Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Analysis of a security protocol in μCRL

J. Pang

REPORT SEN-R0201 JANUARY 31, 2002

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2001, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Analysis of a Security Protocol in μCRL

Jun Pang
CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Email: Jun.Pang@cwi.nl

ABSTRACT

With the growth and commercialization of the Internet, the security of communication between computers becomes a crucial point. A variety of security protocols based on cryptographic primitives are used to establish secure communication over insecure open networks and distributed systems. Unfortunately, security protocols often contain serious errors. Formal verification can be used to obtain assurance that a protocol cannot be attacked by an intruder.

In this paper, we present how the process-algebraic language μCRL can be used to specify and analyze security protocols. To illustrate the feasibility of our approach, we analyze the Needham-Schroeder public-key protocol and reproduce the error found by Gavin Lowe [Low96a]. Two more definitions of authentication are studied. We give some remarks on our approach and discuss some possible directions for future work.

2000 Mathematics Subject Classification: 68N30 [Specification and verification]; 68Q60 [Model checking]; 68M12 [Network protocols]
1998 ACM Computing Classification System: D.2.1 [Requirements/Specifications]; D.2.4 [Model checking]; C.2.2 [Protocol verification]
Keywords and Phrases: μCRL, model checking, process algebra, protocol verification, the Needham-Schroeder public-key protocol
Note: This research is supported by the Dutch Technology Foundation STW under the project STW CES5008: Improving the quality of embedded systems by formal design and systematic testing.

1. Introduction

With the growth and commercialization of the Internet, the security of communication between computers becomes a crucial point. A variety of security protocols based on cryptographic primitives are used to establish secure communication over insecure open networks and distributed systems. Security protocols generally also provide authentication. Unfortunately, open networks and distributed systems are vulnerable to hostile intruders, and security protocols often contain serious errors.

Formal methods are mathematically based techniques for specifying and verifying software and hardware systems. Their mathematical underpinning allows formal methods to specify systems in a more precise, more consistent and non-ambiguous fashion. Also, formal methods allow to formally simulate, validate and reason about system models. This makes it possible to use formal description and verification to obtain assurance that a protocol cannot be attacked by an intruder.

Algebraic approaches to the study of concurrent systems focus on the manipulation of process descriptions. *Process algebras* such as CCS [Mil89], CSP [Ros98] and ACP [BK84, Fok00] are well suited for the study of elementary behavioral properties of distributed systems. However, when it comes to the study of more realistic systems, these languages turn out to lack the ability to handle data adequately.

In order to solve this problem, the language μCRL [GP95] has been developed. This language combines the process algebra ACP with equational *abstract data types* [LEW96]. This is done by parameterizing actions and process terms with data. To each μCRL specification there belongs a *transition system*, in which the states are process terms and the edges are labeled with actions.

The μCRL toolset [Wou01, BFG$^+$01] (see `http://www.cwi.nl/~mcrl`) supports the analysis and manipulation of μCRL specifications. A μCRL specification can be automatically transformed into a *linear process operator* (LPO). All other tools in the μCRL toolset use LPOs as their starting point. There are a number of tools that allow optimizations on the level of LPOs. The *instantiator* can generate a labeled transition system (LTS) from an LPO if this transition system consists of finitely many states. The LTS can be visualized, analyzed and minimized by the μCRL toolset in combination with the CÆSAR ALDÉBARAN DEVELOPMENT PACKAGE (CADP) [FGK$^+$97].

In this paper, we present how the process-algebraic language μCRL can be used to specify security protocols. The behavior of participants of a security protocol can be modeled by processes. The abstract data types in μCRL can be used to abstract from the complex cryptographic primitives and to model the knowledge databases of agents. We define some *security actions* to indicate the critical points in the protocol and these *security actions* can contain the information relevant to the properties we want to verify. The resulting specification is analyzed with the μCRL toolset and CADP. To illustrate the feasibility of our approach, we analyze a standard case study, the Needham-Schroeder public-key protocol, and reproduce a known error. We also study two other definitions of authentication and the property of secret. Our approach resembles the method used by Leduc and Germeau [LG00].

RELATED WORK

Over the past twenty years, many methods have been developed for mechanically analyzing security protocols. In this part, we restrict ourselves on those works using general verification methods to analyze security protocols.

Among these methods, model checking uses general purpose tools to treat a protocol as any other program and attempts to prove its correctness. LOTOS [Var90], state machines [Var89] and Petri nets [NT95] have been used for this purpose. The protocol and its correctness requirements are specified, and then they are investigated by using tools that are available for the formalism used.

The Failures Divergences Refinement Checker (FDR), a model checker for CSP, is used to analyze the Needham-Schroeder public-key protocol in [Low96a]. The agents of the protocol and an intruder are modeled as CSP processes. FDR takes an implementation and a specification of the protocol as input, and checks whether the implementation refines the specification. A security error was discovered. They adapted the protocol to remove this error and detected no further attacks. They also prove correctness of the protocol of arbitrary size. A compiler CASPER has been designed for the analysis of security protocols, which takes a description of a security protocol and produces a CSP description of the same protocol, suitable for checking using FDR. This approach has been applied to several protocols, and has produced a number of attacks [Low96b].

In [MMS97], a finite state exploration tool, Murϕ, is used to analyze several security protocols. A methodology is also presented there. In Murϕ, a process is modeled by a set of related rules. The parallel composition of two processes is modeled by a simple union of the rules of the two processes. The correctness properties can be specified as invariants in Murϕ. When the system reaches a state where some invariant does not hold, an error trace can be given. The Needham-Schroeder public-key protocol was analyzed. Murϕ was able to reproduce the error found by Gavin Lowe [Low96a] and no additional attack could be found.

How LOTOS can be used to specify security protocols is presented in [LG00]. Security properties can be modeled as safety properties and checked automatically by a model-based verification tool. This technique is illustrated on a concrete registration protocol. An error is found and corrected.

The model checking techniques can be used to find subtle errors in a system or protocol, but it cannot prove correctness when no error is found. Another approach is using theorem provers to state and prove the properties of security protocols.

Schneider developed a specific theory based on the CSP semantic framework [Sch97]. The language of CSP can be used to specify the security protocols precisely and this theory can reason about the properties formally. It has been successfully applied on analysis of a fair non-repudiation protocol

[Sch98].

[Pau98] describes how a security protocol can be inductively defined as sets of traces, where a trace is a list of communication events, and properties can be proved by rule induction, with machine support from the proof tool Isabelle/HOL. Both the original version and improved version of the Needham-Schroeder public-key protocol are studied in [Pau97]. The properties proved by Isabelle highlight the distinctions between these two versions. In this approach, the human effort required to analyze a protocol can be little, and yielding a proof script only takes few minutes to run. The author also points out that the inductive method performs a better analysis than model checking, but the cost of using is greater.

More thorough state-of-the-art surveys in the application of formal methods to the analysis of cryptographic protocols can be found in [Mea95, GNG97, GSG99, But99].

Structure of the paper

In Section 2, we give a brief overview of the μCRL specification language and its toolset. The Needham-Schroeder public-key protocol is introduced in Section 3. Then in Section 4, we give the algebraic specifiation of this protocol. The analysis process is presented in Section 5. We show that the improved Needham-Schroeder public-key protocol with a rational intruder meets all the requirements in Section 6. In Section 7, we give some remarks on our approach. And we draw some conclusions and discuss possible directions for future work in Section 8.

