Declaration

I, Stylianos Basagiannis, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the words of other authors in any form e.g. ideas, equations, figures, text, tables, programs etc are properly acknowledged. A list of references employed is included.

Signed: _____

Date: _____

Abstract

This paper explores the application of formal analysis in security protocols with the help of the SPIN model checker. After a literature presentation on formal analysis, a method is proposed of building a PROMELA model of the Needham Schroeder Public Key Authentication Protocol and of an intruder attack upon this. With the help of the SPIN model checker we can validate the correct functioning of the protocol while it can identify to us an intruder's attack on it.

Acknowledgements

I am most grateful to my supervisor Dr Lilia Georgieva for all the guidance and valuable advice that she has given to me throughout the completion of this MSc dissertation.

I also like to thank my friends and roommates Georgios Rizos and Vasileios Rizikianos for the valuable discussions that we have during my studies.

Special thanks to my parents that supported me throughout my postgraduate studies.

Contents

List of Figures

Chapter 1

Introduction

1. Introduction

1.1 Introduction to Formal Analysis of Security Protocols

It is commonly known that as much as technology progress nowadays, one of the main factors that have a basic role during this evolvement is security. Beginning from the smallest system and as usual required from the biggest, security has a unique place between any kinds of communication being established. Today, one of the essential commerce mediums that have evolved among with technology and its aspects is the internet.

Starting from the simplest action of sending or receiving emails, creating peer to peer connections among different hosts, communicating and exchanging sensitive information (establishing e-transactions), the world wide web will always be in the need of the maximum security possible. Knowing that protocols are actually the "procedures" that any kind of data is being sent through the internet, results in applying all the efforts of security into them. One of these efforts is the formal analysis techniques [1], which come to improve the function and the quality of service (Qos) of communication systems. As a result, many formal techniques have being applied today on e-payment and all e-commerce transactions [2], trying to ensure the security properties required by them. All these solutions are based on some kind of cryptographic protocol which, in turn, uses basic cryptographic operations such as encryption, digital signatures and certificates.

Despite their simplicity, these protocols have revealed themselves to be error prone, especially because of the difficulty generally found in foreseeing all the possible attacks. For this reason, researchers have been working on the use of formal verification techniques [3] in order to analyze the vulnerability of such protocols.

1.2 Summary of the Project goals

The application of formal methods trying to ensure and prove all the security properties that a protocol should adhere at any point of its function, is the basic reason why formal analysis tends to be a valuable part of the developing cycle of a system. Given such a promise, this project aims to discuss issues that have to do with formal methods being applied to security protocols [3] that are being used today in a great variety of systems and communication between them. Such systems are mostly distributed systems that are located not only among the internet but also among local and wide area networks.

Being more specific about the outcomes that this project aims to produce, at first some important issues [4] that have to do with formal methods must be mentioned. Having the opportunity to use the SPIN model checker [5,7], an automated model checking tool that provide us with the ability of checking systems about safety and liveness properties, a short introduction about the tool is going to be presented.

Moving on to the area of Security protocols, a description of this is going to take place, giving attention to the Security Protocol that we are going to model and check with the model checker. The specified protocol will be the Needham Schroeder Public key Protocol [6] which is going to be discussed in the dissertation.

Main aim of this project it to check the Needham Schroeder protocol with the SPIN model checker for safety and liveness properties. After the protocol is modelled in PROMELA the specification language that SPIN takes as input [7], the tool will produce to us useful information about whether or not the protocol satisfies properties such as unreachable code, infinite cycles and deadlocks. Also an effort is going to be made in order to model an invasion of an intruder trying to "cut off" the communication between two points that

communicate through the Needham Schroeder Security protocol [6]. Finally we will study the results that the model checker produces to us after a successful modelling of the protocol and of the properties wanted to be checked.

This project sets out to investigate various research questions that have to do with security violation of communication protocols [1]. One of these questions will form the focus of the study and the answers of it will meet the objectives of the project. This question has to do with whether or not can we apply formal techniques to security protocols in order to check these protocols for correction. Also a major question is the possibility that the protocol can have "malfunctioning" during its operation by reacting error ness while security is being breached form the external environment.

1.3 Project Aims and objectives

The fact that Internet is becoming a very efficient commercial medium nowadays makes the guarantee of security a necessity for every distributed protocol running over it. The security of a computer system may be viewed from two points of view according to [4] *safety* or the possibility that the system may harm the environment and *security* or avoiding the possibility that the environment may harm the system. These are the main themes that we are going to focus in this project setting the objectives of the project as follows:

I. At first we have to model a security protocol such as Needham Schroeder public key protocol. The protocol is going to be model in its basic function between two host points trying to communicate. The whole model is going to be created with the PROMELA specification language, which is the input language of the model checker that we are going to use.

II. Then, using the SPIN model checker we are going to test the above model for safety and liveness properties. Also we will try to model an intruder invasion during the protocol communication, in order to observe the results that SPIN is going to produce us about the behaviour of the Needham Schroeder public key protocol.

1.4 Summary Content of the Dissertation

In the following lines a more specified description of each of the chapters that are going to be included to this project is given, presenting all the issues that each chapter is going to cover.

Chapter 1: Introduction

In this chapter, an introduction to the aim of the project is outlined, giving a specific point in the structure of the project describing the tasks, the design and the implementation of it.

Chapter 2: Background and Literature

In chapter 2 of the project, an introduction is being made essential knowledge needed for understanding the basic issues concerning formal analysis and security protocols. Also all the necessary background and literature are going to be presented, focusing on formal analysis, security protocols, the Needham Schroeder Security Protocol and the SPIN model checker.

Chapter 3: The SPIN model checker

Chapter 3 gives a detailed overview about the SPIN model checker, the tool that we use in this project to apply formal analysis to the security protocol that is going to be chosen.

Chapter 4: Requirements capture of the model

Chapter 4 discusses the user requirements and the system requirements that we will take into consideration during this study. We will discuss the way of modelling a Security Protocol focusing on its functioning during communication. All the needed requirements are going to be defined carefully in order for the implementation of the model to be successful and therefore the results of the tool produces based on the model.

Chapter 5: Implementation of the model

In Chapter 5, we focus on the design of the model that represents the Needham Schroeder protocol. Having discussed in the previous chapter the structure of the protocol, the basic modelling is presented here showing in a direct way how each step of the protocol is modelled in PROMELA. Also in this chapter an attempt of modelling an intruder trying to interfere the communication through Needham Schroeder is going to take place in order to ensure that the protocol is powerful enough to external threats.

.

Chapter 6: Simulation and Verification Results

This chapter is going to present and explain the results that are going to be produced for every property (that has been agreed in first place) we are going to check after applying the SPIN model checker on our model.

Chapter 7: Conclusion - future work

Conclusions and discussion about work being done so far referring to the results that have been produced from Chapter 6.

Appendix A: PROMELA source code that represents Needham Schroeder Public key protocol (protocol model) and the intruder's attack (intruder's attack model)

Appendix B: All the simulation and verification results that the SPIN model checker has generated from our PROMELA models.

Chapter 2

Background and Literature

2. Background and Literature

2.1. *Introduction*

As we previously said, a way of testing systems for correctness a consistency is the Formal Analysis[1], performed through formal techniques. Formal methods have already proven their capacity in protocol analysis and design, but are still under-employed by designers in these stages of protocol development [8]. The tendency is for them to only be used when the protocol has been defined and designed to validate or discover errors. Such errors we try to discover by applying formal methods to a security protocol, based on a cryptographic algorithm which implements the specific protocol. Although that, formal analysis are today one of the best ways of analysing and understanding the way a system is functioning, trying to correct any possible errors that may occur in a specific state of the system at a specific time of it.

2.2. *Formal Analysis and Methods today*

Before discussing how formal methods and analysis can help us today we must give a definition of what these terms actually mean. According to [9], a formal method is that area of computer science that is concerned with the application of mathematical techniques to the design and implementation of computer hardware and (more usually) software.

"That part of computer science concerned with the application of mathematical methods to the production of computer software". (Jones, 1986) [9]

[1] Referring to the term Formal Analysis we must add that Formal Analysis is the procedure in which formal methods (often mentioned as formal techniques) are been applied to a system, in order for the initial system to be checked and re-designed in a better way.

Today, formal methods are a valuable part of every system being developed. The reason for this is that formal analysis in combination with formal verification can provide to any developer with important information about the correctness of its product, proving to him with detailed feedback and counter – examples for the system being checked.
Formal methods can help according to [10] (especially in designing a security protocol) to:

- specify the system's boundary, for example the interface between the system and its environment.

- characterize a system's behaviour precisely; some of the current systems are even able to reason about real-time behaviour.

- precisely define the system's desired properties.

- prove that a system meets its specification; some methods can even provide counterexamples if this is not the case.

- force the designer to think about the protocol in a proper and thorough way: he must have a clear idea of what exactly he wants to achieve and must make assumptions explicit and non-ambiguous.

According to the above principles, we can conclude that the designer of any system should always bear in mind that designing and evaluating the system is not always enough in order to prove its correctness. Moreover there are techniques that are helping us to at first to define the system in its environment and then starting to implement it according to the rules that we want to be obeyed.

2.2.1. Application of Formal Analysis

Today industrial acceptance of formal methods has to do with the easiness of using formal methods and tools at any part of the development cycle of a software product. After the usefulness of the formal methods and the results that they can provide to us, an increasing number of companies are directed to the adoption of formal verification techniques to validate the design of their product, trying to integrate these techniques within the existing development process. Industries are keener to accept formal verification a technique assessing the quality attributes of their products, obtained by a traditional life cycle, rather than a fully formal life cycle development, due to the lower training and innovation costs. There are today communities and organizations that investigate formal methods in order for even better techniques to be invented and to be used in industry.

One of them is the Formal Methods && Tools Group of ISTI-CNR [12] which is active in the fields of development and application of formal notations, methods and software support tools for the specification, design and verification of complex computer systems. The basic aim of those groups is to develop error free systems that must meet real-time and security constraints and are used in safety critical missions where human factors play a major role. And that can be accomplished with the verification of correctness of the system through formal methods.

2.2.2. Formal Analysis of Security Protocols

The design of secure cryptographic protocols is a very complex and difficult process. Until recently, researchers were orientated towards the use of formal methods for the analysis and verification of existing protocols. These methods have proved successful at discovering flaws with existing protocols, sometimes previously unrecognised ones. Apart from that, and considering

the rapid progress of technology and its requirements for secure automation, a great deal of doubt remains as to whether any of the existing techniques is sufficient to provide a proof that a given protocol is correct [12]. This situation has an analogy in the verification process of general purpose computer programs, where reliable testing techniques allow many bugs to be found, but will not provide a basis for complete proof of correctness. Therefore formal analysis contributes to a major part of the design of specific methods, implementing tools that could aid the initial production of cryptographic protocols.

2.3. *Introduction to Security Protocols*

The second part of this chapter is being covered by reference to security protocols. Security protocols are going to be discussed, giving a special view of point into the Needham Schroeder protocol. In no doubt, it is known that security protocols are a critical element of the infrastructures needed for secure communication and processing information. Most security protocols are quite simple if only their length is considered. However, the properties they are supposed to ensure are subtle, and therefore it is hard to get protocols correct just by informal reasoning. The history of security protocols and cryptography is full of examples [11], where weaknesses of supposedly correct protocols or algorithms were discovered even years later.

Being used all around every communication in the whole world, security protocols are excellent candidates for rigorous formal analysis. They are critical components of distributed security, are very easy to express and very difficult to evaluate by hand. To design and verify security protocols such as Needham Schroeder for example, one needs to have at least a basic understanding of cryptography since encryption and authentication algorithms are the basic blocks of protocols.

2.4. *Security Protocols today*

When thinking about security and especially situations that has to do with security protocols, we should bear in mind the following aspects, according to [1]. Main aim of every security protocol and moreover for every system that wants to be characterized as secure, is to provide (quality) services; these services are different aspects for every system, and each of it has a way of being implemented among the system. These services according to [11] are:

- Authentication: establishing the identity of the users of the system
- Privacy: ensuring that data cannot be viewed by non-authorised users
- Integrity: avoiding data being modified by non-authorised users
- Non – repudiability : the capacity to prove the identity of the source or the receiver of a message

A security protocol is typically designed to provide one or more of these services. The particular mechanisms which are implemented in order to provide a service can vary and the most common mechanisms are digital signature, encipherment, hash functions and authentication exchange [11].

2.4.1. Cryptographic Protocols

Cryptography has long been regarded as the basic way to protect the confidentiality of information among communication networks. As it is previously said, the use of cryptography is been extended to every system that wants to satisfy the basic security services [1]. Procedures that apply cryptography are largely being used at the moment for message authentication, personal identification, digital signatures, electronic money transfer and other critical applications [3, 11]. Such procedures are the security protocols which embed a cryptographic mechanism in order to secure their operation. Even if we assume that the cryptography in such

procedures is completely reliable, weaknesses may result from the way in which it is used and assembled especially for communication protocols.

There are many examples of systems that at the beginning of their use, they consider to be secure and error free. One example of that is the Needham-Schroeder public key protocol (1978) [6], which were believed to be correct for several years until shown to be flawed by Lowe in 1996 in [28] (using formal methods). Another example comes from the industrial applications, such as the Java programming language (which was found to have type flaws leading to security holes) leading all the security teams of a number of companies to and the recently announcement of security holes in Netscape Navigator and Internet Explorer [19]. Many of these errors could possibly have been avoided if formal analysis and formal methods have been applied on them.

2.4.2. The Needham Schroeder Security Protocol

The security protocol we are going to use in this project is the Needham Schroeder public key protocol, based on the Needham Schroeder cryptographic algorithm. The Needham-Schroeder Public Key Protocol is a well known authentication protocol that dates back to 1978 [6]. It aims to establish mutual authentication between an initiator A and a responder B, after which some session S involving the exchange of messages between A and B is taking place. As its name clearly suggests, the protocol uses public key cryptography [6].

Each agent H possesses a public key, denoted $PK(H)$, and a secret key $SK(H)$, which can be used to decrypt the messages encrypted with $PK(H)$. While $SK(H)$ should be known only by H, any other agent can obtain $PK(H)$ from a key server. Any agent can encrypt a message x using H's public key to produce the encrypted message $fxgPK(H)$. Only the agents that know H's secret key can decrypt this message in order to obtain x. This property should

ensure x secrecy. At the same time, any agent H can sign a message x by encrypting it with its own secret key, $fxgSK(H)$ in order to ensure its integrity. Any agent can decrypt $fxgSK(H)$, using H's public key. The complete Needham-Schroeder Public Key Protocol discussed in [6] involves seven steps and it is described in figure 2.4.1, where A is an initiator agent who requests to establish a session with a responder agent B and S is a trusted key server. Any run of the protocol opens with A requesting B's public key to the trusted key server S (step 1). S responds sending the message 2. This message, signed by S to ensure its integrity, contains B's public key, $PK(B)$, and B's identity. If S is trusted, this should assure A that $PK(B)$ is really B's public key. It is worth noting that the protocol assumes that A can obtain $PK(A)$, needed to decrypt message 2, in a reliable way. If this assumption is not true, an intruder could try to replace S providing an arbitrary value that A thinks to be $PK(A)$. Note also, that as pointed out in [6], there is no guarantee that $PK(B)$ is really the current B's public key, rather that a replay of an old and compromised key (however this attack can be

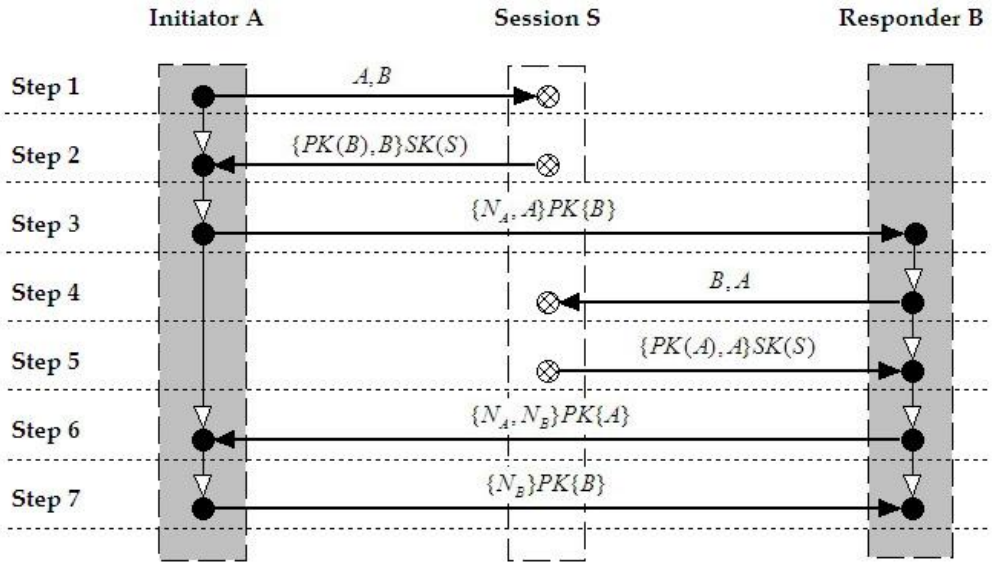

Figure 2.4.1: The complete Needham Schroeder public key protocol [3]

easily prevented using timestamps[2]). Once obtained *B*'s public key, *A* selects a nonce[3] N_A and sends message 3 to *B*. This message can only be understood by *B*, being encrypted with its public key, and indicates that someone, supposed to be *A*, wishes to authenticate himself to *B*. After having received message 3, *B* decrypts it using its secret key to obtain the nonce N_A and then requests *A*'s public key to *S* (steps 4 and 5). At this point, *B* sends the nonce N_A to *A*, along with a new nonce N_B, encrypted with *A*' a public key (message 6). With this message, *B* authenticates itself to *A*, since, receiving it, *A* is sure that it is communicating with *B*, being *B* the only agent that should be able to obtain N_A decrypting message 3. To finish the protocol run, *A* returns the nonce N_B to *B* in order to authenticate itself to *B* (message 7).

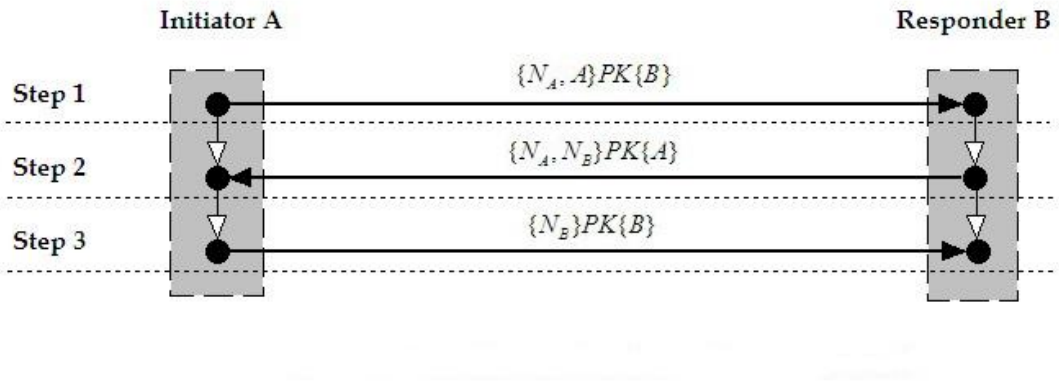

Figure 2.4.2: The reduced Needham Schroeder public key protocol [3]

Looking at the protocol, it is easy to observe as four of the seven steps can be removed if we assume that *A* and *B* already know each other's public keys.

