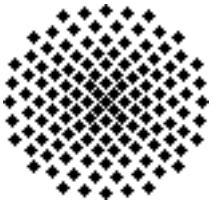

# University of Stuttgart

Faculty of Computer Science

**Date:** 05.12.02

**CR-Classification:** C.2.2, D.2.4, D.4.6, I.6.3, I.6.7, K.3.1

Technical Report No. 2002/09

## ProductivE - A PROMELA driven constructivist environment to learn security protocols

Matthias Papesch
Cora Burger

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

**Abstract**

To overcome problems in understanding security protocols, a constructivist approach is used. Learners are enabled to experiment with a given protocol, either alone or in a team of co-learners. This means, to use pre-defined and automatically generated PROMELA building blocks for all communicating parties and to support students in playing some roles in an interactive way. To explain these building blocks more detailed, the Needham-Schroeder-Public-Key authentication protocol is given as an example.

All building blocks were realized by means of the preprocessor. A proper combination of simulation and visualization components cares for hiding unnecessary details and restricts on the essentials. The SPIN validator is used to check the correctness of students' solutions or generate trail files to give a hint in case of a quandary. To allow collaborative experiments, the whole facility is embedded into a framework for application sharing. The latter is equipped with components for annotation and note taking as well as for recording.

# 1 Introduction

When teaching computer networks and distributed systems, security is one big issue since it is required heavily to protect privacy, identity, autonomy and resources of people in the electronic world. The challenge behind consists in performing this task in front of hackers where the form of attack is hard to foresee. Consequently, protocols being meant to achieve security are not understood in a straightforward manner. Learners have to find out about possible flaws and corrections when trying to comprehend how these protocols work.

With a teacher at hand for explanation 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 similar problem, the one of designing security protocols has already been studied by [5] and [6]. They used SPIN, a model checking tool based on the specification language PROMELA [4] to tackle the complexity of such protocols. Due to the similarities between design and teaching situation, it is obvious to build a learning environment as well on some model checking tool like SPIN. This tool offers the required functionality for the most part. First of all, for a given protocol, people can gain an impression of its behavior through guidance of trail files being generated automatically by the model checker. This can be used for correct working as well as for protocol flaws. In a next step, learners should be able to examine various paths on their own, maybe sometimes even by stepping back and forth in one or the other way. With regard to security protocols, this could mean to confront learners with a simulation of the protocol and, e.g., have them launch different kinds of attacks on the protocol. This can be achieved by literally turning them into a process which is part of the simulation. Hence, with suitable support, students can act on behalf of this process. They may create local variables and have access to all global symbols. Thus, they may receive messages from or send arbitrary messages to global channels at any instant. Certainly, this possibility requires access to process-private data to be restricted, since otherwise the task of breaking the protocol turns into a mere reading assignment without mental effort.

When playing this kind of games in a collaborative manner, where each student controls one out of multiple processes, this can even come close to being a fun thing to do. As a consequence, the effectiveness of learning will be augmented. For instance, let three students engage in roles being typical for security protocols. Following the tradition in literature (cf. [10]), their names be *Alice*, *Bob*, and *Eve* as well as an already running trusted server acting as a *Notary* for all of them. A possible assignment might consist of Alice and Bob having to figure out how to communicate privately. As always, Eve must try to get hold of their conversation or disturb it. In a more general sense, learners have to find out solutions to close gaps. The latter have to be checked automatically, e.g. against correctness requirements defined for this protocol.

To satisfy the demands of teaching in general and of security protocols in particular, it doesn't suffice to simply have some add-on to the original concepts of PROMELA and SPIN. In fact, the whole system has to be embedded into a collaborative learning environment for private studies. The focus of this report lies on elaborating all these facts more detailed, thus deriving a generic way to support the scenarios as described above. To facilitate the treatment of PROMELA specification for students, the next section mainly describes PROMELA building blocks . Their usage will be concretized in an example in Sect. 3 (the complete specification can be found in appendix 7). After that, Sect. 4 demonstrates, how the teaching facility around SPIN can be combined with a suitable visualization facility as well as be integrated into a framework for application sharing to enable collaborative usage. The report closes with related work, conclusions and outlook.

# 2 PROMELA Building Blocks for Security Protocols

At first sight, PROMELA is nothing else than a derivative from the C programming language where main is called init and additional constructs exist for processes (proctypes), connections (channels), and for the declaration of messages (mtypes). But since it serves the special purpose of specifying and validating communication protocols, some of its concepts are different from C and need special treatment. These are, e. g. facilities to describe correctness requirements by means of assertions and temporal claims (for details to PROMELA, please refer to [4]). Hence, when designing an environment to support teaching in the area of security protocols, these pecularities have to be taken into account.

To enable an interactive or even collaborative mode, there is urgent need for on-the-fly usage of PROMELA commands and for the generation of intruder models. Students have to concentrate on the security protocol at hand instead of being bothered with PROMELA specifics. As a consequence, these building blocks have to be designed rather carefully. As explained more detailed in the following, templates are needed for entities, for data and for sessions to be able to complete message templates. By means of these templates, learners can prepare and execute send and receive statements in a comfortable way by mouse clicks only, without a single error-prone keyboard input. More experienced students can even try to specify protocols partially or completely by themselves. For this case again, they would rely on suitable templates.

In PROMELA, some of these templates can be realized by means of pre-processor macros. All templates are described more detailed in the following.

## 2.1 Entities and Data Templates

The entities involved in a security protocol session are typically always the same. An initiator (Alice), a responder (Bob), an intruder (Eve) and last but not least a trusted Notary (cf. Sect. 1). Since entities are equipped with attributes, it may become handy to have their identification as an array index, to ease access, e.g., when looking up the public key of an entity. On the other hand, such identifications appear in messages. Hence, it would be of great help to have a symbolic constant represent an entity. To this end, a pre-processor macro combines both by allowing the definition of entities as `mtype`s as well as their usage as array indices. This is subject to the condition only, that they must be declared consecutively.

Entities may also comprise certain data. To make sure that all entities have equal data associated with same names, a template for the entities' data structure can be defined. Below a minimum sample for a public key infrastructure is shown, which will be used later in Sect. 2.6.

```
#define MAX_PROC  4
#define ENTITY_INDEX(e) (e % MAX_PROC)
mtype = {ALICE, BOB, EVE, NOTARY};

typedef entity_data_NSPK {
  chan secure;          /* the secure channel to the notary */
  int sk, pk[MAX_PROC]; /* the secret key / public keys     */
};
```

## 2.2 Session Templates

All messages being exchanged between two parties, are referred to as a *session*. By means of typedef, a data structure is introduced which contains an entry for each message of the session. The initiator and responder both can store the history and have access to message contents via the same symbols. To reflect the direction of a certain message in this table, an additional index is appended to the message name. According to agreement, an odd index represents `initiator` → `responder`.

Besides this, 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.

While the types of sessions are protocol specific, a limited set of sessions can occur only, namely initiator-responder, initiator-intruder and intruder-responder. The view of initiator and responder is restricted to the single session they are involved. In contrast to this, according to its nature, the intruder has access to any session.

## 2.3 Templates and Processing of Messages

For interactive games, learners must be able to act on behalf of the above mentioned entities. As soon as they actively participate in a simulation run, the content of messages must be in a form which is readable and meaningful (as long as it is not encrypted). On the other hand, for better treatment during simulation and validation, messages should be represented by integers. To solve this contradiction, two distinct kinds of mapping are introduced. Both are mapping integer keys to string messages and vice versa, one for the human readable and the other one for the encrypted case. This cares for keeping the encryption state consistent.

