114 * -----
115 * After the initiator receives the SESSION_KEY message, it contacts
116 * the responder. The message includes the second part of the SESSION_KEY
117 * message, which contains the information intended for the responder
118 * and encrypted with its key.
119 *
120 * int session_key_r
121 * int initiator_r
122 */
123 MESSAGE(EXAM, REQUEST_SESSION, int session_key_r; int initiator_r);
124
125 /*
126 * CHALLENGE
127 * -----
128 * The responder extracts the session key and the initiator from the
129 * REQUEST_SESSION message. To find out whether the initiator can
130 * actually read messages encrypted with the session key, a CHALLENGE is

```



```

131 * sent.
132 *
133 * nonce2 : a nonce value to test whether the initiator can read messages
134 * encrypted with the session key
135 * responder: the identity of the responder
136 */
137
138 MESSAGE(EXAM, CHALLENGE, int nonce2; int responder);
139
140 /*
141 * CONFIRM
142 * -----
143 * The confirm message proves the responder, that the initiator can
144 * read messages encrypted with the session key. It includes the decremented
145 * nonce2 value from the CHALLENGE message.
146 *
147 * nonce2dec : nonce2, decremented by 1
148 */
149 MESSAGE(EXAM, CONFIRM, int nonce2dec);
150
151 /*
152 * DATA
153 * -----
154 * The final message, which does not really belong to the protocol. Its
155 * purpose is to use the established connection for a real communication.
156 * Access to and acceptance of the message data indicates the outcome
157 * of the protocol.
158 *
159 * message : the actual message
160 */
161 MESSAGE(EXAM, DATA, int message);
162
163
164 /*
165 * symbolic constants for the entities and nonce values
166 */
167 mtype = {ALICE, BOB, EVE, NOTARY,
168         ANNOUNCE,
169         NAB, NAE, N2B, NEB, N2E};
170
171 /*
172 * EXAM Entity Data
173 * -----
174 * Each entity has some data associated with it.
175 *
176 * secure : a secure channel to the Notary.
177 * notary_key: a shared secret with the Notary
178 */
179 typedef entity_data_EXAM {
180     chan secure;
181     int notary_key;
182 };
183

```



```

184 /*
185 * EXAM Session Data
186 * -----
187 * The session data includes the information which is negotiated among the
188 * initiator and the responder during the message exchange.
189 *
190 * initiator : the initiating end of the connction
191 * responder : the responding end of the connction
192 * nonce1 : the nonce to authenticate the notary against the initiator
193 * nonce2 : the nonce to authenticate the initiator against the responder
194 * session_key : the session key to encrypt the user data
195 * payload : the user data
196 */
197 typedef session_data.EXAM {
198     mtype initiator, responder;
199     int noncel, nonce2, session_key , payload;
200 };
201
202 /*
203 * EXAM Session
204 * -----
205 * A session stores all messages which are exchanged among communicating
206 * parties and a session data structure which ends up containing the
207 * negotiated parameters.
208 */
209 typedef session_EXAM {
210     session_data_EXAM sd;
211     msg_REQUEST_KEY request_key;
212     msg_SESSION_KEY session_key;
213     msg_REQUEST_SESSION request_session;
214     msg_CHALLENGE challenge;
215     msg_CONFIRM confirm;
216     msg_DATA data;
217 };
218
219 /*
220 * ANNOUNCE
221 * -----
222 * An announcement is sent at two points,
223 * 1. before the initiator sends the user data
224 * 2. after the responder receives the user data
225 *
226 * sender : the proclaimed sender
227 * receiver : the proclaimed receiver
228 * data : the user data involved
229 */
230 typedef msg_ANNOUNCE {
231     mtype sender, receiver;
232     int data;
233 };
234
235 /*
236 * CHECK

```



```

237 * -----
238 * A data structure to store information about a started protocol session.
239 * s_correct : indicates whether the sender specified its real identity
240 * data      : the user data involved
241 */
242 typedef check {
243     bit s_correct;
244     int data;
245 };
246
247
248 /*****
249 * Alice
250 * -----
251 * This process acts as the initiator .
252 *
253 * init_alice : the channel to receive the initialization data from the Notary
254 *****/
255 proctype alice(chan init_alice)
256 {
257     /*
258     * local data
259     */
260     entity_data_EXAM ed;
261     session_EXAM exam;
262     msg_ANNOUNCE announce;
263
264     /*
265     * receive the initializatin from the notary
266     *
267     * ed.secure      : a secure channel to the notary
268     * ed.notary_key : a shared secret with the notary
269     *
270     * exam.sd.initiator : self
271     * exam.sd.responder : the party to contact
272     * exam.sd.nonce1   : the nonce to authenticate the notary
273     * exam.sd.nonce2   : <undefined>
274     * exam.sd.session_key: <undefined>
275     * exam.sd.payload  : the data to send
276     */
277     init_alice ?ed, exam.sd;
278
279     /*
280     * build the announcement message, which will inform the notary of
281     * the intended communication. The message is sent to the Notary
282     * over the secure channel.
283     */
284     send_ANNOUNCE:
285     d_step {
286         announce.sender = ALICE;
287         announce.receiver = exam.sd.responder;
288         announce.data = exam.sd.payload;
289     };

```