[2] Timestamps are considering being time – signatures of a document. According to the specific message that the reference is being done, timestamps secure the integrity of a message by asserting the time stamp into it.

[3] A nonce is a random number generated with the purpose to be used in a single run of the protocol.

Indeed, messages 1, 2, 4 and 5 make up a protocol that aims to obtain *A*'s and *B*'s public keys from a trusted key server *S*, whereas messages 3, 6 and 7 make up the real authentication protocol. In the following of this paper, we will assume that all the agents already know each other's public keys and so we focus our attention on the reduced protocol obtained removing messages 1, 2, 4 and 5 and described in figure 2.4.2. The reason of doing that is that this specification is going to help us later for the modelling of the protocol to the specification language.

2.5. *Applying Formal methods to Security Protocols*

The detection and prevention of bugs are in fact two of the main reasons for using formal methods and related approaches: the specification of a system is an essential tool for analysis, and may help to discover many design errors [13]. If the specification is given in an executable language, system execution can be simulated, making it easier to verify certain properties (early prototyping). Other reasons to use formal specifications typically include the need to express user requirements clearly, and to produce a reference guide for the system implementer [13].

In the formal analysis approach, a security protocol (or architecture) is commonly described as a process in an executable specification language. This process is designed to act in a hostile environment, usually represented as another process of the language (the attacker). In the worst-case analysis scenario, the attacker has complete control over the communication network, for example it can intercept, fake and "steal" all communications [11]. The entire system can be analyzed by applying specific techniques. For instance, security is sometimes analyzed by comparing the state-space resulting from the execution of the protocol with and without the attacker. The differences may represent possible attacks that have to be carefully studied. It is worth

noting that the attacker is able to deduce new messages from the messages it has received during a computation.

The basic algebraic features of cryptographic functions are represented as rewriting rules for terms of a language that denotes cryptographic messages [8]. This means that there may be rules that allow an attacker to discover a message encrypted with a certain key when the attacker also holds the correct decryption key. Analysis methods of this type can be also implemented in automated software tools. These tools can be used by (reasonably) non-expert people and, hopefully, by the end-user of a security application in order to achieve a better comprehension of the security mechanisms offered by the application itself.

2.6. Formal Analysis and the SPIN model checker

Trying to apply Formal analysis, we have chosen a tool called the SPIN model checker with which we are going to perform formal analysis into the security protocol that we want to check.

SPIN is a tool for analyzing the logical consistency of concurrent systems, specifically of data communication protocols [5]. It uses a modelling language called PROMELA (Process MEta LAnguage) to describe the behaviour and interaction between different processes; invariants can be represented as assertions. The tool SPIN works on the PROMELA description that we have created directly from the system that we want to check and checks for absence of deadlocks, unspecified receptions, and un-executable code. SPIN is designed to attack software problem, but is also helpful in logical design, because it provides a method of formal verification.

There are two different capabilities that the tool can provide us for our model to be performed onto [5]**.** *Simulation,* where SPIN can work with the

predefined PROMELA description by performing a random simulation of it and *Verification,* where SPIN after simulating the PROMELA model, it can then generate a C program, which in his turn it performs an exhaustive search in the whole state space of the Model, to verify whether or not the design is error-free. There also other kinds of verification checking that the tool can offer us, but we are only going to work with the default settings and options of the model checker initially has.

In this project we explore the possibility of using SPIN, which is one of the most powerful model checkers [13], to verify cryptographic protocols. The main idea is that the protocol configuration to be checked (i.e. the protocol sessions included in the model) can be statically analyzed simulated and verified in order to collect data-flow information, which can be used to simplify the intruder knowledge representation and generally security violated issues. For example [7], such a preliminary analysis can identify which data can potentially be learned by the intruder and which cannot, thus avoiding the representation of knowledge elements that will never occur. Similarly, it is possible to foresee which messages that the intruder could build will never be accepted as valid by any protocol agent, and avoid their generation. In the following chapter we are going to have a detailed description of the SPIN model checker and its capabilities, among with an approach of applying SPIN to a security protocol.

Chapter 3

The SPIN Model Checker

3. The SPIN Model Checker

3.1 Introduction to the SPIN model checker

SPIN is an efficient verification system for models of distributed software systems. It has been used to detect design errors in applications ranging from high-level descriptions of distributed algorithms to detailed code for controlling telephone exchanges [7]. SPIN verification models are focused on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. Process interactions can be specified in SPIN with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these. In focusing on asynchronous control in software systems, rather than synchronous control in hardware systems, SPIN distinguishes itself from other well-known approaches to model checking [5].
As a formal methods tool according to [7], SPIN aims to provide:

- an intuitive, program-like notation for specifying design choices unambiguously, without implementation detail
- a powerful, concise notation for expressing general correctness requirements and
- a methodology for establishing the logical consistency of design choices matching the correctness requirements

In SPIN the notations are chosen in such a way that the logical consistency of a design can be demonstrated mechanically by the tool. SPIN accepts design specifications written in the verification language PROMELA and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL) [7].

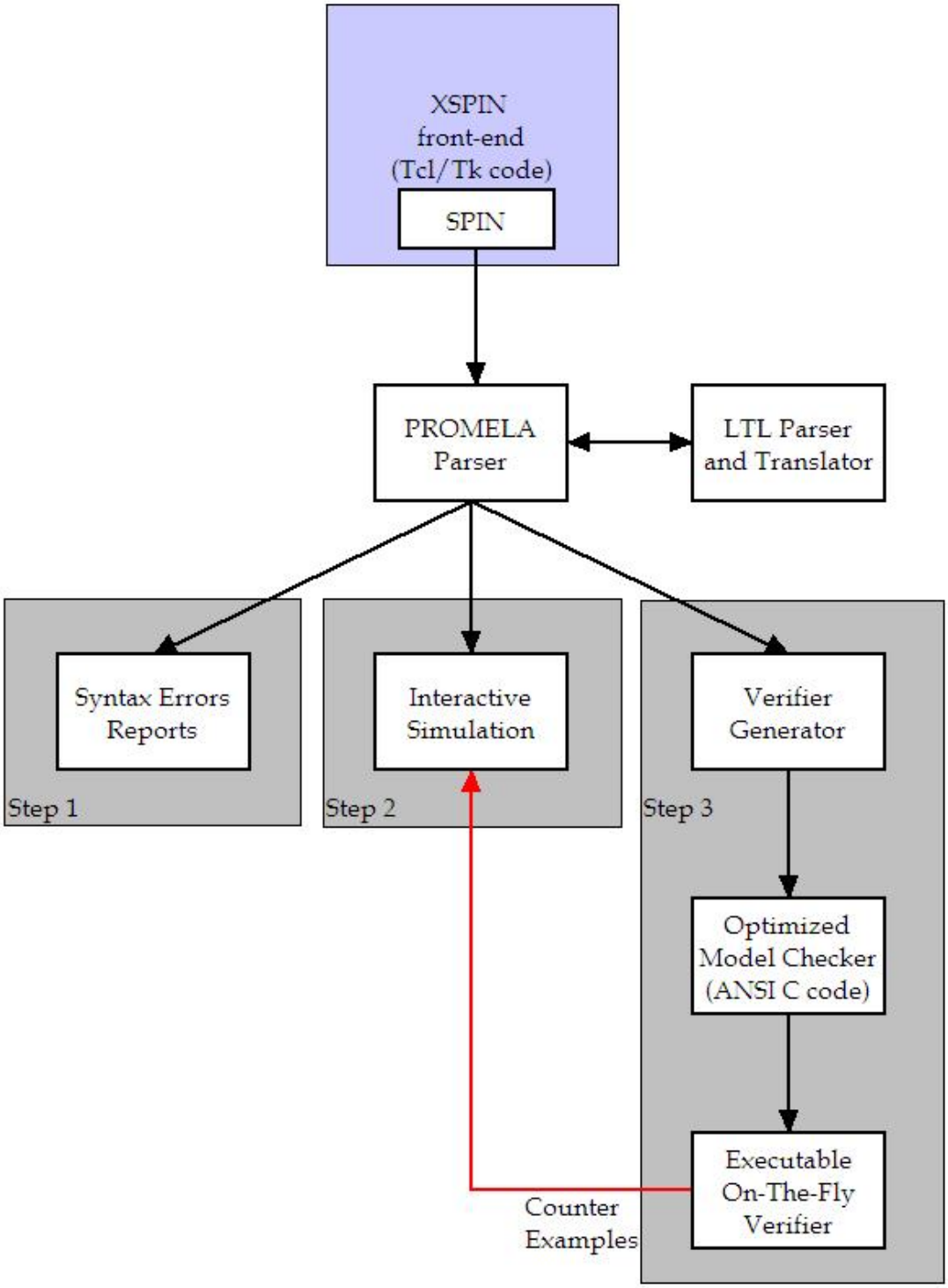

Figure 3.1: The structure of the SPIN model checker [7]

The basic structure of the SPIN model checker is illustrated in Figure 3.1. The typical mode of working is to start with the specification of a high level model of a concurrent system, or distributed algorithm, typically using SPIN's graphical front-end XSPIN. After fixing syntax errors, interactive simulation is

performed until basic confidence is gained that the design behaves as intended. Then, in a third step, SPIN is used to generate an optimized on-the-fly verification program from the high level specification. This verifier is compiled, with possible compile-time choices for the types of reduction algorithms to be used, and executed. If any counter-examples to the correctness claims are detected, these can be fed back into the interactive simulator and inspected in detail to establish, and remove, their cause.

3.2 Safety and Liveness properties

Various classifications of program properties appear in the literature, where each class is usually characterised by a canonical temporal formula scheme [15, 16]. In our work, in association with the SPIN model checker we consider two popular classes of properties, safety and liveness, also referred to as invariance and eventuality [15].

Informally, a safety property claims that "something bad" does not happen in the system that is been checked [15]. SPIN give us the opportunity of checking the above safety properties:

- Assertions[4], which are statements local or global in our model that can be true or false during a random simulation-run using SPIN [**32**]. In this case we define assertions either expressing a desired behaviour that our model (and indirectly our system want to have) or an erroneous one.
- Invalid End-states, which are erroneous states that our system can come into [26], for example deadlock or livelock.

[4] Defining assertions in our model we can using SPIN not only to verify these assertions but also to detect our model for deadlock or livelock situations in respect to the pre-defined assertions

On the other hand, a liveness property claims that "something good" eventually happens, meaning that a program eventually enters a desirable state [21]. SPIN also give us the opportunity of checking the above liveness properties:

- Progress-State Labels, which are labels that allow the designer to explicitly define a notion of progress and thus verify that certain events, do actually occur [26]. A verification of a model that in it we have defined a progress state label will fail, if there exists an infinite execution cycle that does not pass from a progress state label; this situation is known as a non-progress cycle.
- Acceptance-State Labels, which are labels that allow the designer to be able to verify that certain events, do not happen infinitely often [26]. A verification of a model that in it we have defined an accept-state label will fail, if there is an execution that visits an accept-state label infinitely often; this situation is known as an acceptance cycle.
- Weak Fairness[5], which is a liveness property where a process of our model that continuously makes a request, will eventually be serviced [26]. SPIN model checker supports checking weak-fairness requirement by expressing it within an LTL property[6].

3.3 Applying SPIN to a security protocol

Being more specific about the SPIN model checker and the way we are going to perform formal analysis in a security protocol, we must at first display our

[5] Fairness is a special case of liveness and relates to the how the underlying process scheduler deals with contention, for example different clients that competing for the same computational resource. There are two notions of fairness: Weak-fairness and Strong-Fairness [26].

[6] SPIN can be used as a full LTL model checking system, supporting all correctness requirements expressible in linear time temporal logic.

reasons of choosing particularly SPIN, as the mean of formal methods in order to verify our protocol. In the above lines we have already described the basic modes that the SPIN model checker can be useful to us. Apart from the simulation and the verification, SPIN has the following features, according to [27]. These are:

- SPIN is created for efficient software verification, not hardware verification. That reason makes the tool an exceptional choice for simulating, verifying and finding errors among several software systems such as operating systems, data communication protocols switching systems, concurrent algorithms and railway signalling protocols.
- SPIN supports a high level language as input (PROMELA) in order to specify systems descriptions. With the help of PROMELA, SPIN produces efficient results when checking the logical consistency of a specification. After applying SPIN on a pre-modelled security protocol, SPIN can provide us with reports on deadlocks, unspecified receptions, flag incompleteness, race conditions and unwanted assumptions about the relative speeds processes.
- SPIN has the feature of performing automate on-the-fly model checking allowing us to avoid the need of pre-constructing a global state graph or a Kripke structure [13], as a prerequisite for the verification of our security protocol properties.
- The tool also can be used as full LTL (Linear Temporal Logic) model checking system supporting all the correctness requirements (which are expressed in linear time temporal logic), having the opportunity to check the model for basic safety and liveness properties. According to that we can not only check a protocol about correctness of its properties but also we can model an "intruder" invasion during its communication with an expression of an LTL formula, trying to see the responding behaviour that that protocol is going to come into.

- Finally, SPIN is a tool that supports both rendezvous and buffered message passing among processes that are synchronised, making the choice of applying SPIN to a security protocol, ideal as far as verifying correctness of communication properties.

A useful example that also empowers the selection of this tool is a former successful appliance of SPIN to the NetBill protocol, according to [4]. In the paper [34], Fanjul, Tuya and Corrales have modeled the basic communication and message passing of the NetBill protocol in PROMELA, stating the well expressiveness of the language that help them structuring and implementing their model. Apart from that, and after applying SPIN to the model, the tool has produced detailed simulation and verification results [4]. These results have shown at first the function of the protocol in analogy with time, giving information about each state the protocol has reached and whether or not the safety properties that have initially been specified are being obeyed by the protocol.

Although the above features that encourage us to choose SPIN as our tool for applying formal analysis to a security protocol, there are today a number of model checkers that can also verify efficiently systems and properties of them. In the following paragraph a short description of such tools is going to be given focusing on the most known model checkers.

3.4 Formal approaches

A large number of model-checking tools have been developed over the years in order for formal analysis to take place. This paragraph provides an overview of some well-known model checkers.

CADP

CADP (CÆSAR – ALDÉBARAN Development Package) [17] is a verification toolbox for the design and verification of communication protocols and distributed systems, specified in the language LOTOS[7] (Language of Temporal Ordering Specifications). CADP accepts low-level specifications in terms of LTSs[8] (Labeled Transition Systems), and also supports intermediate formats that allow verification of protocol descriptions written in other languages such as SDL (Specification Description Language) [18]. The toolbox contains several closely interconnected components accessible through a graphical user-interface. The functionalities offered include interactive or random simulation, partial and exhaustive deadlock detection, and verification of behavioural specifications with respect to various equivalence relations, as well as verification of branching time temporal logic specifications in the logic XTL (eXecutable Temporal Language). LTSs may be represented either explicitly or implicitly in terms of BDDs[9] (Binary Decision Diagrams). On-the-fly verification can be applied, and so can compositional state-space generation. A number of case studies have been performed with the CADP toolbox (an extension of the initial CADP model checker), including several industrial applications [19].

Concurrency Workbench

The Concurrency Workbench [20] is a tool that incorporates several verification strategies. A system is modelled as a CCS[10] (Calculus of

[7] LOTOS is an international formal specification technique for specifying concurrent and distributed systems [**29**].

[8] LTS is a description of a specification of a system which contains all the states a component may reach and all the transitions it may perform [30].

[9] BDDs are diagrams that offer an efficient way to represent and manipulate Boolean functions [13].

[10] CCS is a classical, widely-accepted formalism for representing systems of concurrent Processes.

Communicating Systems) process [21]. Processes are then interpreted as LTSs for verification purposes. The tool supports three different approaches to verification. Firstly, it checks behavioural equivalence between the LTS of the system and that of its specifications. LTS minimisation can be performed with respect to these notions of equivalence. Although minimisation is a facility provided by the tool, compositional state-space generation is not mentioned as a possibility in [20], nor is any attention given to the problem of intermediate state explosion. Secondly, preorder checking can be performed between the system and its specifications. Thirdly, the tool supports model checking of specifications written in a modal logic based on the propositional μ-calculus. However, formulas in this logic are unintuitive and difficult to understand. For this reason, the tool offers the facility of user-defined macro identifiers. In this way, users are able to code intuitively well-understood operators as macros.

There are other extensions of the specific tool that have been created in order to achieve higher rates of effectiveness upon the systems to be checked. One of these is the NCSU Concurrency Workbench[11] (North Carolina State University where the tool have created), an extension of the Concurrency Workbench [20]. Finally, (another extension of the initial tool) the Concurrency Factory [20] can be viewed as a next-generation Concurrency Workbench which is a development environment for concurrent systems, with a focus on usability. It allows non-experts to design concurrent systems using GCCS[12] (Global Command and Control System), a graphical version of the process algebra CCS.

---

[11] The NCSU Concurrency Workbench is a tool for verifying finite-state systems. Its key feature is its flexibility during verification of a system.

[12] GCCS is a graphical language which resembles informal design diagrams drawn by engineers.

COSPAN

Cospan [22] takes the automata-theoretic approach to verification. It performs verification by checking inclusion of the language of the system in that of its desirable properties, such as safety properties. The native language is S/R (Selection/Resolution) but interfaces have been written for the commercial hardware description languages Vérilog[13] and VHDL[14] (which stands for Very high speed integrated circuit Hardware Description Language) .The semantic model is founded on ω-automata. In general, a system consists of a collection of such automata. To facilitate property specification as w-automata, a library of parameterised automata is provided. Counterexamples are returned when property violations are detected in a system. COSPAN can use either symbolic (BDD-based) or explicit state-enumeration algorithms. The latter invoke caching and bit-state hashing options (as related to on-the-fly verification), as well as minimisation algorithms. Several other reduction strategies are supported such as automated localisation reduction, symmetry reduction, and user-defined reduction (based on the idea of abstract interpretation). COSPAN also supports top-down design development through successive refinements.

FC2TOOLS

FC2TOOLS (the next-generation AUTO/GRAPH[15]) [23] is a verification tool-set that supports graphical specification of concurrent systems. Reachability analysis, minimisation, equivalence checking, and model abstraction can be performed on automata represented either symbolically or explicitly. Moreover, compositional minimisation can be applied on a hierarchical

[13] Vérilog is a hardware description language; a textual format for describing electronic circuits and systems [31].

[14] VHDL is a programming language that has been designed and optimized for describing the behaviour of digital systems [32]

15 AUTO/GRAPH (also named atg) is a graphical editor for process algebra terms and automata. AUTO/GRAPH pictures are translated into Fc2 format for interface for later use with the model checker [33].

network of processes, although intermediate state explosion is not addressed. The tool-set also supports the specification of properties in terms of automata, and implements on-the-fly techniques for checking them.