Also, for interactive simulation, each message field has to be denoted with its meaning to enable unambiguous completion by learners. This can be achieved by using a data structure for each message type, which is appended to the necessary fields, which allow selective reception (message type, sender and receiver). A designated pre-processor macro simplifies these definitions and even creates the necessary channel at the same time. This does no harm, since only a few messages are exchanged being in a different format each. Two more useful macros encapsulate the send and receive action.

```
#define SND(m, s, r, d) net_##m!m(s, r, d);
#define RCV(m, s, r, d) net_##m?m(s, r, d);
#define MESSAGE(m, d)   mtype { m  };  typedef msg_##m { d; };   \
        chan net_##m = [0] of {mtype, mtype, mtype, msg_##m}
```

To exchange messages, one has to proceed as follows:

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.
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.

## 2.4 Special Validation and Simulation Demands

Looking more detailed at the data involved, leads to a further aspect. Whereas for 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 small 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 are shown below.

```
#define RND(x) x
#define GEN_KEY_SYM(id)      RND(id)
#define ENCRYPT_S(val, key) ((val ^ ((key << 8) & ~255)))
#define DECRYPT_S(val, key) ((val ^  (key << 8)))
#define GEN_KEY_PUB(id)     (~id & 255)
#define GEN_KEY_SEC(id)     ( id & 255)
#define ENCRYPT_A(val, key) ((val | (( key & 255) << 8) ))
#define DECRYPT_A(val, key) ((val ^ ((~key & 255) << 8) ))
#define SIGN(val, key)      ENCRYPT_A(val, key)
#define VRFY(val, key)      DECRYPT_A(val, key)
```

## 2.5 Model of an Omnipotent Intruder

The mechanisms as described above, mainly support the case of learners playing by themselves. In contrast to this, intruder models are autmatically generated to take over this functionality in general. As such, they serve demonstration as well as evaluation purposes to find out about the correctness of a given protocol or solution as delivered by learners. For the generation of intruders, a complete solution is available by the work of [6] and [9] who have already examined intruder models extensively.

As laid out in the subsection about session templates, an intruder is aware of all running sessions by storing the content of any message received into its table of session data. 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.

## 2.6 The Notary

The notary is needed in any case, for simulations with learners playing interactively as well as with a generated intruder. It is an unbribable entity whose main task consists in deciding about success or failure of attacks. To exclude any manipulation from the side of learners acting on behalf of some entity, the notary cares for avoiding the exposure of global data being critical to its decision. To this end, it replaces the `init` process, since it must be the only active process in the specification. The following two functions are performed by the notary:

*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 by other entities. 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 authenticity and stores the result as well as all other values for this initiator-responder combination.

Table 1: Derivation of outcome from announcement messages.

| Sender | Initiator | Responder | Receiver | Outcome |
|---|---|---|---|---|
| *Alice* | *Alice* | *Bob* | *Bob* | |
| *Alice* | *Alice* | *Eve* | *Eve* | success |
| *Eve* | *Eve* | *Bob* | *Bob* | |
| *Eve* | *Alice* | *Bob* | *Bob* | impersonation |
| *Alice* | *Alice* | *Bob* | *Eve* | capture |

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. 1.

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.

```
active proctype notary() {
[...]
success:       false;
failure:       false;
capture:       false;
impersonation: false;
}
```

```
never {
  do
  :: notary[0]@OUTCOME -> break;
  :: skip;
  od;
}
```

# 3 Deriving an example

The simplified Needham-Schroeder-Public-Key authentication protocol (NSPK) (cf. [6]) is a useful example to demonstrate how a specification can be derived for teaching purposes. A specification of this protocol is existing already in the PROMELA Database [**?**] but has to be modified according to the concepts as explained in the last section. In NSPK, the participating entities Alice and Bob exchange messages as shown in Tab. 2. Since presenting the entire model, including Eve and Notary, would waste too much space, we will just focus on the main ideas. The complete version can be found in appendix 7.

Table 2: Message flow of the NSPK-protocol

| Sender | Message Name | Message Format | Receiver |
|---|---|---|---|
| *Alice* | `REQUEST_1` | $\{N_a, Alice\}^{PK_{Bob}}$ | *Bob* |
| *Bob* | `CHALLENGE_2` | $\{N_a, N_b\}^{PK_{Alice}}$ | *Alice* |
| *Alice* | `CONFIRM_3` | $\{N_b\}^{PK_{Bob}}$ | *Bob* |
| *Alice* | `DATA_5` | $\{*\}^{N_b}$ | *Bob* |

**Step 1: Message Flow** Based on data involved in the message flow, session members, entity and session data structures are constructed.

```
MESSAGE(REQUEST_1,  int nonce1; \
                    int party);
MESSAGE(CHALLENGE_2,int nonce1; \
                    int nonce2);
MESSAGE(CONFIRM_3,  int nonce2);
MESSAGE(DATA_5,     int message);
```

```
typedef session_NSPK {
  session_data_NSPK sd;
  msg_REQUEST_1   request;
  msg_CHALLENGE_2 challenge;
  msg_CONFIRM_3   confirm;
  msg_DATA_5      data;
};
```

**Step 2: Entity Data.** Each NSPK entity needs the necessary data for public key encryption already indicated in Sec. 2.1.

**Step 3: Session Data.** A NSPK session is established between an initiator and responder. Since each party knows its own identity, it needs only to remember its counterpart. During each session, two nonces and a payload message are exchanged and stored into the session data.

```
typedef session_data_NSPK {
  int party;          /* the other party in the session     */
  int nonce1, nonce2; /* nonces of initiator and responder  */
  int payload;        /* the message to be sent             */
};
```

**Step 4: Initiator** By convention, the initiator sends messages ending with an odd index and waits for those with an even index. Message fields are filled with values from entity and session data after encryption has been applied to them. The sample below shows how an initiator starts the protocol session by sending a message. It is noteworthy to see, that the Notary is informed before any other action is taken via the private channel `ed.secure`.

```
#define ENC_A(val) ENCRYPT_A(val, ed.pk[ENTITY_INDEX(ini.sd.party)])
proctype alice(chan init_alice) {
  msg_ANNOUNCE      announce;
  entity_data_NSPK  ed;  session_NSPK      ini;
  init_alice?ed, ini.sd, scratch_nspk; /* accept initialization    */
send_ANNOUNCE: d_step {                /* build announcement ..    */
  announce.receiver = ini.sd.party;
  announce.sender   = ALICE;        announce.data = ini.sd.payload;}
  ed.secure!ANNOUNCE(announce);        /* .. and send it           */
send_REQUEST_1: d_step {               /* build the request ..     */
    ini.request.nonce1 = ENC_A(ini.sd.nonce1);
    ini.request.party  = ENC_A(ALICE);         }
                                       /* .. and send it           */
  SND(REQUEST_1, ALICE, ini.sd.party, ini.request);
  [...]
};
```