```

290
291   ed.secure!ANNOUNCE(announce);
292
293   #ifdef REPLACE_ALICE
294     #include REPLACE_ALICE
295   #else
296     /*
297      * build the REQUEST_KEY message and send it.
298      * the message is not encrypted.
299      */
300   send_REQUEST_KEY:
301     d_step {
302       exam.request_key.nonce1 = exam.sd.nonce1;
303       exam.request_key.initiator = exam.sd.initiator;
304       exam.request_key.responder = exam.sd.responder;
305     };
306   SND(REQUEST_KEY, ALICE, NOTARY, exam.request_key);
307
308   /*
309    * Receive the SESSION_KEY message.
310    *
311    * the entire message is encrypted with the secret shared with the
312    * Notary, ed.notray_key. The session_key is extracted, so is the
313    * data which goes into SESSION_REQUEST.
314    * initiator and nonce1 must match the values sent to the notary.
315    */
316   receive_SESSION_KEY:
317     d_step {
318       RCV(SESSION_KEY, NOTARY, ALICE, exam.session_key);
319
320       exam.sd.session_key
321         = DECRYPT_S(exam.session_key.session_key_i, ed.notary_key);
322
323       exam.request_session.session_key_r
324         = DECRYPT_S(exam.session_key.session_key_r, ed.notary_key);
325
326       exam.request_session.initiator_r
327         = DECRYPT_S(exam.session_key.initiator_r, ed.notary_key);
328     };
329
330   /*
331    * block, if
332    *   - the initiator in the SESSION_KEY message is a different one
333    *   - the nonce is not the one sent to the Notary with the REQUEST_KEY
334    */
335   (DECRYPT_S(exam.session_key.initiator_i, ed.notary_key) == ALICE);
336   (DECRYPT_S(exam.session_key.nonce1, ed.notary_key) == exam.sd.nonce1);
337
338   /*
339    * Send the REQUEST_SESSION
340    * The entire message is already composed by receiving the SESSION_KEY
341    * message.
342    */

```



```

343 send_REQUEST_SESSION:
344     SND(REQUEST_SESSION, ALICE, exam.sd.responder, exam.request_session);
345
346     /*
347     *  Receive the CHALLENGE
348     *  The responder checks whether a message encrypted with the session key
349     *  can be read.
350     */
351 receive_CHALLENGE:
352     d_step {
353         RCV(CHALLENGE, eval(exam.sd.responder), ALICE, exam.challenge);
354         exam.sd.nonce2 = DECRYPT_S(exam.challenge.nonce2 , exam.sd.session_key);
355     };
356
357     /*
358     *  block if the challenge comes from a different party
359     */
360     (DECRYPT_S(exam.challenge.responder, exam.sd.session_key)
361      == exam.sd.responder);
362
363
364     /*
365     *  Send the CONFIRM message
366     *  ensures the responder that messages encrypted with the session
367     *  key can be read
368     */
369 send_CONFIRM:
370     exam.confirm.nonce2dec
371     = ENCRYPT_S(MODIFY(exam.sd.nonce2), exam.sd.session_key);
372
373     SND(CONFIRM, ALICE, exam.sd.responder, exam.confirm);
374
375     /*
376     *  The connection is set up authentication is provided and a key
377     *  is established .
378     *
379     *  Send the DATA message
380     *  contains the payload, encrypted with the session key
381     */
382 send_DATA:
383     exam.data.message = ENCRYPT_S(exam.sd.payload, exam.sd.session_key);
384
385     SND(DATA, exam.sd.initiator, exam.sd.responder, exam.data);
386 #endif
387
388 }
389
390 /*****
391 *  Bob
392 *  ---
393 *  This process acts as the responder.
394 *
395 *  init_bob : the channel to receive the initialization data from the Notary

```



```

396  *****/
397  proctype bob(chan init_bob)
398  {
399  /*
400   * local data
401   */
402  entity_data_EXAM ed;
403  session_EXAM exam;
404  msg_ANNOUNCE announce;
405
406  /*
407   * receive the initializatin from the notary
408   *
409   * ed.secure : a secure channel to the notary
410   * ed.notary_key : a shared secret with the notary
411   *
412   * exam.sd.initiator : <undefined>
413   * exam.sd.responder : self
414   * exam.sd.nonce1 : <undefined>
415   * exam.sd.nonce2 : the nonce to authenticate the initiator
416   * exam.sd.session_key: <undefined>
417   * exam.sd.payload : <undefined>
418   */
419  init_bob?ed, exam.sd;
420
421  #ifdef REPLACE_BOB
422  #include REPLACE_BOB
423  #else
424
425  /*
426   * receive the REQUEST_SESSION message
427   * it is encrypted with the key shared with the Notary.
428   *
429   * exam.request_session.session_key_r : the key for the session
430   * exam.request_session.initiator_r : the initiator of the session
431   */
432  receive_REQUEST_SESSION:
433  d_step {
434  RCV(REQUEST_SESSION, \
435  exam.sd.initiator, \
436  eval(exam.sd.responder), \
437  exam.request_session);
438
439  exam.sd.session_key
440  = DECRYPT_S(exam.request_session.session_key_r, ed.notary_key);
441
442  exam.sd.initiator
443  = DECRYPT_S(exam.request_session.initiator_r, ed.notary_key);
444  };
445
446  /*
447   * send the CHALLENGE message
448   * it contains a nonce value encrypted with the session key from

```