2. A short introduction to the μCRL language

The μCRL language is based on the *process algebra* ACP extended with a formal treatment of data. It is a language for specifying distributed systems and protocols in an algebraic style. The μCRL specification consists of two parts. One part specifies the data types, the second part specifies the processes. Processes are represented by process terms. Process terms consist of action names and process names with zero or more data parameters combined with process-algebraic operators: sequential composition ($\cdot$), non-deterministic choice ($+$), parallelism ($\|$) and communication ($|$), encapsulation (∂), hiding (τ), renaming (ρ) and recursive declarations. The data part contains equational specifications: one can declare sorts and functions working upon these sorts, and describe the meaning of these functions by equational axioms. A conditional expression ($_ \lhd _ \rhd _$) enables that data elements influence the course of a process, and an alternative quantification operator ($\sum$) provides the possibly infinite choice over some sort. The syntax and semantics of μCRL are given in [GP95].

The μCRL toolset is a collection of tools for analyzing and manipulating μCRL specifications. An overview of the toolset is given in Appendix I. μCRL and its toolset have been successfully used to analyze a wide range of protocols and distributed systems; recent case studies are described in the literature [AL99, AL01, GPW01, Use01].

3. Needham-Schroeder public-key protocol

The Needham-Schroeder public-key protocol [NS78] aims to provide mutual authentication between an initiator A and a responder B, after which some session involving the exchange of messages can take place. Both the initiator and the responder want to be assured of the identity of the other. The formal presentation of this protocol can be decomposed into several meaningful parts. As in [Low96a, MMS97], we present a simplified form of the protocol, which can be described by the following three steps.

In step 1, the initiator A seeks to establish a connection with the responder B by selecting a nonce N_a, and sending it along with its identity to B, both encrypted with B's public key K_b (*Message 1*). When B receives this message, it decrypts the message to obtain the knowledge of N_a. It then returns the nonce N_a with a new nonce N_b to A (*Message 2*). Both nonces are encrypted with A's public key. When A receives this message, it decrypts it and concludes that it is talking to B, since only B should be able to decrypt A's initial message containing nonce N_a; B is authenticated. A then returns the

Message 1.	$A \rightarrow$	$B : \{N_a, A\}_{Kb}$
Message 2.	$B \rightarrow$	$A : \{N_a, N_b\}_{Ka}$
Message 3.	$A \rightarrow$	$B : \{N_b\}_{Kb}$

Table 1: Needham-Schroeder Public-Key Protocol

nonce N_b to B, encrypted with B's public key (*Message 3*). In the same fashion, A is authenticated after step 3.

4. Algebraic specification

The starting point of verifying a protocol or a distributed system with μCRL is to get its algebraic specification. This generally involves the work to abstract the protocol by identifying the key behaviors of the protocol participants[1], and to understand the way that each participant will respond to any possible message.

4.1 Data types

As explained in Section 2, every μCRL specification has two parts, one defines the data types and the other gives the specification of behavior. The behavior part describes the exchange of messages. It does not consider the data transferred by these messages. The data part expresses which kinds of data are used and the operations on them. By this, we can abstract away the complex cryptographic primitives, such as encryption and decryption. Agents can access and achieve information from the encrypted message if and only if they have the right private key. In our specification, we model the knowledge database for agents as a set. The specification of a set in μCRL is defined as follows:

```
sort  DSet
func  ema: →DSet
      set: D×DSet→DSet
map   add: D×DSet→DSet
      remove: D×DSet→DSet
      test: D×DSet→Bool
      empty: DSet→Bool
      if: Bool×DSet×DSet→DSet
var   d,d': D
      s,s': DSet
rew   add(d,s) = if(test(d,s),s,set(d,s))
      remove(d,ema) = ema
      remove(d,set(d's)) = if(eq(d,d'),s,set(d',remove(d,s)))
      test(d,ema) = F
      test(d, set(d',s)) = if(eq(d,d'),T,test(d,s))
      empty(ema) = T
      empty(set(d,s)) = F
      if(T,s,s') = s
      if(F,s,s') = s'
```

DSet is defined as a set of elements, which are of type *D*. The constructors of this sort are *ema* and *set(d,s)*. *ema* stands for the empty set, while *set(d,s)* inserts an element d into a set s. The function *remove(d,s)* removes all occurrences of d in set s. Function *test(d,s)* tells us whether the element d is

[1] In the rest of this paper, we prefer to use 'agent' instead of 'protocol participant'.

in set *s*. The function *empty(s)* is used to judge whether a set is empty, or not. The choice between two sets is achieved by the function *if(b,s,s')*, where *b* ranges over *Bool*. It means that if *b* holds then *s* is selected, otherwise *s'*. And the function *add(d,s)* only inserts the element *d* into set *s* if *test(d,s)* fails. By this, we guarantee that an element can appear only once in a set.

To specify the Needham-Schroeder public-key protocol, we assume that there are processes of *Initiator*, *Responder* and *Intruder*. Each of them is assigned with a unique identity, which is specified by the sort *Identity*. We define the set of *Nonce* instead of random nonce generation. Each nonce can only be used once. In one instance of the protocol, there are three kinds of messages exchanged between an initiator and a responder. They are specified as sorts of *Message1*, *Message2* and *Message3*. The sets of these three types of message are defined as *Message1Set*, *Message2Set* and *Message3Set*. We use the identity of an agent as its public key and private key instead of explicit definitions. The message is encrypted by adding the identity of the receiver as a part of this message. (Here, the identity plays the role of a public key.) When an agent gets a message, it will try to decrypt the message by checking whether the key of the message equals its identity. (Here, the identity plays the role of a private key.) The details of the data part of the μCRL specification can be found in the Appendix II.

4.2 Process behavior

In this section, we focus on the behavioral part of the algebraic specification.

Adding an intruder To verify the correctness of a security protocol, normally we need to put the agents into a hostile environment by adding an intruder into the protocol. In μCRL, we can model an intruder as a process which can mimic attacks of a real-world intruder. We refer to a general set of modeling assumptions with wider applicability as the *Dolev-Yao model* [DY83]. In this model, the protocol intruder has the following capabilities:

1. It is a user of the computer network, the other agents may try to set up a session with it;
2. It can overhear any message exchanged among the agents;
3. It decrypts messages that are encrypted with its own public key and store parts of a message in its knowledge database;
4. It may replay an old message it has seen before, even if it cannot decrypt the encrypted part;
5. It can generate messages using any combination of its knowledge database and send them.

In μCRL specification, the intruder process typically consists of a set of variables that contains the intruder's knowledge and a set of actions that the intruder may take. The introduction of an intruder into a protocol will change the communications between agents as shown in Figure 1.

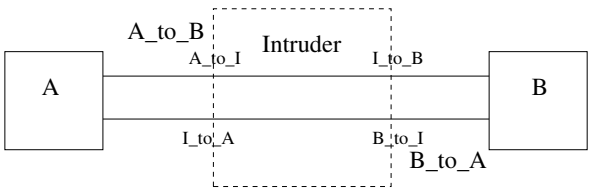

Figure 1: Agents with an intruder

Specifying the agents Agents in a security protocol are linked together by communication channels. These channels are insecure, meaning that they can be eavesdropped by an intruder. This means that, the communication between two agents A and B can be divided into communication between agent A and an intruder and communication between an intruder and agent B. By the encapsulation operator in μCRL, we can enforce the actions of the agents A, B and the intruder into communication.