**Step 5: Responder** For the responder, a piece of code is shown which contains the reception of a message, a condition check and a notification to the Notary.

```
proctype bob(chan init_bob) { [...]
  init_bob?ed, res.sd, scratch_nspk;   /* accept initialization    */
[...]
receive_CONFIRM_3:                     /* .. accept the confirm .. */
  RCV(CONFIRM_3, eval(res.sd.party), BOB, res.confirm);
                                       /* .. nonce2 verification   */
  (DECRYPT_A(res.confirm.nonce2, ed.sk) == res.sd.nonce2);
receive_DATA_5: d_step {               /* .. accept the data ..    */
  RCV(DATA_5, eval(res.sd.party), BOB, res.data);
                                       /* .. and unpack it         */
```

```
  res.sd.payload = DECRYPT_S(res.data.message, res.sd.nonce2);  };
send_ANNOUNCE: d_step {     announce.sender   = res.sd.party;
  announce.receiver = BOB;  announce.data     = res.sd.payload; };
  ed.secure!ANNOUNCE(announce);
};
```

**Step 6: Intruder** The intruder functionality has already been covered in detail in [6], and is only slightly modified to fit the data structures used in this paper. Basically, the process generated for Eve needs to keep up with three sessions. For all messages, the address fields may be filled with randomly chosen entities and there are three `session_data` to use values from. Moreover, messages from a captured session may be replayed as a whole. For a complete implementation see Appendix A. The reader may notice, that there are actually *four* sessions defined. The extra one is used temporarily in send/receive statements, so as not to overwrite any values in a stored session.

**Step 7: Notary** As explained above, the notary creates and initializes all processes and data structures at first. After that, it waits for incoming `ANNOUNCE`-messages. On reception, it determines the outcome of the simulation according to each message (cf. Tab. 1).

```
#define I_ALICE  ENTITY_INDEX(ALICE)
#define CP_PK(src, dst) ed[dst].pk[src] = ed[src].pk[src];
#define SESSION (ENTITY_INDEX(ann.sender)                    \
                 + (MAX_PROC * ENTITY_INDEX(ann.receiver)));
typedef check {  bit  s_correct;  int  data; };

active proctype notary() {
  chan secure[MAX_PROC] = [0] of {mtype, msg_ANNOUNCE};
  chan ini[MAX_PROC]    = [0] of {entity_data_NSPK,
                                  session_data_NSPK,
                                  session_data_NSPK};
  entity_data_NSPK ed[MAX_PROC];
  session_data_NSPK sd_alice_bob, sd_alice_eve,
                    sd_bob_alice,
                    sd_eve_bob, sd_eve_alice ;
   /* entity data initialization */
   ed[I_ALICE ].secure = secure[0]; ed[I_ALICE ].sk = GEN_KEY_PUB(5);
   ed[I_ALICE ].pk[I_ALICE ] = GEN_KEY_SEC(5);
   CP_PK(I_ALICE, I_BOB); CP_PK(I_ALICE, I_EVE);
[...]
   /* session data initialization */
   sd_alice_bob.party   = BOB;  sd_alice_bob.nonce1  = RND(NAB);
   sd_alice_bob.payload = 35;
[...]
  atomic {                 run alice(ini[I_ALICE]);
    run bob  (ini[I_BOB]); run eve  (ini[I_EVE  ]);
    ini[I_BOB]!ed[I_BOB], sd_bob_alice, scratch_nspk;
    ini[I_EVE]!ed[I_EVE], sd_eve_bob,   sd_eve_alice;
    if :: ini[I_ALICE]!ed[I_ALICE], sd_alice_bob, scratch_nspk;
       :: ini[I_ALICE]!ed[I_ALICE], sd_alice_eve, scratch_nspk;
    fi;
  [...] /* outcome determination */
  check storage[MAX_PROC2];   msg_ANNOUNCE ann;
  mtype entity      = 0;      int session_count = 0;
  do :: atomic { if :: ed[I_ALICE].secure?ANNOUNCE(ann)
                    -> entity = ALICE;
                    [...]
                 fi;
```

```
  -> if :: (ann.data != 0)
        -> if :: (storage[SESSION].data == 0 && ann.data != 0)
              ->  storage[SESSION].s_correct
                     = (entity == ann.sender);
              ->  storage[SESSION].data      = ann.data
              ->  session_count++;
              :: ((storage[SESSION].data != 0 )
                 &&  (storage[SESSION].data == ann.data)
                 &&  (entity != ann.sender))
              -> if :: (entity == ann.receiver)
                    -> if :: (storage[SESSION].s_correct)
                          -> session_count--;
                          -> storage[SESSION].data = 0;
                          -> if :: (session_count == 0)
                                -> goto success;
                                :: else -> skip
                             fi;
                          :: else -> goto impersonation;
                       fi;
                    :: else -> if :: (storage[SESSION].s_correct)
                                  -> goto capture;
                                  :: else -> goto failure;
                               fi;
                 fi;
              :: else -> goto failure;
           fi;
        :: else -> goto failure;
     fi;
   -> entity  = 0; };
  od;
 [...]  }
```

# 4 The Environment

The PROMELA driven constructivist environment (ProDuctivE) aims at supporting the generation of PROMELA models to enable an extensive and collaborative investigation of security protocols. Besides offering a user interface to SPIN, it integrates the existing projects HiSAP and NUSS (see explanations below).

## 4.1 Architecture

An architectural overview is given in Fig. 1. The general setup is a client-server one. The server performs the simulation of the PROMELA model, projects the structure into a simplified view and invokes the SPIN validator if required. Hence, SPIN compliance can be achieved. The resulting trail file can then be used to generate a certain execution sequence. Storing events and corresponding data in a database, is one cornerstone for an undo/redo facility. Two sorts of clients exist. The first one is a generating client, which supports the implementation of PROMELA models and their proper configuration. The second one is a constructivist client which allows to investigate such PROMELA models. All clients have access to the global view provided by the server and may even duplicate it to move around in the history. Moreover, depending on current access rights (cf. floor control below), the simulation itself and movements on the global history can be controlled. Hence, learners can undo and redo actions in the local history at will and in the global one according to their rights.

While the main components are written in Java, the usage of SPIN with its need for a C

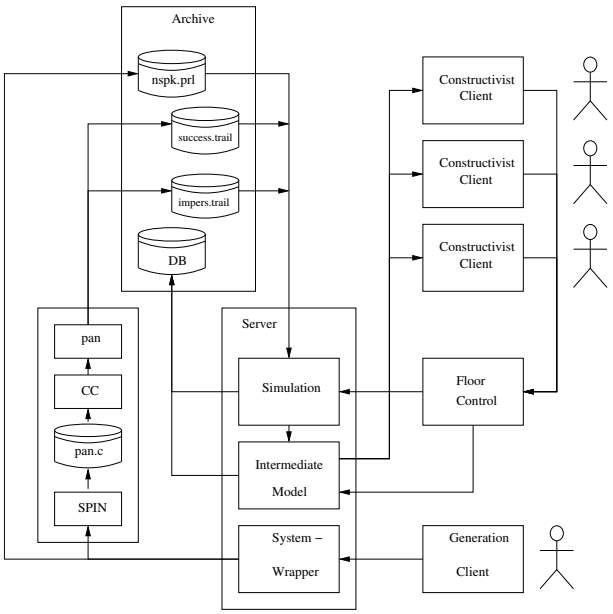


Figure 1: The ProDuctivE architecture

compiler induces system dependencies. Since all components related to SPIN are placed on the server, clients become independent from the underlying system.

## 4.2 Visualization issues

The PROMELA model and its simulation produce a lot of information, but only a fraction is needed to support learners in a proper way. For instance, it is common practice for protocols to have a separate process serve as a network or a router. It receives messages on one PROMELA channel and forwards them to a destination process which listens on another one. If it is not necessary to visualize the network process, both channels can be presented as just one. Hence, by transforming, filtering and generalizing information, the perception of important issues can be improved tremendously.

