A FRAMEWORK FOR MODEL CHECKING OBJECT ORIENTED SECURITY PROTOCOL IMPLEMENTATIONS

by

PARIKSHIT A SINGH

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2005

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. David Kung for providing guidance, support and motivation through the course of this research effort.

I am grateful to Dr. Che and Dr. Lei for serving on my committee. Finally, I would like to thank all friends and members of family for their support during my Masters education.

June 7, 2005

ABSTRACT

A FRAMEWORK FOR MODEL CHECKING OBJECT ORIENTED SECURITY PROTOCOL IMPLEMENTATIONS

Publication No. _____

Parikshit A Singh, M.S.

The University of Texas at Arlington, 2005

Supervising Professor: Dr. David Kung

With the rapid growth of the Internet, more and more vendors see the Internet as a viable marketplace. Since the Internet is public, providing security in the presence of malicious intruders has become paramount. Security protocols have been proposed to protect systems. These protocols work by exchanging messages, many of which are encrypted. Though it may take a long time for an intruder to break the underlying encryption employed by the protocol, it is possible for the intruder to intervene in the authentication process. It may take years before a crucial loophole is discovered in a security protocol. Until then, its implementation may remain in use. There are several

methods for verifying security protocols from their specifications. A specification that is successfully verified for some properties does not imply that the implementation created from it will also satisfy those properties. In this paper, we show a framework for model checking object oriented implementations of security protocols. According to this framework, we reverse engineer security protocol implementations to UML sequence diagrams for a particular use case. The sequence diagrams are then converted to state machines for each principal participating in the use case. The state machine of each principal is used to generate its Promela model. Promela is the language used by the SPIN model checker. Once we have a Promela model for each principal involved in the use case, we can use the SPIN model checker to check if a particular property is satisfied. As a case study, we use an implementation of the NSPK protocol and check if the implementation satisfies the property of *authenticity*. We conclude by showing that the implementation is prone to an attack from an intruder and that the property of *authenticity* is violated.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

CHAPTER 1

INTRODUCTION

With the increased usage of distributed systems (like the Internet), where resources and data are shared among several users located anywhere in the world, the need for secure communication is paramount. Therefore various security protocols were developed. Security protocols have three primary properties viz. authenticity, secrecy and integrity.

In this work we show a framework for model checking OO implementation of security protocols. We achieve this by reverse engineering the implementation to sequence diagrams and control-flow state machines. Control-flow state machines are defined in section 5.2. Then we generate Promela code from control-flow state machine to verify a given property. To illustrate the framework, we use our Java implementation of the NSPK protocol and reverse engineer it to Promela code. As a first step, we select a use case and generate a sequence diagram for each principal participating in the use case. Next, we translate the sequence diagrams to control-flow state machines through well-defined rules. Finally, we generate Promela code for each principal from its control-flow state machine and prove that the Promela model is prone to attack by an Intruder. The main contributions of this paper are:

- Formal transformational semantics for UML sequence diagram and control-flow state machine

- A method for translating sequence diagrams to control-flow state machines
- A procedure to generate Promela code from the control-flow state machine

The structure of the paper is as follows: Section 2 gives a brief overview of concepts like Model Checking, security protocol, UML sequence diagram, state machine, Promela and the SPIN model checker. These are helpful to understand the concept outlined in this paper. Section 3 gives a brief account of work that has already been done in areas related to our work. Sections 4, 5 and 6 explain the detailed design of our framework with definitions, lemmas, assumptions, rules and figures of the initiator sequence diagram and control-flow state machine. Section 7 is a case study of the NSPK protocol to demonstrate the framework. Section 8 summarizes the paper and also highlights the future work. Appendices A and B provide complete sequence diagrams for the implementation of the initiator and responder of NSPK [Clarke, E. M. and Jacob, J., 1997] protocol. Appendix C provides the Promela code for the authentication process employed by the initiator and responder in the NSPK protocol.

CHAPTER 2

BACKGROUND

2.1 SPIN, Promela and Model Checking

The term model checking designates a collection of techniques for the automatic analysis of reactive systems. Subtle errors in the design of safety-critical systems that often elude conventional simulation and testing techniques can be (and have been) found in this way. Because it has been proven cost-effective and integrates well with conventional design methods, model checking is being adopted as a standard procedure for the quality assurance of reactive systems. The inputs to a model checker are a (usually finite-state) description of the system to be analyzed and a number of properties, often expressed as formulas of temporal logic, that are expected to hold of the system. The model checker either confirms that the properties hold or reports that they are violated. In the latter case, it provides a counterexample: a run that violates the property. Such a run can provide valuable feedback and points to design errors. In practice, this view turns out to be somewhat idealized: quite frequently, available resources only permit to analyze a rather coarse model of the system. A positive verdict from the model checker is then of limited value because bugs may well be hidden by the simplifications that had to be applied to the model. On the other hand, counter-examples may be due to modeling artifacts and no longer correspond to actual system runs. In any case, one should keep in mind that the object of analysis is always an *abstract model* of

the system. Standard procedures such as code reviews are necessary to ensure that the abstract model adequately reflects the behavior of the concrete system in order for the properties of interest to be established or falsified. Model checkers can be of some help in this validation task because it is possible to perform “sanity checks”, for example to ensure that certain runs are indeed possible or that the model is free of deadlocks [Merz, S., 2001].

SPIN [Holzmann, G. J., 1997] is the most widely used formal verification tool today. SPIN can be used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, etc. The tool checks the logical consistency of a specification. To verify a design, a formal model is built using Promela, SPIN's input language. SPIN can be used in three basic modes:

- As a protocol simulator, allowing for rapid prototyping with a random, guided, or interactive simulations
- As an exhaustive state space analyzer, capable of rigorously proving the validity of user specified correctness requirements
- As a bit-state space analyzer that can validate even very large protocol systems with maximal coverage of the state space (a proof approximation technique).

Promela (Protocol Meta Language) is a non-deterministic language, loosely based on Dijkstra's guarded command language notation and C.A.R. Hoare's language CSP, extended with some powerful new constructs. It contains the primitives for specifying asynchronous (buffered) message passing via channels, with arbitrary

numbers of message parameters. It also allows for the specification of synchronous message passing systems. Mixed systems, using both synchronous and asynchronous communications, are also supported.

The language can model dynamically expanding and shrinking systems: new processes and message channels can be created and deleted on the fly. Message channel identifiers can be passed from one process to another in messages. Correctness properties can be specified as standard system or process invariants (using assertions), or as general linear temporal logic requirements (LTL), either directly in the syntax of next-time free LTL, or indirectly as Buchi Automata (expressed in Promela syntax as Never Claims).

2.2 Security Protocols

Security protocols ensure that information gets exchanged in a secure manner. Because security protocols are so important, it becomes essential to ensure that they are indeed correct. Important kinds of correctness criteria are:

- Authenticity - a principal is actually whoever he/she claims to be.
- Secrecy - the contents of a secure communication must not be leaked to outsiders.
- Integrity - no outsider must be able to interfere with the communication.

These criteria can be verified by employing formal techniques like model checking. To ensure a secure communication, security protocols employ the mechanism of authenticating each entity involved in communication. The primary aim is to ensure that none of the above three correctness properties are compromised. There are several

security protocols being used in applications in various areas of computing. They are thoroughly verified for any flaws before being implemented. Our approach provides a way to model check such implementations (if they are object oriented). We view the authentication process employed by each principal in different states, which allows us to model check the process.

2.3 UML

Unified Modeling Language (UML) is a standardized notation for object-oriented analysis and design. It is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a system. A UML model is an abstraction and does not contain every detail of the system. A model is represented as a set of diagrams in UML; each of these diagrams has its own set of elements. Some UML diagrams pertinent to our work are:

- Class diagrams: They describe the static structure of the system, class abstraction and relationships between classes.
- Sequence Diagrams: Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.
- State Chart Diagrams: State chart diagrams describe the dynamic behavior of a system in response to external event. State chart diagrams are especially useful in modeling reactive objects whose states are triggered by specific events. There are other diagrams too, but we do not use them in our work.

CHAPTER 3

RELATED WORK

There has been a lot of work in the areas of protocol verification using the SPIN model checker, formalizing UML state machines and reverse engineering, all of which is related to our framework. UML is used to model all kinds of systems. Modeling tools are available that make creation of UML models easy and can also generate code from some of the UML diagrams. But a UML model must still be verified as it can have behaviors not expected by the designers. This requires formalization of UML state machines. Some work has already been done in this direction [Latella, D., Majzik, I. and Massink, M., 1999]. This formalization is needed for code generation, simulation and verification of UML state charts. Lacey and DeLoach [Lacey, T. and DeLoach, S., 2000] transform state transition diagrams to formal modeling language Promela to verify multiagent conversations.

UML state charts have become a successful specification method for describing dynamic aspects of object behaviors. Model checking techniques have been applied to state charts and to variants of state charts [Tenzer, J., Stevens, P., 2003]. Usually, the encoding is based on a static precomputation of the possible transitions between state configurations. Lilius and Paltor [Paltor, I., Lilius, J., 1999] have defined an operational semantics of UML state machines in Promela as a basis for vUML, based on the SPIN model checker. In the vUML system, the UML model is translated to Promela, input

language for SPIN. If SPIN finds an error in the system it produces error trace of the counter-example. This counter-example is translated to UML sequence diagram and displayed to the user. The user does not need to know PROMELA or SPIN to use vUML. vUML is targeted towards the designer of object oriented systems to verify UML designs. Our framework targets the tester of object oriented code. We reverse engineer object oriented code to interaction diagram and control-flow state machine. Then we verify if some desirable properties are satisfied by the implementation. HUGO[Schafer, T., Knapp, A. and Merz, S., 2001], proposed by Schafer, Knapp, Merz, is a prototype tool designed to automatically verify whether the interactions expressed by a collaboration diagram can indeed be realized by a set of state machines. It compiles the state machines into a Promela model, and collaborations into sets of Buchi automata ("never claims"). The model checker SPIN is then called upon to verify the model against the automata. Model checking has proven to be a very useful technique for verifying security protocols. By modeling protocols as finite state machines and examining all possible execution traces, model checking can be used to find errors in system design. Security protocols are very subtle and can have bugs that are difficult to find. By examining all execution traces of a security protocol in the presence of a malicious user with well-defined capabilities, it can determine if a protocol does indeed enforce its security guarantees. If not a sample trace of an attack on a protocol can be provided. However, model checking based approaches are applicable to finite state systems and suffer from the state explosion problem.

