Teaching Formal Methods Based on Rewriting Logic and Maude

Peter Csaba Ölveczky

Department of Informatics, University of Oslo

Abstract. This paper advocates teaching formal methods based on rewriting logic and the Maude tool for the purpose of widening access to formal methods. On the one hand, Maude's elegant, intuitive, and expressive programming/modeling language, its high-performance analysis methods, and some of its academic and industrial applications should make it appealing to a wide range of computer science students. On the other hand, teaching rewriting logic allows us to naturally incorporate substantial formal methods theory, such as equational logic and inductive theorem proving, TRS theory, and model checking. This paper also gives an overview of the content of – and the student feedback to – an introductory formal methods course based on rewriting logic that has been given at the University of Oslo since 2002.

1 Introduction

The slogan of TFM'09 is *widening the access to formal methods*. Key challenges that must be addressed by formal methods courses aiming at introducing formal methods to students with limited awareness and/or interest in formal methods include:

1. The need for formal methods must be well motivated to possibly skeptical students.
2. Applying formal methods should be reasonably easy, fun, and elegant.
3. The selected formalisms must appear to be relevant, both w.r.t. the student's future specialization – where we hope (s)he will apply formal methods – and elsewhere outside academia. No matter how nice a formalism and a tool are, if they are not used outside academia, the students will not be motivated to use the tool.
4. At the same time, the course should provide a serious introduction to substantial aspects of formal methods theory, and should exhibit some of the success stories of formal methods.

This paper advocates the use of *rewriting logic* [1, 2] and its associated high-performance tool Maude [3] as a basis for introductory courses in formal methods that interest students while giving a good introduction to formal methods.

Rewriting logic extends equational logic and term rewriting to model dynamic systems. In rewriting logic, the static parts of the system are modeled as

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, pp. 20–38, 2009.

an algebraic equational specification, and dynamic state changes are modeled as rewrite rules on the equivalence classes of terms induced by the equational theory. There is by now ample evidence that, despite the simplicity of the formalism, rewriting logic is quite expressive and general, and can be used to model a wide range of distributed systems. In particular, it provides a nice and simple model for concurrent objects [4]. Maude [3] is a freely available high-performance state-of-the-art formal tool based on rewriting logic. Maude supports the simulation/rewriting, reachability analysis, and linear temporal logic model checking of rewrite theories.

The challenges (1) to (4) above are (or can be) addressed by a rewriting-logic-based course as follows:

1. There is a wide range of critical distributed systems where formal analysis has proved indispensable. The model checking effort of Lowe to find an attack on the NSPK cryptographic protocol after 17 years is but one example that should appeal to students.
2. The functional programming style of Maude is fairly elegant – as I try to convey in Section 4 – and is typically enjoyed by students who like programming. Likewise, the object-oriented and rule-based way of modeling distributed systems has been shown to be intuitive and easily understandable also for people without formal methods background [5].
3. Since rewriting logic is fairly expressive and can be applied to a wide range of distributed systems, it should be relevant to students who will pursue other fields than formal methods. Furthermore, Maude is increasingly being used outside academia, with some "sexy" applications, such as its use at Microsoft to find previously unknown security flaws in web browsers [6], the use in the Japanese car industry to find bugs in embedded automotive software, etc.
4. At the same time, a rewriting-logic-based course naturally includes a fair amount of formal theory, including formal proofs and deduction systems, classic term rewrite system theory, algebra, and (linear temporal logic) model checking, often mentioned as one of the success stories of formal methods.

To test the hypothesis about the suitability of widening access to formal methods through rewriting logic and Maude, I have developed an introductory course at the Department of Informatics, University of Oslo, that has been given since 2002. University of Oslo should be a suitable candidate for this experiment, since formal methods typically do not attract many students, and since many students at the department are somewhat weak and uninterested in mathematics. This paper gives an overview of the contents of our course, and on our experiences and student feedback on this course since 2002.

The course consists of two parts: (i) equational specifications and their analysis, and (ii) rewrite specifications and their analysis. The first part introduces classic (order-sorted) equational logic and term rewrite system (TRS) theory. In particular, we study: the definition of the usual data types (lists, sets, binary trees, ...) in Maude; classic TRS theory including proving termination of and confluence of an equational specification; equational logic; inductive theorems; and some small examples such as quick-sort and merge-sort. Part (ii) introduces:

rewriting logic and its proof theory; modeling distributed concurrent objects in rewriting logic; modeling a wide range of communication forms; temporal logic; reachability analysis and LTL model checking in Maude; and a set of larger examples, such as the two-phase commit protocol for distributed databases, the TCP, alternating bit, and sliding windows communication protocols, and the NSPK cryptographic protocol.