FDR

FDR is a tool based on the theory of CSP (Communicating of Sequential Processes) [24]. FDR establishes whether a property holds for a system, by checking that the system refines its property in the traces, failures, or failures-divergences model. The standard model used is that of failures divergences, hence the name of the tool (Failures-Divergence Refinement). Both the system and the property are specified in a machine-readable version of CSP, and their specifications are translated into finite LTSs. For checking refinement, the property LTS is normalised before model checking is applied. This can be a problem, as normalisation may increase the size of an LTS exponentially. Additionally, a system can be developed by a series of stepwise refinements, starting with a specification process and gradually refining it into an implementation. Finally, FDR supports compositional minimisation, where intermediate systems can be simplified with a variety of compression techniques.

SMV

SMV [25] is a tool for checking finite-state systems against specifications in the temporal logic CTL. It supports a flexible specification language and uses an OBDD-based **(**Ordered Binary Decision Diagrams) symbolic model-checking algorithm for efficiently checking whether CTL (Computation Tree logic) specifications are satisfied by the system. The tool has been used to verify several industrial designs such as the Futurebus+ and the Gigamax protocols [2**].**

Chapter 4

Requirements Capture of the Model

4. Requirements Capture of the Model

4.1 Introduction to requirements capture

It is an important thing, in every implementation of a project for the requirements and the specifications of it to have been stated in the firsts steps of the designing procedure. The structure of the chapter is created in this way in order to present all the necessary rules that must be obeyed according of course to the nature of the problem that our project wants to represent.

Before starting with the definition of the requirements we should make an introduction about the correct capture of them, in order for our model to be built according to specific guidelines. Looking at previous works of modelled protocols in PROMELA [2] we can see examples of rules, requirements and specifications of such models.

4.2 Requirements of the implementation

In order to represent a cryptographic protocol into a representative model that will allow us to apply formal methods on it, we must take into consideration principles and rules that the protocol need to obey to, for a successful function of it. Trying to implement these rules we assume at the beginning, that the algorithm being modelled is perfect and the implementation of its cryptographic algorithm is being done in an abstract way. Such an assumption is being based in [4] previous work, during the modelling of the NetBill protocol.

The modelling of a cryptographic protocol such as the Needham Schroeder in our case is a set of principals which send messages between agents according to the protocol's basic function. In order for the correct modelling of our second section model we must agree that before moving to this point, that the

protocol is modelled in an ideal way having proved that there are no errors during its functioning. In our case, we are going to separate the whole phase of our project into two parts: the first[16] one (part 1) will be the modelling of the basic function Needham Schroeder protocol, using two agents (Initiator Alice and Responder Bob), and the second part (part 2) will be the modelling of an intruder attack during the communication between these two agents.

We assume for the Needham Schroeder public key protocol that the following principles are valid during the communication and the exchange of messages:

- Agents A, B and I are the initiator, the responder and the intruder respectively, who try to establish communication and exchange a secret represented by their pair of nonces (for the first two agents), while the intruder tries to interfere this communication by repeating messages (freshness attack).
- The goal of the protocol is to try and convince each other of their presence and identity.
- The goal of the intruder is to convince either agent A or B that the communication is correct in both directions of it.
- Initiator A knows responder's B public key and the responder B knows initiator's A public key (so there is no need for communication by the two hosts with a trusted session S that will provide each of them the needed public keys.). The intruder I, know also the public keys of the agents A and B.
- The only way for an agent A to decrypt an encrypted message coming from an agent B is to know B's corresponding key.
- An encrypted message in no way does reveal its key that was used to be encrypted.

[16] We are going to refer to the part 1 model of our project also as protocol model, while for the part 2 model we are going to refer to us the intruder's attack model.

- Initially, each agent has a (non-corrupted) pair of keys and has made up a nonce.
- There is sufficient redundancy in messages so that the decryption algorithm can detect whether a ciphertext was encrypted with the expected key.
- The result that we expect from our first section model is that the protocol is well functioning between the agents A and B (proof that is produced from the SPIN model checker output).
- The result that we expect from our second section model is that whether or not an intruder can breach security into the communication of the protocol between at least two [34] agents A and B.

4.3 Moving to the PROMELA model

According to our project plan of the implementation that have been agreed to follow, we should take a closer look to the modelling procedures of our two models, giving details about their initial design and their "constructing" rules that their based on.

Starting from the part 1 model, we must create a PROMELA model of the Needham Schroeder cryptographic protocol instance, to be used for security property verification. Such a procedure is illustrated using the reduced situation of the Needham-Schroeder public key authentication protocol as it has been mentioned previously in this paper (see chapter 2 page 15). The basic mechanism that we are going to model is agreed to the diagram of the protocol showed in the figure 4.3.1 below.

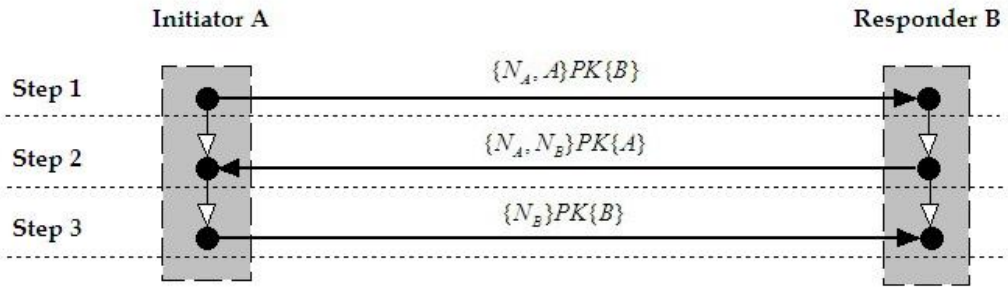

Figure 4.3.1: The reduced Needham Schroeder protocol

As we have mentioned in previous chapter, the Reduced Needham Schroeder Public Key protocol requires three basic steps to be completed. At first the initiator A sends a message, including its generated nonce (Na) and his identity, to the responder B, encoded with B's public key. Then B is responding with a second message including its generated nonce (Nb) and nonce Na encoded with A's public key. When initiator A receives this message he decrypts the message and sends back to B its nonce Nb encoded again with B's public key. Before the communication starts we have assumed that this is an instance of the protocol where Initiator A and Responder B knows the public keys of each other, so there is no need for communication with a trusted session S in order to acquire public keys.

Using the SPIN model checker and its capability of the correct representation of protocol in its input (PROMELA) language, both the two agents that we are going to use in our model are represented as different (active) processes that communicate with each other by exchanging messages using a shared synchronized communication channel. This means that in the simplest communication attempt between an initiator A and a responder B, each agent who is sending a message to the other one should wait for a message-response from the other until the moment that he will receive one.

In a different situation and having noticed the absence of time nonces in our model, SPIN model checker is going to indicate us a time error problem, as the one of the two processes cannot produce the corresponding message and though the modeling of our protocol has design errors. After the exchange of each message, an integrity check of the message should take place from each agent-process, by comparing the produced and nonce that have been sent to each other.

After having the results that indicates us that the protocol model in part 1 is correct, we are going to move to the modeling of the intruder's attack to the protocol. The whole idea of finding an attack to a protocol is being based on the theorem of reducing the number of agents that participate to this communication. According to this theorem being defined in [34], in order to find a possible security "hole" (such as a successful intruder's attack) in a protocol, if there is one such a security-error , than we can identify this by the use of the minimum number of agents taking part in the protocol's communication.

In our case, the part 2 model of our project that has to do with the intruder's definition is an extension of our part 1 model, as shown in figure 4.3.2. In this part, the process of the intruder is being defined while the predefined processes of the agents A and B have been slightly changed in the point of the determination of the partner that they (?think) that they communicate.

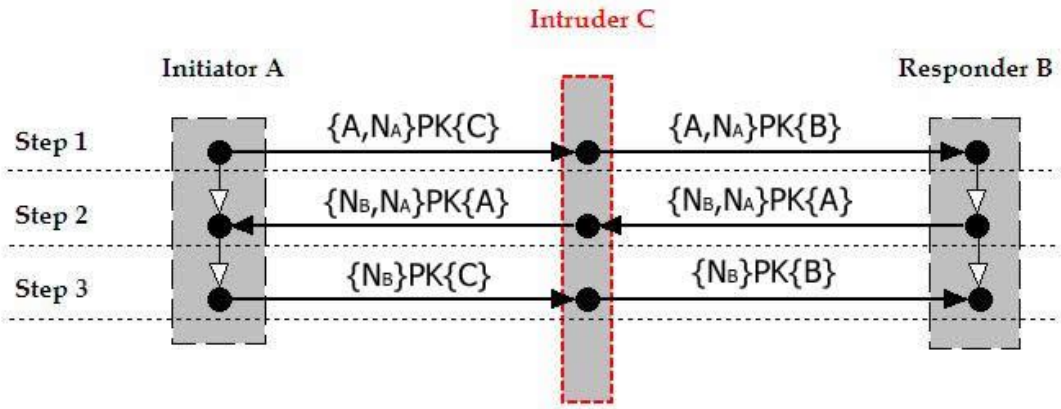

Figure 4.3.2: Intruder's attack in the reduced Needham Schroeder protocol

The basic mechanism of the intruder is to gain as much knowledge as he can, during the communication of agents A and B. Every time the intruder intercepts a message, he increases his knowledge. This point can be crucial as the intruder does not have (in our case) the possibility of decrypting the intercepted message, so the knowledge that he gains is simply a packet of encrypted data that has a certain destination (agent B). Having this, the intruder can send repeatedly this message to the network in order to fake himself of being the first agent who sent the message (the agent whom he has the packet acquired from). Since we are interested in modelling the most powerful intruder, we assume it always learns as much as possible from the intercepted messages.

4.4 Requirements conclusion

After defining the appropriate requirements for our model, we should get started with the actual implementation of it. Before that we should address that requirements capture has an important role in the begging of the implementation of each system (real or not).

In previous works [4] of modeling protocols, we can see that an actual protocol cannot be always perfectly modeled, as the real time states that the protocol can come into are quite large in number. The solution to this is the restriction of our modeling by concluding the situations that we want actually to examine the behavior of the protocol into it. That is the reason why in most cases of modeling we always take into account principles that can help us focus on the specific point of function of it in which we want to check. This reduction of the possible states of the model not only can be helpful in the modeling procedure but also can be proved as a wise solution in most of cases where error-find methods can be time consuming. In chapter 5 we are going to proceed into the actual implementation of our model. The best way of this is by creating at first diagrams of the communication of the protocol that represents the actual communication of the agents step by step. Afterwards we can proceed to the coding of the model according to the principles we have defined that also the representative diagrams must be comply with.

Chapter 5

Implementation of the Model

5. Implementation of the Model

5.1 Introduction to the implementation.

In this chapter we are going to start implementing the PROMELA model of our project with which we perform the formal analysis upon the Needham Schroeder protocol. All the steps of the initial design of the model are going to be described (discrete steps of communication of the protocol), followed from the description of the actual coding of the protocol. At first we should keep in our mind the principles that have been in defined in chapter 4 in order to be close to our goal. As we previously said we separate our implementation phase into two separate parts: we implement the model of the Needham Schroeder protocol, referring to it as protocol model (part 1) and later, based on this model we implement the intruder's attack model (part 2).

5.2 Modeling of the protocol model

We separate the protocol model implementation into the description phase and into the implementation (coding) phase. In the first phase a detailed description of each execution step of the agents is given, while in the second phase the actual coding of the model is presented and explained.

5.2.1 Description of the Initiator (A) and the Responder (B)

At the beginning of the first section of our coding procedure (which is the implementation of the Needham Schroeder protocol), we should address that the basic processes of our model will be the two agents that are trying to establish communication, initiator A and the responder B. We try to create a sequence of required actions that each of the agents must follow during the exchange of messages between them. Such actions have a connection for agents A and B for the reason that A and B must be synchronized for their

actions during their communication. This synchronization mechanism is based on the actual function of the Needham Schroeder public key protocol and must be verified with the initial options that have been defined at the start of chapter 4. Before moving to the actual part of coding the model, we should consider each agent as a process, defining with the correct sequence the appropriate actions-steps that must occur for the correct function of the protocol.

At first the communication is started from the initiator A, which is actually the signal for the starting of the actual mechanism of the protocol. Then the agent A must choose non-deterministically a partner for the communication which in our case is the responder B. As we can see from the figure 5.2.1, the third action for A is to construct the first message (the first step of the protocol) which contains its identity and a generated nonce Na, all together encrypted with B's public key (the key we have assumed that is already known from the initiator A) and send it to B.

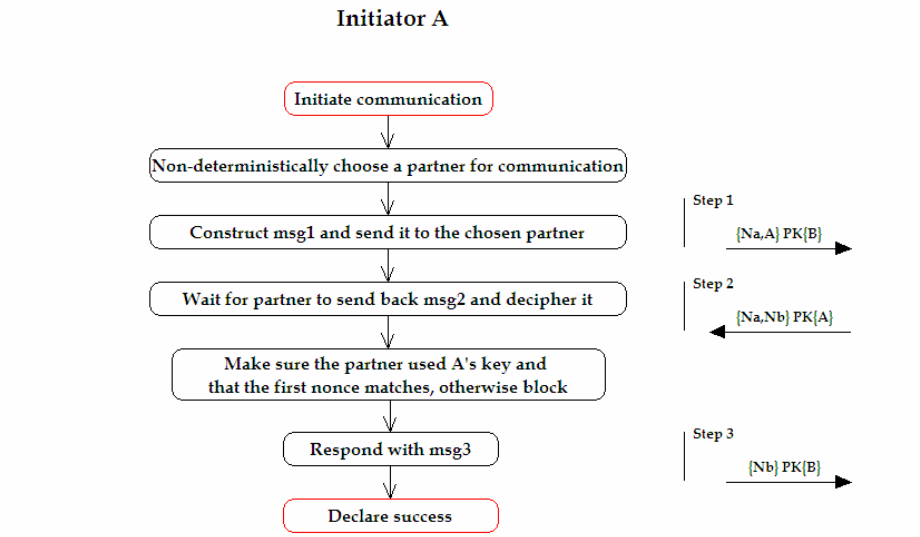

Figure 5.2.1: Sequence description of the initiator A

As we can see from the figure 5.2.2, the Responder B is currently waiting to receive the first message (msg1) and then he will try to decipher the message with his private key in order to acquire the A's nonce and its identity. After

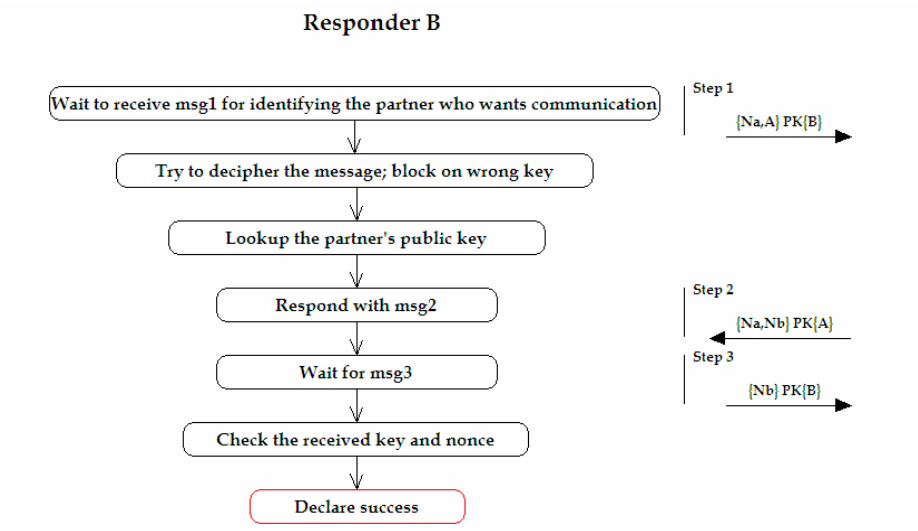

Figure 5.2.2: Sequence description of the responder B

looking up A's public key (we have assumed that B has already A's public key acquired from a previous communication between them), B is responding to A with the second message (msg2) containing the nonce Na and its generated nonce Nb, both encrypted with A's public key (second step of the protocol). When agent A receives the second message he at first check if the message is encrypted with his key; then he check if the received nonce (from his partner) matches the nonce that at first he has generated. After the checking he then constructs the third message (msg3) corresponding to the third step of the Needham Schroeder protocol and send it back to the responder B followed by the declaration that his status is ok, a message that contains B's nonce Nb encrypted with B's public key. Finally, B is receiving the message and after decrypting it and checking if the nonce that he received matches with the one already created, he declares that his status is ok, meaning that he is sure that is going to communicate with partner A.

5.2.2 Implementation of the protocol model

At first we must mention the agents that are going to take part to this communication. In our case we are going to have agent A which is the initiator (for our easiness of coding we refer to this agent with the name Alice)

and the agent B which is the Responder (for our easiness we are going to refer to this agent with the name Bob). For the best modelling of the protocol we must use names that can help us through the entire implementation of the model. This leads to the definition of a set of identifiers, keys, nonces and data that are going to be used to our model as variables. The resulting set of names that we use into our first section model will be the following:

```
mtype = {msg1, msg2, msg3, alice, bob, nonceA, nonceB,
                    keyA, keyB, OK};
```

where msg1, msg2, msg3 are the three messages that are being exchanged during the communication of the two agents (3 messages for 3 steps respectively), Alice is the Initiator agent and Bob the Responder one, nonceA is the nonce generate from the initiator and nonceB is the nonce generated from the responder, keyA is the initiator's public key where keyB is the responder's public key. Finally OK is the message that is being send to the message sequence chart (MSC) that the SPIN model checker is producing, indicating that the status of initiator and the status of the responder is successful, meaning that the communication being established is secure.

