Modeling and Analyzing Protocols in Maude

Peter C. Ölveczky

University of Illinois at Urbana-Champaign

and

University of Oslo

Content

1. Motivation
   - why specify and analyze protocols?
   - specification formalisms and analysis techniques
2. High-level description of the Maude specification language and analysis tool
3. Introduction to Maude and rewriting logic
4. Overview of larger protocol analysis efforts in Maude
   - detailed examples: broadcast protocol, NSPK security protocol
   - overview: advanced new broadcast and multicast protocols for active networks
5. Experiences and concluding remarks

Content III

This talk is

- fairly introductory and tries to give an overview
- not meant to be exhaustive
- based on my own experiences in research and teaching
  - some of my work based on collaboration with very well-known protocol design groups in USA

Part 1. Motivation

“Protocols”

A protocol can mean different things

- this talk: “communication protocols”
- high-level description/specification of a distributed computer system
- not implementation
- think . . .
  - distributed database protocol
  - multicast/broadcast protocol
  - security protocol
  - wireless sensor network protocol
  - . . .

Motivation

Why analyze specifications and not just implementations?

- discover errors as early as possible during system development
- much cheaper to correct errors discovered early
- easier to discover errors at the specification level?
- if specification is correct, implementation is fairly easy (?)
- example: Voyager and Galileo spacecraft systems testing:
  - 197 critical defects identified — only 3 due to coding errors!
  - 50% errors/omissions in requirements
  - 25% design — 25% interface

Conclusion: need analysis early and along the system development process

Motivation (cont.)

Why analyze distributed systems protocols?

- Distributed systems particularly difficult to reason about
- Example: Needham-Schroeder public-key authentication protocol
  - well-known and well-used security protocol published in 1978
  - usually described in three lines:

$$\begin{array}{lll} \text{Message } 1. & A \rightarrow B: & A.B.\{N_a.A\}_{PK(B)} \\ \text{Message } 2. & B \rightarrow A: & B.A.\{N_a.N_b\}_{PK(A)} \\ \text{Message } 3. & A \rightarrow B: & A.B.\{N_b\}_{PK(B)} \end{array}$$

  - possible attack undiscovered for 18 years!
  - discovered during tool-based formal analysis by Lowe

Motivation (cont.)

Other end of spectrum:

- Distributed systems protocols large and ubiquitous
- AER/NCA multicast protocol for active networks:
  - under development in 1999/2000
  - 40 dense pages of specification
  - incorporates many different state-of-the-art techniques for multicast
- New and large protocols developed by Internet Engineering Task Force
- New sophisticated security protocols

Protocol Specification Formalism

Currently often informal specifications:

- prose/pseudo-code or UML/SDL/C-like specification formalism
  - informal $\Rightarrow$ no clear meaning $\Rightarrow$ cannot be tested
  - ambiguous
  - usually contains crucial implicit assumptions
- C/Java
  - too low level (and unreadable) for specification purposes
  - handling of concurrency? (threads)
  - analysis?

Examples of Specifications: AER/NCA protocol

```
  B.8 Processing a Buffered Data Packet

  This use case begins when a received data
packet is buffered. The following processing
is performed (seq is the data packet sequence
number):

    Call Use Case B.4 using seq
    if (application is using an asynchronous
        receive mode)
    {  Check for data packets to deliver to the
        application.    Update readNextSeq  } ...
```

Examples of Specifications: RBP protocol

Reliable broadcast protocol for static topologies:

1. ...
2. If node i processes the message from neighbor node q when it is already active for source j, it simply adds node q to the set SN_j^i.
3. When node i receives an acknowledgment from every neighbor node in the set $(N^i - SN_j^i)$, it sends an acknowledgment to every node in the set $(SN_j^i - s_j^i)$.
4. ...

Analysis of Informal Specifications

- Informal specification not analyzed
- Testing/simulation on testbeds/simulation tools
  - lots of work
  - usually “adds” assumptions inherent in the tool
  - limited analysis capabilities
- “Proofs” of properties
  - hard
  - without formal basis?
  - often wrong
    - “incorrect” proof of NSPK: “wrong” assumptions
    - incorrect proof of RBP: not all scenarios considered

Specification Formalism Criteria

Desired criteria for protocol specification formalisms:

- Clear, unambiguous meaning
- Easy to use and understand
- Easy to model systems
  - intuitive formalism where systems can be modeled at appropriate level of abstraction
  - ("programming concurrent systems only feasible if you don't have to worry about concurrency" (Langmyhr))
- Should support tool-based analysis (such as prototyping)

Specification Formalism Criteria (cont.)

General specification formalism in which many different (aspects of) systems can be naturally modeled

- no need to learn a new formalism/tool for each new (kind of) system
- different parts of a system can be modeled in the same formalism
- larger systems: data types, dynamic behavior, communication
- can use the formalism at many stages in the system development process (architectural specification to code)

or more restricted formalism?

- can be optimized for certain domains (e.g., model checkers for digital circuits)
- integration of different formalisms in a system difficult

Formal Specification Languages

- Mathematics/logic-based formalisms
- Specification defines a precise mathematical model of the system which can be subjected to (automated) mathematical analysis
  - what properties are consequences of the design?
- Usually more abstract (high-level) specification
  - more readable; only essential aspects are modeled
  - less work in developing a specification

Formal Methods Skepticism

Formal methods sometimes met with skepticism:

- Formal specification languages sometimes non-intuitive and mathematically difficult
  - e.g. ugly syntax
- Flaky/ever-changing/inefficient/user-unfriendly tool support
- Belief that formal methods is only about program/protocol verification, which is usually very hard
- Sometimes lack of support for useful programming/specification paradigms such as OO

Increasing Acceptance of Formal Methods

- Modern distributed system too hard to handle
  - need formal modeling and analysis
- Cost and criticality makes formal analysis worthwhile
- Success stories:
  - NSPK attack during formal analysis
    - followed by formal analysis of many security protocols
  - "Pentium bug" could be found by "routine" PVS analysis
  - advanced model checking techniques can analyze digital circuits designs with 500.000 gates
    - chip makers have in-house model checkers
  - . . . and many more . . .