Each of the two agents is defined as a process. As expressed in Figure 1, a message between an initiator and a responder can be eavesdropped by an intruder. For example, we define an action for an initiator of sending a *Message1* in the form of *s_mes1_i_to_t(ini,mes1)*, where *ini* is the identity of the sender, *mes1* is of data type *Message1*, *s_mes1* means sending a message of type *Message1*, and *i_to_t* indicates that this communication happens between an initiator and a intruder and the direction is from the initiator to the intruder. By this way, we can define the behavior of agents by actions parameterized with data. The three key steps of the protocol are modeled by the communications of actions.

comm	s_mes1_i_to_t	r_mes1_t_from_i	= c_mes1_it
	s_mes1_t_to_r	r_mes1_r_from_t	= c_mes1_tr
	s_mes2_r_to_t	r_mes2_t_from_r	= c_mes2_rt
	s_mes2_t_to_i	r_mes2_i_from_t	= c_mes2_ti
	s_mes3_i_to_t	r_mes3_t_from_i	= c_mes3_it
	s_mes3_t_to_r	r_mes3_r_from_t	= c_mes3_tr

The behavior of an initiator is modeled by a process parameterized with an identity *ini* and a nonce set *naset*, which starts the protocol by sending an initial message *Message1* to some responder. The action *I_running(ini,res)* reports that the initiator has recently run the protocol with a responder with identity *res*. After that, it waits and checks the reply. If it can decrypt the message and succeed in checking the included nonce, it concludes that it completes a run with the responder by an action *I_commit* (This action can be parameterized with any data relevant to the properties we want to verify.), and replies by sending a message *Message3* (modeled by an action *s_mes3_i_to_t*), after which it removes the used nonce from the set; otherwise, it only removes the used nonce. This is specified in μCRL as follows:

Initiator(ini:Identity,naset:NonceSet)=
$\sum_{\text{res:Identity}} \sum_{\text{n:Nonce}}$ I_running(ini,res).
s_mes1_i_to_t(ini,mes1(res,n,ini)).
  $\sum_{\text{int:Identity}} \sum_{\text{m2:Message2}}$ r_mes2_i_from_t(ini,m2).
  (I_commit(ini,res).s_mes3_i_to_t(ini,mes3(res,getNonce2(m2))).
  Initiator(ini,remove(n,naset))
  $\lhd$ and(eq(getKey(m2),ini),eq(n,getNonce1(m2))) $\rhd$
  Initiator(ini,remove(n,naset)))
$\lhd$ and(not(isInitiator(res)), test(n,naset)) $\rhd$ δ

We present the responder, with an identity *res* and a nonce set *nbset*, by the process *Responder*. The responder is supposed to wait for *Message1*. It decrypts the message, obtains the nonce and returns it with its own nonce *nb* (sending a message *Message2*). It also will indicate that it has recently been running the protocol with an initiator by an action *R_running*. After that, it waits and checks the reply of a *Message3*. If it succeeds, it concludes that it is talking with the initiator by an action *R_commit*, and then it removes the used nonce from the set; otherwise, it only removes the old nonce.

Responder(res:Identity,nbset:NonceSet)=
$\sum_{\text{m1:Message1}}$ r_mes1_r_from_t(int,res,m1).

$\sum_{\text{n:Nonce}}$ R_running(getID(m1),res).
s_mes2_r_to_t(res,getId(m1),mes2(getId(m1),getNonce(m1),n)).
$\sum_{\text{m3:Message3}}$ r_mes3_r_from_t(int,res,m3).
(R_commit(getId(m1),res).Responder(res,remove(n,nbset))
$\lhd$ and(eq(getKey(m3),res),eq(getNonce(m3),n)) $\rhd$
Responder(res,remove(n,nbset)))
$\lhd$ eq(getKey(m1),res) $\rhd$ Responder(res,nbset)
$\lhd$ test(n,nbset) $\rhd$ δ

Modeling the intruder In our model, we consider an intruder with a unique identity. The state of an intruder is represented by the knowledge it has acquired. The process is parameterized by the three sets of messages *m1s*, *m2s*, *m3s* and a set of nonces *ncset*. The sets *m1s, m2s* and *m3s* store the old messages that it has overheard but it cannot decrypt. *ncset* contains the nonces of itself and the nonces of the initiators and responders that it has learned by decrypting the messages it has seen before. The leakage of a secret (e.g. nonce *na*) is represented by an action *T_leaking(na)* when the intruder knows a nonce by decrypting a message and stores the nonce into its set *ncset*. This process is defined in μCRL as follows:

Intruder(int:Identity,m1s:Message1Set,m2s:Message2Set,
m3s:Message3Set,ncset:NonceSet)=
$\sum_{\text{ini:Identity}} \sum_{\text{m1:Message1}}$ r_mes1_t_from_i(ini,m1).
(T_leaking(getNonce(m1)).
Intruder(int,m1,m2s,m3s,add(getNonce(m1),ncset))
$\lhd$ eq(getKey(m1),int) $\rhd$
Intruder(int,add(m1,m1s),m2s,m3s,ncset)) +
$\sum_{\text{res:Identity}} \sum_{\text{m2:Message2}}$ r_mes2_t_from_r(res,m2).
(T_leaking(getNonce1(m2)).T_leaking(getNonce2(m2)).
Intruder(int,m1s,m2s,m3s,add(getNonce2(m2),add(getNonce1(m2),ncset)))
$\lhd$ eq(getKey(m2),int) $\rhd$
Intruder(int,m1s,add(m2,m2s),m3s,ncset)) +
$\sum_{\text{ini:Identity}} \sum_{\text{m3:Message3}}$ r_mes3_t_from_i(ini,m3).
(T_leaking(getNonce(m3)).
Intruder(int,m1s,m2s,m3s,naset,add(getNonce(m3),ncset))
$\lhd$ eq(getKey(m3),int) $\rhd$
Intruder(int,m1s,add(m3,m2s),m3s,ncset)) +
$\sum_{\text{res:Identity}} \sum_{\text{m1:Message1}}$ s_mes1_t_to_r(res,m1).
Intruder(int,m1s,m2s,m3s,ncset)
$\lhd$ and(test(m1,m1s),isResponder(res)) $\rhd$ δ +
$\sum_{\text{res:Identity}} \sum_{\text{ini:Identity}} \sum_{\text{n:Nonce}}$ s_mes1_t_to_r(res,mes1(res,n,ini)).
Intruder(int,m1s,m2s,m3s,ncset)
$\lhd$ and(and(test(n,ncset),isResponder(res)),isInitiator(ini)) $\rhd$ δ +
$\sum_{\text{ini:Identity}} \sum_{\text{m2:Message2}}$ s_mes2_t_to_i(ini,m2).
Intruder(int,m1s,m2s,m3s,ncset)
$\lhd$ and(test(m2,m2s),isInitiator(ini)) $\rhd$ δ +
$\sum_{\text{ini:Identity}} \sum_{\text{n1:Nonce}} \sum_{\text{n2:Nonce}}$ s_mes3_t_to_i(ini,mes2(ini,n1,n2)).
Intruder(int,m1s,m2s,m3s,ncset)
$\lhd$ and(and(test(n1,ncset),test(n2,ncset)),isInitiator(ini)) $\rhd$ δ +
$\sum_{\text{res:Identity}} \sum_{\text{m3:Message3}}$ s_mes3_t_to_r(res,m3).
Intruder(int,m1s,m2s,m3s,ncset)
$\lhd$ and(test(m3,m3s),isResponder(res)) $\rhd$ δ +
$\sum_{\text{res:Identity}} \sum_{\text{n:Nonce}}$ s_mes3_t_to_r(res,mes3(res,n)).

Intruder(int,m1s,m2s,m3s,ncset)
$\lhd$ and(test(n,ncset),isResponder(res)) $\rhd$ δ

4.3 A small system
We define a small Needham-Schroeder public-key protocol with one initiator, one responder and one intruder:

Asmallsystem=∂_H (Initiator(a1,set(na,ema)) || Responder(b1,set(nb,ema))
|| Intruder(c1,ema1,ema2,ema3,set(nc,ema)))

Here,

$$H=_{def.} \left\{ \begin{array}{l} \text{s_mes1_i_to_t, r_mes1_t_from_i, s_mes1_t_to_r, r_mes1_r_from_t,} \\ \text{s_mes2_r_to_t, r_mes2_t_from_r, s_mes2_t_to_i, r_mes2_i_from_t,} \\ \text{s_mes3_i_to_t, r_mes3_t_from_i, s_mes3_t_to_r, r_mes3_r_from_t} \end{array} \right\}$$

Initially, both initiator and responder have only one nonce, the intruder's its knowledge of the world is empty, and it has a nonce *nc*.