In ProDuctivE, a special interpreter written in Java accepts PROMELA input, and a subset of the C preprocessor directives. It performs the simulation by using classes which are named HiProcess, HiChannel and HiMessage. Each simulation class supports the attachment of event listeners for visualization purposes. That means, these listeners propagate simulation events towards suitable visualization components. As a consequence, these events are mapped into appropriate visualization actions. The relevant events are summarized in Tab. 3.

Visualization components were drawn from HiSAP (Highly interactive Simulation of Algorithms and Protocols), a toolkit consisting of Java classes for interactive simulation and visualization of communication protocols (cf. [2]). Basic visualization components are *nodes*, *connections* and *protocols*. Two nodes are connected by a connection. A node can send to or receive from a connection. Each node is associated with a strategy, which decides when such an action should be taken. The message flow between nodes can be visualized in several ways, like a Message-Sequence-Diagram or a topological view, where images represent the nodes and connections and animated messages travel along these connections. All these visualization mechanisms are shown in Fig. 2.

Table 3: Emitted events and actions they trigger.

| | | |
|---|---|---|
| **Simulation** | | |
| BeginSimulation | simulation | Initialize the views |
| EndSimulation<br>AbortSimulation | | Clean up the views |
| ChannelCreated | channel | Add the handlers for the new channel |
| RuntimeError | | |
| **Process** | | |
| ChannelAccessGained | process | Connect the process with the channel |
| ChannelAccessLost | | Disconnect the process from the channel |
| ChannelExclusiveRead<br>ChannelExclusiveWrite | channel | Change the view of the connection |
| StateReached | process | Adapt visual representation |
| **Message** | | |
| [Rv]MessageSent | process<br>message<br>channel | Trigger the sending of a message from the process to the channel |
| MessageDequeued | | Trigger the sending of a message from the channel to the process |

### 4.3 Distributed Collaboration

With the functionality described so far, only a single person can investigate a certain protocol. The next step aims toward multi-player mode. To this end, the NUSS (Notebook University Stuttgart) framework is used (NUSS supersedes the former project SASCIA which was described in [1]). It provides mechanisms for application sharing in the teaching area. This means especially, to enable annotation and notetaking on application windows as well as recording of application usage and annotations. Moreover, a floor control component in combination with a role concept care for having learners take over different parts in controlling the application. With regard to this paper, simulation and validation of security protocols is the application in question. In this context, the following roles are involved: generator of a promela model, initiator of a validation, *Alice*, *Bob*, *Eve* and the *Notary*. The integration of the SPIN based teaching facility into NUSS is performed within [7].

## 5 Related Work

The application of model checking with SPIN to teaching has been mentioned in [3] and [8]. Since they restrict on protocols like mutual exclusion and dining philosophers, they have been able to use PROMELA and SPIN in the original form. Collaboration between learners has not been thought of yet.

Other approaches aim at offering graphical user interfaces and visualization for SPIN. In the first place, this holds for XSPIN as described in [4]. During simulation runs, the message sequence chart as well as time sequence and data value panels are shown. ProDuctivE contains two more facilities designed especially for teaching purposes: customizable views, e.g. a topology view and the possibility to step back and forth in the simulation thus exploring various paths.

This paper has drawn profit from the work of [6]. The authors developed a universal intruder model which unveils a known flaw in the Needham-Schroeder-Public-Key authentication protocol (cf. [10]). Their model is based on a static analysis of message exchange among the participating processes. The intruder model itself keeps track of the data available to it and modifies global variables to indicate success. The derived intruder model completely exploits the results of the

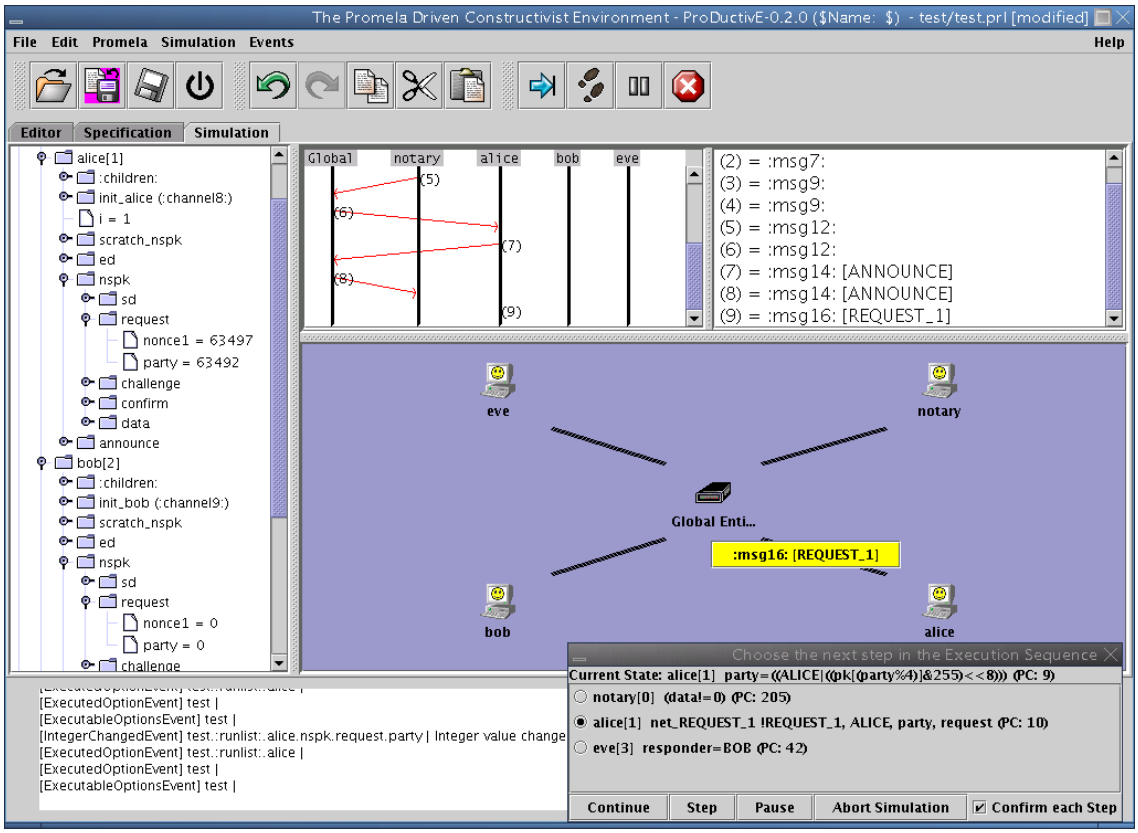


Figure 2: A screenshot of the ProDuctivE prototype.

static analysis, and thus should discover any failure. While this approach keeps the state space small for an efficient validation, it does not consider the peculiarities needed for teaching sessions as described above. This holds especially for the integration of learners into simulation runs.

# 6 Conclusions, outlook

The usage of PROMELA and SPIN for teaching in the area of security protocols was described. Currently, further security protocols (e. g. protocols for Secure Socket Layer version 2.0 and 3.0) are going to be specified in PROMELA. An extension to further examples like network protocols is planned. Moreover, a dialog based security protocol wizard is within reach

With the security protocols at hand, experiments in class will take place during summer term 2003. Only then, one can find out about the acceptance of students as well as advantages and shortcomings of the environment that is empowered by PROMELA, SPIN and collaboration.

