Using the ASTRAL Model Checker for Cryptographic Protocol Analysis

Zhe Dang
Richard A. Kemmerer
Reliable Software Group
Department of Computer Science
University of California
Santa Barbara, CA 93106

August 3, 1997

Abstract

ASTRAL is a formal specification language for real-time systems. It is intended to support formal software development, and therefore has been formally defined. This paper focuses on the mechanism of the ASTRAL model checker and how it can be used to analyze encryption protocols.
Key Words: Encryption protocols, Formal methods, Formal specification and verification, Real-time systems, Timing requirements, Proof obligations, State machines, ASTRAL.

1 Introduction

ASTRAL[GK91] is a high-level formal specification language for real-time system. It is provided with structuring mechanisms that allow one to build modularized specifications of complex systems with layering[CGK97]. A real-time system is modeled by a collection of process specifications and a single global specification. Each process specification consists of a sequence of levels; each level is an abstract view of the process being specified. In [CKM94] formal proofs in ASTRAL, which include interlevel proofs and intralevel proofs, are discussed and a formal framework for generating intralevel proof obligations in ASTRAL is presented. The potential unbounded nondeterminism in ASTRAL transition exit assertions and the validity of the critical requirement statements, which may involve quantifiers over unbounded ASTRAL types, make ASTRAL specifications undecidable. The ASTRAL model checker presented in this paper will be used to verify ASTRAL specification under some restrictions.

The ASTRAL Software Development Environment (SDE) is an integrated set of design and analysis tools based upon the ASTRAL formal framework, which includes a syntax-directed editor, a specification processor, a verification condition generator, and a browser kit. The ASTRAL model checker is added as a new component to the SDE. It checks satisfiability of critical requirements of an ASTRAL specification by enumerating possible runs of transitions within a given time bound. It prompts the user with run-time messages whenever an error is encountered in the run of a specification.

In this paper the use of the ASTRAL model checker for cryptographic protocol analysis is presented. The classic benchmarks being investigated include the Needham-Schroeder public-key authentication protocol and the TMN protocol.

2 Overview of ASTRAL and ASTRAL model checker

An ASTRAL specification includes the global specification and process specifications. The global specification contains declarations of process instances, global constants and nonprimitive types that may be shared by process types, and system level critical requirements. There is a process specification for each process type declared in the global specification. In the Needham-Schroeder public key protocol specification, three process types are declared in the global specification:

```
PROCESSES
        Initiator: array [ 1..NumberInitiator ]  of ProcessInitiator,
        Responder: array [ 1..NumberResponder ]  of ProcessResponder,
        Intruder: ProcessIntruder
```

This indicates that there are NumberInitiator initiators in the model of the protocol, each of process type ProcessInitiator. There are also NumberResponder precesses of type ProcessResponder and there is one Intruder process.

In the process specification of ProcessInitiator there are two parameterized transitions declared:

```
        TRANSITION Initiate ( j: RESP_INTR_ID )
```

and

```
        TRANSITION Receive ( j: RESP_INTR_ID )
```

where parameters i and j are of RESP_INTR_ID type, which basically comprises all identifiers of process instances of ProcessResponder and ProcessIntruder. The type ID is one of the primitive types of ASTRAL. Each process instance has a unique identifier and "self" is used to refer to its own identifier. The body of an ASTRAL transition includes pairs of Entry/Exit assertions with duration of each pair indicated. Entry assertion must be satisfied at the time the transition starts, whereas the Exit assertion will hold after the time indicated by the duration when the transition triggers. For example,

```
        TRANSITION Receive ( j: RESP_INTR_ID )
        Entry [Time:1]
                state = init
                & Intruder.to = self
                & Intruder.from = j
                & Intruder.pkey = PublicKey ( self )
                & Intruder.nonce_a = nonce_a
                & Intruder.source = self
                & Intruder.kind = Msg_Type_2
        Exit
                state = commit_inter
                & nonce_b = Intruder.nonce_b
                & pkey = PublicKey ( j )
                & to = j
                & from = self
                & kind = Msg_Type_3
                & source = self
```

which says if current state is `init` and the message on the network (modeled as process instance Intruder) encrypted in its own public key and of type `Msg_Type_2`, then after duration 1 time unit, the Exit assertion is satisfied with the state changes to `commit_inter` and a new message of type `Msg_Type_3` is put on the network. The value of Intruder.kind in the Entry assertion is that of variable kind of process instance Intruder at the time when the transition fires. And `source = self` in the Exit assertion states that after 1 time unit, the value of variable source may change and the value should satisfy `source = self`.