Increasing Acceptance of Formal Methods (cont.)

- Need for verification/certification of critical code
  - mobile code
  - airplanes/rockets/power plants/. . .
  - electronic voting/commerce
- Better understanding of when/what formal analysis is feasible
- Success stories + criticality $\Rightarrow$ more money $\Rightarrow$ better tools and techniques $\Rightarrow$ more success stories $\Rightarrow$ $\cdots$
- Increasing maturity and performance of formal methods and tools
  - can handle larger systems
- Formal methods impact not only verification/analysis
  - high-level, “powerful,” advanced programming/modeling languages

What Formal Methods?

Potential requirements of formal methods:

- User-friendly and intuitive formalism and meaning
  - should be usable by engineers/protocol developers
- Support for object-oriented and modular specification and analysis
- Efficient and stable tool support for automated simulation/analysis
- General/flexible
  - different kinds/aspects/parts/levels of abstraction of a system modeled in the same formalism
  - e.g. static/dynamic properties, different kinds of communication

Goals contradictory: more restrictive formalism $\Rightarrow$ analysis can be fine-tuned/optimized

Part 2. Maude Overview

Maude

- Maude is a state-of-the-art specification language and analysis tool for concurrent systems
- Developed at SRI International and Universities of Illinois and Madrid under leadership of José Meseguer
- Supports
  - high-level modeling/specification of concurrent systems
  - a wide range of analysis techniques
- Declarative specification language focusses on generality and ease of specification
- Mature and high-quality tool

Maude (cont.)

- Intended to be used on serious applications!
  - focus on performance (millions of rewrites per second)
  - e.g., efficient implementation of integers not restricted to 32/64 bits
  - user-friendly syntax
  - intuitive and easy to understand specification language
  - no “extralogical hacks”
- Solid mathematical foundations
- Well-suited to specify distributed object-based systems
- Free at `http://maude.cs.uiuc.edu`
- Maude object-oriented specifications easy to understand for students/protocol developers without formal methods experience

Specification in Maude

- Maude specifications are rewriting logic theories
  - state space/data types defined by algebraic specifications (equations)
    - generality
    - state-of-the-art algebraic specification formalism
    - a "state" is represented by (an equivalence class of) a term
  - dynamic behavior defined by rewrite rules
- Simple and general logic (by Meseguer) in which many different kinds of systems can be "naturally" represented
- Concurrency in the logic
  - specify the atomic transition patterns
  - (specify a concurrent system without worrying about concurrency)

Specification in Maude (cont.)

- Tight integration of static and dynamic aspects of a system
- A behavior is a (possibly infinite) sequence of rewrite steps from an initial state
- User-friendly syntax: A/C-operators and "mix-fix" syntax
- Natural model of concurrent objects
- Abstract model of communication:
  - no fixed "communication primitives"
  - generality: different kinds of communication can be easily modeled (unicast/multicast/channels/ether/lossy/transmission delay/...)
- Efficient built-in modules for (unbounded) integers, rationals, reals, and strings

Analysis in Maude

- Maude specifications are executable (under reasonable conditions)
- Maude tool supports a range of increasingly stronger validation techniques:
  1. rewriting/prototyping/simulation:
     - simulate one possible behavior from one initial state
  2. exhaustive search:
     - search for reachable states in all possible behaviors from one initial state
       - search for “good” or “bad” states
       - search for deadlocks
       - breadth-first search
       - may search forever

3. linear temporal logic model checking:
   - check whether each behavior from one initial state satisfies a temporal property
   - example of properties:
     * "will one reach a desired state in all behaviors?"
     * "will a request eventually be followed by a response?"
     * "once the desired values are found, will they remain unchanged?"
   - performance comparable to SPIN
   - set of reachable states from initial state must be finite
     * if not, model check an abstraction of the system

- Maude tool does not yet explicitly support

4. narrowing analysis
   - analyze many/all behaviors from many initial states

5. Verification
   - prove a property for all possible behaviors from all allowed initial states
   - Clavel's inductive theorem prover for equational properties
   - mathematical model can be verified "by hand"
   - hard, only when lighter methods have been used

- User-defined strategies specific execution/analysis strategies
  - written in Maude using Maude's meta-programming facilities
  - specification and analysis strategy written in the same language

Part 3. Maude Introduction

Maude Modules

Maude modules are rewriting logic theories

- state space/data types/static parts modeled by equational specifications
- dynamic behavior modeled by rewrite rules

Equations

Equations appear everywhere:

$$e = mc^2$$

$$(x + y)^2 = x^2 + 2xy + y^2$$

$$n! = \text{if } n > 1 \text{ then } n \cdot (n - 1)! \text{ else } 1$$

- Usually used from left to right to simplify an expression until no equation can be "further applied":

$$2! = 2 \cdot 1! = 2 \cdot 1 \cdot 0! = 2 \cdot 1 \cdot 1$$

- Equations must be terminating and confluent

Equational Specifications

- Sorts are the “data types”
- Terms making up the state space defined by
  - function symbols
  - constants
- Equations define equalities between terms
  - some functions are constructors
  - others are defined functions; defined by equations
- Used left to right to compute the normal form (“value”) of an expression

Example: Factorial Function

Built-in module NAT defines sort Nat, the natural numbers, and usual functions; we define the factorial function fact:

```
fmod FACTORIAL is protecting NAT .
  op fact : Nat -> Nat .
  var N : Nat .
  eq fact(N) = if N > 1 then N * fact(sd(N, 1))
               else 1 fi .
endfm
```