```

449  * the REQUEST_SESSION message
450  */
451  send_CHALLENGE:
452  d_step {
453      exam.challenge.nonce2 = ENCRYPT_S(exam.sd.nonce2, exam.sd.session_key);
454      exam.challenge.responder = ENCRYPT_S(BOB, exam.sd.session_key);
455  };
456
457  SND(CHALLENGE, BOB, exam.sd.initiator, exam.challenge);
458
459  /*
460  * receive the CONFIRM message
461  * contains the modified nonce2 from the CHALLENGE message
462  */
463  receive_CONFIRM:
464  RCV(CONFIRM, eval(exam.sd.initiator), eval(exam.sd.responder), exam.confirm);
465
466  /*
467  * block if the initiator could not provide the correctly modified nonce
468  */
469  (DECRYPT_S(exam.confirm.nonce2dec, exam.sd.session_key)
470   == MODIFY(exam.sd.nonce2));
471
472  receive_DATA:
473  d_step {
474      RCV(DATA, eval(exam.sd.initiator), BOB, exam.data);
475
476      exam.sd.payload
477      = DECRYPT_S(exam.data.message, exam.sd.session_key);
478  };
479
480  #endif /* REPLACE_BOB */
481
482  /*
483  * send the ANNOUNCE message
484  * this message tells the notary,
485  * – which data has been received
486  * – who is the receiver
487  * – who is supposed to be the sender
488  * the message will be sent over the secure channel
489  */
490  send_announce:
491  d_step {
492      announce.sender = exam.sd.initiator;
493      announce.receiver = BOB;
494      announce.data = exam.sd.payload;
495  };
496
497  ed.secure!ANNOUNCE(announce);
498  }
499
500  /*****
501  * Eve

```



```

502 * ---
503 * This process acts as the intruder.
504 * It may take the role of the initiator or the receiver or just
505 * send arbitrary messages.
506 *
507 * init_eve : the channel to receive the initialization data from the Notary
508 *****/
509 proctype eve(chan init_eve)
510 {
511     /*
512     * local data
513     */
514     entity_data.EXAM ed;
515
516     session_EXAM exam_actual, exam_init, exam_respond, exam_capture;
517     msg_ANNOUNCE announce;
518
519     mtype initiator, responder;
520
521     /*
522     * receive the initialization from the notary
523     *
524     * ed.secure : a secure channel to the notary
525     * ed.notary_key : a shared secret with the notary
526     *
527     * exam_init.sd.initiator : self
528     * exam_init.sd.responder : the party to contact
529     * exam_init.sd.nonce1 : the nonce to authenticate the notary
530     * exam_init.sd.nonce2 : <undefined>
531     * exam_init.sd.session_key: <undefined>
532     * exam_init.sd.payload : the data to send
533     *
534     * exam_respond.sd.initiator : <undefined>
535     * exam_respond.sd.responder : self
536     * exam_respond.sd.nonce1 : <undefined>
537     * exam_respond.sd.nonce2 : the nonce to authenticate the initiator
538     * exam_respond.sd.session_key: <undefined>
539     * exam_respond.sd.payload : <undefined>
540     */
541     init_eve?ed, exam_init.sd, exam_respond.sd;
542
543     /*
544     * if this macro is defined, a specific behavior will be
545     * executed, otherwise, a general approach is launched
546     */
547     #ifdef SPECIFIC_EVE
548     /*
549     * intercept the REQUEST_KEY, ...
550     */
551     receive_REQUEST_KEY:
552     RCV(REQUEST_KEY, ALICE, NOTARY, exam_capture.request_key);
553
554     /*

```



```

555     * ... modify it to specify EVE to be the responder and re-send it
556     */
557 send_REQUEST_KEY
558     exam_capture.request_key.responder = EVE;
559     SND(REQUEST_KEY, ALICE, NOTARY, exam_capture.request_key);
560
561     /*
562     * receive the REQUEST_SESSION and extract the session key
563     */
564 receive_REQUEST_SESSION:
565     RCV(REQUEST_SESSION, ALICE, BOB, exam_capture.request_session);
566
567     exam_capture.sd.session_key
568     = DECRYPT_S(exam_capture.request_session.session_key_r, ed.notary_key);
569
570     /*
571     * send the challenge, encrypted with the session key
572     */
573 send_CHALLENGE:
574     d_step {
575         exam_capture.challenge.nonce2
576         = ENCRYPT_S(exam_respond.sd.nonce2, exam_capture.sd.session_key);
577
578         exam_capture.challenge.responder
579         = ENCRYPT_S(BOB, exam_capture.sd.session_key);
580     };
581     SND(CHALLENGE, BOB, ALICE, exam_capture.challenge);
582
583
584     /*
585     * receive the CONFIRM and block if the modified nonce does not match
586     */
587 receive_CONFIRM:
588     RCV(CONFIRM, ALICE, BOB, exam_capture.confirm)
589
590     (DECRYPT_S(exam_capture.confirm.nonce2dec, exam_capture.sd.session_key)
591     == MODIFY(exam_respond.sd.nonce2));
592
593     /*
594     * send the announcement to the notary
595     */
596 send_ANNOUNCE:
597     d_step {
598         RCV(DATA, ALICE, BOB, exam_capture.data);
599
600         exam_capture.sd.payload
601         = DECRYPT_S(exam_capture.data.message, exam_capture.sd.session_key);
602
603         announce.sender = ALICE;
604         announce.receiver = BOB;
605         announce.data = exam_capture.sd.payload;
606     };
607     ed.secure!ANNOUNCE(announce);

```