ASTRAL specifications also include environmental assumptions and critical requirements. An environment clause formalizes the assumptions that must always hold on the behavior of the environment to guarantee some desired system properties, see [CGK97] for a detailed description of ASTRAL.

One way to verify an ASTRAL specification is with an ASTRAL model checker. However, due to the undecidability of ASTRAL which we will discuss further, several restrictions will apply to the specifications the model checker can handle.

The model checker procedure is composed of the following steps:

- Assign a model checker time bound. ASTRAL abstract machine is, in general, an infinite state machine with unlimited time bounds whenever the system can evolve. However, in order to make the model checker work, a finite time bound needs to be set by the user.
- Place a call index I to exported transitions in the specification. According to ASTRAL semantics, an exported transition must be called by the external environment before it starts. The model checker will verify if such an index is suitable by checking that it satisfies the condition that for any exported transition T, a call need not be placed to it at current time if T was already called but has not yet started.
- Apply the call index to the current state and select an enabled transition index. Such selection should obey the ASTRAL semantics, which says at most one transition is firing for each process at any time, and a transition is enabled if its entry assertion is satisfied and no other transition in the same process instance is firing at the current time, also if it is an exported transition, it should be in the call index. Then by updating Call time and Start time of transitions in the two indices according to the combination of call index and enabled transition index.
- Evolve the system forward by applying the selected combination of call index and enabled transition index. Check and report any violation of the critical requirements.
- Evolve the system backward if we run out of call indices and enabled transition indices or the system time bound is reached.

The basic task of the model checker is to verify, given a snapshot of current global state, the satisfiability of the ASTRAL specification critical requirements. The evolution of the ASTRAL abstract state machine is simulated within the given time bound to detect any violation.

The model checker is integrated to the ASTRAL SDE, coded in C++ with graphical user interface, running on Sun SparcStation 4 with UNIX Solaris 2.5.1.

3 Handling unbounded nondeterminism

An ASTRAL specification has only finite process instances and each process instance has finite transitions. The initial statement of a process and the exit assertion of transitions may involve unbounded nondeterminism. For example, in modeling nonce generation of a security protocol like Needham-Schroeder's , we use the following transition in process ProcessInitiator:

```
TRANSITION Initiate ( j: RESP_INTR_ID )
ENTRY    [TIME:1]
        state = idle
EXIT
        state = init
        & nonce_a = nonce_a
        & to = j
        & from = self
        & pkey = PublicKey ( j )
        & source = self
        &  kind = Msg_Type_1
```

which among other things prepares a `nonce_a` for initiating the protocol. The EXIT assertion says after firing the transition with duration 1, `nonce_a` could be any fixed value from its domain as declared which certainly satisfies `nonce_a=nonce_a`. Hence the nondeterministic choices of `nonce_a`'s value are unbounded provided it is of type integer.

The expressibility of ASTRAL formulas shown as invariant statements etc., is the same as PA (Peano Arithmetic), therefore the validity of ASTRAL formula is undecidable. This adds extra difficulties in handling the unbounded nondeterminism. In our model checker, we restrict the ASTRAL formula $F(\vec{X})$ to the following generalized form (or equivalent):

$$\forall \vec{y}_1 \exists \vec{y}_2 \cdots Q \vec{y}_n I(\vec{y}_1 \cdots \vec{y}_n, \vec{X})$$

(Q is $\forall$ if n is odd and is $\exists$ otherwise.) where I is an arithmetic expression and $\vec{y}_1 \cdots \vec{y}_n$ are either explicitly bounded or their bounds can be resolved by context. $\vec{X}$ is composed of local variables or exported variables from other process instances. Without loss of generality, assume $\vec{X}$ are exactly those variables which can not be resolved to deterministic value (otherwise we just hide them away). Furthermore, we restrict an Exit assertion to the following form:

$E = E\&E|E or E|\neg E|A > A|A < A|A = A|A <= A|A >= A$

The **atomic assertion** A in the formula above is defined by,

$A = X|C|A + A|A - A$

where

$C = P.X|n|X'$

In the formula above, C holds all deterministic values including exported variables P.X from another process P, constant expression n, and X' (the value that holds for X before starting the transition). All the other variables $\vec{X}$ are supposed to be changed after the transition. Without loss of generality, let $E(\vec{X})$ stand for any Exit assertion as above. Define the **Boolean structure** B_E as follows:

$B_{E_1 \& E_2} = B_{E_1} \& B_{E_2}$

$B_{E_1 \ or \ E_2} = B_{E_1} \ or \ B_{E_2}$

$B_{\neg E} = \neg B_E$

$B_A = b_A$

where b_A is a Boolean variable indexed by A. Let $\vec{b}_A$ denote all Boolean variables in B_E and define $\vec{b}_A(\vec{X})$, an **assignment** of $\vec{b}_A$ according to $\vec{X}$, as $\vec{b}_A(\vec{X}) = A(\vec{X})$.

Example.

```
EXIT
   ((X=Y+1) or (Y=Z)) & X>Z
```

whose Boolean structure is

$(b_0 \ or \ b_1) \ \& \ b_2$

For $< X, Y, Z > = < 3, 1, 1 >$, the $\vec{b}_A$ will be $< b_0, b_1, b_2 > = < F, T, T >$.

Say $B_E(\vec{X}_1) = B_E(\vec{X}_2)$ if $E(\vec{X}_1) = E(\vec{X}_2)$ and $\vec{b}_A(\vec{X}_1) = \vec{b}_A(\vec{X}_2)$.

Denote $S(E) \subseteq S$ to be the **solution space** of $\vec{X}$ with $S(E) = \{\vec{X} : E(\vec{X}) = TRUE\}$.

Define $S/E(\vec{X})$ to be a **partition** of $S(E)$:

(1). $\forall \vec{X} \in S, E(\vec{X}) = TRUE \iff \vec{X} \in \cup(S/E(\vec{X}))$,

(2). $\forall \vec{X}_1, \vec{X}_2 \in S, E(\vec{X}_1) = E(\vec{X}_2) = TRUE \wedge B_E(\vec{X}_1) = B_E(\vec{X}_2) \iff \exists S_0 \in S/E(\vec{X}), \vec{X}_1 \in S_0 \wedge \vec{X}_2 \in S_0$.

THEOREM. $S/E(\vec{X})$ is finite.

Let $S(F)$ be the solution space of F.

THEOREM. $S(E) \subseteq S(F)$ is undecidable.

Based upon these theorems, the best thing we can do with the model checker is to check F by guessing the solution as follows:

Let $\{\vec{X}_1, \cdots, \vec{X}_k\}$ be samples chosen from $S_i \in S/E(\vec{X})$. Check F with these $\vec{X}_1, \cdots, \vec{X}_k$ respectively. F may be satisfied by these values but it may fail with other possible values. Because of the undecidability, in general we can't detect this failure. Hence by this algorithm, each transition branches into finite successors.

4 Analysis of cryptographic protocols

We formulate each participant in the cryptographic protocol as an ASTRAL process instance, which are generally users, key server(s) and intruder. We integrate the network with the intruder as one process in the sense that the intruder will control the network and each participant communicates with others via the intruder. In this way, the intruder can overhear, delete, or store each message and generate new messages by using his knowledge of overheard messages. For users (as well as the key server) the transitions in the process include sending messages, receiving messages, and generating nonces (or session keys). For the intruder, on the other hand, the transitions include storing (overhearing) messages, relaying messages, generating nonces, and attacking other honest participants by placing malicious messages into the network. So, in our model, the intruder plays three roles: a legitimate user, a spy, and the network.