The red(uce) command computes the "value" of a term:

```
Maude> red fact(321) .
result NzNat: 679269174457380047028785170185919186947307915378873794717504834800056690.....
```

Lists of Natural Numbers I

We define sort `List` of lists of natural numbers:

- constructors `nil` and `app`(end)
- one defined symbol `length`:

```
fmod LIST1 is protecting NAT .
  sort List .
  op nil : -> List [ctor] .
  op app : List Nat -> List [ctor] .
  op length : List -> Nat .
  var L : List .   var N : Nat .
  eq length(nil) = 0 .
  eq length(app(L, N)) = 1 + length(L) .
endfm
```

Mix-fix Syntax

- The list “1 2 3” is represented by the term

```
app(app(app(nil, 1), 2), 3)
```

- Syntax inconvenient for later matching in rules
- Can have “mix-fix” syntax in operators:

```
op _+_ : Nat Nat -> Nat .
op if_then_else_fi : Bool Nat Nat -> Nat .
op _! : Nat -> Nat .
```

“Mix-fix” Lists I

```
fmod LIST2 is protecting NAT .
  sort List .
  op nil : -> List [ctor] .
  op _++_ : List Nat -> List [ctor] .
  op length : List -> Nat .
  var L : List .  var N : Nat .
  eq length(nil) = 0 .
  eq length(L ++ N)  = 1 + length(L) .
endfm
```

- List “1 2 3” represented by `(((nil ++ 1) ++ 2) ++ 3)`
- Can still be nicer!

Associativity and Commutativity

A binary function symbol can be declared associative and/or commutative:

- Associative: `op f : s s -> s [assoc] .`
  - $\texttt{f}(x, \texttt{f}(y, z))$ treated exactly as $\texttt{f}(\texttt{f}(x, y), z)$
  - $\texttt{f}(\texttt{f}(\texttt{f}(v, x), y), z)$ can be written as $\texttt{f}(v, x, y, z)$
  - `assoc` essentially lists
- Commutative: `op g : s s -> s [comm] .`
  - $\texttt{g}(x, y)$ equals $\texttt{g}(y, x)$ for all x, y

Associativity and Commutativity

- Associative and commutative:

  `op h : s s -> s [assoc comm] .`

  - $\texttt{h}(x_1, x_2, x_3, x_4) = \texttt{h}(x_3, x_1, x_4, x_2)$
  - essentially multisets
  - convenient!

- Identity: `op f : s s -> [id: t] .`

  - $\texttt{f}(x, \texttt{t})$ is always equal to x.

- All reduction, rewriting modulo these attributes

Subsorts

A (sub)sort can be “included” in another sort:

```
subsort Nat < Int .  --- a nat is also an int
subsort Nat < List . --- a number is a list
subsort Nat < Mset . --- a number is a multiset
```

Lists III

```
fmod LIST is protecting NAT .
  sort List .      subsort Nat < List .
  op nil : -> List [ctor] .
  --- list concatenation:
  op _++_ : List List -> List [ctor assoc id: nil] .
  op length : List -> Nat .
  op rev : List -> List .   --- reverse a list
  var L : List .   var N : Nat .
  eq length(nil) = 0 .
  eq length(L ++ N) = 1 + length(L) .
  eq rev(nil) = nil .
  eq rev(L ++ N) = N ++ rev(L) .
endfm
```

Lists III (cont.)

- List “1 2 3” represented as `1 ++ 2 ++ 3`
- If we use ___ instead of _++_, then the list would be represented

  `1 2 3`

Dymanic Behavior: Rules

- Static: $1 + 1$ is 2, and 2 is $1 + 1$; `length(1 ++ 5)` is `2`
- Dynamic behavior: person 35 years old can become 36 years old, but person 36 years old cannot become 35
  - not equations, but (labeled) rules:

```
op person : String Nat -> Person [ctor] .
var X : String .  var N : Nat .

rl [birthDay] :
    person(X, N) => person(X, N + 1) .
```

Rewriting Logic

- Logic developed by José Meseguer
- A theory $\mathcal{R}$ is a tuple (Σ, E, R)
- The deduction rules of the logic defines which sequents hold: $\mathcal{R} \vdash [t]_E \longrightarrow [t']_E$ if state $[t]_E$ can rewrite to $[t']_E$ in zero or more rewrite steps
- Logic about concurrency: e.g., $[f(a, b)]_E$ rewrites to $[f(a', b')]_E$ in one step if
  - $[a]_E$ rewrites to $[a']_E$ in one step, and
  - $[b]_E$ rewrites to $[b']_E$ in one step
- $[$`person("Peter", 35)`$]_E \longrightarrow [$`person("Peter", 44)`$]_E$ holds in the example above

Objects

- An object can be represented as a term

  < O : C | att_1 : val_1, ..., att_n : val_n >

  - O is the object identifier of sort `Oid`
  - C is the class of the object
  - att_1 to att_n are the attributes of the object
  - val_1 to val_n are the current values

- Example:

```
< "Peter" : Person | age : 35, status : single >
```

- Maude syntax for class declarations:

```
class Person | age : Nat, status : Status .
```

Objects (cont.)

- Example rule for aging:

```
var X : String .  var N : Nat .

crl [birthDay] :
      < X : Person | age : N >   =>
         < X : Person | age : N + 1 >
      if N < 1000 .
```

- Attributes which don’t matter can be omitted in a rule

Configurations

A state (or configuration) of a distributed object system is seen a multiset of

- objects
- messages traveling between objects

```
sorts Object Msg Configuration .
subsorts  Object Msg < Configuration .
op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration [ass
```

A state can be e.g.

```
< "Peter" : Person | age : 35, status : single >
< "Ronaldo" : Person | age : 27, status : single >
< "Lizzie" : Person | age : 32, status : single >
```

Messages

Messages are defined as terms of sort `Msg`:

```
msgs  marry? yes no : Oid Oid -> Msg .
```

Send a `marry?` message:

```
crl [propose] :
    < X : Person | age : N, status : single >
  =>  < X : Person | status : waitFor(Y) >
      marry?(Y, X)
   if N > 15 .
```

Messages (cont.)