```

608
609 #else /* EVE_SPECIFIC */
610
611 #ifdef REPLACE_EVE
612 #include REPLACE_EVE
613 #else
614
615 /*
616  * just loop around and send and receive arbitrary messages
617  */
618 do
619 /*
620  * select the type of the message and fill the fields .
621  * 1. with values from the captured, unreadable session
622  * 2. with arbitrary values
623  */
624
625 :: atomic {
626 /*****
627  * REQUEST_KEY
628  * -----
629  * encryption: none. -> no replay needed
630  * senders : ALICE, EVE
631  * receivers : NOTARY
632  *
633  * initiator : ALICE, EVE
634  * responder : BOB, EVE
635  * nonce1 : init ,respond,capture}.sd.nonce1
636  */
637 if
638 :: initiator = ALICE;
639 :: initiator = EVE;
640 fi ;
641
642 responder = NOTARY;
643
644 if
645 :: ( initiator != EVE || exam_init.sd.session_key == 0)
646 && (initiator != ALICE || (exam_capture.sd.session_key == 0
647 && exam_capture.request_key.nonce1 != 0))
648 /*
649  * the guard prevents a non-progress loop of
650  * REQUEST_KEY -> SESSION_KEY messages
651  *
652  *
653  * compose message of possible values
654  */
655 -> if
656 :: exam_actual.request_key.initiator = ALICE
657 -> exam_actual.request_key.responder = EVE;
658 :: exam_actual.request_key.initiator = ALICE
659 -> exam_actual.request_key.responder = BOB;
660 :: exam_actual.request_key.initiator = EVE

```



```

661         -> exam_actual.request_key.responder = BOB;
662         fi ;
663
664     -> if
665         :: exam_actual.request_key.nonce1 = exam_init.sd.nonce1;
666         :: exam_actual.request_key.nonce1 = exam_respond.sd.nonce1;
667         :: exam_actual.request_key.nonce1 = exam_capture.sd.nonce1;
668         fi ;
669     fi ;
670
671     SND(REQUEST_KEY, initiator, NOTARY, exam_actual.request_key);
672
673     /*
674     *  reset
675     */
676     initiator = 0;
677     responder = 0;
678     exam_capture.request_key.nonce1 = 0;
679     exam_actual.request_key.initiator = 0;
680     exam_actual.request_key.responder = 0;
681     exam_actual.request_key.nonce1 = 0;
682 };
683
684 :: atomic {
685 /*****
686 *
687 *  SESSION_KEY
688 *  this message is completely encrypted with a key which is
689 *  not available under any circumstances. Thus, it can only
690 *  be replayed.
691 *
692 *  encryption   : shared secret among initiator & Notary -> replay only
693 *  senders      : NOTARY
694 *  receivers    : ALICE, EVE
695 *
696 *  nonce1      :
697 *  initiator_i  :
698 *  session_key_i :
699 *  session_key_r :
700 *  initiator_r  :
701 */
702     responder = NOTARY;
703     if
704         :: initiator = ALICE;
705         :: initiator = EVE;
706     fi ;
707
708     if
709         :: exam_actual.session_key.nonce1
710            = exam_capture.session_key.nonce1;
711
712         exam_actual.session_key.initiator_i
713            = exam_capture.session_key.initiator_i ;

```



```

714
715     exam_actual.session_key.session_key_i
716         = exam_capture.session_key.session_key_i;
717
718     exam_actual.session_key.session_key_r
719         = exam_capture.session_key.session_key_r;
720
721     exam_actual.session_key.initiator_r
722         = exam_capture.session_key.initiator_r;
723     fi;
724
725     SND(SESSION_KEY, responder, initiator, exam_actual.session_key);
726
727     /*
728     *  reset
729     */
730     initiator = 0;
731     responder = 0;
732     exam_actual.session_key.nonce1      = 0;
733     exam_actual.session_key.initiator_i  = 0;
734     exam_actual.session_key.session_key_i = 0;
735     exam_actual.session_key.session_key_r = 0;
736     exam_actual.session_key.initiator_r  = 0;
737     };
738     :: atomic {
739     /*****
740     *
741     *  RQUEST_SESSION
742     *  this message is completely encrypted with a key which is
743     *  not available under any circumstances. Thus, it can only
744     *  be replayed.
745     *
746     *  encryption   : shared secret among responder & Notary -> replay only
747     *  senders       : ALICE, EVE
748     *  receivers      : BOB, EVE
749     *
750     *  session_key_r :
751     *  initiator_r  :
752     */
753     if
754     :: initiator = ALICE; responder = BOB;
755     :: initiator = ALICE; responder = EVE;
756     :: initiator = EVE; responder = BOB;
757     fi;
758
759     if
760     :: exam_actual.request_session.session_key_r
761         = exam_capture.request_session.session_key_r;
762
763     exam_actual.request_session.initiator_r
764         = exam_capture.request_session.initiator_r
765
766     :: exam_actual.request_session.session_key_r

```



```

767         = exam_init.request_session . session_key_r ;
768
769     exam_actual.request_session . initiator_r
770     = exam_init.request_session . initiator_r
771
772     :: exam_actual.request_session . session_key_r
773     = exam_respond.request_session.session_key_r;
774
775     exam_actual.request_session . initiator_r
776     = exam_respond.request_session . initiator_r
777     fi ;
778
779     SND(REQUEST_SESSION,      \
780         initiator , responder, \
781         exam_actual.request_session);
782
783     /*
784     * reset
785     */
786     initiator = 0;
787     responder = 0;
788     exam_actual.request_session . session_key_r = 0;
789     exam_actual.request_session . initiator_r   = 0;
790     };
791
792     :: atomic {
793     /*****
794     * CHALLENGE
795     *
796     * encryption: exam_{init,respond,capture}.sd.session_key
797     * sender      : BOB, EVE
798     * receiver   : ALICE, EVE
799     *
800     * nonce2     : exam_{init,respond,capture}.sd.nonce2
801     * responder  : BOB, EVE, exam_{init,repsond,capture}.sd.responder
802     */
803     if
804     :: initiator = ALICE; responder = BOB;
805     :: initiator = ALICE; responder = EVE;
806     :: initiator = EVE; responder = BOB;
807     fi ;
808
809     if
810     ::
811     /*
812     * captured message
813     */
814     exam_actual.challenge.nonce2 = exam_capture.challenge.nonce2
815     -> exam_actual.challenge.responder = exam_capture.challenge.responder
816
817     ::
818     /*
819     * arbitrary values

```