Needham-Schroeder Public Key Protocol

The Needham-Schroeder Public Key Protocol serves to exchange nonces between the initiator and the responder using public keys for mutual authentication. The full protocol has seven steps as follows:

1. $A \to S : A, B$
2. $S \to B : \{K_b, B\}_{K_s^{-1}}$
3. $A \to B : \{N_a, A\}_{K_b}$
4. $B \to S : B, A$
5. $S \to B : \{K_a, A\}_{K_s^{-1}}$
6. $B \to A : \{N_a, N_b\}_{K_a}$
7. $A \to B : \{N_b\}_{K_b}$

By assuming that each participant already knows all the public keys, we can simplify the protocol into 3 messages:

1. $A \to B : \{N_a, A\}_{K_b}$
2. $B \to A : \{N_a, N_b\}_{K_a}$
3. $A \to B : \{N_b\}_{K_b}$

To initiate the protocol, initiator A generates a nonce N_a, encrypts it with its own ID under responder B's public key K_b and sends it to B. Then B knows N_a by decrypting the message, and generates a nonce N_b, encrypts N_b with N_a under A's public key K_a. When A receives the second message, A knows that B got the initial message by comparing the nonce N_a in these two messages. After A sends the last message to B, in the same way, B authenticates A.

The ASTRAL specification of the protocol is in the appendix. The processes ProcessInitiator and ProcessResponder perform the message passing described in the protocol via the intruder process, ProcessIntruder. ProcessIntruder, serves as the network and the intruder, has legitimate status, can either perform legal activities such as initiating, sending and receiving messages allowed by the protocol, relaying messages to other participants, or maliciously producing messages by combining nonces, keys, and user_ids it knows. The knowledge of the intruder is held in its local variables:

```
know_nonce_a(PID):Integer,
know_nonce_b(PID):Integer
```

where, as shown in the protocol, two nonces play different roles when the intruder can wisely use them. In addition,

```
ifknow_nonce_a(PID):Boolean,
ifknow_nonce_b(PID):Boolean,
```

are Boolean variables indicating if the intruder knows the nonce. Though for efficiency, we will not explicitly encrypt each message, the entry condition shows that only if the message is encrypted under the intruder's own public key, does the intruder know the content of the message, otherwise, it can only store, relay or delete it.

The intruder can arbitrarily delay any messages at will by putting nonsense messages on the network that other honest participants will never acknowledge. The timing condition is not critical in this specification though we put duration 1 time unit to each transition (in ASTRAL null duration transitions are not allowed). The correctness of the protocol can be formulated into ASTRAL global invariant as follows:

```
Forall j:RESP_ID,i:INIT_ID
     ((j.state=commit_final & i=j.source)
               -> i.Start(Initiate(j))>=0)
```

which says for any responder, if it committed to an initiator with the identifier stored in the field source, then this initiator really started this instance of the protocol. Our test uncovered the following well-known attack:

1. $A \rightarrow I : \{N_a, A\}_{K_i}$
2. $I(A) \rightarrow B : \{N_a, A\}_{K_b}$
3. $B \rightarrow I(A) : \{N_a, N_b\}_{K_a}$
4. $I \rightarrow A : \{N_a, N_b\}_{K_a}$
5. $A \rightarrow I : \{N_b\}_{K_i}$
6. $I(A) \rightarrow B : \{N_b\}_{K_b}$

in 135 seconds, exploring 3657 states with time bound 11 (t starts from 0). The system configuration is 1 initiator , 1 responder and 1 intruder. Note that in fact in our model checker these states are all the legal states traversed by the system instead of the states really checked. As discussed above, at each moment, the model checker will check all possible combinations of calls and transitions, and select only legal transitions to run and discard all the others.

One advantage of ASTRAL is that we can control the behavior of our model checker via the environment (either global or local) clauses, which will significantly reduce the search space of our model checker without rewriting the specification. As in our specification, if we add

```
ENVIRONMENT
       Forall i:INIT_ID,j:RESP_INTR_ID
          (i.Call(Initiate(j),0) &
            Forall t:Time (t>0->i.Call(Initiate(j),t)))
```

which says that any call to the exported transition `Initiate` that initiates the protocol only happens in time t=0. Our model checker will report the same attack under the same system configuration within 65 seconds with only 587 states explored (time bound still is 11). For larger protocols, we could formulate our preoccupied observation into the environment clause in order to uncover some weakness of the protocol.