By giving a range of larger examples, I have tried to convey the difficulty of designing distributed systems, and how such systems can be modeled. Most importantly, the NSPK crypto-protocol case study serves as a very nice motivating example for formal model checking in today's iBanking society: a three-line protocol which is so hard to understand and get right that its flaws went undiscovered for 17 years, and whose critical flaw was discovered by formal model checking techniques similar to those presented in the course.

As further explained in Section 6, our experiences are mostly positive. The course consistently gets very positive student evaluation, but has one substantial weakness. Furthermore, although the course does not attract as many students as we would have liked, the course does attract significantly more students than our previous formal methods course, and is mostly taken by students who major in other fields, such as computer networks, computational linguistics, mathematical logic, and so on.

Section 2 gives some background on rewriting logic and Maude. Section 4 gives an overview of our course, and gives some samples of the Maude specifications to allow the reader to form a first impression about the suitability of basing a formal methods course on Maude. The paper also discusses experiences and student feedback on the course (Section 6), course material (Section 5), and related courses (Section 7). Section 9 gives some concluding remarks.

2 Rewriting Logic and Maude

Rewriting logic [1,2] is a logic of change that was developed by José Meseguer in the early 1990-ies. A rewriting logic specification is a rewrite theory (Σ, E, R), where (Σ, E) is an algebraic equational specification – that may be unsorted, many-sorted, order-sorted, or a membership equational logic [7] specification – with Σ an algebraic signature and E a set of equations (and possibly membership axioms), and where R is a set of labeled conditional rewrite rules of the form

$$l : \ [t]_E \longrightarrow [t']_E \ \textbf{if} \ cond,$$

with l a *label* and t and t' Σ-terms. Such rules specify the system's local transition patterns. The state space and functions of a system are thus specified by an equational specification, whereas the dynamic state changes are modeled by rewrite rules. Despite its simplicity, rewriting logic has been shown to be an expressive model of concurrency in which many other models of concurrency and communication can be naturally represented [1,8].

Maude [3] is a mature high-performance language and tool supporting the specification and analysis of rewrite theories. Maude assumes that the equations

are terminating and confluent. Maude executes rewrite rules by reducing the state to its equational normal form before applying a rewrite rule. We briefly summarize the syntax of Maude. Operators are introduced with the `op` keyword. They can have user-definable syntax, with underbars '`_`' marking the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. There are three kinds of logical statements, namely, *equations*—introduced with the keywords `eq`, or, for conditional equations, `ceq`—*memberships*—declaring that a term has a certain sort and introduced with the keywords `mb` and `cmb`—and *rewrite rules*—introduced with the keywords `rl` and `crl`. The mathematical variables in such statements are declared with the keywords `var` and `vars`.

Full Maude [3] is a prototype extension of Maude – implemented in Maude – that provides convenient syntax for object-oriented specification. In object-oriented Full Maude modules one can declare *classes* and *subclasses*. A class declaration

`class` C `|` att_1 `:` s_1 `, ... ,` att_n `:` s_n `.`

declares an object class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term

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

where O is the object's name, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In an object-oriented system, the state, which is usually called a *configuration*, is a term of the built-in sort `Configuration`. It has typically the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative and having the `none` multiset as its identity element, so that order and parentheses do not matter, and so that rewriting is *multiset rewriting* supported directly in Maude. The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

```
rl [l] : m(O,w)  < O : C | a1 : x, a2 : O', a3 : z >   =>
                 < O : C | a1 : x + w, a2 : O', a3 : z >  m'(O') .
```

defines a family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions have the effect of altering the attribute `a1` of the object `O` and of sending a new message. "Irrelevant" attributes (such as `a3`, and the *right-hand side occurrence* of `a2`) need not be mentioned in a rule.

As mentioned, rewrite theories are executable under fairly mild conditions. Maude supports a wide range of analysis strategies for rewrite theories, including simulation, reachability analysis, and linear temporal logic model checking.

Maude's *rewrite* command simulates *one* behavior of the system, possibly up to a certain number of rewrite steps. It is written with syntax

`rew` [[n]] t `.`

where t is the initial state, and n is the (optional) bound of the number of rewrite steps to execute.

Maude's *search* command uses explicit-state breadth-first search to search for states that are reachable from a given initial state t, and that match a *search pattern*, and satisfy a *search condition*. The command which searches for *one* state satisfying the search criteria has syntax

`search [1]` t `=>*` *pattern* `such that` *cond* `.`

Maude caches the visited states, so that the search command terminates if the state space reachable from t is finite, or if the desired number of (un)desired states are reachable from the initial state.