Two rules read a `marry?` request:

```
crl [accept] :
    marry?(Y, X)
    < Y : Person | age : N, status : single >
  =>  < Y : Person | status : engaged(X) >
      yes(X, Y)
    if N > 15 .

rl [reject] :
    marry?(Y, X)  < Y : Person | >
  =>  < Y : Person | >  no(X, Y) .
```

Messages (cont.)

Rules for accepting the response:

```
rl [yes] :
   yes(X, Y)
   < X : Person | status : waitFor(Y) >  =>
      < X : Person | status : engaged(Y) > .

rl [no] :
   no(X, Y)
   < X : Person | status : waitFor(Y) >  =>
     < X : Person | status : single > .
```

Message Transmission

Message transmission modeled abstractly sincew we have multisets:

```
< "Peter" : Person | ..., status : waitFor("Lizzie") >
marry?("Lizzie", "Peter")
< "Ronaldo" : Person | ..., status : single >
< "Lizzie" : Person | ..., status : single >
```

is exactly the same as

```
< "Peter" : Person | ..., status : waitFor("Lizzie") >
< "Ronaldo" : Person | ..., status : single >
marry?("Lizzie", "Peter")
< "Lizzie" : Person | ..., status : single >
```

which rewrites to

```
< "Peter" : Person | ..., status : waitFor("Lizzie") >
< "Ronaldo" : Person | ..., status : single >
< "Lizzie" : Person | ..., status : engaged("Peter") >
yes("Peter", "Lizzie")
```

Communication

- Example of asynchronous communication
  1. `"Peter"` sends `marry?` message
  2. `"Lizzie"` sends `yes` (or `no`) response
  3. `"Peter"` rads response

  things can happen in-between
- Synchronous communication: both objects together:

```
rl [wedding] :
   < X : Person | status : engaged(Y) >
   < Y : Person | status : engaged(X) >
    =>
      < X : Person | status : married(Y) >
      < Y : Person | status : married(X) > .
```

Modeling Multicast

- Multicast: send message to a set of objects
- Multisets of object identifiers:

```
sort OidSet .    subsort Oid < OidSet .
op none : -> OidSet [ctor] .
op _;_ : OidSet OidSet -> OidSet
                  [ctor assoc comm id: none] .
```

- A message `multiM(OS, O)` can be defined to be equal to a message `M(O', O)` for each `O'` in the `OidSet` `OS`:

```
eq multiM(none, O) = none .
eq multiM(OS ; O', O) =
       M(O', O)  multiM(OS, O) .
```

- Many other forms of communication can be easily defined

Analysis in Maude

- Rewriting: `rew [`n`]` t `.`
- Search for certain reachable states:
  - `search [`n`]` t `=>*` t' `such that` $condition$ `.`
  - `search [`n`]` t `=>!` t' `such that` $condition$ `.`
    - t is a ground term (the initial state)
    - t' is a term, with variables, to search for
    - $cond$ is a condition on the variables
    - `=>!` searches for deadlocks/final states
    - `[`n`]` searches for n solutions/simulates n rewrite steps
- Temporal logic model checking
- User-defined analysis strategies

Part 4. Protocol Analysis in Maude: Examples

A very simple broadcast protocol

A Trivial Broadcast Protocol

- Topology: graph — each node knows its neighbors
- Aim: Broadcast: All nodes reachable from initiator node should see the message
- Protocol:
  1. initiator sends message to its neighbors
  2. when a node first receives a message, it stores the message, and forwards it to its other neighbors
  3. when a node sees a message it has seen, it ignores the message

Broadcast: Data Types

```
(omod BROADCAST is  protecting STRING .
  subsort String < Oid .

  msgs m1 m2 m3 m4 m5 : -> Msg .
  msg broadcast : Msg Oid -> Msg .

  sort OidSet .  subsort Oid < OidSet .
  op none : -> OidSet [ctor] .
  op _;_ : OidSet OidSet -> OidSet
              [ctor assoc comm id: none] .
```

- `m1` to `m5` are the possible message contents to be broadcast
- `broadcast` starts the protocol

The Node Class

```
class Node | neighbors : OidSet,
             msgRead : Configuration .
```

- `msgRead` is either `none` or contains the message to be stored
- `neighbors` is (object identifiers of) the neighbors

The Message Wrappers

```
vars O O' : Oid .  var OS : OidSet .
var M : Msg .

msg msg_from_to_ : Msg Oid Oid -> Msg .
op multimsg_from_to_ : Msg Oid OidSet ->
                       Configuration [ctor] .

eq multimsg M from O to none = none .
eq multimsg M from O to O' ; OS =
      (msg M from O to O')
      (multimsg M from O to OS) .
```

- definition of `multimsg`

Broadcast Example: Rules

1. Initiator sends the message M to be broadcast to all its neighbors:

```
rl [startBroadcast] :
   broadcast(M, O)
   < O : Node | neighbors : OS,
                msgRead : none >
  =>
   < O : Node | msgRead : M >
   multimsg M from O to OS .
```

Broadcast Example: Rules (cont.)

2. A node receives message `M` for first time (`msgRead` is `none`), and forwards it:

```
rl [readAndForward] :
   (msg M from O to O')
   < O' : Node | neighbors : O ; OS,
                 msgRead : none >
  =>
   < O' : Node | msgRead : M >
   multimsg M from O' to OS .
```

Broadcast Example: Rules (cont.)