TMN Protocol

The TMN protocol is intended to be used in mobile phone systems for session key distribution. It is as follows:

$A \rightarrow S : B, \{N_a\}_{K_s}$
$S \rightarrow B : A$
$B \rightarrow S : A, \{N_b\}_{K_S}$
$S \rightarrow A : B, N_b + N_a$

At the end of the protocol, nonce $\{N_b\}$ is taken as a shared secret between initiator A and responder B.

Similar to the case of the Needham-Schroeder protocol, our model is composed of processes representing initiators, responders, servers and the intruder. We also put a counter on each process to indicate the timestamp of the message being sent to the network (the intruder): after each transition's firing, the exit assertion will increase the counter by one so that other processes will know whether this message was already fetched. The correctness of the protocol is formulated as an ASTRAL global invariant:

```
Forall j:RESP_ID,i:INIT_ID
  (i.share_secret(j)>0->~Intruder.ifknow_nonce(j))
```

that is, if an initiator shares a secret with a responder, then the intruder doesn't know the secret.

The intruder can impersonate other legitimate users. Under the simplest setup including 1 initiator, 1 responder, 1 server and the intruder, the model checker finds the following bug:

$A \to S : B, \{N_a\}_{K_s}$
$S \to B : A$
$I(B) \to S : A, \{N_i\}_{K_S}$
$S \to A : B, N_i + N_a$

after searching 3300 states within time bound 8 in 120 seconds. And no other bugs were found with 38,000 states explored in 962 seconds.

5 Discussion

In ASTRAL two categories of critical requirements are explicitly distinguished, one is ASTRAL invariant clause, the other is ASTRAL schedule clause. The former will always hold regardless of the validity of environmental assumptions, while the later will hold if environmental assumptions are satisfied. Therefore, even if the current state constructed by the model checker does not satisfy the environmental assumptions, we still need to check the validity of invariant clauses. However, for the user's convenience, our model checker has the option of checking only the states satisfying the environmental assumptions. Therefore, if a user makes this choice when using the model checker, he could significantly reduce the search space by imposing stronger restrictions on the environmental assumptions of the specification without rewriting the specification.

The efficiency of using our model checker to analyze secure protocols is heavily dependent on the following factors:

- depth of the anticipated run of a protocol coded in ASTRAL. Different modeling methods differ dramatically. For example, our first version of Needham-Schroeder public-key authentication protocol specification, the message passing between two parties is not relayed by the network (the intruder): each participant including the intruder has local variable called Msg which can be seen by others under the condition that the observer can decrypt it. In the same manner, the intruder has the ability to accumulate his knowledge of other participants by overhearing and decrypting whenever he can, and to replay an old message or construct a malicious message which may fool an honest agent. Though this method may not work well in other cases, our test results show that the same bug is found within 62 seconds with 3600 states explored. In this case the time bound is only 7. However, this method produces a messy low level ASTRAL specification instead of a clean high level ASTRAL specification.

- avoid unnecessary exporting and idling of transitions . Exported transitions may cause "idling" if it's not called by external environment. That is, even though the entry assertion is satisfied, the transition is still not firable. We can not put firm restrictions on calls to exported transitions except as stated in the environment clause.

- using simple ASTRAL types. ASTRAL provides rich data types and their operations. However, complex types like List of List or List of Structure compositions need more time to resolve and more space to allocate.

- using fewer process instances and fewer transition instances in a specification. Like the examples in this paper, we combine the network with the process Intruder, in the sense that the intruder will fully control the network.

Our model checker missed a bug in TMN which is uncovered by FDR [LR97], because it required too large of a time bound under the current ASTRAL coding of the specification. The simpler bug shown in this paper is also uncovered by Murφ[MMS97] .

Results presented in this paper are preliminary. As our experience specifying protocols in ASTRAL grows, we believe the results can be improved further. We intend to apply the model checker in investigating some real time protocols in which ASTRAL will show its advantages as a high level real time specification language.