5. Analysis process

Several approaches have been developed for analyzing security protocols. We take the *explicit intruder method* [DM99] as the basis of our analysis approach and adopt it with our verification experience based on μCRL and its toolset. The whole process may have this sequence of steps: specify the protocol; model the intruder; state the correctness properties and verify the protocol. This verification process is described in Figure 2.

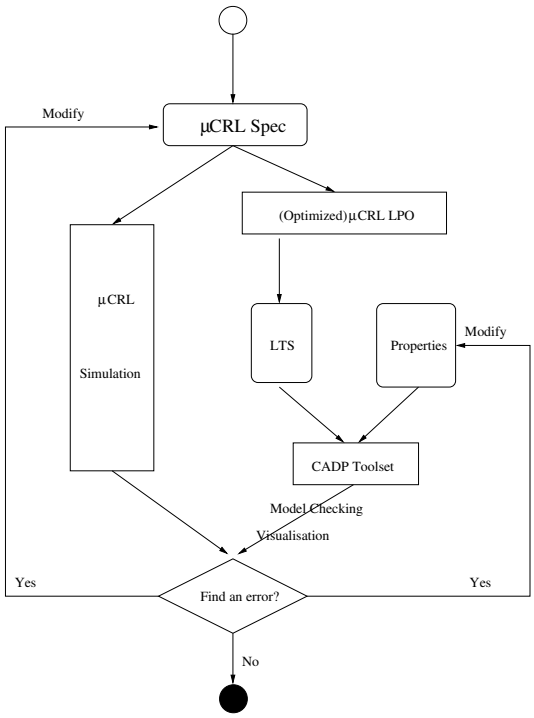

Figure 2: Verification (model checking) process in μCRL

The specification of the Needham-Schroeder public-key protocol has been discussed in Section 4. In this section, we will give the requirements of this protocol and present our verification result. Before that, we will show how to state the correctness properties and verify a protocol in our approach.

5.1 Stating the correctness properties

Model checking is an automatic technique to determine whether a state transition system satisfies certain requirements [CW96]. It has been successfully applied to a large number of communication protocols. In order to check whether a certain requirement holds, it should be expressed as a temporal logic formula first. A model checker searches the reachable states of a certain labeled transition system to determine whether this formula holds. If the model checker finds that the formula does not hold, it presents a fragment of the transition system that violates the requirement.

The temporal logic used as input language for EVALUATOR[2] is called *regular alternation-free μ-calculus*. It is an extension of the alternation-free fragment of the modal μ-calculus with action predicates and regular expressions over action sequences. The syntax of this logic is expressed in [MS00].

Similar to what Leduc and Germeau have done [LG00], we need to define some *security actions* (e.g. actions *I_running*, *I_commit* and *T_leaking* in the μCRL specification of Section 4.2) for agents to determine the critical points in the specification. These actions can be parameterized with any data relevant to the properties we want to model check. The parameters of these actions play an important role when model checking the properties. This is illustrated in the following discussion (see Section 5.3). By this, we can abstract away from the details of communication and only focus on these actions.

The correctness requirements of a security protocol are always related to authentication and rely on the fact that the intruder does not know some secret. In this paper, we study three kinds of authentication, and show that the Needham-Schroeder public-key protocol and its improved version can or cannot guarantee secure communication at different level. The property of authentication in our approach has the form "whenever agent A completes a run ..., then agent B has recently been running the protocol ...". The '...' part stands for the precise information relevant to different definitions of authentication. The fact that the intruder does not know some secret can be characterized as safety properties, stating that "nothing bad will ever happen".

5.2 Verifying the protocol

The μCRL specification can be automatically linearized into an LPO, and we generate the transition system from an (optimized) LPO. The generated LTS and the properties that we want to verify for this LTS, formulized in temporal logic, are treated as input to the model checker EVALUATOR. When we find a violated property, we analyze a number of diagnostic sequences produced by EVALUATOR or the simulator in the μCRL toolset in order to find out whether it is one among the following:

1. Property error: the property is incorrectly stated;
2. Specification error: the specification itself is not a correct representation of the protocol;
3. Protocol error: the error is with the protocol's design.

Most of the property errors we find are due to the fact that we define them too strongly. The specification errors can be corrected by modifying the corresponding parts in the specification, as long as we figure out the reasons. The protocol errors are most interesting to us. Much effort is spent on finding the real causes and the solutions to them. After repairing the detected errors, the whole process is repeated until no more errors can be found.

[2] A model checker among the the CÆSAR ALDÉBARAN DEVELOPMENT PACKAGE (CADP).

5.3 Requirements of the protocol

Since the Needham-Schroeder public-key protocol has been studied by many researchers, using different formal analysis approaches, the correctness requirements are already well known. We catch three kinds of definition for authentication from [Low96b] and formulate them in the *regular alternation-free μ-calculus*.

The mostly investigated definition of authentication is: Whenever an agent A completes a run apparently with B, then agent B has recently been running the protocol apparently with A. The attack on the Needham-Schroeder public-key protocol [Low96a] will break this. The properties of authentication are defined in *regular alternation-free μ-calculus* as follows:

A3: The *Responder* is correctly authenticated.

$$[(\neg\ \mathsf{R_running(a1,b1)})^* \cdot \mathsf{I_commit(a1,b1)}]\ \mathsf{F}$$

It says that if an execution sequence does not contain an action *R_running(a1,b1)*, then in the resulting state an initiator (with identity $a1$) cannot believe that it is talking with a responder (with identity $b1$).

A4: The *Initiator* is correctly authenticated.

$$[(\neg\ \mathsf{I_running(a1,b1)})^* \cdot \mathsf{R_commit(a1,b1)}]\ \mathsf{F}$$

It sates that if an execution sequence does not contain an action *I_running(a1,b1)*, then in the resulting state a responder (with identity $b1$) cannot believe that it completes a run with an initiator (with identity $a1$).

A stronger definition insists that the two agents agree on each other's state, there is a one-one relationship. It is expressed as follows: Whenever an agent A completes a run apparently with B, then agent B has recently been running the protocol apparently with A; and the two agents agree upon all data values used in the run. The *regular alternation-free μ-calculus* code is given below:

A1: The *Responder* is strongly authenticated.

$$[(\neg\ \mathsf{I_commit(a1,b1,na,nb)})^* \cdot \mathsf{I_commit(a1,b1,na,nb)}]$$
$$\mu\ \mathsf{X} \cdot (\langle \mathsf{T} \rangle \mathsf{T} \wedge [\neg\ \mathsf{R_commit(a1,b1,na,nb)}]\ \mathsf{X})$$

It states that after an action *I_commit(a1,b1,na,nb)*, the reachability of an action *R_commit(a1,b1,na,nb)* is inevitable. This is expressed using fixed point operators.

A2: The *Initiator* is strongly authenticated.

$$[(\neg\ \mathsf{I_commit(a1,b1,na,nb)})^* \cdot \mathsf{R_commit(a1,b1,na,nb)}]\ \mathsf{F}$$

A weaker definition is: Whenever an agent A completes a run apparently with B, then agent B has recently been running the protocol. Note that agent B may run the protocol with some other agent and never have heard of A.

A5: The *Responder* is weakly authenticated.

$$[(\neg\ \mathsf{R_running(b1)})^* \cdot \mathsf{I_commit(a1,b1)}]\ \mathsf{F}$$

A6: The *Initiator* is weakly authenticated.