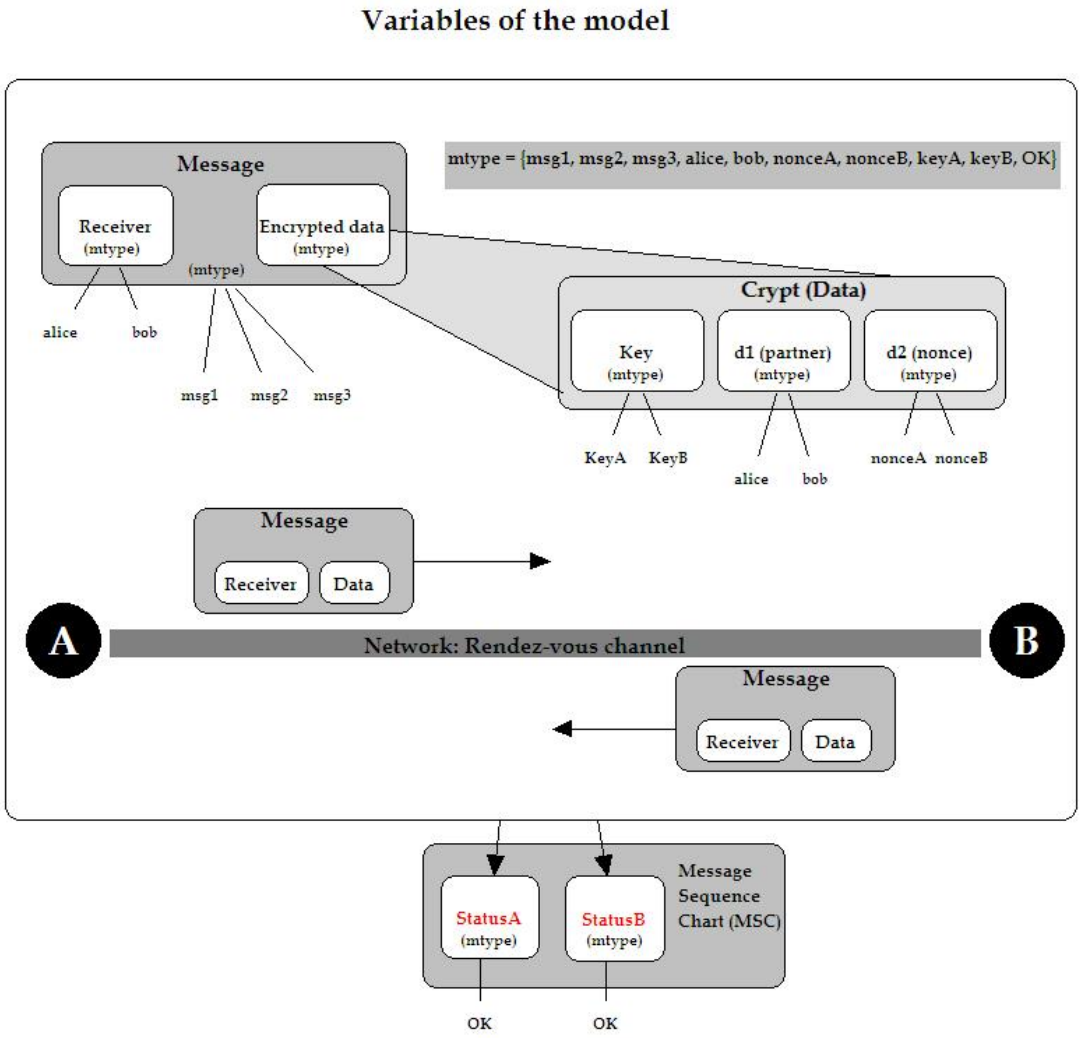

Figure 5.2.3: Variables and possible values of the PROMELA model

The second step in the construction of the model is the definition of the channels used by the agents to communicate with each other. A global channel is defined in order to pass the messages from the one agent to another. As we have mentioned in the above lines we must create a synchronization mechanism which will allow the perfect communication between the two agents A and B. In both three steps of the protocol the two agents must wait each other for authentication and checking controls upon the received messages. That is the reason why we choose to create a rendez-vouz channel named network and is defined as followed:

```
chan network = [0] of {mtype, mtype, Crypt};
```

A diagram of the variables that are being used to our model is shown in the figure 5.2.3. In the diagram we can see not only the variables that we are going to use to our model but also the possible values that these variables can get, during the simulation run of the model.

A simple message that the two agents are exchanging during their communication, this is modelled as a tuple of an mtype for the receiver and a triple mtype data which contains agent's public key, identity of the partner and the nonce that has been generated (three mtypes). The message that we model is defined as follows:

```
typedef Crypt { mtype key, d1, d2;}
```

Also, we define mtypes partnerA, partnerB and statusA, statusB for the partners of the agents the first two and for the declaration of the status of each partner the last ones. This definition is the following:

```
mtype partnerA, partnerB;
mtype statusA, statusB;
```

Moving to the implementation of the agents of our model, we start from the modelling of the initiator A, which we refer to with the name Alice, for the easiness of coding and understanding of the protocol. After the declaration of the appropriate local variables of the proctype, Alice must choose non-deterministically the agent that she wants to communicate with (this is a crucial point for the intruder's invasion into the communication as we are going to see later into the second section of our model). If the partner of Alice is Bob, then the key that Alice is using to encrypt her message should be Bob's public key, keyB.

```
if
:: partnerA = bob; partner_key = keyB;
```

```
:: skip;
fi
```

Then Alice must construct the first message (msg1) that contains the essential information to be sent to her partner. This deterministic step is defined as follows:

```
d_step {
     data.key = partner_key;
     data.d1 = alice;
     data.d2 = nonceA;
     }
```

We can see that have no modelled the first message of the protocol which is an encrypted with keyB message (partner_key) containing Alice's identifier and the generated nonceA that Alice has produced. After that the message is being sent to Bob (partnerA) through the channel network,

```
network! msg1(partnerA, data);
```

while Alice is entering a waiting situation until she receives the second message from her partner Bob.

```
network?  msg2(alice, data);
```

We can see the whole procedure of the active proctype Alice in the figure 5.2.4.

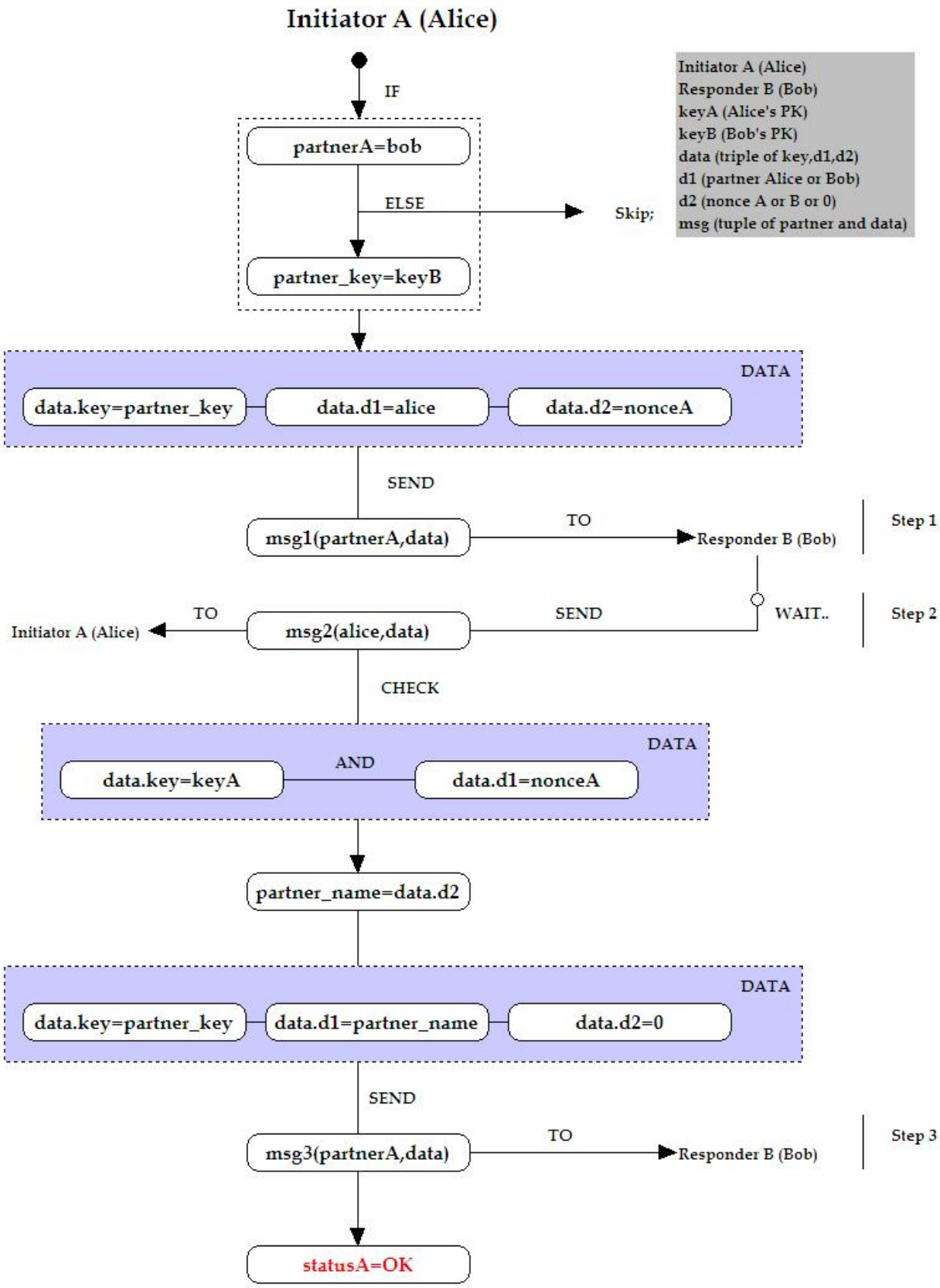

Figure 5.2.4: PROMELA Diagram of the Initiator Alice

At this point and after receiving msg2 from Bob, Alice must check the authenticity of the msg2 that he has received. He must make sure that the partner used A's key and that the first nonce matches, otherwise he is going to block and the model will declare an error.

```
end_errA:
(data.key == keyA) && (data.d1 == nonceA);
```

The initiator Alice has now from his side to construct the third message (and the third step of the protocol). He first assigns his partner's nonce (nonceB) that he had received (from the msg2),

```
partner_nonce = data.d2;
```

and then he constructs his third message, with the following code:

```
d_step {
    data.key = partner_key;
    data.d1 = partner_nonce;
    data.d2 = 0;
       }
```

and sends the message to his partner (through the network channel):

```
network ! msg3(partnerA, data);
```

After the end of each coding step, initiator Alice can declare success from his part as following:

```
statusA = OK;
```

Moving to the implementation of the responder Bob which is diagrammatically shown in figure 5.2.5, the specific agent enters the model in a waiting situation as he must first accept the first (and initiative) message from the agent (Alice) that wants to communicate with him. So at first we define,

```
network ? msg1(bob, data);
```

After receiving msg1, he must check the data that he has just received and block the model if he finds an error. At first he checks for the message if it is encrypted with his public key (keyB),

```
end_errB1:
(data.key == keyB);
```

and assign his partner to the identity variable data.d1

```
partnerB = data.d1;
```

From his side now, he must construct the second message to be sent to his partner. This is the point where initiator Alice is waiting to receive msg2 (actually the second step of the protocol) in order to move on to its procedure. Bob has at first to acquire Alice's public key and also construct the part of the second message. This is done as follows:

```
d_step {
    partner_nonce = data.d2;
    if
    :: (partnerB == alice) -> partner_key
= keyA;
    ::  skip;
    fi;
```

```
data.key = partner_key;
data.d1 = partner_nonce;
data.d2 = nonceB;
 }
```

After doing that, he then send the msg2 to Alice through the network channel (this is the point where Alice is receiving msg2 and continues to construct msg3) :

```
network ! msg2(partnerB, data);
```

and enters a second wait situation where Bob must wait to receive msg3 from Alice,

```
network ? msg3(bob, data);
```

After Alice has sent msg3 to Bob and declares success of its procedure, Bob receives the message, continuing to the authentication point of the protocol, where he has to check if the message that he has received is encrypted with his public key and whether or not his nonce that has sent to the initiator matches with the one that has at first generated.

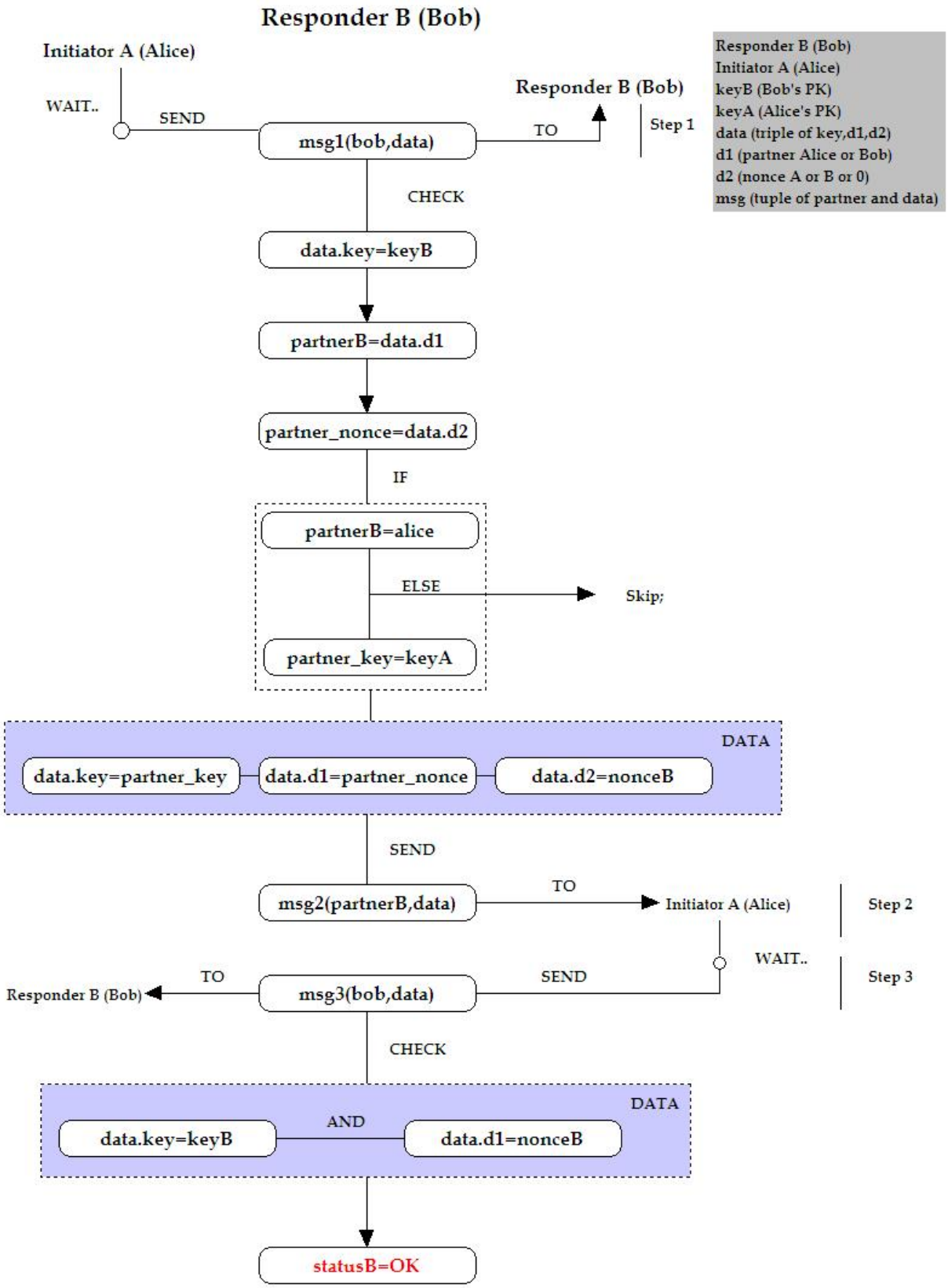

Figure 5.2.5: PROMELA Diagram of the responder Bob

```
end_errB2:
(data.key == keyB) && (data.d1 == nonceB);
```

We can observe the mechanism of the active proctype Bob from the figure 5.2.5 also in parallel with the three steps of the Needham Schroeder protocol. Provided that no errors has been found to the whole procedure, Bob declares from is side that his status was successful,

```
statusB =OK;
```

After completing the first section of our model, we should state that the correction of our model can be seen and proved through the simulation and the verification outputs that the SPIN model checker produces to us (see chapter 6). The reader can see the complete Needham Schroeder model at the appendix A.

In order to move to the implementation of an intruder to the Needham Schroeder protocol, it should be mentioned that the first part of our model so far, is the basis of the part 2 model, as the agents Bob and Alice remain unchanged provided the proof of their successful communication. The complete Needham Schroeder diagram can be seen at figure 5.26.

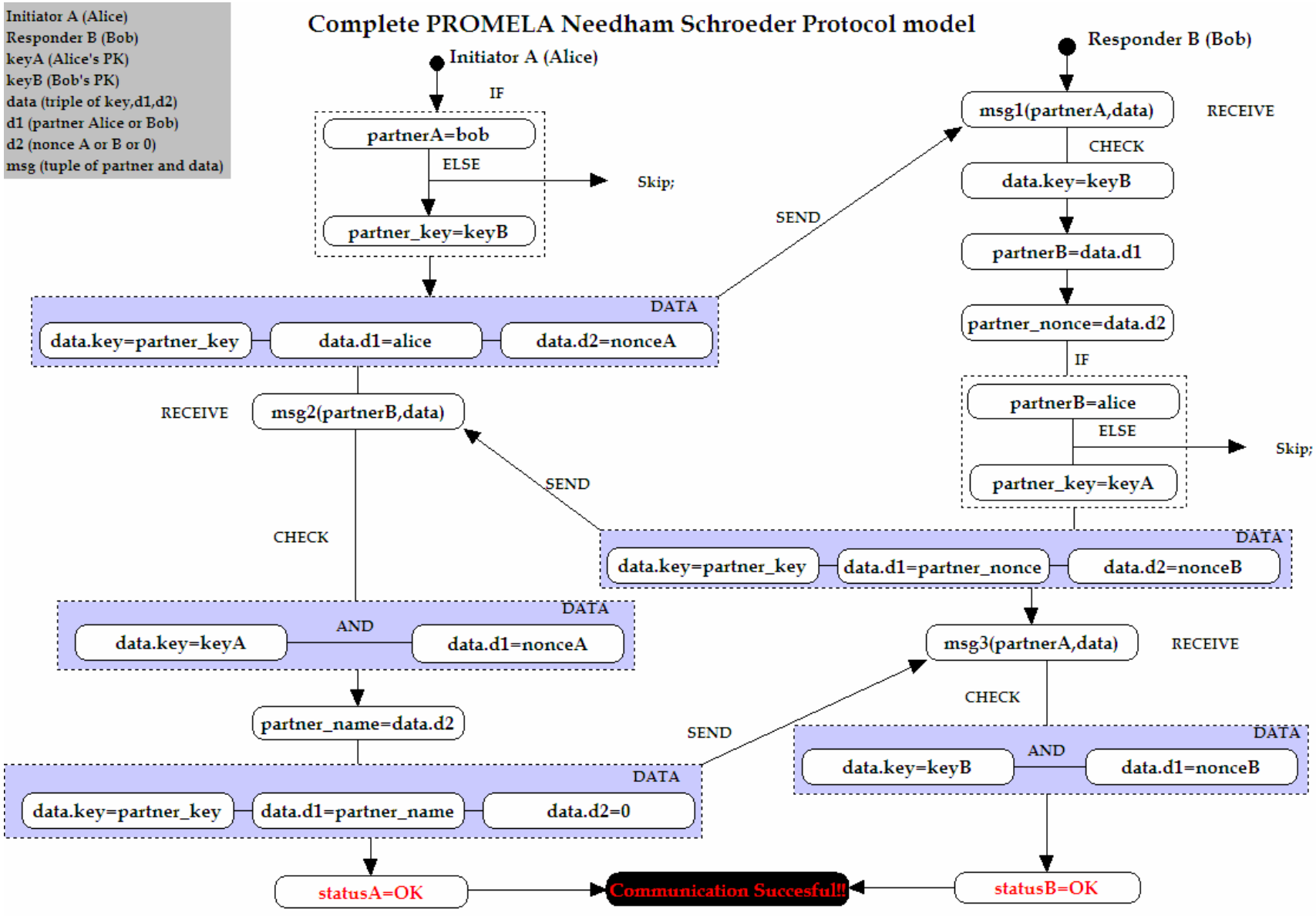

Figure 5.2.6: Complete Needham Schroeder PROMELA model diagram

5.3 *Modeling of the Intruder's attack*

Following the same procedure as the protocol model, we separate the intruder's attack model implementation into the description phase and into the implementation (coding) phase. In the first phase a detailed description of each execution step of the intruder among the agents is given, while in the second phase the actual coding of the intruder is presented and explained.