References

[GK91] C. Ghezzi and R. Kemmerer, *ASTRAL: An assertion language for specifying real-time systems,* Proceedings of the 3rd European Software Engineering Conference, 1991, pp.122-146.

[CGK97] A. Coen-Porisini, C. Ghezzi and R. Kemmerer, *Specification of realtime systems using ASTRAL*, to appear IEEE Transactions on Software Engineering, 1997.

[CKM94] A. Coen-Porisini, R. Kemmerer and D. Mandrioli, *A formal framework for ASTRAL intralevel proof obligations,* IEEE Transactions on Software Engineering, vol. 20 No. 8 1994, pp.548-561.

[LR97] G. Lowe and A. W. Roscoe, *Using CSP to detect errors in the TMN protocol*, to appear IEEE Transactions on Software Engineering, 1997.

[MMS97] J. C. Mitchell, M. Mitchell, and U. Stern, *Automated analysis of cryptographic protocols using Murphi*, Proceedings of the IEEE Symposium on Security and Privacy, 1997, pp.141-151.

Appendix: ASTRAL specification of Needham-Schroeder protocol

```
SPECIFICATION Needham_Schroeder_Protocol
  GLOBAL SPECIFICATION NS_GLOBAL
    PROCESSES
              Initiator: array [ 1..NumberInitiator ]  of ProcessInitiator,
              Responder: array [ 1..NumberResponder ]  of ProcessResponder,
              Intruder: ProcessIntruder /* Intruder is used to model the intruder and the network */
    TYPE
              PID: typedef i: ID ( IDTYPE  ( i )  = ProcessResponder
                                 |  IDTYPE  ( i )  = ProcessInitiator
                                 |  IDTYPE  ( i )  = ProcessIntruder ) ,
              INIT_RESP_ID: typedef i: ID ( IDTYPE  ( i )  = ProcessResponder
                                 |  IDTYPE  ( i )  = ProcessInitiator ) ,
              RESP_INTR_ID: typedef i: ID ( IDTYPE  ( i )  = ProcessResponder
                                 |  IDTYPE  ( i )  = ProcessIntruder ) ,
              INIT_INTR_ID: typedef i: ID ( IDTYPE  ( i )  = ProcessInitiator
                                 |  IDTYPE  ( i )  = ProcessIntruder ) ,
              RESP_ID: typedef i: ID ( IDTYPE  ( i )  = ProcessResponder ) ,
              INIT_ID: typedef i: ID ( IDTYPE  ( i )  = ProcessInitiator ) ,
              AGENT_STATE:  ( idle, resp, init, commit_inter, commit_final ) ,
              MsgKind:  ( Msg_Type_1, Msg_Type_2, Msg_Type_3 )
    CONSTANT
              PublicKey ( PID ) : Integer,
              NumberInitiator: Integer,
              NumberResponder: Integer
    AXIOM
              Forall i: PID, j: PID
                       ( i ~= j
                    ->   PublicKey ( i )  ~= PublicKey ( j )  )
    INVARIANT
              forall j: RESP_ID, i: INIT_ID
                       (  ( j.state = commit_final
                          & i = j.source )
                    ->   i.Start ( Initiate ( j )  )  >= 0 )
  END NS_GLOBAL
```

```
PROCESS SPECIFICATION ProcessInitiator
  LEVEL Top_Level
    IMPORT
              AGENT_STATE, PID, RESP_INTR_ID, PublicKey, Intruder, MsgKind
    EXPORT
              Initiate, state, source, to, from, pkey, nonce_a, nonce_b
    VARIABLE
              state: AGENT_STATE,
              pkey: Integer,
              nonce_a: integer,
              nonce_b: integer,
              source: PID,    /* source is who started the protocol */
              to: PID,
              from: PID,
              kind: MsgKind
    INITIAL
              state = idle
            & nonce_a = 0
            & pkey = PublicKey ( self )
            & nonce_b = 0
            & source = self
            & to = self
            & from = self

    TRANSITION Initiate ( j: RESP_INTR_ID )
      ENTRY           [ TIME : 1 ]
                state = idle
      EXIT
                state = init
              & nonce_a = nonce_a  /* get a fresh nonce */
              & to = j
              & from = self