One of the earliest attempts at formalizing security protocols involved the development of a new logic that could express and deduce security properties. The earliest such logic is referred to as the BAN Logic proposed by Burrows, Abadi and Needham [Burrows, M., Abadi, M. and Needham, R., 1989]. BAN logic syntax provides constructs for expressing intuitive properties and creating rules, which can be used to deduce security properties based on assumptions made about the protocol. A number of extensions to BAN logic have been proposed since its introduction and it remains popular because of its simplicity and high level of abstraction. Dolev and Yao [Dolev, D., Yao, A., 1989] took a different approach to achieve this formalism. Their approach was to model a protocol by defining a set of states and a set of transitions taking into account the intruder, exchange of messages between principals and knowledge of principals. The state space could then be checked to see if a particular state could be reached. This approached was adapted and extended by Meadows in her PROLOG based protocol analyzer [Meadows, C., 1994], which later evolved to become the NRL protocol analyzer [Meadows, C., 1994]. In her system, the user can model the protocol as set of rules, which describe how intruder can generate knowledge by encryption and decryption of messages and by receiving responses from other principals. To perform verification user has to provide description of the insecure state. The model checker searches backwards to find an initial state. If initial state is found then the system is said to be insecure else it is considered an unreachable state. Woo and Lam [Woo, T. Y. C. and Lam, S. S., 1993] proposed an intuitive model for authentication protocols. Their model resembles sequential programming where

principals are modeled independently. Other work in this area has involved trying to use generic verification tools to verify security protocols. Lowe [Lowe, G., 1996] uses the FDR model checker for CSP [Hoare, C. A. R., 1985] to verify security protocols. In his approach, principals and intruders are modeled as CSP processes. The intruder process can deduce new knowledge by using some inference rules on a set of existing facts. All communication is assumed to go through the intruder. Security properties are specified in trace semantics. FDR verifies a security protocol by enumerating all the behaviors of the protocol and checking whether they are allowed by the trace semantics of desirable security properties. Brutus developed by Clarke et al. is a special-purpose model checker for verifying security protocols [Marrero, W., Clarke, E. M. and Jha, S., 1997]. In Brutus, principals and the intruder are modeled as named processes, and a protocol is modeled as an asynchronous composition of a set of named processes. Security properties are specified using a variant of linear-time temporal logic. The model-checking algorithm employed in Brutus is based on state exploration. We take a different approach, since we start from the implementation (object oriented) of the protocol. This allows us to find problems in the implementation that cannot be found in specification testing. We get the sequence diagram from the implementation for a specific use case and generate the control-flow state machine from it. This allows us to model check the implementation for specific properties. The next section provides an overview of the framework.

CHAPTER 4

APPROACH

4.1 Overview

In this section we provide an overview of our approach. We assume that the Java programming language is used but the result is not limited to Java. The framework consists of the following steps:

- Reverse engineer the Java implementation to sequence diagram using an appropriate tool like Together Control Center.
- Convert the sequence diagram of each participating principal to its control-flow state machine.
- Write Promela code for each principal from its control-flow state machine.
- Verify if a desired property is satisfied for the specific use case using the SPIN model checker.

We propose the use of a tool like OOTWorks or Together Control Center to generate sequence diagrams from Java source code. In order to generate a sequence diagram using Together Control Center, we do not need to map the diagram to a use case. We key into a method that triggers a sequence of events of the object. These events define the phase or aspect of the object during which, we can conclusively determine if a desired property is violated.

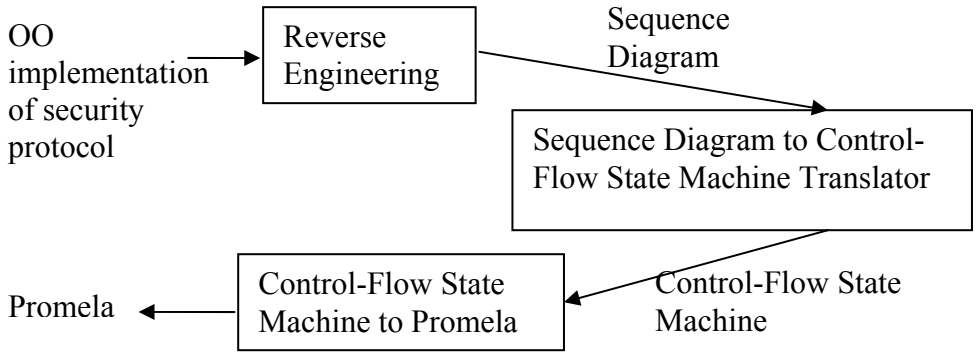


Figure 4.1 Block diagram of the framework

We provide an outline of the framework in the steps 1 through 5 below. The first component in figure 4.1 is *Reverse Engineering* to generate sequence diagram. This component is broken up into steps 1 and 2 below. Steps 3 and 4 explain the flow of the remaining block diagram.

Step 1: *Identify the use case and generate sequence diagram for the use case*

Identify the use case to generate the sequence diagram. The use case depends on the property to be verified. In order to generate a sequence diagram using Together Control Center, we need not map the sequence diagram to a use case.

Step 2: *Generate sequence diagram for each principal and exclude objects from the sequence diagram that are not directly related to the pertinent use case.*

First, select the method of the principal that triggers a sequence of interactions which form the use case. Then generate the sequence diagram using the utility provided by Together Control Center. This process will generate a sequence of interactions or events of the specific object or principal. In other words, it will generate a sequence

diagram of a selected principal. Together also allows the user to exclude selected classes before generating sequence diagrams for any use case. APIs like java.lang.System, java.lang.String, java.net.*, java.math.*, which are not directly related to the use case can be safely excluded. For example, appendix A shows the sequence diagram of the initiator of the NSPK protocol for the authentication use case. We removed objects not directly related to the use case and generated the sequence diagram shown in figure 7.2. This diagram, although less detailed, is fully informative and easy to understand.

Step 3: *Create control-flow state machine for each participating principal from its sequence diagram*

Control-flow state machine is a control flow graph of the principal that has state nodes and transition arcs interconnecting the nodes. It is different from the attribute-value based state machine mentioned in the UML specification. We provide more details on differences between the two in section 5.2

Step 4: *Write Promela code for each principal from its control-flow state machine and use the SPIN model checker to verify properties of the selected use case.*

Use the control-flow state machine of a principal to write its corresponding Promela model. We outline directions to generate a Promela model from a control-flow state machine in sections 6.2.1 and 6.2.2. Specifiy the property chosen for verification in linear temporal logic (LTL), so that it can be verified using the SPIN [Holzmann, G. J., 1997] model checker.

From the next section we explain in detail how to follow the above steps. We provide a set of formal rules to generate control-flow state machine from a sequence diagram. To provide sufficient background for the rules, we detail formal transformational semantics of the UML sequence diagram. Then we give an explanation of control-flow state machine and present its semantics in context of our work. The semantics of sequence diagram and control-flow state machine are explained in section 5. Section 6 explains rules to convert sequence diagram to control-flow state machine and generate Promela model from the control-flow state machine respectively.

CHAPTER 5

SEMANTICS

5.1 Sequence Diagram

We provide transformational semantics for a sequence diagram that enable us to transform it to a control-flow state machine for a principal. Our definition of sequence diagram has two dimensions, viz. the vertical dimension and the horizontal dimension. The former represents time, and the latter represents different objects. This also forms the basis for the definition of the UML sequence diagram. We define a sequence diagram as a tuple with two elements, viz. the set of objects (horizontal dimension) and the set of events in time (vertical dimension).

Definition 1 – Total Sequence Diagram: A total sequence diagram for use case 'u' is the tuple $D(u) =< O, Q >$, where O is the set of objects participating in the use case and Q is the set of events or interactions among objects in O.

The set of events Q in definition 1 is the relation $Q \subseteq O \times O \times M \times N$ where,

- O *is the set of all objects*
- M is the set of all method calls in the sequence diagram
- N is the set of non-zero natural numbers

For example, let the sequence diagram in the figure 5.1, shown below be $D(u) =< O, Q >$. The events q_i, q_j, q_k and $q_l \in Q$.

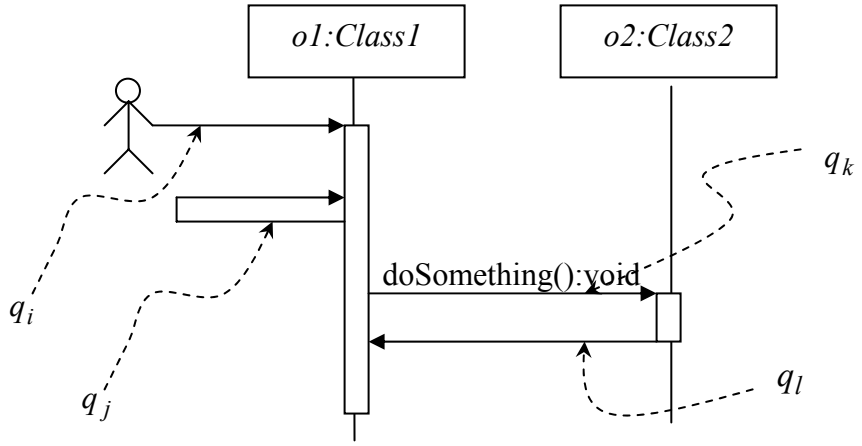


Figure 5.1 Sequence diagram, $q_k = (o_1, o_2, m, n)$, $m =$ doSomething():void and $n = 3$

For an event q , $q|_N$ is a value that indicates the relative order of its appearance in the sequence diagram. For the first event (the event that triggers the use case), this value is '1'. For the second event, this value is '2' and so on. For example, in figure 5.1, $q_i|_N = 1$ and $q_k|_N = 3$. Therefore, if an event q_i appears before another event q_j , as in the case of figure 5.1, then $q_i|_N < q_j|_N$.

5.1.1 Sequence Diagram of a Principal

A total sequence diagram of a security protocol implementation contains all principals and objects participating in the use case. We can, however focus on a particular principal and generate a sequence diagram, which shows the principal's interactions only.

Definition 2 – Sequence Diagram of a principal: Given the total sequence diagram for a use case 'u' as $D(u) =< O, Q >$, the sequence diagram of a principal ' p ' is the

tuple $D(u)|_p =< O_p, Q_p >$, where O_p is the set of objects excluding all principals except p and Q_p is the set of events or interactions among objects in O_p.

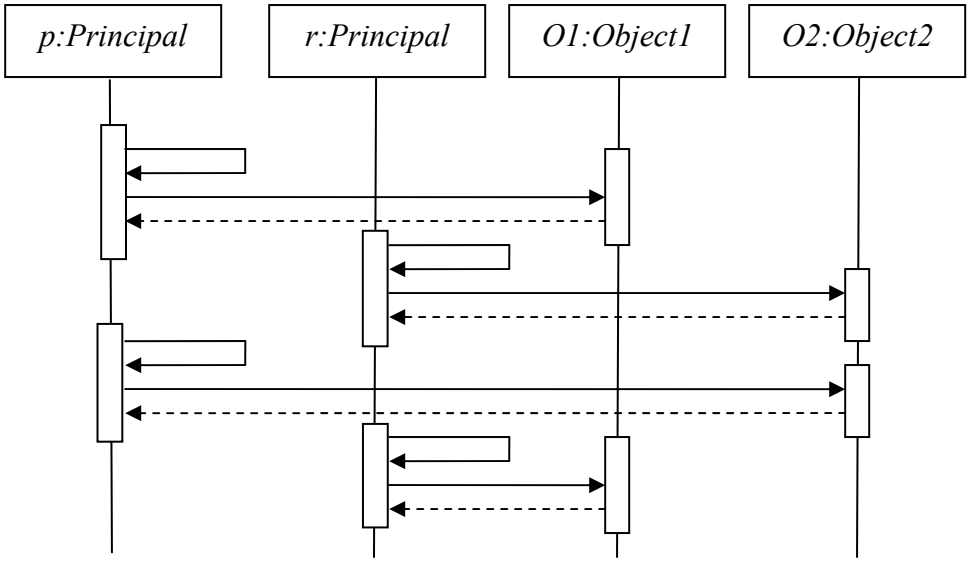


Figure 5.2 Total sequence diagram showing principals *p* and *r*

In definition 2, we have the following.