Maude's *linear temporal logic model checker* [9] analyzes whether each behavior from an initial state satisfies a temporal logic formula. *State propositions* are terms of sort `Prop`, and their semantics should be given by (possibly conditional) equations of the form

statePattern `|=` *prop* `=` b

for b a term of sort `Bool`, which defines the state proposition *prop* to hold in all states t where t `|=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), `O` ("next"), and `U` ("until"). The model checking command has syntax

`red modelCheck(`t`,` *formula*`)` `.`

for t the initial state and *formula* the temporal logic formula. The model checking terminates if the state space reachable from t is finite.

Rewriting logic is an active area of research, in which more than a thousand research papers have been published. Some of the applications of rewriting logic and Maude include: defining the formal semantics for a wide range of programming and modeling languages [10]; work at Microsoft that discovered previously unknown address and status bar spoofing attacks in web browsers [6]; developing analysis tools for programming languages, such as the JavaFAN [11] tool for efficiently analyzing multi-threaded Java programs; analysis of advanced security, communication, and wireless sensor network protocols (see e.g. [12, 13, 5, 14, 15, 16]); modeling of cell biology to simulate and analyze biological reactions [17, 18]; finding several bugs in embedded software used by major car makers; implementing the latest version of the NRL PA crypto-protocol analysis tool [19]; implementing extensions of rewriting logic, such as Real-Time Maude tool [20, 21] for real-time systems and the PMaude tool for probabilistic systems [22]. The paper [23] provides an early bibliography and roadmap of the use of Maude around the world.

3 Prerequisites and Course Duration

Our course is aimed at third-year students at the Department of Informatics at the University of Oslo. The course basically starts "from scratch", as it is not assumed that students have significant mathematical background; they may for example never have seen a proof system before. It is also *not* assumed that the students have any experience with functional or logic programming.

The course consists of 14 lectures, each lasting between two and three 45-minute "hours".

4 Course Overview and Sampler

This section gives an overview of the course content, and some samples to get a first impression of the course. The course is divided into two roughly equal-sized parts: modeling and analyzing, respectively, the *static* and the *dynamic* parts of the systems. The static part corresponds to "classical" term rewrite system (TRS) theory and specification of data types in Maude.

4.1 The Static Part

Defining Data Types in Maude. In Maude, data types are defined by an algebraic *equational specification*, where a *signature* declares a set of sorts and function symbols (or *operators*); the operators are divided into *constructors* (`ctor`) that define the carrier of the sort, and *defined functions*. The first such specification given in the course is the following module, that defines the natural numbers, together with an addition function, in a Peano style:

```
fmod NAT-ADD is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .

  vars M N : Nat .

  eq 0 + M = M .
  eq s(M) + N = s(M + N) .
endfm
```

The underscore in the function declaration gives the place of the arguments in a "mix-fix" notation. It should be noted that, for convenience and efficiency, common data types, such as integers, floating-point numbers, Booleans, strings, etc., are built-in in Maude.

As mentioned in Section 2, the equational specification may be order-sorted (that is, include subsort declarations), and operators can be declared to be *associative* (`assoc`), *commutative* (`comm`), and/or to have an *identity element* (`id`). Matching is then performed *modulo* such axioms. Using these features, lists of integers can be defined as follows, where the sort `NeList` defines *non-empty* lists:

```
sorts List NeList .
subsorts Int < NeList < List .

op nil : -> List [ctor] .
op __ : List List -> List [assoc id: nil ctor] .
op __ : NeList NeList -> NeList [assoc id: nil ctor] .
```

List concatenation is defined by the juxtaposition operator (`__`). Since `Int` is a subsort of `List`, the number `6` is also a list. Therefore, `6 32` is a two-element list, and `(6 32) 17` is a list. Since `__` is declared to be associative, `(6 32) 17` and `6 (32 17)` are the same list, and can be written `6 32 17`. Since list concatenation is declared to have `nil` as the identity element, any list is either of the form `nil` or $n\ l$ (or $l\ n$), for n an integer and l a `List`.

The definition of the usual list functions seems to be more elegant than in most languages:

```
op length : List -> Nat .
ops first last : NeList -> Int .
op rest : NeList -> List .
op reverse : List -> List .

vars I J K : Int .    vars L L' : List .    vars NEL NEL' : NeList .

eq length(nil) = 0 .                    eq reverse(nil) = nil .
eq length(I L) = 1 + length(L) .        eq reverse(L I) = I reverse(L).

eq first(I L) = I .                     eq rest(I L) = L .

eq last(L I) = I .
```

Using this representation of lists, the well known merge-sort function can be specified fairly elegantly in Maude:

```
op mergeSort : List -> List .
op merge : List List -> List [comm] .

eq mergeSort(nil) = nil .
eq mergeSort(I) = I .
ceq mergeSort(NEL NEL') = merge(mergeSort(NEL), mergeSort(NEL'))
      if length(NEL) == length(NEL')  or length(NEL) == s length(NEL') .

eq merge(nil, L) = L .
ceq merge(I L, J L') = I merge(L, J L') if I <= J .
```

In the same way, we have defined the usual data types, such as binary trees, and multisets. Indeed, any computable data type can be defined as a confluent and terminating equational theory [24].