[(¬ I_running(a1))* · R_commit(a1,b1)] F

The fact that the intruder does not know some secret can be expressed in *regular alternation-free* μ*-calculus* as follows:

S1: *na* is a secret.

[(¬ I_running(a1,c1))* · T_leaking(na)] F

S2: *nb* is a secret.

[(¬ R_running(c1,b1))* · T_leaking(nb)] F

5.4 Verification results

In our verification, we were able to discover the protocol error described in [Low96a]. Property A3, S1 and S2 were proved as *false* by the model checker. The diagnostic sequence produced by EVALUATOR is shown in Table 2. It means that the responder *b1* commits to a session with an initiator *a1* even though this initiator *a1* is not trying to establish a session with the responder *b1*. After fixing the protocol as Gavin Lowe did in [Low96a] (see Table 3), the improved protocol was shown to satisfy the properties (A3 and A4) of authentication and the properties S1 and S2.

initial state
I_running(a1,c1)
c_mes1_it(a1,mes1(c1,na,a1))
T_leaking(na)
c_mes1_tr(b1,mes1(b1,na,a1))
R_running(a1,b1)
c_mes2_rt(b1,mes2(a1,na,nb))
c_mes2_ti(a1,mes2(a1,na,nb))
I_commit(a1,c1)
c_mes3_it(a1,mes3(c1,nb))
T_leaking(nb)
c_mes3_tr(b1,mes3(b1,nb))
R_commit(a1,b1)
goal state

Table 2: One diagnostic sequence produced by EVALUATOR on property A3

Message 2. $B \rightarrow A : \{N_a, N_b, B\}_{Ka}$

Table 3: *Message2* in improved Needham-Schroeder Public-key Protocol

Table 4 summarizes the verification result on all the properties listed in Section 5.3. It shows that both the protocol and its improved version can satisfy the properties of weaker authentication (A5 and A6). The protocol satisfies neither A1 nor A2, while the improved version satisfies A2 but not A1.

	A1	A2	A3	A4	A5	A6	S1	S2
Improved NS-PKP	False	False	False	True	True	True	False	False
NS-PKP	False	True	True	True	True	True	True	True

Table 4: Verification result on the properties

6. The improved protocol with a rational intruder

When we verified the improved Needham-Schroeder public-key protocol on property A1, one diagnostic sequence produced by EVALUATOR was found as in Table 5. It says that the run of the protocol goes very well except for the last step, where the intruder sends one $Message3$ composed by its nonce, while the reponder is expecting a nonce of itself. It fails since as the intruder's behavior is nondeterministic, it can send any message based on its knowledge. But most of the messages may not be actually accepted by the agents. If we have an intruder who is rational, meaning that the intruder will use information that the other agents will actually accept, then we can show that the improved protocol can satisfy property A1. In [MMS97], it is said to be an efficient way to improve the running time and space of analysis by letting the intruder avoid the generation of useless messages.

initial state
I_running(a1,b1)
c_mes1_it(a1,mes1(b1,na,a1))
c_mes1_tr(b1,mes1(b1,na,a1))
R_running(a1,b1)
c_mes2_rt(b1,mes2(a1,na,nb,b1))
c_mes2_ti(a1,mes2(a1,na,nb,b1))
I_commit(a1,b1,na,nb)
c_mes3_it(a1,mes3(b1,nb))
c_mes3_tr(b1,mes3(b1,nc))
goal state

Table 5: One diagnostic sequence produced by EVALUATOR on property A1

To achieve a rational intruder, we only need to change the specification of the intruder slightly. We parameterized the process with another two new sets, *naset* and *nbset*. The nonces of the initiators that it has learned from the system are stored into *naset*; while the nonces of the responders are put into *nbset*. When the intruder fakes a $Message2$, it will use the nonces in *naset*, since an initiator is expecting a $Message2$ containing its nonce. When the intruder fakes a $Message3$, it will use the nonces in *nbset*, since a responder is expecting a $Message3$ with its nonce. After this, the improved protocol with a rational intruder was shown to meet property A1.

7. Discussion

This is the first time that μCRL and its toolset were used to verify security protocols. In this section, we make some remarks on our approach.

7.1 Do we model the intruder powerful enough?

In our analysis, we refer to a general model of the intruder. And initially, the knowledge databases of the intruder are empty. The question is: do we model the intruder powerful enough? Our answer is no. In the real world, the situation is more complex. When the intruder gets to know some private key,

it can achieve more information by decrypting the old messages it has seen before. We don't consider this in our specification of the intruder. In fact, this will lead to the problem of state explosion and make our approach infeasible.

7.2 Can we use the proof theory in μCRL?

The model checking technique can only find errors of a system or protocol, but it cannot prove there is no error in a system or protocol. In addition, it can only deal with finite state systems. Proving a system correct is more convincing. The language μCRL has also been the basis for the development of a proof theory (the *cones and foci method*) [GP94] that has enabled the formal verification of protocols and distributed systems in a precise and logical way. To prove a system correct, the axioms of μCRL are applied to a μCRL specification of that system (i.e. the implementation of the system) to show that the specification is branching bisimular [GW96] with a specification of the intended external behavior (i.e. the specification of the system). It can be written in μCRL notation as follows:

$$\tau_I(\partial_H(Implementation)) \simeq_b Specification$$

A prototype tool [Pol01] based on an extension of *binary decision diagrams* has been implemented, which can check invariants and the correct criteria associated with a state mapping between a specification and its implementation.

However, security protocols in practice tend to be very complicated, and it is difficult to identify the external behavior relevant to security properties. One could formulate the properties S1 and S2 as invariants, which one could then try to prove using the μCRL theorem prover. To achieve such a correct proof requires a lot of information at LPO level, this is time consuming and needs analysis experience with μCRL specifications. We leave this as future work.

7.3 Can we reduce the transition system?

The running time and space of analysis of a distributed system or a protocol can be decreased if we reduce the size of the generated transition systems. But usually we want to run the protocol against the most powerful and most nondeterministic intruder, this will cause the size of transition system to be very large.

The first way of reduction is that we can model the behavior of the intruder less nondeterministic, while the intruder's capabilities are not weakened. In our study on the Needham-Schroeder public-key protocol, we slightly changed the specificaiton of the intruder by allowing the intruder to send a message only when it receives a message of the same type. By this way, we could reduce the transition system to an extremely small size. In Section 6, a rational intruder is introduced. This is another efficient way to improve the running time and space of analysis by letting the intruder avoid the generation of useless messages. This is also metioned in [MMS97]. Experiments have shown that by adding more agents into the protocol, the size of transition systems can increase very fast. Hence, restricting the number of agents in the system can be another way to fight state explosion. But in the meanwhile, we need to be sure that if there is an attack on a system running this protocol, it can be found in a smaller system (as Gavin Lowe did in [Low96a]). In μCRL specification, the initial state of the agents and the intruder decides the size of the generated transition system. By carefully assigning values to the parameters, we can also reduce the transition systems.

8. Conclusion and future work

This paper presents a formal analysis approach to verify security protocols with μCRL and its toolset. We take the Needham-Schroeder public-key protocol as a case study. The algebraic specification and the analysis process are given in details. The known error can be reproduced by model checking. Furthermore, we also can study another two definitions of authentication and discuss the improved Needham-Schroeder public-key protocol with a rational intruder can meet all the requirements.

Compared with works using general purpose verification methods to analyze security protocols, our approach has achieved some success. There are two weaknesses in our approach, the second one is

also true with other approaches based on finite state exploration techniques:

1. The data types in μCRL language can only contain finitely many elements. Due to this, we cannot model the generation of nonces from an infinite domain in μCRL;
2. Modeling the intruder in full generality and adding more agents into the protocol can lead to the problem of state explosion and make our approach infeasible.