# 7 Acknowledgement

Our deep gratitude is to Kurt Rothermel, the leader of the department Distributed Systems of the Institute for Parallel and Distributed Systems at the University of Stuttgart. He contributed some of the original ideas. Moreover, we thank Stella Papakosta who temporarily was advisor of the first author.

The work was performed in the project ITO (Information Technology Online) being funded

by the German Ministry of Education. The second author is holding a Margarethe-von-Wrangell scholarship by the state Baden-Württemberg in Germany.

## References

[1] Cora Burger, Stella Papakosta, and Kurt Rothermel. Application sharing in teaching context with wireless networks. In *Proceedings of World Congress NETWORKED LEARNING IN A GLOBAL ENVIRONMENT - Challenges and Solutions for Virtual Education*, 2002. URL: http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL_view.pl?id=INPROC-2002-04&inst=VS&mod=&engl=.

[2] Cora Burger and Kurt Rothermel. A framework to support teaching in distributed systems. *Journal of Educational Resources in Computing (JERIC)*, 1(1es):3, 2001. URL: http://doi.acm.org/10.1145/376697.376698.

[3] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Protocol Verification with Heuristic Search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, 2001. URL: http://www.informatik.uni-freiburg.de/~edelkamp/publications/protocol.pdf.

[4] Gerard J. Holzmann. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[5] Audun Jøsang. Pre-study of: Security Protocol Verification using SPIN. In *Proceedings of SPIN95, the First SPIN Workshop*, 1995. URL: http://netlib.bell-labs.com/netlib/spin/ws95/papers.html#F.

[6] Paolo Maggi and Riccardo Sisto. Using SPIN to Verify Security Properties of Cryptographic Protocols. In *Proceedings of the 9-th SPIN Workshop 2002*, 2002.

[7] Matthias Papesch. Constructivist environment for selfstudies in the area of security protocols. Master's thesis, Universität Stuttgart, 2003. URL: http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL_view.pl?id=DIP-2033&inst=VS&mod=&engl=.

[8] John Regehr. Using SPIN to Help Teach Concurrent Programming, 1998. URL: http://citeseer.nj.nec.com/regehr98using.html.

[9] Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *modelling and analysis of security protocols*. Addison-Wesley, 2001.

[10] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, 1996.