```

820     */
821     if
822         :: exam_actual.sd.session_key = exam_init.sd.session_key ;
823         :: exam_actual.sd.session_key = exam_respond.sd.session_key;
824         :: exam_actual.sd.session_key = exam_capture.sd.session_key;
825     fi ;
826     -> if
827         :: exam_actual.challenge.nonce2
828             = ENCRYPT_S(exam_init.sd.nonce2, exam_actual.sd.session_key);
829
830         :: exam_actual.challenge.nonce2
831             = ENCRYPT_S(exam_capture.sd.nonce2,exam_actual.sd.session_key);
832
833         :: exam_actual.challenge.nonce2
834             = ENCRYPT_S(exam_respond.sd.nonce2,exam_actual.sd.session_key);
835     fi ;
836     -> if
837         :: exam_actual.challenge.responder
838             = ENCRYPT_S(BOB, exam_actual.sd.session_key);
839         :: exam_actual.challenge.responder
840             = ENCRYPT_S(EVE, exam_actual.sd.session_key);
841     fi ;
842 fi ;
843
844     SND(CHALLENGE, responder, initiator, exam_actual.challenge);
845
846     /*
847     * reset
848     */
849     initiator = 0;
850     responder = 0;
851     exam_actual.sd.session_key      = 0;
852     exam_actual.challenge.nonce2    = 0;
853     exam_actual.challenge.responder = 0;
854 };
855 :: atomic {
856 /******
857 * CONFIRM
858 * this message is not encrypted, thus the captured data may be
859 * combined with arbitrary values.
860 *
861 * nonce2dec: {init,respond,capture}.sd.nonce2 - 1
862 */
863     if
864         :: initiator = ALICE; responder = BOB;
865         :: initiator = ALICE; responder = EVE;
866         :: initiator = EVE; responder = BOB;
867     fi ;
868
869     if
870         ::
871     /*
872     * captured message

```



```

873     */
874     exam_actual.confirm.nonce2dec = exam_capture.confirm.nonce2dec
875
876     ::
877     /*
878     * arbitrary values
879     */
880     if
881     :: exam_actual.sd.session_key = exam_init.sd.session_key ;
882     :: exam_actual.sd.session_key = exam_respond.sd.session_key;
883     :: exam_actual.sd.session_key = exam_capture.sd.session_key;
884     fi ;
885     -> if
886     :: exam_actual.confirm.nonce2dec
887         = ENCRYPT_S(exam_init.sd.nonce2 - 1,
888                   exam_actual.sd.session_key);
889
890     :: exam_actual.confirm.nonce2dec
891         = ENCRYPT_S(exam_respond.sd.nonce2 - 1,
892                   exam_actual.sd.session_key);
893
894     :: exam_actual.confirm.nonce2dec
895         = ENCRYPT_S(exam_capture.sd.nonce2 - 1,
896                   exam_actual.sd.session_key);
897     fi ;
898     fi ;
899
900     SND(CONFIRM, initiator, responder, exam_actual.confirm);
901     /*
902     * reset
903     */
904     initiator = 0;
905     responder = 0;
906     exam_actual.sd.session_key = 0;
907     exam_actual.confirm.nonce2dec = 0;
908 };
909
910 :: atomic {
911 /******
912 * DATA
913 * this message is encrypted, thus the captured data may be
914 * combined with arbitrary values.
915 *
916 * message: {init,respond,capture}.sd.payload
917 */
918     if
919     :: initiator = ALICE; responder = BOB;
920     :: initiator = ALICE; responder = EVE;
921     :: initiator = EVE; responder = BOB;
922     fi ;
923
924     if
925     :: /* captured message */

```



```

926     exam_actual.data.message = exam_capture.data.message
927   :: /* arbitrary values */
928     if
929       :: exam_actual.sd.session_key = exam_init.sd.session_key ;
930       :: exam_actual.sd.session_key = exam_respond.sd.session_key;
931       :: exam_actual.sd.session_key = exam_capture.sd.session_key;
932     fi ;
933   -> if
934     :: exam_actual.data.message
935       = ENCRYPT_S(exam_init.sd.payload, exam_actual.sd.session_key);
936
937     :: exam_actual.data.message
938       = ENCRYPT_S(exam_respond.sd.payload, exam_actual.sd.session_key);
939
940     :: exam_actual.data.message
941       = ENCRYPT_S(exam_capture.sd.payload, exam_actual.sd.session_key);
942     fi ;
943   fi ;
944
945   SND(DATA, initiator, responder, exam_actual.data);
946
947   /*
948   * reset
949   */
950   initiator = 0;
951   responder = 0;
952   exam_actual.sd.session_key = 0;
953   exam_actual.data.message = 0;
954 } ;
955
956 /*****
957 * Messages to receive in a session EVE is involved in.
958 * These are decryptable.
959 *
960 * SESSION_KEY : NOTARY -> EVE
961 * REQUEST_SESSION : ALICE -> EVE
962 * CHALLENGE : BOB -> EVE
963 * CONFIRM : ALICE -> EVE
964 * DATA : ALICE -> EVE
965 */
966
967 :: d_step {
968 /*****
969 * SESSION_KEY
970 * nonce1, initiator_i, session_key_i, session_key_r, initiator_r
971 */
972   RCV(SESSION_KEY, NOTARY, EVE, exam_init.session_key);
973
974   exam_init.sd.session_key
975     = DECRYPT_S(exam_init.session_key.session_key_i, ed.notary_key);
976
977   exam_init.request_session.session_key_r
978     = DECRYPT_S(exam_init.session_key.session_key_r, ed.notary_key);

```