3. If a message M is already seen, it is ignored:

```
rl [readSeenMsg] :
   (msg M from O to O')
   < O' : Node | neighbors : OS,
                 msgRead : M >
  =>
   < O' : Node | > .
```

Broadcast: Define Initial State

We define the following initial state to test the system:

```
op initState : -> Configuration .
eq initState =
     broadcast(m4, "b")
     < "a" : Node | neighbors : "b" ; "e",
                    msgRead : none >
     < "b" : Node | neighbors : "a" ; "d",
                    msgRead : none >
     < "c" : Node | neighbors : "d", msgRead : none >
     < "d" : Node | neighbors : "b" ; "c" ; "e",
                    msgRead : none >
     < "e" : Node | neighbors : "a" ; "d",
                    msgRead : none > .
```

Broadcast: Execute Initial State

- Simulate one bahvaior:

```
Maude> (rew initState .)

result Configuration :
< "a" : Node | msgRead : m4, neighbors :("b" ; "e")>
< "b" : Node | msgRead : m4, neighbors :("a" ; "d")>
< "c" : Node | msgRead : m4, neighbors : "d" >
< "d" : Node | msgRead : m4, neighbors :("b" ; "c" ; "
< "e" : Node | msgRead : m4, neighbors :("a" ; "d")>
```

- Is this promising?
- The rewrite can be traced
- All are all behaviors from `initState` OK?

Broadcast: Search

- Check all possible "results" from `initState`:

```
Maude> (search
            initState =>! C:Configuration .)
```

- Any state matches the variable `C:Configuration`
  - search returns all terminated states

Broadcast: Search (cont.)

The result of the search is

```
Solution 1
C:Configuration <- < "a" : Node | msgRead : m4, neighbors
< "b" : Node | msgRead : m4, neighbors :("a" ; "d")>
< "c" : Node | msgRead : m4, neighbors : "d" >
< "d" : Node | msgRead : m4, neighbors :("b" ; "c" ; "e"
< "e" : Node | msgRead : m4, neighbors :("a" ; "d")>

No more solutions.
```

- Result promising?
- Conclusion: trivial protocol does what it is supposed to do for `initState`, but is not analyzed for other initial states

Needhan-Schroeder public-key authentication protocol (NSPK)

NSPK

Needhan-Schroeder public-key authentication protocol (NSPK)

- Famous and often cited security/cryptographic protocol from 1978
- Goal: authenticate a connection in an unprotected network where intruders may eavesdrop and may fake messages
  - two agents must be sure that they are talking to each other
  - the Internet bank must know that it is in touch with Scrooge, and not with the bandits, and vice versa
  - field general must be sure he talks to Pentagon and not to al-Sadr, and vice versa

Public Key Cryptography

- Each agent A has public key $PK(A)$ and private key $PrvK(A)$
- All public keys are known to everyone
- Something encrypted with $PK(A)$ can only be decrypted with $PrvK(A)$, and vice versa
- Assumptions in crypto-protocols:
  - cannot guess correctly keys and randomly generated numbers
  - cannot decrypt encrypted message if decryption key not known
- Public-key cryptography used to e.g.
  - authentication
  - establishment of secret keys for symmetric-key cryptography
  - digital signatures

NSPK

- A nonce is a fresh "randomly generated" natural number
  - cannot be guessed
  - can be secret key to be agreed upon
- Notation: plaintext X encrypted with key K written $\{X\}_K$
- The protocol:

Message $1.$	$A \rightarrow B:$	$A.B.\{N_a.A\}_{PK(B)}$
Message $2.$	$B \rightarrow A:$	$B.A.\{N_a.N_b\}_{PK(A)}$
Message $3.$	$A \rightarrow B:$	$A.B.\{N_b\}_{PK(B)}$

- N_a and N_b are nonces

Modeling Nonces and Keys

- Nonce: `nonce`(A, i) models the ith nonce generated by A, instead of a natural number:
  - avoid over-specification
  - abstraction: it is not necessary to know the actual numeric value

```
sort Nonce .
op nonce : Oid Nat -> Nonce [ctor] .
```

- Keys: the actual numeric value not interesting:

```
sort Key .
op pubKey : Oid -> Key [ctor] .
```

Modeling the Messages

The protocol had three kinds of messages:

Message $1.$ $A \rightarrow B : \quad A.B.\{N_a.A\}_{PK(B)}$

Message $2.$ $B \rightarrow A : \quad B.A.\{N_a.N_b\}_{PK(A)}$

Message $3.$ $A \rightarrow B : \quad A.B.\{N_b\}_{PK(B)}$

Parts to be encrypted:

```
sorts MsgContent EncrMsgContent .
op _;_ : Nonce Oid -> MsgContent [ctor] . --- 1
op _;_ : Nonce Nonce -> MsgContent [ctor] . --- 2
subsort Nonce < MsgContent .                --- 3
```

Modeling the Messages (cont.)

Encrypted message content:

```
op encrypt_with_ : MsgContent Key
                   -> EncrMsgContent [ctor] .
```

... and sender's and receiver's `Oid`'s:

```
msg msg_from_to_ : EncrMsgContent
                   Oid Oid -> Msg .
```

The Classes

- Protocol defined for one run of the protocol
- Our model: many concurrent runs and many agents possible
- Classes `Initiator` and `Responder`, and later also `Intruder`
- Some agents may be both initiators and responders:

```
class InitiatorAndResponder .
subclass
  InitiatorAndResponder < Initiator Responder .
```

Initiator Sessions

Initiator must keep track of where in the protocol he is, for each agent:

```
sort InitSessions .
op emptySession : -> InitSessions [ctor] .
op __ : InitSessions InitSessions ->
                                  InitSessions
      [ctor assoc comm id: emptySession] .

op notInitiated : Oid -> InitSessions [ctor] .
op initiated : Oid Nonce -> InitSessions [ctor] .
op trustedConnection : Oid -> InitSessions
                                     [ctor] .
```