# A PROMELA implementation of the NPSK protocol

```
1  /*
2   * File: nspk.prl
3   *
4   * Description: This is an implementation of a simplified version of
5   * the Needham-Schroeder-Public-Key authentication protocol. The
6   * simplification is such, that the trusted server which keeps all
7   * public keys is left out.
8   *
9   * The goal is to unveil the flaw which inhabits this protocol.
10  * The
11  *
12  * 4 proctypes are defined, each with a specific purpose.
13  *
14  * Alice: Tries to iniatiate a session with either Bob or Eve.
15  *
16  * Bob: Responds to an initiation request from either Alice or Eve.
17  *
```

```
18   * Eve: Either initiates a session with Bob,
19   *      responds to a request from Alice
20   *      or tries to interfere with a session between Alice and Bob.
21   *
22   * Notary: Initializes the other entities and establishes shared secrets
23   *         with them. Also decides the outcome of the protocol.
24   *
25   */
26
27  #ifndef HISPIN
28
29  /*
30   * Random Number Support
31   * ---------------------
32   *
33   * For a validation, a random value can be re-used. Not so for an
34   * interactive simulation. This macro provides a possibility to indicate
35   * that the interpreter should replace the given value with a randomly
36   * generated one. The validator, however, may just replace the value with the
37   * noted default.
38   */
39  #define RND(x) x
40
41  /*
42   * Symmetric Cryptography
43   * ----------------------
44   *
45   * To perform symmetric encryption, three macros are defined.
46   * The first creates a key, which can be used for en- and decryption
47   * with the functions implemented by the other two functions, respectively.
48   *
49   * ^  : bitwise exclusive or
50   * << : shift left
51   * ~  : bitwise negation
52   */
53  #define GEN_KEY_SYM(id) RND(id)
54  #define ENCRYPT_S(val, key) ((val ^ ((key << 8) & ~255)))
55  #define DECRYPT_S(val, key) ((val ^ (key << 8)))
56
57  /*
58   * Asymmetric Cryptography
59   * -----------------------
60   *
61   * The difference to symmetric encryption is, that asymetric requires two
62   * different keys. A preliminary solution is to use the default value to
63   * lookup the wanted key. A duplicate key error should be created, when
64   * a key with the same value is created twice.
65   *
66   * The additional functionality gained is the possibility to create
67   * signatures and to verify them. The implented signature function simply
68   * returns the encrypted value.
69   */
70  #define GEN_KEY_PUB(id) (~id & 255)
71  #define GEN_KEY_SEC(id) ( id & 255)
72  #define ENCRYPT_A(val, key) ((val | (( key & 255) << 8) ))
73  #define DECRYPT_A(val, key) ((val ^ ((~key & 255) << 8) ))
```

```
74
75  #define SIGN(val, key)  ENCRYPT_A(val, key)
76  #define VRFY(val, key) DECRYPT_A(val, key)
77
78  #endif
79
80  /*
81   * Improved Message handling
82   * ------------------------
83   * These macros provide a more comfortable way to handle messages
84   * interactively . If the entire message content is contained within a
85   * data structures , its components can be accessed by a name. This is
86   * clearly more convenient than the anonymous, positional access "regular"
87   * channels use.
88   *
89   * MESSAGE macro
90   * -------------
91   * The message maco makes sure, that
92   * - an mtype for the message is created
93   * - each protocol specific message is described by a typedef'ed structure
94   * - a global channel is created , which takes 4 arguments
95   *       1. the type of the message
96   *       2. the mtype describing the sender
97   *       3. the mtype describing the receiver
98   *       4. a structure of the correct type
99   *     Messages can only be blocked according to the sender and the receiver,
100  *     but not on the content.
101  *
102  * SND/RCV macros
103  * ----------
104  * The SND/RCV macro provide a simple invocation of a send/receive
105  * operations for a message defined with the MESSAGE macro. They take
106  * - the message type
107  * - the sender
108  * - the receiver
109  * - a data structure
110  * as arguments. They create the correct statement for the channel
111  * created through the MESSAGE macro.
112  *
113  * These macros use the concatenation operator '##', which merges two
114  * string tokens into a single one. (see man cpp)
115  */
116 #define MESSAGE(s, m, d)                                        \
117         mtype { m };                                            \
118         typedef msg_##m { d; };                                 \
119         chan net_##m = [0] of {mtype, mtype, mtype, msg_##m }
120
121 #define SND(m, s, r, d) net_##m!m(s, r, d);
122 #define RCV(m, s, r, d) net_##m?m(s, r, d);
123
124 /*
125  * Constants
126  */
127 #define OUTCOME capture
128 #define MAX_PROC 4
129 #define MAX_PROC2 16
```

```
130 #define SESSION (ENTITY_INDEX(sender) + MAX_PROC * ENTITY_INDEX(receiver))
131
132 #define ENTITY_INDEX(e) (e % MAX_PROC)
133
134 #define I_ALICE ENTITY_INDEX(ALICE)
135 #define I_BOB   ENTITY_INDEX(BOB)
136 #define I_EVE   ENTITY_INDEX(EVE)
137 #define I_NOTARY ENTITY_INDEX(NOTARY)
138
139 /*
140  * Request
141  * --------
142  * - nonce1 : a nonce value to identify the initiator
143  * - party  : the party to talk to
144  */
145 MESSAGE(NSPK, REQUEST_1, int nonce1; int party);
146
147 /*
148  * Challenge
149  * ----------
150  * - nonce1 : the original nonce to identify the initiator
151  * - nonce2 : another nonce to identify the responder
152  */
153 MESSAGE(NSPK, CHALLENGE_2, int nonce1; int nonce2);
154
155 /*
156  * Confirm
157  * --------
158  * - nonce2 : the nonce to identify the responder
159  */
160 MESSAGE(NSPK, CONFIRM_3, int nonce2);
161
162 /*
163  * Data
164  * ----
165  * - message: the actual message value
166  *
167  */
168 MESSAGE(NSPK, DATA_5, int message);
169
170 mtype = {ALICE, BOB, EVE, NOTARY,
171          ANNOUNCE,
172          NAB, NAE, N2B, NEB, N2E};
173
174 /*
175  * sesison_data_NSPK
176  * ------------------
177  * contains all session specific data.
178  *
179  * - party          : the other end
180  * - nonce1, nonce2 : nonces to identify each end
181  * - payload        : the actual message
182  */
183 typedef session_data_NSPK {
184   mtype party;
185   int nonce1, nonce2, payload;
```

```
186 };
187
188 /*
189  * session_NSPK
190  * ------------
191  * contains an instance of each message and one session_data.
192  *
193  */
194 typedef session_NSPK {
195   session_data_NSPK  sd;
196   msg_REQUEST_1    request;
197   msg_CHALLENGE_2 challenge;
198   msg_CONFIRM_3    confirm;
199   msg_DATA_5         data;
200 };
201
202 /*
203  * Announce
204  * --------
205  * - sender   : the proclaimed sender
206  * - receiver : the proclaimed receiver
207  * - data     : the message of the session
208  */
209 typedef msg_ANNOUNCE {
210   mtype sender, receiver;
211   int data;
212 };
213
214 /*
215  *  stores the message and the correctnes of the sender
216  */
217 typedef check {
218   bit   s_correct;
219   int   data;
220 };
221
222 /*
223  * entity_data_NSPK
224  * ----------------
225  * - secure : a private channel shared with the notary
226  * - sk     : a secret key
227  * - pk[]   : a set of public keys
228  */
229 typedef entity_data_NSPK {
230   chan secure;
231   int sk, pk[MAX_PROC];
232 };
233
234
235 /*
236  *  Alice
237  */
238 proctype alice(chan init_alice) {
239   entity_data_NSPK ed;
240
241   session_NSPK      nspk;
```

```
242     msg_ANNOUNCE announce;
243
244     /*
245      * just to ignore the empty field
246      * prevents a warning
247      */
248     session_data_NSPK scratch_nspk;
249
250     /*
251      * receive the initialization from the notary.
252      */
253      init_alice ?ed, nspk.sd, scratch_nspk;
254
255     /*
256      * send the announcement on the secure channel to the Notary.
257      */
258   send_ANNOUNCE: d_step {
259     announce.sender   = ALICE;
260     announce.receiver = nspk.sd.party;
261     announce.data     = nspk.sd.payload; }
262