Confluence and Termination. Maude's rewrite engine executes equational specifications by reducing a term to its normal form. Maude therefore assumes

that a specification is *confluent* and *terminating*, modulo associativity and commutativity of the function symbols so declared. This gives us the motivation to study these properties, which in my introductory course are studied only for unsorted systems without associativity, commutativity, and conditional equations. After defining formally what it means to perform a simplification step, confluence is studied in the usual way, assuming termination, and using the critical pair's lemma to check local confluence.

The course deals a fair amount with theoretical and practical aspects of proving termination. We study both "weight function" mappings of ground terms onto well-founded domains, and the elegant theory of Dershowitz' simplification orderings [25, 26] that lead to the multiset and lexicographic path orderings as well as other termination orderings. Although these techniques apply to term rewrite systems, it is the hope and motivation that the students are able to adapt their understanding of termination to also analyze termination of imperative programs such as the Euclidean algorithm for finding the greatest common divisor of two natural numbers `m` and `n`:

```
int gcd(int m, int n) {   // m,n > 0
  int x := m;   int y := n;
  int r := x % y;
  while (r>0) {
    x := y;
    y := r;
    r := x % y;
  }
  return y;
}
```

Equational Logic. The course introduces equational logic (again, in its simplest, unsorted case). For many of our students, this is the first time they see a formal deduction system. The usual undecidability and decidability results are given. Finally, we study *inductive theorems* and induction on data types, and prove basic inductive theorems on toy problems, such as that reversing a binary tree twice yields the original tree.

Although the described course does not use it, Maude has an associated *inductive theorem prover* that can assist in the proof of inductive theorems [27].

4.2 Modeling and Analyzing Dynamic Systems in Maude

Part II of the course deals with the formal modeling of *dynamic* systems as *rewriting logic* theories, and their formal analysis in the Maude tool. Since two of the main goals of the course is to (i) give the students some intuition about the difficulty of designing distributed systems, and (ii) teach the students to formally model designs of such systems, we focus on modeling and analyzing a range of examples from different domains: transport protocols, transaction protocols for distributed database systems, and cryptographic protocols.

Rewriting Logic. We introduce *rewriting logic* and its proof theory; in particular, this proof system allows us to reason about what actions can be performed concurrently. We show that if the state has a *multiset* structure, then each element in the multiset could be involved in one rewrite "action" in a concurrent rewrite step; that is, distributed objects can performs actions concurrently.

We illustrate such modeling with simple examples, such as modeling the life (the age and marital status) of a collection of persons, various kinds of games, etc. For example, (the scores of) a never-ending soccer game is modeled by the following module, where a typical state could be the term `"PSV" - "Ajax" 3 : 2`:

```
mod GAME is protecting NAT + STRING .
  sort Game .
  op _-_ _:_ : String String Nat Nat -> Game [ctor] .

  vars HOME AWAY : String .     vars M N : Nat .

  rl [home-goal] :
     HOME - AWAY M : N  =>  HOME - AWAY M + 1 : N .

  rl [away-goal] :
     HOME - AWAY M : N  =>  HOME - AWAY M : N + 1 .
endm
```

Formal Analysis in Maude. Since, in contrast to equational theories, the rewrite rules need not be confluent or terminating, the Maude tool offers different formal analyses. We cover rewriting for simulation and search for reachability analysis. For example, in the soccer game, the following search command checks whether it is possible to reach a state in which the away team leads by at least three goals[1]:

```
Maude> search [1] "Ajax" - "PSV" 0 : 0
                  =>*
                  "Ajax" - "PSV" M:Nat : N:Nat such that N:Nat >= M:Nat + 3 .
```

Concurrent Objects. We then introduce concurrent objects by the simple example of modeling the lives of a collection of persons, where the class `Person` is declared as follows:

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

  sort Status .
  op single : -> Status [ctor] .
  ops engaged married separated : Oid -> Status [ctor] .
```

[1] Variables declared on the fly have the form *var*:*sort*.

The following conditional rewrite rule, involving two objects, models the engagement of two single persons who are both older than 15:

```
vars N N' : Nat .   vars X X' : String .