The Initiator Class

Needs a counter for generating nonces:

```
class Initiator | initSessions : InitSessions,
                  nonceCtr : Nat .
```

Initiator Rule 1

An agent wishes to establish an authenticated connection:

Message 1. $A \rightarrow B : \quad A.B.\{N_a.A\}_{PK(B)}$

```
vars A B : Oid .                    vars M N : Nat .
vars NONCE NONCE' : Nonce .  var IS : InitSessions .
rl [start-send-1] :
   < A : Initiator | initSessions : notInitiated(B) IS,
                     nonceCtr : N >
    =>
   < A : Initiator | initSessions :
                       initiated(B, nonce(A, N)) IS,
                     nonceCtr : N + 1 >
   msg (encrypt (nonce(A, N) ; A) with pubKey(B))
     from A to B .
```

Initiator Rule 2

Send message 3 when it receives message 2 with expected nonce:

Message 2. $B \rightarrow A$: $B.A.\{N_a.N_b\}_{PK(A)}$

Message 3. $A \rightarrow B$: $A.B.\{N_b\}_{PK(B)}$

```
rl [read-2-send-3] :
   (msg (encrypt (NONCE ; NONCE') with pubKey(A))
      from B to A)
   < A : Initiator | initSessions :
                       initiated(B, NONCE) IS >
    =>
   < A : Initiator | initSessions :
                       trustedConnection(B) IS >
   msg (encrypt NONCE' with pubKey(B)) from A to B .
```

Responder Class

A responder must know what nonces it has received, and needs a counter to generated new nonces:

```
class Responder | respSessions : RespSessions,
                  nonceCtr : Nat .

sort RespSessions .
op __ : RespSessions RespSessions -> RespSessions
        [ctor assoc comm id: emptySession] .

op responded : Oid Nonce -> RespSessions [ctor] .
op trustedConnection : Oid -> RespSessions [ctor] .
```

Responder Rule 1

Answer message 1 by sending message 2:

Message 1. $A \rightarrow B$: $A.B.\{N_a.A\}_{PK(B)}$

Message 2. $B \rightarrow A$: $B.A.\{N_a.N_b\}_{PK(A)}$

```
crl [read-1-send-2] :
    (msg (encrypt (NONCE ; A) with pubKey(B))
       from A to B)
    < B : Responder | respSessions : RS, nonceCtr : N >
     =>
    < B : Responder | respSessions :
                        responded(A, nonce(B, N)) RS,
                      nonceCtr : N + 1 >
    (msg (encrypt (NONCE ; nonce(B, N)) with
       pubKey(A)) from B to A) if not A inSession RS .
```

Responer Rule 2

Receive message 3 with its own nonce and can be sure to have contact:

Message 3. $A \rightarrow B : \quad A.B.\{N_b\}_{PK(B)}$

```
rl [read-3] :
   (msg (encrypt NONCE with pubKey(B))
      from A to B)
   < B : Responder | respSessions :
                        responded(A, NONCE) RS >
    =>
   < B : Responder | respSessions :
                        trustedConnection(A) RS > .
```

Validation of Intruder-less Protocol

- Specified NSPK for multiple protocol sessions
- No intruder defined
- Can check whether the protocol works without intruders
  - no deadlocks
  - can establish desired connections
  - no undesired connection can be established

Check Security of Protocol

How to check whether the protocol is safe against eavesdroppers/intruders/attacker?

- Think? No, done by smarter people!
- Model “smart” intruders?
  - probably thought a lot about smart intruders since 1978
- Define all possible (stupid and smart) behaviors by the intruder and search in Maude for a state which breaks the protocol?
  + no thinking required!
  + potential flaws in protocol usually due to unexpected scenarios
  + RBP flaw this way after 2 years
  - search can be too large for machine memory

The Intruders

- An intruder is no superman, but can
  - eavesdrop or intercept messages
  - decrypt messages encrypted with own key; to learn new nonces
  - send “fake” messages with known nonces
  - send encrypted messages it could not decrypt, but can change plaintext parts of these (sender/receiver)
- Intruder can also be an ordinary agent who wants to participate in ordinary runs of the protocol
- Can not guess nonces, keys
- We model all possible intruder behaviors

The Intruder Class

An intruder is an ordinary agent, and has, in addition, attributes to store all

- agent names
- nonces
- encrypted message contents

it has seen:

```
class Intruder | initSessions : InitSessions,
                 respSessions : RespSessions,
                 nonceCtr : Nat,
                 agentsSeen : OidSet,
                 noncesSeen : NonceSet,
               encrMsgsSeen : EncrMsgContentSet .
```

The Intruder Rules

13 rules:

- 4 rules corresponding to ordinary protocol rules
  - stores received names and nonces
- stealing/overhearing messages it can or cannot decrypt
- sending fake messages with known nonces and names
- sending fake messages with unknown encrypted content

Intercepting/Overhearing Messages

Rule for eavesdropping a message it cannot decrypt:

```
crl [overhear-but-not-understand] :
    (msg (encrypt MSGC with pubKey(O)) from O' to O)
    < I : Intruder | agentsSeen : OS,
                     encrMsgsSeen : ENCRMSGS >
     =>
    < I : Intruder | agentsSeen : OS ; O ; O',
                     encrMsgsSeen :
              (encrypt MSGC with pubKey(O)) ENCRMSGS >
    (msg (encrypt MSGC with pubKey(O)) from O' to O)
    if O =/= I .
```

Intercepting the message: remove `(msg ... from O' to O)`

from right-hand side of rule

Sending Fake Messages I

Sends a fake message with a content it does not know:

```
crl [send-encrypted] :
    < I : Intruder | encrMsgsSeen :
                     (encrypt MSGC with pubKey(B))
                     ENCRMSGS,
                        agentsSeen : A ; OS >
     =>
    < I : Intruder | >
    (msg (encrypt MSGC with pubKey(B))
         from A to B)
    if A =/= B /\ B =/= I .
```

Sending Fake Messages II

Use known nonces and names to send a fake message of type 1 with completely arbitrary “senders” and “receivers”:

```
crl [send-1-fake] :
    < I : Intruder | agentsSeen : A ; B ; OS,
                     noncesSeen : NONCE NSET >
     =>
    < I : Intruder | >
    (msg (encrypt (NONCE ; A) with pubKey(B))
         from A to B)
    if A =/= B /\ B =/= I .
```

Analysis I

Is it possible to break the protocol?