263     ed.secure!ANNOUNCE(announce);
264
265     /*
266      * send the request message, encrypted with the public key of the
267      * other party
268      */
269   send_REQUEST_1: d_step {
270       nspk.request.nonce1
271         = ENCRYPT_A(nspk.sd.nonce1, ed.pk[ENTITY_INDEX(nspk.sd.party)]);
272       nspk.request.party
273         = ENCRYPT_A(ALICE, ed.pk[ENTITY_INDEX(nspk.sd.party)]);
274     }
275     SND(REQUEST_1, ALICE, nspk.sd.party, nspk.request);
276
277     /*
278      * receive the challenge, which is encrypted with this public key
279      * make sure, that the nonce matches.
280      */
281   receive_CHALLENGE_2:
282     RCV(CHALLENGE_2, eval(nspk.sd.party), ALICE, nspk.challenge);
283
284     nspk.sd.nonce2 = DECRYPT_A(nspk.challenge.nonce2, ed.sk);
285
286     (DECRYPT_A(nspk.challenge.nonce1, ed.sk) == nspk.sd.nonce1);
287
288     /*
289      * send the confirm message.
290      */
291   send_CONFIRM_3:
292     nspk.confirm.nonce2
293       = ENCRYPT_A(nspk.sd.nonce2, ed.pk[ENTITY_INDEX(nspk.sd.party)]);
294
295     SND(CONFIRM_3, ALICE, nspk.sd.party, nspk.confirm);
296
297     /*
```

```
298     * send the data message
299     */
300   send_DATA_5:
301     nspk.data.message = ENCRYPT_S(nspk.sd.payload, nspk.sd.nonce2);
302
303     SND(DATA_5, ALICE, nspk.sd.party, nspk.data);
304   }
305
306   /*
307    *  Bob
308    */
309   proctype bob(chan init_bob) {
310     entity_data_NSPK ed;
311
312     session_NSPK      nspk;
313     msg_ANNOUNCE announce;
314
315     session_data_NSPK scratch_nspk;
316
317     /*
318      * receive the initialization from the notary.
319      */
320     init_bob?ed, nspk.sd, scratch_nspk;
321
322   receive_REQUEST_1: d_step {
323     RCV(REQUEST_1, nspk.sd.party, BOB, nspk.request);
324
325     nspk.sd.nonce1 = DECRYPT_A(nspk.request.nonce1, ed.sk);
326     nspk.sd.party  = DECRYPT_A(nspk.request.party, ed.sk); }
327
328     /*
329      *  send the challenge, encrypted with the other party's public key
330      */
331   send_CHALLENGE_2: d_step {
332     nspk.challenge.nonce1
333       = ENCRYPT_A(nspk.sd.nonce1, ed.pk[ENTITY_INDEX(nspk.sd.party)]);
334
335     nspk.challenge.nonce2
336       = ENCRYPT_A(nspk.sd.nonce2, ed.pk[ENTITY_INDEX(nspk.sd.party)]); };
337
338     SND(CHALLENGE_2, BOB, nspk.sd.party, nspk.challenge);
339
340     /*
341      *  receive the confirm message, unpack it and make sure nonce2 is
342      *  is the one previously sent
343      */
344   receive_CONFIRM_3:
345     RCV(CONFIRM_3, eval(nspk.sd.party), BOB, nspk.confirm);
346
347     (DECRYPT_A(nspk.confirm.nonce2, ed.sk) == nspk.sd.nonce2);
348
349     /*
350      *  receive the data message encrypted with nonce2
351      */
352   receive_DATA_5: d_step {
353     RCV(DATA_5, eval(nspk.sd.party), BOB, nspk.data);
```

```
354
355   nspk.sd.payload = DECRYPT_S(nspk.data.message, nspk.sd.nonce2); };
356
357   /*
358    * announce the reception of the data message to the notary
359    */
360 send_respond_3: d_step {
361   announce.sender   = nspk.sd.party;
362   announce.receiver = BOB;
363   announce.data     = nspk.sd.payload; };
364
365   ed.secure!ANNOUNCE(announce);
366
367 }
368
369 /*
370  * Eve
371  */
372
373 proctype eve(chan init_eve) {
374   entity_data_NSPK ed;
375
376   session_NSPK      nspk_actual, nspk_init, nspk_respond, nspk_capture;
377   msg_ANNOUNCE announce;
378
379   mtype initiator, responder;
380
381   /*
382    * receive the initialization
383    */
384   init_eve?ed, nspk_init.sd, nspk_respond.sd;
385
386   do
387   :: atomic {
388   /*
389    * send an arbitrarily created message
390    */
391       if
392       :: initiator = ALICE; responder = BOB;
393       :: initiator = ALICE; responder = EVE;
394       :: initiator = EVE;  responder = BOB;
395       fi;
396
397       if
398       /* REQUEST_1(nonce1, party) */
399       :: if
400          :: nspk_actual.request.nonce1 = nspk_capture.request.nonce1
401          -> nspk_actual.request.party = nspk_capture.request.party;
402
403          :: if
404             :: nspk_actual.request.party = ENCRYPT_A(ALICE, ed.pk[I_BOB]);
405             :: nspk_actual.request.party = ENCRYPT_A(BOB, ed.pk[I_BOB]);
406             :: nspk_actual.request.party = ENCRYPT_A(EVE, ed.pk[I_BOB]);
407             fi;
408
409             if
```

```
410                 :: nspk_actual.request.nonce1
411                     = ENCRYPT_A(nspk_init.sd.nonce1, ed.pk[I_BOB]);
412
413                 :: nspk_actual.request.nonce1
414                     = ENCRYPT_A(nspk_init.sd.nonce2, ed.pk[I_BOB]);
415
416                 :: nspk_actual.request.nonce1
417                     = ENCRYPT_A(nspk_respond.sd.nonce1, ed.pk[I_BOB]);
418
419                 :: nspk_actual.request.nonce1
420                     = ENCRYPT_A(nspk_respond.sd.nonce2, ed.pk[I_BOB]);
421               fi;
422             fi;
423
424             SND(REQUEST_1, initiator, responder, nspk_actual.request);
425
426             d_step {
427               nspk_actual.request.party = 0;
428               nspk_actual.request.nonce1 = 0; };
429
430           /* CHALLENGE_2(nonce1, nonce2) */
431           :: if
432              :: nspk_actual.challenge.nonce1 = nspk_capture.challenge.nonce1;
433              -> nspk_actual.challenge.nonce2 = nspk_capture.challenge.nonce2;
434              :: if
435                 :: nspk_actual.challenge.nonce1
436                     = ENCRYPT_A(nspk_init.sd.nonce1, ed.pk[I_ALICE]);
437
438                 :: nspk_actual.challenge.nonce1
439                     = ENCRYPT_A(nspk_init.sd.nonce2, ed.pk[I_ALICE]);
440
441                 :: nspk_actual.challenge.nonce1
442                     = ENCRYPT_A(nspk_respond.sd.nonce1, ed.pk[I_ALICE]);
443
444                 :: nspk_actual.challenge.nonce1
445                     = ENCRYPT_A(nspk_respond.sd.nonce2, ed.pk[I_ALICE]);
446               fi;
447
448               if
449               :: nspk_actual.challenge.nonce2
450                     = ENCRYPT_A(nspk_init.sd.nonce1, ed.pk[I_ALICE]);
451
452               :: nspk_actual.challenge.nonce2
453                     = ENCRYPT_A(nspk_init.sd.nonce2, ed.pk[I_ALICE]);
454
455               :: nspk_actual.challenge.nonce2
456                     = ENCRYPT_A(nspk_respond.sd.nonce1, ed.pk[I_ALICE]);
457
458               :: nspk_actual.challenge.nonce2
459                     = ENCRYPT_A(nspk_respond.sd.nonce2, ed.pk[I_ALICE]);
460               fi;
461             fi;
462
463             SND(CHALLENGE_2, responder, initiator, nspk_actual.challenge);
464
465             d_step {
```

```
466             nspk_actual.challenge.nonce1 = 0;
467             nspk_actual.challenge.nonce2 = 0; }
468
469         /* CONFIRM_3(nonce2) */
470         :: if
471            :: nspk_actual.confirm.nonce2 = nspk_capture.confirm.nonce2;
472            :: if
473               :: nspk_actual.confirm.nonce2
474                  = ENCRYPT_A(nspk_init.sd.nonce1, ed.pk[I_BOB]);
475
476               :: nspk_actual.confirm.nonce2
477                  = ENCRYPT_A(nspk_init.sd.nonce2, ed.pk[I_BOB]);
478
479               :: nspk_actual.confirm.nonce2
480                  = ENCRYPT_A(nspk_respond.sd.nonce1, ed.pk[I_BOB]);
481
482               :: nspk_actual.confirm.nonce2
483                  = ENCRYPT_A(nspk_respond.sd.nonce2, ed.pk[I_BOB]);
484               fi;
485            fi;
486
487            SND(CONFIRM_3, initiator, responder, nspk_actual.confirm);
488            nspk_actual.confirm.nonce2 = 0;
489
490         /* DATA_5(payload) */
491         :: if
492            :: nspk_actual.data.message = nspk_capture.data.message
493            :: if
494               :: nspk_actual.data.message
495                  = ENCRYPT_S(nspk_init.sd.payload, nspk_init.sd.nonce2)
496
497               :: nspk_actual.data.message
498                  = ENCRYPT_S(nspk_respond.sd.payload, nspk_init.sd.nonce2)
499
500               :: nspk_actual.data.message
501                  = ENCRYPT_S(nspk_init.sd.payload, nspk_respond.sd.nonce2)
502
503               :: nspk_actual.data.message