              & pkey = PublicKey ( j )
              & source = self
              & kind = Msg_Type_1

    TRANSITION Receive ( j: RESP_INTR_ID )
      ENTRY           [ TIME : 1 ]
                state = init
              & Intruder.to = self
              & Intruder.from = j
              & Intruder.pkey = PublicKey ( self )
              & Intruder.nonce_a = nonce_a  /* the nonce is fresh */
              & Intruder.source = self
              & Intruder.kind = Msg_Type_2
      EXIT
                state = commit_inter
              & nonce_b = Intruder.nonce_b
              & pkey = PublicKey ( j )
              & to = j
              & from = self
              & kind = Msg_Type_3
              & source = self
  END Top_Level
END ProcessInitiator
```

```
PROCESS SPECIFICATION ProcessResponder
  LEVEL Top_Level
    IMPORT
              INIT_INTR_ID, PublicKey, AGENT_STATE, PID, Intruder, MsgKind
    EXPORT
              state, source, nonce_a, nonce_b, pkey
    VARIABLE
              state: AGENT_STATE,
              pkey: Integer,
              nonce_a: Integer,
              nonce_b: Integer,
              source: PID,
              to: PID,
              from: PID,
              kind: MsgKind
    INITIAL
              state = idle
            & nonce_a = 0
            & to = self
            & from = self
            & source = self
            & nonce_b = 0
            & pkey = PublicKey ( self )

    TRANSITION Respond ( i: INIT_INTR_ID )
      ENTRY           [ TIME : 1 ]
                state = idle
              & Intruder.pkey = PublicKey ( self )
              & Intruder.to = self
              & Intruder.from = i
              & Intruder.kind = Msg_Type_1
      EXIT
                state = resp
              & nonce_b = nonce_b  /* get a fresh nonce */
              & pkey = PublicKey ( Intruder.source )
              & nonce_a = Intruder.nonce_a
              & to = i
              & from = self
              & source = Intruder.source
              & kind = Msg_Type_2

    TRANSITION Receive ( i: INIT_INTR_ID )
      ENTRY           [ TIME : 1 ]
                state = resp
              & Intruder.pkey = PublicKey ( self )
              & Intruder.nonce_b = nonce_b  /* the nonce is fresh */
              & Intruder.to = self
              & Intruder.from = i
              & Intruder.kind = Msg_Type_3
      EXIT
                state = commit_final
              & source = Intruder.source
              & pkey = PublicKey ( self )
              & from = self
              & to = i
  END Top_Level
END ProcessResponder
```

```
PROCESS SPECIFICATION ProcessIntruder
  LEVEL Top_Level
    IMPORT
              INIT_ID, RESP_ID, INIT_RESP_ID, INIT_INTR_ID, RESP_INTR_ID,
              PublicKey, AGENT_STATE, PID, MsgKind, Initiator.state,
              Responder.state
    EXPORT
              source, nonce_a, nonce_b, pkey, to, from
    VARIABLE
              ifknow_nonce_a ( INIT_RESP_ID ) : Boolean,
              ifknow_nonce_b ( INIT_RESP_ID ) : Boolean,
              know_nonce_a ( INIT_RESP_ID ) : Integer,
              know_nonce_b ( INIT_RESP_ID ) : Integer,
              pkey: Integer,
              nonce_a: Integer,
              nonce_b: Integer,
              source: PID,
              to: PID,
              from: PID,
              kind: MsgKind,
              status ( INIT_RESP_ID ) : AGENT_STATE
    INITIAL
              forall i: INIT_RESP_ID
                       ( ifknow_nonce_a ( i )  = False
                       & ifknow_nonce_b ( i )  = False )
            & source = self
            & to = self
            & from = self
            & kind = Msg_Type_1
            & pkey = PublicKey ( self )
            & forall i: INIT_RESP_ID
                       ( status ( i )  = idle )