crl [engagement] :  < X : Person | age : N, status : single >
                    < X' : Person | age : N', status : single >
                   =>
                    < X : Person | status : engaged(X') >
                    < X' : Person | status : engaged(X) >
                   if N > 15 /\ N' > 15 .
```

A married person can initiate a separation, for example by sending a message to his/her spouse. The following declares a `separate` message, and shows the rules for sending, respectively receiving, such a message[2]:

```
msg separate : Oid -> Msg .

rl [sendSep] :  < X : Person | status : married(X') >
               =>
                < X : Person | status : separated(X') >
                separate(X') .

rl [recvSep] :  separate(X)
                < X : Person | status : married(X') >
               =>
                < X : Person | status : separated(X') > .
```

The course also presents an object-oriented version of the mandatory *dining philosophers* example.

Communication Protocols. After defining ways of modeling a wide range of communication models (unordered/ordered transmission; reliable/unreliable; unicast/multicast/broadcast, ...), we specify a set of transport protocols, such as TCP-like sequence number based protocols, the alternating bit protocol, and (as a student homework) different versions of the sliding window protocol. Maude search is used to analyze the protocols. From an educational perspective, the sliding window protocol is a very good example for illustrating that model checking distributed systems takes a long time in a highly nondeterministic setting where any message may get lost (or duplicated).

The Two-Phase Commit Protocol for Distributed Databases. We model and analyze the *two-phase commit protocol* for transactions in distributed database systems with replicated data, as the protocol is described in the textbook [28]. Again, we analyze our model by searching for *final* states of protocol runs, and automatically find the well known facts that the multi-database is consistent after a run if and only if messages cannot get lost.

[2] As we show in the course, separation is not this simple.

The NSPK Cryptographic Protocol. The *Needham-Schroeder public key* (NSPK) authentication protocol is a frequently used and cited protocol from 1978. NSPK is, for example, cited in *Handbook of Applied Cryptography* [29] from 1996, without any error in the protocol being mentioned. In crypto-protocol notation, NSPK is described as follows:

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

We define an object-oriented Full Maude model of the protocol for multiple agents and protocol runs. An agent which can *initiate* a run of the protocol is modeled as an object of the following class `Initiator`:

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

The initiator needs to know the nonce it sent to the responder in Message 1, so that it can check whether this is the same nonce that it receives in Message 2. In our setting, where an initiator may be simultaneously involved in many runs of the protocol with different responders, the initiator must store information about the nonces of all these runs. In the attribute `initSessions` an initiator A stores such information in a multiset of elements of the following three kinds:

- `notInitiated`(B) indicates that A can/will initiate contact with B but has not yet done so;
- `initiated`(B, N) indicates that A has sent Message 1 to B with nonce N and is waiting for Message 2 from B; and
- `trustedConnection`(B) indicates that A has established (what she thinks is) an authenticated connection with B.

The data type representing this kind of information is defined as follows:

```
sorts Sessions InitSessions .          subsort Sessions < InitSessions .
op emptySession : -> Sessions [ctor] .
op __ : InitSessions InitSessions -> InitSessions
                                       [ctor assoc comm id: emptySession] .
op __ : Sessions Sessions -> Sessions [ctor assoc comm id: emptySession] .

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

The dynamics of the protocol is described by four rewrite rules, two for the initiator and two for the responder. The rules for the initiator are described next.

The first rule models the sending of Message 1. The agent `A` has `notInitiated(B)` in its `initSessions` attribute, which indicates that it is interested in establishing an authenticated connection with `B`. The initiator generates a fresh nonce `nonce(A, N)`, encrypts this nonce together with its identifier with

the public key of B (`encrypt ... with pubKey(B)`), and adds its own and B's name (`msg ... from A to B`) to this message and sends it out into the configuration. Agent A must remember that it has `initiated` contact with B with nonce `nonce(A, N)` and must also increase its nonce counter. All this happens in the following rule:

```
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 .
```

The next rule models the reception of Message 2 from, and the sending of Message 3 to, an agent B who sent a pair of nonces encrypted with A's public key. If the first nonce (`NONCE`) in the message received (and decrypted) by A is the same as the nonce stored in A's `initSessions` attribute for B, the agent A figures out that it has established an authenticated connection with B, and sends Message 3 (B's nonce (`NONCE'`) encrypted with B's public key) to 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 .
```

The Dolev-Yao intruder is modeled as a class `Intruder` with 14 simple rewrite rules. For example, the following rule models the case when the intruder intercepts a message that it cannot decrypt. In that case, the intruder just stores the message content in its `encrMsgsSeen` attribute, and stores the new names it learns in its `agentsSeen` attribute:

```
vars I O O' : Oid .   var OS : OidSet .   var MSGC : MsgContent .
var ENCRMSGS : EncryptedMsgContentSet .