- $O_p = (O - P) \cup \{p\}$, where P is the set of principals and O is the set of all objects in the total sequence diagram $D(u) =< O, Q >$.
- $Q_p \subseteq O_p \times O_p \times M_p \times N$, where Q_p represents all time-ordered *outgoing* and *incoming* events of the principal p.
- $$Q_p = \{< o_i, o_j, m, n > | (\exists o, p \in O_p)(\exists m \in M_p)(\exists n \in N) \\ (< p, o, m, n > \in Q) \wedge (< o, p, m, n > \in Q) \wedge (< p, p, m, n > \in Q)\}$$

As an example for definition 2, compare figures 5.2 and 5.3. Figure 5.2 shows the total sequence diagram assuming there are two principals *p* and *r* in the set P. On

the other hand, figure 5.3 shows the sequence diagram of principal *p*. From this point forward, we shall refer to events for a particular principal, i.e. Q_p. This allows us to separate roles of each principal in P and construct its control-flow state machine.

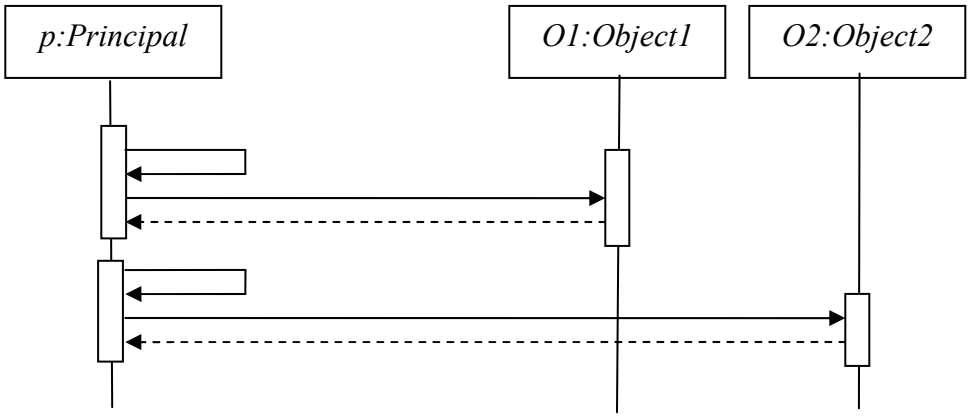


Figure 5.3 Sequence diagram of principal *p*

5.1.2 Scopes in a Sequence Diagram

A scope is a block of code between an opening curly brace, i.e. '{', and a closing curly brace, i.e. '}'. Together Control Center specifically highlights[1] scopes due to looping constructs and conditional statements. We make use of only those scopes that result due to the occurrences of looping constructs and conditional statements.

Figure 5.4 shows a sequence diagram with various events for a principal *p*. Events q_i and q_j are in the outermost scope and event q_k is in a scope *nested* within the outermost scope. For a principal *p*, we denote the outermost scope as $S_p(0)$ and

[1] Scopes due to looping constructs and conditional statements are highlighted dark in the sequence diagrams generated by Together Control Center. This highlighting is visible in the sequence diagram shown in figure 10.

inner scopes as $S_p(i), i = 1,2,...$ based on the level of nesting. For example, event q_i in figure 5.4 is in scope $S_p(0)$ and event q_k is in scope $S_p(1)$. In order to distinguish scopes at the same level of nesting within a conditional construct ("*if-else*" or "*switch"* like constructs), we assign a *superscript* number starting from '0' to every scope within such a construct. As an example, scopes $S_p^0(1)$ and $S_p^1(1)$ in figure 5.4 are distinguished by their superscript numbers 0 and 1 respectively.

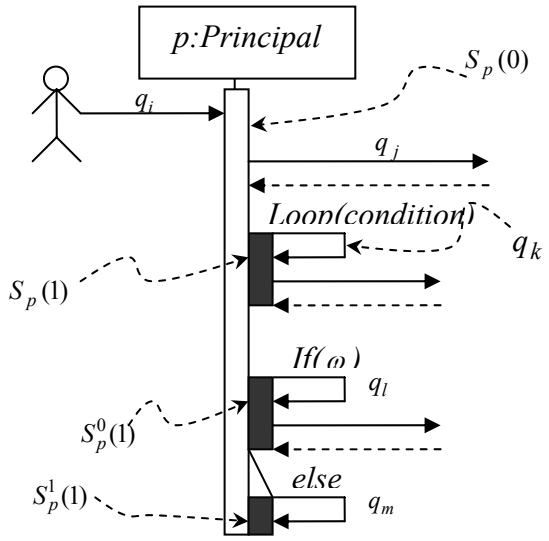


Figure 5.4 Sequence diagram

5.1.3 Precedence among Events in a Sequence Diagram

Definition 3 – Precedence and Precedence Relation: For any two events $q_i, q_j \in Q_p$, q_i precedes q_j implies that the procedural control reaches q_i before it reaches q_j. This is denoted as $q_i \prec q_j$ (read as q_i precedes q_j). The relation between q_i and q_j is the Precedence relation.

For example, in the sequence diagram shown in figure 5.5, procedural control reaches q_i before q_j, hence $q_i \prec q_j$. A related and more significant term – *immediate precedence* is defined below.

Definition 4 – Immediate Precedence: For any two events $q_i, q_j \in Q_p$ such that $q_i \prec q_j$, if there is no event $q_X \in Q_p$ such that $q_i \prec q_X$ and $q_X \prec q_j$ then q_i immediately precedes q_j. We write this relation as $q_i \prec^* q_j$.

As an example, consider the sequence diagram in figure 5.5. The relation between events q_i and q_j is $q_i \prec^* q_j$ (q_i *immediately precedes* q_j) but the relation between events q_i and q_k is $q_i \prec q_k$ (q_i *precedes* q_k).

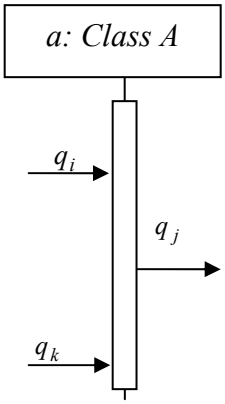


Figure 5.5 Sequence diagram - $q_i \prec^* q_j$ and $q_i \prec q_k$

Corollary 1: $(\forall q_i, q_j \in Q_p)(q_i \prec^* q_j \rightarrow q_i \prec q_j)$. The reverse is not necessarily true.

Lemma 1: The presence of an unconditional jump (like 'break') affects precedence relations.

Procedural control changes due to presence of unconditional jumps. As per definition 3, precedence relies on procedural control, and therefore it changes due to the presence of unconditional jumps. The figure 5.6 below illustrates this assertion. In figure 5.6, procedural control will never reach q_k, therefore there will be only two precedence relations viz. $q_i \prec^* q_j$ and $q_j \prec^* q_l$.

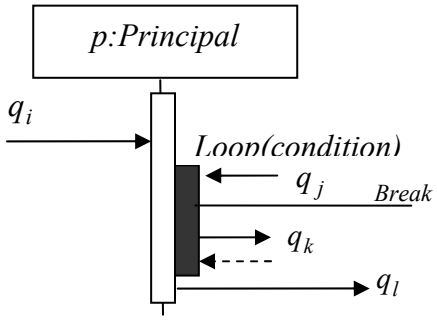


Figure 5.6 Sequence diagram - $q_i \prec^* q_j$ and $q_j \prec^* q_l$

5.2 Control-Flow State Machine

Definition 5 - State: A state in a Control-flow State Machine is a condition during the life of the principal or an interaction during which it performs some action. To some extent, this definition is similar to the definition of a state specified in the Unified Modeling Language Specification, version 1.5 (formal/03-03-01), March 2003.

The OMG UML specification depicts the state machine as a specification that describes all possible behaviors of some dynamic model element. Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by dispatching a series of event instances [Object Management

Group. OMG Unified Modeling Language Specification, version 1.5 (formal/03-03-01). March 2003].

In our case, the model element is a principal of a security protocol and its control-flow state machine describes its behavior for a specific aspect of its lifetime. We also state that the control-flow state machine is not a complete state machine in itself. It is a control flow graph of the principal that has state nodes and transition arcs interconnecting the nodes. It is different from the attribute-value based state machine in the UML specification. The control-flow state machine is like a control flow graph, whereas the state machine in UML has attributes and other features that we can safely exclude to check control flow intensive properties.

5.2.1 Control-Flow State Machine of a Principal

Definition 6 – Control-Flow State Machine of a principal: The Control-Flow State Machine of a principal ' p ' is the tuple $SM_p =< \Sigma_p, \gamma_p >$, which consists of a set of states Σ_p and a transition relation γ_p.

The transition relation is $\gamma_p \subseteq \Sigma_p \times \Lambda \times \Sigma_p$, where

- Σ_p is the set of all states in SM_p.

$\Sigma_p = \{\sigma \mid \sigma \text{ is a state in the state machine of principal } p\}$

- Λ is the set of all transitions taken by principal p between states in the set Σ_p.

$\Lambda = \{\lambda[\phi] \mid \lambda \text{ is the transition and } \phi \text{ is the condition under which it takes place}\}$

superscript numbers 0 and 1 respectively.

CHAPTER 6

RULES

6.1 Rules to Generate Control-Flow State Machine of a Principal from its Sequence Diagram

In this part, we describe rules to generate control-flow state machine of a principal from its sequence diagram. In other words, given a sequence diagram $D|_p =< O_p, Q_p >$, we explain how to generate the control-flow state machine $SM_p =< \Sigma_p, \gamma_p >$ of principal *p*. Rule I below, is our fundamental rule for creating states of a principal's control-flow state machine from its sequence diagram $D|_p =< O_p, Q_p >$.

Rule I: For all events q_j, q_X and q_k within the same scope, where q_X is between the two incoming events q_j and q_k, such that $q_j \prec^* q_X \prec^* q_X^r \prec^* q_k$, where q_X is an outgoing event and its return event is q_X^r

Do the following.

- **Create the state** - Create a state corresponding to event q_X and denote it σ^{qX}. Represent the state according to UML specifications as a "rounded rectangle" with two compartments, viz. the name compartment and the internal transitions compartment.
- **Create transitions to and from the state** -

- Create transition $\lambda^{qj}[\phi_j]$ (corresponding to event q_j) leading into state σ^{qX}.
- Create transition $\lambda^{qk}[\phi_k]$ (corresponding to event q_k) leading out of state σ^{qX}.
- For the transitions $\lambda^{qj}[\phi_j]$ and $\lambda^{qk}[\phi_k]$, ϕ_j and ϕ_k are the respective conditions under which the events q_j and q_k are fired. ϕ_j = ϕ_k = NULL if both q_j and q_k are not within scopes of conditional constructs (like "if-else")

- **Name the state** - Enter the name in the name compartment of the "rounded rectangle" that diagrammatically represents σ^{qX}.
- **Provide an action expression for the state:** Enter the action expression as "m_X / a_X" in the internal transitions compartment, where
  - $m_X = q_X|_M$ is the method name of event q_X
  - a_X is the action performed by the principal when it is in the state σ^{qX}.

Example 1: Consider the sequence diagrams shown in figure 6.1.

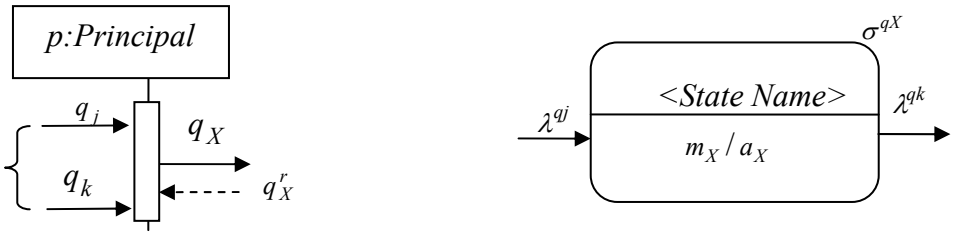


Figure 6.1 $q_j \prec^* q_X \prec^* q_k$, $q_j \prec^* q_X \prec^* q_X^r \prec^* q_k$ and σ^{qX}

The precedence relations among events in figure 6.1 are $q_j \prec^* q_X \prec^* q_k$ and $q_j \prec^* q_X \prec^* q_X^r \prec^* q_k$. We apply rule I to these precedence relations to get the state σ^{qX} with transitions λ^{qj} and λ^{qk} shown on the right hand side of figure 6.1.