```

979
980     exam_init.request_session . initiator_r
981         = DECRYPT_S(exam_init.session_key.initiator_r, ed.notary_key);
982
983     };
984
985     :: d_step {
986     /*****
987     * REQUEST_SESSION
988     */
989     RCV(REQUEST_SESSION, ALICE, EVE, exam_respond.request_session);
990
991     exam_respond.sd.session_key
992         = DECRYPT_S(exam_respond.request_session.session_key_r,
993                   ed.notary_key);
994
995     exam_respond.sd.initiator
996         = DECRYPT_S(exam_respond.request_session.initiator_r,
997                   ed.notary_key);
998
999     };
1000
1001     :: d_step {
1002     /*****
1003     * CHALLENGE
1004     */
1005     RCV(CHALLENGE, BOB, EVE, exam_init.challenge);
1006
1007     exam_init.sd.nonce2
1008         = DECRYPT_S(exam_init.challenge.nonce2, exam_init.sd.session_key);
1009
1010     };
1011
1012     :: RCV(CONFIRM, ALICE, EVE, exam_respond.confirm);
1013     /*****
1014     * CONFIRM
1015     */
1016
1017     :: d_step {
1018     /*****
1019     * DATA
1020     */
1021     RCV(DATA, ALICE, EVE, exam_respond.data);
1022
1023     exam_respond.sd.payload
1024         = DECRYPT_S(exam_respond.data.message, exam_respond.sd.session_key);
1025
1026     announce.sender = ALICE;
1027     announce.receiver = EVE;
1028     announce.data    = exam_respond.sd.payload;
1029     };
1030
1031     ed.secure!ANNOUNCE(announce);

```



```

1032     d_step {
1033         announce.sender = 0;
1034         announce.receiver = 0;
1035         announce.data = 0;
1036     };
1037
1038     /*****
1039     * Messages to receive in a session EVE is not involved in.
1040     * Some of them are clear, some encrypted.
1041     *
1042     * REQUEST_KEY : ALICE -> NOTARY not encrypted
1043     * SESSION_KEY : NOTARY -> ALICE encrypted with notary_key
1044     * REQUEST_SESSION : ALICE -> BOB encrypted with notary_key
1045     * CHALLENGE : BOB -> ALICE encrypted with session_key
1046     * CONFIRM : ALICE -> BOB encrypted with session_key
1047     * DATA : ALICE -> BOB encrypted with session_key
1048     */
1049
1050     :: RCV(REQUEST_KEY, ALICE, NOTARY, exam_capture.request_key)
1051     /*****
1052     * REQUEST_KEY
1053     */
1054     -> exam_capture.sd.nonce1 = exam_capture.request_key.nonce1;
1055
1056     :: RCV(SESSION_KEY, NOTARY, ALICE, exam_capture.session_key)
1057     /*****
1058     * SESSION_KEY
1059     */
1060
1061
1062
1063     :: RCV(REQUEST_SESSION, ALICE, BOB, exam_capture.request_session);
1064     /*****
1065     * REQUEST_SESSION
1066     */
1067     -> exam_capture.sd.session_key
1068         = DECRYPT_S(exam_capture.request_session.session_key_r, ed.notary_key);
1069
1070     :: RCV(CHALLENGE, BOB, ALICE, exam_capture.challenge);
1071     /*****
1072     * CHALLENGE
1073     */
1074     if
1075         :: exam_capture.sd.nonce2 = DECRYPT_S(exam_capture.challenge.nonce2,
1076                                             exam_capture.sd.session_key);
1077
1078         :: exam_capture.sd.nonce2 = DECRYPT_S(exam_capture.challenge.nonce2,
1079                                             exam_init.sd.session_key);
1080
1081         :: exam_capture.sd.nonce2 = DECRYPT_S(exam_capture.challenge.nonce2,
1082                                             exam_respond.sd.session_key);
1083     fi;
1084

```