crl [intercept-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 >
   if O =/= I .
```

Another rule can then spontaneously send *any* fake message among the messages the intruder has seen out into the configuration, using any agent `A` it has seen[3] as sender:

```
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 .
```

The following search command finds the well known attack on NSPK. In the initial state, `"Scrooge"` does *not* want to have a contact with the `"Bank"`. In the search command we check whether from such a state, it is possible to reach a state where the `"Bank"` thinks it has an authenticated connection with `"Scrooge"`:

```
Maude> (search [1]
         < "Scrooge" : Initiator |
                       initSessions : notInitiated("Beagle Boys"), ... >
         < "Bank" : Responder | respSessions : emptySession, nonceCtr : 1 >
         < "Beagle Boys" : Intruder |
                           initSessions : notInitiated("Bank"), ... >
       =>*
         C:Configuration
         < "Bank" : Responder | respSessions : trustedConnection("Scrooge")
                                               RS:RespSessions > .)
```

Maude does find a behavior leading to the bad state, and can exhibit this behavior as explained in [30].

This example is excellent for motivating the students, for illustrating the difficulties of designing distributed systems, and for showing the usefulness of formal model checking. The protocol is described in *three* lines of specification. Yet, due to concurrent runs, it is so hard to understand that it took 17 years to find the error, which was found using exactly the same kind of analysis we are doing: exhaustive model checking of a formal model of the protocol [31].

Temporal Properties and LTL Model Checking. Finally, we explain different classes of requirements, and show how invariants can be validated by Maude's search command. More complex temporal system properties can be formalized in linear temporal logic (LTL), and Maude's LTL model checker can be used to analyze whether a system satisfies its requirements. However, to avoid introducing yet another logic to my students, I typically postpone teaching temporal logic to the follow-up course.

[3] Both `encrMsgsSeen` and `agentsSeen` are declared to hold *multisets*. Therefore, the rule can nondeterministically select *any* of the agents and *any* of the messages the intruder has stored.

5 Teaching Material

Fairly mature lecture notes (340 pages) for my course are available on the web at `http://peterol.at.ifi.uio.no/inf3230-lecturenotes.html`. These lecture notes start from scratch and contain many examples and exercises, and should be accessible for people without any formal methods experience. These notes are also suggested reading for the introductory formal methods course CS 476 at the University of Illinois at Urbana-Champaign (UIUC), and seem to have been read by a fair amount of people at UIUC. These notes are also one of the main sources that are recommended to people who want to get a gentle introduction to Maude.

The Maude book [3] is a very useful reference material on the Maude system and on rewriting logic, but, in my view, requires some previous knowledge about term rewriting or similar formal methods. Although no course book exists for the course CS 476 at UIUC, the slides for that much more advanced introductory course are quite comprehensive and can almost be studied as a course book.

6 Evaluation and Impact

This section briefly summarizes anonymous student feedback, my own impression of the students' experiences, and some of the impact this course has had in Oslo.

6.1 Student Feedback

The comments and evaluations from university's anonymous web-based feedback system have been overwhelmingly positive. Unfortunately, most comments are non-constructive comments of the form "Very interesting and exciting". Some write that that it is good that "theory and practice are well interleaved," and other thought programming in Maude was fun. One person told me that he did not really understand the sliding window protocol in the network course, but understood it very well in this course, where implementation details are abstracted away to focus on the essence of the protocol.

The negative feedback overwhelmingly concerns Full Maude, the prototype extension of Maude that is used to model and analyze object-oriented Maude specifications. Unfortunately, Full Maude's slight lack of robustness and, in particular, its cryptic, non-existing, and/or misplaced error messages make it a frustrating experience for some students to get larger specifications, such as the sliding window protocol, right.

As for the difficulty of the course, 5% of my students in 2003 found the course "difficult," 55% found the course "somewhat difficult," and the remaining 40% found it "neither easy nor difficult."

Despite the very positive student feedback each year, the course still does not attract as many students as I would like. The reason *may* lie in the way the students have to select courses, and in the freedom they have to select courses outside their specialization. Typically, 20 to 30 students take the exam each year.

6.2 Impact in Oslo

I am not very much aware of what former students of the course are doing. Two cases that are worth mentioning are:

1. A former student and teaching assistant in my course started a company five years ago, selling a product/service that is implemented in Maude. The company is still doing well, and has also employed another former student and TA of mine to program in Maude.