The example provided to support rule I above demonstrates the case where there is a single transition to exactly one state and a single transition leading out of it. Rule I can be applied to cases where there are multiple states and transitions. Below, we discuss two such cases and explain how rule I may be applied to them. The rule, however, is not limited to the two cases. All references made hereinafter to the return event of an outgoing event, say q will be denoted as q^r

Case 1 – Transitions to/from multiple states: In the sequence diagram illustrated in figure 6.2, if *condition* is true then procedural control will transfer from q_l to q_i and then to q_n. Then by definition of precedence we will have $q_l \prec^* q_i \prec^* q_i^r \prec^* q_n$. However, if *condition* is false then we will have $q_m \prec^* q_j \prec^* q_j^r \prec^* q_n$. The precedence relationships are shown diagrammatically on the right hand side of figure 6.2.

Apply rule 1 to the precedence relations in figure 6.2 to get states and transitions leading into and leading out of the states as shown in the figure below.

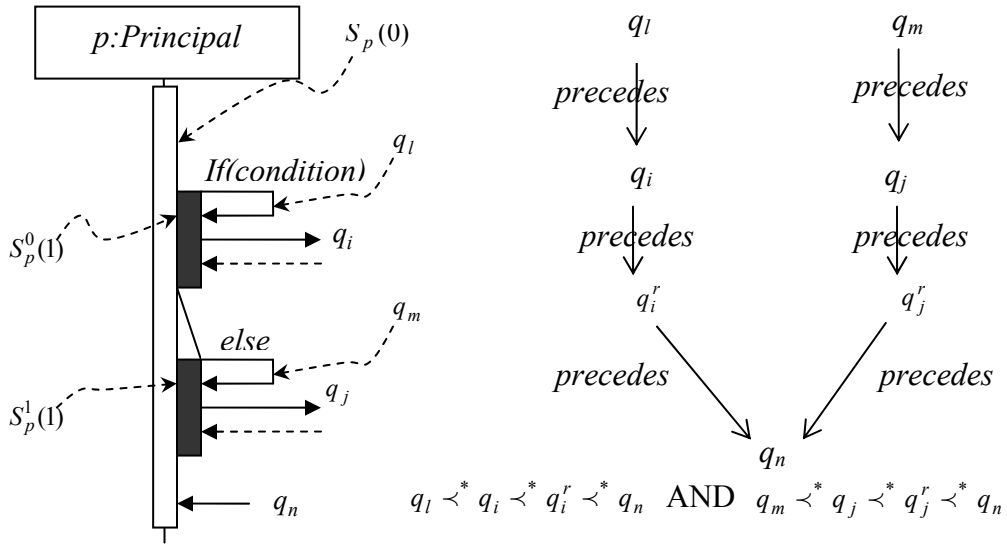


Figure 6.2 Sequence diagram and the corresponding precedence relations

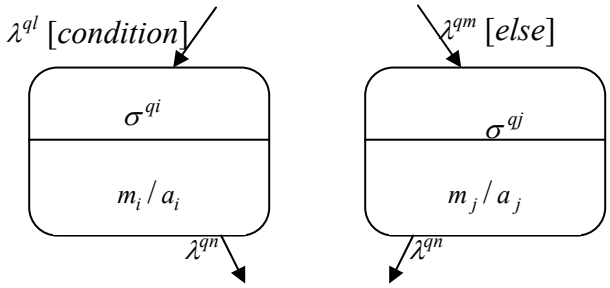


Figure 6.3 Rule I applied to case 1

Case 2 – Multiple transitions to/from one and only one state: In the sequence diagram illustrated in figure 6.4, if *condition* is true then procedural control will transfer from q_l to q_i and then to q_m. Then by definition of precedence we will have $q_l \prec^* q_i \prec^* q_i^r \prec^* q_m$. Similarly, if *condition* is false then we will have

$q_l \prec^* q_i \prec^* q_i^r \prec^* q_n$. The precedence relationships are shown diagrammatically on the right hand side of figure 6.4.

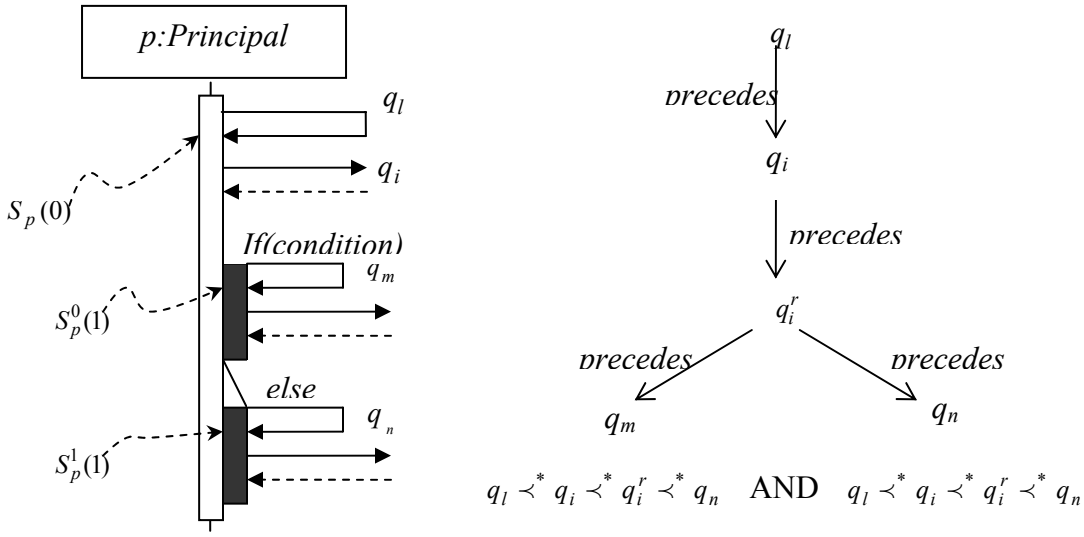


Figure 6.4 Sequence diagram and the corresponding precedence relations

Apply rule 1 to the precedence relations in figure 6.4 to get states and transitions leading into and leading out of the states as shown in the figure below.

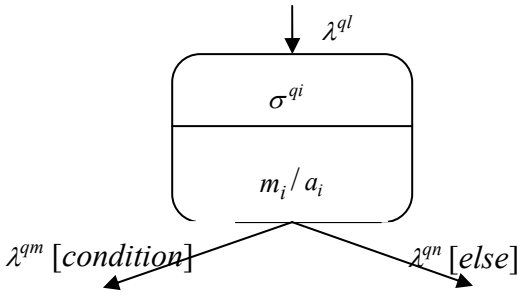


Figure 6.5 Rule I applied to case 2

Rule II: Do not create more than one state with the same name in the control-flow state machine.

This rule is adopted for clarity and to avoid unnecessary confusion. If a state σ^{qX} is created using rule 1, then we explicitly prohibit creating another state that is same as σ^{qX}. The OMG Unified Modeling Language Specification, version 1.5, March 2003 states, "It is undesirable to show the same named state more than once in the same diagram, as confusion may ensue". If there is a circumstance in which, application of rule I is resulting in the creation of a new state that is the same as an already created state, then do not create the new duplicate state. Instead direct the transition to the already created state.

6.1.1 Directions to create Control-Flow State Machine from Sequence Diagram

The directions mentioned below outline steps to generate a control-flow state machine of a principal from its sequence diagram. They are based on the following assumptions.

- The sequence diagram $D|_p =< O_p, Q_p >$ of the principal (*p* in this case) is defined.
- Rules I and II described in section 6.1 (herein referred to as rule I and rule II) are defined
- Unconditional jumps (like *break*) within looping constructs are visible in the sequence diagram

We use the following conventions for algorithms in figures 6.6 and 6.7. The last two points are inspired from the conventions adopted by Cormen et al. in [Cormen, T. H., Leiserson, C. E., Rivest and R. L., Stein, C., 2001].

- The event q^r represents the return event of an outgoing event q.
- References made to any transition λ^q imply $\lambda^q[\phi]$, where ϕ is the condition.
- "*NULL*" is used to represent an empty set.
- Multiple assignment like $i \leftarrow j \leftarrow e$ assigns to variables i and j the value of e.
- The symbol "$\triangleright$" indicates that the remainder of the line is a comment.

Now, follow the directions given below to create a control-flow state machine $SM_p =< \Sigma_p, \gamma_p >$ for principal p.

(i) The *Start* state: Create the first state as the *Start* state. Create the transition leading to this state as λ^{qi}, where q_i is the event that triggers the use case.

Input: Sequence diagram $D|_p =< O_p, Q_p >$

Let q_i be the event that triggers the use case

Let q_j be an incoming event such that $q_i \prec^* q_j$

Initially, let $\Sigma_p \leftarrow \Lambda \leftarrow \gamma_p \leftarrow NULL$

Do $\Sigma_p \leftarrow \Sigma_p \cup \{\sigma^{START}\}$

$\Lambda \leftarrow \Lambda \cup \{\lambda^{qi}, \lambda^{qj}\}$

At this point the relation γ_p is *NULL*

Figure 6.6 Algorithm to create the *Start* state of control-flow state machine

Proceed with event q_j from the algorithm in figure 6.6.

Let $Q_{\sigma,out} \leftarrow Q \leftarrow NULL$ ▷ Let $Q_{\sigma,out}$ and Q be empty.

Initially, let $Q_{\sigma,in} \leftarrow Q_\sigma \leftarrow \{q_j\}$ ▷ Initialize $Q_{\sigma,in}$ and Q_σ

CreateSM ($Q_{\sigma,in}$, Q_σ **)**

1. $\forall q_a \in Q_{\sigma,in}$
2. **Do** Let $q_{\sigma,in} \leftarrow q_a$
3. $\forall q, q_b \in Q_p$ **such that** ($q_{\sigma,in} \prec^* q \prec^* q^r \prec^* q_b$) **and** q_b is incoming
4. **Do** Let $q_{\sigma,out} \leftarrow q_b$
5. Apply rule I to $q_{\sigma,in}$, q and $q_{\sigma,out}$ ▷ Represent σ^q in UML.
6. $\Sigma_p \leftarrow \Sigma_p \bigcup \{\sigma^q\}$ ▷ Add the state to the set of states.
7. $Q \leftarrow Q \bigcup \{q\}$ ▷ Q_σ will have events forming
   ▷ the most recently created state.
8. $\forall q_c \in Q_\sigma$ **such that** $q_c \prec^* q_{\sigma,in}$ **or** $q_c^r \prec^* q_{\sigma,in}$
9. **Do** $\Lambda \leftarrow \Lambda \bigcup \{\lambda^{q\sigma,in}, \lambda^{q\sigma,out}\}$ ▷ Transitions into and out of σ^q .
10. $\gamma_p \leftarrow \gamma_p \bigcup \{(\sigma^{qc}, \lambda^{q\sigma,in}, \sigma^q)\}$ ▷ Add ordered-pair to relation γ_p
11. $Q_{\sigma,out} \leftarrow Q_{\sigma,out} \bigcup \{q_{\sigma,out}\}$
12. $Q_{\sigma,in} \leftarrow Q_{\sigma,out}$
13. $Q_\sigma \leftarrow Q$
14. $Q_{\sigma,out} \leftarrow Q \leftarrow NULL$ ▷ Reinitialize $Q_{\sigma,out}$ and Q_σ .
15. **If** $|Q_{\sigma,in}| > 0$ **Then** ▷ If $Q_{\sigma,in}$ is non-empty,
16. **Do CreateSM (** $Q_{\sigma,in}$, Q_σ **)** ▷ Recursively call CreateSM.
17. **Else**
18. **Do return** $SM_p =< \Sigma_p, \gamma_p >$ ▷ Output: SM_p

Figure 6.7 Algorithm to complete the control-flow state machine initiated in figure 6.6

(ii) Apply rule I subsequently, beginning with the event(s) that form transition(s) leading out of the most recently created state.