Encouraged by the work presented in this paper, we can list some further directions of research.

1. Try to specify and analyze other security protocols, and hope to discover new errors;
2. Apply our approach to more complicated e-commerce protocols, where security plays an important role, e.g. the electronic payment protocols;
3. Combine the approach with the design of new security protocols;
4. Applying techniques from process algebra and theorem proving for the formal analysis of security protocols.

ACKNOWLEDGEMENTS
Earlier versions of this paper were read by Wan Fokkink, his comments and subsequent discussions led to many improvements. Jaco van de Pol is thanked for teaching me how to use the prover in the μCRL toolset to check an invariant for this case study. Thanks also to Ade Azuat for helpful discussions. The author has benefited a lot from discussions with people at PAM (Process Algebra Meetings at CWI), especially Erik de Vink and Soujke Mauw.

References

[AL99] T. Arts and I.A. van Langevelde. How μCRL supported a smart redesign of a real-world protocol. In *Proceedings of FMICS'99*, pages 31–53, 1999.

[AL01] T. Arts and I.A. van Langevelde. Correct performance of transaction capabilities. In *Proceedings of ICACSD'2001*, pages 35–42. IEEE Computer Society, June 2001.

[BFG+01] S.C.C. Blom, W.J. Fokkink, J.F Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specification. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings 13th Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, 2001.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60:109–137, 1984.

[But99] L. Buttyan. Formal methods in the design of cryptographic protocols (state of the art). Technical Report SSC/1999/16, Swiss Federal Institute of Technology, 1999.

[CW96] E.M. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28:626–643, 1996.

[DM99] N.A. Durgin and J.C. Mitchell. Analysis of security protocols. In M. Broy and R. Steinbruggen, editors, *Calculational System Design*, pages 369–395. ISO Press, 1999.

[DY83] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29(2), 1983.

[FGK+97] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In R. Alur and T.A. Henzinger, editors, *Proceedings 8th Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer-Verlag, 1997.

[Fok00] W. J. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. Springer-Verlag, 2000.

[GL01] J.F. Groote and B. Lisser. Computer assisted manipulation of algebraic process specifications. Technical Report SEN-R0117, CWI, Amsterdam, 2001.

[GNG97] S. Gritzalis, N. Nikitakos, and P. Georgiadis. Formal methods for the analysis and design of cryptographic protocols: A state-of-the-art review. In *Proceedings of the IFIP Working Conference on Communications and Multimedia Security*, volume 3, pages 119–132, 1997.

[GP94] J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computing Series, pages 231–250. Springer-Verlag, 1994.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer-Verlag, 1995.

[GPW01] J.F. Groote, J. Pang, and A.G. Wouters. A balancing act: Analyzing a distributed lift system. In S. Gnesi and U. Ultes-Nitsche, editors, *Proceedings of 6he International Workshop on Formal Methods for Industrial Critical Systems*, pages 1–12, 2001. The extended version also appears as CWI technical report SEN-R0111, 2001.

[GSG99] S. Grizalis, D. Spinellis, and P. Georgiadis. Security protocols over open networks and distributed systems: Formal methods for their analysis, design, and verification. *Computer Communications*, 22:697–709, 1999.

[GW96] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43:555–600, 1996.

[LEW96] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley/Teubner, 1996.

[LG00] G. Leduc and F. Germeau. Verification of security protocols using Lotos-method and application. *Computer Communications*, 23:1089–1103, 2000.

[Low96a] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer-Verlag, 1996.

[Low96b] G. Lowe. Some new attacks upon security protocols. In *9th IEEE Computer Security Foundations Workshop*, pages 162–169. IEEE Press, 1996.

[Mea95] C. Meadows. Formal verification of cryptographic protocols: A survey. In *Advances in Cryptology - ASIACRYPT'94*, pages 135–150. Springer-Verlag, 1995.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MMS97] J.C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murϕ. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[MS00] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-cheching for regular alternation-free mu-calculus. In I. Schieferdecker and A. Rennoch, editors, *Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000 (Berlin, Germany)*, April 2000.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21:120–126, February 1978.

[NT95] B. Nieh and S. Tavares. Modelling and analysing cryptographic protocols using petri nets. In *Proceedings of AUSCRYPT'92*, 1995.

[Pau97] L.C. Paulson. Mechanized proofs of security protocols: Needham-Schroeder with public keys. Technical Report Report 413, Computer Laoratory, University of Cambridge, 1997.

[Pau98] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Computer Security*, 6:85–128, 1998.

[Pol01] J.C. van de Pol. A prover for the μCRL toolset with applications - version 1. Technical Report SEN-R0106, CWI, Amsterdam, 2001.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[Sch97] S. Schneider. Verifying authentication protocols with CSP. In *Proceedings of the IEEE Computer Security Foundations Workshop X*, pages 3–17. IEEE Computer Society Press, 1997.

[Sch98] S. Schneider. Verifying authentication protocols with CSP. In *Proceedings of the IEEE Computer Security Foundations Workshop XI*, pages 54–65. IEEE Computer Society Press, 1998.

[Use01] Y.S. Usenko. State space generation for the HAVi leader election protocol. *Science of Computer Programming*, 2001. To appear.

[Var89] V. Varadharajan. Verification of network security protocols. *Computers and Security*, 8(8):693–708, 1989.

[Var90] V. Varadharajan. Use of a formal description technique in the specification of authentication protocols. *Computer Standards and Interfaces*, pages 203–215, 1990.

[Wou01] A. G. Wouters. Manual for the μCRL toolset. Technical Report SEN-R0130, CWI, Amsterdam, 2001.

Appendix I
Overview of the μCRL Toolset

The μCRL toolset is a collection of tools for analyzing and manipulating μCRL specifications. A μCRL specification can be automatically transformed into a LPO. All other tools in the μCRL toolset use LPOs as their starting point. The μCRL toolset comprises five tools (*constelm, sumelm, parelm, structelm* and *rewr*) that target the automated simplification of LPOs while preserving bisimilarity [GL01]. These tools do not require the generation of the LTS belonging to an LPO, thus circumventing the ominous state explosion problem. The simplification tools are remarkably successful at simplifying the LPOs belonging to a number of existing protocols. In some cases these simplifications lead to a substantial reduction of the size of the corresponding LTS. The main relations between the tools is described in Figure I.1.

1. *mcrl* checks whether a specification in (timed) μCRL is well formed and linearizes certain μCRL specifications;
2. *msim* allows interactive simulation of a system described in μCRL;
3. *instantiator* generates a finite transition system from a linearized μCRL specification;
4. *pp* pretty prints a linearized μCRL specification;
5. *rewr* normalizes a linearized μCRL specification;
6. *constelm* removes from a linearized μCRL specification the data parameters that are constant throughout any run of the process;
7. *parelm* removes from a linearized μCRL specification the data parameters and sum variables that do not influence the behavior of the system;
8. *structelm* expands the composite data types of a linearized μCRL specification;
9. *sumelm* replaces in a linearized μCRL specification the sum variables that must be equal to a certain data term by that data term.