5.3.1 Description of the Intruder

It the beginning of the second part of our coding procedure which is the implementation of the intruder's attack on the previously defined Needham Schroeder protocol model, we should address that the basic action that we follow is the implementation of the intruder's process (proctype) in our section one model. At first we create a sequence of actions where the intruder is going to interrupt the communication between two agents as shown in figure 5.3.1.

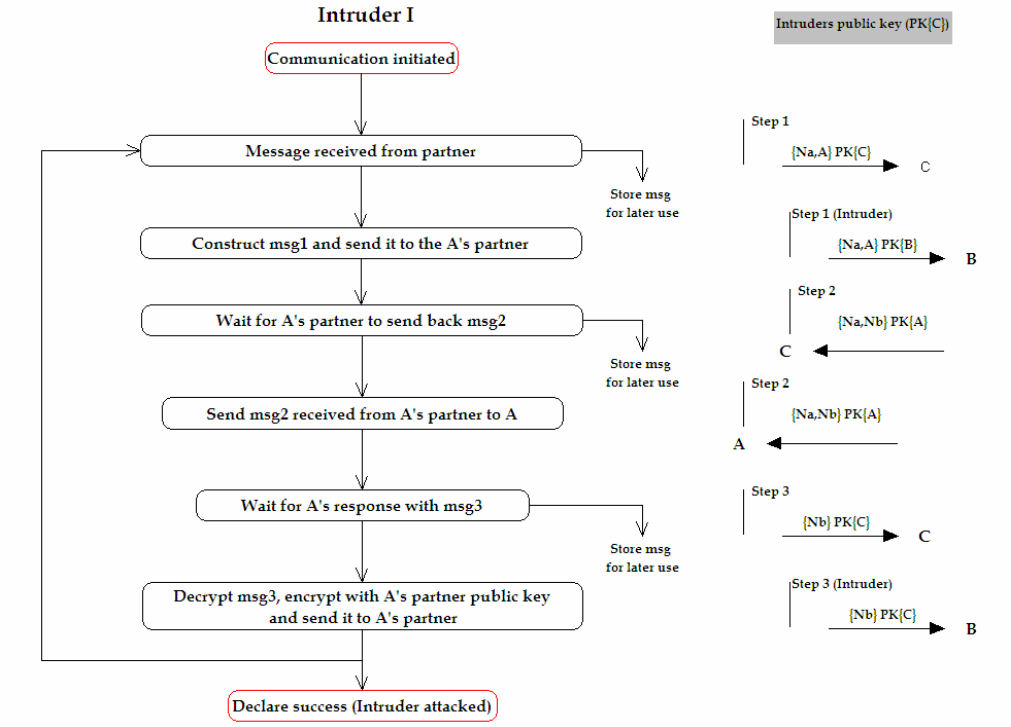

Figure 5.3.1: Sequence Description of the Intruder

As we can see from the figure, the intruder's procedure can be consider as an infinite loop that wait to intercept message or to send previously intercepted messages to both of the agents. This loop is in order with the sequence steps of the Needham Schroeder protocol basic communication steps. But apart from that and because the intruder doesn't know the exact point of the communication of the two agents, he continuously sends messages as he aims into interrupt the communication at any point using any necessary mean. We must also refer to the fact that the intruder is the man on the middle in this case, as he plays the role of the agent both for the initiator A and the responder B, without A or B knowing that. Observing the sequence diagram we can see that there is a complete analogy to the diagrams of the initiator and the responder, as the intruder is constructed in a way of breaching security in the protocol.

Finally, we must focus on the fact that the intruder can interrupt into the protocol in two ways. The first is to construct fresh messages and send them to an agent faking his identity to him and the second one is to use previously intercepted messages from an older communication between the two agents. This is the point where the intruder gains as much knowledge as he can in order to use it in a future communication. We use both these options in our second section model as we have previously agreed in this paper that the intruder gains the most possible knowledge that he can.

5.3.2 Implementation of the Intruder's attack model

Before starting with the implementation of the intruder's proctype, we must refer to the changes (additional code) which we make to our first section model. During the selection of the partner of the initiator A, knowing that this selection is non-deterministic, Alice can easily choose her partner to be either the

responder B (Bob) or the intruder without knowing so as the communication and authentication techniques have not been started. The following PROMELA code represents this:

```
if
:: partnerA = bob; partner_key = keyB;
:: partnerA = intruder; partner_key = keyI;
fi
```

Because the reason that we want the SPIN model checker to find (and prove) the attack on the protocol, the structure of the intruder will be simply all the different actions that the intruder can perform. After the definition of the needed local variables, adding a variable icp_type which represents the intruder's intercepted message,

```
mtype msg;
Crypt data, intercepted;
mtype icp_type;
```

we move to the main body of the intruder which is concluded by an infinite loop while the intruder is always active sending/receiving messages. In the particular point the intruder can generate his own nonce and send it to Bob or to fake himself of being A. That is why he at first sends to Bob a previously intercepted message (msg1) with the use of known nonces or constructs msg1 by himself.

```
if
:: icp_type == msg1 ->
 network ! msg1(bob, intercepted);
:: data.key = keyB;
     if
```

```
        :: data.d1 = alice;
        :: data.d1 = intruder;
        fi;
        if
        :: knowNA -> data.d2 = nonceA;
        :: knowNB -> data.d2 = nonceB;
        :: data.d2 = nonceI;
        fi;
           network ! msg1(bob, data);
fi;
```

We can see that the nonce that the intruder is sending is either previously known nonces of A or B, or his nonce (nonceI). In the exact same way and beginning as if the intruder takes part in the communication for his first time, the intruder can either construct msg2 and send it to Alice A, or he can send a previously intercepted msg2 received from B.

```
if
:: icp_type == msg2 -> network ! msg2(alice,
intercepted);
:: data.key = keyA;
          if
        :: knowNA -> data.d1 = nonceA;
        :: knowNB -> data.d1 = nonceB;
        :: data.d1 = nonceI;
        fi;
        if
        :: knowNA -> data.d2 = nonceA;
        :: knowNB -> data.d2 = nonceB;
        :: data.d2 = nonceI;
        fi;
```

```
network ! msg2(alice, data);
fi;
```

In the third step of the protocol, the intruder acts exactly in the same way:

```
if
:: icp_type == msg2 -> network ! msg3(bob,
intercepted);
:: data.key = keyB;
     if
     :: knowNA -> data.d1 = nonceA;
     :: knowNB -> data.d1 = nonceB;
     :: data.d1 = nonceI;
     fi;
data.d2 = 0;
network ! msg3(bob, data);
fi;
```

So far we saw that the intruder has completed all the sending actions that he can perform in order to interrupt the communication of the two agents. In this point of the proctype we must implement the intruder's interception procedures of exchanged messages. That is why we put the intruder in a wait situation of message in order for him know at first the identity of the agent that has sent the message to him and then (if it is possible) to extract the received nonces.

```
network ? msg (_, data) ->
if /* Store the data field for later use */
:: d_step {
       intercepted.key = data.key;
       intercepted.d1 = data.d1;
       intercepted.d2 = data.d2;
```

```
                icp_type = msg;
            }
        :: skip;
        fi;
```

This is the way how the intruder is intercepting the message and constructs his own fake message using it to breach the security of the protocol every time he wished to. Also, an implementation of a decryption mechanism of the intruder is needed in order for the intruder to decrypt messages which are encrypted with his public key, keyI, in case that an agent has chosen him as his partner.

```
if
:: (data.key == keyI) ->
    if
    :: (data.d1 == nonceA || data.d2 ==
    nonceA) -> knowNA = true;
    :: else -> skip;
    fi;
    if
    :: (data.d1 == nonceB || data.d2 ==
    nonceB) -> knowNB = true;
    :: else -> skip;
    fi;
:: else -> skip;
fi;
```

Completing the implementation of the intruder, we must mention that the intruder has been created gaining the most possible knowledge that he can intercept from a two agent communication. The correction of our second section model can be seen and proved through the simulation and the verification

outputs that the SPIN model checker produces to us. The reader can see the complete Needham Schroeder model at the appendix A. In our case we expect the SPIN model checker to find an error (invalid end-state) during the simulation run of model. In the following figure 5.3.2, a diagram of the attack of the intruder to the Needham Schroeder protocol is shown.

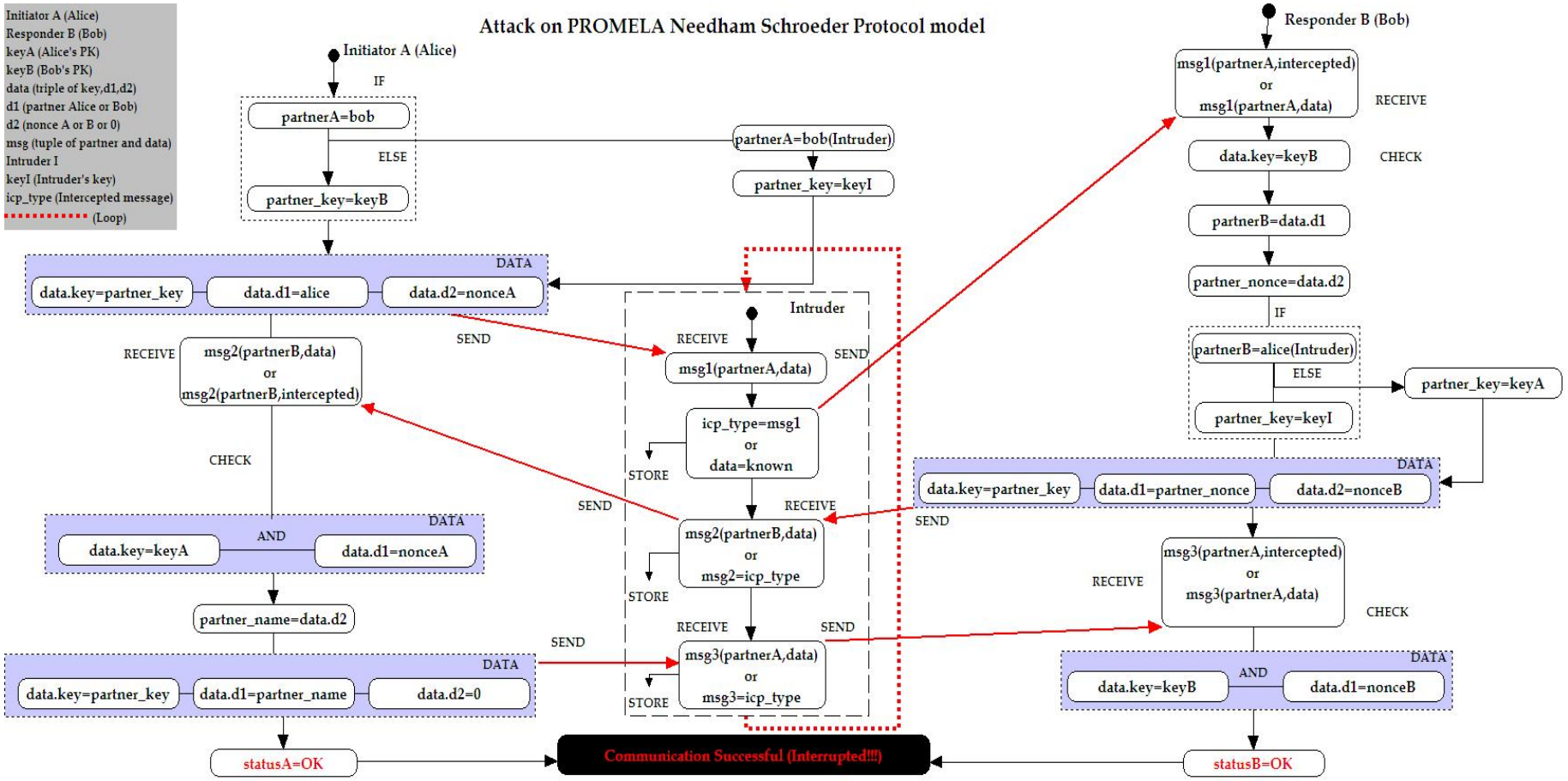

Figure 5.3.2: Intruder's attack on the Needham Schroeder Protocol model

5.4 Conclusion of the implementation

Before presenting the results that the SPIN model checker outputs, while performed upon our two models, we should mention that both of our models are implemented based upon the principles and rules that we have defines in chapter 4. We expect from the simulation and the verification results that the SPIN model checker is going to find no errors on our first model, meaning that the function of the protocol is correct, while in the second section model we expect the model checker to produce an error (communication interrupted) because of the intruder's attack. Chapter 6 is the part of the paper where the results of the simulation and the verification are been shown and explained.

Chapter 6

Simulation and Verification Results

6. Simulation and Verification Results

6.1 Before the SPIN model checking

In this chapter we present the results that the SPIN model checker produces, given as inputs our predefined models. Our effort must focus in the beginning on the first section model which represents the Needham Schroeder protocol. We must ensure that this model is functioning correctly in order to proceed to the part 2 model; this proof is provided by both the simulations and the verification results that the SPIN model checker is returning back to us.

6.2 Formal Analysis of the protocol model

We separate the formal analysis of the protocol model in the simulation results and the verification results.

6.2.1 Simulation results of the protocol model

The first step that we have to do is the correct coding of the models. After the appropriate modifications, we insert our first section model (for the complete model see Appendix A, ex1.prom) and we run a syntax checking in order to eliminate all possible errors. A snapshot of the SPIN model checker with the protocol model is shown in figure 6.2.1.

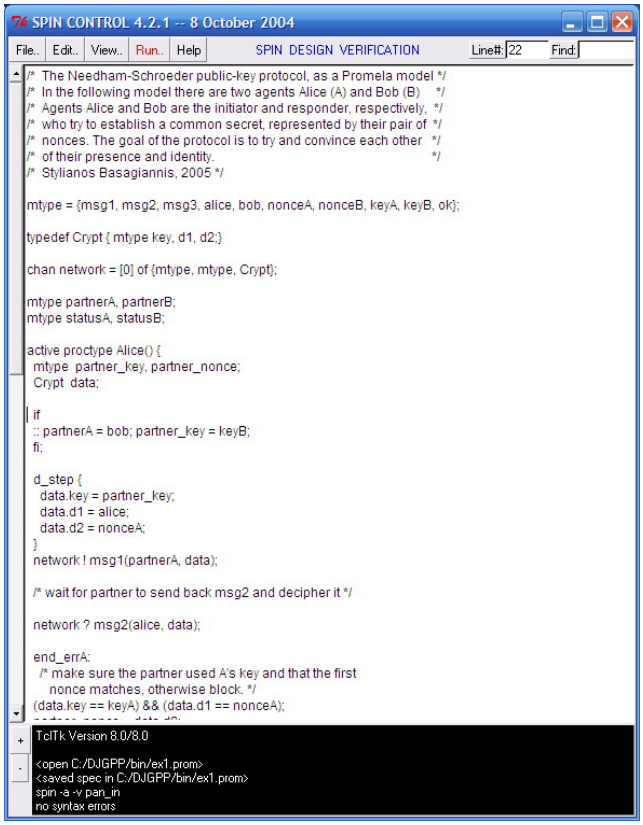

Figure 6.2.1: Protocol model with no syntax errors

If the message that the SPIN model checker returns, is “no syntax errors”, during the syntax check error procedure, we can move to configure the simulation options of our model. According to the principles that we have defined in chapter 4 (page 32), we have to run a simulation, which will show a successful communication between two agents that use the Needham Schroeder protocol. This simulation run can produce a detailed information of each execution step of our model, a time sequence panel for each of these steps, a data value panel showing a the values of each variable (global or local) that we use in our model and finally a message sequence chart that show us the messages being exchanged among the proctypes using channels that we have defined into our model.

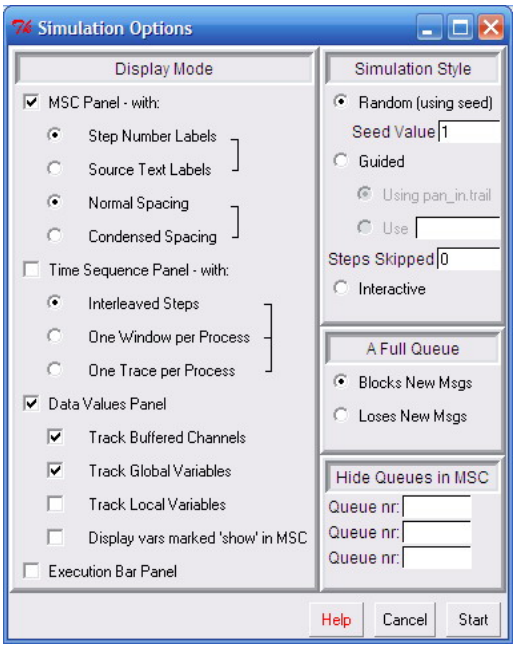

Figure 6.2.2: Simulation options (section 1)

At first we select to run a random simulation style using a seed value (seed value=1). We choose in the simulation options, as we can see from the figure 6.2.2, that for our simulation run we want from the SPIN model checker to produce a Message Sequence Chart (MSC) panel in order to observe the messages being exchanged using the defined channel "network". Also we select the Data values panel to be shown t in order to see in every step of the simulation run, the values that the global variables (statusA, statusB, partnerA, partnerB) have[17].

Starting the simulation, a simulation output is generated showing to us a detailed behavior of the agents Alice and Bob that in every step of the running process. In the summary report of the simulation output, we can see that SPIN model checker provides a summary in which our agents-proctypes have successfully been created and have completed a random run in a specified depth of the state space of our model. We can observe now the correction of the communication of the two agents in the Message Sequence Chart of the SPIN model checker in figure 6.2.3.

[17] We choose in the particular options, the default simulation options that the SPIN model checker has.