(iii) At each stage of applying rule I, verify that rule II is never violated. In other words, ensure that more than one state with the same name is never created.

(iv) The End state: Label the state that indicates a successful completion of the use case as the End state.

(v) The Invalid End state: Optionally, label the state that indicates an unsuccessful completion of the use case as the Invalid End state.

Example 2: This example is an exercise to demonstrate the application of rules explained in section 6.1 on a specific use case of the NSPK protocol. Consider a part of the sequence diagram of the Initiator (left hand side of the figure 6.8) for the authentication use case. The full sequence diagram is given in figure 7.2.

We apply rules and directions provided in section 6.1 and generate states and transitions in the following manner.

- The *Start* state: The event q_i that triggers the use case has label *processInputAndSend(String[]):void*. The events q_i and q_j (labeled *createMessage*) satisfy the relation $q_i \prec^* q_j$. Therefore, proceeding as directed in point (i) of section 6.1.1, we create the *Start* state with the transition labeled *processInputAndSend(String[]):void* leading to this state. At this stage, we have $\Sigma_p = \{\sigma^{START}\}$ and $\Lambda = \{\lambda^{qi}, \lambda^{qj}\}$.

- Continuing forward, we work as directed in the algorithm shown in figure 6.7 to support points (ii) and (iii) of section 6.1.1. We begin with the "outer" loop at line 1

with $Q_{\sigma,in} = \{q_j\}$, where q_i has the label *createMessage*. The "inner" loop begins at line 3, for all events q, q_b in Q_p such that $q_{\sigma,in} \prec^* q \prec^* q_b$, where $q_{\sigma,in}$ is q_i. The events q_X, q_k in Q_p and q_X, q_l in Q_p pass the condition to enter the "inner" loop. Thus, we make two passes through the "inner" loop, one each for $q_j \prec^* q_X \prec^* q_X^r \prec^* q_k$ and $q_j \prec^* q_X \prec^* q_X^r \prec^* q_l$, before the "inner" loop terminates. The "outer" loop also terminates since we started with just one element q_i in $Q_{\sigma,in}$. At this stage, we have $Q_{\sigma,out} = \{q_k, q_l\}$, which is assigned to $Q_{\sigma,in}$ at line 12. Also, the set of states Σ_p has a newly added state viz. the *Encrypt* state or σ^{qX}, and the set of transitions Λ, populated with newly added transitions λ^{qk} and λ^{ql} becomes $\{\lambda^{qi}, \lambda^{qj}, \lambda^{qk}, \lambda^{ql}\}$. At line 16, we make a recursive call to *CreateSM*($Q_{\sigma,in}, Q_\sigma$), with $Q_{\sigma,in} = \{q_k, q_l\}$ and $Q_\sigma = \{q_X\}$ as arguments.

- Now starting with the "outer" loop at line 1, we have $Q_{\sigma,in} = \{q_k, q_l\}$, $Q_\sigma = \{q_X\}$ and the empty sets $Q_{\sigma,out}$ and Q. First, let us iterate with $q_{\sigma,in} = q_k$. This allows only one pass through the "inner" loop, i.e. for $q_k \prec^* q_Y \prec^* q_Y^r \prec^* q_m$. As a result, we add the *Sending* state to Σ_p along with the transition λ^{qm} to Λ. Next, we iterate through the "outer" loop with $q_{\sigma,in} = q_l$. This also allows us exactly one pass through the "inner" loop, i.e. for $q_l \prec^* q_Z \prec^* q_Z^r \prec^* q_n$. The "inner" and "outer" loops terminate with $Q_{\sigma,in} = \{q_m, q_n\}$ at line 12 and $Q_\sigma = \{q_Y, q_Z\}$ at line 13. At

this stage, the set of states Σ_p has the *Start, Encrypt* and *Sending* states, added in that order. The set of transitions Λ has elements $\{\lambda^{qi}, \lambda^{qj}, \lambda^{qk}, \lambda^{ql}, \lambda^{qm}, \lambda^{qn}\}$. The right hand side of figure 6.8 shows the states and transitions created during the iterations described above. See the complete control-flow state machine of the initiator in figure 7.2.

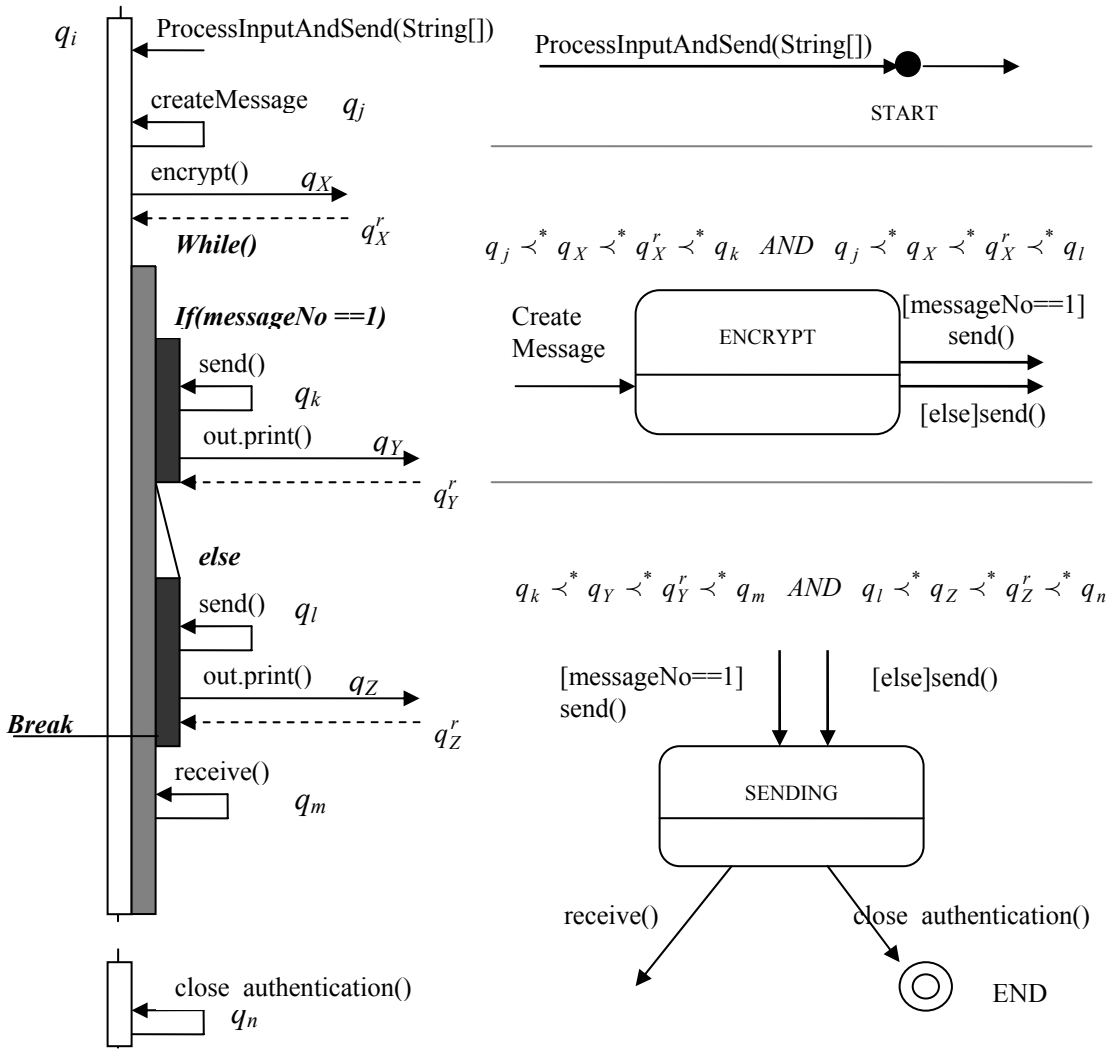


Figure 6.8 Parts of sequence diagram and control-flow state machine of the initiator of the NSPK protocol

Now starting with the "outer" loop at line 1, we have $Q_{\sigma,in} = \{q_k, q_l\}$, $Q_\sigma = \{q_X\}$ and the empty sets $Q_{\sigma,out}$ and Q. First, let us iterate with $q_{\sigma,in} = q_k$. This allows only one pass through the "inner" loop, i.e. for $q_k \prec^* q_Y \prec^* q_Y^r \prec^* q_m$. As a result, we add the *Sending* state to Σ_p along with the transition λ^{qm} to Λ. Next, we iterate through the "outer" loop with $q_{\sigma,in} = q_l$. This also allows us exactly one pass through the "inner" loop, i.e. for $q_l \prec^* q_Z \prec^* q_Z^r \prec^* q_n$. The "inner" and "outer" loops terminate with $Q_{\sigma,in} = \{q_m, q_n\}$ at line 12 and $Q_\sigma = \{q_Y, q_Z\}$ at line 13. At this stage, the set of states Σ_p has the *Start, Encrypt* and *Sending* states, added in that order. The set of transitions Λ has elements $\{\lambda^{qi}, \lambda^{qj}, \lambda^{qk}, \lambda^{ql}, \lambda^{qm}, \lambda^{qn}\}$. The right hand side of figure 6.8 shows the states and transitions created during the iterations described above. See the complete control-flow state machine of the initiator in figure 7.2.

6.2 Promela

6.2.1 Introduction

In this section we introduce the basics of constructing a Promela model. Promela has a construct called *proctype* that models a principal. The *proctype* of each principal will contain its part of the conversation. The idea is to begin the *proctype* in the START state and end in the END state, while moving from one state to another only if there is a transition from that state to the other state.

The initial steps to write the Promela code are to declare and define messages for communication between the principals and define a channel for the communication.

Messages must be declared using an *mtype* declaration. This declaration would typically contain keys, nonces and identities of all principals, e.g.

```
mtype = {<principal1>, <principal2>, …,
                <nonce1>, <nonce2>, …,
                <key1>, <key2>, …,
                <msg1>, <msg2>, …};
```

The next step is to define the general form of message shared by all principals and the channel to send messages. The message can consist of an encrypted part and unencrypted part, where each part is composed of 'mtype' enumeration types.

```
typedef message = {<key>, <mtype>, <mtype>, …};
```

Once the message is declared, the channel for communication may be declared. This is made possible with the keyword *chan* as shown below.

```
chan network = [0] of {…};
```

An index '*[0]'* indicates a synchronous channel, i.e. there is no buffer for the messages. This indicates that there can be at most one message in the channel at any given time. A state can be expressed in Promela by using a label as "STATE-1:", which is similar to a label in the C programming language.