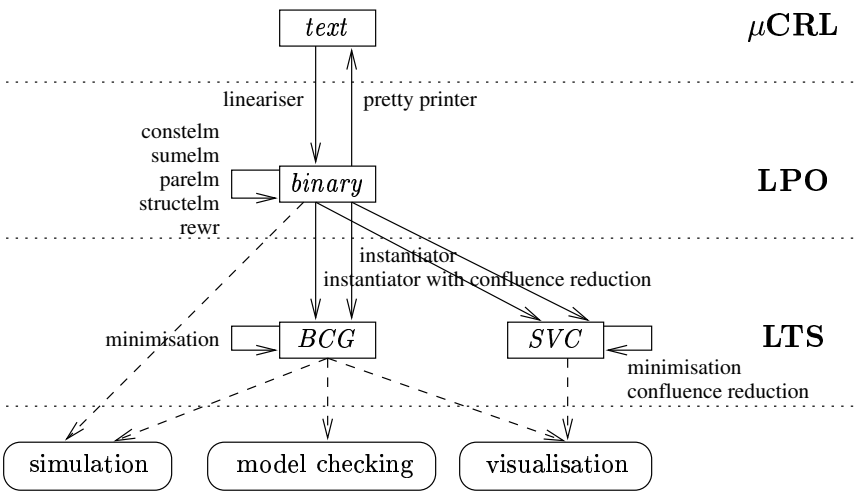

Figure I.1: The main components of the μCRL toolset

Appendix II
The μCRL specification of the Needham-Schroeder Public-key Protocol

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This is the specification of the NS public-key protocol in mCRL.
% Jun Pang, Aug 2001, CWI, Amsterdam, The Netherlands

% One Initiator
% One Responder
% One Intruder
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Bool
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Bool
func T,F:->Bool
map  if:Bool#Bool#Bool->Bool
     not:Bool->Bool
     and:Bool#Bool->Bool
     or:Bool#Bool->Bool
     eq:Bool#Bool->Bool
var  b,b':Bool
rew  if(T,b,b')=b     if(F,b,b')=b'
     not(T)=F         not(F)=T
     and(T,b)=b       and(F,b)=F     and(b,T)=b       and(b,F)=F
     or(T,b)=T        or(F,b)=b      or(b,T)=b        or(b,F)=b
     eq(b,b)=T        eq(T,F)=F      eq(F,T)=F
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Nonce
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Nonce
func na,nb,nc:->Nonce
map  eq:Nonce#Nonce->Bool
rew  eq(na,na)=T            eq(na,nb)=F            eq(na,nc)=F
     eq(nb,na)=F            eq(nb,nb)=T            eq(nb,nc)=F
     eq(nc,na)=F            eq(nc,nb)=F            eq(nc,nc)=T
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% NonceSet
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort NonceSet
func ema:->NonceSet
     set:Nonce#NonceSet->NonceSet
map  add:Nonce#NonceSet->NonceSet
     remove:Nonce#NonceSet->NonceSet
     test:Nonce#NonceSet->Bool
     empty:NonceSet->Bool
     if:Bool#NonceSet#NonceSet->NonceSet
var  n1,n1':Nonce
     s1,s1':NonceSet
rew  empty(ema)=T
     empty(set(n1,s1))=F
     test(n1,ema)=F
     test(n1,set(n1',s1))=if(eq(n1,n1'),T,test(n1,s1))
     add(n1,s1)=if(test(n1,s1),s1,set(n1,s1))
     remove(n1,ema)=ema
     remove(n1,set(n1',s1))=if(eq(n1,n1'),remove(n1,s1),
                                          set(n1',remove(n1,s1)))
     if(T,s1,s1')=s1
     if(F,s1,s1')=s1'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Message1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Message1
func mes1:Identity#Nonce#Identity->Message1 % Destination is the key
map  getNonce:Message1->Nonce
     getId:Message1->Identity
     getKey:Message1->Identity
     eq:Message1#Message1->Bool
var  n1,n1':Nonce
     dest1,id1,dest1',id1':Identity
rew  getNonce(mes1(dest1,n1,id1))=n1
     getId(mes1(dest1,n1,id1))=id1
     getKey(mes1(dest1,n1,id1))=dest1
     eq(mes1(dest1,n1,id1),mes1(dest1',n1',id1'))=
     and(eq(dest1,dest1'),and(eq(n1,n1'),eq(id1,id1')))
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Message2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Message2
func mes2:Identity#Nonce#Nonce->Message2
map  getNonce1:Message2->Nonce
     getNonce2:Message2->Nonce
     getKey:Message2->Identity
     eq:Message2#Message2->Bool
var  n21,n22,n21',n22':Nonce
     dest2,dest2':Identity
rew  getNonce1(mes2(dest2,n21,n22))=n21
     getNonce2(mes2(dest2,n21,n22))=n22
     getKey(mes2(dest2,n21,n22))=dest2
     eq(mes2(dest2,n21,n22),mes2(dest2',n21',n22'))=
     and(eq(dest2,dest2'),and(eq(n21,n21'),eq(n22,n22')))
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Message3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Message3
func mes3:Identity#Nonce->Message3
map  getNonce:Message3->Nonce
     getKey:Message3->Identity
     eq:Message3#Message3->Bool
var  n3,n3':Nonce
     dest3,dest3':Identity
rew  getNonce(mes3(dest3,n3))=n3
     getKey(mes3(dest3,n3))=dest3
     eq(mes3(dest3,n3),mes3(dest3',n3'))=
     and(eq(dest3,dest3'),eq(n3,n3') )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Message1Set
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Message1Set
func ema1:->Message1Set
     set1:Message1#Message1Set->Message1Set
map  add:Message1#Message1Set->Message1Set
     remove:Message1#Message1Set->Message1Set
     test:Message1#Message1Set->Bool
     empty:Message1Set->Bool
     if:Bool#Message1Set#Message1Set->Message1Set
var  m1,m1':Message1
     s1,s1':Message1Set
rew  empty(ema1)=T
     empty(set1(m1,s1))=F
     test(m1,ema1)=F
     test(m1,set1(m1',s1))=if(eq(m1,m1'),T,test(m1,s1))
     add(m1,s1)=if(test(m1,s1),s1,set1(m1,s1))
     remove(m1,ema1)=ema1
     remove(m1,set1(m1',s1))=if(eq(m1,m1'),remove(m1,s1),set1(m1',remove(m1,s1)))
     if(T,s1,s1')=s1
     if(F,s1,s1')=s1'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Message2Set
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Message2Set
func ema2:->Message2Set
     set2:Message2#Message2Set->Message2Set
map  add:Message2#Message2Set->Message2Set
     remove:Message2#Message2Set->Message2Set
     test:Message2#Message2Set->Bool
     empty:Message2Set->Bool
     if:Bool#Message2Set#Message2Set->Message2Set
var  m2,m2':Message2
     s2,s2':Message2Set
rew  empty(ema2)=T
     empty(set2(m2,s2))=F
     test(m2,ema2)=F
     test(m2,set2(m2',s2))=if(eq(m2,m2'),T,test(m2,s2))
     add(m2,s2)=if(test(m2,s2),s2,set2(m2,s2))
     remove(m2,ema2)=ema2
```