- Start with three actors: ”Peter”, ”Bank”, and the intruder ”Walker”
- ”Peter” does not want to establish connection with the ”Bank”
- If we reach a state where ”Bank” thinks it has established contact with ”Peter”, the protocol is broken

The Initial State

```
eq intruderInit =
    < "Peter" : Initiator | initSessions :
                             notInitiated("Walker"),
                            nonceCtr : 1 >
    < "Bank" : Responder | respSessions : emptySession,
                           nonceCtr : 1 >
    < "Walker" : Intruder | initSessions :
                              notInitiated("Bank"),
                            respSessions :
                              emptySession,
                            nonceCtr : 1,
                            agentsSeen :
                              "Bank" ; "Walker",
                            noncesSeen : emptyNonceSet,
                            encrMsgsSeen : emptyEncrMsg > .
```

Search for Bad State

Is there a reachable state where the ”Bank” thinks that it has estanblished connection with ”Peter”?

```
(search [1]
   intruderInit =>*
     C:Configuration
     < "Bank" : Responder |
                  respSessions :
                     trustedConnection("Peter")
                     RS:RespSessions,
                  ATTS:AttributeSet > .)
```

Search for Attacks

- Search failed to terminate (memory problems)
- Intruder too general and nondeterministic: can perform 40 actions in each step with minimal information
  - 10^{40} executions in 10 steps!
- Instead, split the search into two parts:
  1. searched for state where the "Bank" had a `responded("Peter", N:Nonce)` in its `respSessions`
  2. found such a state!
  3. searched for the attack from that state
  4. found such an attacked state!

Search for Attacks (cont.)

- To ensure that the original protocol, and not our model, is wrong, we looked at the path leading to the undesired state, and checked whether that behavior corresponds to a legal NSPK-execution:

$$
\begin{array}{lrcl}
S1.M1: & P \rightarrow W & : & P.W.\{N_p.P\}_{PK(W)} \\
S2.M1: & W(P) \rightarrow B & : & P.B.\{N_p.P\}_{PK(B)} \\
S2.M2: & B \rightarrow W(P) & : & B.P.\{N_p.N_b\}_{PK(P)} \\
S1.M2: & W \rightarrow P & : & W.P.\{N_p.N_b\}_{PK(P)} \\
S1.M3: & P \rightarrow W & : & P.W.\{N_b\}_{PK(W)} \\
S2.M3: & W(P) \rightarrow B & : & P.B.\{N_b\}_{PK(B)}
\end{array}
$$

- Conclusion: the protocol is flawed!

Discussion

- Classic protocol from 1978 described in Handbook of Applied Cryptography from 1996 without mention of problems
- Error found in 1995 by Lowe during automated formal analysis
- Could find attack without any insight
  - trivial model of protocol from description
  - trivial model of all possible behaviors of intruder
  - had to split search in two (only place to think)
- Search space mirrors problem's complexity
- Could define a "iterative bounded depth-first search" strategy using Maude's meta-programming facilities
- With somewhat smarter intruder: Maude search in 10 seconds!

NSPK in Maude: Conclusion

Maude analysis + (minimal insight or smarter search strategy)

can break classic protocol

Reliable broadcast protocol (RBP)

Reliable Broadcast Protocol

- So far: examples of small, well-known protocols
- Next: Maude specification and analysis of serious protocols under development
- Reliable broadcast in dynamic networks (RBP)
  - by Garcia-Luna and Zhang, published in IEEE ICC'96
  - for wireless, mobile systems
  - protocol "verified and analyzed" according to paper
  - simulation results in paper
  - for dynamic topologies
    - explicitly described also for static case

RBP in Maude

- Modeled static case in Maude in 1998
- Rewriting/simulating different topologies gave desired results
- Search for all possible final states discovered design error in RBP
- Suggested modified protocol and modeled and analyzed that
- New version of RBP for dynamic topology developed (in joint work with protocol developers) and modeled and analyzed in Maude

Updated Protocol for Static Case

1. The source node sends the message to all neighbors.
2. When a node n receives the message for the first time, it remembers the sender (the *parent* of n), and sends the message to all its neighbors except its parent.
3. When a node n receives a message it has already seen, the node sends an acknowledgment to the sender of this latest message (its *sibling*).
4. When a node has received an acknowledgment from all its neighbors except its parent, it sends an acknowledgment to the parent and its cycle is finished.
5. The protocol terminates when the source has received acknowledgments from all its neighbors.