2. Inspired by the use of Maude to find the attack on NSPK, a former student went on to do a Ph.D. in crypto-analysis using Maude, including defining his own protocol analyzer on top of Maude. The person is currently analyzing critical infrastructure in the Norwegian banking sector.

In addition, it is worth remarking that Maude is now frequently used in neighboring research groups at the department. For example, Maude is used to implement proof search strategies in the logics group, the Creol object-oriented language [32] developed at the department is interpreted in Maude, and a student in the linguistics research group (!) has analyzed an IETF-developed multicast protocol using Maude [14].

7 Related Courses

I am only aware of one other "introductory" formal methods course based on rewriting logic. It is given at UIUC by José Meseguer, who developed rewriting logic. That course is significantly more theoretically challenging and comprehensive than the one in Oslo. In addition to treating the theory of rewriting logic and its analysis methods in depth, it also focuses on the verification of sequential imperative programs. There is less focus on larger examples, although the NSPK case study is covered. Another difference is that the UIUC course does not have a course book (but an extensive set of slides). Indeed, my lecture notes are recommended supplementary reading in the UIUC course.

Classic term rewriting theory has been taught many places for years. One difference between many of those and the the first part of the Oslo course is our focus on defining functions in an executable language such as Maude.

Likewise, analyzing dynamic systems using process algebras, model checkers like Spin, various kinds of transition systems, has been much taught. One of the differences of using Maude is the *modeling convenience* and *expressiveness* of the Maude formalism, and the ability to perform both simulation, reachability analysis, and LTL model checking. Another attractive feature of Maude is its very simple and intuitive functional programming style language, which is typically appealing to students, and which makes it far easier to model a system system that, say, in Promela/Spin. Another difference is our focus on case studies from different domains.

8 Follow-Up Courses

Due to the large amount of interesting research being performed using rewriting logic and Maude, there are plenty of appealing subjects to choose from for an advanced follow-up course based on rewriting logic. The follow-up course at the University of Oslo teaches the following topics:

- A student project formalizes and analyzes a published communication protocol which is claimed to be correct, but where a simple Maude search finds an unexpected deadlock.
- Linear temporal logic and its model checking in Maude.
- Meta-programming in Maude.
- Specification and analysis of real-time systems using Real-Time Maude [20].
- Other analysis methods, including narrowing analyses and the use of Maude's inductive theorem prover (ITP) [27].
- Modeling cell biology and and analyzing biological cell reactions [17, 18].
- Study the work in [6] on finding the attacks in web browsers.

Other topics of general interest include:

- Grigore Roşu at UIUC teaches a course on how a wide range of programming languages can be given a rewriting logic semantics and can be analyzed using Maude.
- Probabilistic rewrite theories [33] and their analysis using PMaude [22].
- Theory on the algebraic denotational semantics of equational and rewrite theories.

This is but a small sample of topics that could be covered by an advanced course.

9 Concluding Remarks

This paper has advocated the use of rewriting logic and its associated high-quality tool Maude as a basis for teaching formal methods with the aim of *widening the access to formal methods*. The reasons for believing that a Maude-based formal methods course may interest people who would not normally consider studying formal methods include:

- The logic and programming language are simple and intuitive: they consist of an algebraic signature, equations, and rewrite rules. That's all. Furthermore, the object-oriented rules are very intuitive and easy to understand also for people without formal methods or Maude knowledge (see, e.g., [5]).
- The functional and fairly elegant programming possible in Maude, that this paper tried to convey with the merge-sort example, should be appealing to people who like to program.
- It is fairly easy – in particular compared to traditional formal languages for concurrency – to model a wide range of distributed systems in Maude.

- The NSPK security protocol analysis provides compelling motivation for the use of formal model checking. Furthermore, it is easy to specify NSPK in Maude, and to find the attack using search in Maude.
- No matter its elegance or nice features, no language will motivate students if it is perceived to be a purely academic language not used in industry. Maude has some "sexy" industrial applications, most notably the work at Microsoft that uncovered previous unknown security flaws in web browsers.

From a formal methods teaching perspective, a fair amount of formal methods theory, including classic TRS theory, proof systems and inductive proofs, as well as different forms of model checking – one of the success stories of formal methods – can naturally be integrated and motivated by the use of Maude for system modeling and analysis.

This paper has also presented an introductory Maude-based formal methods course given at the University of Oslo since 2002. This course, also aimed at – and taken by – students who will not necessarily pursue formal methods further, has consistently received positive student feedback, and comes with a fairly mature course book that is freely available and should be suitable starting point for studying formal methods.