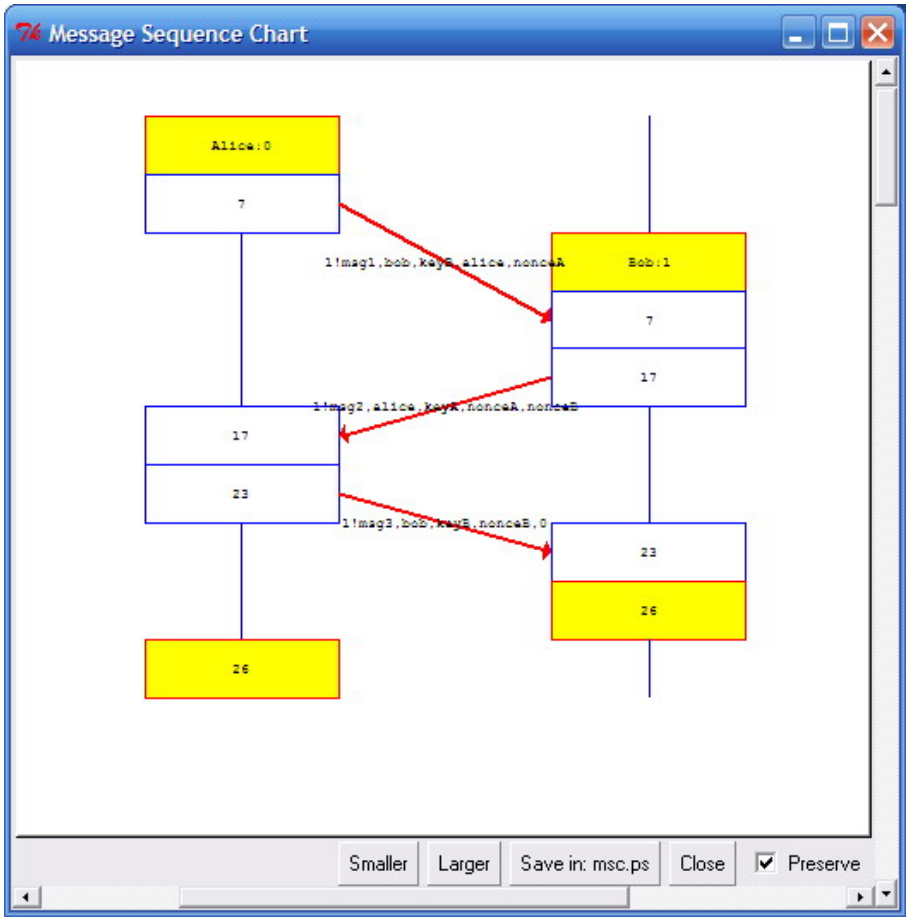

Figure 6.2.3: Message Sequence Chart of the protocol model

At first, as we would expected, initiator Alice begins communication at the simulation step 7, where she sends the first message to her chosen partner Bob. We can see that the first message is encrypted with Bob's public key and contains Alice's identity and the nonceA that Alice has generated. After the receiving of the message from Bob, and the checking that the agent-proctype has to do, he replies to Alice with the second message encrypted with Alice's public key, containing Alice's nonceA and his newly generated nonce, nonceB (step 17). Agent Alice is now the one who has to check about the msg2 that she has received. In step 23 Alice sends the third message to Bob, encrypted with Bob's public key, containing the nonceB that Bob has generated. We can see that also from the Data Values panel shown in figure 6.2.4 where statusA=OK. Bob has received the message and after the appropriate checking he declares that his status is OK at the step 26.

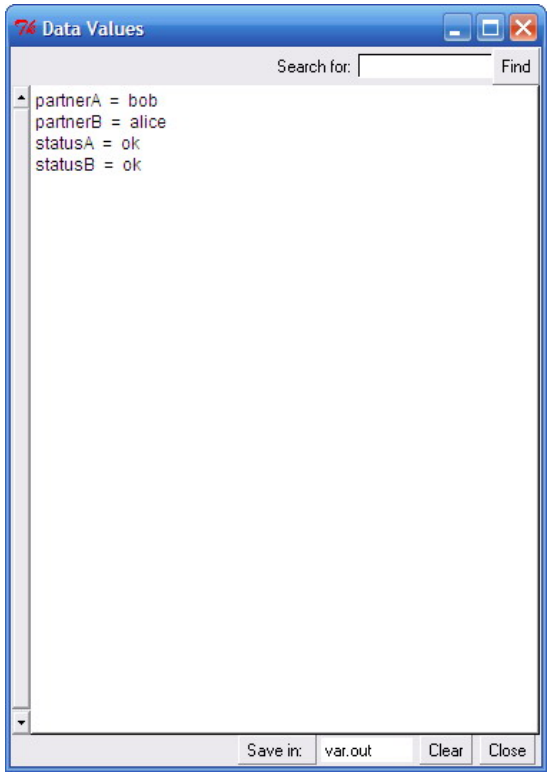

Figure 6.2.4: Data Values of the protocol model

As the simulation results have shown to us, the protocol model that we have implemented is functioning according to the principles of the Needham Schroeder protocol. The proof that there are no errors in our model is being produced from the verification results of the SPIN model checker, described in the next paragraph.

6.2.2 Verification results of the protocol model

Moving to the verification of the protocol model, in the same way as the simulation, we have to configure the verification options that the SPIN model checker has to agree on. At first we have to set the advanced verification options that have to do with the state space exploration of the model, the physical memory we set available for the search and the maximum depth that we want the search to reach. We choose a search depth at 10000 steps, as we want the SPIN model checker to find possible errors in the model using a sufficient search depth. The advanced verification options can be seen in the figure 6.2.5.

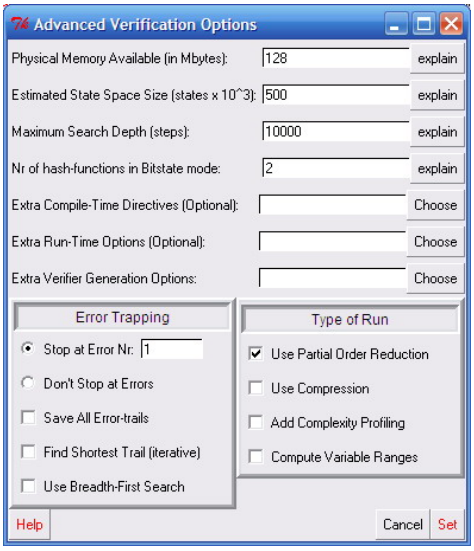

Figure 6.2.5: Advanced Verification options for the protocol model

Moving to the basic verification options, we choose for the correctness properties field for our model to be checked for Safety[18] state properties and especially for invalid end-states of the model, as we don't have defined assertions to our model. Also we want the tool to report any unreachable code that can be found in our model, during an exhaustive search that we also have selected to be performed from the Search mode field, as we can see from the figure 6.2.6.

Figure 6.2.6: Basic Verification Options for the protocol model

[18] Definition of Safety properties in formal analysis can be seen in Chapter 2

Proceeding to the verification, the SPIN model checker is going to produce us a verification output shown in the figure 6.2.7. As we can see from the results, SPIN has completed a full state space search (searching for invalid end-states), using a state vector which holds the value of all variables as well as program counters represented by 36bytes, reaching a 19 deepest stack depth and finding 0 errors in our model. As we can see from the figure 6.2.7, SPIN produces also information about the memory that has been spent in order to perform this search, among with the states that our model have already stored and matched in the seen set[19]. Finally in the verification output we can see that there are two unreached states of the nineteen (see figure 6.2.7) where for the agent-proctype Alice and three for the agent-proctype Bob. If we turn back to our simulation output and compare the output with our defined model, we can see that the error states have not been executed.

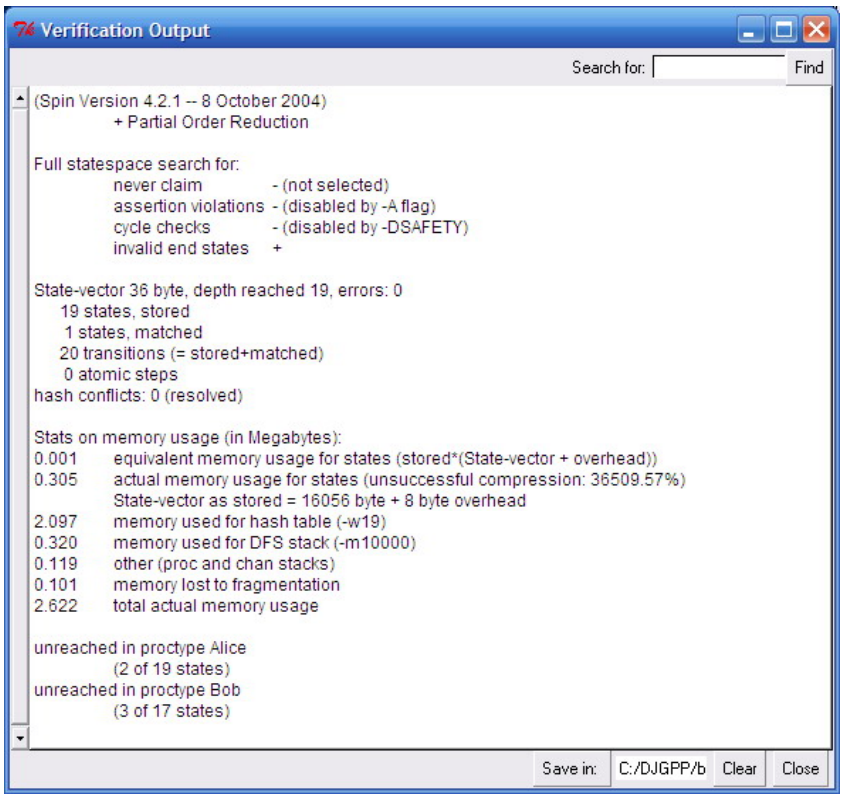

Figure 6.2.7: Verification Output of the protocol model

[19] Seen set is a set of arrays that the SPIN model checker creates in order to store states of the model using them to backtrack from a specific point of its search when a state exists in the seen set.

This means that the protocol model is correctly functioning, thus our implementation of the protocol model is correct. The reader can observe the complete verification results in the Appendix B.

After the verification report that the SPIN model checker has produced to us, that our section 1 model is correct and has no errors, we can move to the simulation and the verification of the intruder's model.

6.3 Formal Analysis of the intruder's attack model

Similarly with the protocol model, we separate the formal analysis of the intruder's model in the simulation results and the verification results.

6.3.1 Simulation results of the intruder's attack

The first step that we have in this model too, is the correct coding of the model that we have implemented. After the appropriate modifications, we insert our second section model and we run a syntax checking in order to eliminate all possible errors.

Figure 6.3.1: Intruder's attack model with no syntax errors

The intruder's attack model without syntax errors checker matches the one in figure 6.3.1.We must add that for the simulation and the verification results of the intruder's attack model, we follow the exact same procedure that we used previously for the protocol model. Providing that, we choose to run a simulation of the intruder's attack model, with the same simulation options. The Message Sequence Chart that the SPIN model checker has generated is the one in figure 6.3.2. We have also selected a random simulation run of our model, so we don't know which agent is going to start the communication (Alice or the Intruder). In our specified simulation run, the Intruder is the one who starts the communication by constructing a msg1 and send it back to Bob pretending to be his partner (for example Alice). If we change the seed value of the simulation options we can see different simulations runs of the model and so different attack points of the intruder to the protocol.

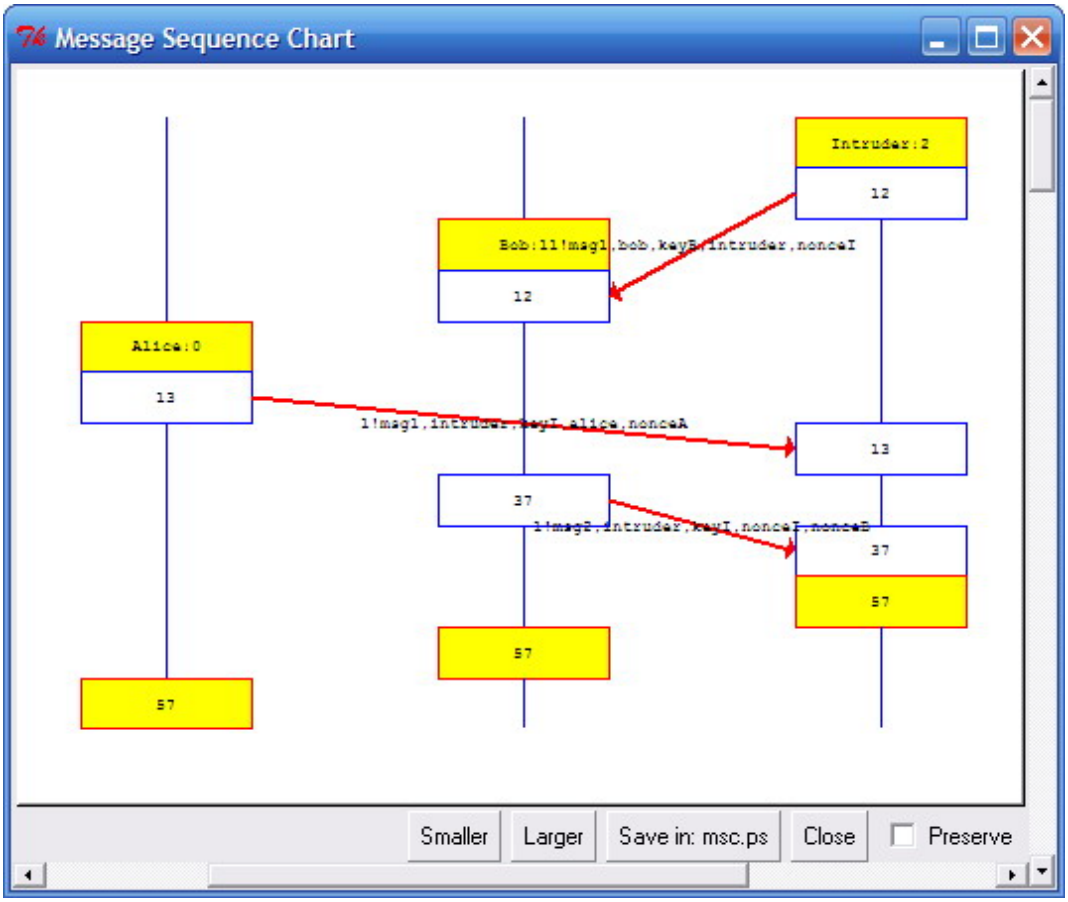

Figure 6.3.2: Message Sequence Chart of the intruder's attack model

As we can see from the message sequence chart given in figure 6.3.2, the agent-proctype intruder is the one that sends at first a msg1 to the agent Bob (step 12). This message contains the (fake) identity of the intruder and a nonce (nonceI) that the intruder has generated. Bob from his side receives the message from the person that wants to communicate with him. Currently he doesn't know that he is communicating with the intruder, and runs the usual checking procedures that he has. Alice, in step 13, sends from her side a msg1 also to the intruder containing her identity, the nonce that she has generated both encrypted with intruder's public key. This is the point where the intruder learns the nonceA that Alice has generated. In this step we can see from the data values panel shown in figure 6.3.3. that Alice has chosen randomly that her partner is the intruder (so she has to use his public key, keyI) and that her nonce is known (knowNA=1). After that, agent Bob has

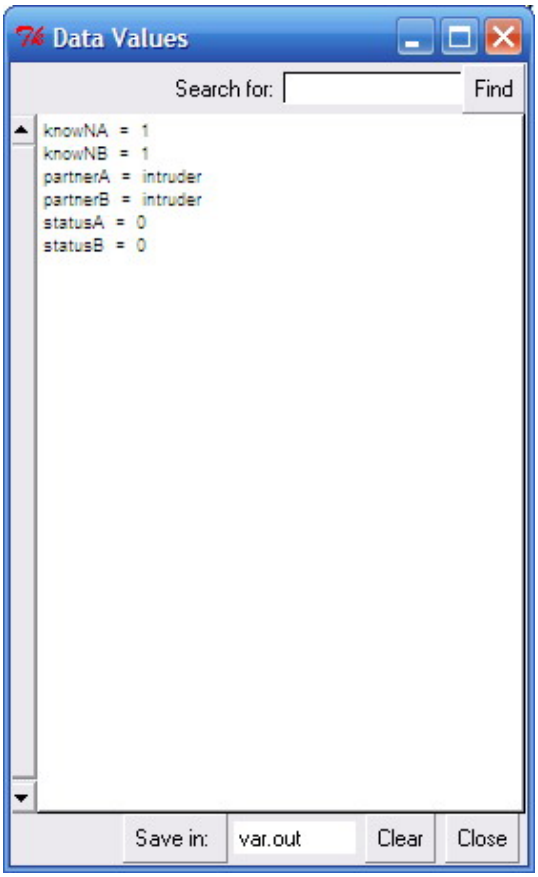

Figure 6.3.3: Data values of the intruder's attack model

found no errors in the msg1 that he had received from the intruder at the beginning of the run so he decides at step 37 to respond the intruder with a msg2 containing intruder's nonce (nonceI), his nonce (nonceB) that he has generated both encrypted with his partner public key which means with intruder's public key. This is the point where intruder gains the knowledge of Bob's nonce (knowNB=1) as we can see from the data values panel. Also from the same window that the SPIN model checker has produced, we can see that none of the agents Bob and Alice has declared success of their procedure, (statusA=0 and statusB=0) meaning that the intruder has successfully managed to interrupt the communication between them

.As the simulation results have shown to us, the intruder's attack model that we have implemented is functioning well as from a first look of the simulation output we can see that the intruder interrupts the communication in the Needham Schroeder protocol model. We should now proceed to the

verification results of our where the SPIN model checker is going to verify the error of the protocol model because of the existence of the intruder.

6.3.2 Verification results of the intruder's attack

Moving to the verification of the protocol model, in the same way as the simulation, we have to configure the verification options that the SPIN model checker has to agree on. We want our intruder's attack model to be checked for invalid end-states regarding the safety properties while we want report of the SPIN model checker of unreachable lines of code in our model. We keep the advanced verification options (memory and search depth specifications) unchanged from the verification of our first section model. Proceeding to the verification, the SPIN model checker produces a verification output shown in the figure 6.3.4.

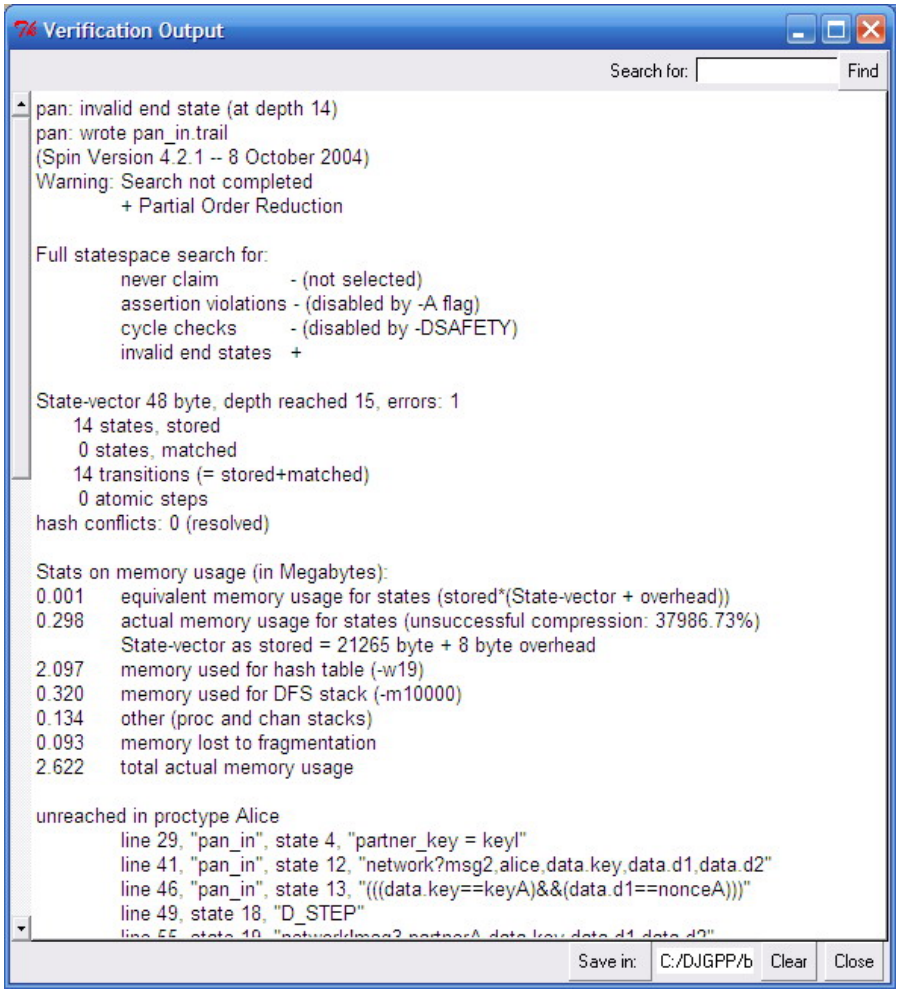

Figure 6.3.4: Verification output of the intruder's attack model

As we can see from the results, from the first lines of the verification output SPIN report us that it has found an invalid end-state error in the state 14 of the model. This verification search has been completed using a state vector represented by 48bytes, reaching a 15 deepest stack depth and finding 1 error in our model. We can also see that the states that SPIN model checker has stored in the seen set was only 14, meaning that in this specific state of the run of the search, SPIN has identified the invalid end-state error.