```

1085     :: RCV(CONFIRM,      ALICE, BOB,  exam_capture.confirm);
1086     /*****
1087     *  CONFIRM
1088     */
1089
1090     :: atomic {
1091     /*****
1092     *  DATA
1093     */
1094         RCV(DATA,          ALICE, BOB,  exam_capture.data);
1095         if
1096             :: exam_capture.sd.payload = DECRYPT_S(exam_capture.data.message,
1097                                                    exam_capture.sd.session_key);
1098
1099             :: exam_capture.sd.payload = DECRYPT_S(exam_capture.data.message,
1100                                                    exam_init.sd.session_key);
1101
1102             :: exam_capture.sd.payload = DECRYPT_S(exam_capture.data.message,
1103                                                    exam_respond.sd.session_key);
1104         fi ;
1105     };
1106
1107     :: atomic {
1108     /*****
1109     *  ANNOUNCEMENT
1110     */
1111         if
1112             :: announce.sender  = ALICE;
1113             -> announce.receiver = BOB;
1114             :: announce.sender  = ALICE;
1115             -> announce.receiver = EVE;
1116             :: announce.sender  = EVE;
1117             -> announce.receiver = BOB;
1118         fi ;
1119
1120         if
1121             :: announce.data    = exam_init.sd.payload;
1122             :: exam_capture.sd.payload
1123             -> announce.data    = exam_capture.sd.payload;
1124             :: exam_respond.sd.payload
1125             -> announce.data    = exam_respond.sd.payload;
1126         fi ;
1127
1128         ed.secure!ANNOUNCE(announce);
1129     };
1130 };
1131 od;
1132 #endif /* REPLACE_EVE */
1133 }
1134 #endif /* SPECIFIC_EVE */
1135
1136 /*****
1137 *  Notary

```



```

1138 * -----
1139 * The Notary serves multiple purposes. It
1140 *
1141 * 1. initializes the other entities . Thus, it can
1142 * establish shared secrets .
1143 *
1144 * 2. decides the protocol outcome.
1145 * The Notary is able to tell the real origin of a
1146 * message. Thus, it can tell whether an entity
1147 * – received a message by the real receiver intended
1148 * for itself -> success
1149 * – gained access to data intended for someone else
1150 * -> capture.
1151 * – pretended to be someone else sender / receiver
1152 * -> impersonate
1153 *
1154 *****/
1155 active proctype notary()
1156 {
1157 /*
1158 * Each process will be handed a 'secure' channel.
1159 * These channels are secure, in so far as they
1160 * are created within the Notary process and no global
1161 * reference exists . These channels carry announcement
1162 * messages. The purpose of these messages is to
1163 * enable the Notary to determine the outcome of a
1164 * protocol session. The Notary is able to tell the
1165 * real origin of a message by the channel on which
1166 * it arrives .
1167 *
1168 */
1169 chan secure[3] = [0] of {mtype, msg_ANNOUNCE};
1170
1171
1172 /*
1173 * Due to limitations in parameter passing,
1174 * the processes are initialized through special channels,
1175 * Each process is given a channel and its first statement
1176 * receives the initializing structures
1177 */
1178 chan ini_alice = [0] of {entity_data_EXAM, session_data_EXAM};
1179 chan ini_bob = [0] of {entity_data_EXAM, session_data_EXAM};
1180 chan ini_eve = [0] of {entity_data_EXAM,
1181 session_data_EXAM,
1182 session_data_EXAM};
1183
1184 entity_data_EXAM ed[MAX_PROC];
1185
1186 session_data_EXAM sd_alice_bob, sd_alice_eve,
1187 sd_bob_alice,
1188 sd_eve_bob, sd_eve_alice ;
1189
1190 d_step {

```



```

1191  /*
1192  *   initialize all entity data structures
1193  *   - secure      : a secure channel to the notary
1194  *   - notary_key: secret symmetric key, shared with the notary
1195  */
1196  ed[IALICE].secure      = secure[0];
1197  ed[IALICE].notary_key = GEN_KEY_SYM(5);
1198
1199  ed[I BOB ].secure      = secure[1];
1200  ed[I BOB ].notary_key = GEN_KEY_SYM(17);
1201
1202  ed[I EVE ].secure      = secure[2];
1203  ed[I BOB ].notary_key = GEN_KEY_SYM(23);
1204
1205  /*
1206  *   initialize the session datas
1207  *
1208  *   an initiating session needs
1209  *   - initiator : who contacts
1210  *   - responder : who should be contacted
1211  *   - nonce1   : to authenticate the Notary
1212  *   - payload  : the data to send
1213  *   these are alice_bob , alice_eve and eve_bob
1214  */
1215
1216  sd_alice_bob . initiator = ALICE;
1217  sd_alice_bob . responder = BOB;
1218  sd_alice_bob . nonce1   = RND(NAB);
1219  sd_alice_bob . payload  = 35;
1220
1221
1222  sd_alice_eve . initiator = ALICE;
1223  sd_alice_eve . responder = EVE;
1224  sd_alice_eve . nonce1   = RND(NAE);
1225  sd_alice_eve . payload  = 45;
1226
1227  sd_eve_bob . initiator   = EVE;
1228  sd_eve_bob . responder   = BOB;
1229  sd_eve_bob . nonce1     = RND(NEB);
1230  sd_eve_bob . payload    = 77;
1231
1232  /*
1233  *   a responding session needs
1234  *   - nonce2 : to authenticate the initiator
1235  */
1236  sd_bob_alice . responder = BOB;
1237  sd_bob_alice . nonce2   = RND(N2B);
1238
1239  sd_eve_alice . responder = EVE;
1240  sd_eve_alice . nonce2   = RND(N2E);
1241  };
1242
1243  atomic {

```