Promela allows switching to a specific label within procedural code by using the *goto* keyword. The following statement shows how to use *goto* to move a label (*STATE-2* in this case).

goto STATE-2;

The symbols '!' and '?' are used for sending and receiving messages respectively. For example, the following Promela code is for receiving a message ("data") from the *network* channel

```
network ? data;
```

The following Promela code is for sending a message ("data") through the *network* channel

```
network ! data;
```

6.2.2 Directions for writing Promela code from Control-Flow State Machine

Follow the directions below to write Promela code for a principal from its control-flow state machine.

- Start writing Promela code by declaring an *mtype,* defining the message and declaring the channel of communication.
- Represent each state within a separate label. For example, use the label "START:" to represent the start state.
- Begin with the "START:" state label and use the *goto* keyword to move control from one state to another, until the final or end state is reached. Use *goto* to transfer control from one state label, say A to another, say B only if there is a transition from state A to state B in the control-flow state machine.
- If a transition in the control-flow state machine from state A to state B is guarded by a condition then enclose the *goto* within an *if-else* Promela construct.

- Within each state label, write the action that is performed when the principal is in that state. If the action is guarded by a condition then enclose it within an *if-else* Promela construct.

Following the above steps, it is straightforward to write Promela code of a principal from its control-flow state machine. Given a Promela model it is possible to verify whether or not a property holds for the model. This can be accomplished with the model checker SPIN [Holzmann, G. J., 1997]. The property may be specified in linear temporal logic (LTL). If the property does not hold true for the Promela model then SPIN provides a counter example.

CHAPTER 7

CASE STUDY – THE NSPK PROTOCOL

We use a Java implementation of the Needham Schroeder Public Key [Clarke, E.M. and Jacob, J., 1997] protocol as a case study to demonstrate our framework. Section 4.1 describes the steps that need be followed in order to apply the framework. We step through all of those steps and explain how to apply them for the NSPK protocol.

Step 1: *Identify the use case and generate sequence diagram for the use case*

We choose the authentication use case of the NSPK protocol. This authentication process has seven steps, which are outlined below. In these steps, *A* and *B* represent *initiator* and *responder* roles respectively. *S* is the server or the trusted third party. *Na* and *Nb* are nonces of *A* and *B* respectively. *Ka* and *Kb* are public keys of *A* and *B* respectively.

(i) A → S : A, B
(ii) S → A : {Kb, B}Ks
(iii) A → B : {Na, A}Kb
(iv) B → S : B,A
(v) S → B : {Ka, A}Ks
(vi) B → A : {Na, Nb}Ka
(vii) A → B : {Nb}Kb

- A → B : {Na, A}Kb
- B → A : {Na, Nb}Ka
- A → B : {Nb}Kb

Figure 7.1 A = Initiator, B = Responder, S = Trusted Third Party

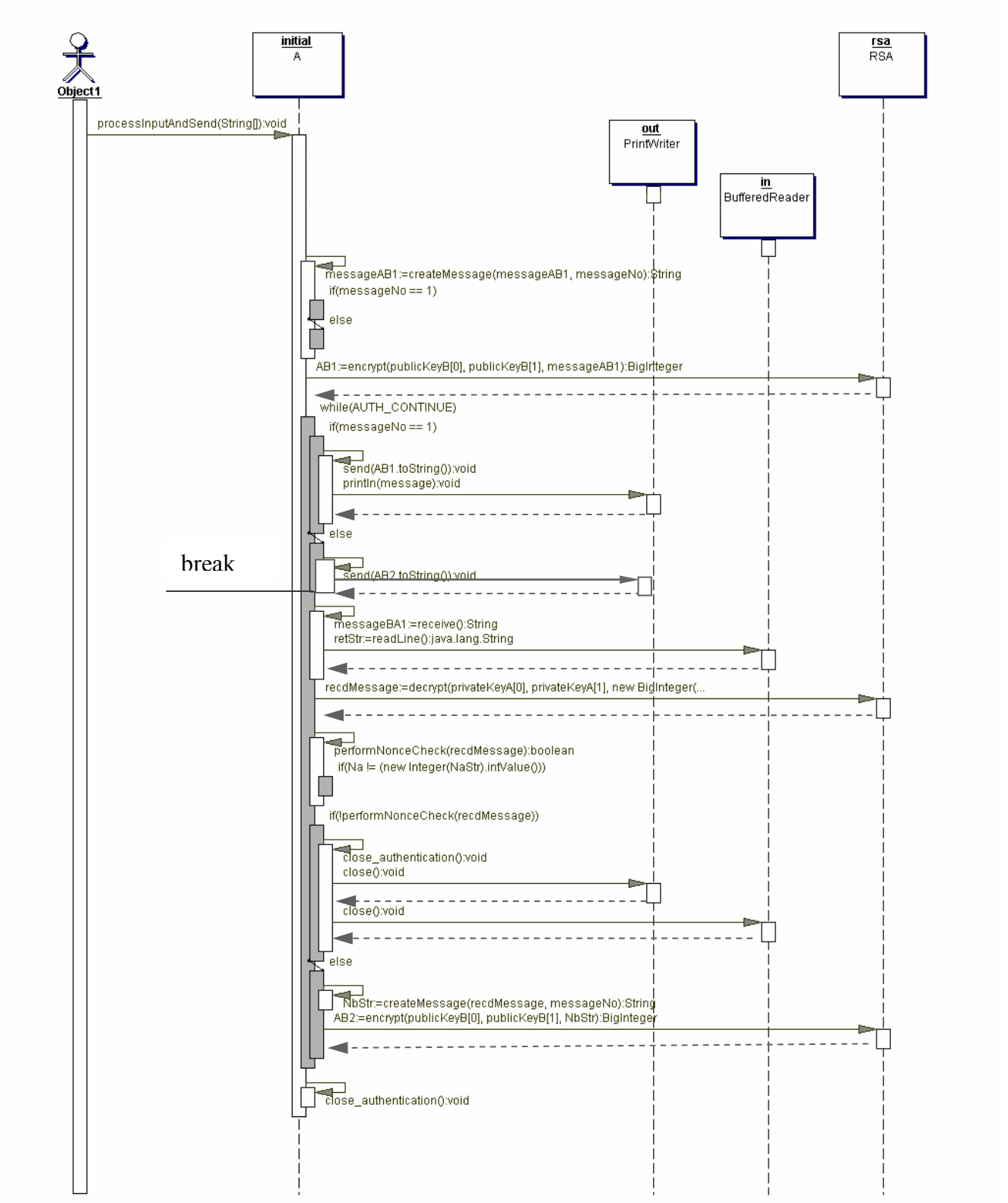


Figure 7.2 Initiator sequence diagram (NSPK protocol) for the authentication use case

As in similar publications, we assume that the communication of all principals with the server or trusted third party is secure. Therefore, we omit all steps that involve the server. This reduces the number of steps to three, as shown on the right hand side of figure 7.1.

The initiator encrypts its nonce, *Na* and identity, *A* with the public key of *B* and sends it to the responder. Once the responder receives the message, it decrypts this message and creates its own nonce, *Nb*. Then it sends the concatenation of *Na* with *Nb* encrypted with the public key of the initiator. After the initiator receives this message from the responder, it decrypts the message and verifies its nonce *Na*. The initiator is now sure that it is communicating with the correct principal. It sends back nonce *Nb* encrypted with the public key of the responder. This ends the initiator's part of the authentication process. The responder decrypts the second message from the initiator and verifies the nonce *Nb* to ensure that it is communicating with the correct principal. This ends the responder's part of the authentication process. We employ Together Control Center 5.0.1 (herein referred to as Together) to generate sequence diagrams of the initiator and responder. Together enables us to reverse engineer source code into sequence diagram for a given use case. The sequence diagram of the initiator for the authentication use case is shown in figure 7.2.

Step 2: *Generate sequence diagram for each principal and exclude objects from the sequence diagram that are not directly related to the pertinent use case.*

Appendix A shows the sequence diagram of the initiator of the NSPK protocol for the authentication use case. We excluded objects not directly related to the pertinent use case and obtained the sequence diagram shown in figure 7.2.

Step 3: *Create control-flow state machine for each participating principal from its sequence diagram*

We use rules I and II in section 6.1 and the directions outlined in section 6.1.1 to create the control-flow state machine for each principal. In the case of the NSPK protocol, it is necessary to create the control-flow state machines of the initiator and responder. Example 2 in section 6.1.1 demonstrates how to generate the initiator's control-flow state machine from its sequence diagram. Proceeding in a similar fashion we construct the control-flow state machine of the responder.

Step 4: *Write Promela code for each principal from its control-flow state machine*

Sections 6.2.1 and 6.2.2 provide an introduction to Promela and outline steps to write Promela code from a control-flow state machine. Following the steps, we make an *mtype* declaration to declare components of messages transferred between principals. The NSPK case study involves two principals, therefore,

```
mtype = {A, B, /* principals */
         na, nb, /* nonces */
         keyA, keyB, /* keys */
         msg1, msg2, msg3 /* keys */
         ok, nok /* flags */};
```

Next, we define the general form of message shared by all principals and the channel to send messages. For NSPK, we declare the encrypted part of the message and the channel of communication as below.

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

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

Proceeding further with the steps outlined in section 6.2.2, we begin writing Promela code with the *start* state and continue to the *end* state. The complete code for the initiator and responder is shown in appendix C.

Initially, our Promela model consists of exactly two communicating principals viz. the initiator and the responder. This enables us to verify the authenticity property in the absence of any form of intruder. Any property to be verified with the SPIN model checker should be specified in LTL (Linear temporal logic). SPIN produces a counter example if the LTL property does not hold true. A trail file is generated to show the counter example. If the property holds true, no such trail is generated. In the absence of any intruder the LTL property holds true, i.e., SPIN produces no counter example. Now, we introduce an intruder (written in Promela) that does some actions in a loop to successfully masquerade as an honest principal. Non-determinism is introduced in the initiator code to allow the initiator to choose between all principals (including the intruder). Similarly, we introduce non-determinism in the responder so that it can also select its partner among all principals. The authenticity property fails to hold true in such a situation. SPIN generates a counter example indicating that the property of