In our case, an invalid end-state error means that SPIN found an execution where our model cannot take another step, and at the point where it has stopped at least one of the processes is at a point that is not the end of its execution. We can see from the verification report the lines of code in our model that has not been executed because of the intruder's interference in the communication of the two agents. The reader can see the complete verification report in the Appendix B, ex1_withintruder.out. Through this verification output we can see the lines of the code (statements) for both of the agents-proctypes that didn't execute due to the intruder's attack. If we change the seed value in the simulation options, and run the simulation we can identify a number of intruder's attacks in the protocol model. But as we have modeled the protocol and the intruder, the most probable attack of him is the one that has been identified from the SPIN verification output. And that because we have implemented the intruder proctype in order that he gains the most knowledge from the protocol communication.

Finally we should state the way that the SPIN model checker proposes to run a guided simulation of the model, after the verification. Provided the error that we have found in the report, we can choose to run the guided simulation in order to find the solution corresponding to the invalid end-state. Doing so, SPIN model checker will stop the simulation at the specific line of code (that is executed) which is causing the invalid end-state. Having this information we

can go back to our model and correct our implementation of the model causing the elimination of this error.

6.4 Conclusion of the results

As we have previously noticed, simulation and verification results can not only show to us the exact execution behavior of our model and the presence of errors, but also we can get usable information about how we can correct our model, and therefore our system that we modeling in a correct way.

The first author that has discovered the intruder's attack on the Needham Schroeder protocol was Lowe in [28]. We have managed to prove this attack through the formal analysis of the protocol using the SPIN model checker as our results have shown.

Chapter 7

Conclusion – Future Work

7. Conclusion – Future Work

7.1 Conclusion

We have presented a way to model cryptographic protocols using PROMELA and a possible intruder's attack on it. The modelling approach we propose consists of specifying the protocol rules, principles and the configuration to be checked directly in PROMELA language, using the SPIN model checker, for the Needham Schroeder Public Key Authentication protocol. Due to the successful appliance that SPIN has towards communication protocol models we managed to apply successfully a formal analysis technique to the Needham Schroeder security protocol. In our case, SPIN model checker has simulated the exact behavior of the protocol (without finding an error on it), while it identified the intruder's attack that exists during the function of it, as Lowe in [28] has first discovered it.

The implementation procedure that we followed was the use of complexity reduction techniques for the simplest definition of the protocol model, and techniques for finding a possible intruder's attack, such as minimizing the number of agents who communicate using the protocol. Following this approach, we succeeded in modelling the correct function of the protocol while we have managed to prove the existence of intruder's attack that can occur.

7.2 Future Work

We have illustrated a procedure to construct a protocol model representing the Needham Schroeder public key authentication protocol and an intruder's attack model on it, using a case study. Future work includes a formal definition of the intruder construction procedure and its implementation in an

automatic intruder model generator which structure depends on the structure of the protocol being on attack.

Another point that we could focus on is implementing the complete Needham Schroeder protocol using all the seven steps (see chapter 2), provided the existence of a trusted session where the two agents don't know each other public keys and must communicate with the trusted agent in order to acquire them. This is an interesting situation, where the intruder has more attacking options provided this kind of communication.

References

[1] Audun J., "Security Protocol Verification using SPIN". Proceedings of the First SPIN Workshop, INRS-Telecommunications, Montreal, Canada, 1995.

[2] Heintze N., Tygar J.D., Wing J., Wong H.C., "Model Checking Electronic Commerce Protocols". Proceedings of the 2nd Usenix Workshop on Electronic Commerce, pages 147 – 164, 1996.

[3] Older S., Shiu-Kai Chin, "Formal methods for assuring security of protocols" Center for systems assurance, electrical engineering and computer science, Syracuse university, USA, 2000.

[4] Fanjul J.G, Tuya J., Corrales J.A., "Formal Verification and Simulation of the NetBill Protocol Using SPIN". Computer Science Department Campus de Viesques at Gijon ,University of Oviedo,2003.

[5] Holzmann G.J., "The SPIN Model Checker: Primer and Reference Manual" Addison-Wesley, ISBN 0-321-22862-6, 2003.

[6] Needham. R., Schroeder, M., "Using encryption for authentication in large networks of computers". Communications of ACM 21, pages 993-999, 1978.

[7] Holzmann G.J., "The Model Checker SPIN". IEEE Trans. on Software Engineering, Vol. 23, No. 5, pages 279-295, 1997.

[8] Buttyan L., "Formal methods in the design of cryptographic protocols". Swiss Federal Institute of Technology, ICA EPFL-Di-ICA, available at

http://icwww.epfl.ch/publications/documents/IC_TECH_REPORT_199938.pdf , 1999 [date accessed 15-5-2005].

[9] Collins M., "Formal Methods". Lectures in Carnegie Mellon University 18-849b Dependable Embedded Systems, available at http://www.ece.cmu.edu/~koopman/des_s99/formal_methods/ , 1998 [date accessed 16-5-2005].

[10] Black P. E., Hall K. M., Jones M. D., Larson T. N., Windley P. J., "A Brief Introduction to Formal Methods". Proceedings of the IEEE 1996, Custom Integrated Circuits Conference (CICC '96), San Diego, California, USA, pages 377-380, 1996.

[11] ISO. IS7498-2, [ISO88] "Basic Reference Model for Open Systems Interconnection, Part 2: Security Architecture". International Organisation for Standardisation, 1988.

[12] Desmedt Y., Burmester M., Millen J., "Design vs. Verification: Is Verification the Wrong Approach?" . Proceedings of the 1997 DIMACS Workshop on Design and Formal Verification of Security Protocols, available at http://dimacs.rutgers.edu/Workshops/Security/program2/yvo-panel.ps, 1997 [date accessed 17-5-2005].

[13] Clarke E. M., Grumberg O., Peled D. A., "Model Checking". MIT Press, ISBN 0-262-03270-8, 1999.

[14] Pandya P., "Formal specification and verification of security protocols". Dresden University of Technology, available at http://www.piware.de/docs/securityprotocols.pdf, 2002 [date accessed 20-5-2005].

[15] Lamport L., "The Temporal Logic of Actions". ACM Transactions on Programming Languages and Systems, vol. 16(3), pages 872-923, 1994.

[16] Manna Z., Pnueli A., "Temporal Verification of Reactive Systems - Safety: Springer- Verlag", 1995.

[17] Fernandez J.C., Garavel H., Kerbrat A., Mateesku R., Mounier L., Sighireanu M., "CADP: A Protocol Validation and Verification Toolbox". Proceedings of the 8th International Conference on Computer-Aided Verification (CAV'96), New Brunswick, NJ, USA, 1996. From Lecture Notes in Computer Science, pages 437-440, R. Alur and T. A. Henzinger, Eds., 2005.

[18] CCITT'93, "SDL - Specification and Description Language". CCITT Z.100, International Consultative Committee on Telegraphy and Telephony, 1993.

[19] Korver H., "Detecting Feature Interactions with CAESAR/ALDEBARAN". Science of Computer Programming, special issue on Industrially Relevant Applications of Formal Analysis Techniques, 1997.

[20] Cleaveland R., Parrow J., Steffen B., "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems". ACM Transactions on Programming Languages and Systems, vol. 15(1), pages 36-72, 1993.

[21] Milner R.," Communication and Concurrency". Prentice-Hall, ISBN 0-13-115007-3, 1989.

[22] Hardin R. H., Har'El Z., Kurshan R. P., "COSPAN,". Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96), USA, 1996. Lecture Notes in Computer Science 1102, pages 423-427, R. Alur , T. A. Henzinger, Eds, 2005.

[23] Bouali A., Ressouche A., Roy V., Simone R. d., "The FC2TOOLS Set". Proceedings of the 8th International Conference on Computer-Aided Verification (CAV'96), USA, 1996. From Lecture Notes in Computer Science, pages 441-445, R. Alur and T. A. Henzinger, Eds, 2005.

[24] Roscoe A.W., "Model-checking CSP in A Classical Mind". Essays in Honour of C.A.R. Hoare, Prentice Hall International Series in Computer Science, Ed.: Prentice- Hall, pages 353-378, 1994.

[25] McMillan K.L. ,"Symbolic Model Checking". Kluwer Academic Publishers, 1993.

[26] Ireland A., "SPIN: Formal Analysis I". From Lecture notes No 6,. School of Mathematics and Computer Science, Heriot – Watt University, UK, 2005.

[27] SPIN model checker official website, available at http://spinroot.com/spin/whatispin.html, [date accessed 21-5-2005].

[28] Lowe G., "An attack on the Needham Schroeder public key authentication protocol". Information Processing letters, available ar http://web.comlab.ox.ac.uk/oucl/work/gavin.lowe/Security/Papers/NSPKP.ps, 1995 [date accessed 23-5-2005].

[29] Magee J., Kramer J., "Concurrency: State models & Java Program" vailable at ftp://ftp.dit.upm.es/pub/lotos/papers/tutorial/ /lotos_language_tutorial.ps [date accessed 24-5-2005].

[30] Magee J., Lectures in Computer science, distributed software engineering section, department of computing, Imperial college, UK, Definition of LTSA, available at **http://www.doc.ic.ac.uk/~jnm/book/ltsa/LTSA.html**, [date accessed 26-5-2005].

[31] Definition of verilog, available at http:/www.eg3.com/eCLIPS/desc/soc_verilog_blank.html [date accessed 25-5-2005].

[32] "Online VHDL Language Guide", available at http://www.acc-eda.com/vhdlref/ [date accessed 25-5-2005].

[33] UNUN IIST, International Institute for Software Technology available at http://www.iist.unu.edu/~alumni/software/other/inria/ www/fc2tools/quick-guide.html#Section4 [date accessed 26-5-2005].

[34] Verma K. J, "Verification of Cryptographic Protocols, L3". Institute for Informatics, TU , Munich, 2004.

Appendix A

Source code (PROMELA)

Appendix A: Source code (PROMELA)

This is the appendix where the source code of our models is presented.

Source code of the protocol model, part 1

The first part has to do with the implementation of the Needham Schroeder protocol. We represent the basic function of the protocol between two agents; one the initiator A (Alice) and the other the responder B (Bob).

```
(ex1.prom)

/*  The Needham-Schroeder public-key protocol, as a Promela model*/
/*  In the following model section 1, there are two agents Alice (A)
/*  and Bob (B)*/
/*  Agents Alice and Bob are the initiator and responder,         */
/*  respectively,*/
/*  who try to establish a common secret, represented by their pair*/
/*  of*/
/*  nonces. The goal of the protocol is to try and convince each   */
/*  other*/
/*  of their presence and identity.                               */
/*  Stylianos Basagiannis, 2005                                   */

mtype = {msg1, msg2, msg3, alice, bob, nonceA, nonceB, keyA, keyB,
ok};

typedef Crypt { mtype key, d1, d2;}

chan network = [0] of {mtype, mtype, Crypt};

mtype partnerA, partnerB;
mtype statusA, statusB;

active proctype Alice() {
  mtype  partner_key, partner_nonce;
  Crypt  data;
```

```
  if
  :: partnerA = bob; partner_key = keyB;
  fi;

  d_step {
    data.key = partner_key;
    data.d1 = alice;
    data.d2 = nonceA;
  }
  network ! msg1(partnerA, data);

  /* wait for partner to send back msg2 and decipher it */

  network ? msg2(alice, data);

  end_errA:
    /* make sure the partner used A's key and that the first
       nonce matches, otherwise block. */
  (data.key == keyA) && (data.d1 == nonceA);
  partner_nonce = data.d2;

  d_step {
    /* respond with msg3 and declare success */
    data.key = partner_key;
    data.d1 = partner_nonce;
    data.d2 = 0;
  }
  network ! msg3(partnerA, data);
  statusA = ok;
}


active proctype Bob() {
  mtype partner_key, partner_nonce;
  Crypt data;

  /* wait for msg1, identifying partner */
  network ? msg1(bob, data);

  /* try to decipher the message; block on wrong key */
```

```
  end_errB1:
  (data.key == keyB);
  partnerB = data.d1;

  d_step {
    partner_nonce = data.d2;
    /* lookup the partner's public key */
    if
    :: (partnerB == alice) -> partner_key = keyA;
    fi;
    /* respond with msg2 */
    data.key = partner_key;
    data.d1 = partner_nonce;
    data.d2 = nonceB;
  }
  network ! msg2(partnerB, data);

  /* wait for msg3, check the key and the nonce, and declare success
*/
  network ? msg3(bob, data);
  end_errB2:
  (data.key == keyB) && (data.d1 == nonceB);
  statusB = ok;
}
```

Source code of the intruder's attack model, part 2

The second part has to do with the implementation of the attack of an intruder to the Needham Schroeder protocol. We represent the basic function of the protocol between two agents; one the initiator A (Alice) and the other the responder B (Bob) and we implement an intruder (I) that is trying to gain knowledge from the communication between the two agents, sending intercepted or fake messages to both of the agents, causing a security error to the protocol.

```
(ex1with_intruder.prom)
```

```
/*  The Needham-Schroeder public-key protocol (section2), as a    /*
/*  Promela model       */
/*  In the following model there are two agents Alice (A) and Bob (B)
/*  and the Intruder (I)      */
/*  Agents Alice and Bob are the initiator and responder,         /*
/*  respectively,                   */
/*   who try to establish a common secret, represented by their  /*
/*    pair of                   */
/*  nonces. The goal of the protocol is to try and convince each */
/*  other                       */
/*   of their presence and identity.The intruder is the process */
/*    that will try to gain knowledge      */
/*   from the other two agents during their communication. If the */
/*    intruder (I) accomplish that   */
/*   then the SPIN model checker will indicate an error (security */
/*    hole) to our model.            */
/*   This section 2 model is an extension of the section1 model. */
/*  Proctypes Alice and Bob are     */
/*  slightly changed and the proctype intruder I is been added at */
/*  the end of the model            */
/*  Stylianos Basagiannis, 2005     */

mtype = {msg1, msg2, msg3, alice, bob, nonceA, nonceB, keyA, keyB,
ok, intruder,nonceI, keyI};

typedef Crypt { mtype key, d1, d2;}

chan network = [0] of {mtype, mtype, Crypt};

mtype partnerA, partnerB;
mtype statusA, statusB;
bool knowNA,knowNB;

active proctype Alice() {
  mtype  partner_key, partner_nonce;
  Crypt  data;

  if
  :: partnerA = bob; partner_key = keyB;
  :: partnerA = intruder; partner_key = keyI;
  fi;
```

```
  d_step {
    data.key = partner_key;
    data.d1 = alice;
    data.d2 = nonceA;
  }
  network ! msg1(partnerA, data);

  /* wait for partner to send back msg2 and decipher it */

  network ? msg2(alice, data);

  end_errA:
    /* make sure the partner used A's key and that the first
       nonce matches, otherwise block. */
  (data.key == keyA) && (data.d1 == nonceA);
  partner_nonce = data.d2;

  d_step {
    /* respond with msg3 and declare success */
    data.key = partner_key;
    data.d1 = partner_nonce;
    data.d2 = 0;
  }
  network ! msg3(partnerA, data);
  statusA = ok;
}

active proctype Bob() {
  mtype partner_key, partner_nonce;
  Crypt data;

  /* wait for msg1, identifying partner */
  network ? msg1(bob, data);

  /* try to decipher the message; block on wrong key */
  end_errB1:
  (data.key == keyB);
  partnerB = data.d1;

  d_step {
```

```
    partner_nonce = data.d2;
    /* lookup the partner's public key */
    if
    :: (partnerB == alice) -> partner_key = keyA;
    :: (partnerB == intruder) -> partner_key = keyI;
    fi;
    /* respond with msg2 */
    data.key = partner_key;
    data.d1 = partner_nonce;
    data.d2 = nonceB;
  }
  network ! msg2(partnerB, data);

  /* wait for msg3, check the key and the nonce, and declare success
*/
  network ? msg3(bob, data);
  end_errB2:
  (data.key == keyB) && (data.d1 == nonceB);
  statusB = ok;
}

active proctype Intruder() {
  /*we want the modelchecker to find the attack, so we list different
     actions it can perform.
  */
  mtype msg;
  Crypt data, intercepted;
  mtype icp_type;   /* type of intercepted message */

  do
  :: /* Send msg1 to B (sending it to anybody else would be foolish).
        May use own identity or pretend to be A; send some nonce
known to I.
     */
     if /* either replay intercepted message or construct a fresh
message */
     :: icp_type == msg1 -> network ! msg1(bob, intercepted);
     :: data.key = keyB;
      if
      :: data.d1 = alice;
      :: data.d1 = intruder;
```

```
      fi;
      if
      :: knowNA -> data.d2 = nonceA;
      :: knowNB -> data.d2 = nonceB;
      :: data.d2 = nonceI;
      fi;
        network ! msg1(bob, data);
     fi;
  :: /* Send msg2 to A. */
     if
     :: icp_type == msg2 -> network ! msg2(alice, intercepted);
     :: data.key = keyA;
        if
      :: knowNA -> data.d1 = nonceA;
      :: knowNB -> data.d1 = nonceB;
      :: data.d1 = nonceI;
      fi;
      if
      :: knowNA -> data.d2 = nonceA;
      :: knowNB -> data.d2 = nonceB;
      :: data.d2 = nonceI;
      fi;
        network ! msg2(alice, data);
     fi;
  :: /* Send msg3 to B. */
     if
     :: icp_type == msg2 -> network ! msg3(bob, intercepted);
     :: data.key = keyB;
      if
      :: knowNA -> data.d1 = nonceA;
      :: knowNB -> data.d1 = nonceB;
      :: data.d1 = nonceI;
      fi;
        data.d2 = 0;
        network ! msg3(bob, data);
     fi;
  :: /* Receive or intercept a message from A or B. If possible,
extract nonces. */
     network ? msg (_, data) ->
     if /* Perhaps store the data field for later use */
     :: d_step {
```

```
        intercepted.key = data.key;
        intercepted.d1 = data.d1;
        intercepted.d2 = data.d2;
          icp_type = msg;
      }
     :: skip;
     fi;
     d_step {
      if /* Try to decrypt the message if possible */
      :: (data.key == keyI) -> /* Have we learnt a new nonce? */
         if
         :: ((data.d1 == nonceA) || (data.d2 == nonceA)) -> knowNA =
true;
           :: else -> skip;
         fi;
         if
         :: (data.d1 == nonceB || data.d2 == nonceB) -> knowNB =
true;
         :: else -> skip;
         fi;
      :: else -> skip;
      fi;
     }
  od;
}
```

Appendix B

Results

Appendix B: Results

In this appendix B we are going to represent the results of both parts 1 and 2 of our models that the SPIN model checker is produced to us. The results are at first the simulation of the models, the data values of the variables that is been used (with their final value at the end of the simulation run) and the message sequence chart (MSC) of the model. After the simulation results, the verification results are also going to be presented.