Maude Model

The total protocol (for static case) has six small rules

- "When a node n receives the message for the first time, it remembers the sender (the *parent* of n), and sends the message to all its neighbors except its parent."

```
rl [RecMsg1] :
   (msg M From A To B)
   < B : Node | nbs : N, parent : none >
    =>
   < B : Node | parent : A >
   (multimsg M from B to N - A) .
```

Maude Model II

"When a node has received an acknowledgment from all its neighbors except its parent, it sends an acknowledgment to the parent and its cycle is finished."

```
rl [ackParent] :
   < A : Node | nbs : B ; N, parent : B,
                recdMsg : N', recdAck : N >
    =>
   < A : Node | parent : none, recdMsg : nil,
                recdAck : nil >
   (ackParent M from A to B) .
```

Initial Test Topology

```
eq test4 =
     (to a Broadcast m)
     < a : Node | nbs : b ; c, parent : none,
                  recdMsg : nil, recdAck : nil >
     < b : Node | nbs : a ; c, ... >
     < c : Node | nbs : a ; b ; d, ... >
     < d : Node | nbs: c, ... > .
```

Search for all final states for this and other initial states gave desired result

(for modified protocol)

AER/NCA protocol suite for reliable and scalable multicast in active networks

Real-Time Maude

- Many sophisticated communication protocols depend crucially on time aspects for their functionality
  - e.g., used to detect a message loss
  - resend a message if not acknowledged within the systems largest round trip time
- Ölveczky and Mesegeur have extended
  - rewriting logic to real-time rewriting logic
  - Maude to the language and tool Real-Time Maude

  to specify and analyze such real-time and hybrid systems

The AER/NCA Protocol

- Existing multicast protocols not both reliable and scalable
  - reliability $\Rightarrow$ sender has to "know" errors and resend lost packets
    $\Rightarrow$ too many acks/nacks to sender; repair packets from sender sent to too many nodes $\Rightarrow$ not scalable
- AER/NCA new protocol suite proposed by Kasera, Bhattacharyya, Keaton, Kiwior, Kurose, Towsley, and Zabele which uses *active* services ("repair servers") inside/co-located with routers along the multicast distribution tree (to execute programs/cache packets)

AER/NCA Protocol

- Real-time protocol: Heavy use of timers, durations of packet transmission, etc.
- Sophisticated protocol (40+ pages of UML specification)
- Four components:
  - Repair service: Error recovery, as close to the place of loss as possible
  - Rate control: Dynamically adjust sending rate based on feedback from “worst” receiver
  - Finding nominee receiver
  - Finding round trip times
- Protocol specified by informal UML-like use cases and simulated/tested using ns, ABONE, CANEs

Real-Time Maude Specification

A use case

```
A4. SPM Timer Service Routine
This use case begins when the SPM timer expires. An
SPM packet is transmitted to the destination multicast
address. The SPM packet contains the current highest
data packet sequence number, which is equal to nextSeq
minus 1. The SPM timer is then reset.
This use case ends when the SPM timer is reset.
```

is modeled by the rule

```
rl [A4] :
  < Q : RSsender | nextSeq : NZN, children : OS,
                   SPMTimer : 0 >   =>
    < Q : RSsender | SPMTimer : 4000 >
    multimsg SPMPacket(NZN - 1) from Q to OS .
```

AER/NCA: Summary

- AER/NCA is a “real” protocol suite
- OO specification style natural and easy to understand $\Rightarrow$ successful cooperation with non-experts in formal methods
- Found significant errors/omissions/problems in original specification while developing and analyzing Real-Time Maude specification
- Found all bugs independently found by the simulation tools, and found some additional serious design errors
- Protocol substantially revised based on Real-Time Maude analysis
- OO techniques: could analyze subprotocols in isolation and in combination
- Explicit time and resource modeling with formal analysis support

Other large communication protocols in Maude

Some Other Communication Protocols in Maude

- New NORM multicast protocol by IETF (Lien) (in Real-Time Maude)
- Spread and Secure Spread (Stehr, Talcott, Denker) for secure group communication
- IPSec (Stehr, Gunter): security architecture for IPv6
- Wireless sensor network protocols (beginning, in Real-Time Maude)
- Sophisticated new scheduling protocols for real-time systems (with Caccamo) (in Real-Time Maude)

Part 5. Experiences and Concluding Remarks

Experiences and Concluding Remarks

- Maude is a high-quality and high-performance tool for specifying and analyzing concurrent systems/protocols
- Intuitive, user-friendly, and general specification language
  - no fixed model of communication $\Rightarrow$ many different kinds of protocols can be modeled
  - generality + tight integration of static and dynamic parts $\Rightarrow$ large and sophisticated new "real" protocols can be modeled
  - OO specifications easily understood by non-formal methods people
  - OO $\Rightarrow$ can model and analyze components and their composition
  - provides a unifying formalism in which many different kind of systems can be understood

Experiences and Concluding Remarks (cont.)

- Formalization process uncovers ambiguities and implicit assumptions
- A wide range of analysis methods
  - can be employed early at almost no cost
  - non-domain expert (me) found serious design errors in all protocols not found by usual testing/simulation/"hand proofs"
  - search/model checking may choke because of state space explosion in distributed systems (use abstractions or strategies!)
- Abstract formalism $\Rightarrow$ used early in the protocol development process
- Impressive collection of sophisticated case studies
  - worked together with protocol developers
  - protocols changed based on Maude modeling and analysis

Experiences and Concluding Remarks (cont.)

- Further analysis strategies definable in the logic itself
  - iterative depth-first search for NSPK
- Specification formalism too general?
  - no explicit constructions for many things
  - too general for “graphical notation”?
  - more restricted formalism could provide optimized analysis techniques (e.g., BDD-based symbolic model checking)
- Specification formalism too restricted?
  - higher order capabilities? (meta-programming)

Extensions of Maude

- Real-Time Maude (Ölveczky, Meseguer)
- Mobile Maude (Maude team)
- PMaude (Kumar, Sen, Meseguer, Agha) for probabilistic theories
- OCC (Stehr; extension to higher order)

Experiences in Teaching

- Maude course first formal methods course at Univ of Oslo
  - theory about termination/confluence/equational logic
  - alternating bit protocol/two-phase commit protocol/dining philosophers/RBP/NSPK/simple distributed algorithms
- 6 times as many students as in previous introductory formal methods course
- Students understand protocols such as NSPK (checked in exam!)
- Students learn different kinds of systems without having to understand more than one formalism

Thank you very much!