```

1244  /*
1245  *  start the processes ...
1246  *  ... and initialize them
1247  */
1248
1249
1250  run alice( ini_alice );
1251  if
1252  :: ini_alice !ed[I_ALICE], sd_alice_bob;
1253  :: ini_alice !ed[I_ALICE], sd_alice_eve;
1254  fi ;
1255
1256  run bob(ini_bob)      -> ini_bob!ed[I_BOB], sd_bob_alice;
1257
1258  run eve(ini_eve)     -> ini_eve!ed[I_EVE], sd_eve_bob, sd_eve_alice ;
1259
1260  };
1261
1262  /*
1263  *  begin with the actual notary functionality
1264  *
1265  *  to store the result of the first ANNOUNCEMENT
1266  *  message. The
1267  *  - [index] : identifies the initiator and the
1268  *               responder for message exchange
1269  *               must match in both ANNOUNCEMENTS
1270  *  - s_correct: origin == initiator
1271  *  - data      : the content of the message,
1272  *               must match in both ANNOUNCEMENTS
1273  */
1274  check storage[MAX_PROC2];
1275
1276  /*
1277  *  temporary values for receiving / sending data
1278  */
1279  msg_ANNOUNCE ann;
1280  msg_REQUEST_KEY rq_k;
1281  msg_SESSION_KEY s.k;
1282
1283  mtype entity      = 0;
1284  int session_count = 0;
1285  int key_count     = 12;
1286  int new_key;
1287
1288  #define INITIATOR_KEY ed[ENTITY_INDEX(rq_k.initiator)].notary_key
1289  #define RESPONDER_KEY ed[ENTITY_INDEX(rq_k.responder)].notary_key
1290
1291  do
1292  :: atomic {
1293      /*
1294      *  wait for a request ...
1295      */
1296      RCV(REQUEST_KEY, entity, NOTARY, rq_k)

```



```

1297
1298     /*
1299     * ... create a new session key ...
1300     */
1301     key_count++;
1302     new_key = RND(57 + key_count);
1303
1304     /*
1305     * ... build the response ...
1306     */
1307     s_k.nonce1      = ENCRYPT_S(rq.k.nonce1, INITIATOR_KEY);
1308     s_k.initiator_i = ENCRYPT_S(rq.k.initiator, INITIATOR_KEY);
1309     s_k.session_key_i = ENCRYPT_S(new_key, INITIATOR_KEY);
1310
1311     s_k.initiator_r
1312     = ENCRYPT_S(ENCRYPT_S(rq.k.initiator, RESPONDER_KEY), INITIATOR_KEY);
1313     s_k.session_key_r
1314     = ENCRYPT_S(ENCRYPT_S(new_key, RESPONDER_KEY), INITIATOR_KEY);
1315
1316     /*
1317     * ... and send it
1318     */
1319     SND(SESSION_KEY, NOTARY, entity, s_k);
1320
1321     /*
1322     * clear the values
1323     */
1324     s_k.nonce1      = 0;
1325     s_k.initiator_i = 0;
1326     s_k.session_key_i = 0;
1327     s_k.initiator_r = 0;
1328     s_k.session_key_r = 0;
1329     entity = 0;
1330     new_key = 0;
1331 };
1332 #undef INITIATOR_KEY
1333 #undef RESPONDER_KEY
1334
1335 #define SESSION (ENTITY_INDEX(ann.sender) \
1336               + (MAX_PROC * ENTITY_INDEX(ann.receiver)))
1337 :: atomic {
1338     /*
1339     * wait for ANNOUNCEMENTS on the secure channels
1340     * the channel determines the entity
1341     */
1342     if
1343     :: ed[I.ALICE].secure?ANNOUNCE(ann) -> entity = ALICE;
1344     :: ed[I.BOB ].secure?ANNOUNCE(ann) -> entity = BOB;
1345     :: ed[I.EVE ].secure?ANNOUNCE(ann) -> entity = EVE;
1346     fi;
1347
1348     if
1349     :: ( ann.data != 0)

```



```

1350      /*
1351      * only accept messages with non-zero data
1352      */
1353      -> if
1354          :: ( storage[SESSION].data == 0 )
1355          /*
1356          * nothing stored so far -> store the first ANNOUNCEMENT
1357          */
1358          -> storage[SESSION].s_correct = (entity == ann.sender)
1359          -> storage[SESSION].data = ann.data
1360          -> session_count++
1361
1362      /*
1363      * prevent entity from fooling itself
1364      */
1365      -> (entity == ann.sender || entity != ann.receiver)
1366
1367          :: (( storage[SESSION].data != 0 )
1368          && (storage[SESSION].data == ann.data)
1369          && (entity != ann.sender))
1370      /*
1371      * second ANNOUNCEMENT, matching data
1372      */
1373      -> if
1374          :: ( entity == ann.receiver )
1375          /*
1376          * the entity is the intended receiver
1377          */
1378          -> if
1379              :: ( storage[SESSION].s_correct )
1380              /*
1381              * the sender of the first announcement was correct
1382              */
1383              -> session_count--;
1384              -> storage[SESSION].data = 0;
1385              -> if
1386                  :: ( session_count == 0 ) -> goto success;
1387                  :: else -> skip
1388                  fi ;
1389              :: else -> goto impersonation;
1390              fi ;
1391          :: else
1392          /*
1393          * the entity is not the intended receiver
1394          */
1395          -> if
1396              :: ( storage[SESSION].s_correct ) -> goto capture;
1397              :: else -> goto failure;
1398              fi ;
1399          fi ;
1400          :: else -> goto failure;
1401          fi ;
1402      :: else -> goto failure;

```



```
1403         fi;
1404         entity = 0;
1405 #undef SESSION
1406     };
1407     od;
1408
1409 success :      false ;
1410 failure :      false ;
1411 capture :      false ;
1412 impersonation: false ;
1413 abort :        false ;
1414 }
1415
1416 never {
1417     do
1418     :: notary[0]@OUTCOME -> break;
1419     :: skip;
1420     od;
1421 }
```

Listing A.2: The complete EXAM protocol (exam.prl).



Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Matthias Papesch)