Simulation Results (protocol model, part 1)

Simulation results of the protocol model. Below are listed the simulation output and the data values panel.

```
(Simulation output, sim1.out)

Starting Alice with pid 0
  0:  proc  - (:root:) creates proc  0 (Alice)
Starting Bob with pid 1
  0:  proc  - (:root:) creates proc  1 (Bob)
spin: warning, "pan_in", global, 'mtype statusA' variable is never
used
spin: warning, "pan_in", global, 'mtype statusB' variable is never
used
  1:  proc  0 (Alice) line  22 "pan_in" (state 3)     [partnerA =
bob]
  2:  proc  0 (Alice) line  23 "pan_in" (state 2)     [partner_key =
keyB]
  3:  proc  0 (Alice) line  26 "pan_in" (state 4)     [.(goto)]
  4:  proc  0 (Alice) line  26 "pan_in" (state 8)     [data.key =
partner_key]
  5:  proc  0 (Alice) line  28 "pan_in" (state 6)     [data.d1 =
alice]
  6:  proc  0 (Alice) line  29 "pan_in" (state 7)     [data.d2 =
nonceA]
  7:  proc  0 (Alice) line  31 "pan_in" (state 9)
      [network!msg1,partnerA,data.key,data.d1,data.d2]
  7:  proc  1 (Bob) line  59 "pan_in" (state 1)
      [network?msg1,bob,data.key,data.d1,data.d2]
  7:  proc  0 (Alice) line  31 "pan_in" (state -)     [values:
1!msg1,bob,keyB,alice,nonceA]
  7:  proc  1 (Bob) line  59 "pan_in" (state -) [values:
1?msg1,bob,keyB,alice,nonceA]
  8:  proc  1 (Bob) line  63 "pan_in" (state 2) [((data.key==keyB))]
  9:  proc  1 (Bob) line  64 "pan_in" (state 3) [partnerB = data.d1]
```

```
 10:  proc  1 (Bob) line  66 "pan_in" (state 12)      [partner_nonce
= data.d2]
 11:  proc  1 (Bob) line  69 "pan_in" (state 7) [((partnerB==alice))]
 12:  proc  1 (Bob) line  70 "pan_in" (state 6) [partner_key = keyA]
 13:  proc  1 (Bob) line  73 "pan_in" (state 8) [.(goto)]
 14:  proc  1 (Bob) line  73 "pan_in" (state 9) [data.key =
partner_key]
 15:  proc  1 (Bob) line  74 "pan_in" (state 10)      [data.d1 =
partner_nonce]
 16:  proc  1 (Bob) line  75 "pan_in" (state 11)      [data.d2 =
nonceB]
 17:  proc  1 (Bob) line  77 "pan_in" (state 13)
      [network!msg2,partnerB,data.key,data.d1,data.d2]
 17:  proc  0 (Alice) line  35 "pan_in" (state 10)
      [network?msg2,alice,data.key,data.d1,data.d2]
 17:  proc  1 (Bob) line  77 "pan_in" (state -) [values:
1!msg2,alice,keyA,nonceA,nonceB]
 17:  proc  0 (Alice) line  35 "pan_in" (state -)     [values:
1?msg2,alice,keyA,nonceA,nonceB]
 18:  proc  0 (Alice) line  40 "pan_in" (state 11)
      [(((data.key==keyA)&&(data.d1==nonceA)))]
 19:  proc  0 (Alice) line  41 "pan_in" (state 12)    [partner_nonce
= data.d2]
 20:  proc  0 (Alice) line  43 "pan_in" (state 16)    [data.key =
partner_key]
 21:  proc  0 (Alice) line  46 "pan_in" (state 14)    [data.d1 =
partner_nonce]
 22:  proc  0 (Alice) line  47 "pan_in" (state 15)    [data.d2 = 0]
 23:  proc  0 (Alice) line  49 "pan_in" (state 17)
      [network!msg3,partnerA,data.key,data.d1,data.d2]
 23:  proc  1 (Bob) line  80 "pan_in" (state 14)
      [network?msg3,bob,data.key,data.d1,data.d2]
 23:  proc  0 (Alice) line  49 "pan_in" (state -)     [values:
1!msg3,bob,keyB,nonceB,0]
 23:  proc  1 (Bob) line  80 "pan_in" (state -) [values:
1?msg3,bob,keyB,nonceB,0]
 24:  proc  0 (Alice) line  50 "pan_in" (state 18)    [statusA = ok]
 25:  proc  1 (Bob) line  82 "pan_in" (state 15)
      [(((data.key==keyB)&&(data.d1==nonceB)))]
 26:  proc  1 (Bob) line  83 "pan_in" (state 16)      [statusB = ok]
 26:  proc  1 (Bob) terminates
 26:  proc  0 (Alice) terminates
2 processes created


(Data values, var1.out)

partnerA  =  bob
partnerB  =  alice
statusA  =  ok
statusB  =  ok
```

Verification Results (protocol model, part 1)

Verification results of the protocol model. Below the verification output is listed.

```
(Verification Output, Var1.out)
(Spin Version 4.2.1 -- 8 October 2004)
      + Partial Order Reduction

Full statespace search for:
      never claim             - (not selected)
      assertion violations    - (disabled by -A flag)
      cycle checks            - (disabled by -DSAFETY)
      invalid end states      +

State-vector 36 byte, depth reached 19, errors: 0
      19 states, stored
       1 states, matched
      20 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.001       equivalent memory usage for states (stored*(State-vector
+ overhead))
0.305       actual memory usage for states (unsuccessful compression:
36509.57%)
      State-vector as stored = 16056 byte + 8 byte overhead
2.097       memory used for hash table (-w19)
0.320       memory used for DFS stack (-m10000)
0.119       other (proc and chan stacks)
0.101       memory lost to fragmentation
2.622       total actual memory usage

unreached in proctype Alice
      (2 of 19 states)
unreached in proctype Bob
      (3 of 17 states)
```

Simulation Results (intruder's attack model, part 2)

Simulation results of the intruder's attack model. Below are listed the simulation output and the data values panel.

```
(Simulation Output, sim2.out)

Starting Alice with pid 0
  0:  proc  - (:root:) creates proc  0 (Alice)
Starting Bob with pid 1
  0:  proc  - (:root:) creates proc  1 (Bob)
Starting Intruder with pid 2
  0:  proc  - (:root:) creates proc  2 (Intruder)
spin: warning, "pan_in", global, 'mtype statusA' variable is never
used
spin: warning, "pan_in", global, 'mtype statusB' variable is never
used
  1:  proc  0 (Alice) line  27 "pan_in" (state 5)     [partnerA =
intruder]
  2:  proc  0 (Alice) line  29 "pan_in" (state 4)     [partner_key =
keyI]
  3:  proc  2 (Intruder) line 104 "pan_in" (state 79) [data.key =
keyB]
  4:  proc  2 (Intruder) line 111 "pan_in" (state 6)  [data.d1 =
intruder]
  5:  proc  2 (Intruder) line 115 "pan_in" (state 7)  [.(goto)]
  6:  proc  0 (Alice) line  32 "pan_in" (state 6)     [.(goto)]
  7:  proc  0 (Alice) line  32 "pan_in" (state 10)    [data.key =
partner_key]
  8:  proc  0 (Alice) line  34 "pan_in" (state 8)     [data.d1 =
alice]
  9:  proc  0 (Alice) line  35 "pan_in" (state 9)     [data.d2 =
nonceA]
 10:  proc  2 (Intruder) line 115 "pan_in" (state 13) [data.d2 =
nonceI]
 11:  proc  2 (Intruder) line 120 "pan_in" (state 14) [.(goto)]
 12:  proc  2 (Intruder) line 120 "pan_in" (state 15)
      [network!msg1,bob,data.key,data.d1,data.d2]
 12:  proc  1 (Bob) line  66 "pan_in" (state 1)
      [network?msg1,bob,data.key,data.d1,data.d2]
 12:  proc  2 (Intruder) line 120 "pan_in" (state -)  [values:
1!msg1,bob,keyB,intruder,nonceI]
 12:  proc  1 (Bob) line  66 "pan_in" (state -) [values:
1?msg1,bob,keyB,intruder,nonceI]
 13:  proc  0 (Alice) line  37 "pan_in" (state 11)
      [network!msg1,partnerA,data.key,data.d1,data.d2]
 13:  proc  2 (Intruder) line 151 "pan_in" (state 52)
      [network?msg,_,data.key,data.d1,data.d2]
 13:  proc  0 (Alice) line  37 "pan_in" (state -)     [values:
1!msg1,intruder,keyI,alice,nonceA]
 13:  proc  2 (Intruder) line 151 "pan_in" (state -)  [values:
1?msg1,0,keyI,alice,nonceA]
 14:  proc  1 (Bob) line  70 "pan_in" (state 2) [((data.key==keyB))]
 15:  proc  2 (Intruder) line 152 "pan_in" (state 59)
      [intercepted.key = data.key]
 16:  proc  2 (Intruder) line 155 "pan_in" (state 54) [intercepted.d1
= data.d1]
 17:  proc  2 (Intruder) line 156 "pan_in" (state 55) [intercepted.d2
= data.d2]
 18:  proc  2 (Intruder) line 157 "pan_in" (state 56) [icp_type =
msg]
 19:  proc  1 (Bob) line  71 "pan_in" (state 3) [partnerB = data.d1]
 20:  proc  1 (Bob) line  73 "pan_in" (state 14)      [partner_nonce
= data.d2]
```

```
 21:  proc  1 (Bob) line  76 "pan_in" (state 9)
      [((partnerB==intruder))]
 22:  proc  1 (Bob) line  78 "pan_in" (state 8) [partner_key = keyI]
 23:  proc  1 (Bob) line  81 "pan_in" (state 10)      [.(goto)]
 24:  proc  1 (Bob) line  81 "pan_in" (state 11)      [data.key =
partner_key]
 25:  proc  1 (Bob) line  82 "pan_in" (state 12)      [data.d1 =
partner_nonce]
 26:  proc  1 (Bob) line  83 "pan_in" (state 13)      [data.d2 =
nonceB]
 27:  proc  2 (Intruder) line 161 "pan_in" (state 60) [.(goto)]
 28:  proc  2 (Intruder) line 161 "pan_in" (state 78)
      [((data.key==keyI))]
 29:  proc  2 (Intruder) line 164 "pan_in" (state 66)
      [(((data.d1==nonceA)||(data.d2==nonceA)))]
 30:  proc  2 (Intruder) line 165 "pan_in" (state 63) [knowNA = 1]
 31:  proc  2 (Intruder) line 168 "pan_in" (state 67) [.(goto)]
 32:  proc  2 (Intruder) line 168 "pan_in" (state 72) [else]
 33:  proc  2 (Intruder) line 170 "pan_in" (state 71) [(1)]
 34:  proc  2 (Intruder) line 172 "pan_in" (state 73) [.(goto)]
 35:  proc  2 (Intruder) line 174 "pan_in" (state 77) [.(goto)]
 36:  proc  2 (Intruder) line 176 "pan_in" (state 80) [.(goto)]
 37:  proc  1 (Bob) line  85 "pan_in" (state 15)
      [network!msg2,partnerB,data.key,data.d1,data.d2]
 37:  proc  2 (Intruder) line 151 "pan_in" (state 52)
      [network?msg,_,data.key,data.d1,data.d2]
 37:  proc  1 (Bob) line  85 "pan_in" (state -) [values:
1!msg2,intruder,keyI,nonceI,nonceB]
 37:  proc  2 (Intruder) line 151 "pan_in" (state -)  [values:
1?msg2,0,keyI,nonceI,nonceB]
 38:  proc  2 (Intruder) line 152 "pan_in" (state 59)
      [intercepted.key = data.key]
 39:  proc  2 (Intruder) line 155 "pan_in" (state 54) [intercepted.d1
= data.d1]
 40:  proc  2 (Intruder) line 156 "pan_in" (state 55) [intercepted.d2
= data.d2]
 41:  proc  2 (Intruder) line 157 "pan_in" (state 56) [icp_type =
msg]
 42:  proc  2 (Intruder) line 161 "pan_in" (state 60) [.(goto)]
 43:  proc  2 (Intruder) line 161 "pan_in" (state 78)
      [((data.key==keyI))]
 44:  proc  2 (Intruder) line 164 "pan_in" (state 66) [else]
 45:  proc  2 (Intruder) line 166 "pan_in" (state 65) [(1)]
 46:  proc  2 (Intruder) line 168 "pan_in" (state 67) [.(goto)]
 47:  proc  2 (Intruder) line 168 "pan_in" (state 72)
      [(((data.d1==nonceB)||(data.d2==nonceB)))]
 48:  proc  2 (Intruder) line 169 "pan_in" (state 69) [knowNB = 1]
 49:  proc  2 (Intruder) line 172 "pan_in" (state 73) [.(goto)]
 50:  proc  2 (Intruder) line 174 "pan_in" (state 77) [.(goto)]
 51:  proc  2 (Intruder) line 176 "pan_in" (state 80) [.(goto)]
 52:  proc  2 (Intruder) line 104 "pan_in" (state 79) [data.key =
keyB]
 53:  proc  2 (Intruder) line 111 "pan_in" (state 6)  [data.d1 =
intruder]
 54:  proc  2 (Intruder) line 115 "pan_in" (state 7)  [.(goto)]
 55:  proc  2 (Intruder) line 115 "pan_in" (state 13) [(knowNA)]
 56:  proc  2 (Intruder) line 116 "pan_in" (state 9)  [data.d2 =
nonceA]
 57:  proc  2 (Intruder) line 120 "pan_in" (state 14) [.(goto)]
timeout
#processes: 3
```

```
 57:  proc  2 (Intruder) line 120 "pan_in" (state 15)
 57:  proc  1 (Bob) line  88 "pan_in" (state 16)
 57:  proc  0 (Alice) line  41 "pan_in" (state 12)
3 processes created
```

(Data Values, var2.out)

```
knowNA  =  1
knowNB  =  1
partnerA  =  intruder
partnerB  =  intruder
statusA  =  0
statusB  =  0
```

Verification Results (intruder's attack model, part 2)

Verification results of the intruder's attack model. Below the verification output is listed.

(Verification Output, ex1with_intruder.out)

```
pan: invalid end state (at depth 14)
pan: wrote pan_in.trail
(Spin Version 4.2.1 -- 8 October 2004)
Warning: Search not completed
      + Partial Order Reduction

Full statespace search for:
      never claim             - (not selected)
      assertion violations    - (disabled by -A flag)
      cycle checks            - (disabled by -DSAFETY)
      invalid end states      +

State-vector 48 byte, depth reached 15, errors: 1
      14 states, stored
       0 states, matched
      14 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.001       equivalent memory usage for states (stored*(State-vector
+ overhead))
0.298       actual memory usage for states (unsuccessful compression:
37986.73%)
      State-vector as stored = 21265 byte + 8 byte overhead
2.097       memory used for hash table (-w19)
0.320       memory used for DFS stack (-m10000)
0.134       other (proc and chan stacks)
0.093       memory lost to fragmentation
2.622       total actual memory usage

unreached in proctype Alice
```

```
      line 29, "pan_in", state 4, "partner_key = keyI"
      line 41, "pan_in", state 12,
"network?msg2,alice,data.key,data.d1,data.d2"
      line 46, "pan_in", state 13,
"(((data.key==keyA)&&(data.d1==nonceA)))"
      line 49, state 18, "D_STEP"
      line 55, state 19,
"network!msg3,partnerA,data.key,data.d1,data.d2"
      line 56, state 20, "statusA = ok"
      line 57, state 21, "-end-"
      (9 of 21 states)
unreached in proctype Bob
      line 88, "pan_in", state 16,
"network?msg3,bob,data.key,data.d1,data.d2"
      line 90, "pan_in", state 17,
"(((data.key==keyB)&&(data.d1==nonceB)))"
      line 91, "pan_in", state 18, "statusB = ok"
      line 92, "pan_in", state 19, "-end-"
      (7 of 19 states)
unreached in proctype Intruder
      line 109, "pan_in", state 2,
"network!msg1,bob,intercepted.key,intercepted.d1,intercepted.d2"
      line 111, "pan_in", state 6, "data.d1 = alice"
      line 111, "pan_in", state 6, "data.d1 = intruder"
      line 116, "pan_in", state 9, "data.d2 = nonceA"
      line 117, "pan_in", state 11, "data.d2 = nonceB"
      line 124, "pan_in", state 19,
"network!msg2,alice,intercepted.key,intercepted.d1,intercepted.d2"
      line 127, "pan_in", state 22, "data.d1 = nonceA"
      line 128, "pan_in", state 24, "data.d1 = nonceB"
      line 126, "pan_in", state 26, "(knowNA)"
      line 126, "pan_in", state 26, "(knowNB)"
      line 126, "pan_in", state 26, "data.d1 = nonceI"
      line 132, "pan_in", state 29, "data.d2 = nonceA"
      line 133, "pan_in", state 31, "data.d2 = nonceB"
      line 131, "pan_in", state 33, "(knowNA)"
      line 131, "pan_in", state 33, "(knowNB)"
      line 131, "pan_in", state 33, "data.d2 = nonceI"
      line 136, "pan_in", state 35,
"network!msg2,alice,data.key,data.d1,data.d2"
      line 140, "pan_in", state 39,
"network!msg3,bob,intercepted.key,intercepted.d1,intercepted.d2"
      line 143, "pan_in", state 42, "data.d1 = nonceA"
      line 144, "pan_in", state 44, "data.d1 = nonceB"
      line 142, "pan_in", state 46, "(knowNA)"
      line 142, "pan_in", state 46, "(knowNB)"
      line 142, "pan_in", state 46, "data.d1 = nonceI"
      line 148, "pan_in", state 49,
"network!msg3,bob,data.key,data.d1,data.d2"
      line 152, "pan_in", state 59, "D_STEP"
      line 152, "pan_in", state 59, "(1)"
      line 161, state 78, "D_STEP"
      line 176, state 82, "-end-"
      (27 of 82 states)
```