Although I believe that the expressive and intuitive formalism of Maude makes it better suited than other model checking systems, such as SMV [34] and Spin [35], for teaching modeling and analysis of complex distributed systems with advanced data types and communication features, much more work comparing Maude-based formal methods teaching with other approaches is needed before significant conclusions can be drawn.

Acknowledgments. I am grateful to Olaf Owe for supporting the development of the described course at the University of Oslo, and to the anonymous reviewers for helpful comments on an earlier version of this paper.

References

1. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
2. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoretical Computer Science 360, 386–414 (2006)
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
4. Meseguer, J.: A logical theory of concurrent objects and its realization in the Maude language. In: Agha, G., Wegner, P., Yonezawa, A. (eds.) Research Directions in Concurrent Object-Oriented Programming, pp. 314–390. MIT Press, Cambridge (1993)
5. Ölveczky, P.C., Meseguer, J., Talcott, C.L.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. Formal Methods in System Design 29, 253–293 (2006)

6. Chen, S., Meseguer, J., Sasse, R., Wang, H.J., Wang, Y.M.: A systematic approach to uncover security flaws in GUI logic. In: IEEE Symposium on Security and Privacy, pp. 71–85. IEEE Computer Society, Los Alamitos (2007)
7. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
8. Meseguer, J.: Rewriting logic as a semantic framework for concurrency: a progress report. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 331–372. Springer, Heidelberg (1996)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual, Version 2.3 (2007), `http://maude.cs.uiuc.edu`
10. Meseguer, J., Rosu, G.: The rewriting logic semantics project. Theoretical Computer Science 373, 213–237 (2007)
11. Farzan, A., Chen, F., Meseguer, J., Rosu, G.: Formal analysis of Java programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004)
12. Denker, G., Meseguer, J., Talcott, C.: Protocol Specification and Analysis in Maude. In: Heintze, N., Wing, J. (eds.) Workshop on Formal Methods and Security Protocols, Indianapolis, Indiana, June 25 (1998)
13. Denker, G., García-Luna-Aceves, J.J., Meseguer, J., Ölveczky, P.C., Raju, Y., Smith, B., Talcott, C.: Specification and analysis of a reliable broadcasting protocol in Maude. In: Hajek, B., Sreenivas, R.S. (eds.) 37th Annual Allerton Conference on Communication, Control, and Computation. University of Illinois, Urbana-Champaign (1999)
14. Lien, E.: Formal modelling and analysis of the NORM multicast protocol using Real-Time Maude. Master's thesis, Department of Linguistics, University of Oslo (2004)
15. Goodloe, A., Gunter, C.A., Stehr, M.O.: Formal prototyping in early stages of protocol design. In: WITS 2005. ACM Press, New York (2005)
16. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theoretical Computer Science 410, 254–280 (2009)
17. Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Talcott, C.: Pathway logic: Executable models of biological networks. Electronic Notes in Theoretical Computer Science 71 (2002)
18. Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J., Sonmez, K.: Pathway logic: Symbolic analysis of biological signaling. In: Pacific Symposium on Biocomputing, Hawaii, pp. 400–412 (2002)
19. Escobar, S., Meadows, C., Meseguer, J.: State space reduction in the Maude-NRL Protocol Analyzer. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 548–562. Springer, Heidelberg (2008)
20. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20, 161–196 (2007)
21. Ölveczky, P.C., Meseguer, J.: The Real-Time Maude tool. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 332–336. Springer, Heidelberg (2008)
22. Agha, G., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. Electronic Notes in Theoretical Computer Science 153, 213–239 (2006)
23. Martí-Oliet, N., Meseguer, J.: Rewriting logic: Roadmap and bibliography. Theoretical Computer Science 285 (2002)

24. Bergstra, J.A., Tucker, J.V.: Initial and final algebra semantics for data type specificartions: Two characterization theorems. SIAM Journal on Computing 12, 366–387 (1983)
25. Dershowitz, N.: Orderings for term-rewriting systems. Theoretical Computer Science 17, 279–301 (1982)
26. Dershowitz, N.: Termination of rewriting. Journal of Symbolic Computation 3, 69–116 (1987)
27. Clavel, M.: The ITP tool home page, `http://maude.sip.ucm.es/itp/`
28. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems, 5th edn. Addison Wesley, Reading (2007)
29. Menezes, A., van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996), `http://www.cacr.math.uwaterloo.ca/hac`
30. Ölveczky, P.C.: Formal modeling and analysis of distributed systems in Maude. Course book for INF3230, Dept. of Informatics, University of Oslo (2009)
31. Lowe, G.: An attack on the Needham-Schroeder public-key authentication protocol. Information Processing Letters 56, 131–133 (1995)
32. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6, 35–58 (2007)
33. Kumar, N., Sen, K., Meseguer, J., Agha, G.: A rewriting based model of probabilistic distributed object systems. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 32–46. Springer, Heidelberg (2003)
34. Clarke, E., Grumberg, O., Long, D.: Verification tools for finite-state concurrent systems. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1993. LNCS, vol. 803. Springer, Heidelberg (1994)
35. Holzmann, G.J.: The model checker SPIN. IEEE Trans. on Software Engineering 23, 279–295 (1997)