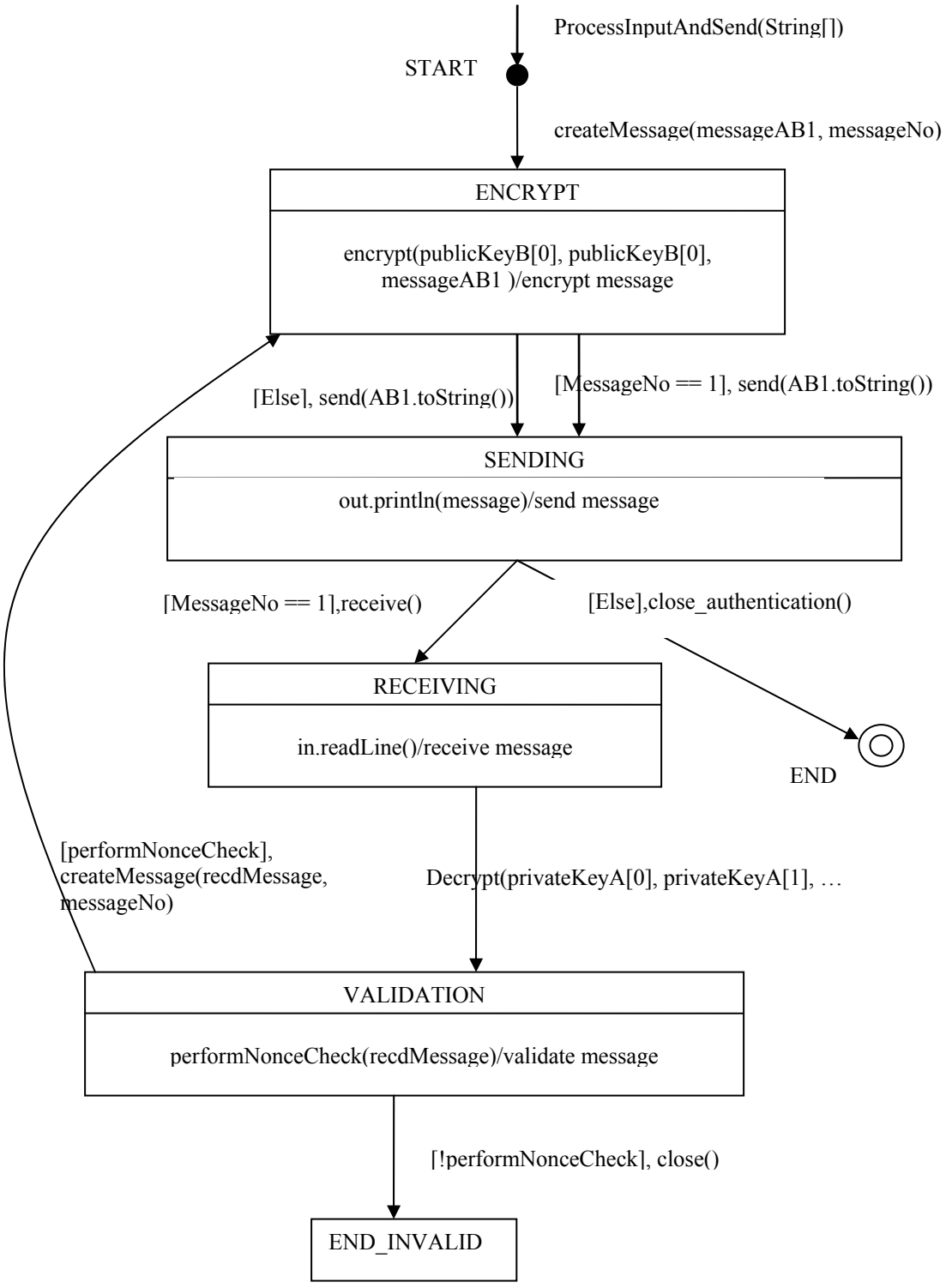


Figure 7.3 Control-flow state machine of initiator generated from the sequence diagram in figure 7.2

authenticity no longer holds true in the presence of an intruder. In this case, we chose the NSPK protocol, where we were aware that an attack undermining the authenticity property is possible. However, the same method can be used to check other properties. This process may also be useful to test the robustness of the implementation.

CHAPTER 8

CONCLUSION AND FUTURE WORK

Our work aims at model checking object oriented security protocol implementations. We reverse engineer the implementation to sequence diagrams, then to control-flow state machines and finally to a verifiable Promela model. This enables us to check properties that should be satisfied by the implementation.

Prior to our work, there has been related research with the aim of applying model checking to check UML diagrams and procedural code. For example, Lilius and Paltor [Paltor, I., Lilius, J., 1999] have defined operational semantics of UML state machines in Promela as a basis for vUML, a tool based on the SPIN model checker. However, vUML is designed to verify UML designs and is targeted towards the designer of object oriented systems. Our framework aims at detecting flaws in implementation by a reverse engineering process that follows a series of well defined steps. While it is essential to verify a design before its implementation is undertaken, we believe it is important to check if the implementation satisfies properties that it was intended to satisfy. This is especially critical for security protocols, where a seemingly benign defect in implementation may result in destructive effects compromising secrecy and other vital properties. Another interesting work that relates to our research is aimed at model checking properties that refer to the control flow of programs. H. Chen et al.

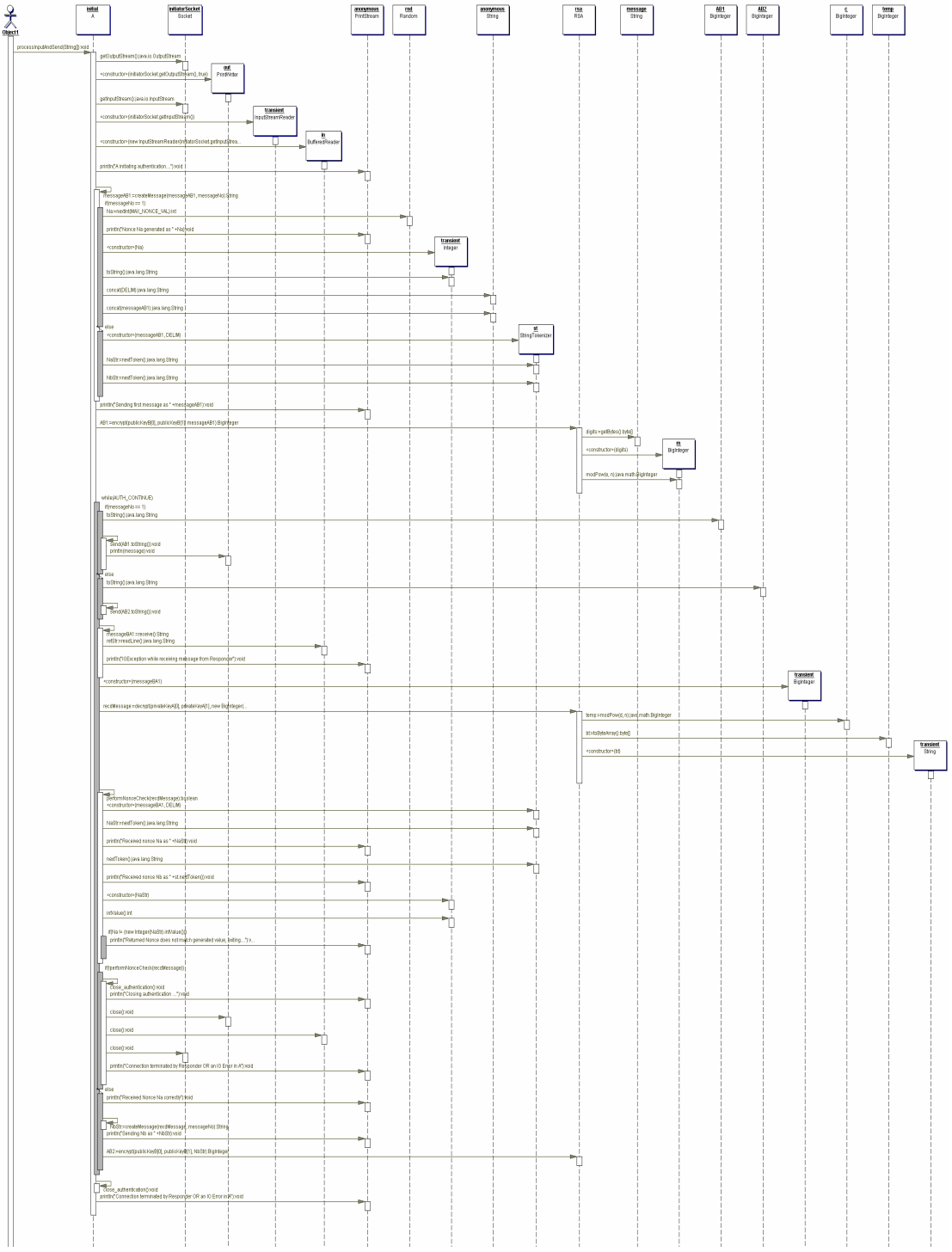
[H. Chen, D. Dean and D. Wagner 2004] have developed a tool (MOPS) for software model checking security-critical applications. However, MOPS is a static or compile-time analysis tool that searches the control flow graph of a program to check if any path may violate a safety property[H. Chen, D. Dean and D. Wagner 2004]. Performing a compile-time analysis of each participating principal without taking into perspective other principals and objects in the system may not yield useful enough information. Therefore it may not be entirely suitable for checking properties of security protocol implementations (or other such applications) wherein different principals (or entities) may run as distinct interacting programs. This requires an analysis of the system comprising of all participating principals and other pertinent objects in a way that resembles their mutual interaction at run-time. We achieve this in the following manner. Initially, we separate individual principals to generate their control-flow state machines, but finally we produce a single Promela model that allows *proctypes* of all principals to run concurrently. Then, we leverage features of the model checker SPIN[Holzmann, G. J., 1997] to check properties of this model.

We have identified major areas where we foresee scope for future research. However, we do not state that future investigation may be limited to those areas. The present framework may be enhanced to model check object oriented implementations other than those of security protocols. For example, distributed applications with object oriented implementations may be reverse engineered to generate control-flow state machines of participating entities. These control-flow state machines can be subsequently converted to Promela for final verification. This process may yield useful

information to identify bugs associated with the implementation. We also see potential for improvement in the final step of the framework. In the final step, we generate a Promela model from control-flow state machines of participating principals. However, we do not work towards efficiency or optimization of the Promela model. An intermediate step to optimize the Promela model before feeding it to the SPIN model checker could be introduced. Any step towards such optimization would be a challenging task, since the optimized Promela model should be generated without loss of information. In other words, the optimized model should be as representative of the control-flow state machines as the original model.