```
     remove(m2,set2(m2',s2))=if(eq(m2,m2'),remove(m2,s2),set2(m2',remove(m2,s2)))
     if(T,s2,s2')=s2
     if(F,s2,s2')=s2'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Message3Set
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Message3Set
func ema3:->Message3Set
     set3:Message3#Message3Set->Message3Set
map  add:Message3#Message3Set->Message3Set
     remove:Message3#Message3Set->Message3Set
     test:Message3#Message3Set->Bool
     empty:Message3Set->Bool
     if:Bool#Message3Set#Message3Set->Message3Set
var  m3,m3':Message3
     s3,s3':Message3Set
rew  empty(ema3)=T
     empty(set3(m3,s3))=F
     test(m3,ema3)=F
     test(m3,set3(m3',s3))=if(eq(m3,m3'),T,test(m3,s3))
     add(m3,s3)=if(test(m3,s3),s3,set3(m3,s3))
     remove(m3,ema3)=ema3
     remove(m3,set3(m3',s3))=if(eq(m3,m3'),remove(m3,s3),set3(m3',remove(m3,s3)))
     if(T,s3,s3')=s3
     if(F,s3,s3')=s3'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Identity
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Identity
func a1,b1,c1:->Identity
map  eq:Identity#Identity->Bool
     isInitiator:Identity->Bool
     isResponder:Identity->Bool
     isIntruder:Identity->Bool
rew  eq(a1,a1)=T            eq(a1,b1)=F            eq(a1,c1)=F
     eq(b1,a1)=F            eq(b1,b1)=T            eq(b1,c1)=F
     eq(c1,a1)=F            eq(c1,b1)=F            eq(c1,c1)=T
     isInitiator(a1)=T     isInitiator(b1)=F     isInitiator(c1)=F
     isResponder(a1)=F     isResponder(b1)=T     isResponder(c1)=F
     isIntruder(a1)=F      isIntruder(b1)=F      isIntruder(c1)=T
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Declaration of actions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act  s_mes1_i_to_t,r_mes1_t_from_i,c_mes1_it:Identity#Message1
     s_mes1_t_to_r,r_mes1_r_from_t,c_mes1_tr:Identity#Message1
     s_mes2_r_to_t,r_mes2_t_from_r,c_mes2_rt:Identity#Message2
     s_mes2_t_to_i,r_mes2_i_from_t,c_mes2_ti:Identity#Message2
     s_mes3_i_to_t,r_mes3_t_from_i,c_mes3_it:Identity#Message3
     s_mes3_t_to_r,r_mes3_r_from_t,c_mes3_tr:Identity#Message3
     I_commit,R_commit:Identity#Identity
     I_running,R_running:Identity#Identity
     T_leaking:Nonce
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initiator
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
proc Initiator(ini:Identity,naset:NonceSet)=
          sum(res:Identity,
              sum(n:Nonce,
                  I_running(ini,res).
                  s_mes1_i_to_t(ini,mes1(res,n,ini)).
                  sum(m2:Message2,
                      r_mes2_i_from_t(ini,m2).
                      (
                      I_commit(ini,res).
                      s_mes3_i_to_t(ini,mes3(res,getNonce2(m2))).
                      Initiator(ini,remove(na,naset))
                      <|and(eq(getKey(m2),ini),eq(n,getNonce1(m2))) |>
                      Initiator(ini,remove(na,naset)
                      )
                  )
              )
              <| and(not(isInitiator(res)), test(n,naset))  |>
              delta
             )
           )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Responder
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Responder(res:Identity,nbset:NonceSet)=
          sum(m1:Message1,
             r_mes1_r_from_t(res,m1).
             sum(n:Nonce,
                    (
                    R_running(getId(m1),res).
                    s_mes2_r_to_t(res,mes2(getId(m1),getNonce(m1),n)).
                    sum(m3:Message3,
                        r_mes3_r_from_t(res,m3).
                        (
                           R_commit(getId(m1),res).
                           Responder(res,remove(n,nbset) )
                           <| and(eq(getKey(m3),res),eq(getNonce(m3),n)) |>
                           Responder(res,remove(n,nbset) )
                        )
                    )
                    <|eq(getKey(m1),res)|>
                    Responder(res,nbset)
                    )
                 <| test(n,nbset) |>
                 delta
            )
        )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Intruder
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Intruder(int:Identity,
              m1s:Message1Set,
              m2s:Message2Set,
              m3s:Message3Set,
              ncset:NonceSet)=
         % Intruder gets message 1
```

```
        sum(ini:Identity,
            sum(m1:Message1,
                r_mes1_t_from_i(ini,m1).
                (  T_leaking(getNonce(m1)).
                   Intruder(int,m1s,m2s,m3s,add(getNonce(m1),ncset))
                   <| eq(getKey(m1),int) |>
                   Intruder(int,add(m1,m1s),m2s,m3s,ncset)
                )
            )
        )+
     % Intruder get messages 2
        sum(res:Identity,
            sum(m2:Message2,
                r_mes2_t_from_r(res,m2).
                (  T_leaking(getNonce1(m2)).
                   T_leaking(getNonce2(m2)).
                   Intruder(int,m1s,m2s,m3s,
                            add(getNonce1(m2),add(getNonce2(m2),ncset)))
                   <| eq(getKey(m2),int) |>
                   Intruder(int,m1s,add(m2,m2s),m3s,ncset)
                )
            )
        )+
    % Intruder get message 3
        sum(ini:Identity,
            sum(m3:Message3,
                r_mes3_t_from_i(ini,m3).
                (  T_leaking(getNonce(m3)).
                   Intruder(int,m1s,m2s,m3s,add(getNonce(m3),ncset))
                   <| eq(getKey(m3),int) |>
                   Intruder(int,m1s,m2s,add(m3,m3s),ncset)
                )
            )
        )+
    % Intruder fake message 1
       sum(res:Identity,
           sum(m1:Message1,
               s_mes1_t_to_r(res,m1).
               Intruder(int,m1s,m2s,m3s,ncset)
               <| and(test(m1,m1s),isResponder(res)) |> delta
           )
       )+
       sum(res:Identity,
           sum(ini:Identity,
               sum(n:Nonce,
                   s_mes1_t_to_r(res,mes1(res,n,ini) ).
                   Intruder(int,m1s,m2s,m3s,ncset)
                   <| and(and(test(n,ncset),isResponder(res)),
                      isInitiator(ini))|>delta
               )
           )
       )+
    % Intruder fake message2
       sum(ini:Identity,
           sum(m2:Message2,
```

```
                        s_mes2_t_to_i(ini,m2 ).
                        Intruder(int,m1s,m2s,m3s,ncset)
                        <| and(test(m2,m2s),isInitiator(ini)) |>delta
                    )
                )+
                sum(ini:Identity,
                    sum(n1:Nonce,
                        sum(n2:Nonce,
                            s_mes2_t_to_i(ini,mes2(ini,n1,n2)) .
                            Intruder(int,m1s,m2s,m3s,ncset)
                            <| and(and(test(n1,ncset),test(n2,ncset)),
                               isInitiator(ini)) |>delta
                        )
                    )
                )+
          % Intruder fake message3
                sum(res:Identity,
                    sum(m3:Message3,
                        s_mes3_t_to_r(res,m3).
                        Intruder(int,m1s,m2s,m3s,ncset)
                        <| and(test(m3,m3s),isResponder(res)) |>delta
                    )
                )+
                sum(res:Identity,
                    sum(n:Nonce,
                        s_mes3_t_to_r(res,mes3(res,n)).
                        Intruder(int,m1s,m2s,m3s,ncset)
                        <| and(test(n,ncset),isResponder(res)) |>delta
                    )
                )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Communication
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
comm  s_mes1_i_to_t | r_mes1_t_from_i = c_mes1_it
      s_mes1_t_to_r | r_mes1_r_from_t = c_mes1_tr
      s_mes2_r_to_t | r_mes2_t_from_r = c_mes2_rt
      s_mes2_t_to_i | r_mes2_i_from_t = c_mes2_ti
      s_mes3_i_to_t | r_mes3_t_from_i = c_mes3_it
      s_mes3_t_to_r | r_mes3_r_from_t = c_mes3_tr
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% A small system
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init encap(
     {s_mes1_i_to_t,r_mes1_t_from_i,
      s_mes1_t_to_r,r_mes1_r_from_t,
      s_mes2_r_to_t,r_mes2_t_from_r,
      s_mes2_t_to_i,r_mes2_i_from_t,
      s_mes3_i_to_t,r_mes3_t_from_i,
      s_mes3_t_to_r,r_mes3_r_from_t},
     Initiator(a1,set(na,ema) )||
     Responder(b1,set(nb,ema) )||
     Intruder(c1,ema1,ema2,ema3,set(nc,ema)) )
```