504                  = ENCRYPT_S(nspk_respond.sd.payload, nspk_respond.sd.nonce2)
505               fi;
506
507               if
508               :: announce.data     = nspk_init.sd.payload;
509               :: announce.data     = nspk_respond.sd.payload;
510               fi;
511
512               announce.sender   = initiator;
513               announce.receiver = responder;
514
515               ed.secure!ANNOUNCE(announce);
516
517               d_step {
518                announce.sender     = 0;
519                announce.receiver   = 0;
520                announce.data       = 0; }
521            fi;
```

```
522
523             SND(DATA_5, initiator, responder, nspk_actual.data);
524             nspk_actual.data.message = 0;
525
526           fi;
527         }
528
529         /*
530          *  receive messages of a session eve is involved - decryptable
531          */
532       :: d_step { RCV(REQUEST_1, ALICE, EVE, nspk_respond.request);
533       -> nspk_respond.sd.nonce1 = DECRYPT_A(nspk_respond.request.nonce1, ed.sk)
534       -> nspk_respond.sd.party = DECRYPT_A(nspk_respond.request.party, ed.sk);};
535
536       :: d_step { RCV(CHALLENGE_2, BOB, EVE, nspk_init.challenge);
537       -> nspk_init.sd.nonce2 = DECRYPT_A(nspk_init.challenge.nonce2, ed.sk); };
538
539       :: d_step { RCV(CONFIRM_3, ALICE, EVE, nspk_respond.confirm);
540       -> nspk_respond.sd.nonce2 = DECRYPT_A(nspk_respond.confirm.nonce2, ed.sk);};
541
542       :: d_step { RCV(DATA_5,   ALICE, EVE, nspk_respond.data);
543       -> nspk_respond.sd.payload
544            = DECRYPT_S(nspk_respond.data.message, nspk_respond.sd.nonce2);
545
546       -> announce.sender = ALICE;
547       -> announce.receiver = EVE;
548       -> announce.data    = nspk_respond.sd.payload; }
549
550         ed.secure!ANNOUNCE(announce);
551         d_step {
552           announce.sender   = 0;
553           announce.receiver = 0;
554           announce.data     = 0; };
555
556         /*
557          *  receive messages of a captured session - unreadable
558          */
559       :: RCV(REQUEST_1, ALICE, BOB, nspk_capture.request);
560
561       :: RCV(CHALLENGE_2, BOB, ALICE, nspk_capture.challenge);
562
563       :: RCV(CONFIRM_3, ALICE, BOB, nspk_capture.confirm);
564
565       :: RCV(DATA_5,    ALICE, BOB, nspk_capture.data);
566
567     od;
568   }
569
570
571   active proctype notary() {
572
573     session_data_NSPK scratch_nspk;
574     chan secure[3] = [0] of {mtype, msg_ANNOUNCE};
575     chan ini[3]    = [0] of {entity_data_NSPK,
576                              session_data_NSPK,
577                              session_data_NSPK};
```

```
578
579     entity_data_NSPK ed[MAX_PROC];
580
581     session_data_NSPK sd_alice_bob, sd_alice_eve,
582                       sd_bob_alice,
583                       sd_eve_bob, sd_eve_alice ;
584
585     /*
586      *  initialization
587      */
588     d_step {
589         ed[I_ALICE].secure        = secure[0];
590         ed[I_ALICE].sk            = GEN_KEY_PUB(5);
591         ed[I_ALICE].pk[I_ALICE] = GEN_KEY_SEC(5);
592
593         ed[I_BOB  ].pk[I_ALICE] = ed[I_ALICE].pk[I_ALICE];
594         ed[I_EVE  ].pk[I_ALICE] = ed[I_ALICE].pk[I_ALICE];
595         ed[I_NOTARY].pk[I_ALICE] = ed[I_ALICE].pk[I_ALICE];
596
597         ed[I_BOB  ].secure        = secure[1];
598         ed[I_BOB  ].sk            = GEN_KEY_SEC(7);
599         ed[I_BOB  ].pk[I_BOB  ] = GEN_KEY_PUB(7);
600
601         ed[I_ALICE].pk[I_BOB  ] = ed[I_BOB  ].pk[I_BOB  ];
602         ed[I_EVE  ].pk[I_BOB  ] = ed[I_BOB  ].pk[I_BOB  ];
603         ed[I_NOTARY].pk[I_BOB] = ed[I_BOB].pk[I_BOB  ];
604
605         ed[I_EVE  ].secure        = secure[2];
606         ed[I_EVE  ].sk            = GEN_KEY_SEC(11);
607         ed[I_EVE  ].pk[I_EVE  ] = GEN_KEY_PUB(11);
608
609         ed[I_ALICE].pk[I_EVE  ] = ed[I_EVE  ].pk[I_EVE  ];
610         ed[I_BOB  ].pk[I_EVE  ] = ed[I_EVE  ].pk[I_EVE  ];
611         ed[I_NOTARY].pk[I_EVE] = ed[I_EVE].pk[I_EVE  ];
612
613         ed[I_NOTARY].sk           = GEN_KEY_PUB(13);
614         ed[I_ALICE].pk[I_NOTARY] = ed[I_NOTARY].pk[I_NOTARY];
615         ed[I_EVE  ].pk[I_NOTARY] = ed[I_NOTARY].pk[I_NOTARY];
616         ed[I_BOB  ].pk[I_NOTARY] = ed[I_NOTARY].pk[I_NOTARY];
617
618         sd_alice_bob.party    = BOB;
619         sd_alice_bob.nonce1   = RND(NAB);
620         sd_alice_bob.payload = 35;
621
622         sd_alice_eve.party    = EVE;
623         sd_alice_eve.nonce1   = RND(NAE);
624         sd_alice_eve.payload = 45;
625
626         sd_bob_alice.nonce2   = RND(N2B);
627
628         sd_eve_bob.party      = BOB;
629         sd_eve_bob.nonce1     = RND(NEB);
630         sd_eve_bob.payload    = 77;
631
632         sd_eve_alice.nonce2   = RND(N2E);
633     }
```

```
634  /*
635   *  create the other processes
636   */
637  atomic {
638    run alice(ini[0]);
639    run bob(ini[1]);
640    run eve(ini[2]);
641
642    ini[1]!ed[I_BOB], sd_bob_alice, scratch_nspk;
643    ini[2]!ed[I_EVE], sd_eve_bob, sd_eve_alice;
644
645  if
646  :: ini[0]!ed[I_ALICE], sd_alice_bob, scratch_nspk;;
647  :: ini[0]!ed[I_ALICE], sd_alice_eve, scratch_nspk;
648  fi;
649  }
650
651  /*
652   *
653   */
654  check storage[MAX_PROC2];
655  msg_ANNOUNCE ann;
656  mtype entity      = 0;
657  int session_count = 0, session = 0;
658
659  do
660   :: atomic {
661  /*
662   *  handle an announcement according to the channel on which it arrives
663   */
664     if
665     :: ed[I_ALICE].secure?ANNOUNCE(ann)
666     -> entity = ALICE;
667
668     :: ed[I_BOB ].secure?ANNOUNCE(ann);
669     -> entity = BOB;
670
671     :: ed[I_EVE ].secure?ANNOUNCE(ann);
672     -> entity = EVE;
673     fi;
674  -> if
675     :: (ann.data != 0)
676     -> session = (ENTITY_INDEX(ann.sender)
677                   + (MAX_PROC * ENTITY_INDEX(ann.receiver)));
678     -> if
679        ::  storage[session].data == 0
680        /*
681         *  the first announcement
682         */
683        -> storage[session].s_correct = (entity == ann.sender);
684        -> storage[session].data      = ann.data
685        -> session_count++;
686
687        :: ((storage[session].data != 0 )
688        /*
689         *  the second announcement
```

```
690             */
691             && (storage[session].data == ann.data)
692             /*
693              * the correct data
694              */
695             && (entity != ann.sender))
696             -> if
697                 :: ( entity == ann.receiver)
698                 /*
699                  * the receiver is honest
700                  */
701                 -> if
702                     :: ( storage[session]. s_correct )
703                     /*
704                      * the sender was honest
705                      */
706                     -> session_count--;
707                     -> storage[session].data = 0;
708                     -> if
709                         :: ( session_count == 0) -> goto success;
710                         :: else                  -> skip
711                         fi;
712
713                   :: else
714                   /*
715                    * the sender was cheating
716                    */
717                   -> goto impersonation;
718                   fi;
719
720                 :: else
721                 /*
722                  * the receiver is cheating
723                  */
724                 -> if
725                     :: ( storage[session]. s_correct ) -> goto capture;
726                     :: else                            -> goto failure;
727                     fi;
728                 fi;
729
730               :: else -> goto failure;
731               fi;
732
733           :: else -> goto failure;
734           fi;
735       -> session = 0;
736       -> entity  = 0; };
737       od;
738
739   success:         false;
740    failure :       false;
741   capture:         false;
742   impersonation:   false;
743   abort:           false;
744   }
745   #ifndef HISPIN
```

```
746 never {
747   do
748    :: notary[0]@OUTCOME -> break
749    :: skip;
750   od;
751 }
752 #endif
```