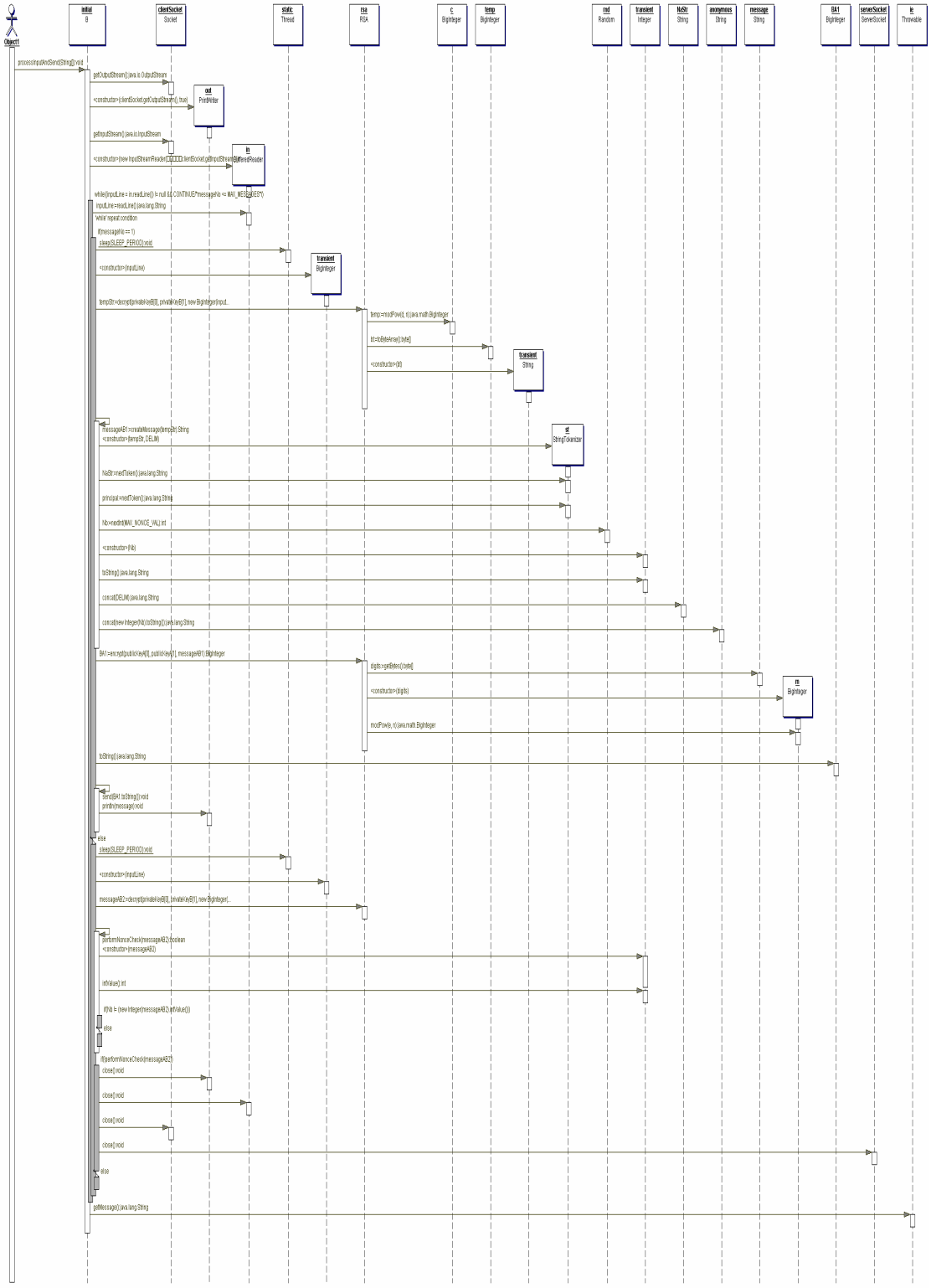
APPENDIX A

SEQUENCE DIAGRAM OF INITIATOR


Object1
initial
A
initiatorSocket
Socket
anonymous
PrintStream
rnd
Random
anonymous
String
rsa
RSA
message
String
AB1
BigInteger
AB2
BigInteger
N
BigInteger
temp
BigInteger
processInputAndSend(String[]):void
getOutputStream():java.io.OutputStream
out
PrintWriter
getInputStream():java.io.InputStream
transient
InputStreamReader
in
BufferedReader
println("A initiating authentication...."):void
transient
Integer
st
StringTokenizer
println("Sending first message as " +messageAB1):void
transient
BigInteger
transient
BigInteger
transient
String
while(AUTH_CONTINUE)
println("Connection terminated by Responder OR an IO Error in A"):void

APPENDIX B

SEQUENCE DIAGRAM OF RESPONDER


Object1
initial
ti
clientSocket
Socket
static
Thread
rsa
RSA
c
BigInteger
temp
BigInteger
rnd
Random
transient
Integer
NoStr
String
anonymous
String
message
String
BA1
BigInteger
serverSocket
ServerSocket
ie
Throwable
processInputAndSend(String[]):void
getOutputStream():java.io.OutputStream
<constructor>(clientSocket.getOutputStream(), true)
out
PrintWriter
getInputStream():java.io.InputStream
<constructor>(new InputStreamReader(CCCCCCclientSocket.getInputStream()))
in
BufferedReader
while((inputLine = in.readLine()) != null && CONTINUE&messageNo <= MAX_MESSAGES/KEY)
inputLine=readLine():java.lang.String
'while' repeat condition
if(messageNo == 1)
sleep(SLEEP_PERIOD):void
<constructor>(inputLine)
transient
BigInteger
tempStr=decrypt(privateKey[0], privateKey[1], new BigInteger(input...
temp=modPow(d, n):java.math.BigInteger
bt=toByteArray():byte[]
<constructor>(bt)
transient
String
messageA81=createMessage(tempStr):String
<constructor>(temp.Str, DELIM)
st
StringTokenizer
hisStr=nextToken():java.lang.String
principal=nextToken():java.lang.String
hb=nextInt(MAX_NOVICE_VAL):int
<constructor>(hb)
toString():java.lang.String
concat(DELIM):java.lang.String
concat(new Integer(hb).toString()):java.lang.String
BA1=encrypt(publicKey[0], publicKey[1], messageA81):BigInteger
digits=getBytes():byte[]
<constructor>(digits)
ttt
BigInteger
modPow(e, n):java.math.BigInteger
toString():java.lang.String
SendBA1.toString():void
print(mymessage):void
else
sleep(SLEEP_PERIOD):void
<constructor>(inputLine)
messageA82=decrypt(privateKey[0], privateKey[1], new BigInteger(...
performNonceCheck(messageA82):boolean
<constructor>(messageA82)
intValue():int
if(hb != (new Integer(messageA82).intValue()))
else
if(performNonceCheck(messageA82))
close():void
close():void
close():void
close():void
else
getMessage():java.lang.String

APPENDIX C

PROMELA CODE

```
mtype = {A_initiator, B_responder, intruder,
         nonceA, nonceB, nonceI,
         keyA, keyB, keyI,
         msg1, msg2, msg3,
         ok, nok};
typedef Crypt { mtype key, d1, d2; }
chan network = [0] of {mtype, mtype, Crypt};
mtype partnerA, partnerB;
mtype statusA, statusB;

active proctype A()
{
mtype  pkey, pnonce;
Crypt  data;
int msgNo = 1;
  if
  :: partnerA = B_responder; pkey = keyB;
  fi;
START:  goto ENCRYPT;
ENCRYPT:
        if
        :: (msgNo == 1) ->
            atomic{data.key = pkey; data.d1 = A_initiator; data.d2 = nonceA;}
                    goto SENDING;
        :: else ->
            atomic{data.key = pkey; data.d1 = pnonce; data.d2 = 0;}
                    goto SENDING;
        fi;
SENDING:
        if
        :: (msgNo == 1) ->
                network ! msg1, partnerA, data; goto RECEIVING;
        :: else ->
                network ! msg3, partnerA, data; goto END_A;
        fi;
RECEIVING:
        network ? msg2, A_initiator, data; goto VALIDATION;
VALIDATION:
        if
        :: (data.d1 == nonceA) ->
                pnonce = data.d2; msgNo++; goto ENCRYPT;
        :: else -> goto INVALID_A;
        fi;
END_A: statusA = ok;
INVALID_A: skip;
}

active proctype B()
{
mtype pkey, pnonce;
Crypt data;
int msgNo_B = 1;

START_B: goto RECEIVING_B;
RECEIVING_B:
        if
        :: (msgNo_B == 1) ->
            network ? msg1, B_responder, data; partnerB = data.d1; goto
ENCRYPT_B;
```

```
        :: else ->
                network ? msg3, B_responder, data; goto VALIDATION_B;
        fi;
VALIDATION_B:
        if
                :: (data.d1 == nonceB) -> goto END_B;
                :: else -> goto INVALID_B;
        fi;
ENCRYPT_B:
        atomic
        {
                if
                :: (partnerB == A_initiator) -> pkey = keyA;
                fi;
                pnonce = data.d2; data.key = pkey; data.d1 = pnonce; data.d2 =
nonceB;
        } goto SENDING_B;
SENDING_B:
        network ! msg2, partnerB, data; msgNo_B++; goto RECEIVING_B;
END_B: statusB = ok;
INVALID_B: skip;
}

active proctype Intruder()
{
  Crypt intercepted;
  mtype msg;
  do
  :: network ? msg, _, intercepted;
  :: (msg == msg1) -> network ! msg1, B_responder, intercepted;
  :: (msg == msg2) -> network ! msg2, A_initiator, intercepted;
  :: (msg == msg2) -> network ! msg3, B_responder, intercepted;
  od;
}
```

Note: To introduce the Intruder with the Initiator and Responder communicating processes, enter non-determinism into the Initiator and Responder code to reflect the Intruder. This modification being straightforward is not shown in this document. We derive inspiration from the tutorial provided by Merz [Merz, S., 2001] for some of the above code.

REFERENCES

Burrows, M., Abadi, M. & Needham, R., 1989. A logic of authentication. Technical Report 39. DEC Systems Research Center, February 1989.

Cawsey, A. Databases and Artificial Intelligence. Online. Available from http://www.cee.hw.ac.uk/~alison/ai3notes/ [accessed 16 January 2004].

Chen H., Dean D. & Wagner D., 2004. Model Checking One Million Lines of C Code, In Symposium on Network and Distributed Systems Security, 2004.

Clarke, E.M. & Jacob, J., 1997. A Survey of Authentication Protocol Literature: Version 1.0. 1997.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C., 2001. Introduction to Algorithms. Second Edition. The MIT Press, 2001, 525-549.

Dolev, D. & Yao, A., 1989. On the security of public key protocols. IEEE transactions on Information Theory. 29(2):198-208, March 1989.

Hoare, C. A. R., 1985. Communicating Sequential Processes. Prentice Hall, 1985.

Holzmann, G. J., 1997. The model checker Spin. IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997.

Lacey, T. & DeLoach, S., 2000. Automatic verification of multiagent conversations. Eleventh Annual Midwest Artificial Intelligence and Cognitive Science Conference, April 15-16, 2000.

Latella, D., Majzik, I. & Massink, M., 1999. Towards a formal operational semantics of UML state chart diagrams. In 3rd International Conference o f Formal

Methods for Open Object-Oriented Distributed Systems, Kluwer Academic Publishers, Boston, 1999.

Lowe, G., 1996. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In Tools and Algorithms for the Construction and Analysis of Systems, volume 1055 of Lecture Notes in Computer Science, pages 147-166. Springer-Verlag, 1996.

Marrero, W., Clarke, E. M. & Jha, S., 1997. Model checking for security protocols. Technical Report TR-CMU-CS-97-139, Carnegie Mellon University, 1997.

Meadows, C., 1992. Applying formal methods to the analysis of a key management protocol. Journal of Computer Security, 1:5-53, 1992.

Meadows, C., 1994. The NRL Protocol Analyzer. An overview. In The Proceedings of The Second International Conference on the Practical Applications of Prolog, 1994.

Merz, S., 2001. Model Checking: A Tutorial Overview. Modeling and Verification of Parallel Processes. LNCS 2067, Springer-Verlag, 2001.

Object Management Group. OMG Unified Modeling Language Specification, version 1.5 (formal/03-03-01). March 2003.

Paltor, I. & Lilius, J., 1999. vUML: A tool for verifying UML models. In R. J. Hall and E. Tyugu, editors, Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99. IEEE, 1999.

Schafer, T., Knapp, A. & Merz, S., 2001. Model Checking UML State Machines and Collaborations. Electronic Notes in Theoretical Computer Science 47, 2001.

Tenzer, J. & Stevens, P., 2003. Modeling Recursive Calls with UML state diagrams. Proceedings of FASE 2003. LNCS 2621, Springer-Verlag, 2003.

Woo, T. Y. C. & Lam, S. S., 1993. A semantic model for authentication protocols. In the Proceedings of the IEEE Symposium on Research in Security and Privacy, 1993.

BIOGRAPHICAL INFORMATION

Parikshit Singh was born on February 13, 1979 in Valsad, a town in western India. He received his Bachelor of Engineering in Chemical Engineering in June 2000. He began his studies for the Masters degree in Computer Science and Engineering in the Spring of 2001. Since then, he worked on a number of academic and industrial projects. He did his internship at Sabre Holdings in Southlake, Texas. The internship lasted for over 10 months. He began working as a Senior Systems Analyst for an Austin based company in the summer of 2004. Since then, he has been working in Austin. Parikshit's research interests include Software Engineering, Formal Methods, Distributed Systems and UML.