    TRANSITION Attack_Init ( i: INIT_ID, j: RESP_ID )
      /* message is to the intruder, decrypt and know nonce_a */
      ENTRY          [ TIME : 1 ]
                i.pkey = PublicKey ( self )
              & i.kind = Msg_Type_1
              & ~ifknow_nonce_a ( i )
      EXIT
                ifknow_nonce_a ( i )  Becomes True
              & know_nonce_a ( i )  Becomes i.nonce_a
      /* produce a message with known nonces */
      EXCEPT         [ TIME : 1 ]
                ifknow_nonce_a ( i )
              & ifknow_nonce_b ( j )
      EXIT
                pkey = PublicKey ( i )
              & nonce_a = know_nonce_a' ( i )
              & nonce_b = know_nonce_b' ( j )
              & from = self
              & to = i
              & kind = Msg_Type_2
              & source = i
      /* message is to the intruder, decrypt and know nonce_b */
      EXCEPT         [ TIME : 1 ]
                i.kind = Msg_Type_3
              & i.pkey = PublicKey ( self )
              & ~ifknow_nonce_b ( i )
      EXIT
                ifknow_nonce_b ( i )  Becomes TRUE
              & know_nonce_b ( i )  Becomes i.nonce_b
```

```
TRANSITION Attack_Resp ( j: RESP_ID, i: INIT_ID )
  /* message is to the intruder, decrypt and know nonce_b */
  ENTRY           [ TIME : 1 ]
            j.pkey = PublicKey ( self )
          & j.kind = Msg_Type_2
          & ~ifknow_nonce_b ( j )
  EXIT
            ifknow_nonce_b ( j )  Becomes TRUE
          & know_nonce_b ( j )  Becomes j.nonce_b
  /* nonce_a known, so produce Msg_Type_1 message */
  EXCEPT          [ TIME : 1 ]
            ifknow_nonce_a ( i )
  EXIT
            pkey = PublicKey ( j )
          & nonce_a = know_nonce_a' ( i )
          & source = i
          & to = j
          & from = self
          & kind = Msg_Type_1
  /* nonce_b known, so produce Msg_Type_3 message */
  EXCEPT          [ TIME : 1 ]
            ifknow_nonce_b ( i )
  EXIT
            pkey = PublicKey ( j )
          & to = j
          & from = self
          & source = i
          & kind = Msg_Type_3
          & nonce_b = know_nonce_b' ( i )
  /* nonce_b known, so produce Msg_Type_3 message */
  EXCEPT          [ TIME : 1 ]
            ifknow_nonce_b ( j )
  EXIT
            pkey = PublicKey ( j )
          & to = j
          & from = self
          & source = i
          & kind = Msg_Type_3
          & nonce_b = know_nonce_b' ( j )

TRANSITION Relay ( i: INIT_RESP_ID )
  ENTRY           [ TIME : 1 ]
            i.pkey ~= PublicKey ( self )
          & status ( i )  ~= i.state
  EXIT
            status ( i )  BECOMES i.state
          & kind = i.kind
          & source = i.source
          & to = i.to
          & from = i.from
          & pkey = i.pkey
          & nonce_a = i.nonce_a
          & nonce_b = i.nonce_b

TRANSITION Replay ( i: INIT_ID, j: RESP_ID )
  /* direct the message to responder j */
  ENTRY           [ TIME : 1 ]
            i.pkey ~= PublicKey ( self )
          & i.to = self
          & status ( i )  ~= i.state
  EXIT
            status ( i )  Becomes i.state
```

```
            & kind = i.kind
            & source = i.source
            & to = j
            & from = self
            & pkey = i.pkey
            & nonce_a = i.nonce_a
            & nonce_b = i.nonce_b
    /* direct the message to initiator i */
    EXCEPT          [ TIME : 1 ]
              j.pkey ~= PublicKey ( self )
            & j.to = self
            & status ( j )  ~= j.state
    EXIT
              status ( j )  Becomes j.state
            & kind = j.kind
            & source = j.source
            & to = i
            & from = self
            & pkey = j.pkey
            & nonce_a = j.nonce_a
            & nonce_b = j.nonce_b
  END Top_Level
 END ProcessIntruder
END Needham_Schroeder_Protocol
```